Let's GoLang(一)：反射

标记

Richard
full stack, gopher, rustacean

学了好长一段时间Go了， 最近准备写一系列文章来总结学习的经验，第一篇是反射。

最开始接触到反射是在Java中，当时的理解是：反射可以在运行时来加载编译好的字节码，而不需要进行编译的过程，非常的灵活。在Go中也是如此：如果我们想要使用在编写程序时不存在的信息，例如变量；或者将文件或网络请求中的数据映射到变量中，在这些情况下，就需要使用反射。 Reflection使我们能够在运行时检查类型，检查、修改和创建变量，函数和结构等，非常强大。

在Go中的反射围绕三个概念构建：Type，Kind和Value，在Go中主要使用标准库中的reflect包来实现调用反射的各种功能。

基本用法

首先让我们来看看reflect.Type。可以使用反射来获取变量var的类型，函数调用varType := reflect.TypeOf(var)，这将返回一个reflect.Type类型的变量，该变量包含有关定义传入变量的类型的各种信息的方法。

第一个方法是Name()，它会返回类型的名称，某些类型（如slice或pointer）没有名称，此方法返回空字符串。

下一个方法是Kind()，非常有用。kind是由 slice，map，pointer, struct，inerface，string，array，func，int或其他一些基本类型构成。Kind() 和Name() 之间的差异可能很难理解，但可以理解为:如果定义一个名为Foo的struct，则kind() 返回struct，Name() 返回Foo。

举个简单的例子吧:

```
type User struct {
    Name string
    Age int
}
user := User{
    "jack",
    20,
}

userType := reflect.TypeOf(user)
fmt.Println(userType.Name())
fmt.Println(userType.Kind())
```

可以看到打印的结果分别是：User, struct。

使用反射时需要注意的一件事：反射包中的所有内容都假定用户知道自己在做什么，如果使用不当，许多函数和方法调用都会造成panic。例如，如果在reflect.Type上调用一个方法，该方法与当前类型不同的类型相关联，那么程序将会崩溃。一种不错的解决方案是，始终记得使用Kind() 来判断反射值的类型

如果变量是pointer，map，slice，channel或array，则可以使用 varType.Elem() 找到包含的类型。

如果变量是struct，则可以使用反射来获取结构中的字段数，并获取reflect.StructField结构中包含的每个字段的结构。 reflect.StructField中包含字段上的name，order，type和tag等信息。

还是看上面那个例子：

```
type User struct {
    Name string `json:"name"`
    Age int `json:"age"`
}

user := User{
    "jack",
    20,
}

userType := reflect.TypeOf(user)
firstField := userType.Field(0)
fmt.Println(firstField.Name, firstField.Type, firstField.Tag)
```

结果是：Name string json:"name"

使用反射创建新的实例

除了用来获取变量的类型之外，还可以使用反射来读取，设置或创建值。首先，使用refVal := reflect.ValueOf(val) 为变量创建一个reflect.Value实例。如果要使用反射来修改值，则必须获取指向变量的指针 refPtrVal := reflect.ValueOf(&val)，如果不这样做，则只能使用反射读取值，但不能修改它。

举个简单的例子：

```
type User struct {
    Name string `json:"name"`
    Age int `json:"age"`
}

user := User{
    "jack",
    20,
}
userValue := reflect.ValueOf(&user).Elem()
userValue.FieldByName("Name").SetString("rose")
fmt.Println(user)
```

打印的结果是：{rose 20}

如果要创建新值，可以使用函数调用 newPtrVal：= reflect.New(varType)，传入reflect.Type。这将返回一个可以修改的指针值。使用Elem().Set()如上所述。就像下面这样：

```
type User struct {
    Name string `json:"name"`
    Age int `json:"age"`
}
user := User{
    "jack",
    20,
}
userType := reflect.TypeOf(user)
newUser := reflect.New(userType)
newUser.Elem().FieldByName("Name").SetString("rose")
newUser.Elem().FieldByName("Age").SetInt(10)
fmt.Println(user, newUser)
```

最后，可以通过调用Interface()方法返回到正常变量。因为Go没有泛型，所以变量的原始类型丢失了，所以我们需要将空接口转换为实际类型才能使用它：

```
user := User{
    "jack",
    20,
}
userInterface := reflect.ValueOf(user).Interface()
originalUser, isOk := userInterface.(User)
if isOk {
    fmt.Println(originalUser.Name, originalUser.Age)
}
```

使用反射创建需要make才能创建类型实例

除了创建内置和用户定义类型的实例之外，还可以使用反射来创建通常需要make函数的实例。可以使用reflect.MakeSlice，reflect.MakeMap和reflect.MakeChan函数制作slice，map或channel。在所有情况下，首先需要调用创建一个reflect.Type，然后调用上面提到的这些方法来获取一个可以使用反射操作的reflect.Value。例如：

```
        intSlice := make([]int, 0)
        initMap := make(map[string]int)


        sliceType := reflect.TypeOf(intSlice)
        mapType := reflect.TypeOf(initMap)
```

```
        reflectSlice := reflect.MakeSlice(sliceType, 0, 0)
        reflectMap := reflect.MakeMap(mapType)


        v := 10
        rv := reflect.ValueOf(v)
        reflectSlice = reflect.Append(reflectSlice, rv)
        intSlice2 := reflectSlice.Interface().([]int)
        fmt.Println(intSlice2)

        k := "hello"
        rk := reflect.ValueOf(k)
        reflectMap.SetMapIndex(rk, rv)
        mapStringInt2 := reflectMap.Interface().(map[string]int)
        fmt.Println(mapStringInt2)
```

使用make创建函数

我们不仅可以使用反射创建存储数据的变量，还可以反使用reflect.MakeFunc函数创建新函数。举个栗子，如果我们想要计算一个函数执行的时长，可以像下面这样：

```
func testMakeFunc(count int)  {
        sum := 0
        for i := 0; i< count; i++ {
                sum += 1
        }
        fmt.Println(sum)
}

func main() {
        funcType := reflect.TypeOf(testMakeFunc)
        funcValue := reflect.ValueOf(testMakeFunc)

        newFunc := reflect.MakeFunc(funcType, func(args []reflect.Value) (results []reflect.Value) {
                start := time.Now()
                out := funcValue.Call(args)
                end := time.Now()
                fmt.Println(end.Sub(start))
                return out
        })
        var count int = 1e8
        newFunc.Call([]reflect.Value{reflect.ValueOf(count)})
}
```

不仅如此，我们还能借助reflect在运行时，根据某些数据来动态的生成struct，非常强大：

```
func testMakeStruct(args ...interface{}) interface{} {
        var structList []reflect.StructField
        for index, value := range args {
                argType := reflect.TypeOf(value)
                item := reflect.StructField{
```

```
                Name: fmt.Sprintf("Item%d", index),
                Type: argType,
            }
            structList = append(structList, item)
        }
        structType := reflect.StructOf(structList)
        structValue := reflect.New(structType)
        return structValue.Interface()
}

func main() {
        structValue := reflect.ValueOf(testMakeStruct(1, true, "hello world"))
        fmt.Println(structValue)
}
```

反射不能做什么?

前面我们说到了反射的各种功能，但是反射不是万能的。例如：反射不能创建方法，因此你无法在运行时实现接口。

反射的作用

大多数情况下，如果使用interface {}类型的参数调用函数，则很可能会使用反射来检查或更改参数的值。

反射的最常见用途是从文件或网络中序列化和反序列化数据。可以将JSON或数据库映射成指定数据结构，比如struct，slice，map等, 也可以反过来。让我们看一下如何通过查看实现JSON解组的Go标准库中的代码来完成这项工作。

为了将JSON字符串中的值映射到变量中，我们调用json.Unmarshal函数。它包含两个参数：

1. 第一个参数是JSON，类型是[]byte
2. 第二个参数是变量，即反序列化JSON生成的变量

如果你看过unmarshal的源码，它是这样的：

```
func (d *decodeState) unmarshal(v interface{}) (err error) {
  <skip over some setup>
  rv := reflect.ValueOf(v)
  if rv.Kind() != reflect.Ptr || rv.IsNil() {
    return &InvalidUnmarshalError{reflect.TypeOf(v)}
  }
  d.scan.reset()
  // We decode rv not rv.Elem because the Unmarshaler interface
  // test must be applied at the top level of the value.
  d.value(rv)
  return d.savedError
}
```

我们可以看到，其中使用反射来验证v是否为正确的变量类型，也就是指针。如果是，则将v的反射版本（称为rv）传递给value方法。 在通过一些函数和方法处理之后，使用反射以不同的方式得到rv，具体取决于JSON是表示数组，对象还是文字。例如，在解析JSON对象时，标准库以多种方式使用反射。 我们可以看到，在处理不同类型时，会执行不同的操作：

```
switch v.Kind() {
        case reflect.Map:
        // Map key must either have string kind, have an integer kind,
        // or be an encoding.TextUnmarshaler.
        t := v.Type()
        switch t.Key().Kind() {
                case reflect.String,
                        reflect.Int, reflect.Int8, reflect.Int16, reflect.Int32, reflect.Int64,
                        reflect.Uint, reflect.Uint8, reflect.Uint16, reflect.Uint32, reflect.Uint64, r
eflect.Uintptr:
                default:
                        if !reflect.PtrTo(t.Key()).Implements(textUnmarshalerType) {
                                d.saveError(&UnmarshalTypeError{Value: "object", Type: v.Type(), Offse
t: int64(d.off)})
                                d.off --
                                d.next() // skip over { } in input
                                return
                        }
        }
        if v.IsNil() {
                v.Set(reflect.MakeMap(t))
        }
        case reflect.Struct:
                // ok
        default:
                d.saveError(&UnmarshalTypeError{Value: "object", Type: v.Type(), Offset: int64(d.of
f)})
                d.off --
                d.next() // skip over { } in input
                return
}
```

```
subv = v
destring = f.quoted
for _, i := range f.index {
        if subv.Kind() == reflect.Ptr {
                if subv.IsNil() {
                        subv.Set(reflect.New(subv.Type().Elem()))
                }
                subv = subv.Elem()
        }
        subv = subv.Field(i)
}
```

上面分别是对map和struct的处理，更多的处理可以看decode的源码。
发布于 2019-01-19 09:47
Go 语言
后端技术