Go Short Interview Questions Explained In Color

REVISION 1

Hawthorne-Press.com

Go Short Interview Questions Explained in Color

Published by

Hawthorne-Press.com
916 Adele Street
Houston, Texas 77009, USA

Table of Contents

Go Short Interview Questions Explained in Color

Introduction

In the process of interviewing for various Golang positions, I have experienced a number interview code tests. Some are long form type tests, where you were given an hour or up to a half a day to complete. Most are short tests given during an interview of 30 to 45 minutes.

Most of the time you are given five to ten minutes to outline your solution. In order to make these examples more complete, I have cleaned up the code. While some of these are original code, others are standard problems that you will encounter.

These short test solutions are often simplistic in nature since they produced under time constraints. This is expected by experienced interviewers.

Go Channel Question

Write a program that calls two go routines where one sends a random number into a channel and other prints the results from that channel.

First, we will write a simplistic solution to the problem suitable for an interview test. Then we will write a more sophisticated answer that is more in keeping with standard practice.

A Simplistic Working Program

We write two functions to support the required go routines. One generates a random integer and sends it to the channel specified in the parameter list. The second function reads the channel given by the input parameter and prints the result to standard output.

The main routine will have seed the random number generator and create an unbuffered integer channel. Then loop over the calls to the go routines for some number of cycles. Since the go routines will be stacked up waiting for execution, we must stop the main routine from leaving immediately, thus we sleep for 10 seconds.

The resulting program is as follows:

```
package main

import (
    "fmt"
    "math/rand"
    "time"
)

func getRand(ch chan int) {
    ch <- rand.Intn(5)
}
func printRand(ch chan int) {
    fmt.Println("Rand Number = ", <-ch)
}

func main() {
    rand.Seed(63)
    ch := make(chan int)
```

```
    for i := 0; i < 5; i++ {
        go getRand(ch)
        go printRand(ch)
    }
    time.Sleep(10 * time.Second)
}
```

A Standard Practice Version

There is one glaring problem with the first program. To prevent the main program from exiting before the Go routines finish, it simply sleeps for ten(10) seconds. While this works, the side effect is that the program prints the five random numbers and does nothing until the sleep period completes. Ideally, we want to exit immediately after the last number is printed. The standard answer for this problem is to provide a mechanism to shut down both the Go routines and the main function when processing is complete. This can be accomplished by creating a channel to signal these routines to exit. In our program, this channel is labeled **done**.

The second issue is creating the functions to support the Go routines. This is probably a matter of the developers taste. I feel that in a case like this, anonymous Go routines are a better choice.

The third issue is who decides when to terminate the program. The function generating the random numbers is effectively free running, depending on the receiver to synchronize the process. Therefore, the receiving function decides when to terminate the program by sending a message into the **done** channel.

The random number generator routine is in red and the receiver routine is in purple.

```
package main

import (
    "fmt"
    "math/rand"
)

func main() {
    rand.Seed(63)
    ch := make(chan int)
    done := make(chan bool)

    go func() {
        for {
            select {
            case ch <- rand.Intn(5):            // Create and send random number into channel
            case <-done:                        // If receive signal on done channel - Return
                return
            default:
            }
        }
    }()
```

```
    go func() {
        for i := 0; i < 5; i++ {
            fmt.Println("Rand Number = ", <-ch)   // Print number received on standard output
        }
        done <- true                              // Send Terminate Signal and return
       return
    }()

    <-done                                        // Exit Main when Terminate Signal received
}
```

The Fibonacci Sequence

This one you should learn by heart. It is one of the most often used interview questions. The formula for generating the Fibonacci sequence is as follows;

$$F_n = F_{n-1} + F_{n-2}$$

We will present two solutions for this problem. The first is an idiomatic Golang solution using channels and a *Generator Pattern*. It also uses a modified "look ahead" method of generating the values:

$$i , j = i+j , i$$

Solution One

The *generator pattern* is a common Golang construction. It is generally a free running loop that is controlled by the channel receiver. It consists of an anonymous Go routine that generates a sequence of values and sends them to a channel. It is enclosed by a function that returns the created channel to the caller of the function.

The main routine calls the enclosing function, **fib_generator(),** and receives a channel in return. This channel will return a sequence of integers until the main routine exits. Since the generator channel is unbuffered , the loop in the main routine will be in lock step with the generator. The anonymous Go routine is shown in red and runs independently.

```
package main

import (
    "fmt"
)

func fib_generator(done) chan int {
    c := make(chan int)                       // Create return channel
    go func() {
        for i, j := 0, 1; ; i, j = i+j, i {
            c <- I                            // Send generated value into the channel
        }
    }()
    return c                                  // return generator channel
}
func main() {
    c := fib_generator()
    for n := 0; n < 12; n++ {
        fmt.Println(<-c)                      // Print channel value
    }
}
```

Solution Two

This solution is a much simpler program. It should be considered if you are asked to produce a Fibonacci sequence program.

It uses a reentrant function, **Fibonacci(),** to generate the sequence values. Startup is provided by simply returning any input value that is less than two(2). Any input value **'n'** greater than or equal to two(2) is used to as follows:

return Fibonacci(n-1) + Fibonacci(n-2).

The complete program is as follows:

```
package main

import (
    "fmt"
)

func Fibonacci(n int) int {                 // Reentrant Fibonacci Function
    if n < 2 {
        return n
    }
    return Fibonacci(n-1) + Fibonacci(n-2)  // Calls itself!
}

func main() {
    for n := 0; n < 20; n++ {
        fmt.Print(Fibonacci(n), " ")        // Print twenty(20) space separated Results
    }
    fmt.Println()

}
```

Interfaces

Interfaces provide a way to describe the behavior of objects. Most interfaces contain only a couple of methods and this is normal for Golang. Complex Interfaces are often created by *embedding* a number of smaller interfaces within such an interface. The thing to remember is that interfaces are values that represent a fixed set of methods. An interface variable can store any concrete (non-interface) value as long the value implements the interfaces methods.

The Empty Interface

The special case is the empty interface (shown in red below). Since an empty interface contains no methods, then it implements all zero(0) methods of any type. The following fragment illustrates this principle.

```
// a is an Empty Interface
var a interface {}
var I int = 5
s:= "This is a string"
// The following are all valid statements
a = i
a = s
```

The Stringer Interface

The most often encountered demonstration of interfaces is the *stringer interface.* This interface is often used to provide special string formatting for particular values. The typical interface signature is as follows:

```
type stringer interface {
    String string          // The implementations will return a string!
}
```

The following is a basic implementation of this type of interface:

```
package main

import (
    "fmt"
    "strconv"
)

// This defines a Stringer interface with one method: "String() string"
type Stringer interface {
    String() string
}

// Defines an unsigned 64 bit number
type Binary uint64

// Defines a method "String() string" for a value of Binary
func (i Binary) String() string {
    return strconv.FormatUint(i.Get(), 2)
}

// Defines a method "Get() uint64" for value Binary
func (i Binary) Get() uint64 {
    return uint64(i)
}

// Main program
func main() {
    b := Binary(356)        // Create Unsigned Binary value for the Decimal Integer 356

    s := Stringer(b)        // Calls interface "Stringer" on "b"

    fmt.Println(s.String()) // "s" is an Stringer Interface Type
}
```

The result of running this program is the result: **101100100.**

However, what if you desired a different layout, say grouped in threes with a separator: **101|100|100**.

To accomplish the formatting above we simply change the implementation of the stringer interface.

```
   func (i Binary) String() string {
          --- implementation ---
  }
```

The new implementation is:

```
func (i Binary) String() string {
    s := strconv.FormatUint(i.Get(), 2)     // Convert Decimal to a binary string representation
    b, count := &bytes.Buffer{}, len(s)%3   // Create a bytes buffer and the initial separator count

    for i, r := range s {                   // Step through string
        if i > 0 && count == i%3 {          // If count equal current modulo of 3 insert separator
            b.WriteRune('|')
        }
        b.WriteRune(r)                      // Insert current value of range loop
    }
    return b.String()                       // Returns bytes buffer as a string
}
```

The tricky part of this process is that we are actually computing the separator position from right to left. That means we have to compute the position of the first separator from the right side of the string. This is accomplished by the variable **count**. If the length of **'s'** is divisible by three then count will be zero(0) and the first separator will be at **'i%3'**. For the string **101100100,** the separator is inserted *before* index **3** and index **6**. The index value of zero(0) is always excluded! The result is **101|100|100.**

I will leave to the reader to figure what the indexes are for various values.

Job Queues

The interviewer asks how would you implement a simple Job Queue? The pseudo code outline is as follows:

- Setup a channel for job requests
- Setup channel for done messages
- Setup an anonymous go routine to insert job requests
- Setup an anonymous go routine process requests
- Setup a control function to terminate jobs as they complete

In a more complete example, the Job Channel might be a structure containing the data, possibly processing instructions, and a done channel to terminate the jobs on completion.

However, in this example, we create a buffered done channel with a length equal to the number of jobs. This serves the purpose of absorbing done messages equal to the number of jobs. The real purpose of the done channel is to synchronize the two go routines and we are simply are dumping them into the done queue after they serve the purpose of synchronization.

```
package main

//
// Demo of "jobs" queue
//
import (
    "fmt"
)
```

```
func main() {

    jobList := [...]string{"job1", "job2", "job3", "job4"}

    jobs := make(chan string)
    done := make(chan bool, len(jobList))

    go func() {
        for _, job := range jobList {
            jobs <- job            // Blocks waiting for a receive
        }
        close(jobs)
    }()

    go func() {
        for job := range jobs {    // Blocks waiting for a send
            fmt.Println(job)       // Do one job
            done <- true           // Send DONE!
        }
    }()

    for i := 0; i < len(jobList); i++ {
        <-done                     // Blocks waiting to receive len(jobList)done messages
    }
}
```

Function Literals And Closures

All functions are in a sense *closures.* Top-level functions "close" over package level variables and thus are not that interesting. However, non-top level functions close over those variables and the variables defined inside the function. They do this each time these variables are created.

The following code we have a *Closure* in the form of an anonymous function literal created by calling the enclosing named function. Each time the enclosing function is called, it returns a reference to the function literal. Each time the variable containing the returned reference is executed, we perform the anonymous function.

The anonymous function literal is shown below in red.

```
package main

import "fmt"

func intSeq() func() int {
    i := 0                      // Initialized only when intSeq() is called!
    return func() int {
        i += 1
        return i
    }
}
func main() {
    nextInt := intSeq()        // nextInt is now a reference to the anonymous function
    fmt.Println(nextInt())     // nextInt() increments 'i' each time is called
    fmt.Println(nextInt())
    fmt.Println(nextInt())



newInts := intSeq()               // nextInts is a new reference to the anonymous function

fmt.Println(newInts())            // Prints '1' because this is a new reference
}
```

Defer Statements And Function Literals

Do you know what is the output of the following program and why?

```
package main

import (
    "fmt"
)

func main() {
    for i := 0; i < 3; i++ {                        // Loop 3 times with count
        defer func() { fmt.Println(i) }()           // Deferred function as closure
    }
    for i := 0; i < 4; i++ {                        // Loop 4 times with count
        defer func(n int) { fmt.Println(n) }(i * 2) // Deferred function as closure
    }
    fmt.Println("Main Done")                         // Print Done Message
}
```

The defer statement defers the execution of a function until the surrounding function returns. In this case, the first thing printed is “Main Done”. However, what happens next?

The difference between the previous program and the one above is that the defer statements ‘Expression’ must always be a *function* call! It is the difference between **defer f** and **defer f()**, and only the second form is allowed.

The first thing to consider is that in both cases the deferred statements will be executed in reverse order.

The second thing is the expression list between the parentheses, which may be empty, greatly affects the outcome:

defer func() { fmt.Println(i) }() // Prints the value of ‘i’ the moment the *defer statement executes.*

defer func(n int) { fmt.Println(n) }(i * 2) // Prints the value of ‘i’ when the *closure statement executes.*

The result is as follows:

```
Main Done
6           // Result of Immediate closure execution
4
2
0
3           // Result of Deferred Execution – ‘i’ is already at 3 when the deferred closure executes
3
3
```

Predefined Constants

What and how many predefined constants are part of the Golang Specification.

Only Three: **true**, **false**, and **iota**.

In Golang, **bool** values may only be **true** or **false** and may not be converted directly to a numeric type. These bool operators only support the values True and False. Uninitiated bool objects are set to False.

The **iota** constant is a different beast. It represents successive untyped integer constants in Golang. The actual value of iota is incremented by one(1) with each successive constant entry and is always initialized as zero. The following example shows the power of this feature.

```
type ByteSize float64

const (
    _           = iota             // ignore first value by using blank identifier (Value was zero)
    KB ByteSize = 1 << (10 * iota)  // Defines an repeatable expression
    MB
    GB
    TB
)
```

Each successive entry is shifted by 10 binary digits, i.e. the expression **1 << (10 * iota)**.

A candidate will often by asked state the results of various iota expressions.

Code That C/C++ And Java Programmers Get Wrong

What is wrong with the following program?

```
package main

import "fmt"

func main() {
     var a int8 = 3
     var b int16 = 4

     sum := a + b

 fmt.Println(sum)
}
```

Developers that are used to implicit promotions and type conversions will often get this question wrong! Golang does not implicitly convert between any intrinsic or custom type, including "obvious" numeric types. The code above generates the following error:

invalid operation: a + b (mismatched types int8 and int16)

The correct code for the sum operation is **sum := a + int8(b).**

What is the output of the following programs?

```
package main

import "fmt"

func main() {

 oSlice := []int{1,3,5,4}
 eSlice := []int{2,4,6,8}
 copy(eSlice, oSlice[2:])
 fmt.Println("eSlice = ", eSlice)
}
```

Do you know why the **eSlice** is equal to **[5 4 6 8]**? If not, you need to study slice mechanics.

```
package main

import "fmt"

func main() {
 x := 1
 y := &x
 fmt.Println(*y)
 *y = 2
 fmt.Println(x)
}
```

The answers are **1** and **2** for the program above. While Golang does not allow pointer arithmetic as in C/C++, it does have **&** (Address of) and ***** (Address Deference) operators. Thus ***y** references the address of the variable **x.**

In the same vein, consider the output of the program below:

```
package main

import "fmt"

func main() {
 x := 1
 incr(&x)
 fmt.Println(incr(&x))
}

func incr(p *int) int {
 *p++
 return *p
}
```

The program above is using address operators to pass by reference to a function. This allows the input parameter to be modified "in place" by passing just its address to the increment function. Notice that the construction ***p++** is modifying the data not the address. *There is no pointer arithmetic in Golang.*

A Final Word

Strive for simplicity in your code. The Occam ’s razor principle applies as strongly for software as it does for many other occupations. In most situations, giving up a little in efficiency to simply your code is often the right decision. Code Simplicity reduces the chances of subtle errors.

As always, comments are welcome at hawthornepresscom@gmail.com.

As to why I have an email address on **gmail** instead of on my website, has to do with AT&T and their inability to handle the concept that not everyone is running a Microsoft OS and my stubborn insistence on running Linux.

Published by

Hawthorne-Press.com
10310 Moorberry Lane
Houston, Texas 77043, USA

Revision History

Date	By	Section	Changes
05/25/2016	C.E. Thornton	All	Initial Document
05/27/2018	C.E. Thornton	Update Address	Update Address