概述

对于面向对象编程的支持Go 语言设计得非常简洁而优雅。因为， Go语言并没有沿袭传统面向对象编程中的诸多概念，比如继承(不支持继承，尽管匿名字段的内存布局和行为类似继承，但它并不是继承)、虚函数、构造函数和析构函数、隐藏的this指针等。

尽管Go语言中没有封装、继承、多态这些概念，但同样通过别的方式实现这些特性：

- 封装：通过方法实现
- 继承：通过匿名字段实现
- 多态：通过接口实现

匿名组合

匿名字段

一般情况下，定义结构体的时候是字段名与其类型一一对应，实际上Go支持只提供类型，而不写字段名的方式，也就是匿名字段，也称为嵌入字段。

当匿名字段也是一个结构体的时候，那么这个结构体所拥有的全部字段都被隐式地引入了当前定义的这个结构体。

```
//人
type Person struct {
    name string
    sex  byte
```

```
    age  int
}
//学生
type Student struct {
    Person // 匿名字段，那么默认Student就包含了Person
的所有字段
    id     int
    addr   string
}
```

初始化

```
//人
type Person struct {
    name string
    sex  byte
    age  int
}

//学生
type Student struct {
    Person // 匿名字段，那么默认Student就包含了Person
的所有字段
    id     int
    addr   string
}
```

```
func main() {
    //顺序初始化
    s1 := Student{Person{"mike", 'm', 18}, 1,
"sz"}
    //s1 = {Person:{name:mike sex:109 age:18} id:1
addr:sz}
    fmt.Printf("s1 = %+v\n", s1)

    //s2 := Student{"mike", 'm', 18, 1, "sz"}
//err

    //部分成员初始化1
    s3 := Student{Person: Person{"lily", 'f', 19},
id: 2}
    //s3 = {Person:{name:lily sex:102 age:19} id:2
addr:}
    fmt.Printf("s3 = %+v\n", s3)

    //部分成员初始化2
    s4 := Student{Person: Person{name: "tom"}, id:
3}
    //s4 = {Person:{name:tom sex:0 age:0} id:3
addr:}
    fmt.Printf("s4 = %+v\n", s4)
}
```

成员的操作

```
var s1 Student //变量声明
    //给成员赋值
    s1.name = "mike" //等价于 s1.Person.name =
"mike"
    s1.sex = 'm'
    s1.age = 18
    s1.id = 1
    s1.addr = "sz"
    fmt.Println(s1) //{{mike 109 18} 1 sz}

    var s2 Student //变量声明
    s2.Person = Person{"lily", 'f', 19}
    s2.id = 2
    s2.addr = "bj"
    fmt.Println(s2) //{{lily 102 19} 2 bj}
```

同名字段

```
//人
type Person struct {
    name string
    sex  byte
    age  int
}

//学生
type Student struct {
```

```
    Person // 匿名字段，那么默认Student就包含了Person的所有字段
    id      int
    addr    string
    name    string //和Person中的name同名
}

func main() {
    var s Student //变量声明

    //给Student的name，还是给Person赋值？
    s.name = "mike"

    //{Person:{name: sex:0 age:0} id:0 addr: name:mike}
    fmt.Printf("%+v\n", s)

    //默认只会给最外层的成员赋值
    //给匿名同名成员赋值，需要显示调用
    s.Person.name = "yoyo"
    //Person:{name:yoyo sex:0 age:0} id:0 addr: name:mike}
    fmt.Printf("%+v\n", s)
}
```

其它匿名字段

非结构体类型

所有的内置类型和自定义类型都是可以作为匿名字段的：

```
type mystr string //自定义类型

type Person struct {
    name string
    sex  byte
    age  int
}

type Student struct {
    Person // 匿名字段，结构体类型
    int    // 匿名字段，内置类型
    mystr  // 匿名字段，自定义类型
}

func main() {
    //初始化
    s1 := Student{Person{"mike", 'm', 18}, 1,
"bj"}

    //{Person:{name:mike sex:109 age:18} int:1
mystr:bj}
    fmt.Printf("%+v\n", s1)

    //成员的操作，打印结果：mike, m, 18, 1, bj
    fmt.Printf("%s, %c, %d, %d, %s\n", s1.name,
s1.sex, s1.age, s1.int, s1.mystr)
```

```
}
```

结构体指针类型

```
type Person struct { //人
    name string
    sex  byte
    age  int
}

type Student struct { //学生
    *Person // 匿名字段，结构体指针类型
    id      int
    addr    string
}

func main() {
    //初始化
    s1 := Student{&Person{"mike", 'm', 18}, 1,
"bj"}

    //{Person:0xc0420023e0 id:1 addr:bj}
    fmt.Printf("%+v\n", s1)
    //mike, m, 18
    fmt.Printf("%s, %c, %d\n", s1.name, s1.sex,
s1.age)
```

```
    //声明变量
    var s2 Student
    s2.Person = new(Person) //分配空间
    s2.name = "yoyo"
    s2.sex = 'f'
    s2.age = 20

    s2.id = 2
    s2.addr = "sz"

    //yoyo 102 20 2 20
    fmt.Println(s2.name, s2.sex, s2.age, s2.id,
s2.age)
}
```

方法

概述

在面向对象编程中，一个对象其实也就是一个简单的值或者一个变量，在这个对象中会包含一些函数，**这种带有接收者的函数，我们称为方法(method)**。 本质上，一个方法则是一个和特殊类型关联的函数。

一个面向对象的程序会用方法来表达其属性和对应的操作，这样使用这个对象的用户就不需要直接去操作对象，而是借助方法来做这些事情。

在Go语言中，可以给任意自定义类型（包括内置类型，但不包括指针类型）添加相应的方法。

方法总是绑定对象实例，并隐式将实例作为第一实参(receiver)，方法的语法如下：

```
func (receiver ReceiverType) funcName(parameters) (results)
```

- 参数 receiver 可任意命名。如方法中未曾使用，可省略参数名。
    - 参数 receiver 类型可以是 T 或 *T。基类型 T 不能是接口或指针。
- 不支持重载方法，也就是说，不能定义名字相同但是不同参数的方法。

为类型添加方法

基础类型作为接受者

```
type MyInt int //自定义类型，给int改名为MyInt

//在函数定义时，在其名字之前放上一个变量，即是一个方法
func (a MyInt) Add(b MyInt) MyInt { //面向对象
    return a + b
}

//传统方式的定义
func Add(a, b MyInt) MyInt { //面向过程
    return a + b
}
```

```
func main() {
    var a MyInt = 1
    var b MyInt = 1

    //调用func (a MyInt) Add(b MyInt)
    fmt.Println("a.Add(b) = ", a.Add(b))
//a.Add(b) =  2

    //调用func Add(a, b MyInt)
    fmt.Println("Add(a, b) = ", Add(a, b))
//Add(a, b) =  2
}
```

通过上面的例子可以看出，面向对象只是换了一种语法形式来表达。方法是函数的语法糖，因为receiver其实就是方法所接收的第1个参数。

注意：虽然方法的名字一模一样，但是如果接收者不一样，那么方法就不一样

结构体作为接受者

方法里面可以访问接收者的字段，调用方法通过点(.)访问，就像struct里面访问字段一样：

```
type Person struct {
    name string
    sex  byte
```

```
    age  int
}

func (p Person) PrintInfo() { //给Person添加方法
    fmt.Println(p.name, p.sex, p.age)
}

func main() {
    p := Person{"mike", 'm', 18} //初始化
    p.PrintInfo() //调用func (p Person) PrintInfo()
}
```

值语义和引用语义

```
type Person struct {
    name string
    sex  byte
    age  int
}

//指针作为接收者，引用语义
func (p *Person) SetInfoPointer() {
    //给成员赋值
    (*p).name = "yoyo"
    p.sex = 'f'
    p.age = 22
}
```

```go
//值作为接收者，值语义
func (p Person) SetInfoValue() {
    //给成员赋值
    p.name = "yoyo"
    p.sex = 'f'
    p.age = 22
}

func main() {
    //指针作为接收者，引用语义
    p1 := Person{"mike", 'm', 18} //初始化
    fmt.Println("函数调用前 = ", p1)   //函数调用前 = {mike 109 18}
    (&p1).SetInfoPointer()
    fmt.Println("函数调用后 = ", p1) //函数调用后 = {yoyo 102 22}

    fmt.Println("============================")

    p2 := Person{"mike", 'm', 18} //初始化
    //值作为接收者，值语义
    fmt.Println("函数调用前 = ", p2) //函数调用前 = {mike 109 18}
    p2.SetInfoValue()
    fmt.Println("函数调用后 = ", p2) //函数调用后 = {mike 109 18}
```

```
}
```

方法集

类型的方法集是指可以被该类型的值调用的所有方法的集合。

用实例实例 value 和 pointer 调用方法（含匿名字段）不受方法集约束，编译器编总是查找全部方法，并自动转换 receiver 实参。

类型 `*T` 的方法集

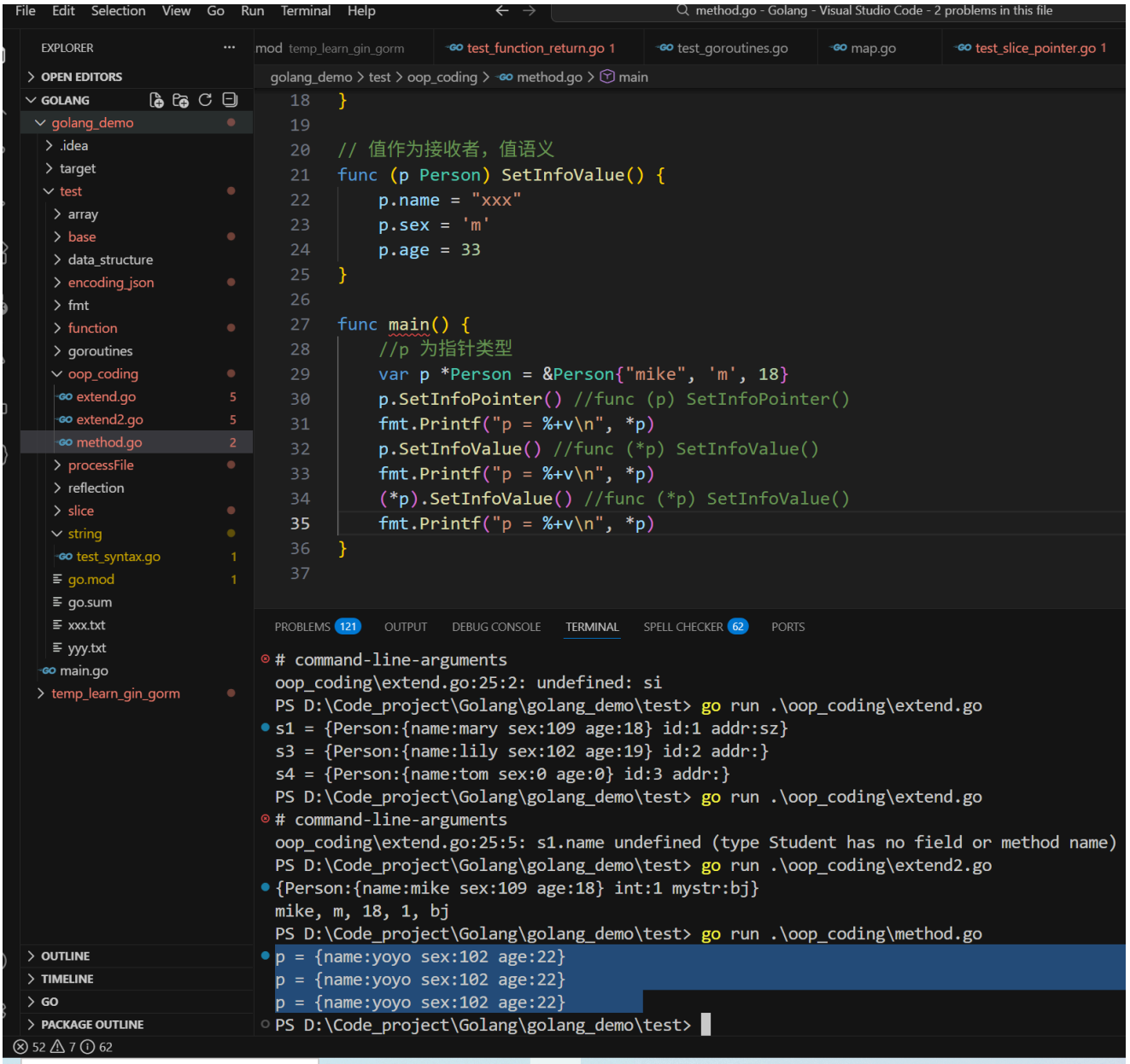
一个指向自定义类型的值的指针，它的方法集由该类型定义的所有方法组成，无论这些方法接受的是一个值还是一个指针。

如果在指针上调用一个接受值的方法，Go语言会聪明地将该指针解引用，并将指针所指的底层值作为方法的接收者。

类型 `*T` 方法集包含全部 receiver T + `*T` 方法：

```
type Person struct {
    name string
    sex  byte
    age  int
}

//指针作为接收者，引用语义
func (p *Person) SetInfoPointer() {
    (*p).name = "yoyo"
    p.sex = 'f'
    p.age = 22
```

```
}

//值作为接收者，值语义
func (p Person) SetInfoValue() {
    p.name = "xxx"
    p.sex = 'm'
    p.age = 33
}

func main() {
    //p 为指针类型
    var p *Person = &Person{"mike", 'm', 18}
    p.SetInfoPointer() //func (p) SetInfoPointer()

    p.SetInfoValue()    //func (*p) SetInfoValue()
    (*p).SetInfoValue() //func (*p) SetInfoValue()
}
```



类型 T 方法集

一个自定义类型值的方法集则由为该类型定义的接收者类型为值类型的方法组成，但是不包含那些接收者类型为指针的方法。

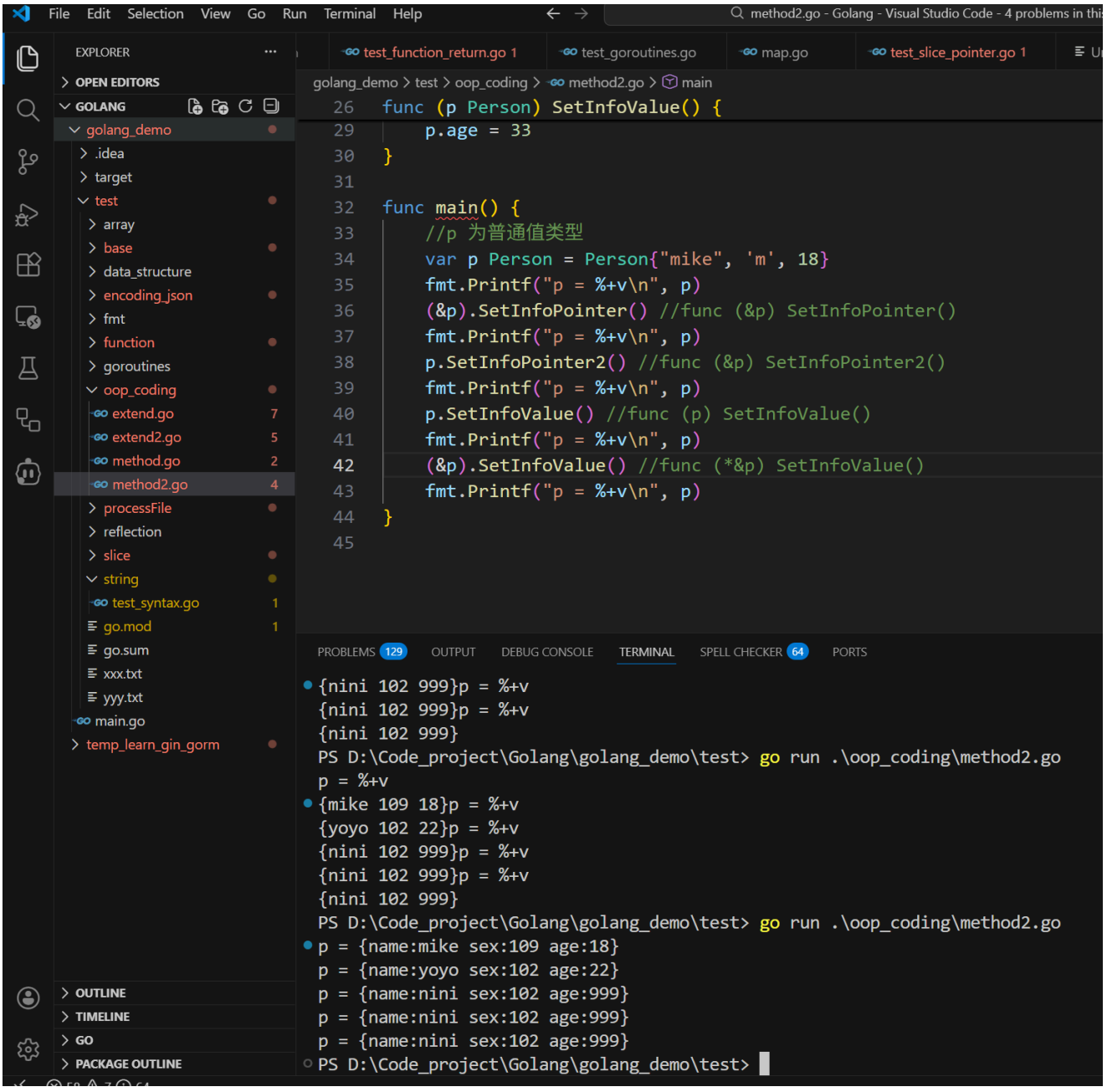
但这种限制通常并不像这里所说的那样，因为如果我们只有一个值，仍然可以调用一个接收者为指针类型的方法，这可以借助于Go语言传值的地址能力实现。

```
package main
```

```go
import "fmt"

type Person struct {

    name string

    sex  byte

    age  int

}

// 指针作为接收者，引用语义

func (p *Person) SetInfoPointer() {

    (*p).name = "yoyo"

    p.sex = 'f'

    p.age = 22

}


// 指针作为接收者，引用语义

func (p *Person) SetInfoPointer2() {
```

```
    (*p).name = "nini"

    p.sex = 'f'

    p.age = 999



// 值作为接收者，值语义
func (p Person) SetInfoValue() {

    p.name = "xxx"

    p.sex = 'm'

    p.age = 33

}

func main() {

    //p 为普通值类型

    var p Person = Person{"mike", 'm', 18}

    fmt.Printf("p = %+v\n", p)
```

```
    (&p).SetInfoPointer() //func (&p) SetInfoPointer()

    fmt.Printf("p = %+v\n", p)

    p.SetInfoPointer2() //func (&p) SetInfoPointer2()

    fmt.Printf("p = %+v\n", p)

    p.SetInfoValue() //func (p) SetInfoValue()

    fmt.Printf("p = %+v\n", p)

    (&p).SetInfoValue() //func (*&p) SetInfoValue()

    fmt.Printf("p = %+v\n", p)

}
```



匿名字段

方法的继承

如果匿名字段实现了一个方法，那么包含这个匿名字段的struct也能调用该方法。

```
type Person struct {
    name string
    sex  byte
    age  int
```

```
}

//Person定义了方法
func (p *Person) PrintInfo() {
    fmt.Printf("%s,%c,%d\n", p.name, p.sex, p.age)
}

type Student struct {
    Person // 匿名字段，那么Student包含了Person的所有字段
    id     int
    addr   string
}

func main() {
    p := Person{"mike", 'm', 18}
    p.PrintInfo()

    s := Student{Person{"yoyo", 'f', 20}, 2, "sz"}
    s.PrintInfo()
}
```

方法的重写

```
type Person struct {
    name string
    sex  byte
```

```
    age  int
}

//Person定义了方法
func (p *Person) PrintInfo() {
    fmt.Printf("Person: %s,%c,%d\n", p.name,
p.sex, p.age)
}

type Student struct {
    Person // 匿名字段，那么Student包含了Person的所有
字段
    id     int
    addr   string
}

//Student定义了方法
func (s *Student) PrintInfo() {
    fmt.Printf("Student: %s,%c,%d\n", s.name,
s.sex, s.age)
}

func main() {
    p := Person{"mike", 'm', 18}
    p.PrintInfo() //Person: mike,m,18

    s := Student{Person{"yoyo", 'f', 20}, 2, "sz"}
    s.PrintInfo()        //Student: yoyo,f,20
```

```
    s.Person.PrintInfo() //Person: yoyo,f,20
}
```

表达式

类似于我们可以对函数进行赋值和传递一样，方法也可以进行赋值和传递。

根据调用者不同，方法分为两种表现形式：方法值和方法表达式。两者都可像普通函数那样赋值和传参，区别在于方法值绑定实例，而方法表达式则须显式传参。

方法值

```
type Person struct {
    name string
    sex  byte
    age  int
}

func (p *Person) PrintInfoPointer() {
    fmt.Printf("%p, %v\n", p, p)
}

func (p Person) PrintInfoValue() {
    fmt.Printf("%p, %v\n", &p, p)
}

func main() {
```

```
    p := Person{"mike", 'm', 18}
    p.PrintInfoPointer() //0xc0420023e0, &{mike
109 18}

    pFunc1 := p.PrintInfoPointer //方法值，隐式传递
receiver
    pFunc1()                     //0xc0420023e0, &
{mike 109 18}

    pFunc2 := p.PrintInfoValue
    pFunc2() //0xc042048420, {mike 109 18}
}
```

方法表达式

```
type Person struct {
    name string
    sex  byte
    age  int
}

func (p *Person) PrintInfoPointer() {
    fmt.Printf("%p, %v\n", p, p)
}

func (p Person) PrintInfoValue() {
    fmt.Printf("%p, %v\n", &p, p)
```

```
}

func main() {
    p := Person{"mike", 'm', 18}
    p.PrintInfoPointer() //0xc0420023e0, &{mike
109 18}

    //方法表达式，  须显式传参
    //func pFunc1(p *Person))
    pFunc1 := (*Person).PrintInfoPointer
    pFunc1(&p) //0xc0420023e0, &{mike 109 18}

    pFunc2 := Person.PrintInfoValue
    pFunc2(p) //0xc042002460, {mike 109 18}
}
```

接口

概述

在Go语言中，接口(interface)是一个自定义类型，接口类型具体描述了一系列方法的集合。

接口类型是一种抽象的类型，它不会暴露出它所代表的对象的内部值的结构和这个对象支持的基础操作的集合，它们只会展示出它们自己的方法。因此接口类型不能将其实例化。

Go通过接口实现了鸭子类型(duck-typing)：“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”。我们并不关心对象是什么类型，到底是不是鸭子，只关心行为。

接口的使用

接口的定义

```
type Humaner interface {
    SayHi()
}
```

- 单方法接口通常以 `er` 结尾

- 接口只有方法声明，没有实现，没有数据字段
- 接口可以匿名嵌入其它接口，或嵌入到结构中

接口的实现

接口是用来定义行为的类型。这些被定义的行为不由接口直接实现，而是通过方法由用户定义的类型实现，一个实现了这些方法的具体类型是这个接口类型的实例。

如果用户定义的类型实现了某个接口类型声明的一组方法，那么这个用户定义的类型的值就可以赋给这个接口类型的值。这个赋值会把用户定义的类型的值存入接口类型的值。

```
type Humaner interface {
    SayHi()
}
```

```go
type Student struct { //学生
    name  string
    score float64
}

//Student实现SayHi()方法
func (s *Student) SayHi() {
    fmt.Printf("Student[%s, %f] say hi!!\n",
s.name, s.score)
}

type Teacher struct { //老师
    name  string
    group string
}

//Teacher实现SayHi()方法
func (t *Teacher) SayHi() {
    fmt.Printf("Teacher[%s, %s] say hi!!\n",
t.name, t.group)
}

type MyStr string

//MyStr实现SayHi()方法
func (str MyStr) SayHi() {
    fmt.Printf("MyStr[%s] say hi!!\n", str)
}
```

```go
//普通函数，参数为Humaner类型的变量i
func WhoSayHi(i Humaner) {
    i.SayHi()
}

func main() {
    s := &Student{"mike", 88.88}
    t := &Teacher{"yoyo", "Go语言"}
    var tmp MyStr = "测试"

    s.SayHi()   //Student[mike, 88.880000] say hi!!
    t.SayHi()   //Teacher[yoyo, Go语言] say hi!!
    tmp.SayHi() //MyStr[测试] say hi!!

    //多态，调用同一接口，不同表现
    WhoSayHi(s)   //Student[mike, 88.880000] say hi!!
    WhoSayHi(t)   //Teacher[yoyo, Go语言] say hi!!
    WhoSayHi(tmp) //MyStr[测试] say hi!!

    x := make([]Humaner, 3)
    //这三个都是不同类型的元素，但是他们实现了interface同一个接口
    x[0], x[1], x[2] = s, t, tmp
    for _, value := range x {
        value.SayHi()
```

```
    }
    /*
        Student[mike, 88.880000] say hi!!
        Teacher[yoyo, Go语言] say hi!!
        MyStr[测试] say hi!!
    */
}
```

通过上面的代码，你会发现接口就是一组抽象方法的集合，它必须由其他非接口类型实现，而不能自我实现。

接口组合

接口嵌入

如果一个interface1作为interface2的一个嵌入字段，那么interface2隐式的包含了interface1里面的方法。

```
type Humaner interface {
    SayHi()
}

type Personer interface {
    Humaner //这里想写了SayHi()一样
    Sing(lyrics string)
}

type Student struct { //学生
    name  string
```

```
    score float64
}

//Student实现SayHi()方法
func (s *Student) SayHi() {
    fmt.Printf("Student[%s, %f] say hi!!\n",
s.name, s.score)
}

//Student实现Sing()方法
func (s *Student) Sing(lyrics string) {
    fmt.Printf("Student sing[%s]!!\n", lyrics)
}

func main() {
    s := &Student{"mike", 88.88}

    var i2 Personer
    i2 = s
    i2.SayHi()     //Student[mike, 88.880000] say
hi!!
    i2.Sing("学生哥") //Student sing[学生哥]!!
}
```

接口转换

超集接口对象可转换为子集接口，反之出错：

```go
type Humaner interface {
    SayHi()
}

type Personer interface {
    Humaner //这里像写了SayHi()一样
    Sing(lyrics string)
}

type Student struct { //学生
    name  string
    score float64
}

//Student实现SayHi()方法
func (s *Student) SayHi() {
    fmt.Printf("Student[%s, %f] say hi!!\n",
s.name, s.score)
}

//Student实现Sing()方法
func (s *Student) Sing(lyrics string) {
    fmt.Printf("Student sing[%s]!!\n", lyrics)
}

func main() {
    //var i1 Humaner = &Student{"mike", 88.88}
    //var i2 Personer = i1 //err
```

```
    //Personer为超集，Humaner为子集
    var i1 Personer = &Student{"mike", 88.88}
    var i2 Humaner = i1
    i2.SayHi() //Student[mike, 88.880000] say hi!!
}
```

空接口

空接口(interface{})不包含任何的方法，正因为如此，所有的类型都实现了空接口，因此空接口可以存储任意类型的数值。它有点类似于C语言的`void *`类型。

```
var v1 interface{} = 1      // 将int类型赋值给interface{}
    var v2 interface{} = "abc" // 将string类型赋值给interface{}
    var v3 interface{} = &v2   // 将*interface{}类型赋值给interface{}
    var v4 interface{} = struct{ X int }{1}
    var v5 interface{} = &struct{ X int }{1}
```

当函数可以接受任意的对象实例时，我们会将其声明为interface{}，最典型的例子是标准库fmt中PrintXXX系列的函数，例如：

```
func Printf(fmt string, args ...interface{})
    func Println(args ...interface{})
```

类型查询

我们知道interface的变量里面可以存储任意类型的数值(该类型实现了interface)。那么我们怎么反向知道这个变量里面实际保存了的是哪个类型的对象呢？目前常用的有两种方法：

- comma-ok断言
- switch测试

comma-ok断言

Go语言里面有一个语法，可以直接判断是否是该类型的变量：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类型的数值，那么ok返回true，否则返回false。
示例代码：

```
type Element interface{}

type Person struct {
    name string
    age  int
}
```

```
func main() {
    list := make([]Element, 3)
    list[0] = 1       // an int
    list[1] = "Hello" // a string
    list[2] = Person{"mike", 18}

    for index, element := range list {
        if value, ok := element.(int); ok {
            fmt.Printf("list[%d] is an int and its
value is %d\n", index, value)
        } else if value, ok := element.(string);
ok {
            fmt.Printf("list[%d] is a string and
its value is %s\n", index, value)
        } else if value, ok := element.(Person);
ok {
            fmt.Printf("list[%d] is a Person and
its value is [%s, %d]\n", index, value.name,
value.age)
        } else {
            fmt.Printf("list[%d] is of a different
type\n", index)
        }
    }

    /*  打印结果:
    list[0] is an int and its value is 1
    list[1] is a string and its value is Hello
```

```
    list[2] is a Person and its value is [mike,
18]
    */
}
```

switch测试

```
type Element interface{}

type Person struct {
    name string
    age  int
}

func main() {
    list := make([]Element, 3)
    list[0] = 1        //an int
    list[1] = "Hello" //a string
    list[2] = Person{"mike", 18}

    for index, element := range list {
        switch value := element.(type) {
        case int:
            fmt.Printf("list[%d] is an int and its
value is %d\n", index, value)
        case string:
            fmt.Printf("list[%d] is a string and
```

```
its value is %s\n", index, value)
        case Person:
            fmt.Printf("list[%d] is a Person and
its value is [%s, %d]\n", index, value.name,
value.age)
        default:
            fmt.Println("list[%d] is of a
different type", index)
        }
    }
}
```