接口 interface

C/C++的接口？提供一个可用的"插头"，别管我怎么供电的，你插进来就可以充电了

Go的接口？实现 多态 的基础

```
class Animal {
    ...
    virtual string Speak() = 0;
};

class Dog : public Animal {
    ...
    string Speak() override {
        return "Haji wang!";
    }
}

class Cat : public Animal {
    ...
    string Speak() override {
        return "Haji mi!";
    }
}

Cat cat; cat.Speak();
Dog dog; dog.Speak();
```

while in Go

```go
// 接口，有点像"父类"了
type Speaker interface {
    Speak() string // 方法method Speak
}

type Dog struct{}

func (d Dog) Speak() string {
    return "Haji wang!"
}

type Cat struct{}

func (c Cat) Speak() string {
    return "Haji mi!"
}

func main() {
    animals := []Speaker{Dog{}, Cat{}}
    for _, animal := range animals {
        fmt.Println(animal.Speak())
    }
}
```

接口定义

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

一种特殊/抽象类型。结构？值？只表现出一堆**方法**，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
果然很抽象😋 👍

接口类型

接口类型 表述了 一系列方法的集合

实现了这些方法的具体类型——这个接口类型的实例
e.g. package io

```
package io
// 接口Reader，包含一个方法Read
type Reader interface {
    Read(p []byte) (n int, err error)
}

// 接口Closer，包含一个方法Close
type Closer interface {
    Close() error
}

// 任何可读/可关闭的类型都可以尝试实现Reader/Closer接口
func (r, io.Reader) Read(p []byte) (int, error) { // io.Reader类型来实例化Read()
    ...
}
```

接口组合

已有接口组成新接口

```
package io
type Reader interface {
    Read(p []byte) (n int, err error)
}
type Closer interface {
    Close() error
}
type Writer interface {
    Write(p []byte) (n int, err error)
}

// 组合
type ReadWriter interface {
    Reader
    Writer
}
type ReadWriteCloser interface {
    Reader
    Writer
    Closer
}
```

也可以新旧混用

```
// 接口ReadWriter，包含两个热乎的新的方法Read & Write
type ReadWriter interface {
    Read(p []byte) (n int, err error)
    Write(p []byte) (n int, err error)
}

// 一个新的Read和一个已有的Writer
type ReadWriter interface {
    Read(p []byte) (n int, err error)
    Writer
}
// 都是一样的效果
```

实现接口的条件

所有方法实现 = 接口实现

一个类型如果拥有一个接口需要的所有方法，那么这个类型就实现了这个接口
如果 `type RW struct{...}` 既实现了 `Read` 也实现了 `Write` ，那么 `RW` 类型就实现了 `ReaderWriter` 这个接口

接口->类型实例？方法全实现了就好办~

接口指定的规则非常简单：表达一个类型属于某个接口只要这个类型实现这个接口。

```
var w io.Writer
w = os.Stdout // √ os.Stdout所属类 *os.File也实现了Write方法
// w = time.Second × // time.Duration does not implement io.Writer (missing method Write)
w.Write([]byte("hello"))
```

```
var w W // W类实现了Writer接口（Write方法）
var rw RW // RW类实现了ReaderWriter接口（Read & Write方法）
w = rw // √ 因为rw也实现了Write方法
rw = w // × 因为w没实现Read方法
```

方法多了可以，我不干就是咯

```
os.Stdout.Write([]byte("hello")) // √ 人家实现了Write方法
os.Stdout.Close() // √ 人家实现了Close方法

var w io.Writer
w = os.Stdout // 由前文可得
w.Write([]byte("hello")) // √
// w.Close() // × 哎，我原来还是一个io.Writer的时候可不会Close，怎么能指望我转行当了os.Stdout就会了
// 哈基io.Writer你这家伙，居然还是没学会吗😊
```

指针？指针？？

指针实现的接口，和我本身有什么关系？😋

```
type Speaker interface {
    Speak() string
}

// Dog类实现了Speak()方法
// 值接收者实现
type Dog struct{}
func (d Dog) Speak() string { return "Woof!" }

// *Cat类实现了Speak()方法
// 指针接收者实现
type Cat struct{}
func (c *Cat) Speak() string { return "Meow!" }

func main() {
    var s Speaker

    s = Dog{}       // 值类型OK
    s = &Dog{}      // 指针类型也OK
                    // 哎？自动解引用发现实现了Speak()方法吗？有点意思😍

    s = Cat{}       // 错误：值类型未实现Speaker
                    // md，不会自动寻址吗www，帮人不到底？😭
    s = &Cat{}      // 正确：指针类型实现了Speaker
}
```

空接口

空——没有

```
var any interface{}
```

没有任何方法——空接口
有什么用？🤔没用😋

咳咳，因为空接口类型对实现它的类型没有要求，所以我们可以将任意一个值赋给空接口类型万能类！😘🥰🥳

```
var any interface{}
any = true // bool
any = 12.34 // float64
any = "hello" // string
any = map[string]int{"one": 1} // map
any = new(bytes.Buffer) // pointer
```

但是 interface{} 没有任何方法，相当于 any 不能进行任何操作，寄😋
类型断言——获取interface{}中值的方法
怎么可能让你万能类就这么润了😘

那啥是类型断言？且听下回分解🥳

flag.Value接口

接口定义

自定义类型**命令行参数**

e.g.

```
ls -l # -l参数表示 以长格式显示（权限、大小等）
```

```
rm -rf filepath(filename)

-r 向下递归，不管有多少级目录，一并删除
-f 直接强行删除，没有任何提示
# filepath(filename) 向-r传递命令行参数
```

flag 包中的 Value 接口

```
type Value interface {
    String() string // 返回该值的字符串表示
    Set(string) error // 解析字符串参数并设置值，返回可能的错误
}
```

flag.Var 函数：
将自定义类型与命令行参数绑定的关键函数

```
func Var(value Value, name string, usage string)
```

value：必须是一个实现了 flag.Value 接口的类型实例
name：命令行参数的名称
usage：命令行参数的帮助信息

完整流程示例

温度展示 & 转换

```go
package main

import (
    "flag"
    "fmt"
    "strings"
)

type Temperature float64

// 为Temperature类型实现两个方法String() & Set()
// 方便后面作为参数传入flag.Var()函数
func (t *Temperature) String() string {
    return fmt.Sprintf("%.2f°C", *t)
}

func (t *Temperature) Set(s string) error {
    var value float64
    var unit string
    // Sscanf()，结构化解析输入，"%f"一个浮点数，"%s"一个字符串
    _, err := fmt.Sscanf(s, "%f%s", &value, &unit)
    if err != nil {
        return err
    }
    switch strings.ToUpper(unit) {
    case "C":
        *t = Temperature(value)
    case "F":
        *t = Temperature((value - 32) * 5 / 9)
    default:
        return fmt.Errorf("invalid unit %q", unit)
    }
    return nil
}

func main() {
    var temp Temperature
    // 为flag.Var()传参~~
    // 1.value 2.名称temp 3.帮助信息(你想说的tips)
    flag.Var(&temp, "temp", "Temperature in Celsius (e.g., 20C) or Fahrenheit (e.g., 68F)")

    // 调用 flag.Parse() 时，flag 包会自动调用你实现的 Set 方法处理用户输入
    flag.Parse()
```

```
    // 输出是默认调用String()方法
    // "%v"的默认行为，如果值的类型实现了 String()，则调用该方法。
    // 如果未实现，则回退到默认的格式化逻辑（如结构体的字段展开）
    fmt.Printf("Current temperature: %v\n", temp)
}
```

运行

命令行中

```
go run interface.go -temp 12C
# Current temperature: 12

go run interface.go -temp 12F
# Current temperature: -11.11111111111111
```