Golang 反射

主讲教师：（大地）
合作网站：www.itying.com （IT 营）
我的专栏：https://www.itying.com/category-79-b0.html

一、反射的引子

有时我们需要写一个函数，这个函数有能力统一处理各种值类型，而这些类型可能无法共享同一个接口，也可能布局未知，也有可能这个类型在我们设计函数时还不存在，这个时候我们就可以用到反射。

1、空接口可以存储任意类型的变量，那我们如何知道这个空接口保存数据的类型是什么？值是什么呢？

1、可以使用类型断言

2、可以使用反射实现，也就是在程序运行时动态的获取一个变量的类型信息和值信息。

2、把结构体序列化成 json 字符串，自定义结构体 Tag 标签的时候就用到了反射

```
package main

import (

    "encoding/json"

    "fmt"

)

type Student struct {
```

```
    ID      int     `json:"id"`
    Gender string `json:"gender"`
    Name    string `json:"name"`
    Sno     string `json:"sno"`
}
func main() {
    var s1 = Student{
        ID:     1,
        Gender: "男",
        Name:   "李四",
        Sno:    "s0001",
    }
    var s, _ = json.Marshal(s1)
    jsonStr := string(s)
    fmt.Println(jsonStr)
}
```

3、后期我们会给大家讲 ORM 框架，这个 ORM 框架就用到了反射技术

ORM:对象关系映射（Object Relational Mapping，简称 ORM）是通过使用描述对象和数据库之间映射的元数据，将面向对象语言程序中的对象自动持久化到关系数据库中。

二、 反射的基本介绍

反射是指在程序运行期间对程序本身进行访问和修改的能力。**正常情况**程序在编译时，变量被转换为内存地址，变量名不会被编译器写入到可执行部分。在运行程序时，程序无法获取自身的信息。**支持反射的语言**可以在程序编译期将变量的反射信息，如字段名称、类型信息、结构体信息等整合到可执行文件中，并给程序提供接口访问反射信息，这样就可以在程序运行期获取类型的反射信息，并且有能力修改它们。

Golang 中反射可以实现以下功能：

1、反射可以在程序运行期间动态的获取变量的各种信息，比如变量的类型 类别

2、如果是结构体，通过反射还可以获取结构体本身的信息，比如结构体的字段、结构体的方法、结构体的 tag。

3、通过反射，可以修改变量的值，可以调用关联的方法

Go 语言中的变量是分为两部分的:

- **类型信息**：预先定义好的元信息。
- **值信息**：程序运行过程中可动态变化的。

在 GoLang 的反射机制中，任何接口值都由是一个**具体类型**和**具体类型的值**两部分组成的。

在 GoLang 中，反射的相关功能由内置的 reflect 包提供，任意接口值在反射中都可以理解为由 reflect.Type 和 reflect.Value 两部分组成，并且 reflect 包提供了 **reflect.TypeOf** 和 **reflect.ValueOf** 两个重要函数来获取任意对象的 Value 和 Type。

三、 reflect.TypeOf()获取任意值的类型对象

在 Go 语言中，使用 reflect.TypeOf()函数可以接受任意 interface{}参数，可以获得任意值的类型对象（reflect.Type），程序通过类型对象可以访问任意值的类型信息。

```
package main

import (

	"fmt"

	"reflect"

)

func reflectType(x interface{}) {

	v := reflect.TypeOf(x)

	fmt.Printf("type:%v\n", v)

}

func main() {

	var a float32 = 12.5

	reflectType(a) // type:float32
```

```
	var b int64 = 100

	reflectType(b) // type:int64

}
```

type Name 和 type Kind

在反射中关于类型还划分为两种：类型（Type）和种类（Kind）。因为在 Go 语言中我们可以使用 type 关键字构造很多自定义类型，而**种类（Kind）就是指底层的类型**，但在反射中，当需要区分指针、结构体等大品种的类型时，**就会用到种类（Kind）**。 举个例子，我们定义了两个指针类型和两个结构体类型，通过反射查看它们的类型和种类。

Go 语言的反射中像数组、切片、Map、指针等类型的变量，它们的.Name()都是返回空。

```
package main

import (

	"fmt"

	"reflect"

)

func reflectType(x interface{}) {

	t := reflect.TypeOf(x)

	fmt.Printf("TypeOf:%v	Name:%v	Kind:%v\n", t, t.Name(), t.Kind())

}

type myInt int64

type Person struct {

	Name string

	Age	int

}

type Animal struct {

	Name string

}

func main() {
```

```
    var a *float32 //  指针
    var b myInt       //  自定义类型
    var c rune        //  类型别名
    reflectType(a) // type: kind:ptr
    reflectType(b) // type:myInt kind:int64
    reflectType(c) // type:int32 kind:int32

    var d = Person{
        Name: "itying",
        Age:    18,
    }
    var e = Animal{Name: "小花"}
    reflectType(d) // type:Person kind:struct
    reflectType(e) // type:Animal kind:struct

    var f = []int{1, 2, 3, 4, 5}
    reflectType(f) //TypeOf:[]int      Name:     Kind:slice
}
```

在 reflect 包中定义的 Kind 类型如下：

```
type Kind uint
const (
    Invalid Kind = iota    //  非法类型
    Bool                          //  布尔型
    Int                           //  有符号整型
```

```
    Int8                // 有符号 8 位整型
    Int16               // 有符号 16 位整型
    Int32               // 有符号 32 位整型
    Int64               // 有符号 64 位整型
    Uint                // 无符号整型
    Uint8               // 无符号 8 位整型
    Uint16              // 无符号 16 位整型
    Uint32              // 无符号 32 位整型
    Uint64              // 无符号 64 位整型
    Uintptr             // 指针
    Float32             // 单精度浮点数
    Float64             // 双精度浮点数
    Complex64           // 64 位复数类型
    Complex128          // 128 位复数类型
    Array               // 数组
    Chan                // 通道
    Func                // 函数
    Interface           // 接口
    Map                 // 映射
    Ptr                 // 指针
    Slice               // 切片
    String              // 字符串
    Struct              // 结构体
    UnsafePointer       // 底层指针
)
```

四、 reflect.ValueOf()

reflect.ValueOf()返回的是 reflect.Value 类型，其中包含了原始值的值信息。reflect.Value 与原始值之间可以互相转换。

reflect.Value 类型提供的获取原始值的方法如下：

方法	说明
Interface() interface {}	将值以 interface{} 类型返回，可以通过类型断言转换为指定类型
Int() int64	将值以 int 类型返回，所有有符号整型均可以此方式返回
Uint() uint64	将值以 uint 类型返回，所有无符号整型均可以此方式返回
Float() float64	将值以双精度（float64）类型返回，所有浮点数（float32、float64）均可以此方式返回
Bool() bool	将值以 bool 类型返回
Bytes() []bytes	将值以字节数组 []bytes 类型返回
String() string	将值以字符串类型返回
...	

1、通过反射获取原始值演示 1

```
package main
import (
	"fmt"
	"reflect"
)

func reflectValue(x interface{}) {
	v := reflect.ValueOf(x)
	var c = v.Int() + 6 //获取反射的原始值
	fmt.Println(c)
}
func main() {
	var a int64 = 100
	reflectValue(a)
}
```

2、通过反射获取原始值演示 2

```
package main

import (
	"fmt"
	"reflect"
)

func reflectValue(x interface{}) {
	v := reflect.ValueOf(x)
	k := v.Kind()
	switch k {
	case reflect.Int64:
		// v.Int()从反射中获取整型的原始值
		fmt.Printf("type is int64, value is %d\n", v.Int())
	case reflect.Float32:
		// v.Float()从反射中获取浮点型的原始值
		fmt.Printf("type is float32, value is %f\n", v.Float())
	case reflect.Float64:
		// v.Float()从反射中获取浮点型的原始值
		fmt.Printf("type is float64, value is %f\n", v.Float())
	}
}
func main() {
	var a float32 = 3.14
	var b int64 = 100
	reflectValue(a) // type is float32, value is 3.140000
	reflectValue(b) // type is int64, value is 100
```

```
    // 将 int 类型的原始值转换为 reflect.Value 类型
    c := reflect.ValueOf(10)
    fmt.Printf("type c :%T\n", c) // type c :reflect.Value
}
```

4、通过反射设置变量的值

- func (v Value) SetBool(x bool)
- func (v Value) SetInt(x int64)
- func (v Value) SetUint(x uint64)
- func (v Value) SetFloat(x float64)
- func (v Value) SetComplex(x complex128)
- func (v Value) SetBytes(x []byte)
- func (v Value) SetString(x string)
- func (v Value) SetPointer(x unsafe.Pointer)
- func (v Value) SetCap(n int)
- func (v Value) SetLen(n int)
- func (v Value) SetMapIndex(key, val Value)
- func (v Value) Set(x Value)

想要在函数中通过反射修改变量的值，需要注意函数参数传递的是值拷贝，必须传递变量地址才能修改变量值。而反射中使用专有的 Elem()方法来获取指针对应的值。

```
package main
import (
    "fmt"
    "reflect"
)
func reflectSetValue1(x interface{}) {
    v := reflect.ValueOf(x)
    if v.Kind() == reflect.Int64 {
        v.SetInt(200) //修改的是副本，reflect 包会引发 panic
    }
```

```
}

func reflectSetValue2(x interface{}) {

	v := reflect.ValueOf(x)

	// 反射中使用 Elem()方法获取指针对应的值

	if v.Elem().Kind() == reflect.Int64 {

		v.Elem().SetInt(200)

	}

}

func main() {

	var a int64 = 100

	// reflectSetValue1(a) //panic: reflect: reflect.Value.SetInt using unaddressable value

	reflectSetValue2(&a)

	fmt.Println(a)

}
```

五、结构体反射

1、与结构体相关的方法

任意值通过 reflect.TypeOf()获得反射对象信息后，如果它的类型是结构体，可以通过反射值对象（reflect.Type）的 NumField()和 Field()方法获得结构体成员的详细信息。

reflect.Type 中与获取结构体成员相关的的方法如下表所示。

方法	说明
Field(i int) StructField	根据索引，返回索引对应的结构体字段的信息。
NumField() int	返回结构体成员字段数量。
FieldByName(name string) (StructField, bool)	根据给定字符串返回字符串对应的结构体字段的信息。
FieldByIndex(index []int) StructField	多层成员访问时，根据 []int 提供的每个结构体的字段索引，返回字段的信息。

FieldByNameFunc(match func(string) bool) (StructField,bool)	根据传入的匹配函数匹配需要的字段。
NumMethod() int	返回该类型的方法集中方法的数目
Method(int) Method	返回该类型方法集中的第 i 个方法
MethodByName(string)(Method, bool)	根据方法名返回该类型方法集中的方法

2、StructField 类型

StructField 类型用来描述结构体中的一个字段的信息。StructField 的定义如下：

```
type StructField struct {
    // 参见 http://golang.org/ref/spec#Uniqueness_of_identifiers
    Name      string   // Name 是字段的名字
    PkgPath   string   //PkgPath 是非导出字段的包路径，对导出字段该字段为""
    Type      Type      // 字段的类型
    Tag       StructTag // 字段的标签
    Offset    uintptr   // 字段在结构体中的字节偏移量
    Index     []int     // 用于 Type.FieldByIndex 时的索引切片
    Anonymous bool      // 是否匿名字段
}
```

3、结构体反射示例

当我们使用反射得到一个结构体数据之后可以通过索引依次获取其字段信息，也可以通过字段名去获取指定的字段信息。

1、获取结构体属性，获取执行结构体方法

```
package main

import (
    "fmt"
    "reflect"
)

//student 结构体
type Student struct {
    Name  string `json:"name"`
    Age   int    `json:"age"`
    Score int    `json:"score"`
}

func (s Student) GetInfo() string {
    var str = fmt.Sprintf("姓名:%v 年龄:%v 成绩:%v", s.Name, s.Age, s.Score)
```

```
	fmt.Println(str)
	return str
}

func (s *Student) SetInfo(name string, age int, score int) {
	s.Name = name
	s.Age = age
	s.Score = score
}

func (s *Student) Print() {
	fmt.Println("打印方法...")
}

//打印字段
func PrintStructField(s interface{}) {

	t := reflect.TypeOf(s)
	// v := reflect.ValueOf(s)

	kind := t.Kind()
	if t.Kind() != reflect.Struct && t.Elem().Kind() != reflect.Struct {
		fmt.Println("传入的不是结构体")
		return
	}
	//1、通过类型变量里面的 Field 可以获取结构体的字段
	field0 := t.Field(0)
	fmt.Println(field0.Name)
	fmt.Println(field0.Type)
	fmt.Println(field0.Tag.Get("json"))

	//2、通过类型变量里面的 FieldByName 可以获取结构体的字段

	field1, _ := t.FieldByName("Age")
	fmt.Println(field1.Name)
	fmt.Println(field1.Type)
	fmt.Println(field1.Tag.Get("json"))

	//3、获取到该结构体有几个字段
	num := t.NumField()
	fmt.Println("字段数量:", num)
}
```

```
//方法
func PrintStructFn(s interface{}) {

    t := reflect.TypeOf(s)
    v := reflect.ValueOf(s)

    if t.Kind() != reflect.Struct && t.Elem().Kind() != reflect.Struct {
        fmt.Println("传入的不是结构体")
        return
    }
    //1、通过类型变量里面的 Method 可以获取结构体的方法
    var tMethod = t.Method(0) //注意
    fmt.Println(tMethod.Name)
    fmt.Println(tMethod.Type)
    //2、通过类型变量获取这个结构体有多少个方法
    fmt.Println(t.NumMethod())

    //3、执行方法 （注意需要使用值变量，并且要注意参数）
    // v.Method(0).Call(nil)
    v.MethodByName("Print").Call(nil)

    //4、执行方法传入参数 （注意需要使用值变量，并且要注意参数）
    var params []reflect.Value //声明了 []reflect.Value
    params = append(params, reflect.ValueOf("张三"))
    params = append(params, reflect.ValueOf(22))
    params = append(params, reflect.ValueOf(100))
    v.MethodByName("SetInfo").Call(params) //传入的参数是 []reflect.Value, 返回[]reflect.Va
lue

    // 5、执行方法获取方法的值
    info := v.MethodByName("GetInfo").Call(nil)
    fmt.Println(info)
}
func main() {
    stu1 := Student{
        Name:  "小明",
        Age:   15,
        Score: 98,
    }
    // PrintStructField(stu1)
    PrintStructFn(&stu1)
}
```

2、修改结构体方法

```
package main

import (
    "fmt"
    "reflect"
)

//student 结构体
type Student struct {
    Name   string `json:"name"`
    Age    int    `json:"age"`
    Score  int    `json:"score"`
}

func (s Student) GetInfo() string {
    var str = fmt.Sprintf("姓名:%v 年龄:%v 成绩:%v", s.Name, s.Age, s.Score)
    return str
}

//反射修改结构体属性
func reflectChangeStruct(s interface{}) {
    t := reflect.TypeOf(s)
    v := reflect.ValueOf(s)

    if t.Elem().Kind() != reflect.Struct {      t.kind() != reflect.Ptr
        fmt.Println("传入的不是结构体指针类型")
        return
    }

    name := v.Elem().FieldByName("Name")
    name.SetString("李四") // 设置值

    age := v.Elem().FieldByName("Age")
    age.SetInt(20) // 设置值

}
func main() {
    stu1 := Student{
        Name:  "小明",
        Age:    15,
        Score: 98,
    }
```

```
    // PrintStructField(stu1)
    reflectChangeStruct(&stu1)

    fmt.Println(stu1.GetInfo())
}
```

六、不要乱用反射

反射是一个强大并富有表现力的工具，能让我们写出更灵活的代码。但是反射不应该被滥用，原因有以下两个。

1. 基于反射的代码是极其脆弱的，反射中的类型错误会在真正运行的时候才会引发 panic，那很可能是在代码写完的很长时间之后。
2. 大量使用反射的代码通常难以理解。