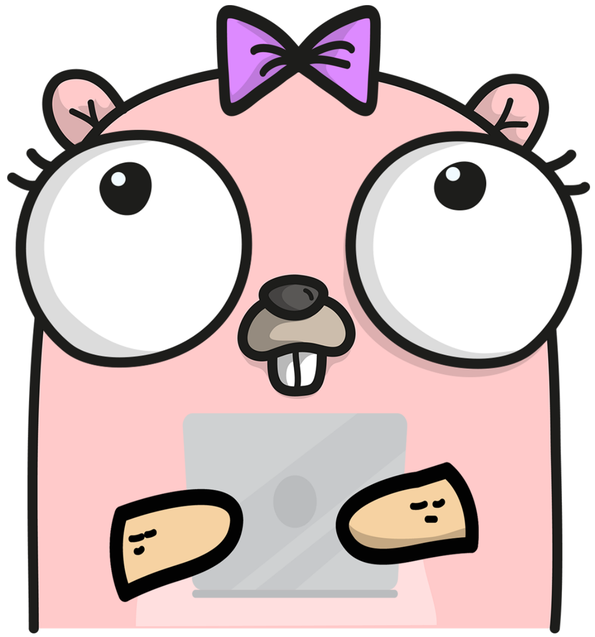
Magic of Reflection

Nastya Kruglikova

Software Engineer
SoftServe



"You don't need reflect to write great code, but knowing it will cement your understanding of Go."

Voltaire, 1767

Agenda

1. Definition of reflection
2. The laws of reflection
3. Examples

Reflection

in computing is the ability of a program to:

- examine its own structure,
- introspect, and modify its own structure and behavior at runtime.

- Python - dir([object]), type, hasattr, setattr
- Ruby - class(), instance_methods(), instance_variables()
- PHP - ReflectionClass
- Java - getClass(), java.lang.reflect
- Haskell - Data.Reflection
- C - sorry, no
- Go - reflect

'reflect' package

1. Provides the capability to reflect
2. Defines two important types
    - reflect.Type
    - reflect.Value

The laws of reflection

by Rob Pike

```
var x float64 = 3.4
v := reflect.ValueOf(x)
fmt.Println("type:", v.Type())
```

```
prints
```

```
type: float64
```

```
var x float64 = 3.4
v := reflect.ValueOf(x)
fmt.Println("type:", v.Type())
```

```
prints
```

```
type: float64
```

```
func ValueOf(i interface{}) Value
```

```
func TypeOf(i interface{}) Type
```

1. Reflection goes from interface value to reflection object.

```
var x float64 = 3.4
v := reflect.ValueOf(x)
fmt.Println("type:", v.Type())
fmt.Println("kind:", v.Kind())
```

```
prints
```

```
type: float64
kind: float64
```

```
type Name string
var x Name = "nastya"
v := reflect.ValueOf(x)
fmt.Println("type:", v.Type())
fmt.Println("kind:", v.Kind())
```

```
prints
```

```
type: main.Name
kind: string
```

```
type Kind uint
const (
        Invalid Kind = iota
        Bool
        Int
        Int8
        Int16
        Int32
        Int64
        Uint
        Uint8
        Uint16
        Uint32
        Uint64
        Uintptr
        Float32
        Float64
        Complex64
        Complex128
        Array
        Chan
        Func
        Interface
        Map
        Ptr
        Slice
        String
        Struct
        UnsafePointer
)
```

```
y := v.Interface()
fmt.Println(y) // 3.4
```

2. Reflection goes from reflection object to interface value.

Reiterating:

Reflection goes from interface values to reflection objects and back again.

```
var x float64 = 3.4
v := reflect.ValueOf(x)
v.SetFloat(7.1) // Error: will panic.
```

```
panic: reflect.Value.SetFloat using unaddressable value
```

```
var x float64 = 3.4
v := reflect.ValueOf(x)
fmt.Println("settability of v:", v.CanSet())
```

```
var x float64 = 3.4
v := reflect.ValueOf(x)
fmt.Println("settability of v:", v.CanSet())

prints

settability of v: false
```

What is settability?

- It's the property that a reflection object can modify the actual storage that was used to create the reflection object.
- Settability is determined by whether the reflection object holds the original item.

Use pointer!

```
var x float64 = 3.4
p := reflect.ValueOf(&x) // Note: take the address of x.
fmt.Println("type of p:", p.Type())
fmt.Println("settability of p:", p.CanSet())
```

Use pointer!

```
var x float64 = 3.4
p := reflect.ValueOf(&x) // Note: take the
address of x.
fmt.Println("type of p:", p.Type())
fmt.Println("settability of p:", p.CanSet())
```

```
prints
```

```
type of p: *float64
settability of p: false
```

Use pointer!

```
var x float64 = 3.4
p := reflect.ValueOf(&x)
v := p.Elem()
fmt.Println("type of p:", v.Type())
fmt.Println("settability of v:", v.CanSet())
```

Use pointer!

```
var x float64 = 3.4
p := reflect.ValueOf(&x)
v := p.Elem()
fmt.Println("type of v:", v.Type())
fmt.Println("settability of v:", v.CanSet())
```

prints

```
type of v: float64
settability of v: true
```

3. To modify a reflection object, the value must be settable.

Use cases

Modifying struct

```
type T struct {
    A int
    B string
}
t := T{23, "skidoo"}
s := reflect.ValueOf(&t).Elem()
typeOfT := s.Type()
for i := 0; i < s.NumField(); i++ {
    f := s.Field(i)
    fmt.Printf("%d: %s %s = %v settability=%t\n", i,
        typeOfT.Field(i).Name, f.Type(), f.Interface(),
f.CanSet())
}

prints

0: A int = 23 settability=true
1: B string = skidoo settability=true
```

Modifying struct

```
s.Field(0).SetInt(77)
s.Field(1).SetString("Sunset Strip")
fmt.Println("t is now", t)
```

```
prints
```

```
t is now {77 Sunset Strip}
```

```
type Foo struct {
    s string
    i int
    j interface{}
}

func main() {
    x := Foo{"hello", 2, 3.0}
    v := reflect.ValueOf(x)

    s := v.Field(0)
    fmt.Printf("%T %v\n", s.String(), s.String())

    i := v.FieldByName("i")
    fmt.Printf("%T %v\n", i.Int(), i.Int())

    j := v.FieldByName("j").Elem()
    fmt.Printf("%T %v\n", j.Float(), j.Float())
}
```

Tags

```
type Member struct {
    Age int `something:"age"`
}
```

Tags

```
type Member struct {
    Age int `something:"age"`
}

func main() {
    member := Member{34}
    t := reflect.TypeOf(member)
    field := t.Field(0)
    fmt.Print(field.Tag.Get("something"))
}
```

SelectCase

```
c1 := make(chan string)
c2 := make(chan string)

select {
case msg1 := <-c1:
    fmt.Println("received ", msg1)
case msg2 := <-c2:
    fmt.Println("received ", msg2)
}
```

reflect.Select

```
func Select(cases []SelectCase) (chosen
int, recv Value, recvOK bool)
```

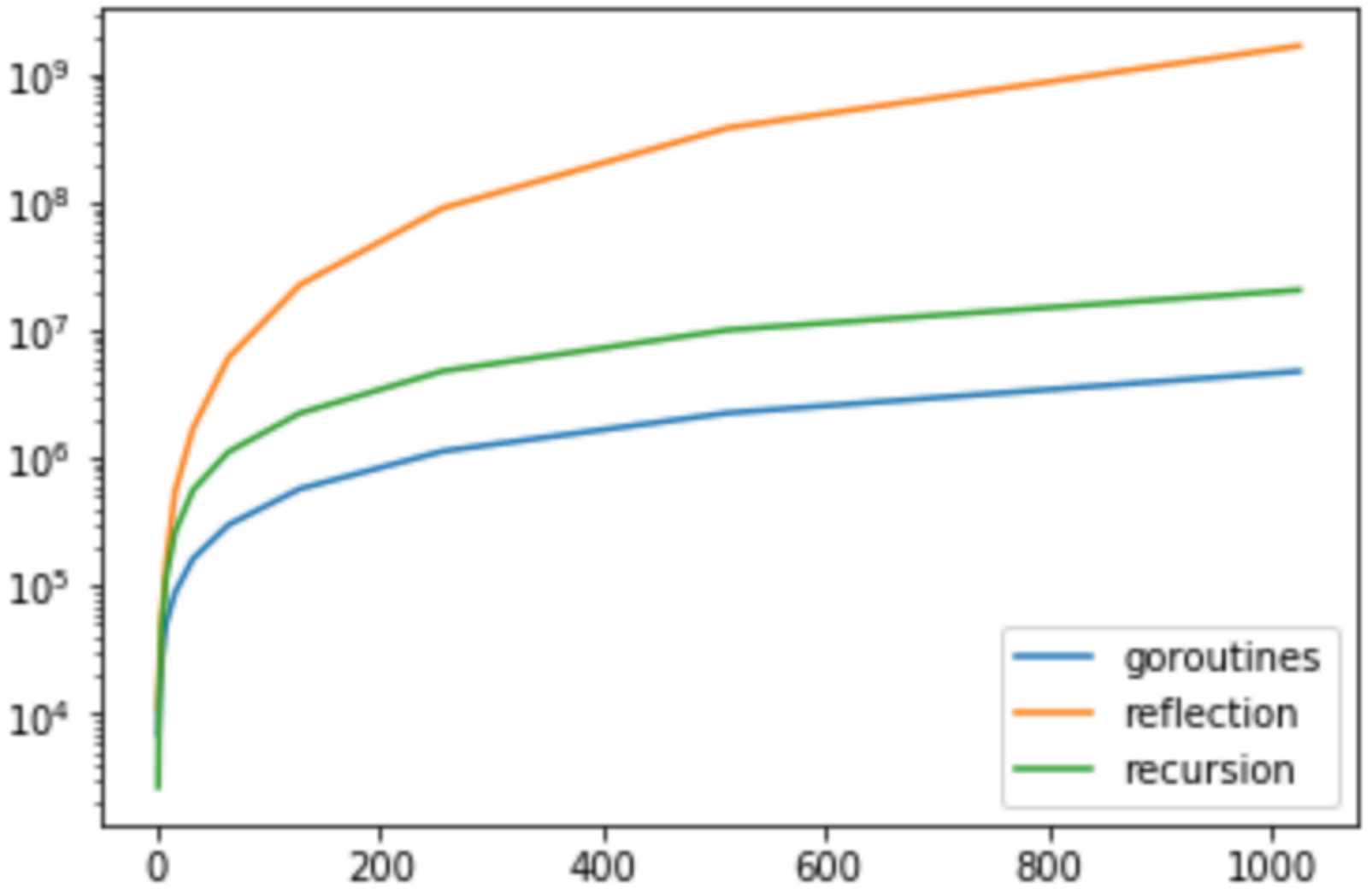
```
type SelectCase struct {
    Dir  SelectDir // direction of case
    Chan Value     // channel to use (for
send or receive)
    Send Value     // value to send (for
send)
}
```

reflect.Select

```
const (
                    SelectDir = iota
    _
    SelectSend                  // case Chan <- Send
    SelectRecv                  // case <-Chan:
    SelectDefault               // default
)
```

reflect.Select

```
cases := make([]reflect.SelectCase,
len(chans))
for i, ch := range chans {
    cases[i] = reflect.SelectCase{Dir:
reflect.SelectRecv, Chan:
reflect.ValueOf(ch)}
}
chosen, value, ok := reflect.Select(cases)
// ok will be true if the channel has not
been closed.
ch := chans[chosen]
msg := value.String()
```


10^9
10^8
10^7
10^6
10^5
10^4
0
200
400
600
800
1000
goroutines
reflection
recursion

reflect.MakeFunc

reflect.MakeFunc

```
func MakeFunc(typ Type, fn func(args []Value)
(results []Value)) Value
```

reflect.MakeFunc

```
func swap(in []reflect.Value) []reflect.Value {
    return []reflect.Value{in[1], in[0]}
}

func main() {
    makeSwap := func(fptr interface{}) {
        fn := reflect.ValueOf(fptr).Elem()
        v := reflect.MakeFunc(fn.Type(), swap)
        fn.Set(v)
    }

    var intSwap func(int, int) (int, int)
    makeSwap(&intSwap)
    fmt.Println(intSwap(1, 3)) // 3 1
}
```

reflect.MakeFunc

```
func swap(in []reflect.Value) []reflect.Value {
    return []reflect.Value{in[1], in[0]}
}

func main() {
    makeSwap := func(fptr interface{}) {
        fn := reflect.ValueOf(fptr).Elem()
        v := reflect.MakeFunc(fn.Type(), swap)
        fn.Set(v)
    }

    var strSwap func(string, string) (string, string)
    makeSwap(&strSwap)
    fmt.Println(strSwap("test", "golang")) // golang
test
}
```

… and more

… and more

func Copy(dst, src Value) int
func DeepEqual(x, y interface{}) bool
func Select(cases []SelectCase) (chosen int, recv Value, recvOK bool)
func Swapper(slice interface{}) func(i, j int)
type ChanDir
  func (d ChanDir) String() string
type Kind
  func (k Kind) String() string
type Method
type SelectCase
type SelectDir
type SliceHeader
type StringHeader
type StructField
type StructTag
  func (tag StructTag) Get(key string) string
  func (tag StructTag) Lookup(key string) (value string, ok bool)
type Type
  func ArrayOf(count int, elem Type) Type
  func ChanOf(dir ChanDir, t Type) Type
  func FuncOf(in, out []Type, variadic bool) Type
  func MapOf(key, elem Type) Type
  func PtrTo(t Type) Type
  func SliceOf(t Type) Type
  func StructOf(fields []StructField) Type
  func TypeOf(i interface{}) Type
type Value
  func Append(s Value, x ...Value) Value
  func AppendSlice(s, t Value) Value
  func Indirect(v Value) Value
  func MakeChan(typ Type, buffer int) Value
  func MakeFunc(typ Type, fn func(args []Value) (results []Value)) Value
  func MakeMap(typ Type) Value
  func MakeMapWithSize(typ Type, n int) Value
  func MakeSlice(typ Type, len, cap int) Value
  func New(typ Type) Value
  func NewAt(typ Type, p unsafe.Pointer) Value
  func ValueOf(i interface{}) Value
  func Zero(typ Type) Value
  func (v Value) Addr() Value
  func (v Value) Bool() bool
  func (v Value) Bytes() []byte
  func (v Value) Call(in []Value) []Value
  func (v Value) CallSlice(in []Value) []Value
  func (v Value) CanAddr() bool
  func (v Value) CanInterface() bool
  func (v Value) CanSet() bool
  func (v Value) Cap() int
  func (v Value) Close()
  func (v Value) Complex() complex128
  func (v Value) Convert(t Type) Value
  func (v Value) Elem() Value
  func (v Value) Field(i int) Value
  func (v Value) FieldByIndex(index []int) Value
  func (v Value) FieldByName(name string) Value
  func (v Value) FieldByNameFunc(match func(string) bool) Value
  func (v Value) Float() float64
  func (v Value) Index(i int) Value
  func (v Value) Int() int64
  func (v Value) Interface() (i interface{})
  func (v Value) InterfaceData() [2]uintptr
  func (v Value) IsNil() bool
  func (v Value) IsValid() bool
  func (v Value) Kind() Kind
  func (v Value) Len() int
  func (v Value) MapIndex(key Value) Value
  func (v Value) MapKeys() []Value
  func (v Value) Method(i int) Value
  func (v Value) MethodByName(name string) Value
  func (v Value) NumField() int
  func (v Value) NumMethod() int
  func (v Value) OverflowComplex(x complex128) bool
  func (v Value) OverflowFloat(x float64) bool
  func (v Value) OverflowInt(x int64) bool
  func (v Value) OverflowUint(x uint64) bool
  func (v Value) Pointer() uintptr
  func (v Value) Recv() (x Value, ok bool)
  func (v Value) Send(x Value)
  func (v Value) Set(x Value)
  func (v Value) SetBool(x bool)
  func (v Value) SetBytes(x []byte)
  func (v Value) SetCap(n int)
  func (v Value) SetComplex(x complex128)
  func (v Value) SetFloat(x float64)
  func (v Value) SetInt(x int64)
  func (v Value) SetLen(n int)
  func (v Value) SetMapIndex(key, val Value)
  func (v Value) SetPointer(x unsafe.Pointer)
  func (v Value) SetString(x string)
  func (v Value) SetUint(x uint64)
  func (v Value) Slice(i, j int) Value
  func (v Value) Slice3(i, j, k int) Value
  func (v Value) String() string
  func (v Value) TryRecv() (x Value, ok bool)
  func (v Value) TrySend(x Value) bool
  func (v Value) Type() Type
  func (v Value) Uint() uint64
  func (v Value) UnsafeAddr() uintptr
type ValueError
  func (e *ValueError) Error() string
Bugs


163


"With great power comes great responsibility"
~Voltaire

When to use reflection?

Clear is better than clever.

Clear is better than clever.
Reflection is never clear.

better

never

References

- Golang 'reflect' package

  https://golang.org/pkg/reflect/

- The Laws of Reflection

  https://blog.golang.org/laws-of-reflection