Golang之interface

一、什么是interface

简单地说，interface是一组method的组合，可以通过interface来定义对象的一组行为。

二、interface类型

interface类型定义了一组方法，如果某个对象实现了某个接口的所有方法，则此对象就实现了此接口。详细语法参考如下例子：

```
type Human struct {
    name string
    age int
    phone string
}
type Student struct {
    Human        //匿名字段Human
    school string
    loan float32
}
type Employee struct {
    Human        //匿名字段Human
    company string
    money float32
}
//Human对象实现Sayhi方法
func (h *Human) SayHi() {
    fmt.Printf("Hi, I am %s you can call me on %s\n", h.name, h.phone)
}
//Human对象实现Sing方法
func (h *Human) Sing(lyrics string) {
    fmt.Println("La la, la la la, la la la la...", lyrics)
}
//Human对象实现Guzzle方法
func (h *Human) Guzzle(beerStein string) {
    fmt.Println("Guzzle Guzzle Guzzle...", beerStein)
}
//Employee重载Human的Sayhi方法
func (e *Employee) SayHi() {
    fmt.Printf("Hi, I am %s, I work at %s. Call me on %s\n", e.name,
e.company, e.phone)
}
//Student实现BorrowMoney方法
func (s *Student) BorrowMoney(amount float32) {
    s.loan += amount
}
//Employee实现SpendSalary方法
func (e *Employee) SpendSalary(amount float32) {
    e.money -= amount
}
//定义interface
type Men interface {
```

```
    SayHi()
    Sing(lyrics string)
    Guzzle(beerStein string)
}
type YoungChap interface {
    SayHi()
    Sing(song string)
    BorrowMoney(amount float32)
}
type ElderlyGent interface {
    SayHi()
    Sing(song string)
    SpendSalary(amount float32)
}
```

从上面代码可知，interface可以被任意的对象实现，比如：Men interface被Human、Student和Employee实现。同理，一个对象可以实现任意多个interface，比如：Student实现了Men和YonggChap两个interface。

任意类型都实现了空interface（我们这样定义：interface{}），即包含0个method的interface。

三、interface值

如果我们定义了一个interface的变量，那么这个变量可以存储实现了这个interface的任意类型的对象。比如：上例中定了一个Men interface类型的变量m，那么m里面可以存储Human、Student或者Employee值。下面举例说明如何使用：因为m能够持有这三种类型的对象，所以可以定义一个包含Men类型元素的slice，这个slice可以被赋予实现了Men接口的任意结构的对象：

```
package main
import "fmt"
type Human struct {
    name string
    age int
    phone string
}
type Student struct {
    Human        //匿名字段
    school string
    loan float32
}
type Employee struct {
    Human        //匿名字段
    company string
    money float32
}
//Human实现Sayhi方法
func (h Human) SayHi() {
    fmt.Printf("Hi, I am %s you can call me on %s\n", h.name, h.phone)
}
//Human实现Sing方法
func (h Human) Sing(lyrics string) {
    fmt.Println("La la la la...", lyrics)
}
//Employee重载Human的SayHi方法
func (e Employee) SayHi() {
    fmt.Printf("Hi, I am %s, I work at %s. Call me on %s\n",
```

```
e.name.e.company, e.phone)
}
//Interface Men被Human、Student和Employee实现
//因为这三个类型都实现了这了两个方法
type Men interface {
    SayHi()
    Sing(lyrics string)
}
func main() {
    mike := Student{Human{"Mike", 25, "222-222-xxx"}, "MIT", 0.00}
    paul := Student{Human{"Paul", 26, "111-222-xxx"}, "Harvard", 100}
    sam := Employee{Human{"Sam", 36, "444-222-xxx"}, "Golang Inc.", 1000}
    Tom := Employee{Human{"Sam", 36, "444-222-xxx"}, "Things Ltd.", 5000}
    //定义Men类型的变量i
    var i Men
    //i能存储Student
    i = mike
    fmt.Println("This is Mike, a Student:")
    i.SayHi()
    i.Sing("November rain")
    //i也能存储Employee
    i = Tom
    fmt.Println("This is Tom, an Employee:")
    i.SayHi()
    i.Sing("Born to be wild")
    //定义了slice Men
    fmt.Println("Let's use a slice of Men and see what happens")
    x := make{[]Men, 3}
    //T这三个都是不同类型的元素，但是他们实现了interface同一接口
    x[0], x[1], x[2] = paul, sam, mike
    for _, value := range x{
        value.SayHi()
    }
}
```

通过上面代码，可以看出interface就是一组抽象方法的集合，必须由其它非interface类型实现，而不能自我实现，Go语言通过interface实现了duck-typing，即“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。”

四、空interface

空interface即interface{}不包含任何的method，因此，所有类型都实现了空interface。空interface对于描述起不到任何作用，因为不包含任何的method，但是空interface在我们需要存储任意类型的数值时相当有用，因为它可以存储任意类型的数值，有点类似于C语言的void *类型。

```
//定义a为空接口
var a interface{}
var i int = 5
s := "hello world"
//a可以存储任意类型的数据
a = i
a = s
```

一个函数把interface{}作为参数，那么它可以接受任意类型的值作为参数，如果一个函数返回interface{}，就可以返回任意类型的值。

五、interface函数参数

interface的变量可以持有任意实现该interface类型的对象，这样我们可以通过定义interface参数让函数接受各种类型的参数。

举例说明：fmt.Println可以接受任意类型的数据，打开fmt的源码能看到定义如下：

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

也就是说，任何实现了String方法的类型都能作为参数被fmt.Println调用：

```
package main
import (
    "fmt"
    "strconv"
)
type Human struct {
    name string
    age int
    phone string
}
//通过这个方法Human实现了fmt.Stringer
func (h Human) String() string {
    return " "+h.name+" - "+strconv.Itoa(h.age)+" years - "+h.phone+" "
}
func main() {
    Bob := Human{"Bob", 39, "000-7777-xxx"}
    fmt.Println("This Human is : ", Bob)
}
```

如果需要某个类型能被fmt包以特殊的格式输出，就必须实现Stringer这个接口。如果没有实现这个接口，fmt将以默认的方式输出。

需要注意的是：实现了error接口的对象（即实现了Error() string的对象），使用fmt输出时，会调用Error()方法，因此不必再定义了String()方法了。

六、interface变量存储的类型

通过前文我们知道interface变量能存储任意类型的数值，只要该类型实现了interface。如果想反向知道该变量里实际存储的类型时什么该当如何？目前常用的有两种方法：

1、Comma-ok断言

在Go语言里，可以通过如下语法来判断interface里存储的数值类型：

```
value, ok = element.(T)
```

这里value就是变量的值，ok是一个bool类型，element是interface变量，T是断言的类型。如果element里面确实存储了T类型的值，则ok返回true，否则返回false。

比如：value, ok := element.(int)

这种方式的问题在于需要一长串的ifelse，断言的类型T越多则ifelse越多，因此，switch来了。

2、switch测试

语法同C语言中的switch，具体如下：

```
switch value := element.(type) {
    case int:
    ...
    case string:
    ...
    default:
    ...
}
```

需要注意的是：element.(type)语法不能在switch外的任何逻辑里面使用，如果要在

switch外面判断一个类型就使用comma-ok。

七、嵌入interface

Go语言里面真正吸引人的是其内置的逻辑语法，就像struct中匿名字段一样，实际上，相同的逻辑在interface中也成立。如果一个interface1作为interface2的一个嵌入字段，则interface2隐式的包含了interface1里面的method。从interface源码中可以看到如下定义：

```
type Interface interface {
    sort.Interface    //嵌入字段sort.Interface
    Push(x interface{})    //a Push method to push elements into the heap
    Pop() interface{}    //a Pop elements that pops elements from the heap
}
```

sort.Interface其实就是嵌入字段，把sort.Interface的所有method隐式包含进来了，也就是下面三个方法：

```
type Interface interface {
    //Len is the number of elements in the collection.
    Len() int
    //Less returns 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)
}
```

另一个例子就是io包下面的io.ReadWriter，它包含了io包下面的Reader和Writer两个interface。

```
//io.ReadWriter
type ReadWriter interface {
    Reader
    Writer
}
```

八、反射

Go语言实现了反射，所谓反射就是动态运行时的状态。我们一般用到的包是reflect包，官方的这篇文章详细地讲解了reflect包的实现原理——《Laws of reflection》。使用reflect一般分成三步：要去反射是一个类型的值（这些值都实现了空interface），首先需要把它转换成reflect对象（reflect.Type或者reflect.Value，根据不同情况调用不同函数）。这两种获取方式如下：

```
t := reflect.TypeOf(i)    //得到类型的元数据，通过t我们能获取类型定义里面的所有元素
v := reflect.ValueOf(i)    //得到实际的值，通过v我们获取存储在里面的值，还可以去改变值
```

转化为reflect对象之后就可以进行一些操作了，即将reflect对象转化成相应的值：

```
tag := t.Elem().Field(0).Tag        //获取定义在struct里面的标签
name := v.Elem().Field(0).String()    //获取存储在第一个字段里面的值
```

获取反射值能返回相应的类型和数值。

```
var x float64 = 3.4
v := reflect.ValueOf(x)
fmt.Println("type:", v.Type())
fmt.Println("kind is float64:", v.kind() == reflect.Float64)
fmt.Println("value:", v.Float())
```

最后，反射的字段必须是可读写的，类似于传值和传引用。按照下面的方式会产生错误：

```
var x float64 = 3.4
v := reflect.ValueOf(x)
```

```
v.SetFloat(7.1)
```

如果要修改相应的值，必须如下才行：

```
var x float64 = 3.4
p := reflect.ValueOf(&x)
v := p.Elem()
v.SetFloat(7.1)
```