李文周的博客

JPG程序员/全栈开发 -- 专注互联网技术，相信代码改变世界。Go语言学习QQ群：645090316 公众号：李文周

Go语言基础之反射

2017年6月25日 | Golang |7860 阅读

本文介绍了Go语言反射的意义和基本使用。

变量的内在机制

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。

反射介绍

反射是指在程序运行期对程序本身进行访问和修改的能力。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。

Go程序在运行期使用reflect包访问程序的反射信息。

在上一篇博客中我们介绍了空接口。 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的数据是什么呢？ 反射就是在运行时动态的获取一个变量的类型信息和值信息。

reflect包

在Go语言的反射机制中，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(我们在上一篇接口的博客中有介绍相关概念)。 在Go语言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` 两个函数来获取任意对象的Value和Type。

TypeOf

在Go语言中，使用 `reflect.TypeOf()` 函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 = 3.14
    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
package main

import (
    "fmt"
    "reflect"
)

type myInt int64

func reflectType(x interface{}) {
    t := reflect.TypeOf(x)
    fmt.Printf("type:%v kind:%v\n", t.Name(), t.Kind())
}

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

    type person struct {
        name string
        age  int
    }
    type book struct{ title string }
    var d = person{
        name: "沙河小王子",
        age:  18,
    }
    var e = book{title: "《跟小王子学Go语言》"}
    reflectType(d) // type:person kind:struct
    reflectType(e) // type:book kind:struct
}
```

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` 。

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
)
```

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

方法	说明
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

通过反射获取值

```
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，然后通过int64()强制类型转换
        fmt.Printf("type is int64, value is %d\n", int64(v.Int()))
    case reflect.Float32:
        // v.Float()从反射中获取浮点型的原始值，然后通过float32()强制类型转换
        fmt.Printf("type is float32, value is %f\n", float32(v.Float()))
    case reflect.Float64:
        // v.Float()从反射中获取浮点型的原始值，然后通过float64()强制类型转换
        fmt.Printf("type is float64, value is %f\n", float64(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
}
```

通过反射设置变量的值

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。

```go
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)
}
```

isNil()和isValid()

isNil()

```go
func (v Value) IsNil() bool
```

`IsNil()` 报告v持有的值是否为nil。v持有的值的分类必须是通道、函数、接口、映射、指针、切片之一；否则IsNil函数会导致panic。

isValid()

```go
func (v Value) IsValid() bool
```

`IsValid()` 返回v是否持有一个值。如果v是Value零值会返回假，此时v除了IsValid、String、Kind之外的方法都会导致panic。

举个例子

`IsNil()` 常被用于判断指针是否为空； `IsValid()` 常被用于判定返回值是否有效。

```
func main() {
    // *int类型空指针
    var a *int
    fmt.Println("var a *int IsNil:", reflect.ValueOf(a).IsNil())
    // nil值
    fmt.Println("nil IsValid:", reflect.ValueOf(nil).IsValid())
    // 实例化一个匿名结构体
    b := struct{}{}
    // 尝试从结构体中查找"abc"字段
    fmt.Println("不存在的结构体成员:", reflect.ValueOf(b).FieldByName("abc").IsValid())
    // 尝试从结构体中查找"abc"方法
    fmt.Println("不存在的结构体方法:", reflect.ValueOf(b).MethodByName("abc").IsValid())
    // map
    c := map[string]int{}
    // 尝试从map中查找一个不存在的键
    fmt.Println("map中不存在的键：", reflect.ValueOf(c).MapIndex(reflect.ValueOf("娜扎")).IsValid())
}
```

结构体反射

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

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 {
    // Name是字段的名字。PkgPath是非导出字段的包路径，对导出字段该字段为""。
    // 参见http://golang.org/ref/spec#Uniqueness_of_identifiers
    Name    string
    PkgPath string
    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
}
```

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。

```
type student struct {
    Name  string `json:"name"`
    Score int    `json:"score"`
}

func main() {
    stu1 := student{
        Name:  "小王子",
        Score: 90,
    }

    t := reflect.TypeOf(stu1)
    fmt.Println(t.Name(), t.Kind()) // student struct
    // 通过for循环遍历结构体的所有字段信息
    for i := 0; i < t.NumField(); i++ {
        field := t.Field(i)
        fmt.Printf("name:%s index:%d type:%v json tag:%v\n", field.Name, field.Index, field.Type,
    }

    // 通过字段名获取指定结构体字段信息
    if scoreField, ok := t.FieldByName("Score"); ok {
        fmt.Printf("name:%s index:%d type:%v json tag:%v\n", scoreField.Name, scoreField.Index, sc
    }
}
```

接下来编写一个函数 `printMethod(s interface{})` 来遍历打印s包含的方法。

```
// 给student添加两个方法 Study和Sleep(注意首字母大写)
func (s student) Study() string {
    msg := "好好学习，天天向上。"
    fmt.Println(msg)
    return msg
}

func (s student) Sleep() string {
    msg := "好好睡觉，快快长大。"
    fmt.Println(msg)
    return msg
}

func printMethod(x interface{}) {
    t := reflect.TypeOf(x)
    v := reflect.ValueOf(x)

    fmt.Println(t.NumMethod())
    for i := 0; i < v.NumMethod(); i++ {
        methodType := v.Method(i).Type()
        fmt.Printf("method name:%s\n", t.Method(i).Name)
        fmt.Printf("method:%s\n", methodType)
        // 通过反射调用方法传递的参数必须是 []reflect.Value 类型
        var args = []reflect.Value{}
        v.Method(i).Call(args)
    }
}
```

反射是把双刃剑

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，原因有以下三个。

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。
3. 反射的性能低下，基于反射实现的代码通常比正常代码运行速度慢一到两个数量级。

练习题

1. 编写代码利用反射实现一个ini文件的解析器程序。