GeekBrains

Go. Уровень 1

Сложные типы данных: структуры, функции и методы

На этом уроке

1. Научимся работать со структурами.
2. Научимся писать собственные функции и методы, понимать разницу между ними.

Оглавление

Структуры

- Введение
- Создание структуры
  - Объявление типа
  - Продолжение создания структуры
- Композиция

Функции

- Объявление функций
  - Вариативные функции (variadic)
- Функции как значения первого класса (first class value)
- Анонимные функции и замыкания
  - Анонимные функции
  - Замыкания (Closure)
- Отложенное исполнение функций (defer)
  - Особенности работы
  - Частая ошибка с замыканиями и циклами
  - Трюк с defer и именованным возвращаемым значением
- Методы

Заключение

Практическое задание

Дополнительные материалы

На первом уроке мы говорили о прекрасном языке Assembly, который также называют языком ассемблера или просто ASM. Он позволял писать программы так, чтобы компьютеру было удобно их понимать, то есть вы буквально пишете последовательность команд процессора. Так неудобно писать большие программы. Хочется, чтобы программа была приближена к задаче, которую она решает, а не к деталям работы процессора.

На этом уроке поговорим о важных концепциях программирования, появившихся в 70-х годах прошлого столетия, которые сделали программирование более удобным для человека и позволили писать большие программы.

Структуры

Введение

В большинстве приложений есть потребность в хранении сгруппированных логически данных. Самый банальный пример — точка с двумя координатами. Ничто не мешает нам использовать для хранения точки два числа.

```
package main

import (
	"fmt"
)

func main() {
	p1X := float64(1)
	p1Y := float64(2)

	fmt.Println("p1 x:", p1X, "y:", p1Y)
}
```

```
> go run app.go
p1 x: 1 y: 2
```

Отлично, тут вроде ничего сложного. Теперь создадим вторую точку и найдём расстояние между ними:

```
package main

import (
	"fmt"
	"math"
)
```

```
func main() {
	p1X := float64(1)
	p1Y := float64(0)

	p2X := float64(-1)
	p2Y := float64(0)

	// math.Pow(x, y) возводит x в степень y
	// math.Sqrt(x) возвращает квадратный корень из x
	dist := math.Sqrt(math.Pow(p1X-p2X, 2) + math.Pow(p1Y-p2Y, 2))

	fmt.Println("dist:", dist)
}
```

```
> go run app.go
dist: 2
```

Это выглядит громоздко и неудобно для чтения. К тому же при увеличении количества точек будет возрастать количество переменных, но мы рассматриваем точки как что-то целостное, а не как две отдельные переменные. На прошлом уроке мы научились работать с мапами. Кажется, они могут помочь.

```
package main

import (
	"fmt"
	"math"
)

func main() {
	p1 := map[string]float64{"x": 1, "y": 0}
	p2 := map[string]float64{"x": -1, "y": 0}
	dist := math.Sqrt(
		math.Pow(p1["x"]-p2["x"], 2) + math.Pow(p1["y"]-p2["y"], 2),
	)
	fmt.Println("dist:", dist)
}
```

```
> go run app.go
dist: 2
```

Уже значительно лучше. Однако мы помним о проблемах мап: их сложно копировать и можно банально описаться в ключах. Кроме того, мапы занимают достаточно много памяти, значительно больше, чем две переменные.

Для решения этих проблем уже в конце 60-х годов прошлого века были придуманы структуры.

Структура — это композитный тип данных, то есть тип данных, который состоит из других типов данных.

Создание структуры

Создадим структуру для точки:

```
package main

import (
	"fmt"
	"math"
)

func main() {
	p1 := struct{ x, y float64 }{1, 0}
	p2 := struct{ x, y float64 }{-1, 0}

	dist := math.Sqrt(math.Pow(p1.x-p2.x, 2) + math.Pow(p1.y-p2.y, 2))

	fmt.Println("dist:", dist)
}
```

```
> go run app.go
dist: 2
```

Этот пример должен продемонстрировать, что для объявления структуры можно использовать анонимный тип. На практике для структур обычно создают именованный тип, чтобы повысить удобство, читаемость и поддерживаемость кода.

Объявление типа

Для объявления типа используется ключевое слово `type`:

```
type Point struct {
	x, y float64
}
```

Итак, мы создали тип `Point` — структуру с **полями** `x` и `y`.

Общая схема объявления нового типа:

```
type NewTypeName BasicType
```

В теории можно объявлять свои типы для проверки логики программы, и в ряде случаев это хорошая практика. Мы рассмотрим такие случаи в дальнейших уроках. Забегая вперёд, скажем, что примером здесь может быть работа с контекстом для передачи данных.

```
type ErrorCode uint32

const (
      ExternalError ErrorCode = iota + 1
      InternalError
      InvalidDataError
)
```

Примечание: структура — логическая композиция, то есть в бинарном исполняемом файле не существует типа данных «структура», она будет «заменена» на эквивалентное количество переменных. Для нас это означает, что структура, состоящая из двух полей, будет занимать столько же памяти, как две переменные этих типов.

Эта особенность позволяет создать **пустую структуру** `struct{}`, которая не будет занимать память вообще. Её можно использовать, например, для создания структуры данных «**множество**» из мапы (`map[string]struct{}`), если важна только уникальность значения.

Продолжение создания структуры

Можно использовать именованную инициализацию:

```
package main

import (
      "fmt"
      "math"
)

type Point struct {
      x, y float64
}

func main() {
      p1 := Point{x: 1, y: 0}
      p2 := Point{x: -1, y: 0}

      dist := math.Sqrt(math.Pow(p1.x-p2.x, 2) + math.Pow(p1.y-p2.y, 2))

      fmt.Println("dist:", dist)
}
```

При именованной инициализации можно инициализировать не все поля, тогда неинициализированные поля будут иметь значение типа по умолчанию:

```
package main

import (
	"fmt"
)

type Point struct {
	x, y float64
}

func main() {
	p1 := Point{x: 1}

	fmt.Println("p1:", p1)
}
```

```
> go run app.go
p1: {1 0}
```

Доступна также пустая инициализация. Все поля будут иметь значение по умолчанию:

```
package main

import (
	"fmt"
)

type Point struct {
	x, y float64
}

func main() {
	p1 := Point{}
	fmt.Println("p1:", p1)
}
```

```
> go run app.go
p1: {0 0}
```

Важно! *Структуры копируются при присваивании, как и другие примитивные типы.*

```
package main

import (
	"fmt"
)

type Point struct {
	x, y float64
}

func main() {
	point := Point{}

	pointCopy := point

	point.x = 1

	fmt.Println("point:", point)
	fmt.Println("pointCopy:", pointCopy)
}
```

```
> go run app.go
point: {1 0}
pointCopy: {0 0}
```

Стоит помнить, что при копировании слайса или мапы копируется только обёртка, а нижележащие структуры не копируются.

```
package main

import (
	"fmt"
)

type StructWithSlice struct {
	number int32
	slice  []int32
}

func main() {
	structWithSlice := StructWithSlice{
		number: 123,
		slice:  []int32{1, 2, 3},
	}

	structWithSliceCopy := structWithSlice

	structWithSlice.number = 321
```

```
	structWithSlice.slice[1] = 100

	fmt.Println("structWithSlice:", structWithSlice)
	fmt.Println("structWithSliceCopy:", structWithSliceCopy)
}
```

```
> go run app.go
structWithSlice: {321 [1 100 3]}
structWithSliceCopy: {123 [1 100 3]}
```

Важно! *Для именования полей структуры действуют такие же правила, как для именования переменных и функций. Название поля не может начинаться с цифры. Если название начинается с заглавной буквы, то поле называется экспортируемым и его можно использовать в других пакетах.*

То же самое справедливо для именования типов.

Композиция

Для полей структуры есть только одно ограничение: нельзя конструировать цепочки типов, которые создают рекурсивные типы. Компилятор начинает думать, что вы хотите создать структуру бесконечного размера.

```
package main

import (
	"fmt"
)

type ExampleStruct struct {
	internal ExampleStruct
}

func main() {
	fmt.Println()
}
```

```
> go run app.go
./prog.go:7:6: invalid recursive type ExampleStruct
```

В остальном можно компоновать структуру как угодно. Например, создадим структуру для круга:

```go
package main

import (
   "fmt"
)

type Point struct {
   X, Y float64
}

type Circle struct {
   Center Point
   Radius float64
}

func main() {
   circle := Circle{
       Center: Point{1, 2},
       Radius: 1,
   }

   // Форматированный вывод
   // Документация https://golang.org/pkg/fmt/#hdr-Printing
   fmt.Printf("circle: %+v\n", circle)
}
```

```
> go run app.go
circle: {Center:{X:1 Y:2} Radius:1}
```

Функции

Мы уже говорили о функциях на первом уроке и постоянно использовали их. Сейчас настало время изучить их подробнее.

Объявление функций

Напомним, что функция объявляется с помощью ключевого слова `func`:

```go
func someFunction(arg1 argType1, arg2 argType2) resultType {
    // function body
}
```

Последовательность, состоящая из названия функции с типами аргументов и результата, называется **сигнатурой функции**.

Функция может ничего не возвращать:

```
func someFunction(arg1 argType1, arg2 argType2) {
    // function body
}
```

Или ничего не принимать:

```
func someFunction() resultType {
    // function body
}
```

Выход из тела функции и возврат результата можно произвести с помощью ключевого слова `return`:

```
func sum(a int32, b int32) int32 {
    return a + b
}
```

Вызывать функцию можно таким образом:

```
result := someFunction(arg1, arg2)
```

Однако в одном из примеров у нас был вот такой вызов:

```
num, err := strconv.ParseInt(line, 10, 64)
```

Очевидно, в этом случае **функция вернула два результата.**

Мы подбираемся к особенностям функций в Go. В Go функция может вернуть любое количество значений:

```
func sumSub(a int32, b int32) (int32, int32) {
    return a + b, a - b
}
```

Более того, результат функции может быть именованным. В этом случае Go создаст переменные под результирующие значения:

```
func sumSub(a, b int32) (sum, subtract int32) {
    sum = a + b
    subtract = a - b
    return
}
```

Эта возможность иногда используется, чтобы прояснить, что возвращает функция. Но чаще всего это применяется для трюков, которые мы рассмотрим чуть дальше в лекции.

Вариативные функции (variadic)

Бывают случаи, когда функция должна принимать неопределённое количество аргументов. Например, функция `fmt.Println` принимает произвольное количество аргументов и печатает их. Объявлять такие функции можно с помощью конструкции `...` .

```
package main

func Println(strs ...string) {
    // strs -- []string
    for i, val := range strs {
        println(i, val)
    }
}

func main() {
      Println("help", "me")

      args := []string{"it's", "ok", "too"}

      Println(args...)  // Можно «распаковать» слайс в variadic-аргументы
}
```

```
> go run app.go
0 help
1 me
0 it's
1 ok
2 too
```

Функции как значения первого класса (first class value)

Функция в Go — это тоже значение, как число, строка, массив и т. д.

Тип функции — это её сигнатура без учёта имени функции и имён её аргументов и возвращаемых значений, то есть тип функции определяется типом её аргументов и типом её возвращаемых значений. Такие **типы** называются **функциональными (function type)**. Значение по умолчанию функциональных типов — `nil`.

Представим ситуацию: в зависимости от пользовательского ввода нам нужно сложить, отнять, умножить или разделить два числа. Это, конечно, можно сделать с помощью `switch`, но иногда удобнее использовать вот такой подход:

```
package main

import (
   "bufio"
```

```
    "fmt"
    "os"
    "strconv"
    "strings"
)

const helpMessage = "command format: 'cmd num1 num2'"

func sum(a, b float64) (float64, error) {
    return a + b, nil
}

func subtract(a, b float64) (float64, error) {
    return a - b, nil
}

func multiply(a, b float64) (float64, error) {
    return a * b, nil
}

func divide(a, b float64) (float64, error) {
    if b == 0 {
        return 0, fmt.Errorf("dividing by zero")
    }
    return a / b, nil
}


// мапа соответствий названия действия функции-обработчику
var cmdMap = map[string]func(float64, float64) (float64, error){
    "sum":      sum,
    "subtract": subtract,
    "multiply": multiply,
    "divide":   divide,
}

func processCmd(cmd string, a, b float64) (float64, error) {
    if cmdFunc, ok := cmdMap[cmd]; ok {
        return cmdFunc(a, b)
    }
    return 0, fmt.Errorf("command not implemented")
}

func main() {
    scanner := bufio.NewScanner(os.Stdin)
    for scanner.Scan() {
        input := scanner.Text()
        tokens := []string{}

        // strings.Split разбивает строку по заданному символу
        for _, token := range strings.Split(input, " ") {
            token = strings.TrimSpace(token) // отрезаем пробелы
            if token != "" {
```

```
                tokens = append(tokens, token)
            }
        }
        if len(tokens) != 3 {
            fmt.Println(helpMessage)
            continue
        }
        cmd := tokens[0]
        num1, err1 := strconv.ParseFloat(tokens[1], 64)
        num2, err2 := strconv.ParseFloat(tokens[2], 64)
        if err1 != nil || err2 != nil {
            fmt.Println(helpMessage)
            continue
        }
        res, err := processCmd(cmd, num1, num2)
        if err != nil {
            fmt.Printf("%s: %v\n", input, err)
            continue
        }
        fmt.Println(res)
    }
}
```

```
> go run app.go
asd
command format: 'cmd num1 num2'
multiply 333 3
999
divide 3 0
divide 3 0: dividing by zero
```

Так как функции — это значения, то можно возвращать и передавать функции в функции:

```
func sum(a, b int32) int32 {
    return a + b
}
func generateSumFunc() func(int32, int32) int32 {
    return sum
}
```

Анонимные функции и замыкания

Анонимные функции

В прикладном программировании иногда требуется написать функцию без названия, чтобы сразу передать её в другую функцию. Такие функции называются анонимными. Для создания таких функций нужно всего лишь пропустить название функции:

```
func generateSumFunc() func(int32, int32) int32 {
   return func(int32, int32) int32 {
      return a + b
   }
}
```

Анонимные функции ничем не отличаются от именованных.

Замыкания (Closure)

Функция может использовать переменные из окружающей её области видимости. Например, рассмотрим функцию `sort.Slice` из пакета sort. Эта функция требует на вход сам слайс и функцию сравнения элементов.

```
package main

import (
   "fmt"
   "sort"
)

func main() {
   slice := []int32{5, 2, 4, 1, 3}

   sort.Slice(slice, func(i, j int) bool {
       // переменная slice используется
       // из внешней области видимости
       return slice[i] < slice[j]
   })

   fmt.Println(slice)
}
```

```
> go run app.go
[1 2 3 4 5]
```

Замыкания часто используют для функций, которые добавляют небольшой слой логики, не изменяя основную логику функции. Такие функции называют middleware или декораторами.

```
package main

import "fmt"

type BinaryI32Op func(a, b int32) int32

func logged2Op(f BinaryI32Op) BinaryI32Op { // это декоратор
	return func(a, b int32) int32 {
		fmt.Println("args:", a, b)
		res := f(a, b)
		fmt.Println("result:", res)
		return res
	}
}

func main() {
	sum := logged2Op(func(a, b int32) int32 {
		return a + b
	})

	sum(1, 2)
	sum(3, 4)
	sum(5, 6)
}
```

```
> go run app.go
args: 1 2
result: 3
args: 3 4
result: 7
args: 5 6
result: 11
```

Заметьте, что функция сложения анонимная и явно нигде не сохраняется в переменную:

- анонимная функция передаётся в функцию `logged`;
- анонимная функция записывается в переменную `f`, но после завершения `logged` все локальные для функции переменные уничтожаются.

Анонимная функция сохраняется внутри замыкания, которое возвращается из функции `logged`. Условно замыкание можно представить как совокупность анонимной функции и списка захваченных переменных из внешней области видимости.

В таком случае говорят, что переменная замыкается в замыкании.

Отложенное исполнение функций (defer)

В некоторых задачах требуется вызвать какую-нибудь функцию-финализатор после завершения определённого блока. Рассмотрим, например, работу с файлом:

```
package main

import (
   "bufio"
   "fmt"
   "os"
   "strconv"
   "strings"
)

func main() {
   f, err := os.Open("some_file")
   if err != nil {
       panic(err)
   }

   s := bufio.NewScanner(f)
   for i := 1; s.Scan(); i++ {
       nums, err := readNums(s.Text())
       if err != nil {
           f.Close() // нужно закрыть файл перед выходом
           panic(err)
       }

       sum, err := sumInverted(nums)
       if err != nil {
           f.Close() // нужно закрыть файл перед выходом
           panic(err)
       }

       fmt.Println(sum)
   }
   f.Close()
}

func readNums(s string) ([]int64, error) {
   // суть функции не важна
   // важно, что она возвращает ошибку
}
func sumInverted(nums []int64) (float64, error) {
   // ...
}
```

В этом примере видно, что нужно закрывать файл перед завершением программы. Иногда этих мест становится настолько много, что в них можно запутаться и забыть закрыть файл. Для таких случаев

предусмотрено отложенное выполнение функции с помощью ключевого слова `defer`. Отложенная функция будет выполнена непосредственно перед выходом из функции, где она была отложена.

Таким образом, функция `main` из примера обретает вид:

```
func main() {
   f, err := os.Open("some_file")
   if err != nil {
       panic(err)
   }
   defer f.Close() // откладываем закрытие файла

   s := bufio.NewScanner(f)
   for i := 1; s.Scan(); i++ {
       nums, err := readNums(s.Text())
       if err != nil {
           panic(err)
       }

       sum, err := sumInverted(nums)
       if err != nil {
           panic(err)
       }

       fmt.Println(sum)
   }
}
```

Особенности работы

Аргументы `defer` вычисляются **сразу же**, а не откладываются.

```
package main

import (
   "fmt"
)

func main() {
   i := 5
   defer fmt.Println("deferred", i)

   i = 10

   fmt.Println(i)
   // вызов отложенной fmt.Println("deferred", i)
   // когда сделали defer, i была равна 5
}
```

```
> go run app.go
10
deferred 5
```

При выходе из функции отложенные функции выполняется в порядке, обратном порядку отложения:

```
package main

import "fmt"

func main() {
      defer fmt.Println("deferred", 1)
      defer fmt.Println("deferred", 2)
      defer fmt.Println("deferred", 3)
}
```

```
> go run app.go
deferred 3
deferred 2
deferred 1
```

Частая ошибка с замыканиями и циклами

Отложенное выполнение часто используют вместе с замыканиями, но в этом случае стоит быть аккуратными.

```
package main

import "fmt"

func main() {
    for i := 0; i < 5; i++ {
        defer func() {
            fmt.Println("deferred", i)
        }()
    }
}
```

Кажется, что по изученным нами правилам вывод должен быть таким:

```
deferred 4
deferred 3
deferred 2
deferred 1
```

```
deferred 0
```

Однако на самом деле вывод будет вот такой:

```
> go run app.go
deferred 5
deferred 5
deferred 5
deferred 5
deferred 5
```

Дело в том, что замыкание захватывает переменные не по значению (то есть не копирует их), а как бы по имени. На последней итерации цикла i будет равна `5`, она не пройдёт проверку `5 < 5`, и цикл завершится. Соответственно, завершится и функция `main`, а значит, выполнятся замыкания, в которых i замкнута по имени.

Чтобы избежать этой ошибки, пользуйтесь тем, что аргументы `defer` вычисляются во время отложения:

```
package main

import "fmt"

func main() {
    for i := 0; i < 5; i++ {
        defer func(i int) {
            fmt.Println("deferred", i)
        }(i)
    }
}
```

```
> go run app.go
deferred 4
deferred 3
deferred 2
deferred 1
deferred 0
```

Немного похожую ошибку с очерёдностью вычисления аргументов можно допустить и при вызове горутин из цикла, но об этом мы поговорим на уроке о горутинах. Пока поэкспериментируйте с тем, как работает замыкание с `defer`. Этот вопрос любят задавать на собеседованиях.

Трюк с defer и именованным возвращаемым значением

Есть интересный трюк, использующий свойства возвращаемых значений и замыканий:

```
package main
```

```
import "fmt"

func calcSomething() (result int64) {
   defer func() {
       fmt.Println("result is", result)
   }()

   return 312324
}

func main() {
   calcSomething()
}
```

```
> go run app.go
result is 312324
```

Несмотря на то, что мы не присваивали явно ничего в переменную `result`, перед вызовом отложенных функций в неё копируется возвращаемое значение. Это можно использовать для финального логирования результата функции.

Методы

Во многих языках программирования есть методы. Метод — это способ задать поведение некоторому абстрактному или конкретному типу данных. То есть методы позволяют определить, как программист будет взаимодействовать с этим типом данных. В Go это сделано с помощью механизма связанных функций (bound functions).

Вспомним структуру точки из начального примера. Хочется привязать к ней логику по вычислению расстояния до другой точки. Сделать это можно с помощью такого синтаксиса.

```
func (selfName boundType) methodName(...) ... {
    // ...
}
```

Сделаем это!

```
package main

import (
   "fmt"
   "math"
)
```

```
type Point struct {
   x, y float64
}

func (p Point) DistanceTo(otherPoint Point) float64 {
   return math.Sqrt(
       math.Pow(p.x-otherPoint.x, 2) + math.Pow(p.y-otherPoint.y, 2),
   )
}

func main() {
   center := Point{x: 0, y: 0}
   p := Point{x: 1, y: 2}

   fmt.Println("dist:", p.DistanceTo(center))
}
```

```
> go run app.go
dist: 2.23606797749979
```

Важно! *Переменная связанного типа — в данном случае `p Point` — ничем не отличается от переменных, передаваемых в функцию, то есть она тоже копируется. Поэтому нельзя изменить переменную с помощью метода, связанного с типом `Point`. Как это исправить, рассмотрим на следующем уроке.*

Примечание: при связывании метода с переменной связанного типа получается обычная функция.

```
package main

import (
   "fmt"
   "math"
)

type Point struct {
   x, y float64
}

func (p Point) DistanceTo(otherPoint Point) float64 {
   return math.Sqrt(
       math.Pow(p.x-otherPoint.x, 2) + math.Pow(p.y-otherPoint.y, 2),
   )
}

func main() {
   center := Point{x: 0, y: 0}
   var distanceToCenter func(Point) float64 = center.DistanceTo

   p := Point{x: 1, y: 2}
```

```
    fmt.Println("dist:", distanceToCenter(p))
}
```

```
> go run app.go
dist: 2.23606797749979
```

Примечание: методы можно объявлять для любого типа внутри пакета. Это значит, что:

1. Методы к одному типу можно объявлять в разных файлах.
2. Методы можно привязывать к примитивным типам вроде `type MyString string`.
3. Нельзя определить собственные методы на функции из сторонних пакетов, например непосредственно на тип `string`.

Заключение

На этом уроке мы познакомились с очень важными для Computer Science концепциями:

1. Структуры (типы-произведения, записи).
2. Анонимные функции и замыкания.
3. Методы.
4. Отложенные функции (финализаторы).

На следующем уроке мы познакомимся с вероятно самой важной концепцией в программировании. Из-за неё мапы и слайсы не копируются, методы не могут изменить структуру и тем, что печатается deferred 5…

Практическое задание

1. Напишите приложение, рекурсивно вычисляющее заданное из стандартного ввода число Фибоначчи.
2. Оптимизируйте приложение за счёт сохранения предыдущих результатов в мапе.
3. Проверьте себя: вам должны быть знакомы следующие ключевые слова Go: `func`, `return`, `defer`, `struct`, `type`.
4. Посмотрите задачи из предыдущих уроков. Как можно улучшить дизайн задач? Что бы вы разбили на отдельные функции или даже пакеты?

Дополнительные материалы

1. Effective Go, главы Initialization, Methods, Functions.
2. Go FAQ: Types.

3. Статья Дэйва Чени про функции как значения первого класса.