



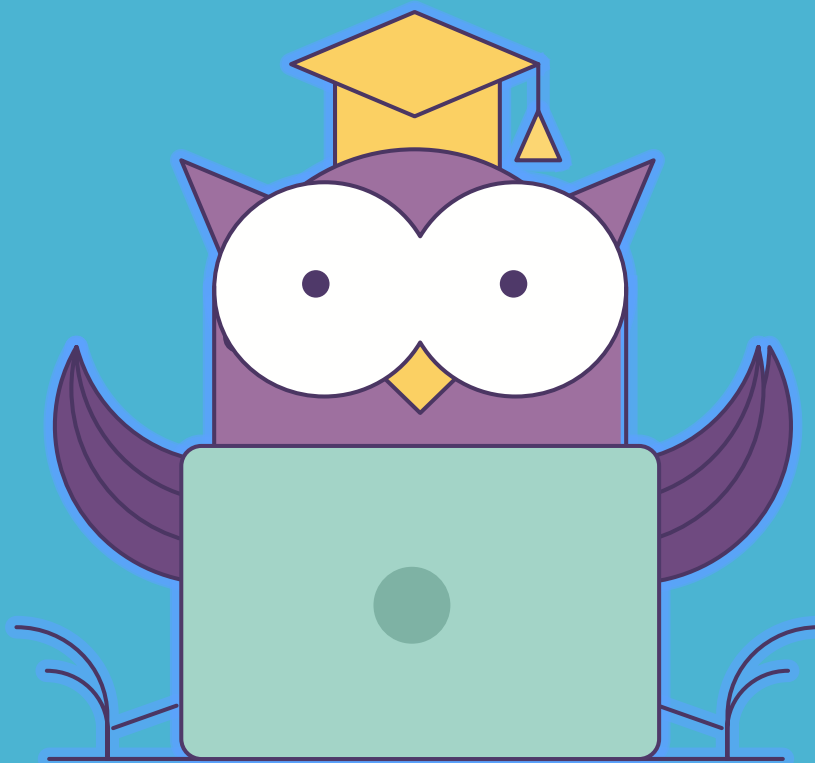
ОНЛАЙН-ОБРАЗОВАНИЕ

# Интерфейсы в Go

Артем Желтак



# Как меня слышно и видно?



## > Напишите в чат

+ если все хорошо

- если есть проблемы со звуком или с видео

!проверить запись!

- Определение и реализация интерфейсов
- Внутренняя структура интерфейсов
- Определение типа значения интерфейса
- Опасный и безопасный type cast
- Конструкций switch
- Где мои generic-и?

Интерфейс - это набор методов:

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

Dog удовлетворяет интерфейсу Duck

```
type Duck interface {  
    Talk() string  
    Walk()  
    Swim()  
}  
  
type Dog struct {  
    name string  
}  
  
func (d Dog) Talk() string {  
    return "AGGGRRRR"  
}  
  
func (d Dog) Walk() { }  
  
func (d Dog) Swim() { }
```

```
func quack(d Duck) {  
    print(d.Talk())  
}  
  
func main() {  
    quack(Dog{})  
}
```

```
type MyVeryOwnStringer struct { s string}

func (s MyVeryOwnStringer) String() string {
    return "my string representation of MyVeryOwnStringer"
}

func main() {
    fmt.Println(MyVeryOwnStringer{"hello"}) // my string representation of MyVeryOwnStringer{}
}
```

```
type Stringer interface {
    String() string
}
```



```
type Hound interface {  
    Hunt()  
}  
type Poodle interface {  
    Bark()  
}  
  
type GoldenRetriever struct{name string}  
  
func (GoldenRetriever) Hunt() { fmt.Println("hunt") }  
func (GoldenRetriever) Bark() { fmt.Println("bark") }  
  
func f1(i Hound) { i.Hunt() }  
func f2(i Poodle) { i.Bark() }  
  
func main() {  
    t := GoldenRetriever{"jack"}  
    f1(t) // "hunt"  
    f2(t) // "bark"  
}
```

```
type Poodle interface {
    Bark()
}

type ScandinavianClip struct{name string}
func (ScandinavianClip) Bark() { fmt.Println("bark") }

type ToyPoodle struct{name string}
func (ToyPoodle) Bark() { fmt.Println("bark") }

func main() {
    var t, sc Poodle

    t = ToyPoodle{"jack"}
    sc = ScandinavianClip{"jones"}

    t.Bark() // "bark"
    sc.Bark() // "bark"
}
```

- Интерфейс — набор методов, которые надо реализовать, чтобы удовлетворить интерфейсу. Ключевое слово `interface`.

```
type Stringer interface {  
    String() string  
}
```

- Тип интерфейс — переменная типа интерфейс, которая содержит значение типа, который реализует интерфейс.

```
var s Stringer
```

Интерфейс может встраивать другой интерфейс, определенный пользователем или импортируемый при помощи qualified name

```
import "fmt"

type Greeter interface {
    hello()
}

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

Пример из 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
}
```

```
type I interface {  
    J  
    i()  
}  
  
type J interface {  
    K  
    j()  
}  
  
type K interface {  
    k()  
    I  
}
```

```
interface type loop involving I
```

имена методов не должны повторяться:

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

то есть не содержать никаких методов:

```
interface{}
```

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



```
func PrintAll(vals []interface{}) {  
    for _, val := range vals {  
        fmt.Println(val)  
    }  
}  
func main() {  
    names := []string{"stanley", "david", "oscar"}  
    PrintAll(names)  
}
```

```
func PrintAll(vals []interface{}) {  
    for _, val := range vals {  
        fmt.Println(val)  
    }  
}  
func main() {  
    names := []string{"stanley", "david", "oscar"}  
    vals := make([]interface{}, len(names))  
    for i, v := range names { vals[i] = v }  
    PrintAll(vals)  
}
```

- это набор сигнатур методов
- который реализуется неявно
- интерфейсы могут встраивать другие интерфейсы
- имена методов не должны повторяться
- интерфейс может быть пустым (не иметь методов), такому интерфейсу удовлетворяет любой тип

<https://play.golang.org/p/U1V7tpVI9il>

Реализовать интерфейс Adult

состоит из динамического типа и значения  
мы можем их посмотреть при помощи %v и %T

```
type Temp int

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

func main() {
    var x fmt.Stringer
    x = Temp(24)
    fmt.Printf("%v %T\n", x, x) // 24 °C main.Temp
}
```

...или с помощью пакета reflect

```
import (  
    "fmt"  
    "reflect"  
)  
  
type MyError struct {}  
  
func (e MyError) Error() string {  
    return "smth happened"  
}  
  
func main() {  
    var e error  
    e = MyError{}  
  
    fmt.Println(reflect.TypeOf(e).Name()) // main MyError  
    fmt.Printf("%T\n", e) // // main MyError  
}
```

nil - нулевое значение для интерфейсного типа

```
type Shape interface {  
    Area() float64  
    Perimeter() float64  
}  
  
func main() {  
    var s Shape  
    fmt.Println("value of s is", s)  
    fmt.Printf("type of s is %T\n", s)  
}
```

```
value of s is <nil>  
type of s is <nil>
```

```
type Rect struct {  
    width float64  
    height float64  
}  
  
func (r Rect) Area() float64 {  
    return r.width * r.height  
}  
  
func (r Rect) Perimeter() float64 {  
    return 2 * (r.width + r.height)  
}  
  
func main() {  
    var s Shape  
    s = Rect{5.0, 4.0}  
    fmt.Printf("type of s is %T\n", s) // type of s is main.Rect  
    fmt.Printf("value of s is %v\n", s) // value of s is {5 4}  
    fmt.Println("area of rectange s", s.Area()) // area of rectange s 20  
}
```

Переменная типа интерфейс I может принимать значение любого типа, который реализует интерфейс I

```
type I interface {  
    method1()  
}  
  
type T1 struct{}  
func (T1) method1() {}  
  
type T2 struct{}  
func (T2) method1() {}  
func (T2) method2() {}  
  
func main() {  
    var i I = T1{}  
  
    i = T2{}  
    fmt.Println(i) //{}  
}
```



Значение интерфейсного типа равно nil тогда и только тогда, когда nil его статическая и динамическая части.

```
type I interface { M() }

type T struct {}
func (T) M() {}

func main() {
    var t *T
    if t == nil { fmt.Println("t is nil") } else {
        fmt.Println("t is not nil")
    }
    var i I = t
    if i == nil { fmt.Println("i is nil") } else {
        fmt.Println("i is not nil")
    }
}
```

```
t is nil
i is not 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)
    }
}
```

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

```
type Callable interface {  
    f() int  
}  
  
type T int  
  
func (t T) f() int {  
    return int(t)  
}  
  
var c Callable  
var t T  
c = t
```

<https://medium.com/golangspec/assignability-in-go-27805bcd5874>

```
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  
}
```

валидно?

Структура (вложенность) не имеет значения - v1 и v2 удовлетворяют I1, I2. Порядок методов также не имеет значения.

```
type I1 interface { M1(); M2() }

type I2 interface { M1(); I3 }

type I3 interface { M2() }

type T struct{}

func (T) M1() {}
func (T) M2() {}

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

валидно?

```
package main

type I1 interface { M1() }

type I2 interface { M1(); M2() }

type T struct{}

func (T) M1() {}

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

Что, если мы хотим присвоить переменной конкретного типа - значение типа интерфейс?

```
type I1 interface {  
    M1()  
}  
  
type T struct{}  
  
func (T) M1() {}  
  
func main() {  
    var v1 I1 = T{}  
    var v2 T = v1  
    _ = v2  
}
```

cannot use v1 (type I1) as type T in assignment: need type assertion

`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



```
var i interface{} = "hello"

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

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

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

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

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

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

проверка типа возможна только для интерфейса:

```
s := 5
i := s.(int)
```

```
Invalid type assertion: s.(int) (non-interface type int on left)
```

можем объединить проверку нескольких типов в один type switch:

```
type I1 interface { M1() }

type T1 struct{}
func (T1) M1() {}

type I2 interface { I1; M2() }

type T2 struct{}
func (T2) M1() {}
func (T2) M2() {}

func main() {
    var v I1
    switch v.(type) {
    case T1:
        fmt.Println("T1")
    case T2:
        fmt.Println("T2")
    case nil:
        fmt.Println("nil")
    default:
        fmt.Println("default")
    }
}
```

как и в обычном switch можем объединять типы:

```
case T1, T2:
    fmt.Println("T1 or T2")
}
```

и обрабатывать default:

```
var v I
switch v.(type) {
default:
    fmt.Println("fallback")
}
```

что-то такое происходит в пакете fmt:

```
type Stringer interface {
    String() string
}

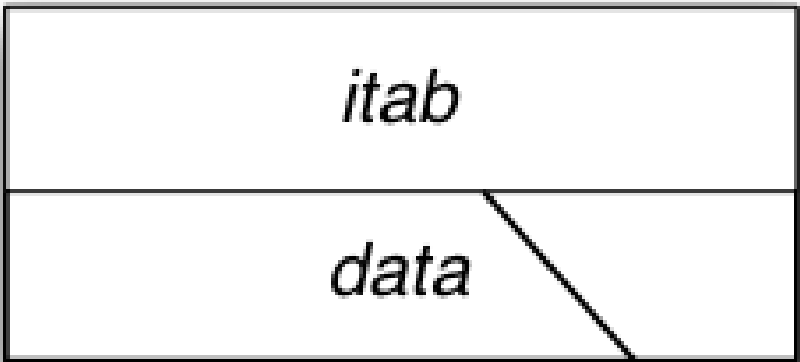
func ToString(any interface{}) string {
    if v, ok := any.(Stringer); ok {
        return v.String()
    }
    switch v := any.(type) {
    case int:
        return strconv.Itoa(v)
    case float:
        return strconv.Ftoa(v, 'g', -1)
    }
    return "???"
}
```

реализовать функцию zoo [https://play.golang.org/p/4zwgnjtDz\\_L](https://play.golang.org/p/4zwgnjtDz_L)

```
type Speaker interface {  
    SayHello()  
}  
  
type Human struct {  
    Greeting string  
}  
  
func (h Human) SayHello() {  
    fmt.Println(h.Greeting)  
}  
...  
var s Speaker  
h := Human{Greeting: "Hello"}  
s := Speaker(h)  
s.SayHello()
```

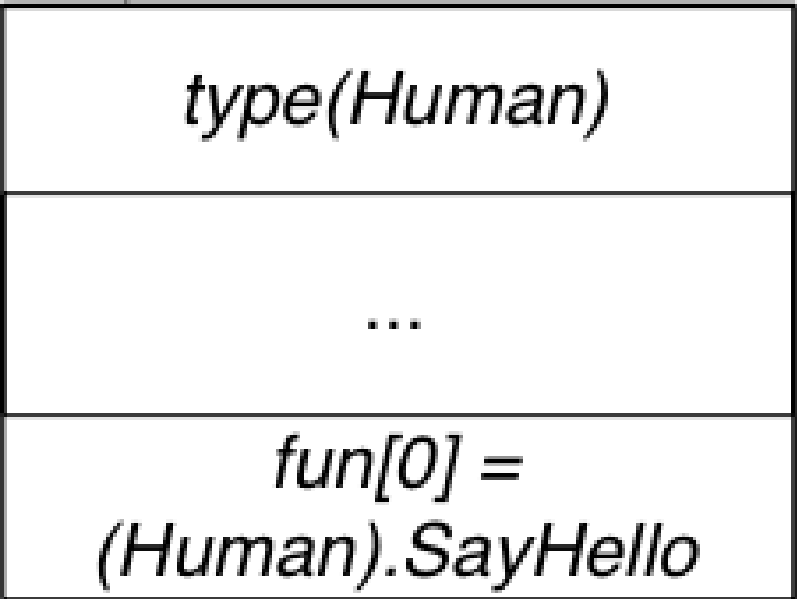
Human{}

s := Speaker(h)



Human{}

*itable(Speaker, Human)*





$s := (\text{interface}\{\}) (h)$

*type(Human)*

*data*

Human{}

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

```
type itab struct { // 40 bytes on a 64bit arch  
    inter *interfacetype  
    _type *_type  
    hash uint32 // copy of _type.hash. Used for type switches.  
    _     [4]byte  
    fun   [1]uintptr // variable sized. fun[0]==0 means _type does not implement inter.  
}
```

[https://github.com/teh-cmc/go-internals/blob/master/chapter2\\_interfaces/README.md](https://github.com/teh-cmc/go-internals/blob/master/chapter2_interfaces/README.md)

```
type Addifier interface{ Add(a, b int32) int32 }

type Adder struct{ id int32 }

func (adder Adder) Add(a, b int32) int32 { return a + b }

func BenchmarkDirect(b *testing.B) {
    adder := Adder{id: 6754}
    for i := 0; i < b.N; i++ {
        adder.Add(10, 32)
    }
}

func BenchmarkInterface(b *testing.B) {
    adder := Adder{id: 6754}
    for i := 0; i < b.N; i++ {
        Addifier(adder).Add(10, 32)
    }
}
```

```
go tool compile -m addifier.go
```

```
Addifier(adder) escapes to heap
```

```
→ addifier go test -bench=.
goos: darwin
goarch: amd64
pkg: strexpan/interfaces/addifier
BenchmarkDirect-8      2000000000      0.60 ns/op
BenchmarkInterface-8   1000000000     13.4 ns/op
PASS
ok      strexpan/interfaces/addifier  2.635s
```

interface type -> concrete type

```
type I interface {  
    M()  
}  
  
type T struct {}  
func (T) M() {}  
  
func main() {  
    var v I = T{}  
    fmt.Println(T(v))  
}
```

cannot convert v(`type` I) to `type` T: need `type` assertion

interface type -> interface type

```
type I1 interface {  
    M()  
}  
type I2 interface {  
    M()  
    N()  
}  
func main() {  
    var v I1  
    fmt.Println(I2(v))  
}
```

```
main.go:16: cannot convert v (type I1) to type I2:  
    I1 does not implement I2 (missing N method)
```

```
type I1 interface {  
    M1()  
}  
  
type T struct{}  
  
func (T) M1() {}  
  
func main() {  
    var v1 I1 = T{}  
    var v2 T = v1  
    _ = v2  
}
```

cannot convert v (type I) to type T: need type assertion

для обычных типов:

```
type I interface {  
    M()  
}  
  
type T struct{}  
  
func (T) M() {}  
  
func main() {  
    var v1 I = T{}  
    v2 := v1.(T)  
    fmt.Printf("%T\n", v2) // main.T  
}
```



для интерфейсов:

```
type I interface {
    M()
}

type T1 struct{}

func (T1) M() {}

type T2 struct{}

func main() {
    var v1 I = T1{}
    v2 := v1.(T2) // impossible type assertion:
                 // T2 does not implement I (missing M method)
    fmt.Printf("%T\n", v2)
}
```

динамические части не совпадают:

```
type I interface {  
    M()  
}  
  
type T1 struct{}  
func (T1) M() {}  
  
type T2 struct{}  
func (T2) M() {}  
  
func main() {  
    var v1 I = T1{}  
    v2 := v1.(T2)  
    fmt.Printf("%T\n", v2)  
}
```

```
panic: interface conversion: main.I is main.T1, not main.T2
```

Можем проверить, выполняется ли приведение при помощи multi-valued type assertion:

```
type I interface { M() }

type T1 struct{}
func (T1) M() {}

type T2 struct{}
func (T2) M() {}

func main() {
    var v1 I = T1{}
    v2, ok := v1.(T2)
    if !ok {
        fmt.Printf("ok: %v\n", ok) // ok: false
        fmt.Printf("%v, %T\n", v2, v2) // {}, main.T2
    }
}
```

```
type I1 interface { M() }

type I2 interface { I1; N() }

type T struct{
    name string
}

func (T) M() {}
func (T) N() {}

func main() {
    var v1 I1 = T{"foo"}
    var v2 I2
    v2, ok := v1.(I2)
    fmt.Printf("%T %v %v\n", v2, v2, ok) // main.T {foo} true
}
```

```
type I1 interface {
    M()
}

type I2 interface {
    N()
}

type T struct {}

func (T) M() {}

func main() {
    var v1 I1 = T{}
    var v2 I2
    v2, ok := v1.(I2)
    fmt.Printf("%T %v %v\n", v2, v2, ok) // <nil> <nil> false
}
```

nil всегда паникует

```
type I interface {  
    M()  
}  
  
type T struct{}  
  
func (T) M() {}  
  
func main() {  
    var v1 I  
    v2 := v1.(T)  
    fmt.Printf("%T\n", v2)  
}
```

```
panic: interface conversion: main.I is nil, not main.T
```

есть: map, slice, etc.

<https://go.dev/doc/faq/whats-new-to-go2#go2draft-generics-overview>

Чтобы реализовать общие алгоритмы мы можем воспользоваться интерфейсами:

```
type Interface interface {  
    // Len is the number of elements in the collection.  
    Len() int  
    // Less reports whether the element with  
    // index i should sort before the element with index j.  
    Less(i, j int) bool  
    // Swap swaps the elements with indexes i and j.  
    Swap(i, j int)  
}
```



```
type Person struct {
    Name string
    Age  int
}
// ByAge implements sort.Interface for []Person based on
// the Age field.
type ByAge []Person

func (a ByAge) Len() int           { return len(a) }
func (a ByAge) Swap(i, j int)      { a[i], a[j] = a[j], a[i] }
func (a ByAge) Less(i, j int) bool { return a[i].Age < a[j].Age }
...
people := []Person{
    {"Bob", 31},
    {"John", 42},
    {"Michael", 17},
    {"Jenny", 26},
}

sort.Sort(ByAge(people))
```

Спасибо за внимание!

