



Объектная модель в Go

Tinkoff.ru





- Errcheck – проверяет, что мы не забыли обработать все ошибки
- Govet – проверяет код на подозрительные конструкции: Printf на соответствие аргументов и управляющей строки
- Gocyclo – проверяет на цикломатическую сложность
- Goconst – ищет повторяющиеся строки и предлагает заменить их на константы



План занятия

-
1. Функции
 2. Структуры
 3. Value vs Pointer semantics
 4. Полиморфизм в Go
 5. Обработка ошибок



ФУНКЦИИ



Синтаксис

Функция — это фрагмент программного кода, к которому можно обратиться из другого места программы.

```
1 // Функция без аргументов
2 func PrintHelloWorld() {
3     fmt.Println("Hello world")
4 }
5
6 // Функция с одним параметром без возвращаемых значений
7 func PrintHello(name string) {
8     fmt.Printf("Hello %s\n", name)
9 }
10
11 // Функция с одним принимаемым и одним возвращаемым значением
12 func MakeGreeting(name string) string {
13     return fmt.Sprintf("Hello %s. Nice to meet you!", name)
14 }
15
```



Синтаксис функций - 2

```
16 // Функция с несколькими аргументами, повторяющимися по типу
17 func RegisterUser(firstName, lastName string, age int) {
18     fmt.Printf("Register user %s %s, age=%d\n", firstName, lastName, age)
19 }
20
21 // Функция с переменным количеством аргументов
22 func PrintStudentRatings(firstName, lastName string, ratings ...int) {
23     fmt.Printf("%s %s ratings: %+v", firstName, lastName, ratings)
24 }
```



Синтаксис функций - 3

```
31 // Функция с несколькими возвращаемыми значениями
32 func MinMax(slice []int) (int, int) {
33     ...
34     return min, max
35 }
36
37 // Функция с именованными возвращаемыми значениями
38 // При return можно не перечислять возвращаемые значения
39 func MinMaxAvg(slice []int) (min, max int, avg float64) {
40     ...
41     return
42     // return min, max, avg // Либо так
43 }
44
45
```



- Простое декларативное имя (Что делает функция)
- Функция должна выполнять **1 действие**
- Следуем принципам Value/Pointer semantics



Конструкция `defer` позволяет сделать гарантированный отложенный вызов функции.

`Defer` позволяет выполнить переданную функцию перед выходом из функции, в которой использовался `defer`.

Несколько `defer`-ов выполняются в обратном порядке



defer

```
1 func readFile(path string) {
2     file, err := os.Open(path)
3     if err != nil {
4         log.Fatal(err)
5     }
6     defer file.Close()
7
8     ...
9 }
10
11
```



defer – 2

```
12 func PrintReverse() {
13     for i := 0; i < 10; i++ {
14         defer func(value int) {
15             fmt.Printf("%d ", value)
16         }(i)
17     }
18     fmt.Println("End PrintReverse")
19 }
20 }
```

```
End PrintReverse
9 8 7 6 5 4 3 2 1 0
```



Паника – это аварийная ситуация, нештатное поведение.
Не путать с ошибкой.

Примеры: выход за границы слайса, ошибка приведения
типа, обращение по nil-адресу

Не паникуем !

1

```
panic("Wow!")
```

Обрабатываем панику



```
1 func panicExample() {
2     fmt.Println("Hello world!")
3
4     defer func() {
5         if r := recover(); r != nil {
6             fmt.Printf("Ops, panic: (%T)%+v\n", r, r)
7         }
8     }()
9
10    panic("Ta-da")
11
12    fmt.Println("Zombie apocalypse has begun!")
13}
14 }
```

```
1: Hello world!
2: Ops, panic: (string)Ta-da
```



Выполняется при первом импорте пакета.

Используется для инициализации переменных, регистрации, проверки состояния программ, одноразовых вычислений.



init

```
1 func init() {
2     Name = "Unknown"
3 }
4
5 var Name = myName()
6
7 func myName() string {
8     return "John"
9 }
10
11 func main() {
12     fmt.Printf("My name is %s\n", Name)
13 }
14 }
```

1: My name is Unknown



ФУНКЦИИ – ЭТО ОБЪЕКТ ПЕРВОГО КЛАССА

Можем использовать функцию как аргумент функции, возвращать из функции функцию, присваивать переменной

```
12 func Filter(nums []int, f func(n int)bool) []int {  
13     result := make([]int, 0, len(nums))  
14     for _, n := range nums {  
15         if f(n) {  
16             result = append(result, n)  
17         }  
18     }  
19  
20     return result  
}
```



Замыкания

```
1 func IsWork(startTime, endTime time.Time) func(now time.Time)bool {
2     return func(now time.Time) bool {
3         if now.After(startTime) && now.Before(endTime) {
4             return true
5         }
6
7         return false
8     }
9 }
10
11 func main() {
12     // ...
13     moex := IsWork(startTime, endTime)
14     if moex(time.Now()) {
15         //...
16
17 }
```

Анонимная функция



```
func Slice(slice interface{}, less func(i, j int) bool)
```

```
1 func main() {
2     //...
3     sort.Slice(candles, func(i, j int) bool {
4         return candles[i].Ts.Before(candles[j].Ts)
5     })
6 }
7
8
9
10
11 }
```



Структуры



Синтаксис

Структура представляет собой агрегированный тип данных, объединяющий нуль или более именованных значений произвольных типов в единое целое

```
1 type Exchange struct {
2     Name string `json:"Exchange"`
3     StartTime time.Time `json:"start_time"`
4     EndTime time.Time `json:"end_time"`
5     stocks []string
6 }
```

Объявление



```
1 type Candle struct {
2     o float64
3     h float64
4     l float64
5     c float64
6     name string
7 }
8
9 func main() {
10    apl := Candle{o:63.2, h:64.6, l:61.2, c:61.2, name:"AAPL"}
11    var sber Candle
12    sber.name = "SBER"
13    fmt.Printf("%v\n%+v", apl, sber)
14
15 }
```

```
1: {63.2 64.6 61.2 61.2 AAPL}
{o:0 h:0 l:0 c:0 name:SBER}
```



Сравнение структур

Структуры можно сравнивать, если все поля структуры можно сравнить

```
1 func main() {
2     c1 := Candle{o: 63.2, h: 64.6, l: 61.2, c: 61.2, name: "AAPL"}
3     c2 := c1
4     if c1 == c2 {
5         c2.name = "SBER"
6         fmt.Printf("C1 : %+v\nC2 : %+v", c1, c2)
7     }
8 }
```

```
1: C1 : {o:63.2 h:64.6 l:61.2 c:61.2 name:APPL}
   C2 : {o:63.2 h:64.6 l:61.2 c:61.2 name:SBER}
```



Пустая структура

```
1 func main() {
2     size := unsafe.Sizeof([10000]struct{}{})
3     fmt.Printf("%d", size)
4 }
```

```
5
6
7
```

```
1: 0
```



1. По значению (память выделяется на стеке, структура полностью копируется)
2. По ссылке (память выделяется на стеке либо в куче, копируется только указатель)



Передача аргументов в функцию

```
1 func updatePriceByValue(c Candle) {
2     c.h += 10
3 }
4
5 func updatePriceByRefernce(c *Candle) {
6     c.h += 10
7 }
8
9 func main() {
10    c := Candle{h:10}
11    updatePriceByValue(c)
12    fmt.Println(c.h)
14    updatePriceByRefernce(&c)
15    fmt.Println(c.h)
16 }
```

```
1: 10
2: 20
```



Методы

Метод объявляется с помощью вариации объявления обычных функций, в котором перед именем функции появляется дополнительный параметр – receiver

```
func (t Time) Add(d Duration) Time
```

```
1 type Exchange struct {
2     Name string `json:"Exchange"`
3     StartTime time.Time `json:"start_time"`
4     EndTime time.Time `json:"end_time"`
5     stocks []string
6 }
7
8
9 func (e *Exchange) updateStartTime(d time.Duration) {
10     e.StartTime = e.StartTime.Add(d)
11 }
```



Конструкторы

```
1 const layout = "15:04"
2 func New() (Exchange, error) {
3     startTime, err := time.Parse(layout, "10:00")
4     if err != nil {
5         return Exchange{}, err
6     }
7     endTime, err := time.Parse(layout, "18:00")
8     if err != nil {
9         return Exchange{}, err
10    }
11    return Exchange{
12        Name:      "NASDAQ",
13        StartTime: startTime,
14        EndTime:   endTime,
15        stocks:   []string{ "AAPL", "AMZN" },
16        }, nil
17 }
```



Встраивание (embedding)

В Go нет наследования, но есть встраивание

Возможность использовать **поля и методы** другой структуры
через анонимное поле



Встраивание (embedding)

```
1 func (u *User) CreatePost(content string) error { ...}
2
3 type Admin struct {
4     User
5     permissions map[string]bool
6 }
7
8 func main() {
9     ...
10    err := admin.CreatePost("new post") // admin.User.CreatePost("new
11 post")
12    if err != nil {
13        log.Fatal(err)
14    }
15    log.Printf("Post was created by %s", admin.Name )
16 }
17 }
```



Value vs Pointer semantics



Value semantics

- Встроенные типы

```
func Replace(s, old, new string, n int) string
func LastIndex(s, sep string) int
func ContainsRune(s string, r rune) bool
```

- Ссылочные типы (map, slice, interface, func, chan) – эти типы внутри себя уже содержат указатель



Value semantics

```
1 type IP []byte
2 type IPMask []byte
3
4 func (ip IP) Mask(mask IPMask) IP {
5     if len(mask) == IPv6len && len(ip) == IPv4len && allFF(mask[:12]) {
6         mask = mask[12:]
7     }
8     if len(mask) == IPv4len && len(ip) == IPv6len && bytesEqual(ip[:12], v4InV6Prefix) {
9         ip = ip[12:]
10    }
11    n := len(ip)
12    if n != len(mask) {
13        return nil
14    }
15    out := make(IP, n)
16    for i := 0; i < n; i++ {
17        out[i] = ip[i] & mask[i]
18    }
19    return out
20 }
```



Пользовательские типы

- если у структуры высокая степень вложенности
- если не уверены на 100%, что структуры
можно постоянно копировать (файлы)



Pointer semantics

```
1 type CustomTs struct {
2     time.Time
3 }
4
5 func (c CustomTs) UnmarshalJSON(data []byte) error {
6     timeString := strings.Trim(string(data), "\\"")
7     t, err := time.Parse ("15:04", timeString)
8     if err != nil {
9         return err
10    }
11    c.Time = t
12    return nil
13 }
14
15 type Quote struct {
16     Ask float64
17     Bid float64
18     Ts CustomTs
19 }
```



Pointer semantics

```
1 func main() {
2     data := []byte(`{"Ask":64.2, "Bid":66.6, "Ts":"18:59"}`)
3     var q Quote
4     if err := json.Unmarshal(data, &q); err != nil {
5         log.Fatal(err)
6     }
7
8     log.Printf("%+v", q)
9 }
```

```
1: {Ask:64.2 Bid:66.6 Ts:0001-01-01 00:00:00 +0000 UTC}
```



Полиморфизъм



Полиморфизм

Возможность писать код, который будет менять свое поведение в зависимости от типа

Go поддерживает концепцию «утиной» типизации («Если это выглядит как утка, плавает как утка и крякает как утка, то это, вероятно, и есть утка.») => Если структура реализует все методы интерфейса, то она удовлетворяет ему



Интерфейс

```
1 type CandleFetcher interface {
2     Fetch() ([]Candle, error)
3 }
4
5 type NasdaqFetcher struct {}
6
7 func (n *NasdaqFetcher) Fetch() ([]Candle, error) { ... }
8
9 type MoexFetcher struct {}
10
11 func (m *MoexFetcher) Fetch([]Candle, error) { ... }
12
13
14
15
```

Интерфейс



```
1 func Candles(fetchers []CandleFetcher) ([]Candle, error) {
2     ...
3     for i := range fetchers {
4         candles, err := fetchers[i].Fetch()
5     }
6     ...
7 }
8 }
```

Интерфейс



```
1 type FeedAPIService struct {
2     ...
3     storage db.Postgres
4     ...
5 }
6
7 // как тестировать ? Что делать, если захотим изменить БД ?
8 func (f *FeedAPIService) FindFeed(params api.Params) ([]Item, error) {
9     ...
10    items, err := storage.Find(params)
11    ...
12 }
```



Интерфейс

```
1 type FeedStorage interface {
2     Find (params api.Params) ([]Item, error)
3 }
4
5 type FeedAPIService struct {
6     ...
7     storage FeedStorage
8     ...
9 }
10
11
12
```



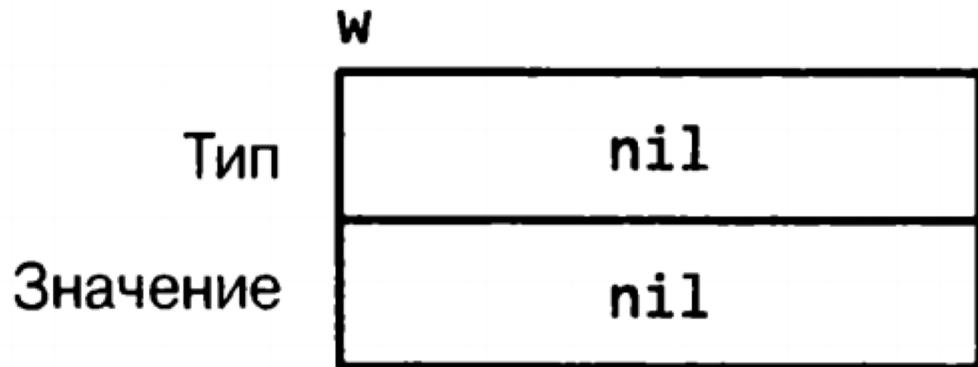
Конвенция имен

```
1 type Stringer interface {
2     String() string
3 }
4
5 type Reader interface {
6     Read(p []byte) (n int, err error)
7 }
8
9 type Writer interface {
10    Write(p []byte) (n int, err error)
11 }
12
13 type Closer interface {
14     Close() error
15 }
```



Устройство

```
1 var w io.Writer
```





Типичная ошибка

```
1 const debug = true
2
3 func main() {
4     var buf *bytes.Buffer
5     if debug {
6         buf = new(bytes.Buffer)
7     }
8     f(buf)
9     if debug {
10
11 }
12 }
13
15 func f(out io.Writer) {
16     if out != nil {
17         out.Write([]byte("выполнено!\n"))
18     }
19 }
```



Запомнить

Каждый тип ассоциирован со своим множеством методов

Methods	Receivers	Values
(t T)		T and *T
(t *T)		*T



Пустой интерфейс

Это интерфейс, который не содержит методов.

Является попыткой реализовать динамическую типизацию в языке со строгой статической типизацией.

Любой объект можно привести к пустому интерфейсу.



Пустой интерфейс

```
1 func main() {
2     var variable interface{}
3     variable = []int{1, 2, 3}
4     fmt.Printf("%+v\n", variable)
5
6     variable = "Hello world"
7     fmt.Printf("%+v\n", variable)
8
9     variable = func() int {
10         return 100
11     }
12     fmt.Printf("%+v\n", variable)
13 }
14 }
```

```
1: [1 2 3]
2: Hello world
3: 0xe2260
```

Приведение типов для структур и интерфейсов



```
1 var empty interface{}
2 user := User{Name: "John"}
3
4 empty = user
5 stringer := empty.(fmt.Stringer)
6 fmt.Printf("%s\n", stringer)
7
8 writer, ok := empty.(io.Writer)
9 if !ok {
10     fmt.Println("Can't cast empty struct to io.Writer")
11 }
12
13 switch empty.(type) {
14 case fmt.Stringer:
15     fmt.Printf("Cast empty to fmt.Stringer: %s\n", empty.(fmt.Stringer))
16 }
17 }
```



Проверить на соответствие интерфейсу можно и на этапе компиляции

```
1 var _ FeedStorage = &Postgres{ }  
2  
3  
4  
5  
6  
7
```



- По возможности избегайте работы с пустым интерфейсом.
- Всегда делайте безопасное приведение типов.



- Композиция является более гибким инструментом расширения проекта
- Композиция позволяет динамически изменять поведение объекта



Errors



```
1 // The error built-in interface type is the conventional interface for
2 // representing an error condition, with the nil value representing no
3 // error.
4 type error interface {
5     Error() string
6 }
7 }
```

- Ошибка – это интерфейс с методом `Error()`.
- **Соглашение:** если функция возвращает объект и ошибку, то если `err == nil`, то объект `!= nil`, и наоборот.
- Р.С. Исключение – слайсы. Из функции можно вернуть `nil`, `nil`

Error



```
1 // https://github.com/golang/go/blob/master/src/errors/errors.go
2 // Package errors implements functions to manipulate errors.
3 package errors
4
5 // New returns an error that formats as the given text.
6 func New(text string) error {
7     return &errorString{text}
8 }
9
10 // errorString is a trivial implementation of error.
11 type errorString struct {
12     s string
13 }
14
15
16 func (e *errorString) Error() string {
17     return e.s
18 }
```

Обработка ошибок – 1



Все ошибки обрабатываем явно. Никаких checkError()

```
1  finderFunc, err := s.makeFinderFunc(filter)
2  if err != nil {
3      return nil, nil, errors.Wrap(err, "can't make finder func")
4  }
5
6  feeds, cursor, err := s.findByPage(filter, userLimit, finderFunc)
7  if err != nil {
8      return nil, nil, errors.Wrap(err, "can't find filtered")
9  }
10
11 if err = s.setIsMarked(feeds, filter); err != nil {
12     return nil, nil, errors.Wrap(err, "can't set IsMarked by filter")
13 }
14 }
```

Обработка ошибок – 2



Чем плох это фрагмент кода?

```
1 func doSomething() error {
2     ...
3     user, err := findUser(userID)
4     if err != nil {
5         return err
6     }
7     ...
8 }
```

Недостаток: в логах нет контекста о проблеме, по логу
нельзя понять последовательность вызова.



Обработка ошибок – 2

Правильно: добавлять дополнительную информацию

```
1 func doSomething() {
2     ...
3     user, err := findUser(userID)
4     if err != nil {
5         // Простой вариант
6         // return fmt.Errorf("can't find user by id %d: %s", userID, err)
7         // Вариант с использование библиотеки pkg/errors
8         return errors.Wrapf(err, "can't find user by id %d", userID)
9     }
10    ...
11 }
```

В Тинькофф мы используем <https://github.com/pkg/errors>



- Unwrap(err error)error
- Is(err, target error) bool
- As(err error, target interface{})bool



Обратная связь

Tinkoff.ru



Спасибо за внимание

Tinkoff.ru