反射 reflect

```
package reflect
```

why reflect?

e.g.检查类型 `fmt.Sprintf`，接收**任意参数**并返回其格式化后的字符串（e.g. string类型返回它自己，int类型先转换为string再返回...）

想一想该如何实现呢？

```
func Sprint(x interface{}) string {
    type stringer interface {
        String() string
    }
    switch x := x.(type) {
    // 如果实现了String()方法，直接return
    case stringer:
        return x.String()
    // 如果是string类型...
    case string:
        return x
    // 如果是int类型...
    case int:
        return strconv.Itoa(x)
    // 如果是int8, int16, int32, int64...
    // 如果是uint8...
    // 如果是...
    // 如果是bool类型...
    case bool:
        if x {
            return "true"
        }
        return "false"
    // 还有一大堆呢
    // 还可能有自定义类型呢？type Weekday int, type struct S{...}
    // 这么多怎么办呢？
    default:
        // array, chan, func, map, pointer, slice, struct
        return "???"
    }
}
```

敲个代码人山人海的多不好呀😊

reflect.Type & reflect.Value

type & value

interface值 = 动态类型 + 动态值

```
func refTandV() {
        var w io.Writer = os.Stdout
        t := reflect.TypeOf(w) // 动态值
        v := reflect.ValueOf(w) // 动态类型
        fmt.Println(t, v)
        // vs fmt.Printf
        fmt.Printf("%T %v\n", w, w) // %T Type, %v value
        // 其实就是靠reflect.T/V实现的!

        // 我们再来看看t和v都是什么类型的
        tt := reflect.TypeOf(t) // *reflect.rtype (reflect.Type)
        tv := reflect.TypeOf(v) // reflect.Value
        fmt.Println(tt, tv)

        // reflect.Type和reflect.Value类型也实现了String()方法（满足fmt.Stringer接口）
        fmt.Println(t.String(), v.String()) // *os.File <*os.File Value>

        // reflect.Value.Type() = reflect.Type
        fmt.Println(v.Type(), t) // 一样的，*os.File *os.File
}
```

正着获取可以，反向重建呢?

```
func val2inter() {
        // interface->value, then value->interface?
        var num int = 3 // <int, 3>
        v := reflect.ValueOf(num) // reflect.Value
        x := v.Interface() // interface{}
        i := x.(int) // int
        fmt.Println(i)
}
```

that's why reflect!

类型? `reflect.Value.Kind()` 方法!

```
func formatAtom(v interface{}) string {
    switch reflect.ValueOf(v).Kind() {
    // reflect.Value.Kind()是有限种类的!
    /// Invalid家族
    case reflect.Invalid:
        return "invalid"
    // Int家族
    case reflect.Int, reflect.Int8, reflect.Int16,
        reflect.Int32, reflect.Int64:
        return strconv.FormatInt(v.Int(), 10)
    // UInt家族
    case reflect.Uint, reflect.Uint8, reflect.Uint16,
        reflect.Uint32, reflect.Uint64, reflect.Uintptr:
        return strconv.FormatUint(v.Uint(), 10)
    // ...floating-point and complex cases omitted for brevity...
    // ...float家族和complex复数家族
    // Bool家族
    case reflect.Bool:
        return strconv.FormatBool(v.Bool())
    // String家族
    case reflect.String:
        return strconv.Quote(v.String())
    // 引用类型家族
    case reflect.Chan, reflect.Func, reflect.Ptr, reflect.Slice, reflect.Map:
        return v.Type().String() + " 0x" +
            strconv.FormatUint(uint64(v.Pointer()), 16)
    // 其他乱七八糟的家族
    default: // reflect.Array, reflect.Struct, reflect.Interface
        return v.Type().String() + " value"
    }
}
```

e.g.递归打印 Display()

对于这样的结构体：

```
type Movie struct {
    Title, Subtitle string
    Year            int
    Color           bool
    Oscars          []string
    Sequel          *string
}
```

想要打印其中的具体信息，来一个Display()函数

乱七八糟的类型—— `reflect.Value.Kind()`！

```
func Display(path string, vi interface{}) {
        v := reflect.ValueOf(vi)
        switch v.Kind() {
        case reflect.Invalid:
                fmt.Printf("%s = invalid\n", path)
        // 对Slice和Array
        // 递归处理每一个元素
        case reflect.Slice, reflect.Array:
                for i := 0; i < v.Len(); i++ {
                        Display(fmt.Sprintf("%s[%d]", path, i), v.Index(i))
                }
        // 对于Struct
        // 递归处理每一个成员
        case reflect.Struct:
                for i := 0; i < v.NumField(); i++ {
                        fieldPath := fmt.Sprintf("%s.%s", path, v.Type().Field(i).Name)
                        Display(fieldPath, v.Field(i))
                }
        // 对于指针
        // 打印时多一个(*...)
        case reflect.Ptr:
                if v.IsNil() {
                        fmt.Printf("%s = nil\n", path)
                } else {
                        Display(fmt.Sprintf("(*%s)", path), v.Elem())
                }
        // 对于接口
        // 打印动态类型，递归处理动态值
        case reflect.Interface:
                if v.IsNil() {
                        fmt.Printf("%s = nil\n", path)
                } else {
                        fmt.Printf("%s.type = %s\n", path, v.Elem().Type())
                        Display(path+".value", v.Elem())
                }
        default:
                fmt.Printf("%s = %v\n", path, v.Interface())
        }
}
```

实例化一个 Movie 并打印

```
func DisplayEx() {
        strangelove := utils.Movie{
                Title:    "Dr. Strangelove",
                Subtitle: "How I Learned to Stop Worrying and Love the Bomb",
                Year:     1964,
                Color:    false,

                Oscars: []string{
                        "Best Actor (Nomin.)",
                        "Best Adapted Screenplay (Nomin.)",
                        "Best Director (Nomin.)",
                        "Best Picture (Nomin.)",
                },
        }
        utils.Display("strangelove", strangelove)
}
```

预期结果：

```
strangelove.Title = Dr. Strangelove
strangelove.Subtitle = How I Learned to Stop Worrying and Love the Bomb
strangelove.Year = 1964
strangelove.Color = false
strangelove.Oscars[0] = Best Actor (Nomin.)
strangelove.Oscars[1] = Best Adapted Screenplay (Nomin.)
strangelove.Oscars[2] = Best Director (Nomin.)
strangelove.Oscars[3] = Best Picture (Nomin.)
strangelove.Sequel = nil
```

（但是运行有点卡不知道怎么回事，是不是递归惹的祸😋）

通过 `reflect.Value` 修改值

只读？也能写！😘

变量？一个变量就是一个**可寻址的内存空间**，里面存储了一个值，并且存储的值可以通过内存地址来更新

可取地址是变量？！😙
判断题：下面哪些是变量？

```
x := 2                   // value   type    variable?
a := reflect.ValueOf(2)  // 2       int     no
b := reflect.ValueOf(x)  // 2       int     no
c := reflect.ValueOf(&x) // &x      *int    no
d := c.Elem()            // 2       int     yes (x)
```

`d := reflect.ValueOf(&x).Elem()` 一通王八拳就成了变量了

1. 先获取x的指针(&x)的 `reflect.Value`
2. `.Elem()` 解引用指针，获取指针指向**实际值**的 `reflect.Value`

这样，v 和 x 共享同一块内存，通过 v 修改值会影响 x!
v就像x的引用，但是是 `reflect.Value` 类型的！ 😋 👍

通过v修改x（如果v是可设置的Value）

```
func modifyValue() {
    var x float64 = 3.4
    v := reflect.ValueOf(&x).Elem() // 必须获取可设置的Value

    if v.CanSet() { // CanSet()检查是否是可取地址并可被修改的
        // v.Set(reflect.ValueOf(7.1))
        v.SetFloat(7.1)
        // 这两种set方法都OK!
        // 还有SetInt(), SetString()...
        fmt.Println(x) // 输出: 7.1
    }
}
```

显示类型和方法集（所有method）

`reflect.Type`！

```
func Print(x interface{}) {
    v := reflect.ValueOf(x)
    t := v.Type()
    fmt.Printf("type %s\n", t)

    for i := 0; i < v.NumMethod(); i++ {
        methType := v.Method(i).Type()
        fmt.Printf("func (%s) %s%s\n", t, t.Method(i).Name,
            strings.TrimPrefix(methType.String(), "func"))
    }
}
```

拿牢玩家 os.Stdout 开刀

```
Print(os.Stdout)
```

输出

```
type *os.File
func (*os.File) Chdir() error
func (*os.File) Chmod(fs.FileMode) error
func (*os.File) Chown(int, int) error
func (*os.File) Close() error
func (*os.File) Fd() uintptr
func (*os.File) Name() string
func (*os.File) Read([]uint8) (int, error)
func (*os.File) ReadAt([]uint8, int64) (int, error)
func (*os.File) ReadDir(int) ([]fs.DirEntry, error)
func (*os.File) ReadFrom(io.Reader) (int64, error)
func (*os.File) Readdir(int) ([]fs.FileInfo, error)
func (*os.File) Readdirnames(int) ([]string, error)
func (*os.File) Seek(int64, int) (int64, error)
func (*os.File) SetDeadline(time.Time) error
func (*os.File) SetReadDeadline(time.Time) error
func (*os.File) SetWriteDeadline(time.Time) error
func (*os.File) Stat() (fs.FileInfo, error)
func (*os.File) Sync() error
func (*os.File) SyscallConn() (syscall.RawConn, error)
func (*os.File) Truncate(int64) error
func (*os.File) Write([]uint8) (int, error)
func (*os.File) WriteAt([]uint8, int64) (int, error)
func (*os.File) WriteString(string) (int, error)
func (*os.File) WriteTo(io.Writer) (int64, error)
```