方法声明

在函数名之前增加一个变量声明，该参数会将该函数附加到该类型上，即相当于为该类型定义了新的独占方法。

调用方法面向对象的理解就是向该对象发送消息，从该角度上看，对象即为接一个方法(消息)的接收器，go中习惯称之为receiver。

> 在Go语言中,我们并不会像其它语言那样用this或者self作为接收器; 我们可以任意的选择接收器的名字。由于接收器的名字经常会被使用到,所以保持其在方法间传递时的一致性和简短性是不错的主意。这里的建议是可以使用其类型的第一个字母,比如这里使用了Point的首字母p。
> 方法的调用，和方法的声明一样，接收器在前，方法名在后。
>
> p.Distance表达式叫做选择器，p.X属性获取也是选择器，比较奇怪的是声明一个p的X()方法会编译不通过。

```
package main

import (
    "math"
    "fmt"
)

type Point struct {
    X, Y float64
}

func Distance(p, q Point) float64 {
    return math.Hypot(q.X-p.X, q.Y-p.Y)
}

func (p Point) Distance(q Point) float64 {
    return math.Hypot(q.X-p.X, q.Y-p.Y)
}

func main() {
    p := Point{1, 2}
    q := Point{4, 6}
    fmt.Println(Distance(p, q))
    fmt.Println(p.Distance(q))
}
```

go可以为任意类型定义方法

Path类型是Path类型的slice
注意`if i > 0`的巧妙设计 每个Path代表一个线段的集合。

```
type Path []Path
func (p Path) Distance() float64 {
    sum := 0
    for i := range p {
        if i > 0 {
            sum += p[i-1].Distance(p[i])
        }
    }
    return sum
}
```

不知不觉中，我们已经给`[]Path`添加了Distance方法。

在go中，除了指针和interface外，我们可以很方便的给数值，字符串，slice，map等添加行为方法很方便，这也和其他语言很大的不同之处。

编译器会根据方法的名字和接收器来决定调用哪一个函数。
不同的类型可以拥有同样的方法名，但同一类型不可有方法名的冲突。

> 方法比之函数的一些好处:方法名可以简短。当我们在包外调用的时候这种好处就会被放大,因为我们可以使用这个短名字,而可以省略掉包的名字,下面是例子:

```
import "geometry"
perim := geometry.Path{{1, 1}, {5, 1}, {5, 4}, {1, 1}}
fmt.Println(geometry.PathDistance(perim)) // "12", standalone function
fmt.Println(perim.Distance()) // "12", method of geometry.Path
```

在Go中调用包外的函数需要带上包名。

基于指针对象的方法

Go中调用一个函数时，会对每一个参数值进行拷贝，**如果一个函数需要更新一个变量，或者函数的其中的一个参数实在太大我们希望能够避免这种默认的拷贝，这种情况下就需要使用到指针**

了。对应到更新接收器的对象的方法上来说，当这个接受者变量本身比较大时，我们就可以用其指针而不是对象来声明方法，如下:

```
func (p *Point) ScaleBy(factor float64) {
    //p.X = p.X * factor
    p.X *= factor
    p.Y *= factor
}

fmt.Printf("%T\n", (*Point).ScaleBy)
// func(*main.Point, float64)
```

> 在现实的程序里,一般会约定如果Point这个类有一个指针作为接收器的方法,那么所有Point的方法都必须有一个指针接收器,即使是那些并不需要这个指针接收器的函数。

只有类型Point和指向它的指针(*Point)才是可能出现在接收器声明里的两种接收器。
不过为了避免歧义，在声明方法时，如果一个类型名本身是指针的话，是不允许出现在接收器中的。

```
type P *int
func (P) f() {/**/} //compile error:invalid receiver type
```

接收器为指针的方法调用有以下几种方式:

```
go
r := &Point{1,2}
r.ScaleBy(2)
fmt.Println(*r)
```

```
p := Point{1,2}
pptr := &p
pptr.ScaleBy(2)
fmt.Println(pptr) // &{2 4}
```

```
p := Point{1, 2}
(&p).ScaleBy(2)
fmt.Println(p) // "{2, 4}"
```

> 不过后面两种方法有些笨拙。幸运的是,go语言本身在这种地方会帮到我们。如果接收器p是一个Point类型的变量,并且其方法需要一个Point指针作为接收器,我们可以用下面这种简短

> 的写法:

```
p.ScaleBy(2)
```

> 编译器会隐式地帮我们用&p去调用ScaleBy这个方法。这种简写方法只适用于“变量”,包括struct里的字段比如p.X,以及array和slice内的元素比如perim[0]。我们不能通过一个无法取到地址的接收器来调用指针方法,比如临时变量的内存地址就无法获取得到:

```
Point{1, 2}.ScaleBy(2) // compile error: can't take address of Point literal
```

> 我们可以用一个 `*Point` 这样的接收器来调用Point的方法,因为我们可以通过地址来找到这个变量,只要用解引用符号 * 来取到该变量即可。编译器在这里也会给我们隐式地插入*这个操作符,所以下面这两种写法等价的:
> `go`
> `pptr.Distance(q)`
> `(*pptr).Distance(q)`
>
> 总结:
> 1. 不管你的method的receiver是指针类型还是非指针类型,都是可以通过指针/非指针类型进行调用的,编译器会帮你做类型转换。
> 2. 在声明一个method的receiver该是指针还是非指针类型时,你需要考虑两方面的内部,第一方面是这个对象本身是不是特别大,如果声明为非指针变量时,调用会产生一次拷 贝;第二方面是如果你用指针类型作为receiver,那么你一定要注意,这种指针类型指向 的始终是一块内存地址,就算你对其进行了拷贝。熟悉C或者C++的人这里应该很快能明白。

```
package main

import (
    "fmt"
)

type Point struct {
    X, Y float64
    s    []int  //看其变化
}

func (p Point) change() {
    p.s = []int{1, 2, 3}
}

func main() {
```

```
    p := Point{1, 2, []int{}}
    p.change()
    fmt.Println("after change() ", p)
}

// after change()  {1 2 []}
// 传递指针时则  after change()  {1 2 [1,2,3]}
```

方法继承

匿名成员可以使用简短的选择器，使得我们可以解决字段类型嵌套的繁琐访问问题，go圣经中解释的很到位了，但是更深一层的理解，匿名成员是成员的继承实现，当拥有了匿名成员，就相当于拥有了该类型内部的导出成员。
而方法也是如此，匿名成员了方法，那么包含该匿名字段的类型也可以调用该方法。

```
package main

import "fmt"

type Human struct {
    name  string
    age   int
    phone string
}

type student struct {
    Human
    school string
}
type employee struct {
    Human
    company string
}

func (h *Human) sayHi(msg string) {
    fmt.Printf("hello,my name is %s, %s\n", h.name, msg)
}

func main() {
    stu := student{Human{"kobe", 12, "1232323"}, "USC"}
    emp := employee{Human{"zhansan", 12, "23232"}, "KFC"}
    stu.sayHi("welcome")
    emp.sayHi("welcome to our company")
}
// hello,my name is kobe, welcome
```

```
// hello,my name is zhansan, welcome to our company
```

方法重写

```
package main

import "fmt"

type Human struct {
    name  string
    age   int
    phone string
}

type student struct {
    Human
    school string
}
type employee struct {
    Human
    company string
}

func (h *Human) sayHi(msg string) {
    fmt.Printf("hello,my name is %s, %s\n", h.name, msg)
}

func (s *student) sayHi(msg string) {
    fmt.Printf("hello,my name is %s ,this is student impl %s\n ", s.name, ms
g)

}
func main() {
    stu := student{Human{"kobe", 12, "1232323"}, "USC"}
    emp := employee{Human{"zhansan", 12, "23232"}, "KFC"}
    stu.sayHi("welcome")
    emp.sayHi("welcome to our company")
}
```

go语言的面向对象的设计是如此精妙和简约。

接口

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。

```
type Men interface {
    sayHi(msg string)
}
```

interface可以被任意的对象实现。

注意:任意的类型都实现了空的接口`interface{}`，也就是包含0个method的interface。

interface值

interface变量的值可以存储实现该接口的任意类型对象。

go的interface实现了"鸭子类型"。

空interface对于描述起不到任何作用，但是空interface在我们需要**存储任意类型的数值**的时候相当有用，它可以存储任意类型的数值。那可以联想到空接口作为参数和返回值是多么任性。

```
a := "hello"
b := 1
var c interface{}
c = a
c = b
```

interface参数

fmt.Println函数都是默认的打印样式，但是我们通过源码看到go提供了Stringer接口只要实现了它的String方法，就将以自定义实现方式打印。

```
package main

import (
    "fmt"
    "strconv"
)

type Human struct {
    name  string
    age   int
    phone string
```

```
}


func (h Human) String() string { //interface stringer
    return "❮" + h.name + " - " + strconv.Itoa(h.age) + " years - ✆ " + h.ph
one + "❯"
}

func main() {
    bob := Human{"jordan",23,"1222222"}
    fmt.Println(bob)
}
```

fmt print.go 中定义了Stringer接口

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

注意:

builtin.go

```
go
type error interface {
Error() string
}
```

注意: 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String()方法了。

查看fmt包下的print.go源码可以清晰解释为何不需要再定义String()方法了。

```go
func (p *pp) handleMethods(verb rune) (handled bool) {
if p.erroring {
return
}
// Is it a Formatter?
if formatter, ok := p.arg.(Formatter); ok {
....
}
// If we're doing Go syntax and the argument knows how to supply it, take care of it now.
```

if p.fmt.sharpV {
...
}
} else {
// If a string is acceptable according to the format, see if
// the value satisfies one of the string-valued interfaces.
// Println etc. set verb to %v, which is "stringable".
switch verb {
case 'v', 's', 'x', 'X', 'q':
// Is it an error or Stringer?
// The duplication in the bodies is necessary:
// setting handled and deferring catchPanic
// must happen before calling the method.
switch v := p.arg.(type) {
case error:
handled = true
defer p.catchPanic(p.arg, verb)
p.fmtString(v.Error(), verb)
return

```
        case Stringer:
            handled = true
            defer p.catchPanic(p.arg, verb)
            p.fmtString(v.String(), verb)
            return
        }
    }
}
return false
```

}
```

# interface变量存储的类型

我们知道interface可以存储任意类型的变量，但是我们如何获取一个interface值的实际类型呢？
有以下几种方式
1. Comma-ok断言

```
go
```
```

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

element为interface变量，ok我bool型，T为断言类型。
如果element中确实存储了T类型的值，ok返回true，否则返回false。

实例:

```go
package main

import (
    "fmt"
    "stronv"
)

type Element interface
type List []Element
type Student struct {
    name string
    age int
}

func (s Student) String() string {
    return "(name:"+ s.name + " - age: " + strconv.Itoa(s.age)+"years)"
}

func main() {
    list := make(List,3)
    list[0] = 1
    list[1] = "hello"
    list[2] = Student{name:"kobe",23}

    for i, e := range list {
        if v, ok := e.(int);ok {
           ...
        }else if v,ok := e.(string); ok {
            ...
        }else if v, ok := e.(Student); ok {
            ...
        }else{
            ...
        }
    }
}
```

使用switch

```
go
```

1. `go for i, e := range list { switch v := e.(T); v { case int: //... case string: //... case Student: //... default: } }` 需要强调的是:element.(type)。switch外面判断一个类型就使用comma-ok语法不能在switch外的任何逻辑里面使用,如果你要在switch外面判断一个类型就使用comma-ok。

interface 嵌套

类似匿名字段的嵌套，interface也可以嵌套，那么被嵌套的interface的所有方法都可以隐式包含进来。

```
type ReadWriter interface {
    Reader
    Writer
}
```

---

layout: post
title: 006-Go基础之四 方法和接口
cateGory: Golang架构师之路
tags: Golang,method,interface

keywords: Golang,method,interface

方法和接口
学习如何为类型定义方法;如何定义接口;以及如何将所有内容贯穿组织起来。

学习了方法和接口，可以这种构造来定义对象及其行为。

方法

- **Go没有class类，可以为结构体类型定义方法**
- 一个方法是含有特殊**接收器参数**的函数
- 接收器参数在它自己的参数列表中，位于`func`关键字和函数名中间。

在下面的示例中，Abs方法有个命名为`v`的、类型为`Vertex`的接收器

```Go
package main

import (
    "fmt"
    "math"
)

type Vertex struct {
    X float64
    Y float64
}

func (v Vertex) Abs() float64 {
    return math.Sqrt(v.X*v.X + v.Y*v.Y)
}

func main() {
    v := Vertex{3, 4}
    fmt.Println(v.Abs())
}
// 5
```

方法和函数

方法只是函数多了个接收器参数。

下面的示例`Abs`写法就是个规则的函数，功能上没有任何的变化。

但是从结构上来说，方法是结构体的一部分，和之后的接口定义实现是有联系的。
```Go
package main

import (
"math"
"fmt"
)

type Vertex struct {
X float64
Y float64
}

/**
```

```
func (v Vertex) Abs() float64 {
return math.Sqrt(v.Xv.X + v.Yv.Y)
}
*/

func Abs(v Vertex) float64 {
return math.Sqrt(v.Xv.X + v.Yv.Y)
}

func main() {
v := Vertex{3, 4}
fmt.Println(Abs(v))
}

// 5
```

方法和类型

前面的案例可以为结构体类型构造方法，其实我们也可以不使用结构体类型。
下面的案例中我们可以看到带有`Abs`方法的数值类型`MyFloat`。

只能为同一包内定义的类型的接收器声明方法。不能为其他包定义的类型(包括`int`等内建类型)的接收器声明方法。

```
type myInt int//必须单独定义一下，如果不定义别名的话，编译不过去。接收器必须是同一包内的。
func (x myInt) add(a,b int)  {}
```

```
package main

import (
    "fmt"
    "math"
)

type MyFloat float64

func (f MyFloat) Abs() float64 {
    if f < 0 {
        return float64(-f)
    }
```

```
    return float64(f)
}

func main() {
    f := MyFloat(-math.Sqrt2)
    fmt.Println(f.Abs())
}

// 1.4142135623730951
```

类型别名进阶

```
package main

import fm "fmt"
func main() {
    fm.Println("hello world")
}
```

上述方式是最基本的取别名方式，简单易懂。

Go通过取别名的方式对任意类型都可以添加方法。其实本质上就是 接收器可以为任意类型构造方法。

```
package main

import fm "fmt"

type TP int

func (tpz *TP) Foo() {
    fm.Println("tpz")
}

func main() {
    var t TP = 100
    t.Foo() // method value
    (*TP).Foo(&t) // method expression
}

// tpz
```

底层类型为int，被我们绑定了Foo方法，可以看出Go的灵巧之处。 接收器可以为任意类型~~构造~~(绑定)方法。

底层为int类型的值调用自定义方法修改值。

```Go
package main

import "fmt"

type myInt int

/*func (x myInt) increase(y int) {
x+=myInt(y) //该案例没有原值改变，说明接收器的传递方式也是传值拷贝
}*/

func (x *myInt) increase(y int) {
*x += myInt(y)
}

func main() {
var a myInt = 10
a.increase(100)
fmt.Println(a)
}
```
注意：接收器是特殊的参数，但其传值方式与普通参数传值拷贝是一样的。
&取址，*取值(指针类型及所指向的空间值)

指针接收器

我们可以使用指针接收器定义方法，从面向对象的角度理解，也可以认为是**为指针接收器定义了方法**(更好理解些)。

这意味着对于某种类型 `T` ，接收器类型拥有这样的的字面量 `*T` 。(当然， `T` 不能是像 `*int` 这样的指针，是因为接收器必须定义在同一包下，而我们开发者自定义的肯定无法修改int作为接收器，自然不能是*int)。

例如，为 `*Vertex` 接收器定义 `Scale` 方法。

指针接收器的方法可以修改接收器指向的值(就如 `Scale` 方法所做)。**由于方法经常需要修改他们**

的接收器，所以指针接收器比值接收器更加通用。

试试从Scale方法移除`*`，并管着程序运行结果是如何变化的。

使用值接收器的话，`Scale`方法将会操作原始`Vertex`的值的一份副本(对于其他函数的参数是一样的，**本质上也是一种针对副本的只读行为**)。Scale方法必须用指针接收器来更改main函数中声明的Vertex的值。

```
package main

import (
    "math"
    "fmt"
)

type Vertex struct {
    X, Y float64
}

func (v Vertex) Abs() float64 {
    return math.Sqrt(v.X*v.X + v.Y*v.Y)
}

func (v *Vertex) Scale(f float64) {
    v.X = v.X * f
    v.Y = v.Y * f
}

func main() {
    v := Vertex{5,12}
    v.Scale(100)
    fmt.Println(v.Abs())
}

// 1300
```

指针和函数

上一节中的案例我们使用函数的方式重写以下。

```
package main

import (
```

```
    "fmt"
    "math"
)

type Vertex struct {
    X, Y float64
}

func Abs(v *Vertex) float64{
    return math.Sqrt(v.X*v.X+v.Y*v.Y)
}

func Scale(v Vertex, f float64) {
    v.X = v.X * f
    v.Y = v.Y * f
}

func main() {
    v := Vertex{3,4}
    Scale(&v,10)//函数的方式就将结构体指针作为参数传入
    fmt.Println(Abs(v))
}

// 50
```

同样，我们再试试删除`Scale`方法的`*`。你知道为什么结果方式了变化吗?还需要哪些修改使该程序编译通过?

```
package main

import (
    "fmt"
    "math"
)

type Vertex struct {
    X, Y float64
}

func Abs(v Vertex) float64 {
    return math.Sqrt(v.X*v.X + v.Y*v.Y)
}

func Scale(v Vertex, f float64) Vertex{
    v.X = v.X * f
    v.Y = v.Y * f
    return v
```

```
}

func main() {
    v := Vertex{3, 4}
    p := Scale(v, 10)
    fmt.Println(Abs(p))
}
```

方法和指针重定向(Methods and pointer indirection)

对比之前两小节的程序，你也会注意到了`带指针参数`的**函数**`必须接受一个指针`：

```
var v Vertex
ScaleFunc(v) // 编译错误
ScaleFunc(&v) // ok
```

但带指针的方法不论是值接收器还是指针接收器，当它们被调用时:

```
var v Vertex
v.Scale(5) // ok
p := &v
p.Scale(10) // ok
```

对于语句`v.Scale(5)`，即使v是一个值并不是一个指针，指针接收器的方法也被自动地调用。也就是说，由于`Scale`方法有个指针接收器，作为便利，Go解释器将语句`v.Scale(5)`解析为`(&v).Scale(5)`

在相反的情况下是相同的。`带值参数`的**函数**`必须接收一个特定类型的值`：

```
var v Vertex
fmt.Println(AbsFunc(v)) // OK
fmt.Println(AbsFunc(&v)) // Compile error!
```

但以值为接收器的方法当它们被调用时，接受者既可以为值也可以为指针:

```
var v Vertex
fmt.Println(v.Abs()) // OK
p := &v
fmt.Println(p.Abs()) // OK
```

在这个案例中，方法调用`p.Abs()`被解释为`(*p).Abs()`
`*`操作符表示指针指向的底层值。

```
package main

import (
    "fmt"
    "math"
)

type Vertex struct {
    X, Y float64
}

func (v Vertex) Abs() float64 {
    return math.Sqrt(v.X*v.X + v.Y*v.Y)
}

func AbsFunc(v Vertex) float64 {
    return math.Sqrt(v.X*v.X + v.Y*v.Y)
}

func main() {
    v := Vertex{3, 4}
    fmt.Println(v.Abs())
    fmt.Println(AbsFunc(v))

    p := &Vertex{5, 12}
    fmt.Println(p.Abs())
    fmt.Println(AbsFunc(*p)) //*p 指针p底层的值
}

// 5
// 5
// 13
// 13
```

值或指针接收器的选择△

有两个原因选择指针接收器。

- 第一:方法可以修改指针所指向的值
- 第二:避免每次方法调用都进行值拷贝。如果是大结构体类型的接收器这将会很高效。

这次的示例中，`Scale`和`Abs`都使用`*Vertex`类型的接收器，即使`Abs`方法不需要修改它的

接收器。

通常情况下，给定类型的方法都应该有值或指针接收器，但不应是两者的混合。

```
package main

import (
    "fmt"
    "math"
)

type Vertex struct {
    X, Y float64
}

func (v *Vertex) Scale(f float64) {
    v.X = v.X * f
    v.Y = v.Y * f
}

func (v *Vertex) Abs() float64 {
    return math.Sqrt(v.X*v.X + v.Y*v.Y)
}

func main() {
    //v := &Vertex{3, 4}
    v := Vertex{3, 4}
    fmt.Printf("Before scaling: %+v, Abs: %v\n", v, v.Abs())
    v.Scale(10)
    fmt.Printf("After scaling: %+v, Abs: %v\n", v, v.Abs())

}

// Before scaling: {X:3 Y:4}, Abs: 5
// After scaling: {X:30 Y:40}, Abs: 50
```

变参的值拷贝

```
package main

import (
    "fmt"
)
```

```go
func main() {
    a, b := 1,2
    fmt.Println("改变前:-->", a,b)
    Change(a,b)
    fmt.Println("改变后:-->", a,b)
}
//func Change(x ...int) {
func Change(x ...int) {
    x[0] = 3
    x[1] = 4
    fmt.Println("改变中:-->", x)
}
// 改变前:--> 1 2
// 改变中:--> [3 4]
// 改变后:--> 1 2
```

```go
package main

import (
    "fmt"
)

func main() {
    a, b := 1, 2
    fmt.Println("改变前:-->", a, b)
    Change(&a, &b)
    fmt.Println("改变后:-->", a, b)
}

//func Change(x ...int) {
func Change(x ...*int) {
    *x[0] = 3
    *x[1] = 4
    fmt.Println("改变中:-->", x)
}
// 改变前:--> 1 2
// 改变中:--> [0xc42000e238 0xc42000e260]
// 改变后:--> 3 4
```

- 引用类型传参时也是值拷贝，注意拷贝的值是指针地址。并且需要注意的是，并非拷贝了指针。

```go
package main
```

```
import (
    "fmt"
)

func main() {
    s := []int{1, 2, 3}
    fmt.Println("改变前:-->", s)
    Change(s)
    fmt.Println("改变后:-->", s)
}
//func Change(x ...int) { //编译错误
func Change(x []int) {
    x[0] = 4
    x[1] = 5
    x[2] = 6
    fmt.Println("改变中:-->", x)
}

// [1 2 3]
// [4 5 6]
// [4 5 6]
```

上述案例表明，变参是传入前就决定了的形式，param(x ...int) 参数x需要是散值，但x却组成了数组形式。
但是当参数实际传入的是数组时，param(x []int)就必须与之对应，否则是无法编译通过的。

Go中一切皆值，一切皆类型

函数也是值，类型则是函数类型。简而言之，函数类型也是值

`a := 3 // 3是int类型的值`

`f := FuncName // f是func类型的值`

```
package main

import "fmt"

func main() {
    a := A
    a()
}

func A() {
```

```
    fmt.Println("A()")
}
// A()
```

函数也是类型，在todd的面向对象讲解中，讲到golang的面向对象和其他语言的区别，明确地指出类型的重要。
最棒的go面向对象的讲解
go面向对象不创建class，只需创建type
不需要实例化，只需要给类型赋值即可。

匿名函数

```
package main

import "fmt"

func main() {
    a := func() {
        fmt.Println("A()")
    }
    a()
}
// A()
```

闭包

```
package main

import "fmt"

func main() {
    f := closure(10)
    fmt.Println(f(1))
    fmt.Println(f(2))
}

func closure(x int) func(y int) int {
    fmt.Printf("%p %v \n", &x, x)
    return func(y int) int {
        fmt.Printf("%p %v \n", &x, x)
        return x + y
    }
}
```

```
// 0xc420082050 10
// 0xc420082050 10
// 11
// 0xc420082050 10
// 12
```

三次x的地址指向都是同一个x

> （1）闭包是一种设计原则，它通过分析上下文，来简化用户的调用，让用户在不知晓的情况下，达到他的目的；
> （2）网上主流的对闭包剖析的文章实际上是和闭包原则反向而驰的，如果需要知道闭包细节才能用好的话，这个闭包是设计失败的；
> （3）尽量少学习。
> 闭包，懂不懂由你，反正我是懂了

接口

接口被定义为一组方法签名的集合。

接口类型的值可以保存**实现其方法的**任意值。

差一个接口方法实现都不叫接口实现

```go
type Human interface {
    SayHi(name string)
    Run()
}

type Man struct {
    Name string
}

func (r *Man) SayHi(name string) {
    fmt.Println(r.Name)
}

func (r *Man) Run() {
    fmt.Println( a: "man is running....")
}

func SayHi(name string) {
    fmt.Println("sayHi:" + name)
}

func Run() {
    fmt.Println( a: "running....")
}
```

```go
package main

import "fmt"

type Printer interface {
    Name() string
    Print()
    Devicer
}

type Devicer interface {
    Start()
}

type HP_Printer struct {
    name string
}

type Sony_Printer struct {
    name string
}

func (this *Sony_Printer) Name() string {
    return this.name
}

//func (hp *HP_Printer) Start() {
func (hp HP_Printer) Start() {
    fmt.Println("hp start()")
}

// func (hp *HP_Printer) Name() string {
func (hp HP_Printer) Name() string {
    return hp.name
}

// func (hp *HP_Printer) Print() {
func (hp HP_Printer) Print() {
    fmt.Println("hp print():", hp.name)
}

func main() {
    // var hp Printer
    // hp = HP_Printer{"hp printer"}
    //hp := &HP_Printer{"hp printer"}
    hp := HP_Printer{"hp printer"}
    fmt.Println(hp.Name())
    hp.Print()
```

```
    stop(hp)

    //sony_printer := Sony_Printer{name: "sony printer"}
    //stop(sony_printer) //需要实现所有的方法，才是Printer实现
    //sony_printer.stop()
}

//func (printer Printer)stop() {
func stop(printer Printer) {
    fmt.Printf("stop %s\n", printer.Name())
}

// hp printer
// hp print(): hp printer
// stop print
```

接口嵌套

```
package main

import "fmt"

type Printer interface {
    Name() string
    Print()
    Devicer
}

type Devicer interface {
    Start()
}

type HP_Printer struct {
    name string
}

type Sony_Printer struct {
    name string
}

//func (hp *HP_Printer) Start() {
func (hp HP_Printer) Start() {
    fmt.Println("hp start()")
}
```

```go
// func (hp *HP_Printer) Name() string {
func (hp HP_Printer) Name() string {
    return hp.name
}

// func (hp *HP_Printer) Print() {
func (hp HP_Printer) Print() {
    fmt.Println("hp print():", hp.name)
}

func main() {
    // var hp Printer
    // hp = HP_Printer{"hp printer"}
    //hp := &HP_Printer{"hp printer"}
    hp := HP_Printer{"hp printer"}
    fmt.Println(hp.Name())
    hp.Print()

    stop(hp)
}

func stop(printer Printer) {
    fmt.Println("stop print")
}
```

接口值

在实现内部，**接口值**可以被看作包含**值和具体类型的元组**`(value, type)`
接口值保存了一个底层的具体类型的具体值。

接口值调用方法时会执行其底层类型的同名方法。

```go
package main

import (
    "fmt"
    "math"
)

type I interface {
    M()
}
```

```
type T struct {
    S string
}

type MyFloat float64

func (t *T) M() {
    fmt.Println(t.S)
}

func (f MyFloat) M() {
    fmt.Println(f)
}

func main() {
    var i I

    i = &T{"hello"}
    describe(i)
    i.M()

    i = MyFloat(math.Pi)
    describe(i)
    i.M()
}

func describe(i I) {
    fmt.Printf("(%v,%T)\n", i, i)
}

// (&{hello},*main.T)
// hello
// (3.141592653589793,main.MyFloat)
// 3.141592653589793
```

底层值为nil的接口值

如果接口中的实际具体值为`nil`，该方法仍将被一个`nil`类型的接收器调用。

在其他一些语言中，这将导致空指针异常，但是在`Go`中通常使用`nil`接收器写个方法来有优雅地处理它。(例如下面的示例中`M`方法)

保存了`nil`具体值的接口其本身并不是`nil`。

```
package main
```

```go
import (
    "fmt"
)

type I interface {
    M()
}

type T struct {
    S string
}

func (t *T) M() {
    if t == nil {
        fmt.Println("<nil>")
        return
    }
    fmt.Println(t.S)
}

func main() {
    var i I
    var t *T
    i = t
    describe(i)
    i.M() //即使为nil，也是很优雅的，未出现空指针异常的，和其他语言不一样

    i = &T{"hello"}
    describe(i)
    i.M()
}

func describe(i I) {
    fmt.Printf("(%v, %T)\n", i, i)
}

// (<nil>, *main.T)
// <nil>
// (&{hello}, *main.T)
// hello
```

nil接口值

nil接口值既不保存值也不保存具体的类型。

nil接口调用方法是运行时错误，因为**接口的元组内并没有包含**能够指明调用哪个**具体**方法的**类型**

```
package main

import (
    "fmt"
)

type T interface {
    M()
}

func main() {
    var t T
    describe(t)
    t.M() //这里直接使用接口调用，接口值的元组连nil都没有，肯定会运行时异常。老生常谈面向对象，最起码要有个实例对象嘛..
}

func describe(t T) {
    fmt.Printf("(%v, %T)\n",t,t)
}

/**(<nil>, <nil>)
panic: runtime error: invalid memory address or nil pointer dereference
[signal SIGSEGV: segmentation violation code=0x1 addr=0x20 pc=0x1088b8a]

goroutine 1 [running]:
main.main()
    /Users/fqc/work/src/myecho/main.go:14 +0x3a
exit status 2*/
```

类型断言

判断是否是某一类型，`if x_obj, ok := Obj.(T); ok {...}`
比如判断Sony_Printer类型是否为Printer接口类型，可以`printer.(Sony_Printer)`
`if hp, ok := printer.(HP_Printer); ok {...}`

```
package main

import (
    "fmt"
)

type Printer interface {
```

```
    Name() string
    Print()
    Devicer
}

type Devicer interface {
    Start()
}

type HP_Printer struct {
    name string
}

type Sony_Printer struct {
    name string
}

func (sy Sony_Printer) Name() string {
    return sy.name
}

func (sy Sony_Printer) Print() {
    fmt.Println(sy.name, "printing...")
}

func (sy Sony_Printer) Start() {
    fmt.Println(sy.name, "start()")
}

func (hp HP_Printer) Start() {
    fmt.Println(hp.name, "start()")
}

func (hp HP_Printer) Name() string {
    return hp.name
}

func (hp HP_Printer) Print() {
    fmt.Println(hp.name, "printing...")
}

func main() {
    hp := HP_Printer{"hp printer"}
    hp.Start()
    hp.Print()
    stop(hp)

    sony := Sony_Printer{"sony printer"}
```

```
    sony.Start()
    sony.Print()
    stop(sony)

}

func stop(printer Printer) {
    if hp, ok := printer.(HP_Printer); ok { //类型断言
        fmt.Println(hp.name, "stop()")
        return
    }
    fmt.Print("Unknown printer.")
}

// hp printer start()
// hp printer printing...
// hp printer stop()
// sony printer start()
// sony printer printing...
// Unknown printer.
```

类型断言提供了底层具体值的访问。

```
t := i.(T)
```

上述语句断言接口值`i`保存了具体类型`T`并将T的底层具体值赋予了变量`t`。

如果`i`没有持有`T`,上述语句将异常(`trigger a panic 出发恐慌..异常`)

为了测试接口值是否持有特定的类型，类型断言可以返回两个值:

1. 底层值
2. 报告断言是否成功的布尔值

```
t, ok := i.(T)
```

如果`i`持有`T`，则`t`将被赋予`i`的底层值，`ok`将为`true`。相反，`ok`则为false，`t`将为`T`类型的默认零值，但没有异常发生。

注意此语法和读取map类似。

```
package main
```

```
import (
    "fmt"
)

func main() {
    var i interface{} = "hello"
    s := i.(string)
    fmt.Println(s)

    s, ok := i.(string)
    fmt.Printf("%v, %v\n", s, ok)

    f, ok := i.(float64)
    fmt.Printf("%v, %v\n", f, ok)

    //f = i.(float64) // panic
    //fmt.Println(f)

    /**
    panic: interface conversion: interface {} is string, not float64

Goroutine 1 [running]:
main.main()
    /Users/fqc/github/Golang_sidepro/src/github.com/fqc/tour/Go_tour.Go:18 +
0x3a7
exit status 2
    */
}

// hello
// hello, true
// 0, false
```

```
package main

import "fmt"

func main() {
    rem := 7.24
    fmt.Printf("%T\n", rem)
    fmt.Printf("%T\n", int(rem))

    var val interface{} = 7
    fmt.Printf("%T\n", val)
    fmt.Printf("%T\n", int(val))//cannot convert val (type interface {}) to
type int: need type assertion
        //fmt.Printf("%T\n", val.(int))
```

```
}
```

空接口

没有一个方法的接口类型被认作为空接口。

```
interface{} //  a special type
```

空接口可以保存任意类型的值。(每种类型都实现了空接口，因为不用实现任何方法)

空接口通常用来处理未知类型的值。例如，`fmt.Print`接收任意数量的`interface{}`空接口类型。

```
package main

import (
    "fmt"
)

func main() {
    var i interface{}
    describe(i)

    i = 43
    describe(i)

    i = "hello"
    describe(i)
}

func describe(i interface{}) {
    fmt.Printf("(%v, %T)\n", i, i)
}

// (<nil>, <nil>)
// (43, int)
// (hello, string)
```

根据Go语言接口的实现原理，我们可以得出结论:**Go的所有类型的方法都实现了空接口**。

```
type empty interface {}
```

前面的一个示例中的stop方法我们进行改造

```
func stop(printer empty) {
    if hp, ok := printer.(HP_Printer); ok {
        fmt.Println(hp.name, "stop()")
        return
    }
    fmt.Print("Unknown printer.")
}
```

接口类型转换

遵循其他语言**子集向上转换**

```
hp := HP_Printer{"hp printer"}
fmt.Println(hp.name)
p := Devcicer(hp)
p.Start()
```

```
package main

import (
    "fmt"
    "math"
)

type Abser interface {
    Abs() float64
}

func main() {
    var a Abser
    f := MyFloat(-math.Sqrt2)
    v := Vertex{3, 4}

    a = f
    fmt.Println(a.Abs())
    a = v // cannot use v(type Vertex) as type Abser, Vertex dose not implem
ent Abser(Abs method has pointer receiver)
    fmt.Println(a.Abs())
}

type MyFloat float64
```

```
func (f MyFloat) Abs() float64 {
    if f < 0 {
        return float64(-f)
    }
    return float64(f)
}

type Vertex struct {
    X, Y float64
}

func (v *Vertex) Abs() float64 {
    return math.Sqrt(v.X*v.X + v.Y*v.Y)
}
```

示例代码将编译出错，`Vertex`(值类型)没有实现`Abser`，因为`Abs`方法仅仅定义了`*Vertex`(指针类型)。接收器的值类型和指针类型是需要区分对待的。但如果是指针类型的时候，调用的时候可以简便使用值的方式调用，Go编译器提供了便利。

上述代码编译错误的解决方案是:

`a = &v`或者 `v := &Vertex{3,4}`

接口与隐式实现

一个类型实现接口：`只是通过实现接口的方法，不需要任何关键字声明`。这里没有明确的声明意图，没有"implements"关键字。

隐式接口**从实现上解耦了接口的定义**，也就是说**实现可以出现在任意包中，无需提前准备**。 实现上解耦接口的定义这句话可以对比其他语言，比如JGoava的接口与实现，都是通过明确的关键字进行声明，而Go语言无需在每一个实现上增加新的接口名称，有利于鼓励明确的接口定义。

```
package main

import (
    "fmt"
)

type I interface {
    M()
}

type T struct {
    S string
```

```
}

func (t T) M() {
    fmt.Println(t.S)
}

func main() {
    i := T{"hello"}
    // var i I = T{"hello"}
    i.M()
}
```

interace机制

go语言的interface机制

附录

https://Godoc.org/Golang.org/x/tools/cmd/guru
https://docs.Google.com/document/d/1_Y9xCEMj5S-7rv2ooHpZNH15JgRT5iM742gJkw5LtmQ/edit#

```
Setting up

Download and build the guru tool and install it somewhere on your $PATH.

$ Go get   Golang.org/x/tools/cmd/guru
$ Go build Golang.org/x/tools/cmd/guru
$ mv guru $(Go env GoROOT)/bin                    (for example)
$ guru -help
Go source code guru.
Usage: guru [flags] <mode> <position>
...
```

describe describe selected syntax: definition, methods, etc

Tips

```
Go tool tour // 英文Go tour
```

```
$GoPATH/bin/Gotour // 中文Go tour  ```



指针or结构


```