Reflection in Go

Joshua Liebow-Feeser
hello@joshlf.com

Overview

- Reflection in Go is reflection on types
- Reflection goes:
  - Static types
  - Dynamic types (interfaces)
  - Reflection objects (sometimes)
  - Dynamic types
  - Static types

Interfaces

```
type Reader interface {
    Read(p []byte) (n int, err error)
}

type byteReader byte
func (b byteReader) Read(p []byte) (n int, err error) {
    for i := range p {
        p[i] = byte(b)
    }
    return len(p), nil
}
```

Interfaces

```
var a int = 1
var b int = a // Checks to make sure that a is an int

var c byteReader = 'a'
var d Reader = c // Checks to make sure that c is a Reader

buf := make([]byte, 10)
d.Read(buf)
```

Overview

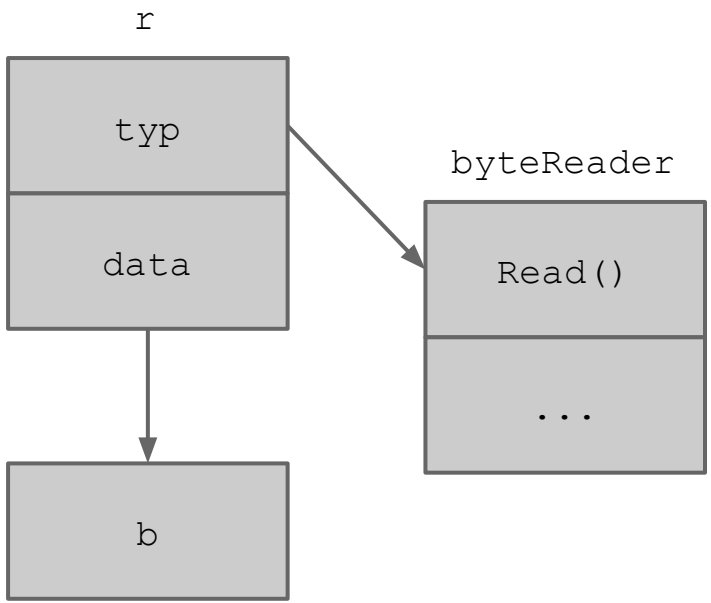
- Reflection goes:
  - Static types
  - Dynamic types (interfaces)
  - Reflection objects (sometimes)
  - Dynamic types
  - Static types

Interfaces

```
type charReader char
func (c charReader) Read(p []byte) (n int, err error) { ... }

buf := make([]byte, 10)

var b byteReader = 'a'
var c charReader = 'b'
var r Reader = c

b.Read(buf)
c.Read(buf)
r.Read(buf)
```

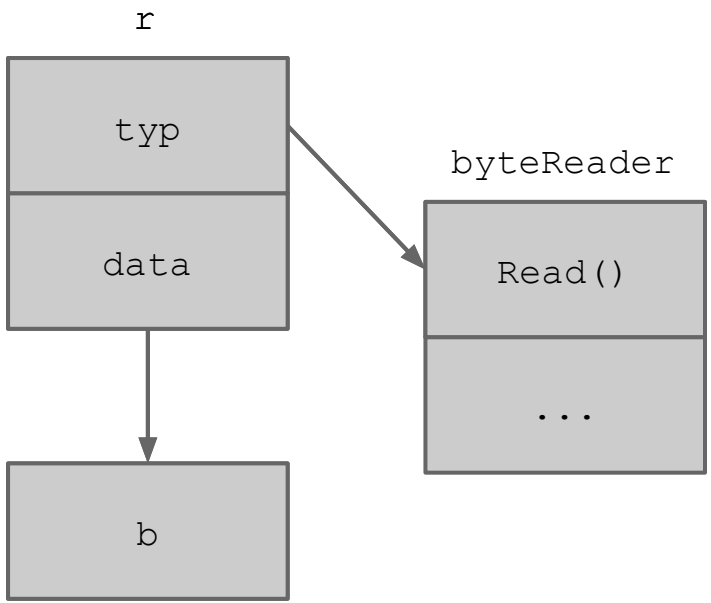
Implementation of Interfaces

```
var r Reader
struct Reader {
    void *data;
    type *typ;
}

var b byteReader = 'a'
r = b
r.typ = byteReader;
r.data = malloc(sizeof(byteReader));
*r.data = b;
```



Implementation of Interfaces

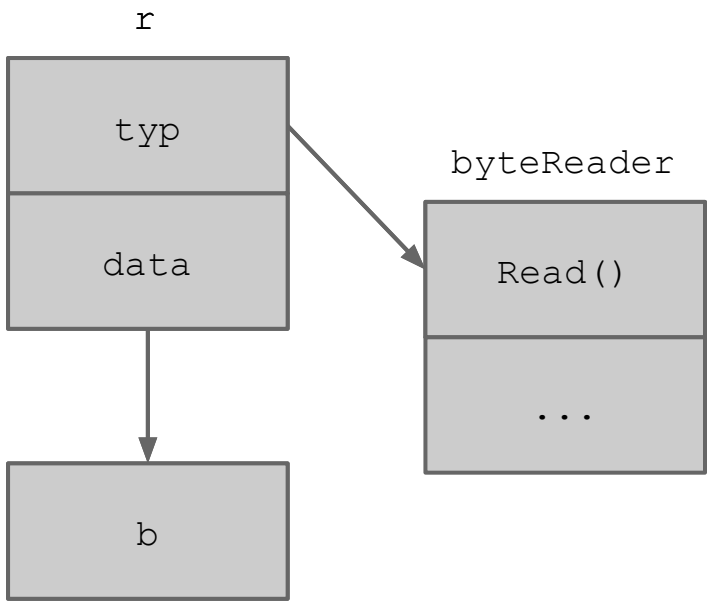
```
var r Reader
struct Reader {
    void *data;
    type *typ;
}

r = byteReader('a')
buf := make([]byte, 10)
r.Read(buf)
r.typ->Read(r.data, buf);
```



Implementation of Interfaces

Take-home

Interface values hold both the **concrete value** and a **pointer to the type** of that concrete value.



Compare Interface Values

```
var r1, r2 Reader
r1 = byteReader('a')
r2 = charReader('b')

if r1 == r2 {
    ...
}
if (r1.typ == r2.typ && *r1.data == *r2.data) {
    ...
}
```

Type Assertion

```
func GetType(val interface{}) string {
    i, ok := val.(int)
    if ok {
        return fmt.Sprint("int:", i)
    }
    return "What is this?"
}
int i = 0; // Zero value of int
bool ok = (val.typ == int);
if (ok)
    i = *val.data;
```

Overview

- Reflection goes:
  - Static types
  - Dynamic types (interfaces)
  - Reflection objects (sometimes)
  - Dynamic types
  - Static types

Type Assertion Against Interface

```
type stringer interface {
    String() string
}
func String(val interface{}) string {
    s, ok := val.(stringer)
    if ok {
        return s.String()
    }
    return "What is this?"
}
```

```
stringer s;
// Set to zero value of interface
s.typ = NULL;
s.data = NULL;
bool ok = impl(val.typ, stringer);
if (ok) {
    s.typ = val.typ;
    s.data = malloc(sizeof(val.typ));
    *s.data = *val.data;
}
```

Overview

- Reflection goes:
  - Static types
  - Dynamic types (interfaces)
  - Reflection objects (sometimes)
  - Dynamic types
  - Static types

Type Switch

```
func GetType(val interface{}) string {
    switch concrete := val.(type) {
    case int:
        return fmt.Sprint("int:", concrete) // concrete has type int
    case bool:
        return fmt.Sprint("bool:", concrete) // concrete has type bool
    default:
        return "What is this?"
    }
}
```

The Reflect Package

```
i := 1
val := reflect.ValueOf(i) // val has type reflect.Value
typ := reflect.TypeOf(i)  // typ has type reflect.Type

i := 1
var iface interface{} = i
val := reflect.ValueOf(iface)
typ := reflect.TypeOf(iface)

typ = val.Type()
```

Overview

- Reflection goes:
  - Static types
  - Dynamic types (interfaces)
  - Reflection objects (sometimes)
  - Dynamic types
  - Static types

Reflection on Types

```
type myStruct struct {
    m map[int]string
    b []bool
}

typ := reflect.TypeOf(myStruct{})
fmt.Println(typ.NumField())          // Prints "2"

mtyp := typ.Field(0).Type            // Represents the type map[int]string
fmt.Println(mtyp.Key(), mtyp.Elem()) // Prints "int string"

styp := typ.Field(1).Type            // Represents the type []bool
fmt.Println(styp.Elem())             // Prints "bool"
```

Reflection on Values

```
var u uint64 = 42
val := reflect.ValueOf(u)
fmt.Println(u == val.Uint())
val.SetUint(52) // panics!

val = reflect.ValueOf(&u)
val.SetUint(52) // panics!

val = reflect.ValueOf(&u).Elem()
val.SetUint(52)
fmt.Println(u) // Prints "52"
```

Reflection Output

```
val := reflect.ValueOf(uint64(52))
val.Println(val.Uint()) // Prints "52"

u := val.Interface().(uint64)
fmt.Println(u) // Prints "52"

var u uint64 = 42
val := reflect.ValueOf(&u).Elem()
val.SetUint(52)
fmt.Println(u) // Prints "52"
```

Overview

- Reflection goes:
  - Static types
  - Dynamic types (interfaces)
  - Reflection objects (sometimes)
  - Dynamic types
  - Static types

Reflection in Practice

```
func Map(slc []T, pred func(T) U) []U
func Map(slc, pred interface{}) interface{} {
    slice := reflect.ValueOf(slc)
    if slice.Type().Kind() != reflect.Slice {
        panic("Expected a slc to be a slice")
    }

    predFn := reflect.ValueOf(pred)
    if predFn.Type().Kind() != reflect.Func {
        panic("Expected pred to be a function")
    }
    ...
}
```

Reflection in Practice

```
func Map(slc []T, pred func(T) U) []U
func Map(slc, pred interface{}) interface{} {
    ...
    sliceType := slice.Type()    // Represents type []T
    elemType := sliceType.Elem() // Represents type T
    predType := predFn.Type()    // Represents type func(T) U? Verify:
    if predType.NumIn() != 1 || predType.NumOut() != 1
       || predType.In(0) != elemType {
        panic("Expected pred to be of type func(T) bool")
    }
    ...
}
```

Reflection in Practice

```
func Map(slc []T, pred func(T) U) []U
func Map(slc, pred interface{}) interface{} {
    ...
    ret := reflect.MakeSlice(reflect.SliceOf(predType.Out(0)),
            slice.Len(), slice.Cap())
    ...
}
```

Reflection in Practice

```
func Map(slc []T, pred func(T) U) []U
func Map(slc, pred interface{}) interface{} {
    ...
    args := make([]reflect.Value, 1)
    for i := 0; i < slice.Len(); i++ {
        args[0] = slice.Index(i)
        ret.Index(i).Set(predFn.Call(args)[0])
    }

    return ret.Interface()
}
```

Reflection in Practice

```
func Map(slc []T, pred func(T) U) []U

ints := []int{1, 2, 3}
pred := func(i int) float64 { return float64(i) }
ret := Map(ints, pred)

floats, ok := ret.([]float64)
if !ok {
    fmt.Println("Oh no - should be of type []float64!")
} else {
    fmt.Println(floats) // Prints "[1.0 2.0 3.0]"
}
```

Links n' Stuff

The reflect package [golang.org/pkg/reflect]

The laws of reflection [blog.golang.org/laws-of-reflection]

Implementation of Interfaces [research.swtch.com/interfaces]

Me [hello@joshlf.com]