GeekBrains

Go. Уровень 1

# История возникновения языка Go. Особенности языка Go.

# Работа с плейграундом и основы тулчейна

# На этом уроке

1. Познакомимся с историей создания и развития Go.
2. Рассмотрим особенности языка Go.
3. Изучим структуру простейшего Go-приложения.
4. Научимся использовать Toolchain и Playground.

# Оглавление

# История создания и развития Go

## До интернета (1950 – 1980)

В древние времена, когда по земле ходили динозавры, а компьютеры использовались исключительно для научных расчётов и запуска ракет в космос, языки программирования были низкоуровневыми, то есть позволяли практически напрямую контролировать происходящее на уровне «железа». В то время появились такие языки, как FORTRAN, BCPL, Pascal, Ada, B, C и прочие.

Появление этих языков было обусловлено несколькими обстоятельствами:

1. Компьютеры медленные.
2. Хочется писать словами, а не кодами ассемблера.
3. Рынок компьютеров мал → специалистов мало, зато уровня не ниже «суперхакер».
4. Компьютерные системы не очень сложные.

Сравнение цикла, написанного на Assembly:

```
xor cx, cx
loop1 nop ; Do something
inc cx
cmp cx, 3
jle loop1
```

С циклом, написанным на C:

```
for (int i = 0; i <= 3; i++) {
   // Do something
}
```

В то время было много разных компьютерных архитектур, соответственно, для каждой архитектуры существовал свой собственный Assembly. Языки того времени позволяли написать один раз компилятор (или транслятор в Assembly) и получить почти полную переносимость ПО с одного компьютера на другой.

# Появление интернета (1980 – 2005)

После появления интернета запросы рынка мгновенно изменились. Теперь было необходимо уметь делать веб-странички быстро, с минимальным количеством уязвимостей, а также обрабатывать большое количество запросов «одновременно».

При этом процессоры всё ещё оставались одноядерными, а производительность на одно ядро продолжала расти. Это привело к появлению огромного количества языков программирования, большинство из которых были скриптовыми. В эти годы появились Java, Perl, Python, Ruby, PHP, JavaScript и другие.

Для обработки большого количества запросов на одном ядре процессора они использовали так называемую fork-модель.

## Fork-модель

Fork — это системный вызов UNIX-подобных операционных систем, который клонирует текущий процесс.

Вначале общий алгоритм работы веб-сервера был таким:

1. Принимаем запрос.
2. Клонируем свой процесс.
3. В процессе-клоне обрабатываем запрос. В главном процессе продолжаем принимать запросы.

Так как системные вызовы — это медленные операции, скоро такой принцип построения сменился prefork-моделью:

1. Процесс запускается и клонирует себя N раз.
2. Принимаем запрос.
3. Отправляем запрос на обработку в один из N процессов-клонов.
4. Продолжаем принимать запросы.

Такая модель до сих пор используется в большом количестве веб-серверов.

У этой модели есть несколько минусов:

1. Процессы-клоны могут занимать много памяти.
2. Контроль за переключением между родительским и процессом-клоном берёт на себя операционная система.
3. Переключение между родительским и процессом-клоном — так называемое переключение контекста — происходит медленно.

Далее люди заметили, что большую часть времени работы веб-сервера занимает ожидание какого-то взаимодействия с внешним миром: ожидание запроса клиента, ожидание отправки ответа по сети, ожидание ответа базы данных и т. д.

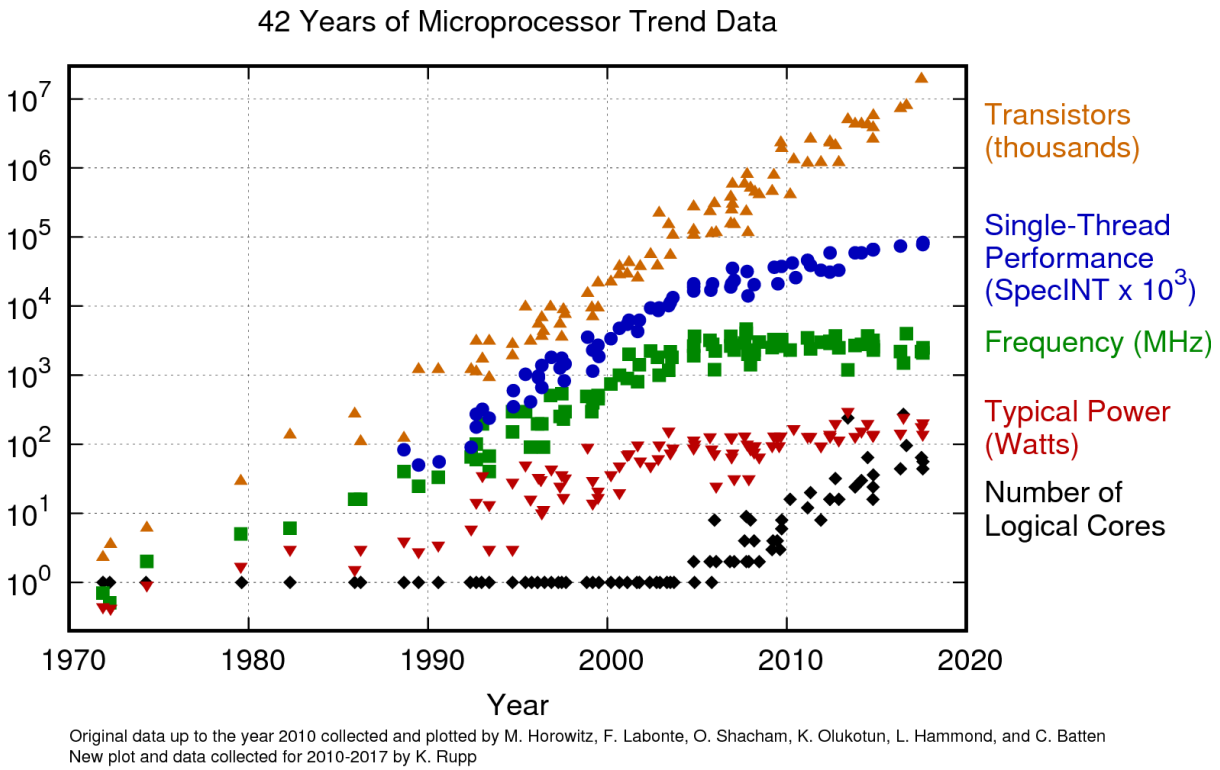
Дабы использовать это время ожидания эффективно, была придумана концепция Event Loop (событийный цикл) в связке с функциями обратного вызова (callback).

## Event Loop

Event Loop (событийный цикл) — это механизм, который позволяет каким-либо образом регистрировать все ожидающие чего-либо файлы, сетевые соединения и прочие файловые дескрипторы в некоторый цикл и прикреплять к каждому функцию, которая будет вызвана, когда ожидающий объект будет готов.

Этот подход помог решить почти все проблемы, но тут мы упёрлись в производительность компьютера.

**Производительность на ядро перестала расти (>2005)**




На графике отражены основные характеристики процессоров за последние ~40 лет.

Заметьте, что приблизительно в 2005 году рост производительности одного ядра процессора замедлился, и производители начали выпускать многоядерные процессоры, чтобы наращивать её.

Появился запрос новой технологии, которая могла бы эффективно использовать все появившиеся процессорные ядра. Большинство языков программирования стали использовать сочетание prefork-модели и event loop, однако, так как они были не приспособлены для этого изначально, в таких решениях усложнилась коммуникация между потоками и процессами. Однако технологий, удовлетворяющих этот запрос, практически не было. Была Java, которая к тому времени была большая и сложная. Были Perl, PHP, Python, Ruby, в которых использование многоядерных систем было «приделано сбоку» и от этого было неудобным. Были Haskell и Erlang, в которых с многоядерностью всё было хорошо, но они требовали глубоких академических знаний из-за того, что функциональные языки программирования возникли из теории категорий.

В связи с этим в 2007 году Робертом Гризмером (Robert Griesemer), Робом Пайком (Rob Pike) и Кеном Томпсоном (Ken Thompson) был спроектирован язык программирования Go (Golang), а в 2012-м была представлена первая стабильная версия компилятора.

Он также использует нечто похожее на сочетание prefork-модели, но вместо процессов-клонов использует потоки, и event loop, но вместо callback-функций использует корутины. Однако эти детали скрыты от разработчика за абстракцией горутины (goroutine), а среда исполнения Go сама распределяет нагрузку по ядрам процессора и обеспечивает общение между горутинами.

Кроме того, программные продукты и системы стали очень сложными, их стало очень много, а человеку необходимо бороться с постоянно возрастающей сложностью. В связи с этим язык Go решили сделать максимально простым и привычным: синтаксис языка напоминает C, а спецификация насчитывает всего 84 страницы.

# Особенности языка Go

Каждый язык программирования можно классифицировать по множеству направлений, таким образом дав языку некоторую характеристику. В этой главе мы рассмотрим особенности дизайна языка Go по нескольким критериям.

## Компилируемость

Языки программирования принято разделять на компилируемые и интерпретируемые.

**Компиляция** — процесс преобразования исходного кода программы в бинарный исполняемый файл (например, .exe) для конкретной архитектуры процессора и операционной системы.

Плюс этого подхода — большую часть проверок на корректность программы можно провести на этапе компиляции. Также стоит упомянуть, что для всех популярных компилируемых языков программирования существуют оптимизирующие компиляторы. Они оптимизируют код по некоторым алгоритмам, например, вырезая неиспользуемые части, а также используют особенности конкретной

архитектуры процессора, например векторизацию с помощью инструкций SSE. По этой причине компилируемые языки обычно быстрее и надёжнее интерпретируемых.

Хочется выделить наиболее эффективную оптимизацию, называемую инлайнингом (inlining / inline expansion). Суть её заключается во «врезании» функции в место её вызова. После этой оптимизации компилятор может понять, что, например, какая-то ветка кода функции не исполняется, и вырезать её. Эта оптимизация — основа для дальнейшей эффективной оптимизации кода.

Минус компиляции — длительность процесса, которая может значительно замедлять разработку. К примеру, ядро операционной системы Linux может компилироваться несколько часов. К тому же необходимость компиляции перед исполнением делает невозможным правку кода «на лету» и затрудняет отладку.

**Интерпретация** — процесс исполнения файлов с исходным кодом программой-интерпретатором. Интерпретатор обычно хранит некоторое внутреннее состояние интерпретируемой программы, читает файл с исходным кодом по строкам и применяет изменения к этому состоянию.

Такой подход позволяет быстро отлаживать программы, так как для их запуска не нужен процесс компиляции. Эти языки часто более высокоуровневые, так как компиляция высокоуровневого языка требует очень много времени. В интерпретаторах таких языков зачастую доступен режим REPL (Read Evaluate Print Loop), что невероятно ускоряет разработку и делает возможным использование языка в качестве повседневного рабочего инструмента.

Однако интерпретируемые языки значительно медленнее (обычно в ~100 раз), чем компилируемые. Также в них обычно проще допустить ошибку в силу отсутствия проверок компилятора.

**Транспиляция** — подход, схожий с компиляцией, однако продукт транспилятора — это не бинарный исполняемый файл, а исходный код на другом языке программирования. Яркие примеры такого подхода: TypeScript и CoffeeScript, фреймворк Svelte, транспилер/полифилер Babel и множество других инструментов работы с JavaScript.

Деление на интерпретацию и компиляцию сейчас стало достаточно условным, так как большинство интерпретаторов либо компилируют исходные файлы в байткод на старте, либо используют JIT-компиляцию, то есть компилируют особо тормозящие куски кода в процессе исполнения. Тут же можно упомянуть язык Java, который является компилируемым в байткод (низкоуровневый не человекочитаемый язык) с последующей интерпретацией этого байткода внутри JVM с возможностью JIT-оптимизаций.

Язык Go в данной классификации можно смело отнести к компилируемым языкам программирования. Этот подход был выбран из-за высокой скорости исполнения программ и надёжности, а его минус в виде низкой скорости компиляции удалось нивелировать за счёт простоты языка.

# Типизация

Каждый язык программирования создан, чтобы работать с данными. Однако данные в компьютере предстают в виде зарядов и отсутствия зарядов, что человеку легко интерпретировать как 1 и 0. Так как данные бывают очень разные, но имеют одинаковое представление, для определения способа работы с ними необходимы типы.

### Пример

Есть ряд из нулей и единиц — 01000001. Его можно интерпретировать как цифру 65 или как букву A (в кодировке ascii). Чтобы определить, что необходимо вывести на экран, необходимо знать тип, ассоциированный с этим набором из нулей и единиц.

Для характеристики типизации есть два основных критерия: отношение к постоянству типа и отношение к преобразованиям типов.

## Постоянство типа

В этом отношении типизация делится на два вида: динамическую и статическую.

При **динамической типизации** переменная не ассоциируется с типом. С типом ассоциируется значение, которое присвоено переменной. Таким образом переменной, которой присвоена строка, в следующей инструкции может быть присвоено число. Тип значения переменной может меняться во время выполнения программы.

Это удобно для быстрой разработки относительно небольших программ, однако это затрудняет отладку и проверку программ. Также требуется дополнительное количество памяти на переменную, чтобы хранить её текущий тип. Такой вид типизации в основном используется в интерпретируемых языках программирования.

При **статической типизации** тип ассоциируется с переменной, таким образом тип значения этой переменной не может меняться во время исполнения. Это удобно для проверки кода, но несколько замедляет разработку. Компилируемые языки программирования обычно статически типизированные, и корректность типов проверяется в процессе компиляции.

Недавно появился смешанный вид, называемый опциональной статической типизацией (optional static typing), при которой часть переменных может быть статически типизированной, а другая часть — динамически типизированной.

Таким образом, компилятор или статический анализатор может проводить проверку типов, а скорость разработки остаётся как при динамической типизации. Этот подход активно используется в Python3 — при помощи mypy, pyre или других статических анализаторов — и TypeScript.

Язык Go относится к языкам со статической типизацией.

## Преобразования типов

По отношению к преобразованиям типов невозможно строго разделить языки, однако каждый язык склоняется к одному из полярных вариантов: строгая (сильная) типизация и нестрогая (слабая) типизация.

При слабой типизации выражения одного типа могут преобразовываться в выражения другого типа, например (1 + 1.5) преобразуется в (1.0 + 1.5), так как дробные числа — это расширение целых чисел. В целом это удобно, но из-за подобного поведения могут возникать различные казусы.

### Пример

В языке Perl слабая типизация и выражение `""` (пустая строка) трактуется как ложь.

Выражение `0` (число 0) трактуется как ложь.

Выражение `"asdasd"` (непустая строка) трактуется как истина.

Однако выражение `"0"` (строка с нулём) трактуется как ложь, так как `"0"` (строка) автоматически преобразуется в `0` (число).

При сильной типизации любые неявные преобразования типов недопустимы.

Практически все современные языки стремятся к снижению количества неявных преобразований типов. В Go практически нет неявных преобразований типов, его можно отнести к языкам с сильной типизацией.

Таким образом, язык Go — строго статически типизированный. Однако в Go есть механизм, называемый интерфейсами (interface), который позволяет немного ослабить строгость типизации. Интерфейсы позволяют задать ограничения (в виде списка методов) на тип, при этом не называя какой-то конкретный тип.

Так, любой тип, удовлетворяющий этим ограничениям, может быть присвоен в переменную. Такой метод организации типизации называется утиной типизацией (duck-typing). Название выбрано по принципу «Если что-то плавает как утка, летает как утка и крякает как утка, мы будем считать это уткой».

На этом этапе уже можно дать обзорную характеристику какому-либо языку:

1. Python3 — опционально статически типизированный с сильной типизацией интерпретируемый язык.

2. JavaScript — слабо динамически типизированный интерпретируемый язык с возможностью JIT-оптимизации.
3. C — слабо статически типизированный компилируемый язык.
4. Go — сильно статически типизированный компилируемый язык.

Однако есть другие аспекты языка, вызывающие интерес.

# Обработка ошибок

Слово «ошибка» имеет отрицательную коннотацию, однако ошибка во время исполнения программы должна быть штатной ситуацией. Так, например, отсутствие открываемого файла на жёстком диске является ошибкой, пропавшая сеть на телефоне является ошибкой и так далее. Все эти ситуации не должны вызывать у программы каких-либо проблем, а должны быть тщательно предусмотрены программистом.

В современных языках программирования сформировалось несколько подходов к обработке и репрезентации ошибок.

## Исключения (Exceptions)

Исключения — самый распространённый в данный момент подход к ошибкам. Он подразумевает наличие специального синтаксиса для «бросания» и обработки исключений. Исключения — это особые объекты языка, которые несут информацию о том, где их создали и в каком месте «поймали». Эта информация называется stack trace или trace back и переводится как «развёртка стека вызова».

```
try:
    raise Exception()  # бросаем исключение
except Exception as e:  # ловим исключение
    process_exception(e)  # обрабатываем исключение
```

Особенность исключения — после «броска» оно «пролетает» вверх по всем вызывающим функциям до места, где его поймают.

```
def f3():
   raise Exception()  # бросаем исключение

def f2():
   f3()  # исключение пролетает здесь
   a = 1 + 2  # строка не выполняется

def f1():
   try:
```

```
        f2()  # исключение вылетает здесь
        b = 3 + 4  # строка не выполняется
    except Exception as e:  # ловим исключение
        ...  # обрабатываем исключение
    c = 5 + 6  # строчка выполняется
```

В силу механики работы исключений, в программе очень удобно установить заградительный уровень, где ловятся все исключения и обрабатываются каким-то общим способом. В этой же механике заключается и слабость: исключения неявные и не принуждают себя обрабатывать.

## Enum (объединения или типы-суммы)

В функциональных языках программирования способ обработки ошибок кардинально другой: каждая функция, в ходе работы которой может возникнуть ошибка, возвращает не ожидаемое от неё значение, а некоторый тип-объединение, который может содержать либо ошибку, либо результат выполнения.

```
match f() { // вызов функции
    Ok(val) => ... // Обработка обычной ситуации
    Err(err) => ... // обработка ошибки
}
```

Такой подход заставляет явно обрабатывать все ошибки, что повышает надёжность программы.

В Go используется похожая на enum-типы система. Функция возвращает два значения: результат и ошибку. Если ошибка не является нулевым значением, то её нужно обработать.

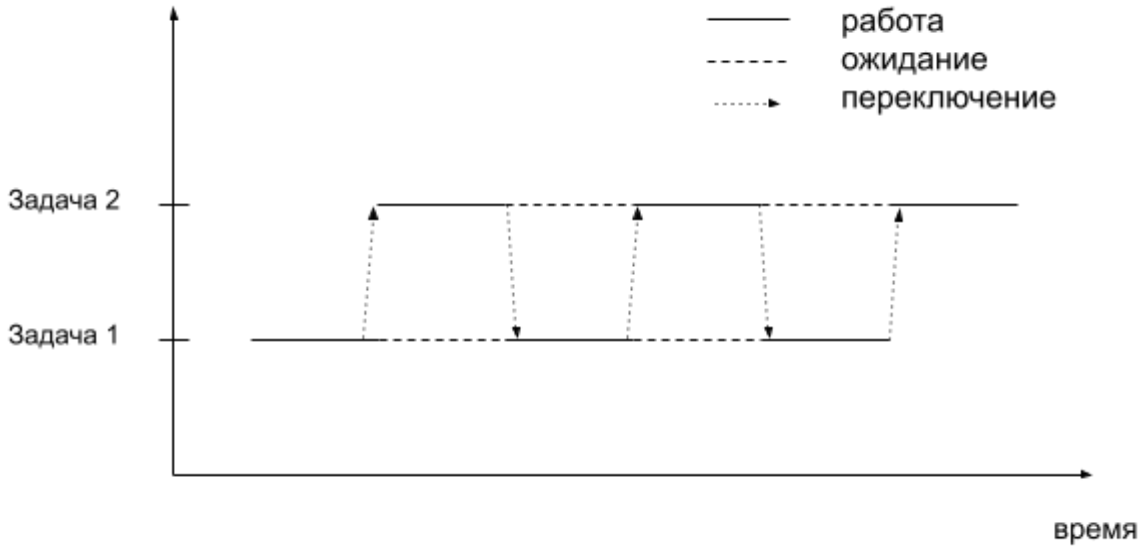
```
    func f() (int, error) {
        // ...
    }

    res, err := f()
    if err != nil {
        // обработка ошибки
    }
```

# Работа с конкурентностью (concurrency)

Перед разговором о конкурентности стоит прояснить, что это вообще такое и зачем нужно. В современном веб-программировании большую часть работы программы занимает ожидание различных событий: ожидание подключения клиента, ожидание получения запроса клиента, ожидание ответа от базы данных и так далее.

Конкурентность позволяет переключиться на другую задачу, пока текущая задача ожидает чего-либо, а позже вернуться к ней. Это позволяет утилизировать время работы программы с максимальной эффективностью.




В разделе с историей мы говорили о событийном цикле (event loop) и функциях обратного вызова (callback). Проблема этого подхода в том, что он порождает множество уровней вложенности (погуглите callback hell). Из-за этого код становится сложно читать и поддерживать.

В Go используется концепция CSP — communicating sequential processes. Она подразумевает запуск множества одновременно работающих задач и общение между ними через систему событий. В Go в качестве абстракции работающей задачи выступают горутины (goroutines), а для передачи сообщений между ними принято использовать каналы (channels). Это делает код менее вложенным, а значит, более читаемым и поддерживаемым.

К тому же в Go программа незаметно для программиста умеет работать со множеством ядер процессора, что в разы облегчает программирование многоядерных систем.

# Структура простейшего Go-приложения

Структуру Go-приложения можно представить иерархически:

```
app
      package_main
            file1.go
            file2.go
            ...
      package1
            file1.go
            file2.go
```

```
            ...
      package2
            file1.go
            file2.go
            ...
      ...
```

## Файлы с кодом

Код программы на Go, как и в большинстве языков программирования, хранится в обычных текстовых файлах в кодировке UTF-8 с расширением **.go**. Текст программы можно разбить на произвольное количество файлов. В простейшем случае текст программы содержится в одном файле.

Примечание для пользователей Windows: убедитесь, что файлы в кодировке UTF-8.

## Понятие пакета (package)

Каждый файл в Go принадлежит определённому пакету. Пакеты используются для разграничения областей видимости, но об этом поговорим позже. В данный момент важно, что в начале каждого файла необходимо указать его пакет.

## Пакет main

У каждой программы есть «точка входа». Это место, где управление передаётся написанному вами коду, до этого момента вы не можете явно контролировать поведение программы.

До точки входа программа инициализирует константы и глобальные переменные, запускает свои механизмы среды исполнения (runtime). Точка входа для Go-приложения — всегда функция main в пакете main.

## Имена и ключевые слова

Напишем простейшее Go-приложение, состоящее из одного пакета и одного файла app.go.

```
package main // говорим, что файл принадлежит пакету main

// func -- объявление функции
func main() { // func main() -- точка входа
   println("Hello, world !") // println -- функция, выводящая в консоль строку
```

```
}
```

Запустить эту программу можно командой `go run app.go`

```
> go run app.go
Hello, world !
```

## Ключевое слово var

Теперь вынесем имя в отдельную переменную, чтобы можно было менять только её, а не всю строку.

```
package main

func main() {
   // var -- объявление переменной
   // string -- тип "строка"
   //  = "world" -- присваивание ей значения "world"
   var name string = "world"

   println("Hello,", name, "!")
}
```

В Go есть ещё два варианта записи для этой конструкции:

```
package main

func main() {
   // то же самое, но без указания типа
   // Go умеет выводить тип сам
   var name = "world"

   println("Hello,", name, "!")
}
```

```
package main

func main() {
   // короткая запись объявления переменной без var и указания типа
   // используется чаще всего
   name := "name"
   name = "world" // перезапись переменной
   println("Hello,", name, "!")
}
```

Можно объявлять глобальные переменные.

```
// в строку
var globalVar1 string = "the cake is a lie" // можно указать тип

// или в блоке
var (
   globalVar2 = 1 // а можно не указывать тип
   globalVar3 = -10
)
```

## Важно

В Go существует параллельное присваивание — присваивание сразу множества переменных:

```
a, b := 1, 2 // a = 1; b = 2
```

Go не допускает наличия в коде неиспользуемых переменных и предоставляет возможность подавить «лишние» переменные:

```
_, b := 1, 2 // значение 1 никуда не запишется, оно подавлено _
```

## Ключевые слова const и iota

Помимо переменных существуют константы. Их отличие в том, что их нельзя изменить в процессе работы программы.

Глобальные:

```
// в строку
const constant1 string = "the cake is a lie" // можно указать тип

// или в блоке
const (
   constant2 = 1 // а можно не указывать тип
   constant3 = -10
)
```

Локальные:

```
func main() {
      const name = "world"
      sayHello(name) // вызов функции
}
```

Для удобного объявления последовательных констант есть ключевое слово `iota`:

```
const (
   zero  = iota // будет равна 0
   one          // будет равна 1
   two          // будет равна 2
   three        // будет равна 3
)

// можно использовать iota в выражении
const (
   ten      = (iota * 3) + 10 // будет равна 10
   thirteen                   // будет равна 13
   sixteen                    // будет равна 16
)
```

## Ключевое слово func

Для знакомства с объявлениями функций вынесем логику приветствия в функцию:

```
package main

// объявление функции
func sayHello(name string) {
   println("Hello,", name, "!")
}

func main() {
   name := "world"

   sayHello(name) // вызов функции
}
```

Любая функция объявляется таким образом:

```
func funcName(param1 type1, param2 type2, ...) returnType {
   // тело функции
}
```

## Ключевое слово import

Для подключения других пакетов используется ключевое слово `import`:

```
package main

import "math"

// можно и так
```

```
// import (
//   "math"
// )

func main() {
   println(math.E)
}
```

```
package main

import (
   // можно переименовать импорт,
   // если имена двух пакетов совпадают
   m "math"
)

func main() {
   println(m.E)
}
```

### Именование функций, переменных и констант

Для именования функций, переменных и констант в Go принято использовать `camelCase`. Имя обязано начинаться с буквы и не содержать пробелов. Теоретически можно использовать любые Unicode-символы — русские буквы, иероглифы, смайлы — однако так делать не надо.

Если имя начинается с заглавной буквы, то оно экспортируется, то есть его можно использовать, если подключить ваш пакет. В противном случае имя будет доступно только внутри вашего пакета.

Имя функции должно отвечать на вопрос «Что она делает?», имя переменной или константы должно отвечать на вопрос «Что это?»

# Toolchain и Playground

## Toolchain

При установке компилятора Go вместе с ним устанавливаются различные инструменты, полезные при разработке. Это та самая команда go.

Вам сейчас будут полезны следующие команды этой утилиты:

- `go build <file>` — сборка приложения в бинарный исполняемый файл;
- `go run <file>` — сборка и исполнение без сохранения исполняемого файла;
- `go fmt <file>` — форматирование файла с исходным кодом;

- `go help <command>` — помощь по команде (например, go help build).

## Редакторы

Go поддерживают почти все современные редакторы кода.

В данный момент самые популярные:

- Goland — лёгкий в настройке, но достаточно тяжеловесный редактор;
- Visual Studio Code — более легковесный, чем Goland, но требует настройки;
- Atom — почти то же самое, что VS Code, но сложнее настраивать;
- Sublime Text — самый быстрый из редакторов, но самый сложный в настройке.

## Playground

Для быстрой проверки маленьких кусочков кода удобно использовать Go Playground. Это простой редактор с возможностью запуска кода прямо в браузере.

Существуют альтернативные реализации, например Go Playspace, но в них не гарантируется корректная работа некоторых возможностей языка.

# Практическое задание

1. Установить тулчейн Go на своё локальное окружение.
2. Установить и настроить любой удобный редактор или IDE для работы с кодом.
3. Создать GitHub- или GitLab-репозиторий (на ваш выбор) для хранения примеров кода в рамках данного курса и склонировать его на локальное окружение.
4. Написать, запустить локально и запушить на GitHub приложение Hello World.
5. Попробовать запуск кода с помощью команды `go run`.
6. Скомпилировать исполняемый файл с помощью команды `go build`.
7. Попробовать работу с онлайн-песочницами The Go Playground и Go Play Space.

# Дополнительные материалы

1. Как установить.
2. Как начать использовать тулчейн.
3. Go Tour.
4. Go Playground.
5. Go Playspace.

6. Go как первый язык программирования.
7. Можно начать читать Effective Go: Introduction, Formatting, Commentary, Names.
8. Хорошая книга по Go для новичков в программировании (на английском): Get Programming with Go
9. Книги по Go, которые пригодятся в течение всех модулей:
    - The Go Programming Language;
    - Go in Action;
    - Go in Practice;
    - Concurrency in Go.
10. YouTube-каналы русскоязычных конференций по Go (можно посмотреть записи предыдущих лет):
    - GopherCon Russia;
    - GolangConf.
11. Что читать и смотреть об истории и современном состоянии Go:
    - На русском: доклад Алексея Палажченко про 10 лет Go.
    - На русском: доклад Фила Кулина «Почему Go такой странный».
    - Статья Роба Пайка об истории возникновения и развития Go.
    - Статья Роба Пайка о Go как языке, созданном для решения задач Google.
    - Доклад Стива Франсиа об истории Go.