

# 反射

## 什么是反射

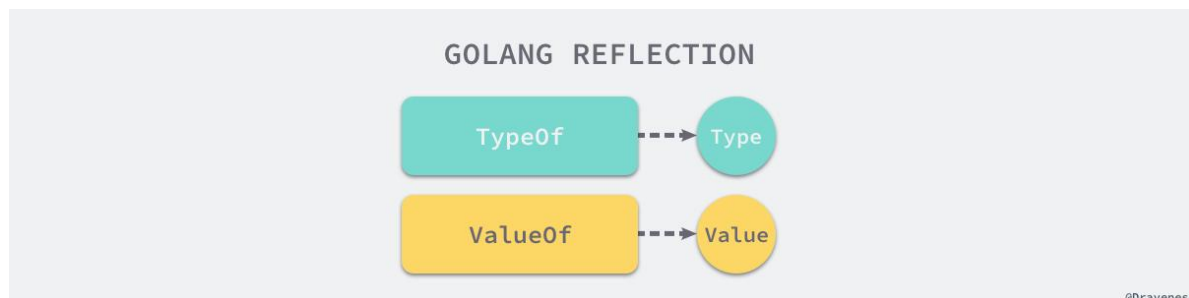
反射其实指的是在程序运行期间对程序本身进行修改和访问的一种能力，它作用在很多动态语言里面。具体一点来说，程序在编写的过程中我们的变量都是有一个类型的，代码编译过后这些变量名就会被去掉了，只有内存地址之类的；但是在某些场景下我们不确定某个变量是什么类型的，比如我们用一个 interface 类型接一个类型，我们不知道里面具体是什么类型，我们只有在程序运行的时候动态访问它，这时候我们就要用到反射。说白了，反射就是能够在程序运行时动态获取数据的类型、属性、字段等等这些信息。具体一点的场景有 json 的序列化和反序列化。

反射是 Go 语言比较重要的一个特性之一，虽然在大多数的应用和服务中并不常见，但是很多框架都依赖 Go 语言的反射机制实现一些动态的功能。作为一门静态语言，Golang 在设计上都非常简洁，所以在语法上其实并没有较强的表达能力，但是 Go 语言为我们提供的 reflect 包提供的动态特性却能够弥补它在语法上的一些劣势。

reflect 实现了运行时的反射能力，能够让 Golang 的程序操作不同类型的对象，我们可以使用包中的函数 TypeOf 从静态类型 interface{} 中获取动态类型信息并通过 ValueOf 获取数据的运行时表示，通过这两个函数和包中的其他工具我们就可以得到更强大的表达能力。

## 概述

在具体介绍反射包的实现原理之前，我们先要对 Go 语言的反射有一些比较简单的理解，首先 reflect 中有两对非常重要的函数和类型，我们在上面已经介绍过其中的两个函数 TypeOf 和 ValueOf，另外两个类型是 Type 和 Value，它们与函数是一一对应的关系：



类型 Type 是 Golang 反射包中定义的一个接口，我们可以使用 TypeOf 函数获取任意值的变量的的类型，我们能从这个接口中看到非常多有趣的方法，MethodByName 可以获取当前类型对应方法的引用、Implements 可以判断当前类型是否实现了某个接口：

(看一下Type和Value)

反射包中的所有方法基本都是围绕着 Type 和 Value 这两个对外暴露的类型设计的，我们通过 TypeOf、ValueOf 方法就可以将一个普通的变量转换成『反射』包中提供的 Type 和 Value，使用反射提供的方法对这些类型进行复杂的操作。

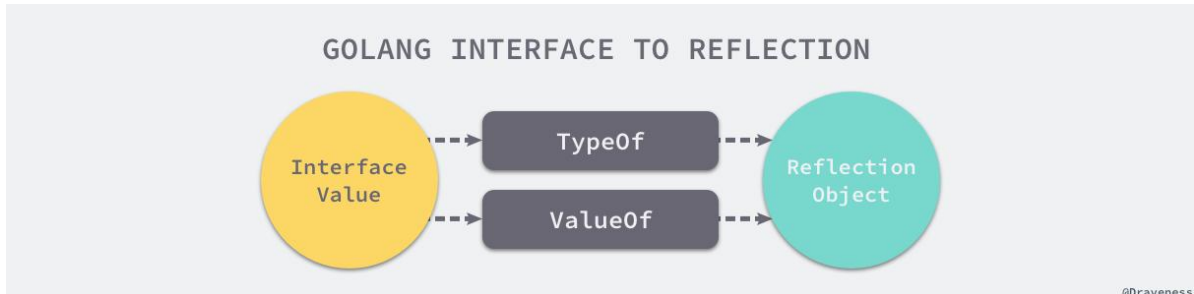
## 反射法则

运行时反射是程序在运行期间检查其自身结构的一种方式，它是 元编程 的一种，但是它带来的灵活性也是一把双刃剑，过量的使用反射会使我们的程序逻辑变得难以理解并且运行缓慢，我们在这一节中就会介绍 Go 语言反射的三大法则，这能够帮助我们更好地理解反射的作用。

1. 从接口值可反射出反射对象；
2. 从反射对象可反射出接口值；
3. 要修改反射对象，其值必须可设置；

## 第一法则

反射的第一条法则就是，我们能够将在 Go 语言中的接口类型变量转换成反射对象，上面提到的 `reflect.TypeOf` 和 `reflect.ValueOf` 就是完成这个转换的两个最重要方法，如果我们认为 Go 语言中的类型和反射类型是两个不同『世界』的话，那么这两个方法就是连接这两个世界的桥梁。



我们通过以下例子简单介绍这两个方法的作用，其中 `TypeOf` 获取了变量的类型而 `ValueOf` 获取了变量的值，如果我们知道了一个变量的类型和值，那么也就意味着我们知道了关于这个变量的全部信息。

(操作)

### TypeOf

```
package main

import (
    "fmt"
    "reflect"
)

// 定义一个函数
func test(re interface{}) {
    // 使用反射获取类型
    rType := reflect.TypeOf(re)
    fmt.Println("rType: ", rType) // int

    // 使用反射获取值
    rVal := reflect.ValueOf(re)
    fmt.Println("rVal: ", rVal) // 10
    // rVal 的值没法直接使用，需要转换
    num := 10 + rVal.Int()
    fmt.Println("num: ", num)
}

func main() {
    // 定义一个基本数据类型
    i := 10
    test(i)
}
```

## 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。

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

```

## 使用反射操作结构体

```

package main

import (
    "fmt"
    "reflect"
)
//定义一个结构体
type Student struct {
    Name string `json:"name"`
    Age int
}
//定义一个函数
func test01(re interface{}) {
    //使用反射获取类型
    rType := reflect.TypeOf(re)
    fmt.Println("rType: ", rType)
    //使用反射获取值
    rVal := reflect.ValueOf(re)
    fmt.Println("rVal: ", rVal)
}

func main() {
    student := Student{"小明", 20}
    test01(student)
}

```

### 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, field.Tag.Get("json"))
}

// 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, scoreField.Type, scoreField.Tag.Get("json"))
}
}

```

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 := "洗洗睡吧。"
    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)
    }
}

```

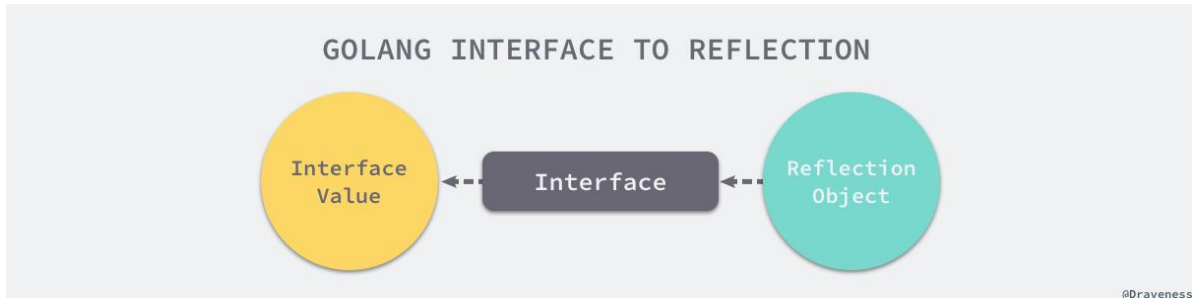
从变量的类型上我们可以获当前类型能够执行的方法 Method 以及当前类型实现的接口等信息；

- 对于结构体，可以获取字段的数量并通过下标和字段名获取字段 StructField；
- 对于哈希表，可以获取哈希表的 Key 类型；
- 对于函数或方法，可以获得入参和返回值的类型；
- ...

总而言之，使用 `TypeOf` 和 `ValueOf` 能够将 Go 语言中的变量转换成反射对象，在这时我们能够获得几乎一切跟当前类型相关数据和操作，然后就可以用这些运行时获取的结构动态的执行一些方法。

## 第二法则

我们既然能够将接口类型的变量转换成反射对象类型，那么也需要一些其他方法将反射对象还原成接口类型的变量，`reflect` 中的 `Interface` 方法就能完成这项工作：



然而调用 `Interface` 方法我们也只能获得 `interface{}` 类型的接口变量，如果想要将其还原成原本的类型还需要经过一次强制的类型转换(类型断言)，如下所示：

```
package main

import (
    "fmt"
    "reflect"
)

//定义一个函数
func test(re interface{}) {
    //使用反射获取值
    rVal := reflect.ValueOf(re)

    //将rVal转化接口
    iV := rVal.Interface()
    //将iV转换为需要的类型
    i := iV.(int)
    num2 := 10 + i
    fmt.Println("num:", num2)
}

func main() {
    //定义一个基本数据类型
    i := 10
    test(i)
}
```

```
package main

import (
    "fmt"
    "reflect"
)

//定义一个结构体
type Student struct {
    Name string `json:"name"`
}
```



```

    Age int
}
//定义一个函数
func test01(re interface{}) {
    //使用反射获取值
    rVal := reflect.ValueOf(re)
    fmt.Println("rVal: ", rVal)

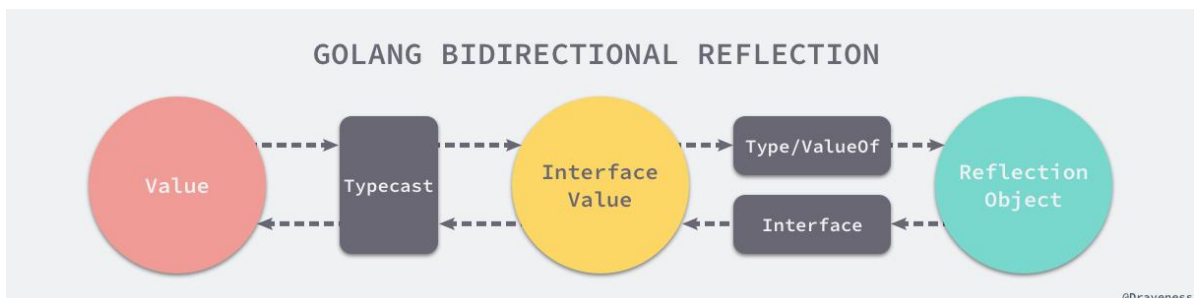
    //这里需要转换
    iV := rVal.Interface()

    student, ok := iV.(Student)
    if !ok {
        fmt.Println("转换失败")
    }
    fmt.Println(student.Name)
}

func main() {
    student := Student{"小明", 20}
    test01(student)
}

```

这个过程就像从接口值到反射对象的镜面过程一样，从接口值到反射对象需要经过从基本类型到接口类型的类型转换和从接口类型到反射对象类型的转换，反过来的话，所有的反射对象也都需要先转换成接口类型，再通过强制类型转换变成原始类型：



当然不是所有的变量都需要类型转换这一过程，如果本身就是 interface{} 类型的，那么它其实并不需要经过类型转换，对于大多数的变量来说，类型转换这一过程很多时候都是隐式发生的，只有在我们需要将反射对象转换回基本类型时才需要做显示的转换操作。

### 第三法则

Go 语言反射的最后一条法则是与值是否可以被更改相关的，如果我们想要更新一个 reflect.Value，那么它持有的值一定是可以被更新的。

```

func main() {
    i := 1
    v := reflect.ValueOf(i)
    v.SetInt(10)
    fmt.Println(i)
}

```

运行上述代码时会导致程序 panic 并报出 reflect: reflect.flag.mustBeAssignable using unaddressable value 错误，仔细想一下其实能够发现出错的原因，Go 语言的 函数调用 都是传值的，所以我们得到的反射对象其实跟最开始的变量没有任何关系，没有任何变量持有复制出来的值，所以直接对它修改会导致崩溃，必须传递变量地址才能修改变量值。

想要修改原有的变量我们只能通过如下所示的方法，首先通过 reflect.ValueOf 获取变量指针，然后通过 Elem 方法获取指针指向的变量并调用 SetInt 方法更新变量的值：

```
func main() {
    i := 1
    v := reflect.ValueOf(&i)
    v.Elem().SetInt(10)
    fmt.Println(i)
}
```

## 修改结构体的值

```
package main

import (
    "fmt"
    "reflect"
)

type Jack struct {
    Name string
}

func main() {
    //修改结构体的值
    jack := &Jack{}
    j := reflect.ValueOf(jack)
    j = j.Elem()
    //获取结构体的字段
    name := j.FieldByName("Name")
    //修改结构体字段的值
    name.SetString("小明")
    fmt.Println("修改后的Name值为:", name.String())
}
```

## 实现原理

我们在上面的部分已经对 Go 语言中反射的三大法则进行了介绍，对于接口值和反射对象互相转换的操作和过程都有了一定的了解，接下来我们就深入研究反射的实现原理，分析 reflect 包提供的方法是如何获取接口值对应的反射类型和值、判断协议的实现以及方法调用的过程。

### 类型和值

Golang 的 interface{} 类型在语言内部都是通过 emptyInterface 这个结体来表示的，其中包含一个 rtype 字段用于表示变量的类型以及一个 word 字段指向内部封装的数据：

```
type emptyInterface struct { typ *rtype word unsafe.Pointer }
```

用于获取变量类型的 `TypeOf` 函数就是将传入的 `i` 变量强制转换成 `emptyInterface` 类型并获取其中存储的类型信息 `rtype`:

```
// 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 any) Type {
    eface := *(*emptyInterface)(unsafe.Pointer(&i))
    return toType(eface.typ)
}

// toType converts from a *rtype to a Type that can be returned
// to the client of package reflect. In gc, the only concern is that
// a nil *rtype must be replaced by a nil Type, but in gccgo this
// function takes care of ensuring that multiple *rtype for the same
// type are coalesced into a single Type.
func toType(t *rtype) Type {
    if t == nil {
        return nil
    }
    return t
}
```

`rtype` 就是一个实现了 `Type` 接口的接口体，我们能在 `reflect` 包中找到如下所示的 `Name` 方法帮助我们获取当前类型的名称等信息：

```
func (t *rtype) String() string {
    s := t.nameOff(t.str).name()
    if t.tflag&tflagExtraStar != 0 {
        return s[1:]
    }
    return s
}
```

`TypeOf` 函数的实现原理其实并不复杂，它只是将一个 `interface{}` 变量转换成了内部的 `emptyInterface` 表示，然后从中获取相应的类型信息。

用于获取接口值 `Value` 的函数 `ValueOf` 实现也非常简单，在该函数中我们先调用了 `escapes` 函数保证当前值逃逸到堆上，然后通过 `unpackEface` 方法从接口中获取 `Value` 结构体：

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

    // TODO: Maybe allow contents of a value to live on the stack.
    // For now we make the contents always escape to the heap. It
    // makes life easier in a few places (see chanrecv/mapassign
    // comment below).
    escapes(i)

    return unpackEface(i)
}
```

```
// unpackEface converts the empty interface i to a Value.
func unpackEface(i any) Value {
    e := (*emptyInterface)(unsafe.Pointer(&i))
    // NOTE: don't read e.word until we know whether it is really a pointer or
    not.
    t := e.typ
    if t == nil {
        return Value{}
    }
    f := flag(t.Kind())
    if ifaceIndir(t) {
        f |= flagIndir
    }
    return Value{t, e.word, f}
}
```

unpackEface 函数会将传入的接口 interface{} 转换成 emptyInterface 结构体然后将其中表示接口值类型、指针以及值的类型包装成 Value 结构体并返回。

TypeOf 和 ValueOf 两个方法的实现其实都非常简单，从一个 Go 语言的基本变量中获取反射对象以及类型的过程中，TypeOf 和 ValueOf 两个方法的执行过程并不是特别的复杂，我们还需要注意基本变量到接口值的转换过程：

```
package main import ( "reflect" ) func main() { i := 20 _ = reflect.TypeOf(i)} $
go build -gcflags="-S -N" main.go...MOVQ $20, ""..autotmp_20+56(SP) // autotmp =
20LEAQ type.int(SB), AX // AX = type.int(SB)MOVQ AX, ""..autotmp_19+280(SP) //
autotmp_19+280(SP) = type.int(SB)LEAQ ""..autotmp_20+56(SP), CX // CX = 20MOVQ
CX, ""..autotmp_19+288(SP) // autotmp_19+288(SP) = 20...
```

我们使用 -S -N 编译指令编译了上述代码，从这段截取的汇编语言中我们可以发现，在函数调用之前其实发生了类型转换，我们将 int 类型的变量转换成了占用 16 字节 autotmp\_19+280(SP) ~ autotmp\_19+288(SP) 的 interface{} 结构体，两个 LEAQ 指令分别获取了类型的指针 type.int(SB) 以及变量 i 所在的地址。

总的来说，在 Go 语言的编译期间我们就完成了类型转换的工作，将变量的类型和值转换成了 interface{} 等待运行期间使用 reflect 包获取其中存储的信息。

## 更新变量

当我们想要更新一个 reflect.Value 时，就需要调用 Set 方法更新反射对象，该方法会调用 mustBeAssignable 和 mustBeExported 分别检查当前反射对象是否可以被设置的和对外暴露的公开字段：

```
func (v Value) Set(x Value) { v.mustBeAssignable() x.mustBeExported() // do not
let unexported x leak var target unsafe.Pointer if v.kind() == Interface { target
= v.ptr } x = x.assignTo("reflect.Set", v.typ, target) if x.flag&flagIndir != 0 {
typedmemmove(v.typ, v.ptr, x.ptr) } else { *(*unsafe.Pointer)(v.ptr) = x.ptr
}}Set
```

Set 方法中会调用 assignTo，该方法会返回一个新的 reflect.Value 反射对象，我们可以将反射对象的指针直接拷贝到被设置的反射变量上：

```
func (v Value) assignTo(context string, dst *rtype, target unsafe.Pointer) Value
{ if v.flag&flagMethod != 0 { v = makeMethodValue(context, v) } switch { case
directlyAssignable(dst, v.typ): fl := v.flag&(flagAddr|flagIndir) | v.flag.ro()
fl |= flag(dst.Kind()) return Value{dst, v.ptr, fl} case implements(dst, v.typ):
if target == nil { target = unsafe_New(dst) } if v.Kind() == Interface &&
v.IsNil() { return Value{dst, nil, flag(Interface)} } x := valueInterface(v,
false) if dst.NumMethod() == 0 { *(*interface{})(target) = x } else {
ifaceE2I(dst, x, target) } return Value{dst, target, flagIndir |
flag(Interface)} } panic(context + ": value of type " + v.typ.String() + " is
not assignable to type " + dst.String())}
```

assignTo 会根据当前和被设置的反射对象类型创建一个新的 Value 结构体，当两个反射对象的类型是可以被直接替换时，就会直接将目标反射对象返回；如果当前反射对象是接口并且目标对象实现了接口，就会将目标对象简单包装成接口值，上述方法返回反射对象的 ptr 最终会覆盖当前反射对象中存储的值。

### 实现协议

reflect 包还为我们提供了 Implements 方法用于判断某些类型是否遵循协议实现了全部的方法，在 Go 语言中想要获取结构体的类型还是比较容易的，但是想要获得接口的类型就需要比较黑魔法的方式：

```
reflect.TypeOf((*)(nil)).Elem()
```

只有通过上述方式才能获得一个接口类型的反射对象，假设我们有以下代码，我们需要判断 CustomError 是否实现了 Go 语言标准库中的 error 协议：

```
type CustomError struct{ func (*CustomError) Error() string { return ""} func
main() { typeOfError := reflect.TypeOf((*error)(nil)).Elem() customErrorPtr :=
reflect.TypeOf(&CustomError{}) customError := reflect.TypeOf(CustomError{})
fmt.Println(customErrorPtr.Implements(typeOfError)) // #=> true
fmt.Println(customError.Implements(typeOfError)) // #=> false}
```

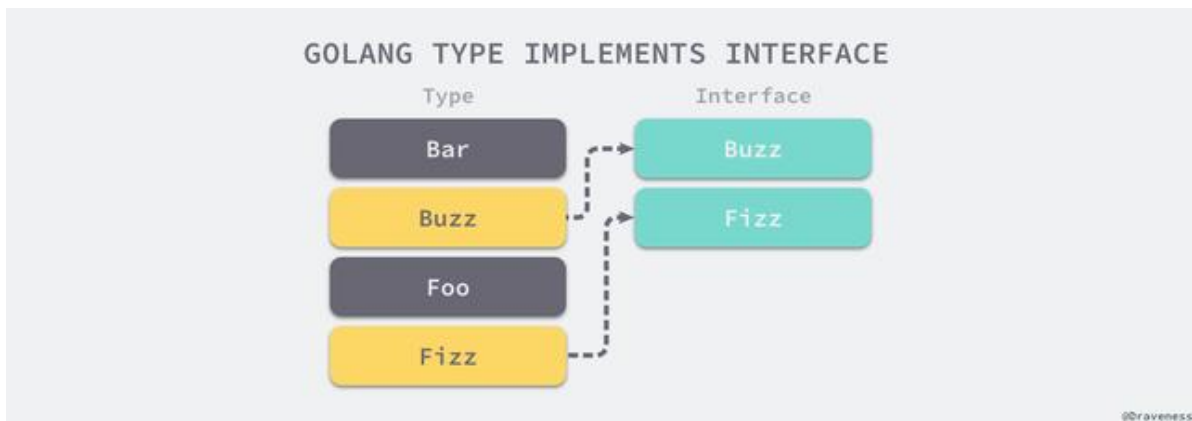
运行上述代码我们会发现 CustomError 类型并没有实现 error 接口，而 \*CustomError 指针类型却实现了接口，这其实也比较好理解，我们在 接口 一节中也介绍过可以使用结构体和指针两种不同的类型实现接口。

```
func (t *rtype) Implements(u Type) bool { if u == nil { panic("reflect: nil type
passed to Type.Implements") } if u.Kind() != Interface { panic("reflect: non-
interface type passed to Type.Implements") } return implements(u.(*rtype), t)}
```

Implements 方法会检查传入的类型是不是接口，如果不是接口或者是空值就会直接 panic 中止当前程序，否则就会调用私有的函数 implements 判断类型之间是否有实现关系：

```
func implements(T, v *rtype) bool { t := (*interfaceType)(unsafe.Pointer(T)) if
len(t.methods) == 0 { return true } // ... v := v.uncommon() i := 0 vmethods :=
v.methods() for j := 0; j < int(v.mcount); j++ { tm := &t.methods[i] tmName :=
t.nameOff(tm.name) vm := vmethods[j] vmName := v.nameOff(vm.name) if
vmName.name() == tmName.name() && v.typeOff(vm.mtyp) == t.typeOff(tm.typ) { if
i++; i >= len(t.methods) { return true } } } return false}
```

如果接口中不包含任何方法，也就意味着这是一个空的 interface{}，任意的类型都可以实现该协议，所以就会直接返回 true。



在其他情况下，由于方法是按照一定顺序排列的，implements 中就会维护两个用于遍历接口和类型方法的索引  $i$  和  $j$ ，所以整个过程的实现复杂度是  $O(n+m)$ ，最多只会进行  $n+m$  次数的比较，不会出现次方级别的复杂度。

### 方法调用

作为一门静态语言，如果我们想要通过 reflect 包利用反射在运行期间执行方法并不是一件容易的事情，下面的代码就使用了反射来执行 `Add(0, 1)` 这一表达式：

```
func Add(a, b int) int { return a + b } func main() { v := reflect.ValueOf(Add)
if v.Kind() != reflect.Func { return } t := v.Type() argv :=
make([]reflect.Value, t.NumIn()) for i := range argv { if t.In(i).Kind() !=
reflect.Int { return } argv[i] = reflect.ValueOf(i) } result := v.Call(argv) if
len(result) != 1 || result[0].Kind() != reflect.Int { return }
fmt.Println(result[0].Int()) // => 1}
```

1. 通过 `reflect.ValueOf` 获取函数 `Add` 对应的反射对象；
2. 根据反射对象 `NumIn` 方法返回的参数个数创建 `argv` 数组；
3. 多次调用 `reflect.Value` 逐一设置 `argv` 数组中的各个参数；
4. 调用反射对象 `Add` 的 `Call` 方法并传入参数列表；
5. 获取返回值数组、验证数组的长度以及类型并打印其中的数据；

使用反射来调用方法非常复杂，原本只需要一行代码就能完成的工作，现在需要 10 多行代码才能完成，但是这也是在静态语言中使用这种动态特性需要付出的成本，理解这个调用过程能够帮助我们深入理解 Go 语言函数和方法调用的原理。

```
func (v value) Call(in []value) []value { v.mustBe(Func) v.mustBeExported()
return v.call("call", in)}
```

`Call` 作为反射包运行时调用方法的入口，通过两个 `MustBe` 方法保证了当前反射对象的类型和可见性，随后调用 `call` 方法完成运行时方法调用的过程，这个过程会被分成以下的几个部分：

1. 检查输入参数的合法性以及类型等信息；
2. 将传入的 `reflect.Value` 参数数组设置到栈上；
3. 通过函数指针和输入参数调用函数；
4. 从栈上获取函数的返回值；

我们将按照上面的顺序依次详细介绍使用 `reflect` 进行函数调用的几个过程。

### 参数检查

参数检查是通过反射调用方法的第一步，在参数检查期间我们会从反射对象中取出当前的函数指针 `unsafe.Pointer`，如果待执行的函数是方法，就会通过 `methodReceiver` 函数获取方法的接受者和函数指针。

```
func (v Value) call(op string, in []Value) []Value { t := (*funcType)
(unsafe.Pointer(v.typ)) var ( fn unsafe.Pointer rcvr Value rcvrtype *rtype ) if
v.flag&flagMethod != 0 { rcvr = v rcvrtype, t, fn = methodReceiver(op, v,
int(v.flag)>>flagMethodShift) } else if v.flag&flagIndir != 0 { fn = *
(*unsafe.Pointer)(v.ptr) } else { fn = v.ptr } n := t.NumIn() if len(in) < n {
panic("reflect: Call with too few input arguments") } if len(in) > n {
panic("reflect: Call with too many input arguments") } for i := 0; i < n; i++ {
if xt, targ := in[i].Type(), t.In(i); !xt.AssignableTo(targ) { panic("reflect: "
+ op + " using " + xt.String() + " as type " + targ.String()) } } nin := len(in)
if nin != t.NumIn() { panic("reflect.Value.Call: wrong argument count") }
```

除此之外，在参数检查的过程中我们还会检查当前传入参数的个数以及所有参数的类型是否能被传入该函数中，任何参数不匹配的问题都会导致当前函数直接 panic 并中止整个程序。

### 准备参数

当我们已经对当前方法的参数验证完成之后，就会进入函数调用的下一个阶段，为函数调用准备参数，在前面的章节 函数调用 中我们已经介绍过 Go 语言的函数调用的惯例，所有的参数都会被依次放置到堆栈上。

```
nout := t.NumOut() frametype, _, retOffset, _, framePool := funcLayout(t,
rcvrtype) var args unsafe.Pointer if nout == 0 { args = framePool.Get().
(unsafe.Pointer) } else { args = unsafe_New(frametype) } off := uintptr(0) if
rcvrtype != nil { storeRcvr(rcvr, args) off = ptrSize } for i, v := range in {
targ := t.In(i).(*rtype) a := uintptr(targ.align) off = (off + a - 1) &^ (a - 1)
n := targ.size if n == 0 { v.assignTo("reflect.Value.Call", targ, nil) continue
} addr := add(args, off, "n > 0") v = v.assignTo("reflect.Value.Call", targ,
addr) if v.flag&flagIndir != 0 { typedmemmove(targ, addr, v.ptr) } else { *
(*unsafe.Pointer)(addr) = v.ptr } off += n }
```

1. 通过 `funcLayout` 函数计算当前函数需要的参数和返回值的堆栈布局，也就是每一个参数和返回值所占的空间大小；
  2. 如果当前函数有返回值，需要为当前函数的参数和返回值分配一片内存空间 `args`；
  3. 如果当前函数是方法，需要向将方法的接受者拷贝到 `args` 这片内存中；
  4. 将所有函数的参数按照顺序依次拷贝到对应 `args` 内存中使用 `funcLayout` 返回的参数计算参数在内存中的位置；通过 `typedmemmove` 或者寻址的放置拷贝参数；
- 准备参数的过程其实就是计算各个参数和返回值占用的内存空间，并将所有的参数都拷贝内存空间对应的位置上。

### 调用函数

准备好调用函数需要的全部参数之后，就会通过以下的表达式开始方法的调用了，我们会向该函数中传入栈类型、函数指针、参数和返回值的内存空间、栈的大小以及返回值的偏移量：

```
call(frametype, fn, args, uint32(frametype.size), uint32(retOffset))
```

这个函数实际上并不存在，它会在编译期间被链接到 `runtime.reflectcall` 这个用汇编实现的函数上，我们在这里并不会展开介绍该函数的具体实现，感兴趣的读者可以自行了解其实现原理。

### 处理返回值



当函数调用结束之后，我们就会开始处理函数的返回值了，如果函数没有任何返回值我们就会直接清空 args 中的全部内容来释放内存空间，不过如果当前函数有返回值就会进入另一个分支：

```
var ret []Value if nout == 0 { typedmemclr(frametype, args) framePool.Put(args)
} else { typedmemclrpartial(frametype, args, 0, retOffset) ret = make([]Value,
nout) off = retOffset for i := 0; i < nout; i++ { tv := t.Out(i) a :=
uintptr(tv.Align()) off = (off + a - 1) &^ (a - 1) if tv.Size() != 0 { fl :=
flagIndir | flag(tv.Kind()) ret[i] = value{tv.common(), add(args, off,
"tv.Size() != 0"), fl} } else { ret[i] = Zero(tv) } off += tv.Size() } } return
ret}
```

1. 将 args 中与输入参数有关的内存空间清空；
2. 创建一个 nout 长度的切片用于保存由反射对象构成的返回值数组；
3. 从函数对象中获取返回值的类型和内存大小，将 args 内存中的数据转换成 reflect.Value 类型的返回值；

由 reflect.Value 构成的 ret 数组最终就会被返回到上层，使用反射进行函数调用的过程也就结束了。

## 总结

我们在这一节中 Go 语言的 reflect 包为我们提供的多种能力，其中包括如何使用反射来动态修改变量、判断类型是否实现了某些协议以及动态调用方法，通过对反射包中方法原理的分析帮助我们理解之前看起来比较怪异、令人困惑的现象。