




# Golang Developer. Professional

[otus.ru](https://otus.ru)

 Проверить, идет ли запись

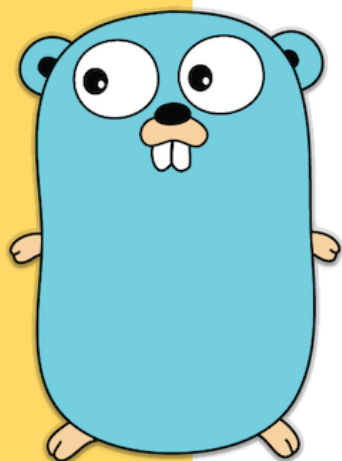
# Меня хорошо видно && слышно?

 Ставим “+”, если все хорошо  
“-”, если есть проблемы

Тема вебинара

# Рефлексия

Алексей Романовский



# Правила вебинара



Активно  
участвуем



Off-topic обсуждаем  
в учебной группе

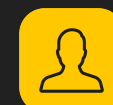


Задаем вопрос  
в чат или голосом



Вопросы вижу в чате,  
могу ответить не сразу

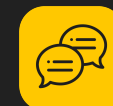
## Условные обозначения



Индивидуально



Время, необходимое  
на активность



Пишем в чат



Говорим голосом



Документ



Ответьте себе или  
задайте вопрос

# План занятия

- Что такое рефлексия?
- Использование рефлексии
  - На уровне языка
  - Пакет reflect
  - Type и Value
  - unsafe.Pointer

# Рефлексия

Что такое рефлексия?

# Рефлексия

**Рефлексия (отражение)** — процесс, во время которого программа может отслеживать и модифицировать собственную структуру и поведение во время выполнения.

```
MyClass.__class__ // Python  
f.GetType().GetMethod("PrintHello").Invoke(f, null); // C#  
QPushButton::staticMetaObject.className(); // C++ Qt
```

[https://en.wikipedia.org/wiki/Reflection\\_\(computer\\_programming\)](https://en.wikipedia.org/wiki/Reflection_(computer_programming))

# Когда нужна рефлексия?



# Когда нужна рефлексия?

- Для работы с типами, которые не удовлетворяют одному интерфейсу.
- Для работы с данными, которые не определены во время компиляции.

# Примеры использования рефлексии:

- `encoding/json` и `encoding/xml`
- `text/template` и `html/template`
- <https://github.com/kelseyhightower/envconfig>

# Рефлексия: не путать с AST

**AST (абстрактное синтаксическое дерево)** — ориентированное дерево, в котором внутренние вершины сопоставлены с операторами языка программирования, а листья — с соответствующими операндами.

```
fset := token.NewFileSet()
f, err := parser.ParseFile(fset, "src.go", src, 0)

// Inspect the AST and print all identifiers and literals.
ast.Inspect(f, func(n ast.Node) bool {
    var s string
    switch x := n.(type) {
    case *ast.BasicLit:
        s = x.Value
    case *ast.Ident:
        s = x.Name
    }
    if s != "" {
        fmt.Printf("%s:\t%s\n", fset.Position(n.Pos()), s)
    }
    return true
})
```

[https://en.wikipedia.org/wiki/Abstract\\_syntax\\_tree](https://en.wikipedia.org/wiki/Abstract_syntax_tree)

<https://golang.org/pkg/go/ast/>

# Рефлексия на уровне языка

Как думаете, о чем речь?

# Рефлексия на уровне языка

Каждая переменная в Go обладает статическим типом, переменные разных типов нельзя присваивать (в большинстве случаев).

```
type MyInt int

var i int
var j MyInt
j = i // ошибка компиляции
```

# Рефлексия на уровне языка: интерфейсы

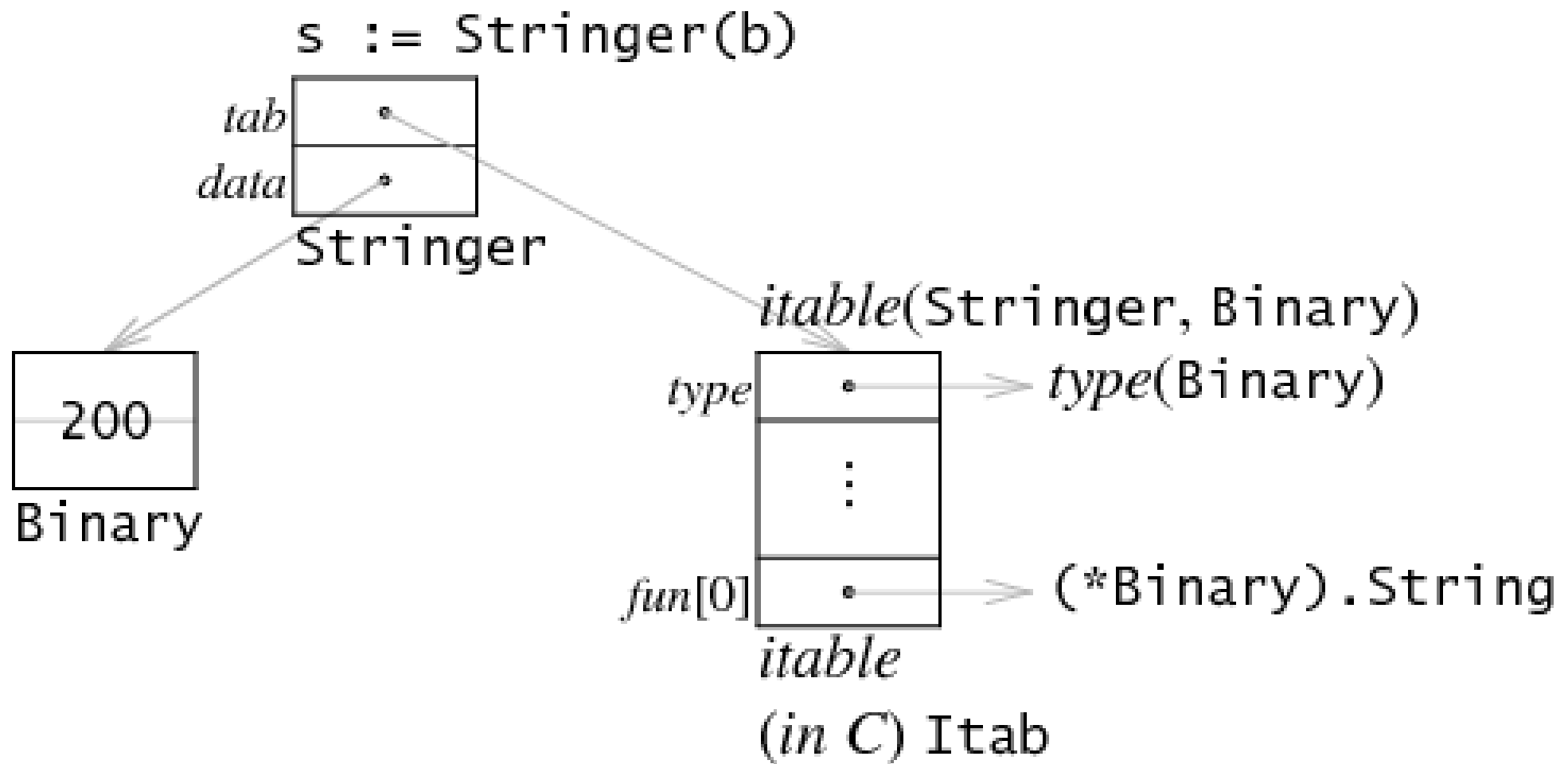
Однако мы можем присвоить переменную типа `T` переменной типа интерфейс `I`, если тип `T` реализует методы `I`.

```
var fh *os.File
fh, _ = os.Open("my.txt")
var rw io.ReadWriter = fh
var r io.Reader = rw
var any interface{} = r
any = []int{0}
```

Для переменной `r`:

- статический тип — `io.Reader`
- динамический тип — `*os.File`

# Рефлексия на уровне языка: интерфейсы



# Рефлексия на уровне языка: type assertion/switch

На уровень языка вынесены следующие возможности рефлексии.

## type assertion

```
var r io.Reader
var f *os.File

f = r.(*os.File) // ?
```

## type switch

```
switch v := i.(type) {
case int:      // here v has type int
    i = v + 1
case string:   // here v has type string
    i = v + "1"
default:      // no match; here v has the same type as i
}
```



# Пакет reflect

Пакет `reflect` в Go представляет API для работы с переменными заранее неизвестных типов.

1. Reflection goes from interface value to reflection object.
2. Reflection goes from reflection object to interface value.
3. To modify a reflection object, the value must be settable.

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

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

# reflect.Value

Значения типа `reflect.Value` представляют собой программную обертку над значением произвольной переменной.

```
i := 42
iv := reflect.ValueOf(i) // тип reflect.Value
```

Какие методы есть у `reflect.Value` ?

```
value.Type() reflect.Type      // вернуть тип обертку над типом
value.Kind() reflect.Kind      // вернуть "базовый" тип

value.Interface() interface{} // вернуть обернутое значение как interface{}
value.Int() int64              // вернуть значение как int64
value.String() string          // вернуть значение как string

value.CanSet() bool            // возможно ли изменить значение ?
value.SetInt(int64)            // установить значение типа int64
value.Elem() reflect.Value     // разыменованье указателя или интерфейса
value.Indirect() reflect.Value // если указатель, то разыменовать
```

Пример: `assert_string`

# reflect.Type и reflect.Kind

`reflect.Kind` представляет собой базовый тип для значения. `reflect.Kind` определяет какие методы имеют смысл для конкретного `reflect.Value`, а какие вызовут панику.

```
const (  
    Invalid Kind = iota  
    Bool  
    Int  
    Int8  
    Int16  
    ...
```

`reflect.Type` представляет собой информацию о конкретном типе: имя, пакет, список методов и т.д.

```
t.Name() string           // имя типа  
t.PkgPath() string        // пакет, в котором определен тип  
t.Size() uintptr          // размер в памяти, занимаемый значением  
t.Implements(u Type) bool // реализует ли интерфейс u?  
t.MethodByName(string name) reflect.Value // метод по имени
```

Большинство методов продублировано в `reflect.Value`

# Пример: изменение значений

Неправильный способ:

```
var x float64 = 3.4
v := reflect.ValueOf(x) // ???
v.SetFloat(7.1) // panic: reflect.Value.SetFloat using unaddressable value
fmt.Println(v.CanSet()) // false
```

Правильный способ:

```
var x float64 = 3.4
p := reflect.ValueOf(&x) // адрес переменной x
fmt.Println(p.Type())    // *float64
fmt.Println(p.CanSet())  // false

v := p.Elem()            // переход по указателю
fmt.Println(v.Type())    // float64
fmt.Println(v.CanSet())  // true
v.SetFloat(7.1)
fmt.Println(x)           // 7.1
```

`(reflect.Value).Elem()` — переходит по указателю или к базовому объекту интерфейса.



# Работа со структурами

Если `v` это рефлексия значения структуры (`reflect.Value`), то

```
v.NumField() int           // возвращает кол-во полей в структуре
v.Field(i int) reflect.Value // возвращает рефлексия для отдельного поля
v.FieldByName(s string) reflect.Value // тоже, но по имени поля
```

Если `t` это рефлексия типа структуры (`t := v.Type()`), то

```
t.NumField() int           // возвращает кол-во полей в структуре
t.Field(i int) reflect.StructField // возвращает рефлексия для конкретного поля
t.FieldByName(name string) (reflect.StructField, bool) // тоже, но по имени поля
```

Свойства `reflect.StructField`

Name	string	// имя поля
Type	reflect.Type	// рефлексия типа поля
Tag	reflect.StructTag	// описание тэгов конкретного поля
Offset	uintptr	// смещение в структуре

Примеры: `struct_to_map`, `map_to_struct`

# Работа с функциями и методами

Получив рефлексию функции/метода, мы можем исследовать количество и типы ее аргументов:

```
f := fmt.Printf
v := reflect.ValueOf(f)
t := v.Type()

t.NumIn()    // количество аргументов
t.NumOut()   // количество возвращаемых значений
a1t := t.In(0) // reflect.Type первого аргумента
o1t := t.Out(0) // reflect.Type первого возвращаемого значения
t.IsVariadic() // принимает ли функция переменное число аргументов ?
```

А также можем вызвать функцию с помощью

```
(reflect.Value).Call([]reflect.Value) []reflect.Value
```

И создавать функции с помощью <https://golang.org/pkg/reflect/#MakeFunc>

Пример: method\_list

# Указатели в Go

В Go указатели на разные типы не совместимы между собой (т.к. сами являются разными типами)

```
type St struct {  
    a, b int  
}  
  
func main() {  
    var b [8]byte  
    bp := &b  
    var sp *St  
    sp = bp // not possible  
    sp = (*St)(bp) // not possible  
}
```

# unsafe.Pointer

Однако тип `unsafe.Pointer` является исключением. Компилятор Go позволяет делать явное преобразование типа любого указателя в `unsafe.Pointer` и обратно (а также в `uintptr`).

```
type St struct {  
    a, b int  
}  
  
func main() {  
    var b [8]byte  
    up := unsafe.Pointer(&b)  
    sp := (*St)(up)  
    sp.a = 12345678  
    fmt.Println(b) // [78 97 188 0 0 0 0 0]  
}
```

<https://goplay.tools/snippet/hpdw55xxlXO>

<https://go101.org/article/unsafe.html>



# Зачем нужен пакет unsafe?

# Зачем нужен пакет unsafe?

В первую очередь он используется в самом Go, например в пакетах `runtime` и `reflect`

```
// src/reflect/value.go
type Value struct {
    typ *rtype
    ptr unsafe.Pointer
    ...
}

func (v Value) SetFloat(x float64) {
    v.mustBeAssignable()
    switch k := v.kind(); k {
    default:
        panic(&ValueError{"reflect.Value.SetFloat", v.kind()})
    case Float32:
        *(*float32)(v.ptr) = float32(x)
    case Float64:
        *(*float64)(v.ptr) = x
    }
}
```

# Минусы рефлексии

# Минусы рефлексии

- Более чувствительный код: узнаем об ошибке не во время компиляции, а только в рантайме.
- В коде нет информации о типах: статические анализаторы не применимы.
- Нужна дополнительная документация (слишком много `interface{}` , не всегда понятно, что именно там ждут).
- Скорость (с небольшой оговоркой).

# Домашнее задание

[https://github.com/OtusGolang/home\\_work/tree/master/hw09\\_struct\\_validator](https://github.com/OtusGolang/home_work/tree/master/hw09_struct_validator)

# Примеры с занятия

[https://github.com/OtusGolang/webinars\\_practical\\_part/tree/master/20-reflection](https://github.com/OtusGolang/webinars_practical_part/tree/master/20-reflection)

# Дополнительные материалы

- Про рефлексю от Роба Пайка: <https://blog.golang.org/laws-of-reflection>
- Про unsafe: <https://go101.org/article/unsafe.html>
- Про интерфейсы от Рассы Кокса: <https://research.swtch.com/interfaces>
- Про интерфейсы от Ивана Данилюка: <https://habr.com/ru/post/276981/>
- Полезные примеры на рефлексю (с пояснениями): <https://github.com/a8m/reflect-examples>

# Следующее занятие

Кодогенерация



# Опрос

Заполните пожалуйста опрос

Ссылка в чате

