链滴

Golang 入门笔记 -09- 反射

作者：zyk
原文链接：https://ld246.com/article/1603176483714
来源网站：链滴
许可协议：署名-相同方式共享 4.0 国际 (CC BY-SA 4.0)

反射是用程序检查其所拥有的结构，尤其是类型的一种能力。反射可以在运行时（不必在编译时）检类型和变量，例如大小、变量、方法和动态调用这些方法。

方法和类型的反射

reflect 包提供了反射功能，它定义两个重要类型：Type 和 Value，分别表示动态类型和值。

有两个常用的方法：

- reflect.TypeOf：返回对象的具体类型。
- reflect.ValueOf：返回对象的值。

反射是先检查一个接口的值，再将变量转换成空接口类型，我们看下这两个函数的定义就能明白了：

```
func TypeOf(i interface{}) Type
func ValueOf(i interface{}) Value
```

reflect.TypeOf

函数 reflect.Typeof() 可以接收任意 interface{} 类型数据，并返回其动态类型。

```
package main

import (
    "fmt"
    "reflect"
)

func main() {
    t := reflect.TypeOf(3)  // a reflect.Type
```

```go
    fmt.Println(t.String()) // "int"
    fmt.Println(t)          // "int"
}
```

由于 reflect.TypeOf 返回的是一个动态类型的接口值，因此它返回的总是具体类型。下面的代码打印是 *os.File，而不是 io.Writer：

```go
package main

import (
    "fmt"
    "io"
    "os"
    "reflect"
)

func main() {
    var w io.Writer = os.Stdout
    fmt.Println(reflect.TypeOf(w)) // "*os.File"
}
```

可以通过 reflect.Type 的 Name() 方法获取**类型名称**，通过 reflect.Type 的 Kind() 方法获取**底层类型**
我们来看一个例子：

```go
package main

import (
    "fmt"
    "reflect"
)

type Enum int // 自定义类型 Enum

func main() {
    var x Enum = 2
    v := reflect.TypeOf(x)

    fmt.Println(v.Name()) // Enum
    fmt.Println(v.Kind()) // int
}
```

reflect.ValueOf

函数 reflect.ValueOf() 可以接收任意 interface{} 类型数据，并返回其值。

我们来看一个例子：

```go
package main

import (
    "fmt"
    "reflect"
)
```

```
type Enum int // 自定义类型 Enum

func main() {
    var x Enum = 2
    v := reflect.ValueOf(x)

    fmt.Printf("%v\n", v)                  // 2
    fmt.Printf("%v", v.Interface().(Enum)) // 2
}
```

通过反射修改值

反射并不能修改所有变量的值，我们来看一个例子：

```
package main

import "reflect"

func main() {
    var x int = 2
    v := reflect.ValueOf(x)

    v.SetInt(5)
}
```

当运行上述代码时，出现了如下的错误：

出现这个错误的原因是：v 是**不可设置**的。我们通过 v := reflect.ValueOf(x) 传递的仅仅是变量 x 的本，并不能更改原始的 x。

我们可以利用 CanSet() 函数判断变量是否可设置：

若 CanSet() 返回 false，表明变量无法设置；true 表可设置。

```
package main

import (
```

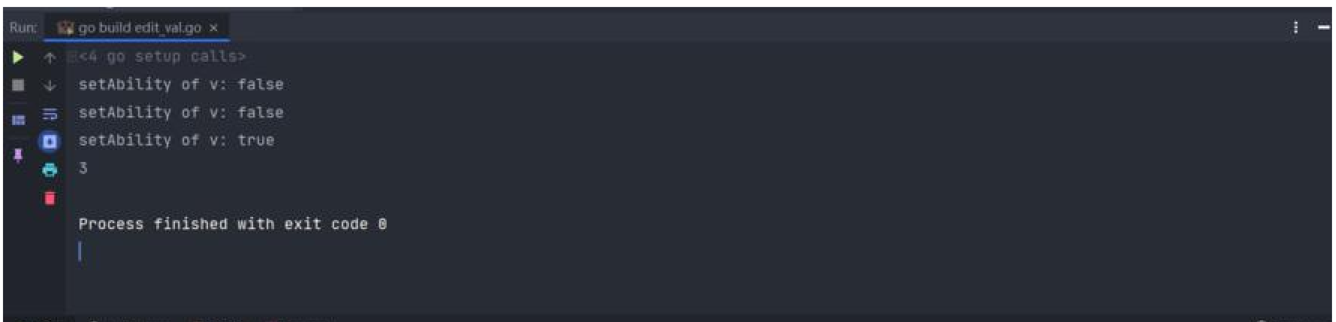
```
    "fmt"
    "reflect"
)

func main() {
    var x int = 2
    v := reflect.ValueOf(x)

    b := v.CanSet()
    fmt.Println(b) // false
}
```

要让 v 可设置，我们可以使用 Elem() 方法，相当于间接使用指针：

v := reflect.ValueOf(x) 只是传递了 x 的拷贝，修改 v 并无法修改原始的 x；若要使修改 v 也能作用到 x 上，需要传递 x 的引用：v := reflect.ValueOf(&x)。

```
package main

import (
    "fmt"
    "reflect"
)

func main() {
    var x = 2
    v := reflect.ValueOf(x)
    fmt.Printf("setAbility of v: %v\n", v.CanSet())

    v = reflect.ValueOf(&x)
    fmt.Printf("setAbility of v: %v\n", v.CanSet())

    v = v.Elem()
    fmt.Printf("setAbility of v: %v\n", v.CanSet())

    v.SetInt(3)
    fmt.Println(v)
}
```

上述代码运行结果为：



反射获取结构体信息

reflect.Type 的 Field() 方法返回 StructField 结构，这个结构用来描述结构体成员的信息：

```
type StructField struct {
    Name string          // 字段名
    PkgPath string       // 字段路径
    Type      Type       // 字段反射类型对象
    Tag       StructTag  // 字段的结构体标签
    Offset    uintptr    // 字段在结构体中的相对偏移
    Index     []int      // Type.FieldByIndex 中的返回的索引值
    Anonymous bool       // 是否为匿名字段
}
```

reflect.Type 中的常用方法如下：

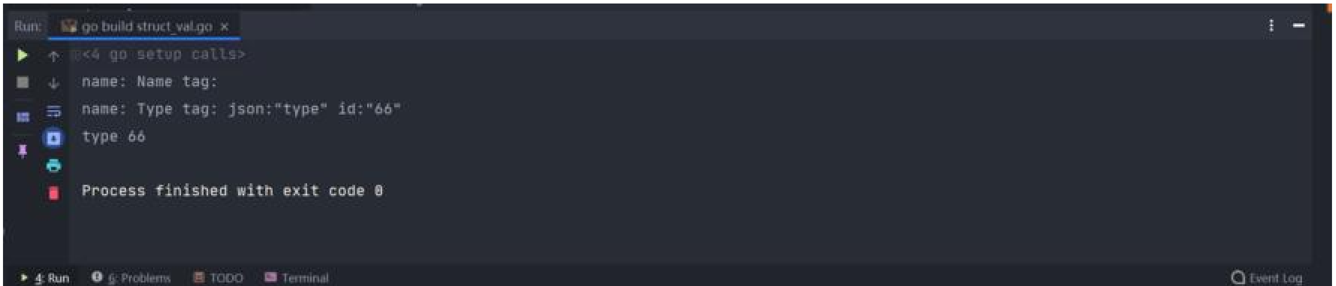
方法	说明
Field(i int) StructField	根据索引，返回索引对应结构体字段的信息。当值不是结构体或索引超界时发生宕机
NumField() int	返回结构体成员字段数量。当类不是结构体或索引超界时发生宕机
FieldByName(name string) (StructField, bool)	据给定字符串返回字符串对应的结构体字段的信息。没有找到时 bool 返回 false，当类型不是结构体索引超界时发生宕机
FieldByIndex(index []int) StructField	多层成员问时，根据 []int 提供的每个结构体的字段索引，返回字段的信息。没有找到时返回零值。当类型不结构体或索引超界时 发生宕机
FieldByNameFunc(match func(string) bool) (StructField,bool)	据匹配函数匹配需要的字段。当值不是结构体或索引超界时发生宕机

我们可以通过实例化一个结构体，然后再利用 reflect.Type 的 FieldByName() 方法查找结构体中指字段：

```
package main

import (
    "fmt"
    "reflect"
)

type Cat struct {
    Name string
    Type int `json:"type" id:"66"`
}

func main() {
    cat := Cat{Name: "Kim", Type: 1} // 创建结构体实例
    typeOfCat := reflect.TypeOf(cat) // 获取结构体实例的反射类型对象

    // 遍历结构体所有成员
    for i := 0; i < typeOfCat.NumField(); i++ {
        // 获取字段类型
        fieldType := typeOfCat.Field(i)
        // 输出成员名称和 tag
        fmt.Printf("name: %v tag: %v\n", fieldType.Name, fieldType.Tag)
    }
```

```
    // 通过字段名, 找到字段类型信息
    if catType, ok := typeOfCat.FieldByName("Type"); ok {
        // 根据名称获取对应 tag
        fmt.Println(catType.Tag.Get("json"), catType.Tag.Get("id"))
    }
}
```

上述代码运行结果如下:



我们也可以通过反射来修改结构体的成员变量，但前提是这些成员变量必须是可导出的（首字母大写，来看一个例子:

```
package main

import (
    "reflect"
)

type Cat struct {
    name string
    Type int `json:"type" id:"66"`
}

func main() {
    cat := Cat{name: "Kim", Type: 1} // 创建结构体实例
    valOfCat := reflect.ValueOf(&cat).Elem() // 通过 Elem() 获取 &cat 的指针实例

    valOfCat.Field(0).SetString("Mi") // 修改 name 字段的值
}
```

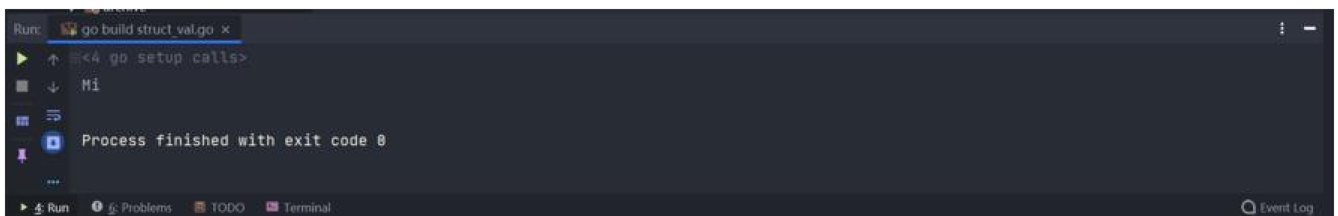
上述代码运行后报错了:

因为 name 字段名是小写字母开头，无法导出。我们将其改为 Name ：

```
package main
```

```
import (
    "fmt"
    "reflect"
)

type Cat struct {
    Name string
    Type int `json:"type" id:"66"`
}

func main() {
    cat := Cat{Name: "Kim", Type: 1} // 创建结构体实例
    valOfCat := reflect.ValueOf(&cat).Elem() // 通过 Elem() 获取 &cat 的指针实例

    valOfCat.Field(0).SetString("Mi") // 修改 Name 字段的值
    fmt.Println(valOfCat.Field(0))
}
```

再次运行，就能修改成功了：



注意：虽然反射在某些场合下很好用，但反射较损耗性能，因此在性能需求较高和高并发的场景下，尽量避免使用反射。