接口

接口interface，和Java类似，是一组**行为规范**的集合，就是定义一组未实现的函数声明。谁使用接口就是参照接口的方法定义实现它们。

```
type 接口名 interface {
    方法1（参数列表1）返回值列表1
    方法2（参数列表2）返回值列表2
    ...
}
```

- 接口命名习惯在接口名后面加上er后缀
- 参数列表、返回值列表参数名可以不写
- 如果要在包外使用接口，接口名应该首字母大写，方法要在包外使用，方法名首字母也要大写
- 接口中的方法应该设计合理，不要太多

Go语言中，使用组合实现对象特性的描述。对象内部使用结构体内嵌组合对象应该具有的特性，对外通过接口暴露能使用的特性。

Go语言的接口设计是非侵入式的，接口编写者无需知道接口被哪些类型实现。而接口实现者只需知道实现的是什么样子的接口，但无需指明实现哪一个接口。编译器知道最终编译时使用哪个类型实现哪个接口，或者接口应该由谁来实现。

接口是约束谁应该具有什么功能，实现某接口的方法，就具有该接口的功能，简而言之，缺什么补什么。

接口实现

如果一个结构体实现了一个接口声明的**所有方法**，就说结构体实现了该接口。

一个结构体可以实现多个接口。

```
package main

import "fmt"

type Person struct {
    name string
    age  int
}

type Sport interface {
    run()
    jump()
}

func (*Person) run() {
    fmt.Println("Run~~~")
}

func (*Person) jump() {
    fmt.Println("Jump~~~")
```

```
}

func main() {
    p := new(Person)
    p.run()
    p.jump()
}
```

在实现一个方法swim。

```
func (*Person) swim() {
    fmt.Println("Swim~~~")
}

func main() {
    p := new(Person)
    p.run()
    p.jump()
    p.swim()

    var s Sport = p // 不报错，Person实现了Sport接口
    s.run()
    s.jump()
    s.swim() // 报错，接口没有该方法
}
```

接口嵌入

除了结构体可以嵌套，接口也可以。接口嵌套组合成了新接口。

```
type Reader interface {
    Read(p []byte) (n int, err error)
}

type Closer interface {
    Close() error
}

type ReadCloser interface {
    Reader
    Closer
}
```

ReadCloser接口是Reader、Closer接口组合而成，也就是说它拥有Read、Close方法声明。

空接口

空接口，实际上是空接口类型，写作`interface {}`。为了方便使用，Go语言为它定义一个别名any类型，即`type any = interface{}`。

空接口，没有任何方法声明，因此，任何类型都无需显式实现空接口的方法，因为任何类型都满足这个空接口的要求。那么，任何类型的值都可以看做是空接口类型。

```go
var a = 500
var b interface{} // 空接口类型可以适合接收任意类型的值
b = a
fmt.Printf("%v, %[1]T; %v, %[2]T\n", a, b)
var c = "abcd"
b = c // 可以接收任意类型
fmt.Printf("%v, %[1]T; %v, %[2]T\n", c, b)

b = []interface{}{100, "xyz", [3]int{1, 2, 3}} // interface{}看做一个整体。切片元素类型任意
fmt.Printf("%v, %[1]T\n", b)
```

接口类型断言

接口类型断言（Type Assertions）可以将接口转换成另外一种接口，也可以将接口转换成另外的类型。

接口类型断言格式 `t := i.(T)`

- i代表接口变量
- T表示转换目标类型
- t代表转换后的变量
- 断言失败，也就是说 i 没有实现T接口的方法则panic
- `t, ok := i.(T)`，则断言失败不panic，通过ok是true或false判断i是否是T类型接口

```go
var b interface{} = 500
if s, ok := b.(string); ok {
    fmt.Println("转换成功，值是", s)
} else {
    fmt.Println("转换失败")
}
```

type-switch

可以使用特殊格式来对接口做多种类型的断言。

```go
var i interface{} = 500
switch i.(type) {
case nil:
    fmt.Println("nil")
case string:
    fmt.Println("字符串")
case int:
    fmt.Println("整型")
default:
    fmt.Println("其他类型")
}
```

`i.(type)`只能用在switch中。

如果想在switch中使用转换的结果，可以用下面的方式

```go
var i interface{} = 500
switch v := i.(type) {
case nil:
    fmt.Println("nil")
case string:
    fmt.Println("字符串", v)
case int:
    fmt.Println("整型", v)
default:
    fmt.Println("其他类型")
}
```

输出格式接口

我们使用fmt.Print等函数时，对任意一个值都有一个缺省打印格式。本质上就是实现打印相关的接口。

```go
// 普通的Print
type Stringer interface {
    String() string
}

// %#v format
type GoStringer interface {
    GoString() string
}
```

通过实现上面的接口，就可以控制值的打印输出格式。

```go
package main

import "fmt"

type Person struct {
    name string
    age  int
}

// fmt.Stringer
func (*Person) String() string {
    return "abc"
}

// fmt.GoStringer
func (*Person) GoString() string {
    return "xyz"
}

func main() {
    p := &Person{"Tom", 20}
    fmt.Println(p)
    fmt.Printf("%+v\n", p)
    fmt.Printf("%#v\n", p)
```

```
25 }
```

	%v	%+v	%#v
默认	&{tom 20}	&{name:tom age:20}	&main.Person{name:"tom", age:20}
实现Stringer接口	abc	abc	
实现GoStringer接口			xyz

接口定义实例 https://gitee.com/go-course/go9/blob/master/projects/vblog/api/apps/blog/interface.go#L14-37