链滴

反射和类型断言

作者：opsxdev
原文链接：https://ld246.com/article/1616684816167
来源网站：链滴

反射和类型断言

1.反射简介

反射机制：在编译时不知道类型的情况下，可更新变量、在运行时查看值、调用方法以及直接对它们布局进行操作，反射也让我们可以把类型当做头等值

- 通过反射获取类型信息

在 Go 语言中通过调用 reflect.TypeOf 函数，我们可以从一个任何非接口类型的值创建一个 reflect.Type 值。reflect.Type 值表示着此非接口值的类型。通过此值，我们可以得到很多此非接口类型的信息当然，我们也可以将一个接口值传递给一个 reflect.TypeOf 函数调用，但是此调用将返回一个表示着接口值的动态类型的 reflect.Type 值。

实际上，reflect.TypeOf 函数的唯一参数的类型为 interface{}，reflect.TypeOf 函数将总是返回一个示着此唯一接口参数值的动态类型的 reflect.Type 值。

```
package main

import (
    "fmt"
    "reflect"
)

func main(){
    //空接口可以存储任意类型数据   空接口类型数据无法计算
    var i interface{} = 123
    //var j interface{} = 456
    //fmt.Println(i + j)  无法进行计算
```

```
    t1 := reflect.TypeOf(i)

    fmt.Printf("%T\n",t1)
    fmt.Println(t1)
}
C:\Users\vSphere\go\src\day04>go run 06.go
*reflect.rtype
int
```

- 接口数据类型进行计算

```
package main

import (
    "fmt"
    "reflect"
)

func main(){
    //空接口可以存储任意类型数据   空接口类型数据无法计算
    var i interface{} = 123
    var j interface{} = 456
    //fmt.Println(i + j)  无法进行计算

    t1 := reflect.TypeOf(i)
    t2 := reflect.TypeOf(j)

    fmt.Printf("%T\n",t1)
    fmt.Println(t1)

    if t1 == t2 {
        v1 := reflect.ValueOf(i).Int()
        v2 := reflect.ValueOf(j).Int()
        fmt.Println(v1 + v2)
    }
}
C:\Users\vSphere\go\src\day04>go run 06.go
*reflect.rtype
int
579
```

- 函数： reflect.ValueOf().Int()

```
package main

import (
    "fmt"
    "reflect"
)

func main(){
    //空接口可以存储任意类型数据   空接口类型数据无法计算
    var i interface{} = 123
    var j interface{} = 456
```

```
    //fmt.Println(i + j)  无法进行计算

    t1 := reflect.TypeOf(i)
    t2 := reflect.TypeOf(j)

    fmt.Printf("%T\n",t1)
    fmt.Println(t1)

    if t1 == t2 {
        v1 := reflect.ValueOf(i).Int()
        v2 := reflect.ValueOf(j).Int()
        fmt.Println(v1 + v2)

        fmt.Println(v1)
        fmt.Printf("%T\n",v1)
    }
}
C:\Users\vSphere\go\src\day04>go run 06.go
*reflect.rtype
int
579
123
int64
```

2.reflect.Type

Type表示go语言的一个类型，它是一个有很多方法的接口，这些方法可以用来识别类型以及透视类的组成部分，比如一个结构的各个字段或者一个函数的各个参数

reflect.Type接口只有一个实现，即类型描述符，接口值中的动态类型也是类型描述符

- Reflect.Type函数接受任何的interface{}参数，并且把接口中的动态类型以reflect.Type形式返回

```
t := reflect.TypeOf(3)    //一个reflect.Type
//上面的TypeOf(3)调用把数值3赋给interface{}参数,把一个具体值赋给一个接口类型时发生一个隐式型转换,转换会生成一个包含两部分内容的接口值:动态类型部分三操作数的值
```

- 因为reflect.TypeOf返回一个接口值对应的动态类型,所以它返回总是具体类型,而不是接口类型

```
var  w  io.Writter = os.Stdout
fmt.Println(reflect.TypeOf(w))   //*os.File而不是io.Writer
```

注意,reflect.Type满足fmt.Stringer,因为输出一个接口值的动态类型在调试和日志中很常用

3.reflect.Value

reflect.Value可以包含一个任意类型的值

reflect.ValueOf函数接受任意的interface{}并将接口的动态值以reflect.Value的形式返回,与reflect.TyeOf类似,reflect.ValueOf的返回值也都是具体值,不过reflect.Value也可以包含一个接口值.

```
v := reflect.ValueOf(3)   //一个reflect.Value
```

```
fmt.Println(v)   //3
fmt.Println(v.String()) //int
```

● 另一个与reflect.Type类似的是,reflect.Value也满足fmt.Stringer,但除非value包含的是一个字串,否则string方法的结果仅仅暴露了类型

```
//调用value的type方法会把它的类型以reflect.Type方式返回
t := v.Type()
fmt.Println(t.String())//int
```

● reflect.ValueOf的逆操作是reflect.Value.Interface方法,它返回一个interface{}接口,与reflect.Value包含同一个具体值

```
v := reflect.ValueOf(3)  // a reflect.Value
x :=v.Interface() //an  interface{}
i := x.(int) //an int
fmt.Printf("%d\n",i)
```

reflect.Value和interface()都可以包含任意的值,二者的区别是空接口隐藏了值的布局信息\内置操作相关方法,所以除非我们知道它的动态类型,并用一个类型断言来渗透进去,否则我们对包含值能做的事很少

4.通用的格式化函数

不用类型分支,我们用reflect.Value的Kind方法来区分不同的类型,尽管有无限种类型,但类型的分类只少数几种:基础类型Bool,String以及各种数字类型;聚合类型Array和Struct;引用类型Chan,Func,Ptr,Slic和Map,接口类型interface;最后还有Invalid表示它还没有任何值

```
// Any formats any value as a string.
func Any(value interface{}) string {
	return formatAtom(reflect.ValueOf(value))
}

// formatAtom formats a value without inspecting its internal structure.
func formatAtom(v reflect.Value) string {
	switch v.Kind() {
	case reflect.Invalid:
		return "invalid"
	case reflect.Int, reflect.Int8, reflect.Int16,
		reflect.Int32, reflect.Int64:
		return strconv.FormatInt(v.Int(), 10)
	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...
	case reflect.Bool:
		return strconv.FormatBool(v.Bool())
	case reflect.String:
		return strconv.Quote(v.String())
	case reflect.Chan, reflect.Func, reflect.Ptr, reflect.Slice, reflect.Map:
		return v.Type().String() + " 0x" +
			strconv.FormatUint(uint64(v.Pointer()), 16)
	default: // reflect.Array, reflect.Struct, reflect.Interface
		return v.Type().String() + " value"
```

```
        }
}
```

该函数把每个值当做一个没有内部结构且不可分割的物体,对于聚合类型以及接口,它只输出了值的类型对于引用类型(通道,函数,指针,slice,map),它输出了类型和以十六进制表示的引用地址

改进,Display递归的值显示器

调试工具函数Display,这个函数对给定的任意一个复杂值x,输出这个复杂值的完整结构,并对找到的每元素标上这个元素的路径

```
//!+Display

func Display(name string, x interface{}) {
        fmt.Printf("Display %s (%T):\n", name, x)
        display(name, reflect.ValueOf(x))
}

//!-Display

// formatAtom formats a value without inspecting its internal structure.
// It is a copy of the the function in gopl.io/ch11/format.
func formatAtom(v reflect.Value) string {
        switch v.Kind() {
        case reflect.Invalid:
                return "invalid"
        case reflect.Int, reflect.Int8, reflect.Int16,
                reflect.Int32, reflect.Int64:
                return strconv.FormatInt(v.Int(), 10)
        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...
        case reflect.Bool:
                if v.Bool() {
                        return "true"
                }
                return "false"
        case reflect.String:
                return strconv.Quote(v.String())
        case reflect.Chan, reflect.Func, reflect.Ptr,
                reflect.Slice, reflect.Map:
                return v.Type().String() + " 0x" +
                        strconv.FormatUint(uint64(v.Pointer()), 16)
        default: // reflect.Array, reflect.Struct, reflect.Interface
                return v.Type().String() + " value"
        }
}

//!+display
func display(path string, v reflect.Value) {
        switch v.Kind() {
        case reflect.Invalid:
```

```
            fmt.Printf("%s = invalid\n", path)
        case reflect.Slice, reflect.Array:
            for i := 0; i < v.Len(); i++ {
                display(fmt.Sprintf("%s[%d]", path, i), v.Index(i))
            }
        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))
            }
        case reflect.Map:
            for _, key := range v.MapKeys() {
                display(fmt.Sprintf("%s[%s]", path,
                    formatAtom(key)), v.MapIndex(key))
            }
        case reflect.Ptr:
            if v.IsNil() {
                fmt.Printf("%s = nil\n", path)
            } else {
                display(fmt.Sprintf("(*%s)", path), v.Elem())
            }
        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: // basic types, channels, funcs
            fmt.Printf("%s = %s\n", path, formatAtom(v))
        }
}
```

go

5.reflect.Value设置值

一个变量是一个可寻址的存储区域,其中包含了一个值,并且它的值可以通过这个地址来更新

示例,调用reflect.ValueOf(&x).Elem()来获得任意x可寻址的value值

```
x := 2 //值类型变量
a := reflect.ValueOf(2) //no
b := reflect.ValueOf(x) //no
c := reflect.Value(&x) //no
d := x.Elem()    //2    int    yes(x)
```

通过变量的CanAddr方法来询问reflect.Value变量是否寻址

```
fmt.Println(d.CanAddr)
```

我们可以通过一个指针间接获取一个可寻址的reflect.Value,从一个可寻址的reflect.Value()获取变量要三步.

- 首先,调用Addr(),返回一个Value,其中包含一个指向变量的指针,
  - 接下来,在这个Value上调用Interface(),会返回一个包含这个指针的interface{}值
  - 最后,如果我们值这个变量的类型,可以使用类型断言把接口内容转换为一个普通指针,之后通过这指针来更新变量

```
x := 2
d := reflect.ValueOf(&x).Elem()   //d代表变量x
px := d.Addr().Interface().(*int) //px:=&x
*px = 3    //x = 3
fmt.Prinln(x) //3
```

平常由编译器来检查的那些可赋值性条件,在这种情况下则是在运行时由Set方法来检查,确保这个值对类型变量是可赋值的是很重要的一件事情,在不可寻址的reflect.Value上调用Set方法也会崩溃

利用反射填充数据结构

decode.go

```
// Package sexpr provides a means for converting Go objects to and
// from S-expressions.
package sexpr

import (
    "bytes"
    "fmt"
    "reflect"
    "strconv"
    "text/scanner"
)

//!+Unmarshal
// Unmarshal parses S-expression data and populates the variable
// whose address is in the non-nil pointer out.
func Unmarshal(data []byte, out interface{}) (err error) {
    lex := &lexer{scan: scanner.Scanner{Mode: scanner.GoTokens}}
    lex.scan.Init(bytes.NewReader(data))
    lex.next() // get the first token
    defer func() {
        // NOTE: this is not an example of ideal error handling.
        if x := recover(); x != nil {
            err = fmt.Errorf("error at %s: %v", lex.scan.Position, x)
        }
    }()
    read(lex, reflect.ValueOf(out).Elem())
    return nil
}

//!-Unmarshal

//!+lexer
type lexer struct {
    scan  scanner.Scanner
    token rune // the current token
```

```
}

func (lex *lexer) next()        { lex.token = lex.scan.Scan() }
func (lex *lexer) text() string { return lex.scan.TokenText() }

func (lex *lexer) consume(want rune) {
    if lex.token != want { // NOTE: Not an example of good error handling.
        panic(fmt.Sprintf("got %q, want %q", lex.text(), want))
    }
    lex.next()
}

//!-lexer

// The read function is a decoder for a small subset of well-formed
// S-expressions.  For brevity of our example, it takes many dubious
// shortcuts.
//
// The parser assumes
// - that the S-expression input is well-formed; it does no error checking.
// - that the S-expression input corresponds to the type of the variable.
// - that all numbers in the input are non-negative decimal integers.
// - that all keys in ((key value) ...) struct syntax are unquoted symbols.
// - that the input does not contain dotted lists such as (1 2 . 3).
// - that the input does not contain Lisp reader macros such 'x and #'x.
//
// The reflection logic assumes
// - that v is always a variable of the appropriate type for the
//   S-expression value.  For example, v must not be a boolean,
//   interface, channel, or function, and if v is an array, the input
//   must have the correct number of elements.
// - that v in the top-level call to read has the zero value of its
//   type and doesn't need clearing.
// - that if v is a numeric variable, it is a signed integer.

//!+read
func read(lex *lexer, v reflect.Value) {
    switch lex.token {
    case scanner.Ident:
        // The only valid identifiers are
        // "nil" and struct field names.
        if lex.text() == "nil" {
            v.Set(reflect.Zero(v.Type()))
            lex.next()
            return
        }
    case scanner.String:
        s, _ := strconv.Unquote(lex.text()) // NOTE: ignoring errors
        v.SetString(s)
        lex.next()
        return
    case scanner.Int:
        i, _ := strconv.Atoi(lex.text()) // NOTE: ignoring errors
        v.SetInt(int64(i))
```

```
        lex.next()
        return
    case '(':
        lex.next()
        readList(lex, v)
        lex.next() // consume ')'
        return
    }
    panic(fmt.Sprintf("unexpected token %q", lex.text()))
}

//!-read

//!+readlist
func readList(lex *lexer, v reflect.Value) {
    switch v.Kind() {
    case reflect.Array: // (item ...)
        for i := 0; !endList(lex); i++ {
            read(lex, v.Index(i))
        }

    case reflect.Slice: // (item ...)
        for !endList(lex) {
            item := reflect.New(v.Type().Elem()).Elem()
            read(lex, item)
            v.Set(reflect.Append(v, item))
        }

    case reflect.Struct: // ((name value) ...)
        for !endList(lex) {
            lex.consume('(')
            if lex.token != scanner.Ident {
                panic(fmt.Sprintf("got token %q, want field name", lex.text()))
            }
            name := lex.text()
            lex.next()
            read(lex, v.FieldByName(name))
            lex.consume(')')
        }

    case reflect.Map: // ((key value) ...)
        v.Set(reflect.MakeMap(v.Type()))
        for !endList(lex) {
            lex.consume('(')
            key := reflect.New(v.Type().Key()).Elem()
            read(lex, key)
            value := reflect.New(v.Type().Elem()).Elem()
            read(lex, value)
            v.SetMapIndex(key, value)
            lex.consume(')')
        }

    default:
        panic(fmt.Sprintf("cannot decode list into %v", v.Type()))
```

```
    }
}

func endList(lex *lexer) bool {
    switch lex.token {
    case scanner.EOF:
        panic("end of file")
    case ')':
        return true
    }
    return false
}

//!-readlist
```

encode.go

```
package sexpr

import (
    "bytes"
    "fmt"
    "reflect"
)

//!+Marshal
// Marshal encodes a Go value in S-expression form.
func Marshal(v interface{}) ([]byte, error) {
    var buf bytes.Buffer
    if err := encode(&buf, reflect.ValueOf(v)); err != nil {
        return nil, err
    }
    return buf.Bytes(), nil
}

//!-Marshal

// encode writes to buf an S-expression representation of v.
//!+encode
func encode(buf *bytes.Buffer, v reflect.Value) error {
    switch v.Kind() {
    case reflect.Invalid:
        buf.WriteString("nil")

    case reflect.Int, reflect.Int8, reflect.Int16,
        reflect.Int32, reflect.Int64:
        fmt.Fprintf(buf, "%d", v.Int())

    case reflect.Uint, reflect.Uint8, reflect.Uint16,
        reflect.Uint32, reflect.Uint64, reflect.Uintptr:
        fmt.Fprintf(buf, "%d", v.Uint())

    case reflect.String:
```

```
        fmt.Fprintf(buf, "%q", v.String())

    case reflect.Ptr:
        return encode(buf, v.Elem())

    case reflect.Array, reflect.Slice: // (value ...)
        buf.WriteByte('(')
        for i := 0; i < v.Len(); i++ {
            if i > 0 {
                buf.WriteByte(' ')
            }
            if err := encode(buf, v.Index(i)); err != nil {
                return err
            }
        }
        buf.WriteByte(')')

    case reflect.Struct: // ((name value) ...)
        buf.WriteByte('(')
        for i := 0; i < v.NumField(); i++ {
            if i > 0 {
                buf.WriteByte(' ')
            }
            fmt.Fprintf(buf, "(%s ", v.Type().Field(i).Name)
            if err := encode(buf, v.Field(i)); err != nil {
                return err
            }
            buf.WriteByte(')')
        }
        buf.WriteByte(')')

    case reflect.Map: // ((key value) ...)
        buf.WriteByte('(')
        for i, key := range v.MapKeys() {
            if i > 0 {
                buf.WriteByte(' ')
            }
            buf.WriteByte('(')
            if err := encode(buf, key); err != nil {
                return err
            }
            buf.WriteByte(' ')
            if err := encode(buf, v.MapIndex(key)); err != nil {
                return err
            }
            buf.WriteByte(')')
        }
        buf.WriteByte(')')

    default: // float, complex, bool, chan, func, interface
        return fmt.Errorf("unsupported type: %s", v.Type())
    }
    return nil
}
```

```
//!-encode
```

pretty.go

```
package sexpr

// This file implements the algorithm described in Derek C. Oppen's
// 1979 Stanford technical report, "Pretty Printing".

import (
    "bytes"
    "fmt"
    "reflect"
)

func MarshalIndent(v interface{}) ([]byte, error) {
    p := printer{width: margin}
    if err := pretty(&p, reflect.ValueOf(v)); err != nil {
        return nil, err
    }
    return p.Bytes(), nil
}

const margin = 80

type token struct {
    kind rune // one of "s ()" (string, blank, start, end)
    str  string
    size int
}

type printer struct {
    tokens []*token // FIFO buffer
    stack  []*token // stack of open ' ' and '(' tokens
    rtotal int      // total number of spaces needed to print stream

    bytes.Buffer
    indents []int
    width   int // remaining space
}

func (p *printer) string(str string) {
    tok := &token{kind: 's', str: str, size: len(str)}
    if len(p.stack) == 0 {
        p.print(tok)
    } else {
        p.tokens = append(p.tokens, tok)
        p.rtotal += len(str)
    }
}
func (p *printer) pop() (top *token) {
    last := len(p.stack) - 1
```

```
    top, p.stack = p.stack[last], p.stack[:last]
    return
}
func (p *printer) begin() {
    if len(p.stack) == 0 {
        p.rtotal = 1
    }
    t := &token{kind: '(', size: -p.rtotal}
    p.tokens = append(p.tokens, t)
    p.stack = append(p.stack, t) // push
    p.string("(")
}
func (p *printer) end() {
    p.string(")")
    p.tokens = append(p.tokens, &token{kind: ')'})
    x := p.pop()
    x.size += p.rtotal
    if x.kind == ' ' {
        p.pop().size += p.rtotal
    }
    if len(p.stack) == 0 {
        for _, tok := range p.tokens {
            p.print(tok)
        }
        p.tokens = nil
    }
}
func (p *printer) space() {
    last := len(p.stack) - 1
    x := p.stack[last]
    if x.kind == ' ' {
        x.size += p.rtotal
        p.stack = p.stack[:last] // pop
    }
    t := &token{kind: ' ', size: -p.rtotal}
    p.tokens = append(p.tokens, t)
    p.stack = append(p.stack, t)
    p.rtotal++
}
func (p *printer) print(t *token) {
    switch t.kind {
    case 's':
        p.WriteString(t.str)
        p.width -= len(t.str)
    case '(':
        p.indents = append(p.indents, p.width)
    case ')':
        p.indents = p.indents[:len(p.indents)-1] // pop
    case ' ':
        if t.size > p.width {
            p.width = p.indents[len(p.indents)-1] - 1
            fmt.Fprintf(&p.Buffer, "\n%*s", margin-p.width, "")
        } else {
            p.WriteByte(' ')
```

```
            p.width--
        }
    }
}
func (p *printer) stringf(format string, args ...interface{}) {
    p.string(fmt.Sprintf(format, args...))
}

func pretty(p *printer, v reflect.Value) error {
    switch v.Kind() {
    case reflect.Invalid:
        p.string("nil")

    case reflect.Int, reflect.Int8, reflect.Int16,
        reflect.Int32, reflect.Int64:
        p.stringf("%d", v.Int())

    case reflect.Uint, reflect.Uint8, reflect.Uint16,
        reflect.Uint32, reflect.Uint64, reflect.Uintptr:
        p.stringf("%d", v.Uint())

    case reflect.String:
        p.stringf("%q", v.String())

    case reflect.Array, reflect.Slice: // (value ...)
        p.begin()
        for i := 0; i < v.Len(); i++ {
            if i > 0 {
                p.space()
            }
            if err := pretty(p, v.Index(i)); err != nil {
                return err
            }
        }
        p.end()

    case reflect.Struct: // ((name value ...)
        p.begin()
        for i := 0; i < v.NumField(); i++ {
            if i > 0 {
                p.space()
            }
            p.begin()
            p.string(v.Type().Field(i).Name)
            p.space()
            if err := pretty(p, v.Field(i)); err != nil {
                return err
            }
            p.end()
        }
        p.end()

    case reflect.Map: // ((key value ...)
        p.begin()
```

```
        for i, key := range v.MapKeys() {
            if i > 0 {
                p.space()
            }
            p.begin()
            if err := pretty(p, key); err != nil {
                return err
            }
            p.space()
            if err := pretty(p, v.MapIndex(key)); err != nil {
                return err
            }
            p.end()
        }
        p.end()

    case reflect.Ptr:
        return pretty(p, v.Elem())

    default: // float, complex, bool, chan, func, interface
        return fmt.Errorf("unsupported type: %s", v.Type())
    }
    return nil
}
```

6.访问结构体字段标签

用结构体字段标签来修改Go结构值的JSON编码方式

下面的Unpack函数做了三件事情,首先,调用req.ParseForm()来解析请求,在这之后req.Form就有了所的请求参数,这个方法对HTTP GET 和 POST请求都适用.接着Unpack函数构造了一个从每个有效字段到对应字段变量的映射,在字段有标签时有效字段名与实际字段名可能会有差别,reflect.Type的Field方会返回一个reflect.StructField类型,这个类型提供了每个字段的名称,类型以及一个可选的标签,它的Ta字段类型为reflect.StructTag,底层类型为字符串,提供了一个Get方法用于解析和提供对于一个特定健子串.

```
// Unpack populates the fields of the struct pointed to by ptr
// from the HTTP request parameters in req.
func Unpack(req *http.Request, ptr interface{}) error {
    if err := req.ParseForm(); err != nil {
        return err
    }

    // Build map of fields keyed by effective name.
    fields := make(map[string]reflect.Value)
    v := reflect.ValueOf(ptr).Elem() // the struct variable
    for i := 0; i < v.NumField(); i++ {
        fieldInfo := v.Type().Field(i) // a reflect.StructField
        tag := fieldInfo.Tag           // a reflect.StructTag
        name := tag.Get("http")
        if name == "" {
            name = strings.ToLower(fieldInfo.Name)
        }
```

```
        fields[name] = v.Field(i)
    }

    // Update struct field for each parameter in the request.
    for name, values := range req.Form {
        f := fields[name]
        if !f.IsValid() {
            continue // ignore unrecognized HTTP parameters
        }
        for _, value := range values {
            if f.Kind() == reflect.Slice {
                elem := reflect.New(f.Type().Elem()).Elem()
                if err := populate(elem, value); err != nil {
                    return fmt.Errorf("%s: %v", name, err)
                }
                f.Set(reflect.Append(f, elem))
            } else {
                if err := populate(f, value); err != nil {
                    return fmt.Errorf("%s: %v", name, err)
                }
            }
        }
    }
    return nil
}

//!-Unpack

//!+populate
func populate(v reflect.Value, value string) error {
    switch v.Kind() {
    case reflect.String:
        v.SetString(value)

    case reflect.Int:
        i, err := strconv.ParseInt(value, 10, 64)
        if err != nil {
            return err
        }
        v.SetInt(i)

    case reflect.Bool:
        b, err := strconv.ParseBool(value)
        if err != nil {
            return err
        }
        v.SetBool(b)

    default:
        return fmt.Errorf("unsupported kind %s", v.Type())
    }
    return nil
}
```

```
//!-populate
```

7.显示类型的方法

reflect.Type显示任意一个值的类型并枚举它的方法

reflect,Type和reflect.Value都有一个叫做Method的方法,每个t.Method(i)从reflect.TYpe调用都会回一个reflect.Method类型的实例,这个结构类型描述了这个方法的名称和类型,而每个v.Method(i)从reflect.Value调用都会返回一个reflect.Value,代表一个方法值,既一个已绑定接受者的方法,使用reflect.Value.Call方法可以调用Func类型的Value

```
//!+print
// Print prints the method set of the value x.
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"))
    }
}
```

8.反射注意事项

1.基于反射的代码都是很脆弱的

2.类型其实也算是某种形式的文档,而反射的相关操作则无法做静态类型检查,所以大量使用反射的代码很难理解的

3.基于反射的函数会比特定类型优化的函数慢一两个数量级

9.类型断言

- 类型断言(一)

```
package main

import "fmt"

type people struct {
    Name string
}

func main(){
    var i interface{} = &people{"墨持"}

    //类型断言
    if v,ok := i.(*people);ok{
        fmt.Print(v.Name)
```

原文链接：反射和类型断言

```
    }else{
        fmt.Println("类型匹配失败")
    }

}
C:\Users\vSphere\go\src\day04>go run 05.go
墨持
```

- 类型断言(二)

```
package main

import (
    "fmt"
)

type people struct {
    Name string
}

func main(){
    var i interface{} = people{"乘志"}

    switch i.(type) {
    case people:
        fmt.Println("people")
    case *people:
        fmt.Println("*people")
    default:
        fmt.Println("类型匹配失败")

    }
}
C:\Users\vSphere\go\src\day04>go run 05.go
people
```