FullStack.Cafe - Kill Your Tech Interview

Q1: What is Go? ☆

Topics: Golang

Answer:

Go is a general-purpose language designed with systems programming in mind. It was initially developed at Google in year 2007 by Robert Griesemer, Rob Pike, and Ken Thompson. It is strongly and statically typed, provides inbuilt support for garbage collection and supports concurrent programming. Programs are constructed using packages, for efficient management of dependencies. Go programming implementations use a traditional compile and link model to generate executable binaries.

Q2: Is Go a new language, framework or library? ☆

Topics: Golang

Answer:

Go isn't a library and not a framework, it's a new language.

Go is mostly in the C family (basic syntax), with significant input from the Pascal/Modula/Oberon family (declarations, packages). Go does have an extensive library, called the runtime, that is part of every Go program. Although it is more central to the language, Go's runtime is analogous to libc, the C library. It is important to understand, however, that Go's runtime does not include a virtual machine, such as is provided by the Java runtime. Go programs are compiled ahead of time to native machine code.

Q3: What are the benefits of using Go programming? ☆☆

Topics: Golang

Answer:

Following are the benefits of using Go programming:

- Support for environment adopting patterns similar to dynamic languages. For example type inference (`x := 0` is valid declaration of a variable `x` of type `int`).
- Compilation time is fast.
- In built concurrency support: light-weight processes (via goroutines), channels, select statement.
- Conciseness, Simplicity, and Safety.
- Support for Interfaces and Type embedding.
- The go compiler supports static linking. All the go code can be statically linked into one big fat binary and it can be deployed in cloud servers easily without worrying about dependencies.

Q4: What is *static type declaration* of a variable in Go? ☆☆

Topics: Golang

Answer:

Static type variable declaration provides assurance to the compiler that there is one variable existing with the given type and name so that compiler proceed for further compilation without needing complete detail about the variable. A variable declaration has its meaning at the time of compilation only, compiler needs actual variable declaration at the time of linking of the program.

Q5: What is *dynamic type declaration* of a variable in Go? ☆☆

Topics: Golang

Answer:

A *dynamic type variable declaration* requires compiler to interpret the type of variable based on value passed to it. Compiler don't need a variable to have type statically as a necessary requirement.

Q6: Can you *declared multiple types of variables* in single declaration in Go? ☆☆

Topics: Golang

Answer:

Yes. Variables of different types can be declared in one go using type inference.

```
var a, b, c =  3,  4,  "foo"
```

Q7: What is a *pointer*? ☆☆

Topics: Golang

Answer:

A **pointer variable** can hold the *address* of a variable.

Consider:

```
var x =  5  var p *int p =  &x
fmt.Printf("x = %d",  *p)
```

Here `x` can be accessed by `*p`.

Q8: Can you *return multiple values* from a function? ☆☆

Topics: Golang

Answer:

A Go function can return multiple values.

Consider:

```go
package main
import "fmt"

func swap(x, y string) (string, string) {
   return y, x
}
func main() {
   a, b := swap("Mahesh", "Kumar")
   fmt.Println(a, b)
}
```

Q9: What are some advantages of using Go? ☆☆

Topics: Golang

Answer:

Go is an attempt to introduce a new, concurrent, garbage-collected language with fast compilation and the following benefits:

- It is possible to compile a large Go program in a few seconds on a single computer.
- Go provides a model for software construction that makes dependency analysis easy and avoids much of the overhead of C-style include files and libraries.
- Go's type system has no hierarchy, so no time is spent defining the relationships between types. Also, although Go has static types, the language attempts to make types feel lighter weight than in typical OO languages.
- Go is fully garbage-collected and provides fundamental support for concurrent execution and communication.
- By its design, Go proposes an approach for the construction of system software on multicore machines.

Q10: Why the Go language was created? ☆☆

Topics: Golang

Answer:

Go was born out of frustration with existing languages and environments for systems programming.

Go is an attempt to have:

- an interpreted, dynamically typed language with
- the efficiency and safety of a statically typed, compiled language
- support for networked and multicore computing
- be fast in compilation

To meet these goals required addressing a number of linguistic issues: an expressive but lightweight type system; concurrency and garbage collection; rigid dependency specification; and so on. These cannot be addressed well by libraries or tools so a new language was born.

Q11: Does Go have *exceptions*? ☆☆

Topics: Golang

Answer:

No, Go takes a different approach. For plain error handling, Go's **multi-value returns** make it easy to report an error without overloading the return value. Go code uses error values to indicate an abnormal state.

Consider:

```
func Open(name string) (file *File, err error)
```

```
f, err := os.Open("filename.ext")
if err != nil {
    log.Fatal(err)
}
// do something with the open *File f
```

Q12: What are *Goroutines*? ☆☆

Topics: Golang

Answer:

Goroutines are functions or methods that run concurrently with other functions or methods. Goroutines can be thought of as light weight threads. The cost of creating a Goroutine is tiny when compared to a thread. Its common for Go applications to have thousands of Goroutines running concurrently.

Q13: What kind of *type conversion* is supported by Go? ☆☆

Topics: Golang

Answer:

Go is very strict about **explicit typing**. There is no automatic type promotion or conversion. Explicit type conversion is required to assign a variable of one type to another.

Consider:

```
i := 55      //int
j := 67.8    //float64
sum := i + int(j) //j is converted to int
```

Q14: How to efficiently *concatenate strings* in Go? ☆☆

Topics: Golang

Problem:

In Go, a `string` is a primitive type, which means it is read-only, and every manipulation of it will create a new string.

So if I want to concatenate strings many times without knowing the length of the resulting string, what's the best way to do it?

Solution:

Beginning with Go 1.10 there is a `strings.Builder`. A Builder is used to efficiently build a string using Write methods. It minimizes memory copying. The zero value is ready to use.

```
package main

import (
    "strings"
    "fmt"
)

func main() {
    var str strings.Builder

    for i := 0; i < 1000; i++ {
        str.WriteString("a")
    }

    fmt.Println(str.String())
}
```

Q15: Have you worked with Go 2? ☆☆☆

Topics: Golang

Answer:

Tricky questions and the answer is no one worked. There is no Go version 2 available in 2018 but there are some movement toward it. Go 1 was released in 2012, and includes a language specification, standard libraries, and custom tools. It provides a stable foundation for creating reliable products, projects, and publications. The purpose of Go 1 is to provide long-term stability. There may well be a Go 2 one day, but not for a few years and it will be influenced by what we learn using Go 1 as it is today.

The possible goals and features of Go 2 are:

- Fix the most significant ways Go fails to scale
- Provide backward compatibility
- Go 2 must not split the Go ecosystem

Q16: Name some advantages of *Goroutines* over *threads* ☆☆☆

Topics: Golang

Answer:

- Goroutines are extremely **cheap to create** when compared to threads. They are only a few kb in stack size and the stack can grow and shrink according to needs of the application whereas in the case of threads the stack size has to be specified and is fixed.
- The Goroutines are **multiplexed** to fewer number of OS threads. There might be only one thread in a program with thousands of Goroutines. If any Goroutine in that thread blocks say waiting for user input, then another OS thread is created and the remaining Goroutines are moved to the new OS thread.
- Goroutines communicate using **channels**. Channels by design **prevent race conditions** (a race condition occurs when two or more threads can access shared data and they try to change it at the same time) from happening when accessing shared memory using Goroutines. Channels can be thought of as a pipe using which Goroutines communicate.

Q17: Is Go an object-oriented language? ☆☆☆

Topics: Golang

Answer:

Yes and no. Although Go has types and methods and allows an object-oriented style of programming, there is no type hierarchy. This is in contrast to most object-oriented languages like C++, Java, C#, Scala, and even dynamic languages like Python and Ruby.

Go Object-Oriented Language Features:

- **Structs** - Structs are user-defined types. Struct types (with methods) serve similar purposes to classes in other languages.
- **Methods** - Methods are functions that operate on particular types. They have a receiver clause that mandates what type they operate on.
- **Embedding** - we can embed anonymous types inside each other. If we embed a nameless struct then the embedded struct provides its state (and methods) to the embedding struct directly.
- **Interfaces** - Interfaces are types that declare sets of methods. Similarly to interfaces in other languages, they have no implementation. Objects that implement all the interface methods automatically implement the interface. There is no inheritance or subclassing or "implements" keyword.

The Go way to implement:

- ** Encapsulation** - Go encapsulates things at the package level. Names that start with a lowercase letter are only visible within that package. You can hide anything in a private package and just expose specific types, interfaces, and factory functions.
- **Inheritance** - composition by *embedding* an anonymous type is equivalent to implementation inheritance.
- **Polymorphism** - A variable of type interface can hold any value which implements the interface. This property of interfaces is used to achieve polymorphism in Go.

Consider:

```go
package main

import (
    "fmt"
)

// interface declaration
type Income interface {
    calculate() int
    source() string
}

// struct declaration
type FixedBilling struct {
    projectName string
    biddedAmount int
}

type TimeAndMaterial struct {
    projectName string
    noOfHours  int
    hourlyRate int
}

// interface implementation for FixedBilling
func (fb FixedBilling) calculate() int {
    return fb.biddedAmount
}

func (fb FixedBilling) source() string {
    return fb.projectName
}

// interface implementation for TimeAndMaterial
func (tm TimeAndMaterial) calculate() int {
    return tm.noOfHours * tm.hourlyRate
}

func (tm TimeAndMaterial) source() string {
    return tm.projectName
}

// using Polymorphism for calculation based
// on the array of variables of interface type
func calculateNetIncome(ic []Income) {
    var netincome int = 0
    for _, income := range ic {
        fmt.Printf("Income From %s = $%d\n", income.source(), income.calculate())
        netincome += income.calculate()
    }
    fmt.Printf("Net income of organisation = $%d", netincome)
}

func main() {
    project1 := FixedBilling{projectName: "Project 1", biddedAmount: 5000}
    project2 := FixedBilling{projectName: "Project 2", biddedAmount: 10000}
    project3 := TimeAndMaterial{projectName: "Project 3", noOfHours: 160, hourlyRate: 25}
    incomeStreams := []Income{project1, project2, project3}
    calculateNetIncome(incomeStreams)
}
```

Q18: What is `rune` type in Go? ☆☆☆

Topics: Golang

Answer:

There are many other symbols invented by humans other than the 'abcde..' symbols. And there are so many that we need 32 bit to encode them.

A **rune** is a builtin type in Go and it's the alias of `int32`. rune represents a Unicode CodePoint in Go. It does not matter how many bytes the code point occupies, it can be represented by a rune. For example the rule literal `a` is in reality the number 97.

A string is *not* necessarily a sequence of runes. We can convert between `string` and `[]rune`, but they are different.

Q19: What is so special about *constants* in Go? ☆☆☆

Topics: Golang

Answer:

Constants in Go are special.

- **Untyped constants.** Any constant in golang, named or unnamed, is untyped unless given a type explicitly. For example an untyped floating-point constant like `4.5` can be used anywhere a floating-point value is allowed. We can use untyped constants to temporarily escape from Go’s strong type system until their evaluation in a type-demanding expression.

```
1       // untyped integer constant
const a = 1
var myFloat32 float32 = 4.5
var myComplex64 complex64 = 4.5
```

- **Typed constants**. Constants are typed when you explicitly specify the type in the declaration. With typed constants, you lose all the flexibility that comes with untyped constants like assigning them to any variable of compatible type or mixing them in mathematical operations.

```
const typedInt int = 1
```

Generally we should declare a type for a constant only if it’s absolutely necessary. Otherwise, just declare constants without a type.

Q20: How do you *swap two values*? Provide a few examples. ☆☆☆

Topics: Golang

Answer:

Two values are swapped as easy as this:

```
a, b = b, a
```

To swap three values, we would write:

```
a, b, c = b, c, a
```

The swap operation in Go is guaranteed from side effects. The values to be assigned are guaranteed to be stored in temporary variables before starting the actual assigning, so the order of assignment does not matter.

FullStack.Cafe - Kill Your Tech Interview

Q1: How to *initialise a struct* in Go? ☆☆☆

Topics: Golang

Answer:

The new keyword can be used to create a new struct. It returns a pointer to the newly created struct.

```
var pa *Student   // pa == nil
pa = new(Student) // pa == &Student{"", 0}
pa.Name = "Alice" // pa == &Student{"Alice", 0}
```

You can also create and initialize a struct with a **struct literal**.

```
b := Student{ // b == Student{"Bob", 0}
    Name: "Bob",
}

pb := &Student{ // pb == &Student{"Bob", 8}
    Name: "Bob",
    Age:  8,
}

c := Student{"Cecilia", 5} // c == Student{"Cecilia", 5}
d := Student{}             // d == Student{"", 0}
```

Q2: How to check if a `Map` *contains* a `key` in Go? ☆☆☆

Topics: Golang

Answer:

```
if val, ok := dict["foo"]; ok {
    //do something here
}
```

`if` statements in Go can include both a condition and an initialization statement. The example above uses both:

- initializes two variables - `val` will receive either the value of "foo" from the map or a "zero value" (in this case the empty string) and `ok` will receive a bool that will be set to `true` if "foo" was actually present in the map

- evaluates `ok`, which will be `true` if "foo" was in the map

If "foo" is indeed present in the map, the body of the `if` statement will be executed and `val` will be local to that scope.

Q3: Is there a `foreach` construct in the Go language? ☆☆☆

Topics: Golang

Answer:

A `for` statement with a `range` clause iterates through all entries of an array, slice, string or map, or values received on a channel. For each entry it assigns iteration values to corresponding iteration variables and then executes the block.

```
for index, element := range someSlice {
    // index is the index where we are
    // element is the element from someSlice for where we are
}
```

If you don't care about the index, you can use `_`:

```
for _, element := range someSlice {
    // element is the element from someSlice for where we are
}
```

Q4: Can Go have *optional parameters*? ☆☆☆

Topics: Golang

Problem:

Or can I just define two functions with the same name and a different number of arguments?

Solution:

Go *does not have optional parameters nor does it support method overloading*:

> Method dispatch is simplified if it doesn't need to do type matching as well. Experience with other languages told us that having a variety of methods with the same name but different signatures was occasionally useful but that it could also be confusing and fragile in practice. Matching only by name and requiring consistency in the types was a major simplifying decision in Go's type system.

Q5: What is the preferred way to handle configuration parameters for a Go program? ☆☆☆

Topics: Golang

Answer:

The **JSON** format worked quite well. The standard library offers methods to write the data structure indented, so it is quite readable.

Consider:

```
{
    "Users": ["UserA","UserB"],
    "Groups": ["GroupA"]
}
```

And to read the file:

```
import (
    "encoding/json"
    "os"
    "fmt"
)

type Configuration struct {
    Users    []string
    Groups   []string
}

file, _ := os.Open("conf.json")
defer file.Close()
decoder := json.NewDecoder(file)
configuration := Configuration{}
err := decoder.Decode(&configuration)
if err != nil {
  fmt.Println("error:", err)
}
fmt.Println(configuration.Users) // output: [UserA, UserB]
```

Q6: What is the difference between the `=` and `:=` operator? ☆☆☆

Topics: Golang

Answer:

In Go, `:=` is for declaration + assignment, whereas `=` is for assignment only.

For example, `var foo int = 10` is the same as `foo := 10`.

Q7: What is the difference between `C.sleep()` and `time.Sleep()`? ☆☆☆

Topics: Golang

Answer:

- `C.sleep()` invokes syscall sleep, which causes idle threads
- `time.Sleep()` is optimized for goroutine so syscall sleep is not involved.

Q8: What are the differences between *unbuffered* and *buffered* channels? ☆☆☆

Topics: Golang

Answer:

- For **unbuffered channel**, the sender will block on the channel until the receiver receives the data from the channel, whilst the receiver will also block on the channel until sender sends data into the channel.
- Compared with unbuffered counterpart, the sender of **buffered channel** will block when there is no empty slot of the channel, while the receiver will block on the channel when it is empty.

Q9: Why would you prefer to use an *empty* `struct{}`? ☆☆☆

Topics: Golang

Answer:

You would use an empty struct when you would want to save some memory. Empty structs do not take any memory for its value.

```
a := struct{}{}
println(unsafe.Sizeof(a))
// Output: 0
```

This saving is usually insignificant and is dependent on the size of the slice or a map. Although, more important use of an empty struct is to show a reader you do not need a value at all. Its purpose in most cases is mainly informational.

Q10: How to *copy* `Map` in Go? ☆☆☆

Topics: Golang

Answer:

You copy a map by traversing its keys. Unfortunately, this is the simplest way to copy a map in Go:

```
a := map[string]bool{"A": true, "B": true}
b := make(map[string]bool)
for key, value := range a {
    b[key] = value
}
```

Q11: What would you do if you need a `hash` displayed in a *fixed order*? ☆☆☆

Topics: Golang

Answer:

You would need to sort that hash's keys.

```
fruits := map[string]int{
    "oranges": 100,
    "apples":  200,
    "bananas": 300,
}

// Put the keys in a slice and sort it.
var keys []string
for key := range fruits {
    keys = append(keys, key)
}
sort.Strings(keys)

// Display keys according to the sorted slice.
for _, key := range keys {
    fmt.Printf("%s:%v\n", key, fruits[key])
}
// Output:
// apples:200
// bananas:300
// oranges:100
```

Q12: How does Go compile so quickly? ☆☆☆☆

Topics: Golang

Answer:

Go provides a model for software construction that makes **dependency analysis** easy and avoids much of the overhead of C-style include files and libraries.

There are three main reasons for the compiler's speed.

- First, all imports must be explicitly listed at the beginning of each source file, so the compiler does not have to read and process an entire file to determine its dependencies.
- Second, the dependencies of a package form a directed acyclic graph, and because there are no cycles, packages can be compiled separately and perhaps in parallel.
- Finally, the object file for a compiled Go package records export information not just for the package itself, but for its dependencies too. When compiling a package, the compiler must read one object file for each import but need not look beyond these files.

Go only needs to include the packages that you are importing *directly* (as those already imported what *they* need). This is in stark contrast to C/C++, where **every single file** starts including x headers, which include y headers etc.

Bottom line: Go's compiling takes linear time w.r.t to the number of imported packages, where C/C++ take exponential time.

Q13: What is an idiomatic way of representing *enums* in Go? ☆☆☆☆

Topics: Golang

Answer:

Use `Iota`. Within a constant declaration, the predeclared identifier `iota` represents successive untyped integer constants. It is reset to 0 whenever the reserved word const appears in the source and increments after each ConstSpec. It can be used to construct a set of related constants:

```
const (
        A = iota
        C
        T
        G
)
```

or

```
type Base int
```

```
const (
        A Base = iota
        C
        T
        G
)
```

Q14: What are the use(s) for *tags* in Go? ☆☆☆☆

Topics: Golang

Answer:

A **tag** for a field allows you to attach meta-information to the field which can be acquired using *reflection*. Usually it is used to provide transformation info on how a struct field is encoded to or decoded from another format (or stored/retrieved from a database), but you can use it to store whatever meta-info you want to, either intended for another package or for your own use.

Consider:

```go
type User struct {
    Name  string `mytag:"MyName"`
    Email string `mytag:"MyEmail"`
}

u := User{"Bob", "bob@mycompany.com"}
t := reflect.TypeOf(u)

for _, fieldName := range []string{"Name", "Email"} {
    field, found := t.FieldByName(fieldName)
    if !found {
        continue
    }
    fmt.Printf("\nField: User.%s\n", fieldName)
    fmt.Printf("\tWhole tag value : %q\n", field.Tag)
    fmt.Printf("\tValue of 'mytag': %q\n", field.Tag.Get("mytag"))
}
```

Output:

```
Field: User.Name
    Whole tag value : "mytag:\"MyName\""
    Value of 'mytag': "MyName"

Field: User.Email
    Whole tag value : "mytag:\"MyEmail\""
    Value of 'mytag': "MyEmail"
```

Q15: How to find a *type* of an object in Go? ☆☆☆☆

Topics: Golang

Problem:

How do I find the type of an object in Go? In Python, I just use `typeof` to fetch the type of object. Similarly in Go, is there a way to implement the same ?

Solution:

I found 3 ways to return a variable's type at runtime:

Using **string formatting**:

```go
func typeof(v interface{}) string {
    return fmt.Sprintf("%T", v)
}
```

Using **reflect package**

```go
func typeof(v interface{}) string {
    return reflect.TypeOf(v).String()
}
```

Using **type assertions**

```go
func typeof(v interface{}) string {
    switch v.(type) {
    case int:
        return "int"
    case float64:
        return "float64"
    //... etc
    default:
        return "unknown"
    }
}
```

Every method has a different best use case:

- string formatting - short and low footprint (not necessary to import reflect package)
- reflect package - when need more details about the type we have access to the full reflection capabilities
- type assertions - allows grouping types, for example recognize all int32, int64, uint32, uint64 types as "int"

Q16: When is the `init()` function run? ☆☆☆☆

Topics: Golang

Answer:

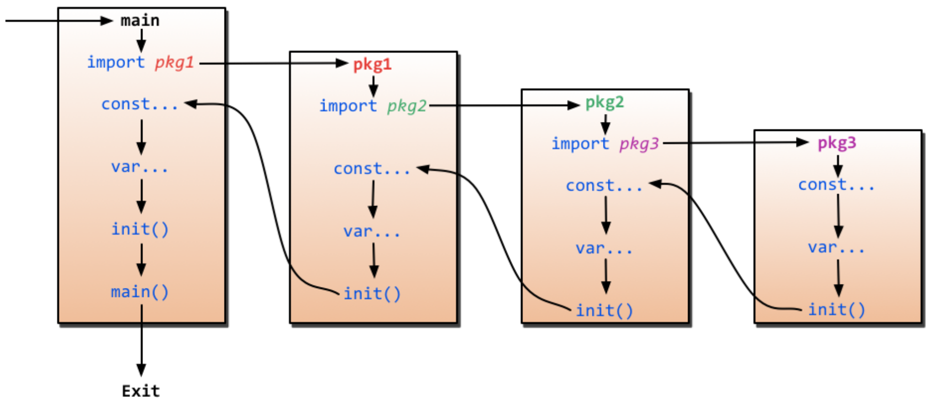
`init()` is called after all the variable declarations in the package have evaluated their initializers, and those are evaluated only after all the imported packages have been initialized.

If a package has one or more `init()` functions they are automatically executed before the main package's main() function is called.

```
import --> const --> var --> init()
```

- If a package imports other packages, the imported packages are initialized first.
- Current package's constant initialized then.
- Current package's variables are initialized then.

- Finally, `init()` function of current package is called.



Q17: How can I check if *two slices are equal*? ☆☆☆☆

Topics: Golang

Answer:

You need to loop over each of the elements in the slice and test. Equality for slices is not defined. However, there is a `bytes.Equal` function if you are comparing values of type `[]byte`.

```go
func testEq(a, b []Type) bool {

    // If one is nil, the other must also be nil.
    if (a == nil) != (b == nil) {
        return false;
    }

    if len(a) != len(b) {
        return false
    }

    for i := range a {
        if a[i] != b[i] {
            return false
        }
    }

    return true
}
```

We also could use `reflect.DeepEqual()`. Slice values are deeply equal when all of the following are true: they are both nil or both non-nil, they have the same length, and either they point to the same initial entry of the same underlying array (that is, &x[0] == &y[0]) or their corresponding elements (up to length) are deeply equal. Note that a non-nil empty slice and a nil slice (for example, []byte{} and []byte(nil)) are not deeply equal.

Q18: List the functions that can *stop* or *suspend* the execution of current goroutine, and explain their differences. ☆☆☆☆

Topics: Golang

Answer:

- `runtime.Gosched`: give up CPU core and join the queue, thus will be executed automatically.
- `runtime.gopark`: blocked until the callback function unlockf in argument list return false.
- `runtime.notesleep`: hibernate the thread.
- `runtime.Goexit`: stop the execution of goroutine immediately and call defer, but it will not cause panic.

Q19: Briefly describe how GC works in GO ☆☆☆☆

Topics: Golang

Answer:

- `MarkWorker` goroutine recursively scan all the objects and colors them into white (inaccessible), gray (pending), black (accessible). But finally they will only be black and white objects.
- In compile time, the compiler has already injected a snippet called write barrier to monitor all the modifications from any goroutines to heap memory.
- When "Stop the world" is performed, scheduler hibernates all threads and preempt all goroutines.
- Garbage collector will recycle all the inaccessible objects so heap or central can reuse.
- If the whole span of memory are unused, it can be freed to OS.
- Perform "Start the world" to wake cpu cores and threads, and resume the execution of goroutines.

Q20: What is `$GOROOT` and `$GOPATH`? ☆☆☆☆

Topics: Golang

Answer:

- `$GOROOT` is the root directory for standard library, including executables, source code and docs.
- `$GOPATH` is the directory for 3rd party packages.

FullStack.Cafe - Kill Your Tech Interview

Q1: Let's talk variable declaration in Go. Could you explain what is a variable `zero value`? ☆☆

Topics: Golang

Answer:

Variable is the name given to a memory location to store a value of a specific type. There are various syntaxes to declare variables in go.

```
// 1 - variable declaration, then assignment
var age int
age = 29

// 2 - variable declaration with initial value
var age2 int = 29

// 3 - Type inference
var age3 = 29

// 4 - declaring multiple variables
var width, height int = 100, 50

// 5 - declare variables belonging to different types in a single statement
var (
      name1 = initialvalue1,
      name2 = initialvalue2
)
// 6 - short hand declaration
name, age4 := "naveen", 29 //short hand declaration
```

If a variable is not assigned any value, go automatically initialises it with the **zero value of the variable's type**. Go is strongly typed, so variables declared as belonging to one type cannot be assigned a value of another type.

Q2: Explain this code ☆☆

Topics: Golang

Problem:

In Go there are various ways to return a struct value or slice thereof. Could you explain the difference?

```
type MyStruct struct {
    Val int
}

func myfunc() MyStruct {
    return MyStruct{Val: 1}
}

func myfunc() *MyStruct {
    return &MyStruct{}
}
```

```
func myfunc(s *MyStruct) {
    s.Val = 1
}
```

Solution:

Shortly:

- the first returns a copy of the struct,
- the second a pointer to the struct value created within the function,
- the third expects an existing struct to be passed in and overrides the value.

Q3: Implement a function that *reverses a slice of integers* ☆☆☆

Topics: Golang

Answer:

```
func reverse(s []int) {
        for i, j := 0, len(s)-1; i < j; i, j = i+1, j-1 {
                s[i], s[j] = s[j], s[i]
        }
}

func main() {
    a := []int{1, 2, 3}
    reverse(a)
    fmt.Println(a)
    // Output: [3 2 1]
}
```

Q4: What is the idiomatic Go equivalent of C's *ternary operator*? ☆☆☆☆

Topics: Golang

Answer:

Suppose you have the following ternary expression (in C):

```
int a = test ? 1 : 2;
```

The idiomatic approach in Go would be to simply use an `if` block:

```
var a int

if test {
  a = 1
} else {
  a = 2
}
```

For inline expression you could also use an immediately evaluated anonymous function:

```
a := func() int { if test { return 1 } else { return 2 } }()
```

Q5: What might be wrong with the following small program? ☆☆☆☆

Topics: Golang

Problem:

```
func main() {
    scanner := bufio.NewScanner(strings.NewReader(`one
two
three
four
`))
    var (
        text string
        n    int
    )
    for scanner.Scan() {
        n++
        text += fmt.Sprintf("%d. %s\n", n, scanner.Text())
    }
    fmt.Print(text)

    // Output:
    // 1. One
    // 2. Two
    // 3. Three
    // 4. Four
}
```

The program numbers the lines in a buffer and uses the `text/scanner` to read the input line-by-line. What might be wrong with it?

Solution:

First, it is not necessary to collect the input in the string before putting it out to standard output. This example is slightly contrived.

Second, the string text is not modified with the `+=` operator, it is created anew for every line. This is a significant difference between strings and `[]byte` slices — strings in Go are non-modifiable. If you need to modify a string, use a `[]byte` slice.

Here's a provided small program, written in a better way:

```
func main() {
    scanner := bufio.NewScanner(strings.NewReader(`one
two
three
four
`))
    var (
        text []byte
        n    int
    )
    for scanner.Scan() {
        n++
        text = append(text, fmt.Sprintf("%d. %s\n", n, scanner.Text())...)
    }
    os.Stdout.Write(text)
    // 1. One
    // 2. Two
    // 3. Three
    // 4. Four
}
```

That is the point of the existence of both `bytes` and `strings` packages.

Q6: What is the difference, if any, in the following two slice declarations, and which one is more preferable? ☆☆☆☆

Topics: Golang

Problem:

```
var a []int
```

and

```
a := []int{}
```

Solution:

The first declaration does not allocate memory if the slice is not used, so this declaration method is preferred.

Q7: When go runtime allocates memory from *heap*, and when from *stack*? ☆☆☆☆☆

Topics: Golang

Answer:

Stack memory is allocated:

- For small objects whose life cycle is only within the stack frame
- For small objects that could escape to heap but actually inlined

Heap memory is allocated:

- For small objects that will be passed across stack frames
- For big objects (>32KB)

Q8: What is the `malloc` threshold of `Map` object? How to modify it? ☆☆☆☆☆

Topics: Golang

Answer:

The default limit is 128. It can be modified by changing the value of `maxKeySize` and `maxValueSize` in runtime.hashmap.

Q9: How to *compare two interfaces* in Go? ☆☆☆☆☆

Topics: Golang

Answer:

You can compare two interfaces with the `==` operator as long as the underlying types are “simple” and identical. Otherwise the code will panic at runtime:

```
var a interface{}
var b interface{}

a = 10
b = 10
println(a == b)
// Output: true

a = []int{1}
b = []int{2}
println(a == b)
// Output: panic: runtime error: comparing uncomparable type []int
```

Both structs and interfaces which contain maps, slices (but not functions) can be compared with the `reflect.DeepEqual()` function:

```
var a interface{}
var b interface{}

a = []int{1}
b = []int{1}
println(reflect.DeepEqual(a, b))
// Output: true

a = map[string]string{"A": "B"}
b = map[string]string{"A": "B"}
println(reflect.DeepEqual(a, b))
// Output: true

temp := func() {}
a = temp
b = temp
println(reflect.DeepEqual(a, b))
// Output: false
```