Compare Functional Programming of Go and Lua

Table of Contents

Introduction

This article compares two programming languages - **Lua** and **Go(lang)**. It covers the most important aspects of these two languages:

- High level overview and use case
- Deeper dive into typing of both languages
- Objects and explain tables in Lua
- Functions and exception handling
- Goroutines and Coroutines

What is Lua?

Lua is a programming language which was designed, implemented and released in 1993 by PUC-Rio, the Pontifical Catholic University of Rio de Janeiro in Brazil. Untill version 4 it had been licensed under BSD and since Version 5 it is under MIT.

In the first version Lua was created as a library. Over the time and continuous development it has grown to a real programming language.

It´s creators goal was to create a powerful, efficient, lightweight and embeddable scripting language. It is based on concepts of procedural, object-oriented , functional and data-driven programming and on data description.

Because of small footprint, is is easy to embed in platforms which support standard C compiler, it is a scripting language which is typed dynamically.

It´s easy to create a program written in two languages with Lua. Lua can be seen as the glue which holds together the hard parts of a program (which is maybe written in *C*).

Currently Lua is **the** leading scripting language in video games. For example "World of Warcraft" and "Angry Birds" use Lua. Because of its popularity Lua has won many awards like the "Front Line Award 2011".

What is Go(lang)?

Go respectively Golang is a programming language which was created in 2009 by Google. The purpose of Go is to be a system programming language which solve current requirements. It is also based on object-oriented and functional programming but combined with some other concepts. Some of those concepts are from the 1970s. But why use such old

stuff? In the 1970´s these concepts were developed in theory but had no real use case. That changed since multi-core CPUs, distributed computing and cloud solutions are widely used nowadays.

With Go Google created a language which should handle the new challenges which came with the big cloud hype and its requirements.

Google wanted to meet the following requirements:

- Compiled (cross compiler for OS: Mac, Windows, Linux and CPU: ARM, x86 + Amd64)
- Static type system with runtime support (reflection, dynamic types)
- Static linker (single binary) → ideal for Docker containers
- Short compile times (the entire Go codebase compiles in <10 seconds)
- Simple (less keywords like C - 25/32)
- Object-oriented, functional, parallel, modular (vgo) and versioned

These requirements were extended with the concept of Hoare´s CSP (1978) and Dijkstra´s guarded commands (1975) in order to get rid of a heavy concurrent approach of threads and to think in this way.

These challenges are solved by Go, and therefore it is now part of the cloud native landscape.

Comparison Table Go vs Lua

Feature	Go	Lua
Multiple Projects	Yes	Yes
Target audience	Cloud computing, Web Development, App Development, Distributed Systems, Embedded system, Systems Programming	Embedded system, Game Development industry
Release Date	2009	1993
Typing	Static	Dynamic
Programming paradigm	Concurrency Oriented, Object-oriented, Imperative programming, reflective	Imperative programming, Object-oriented, Functional, Metaprogramming
Supported VCS	Any	Git
Embeddable	Yes	Yes
Asynchronously	Goroutines, Channels, Multi-Way cconcurrent control, Locks (Usually not needed!)	Coroutines
Exception Handling	panic/recover	Lua ends the current chunk and returns to the application pcall (like try/catch)

Feature	Go	Lua
Public/Private	Implicit by default by Spelling/Writing	Privacy by Local Variables or keep Objects/Tables in Closure
Objects	By embedding and by interfaces	Tables
Interfaces	Yes	No
Compiled	Yes	Yes
Inheritance	Multiple by Interfaces	Multiple Prototyping

Typing

One of the biggest difference between Lua and Go is their typing. Lua is a dynamic typed scripting language and Go on the other hand is a simplified static typed language. Both concepts have their pros and cons which will not be discussed in this article. This article shows a high level overview about the concepts and how things are done in the 2 languages.

Assigning Variables

Lua and Go allows multi assignment like:

```
a,b = 1,2   // Lua
a,b := 1,2  // Go - uses ':=' to declare and set values
```

Very interesting in this example is that Go **knows** the type of the variable by declaring it with a value. This could also be done explicitly by:

```
a int
a = 1
```

but its still static.

Multi assignment allows funny things like to swap without a swap function

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

or to receive multiple return values of a function without complication handling (Exception Handling)

```
result, error = f()   // Lua
result, error := f()  // Go
```

Dynamic Typing - Lua

Languages with dynamic typing check the type during runtime. This means it is not important or the programmer does not have to care about which variable holds which type. A variable can handle values of different types (at different

times). The following example makes this clear:

```
a = 1
a = "one"
a = f()
...
```

There is no need for casting to get an int into a double/float or somewhere else. But there is a lot to do for the programmer, if he wants build a stable program. He has to check his variable during input, handling and output. He can´t be sure if the variable is a real numeric value or only a string, which only looks numeric. Even worse if you add to an int an object because in both variable was an int at program beginning. But during runtime one was overridden by any object.

On the other hand, if you know there are only correct types and variables, it is very easy to handle them. The casting operation always works correct, the code looks very clean, and one can handle different types the same way.

Static Typing - Go

For **Go** is it very important to have static typing with a lot of syntactic sugar. In Go one can't assign different types to a variable.

```
a := 1      // a will declared as a variable from the type integer an gets the value
```

```
// equivalent to
a int
a = 1
```

```
a = "one" // will occur a panic Type Error
```

This concept makes it very easy for developers to know which values he or she needs for calling functions or handling return values because at all times it is clear which variables, parameters or return values have to be handled.

Of course static typing has problems and ugly sides. If there is an array of objects and If it is clear that only integer values are in it, they have to be casted explicitly.

```
arr_1 := []any{2, 3, 4}
a ;= 1
a := a + arr_1[0].(int)
or
arr_2 := []int{1, 2, 3}
b := 2 + arr_2[0]
```

Another important aspect is that if one declares variables one must also declare the type. This is not a big deal, but it´s good to know and kind of syntactic sugar.

The static approach has also effects on functions. The parameters can only be declared with types. If the parameters are not be clear while creating the function or used in several ways, one has to cast explicitly like:

```
func foo (a string, b int) string{
    return "Some Value"
}
type Any interface{}
func foo_2(a Any, b Any) Any{
```

```
    x string
    b int
    x = a.(string)
    y = b.(int)
    return "Some Value"
}
result string
result = foo_2("Some", 2).(string)
```

The example shows that the function params can only be a string for *a* and an integer for *b*. *foo* must return a string. In the second function, one has to cast the params and return a value to fit the correct type. The main benefit is that the developer can easily see the correct type and use it in the correct way. IDE´s and finally the compiler are able to find type errors. In the worst case, in function *foo_2* still a type error occurs during runtime. Go allows type save casts but if it´s cast in the wrong type an error occurs.

Static typing helps developers to prevent errors. Readability and understanding of the code depends on the developer who uses it.

Let's take a look on Lua functions to see how it looks there:

```
function foo (n)
    n = n or 1
    n + 1
    return n
end
```

Without saying which concept is better (dynamic or static), can you answer the following questions?

- Which type has the parameter? How should *foo* be called?
- Will you return something? Which type will it have?
- Must the parameter be set?

There still exists documentation and the function parameter naming shows if it's needed and which type has to be used, but there is no compiler or IDE support for this kind of typing.

If foo is called like *foo("abc")*, an invalid-type-exception will occur during runtime and in the worst case the program will break.

One of the benefits of dynamic typing is that it is much easier to write, one doesn´t have to care about explicit typing and variables can be reused for multiple purposes. Code can be much smaller and during development the developer knows which types he uses. Anonymous functions are called in an explicit context where the types are known. Why types be defined? There is only one way to use them and it´s absolutely clear!

Objects in Lua (Go Objects will be discussed in other articles)

Objects in Lua are called *Tables*. They are kind of associative arrays. These arrays store different kinds of values. These values can be indexed by number or string. Tables have no fixed size as known from other languages and can grow dynamically during runtime. Take a look at how *Tables* can look like:

```
TableA = { 1, "One", boolean}
print(TableA[2])                                    // One - Lua starts to index by 1
TableB = { first = "ONE", second= 2 , third = false}
print(TableA["second"])                             // 2
TableC = {first = 1, second = 2, 3}
print(TableC["first"}, TableC.second, TableC[1])    // 1 2 3 - Because 3 is the first none named value!!!
```

This data structure will be used for every structure in Lua like ordinary arrays, symbol tables, sets, records and queues.

Tables in Lua are neither values nor variables, they are objects. As such there are no hidden copies or new tables created behind the scenes, the program manipulates tables by references.

If there is the need for a new table, one can just create it like in the example above. There is no need for a constructor.

To get the same behavior as for objects or classes in other languages the tables has to be extended with first-class functions. That can be done in several ways like:

```
LanguageTable = {
    de = "", en = "",
    New = function()
        helloT = {}
        for k, v in pairs(HelloTable) do
            helloT[k] = v
        end
        return helloT
    end,
    german = function(param)
        print(param.de)
```

```
    end,
    english = function(param)
        print(param.en)
    end
}
a = LanguageTable.New()
a.de = "Servus"
a.en = "HI"
a.german(a)                 // Servus
a.english(a)                // HI
```

or with more syntactical sugar and more functionality like a meta table

```
LanguageTable = {
    de = "", en = "",
    mt = {},
    New = function()
        helloT = {}
        setmetatable(helloT, LanguageTable.mt)   // setmetatable() came with Lua
        return helloT
    end,
    german = function(self) // convention
        print(self.de)
    end,
    english = function(self)
        print(self.en)
    end
}
LanguageTable.mt.__eq = function(lt1, lt2) // __eq stands for operator Equals
   return lt1.de == lt2.de and lt1.en == lt2.en
```

```
end
 LanguageTable.mt.__index = LanguageTable   // allows to call functions from super.
```

Let´s have a look on the example before showing us how it is used. In the *New/Constructor*-Function a *setmetatable*-Function (which came with Lua) appears and assigns the table to a newly created table which we return. That is used by overloading the operations during the __-Notation of *equals* and *index*. During equals it is, because it is a simple equals implementation, *index* is very interesting because it shows how *super()* from other languages is included. *index = LanguageTable* says, should there be no value in the current object, take value from *LanguageTable*.

```
a = LanguageTable.New()
a.de = "Servus"
a.en = "HI"
a:german()                  // Servus - :-Notation passes self as parameter
a:english()                 // HI
b = LanguageTable.New()
b.de = "Hallo"
b.en = "Hello"
print(a == b)              // false
c = LanguageTable.New()
c.de = "Hallo"
c.en = "Hello"
print(b == c)              // true
```

The usage shows a few little differences. With the *:*-Notation there is no need to pass the table into the function. The Equals operator now works for tables. And *index* allows to call *german* and *english* from the template table.

There is still a lot to say about tables in Lua but for this article it is sufficient.

More Characteristics of Lua and Go

Importing Modules

In Lua the creators say with a smile that the import is maybe to dynamic. The linking to the "math"-library is never checked. During execution it is there or the program throws an error:

```
local m = require "math"
print(m.sqrt(10))
```

Go has static linking. Special is that the whole Path to the imported Package/Library has to be written. If the developer uses a state of the art IDE in which, IDE handles the imports for the Developer, then he doesn´t have to care about it.

```
import "fmt"
fmt.println("Hello World")
```

Pointers / References / Call By Value

Lua doesn´t offer Pointers (of course internally it uses references to memory) depending on the data-type where references or values are copied. Lua handles allocation and deallocation of strings and other objects.

Or more simple: All types are passed by value, but functions, tables, userdata and threads are reference types. An exception is String which is immutable and will be handled as a reference to a newly created string. So it has same behaviors like a value type, but with better performance.

Go offers Pointers and all functionality which come with this opportunity.

Functions

Now the basics are clear. Let's take a deeper dive into the functions of both languages.

Scope

Lua uses lexical scoping. That means unlike global variables, local variables have their scope limited to the block where they are declared. A block is the body of a control structure, the body of a function or a chunk (the file or string with the code where the variable is declared). It´s the same for functions, so we can create typical closures like:

```
function sequence ()
    local i = 0
    return function ()
        i ++
        return i
    end
end
```

Go has nearly the same behavior except global variables, Go doesn´t contain them. The scoping of Go is called lexical blocks, which is similar to Lua´s lexical scope. The syntactic block is a sequence of statements enclosed in braces that surround the body of a function or loop. There is a lexical scope for the entire source code, called the universal scope: For

each package, file, function, loop, switch, switch-case, select and of course for each lexical scope. Imports are in the file level scope. Closures in Go look nearly the same as in Lua:

```
func sequence() func() int  {
    i := 0
    return func() int {
        i++
        return i
    }
}
```

As we can see both can handle closures and functions as first class values/citizens which means you can treat functions as values. Functions can be function parameters, return values (higher-order functions) or stored in variables.

Function Nesting

As expected both languages have function nesting like we see in **Lua**

```
function foo(x)
    function p(y)
        print(y)
    end
    p(2*x)
end
```

and **Go**

```
func foo(x int) {
    b := func(y int) {
        fmt.Println(y)
    }
    b(2 * x)
 }
```

Anonymous Functions

Beside function nesting there are in both languages anonymous functions

```
add = (function (x,y) return x+y end)   // Lua
add := func(x int, y int) int {         // Go
        return x + y
    }
```

Example Map-Function

The "canonical" example of a function that takes another function as a parameter is *map*. Unfortunately *map* does not come with **Lua**, so we'll have to code it ourselves.

```
function map(func, array)
    local new_array = {}
    for i,v in ipairs(array) do // ipairs returns simple said the key and value
        new_array[i] = func(v)
    end
```

```
    return new_array
end
```

This is a simple map implementation that only works with one array. But it works well:

```
return table.concat(map(double, {1,2,3}),",") // 2,4,6
```

Its very funny that **Go** as well as Lua don´t include a map function. Therefore, we compare the code ourselves:

```
func Map(foo func(interface{}) interface{}, arr []interface{}) interface{} {
    temp := new([]interface{})
    for _, v := range arr {
        *temp = append(*temp, foo(v))
    }
    return temp
}
```

In the Go example interface{} is used very often in order to use Map with every type. It looks very ugly and does not support the developer in how the function should be used. To use it more explicit and with the common *Object.Function()*-Notation, functions on Objects Map can be written for an explicit type.

```
   func (s *SomeType) Map(mapperFunction AnyInterface) SomeReturnValue {
    for i, el := range s.data {
        s.data[i] = mapperFunction(el)
    }
return s
   }
```

```
    s := make(SomeType)
    s.Map(AnyFunction)
```

Exception handling

Lua uses function nesting for its error handling. The *pcall*-Function (Protacted Call) takes a function as parameter and calls that function. *pcall* returns two values, an ok-Value in case all is ok :) and a second value with the error message should an error occur during calling the function . This is a very good example how function nesting and multi assignment work.

```
local ok, err = pcall(function() <block/error> end)
if not ok then
    print(err) // error handling
end
```

For simple semantic one only needs 2 functions – this shows that functional programming is a major part of Lua.

For **Go** its nearly the same. For functions which are intended to return an error, like http calls, that function has to return an error by design in case of an error. Therefore there is no need for wrapping functions into a *pcall* like:

```
ok, err := http.Get(url)
if err != nil {
    fmt.println(err) // error handling
}
```

this example shows a common case in which the program should not crash. We only need to handle the error, maybe across calling the Get again or something else.

The second handling strategy is for unexpected errors the *panic*-Function which is reserved for *wrong* states and behaviour of the program. *Panic* can be cached to cleanup a Webserver, to write into a logfile, to stop the program more controlled or maybe to recover.

Panics could be thrown very easy.

```
panic("42")
```

and similar simple cached

```
func Parse(input string) (s *Syntax, err error){
    defer func() {                          // nearly same as finally
        if p:= recover(); p != nil {
            err = fmt.Errorf("internal error: %v", p)
        }
    }()
    // .. parser ..
}
```

Go includes by design the *pcall* from Lua. But the *pcall* needs to be built into the functions as an expected behavior, as an additional return value. This is a better approach and has better performance wrapping every unsafe function into a

pcall. The Panic mechanism allows (depending of the developers intention) to recover the program. Do not forget that sometimes it is the right response to panic and maybe break the program.

Goroutines vs Coroutines or Async and Threading

Lua offers coroutine which are similar to the well known thread (in the sense of multithreading): a line of execution, with its own stack, its own local variables, and its own instruction pointer; but sharing global variables and mostly anything else with other coroutines. The main difference between threads and coroutines is that, conceptually (or literally, in a multiprocessor machine), a program with threads runs several threads concurrently. Coroutines, on the other hand, are collaborative: A program with coroutines is, at any given time, running only one of its coroutines and this running coroutine only suspends its execution when it explicitly requests to be suspended.

A coroutine has 3 states: *suspended*, *running*, *dead*. It can be stored in variables and coroutines has functionality to get its state, close, yield and creating them.

```
co = coroutine.create(function ()
       for i=1,10 do
         print("co", i)
         coroutine.yield()
       end
     end)
```

Now, when we resume this coroutine, it starts its execution and runs until the first yield:

```
coroutine.resume(co)    // 1
```

If the status will be checked it returns the *suspended*-state.

```
print(coroutine.status(co))   --> suspended
```

This can be done until the for-loop is ending. Then the state of the coroutine will change to dead. It can no longer be called without an exception.

For example downloading different files using http. It can be downloaded in sequence (take a long time) or if there is currently no data available the coroutines can yield and another coroutine can run and so on.

```
function download (host, file)
     local c = assert(socket.connect(host, 80))// creates connection
     local count = 0                           // counts number of bytes read
     c:send("GET " .. file .. " HTTP/1.0\r\n\r\n")
     while true do
       local s, status = receive(c)
       count = count + string.len(s)
       if status == "closed" then break end
     end
     c:close()
     print(file, count)
   end
```

```
function receive (connection)
  connection:timeout(0)                       // do not block
  local s, status = connection:receive(2^10)
  if status == "timeout" then
    coroutine.yield(connection)
  end
  return s, status
end
```

The next function ensures that each download runs in an individual thread:

```
threads = {}                                  // list of all live threads
function get (host, file)
  local co = coroutine.create(function ()     // create coroutine
    download(host, file)
  end)
  table.insert(threads, co)                   // insert into list
end
```

Coroutines are a kind of collaborative multithreading. They are not constructed as real multithreading like Goroutines. While a coroutine is running, it cannot be stopped from the outside! However, with non-preemptive multithreading, whenever any thread calls a blocking operation, the whole program blocks until the operation completes.

For several applications this is not a problem, even better, it´s much easier. Developers have not to worry about to lock or unlock functions/variables. There can´t be synchronization bugs among threads. The only thing developers have to care about is to *yield* and *resume* coroutines to prevent deadlocks or to let coroutines wait until the end of time.

Go follows totally other concepts of async and threading. Go supports multithreading in form of Goroutines which are very light and multiplexed. This means a Goroutine can be executed on several OS threads. This concept offers the opportunity to use all cores of a machine. Goroutines in general do not use blocking. It is possible but unusual. Goroutines uses communication to pass data from one routine into another. This method is inspired by Hoare´s CSP (1978) and Dijkstra´s guarded commands (1975). In these concepts there is no need to share memory or variables to pass data between Goroutines, they communicate to handle that. For that Go includes so called channels through which can be written or read. Depending on the use case, with or without buffering. The default channel has no buffer size, so if a Goroutine would write into a channel it waits for a receiver on the other side. Similar to that the receiver waits until someone writes into the channel. With these simple rules routines can be synchronized.

```
type Ball struct{ hits int }                    // Ball contains the number of hits.
func main() {
    table := make(chan *Ball)
    go player("ping", table)
    go player("pong", table)
    table <- new(Ball)                          // game on; toss the ball
    time.Sleep(1 * time.Second)
    <-table                                     // game over; grab the ball
}
func player(name string, table chan *Ball) {
    for {
        ball := <-table
        ball.hits++
        fmt.Println(name, ball.hits)
        time.Sleep(100 * time.Millisecond)
        table <- ball
    }
}
```

This example shows a lot. Start with definitions of unusual operators and reserved words:

- *chan* is the type of Channels
- *go*-command tells the machine that this should be started asynchronously as an independent Goroutine. (very easy syntax)
- *some_value* means to write *some_value* into the channel
- a = <- *some_channel* takes the value from a channel

Lets look deeper into the example above.

- In the first line a struct is created which contains the hits (boring)
- The main-function creates a table which is a channel of Ball-Pointer
- Then two player-functions are launched. Both get a name and take a channel of Ball-Pointer. And start in an infinite loop: wait to read from the table, hit the ball, write the hits to the console, wait and write to the table channel.
- Now the game starts: The first message is written into the table channel (do not forget that two player-functions are started and waiting for the first message to handle the ball)
- Wait
- Take a message from the channel (⇒ the two players both wait for a message at the table-channel but none is there) GAME OVER!

It is important to understand is that a Goroutine is not a Thread. It's much lighter and does not share variables or memory. It passes data using channels. That needs new control structures like the *select*-Statement which is nearly the same as the usual *switch*-statements only for sequential Goroutine handling. And of course, if it is needed Go offers blocking:

```
var someThing sync.Mutex
func BlockingExample() {
    something.Lock()
    defer something.Unlock()
...
}
```

Summary

Best to see in the *Goroutine vs Coroutine* chapter, **Lua** and **Go** are made for different usage. **Lua** is made as an embeddable lightweight dynamic script language and does a good job for such approaches. While **Go** has been designed as a cloud programming server language, optimized for the challenges of this discipline. It is lightweight, static and fast to compile multi-threading concurrent language.

Bibliography

Web

- https://www.lua.org

- https://www.youtube.com/watch?v=wdRGOE1N-FA (Talk von LUA in Moskau by Roberto Ierusalimschy)
- https://www.youtube.com/watch?v=f6kdp27TYZs&feature=youtu.be&t=1 (Rob Pike Google I/0 2012 - Go Concurrency Patterns)
- https://pragprog.com/magazines/2013-05/a-functional-introduction-to-lua
- http://vschart.com/compare/lua/vs/go-language

Books

- The Go Programming Language by Donovan, Kernighan
- Programming in Lua by Roberto Ierusalimschy, Lua.org, December 2003

Lecture

- Johannes Weigend at Technical University of Applied Sciences Rosenheim (There i stole a lot of the Go stuff)

by Dominik Ampletzer