Go 边看边练 -《Go 学习笔记》系列（十二）

作者：88250
原文链接：https://ld246.com/article/1438845728987
来源网站：链滴

上一篇： https://hacpai.com/article/1438763483577

ToC

6.1 接口定义

接口是一个或多个方法签名的集合，任何类型的方法集中只要拥有与之对应的全部方法，就表示它 "现" 了该接口，无须在该类型上显式添加接口声明。

所谓对应方法，是指有相同名称、参数列表 (不包括参数名) 以及返回值。当然，该类型还可以有其他法。

- 接口命名习惯以 er 结尾，结构体。
- 接口只有方法签名，没有实现。
- 接口没有数据字段。
- 可在接口中嵌入其他接口。
- 类型可实现多个接口。

<iframe style="border:1px solid" src="https://wide.b3log.org/playground/258b8a46339ef054189ed004dd4f3b8.go?embed=true" width="99%" height="750"></iframe>

空接口 interface{} 没有任何方法签名，也就意味着任何类型都实现了空接口。其作用类似面向对象言中的根对象 Object。

```
func Print(v interface{}) {
    fmt.Printf("%T: %v\n", v, v)
}

func main() {
    Print(1)
    Print("Hello, World!")
}
```

输出：

```
int: 1
string: Hello, World!
```

匿名接口可用作变量类型，或结构成员。

<iframe style="border:1px solid" src="https://wide.b3log.org/playground/bb45f4ffd96bf6c852953837bc8f8de.go?embed=true" width="99%" height="600"></iframe>

6.2 执行机制

接口对象由接口表 (interface table) 指针和数据指针组成。

runtime.h

```
struct Iface
{
    Itab* tab;
    void* data;
};

struct Itab
{
    InterfaceType* inter;
    Type* type;
    void (*fun[])(void);
};
```

接口表存储元数据信息，包括接口类型、动态类型，以及实现接口的方法指针。无论是反射还是通过口调用方法，都会用到这些信息。

数据指针持有的是目标对象的只读复制品，复制完整对象或指针。

<iframe style="border:1px solid" src="https://wide.b3log.org/playground/1109087790153bdbaf0d0d139c79999.go?embed=true" width="99%" height="520"></iframe>

接口转型返回临时对象，只有使用指针才能修改其状态。

<iframe style="border:1px solid" src="https://wide.b3log.org/playground/0e60e29cd72bff027b64fe97e7394a7.go?embed=true" width="99%" height="520"></iframe>

只有 tab 和 data 都为 nil 时，接口才等于 nil。

```
var a interface{} = nil // tab = nil, data = nil
var b interface{} = (*int)(nil) // tab 包含 *int 类型信息, data = nil

type iface struct {
    itab, data uintptr
}

ia := *(*iface)(unsafe.Pointer(&a))
ib := *(*iface)(unsafe.Pointer(&b))

fmt.Println(a == nil, ia)
fmt.Println(b == nil, ib, reflect.ValueOf(b).IsNil())
```

输出：

```
true {0 0}
false {505728 0} true
```

6.3 接口转换

利用类型推断，可判断接口对象是否某个具体的接口或类型。

```
type User struct {
    id int
    name string
}

func (self *User) String() string {
    return fmt.Sprintf("%d, %s", self.id, self.name)
}

func main() {
    var o interface{} = &User{1, "Tom"}

    if i, ok := o.(fmt.Stringer); ok { // ok-idiom
        fmt.Println(i)
    }

    u := o.(*User)
    // u := o.(User) // panic: interface is *main.User, not main.User
    fmt.Println(u)
}
```

还可用 switch 做批量类型判断，不支持 fallthrough。

```
func main() {
    var o interface{} = &User{1, "Tom"}
```

```
    switch v := o.(type) {
    case nil: // o == nil
        fmt.Println("nil")
    case fmt.Stringer: // interface
        fmt.Println(v)
    case func() string: // func
        fmt.Println(v())
    case *User: // *struct
        fmt.Printf("%d, %s\n", v.id, v.name)
    default:
        fmt.Println("unknown")
    }
}
```

超集接口对象可转换为子集接口，反之出错。

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

type Printer interface {
    String() string
    Print()
}

type User struct {
    id int
    name string
}

func (self *User) String() string {
    return fmt.Sprintf("%d, %v", self.id, self.name)
}

func (self *User) Print() {
    fmt.Println(self.String())
}

func main() {
    var o Printer = &User{1, "Tom"}
    var s Stringer = o
    fmt.Println(s.String())
}
```

6.4 接口技巧

让编译器检查，以确保某个类型实现接口。

```
var _ fmt.Stringer = (*Data)(nil)
```

某些时候，让函数直接 "实现" 接口能省不少事。

```
type Tester interface {
    Do()
}

type FuncDo func()
func (self FuncDo) Do() { self() }

func main() {
    var t Tester = FuncDo(func() { println("Hello, World!") })
    t.Do()
}
```

下一篇： https://hacpai.com/article/1438938175118

- **本系列是基于雨痕的《Go 学习笔记》（第四版）整理汇编而成，非常感谢雨痕的辛勤付出与分享！**
- 转载请注明：文章转载自： **黑客与画家的社区** [https://hacpai.com]
- 如果你觉得本章节做得不错，请在下面打赏一下吧~

社区小贴士

- 关注标签 [golang] 可以方便查看 Go 相关帖子
- 关注作者后如有新帖将会收到通知