GO Concurrency Basics and Patterns Explained in Color

REVISION 1

Hawthorne-Press.com

Go Concurrency Basics Explained In Color

Published by

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

Contents

GO CONCURRENCY BASICS AND PATTERNS EXPLAINED IN COLOR

This Document should help new users of the Go Language see the relationships involved in concurrency issues. The code presented here is from a Google talk on "Go Concurrency Patterns" and additional concepts from "Advanced Go Concurrency Patterns".

The code and some textual information have been modified and comments added to guide the new user. The comments are extensive and often redundant, but they are offered here to assure that the user understands how every part of the program functions.

This document builds from simple concepts to more advanced topics. Also included are sections on important design patterns.

All code has been tested with go 1.6 on an Ubuntu 64.biit 12.04 System running Gnome.

CHANNELS AND CONCURRENCY

We will start our journey with a toy program called **boring.go**.

```
package main

import (        // Import Supporting Packages
    "fmt"       // Print Formatting
    "math/rand" // Math - Random Number Functions
    "time"      // Time - Time Functions
)

func main() {        // Program starts here
    boring("boring") // Call Subroutine
}                    // Exits the program

func boring(msg string) {    // A function with string parameter
    for i := 0; ; i++ {      // Infinite loop with loop count
        fmt.Println(msg, i)  // Print Input String and count
        // Sleep and Random amout of time
        time.Sleep(time.Duration(rand.Intn(1e3)) * time.Millisecond)
    }
}
```

With this basic function, we will explore concurrency with Goroutines. The first thing to change is to prefix the *boring* routine with the **go** Keyword**.**

If all you do is plug **go** in front of the boring function, the main program will exit before the boring Goroutine completes. This is because once a Goroutine is launched by the **go** command; the program continues and, in this case, exits. We will have to add code to wait for the Goroutine to

finish before exiting the program. One simplistic method is to print a message and wait a couple of seconds. The modified main routine is shown below:

```
func main() {
   go boring("boring")
   fmt.Println("I'm Waiting")
   time.Sleep(2 * time.Second)
   fmt.Println("You're boring, I'm leaving!")
}
```

NOTE*: Any modified code, or changes, in this and future examples will be shown in red. Comments will be shown in green. The unchanged code will be in blue.*

Obviously, just putting waits in your programs is not a viable solution.

Basic Channels

In order for Goroutines to function in a synchronized way, which is often very important, they must be able to communicate with each other and the controlling program. This is accomplished in **Go** with *Channels*.

The following are examples of basic channel operations:

```
// Declaring and initializing a channel passing an Integer!

var c chan int         // Defines a variables type
c = make(chan int)     // Create a Variable of type Channel of Integer

// Or more easily
c := make(chan int)    // Same result in one step

// Sending on a channel (Push an integer '1' into channel 'c'
c <- 1

// Receiving on a channel
// The direction of the "arrow" indicates the direction of data flow

value <- c     // Variable 'value' set to the value of channel 'c'
```

Now that we have the basics, we will rewrite our routine to use a channel to synchronize the Goroutine with the main program.

The boring routine is modified to create a string with an incrementing number embedded in it. It will continue to create such a string each time it is executed. It returns that string in a channel.

The Main function will call the boring function five times, each time waiting for the channel to return a string and printing the returned string.

```
package main

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

func main() {
   c := make(chan string)              // Create Channel of String
   go boring("boring!", c)             // Execute the boring Goroutine
   for i := 0; i < 5; i++ {            // Loop 5 times and generate a count
      fmt.Printf("You Say: %q\n", <-c) // Received Channel value is a string
   }
   fmt.Println("You're boring, I'm Leaving.")  // Format and Print result
}

func boring(msg string, c chan string) { // Function two string parameters
   for i := 0; ; i++ {                   // Infinite loop with counter
      c <- fmt.Sprintf("%s %d", msg, i)  // Format output string and
                                         // return it on channel parameter
      time.Sleep(time.Duration(rand.Intn(1e3)) * time.Millisecond) // Delay
   }
}
```

Channels come in two flavors, buffered and non-buffered. Non-buffered channels, for both send and receive, are blocking. When sending a value into a non-buffered channel it will block until a receiver is ready. Conversely, requesting a value from a channel will block until a value is available.

Buffered channels block only when the buffer is full. The user can send into a buffered channel without blocking until the buffer is full. You can receive from a channel without blocking as long as the buffer has values.

Before we continue with the boring routine, we have to discuss *Function Literals*, as they will play a part in improving the boring program.

Function Literal Pattern

The definition of a *Function Literal* is as follows:

```
   Func (ch chan int) { ch <- ACK} (reply)
```

The "func(ch chan int) { ch <- ACK }" is the definition of the closure.

The (reply) means "execute this closure with parameter reply". Note that the reply is an expression, *not necessarily a Go Channel*.

An example with a replyChan is:

```
   func(n int) { fmt.Println(n) }(reply)
```

If the function has a parameter, the **(reply)** must be specified.

The following code illiterates the function literal forms:

```
    nn := 77
    go func() { fmt.Println(nn) }()          // Result = 77

    go func(n int) { fmt.Println(n) }(88)  // Result = 88
```

The following program example illustrates the interesting properties of a closure:

```
package main

import (
    "fmt"
)

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

The result of running the above program is as follows:

Main Done
1
0
3
3
3

The *defer* statements are not executed until main is complete. They are then unwound in inverse order. However, the important point is the values printed by the defer functions.

Notice that the first two values printed are 1 and 0. These values were printed by the second loop and they use the value of ‘i’ that was present when the *closure was executed.*

The last three numbers are all three. These values are printed at the time the *actual defer statement executes*. The value of ‘i’ after the loop has completed and all the defer statements execute is 3!

Generator Pattern

A “Generator” is a function the returns a channel. Our boring function executes and waits for a channel to be returned. This makes a good candidate for turning into a Generator.

The boring function runs an infinite loop using an *autonomous Go function* to create a message and return the message in the specified channel. Autonomous **Go** functions are another name for Goroutines executing as *Function Literals*.

The new versions of “main” and “*boring”* are shown below:

```
func main() {
   c := boring("boring!")  // ‘c’ variable is a function returning a channel
   for i := 0; i < 5; i++ {
      fmt.Printf("You Say: %q\n", <-c) // The function ‘c’ returns a channel
   }
   fmt.Println("You're boring, I'm Leaving.")
}

func boring(msg string) chan string { // returns receive-only channel of
strings
c := make(chan string)
go func() {               // we launch the Goroutine inside the function
   for i := 0; ; i++ {
      c <- fmt.Sprintf("%s %d", msg, i) // The string returned in a channel
      time.Sleep(time.Duration(rand.Intn(1e3)) * time.Millisecond)  // Delay
   }()
   return c               // returns channel to the caller
}
```

The Go programmer will find a number of different versions of the Generator Pattern and Autonomous Functions.

Now that you have a Generator function, you can turn it into a *Service*. The same service can be reused for different users. Examine the follow modification of the main function. Notice that all we do is rename the ‘chan’ and ‘msg’.

```
func main() {
   joe := boring("joe")   // Calls same boring generator as above.
   ann := boring("Ann")
   for i := 0; i < 5; i++ {
      fmt.Println(<-joe) // Sync on receive of Channel from generator.
      fmt.Println(<-ann)
   }
   fmt.Println("You're boring, I'm Leaving.")
}
```

This function acts in sequence, each message in turn. However, if we wish to handle messages with different timing, we are going to have to use a *Fan-In Function*. This again is a basic pattern the will be used in many places.

Fan-in Pattern (Or Channel Multiplexing)

A Fan-In function is a modification of the generator function. It has two or more generators, as function literals, which copy "their" input to the common return channel.

First, notice the new function literal, in purple, it simply contains an infinite loop that copies a specific input to the common return channel. It will copy input to output as long as there are values to copy and a channel to receive the results.

```
func fanin(input1, input2 <- chan string) <- chan string {
   c := make(chan string)
   go func() { for {c <- <-input1 } }() // Copy input1 to chan 'c'
   go func() { for {c <- <-input2 } }() // Copy input2 to chan 'c'
   return c
}
```

The code above is perfectly correct and executes without a problem. However, it is not coded in the *Go idiom*. The Go FMT Command will rearrange the code as follows:

```
// Formatted Fan-in Function
func fanin(input1, input2 <-chan string) <-chan string {
    c := make(chan string)
    go func() {
        for {
            c <- <-input1
        }
    }()        // Copy input1 to chan 'c'
    go func() {
        for {
            c <- <-input2
        }
    }()        // Copy input2 to chan 'c'
    return c
}
```

This fan-in pattern can be used for any number of channels. A feature of this pattern is that the inputs do not have to wait on each other. The channel ‘c’ may block waiting for a receiver, but there is no receiving order imposed by this technique. Each channel can run at its own rate. The following modification of the main routine is below:

```
func main() {
   c := fanin(boring("joe"), boring("ann"))
   for i := 0; i < 10; i++ {
      fmt.Println(<-c)
   }
   fmt.Println("You're boring, I'm Leaving.")
}
```

The result of this new fan-in program is:

joe 0
ann 0
ann 1
…
joe 4

Returning Fan-in Channels in Lock Step

At this point, we have used channels for either signaling or sending data. Channels are first-class values and can do both. This next program is an example of passing a structure with data and a wait flag. Pay attention to the parts in red in the following program:

```
package main    // Generator "Service" with wait flag

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

type Message struct {  // Message Structure:
   str string          // Data
   wait chan bool      // Wait Flag
}

func main() {
   c:= fanin(boring("Joe"), boring("Ann"))
      for i := 0; i < 5; i++ {
      msg1 := <-c; fmt.Println(msg1.str)
      msg2 := <-c; fmt.Println(msg2.str)
      msg1.wait <- true
      msg2.wait <- true
   }
   fmt.Println("You're boring, I'm Leaving.")
}

func boring(msg string) <-chan Message { // returns a channel of strings
c := make(chan Message)                  // Create Message Structure
waitforit := make(chan bool)             // Wait Chan: Shared by all messages

go func() { // we launch the goroutine inside the function
   for i := 0; ; i++ {
      c <- Message{ fmt.Sprintf("%s %d", msg, i), waitforit}  // Return Msg
      time.Sleep(time.Duration(rand.Intn(1e3)) * time.Millisecond) // Delay

      <- waitforit  // Wait Here until waitforit channel has a value!
      }
   }()
   return c // returns channel to the caller
}
```

```
func fanin(input1, input2 <- chan Message)  <- chan Message {
   c := make(chan Message)
   go func() { for {c <- <-input1 } }() // Copy input1 to chan 'c'
   go func() { for {c <- <-input2 } }() // Copy input2 to chan 'c'
   return c
}
```

The first time a new user encounters this type of function, they usually do not understand it. Therefore, we will discuss the concepts buried primarily in the new boring routine.

The first three lines are straightforward. A Function Definition taking a string parameter and returns a channel of type *Message*. Thus, we create the channel 'c' of type Message. We also create a new channel of type *Bool* named '*waitforit*' and store it in the Message Structure as the 'wait' channel.

After launching the anonymous Goroutine, Message channel 'c' is returned to the callers of the boring function.

The rest of the magic is inside the anonymous Goroutine, a Function Literal. It enters an infinite loop that will send formatted messages into channel 'c'. When the main function receives the message from boring, it prints the string in the Message Structure, and sends a *True* into the Message '*wait*' channel. The boring routine waits for a value from '*waitforit*' channel l and return to the top of the loop. (Remember that '*Wait*' and '*Waitforit*' are the same channel). This cycle continues until the Goroutine is stopped.

The main function calls boring for each message and then waits for both replies. The printed results always occur in pairs, a message from "Joe" and "Ann". However, the order of each pair is undetermined due to the random delay each time a message is processed.

Notice that both channels share the '*wait*' channel. This would be a problem except for the serial lock-step nature of this program. The program has to wait until both '*wait*' Channels have been received before the next messages are sent!

Select Statement Pattern

Select Statement is a built-in feature of **Go**.

This statement is a similar to a **switch** statement but controls execution based on what actions can proceed depending on the state of concurrency operations. In other words, it allows selecting actions based on which channels are in a non-blocking state. If multiple channels are unblocked, the one executed will be randomly selected.

Using the select statement, we can now simplify the fan-in function.

The new iteration of the fan-in function uses the **Go select** statement to merge the input channels.

One of the first thing an inexperienced **Go** programmer will be concerned about is the channel 's' is defined twice, once in each **case** statement. This was included into the program to illustrate a

point about *Scope*. Each **case** statement has its own scope. Just as a variable can be redefined inside a function, it can be redefined inside the scope of a **case** statement.

Further, the 's' Channels are completely separate entities. They are different channels!

```
func fanin(input1, input2 <-chan Message) <-chan Message {
    c := make(chan Message)        // Create Message Channel 'c'

    go func() {                    // Example of an anonymous function
        for { // Infinite loop
            select {               // Select the input that is unblocked
            case s := <-input1:    // Receive Input 1 when unblocks
                c <- s             // Copy channel 's' to channel 'c'
            case s := <-input2:    // Receive Input 2 when unblocks
                c <- s             // Copy channel 's' to channel 'c'
            }
        }
    }()       // End of Anonymous Goroutine

    return c  // Return Message Channel 'c' to Caller
}
```

Time-out Pattern

The function *time.After* returns a channel that blocks for a specified time and then after that interval, the channel delivers the current time once. An example of this is shown below:

```
func main() {
   c := boring("Boring")                  // Call boring function
   for {        // Infinite Loop
      select {                            // Select on Channels
      case s := <-c:                      // Get string from channel 'c'
          fmt.Println(s)                  // Print the string
      case <-time.After(1 * time.Second): // Will timeout after one second.
        fmt.Printf("You're too slow.")    // Print Message due to timeout
        return                            // Exit Program
      }
   }
}
```

One point you should take away from this example is that Channels are used in many package API's. **Go** is built around the notion that information is shared between concurrent processes by *Communication*!

The programmer also can create the timeout outside the **select.** A channel variable is a first-class object.

The user could use it to create an overall program "death clock", if such a bizarre thing was required.

```
func main() {
    c := boring("Boring")                  // Create Channel for boring
    timeout := time.After(5 * time.Second) // Create a Five Second timeout
         ::          ::       ::
    case <- timeout:                       // Place timeout in select group
         ::          ::       ::
```

Handshake Synchronization Pattern

The following is a modified version of the boring program. It synchronizes with the main function through a channel passed to the boring routine.

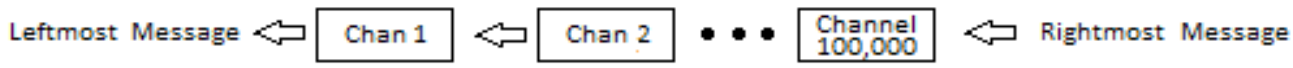
The main function creates a channel ‘quit’ and passes it along with a string to the boring routine in the message channel ‘c’. The boring routine will create messages and send them back on that channel until it receives a message on the ‘quit’ channel. At which time it executes the ‘cleanup’ function, prints a quit message and returns, thus terminating the boring routine.

The main function starts the boring routine, allows it to run for a random amount of time and then sends a string into the ‘quit’ channel.

```
package main

import  "fmt"
import   "math/rand"

func main() {
   quit := make(chan string)             // Make ‘quit’ channel
   c := boring("Boring", quit)           // Create Boring Function variable
   for i := rand.Intn(10); i >= 0; i-- { // Loop for random amount of time
       fmt.Println(<-c)                  // Print returned message
   }
   quit <- "Bye!"                        // At end delay – Terminate boring
   fmt.Printf("Joe says: %q\n", <-quit)  // Print main function goodbye msg
   }

func cleanup() {                         // Clean up function
   fmt.Println("Cleaning Up!")           // Print goodbye message
}

func boring(msg string, quit chan string) <- chan string {
c := make(chan string)                    // create return channel
go func() {                               // Autonomous Function
   for i := 0; ; i++ {                    // Infinite Loop
      select {
      case c <- fmt.Sprintf("%s %d", msg, i): // Send message into channel
        // do Nothing
      case <- quit:                       // Wait for ‘quit’ Message
        cleanup()                         // Clean up and exit boring
        quit <- "See you!"
        return
      }
    }
  }()
 return c    // returns channel to the caller
}
```

Pipeline Pattern

This particular demo program is designed to show a couple of interesting features of the **Go** Language. The first is that channels can be chained together in a *Pipeline*!



The first thing to understand is that all the Goroutines and associated Channels were *created before the transfer starts.* One of the main **Go** concepts is that Goroutines are very lightweight. Before this process begins we have created more than100,000 Goroutines and Channels. When we run the program the entire time to setup this chain and execute it will take less than 2 -3 seconds on a modern desktop system.

The code is show below:

```
package main

import "fmt"

func f(left, right chan int) { // Primary transfer function
    left <- 1 + <-right        // Left Channel = 1 + Right Channel
}

func main() {
    const n = 100000           // Number of Goroutines to execute
    leftmost := make(chan int) // Make Primary Channels
    right := leftmost          // Initialize to leftmost Channel
    fmt.Println("Right = ", right)
    left := leftmost // Initialize to leftmost Channel
    fmt.Println("Leftmost = ", leftmost)
    for i := 0; i < n; i++ {
        right = make(chan int) // Make another new right Channel
        go f(left, right)      // Create and call new GO routine
        left = right           // Transfer Right Channel to Left
    }
    // Uncomment next line to indicate the end of the setup period
    //   fmt.Println("100,000 GO Routines Created - Start the chain")

    go func(c chan int) { c <- 1 }(right) // Send 1 into Rightmost Channel
    fmt.Println(<-leftmost)               // Print leftmost Channel Value

}
```

The function *'f'* is a transfer function that copies the "right" channel to the "left" channel after adding one to its value.

The Main function stores a created channel in both 'left' and 'right' variables. It then enters a loop that creates a new right channel and calls function 'f'. At this point, the **for** loop creates all the Goroutines and channels. The channels remain blocked waiting for the *last right channel to receive a value.*

The main routine then uses the Function Literal 'func' to send a '1' into the rightmost channel, thus initializing the rightmost channel and kicking off the transfer loop. The rightmost channel is the last "right" channel after the initializing loop has completed. This initializing line of code is as follows:

```
Go func(c chan int) {c <- 1}(right)        // Calls an anonymous function literal
```

Since func has a parameter, the closure must supply a reply expression, (right). After substituting 'right' for the channel parameter 'c', the effective statement is as follows: If you doubt this, make the substitution.

```
right <- 1
```

Producer-Consumer Pattern

When there is an on-going process that is generating data, something is going to have to be a consumer. Whether that is a data file, another process, or a displaying something, the data is going to be consumed. It is generally a good idea to make such dual co-dependent processes explicit.

```
package main

import ("fmt")

var done = make(chan bool)      // Unbuffered Done Channel
var msgs = make(chan int)       // Unbuffered Message Channel

func produce () {               // Producer Function
    for i := 0; i < 10; i++ {
        msgs <- I               // Generate Data (PRODUCE MESSAGES)
    }
    done <- true                // End of message generation
}

func consume () {               // Consumer Function
    for {
      msg := <-msgs             // Get Message
      fmt.Println(msg)          // Print Message (CONSUME MESSAGES)
   }
}

func main () {                  // Main Function
   go produce()                 // Execute Producer  (Goroutine)
   go consume()                 // Execute Consumer  (Goroutine)
   <- done                      // Wait for value on “done” channel
                                // -- Notice: The value is discarded
}
```

The code above is a simple case, but it makes the Producer-Consumer Pattern obvious. The connection between them is a unbuffered channel. The data can be produced at a different rate than the consumption process. If the channel is unbuffered as above, then the processes proceed in lock step. If you buffer the channel then the “done” code will have to be more complex.

Error Reporting Pattern

A good programmer will separate error checking, reporting, and normal program logic. These categories should be handled in separate pieces of code. For example:

```
func httpRequestHandler(w http.ResponseWriter, req *http.Request) {

    err := func () os.Error { // Function Literal Variable for Error Checking

        if req.Method != "GET" {                          // Check Error
            return os.NewError("expected GET")            // Log Error
        }

        if input := parseInput(req); input != "command" { // Check Error
            return os.NewError("malformed command")       // Log Error
        }
    } ()                         // Return Nil if ok, otherwise return error

    if err != nil {              // Using 'err' invokes the processing above!
        w.WriteHeader(400)       // Display Error 400
        io.WriteString(w, err)   // Display Error String
        return                   // Return to Caller
    }

    doSomething() ...            // Normal Processing
...
```

In the 'real' world, the error code would be more extensive to allow the same error construct to be used in more places.

With the variable *err* available the processing of errors becomes much simpler and more obvious.

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 walks of life. In most situations, giving up a little in efficiency to simplify 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.

Revision History

Date	By	Section	Changes
03/13/2016	C.E. Thornton	All	Initial Document
05/27/2018	C.E. Thornton	Copyright	Update Address