Go 语言接口

接口类型是一种抽象的类型。它不会暴露出它所代表的对象的内部值的结构和这个对象支持的基础操作的集合；它们只会表现出它们自己的方法。也就是说当你有看到一个接口类型的值时，你不知道它是什么，唯一知道的就是可以通过它的方法来做什么。

接口（interface）定义了一个对象的行为规范，只定义规范不实现，由具体的对象来实现规范的细节。

接口是一种新的**类型定义**，它把所有的**具有共性的方法**定义在一起，任何其他类型只要实现了这些方法就是实现了这个接口。

接口（interface）是一种类型，一种抽象的类型。

接口做的事情就像是定义一个协议（规则），只要一台机器有洗衣服和甩干的功能，我就称它为洗衣机。不关心属性（数据），只关心行为（方法）。

为什么要使用接口

```
type Cat struct{}

func (c Cat) Say() string { return "喵喵喵" }

type Dog struct{}

func (d Dog) Say() string { return "汪汪汪" }

func main() {
    c := Cat{}
    fmt.Println("猫:", c.Say())
    d := Dog{}
    fmt.Println("狗:", d.Say())
}
```

上面的代码中定义了猫和狗，然后它们都会叫，你会发现main函数中明显有重复的代码，如果我们后续再加上猪、青蛙等动物的话，我们的代码还会一直重复下去。那我们能不能把它们当成“能叫的动物”来处理呢？

像类似的例子在我们编程过程中会经常遇到：

比如一个网上商城可能使用支付宝、微信、银联等方式去在线支付，我们能不能把它们当成“支付方式”来处理呢？

比如三角形，四边形，圆形都能计算周长和面积，我们能不能把它们当成“图形”来处理呢？

比如销售、行政、程序员都能计算月薪，我们能不能把他们当成“员工”来处理呢？

Go语言中为了解决类似上面的问题，就设计了接口这个概念。接口区别于我们之前所有的具体类型，接口是一种抽象的类型。当你看到一个接口类型的值时，你不知道它是什么，唯一知道的是通过它的方法能做什么。

> 只有当有两个或两个以上的具体类型必须以相同的方式进行处理时才需要定义接口。不要为了接口而写接口，那样只会增加不必要的抽象，导致不必要的运行时损耗。

接口的定义

每个接口由数个方法组成，接口的定义格式如下：

```
type 接口类型名 interface{
    方法名1( 参数列表1 ) 返回值列表1
    方法名2( 参数列表2 ) 返回值列表2
    …
}
```

- 接口名：使用type将接口定义为自定义的类型名。Go语言的接口在命名时，一般会在单词后面添加er，如有写操作的接口叫Writer，有字符串功能的接口叫Stringer等。接口名最好要能突出该接口的类型含义。
- 方法名：当方法名首字母是大写且这个接口类型名首字母也是大写时，这个方法可以被接口所在的包（package）之外的代码访问。
- 参数列表、返回值列表：参数列表和返回值列表中的参数变量名可以省略。

> 实现接口必须实现接口中所有方法

示例：

```
package main

import "fmt"

// Sayer 接口
type Sayer interface {
    say()
}

//定义dog和cat两个结构体：
type dog struct{}

type cat struct{}

// dog实现了Sayer接口
func (d dog) say() {
    fmt.Println("汪汪汪")
}

// cat实现了Sayer接口
func (c cat) say() {
    fmt.Println("喵喵喵")
}

func main() {
    var x Sayer // 声明一个Sayer类型的变量x
    a := cat{}  // 实例化一个cat
    b := dog{}  // 实例化一个dog
    x = a       // 可以把cat实例直接赋值给x
    x.say()     // 喵喵喵
    x = b       // 可以把dog实例直接赋值给x
    x.say()     // 汪汪汪
}
```

```
#结果
喵喵喵
汪汪汪
```

接口值接收者和指针接收者

本质上和**方法**的值类型接收者和指针类型接收者的思考方法是一样的，值接收者是一个拷贝，是一个副本，而指针接收者，传递的是指针。

示例：

```
package main

import "fmt"

//定义Mover接口
type Mover interface {
    move(name string)
}

//定义dog结构体
type dog struct{}

//值接收者实现接口
func (d dog) move(name string) {
    fmt.Println(name + "会旋转跳跃后空翻")
}

//定义cat结构体
type cat struct{}

//指针接收者实现接口
func (c *cat) move(name string) {
    fmt.Println(name + "会撒娇卖萌")
}

func main() {
    var x Mover
    var wangcai = dog{} // 旺财是dog类型
    x = wangcai          // x可以接收dog类型
    x.move("旺财")
    var fugui = &dog{} // 富贵是*dog类型
    x = fugui           // x可以接收*dog类型
    x.move("富贵")

    var mimi = cat{} //咪咪是cat类型
    // x = mimi          //x不能接收cat值类型  必须 x=&mimi
    x = &mimi
    x.move("咪咪")
    var miaomiao = &cat{} //喵喵是cat类型
    x = miaomiao           // x可以接收*cat类型
    miaomiao.move("喵喵")
}
#结果
旺财会旋转跳跃后空翻
```

```
富贵会旋转跳跃后空翻
咪咪会撒娇卖萌
喵喵会撒娇卖萌
```

类型与接口的关系

- 一个类型可以实现多个接口
- 多个类型可以实现同一个接口（多态）
- 在 Go 语言中，接口和类型之间是多对多的关系，即一个类型可以实现多个接口，同时，一个接口也可以被多个类型所实现。

一个类型实现多个接口

一个类型可以同时实现多个接口，而接口间彼此独立，不知道对方的实现。 例如，狗可以叫，也可以动。我们就分别定义Sayer接口和Mover接口。

示例：

```
package main

import "fmt"

//定义Sayer 接口
type Sayer interface {
    say()
}

//定义Mover接口
type Mover interface {
    move()
}

//定义dog结构体
type dog struct {
    name string
}

// 实现Sayer接口
func (d dog) say() {
    fmt.Printf("%s会叫汪汪汪\n", d.name)
}

//实现Mover接口
func (d dog) move() {
    fmt.Printf("%s会旋转跳跃后空翻\n", d.name)
}

func main() {
    var x Sayer
    var y Mover

    var wangcai = dog{name: "旺财"}
    x = wangcai  //旺财实现了Sayer接口
    y = wangcai  //旺财实现了Mover接口
    x.say()
```

```
    y.move()
}
#结果
旺财会叫汪汪汪
旺财会旋转跳跃后空翻
```

多个类型实现同一接口

Go语言中不同的类型还可以实现同一接口。

```
package main

import "fmt"

//定义Sayer 接口
type Sayer interface {
    say()
}

//定义dog结构体
type dog struct {
    name string
}

//定义cat结构体
type cat struct {
    name string
}

// dog实现Sayer接口
func (d dog) say() {
    fmt.Printf("%s会叫汪汪汪\n", d.name)
}

// cat实现Sayer接口
func (c cat) say() {
    fmt.Printf("%s会叫喵喵喵\n", c.name)
}

func main() {
    var x Sayer

    var wangcai = dog{name: "旺财"}
    var mimi = cat{name: "咪咪"}
    x = wangcai //旺财实现了Sayer接口
    x.say()

    x = mimi //咪咪实现了Sayer接口
    x.say()
}
#结果
旺财会叫汪汪汪
咪咪会叫喵喵喵
```

接口嵌套

接口与接口间可以通过嵌套创造出新的接口，嵌套得到的接口的使用与普通接口一样。

一个接口可以包含一个或多个其他的接口，这相当于直接将这些内嵌接口的方法列举在外层接口中一样。

只有实现接口中所有的方法，包括被包含的接口的方法，才算是实现了接口。

```
// Sayer 接口
type Sayer interface {
    say()
}

// Mover 接口
type Mover interface {
    move()
}

// 接口嵌套
type animal interface {
    Sayer
    Mover
}
```

示例：

```
package main

import "fmt"

//定义Sayer 接口
type Sayer interface {
    say()
}

//定义Mover 接口
type Mover interface {
    move()
}

// 接口嵌套 Animal包含Sayer和Mover接口
type Animal interface {
    Sayer
    Mover
}

//定义dog结构体
type dog struct {
    name string
}

//定义cat结构体
type cat struct {
    name string
```

```
}

// dog实现Sayer接口
func (d dog) say() {
    fmt.Printf("%s会叫汪汪汪\n", d.name)
}

// cat实现Sayer接口
func (c cat) say() {
    fmt.Printf("%s会叫喵喵喵\n", c.name)
}

// dog实现Mover接口
func (d dog) move() {
    fmt.Printf("%s会叫后空翻\n", d.name)
}

// cat实现Mover接口
func (c cat) move() {
    fmt.Printf("%s会叫撒娇卖萌\n", c.name)
}

func main() {
    var x Animal

    var wangcai = dog{name: "旺财"}
    var mimi = cat{name: "咪咪"}

    x = wangcai //旺财实现了Sayer接口
    x.say()
    x.move()

    x = mimi //咪咪实现了Sayer接口
    x.say()
    x.move()
}
#结果
旺财会叫汪汪汪
旺财会叫后空翻
咪咪会叫喵喵喵
咪咪会叫撒娇卖萌
```

空接口

空接口是指没有定义任何方法的接口。因此任何类型都实现了空接口。

空接口类型的变量可以存储任意类型的变量。

示例:

```
package main

import "fmt"

func main() {
```

```
    // 定义一个空接口x
    var x interface{}
    s := "包子"
    x = s
    fmt.Printf("type:%T value:%v\n", x, x)
    i := 100
    x = i
    fmt.Printf("type:%T value:%v\n", x, x)
    b := true
    x = b
    fmt.Printf("type:%T value:%v\n", x, x)
}
#结果
type:string value:包子
type:int value:100
type:bool value:true
```

空接口的应用

空接口作为函数的参数

使用空接口实现可以接收任意类型的函数参数。

```
// 空接口作为函数参数
func show(a interface{}) {
    fmt.Printf("type:%T value:%v\n", a, a)
}
```

空接口作为map的值

使用空接口实现可以保存任意值的字典。

```
package main

import "fmt"

func main() {
    // 空接口作为map值
    var studentInfo = make(map[string]interface{})
    studentInfo["name"] = "包子"
    studentInfo["age"] = 18
    fmt.Println(studentInfo)
}
#结果
map[age:18 name:包子]
```

空接口比较

Go 语言中的空接口在保存不同的值后，可以和其他变量一样使用 `==` 进行比较操作。

- 类型不同的空接口间的比较结果不相同
- 不能比较空接口中的动态值

类 型	说 明
map集合	不可比较，如果比较，程序会报错
切片	不可比较，如果比较，程序会报错
通道(channel)	可比较，必须由同一个 make 生成，也就是同一个通道才会是 **true**，否则为 false
数组	可比较，编译期知道两个数组是否一致
结构体	可比较，可以逐个比较结构体的值
函数	可比较

示例：

```
package main

import "fmt"

func main() {
    // 两个类型相同值相同的接口
    var intvalue1 interface{} = 1024
    var intvalue2 interface{} = 1024
    fmt.Println(intvalue1 == intvalue2)
    // 比较两个类型相同值不相同的接口结果为 false
    var intvalue3 interface{} = 1024
    var intvalue4 interface{} = 10
    fmt.Println(intvalue3 == intvalue4)
    // 比较两个类型不同的接口结果为 false
    var intvalue5 interface{} = 1024
    var intvalue6 interface{} = 1024.0
    fmt.Println(intvalue5 == intvalue6)
    // 空接口中的动态值不能比较 这里报错
    var sliceValue1 interface{} = []int{1024}
    var sliceValue2 interface{} = []int{1024}
    fmt.Println(sliceValue1 == sliceValue2)
}
#结果
true
false
false
panic: runtime error: comparing uncomparable type []int
```

类型断言

因为在 Go语言中，接口变量的动态类型是变化的，有时我们需要知道一个接口变量的动态类型究竟是什么，这就需要使用类型断言，类型断言就是对接口变量的类型进行检查。

接口值

一个接口的值（简称接口值）是由一个具体类型和具体类型的值两部分组成的。这两部分分别称为接口的动态类型和动态值。

想要判断空接口中的值这个时候就可以使用类型断言。

语法格式

```
x.(T)
```

`x`：表示类型为interface{}的变量
`T`：表示断言x可能是的类型。

该语法返回两个参数，第一个参数是x转化为T类型后的变量，第二个值是一个布尔值，若为true则表示断言成功，为false则表示断言失败。

实际完整写法为：

```
value, ok := x.(T)
```

参数

参数	描述
x	接口类型变量。
T	需要断言（判断）的类型。

返回值

返回值	描述
value	断言后的返回的数据。
ok	断言成功与否的bool变量表示。

示例：

```
func main() {
    var x interface{}
    x = "hello word! "
    v, ok := x.(string)
    if ok {
        fmt.Println(v)
    } else {
        fmt.Println("类型断言失败")
    }
}
```

如果要断言多次就需要写多个if判断，这个时候我们可以使用switch语句来实现：

```
func justifyType(x interface{}) {
    switch v := x.(type) {
    case string:
        fmt.Printf("x is a string, value is %v\n", v)
    case int:
        fmt.Printf("x is a int is %v\n", v)
    case bool:
        fmt.Printf("x is a bool is %v\n", v)
    default:
        fmt.Println("unsupport type! ")
    }
}
```