反射

概念

何为反射？在**运行期**，语言提供可以获取对象的类型、内存结构等信息的能力。就如同照镜子一样，通过反射看到自身的特征。

类型Type

使用TypeOf可以获取类型信息，其签名为`func reflect.TypeOf(i any) Type`。Type是一个接口，定义了很多方法可供调用。

reflect/type.go/Type定义

```
type Type interface {
    Method(int) Method // 字典序索引获取方法
    NumMethod() int // 方法的个数
    MethodByName(string) (Method, bool) //  根据名字获取方法
    Name() string // 获取类型名称
    PkgPath() string // 包路径
    Size() uintptr // 内存占用
    String() string // 类型的字符串表达
    Kind() Kind // *** 类型的种类

    Implements(u Type) bool // 该类型是否实现接口u
    AssignableTo(u Type) bool // 该类型是否能赋值给另一种类型u
    ConvertibleTo(u Type) bool // 该类型是否转换为另一种类型u

    Elem() Type // 解析指针，如果Kind不是Array、Chan、Map、Pointer、Slice则panic

    // 结构体
    Field(i int) StructField // 结构体索引i的字段
    FieldByIndex(index []int) StructField //
    FieldByName(name string) (StructField, bool) //
    FieldByNameFunc(match func(string) bool) (StructField, bool) //
    NumField() int //

    Len() int // 容器的长度，如果不是Array返回panic
    Key() Type // 返回map的key类型，如果不是map返回panic

    // 函数
    In(i int) Type // 返回一个函数类型第i个索引的入参类型，如果不是Func或i超界则panic
    Out(i int) Type // 出参
    NumIn() int
    NumOut() int
}
```

- Elem()，返回Array、Chan、Map、Pointer、Slice类型的元素的类型，因为容器类型是壳，如同一个盒子，如果你关心盒子里面的东西的类型，就需要Elem()。

```go
package main

import (
    "fmt"

    "reflect"
)

func main() {
    // 数组
    a := [3]int{1, 3, 5}
    t := reflect.TypeOf(a)
    fmt.Printf("%v, %v\n", t, t.Kind()) // 类型是[3]int，种类是array
    fmt.Println(t.Len())                // Len只有array可用
    fmt.Println(t.Elem())               // 元素的类型int
    fmt.Println(t.Elem().Kind())        // 元素的种类int

    // Map
    b := map[string]int{"a": 111, "b": 222}
    t = reflect.TypeOf(b)
    fmt.Printf("%v, %v\n", t, t.Kind()) // 类型是map[string]int，种类是map
    fmt.Println(t.Key())                // 只有map可用

    // 函数
    c := func(a, b int) (int, error) { return a + b, nil }
    t = reflect.TypeOf(c)
    fmt.Printf("%v, %v\n", t, t.Kind()) // 类型是func(int,int)(int,error)，种类是func
    for i := 0; i < t.NumIn(); i++ {    // 入参，参数列表
        fmt.Println(i, t.In(i))
    }
    for i := 0; i < t.NumOut(); i++ { // 出参，返回值
        fmt.Println(i, t.Out(i))
    }
}
```

输出如下

```
[3]int, array
3
int
int
array

map[string]int, map
string

func(int, int) (int, error), func
0 int
1 int
0 int
1 error
```

Type和Kind

从中文角度来看，Type和Kind意思接近，这里把Type翻译成类型，把Kind翻译成种类。意思很相近，怎么区分他们呢?

从上例中，可以看出Type更具体，Kind更概括。以数组为例，Type具体指几个元素已经元素类型的数组，Kind指数组这个大的种类，包含所有数组。

那既然有了Type类型，为什么还要Kind种类?

例如某些使用场景中，只是需要使用数组类型就行，使用Kind判断更合适、更宽松。

结构体Type

```go
package main

import (
    "fmt"

    "reflect"
)

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

func (p Person) GetName() string {
    return p.Name
}

func (p Person) getAge() int {
    return p.age
}

func main() {
    // 结构体

    tom := Person{Name: "tom", age: 20}
    t := reflect.TypeOf(tom)
    fmt.Println(t, t.Kind(), t.Size()) // 类型是Person，种类是struct，占据内存字节数
    fmt.Println("~~~~~~~~~~~~~~~~~~~~~~~~~~")
    // 字段
    for i := 0; i < t.NumField(); i++ {
        f := t.Field(i)
        fmt.Println(i, f)
        fmt.Println(
            f.Name,            // 字段名
            f.Index,           // 字段索引
            f.Type,            // 字段类型
            f.Type.Kind(),     // 字段类型的种类
            f.Offset,          // 相当于结构体首地址该字段值的偏移，string占16字节
            f.Anonymous,       // 是否匿名成员，就是没有名字。注意不要和可见性混淆
```

```
            f.IsExported(),     // 是否导出，包外可见否
            f.Tag,              // 本质上就是string类型
            f.Tag.Get("json"), // 获取结构体字段定义后面反引号部分的tag
        )
    }
    // 方法，只包括导出的方法，也不包括receiver是该结构体指针的方法
    for i := 0; i < t.NumMethod(); i++ {
        m := t.Method(i)
        fmt.Println(i, m)
        fmt.Println(
            m.Name,          // 方法名
            m.Index,         // 方法索引
            m.Type,          // 方法类型，函数签名
            m.Type.Kind(),   // 方法类型的种类
            m.IsExported(), // 是否导出。当然未导出的看不到
            m.Func,          // reflect.Value
        )
    }
}
```

指针类型

将上面代码改成结构体的指针，运行报错

```
func main() {
    // 结构体
    tom := &Person{Name: "tom", age: 20}
    t := reflect.TypeOf(tom)
    fmt.Println(t, t.Kind()) // 类型是Person，种类是struct
    fmt.Println("~~~~~~~~~~~~~~~~~~~~~~~~~~")
    // 字段
    fmt.Println(t.NumField()) //reflect: NumField of non-struct type *main.Person
}
```

很明显，指针类型不能调用NumField()方法，这是结构体才能调用的，这时候就要使用Elem()方法来解析指针，相当于对指针类型变量做了*操作获取元素。

```
func main() {
    // 结构体
    tom := &Person{Name: "tom", age: 20}
    t := reflect.TypeOf(tom)
    fmt.Println(t, t.Kind()) // 类型是Person，种类是struct
    fmt.Println("~~~~~~~~~~~~~~~~~~~~~~~~~~")
    // 字段
    // fmt.Println(t.NumField()) //reflect: NumField of non-struct type *main.Person
    fmt.Println(t.Elem().NumField())
}
```

使用了Elem()方法后，下面写法都差不多

```
package main

```

```go
import (
    "fmt"

    "reflect"
)

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

func (p Person) GetName() string {
    return p.Name
}

func (p Person) getAge() int {
    return p.age
}

func main() {
    // 结构体
    tom := &Person{Name: "tom", age: 20}
    t := reflect.TypeOf(tom)
    fmt.Println(t, t.Kind()) // 类型是Person，种类是struct
    fmt.Println("~~~~~~~~~~~~~~~~~~~~~~~~~~")
    // 字段
    // fmt.Println(t.NumField()) //reflect: NumField of non-struct type *main.Person
    for i := 0; i < t.Elem().NumField(); i++ {
        f := t.Elem().Field(i)
        fmt.Println(i, f)
        fmt.Println(
            f.Name,              // 字段名
            f.Index,             // 字段索引
            f.Type,              // 字段类型
            f.Type.Kind(),       // 字段类型的种类
            f.Offset,            // 相当于结构体首地址该字段值的偏移，string占16字节
            f.Anonymous,         // 是否匿名成员，就是没有名字。注意不要和可见性混淆
            f.IsExported(),      // 是否导出，包外可见否
            f.Tag,               // 本质上就是string类型
            f.Tag.Get("json"), // 获取结构体字段定义后面反引号部分的tag
        )
    }

    // 方法，只包括导出的方法，也不包括receiver是该结构体指针的方法
    for i := 0; i < t.Elem().NumMethod(); i++ {
        m := t.Method(i)
        fmt.Println(i, m)
        fmt.Println(
            m.Name,          // 方法名
            m.Index,         // 方法索引
            m.Type,          // 方法类型，函数签名
            m.Type.Kind(),   // 方法类型的种类
            m.IsExported(), // 是否导出。当然未导出的看不到
            m.Func,          // reflect.Value
```

```
        )
    }
    fmt.Println("~~~~~~~~~~~~~~~~~~~~~~~~~~")

    // 方法，使用指针访问，可以访问所有receiver的导出的方法
    for i := 0; i < t.NumMethod(); i++ {
        m := t.Method(i)
        fmt.Println(i, m)
        fmt.Println(
            m.Name,          // 方法名
            m.Index,         // 方法索引
            m.Type,          // 方法类型，函数签名
            m.Type.Kind(),   // 方法类型的种类
            m.IsExported(), // 是否导出。当然未导出的看不到
            m.Func,          // reflect.Value
        )
    }
}
```

注意，方法签名中的receiver的变化。

内存对齐

```
package main

import (
    "fmt"

    "reflect"
)

type Person struct {
    Name   string `json:"name"`
    Active bool
    Gender uint8
    Age    int
}

func main() {
    // 结构体
    tom := Person{Name: "tom", Active: true, Gender: 1, Age: 20}
    t := reflect.TypeOf(tom)
    fmt.Println(t, t.Kind()) // 类型是Person，种类是struct
    fmt.Println("~~~~~~~~~~~~~~~~~~~~~~~~~~")
    // 字段
    for i := 0; i < t.NumField(); i++ {
        f := t.Field(i)
        fmt.Printf("%d, %-8s\t%-8s\t%-8s\t%d\n",
            i,
            f.Name,         // 字段名
            f.Type,         // 字段类型
            f.Type.Kind(), // 字段类型的种类
            f.Offset,       // 相当于结构体首地址该字段值的偏移，string占16字节)
        )
    }
```

```
    fmt.Println(t.Size())
}
```

```
main.Person struct
~~~~~~~~~~~~~~~~~~~~~~~~~~
0, Name      string      string      0
1, Active    bool        bool        16
2, Gender    uint8       uint8       17
3, Age       int         int         24
32
```

可以看出这里对齐是凑够8个字节

```
type A struct {
    height uint16 // 2字节
    age    int64  // 偏移8字节，为了对齐，height凑够8
} // 总16字节

type B struct {
    gener  byte
    age    int
    height uint16
    active bool
} // 共24字节，age偏移8，height和active凑够8
```

接口

```
type Runner interface {
    run()
}

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


tom := Person{"Tom", 20}
t1 := reflect.TypeOf(tom)
t1.Implements(需要接口Runner的Type，也就是要获取接口类型Type)
```

怎么拿到接口的类型呢？如果接口能实例化倒还好说，但是接口是方法的声明。很容易写成下面的形式

```
tom := Person{"Tom", 20}
t1 := reflect.TypeOf(tom)
var b Runner = &tom
t2 := reflect.TypeOf(b)
fmt.Println(t1, t2) // main.Person *main.Person
t1.Implements(t2) // panic: reflect: non-interface type passed to
Type.Implements
```

也就是说通过上面的方式，无法获取接口类型，所以，要使用一种特殊做法。

```go
package main

import (
    "fmt"

    "reflect"
)

type Runner interface {
    run()
}

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

// func (Person) run() {
// }

func main() {
    tom := Person{"Tom", 20}
    t1 := reflect.TypeOf(tom)
    t2 := reflect.TypeOf((*Runner)(nil)) // *Runner的类型
    fmt.Println(t2, t2.Kind())
    t3 := t2.Elem() // 解析指针获得实例的类型
    fmt.Println(t3, t3.Kind())
    if t1.Implements(t3) {
        fmt.Println("实现了")
    } else {
        fmt.Println("未实现")
    }
}
```

`(*Runner)(nil)` 说明：

- nil是指针的零值，也就是空指针
- *Runner把空指针强制类型转换为*Runner类型空指针

Value

Value是一个结构体

```go
type Value struct {
    // typ holds the type of the value represented by a Value.
    typ *rtype

    // Pointer-valued data or, if flagIndir is set, pointer to data.
    // Valid when either flagIndir is set or typ.pointers() is true.
    ptr unsafe.Pointer

    // flag holds metadata about the value.
    // The lowest bits are flag bits:
```

```
    //  - flagStickyRO: obtained via unexported not embedded field, so read-only
    //  - flagEmbedRO: obtained via unexported embedded field, so read-only
    //  - flagIndir: val holds a pointer to the data
    //  - flagAddr: v.CanAddr is true (implies flagIndir)
    //  - flagMethod: v is a method value.
    // The next five bits give the Kind of the value.
    // This repeats typ.Kind() except for method values.
    // The remaining 23+ bits give a method number for method values.
    // If flag.kind() != Func, code can assume that flagMethod is unset.
    // If ifaceIndir(typ), code can assume that flagIndir is set.
    flag

    // A method value represents a curried method invocation
    // like r.Read for some receiver r. The typ+val+flag bits describe
    // the receiver r, but the flag's Kind bits say Func (methods are
    // functions), and the top bits of the flag give the method number
    // in r's type's method table.
}
```

reflect.ValueOf(rawValue)返回reflect.Value类型，包含着rawValue的值的有关信息。

```
var a int = 100
v := reflect.ValueOf(a)
fmt.Printf("%T: %[1]v\n", v) // reflect.Value: 100
```

Value和原始值

reflect.Value 与原始值之间可以通过 `值包装` 和 `值获取` 相互转化。

方法签名	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 int64 类型返回，所有有符号整型均可以此方式返回 如果需要的是int类型，则需要强制类型转换
Uint() uint64	将值以 uint64 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

```
var a = 100
v := reflect.ValueOf(a) // 原始值 => Value，原始值包装
i := v.Interface() // 等价于 var i interface{} = (v's underlying value)，Value => 原始值，值获取
```

```
package main

import (
    "fmt"

    "reflect"
)

func main() {
    var a = 100
    v := reflect.ValueOf(a)
    i := v.Interface() // 等价于 var i interface{} = (v's underlying value)
    // 1 接口类型断言
    j := i.(int) // 获得int类型原始值，断言失败panic
    fmt.Printf("%T %[1]d\n", j)
    j1, isStr := i.(string) // 断言失败与否看isStr，失败不panic
    fmt.Println(j1, isStr)
    // 2 值获取
    k := v.Int() // 返回的是int64
    fmt.Printf("%T %[1]d\n", k)
    fmt.Println(int(k)) // 强制类型转换为int获得原始值
}

```

空值和有效性判断

方法签名	说明
IsValid() bool	判断值是否有效。 当值本身非法时，返回 false，例如 reflect.ValueOf(nil).IsValid()就是false 常用来判断返回值是否有效
IsZero() bool	值是否是零值。如果值无效则panic
IsNil() bool	值是否为 nil。 必须是chan、func、interface、map、pointer、slice，否则panic。 类似于语言层的`v== nil`操作 常用来判断指针是否为空

```
package main

import (
    "fmt"

    "reflect"
)

func main() {
```

```
    var a = 100
    v := reflect.ValueOf(a)
    fmt.Println(
        v.IsValid(),  // true
        // v.IsNil(), // 必须是chan、func、interface、map、pointer、slice，否则panic
        v.IsZero(),   // false
    )
    var b *int
    v = reflect.ValueOf(b)
    fmt.Println(
        v.IsValid(), // true
        v.IsNil(),   // true
        v.IsZero(),  // true
    )

    v = reflect.ValueOf(nil)
    fmt.Println(
        v.IsValid(), // false，因为nil是用来给某种类型做零值，直接用nil不知道其类型，所以无效
    )
}
```

注意，上例中reflect.ValueOf(nil).IsValid()为false，而reflect.ValueOf(b).IsValid()为true，因为b是有类型的，它是*int不过是空指针罢了，所以有效，而nil不是。

反射和结构体

方法签名	说明
Field(i int) Value	根据索引，返回索引对应的结构体成员字段的反射值对象。当值不是结构体或索引超界时panic
NumField() int	返回结构体成员字段数量。当值不是结构体或索引超界时发生宕机
FieldByName(name string) Value	根据给定字符串返回字符串对应的结构体字段。没有找到时返回零值，当值不是结构体panic
FieldByIndex(index []int) Value	多层成员访问时，根据 []int 提供的每个结构体的字段索引，返回字段的值。 没有找到时返回零值，当值不是结构体panic
Method(i int) Value	根据索引，返回索引对应的结构体成员方法的反射值对象。当值不是结构体或索引超界时panic
MethodByName(name string) Value	根据给定字符串返回字符串对应的结构体方法。没有找到时返回零值，当值不是结构体panic

```
package main

import (
    "fmt"
    "reflect"
)

type person struct {
```

```
    name string
    age  int
}

func (p person) GetName() string {
    return p.name
}

func main() {
    var a = person{}
    a.name = "tom"
    // a.age = 100
    v := reflect.ValueOf(a)
    t := v.Type()
    fmt.Println(v, t, t.Kind())
    fmt.Println("~~~~~~~~~~~~~~~~~~~~~~~~~~")
    // 下面使用Type和Value遍历字段的区别
    for i := 0; i < t.NumField(); i++ {
        f := t.Field(i)
        fmt.Println(
            f.Name, f.Index, f.Offset,
            f.Anonymous, f.IsExported(), // Type字段类型信息
        )
    }

    for i := 0; i < v.NumField(); i++ {
        f := v.Field(i)
        fmt.Println(i, f.IsValid(), f.IsZero()) // Value关注字段的值
    }
    fmt.Println("~~~~~~~~~~~~~~~~~~~~~~~~~~")
    fmt.Println(
        v.FieldByName("name").IsZero(),            // 通过v找底层结构体的name字
段
        v.Field(1).IsValid(), v.Field(1).IsZero(), // 字段age
        v.FieldByIndex([]int{1}).IsValid(),  // 字段age
        v.FieldByName("score").IsValid(),    // score字段不存在，无效
        v.MethodByName("GetName").IsValid(), // GetName方法不存在，无效
    )
}
```

反射调用函数

```
package main

import (
    "fmt"

    "reflect"
)

type person struct {
    name string
    age  int
}

```

```go
func (p person) GetName(prefix, suffix string) string {
    return fmt.Sprintf("%s %s %s", prefix, p.name, suffix)
}

func main() {
    var a = person{}
    a.name = "tom"
    // a.age = 100
    v := reflect.ValueOf(a)          // 结构体的Value
    vf := v.MethodByName("GetName") // 函数的Value
    fmt.Println(v, vf)

    // 构建参数列表，需要2个入参
    p1 := reflect.ValueOf("!!")
    p2 := reflect.ValueOf("##")
    inParams := []reflect.Value{p1, p2}
    outParams := vf.Call(inParams) // Call调用需要[]reflect.Value的参数列表
    fmt.Println(outParams)
}
```

主要是参数列表的构建有点麻烦。

反射修改值

方法签名	说明
Elem() Value	取值指向的元素值，类似于语言层 * 操作。 当值类型不是指针或接口时panic，空指针时返回 nil 的 Value
Addr() Value	对可寻址的值返回其地址，类似于语言层 & 操作。当值不可寻址时panic
CanAddr() bool	表示值是否可寻址
CanSet() bool	返回值能否被修改。要求值可寻址且是导出的字段

方法签名	说明
Set(x Value)	将值设置为传入的反射值对象的值
SetInt(x int64)	使用 int64 设置值。当值的类型不是 int、int8、int16、 int32、int64 时会发生宕机
SetUint(x uint64)	使用 uint64 设置值。当值的类型不是 uint、uint8、uint16、uint32、uint64 时会发生宕机
SetFloat(x float64)	使用 float64 设置值。当值的类型不是 float32、float64 时会发生宕机
SetBool(x bool)	使用 bool 设置值。当值的类型不是 bod 时会发生宕机
SetBytes(x []byte)	设置字节数组 []bytes值。当值的类型不是 []byte 时会发生宕机
SetString(x string)	设置字符串值。当值的类型不是 string 时会发生宕机

如果CanSet() 返回false，调用以上Set*方法都会panic

值可修改条件之一：可被寻址

```go
package main

import (
    "fmt"

    "reflect"
)

func main() {
    var a int = 100
    v := reflect.ValueOf(a)
    fmt.Println(v.CanAddr(), v.CanSet()) // false false
    // 不可寻址，不可设置，如果像下面这样强行设置会panic
    v.SetInt(200) // reflect.Value.SetInt using unaddressable value
    fmt.Println(v)
}
```

上面程序调整成指针，如下

```go
package main

import (
    "fmt"

    "reflect"
)

```

```go
func main() {
    var a int = 100
    v := reflect.ValueOf(&a).Elem()      // a指针指向的元素的Value
    fmt.Println(v.CanAddr(), v.CanSet()) // true true
    v.SetInt(200)
    fmt.Println(v, int(v.Int()))
}
```

值可修改条件二：被导出

结构体成员中，如果字段没有被导出，不使用反射也可以被访问，但不能通过反射修改，代码如下：

```go
package main

import (
    "fmt"

    "reflect"
)

type person struct {
    Name string
    age  int
}

func (p person) GetName(prefix, suffix string) string {
    return fmt.Sprintf("%s %s %s", prefix, p.Name, suffix)
}

func main() {
    v := reflect.ValueOf(person{Name: "Tom", age: 20})
    vf1 := v.FieldByName("Name") // 导出字段
    fmt.Println(vf1, vf1.CanAddr(), vf1.CanSet()) // false false
    // vf1.SetString("Jerry") // reflect.Value.SetString using unaddressable value
    vf2 := v.FieldByName("age") // 未导出字段
    fmt.Println(vf2, vf2.CanAddr(), vf2.CanSet()) // false false
    vf2.SetInt(30) // reflect.Value.SetInt using value obtained using unexported
}
```

错误提示age是未导出字段，但是Name也不行，因为不可寻址。因此，需要使用结构体指针。

```go
package main

import (
    "fmt"

    "reflect"
)

type person struct {
    Name string
```

```
    age  int
}

func main() {
    v := reflect.ValueOf(&person{Name: "Tom", age: 20})
    vf1 := v.Elem().FieldByName("Name")           // 导出字段
    fmt.Println(vf1, vf1.CanAddr(), vf1.CanSet()) // true true
    vf1.SetString("Jerry")                        // 成功修改
    vf2 := v.Elem().FieldByName("age")            // 未导出字段
    fmt.Println(vf2, vf2.CanAddr(), vf2.CanSet()) // true false
    // vf2.SetInt(30) // reflect.Value.SetInt using value obtained using unexported
    fmt.Println(v) // 名字已经变成了Jerry了
}
```

反射创建实例

需要用到`reflect.New(typ reflect.Type) reflect.Value`，简单讲就是将Type New成Value。

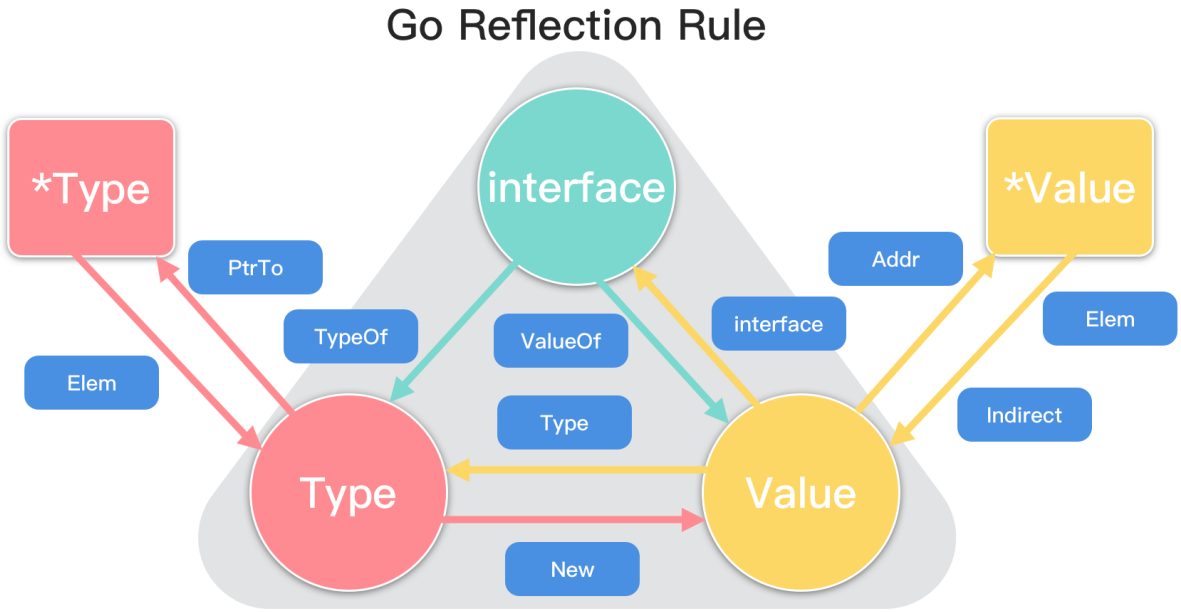
```
package main

import (
    "fmt"
    "reflect"
)

func main() {
    var a int = 100
    t := reflect.TypeOf(a) // 提取类型信息
    v := reflect.New(t)    // 创建一个该类型的新的零值返回指针的Value，相当于new(int)
    fmt.Println(v, v.Elem(), v.Type(), v.Kind()) // 内存地址 0 *int ptr
}
```

- Value.Type()，Value => Type
- Value.Kind()，Value => Kind
- Type.New()，Type => 指定类型新零值的指针的Value

总结



反射可以从接口值当中通过TypeOf()获取Type对象，通过ValueOf()获取Value对象

Type对象只有类型信息，如果想获得值、修改值，可以通过New()获得Value对象，这个Value对象是指向该类型的零值的指针

Value对象可以通过interface()转换成接口值

value对象可以通过Type()返回Type对象

反射的弊端

- 代码难以阅读，难以维护
- 编译期间不能发现类型错误，覆盖测试难度很大，有些Bug需要线上运行时才可能发现，并造成严重后果
- 反射性能很差，通常比正常代码慢一到两个数量级。如果性能要求高时，或反复调用的代码块里建议不要使用反射

反射主要应用场合就是写库或框架，一般用不到，再一个面试时候极低概率被问到。

自行实现Json序列化 https://zhuanlan.zhihu.com/p/424695673