

Язык программирования



Знакомство

расскажет Михаил В. Шохирев

Клуб программистов
Шадринск
2025-2026

О чём поGоворим



История: кем, когда, где и как создавался язык.

Цели: зачем создавался язык. Улучшение *технологии разработки*.

Особенности: чем Go отличается от других языков. Корни языка.

Компиляция (для разных платформ) и выполнение.

Синтаксис: правописание и стиль. Управляющие конструкции. Данные.

Модульность: функции, методы. Пакеты. Объектное программирование.

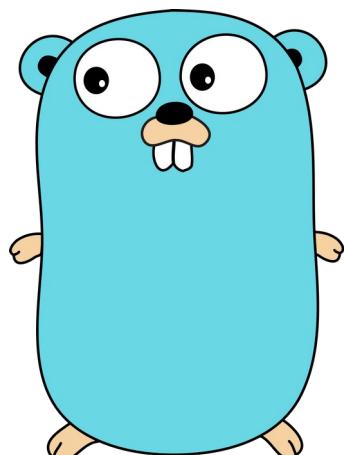
Интерфейсы: типы для действий (*contracts*), ограничения (*constraints*) для *generics*.

Многозадачность: concurrency (*goroutines*, *channels*, *select*).

Инструменты: *gofmt*, *go command*. *Go tools*. IDE и редакторы.

Применение: где, как и почему лучше использовать Go. Рейтинги.

Критика: недостатки Go и альтернативы ему. Мои впечатления.



Шустрый суслик
Gopher

(автор: Renée French)

Go = "C for the 21st century"

Go — простой быстро компилируемый язык программирования со статической типизацией, ориентированный на высокопроизводительную работу в сети и эффективное выполнение (в виде “родных” исполняемых файлов), реализующий одновременное выполнение подпрограмм, с многочисленными надёжными стандартными библиотеками, легко осваиваемый, удобный для сопровождения.

Проектировщики:

Robert Griesemer, Rob Pike, Ken Thompson (раньше работали вместе) – почти 2 года им никто не мешал спокойно проектировать язык. В язык включалось только то, что было одобрено всеми тремя создателями, каждый из которых имел ценный опыт разработки разных языков программирования.

Разработчики: команда в Google + Go community.

Цель: система программирования для разработки больших надёжных высоконагруженных быстро работающих серверных программных комплексов с распараллеливанием выполнения, которые будут развиваться в течение длительного времени большой командой разработчиков.

На проектирование Go повлияли языки C, Oberon-2, Active Oberon, Oberon, Modula-2, Modula, Pascal, Alef, Newsqueak, Squeak, CSP, Smalltalk, Limbo, APL, BCPL, occam.

Разработчиков первоначально объединило их общее недовольство языком C++: они хотели сделать язык с простым синтаксисом, но отвечающий современным требованиям к разработке программ.

Go: создатели



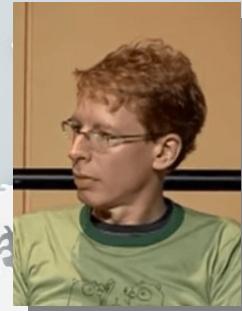
Ken Thompson,
США
«старая
инженерная
школа»):
разработчик
языка C, ОС Unix,
Plan 9, Inferno,
утилит grep, ed,
QED, кодировки
UTF-8.



Rob Pike, Канада
«следующее
поколение»,
специалист по
concurrency):
разработчик
window system for
Unix; ОС **Plan 9** и
Inferno, UTF-8,
sam, acme, языков
Sawzall, Limbo,
Newsqueak.



Go = C tokens +
Oberon structure
(& strictness)



Robert Griesemer,
Швейцария
«ученик Никласа
Вирта (Pascal, Modula,
Oberon), европейская
школа»):
разработчик V8 JS
engine, Java HotSpot
VM, языка Sawzall (a
programming language
for vector computers),
системы Strongtalk.

«Our original goal was not to create
a new programming language, it was
to create a better way to write
programs.»

Rob Pike

«Нашей изначальной целью было не
разработать ещё один новый язык
программирования, а создать
лучший способ писать программы.»

Rob Pike

Наверное, это последний язык,
разработанный «классиками»,
которые создали Unix.

Go: как достигнуты цели создания языка

Разработчики хотели не просто создать новый язык, но разработать лучшую технологию разработки программ (большой командой разработчиков в течение длительного времени):

- стабильная спецификация языка: совместимость с предыдущими и последующими версиями;
- логичный краткий синтаксис: легко освоить и однозначно понимать всеми в команде;
- строгая типизация, объявления, импорты: компилятор контролирует программистов;
- быстрый компилятор: минимизирует время сборки больших программных систем;
- легкая кросс-компиляция для разных ОС и архитектур одни и те же исходники;
-
- интерфейсы с неявным соответствием: позволяют расширять готовые системы;
- композиция вместо наследования: даёт основу для независимого развития компонентов;
- статическая диспетчеризация методов: обеспечивает быстрое выполнение;
- легковесные *goroutine*-ы: структурируют программу для одновременного выполнения процессов;
- каналы обеспечивают удобную синхронизацию и обмен данными между процессами;
- тип данных *error*: предоставляет все средства языка для явной обработки ошибок;
- единый стиль оформления исходников: задаётся утилитой **go fmt**;
-
- богатая и надёжно работающая стандартная библиотека: предоставит готовые компоненты;
- удобная распределённая система управления внешними пакетами с идентификацией по URL;
- мощный набор стандартных инструментов всегда под рукой: **go command**, **go tool command**;
- заложены широкие возможности для автоматизации за счёт расширения доступных инструментов;
-
- открытый исходный код: привлекает сообщество для развития системы программирования.

Система разработки Go (язык + инструменты) спроектирована для создания надёжно работающего большого серверного многозадачного ПО.

ХРОНОЛОГИЯ И СОВМЕСТИМОСТЬ

2007-09 началась **разработка** Go в компании Google; проектированием занимались: Robert Griesemer, Rob Pike и Ken Thompson (~ в течение 1 года).

2008-03 1-й проект (draft) спецификации языка.

2009-11-10 был официально **представлен** язык Go.

2011-03-16 go r56: based on release weekly.2011-03-07.1

2012-03-28 **go1.0**: language & a set of core libraries.

2013-05-13 go1.1: ~30%-40% performance improvement of compiled code.

2015-08-19 go1.5: compiler & runtime written in Go (+ a little assembler); dynamic libraries.

2017-08-24 go1.9: type aliases.

2018-08-24 go1.11: modules; experimental port to WebAssembly.

2020-02-25 go1.14: Go modules for production use; overlapping interfaces.

2022-03-15 go1.18: generics. Built-in fuzz testing.

2023-08-08 go1.21: min, max, clear built-in functions.

2024-02-06 go1.22: *math/rand/v2* package; PGO (Profile-guided Optimization) in compiler.

2024-08-13 go1.23: range over function types; *iter*, *unique*, *structs* packages.

2025-02-11 go1.24: generic type aliases; weak pointers; post-quantum cryptography; FIPS mode.

2025-08-12 go1.25: experimental GC (10-40% faster), many changes in the standard library.



Спецификация языка и стандартной библиотеки обратно совместимы с версиями Go 1.x.

Поэтому многие крупные компании, выждав время, убедились в долговременной поддержке языка и стали применять его в своих важных проектах.

Установка

Реализации:

1. Официальный компилятор (Google) для ОС AIX, Android, *BSD, iOS, Linux, macOS, Plan 9, Solaris, Windows (на разных аппаратных архитектурах) и для WebAssembly (WASM).
2. gofrontend + libgo для GCC и других компиляторов.
3. **TinyGo** для embedded systems (MCU) и WebAssembly.
4. **GopherJS** – кросс-компилятор из Go в JavaScript.
5. **TamaGo** — средство разработки программ, работающих на "голом железе" без ОС.

Поддерживаются практически все **архитектуры**: i386, amd64, ARM, RISC-V, MIPS, ppc64, ...

```
go tool dist list
```

Лёгкая кросс-компиляция:

```
GOOS=целевая_ОС GOARCH=целевая_архитектура go build .
```

Установка (описание <https://golang.org/doc/install>):

```
sudo apt-get install golang
```

Обновление:

```
go get go@1.25.6 # или go get go@latest
```

The Go Playground ~ интерактивное выполнение программ в браузере:

```
https://play.go.dev/
```

Проверил
работу Go под
GNU/Linux,
MS Windows,
Raspberry Pi Pico
(микроконтроллер).
А также
кросс-компиляцию:
в ELF, EXE, APK,
WASM.

Пример с приветом

```
package main                                // все программы принадлежат к своему пакету
import (
    "fmt"                                     // подключить...
    "os"                                       // ... пакет форматированного вывода
)                                            // ... и взаимодействия с ОС
const world = "世界"                        // для исходного кода и литералов: только UTF-8

func main() {
    var s string = world
    if len(os.Args) > 1 {
        s = os.Args[1]
    }
    fmt.Printf("Привет, %s!\n", s) // вызов функции из импортированного пакета
}
```

```
$ go run helloworld.go
Привет, 世界!
$ go build helloworld.go
$ ./helloworld мир
Привет, мир!
$ GOOS=windows GOARCH=amd64 go build helloworld.go
$ ls helloworld*
helloworld helloworld.exe helloworld.go
```

Каталоги и пути

```
$ echo $GOROOT
/usr/local/go
$ echo $GOPATH
~/go

$ cd $GOPATH/src

$ mkdir sample && mkdir sample/module
$ cd sample/module
```

1. Напишите program.go, которая будет использоваться в main.go.

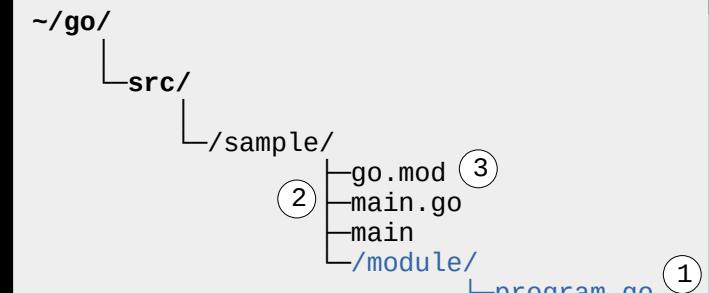
```
$ cd $GOPATH/src/sample
# 2. Напишите main.go, которая делает import "sample/module"
```

3. Объявите модуль sample в go.mod

```
$ go mod init sample
```

```
$ go run main.go
```

```
$ go build main.go
$ ./main
```



// go.mod

```
module sample
go 1.25.5
```

// main.go

```
package main
import "sample/module"
func main() {
    object := module.Type{}
    object.Method()
}
```

// module/program.go

```
package module
type Type struct {}
func (t Type) Method() {
    println("Method() of Type")}
```

Синтаксис: особенности

Простой синтаксис. Минимум синтаксических конструкций. Однозначное выражение действий (без TIMTOWTDI).

Каждое утверждение (statement) начинается с ключевого слова (25 keywords). Исходники в UTF-8.

Ошибки – это тип данных **error**. Нет исключений (exceptions). Есть `panic()` и `recover()`.

Нет классов, но в `struct` описываются поля, и для всех типов данных можно определять методы: `func (o T)m()`.

`interface` описывает тип с набором методов, другие типы могут неявно соответствовать интерфейсу, реализуя этот набор.

`;` как перевод строки (line feed) — автоматически вставляется компилятором, где необходимо; нельзя переносить { на новую строку.

`,` запятая обязательна в конце строки в списке, если нет) как завершителя списка.

`_` “пустая переменная” (blank identifier) для игнорирования значения.

`:=` простое объявление с выводом типа из значения (inferred type) и инициализация переменной в функции.

Все объявленные переменные получают начальное zero value (`0, false, "", nil` для интерфейсов и ссылочных типов).

Имена с заглавной буквы (Capitalized) экспортятся (видны вне пакета). Область видимости имён — пакет (package).

Все имена со строчной буквы видны во всех файлах внутри одного пакета. 1 пакет = 1 каталог.

Функции — полноценные типы: multiple return values, named return values, bare return, variadic functions, anonymous functions.

`func init() { }` // инициализирующие функции в файлах пакета.

`defer f()` // отложенное исполнение действий перед завершением функции: появилось в Go.

`go f()` // запустить любую функцию как goroutine для одновременного выполнения.

Обмен данными через каналы: `channel <- value; value <- channel; select / case / default`

Безтиповые константы (untyped constants) в языке со строгой типизацией!

`rune` // тип данных для “символа” (code point) в кодировке UTF-8.

`iota` для перечисления (enumerator) именованных целых значений.

`import "package"; var declared` // если не используются — программа не скомпилируется!

Как указывали сами создатели языка Go, его синтаксис основан на «C tokens» и «Oberon structure».

Пунктуация

() список: импортов, констант, параметров, возвращаемых значений, ...

[] размер массива, показатель среза элемент массива, среза, словаря

[] тип параметра в generics

{ } блок определения

{ } блок начальных значений

{ } блок кода

: отделяет индекс или ключ от значения

: ставится после метки

... размер массива вычисляется по значениям

... список параметров переменной длины

... список аргументов переменной длины

; разделитель выражений в for и if

; перевод строки (разделитель утверждений)

:= объявление и присваивание (в функции)

, разделитель в списке

. разделитель получателя и метода

<- запись в канал и чтение из канала

```
import ( "fmt" ); const ( answer = 42 ); var ( five = 42 )
func f(x float64) float64 { return 0.0 }

array [size]int;      slice []string
array[index];        slice[index];
value = map[key]
func f[T any](a []T) T { return a[0] }; e := f[int](someSlice))
```

```
type struct Point { x, y int32 }
ipAddress = [4]int{127, 0, 0, 1}
func answer() int { return 42 }
```

```
gender := [2]string{ 0: "Female", 1: "Male" }
language := map[string]int { "Go": 2007 }
label:
```

```
shadrinsk := [...] float32 { 56.05, 63.38 }
func sum(numbers ...int) (sum int) { /* range numbers */ }
integers := []int{1,2,3,4,5}; sum(integers...)
```

```
for i := 0; i < n; i++ { if y := f(i); y > 0 { println(y) } }
version := 1.0; released := 2012; fmt.Println(version)
```

```
site := "https://go.dev/"
```

```
place = findLocation(latitude, longitude)
```

```
result = object.method()
```

```
channel <- value; value = <-channel
```

Синтаксис: что исключено из Go

Чтобы сделать синтаксис кратким и понятным, сознательно исключены:

- Описатели видимости имён (`private`, `public`, ...) — её показывает *Capitalization*.
- Особые описатели для имён (`auto`, `static`, ...) — расположение `const` и `var` говорит об области действия.
- Обозначения типа и значности у констант (`42UL`, `7F`) — применение `untyped const` очень естественно.
- Дополнительная нотация для `<generics>` — поскольку это лишь [параметризация] и ограничения для типов.
- Ключевые слова структурного программирования (`procedure`, ...) — назначение выводится из контекста.
- Дополнительные управляющие конструкции (`elsif` / `elif`, ...) — лишь замусоривают исходники.
- Различные ключевые слова для разных типов циклов (`while`, `until`) — достаточно одного `for`.
- Ключевые слова ООП (`class`, ...) — объектный подход обходится только `type` и `func`.
- Зарезервированные слова для особых методов (`constructor`, `new`, `self`, ...) — допустимы любые имена.
- Ограничения ООП — методы можно определять для любого типа.
- Наследование (`inheritance`) делает отношения между типами жёсткими — поэтому `composition`.
- Явное указание реализации (`implements`) интерфейсов — неявное соответствие даёт больше свободы.
- Деструкторы — сборщик мусора (GC) эффективно освобождает память.
- Исключения (`exceptions`) нарушают ход выполнения — подпрограммы возвращают результаты и `error`.
- Арифметика с указателями (`pointer arithmetic`) — источник ошибок.
- «`;`» как разделитель утверждений — автоматически вставляется на месте перевода строки.
- Значения по умолчанию для аргументов — ведут к нарушению ISP (принцип разделения интерфейсов).
- Перегрузка функций (`functions overriding`) — снижает понимание программ.
- Автоматическое преобразование типов (`automatic type conversion`) — явное понимается однозначно.

Синтаксис: keywords

Go (48)

25 keywords:

| | |
|-------------|-----------------------|
| break | 19 data types: |
| case | bool |
| chan | byte |
| const | complex64 |
| continue | complex128 |
| default | float32 |
| defer | float64 |
| else | int |
| fallthrough | int8 |
| for | int16 |
| func | int32 |
| go | int64 |
| goto | rune |
| if | string |
| import | uint |
| interface | uint8 |
| map | uint16 |
| package | uint32 |
| range | uint64 |
| return | uintptr |
| select | |
| struct | |
| switch | |
| type | |
| var | |

4 constants:

| |
|-------|
| false |
| iota |
| nil |
| true |

C++ (108)

| |
|-----------------|
| alignas |
| alignof |
| and |
| and_eq |
| asm |
| atomic_cancel |
| atomic_commit |
| atomic_noexcept |
| auto |
| bitand |
| bitor |
| bool |
| break |
| case |
| catch |
| char |
| char8_t |
| char16_t |
| char32_t |
| class |
| compl |
| concept |
| const |
| consteval |
| constexpr |
| constinit |
| const_cast |
| continue |
| contract_assert |
| co_await |
| co_return |
| co_yield |

**decltype
default
delete
do
double
dynamic_cast
else
enum
explicit
auto
bitand
bitor
bool
break
case
catch
char
char8_t
char16_t
char32_t
class
compl
concept
const
consteval
constexpr
constinit
const_cast
continue
contract_assert
co_await
co_return
co_yield**

**reflexpr
register
reinterpret_cast
requires
return
short
signed
sizeof
static
export
extern
false
float
for
friend
goto
if
inline
int
long
mutable
namespace
new
noexcept
not
not_eq
nullptr
operator
or
or_eq
private
protected
public**

| |
|------------------|
| reflexpr |
| register |
| reinterpret_cast |
| requires |
| return |
| short |
| signed |
| sizeof |
| static |
| export |
| extern |
| false |
| float |
| for |
| friend |
| goto |
| if |
| inline |
| int |
| long |
| mutable |
| namespace |
| new |
| noexcept |
| not |
| not_eq |
| nullptr |
| operator |
| or |
| or_eq |
| private |
| protected |
| public |

В дополнение к 98 ключевым словам в C++23 есть ещё 10 идентификаторов со специальным значением:

| |
|-----------------------------------|
| final |
| override |
| transaction_safe |
| transaction_safe_dynamic |
| import |
| module |
| pre |
| post |
| trivially_relocatable_if_eligible |
| replaceable_if_eligible |

Ещё 24 токена распознаются препроцессором:

| |
|--------------------|
| if |
| elif |
| else |
| endif |
| ifdef |
| ifndef |
| elifdef |
| elifndef |
| define |
| undef |
| include |
| embed |
| line |
| error |
| warning |
| pragma |
| defined |
| _has_include |
| _has_cpp_attribute |
| _has_embed |
| export |
| import |
| module |
| _Pragma |

C# (127)

78 keywords + context keywords

Free Pascal (106)

54 keywords + reserved words

Kotlin (95)

31 hard, 17 soft & 29 modifier keywords + 18 data types

Java (70)

53 keywords + reserved words

У языка Go краткий, легко запоминающийся синтаксис для однозначной записи алгоритма.

Управление выполнением: ветвление

```
if условие {  
    действие1()  
} else {  
    действие2()  
}
```

```
switch выражение; условие {  
    case значение1:  
        действие1()  
    case значение2, значение3:  
        действие23()  
        fallthrough  
    case значениеN:  
        действиeN()  
        break  
default:  
    действиeПоУмолчанию()  
}
```

```
if выражение; условие { // if x:=f(); x > 0  
    действие1()  
} else {  
    действие2()  
}
```

```
switch {  
    case условие1:  
        действие1()  
    case условиеN:  
        действиеN()  
}
```



```
switch значение := интерфейс.(type) {  
    case значениеТипа1:  
        действие1()  
    case значениеТипaN:  
        действиеN()  
}
```

Управление выполнением: циклы

```
// итерационный: (i := 0; i < n; i++)
for инициализация; условие; изменение {
    обработка(данных)
}
```

```
// == while
for условие {
    обработка(данных)
}
```

```
// бесконечный цикл
метка:
for {
    обработка(данных)
    if условие { continue }
    if условие { break метка }
}
```

```
// перебор целых чисел от 0 до < число
for значение := range число {
    обработка(данных)
}
```

```
// перебор array или slice
for индекс, элемент := range коллекция {
    обработка(данных)
}
```

```
// перебор ассоциативного массива = map
for ключ, значение := range отображение {
    обработка(данных)
}
```

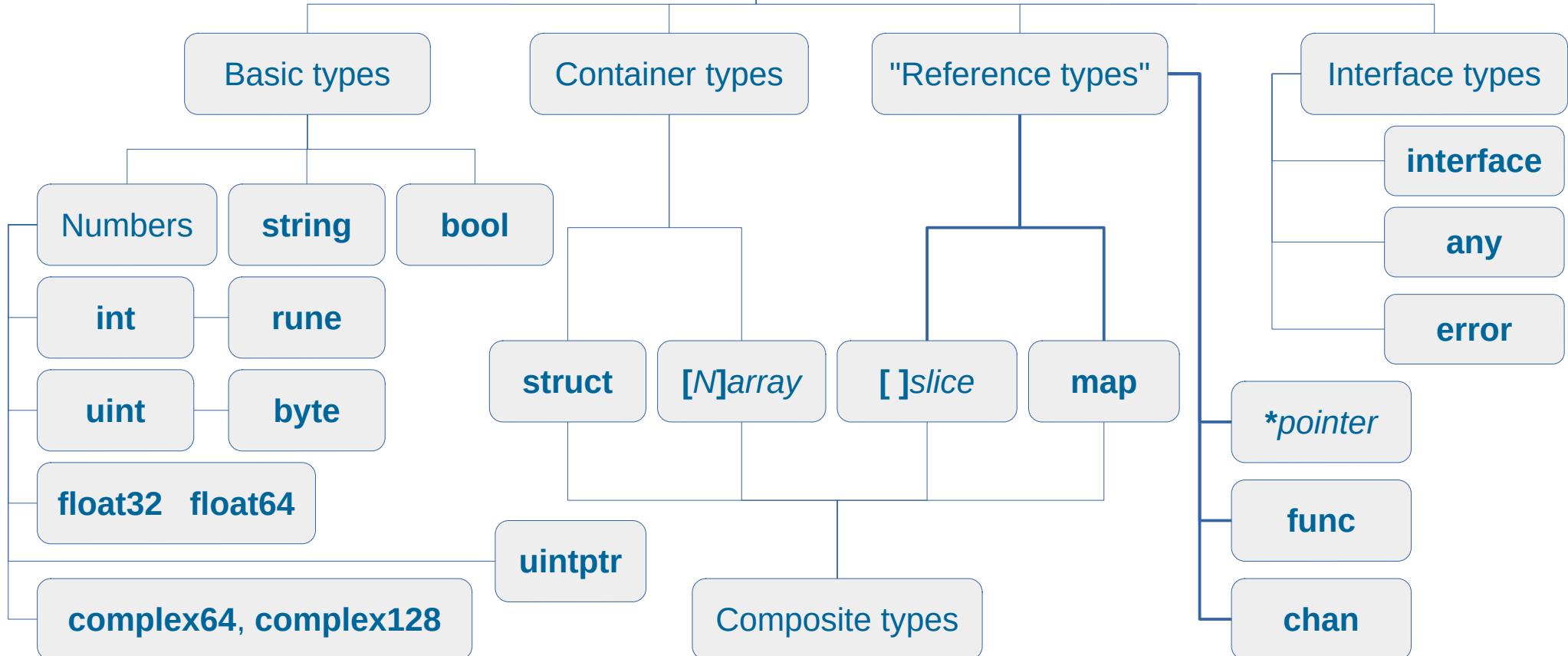
```
// получение из channel
for значение := range канал {
    обработка(данных)
}
```

Типы данных



int, int8, int16, int32 = rune, int64
uint, uint8 = byte, uint16, uint32, uint64

Data types in Go



Константы: const

Безтиповые константы (untyped constants): // математически точные, не требуются указания типа (42LL, 7UL, 3F, etc.)

```
const (
    e = 2.71828182845904523536028747135266249775724709369995957496696763
    π = 3.14159265358979323846264338327950288419716939937510582097494459
    π2 = π * π
)
var pi2 float32 = π2 // при использовании значение константы усекается до размера типа
```

Константы с заданным типом (typed constants):

```
const (
    b byte = 0xf          // байт
    x complex128 = 2+5i   // комплексное число
    Big float64 = 1 << 100 // с плавающей точкой: 1 со 100 нулями
    i int32 = -273         // целое
    Go rune = '碁'        // символ
    language string = "Go" // строка
)
```

Надо быть очень
искусным
инженером, чтобы
реализовать
безтиповыe
константы в
языке со строгой
типизацией. ☺

Предопределённые константы (predefined constants):

```
var zeroPointer *int = nil           // nil нельзя присвоить константе
const t, f bool = true, false        // логические значения

type Weekday int                   // 0 и авто-увеличение значений
const ( Sun Weekday = iota; Mon; Tue; Wed; Thu; Fri; Sat ) // 0;1;2;3;4;5;6
```

Объявление и присваивание: var

// При объявлении новой переменной у неё **всегда** есть начальное значение (zero value).

```
var (
    i      int          // 0
    G, o   rune         // 0, 0
    s      string        // ""
    tube   chan string  // nil
    ok     bool          // false
    x      float64      // 0.0
    answer = 42         // тип int выведен из присвоенного значения
)
```

В программах на Go нет
неинициализированных
переменных.

// объявление новой переменной и присваивание значения (в функции)

```
j := 25           // тип выводится из присваиваемого значения
t, f := true, false // параллельное присваивание
```

// присваивание значения уже объявленным переменным

```
s = "Go"          //
G, o = 'G', 'o'    // параллельное присваивание (tuple assignment)
i, j = j, i       // обмен значений i и j
```

Типы данных: struct



Структура (**struct**) — набор разнотипных полей: обычно для описания типа

```
type User struct {  
    id int  
    name, password string  
}  
  
u := User {  
    name: "Ken Thompson",  
    id: 42,  
}  
  
var u2 User  
u2 = u  
u2.id += 1  
  
// объявление переменной и инициализация полями значениями по порядку  
u3 := User{u.id, u.name, u.password} // следования полей  
  
var u4 User = User{}  
// инициализация пустой структурой
```

Указатели: pointer

pointer ~ указатель (на значение определённого типа):

```
type BaseType string
```

```
var v BaseType = "Значение"           // переменная типа BaseType
var p *BaseType                  // указатель на переменную типа BaseType
p = &v                         // ссылка на значение переменной типа BaseType
c := *p                         // значение переменной типа BaseType по ссылке на v
                                // нет вычислений адресов по указателям
```

// Если в функцию передать значение по ссылке, его можно изменить

```
func changeValue(p *BaseType) {
    var newValue BaseType = "Новое значение"
    *p = newValue
}
```

```
type BaseType = string
```

Контейнерные типы: container types

[размер]Type // массив (array) определённой длины:

```
var punchCard [80]rune // 80 * 0
localhost := [4]int { 127, 0, 0, 1 }
gender     := [2]string { 0:"Female", 1:"Male" }
location   := [...]float32 { 56.05, 63.38 } // ... = number of initial values
```

[]Type // срез (slice) – динамический массив («переменной» длины),
// (на самом деле – удобная надстройка над обычным массивом):

```
primes := []int { 2, 3, 5, 7, 11, 13, 17, 19, 23 }
messages := make([]string, 0, 1024)
messages = append(messages, "OK")
```

map[KeyType]ValueType // ассоциативный массив = отображение = карта:

```
languages := map[string]int { "Go": 2007 }
languages["Kotlin"] = 2011
```

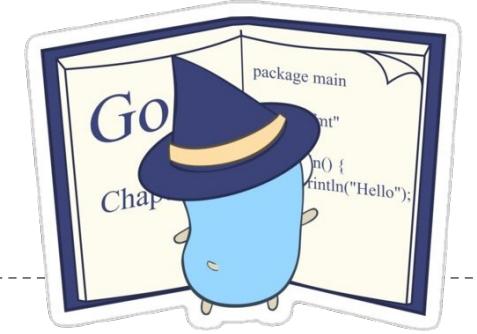
```
type Coordinates map[[2]float32]string
places := make(Coordinates)
places[location] = "Шадринск"
places[[2]float32{33.54, -118.05}] = "Norwalk"
```

Функции: func main(); func init()

// Главная функция в пакете main, с которой начинается выполнение программы.

```
package main
```

```
func main() {  
    обработка(&данных)  
}
```



// в каждом пакете может быть несколько «инициализирующих» функций,

```
func init() {  
    инициализация(&данных)  
}
```

// которые выполняются при загрузке пакета в порядке их описания

```
func init() {  
    инициализация(&других_данных)  
}
```

// и могут располагаться в разных файлах этого пакета

ФУНКЦИИ: func

```
// функция без возвращаемого значения = процедура
func debug(m string) { println(m) }
debug("побочный эффект")

// функция с одним возвращаемым значением
func save(u User) (error) { e := database.update(u); return e }
err := save(newUser)

// функция с одним именованным возвращаемым значением
func save(u User) (e error) { e = database.update(u); return } 
err := save(newUser)

// функция с несколькими возвращаемыми значениями: возможно, именованными
func add(u User) (id int, e error) { id, e = database.insert(u); return id, e }
userId, err := add(newUser)

// функция с переменным списком параметров = variadic function
func saveAll(users ...User) (e []error) { e = database.updateAll(users); return s }
arguments := []User{user1, user2, user3}
possibleErrors1 := saveAll(arguments...)
possibleErrors2 := saveAll(user4, user5, user6)

// в функции передаются копии значений аргументов (но можно передавать указатели)
```

В Go подпрограммы описываются как **func**, но они могут не возвращать результата. А могут возвращать несколько любых значений.

Это часто применяется, когда нужно вернуть результат, а также сообщить, была ли ошибка при вычислении результата.

Типы-функции: func type

type ИмяТипа func(типы, параметров) (типы, возвращаемых, значений)

```
type F1 func(int, int) int // тип функции = её сигнатура
```

```
// у любой функции есть тип, например: func(int, int) int
func add(x, y int) int { return x+y } // соответствует типу F1
```

// функция как значение переменной

```
var f1 F1 = add // присваивание объявленной функции
fa := func() { println("anonymous") } // анонимная функция типа func()
```

// функция как возвращаемое значение

```
func returnsFunc() F1 { return add } // возвращает функцию типа F1
f2 := returnsFunc()
y := f2(40, 2)
```

// функция как параметр (передаётся в функцию)

```
func receivesFunc(a, b int, f F1) (r int) { r = f(a, b); return r }
sum := receivesFunc(21, 21, add)
production := receivesFunc(21, 2, func(x, y int) int { return x*y } )
```

// определение и вызов анонимной функции

```
func() { println("lambda") }() // lambda типа func()
```

Функции в Go – это полноценные типы данных, с которыми можно обращаться, как с другими типами: хранить в массивах, передавать как аргументы, отправлять через каналы и т. д.

Методы: func (t T) f()

К любому типу данных можно присоединить поведение с помощью методов.
Метод – это функция, объявленная с объектом-получателем (receiver):

```
func (object Type) method(parameter Type) valueType { /* ... */ }
```

```
type Celsius float32
func (t Celsius) String() string { return fmt.Sprintf("%g°C", t) }
var t Celsius = 37.0
println(t.String()) // 37°C
```

```
type Album struct { name, artist string; year, length int; media string }
a := Album{ "Dark Side of the Moon", "Pink Floyd", 1973, 44, "катушка 18 см" }
```

```
type TapeRecorder struct {
    Model string
}
func (r TapeRecorder) play(a Album) {
    fmt.Printf("Playing album '%s' by '%s' for %d minutes...\n",
        a.name, a.artist, a.length)
}
```

```
recorder := TapeRecorder{ Model: "Нота 203-1 стерео" }
recorder.play(a) // Compile-Time Resolution
```

Объектное программирование: type struct + func



// Нет классов, но можно описывать типы объектов на основе struct:

```
package user
type User struct { // тип описывает структуру данных
    login, email string
}
```

«Object- but not type-oriented»
R. Griesemer

// К такому типу можно присоединить поведение с помощью методов:

```
func (u User) Login() string { return u.login }
func (u User) Email() string { return u.email }
func (u *User) SetEmail(mailbox string) { u.email = mailbox }
```

// Это не конструктор, а обычная функция, которую можно назвать New или NewUser

```
func New(l, e string) (u User) { u = User{login: l, email: e}; return u }
```

```
package main
```

```
import ( "sample/oop/user"; "fmt" )
```

```
func main() {
```

```
    mike := user.New("mshock", "mshock@caiman-club.org")
```

```
    mike.SetEmail("librarian@caiman-club.org")      // static dispatch of methods
```

```
    fmt.Printf("%v %v\n", mike.Login(), mike.Email())
```

```
}
```

Объектное программирование: composition



```
package user
type UserRole struct {
    name string
    id   int
}
func (r UserRole) Role() string { return r.name }

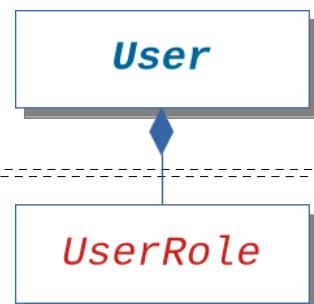
type User struct { // struct embedding (внедрение, включение одной структуры в другую)
    login, email string
    UserRole      // composition via struct embedding: описать anonymous field как имя типа
}
func New(l, e string) (u User) {
    u = User{login: l, email: e, UserRole: UserRole{"Гость", 9}}
    return u
}

-----  

package main
func main() {
    vlad := user.New("pirogov", "chairman@caiman-club.org")

    fmt.Printf("%v %v %v\n", vlad.Login(), vlad.Email(), vlad.Role())
}
// Внедрённый тип UserRole передаёт (promote) свои поля и методы внедряющему типу User
```

Inheritance ~ «is a...»
vs
Composition ~ «has a...»



Интерфейсные типы: interface types

interface – это абстрактный тип данных для описания поведения: описывает набор сигнатур для методов (set of method signatures) без их реализации:

```
type Messenger interface {  
    Send(user, message string) error  
    Receive() string, error  
}
```

Любой конкретный тип будет неявно соответствовать ранее описанному интерфейсу, если реализует все методы этого интерфейса.

```
type Telegram struct { api TelegramAPI } // этот тип соответствует типу Messenger  
func (t Telegram) Send(u, m string) (e error) { e = api.send(u, m); return e }  
func (t Telegram) Receive() (m string, e error) { m, e = api.receive(); return m, e }  
  
// У переменной абстрактного типа динамически появляется конкретный тип (underlying type) и значение  
var telegram Messenger = Telegram{ api: tg.NewClient(userID) }  
  
telegram.Send("@pirogov", "Знакомство с языком Go") // вызывается реализованный метод  
  
var icq, skype, whatsapp, viber, signal, discord Messenger  
// Объявлены переменные абстрактного типа, которым можно присваивать значения типов, соответствующих  
// контракту (интерфейсу); но пока у них нет конкретного типа (реализации), а значение = nil
```

Интерфейсы: embedding

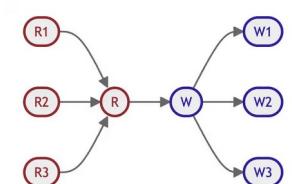
```
package io // пакет из стандартной библиотеки
type Reader interface { Read(b []byte) (int, error) }
type Writer interface { Write([]byte) (int, error) }
```

// Интерфейс может включать в себя (embed) другие интерфейсы:

```
package os
type File interface {
    io.Reader // embedded interfaces
    io.Writer
    // ...
}
```

```
package io // абстрактные действия можно легко комбинировать:
func Copy(dst Writer, src Reader) (written int64, err error)
func MultiReader(readers ...Reader) Reader
func MultiWriter(writers ...Writer) Writer
```

```
_, err := io.Copy(io.MultiWriter(w1, w2, w3), io.MultiReader(r1, r2, r3))
```



Интерфейсы: any

```
// Конкретный тип может соответствовать (satisfy) нескольким интерфейсам

    type Telegram struct { /* ... */ }
// Соответствует типу Messenger, реализуя Send() и Receive()

// а также может начать соответствовать типу io.Reader:
func (t Telegram) Read(b []byte) (int, error) { /* ... */ }

// any == interface{}
type SatisfiedByAnyType interface{} // пустой интерфейс без методов, ему
                                    // соответствует объект любого типа

func printType(i any) {
    switch t := i.(type) {
    case int:
        fmt.Println("Integer:", t)
    case float64:
        fmt.Println("Float:", t)
    case string:
        fmt.Println("String:", t)
    default:
        fmt.Printf("Unknown type: %T\n", t)
    }
}

// определяет конкретный тип значения
// приведение типа (type assertion)

if v, ok := i.(Type); ok {
    // удалось преобразовать к типу Type
    // можно использовать v
}
```

Интерфейсы: полиморфизм

```

type Flyer interface { fly() string }          // 1-й интерфейс с методом
// все типы, которые реализуют метод fly(), будут соответствовать типу Flyer

type Bird struct { Name string }              // пользовательский тип Bird
func (b Bird)fly() string {                  // реализует метод fly() и поэтому
    return "flying..."                      // соответствует интерфейсу Flyer
}

type Swimmer interface { swim() string }       // 2-й интерфейс с методом

type Penguin struct { Name string }           // пользовательский тип Penguin
func (f Penguin)swim() string { return "swimming..." } // соответствует сразу
func (b Penguin)fly() string { return "I can fly under water!" } // 2-м интерфейсам

var s = Bird{"Sparrow"}
var p = Penguin{"Gentoo"}                     // Flyer – «обобщённый» тип данных, которому
// соответствуют и Bird, и Penguin
birds := []Flyer{s, p, Bird{"Dove"}}
for _, b := range birds {                    // polymorphism
    fmt.Println(b, b.fly())
}

```

Интерфейс: sortable — описание + соответствие

```
// чтобы отсортировать любой список, нужно 3 функции:
type sortable interface {
    LessEqual(i, j int) bool          // сравнить 2 элемента
    Len() int                         // узнать длину списка
    Swap(i, j int)                   // поменять элементы местами
}

type ListOfIntegers []int // реализовав 3 метода для int, можем сортировать целые числа
func (l ListOfIntegers) LessEqual(i, j int) bool { return l[i] <= l[j] }
func (l ListOfIntegers) Len() int { return len(l) }
func (l ListOfIntegers) Swap(i, j int) { l[i], l[j] = l[j], l[i]; return }

func sortIntegers() {
    ints := ListOfIntegers{123, 789, 234, 987, 345, 890, 567, 678, 456, 876, 765, 654, 543, 432, 321}
    QuickSort(ints, true)
}

type vampires []vampire
type vampire struct {
    name      string
    prefersWomen bool
    killCount int
}
func (v vampires) LessEqual(i, j int) bool { return v[i].killCount <= v[j].killCount }
func (v vampires) Len() int { return len(v) }
func (v vampires) Swap(i, j int) { v[i], v[j] = v[j], v[i] }

var vampz vampires = vampires{vampire{"Dracula", true, 100001}, vampire{"Sava Savanović", false, 3}}
// QuickSort(vampz, false)

func QuickSort(s sortable, isAscending bool) {
    sort(s, 0, s.Len()-1, isAscending)
}

func sort(s sortable, start, end int, isAscending bool) { /* ... */ }

func partition(s sortable, start, end int, isAscending bool) int { /* ... */ }
```

Обобщённые типы: generics

// Generics описываются с помощью ограничений (constraints) на обобщённый тип в функции:

```
func First[T any](a []T) (result T, err error) {
    if len(a) == 0 {
        return result, errors.New("Slice is empty!")      // пустой результат и ошибка
    }
    return a[0], err                                         // 1-й элемент и nil (ошибки нет)
}
func Last[T any](a []T) (result T, err error) {
    if len(a) == 0 {
        return result, errors.New("Slice is empty!")      // пустой результат и ошибка
    }
    return a[len(a)-1], err                                // последний элемент и nil
}
```

```
sliceOfIntegers := []int{1, 2, 3, 4, 5}
```

```
// если тип параметра можно вывести из переменной, то его можно не указывать
first, err := First(sliceOfIntegers)
```

```
last, err := Last[int](sliceOfIntegers) // но можно и указать явно
```

```
sliceOfStrings := []string{"Вышел", "зайчик", "погулять"}
```

```
fmt.Println(First(sliceOfStrings))
```

```
fmt.Println(Last[string](sliceOfStrings))
```

Monomorphization ~
 using generics, Go can generate specific versions of functions for each type, which enhances performance by avoiding the overhead associated with dynamic dispatch.

Пакеты: package

Пакет — это набор (логически связанных) исходных файлов, расположенных в одном каталоге (directory). В начале каждого файла должна описываться его принадлежность к пакету фразой

```
package packageName // site/path/packageName
```

Пакет — единица видимости имён (типов, констант, переменных, полей, функций):

- Все имена видны во всех файлах одного пакета.
- Имена в пакете, начинающиеся с Заглавной буквы экспортятся: они видны в программе, которая импортировала пакет фразой

```
import "packageName" // название пакета – это строка
```

```
var result packageName.Type = packageName.Func(packageName.Const, packageName.Var)
```

Пакеты (не из стандартной библиотеки) могут располагаться где угодно, их полные адреса (локальные пути или URL) должны быть перечислены в конфигурационном файле

```
go.mod
```

package **main** — это специальное имя пакета, которое означает, что этот пакет содержит код, который будет скомпилирован в двоичный исполняемый файл. В одном из файлов (обычно, в **main.go**) этого пакета должна быть функция **main()**, с которой начнётся выполнение.

Имена: области видимости

Именование в языке программирования крайне важно для понимания (readability).

Области видимости (scopes) управляют поведением имён.

В Go — очень простая иерархия областей видимости (scope hierarchy):

- общая (universe): встроенные (predeclared) идентификаторы вроде `int` и `string`;
- пакет (package): все исходные файлы пакета находятся в общей области;
- файл (file): только для переименования имён пакетов в `import`;
- функция (func): тело функции;
- блок (block): тело блока (`if`, `for`, `switch`, `select`).



В языках C, C++, Java имя в исходнике может относиться к какой угодно части программы.

В Go при импорте пакета не возникает неожиданностей:

- добавление импортированного имени к текущему пакету не сломает другой пакет;
- имена не просачиваются через границы пакетов;
- любое имя всегда определено в конкретном пакете: в этом или в импортированном;
- имя `v` или `V` (без указания пакета) определено в одном из файлов текущего пакета;
- `x.V` понимается однозначно: найди пакет "x", V будет определено в нём,
- и есть только одно такое `x.V`.

Это сильно упрощает восприятие (readability), а значит понимание программы и её надёжность.

Модули: mod

Модуль — это набор пакетов, которые распространяются (с определённым номером версии) как единое целое. Модули могут загружаться прямо из систем управления версиями исходников или с общедоступных серверов.

Модуль идентифицируется путём до модуля (*module path*), который объявляется в файле `go.mod` вместе с информацией о зависимостях модуля.

```
# создать файл go.mod с именем модуля  
go mod init path/to/module/moduleName
```

Например:

```
go mod init local/module
```

```
go mod init caiman-club.org/go/mshock/presentation
```

Модули в Go – это (стандартные или собственные или от других разработчиков) библиотеки, которые импортируются, чтобы использовать их функциональность.

Главный каталог модуля (*module root directory*) — это каталог, содержащий файл `go.mod`. Когда модуль состоит из нескольких пакетов, они располагаются в подкаталогах главного каталога модуля. К ним можно обращаться по относительным путям:

```
import local/module/package
```

Многозадачность: concurrency

- **concurrency** ~ одновременность = взаимодействие множества процессов, которые могут выполняться одновременно, если позволяет «железо» и ОС

"Concurrency is the composition of independently execution things." – Rob Pike

Concurrency — это способ структурировать программу, согласовывая взаимодействие процессов (возможно, возможно, одновременно).

Concurrency — это о том, как **организовать** одновременную обработку многих вещей («**dealing** with a lot of things at once»).

В программе, спроектированной на основе **concurrency**, процессы не обязательно будут автоматически выполняться параллельно (например, из-за аппаратных ограничений).

- **parallelism** ~ параллелизм = физическое параллельное выполнение множества процессов (на одном или нескольких CPU / узлах)

"Parallelism is the simultaneous execution of multiple things." – Rob Pike

Parallelism — это параллельное выполнение нескольких (независимых, возможно, взаимосвязанных) процессов.

Parallelism — это о том, как **выполнить** обработку многих вещей параллельно («**doing** a lot of things at once»).

Программа, спроектированная на основе **concurrency**, организует взаимодействие процессов, учитывая их возможный **параллелизм**.

If you have only one processor, your program can still be concurrent but it cannot be parallel.

Многозадачность: средства

Многозадачность в Go реализована на основе CSP (communicating sequential processes: C. A. R. Hoare, 1978).

Для управления многозадачностью в язык встроено несколько механизмов:

- Подпрограммы (goroutines) для одновременного выполнения:

```
go f() // запустить любую функцию как подпроцесс
```

- Каналы (channel) для обмена данными и синхронизации выполнения:

```
var channel = make(chan Type)
channel <- value // отправить значение в канал и ждать
value = <-channel // ждать и получить значение из канала
```

- Выбор (select) для обработки нескольких потоков данных через каналы:

```
select {
case <-ch1: // если что-то появится в канале #1
case v2 := <-ch2: // если удалось получить значение из канала #2
case ch3 <- v3: // если удалось отправить в канал #3
default: // обработать другое событие
}
```

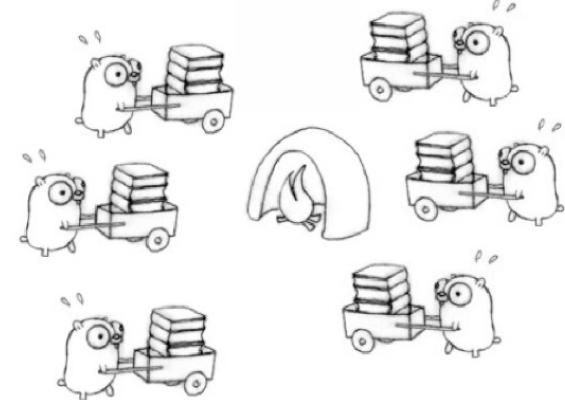
Многозадачность: goroutines

Подпрограммы goroutines — это легковесные потоки, которые выполняются одновременно с потоком главной программы (*main go thread*) и управляются системой исполнения (Go runtime).

```
func f(n int) { println(n) }

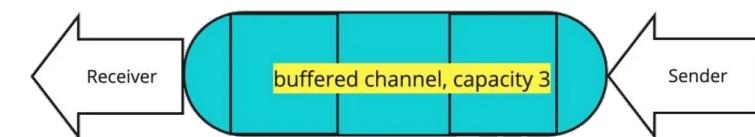
func main() {
    for n := range 5 { // любую функцию можно запустить через go
        go f(n+1) // запустить 5 экземпляров f() одновременно с main()

        // можно запустить анонимные функции
        go func () { println(n+1) }()
    }
    // дождаться завершения подпрограмм
    time.Sleep(5 * time.Second)
    println("Вышел зайчик погулять.")
}
```



Многозадачность: каналы

```
// объявить переменную для канала обмена данными указанного типа  
var channel chan T  
  
// создать канал  
channel = make(chan T, размерБуфера) // default size = 1  
  
// отправить значение в канал  
channel <- value  
// и ждать, пока не будет прочитано значение из канала  
  
// ждать, пока не будет записано значение в канал  
// прочитать значение из канала в переменную  
value := <-channel  
  
// ждать и прочитать из канала, игнорируя значение  
// <-channel  
  
close(channel) // закрыть канал
```



Каналы: пример

```

var club chan string          // объявить канал для строк, значение nil
club = make(chan string)      // выделить память каналу для строк

// club <- "Разговор о Go"      // будет deadlock !!!
// отправить в канал значение из одновременно выполняемой функции
go sendMessage(club, "Предложен разговор о языке Go.")
received := <-club            // получить значение из канала в переменную

go sendMessage(club, "Разговор о языке Go запланирован.")
go sendMessage(club, "Разговор о языке Go состоялся.")

m2, m1 := <-club, <-club      // получить 2 сообщения
close(club)
message, ok := <-club          // проверить доступность канала
if !ok {                        // канал закрыт
    fmt.Println("Разговор завершился.")
}
// сколько раз отправлено,
// столько раз получено

```

```
func sendMessage(ch chan<- string, s string) {
    ch <- s
}
```

Инструменты: go tools



Работа с модулями

```
go mod  
go get  
go install  
go list  
go doc  
go fix  
go vet
```

...

Компиляция и сборка

```
go fmt  
go build  
go generate  
go run  
go clean  
go tool asm  
go tool compile  
go tool link
```

...

Тестирование, измерение

```
go test  
go telemetry  
go bug  
go tool pprof  
go tool cover  
go tool covdata
```

...

В системе программирования на Go есть много стандартных инструментов, из которых можно собирать конвейеры для автоматического выполнения повторяющихся действий при разработке ПО.

Инструменты: go command

`go command [аргументы...]` # в одну команду `go` интегрированы все команды:

| | |
|------------------------|---|
| <code>bug</code> | оформить отчёт об ошибке (<code>bug report</code>) |
| <code>build</code> | собрать исполняемую программу со всеми зависимостями |
| <code>clean</code> | удалить объектные файлы и почистить файлы в кэше |
| <code>doc</code> | показать документацию на пакет |
| <code>env</code> | вывести информацию о переменных окружения для Go |
| <code>fix</code> | обновить пакеты с изменениями в API |
| <code>fmt</code> | переформатировать исходники к стандартному виду |
| <code>generate</code> | сгенерировать файлы Go по указаниям в исходниках |
| <code>get</code> | скачать и установить пакеты, импортированные в этом модуле |
| <code>install</code> | скомпилировать и установить пакеты и зависимости |
| <code>list</code> | вывести список пакетов или модулей |
| <code>mod</code> | подкоманды для обслуживания файла <code>go.mod</code> |
| <code>work</code> | подкоманды для обслуживания <code>workspace</code> |
| <code>run</code> | скомпилировать и сразу выполнить программу на Go |
| <code>telemetry</code> | управлять настройками и данными телеметрии |
| <code>test</code> | выполнить тесты для пакетов: <code>./... #</code> для всех |
| <code>tool</code> | запустить указанный инструмент |
| <code>version</code> | вывести версию Go |
| <code>vet</code> | сделать отчёт о потенциально ошибочных конструкциях в пакетах |

В одной команде `go` – основные стандартные инструменты системы программирования Go.

Инструменты: go tool command

`go tool [-n] command [arguments...]` # запускает такие инструменты:

| | |
|-------------------------|---|
| <code>addr2line</code> | читает адреса и выводит имена функций & место в исходнике (<code>file:line</code>) |
| <code>asm</code> | ассемблирует <code>x.go</code> в <code>x.o</code> , чтобы объединить с другими объектами в архив пакета |
| <code>buildid</code> | выводит или перезаписывает (с <code>-w</code>) <i>build ID</i> в указанном файле |
| <code>cgo</code> | преобразует исходные Go файлы в несколько исходных Go и C файлов |
| <code>compile</code> | компилирует файлы пакета в один объектный файл |
| <code>covdata</code> | генерирует отчёты из выходных файлов coverage testing (2-го поколения) |
| <code>cover</code> | анализирует данные покрытия сгенерированные ' <code>go test -coverprofile=cover.out</code> ' |
| <code>doc</code> | <code>== go doc</code> |
| <code>fix</code> | находит программы со старыми API и исправляет их для использования новых API |
| <code>link</code> | объединяет главный объектный файл и зависимости в исполняемый двоичный файл |
| <code>nm</code> | выводит список символов из объектного / исполняемого файла или архива |
| <code>objdump</code> | дизассемблирует исполняемые файлы |
| <code>pack</code> | простая версия традиционной Unix-команды <code>ar</code> с нужными для Go операциями |
| <code>pprof</code> | средство визуализации и анализа метрик о выполнении (performance profile) |
| <code>preprofile</code> | делает промежуточное представление данных <code>pprof</code> для применения в PGO |
| <code>test2json</code> | преобразует вывод <code>go test</code> в машинно-читаемый поток JSON |
| <code>trace</code> | средство просмотра файлов трассировки, сгенерированных <code>go test -trace</code> |
| <code>vet</code> | изучает исходники на Go и делает отчёт о подозрительных конструкциях |

Инструменты: IDEs

▲ IDEs And Text Editor Plugins @ go.dev:

- **Visual Studio Code** + plug-in (Microsoft) // бесплатно
- **GoLand** (IDE by JetBrains) // можно бесплатно попробовать в течение 30 дней
- **LiteIDE** (open source and cross-platform Go IDE)
- **Komodo IDE** (open source cross-platform IDE with built-in Go support)
- **Komodo Edit** + plug-in (open source cross-platform text editor)
- **Geany** (free cross-platform programmer's text editor)
- **jEdit** (open-source, cross-platform text editor: Java)
- **Notepad++** (freeware text & source code editor: Windows)
- **Kate** (cross-platform text editor with Go support out-of-the-box: KDE)
- **Sublime Text** (commercial text editor: macOS, Windows, Linux)
- **TextMate** (commercial text editor: macOS)
- **vim & Neovim+ vim-go plugin** (open-source, cross-platform text editor)

...

... Atom, BBEdit, Chime, CodeLobster IDE, Coding Rooms, emacs, Gitpod, IDEOne, Jdoodle, OneComplier, OnlineGDB, Micro, Nova, zed, Zeus IDE, ...



Распространение: компании

Го применяется в (> 40% *IT technology companies worldwide*):

Alibaba, Amazon, American Express, Apple, Armut (*C#→Go*), Baidu, BBC, bitly, ByteDance (*TikTok/Douyin*), Canonical, Capital One, CERN, Cloudflare, Cockroach Labs, Curve, DataDog, dailymotion, Docker, DropBox (*Python→Go*), GitHub, Google, gov.uk, Heroku, Huawei, IBM, InfluxDB, Intel, K8s, Kubernetes, Meta, Microsoft, Monzo Bank, Mozilla (*Rust & Go*), Netflix (*Java→Go*), New York Times, Oracle, PayPal, Pinterest, Qiniu, Reddit, RedHat, Riot Games, Salesforce (*Python, C→Go*), SendGrid, Slack, Stream (*Python→Go*), SoundCloud, Terraform, The Economist, The New York Times, Twitch, Uber, Walmart, YouTube, X / Twitter, многих других организациях и проектах *open-source*.

В России (всеми крупными компаниями, и не только):

Яндекс, ЦУМ, УГМК-Телеком, Точка, Тинькофф, Совкомбанк Технологии, СКБ Контур, Ситимобил, СберTex, Ростелеком, Онлайн-кинотеатр Иви, МТС+MWS, МойОфис, Магнит.Tech, Лаборатория Касперского, Купер, ИТ-Холдинг Т1, ИнГосСтрах Банк, ДомКлик, Группа Астра, ГНИВЦ, ГК Юзтех, АйТи Инновация, YADRO, X5 Digital, Wildberries, VK (*PHP→Go*), Viasat Tech, Tutu, Tele2, Selectel, S8.Capital, Ozon, Okko, Mail.ru Group, Lamoda Tech, iSpring, IBS, Delivery Club, Cloud.ru, Boxberry: IT, Beeline, Avito, 2GIS и многие другие...

Применение: software

Области применения ▲ Go:

- DevOps & SRE ▲ (Development Operations & Site Reliability Engineering)
- Cloud & Distributed Network Services ▲ & Databases
- Web Development ▲ (frameworks, toolkits, engines, servers)
- System Automation & CLIs ▲, Utilities & Stand-Alone Tools
- ... AI clients via API & AI Libraries & Tools (*GenKit, LocalAI, Ollama, ...*)
- ... IoT & embedded systems (*TinyGo*)
- ... UI: GUI (Linux, MacOS, iOS, Android, Windows), WUI (GopherJS, WASM), TUI / CLI

Software на Go ▲ :

ADK for Go (AI) @ Google, Allegro (eCommerce), **AmneziaWG**, AKS [Azure Container Service] @ Microsoft, AresDB @ Uber, Badoo, bilibili (video sharing), Buffalo (web framework), Caddy (web server), CockroachDB, Digger (IaC), Docker, Drone (CD), DropBox (backend), ent @ Meta, etcd (key-value DB), Flamingo (web framework), Galène (videoconferencing), GenKit (AI) @ Google, Gin (web framework), GitLab, Go Ethereum, Google Cloud, GoLand @ JetBrains, Gorgonia (ML), Gorilla (web toolkit), Grafana, gravitational/teleport (access proxy), Harvester (HCI), Hugo (website engine), Hyperledger Fabric (blockchain), InfluxDB, Istio (service mesh), JuiceFS, Kubernetes, LangChainGo, Lantern, LocalAI, LXD @ Canonical, Mattermost (messaging platform), MinIO (object storage), Monzo (banking app), NATS (messaging), NSQ (messaging), Ollama (89%), OpenShift (containerization), Podman, Prometheus (monitoring & alerting toolkit), Rend (large scale data caching @ Netflix), RoadRunner (application server for PHP), SoundCloud, SourceCraft (software development platform) @ Yandex, Soundscape (music streaming), Tendermint Core, Terraform (IaC), Timesheets (project management), Traefik (reverse proxy and load balancer), Twitch (live-streaming), TypeScript @ Microsoft, V2Ray, VITESS @ YouTube, Zabbix agent2, ...

Top 60+ Open-source Apps Written with Golang in 2024 ▲

Применение: GUI

GUI: cross-platform graphical apps (Linux, Windows, MacOS, iOS, Android)

Fyne.io

Fyne.io

- Docs
- Apps
- Add-ons
- Conference**
- Blog
- Events
- Support

Gallery

Widgets (Select one)

- Accordion
- Type or select
- Check
- Card
- Entry
- Form
- Text
- Image
- Table
- List
- Input
- CheckGroup Item 1
- Radio Item 1

Our Vision

Are you ready to build the future with us?

GioUI.org

Gio

- Home
- Newsletter
- Install
- Learn
- Showcase
- Anvil
- Chapar
- Cryptopower
- G45W
- godcr
- gotraceui
- Photon
- Protonet
- scapmi
- Sointu
- Sprig
- Transito
- Wormhole William
- Architecture
- Community

godcr

A wallet for the Decred cryptocurrency.

gotraceui

an efficient frontend for Go execution traces

Photon

A fast RSS reader as light as a photon.

Protonet

Peer-to-peer chat application.

scapmi

Simultaneously Copy And Paste Multiple Items

Sointu

A modular software synthesizer to produce music for 4k intros.

Sprig

A client for the Arbor chat system.

Transito

Public transit app to download GTFS data & calculate routes offline.

Популярность: рейтинги

TIOBE index ▲ (since 2009):

Now: #16 (Jan 2026) ← #8 (Oct 2025) ← #13 (Nov 2023)

Highest Position (before): #7 (Apr 2024, Jul 2025)

Lowest Position: #122 (May 2015)

Language of the Year: 2009, 2016

Cloudflare Radar ▲ API Client Language popularity: #1 (2024)

JetBrains ▲ Top-paid employees by programming language: #2 (2024)

GitHub Octoverse ▲ Top 10 fastest growing languages in 2024: #3

JetBrains ▲ Language Promise Index: #4 (2024)

IEEE Spectrum ▲ Top Programming Languages: #10 (2025)

Crossover ▲ Top 10 In-Demand Programming Languages for 2025: #10

ZDnet ▲ The most popular programming languages in 2025: #10

StackOverflow ▲ #12 Most popular techs: language (professionals) (2025)

PYPL ▲ : #14 (Oct 2025)

RedMonk ▲ Programming Language Rankings: #12 (Jun 2024)

Statista ▲ Most used programming languages among developers (2024): #12

GeeksForGeeks ▲ 20 Best Programming Languages to Learn in 2025: #13

Is Golang Still Growing?
Go Language Popularity
Trends in 2024 ▲ @
JetBrains

Golang in 2025: Usage,
Trends, and Popularity

При критике языка Go упоминаются следующие недостатки:

- Синтаксис слишком простой, мало syntactic sugar.
- Нет полноценного ООП.
- Явная обработка ошибок: смущает разработчиков, кто привык к исключениям (многословность, нет прерывания потока выполнения).
- Ограниченный вывод типов (type inference): явное указание типов параметров снижает простоту и выгоды от шаблонного кода.
- Ограничения (constraints) задаются только интерфейсами и могут ограничивать гибкость generics в определённых сценариях работы.
- Синтаксис непривычный: использование [] в типах параметров и в описании ограничений для generics снижает читабельность.
- Нет перегрузки функций (function overloading).
- Нет перегрузки операций (operator overloading) или добавления ключевых слов (keyword extensibility).
- Нет возможности объявить неизменяемость (immutability declarations), кроме const.
- Не хватает значений по умолчанию для параметров функций (default values for arguments) — сделано намеренно.
- Использование nil и weak type safety?
- Диспетчер сопрограмм (goroutines scheduler) управляет их выполнением, что может привести к недетерминированному поведению.
- Сборщик мусора может иногда вносить недопустимые задержки при выполнении программ (not real-time).
- Странный шаблон при форматировании даты и времени: "Mon Jan 2 15:04:05 -0700 2006".
- Обескураживающий результат при проверке интерфейсов на nil (true только, если и значение, и тип == nil).
- Рассогласование ссылок при изменении среза в функции (когда меняется ссылка на данные при изменении его размера).
- Нет проверки значений на соответствие перечислению, объявленному через iota.
- В некоторых случаях требуется более низкоуровневое управление распределением памяти, как в Rust.

По-моему, эти претензии предъявляют те, кто не понял, для чего создавался Go, и хотят сделать из него другой язык, ещё один C++ / C# / Java.

Во многих проектах разработчики сочетают применение Go с использованием других новых языков: «Rust vs. Go: Why They're Better Together» ▲ .

Go: мои впечатления



Синтаксис лаконичный и понятный, но с некоторыми непривычными конструкциями.

Логично спроектирован, предсказуем. Исходники хорошо понимаются. Можно изучить легко и быстро!
Непривычно после динамического Ruby: все объявления и преобразования надо делать явно.

Очень строгий компилятор: переменная не используется – код компилироваться не будет!

Strong typing и другие строгости важны для надёжности больших программ.

Осознал, что явная работа с ошибками дисциплинирует программиста: о них надо думать постоянно.

Убедился в преимуществах отказа от традиционного ООП в пользу объектного подхода в Go.

Полюбил интерфейсы в Go — основу динамичности и гибкости при разработке.

Довольно низкоуровневый: напоминает Си, но современный и более надёжный, очень мощный.

Очень быстро компилируется. Удобно сразу выполнить: `go run program.go`

Легко скомпилировать исполняемую программу для другого «железа» и ОС.

Действительно очень быстро выполняется: кое-что переписал на Go с Python и Ruby, сравнил скорость.

Очень много стандартных библиотек – на все случаи жизни. И они всё время улучшаются.

Легко подключать и обновлять сторонние модули (которых неимоверно много).

Хорошая документация на библиотеки (с исполняемыми примерами).

Много сайтов с примерами – изучать легко и приятно.

Хорошие инструменты в комплекте — можно разрабатывать без IDE.

На Си писать сложно и ненадёжно. А на Go пишу с удовольствием!

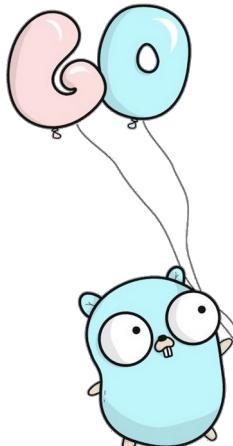
«Go isn't like C, because it is garbage-collected and has a real run-time, but it *is* like C in that **you can fit the whole language in your head.**»

Sinclair Target

Мой опыт: советы

Чему я научился, изучая Go:

- Из функций, в которых возможна непредвиденная ситуация (а их большинство!), нужно всегда возвращать `error`, который будет о ней сообщать (можно управлять уровнем подробности).
- Надо чистить программу от неиспользуемых переменных и ненужных включений (`import`).
- Лучше разбивать исходники по пакетам в подкаталогах, которые специализируются на конкретных наборах действий (конфигурация, хранение данных, API, ...) — не классы!
- Когда пишешь пакет, лучше сначала все имена сделать неимпортируемыми (`names`), и делать их видимыми извне (`Names`) при необходимости.
- Сначала тестируешь внутреннюю логику программ пакета (как «белый ящик»), а потом его интерфейс (как «чёрный ящик»).

- 
- Если функция возвращает `error` — надо обязательно проверять, не возникла ли ошибка, не игнорировать проверку.
 - Надо не лениться и всё объявлять и преобразовывать явно.
 - Удобно сочетать типы с методами и просто функции, которые что-то обрабатывают.
 - Не сразу стал отвыкать от объектно-ориентированного мышления...
 - При обработке данных по сети concurrency может сильно ускорить выполнение.
 - Быстрые исполняемые файлы для разных ОС (из одних исходников) очень удобны: просто скопировал и запустил!

КНИГ МНОГО (лучше читать на английском: свежие версии и без ошибок перевода)



ССЫЛКИ ▲

- go.dev/ // Официальный сайт языка
- go.dev/play/ // Go Playground ~ выполнение в браузере
- go.dev/ref/spec // Спецификация языка (!!!)
- github.com/golang/go // Исходники
- go.dev/doc/ // Документация
- go.dev/doc/code // How to Write Go Code
- pkg.go.dev/std // стандартная библиотека
- gobyexample.com // Go в примерах
- go.dev/doc/modules/layout // Структура каталогов
- github.com/golang-standards/project-layout // Стандартный макет [большого] Go проекта
- tour.golang.org // Экскурсия по возможностям Go
- golangdocs.com // Примеры конструкций
- appliedgo.net/why-go/ // 15 Reasons I Love Go
- [awesome-go](https://awesome-go.com) // Подборка библиотек и инструментов: для «всего на свете»

- go.dev/doc/effective_go // "Effective Go" бесплатная web-книга
- gopl.io // "The Go Programming Language" by A.A.A.Donovan & B.W.Kernighan

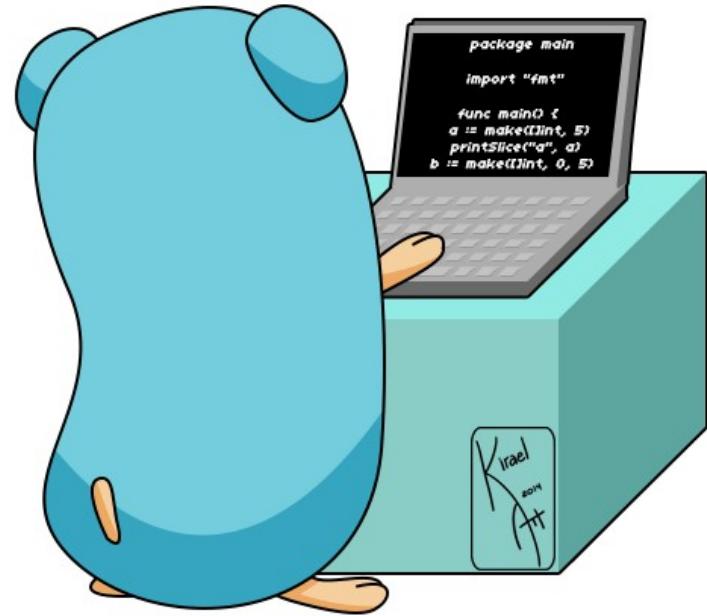
- w3schools.com/go/ @ w3schools // Справочник
- [Самоучитель по Go для начинающих](#) @ proglib.io // Самоучитель
- Учебник для начинающих @ uproger // Учебник
- [Дорожная карта Go-разработчика](#) @ proglib.io // План изучения
- lyceum.yandex.ru/go // Яндекс-лицей: Программирование на Go
- start.practicum.yandex/go-basics/ // Яндекс-практикум: Основы Go
- Книги по Go @codelibs.ru // Учебники по Go

- tinygo.org // TinyGo: Go on embedded systems & WebAssembly
- scriggo.com // Go embeddable interpreter

Готов ответить на вопросы



Ссылка на презентацию



Словарик

blank identifier ~ пустой идентификатор (_) для игнорирования значения

built-in constants ~ `false`, `true`, `iota`, `nil`

communicating sequential processes ~ взаимодействие последовательных процессов

concurrency ~ свойство программы, допускающее одновременное выполнение нескольких вычислительных процессов

CSP = communicating sequential processes

deferred function ~ функция с отложенным выполнением через применение `defer`

GC = garbage collector ~ сборщик мусора

GopherJS ~ кросс-компилятор с Go на JavaScript

gopher ~ программист на Go

gopl ~ книга "The Go Programming Language", авторы: A.A.A.Donovan & B.W.Kernighan (gopl.io)

goroutine ~ подпрограмма, запущенная для одновременного выполнения, возможно, выполняемая параллельно

multitasking ~ многозадачность

naked return ~ `return` without an expression: named return value is returned

parallelism ~ параллелизм = параллельное выполнение вычислений

predeclared names ~ like `int` and `true` for built-in constants, types and functions

process ~ процесс

subprocess ~ подпроцесс

subtask ~ подзадача

task ~ задача

type assertion ~ выведение типа переменной из её значения

untyped constants ~ безтиповые константы

zero value ~ нулевое начальное значение