



吴德宝AllenWu

2018年02月03日 阅读 10113

# Golang的反射reflect深入理解和示例

[TOC]

## Golang的反射reflect深入理解和示例

【记录于2018年2月】

### 编程语言中反射的概念

在计算机科学领域，反射是指一类应用，它们能够自描述和自控制。也就是说，这类应用通过采用某种机制来实现对自己行为的描述（self-representation）和监测（examination），并能根据自身行为的状态和结果，调整或修改应用所描述行为的状态和相关的语义。

每种语言的反射模型都不同，并且有些语言根本不支持反射。Golang语言实现了反射，反射机制就是在运行时动态的调用对象的方法和属性，官方自带的reflect包就是反射相关的，只要包含这个包就可以使用。

多插一句，Golang的gRPC也是通过反射实现的。

### interface 和 反射

在讲反射之前，先来看看Golang关于类型设计的一些原则

- 变量包括（type, value）两部分
  - 理解这一点就知道为什么nil != nil了
- type 包括 static type和concrete type. 简单来说 static type是你在编码是看见的类型(如int, string)，concrete type是runtime系统看见的类型



接下来要讲的反射，就是建立在类型之上的，Golang的指定类型的变量的类型是静态的（也就是指定int、string这些的变量，它的type是static type），在创建变量的时候就已经确定，反射主要与Golang的interface类型相关（它的type是concrete type），只有interface类型才有反射一说。

在Golang的实现中，每个interface变量都有一个对应pair，pair中记录了实际变量的值和类型：

```
(value, type)
```

value是实际变量值，type是实际变量的类型。一个interface{}类型的变量包含了2个指针，一个指针指向值的类型【对应concrete type】，另外一个指针指向实际的值【对应value】。

例如，创建类型为\*os.File的变量，然后将其赋给一个接口变量r：

```
tty, err := os.OpenFile("/dev/tty", os.O_RDWR, 0)

var r io.Reader
r = tty
```

接口变量r的pair中将记录如下信息：(tty, \*os.File)，这个pair在接口变量的连续赋值过程中是不变的，将接口变量r赋给另一个接口变量w：

```
var w io.Writer
w = r.(io.Writer)
```

接口变量w的pair与r的pair相同，都是(tty, \*os.File)，即使w是空接口类型，pair也是不变的。

interface及其pair的存在，是Golang中实现反射的前提，理解了pair，就更容易理解反射。反射就是用来检测存储在接口变量内部(值value；类型concrete type) pair对的一种机制。

## Golang的反射reflect

### reflect的基本功能TypeOf和ValueOf

既然反射就是用来检测存储在接口变量内部(值value；类型concrete type) pair对的一种机制。那么在Golang的reflect反射包中有怎样的方式可以让我们直接获取到变量内部的信息呢？它提供



```
// ValueOf returns a new Value initialized to the concrete value
// stored in the interface i. ValueOf(nil) returns the zero
func ValueOf(i interface{}) Value {...}
```

翻译一下：ValueOf用来获取输入参数接口中的数据的值，如果接口为空则返回0

```
// TypeOf returns the reflection Type that represents the dynamic type of i.
// If i is a nil interface value, TypeOf returns nil.
func TypeOf(i interface{}) Type {...}
```

翻译一下：TypeOf用来动态获取输入参数接口中的值的类型，如果接口为空则返回nil

reflect.TypeOf()是获取pair中的type，reflect.ValueOf()获取pair中的value，示例如下：

```
package main

import (
    "fmt"
    "reflect"
)

func main() {
    var num float64 = 1.2345

    fmt.Println("type: ", reflect.TypeOf(num))
    fmt.Println("value: ", reflect.ValueOf(num))
}
```

运行结果：

```
type: float64
value: 1.2345
```

## 说明

1. reflect.TypeOf：直接给到了我们想要的type类型，如float64、int、各种pointer、struct等等真实的类型
2. reflect.ValueOf：直接给到了我们想要的具体的值，如1.2345这个具体数值，或者类似{"Allen.Wu" 25} 这样的结构体struct的值



## 从reflect.Value中获取接口interface的信息

当执行reflect.ValueOf(interface)之后，就得到了一个类型为“reflect.Value”变量，可以通过它本身的Interface()方法获得接口变量的真实内容，然后可以通过类型判断进行转换，转换为原有真实类型。不过，我们可能是已知原有类型，也有可能是未知原有类型，因此，下面分两种情况进行说明。

### 已知原有类型【进行“强制转换”】

已知类型后转换为其对应的类型的做法如下，直接通过Interface方法然后强制转换，如下：

```
realValue := value.Interface().(已知的类型)
```

示例如下：

```
package main

import (
    "fmt"
    "reflect"
)

func main() {
    var num float64 = 1.2345

    pointer := reflect.ValueOf(&num)
    value := reflect.ValueOf(num)

    // 可以理解为“强制转换”，但是需要注意的时候，转换的时候，如果转换的类型不完全符合，则直接panic
    // Golang 对类型要求非常严格，类型一定要完全符合
    // 如下两个，一个是*float64，一个是float64，如果弄混，则会panic
    convertPointer := pointer.Interface().(*float64)
    convertValue := value.Interface().(float64)

    fmt.Println(convertPointer)
    fmt.Println(convertValue)
}
```

运行结果：

0xc42000e238

1.2345



2. 转换的时候，要区分是指针还是指
3. 也就是说反射可以将“反射类型对象”再重新转换为“接口类型变量”

## 未知原有类型【遍历探测其Filed】

很多情况下，我们可能并不知道其具体类型，那么这个时候，该如何做呢？需要进行遍历探测其Filed来得知，示例如下：

```
package main

import (
    "fmt"
    "reflect"
)

type User struct {
    Id    int
    Name string
    Age   int
}

func (u User) ReflectCallFunc() {
    fmt.Println("Allen.Wu ReflectCallFunc")
}

func main() {

    user := User{1, "Allen.Wu", 25}

    DoFiledAndMethod(user)

}

// 通过接口来获取任意参数，然后一一揭晓
func DoFiledAndMethod(input interface{}) {

    getType := reflect.TypeOf(input)
    fmt.Println("get Type is :", getType.Name())

    getValue := reflect.ValueOf(input)
    fmt.Println("get all Fields is:", getValue)

    // 获取方法字段
```



```
for i := 0; i < getType.NumField(); i++ {
    field := getType.Field(i)
    value := getValue.Field(i).Interface()
    fmt.Printf("%s: %v = %v\n", field.Name, field.Type, value)
}

// 获取方法
// 1. 先获取interface的reflect.Type, 然后通过.NumMethod进行遍历
for i := 0; i < getType.NumMethod(); i++ {
    m := getType.Method(i)
    fmt.Printf("%s: %v\n", m.Name, m.Type)
}
}
```

运行结果:

```
get Type is : User
get all Fields is: {1 Allen.Wu 25}
Id: int = 1
Name: string = Allen.Wu
Age: int = 25
ReflectCallFunc: func(main.User)
```

## 说明

通过运行结果可以得知获取未知类型的interface的具体变量及其类型的步骤为：

1. 先获取interface的reflect.Type，然后通过NumField进行遍历
2. 再通过reflect.Type的Field获取其Field
3. 最后通过Field的Interface()得到对应的value

通过运行结果可以得知获取未知类型的interface的所属方法（函数）的步骤为：

1. 先获取interface的reflect.Type，然后通过NumMethod进行遍历
2. 再分别通过reflect.Type的Method获取对应的真实的方法（函数）
3. 最后对结果取其Name和Type得知具体的方法名
4. 也就是说反射可以将“反射类型对象”再重新转换为“接口类型变量”
5. struct 或者 struct 的嵌套都是一样的判断处理方式

## 通过reflect.Value设置实际变量的值



示例如下：

```
package main

import (
    "fmt"
    "reflect"
)

func main() {

    var num float64 = 1.2345
    fmt.Println("old value of pointer:", num)

    // 通过reflect.ValueOf获取num中的reflect.Value，注意，参数必须是指针才能修改其值
    pointer := reflect.ValueOf(&num)
    newValue := pointer.Elem()

    fmt.Println("type of pointer:", newValue.Type())
    fmt.Println("settability of pointer:", newValue.CanSet())

    // 重新赋值
    newValue.SetFloat(77)
    fmt.Println("new value of pointer:", num)

    ////////////////
    // 如果reflect.ValueOf的参数不是指针，会如何？
    pointer = reflect.ValueOf(num)
    //newValue = pointer.Elem() // 如果非指针，这里直接panic，"panic: reflect: call of reflect.Va
}
```

运行结果：

```
old value of pointer: 1.2345
type of pointer: float64
settability of pointer: true
new value of pointer: 77
```

## 说明

1. 需要传入的参数是\* float64这个指针，然后可以通过pointer.Elem()去获取所指向的Value，**注意一定要是指针。**
2. 如果传入的参数不是指针，而是变量，那么



3. `newValue.CantSet()`表示是否可以重新设置其值，如果输出的是true则可修改，否则不能修改，修改完之后再进行打印发现真的已经修改了。
4. `reflect.Value.Elem()` 表示获取原始值对应的反射对象，只有原始对象才能修改，当前反射对象是不能修改的
5. 也就是说如果要修改反射类型对象，其值必须是“addressable”【对应的要传入的是指针，同时要通过Elem方法获取原始值对应的反射对象】
6. `struct` 或者 `struct` 的嵌套都是一样的判断处理方式

## 通过reflect.ValueOf来进行方法的调用

这算是一个高级用法了，前面我们只说到对类型、变量的几种反射的用法，包括如何获取其值、其类型、如果重新设置新值。但是在工程应用中，另外一个常用并且属于高级的用法，就是通过reflect来进行方法【函数】的调用。比如我们要做框架工程的时候，需要可以随意扩展方法，或者说用户可以自定义方法，那么我们通过什么手段来扩展让用户能够自定义呢？关键点在于用户的自定义方法是未知的，因此我们可以通过reflect来搞定

示例如下：

```
package main

import (
    "fmt"
    "reflect"
)

type User struct {
    Id    int
    Name  string
    Age   int
}

func (u User) ReflectCallFuncHasArgs(name string, age int) {
    fmt.Println("ReflectCallFuncHasArgs name: ", name, ", age:", age, "and orignal User.Name:",
}

func (u User) ReflectCallFuncNoArgs() {
    fmt.Println("ReflectCallFuncNoArgs")
}

// 如何通过反射来进行方法的调用？
// 本来可以用u.ReflectCallFuncXXX直接调用的，但是如果要通过反射，那么首先要将方法注册，也就是MethodE
```



```
// 1. 要通过反射来调用起对应的方法，必须要先通过reflect.ValueOf(interface)来获取到reflect.Value,
getValue := reflect.ValueOf(user)

// 一定要指定参数为正确的方法名
// 2. 先看看带有参数的调用方法
methodValue := getValue.MethodByName("ReflectCallFuncHasArgs")
args := []reflect.Value{reflect.ValueOf("wudebao"), reflect.ValueOf(30)}
methodValue.Call(args)

// 一定要指定参数为正确的方法名
// 3. 再看看无参数的调用方法
methodValue = getValue.MethodByName("ReflectCallFuncNoArgs")
args = make([]reflect.Value, 0)
methodValue.Call(args)
}
```

运行结果:

```
ReflectCallFuncHasArgs name: wudebao , age: 30 and origal User.Name: Allen.Wu
ReflectCallFuncNoArgs
```

## 说明

1. 要通过反射来调用起对应的方法，必须要先通过reflect.ValueOf(interface)来获取到reflect.Value，得到“反射类型对象”后才能做下一步处理
2. reflect.Value.MethodByName这.MethodByName，需要指定准确真实的方法名字，如果错误将直接panic，MethodByName返回一个函数值对应的reflect.Value方法的名字。
3. []reflect.Value，这个是最终需要调用的方法的参数，可以没有或者一个或者多个，根据实际参数来定。
4. reflect.Value的 Call 这个方法，这个方法将最终调用真实的方法，参数务必保持一致，如果reflect.Value.Kind不是一个方法，那么将直接panic。
5. 本来可以用u.ReflectCallFuncXXX直接调用的，但是如果要通过反射，那么首先要将方法注册，也就是MethodByName，然后通过反射调用methodValue.Call

## Golang的反射reflect性能



```
Field field = clazz.getField("hello");
field.get(obj1);
field.get(obj2);
```

这个取得的反射对象类型是 `java.lang.reflect.Field`。它是可以复用的。只要传入不同的obj，就可以取得这个obj上对应的 field。

但是Golang的反射不是这样设计的:

```
type_ := reflect.TypeOf(obj)
field, _ := type_.FieldByName("hello")
```

这里取出来的 field 对象是 `reflect.StructField` 类型，但是它没有办法用来取得对应对象上的值。如果要取值，得用另外一套对object，而不是type的反射

```
type_ := reflect.ValueOf(obj)
fieldValue := type_.FieldByName("hello")
```

这里取出来的 fieldValue 类型是 `reflect.Value`，它是一个具体的值，而不是一个可复用的反射对象了，每次反射都需要malloc这个reflect.Value结构体，并且还涉及到GC。

## 小结

Golang reflect慢主要有两个原因

1. 涉及到内存分配以及后续的GC；
2. reflect实现里面有大量的枚举，也就是for循环，比如类型之类的。

## 总结

上述详细说明了Golang的反射reflect的各种功能和用法，都附带有相应的示例，相信能够在工程应用中进行相应实践，总结一下就是：

- 反射可以大大提高程序的灵活性，使得interface{}有更大的发挥余地



- 反射可以将“接口类型变量”转换为“反射类型对象”
  - 反射使用 `TypeOf` 和 `ValueOf` 函数从接口中获取目标对象信息
- 反射可以将“反射类型对象”转换为“接口类型变量”
  - `reflect.value.Interface().(已知的类型)`
  - 遍历`reflect.Type`的Field获取其Field
- 反射可以修改反射类型对象，但是其值必须是“addressable”
  - 想要利用反射修改对象状态，前提是 `interface.data` 是 `settable`,即 `pointer-interface`
- 通过反射可以“动态”调用方法
- 因为Golang本身不支持模板，因此在以往需要使用模板的场景下往往就需要使用反射(reflect)来实现

## 参考链接

- [The Go Blog](#) : 其实看官方说明就足够了！
- [官方reflect-Kind](#)
- [Go语言的反射三定律](#)
- [Go基础学习五之接口interface、反射reflection](#)
- [提高 golang 的反射性能](#)

关注下面的标签，发现更多相似文章

API

后端

设计

Go

### 安装掘金浏览器插件

打开新标签页发现好内容，掘金、GitHub、Dribbble、ProductHunt 等站点内容轻松获取。快来安装掘金浏览器插件获取高质量内容吧！



输入评论...



panpanla python运维  
赞哦

1月前



回复



lukex DBA  
非常好，赞

2月前



回复



低轨道粒子炮  
学习到了！！！！

3月前



回复



happy123  
，明天继续撸，。牛

11月前



回复

相关推荐

专栏 · TIGERB · 7小时前 · Go / PHP

电商设计手册之基础商品信息



44



7

专栏 · Denny-coder · 1天前 · API

我们为什么需要API管理系统？



4



1

专栏 · EatherToo · 17小时前 · 后端

Spring入门学习手册 2：怎么用注解来DI/IOC



3



专栏 · 热爱coding的稻草 · 19小时前 · Go

Go嵌套并发实现EDM，附坑点分析#1



7



热 · 专栏 · 胡七筒 · 5天前 · JavaScript



aohe · 2天前 · Go / 后端

Golang 需要避免踩的 50 个坑 ( 二 )

👍 26    💬 2

幸运儿 · 5天前 · 设计 / 设计师 / 前端

如何和设计师成为好朋友？

👍 33    💬 26

热 · 专栏 · 胡七筒 · 7天前 · JavaScript

程序猿生存指南-48 何为爱情

👍 34    💬 42

专栏 · CoderHG · 2天前 · 设计

实现一键式自动化操作 ( 快速创建 Python 与 Shell 文件 )

👍 5    💬

