面向对象编程——接口

目录：

一、什么是接口?

(一)、概念

1、面向对象语言中，接口用于定义对象的行为。接口只指定对象应该做什么，实现这种行为的方法(实现细节)是由对象来决定。

2、在Go语言中，接口是一组方法签名。

- 接口只指定了类型应该具有的方法，类型决定了如何实现这些方法。
- 当某个类型为接口中的所有方法提供了具体的实现细节时，这个类型就被称为实现了该接口。
- 接口定义了一组方法，如果某个对象实现了该接口的所有方法，则此对象就实现了该接口。

3、Go语言的类型都是隐式实现接口的。任何定义了接口中所有方法的类型都被称为隐式地实现了该接口。

（二）、接口的定义语法及示例

1、定义接口

```
type 接口名字 interface {
  方法1([参数列表]) [返回值]
```

```
  方法2([参数列表]) [返回值]
  ...
  方法n([参数列表]) [返回值]
}
```

2、定义结构体

```
type 结构体名 struct {
  //属性
}
```

3、实现接口方法

```
func (变量名  结构体类型) 方法1([参数列表]) [返回值] {
  //方法体
}

func (变量名  结构体类型) 方法2([参数列表]) [返回值] {
  //方法体
}
...
func (变量名  结构体类型) 方法n([参数列表]) [返回值] {
  //方法体
}
```

4、示例代码：

```
package main
import "fmt"

type Phone interface {
  call()
}

type AndroidPhone struct {
}
```

```
type IPhone struct {
}

func (a  AndroidPhone) call() {
  fmt.Println("我是安卓手机，我可以打电话!")
}

func (i  IPhone) call() {
  fmt.Println("我是苹果手机，我可以打电话!")
}

func main() {
  //定义接口类型的变量
  var phone Phone
  phone = new(AndroidPhone)
  phone.call()

  phone = new(IPhone)
  phone.call()
}
```

运行结果：

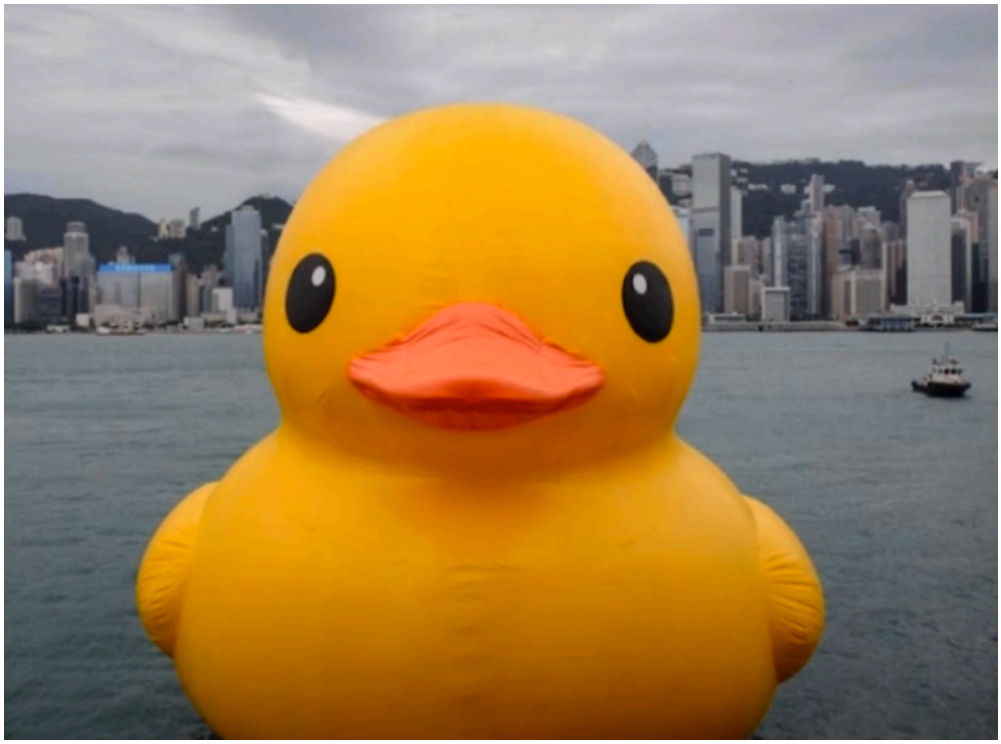
```
我是安卓手机，我可以打电话!
我是苹果手机，我可以打电话!
```

【思考：】

- 并没有见到上述案例中出现AndroidPhone及iPhone实现接口Phone的语句。那么为什么 new(AndroidPhone)以及new(IPhone)可以直接赋值给接口变量phone呢？——隐式实现接口
- go没有 implements, extends 关键字
- 其实这种编程语言叫做duck typing编程语言。

(三)、duck typing

- 编程语言中的鸭子类型



1、大黄鸭是鸭子吗?

- 鸭子：脊索动物门、脊椎动物亚门、鸟纲雁形目
  - 大黄鸭无生命：不是鸭子
- duck typing
  - When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.
  - "当看到一只鸟走起来像鸭子、游泳起来像鸭子、叫起来也像鸭子，那么这只鸟就可以被称为鸭子。"
  - 扩展后，可以理解为：“看起来像鸭子，那么它就是鸭子”。
  - 描述事物的外部行为而非内部结构。
  - 在鸭子类型中，关注的不是对象的类型本身，而是它是如何使用的。

2、duck typing编程语言

- 一般来讲，使用 duck typing 的编程语言往往被归类到“动态类型语言”或者“解释型语言”里，比如 Python, Javascript, Ruby 等等；
- 而非duck typing语言往往被归到“静态类型语言“中，比如 C/C++/Java。

3、非duck typing语言

- 以 Java为例， 一个类必须显式地声明：“类实现了某个接口”， 然后才能用在这个接口可以使用的地方。
- 如果你有一个第三方的 Java 库，这个库中的某个类没有声明它实现了某个接口，那么即使这个类中真的有那些接口中的方法，你也不能把这个类的对象用在那些要求用接口的地方。
- 但如果在duck typing的语言中， 你就可以这样做，因为它不要求一个类显式地声明它实现了某个接口。

4、动态类型的好处

- 动态类型的好处很多，Python代码写起来很快。但是缺陷也是显而易见的：错误往往要在运行时才能被发现。
- 相反，静态类型语言往往在编译时就是发现这类错误：如果某个变量的类型没有显式声明实现了某个接口，那么，这个变量就不能用在要求一个实现了这个接口的地方。

5、Go 类型系统采取了折中的办法：

- 静态类型语言
- 之所以说这是一种折中的办法，原因如下：
  - 第一，结构体类型T不需要显式地声明它实现了接口 I。只要类型 T 实现了接口 I 规定的所有方法，它就自动地实现了接口 I。 这样就像动态语言一样省了很多代码，少了许多限制。
  - 第二，将结构体类型的变量显式或者隐式地转换为接口 I 类型的变量 i。这样就可以和其它静态类型语言一样，在编译时检查参数的合法性。

7、示例代码：

```
package main
import "fmt"

type ISayHello interface {
  SayHello()
}

type Person struct{}
type Duck struct{}
```

```go
type Duck2 struct{}

func (person Person) SayHello() {
  fmt.Printf("Hello!")
}

func (duck Duck) SayHello() {
  fmt.Printf("ga ga ga!")
}

func greeting(i ISayHello) {
  i.SayHello()
}

func main() {
  //person := Person{}
  //duck := Duck{}
  person := new(Person)
  duck := new(Duck)

  //以下输出跟接口没有关系
  fmt.Println("非接口调用形式")
  person.SayHello()
  duck.SayHello()
  fmt.Println("\n---------------------")

  //定义接口变量。
 fmt.Println("接口调用形式")
  var i ISayHello
  i = person
  greeting(i)

  i = duck
  greeting(i)
```

```
    //可否将一个未实现接口方法的结构体对象赋值给接口呢?
    //i = new(Duck2)
}
```

运行结果：

非接口调用形式
Hello!ga ga ga!

接口调用形式
Hello!ga ga ga!

【备注：接口的用法】

- 用法一：一个函数如果接收接口类型作为参数，那么实际上可以传入该接口的任意实现类对象作为参数。
- 用法二：定义一个接口变量，那么实际上可以赋值任意实现了该接口的对象。

如果定义了一个接口类型的容器（数组或切片），实际上该容器中可以存储任意的实现类对象。

（四）、 多态：

1、概念：

- 事物的多种形态
- Go中的多态性是在接口的帮助下实现的。定义接口类型，创建实现该接口的结构体对象。
- 定义接口类型的对象，可以保存实现该接口的任何类型的值。Go语言接口变量的这个特性实现了Go语言中的多态性。
- 接口类型的对象，不能访问其实现类中的属性字段。

2、多态示例代码：

```
package main
import "fmt"
```

```go
type Income interface {
  calculate() float64
  source() string
}

//固定账单项目
type FixedBilling struct {
  projectName  string
  biddedAmount float64 //招标总额
}

//定时和材料项目（定时生产项目）
type TimeAndMaterial struct {
  projectName string
  workHours   float64 //工作时长
  hourlyRate  float64 //每小时工资率
}

//固定收入项目
func (fb FixedBilling) calculate() float64 {
  return fb.biddedAmount
}

func (fb FixedBilling) source() string {
  return fb.projectName
}

//定时生产项目
func (tm TimeAndMaterial) calculate() float64 {
  return tm.workHours * tm.hourlyRate
}

func (tm TimeAndMaterial) source() string {
  return tm.projectName
}
```

```
//假设该组织通过广告找到了新的收入来源。让我们看看如何简单地添加新的收入方式和计算总收入，而不用对calculateNetIncome函数做任何更改。由于多态性，这样是可行的。
//首先让我们定义Advertisement类型和calculate()和source()方法。
//广告类型有三个字段adName, costPerclick(每次点击的花费，cost per click)。
type Advertisement struct {
  adName      string
  costPerclick float64
  clickCount   int
}

func (a Advertisement) calculate() float64 {
  return a.costPerclick * float64(a.clickCount)
}

func (a Advertisement) source() string {
  return a.adName
}

//计算和打印总收入的calculateNetIncome函数
func calculateNetIncome(ic []Income) {
  netincome := 0.0
  for _, income := range ic {
    fmt.Printf("收入来源： %s = $%.2f \n", income.source(),
income.calculate())
    netincome += income.calculate()
  }
  fmt.Printf("公司净收入合计 = $%.2f ", netincome)
}

func main() {
  project1 := FixedBilling{projectName: "项目1", biddedAmount: 5000}
  project2 := FixedBilling{projectName: "项目2", biddedAmount: 10000}
  project3 := TimeAndMaterial{projectName: "项目3", workHours: 100,
hourlyRate: 40}
  project4 := TimeAndMaterial{projectName: "项目4", workHours: 250,
hourlyRate: 20}
```

```
    project5 := Advertisement{adName: "广告5", costPerclick: 0.1, clickCount:
10000}
    incomeStreams := []Income{project1, project2, project3, project4, project5}
    calculateNetIncome(incomeStreams)
}

//说明:
// 没有对calculateNetIncome函数做任何更改，尽管添加了新的收入方式。全靠多
态性而起作用。
// 由于新的Advertisement类型也实现了Income接口，可以将它添加到
incomeStreams切片中。
// calculateNetIncome函数也在没有任何更改的情况下工作，因为它可以调用
Advertisement类型的calculate()和source()方法。
```

（五）、空接口

1、概念

- 空接口：该接口中没有任何的方法。任意类型都可以实现该接口。
- 空interface这样定义：interface{}，也就是包含0个method的interface。
- 用空接口表示任意数据类型。类似于java中的object。
- 空接口常用于以下情形：
    - 1、println的参数就是空接口
    - 2、定义一个map：key是string，value是任意数据类型
    - 3、定义一个切片，其中存储任意类型的数据

2、示例代码：

```
package main
import "fmt"

//定义空接口
type A interface {
}

type Cat struct {
    name string
```

```
    age  int
}

type Person struct {
    name string
    sex  string
}

func main() {
    // 用空接口表示任意数据类型。类似于java中的object
    var a1 A = Cat{name: "Mimi", age: 1}
    var a2 A = Person{"Steven", "man"}
    var a3 A = "Learn golang with me!"
    var a4 A = 100
    var a5 A = 3.14

    fmt.Printf("%T，%v \n", a1, a1)
    fmt.Printf("%T，%v \n", a2, a2)
    fmt.Printf("%T，%v \n", a3, a3)
    fmt.Printf("%T，%v \n", a4, a4)
    fmt.Printf("%T，%v \n", a5, a5)
    fmt.Println("-----------------------")

    //1、println的参数就是空接口
    fmt.Println("println的参数可以是任何数据类型，用空接口表示\n", 100,
3.14, Cat{"小天", 2})

    //2、定义一个map：key是string，value是任意数据类型
    map1 := make(map[string]interface{})
    map1["name"] = "Daniel"
    map1["age"] = 13
    fmt.Println(map1)
    fmt.Println("-----------------------")
```

```
    //3、定义一个切片，其中存储任意类型的数据
    slice1 := make([]interface{}, 0, 10)
    slice1 = append(slice1, a1, a2, a3, a4, a5)
    fmt.Println(slice1)

    testInterface(slice1)
  }

  func testInterface(s []interface{}) {
    for i := range s {
      fmt.Println("第", i+1, "个数据：")
      switch ins := s[i].(type) {
      case Cat:
        fmt.Println("\tcat对象：", ins.name, ins.age)
      case Person:
        fmt.Println("\tperson对象：", ins.name, ins.sex)
      case int:
        fmt.Println("\tint类型：", ins)
      case string:
        fmt.Println("\tstring类型：", ins)
      case float64:
        fmt.Println("\tfloat64类型：", ins)
      }
    }
  }
```

（六）、接口对象转型

1、方式一：

- instance，ok := 接口对象.(实际类型)
- 如果该接口对象是对应的实际类型，那么instance就是转型之后对象，ok的值为true
- 配合if ... else if...语句使用

2、方式二：

- 接口对象.(type)
- 配合switch...case语句使用

3、示例代码：

```
package main
import (
  "fmt"
  "math"
)

//1.定义一个接口
type Shape interface {
  perimeter() float64
  area() float64
}

//2.矩形
type Rectangle struct {
  a, b float64
}

//3.三角形
type Triangle struct {
  a, b, c float64
}

//4.圆形
type Circle struct {
  radius float64 //半径
}

//实现接口的方法
func (r Rectangle) perimeter() float64 {
  return 2 * (r.a + r.b)
}
```

```go
func (r Rectangle) area() float64 {
  return r.a * r.b
}

func (t Triangle) perimeter() float64 {
  return t.a + t.b + t.c
}

func (t Triangle) area() float64 {
  p := t.perimeter() / 2 //半周长
  //海伦公式
  s := math.Sqrt(p * (p - t.a) * (p - t.b) * (p - t.c))
  return s
}

func (c Circle) perimeter() float64 {
  return 2 * math.Pi * c.radius
}

func (c Circle) area() float64 {
  return math.Pow(c.radius, 2) * math.Pi
}

//测试函数
func testShape(s Shape) {
  fmt.Printf("周长：%.2f ， 面积：%.2f \n", s.perimeter(), s.area())
}

func main() {
  var s Shape
  s = Rectangle{3, 4}
  testShape(s)

  s = Triangle{3, 4, 5}
  testShape(s)
```

```
    s = Circle{1}
    testShape(s)
}
//接口对象转型——方式1
func getType(s Shape) {
    if instance, ok := s.(Rectangle); ok {
        fmt.Printf("矩形：长度为%.2f， 宽为%.2f， \t", instance.a,
instance.b)
    } else if instance, ok := s.(Triangle); ok {
        fmt.Printf("三角形：三边分别为%.2f， %.2f， %.2f， \t", instance.a,
instance.b, instance.c)
    } else if instance, ok := s.(Circle); ok {
        fmt.Printf("圆形：半径为%.2f， \t", instance.radius)
    }
}

//接口对象转型——方式2
func getType2(s Shape) {
    switch instance := s.(type) {
    case Rectangle:
        fmt.Printf("矩形：长度为%.2f， 宽为%.2f， \t", instance.a,
instance.b)
    case Triangle:
        fmt.Printf("三角形：三边分别为%.2f， %.2f， %.2f， \t", instance.a,
instance.b, instance.c)
    case Circle:
        fmt.Printf("圆形：半径为%.2f， \t", instance.radius)
    }
}
```

（七）、系统内置接口

1、fm包下的Stringer()接口

- 实现对象的格式化打印

- 示例代码：

```
//系统接口fmt.Stringer()
func (p Triangle) String() string {
  //fmt.Stringer()
  return fmt.Sprintf("Triangle对象 —— 属性分别为：%.2f，%.2f， %.2f
\n", p.a, p.b, p.c)
}
```