

GO-07 01: Кодогенерация. Рефлексия

Куплено
благодаря
Skladchik.com

Описание:

В этом модуле мы поговорим о генерации кода в go. Go является Тьюринг-полным языком программирования, а одно из свойств полноты по Тьюрингу - это то, что программа может написать другую программу самостоятельно. Например, go test перед запуском сканирует все пакеты, которые ему нужно протестировать, а затем пишет обертку над всем этим (новую программу), компилирует и запускает. Еще один пример кодогенерации - это протокол protobuf и генерация клиента и сервера на основе proto схемы. Генерация кода - это огромная часть работы любого компилятора языка, ведь компилятор, анализируя грамматику языка, генерирует новый код (будь то промежуточное представление или непосредственно байт-код). Go из коробки имеет мощные механизмы для генерации кода, и одним из них является рефлексия.

Рефлексия - это способность программы анализировать свои объекты, менять свою структуру и поведение прямо во время выполнения (в runtime).

Пакет reflect

Пакет reflect представляет продвинутое API для работы с переменными заранее неизвестных типов. Его две главные сущности, с которыми пакет работает, это:

- reflect.Type, которая является оберткой с методами над типом произвольной переменной. Для того чтобы начать работать с типом, достаточно поместить ваше неопределенное значение в метод reflect.TypeOf.

```
var v interface{} = "value"  
valueType := reflect.TypeOf(v)  
valueType.Name() // string
```

- reflect.Value, которая является оберткой с методами над значением произвольной переменной. Для того чтобы начать работать с типом, достаточно поместить ваше неопределенное значение в метод reflect.ValueOf.

```
var v interface{} = "value"  
value := reflect.ValueOf(v)  
value.Kind() // string
```

Однако у сущности reflect.Value есть метод Type(), который позволяет в любой момент времени вытащить информацию о типе переменной, поэтому саму сущность reflect.Type используют нечасто.

```
var v interface{} = "value"  
reflectValue := reflect.ValueOf(v)  
valueType := reflectValue.Type()  
valueType.Name() // string
```

Также важно знать, что `reflect.ValueOf` и `reflect.TypeOf` не заимствуют значение, которое им передают, а создают новую копию этого значения, но с более подробной информацией о нем.

Давайте посмотрим еще на один пример.

```
func stringAssertion(v interface{}) (string, error) {
    reflectValue := reflect.ValueOf(v)
    if reflectValue.Kind() == reflect.String {
        return reflectValue.String(), nil
    }

    return "", errors.New("value is not string")
}
```

У нас есть функция `stringAssertion`, в которой мы работаем со значением, тип которого мы заранее не знаем, и нам это значение нужно попытаться привести к строке. При помощи метода `Kind()` мы можем получить базовый тип переданного нам значения и сравнить с заранее имеющимися в пакете `reflect` константами для каждого типа. Информация о базовом типе хранится в виде числа `uint`, а все константы базовых типов представляют из себя `iota` перечисление. Вот вырезка из реализации пакета `reflect`:

```
// The zero Kind is not a valid kind.
```

```
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
)
```

Также пакет `reflect` позволяет работать со структурами, вытаскивая любую информацию о них, такую как:

- имя поля;
- тип поля;
- тег поля (например, `json tag json:"tag_name"`);
- значение поля;
- и т.д.

```
type User struct {
    Username string `json:"name"`
    Age      uint   `json:"age"`
}

func main() {
    user := User{"ivan", 25}
    reflectTypeUser := reflect.TypeOf(user)
    field := reflectTypeUser.Field(0)
    fmt.Println(field.Name, field.Type.Name(), field.Tag)
}
```

Важно рассказать еще об одном свойстве рефлект объектов - устанавливаемость.

Давайте взглянем на пример:

```
var x float64 = 2.9
v := reflect.ValueOf(x)
v.SetFloat(0.8) // Error
```

Казалось бы, почему этот код должен падать с ошибкой? Ответ здесь кроется в том, что `reflect.ValueOf` содержит в себе копию `x`, а не само его значение. Чтобы вдруг не упасть с ошибкой, можно заранее проверить, является ли объект устанавливаемым через метод `CanSet`:

```
var x float64 = 2.9
```

```
v := reflect.ValueOf(x)
v.CanSet() // false
```

Иначе будет выброшена паника.

Для того чтобы сделать значение устанавливаемым, достаточно вызвать метод Elem, который возьмет значение через unsafe указатель, и теперь мы можем смело менять нашу переменную.

```
var x float64 = 2.9
v := reflect.ValueOf(&x)
e := v.Elem()
e.CanSet() // true
e.SetFloat(1.1)
x // 1.1
```

Полезные ссылки:

- [The Laws of Reflection](#)
- [Рефлексия в Go](#)
- [Законы рефлексии в Go](#)

Задание:

В этом задании нам будет необходимо написать две функции с использованием рефлексии.

1. Функция StructToMap, которая из структуры делает мапу вида map[string]interface{}:

```
// Интерфейс функции
func StructToMap(item interface{}) map[string]interface{}
```

2. Функция MapToStruct, которая при помощи мапы и указателя на структуру заполняет эту структуру данными из мапы (у вас получится что-то наподобие json.Unmarshal, но попроще):

```
// Интерфейс функции
func MapToStruct(mp map[string]interface{}, item interface{}) error
```

Условия

- Все тесты из пакета module07/internal/convertor должны успешно проходить.
- Функции должны уметь работать с вложенными структурами.
- Функции должны уметь работать с тегом keyname:"tag_name". Если тег у структуры есть, тогда ключами в мапе должны быть именно теги в обеих функциях, если тег отсутствует, тогда ключами будут являться имена полей структуры.

- Функции должны работать в обе стороны, то есть, если мы через функцию StructToMap сгенерировали карту, тогда через функцию MapToStruct, передав туда сгенерированную карту, должны получить исходную структуру и наоборот (это все проверяется в тестах).

Порядок действий:

1. Сделайте форк проекта [module07](#) в группу `golang_users_repos/<your_gitlab_id>`.
2. Создайте в вашем проекте `module07` новую ветку `module07_01`.
3. В пакете `module07/internal/convertor` заполните логикой функции `StructToMap` и `MapToStruct`.
4. Проверьте, что все тесты проходят успешно (тесты можно запустить при помощи команды `make test_01`).
5. В качестве ответа пришлите ссылку на `merge request` в ветку `master` вашего проекта ветки `module07_01`.