1

Exceptions in Lua

John Belmonte

Despite the well known advantages of using exceptions for program errors, the mechanism is underutilized in Lua — both in quantity and quality. One aspect of this relates to the Lua core and standard library, which tend to raise exceptions only in the most serious situations such as parse errors, type errors, and invalid arguments. When exceptions are thrown, they are exclusively string values which are not enumerated as part of the API. Tables, the primary data structure, yield nil for a nonexistent key rather than raise an error. All of this leads to an unspoken bias in Lua that exceptions are something to be thrown but rarely caught — that they are serious errors which normally go unhandled. In the few situations where we do catch them, no distinction is made with respect to the cause of the error.

The core and standard libraries arguably work well as they are, and their use of errors may not warrant meddling. But why are exceptions also underutilized within Lua programs and third party modules? One problem is the unfriendliness of Lua's protected call interface to programmers expecting a native try–catch construct. This in turn discourages library authors from using exceptions for fear of alienating users. The inability to use coroutines within a protected call also works to limit uptake by libraries.

Lua possesses the necessary building blocks for exceptions; however, rough edges appear when one tries to assemble them. This perpetuates disuse of exceptions and strengthens anti-exception patterns such as signaling errors by way of return values. To break the cycle, we first need to promote idioms and know-how for richer use of exceptions. As more Lua developers encounter

the same rough spots, the necessary motivation will exist for some incremental improvements in the core and language itself.

This gem intends to start the process by presenting some exception tools and know-how for Lua. First we spell out a criteria as to when a function should raise an exception versus simply return an error status. For handling the exceptions, we present a simple try–catch idiom that works with today's stock Lua. We then cover why custom error objects are important and address gaps in Lua regarding their use. Finally, we set out to find the right pattern for exception safety in Lua.

What is an error?

What failure situations should be considered a first-class error, warranting the use of exceptions? Calling a function with invalid arguments is an obvious error. In contrast, a negative result from a string matching function is normally not considered an error. In between these is an expanse of various error-like situations. What about an attempt to append to a read-only file; a failed hash table lookup; a database conflict; or an HTTP connection failure? We need a guideline for evaluating these.

On this subject, "Programming in Lua" suggests that if an error cannot be easily avoided, it should be signaled with a return code rather than exception. This logic is geared towards letting you handle error situations without the need for a try–catch — a decidedly conservative view on the use of exceptions. What effect does it have on a program?

Let's consider a Lua program which outputs the length of a file given its name on the command line:

```
local f = io.open(arg[1])
local length = f:seek('end')
print(length)
```

The program lacks error handling — it may be the work of a novice programmer or a lazy expert programmer. How does it behave when things go wrong? Let's try an input file, "abc", which doesn't exist:

```
$ lua file-length.lua abc
lua: file-length.lua:2: attempt to index local 'f' (a nil value)
```

The good news is that an unhandled exception occurred, causing the program to return a non-zero exit code. This is the bare minimum behavior we need from a command-line program on error. The error message, however, is not very helpful. In this simple program we can look at the source code and quickly deduce that `io.open` returned nil instead of a file object, causing an error on call of the seek method. In a complex program, debugging could be much more difficult. The file object could be passed to a different place in the program, and perhaps not used until long after the `io.open` call.

Wrapping the `io.open` call in assert would address this error situation, producing an exception with an accurate location and message.[1] However, the novice programmer didn't consider that, and the expert programmer either didn't think his program would be used so foolishly, or didn't care. In large programs such negligence can go unnoticed until a certain obscure code path is encountered. Arguably, it's better not to present the opportunity for an oversight.

A more liberal guideline for errors is this: if a failure situation is most often handled by the immediate caller of your function, signal it by return value. Otherwise, consider the failure to be a first-class error and throw an exception. The effect is to use exceptions when errors are communicated two levels up the call stack or higher (including possible program termination). This is intended to extract the best value from exceptions. When an error is likely to traverse several levels, we relieve intermediate code from having to propagate the error — a task which is error prone and clutters both code and API. On the other hand, when a failure is usually consumed by the caller, we spare the extravagance and expense of a throw and catch.

What is the outcome when this guideline is applied to `io.open`? It's subjective, but programs usually have a strong dependency on the files they open. When a problem occurs — whether it be a full storage device, permission error, or missing file — it tends to require handling at a high level in the program, if it is handled at all. It's a good guess that the error will be traveling up past the immediate caller of the I/O function.

A simple try–catch construct

Now that we've planted the seed for more exceptions, we can focus on how to catch them. As mentioned, Lua lacks the common try–catch construct for dealing with exceptions, which may put off some programmers. By creating something in pure Lua close to that familiar construct, perhaps we can lower the barrier to more extensive use of exceptions.

Lua supports catching of exceptions through a functional interface, namely `pcall`. It expects that the code to be attempted is itself defined as a function. Those constraints leave us with few options — our try–catch will have to be functional also, with the "try" and "catch" blocks of code passed in as functions. Nonetheless, with the help of in-line anonymous functions and some creative formatting, we can approach the feel of a native try–catch construct. Here is a template for use of our utility function, simply called "try":

[1] Wrapping a call with assert assumes it follows the convention of returning a nil and error message tuple on failure. The convention can't be used, however, if nil or false happen to be valid outputs. It can also interfere with code readability when a function has multiple outputs and the caller elects not to wrap with assert (e.g., a function returns coordinates x and y, but on error y doubles as a message).

```
try(function()
    -- Try block
end, function(e)
    -- Catch block.  E.g.:
    --   Use e for conditional catch
    --   Re-raise with error(e)
end)
```

The catch function, should it be invoked, receives the error object as an argument. After inspecting the error, it can elect to either suppress the exception by taking no action, re-raise the existing error, or throw a different error.

A notable limitation of using functions to define our code blocks is that flow control statements, such as return and break, cannot cross outside the try–catch. For example, the following code would not work as expected:

```
function foo()
    try(function()
        if some_task() then
            return 10 -- does not cause foo() to return 10
        end
    end, function(e)
        -- ...
    end)
    return 20
end
```

Lua's `pcall` operates by calling the function given to it. Any exception will be trapped, returning nil and the error object. Based on that, the definition of our try function is trivial:

```
function try(f, catch_f)
    local status, exception = pcall(f)
    if not status then
        catch_f(exception)
    end
end
```

Unfortunately coroutines do not mix well with `pcall`, so this will preclude their use within our try block. The problem is well known and has various workarounds, ranging from a `pcall` replacement implemented in pure Lua to an extensive Lua core patch.

Custom error objects

Putting our new try–catch construct to use, let's say we have a transactional database application. If a database conflict error occurs — perhaps because two programs tried to increment the same balance field of some record — we'd like to retry the transaction. Coding our simplistic example:

```
try(function()
    do_transaction()
end, function(e)
    log('Retrying database transaction')
    do_transaction()
end)
```

The issue here is that we end up retrying the transaction not only when there is a database problem, but also for any other error. This could mask bugs such as calling a function with the wrong arguments, producing strange program behavior. Clearly we want to be more selective by handling only the errors we understand and letting the rest pass through. Given the common practice of throwing strings, however, this becomes tricky. We are faced with fragile parsing of exception messages which may change in the future, especially if they originate from a third party's module.

To address this problem, we take advantage of the often-overlooked ability of `error` to throw values other than strings. A table is the natural choice, leaving room for expanded functionality by way of methods and internal state. The database module might simply contain the following definition:

```
ConflictError = {}
```

This approach serves not only to allow positive identification of an exception, but also to enumerate the errors which can be raised by a module — it should be considered part of the API. Now the database module can signal a conflict with `error(ConflictError)` and our catch function can be refined as follows:

```
function(e)
    if e == db.ConflictError then
        log('Retrying database transaction')
        do_transaction()
    else
        error(e)  -- re-raise
    end
end
```

A new problem is lurking however. What if the database conflict should go unhandled? Let's simulate the situation in Lua's interactive interpreter:

```
> error({})
(error object is not a string)
```

Unfortunately, the uncaught exception handler which lives inside Lua's standard interpreter refuses to do anything with a non-string error value. We're missing the human-readable message and call stack which are essential for locating the source of the error. The required improvement to the interpreter is minor however: just pass the error value through `tostring` before invoking `debug.traceback`. This change is planned for the next version of Lua. With this change in the interpreter, and by enhancing our error object with an appropriate `__tostring` metamethod, the behavior becomes:

```
> MyError = setmetatable({},
>>   {__tostring = function() return 'My error occurred' end})
> error(MyError)
My error occurred
stack traceback:
        [C]: in function 'error'
        stdin:1: in main chunk
        [C]: ?
```

While this is a significant improvement, there is sill one detail missing from the trace: the file name and line number of the exception. Normally, with a string value, the error function adds this information at the point of the exception by prefixing it to the string. For other value types the location is omitted. While the association between an error and its location might best be maintained by the Lua core, such a change would be substantial. A compromise is to alter the error function to store the location in a field (assuming the value is a table) and have it picked up by the interpreter's handler. This location fix and the aforementioned `tostring` fix are available together as a "custom errors" patch to Lua. (See `http://lua-users.org/wiki/LuaPowerPatches`.)

Continuing with our database application, suppose we wish to catch any exception specific to the database module. Or perhaps the module author decides to distinguish between read and write conflicts using separate error types, while our handler remains interested in both cases. It would be unfortunate to have to spell out each error to be caught when all we mean is "any database module error" and "any conflict error", respectively. This suggests the need for an error hierarchy, where we can test if a certain instance belongs to a given class of errors.

In other languages, an error hierarchy tends to be defined by class inheritance. In Lua we are free to do the same, but without a standard class system the error values from various modules and our own code will lack a common root and API. As a compromise, the database module author might make a utility available for testing inheritance among the module's own objects. The implementation should be robust, yielding a negative result for foreign values. Our catch function then becomes:

```
function(e)
    if db.instance_of(e, db.ConflictError) then
        log('Retrying database transaction')
        do_transaction()
    else
        error(e)
    end
end
```

Note that such an inheritance test becomes mandatory should we choose to make error objects something more than a simple table constant. For an error

having internal state, a new instance must be created for each exception thrown. In that case equality cannot be used to identify the exception.

The argument for custom errors is that a human-readable error message, while essential, should be only one component of a richer error object. Errors should be enumerated as part of an API, providing the ability to positively identify exceptions and perhaps locate their place within a hierarchy of errors. Custom error objects can also serve to store arbitrary state at the time of an exception, which may be useful for debugging and error reporting. All of this is light work for Lua tables, although the need for hierarchy testing does present an interoperability issue between modules.

Exception safety

With exceptions comes the issue of exception safety — proper cleanup of acquired resources and program state when an exception does occur. Acquired resources might include memory allocated from special pools, device handles, and mutex objects. Consider the following simplistic function to paint a logo onto the screen:

```
function display_logo(display_buffer, x, y)
    local canvas = allocate_canvas(50, 50)
    render_logo(canvas)
    display_buffer:lock()
    display_buffer:copy(canvas, x, y)
    display_buffer:unlock()
    canvas:free()
end
```

During the course of this function we acquire a graphic canvas (perhaps off-screen video memory) and a lock on the display buffer. If the `render_logo` function happens to throw an exception then the canvas may not be freed in a timely manner — it may happen automatically when the canvas value is garbage collected, but we don't know when that will be. More seriously, if the `display_buffer:copy` call throws an exception because the input coordinates are out of range, the display is never unlocked. Clearly, if resources like this are going to be exposed to the scripting environment, we need a way to free them even if an exception occurs.

Even if we decide not to expose management of critical resources to scripting, there are common cases where we must ensure that some program state is restored despite an error. Say we'd like the text output of a certain third party function directed to a file, but the module has been hard-coded to use standard output. We could work around the limitation by changing Lua's default output temporarily:

```
local out = io.output()
io.output(log_file)
somelib.do_task()
io.output(out)
```

The problem here is that if `do_task` throws an exception, the default output will never be restored. One may argue that restoring this state doesn't matter because the process will be terminated anyway. This overlooks the possibility that the exception may be handled at a higher level in the execution stack, allowing the program to continue. That a certain error is too dire to be intercepted usually turns out to be a myopic view since, at the highest level of the program, there are always options such as reattempting an operation or switching to a failover routine. This makes proper exception safety especially important when implementing a library, where the author cannot imagine all usage scenarios.

Now that we've identified the need for exception safety, how is it accomplished? The solutions are all variations on one theme: install cleanup code to be run on exit of the current scope, whether that be normally or by exception. Traditionally, programming languages have two mechanisms available for this. One is the try-finally construct, where the scope is defined by a "try" block, and the cleanup code placed in a "finally" block which always runs afterward. As a language construct, however, try-finally has fallen out of fashion. To consider why, let's pretend that Lua supported `try..finally..end` and apply it to our `display_logo` function:

```
function display_logo(display_buffer, x, y)
    local canvas = allocate_canvas(50, 50)
    try  -- no such thing in Lua
        render_logo(canvas)
        display_buffer:lock()
        try
            display_buffer:copy(canvas, x, y)
        finally
            display_buffer:unlock()
        end
    finally
        canvas:free()
    end
end
```

The issue is one of code readability and maintenance. A nested try-finally is needed for each consecutive resource acquired, making the flow of the original program difficult to follow. Moreover, although having "try" come before "finally" is most intuitive and the common layout, it tends to maximize the distance between acquisition and cleanup code. This problem becomes more pronounced as the size of the function grows — to the point where the programmer cannot see them on the screen together, and could modify one without considering the other.

The other traditional mechanism for exception safety is the use of a custom object which is referenced solely by the local scope. The cleanup code exists in the destructor of the object so that it will be invoked as the value goes out of scope. Rather than defining an ad hoc type for each cleanup situation — which becomes verbose and a burden to maintain — the use of a generic "scope

manager" object is becoming common (e.g., the C++ scope guard pattern, or the D "scope" statement). A scope manager allows the registration of arbitrary code which will be called at scope exit. Since registration can take place multiple times and throughout the scope, it enables natural placement of cleanup code. In some languages it's possible for the manager to know if the scope exited normally or by exception, further enhancing the utility of this pattern.

Unfortunately, Lua provides no way to hook into scope exit.[2] As object destruction is subject to the whim of the garbage collection system, the trick of using an object referenced only by the local scope does not provide deterministic cleanup. As in our try–catch implementation, however, it's possible to approximate such a hook by way of an explicit function scope and `pcall`. We'll use that to create a simple scope manager in Lua for our cleanup needs.

A simple scope manager

We define a utility function "scope", which takes a single function argument and calls it. Within the environment of the given function, an `on_exit` function is made available for registering cleanup functions. Here is how the scope utility looks when applied to our `display_logo` example:

```
function display_logo(display_buffer, x, y)
    scope(function()
        local canvas = allocate_canvas(50, 50)
        on_exit(function() canvas:free() end)
        render_logo(canvas)
        display_buffer:lock()
        on_exit(function() display_buffer:unlock() end)
        display_buffer:copy(canvas, x, y)
    end)
end
```

Notice that no nesting is needed for consecutively acquired resources as in the try-finally solution. Also, each piece of cleanup code is positioned logically so that, as the code is read from top to bottom, one can see exactly when it becomes active within the scope.

To round out our cleanup utility, we'll make two more registration functions available within the scope: `on_failure` and `on_success`. The `on_failure` hook might be used to roll back a pending database transaction or other tentative state change. Although a try–catch could be used here instead, `on_failure` is more readable and avoids having the user take responsibility for re-raising the caught error. The `on_success` hook will likely be least used of the three, but

[2] The ability to hook into scope exit is the only fundamental building block I've noticed as missing from Lua 5.1. I hope that this can be resolved in a future version of the language — perhaps by creating a new class of variable which notifies its value when it goes out of scope, or by adding a construct along the lines of Python's "with" statement.

again it offers more flexibility on placement of cleanup code. Here is the scope function implementation:

```
function scope(f)
    local function run(list)
        for _, f in ipairs(list) do f() end
    end
    local function append(list, item)
        list[#list+1] = item
    end
    local success_funcs, failure_funcs, exit_funcs = {}, {}, {}
    local manager = {
        on_success = function(f) append(success_funcs, f) end,
        on_failure = function(f) append(failure_funcs, f) end,
        on_exit = function(f) append(exit_funcs, f) end,
    }
    local old_fenv = getfenv(f)
    setmetatable(manager, {__index = old_fenv})
    setfenv(f, manager)
    local status, err = pcall(f)
    setfenv(f, old_fenv)
    -- NOTE: behavior undefined if a hook function raises an error
    run(status and success_funcs or failure_funcs)
    run(exit_funcs)
    if not status then error(err, 2) end
end
```

Like the try–catch implementation, this scope hook suffers from an incompatibility with coroutine yield, and the inability to use flow control statements across the scope's boundary (i.e., `return`, `break`, etc.). A more fundamental limitation exists however: cleanup code itself must not raise an exception. Allowing this would create at least two ambiguities: 1) if an exception happens in one piece of cleanup code, should the entire cleanup contract be invalidated? 2) if there are multiple, logically parallel exceptions, which is to be propagated? The situation is best avoided and, in the implementation presented, its behavior is left undefined.

A slightly different design for the scope utility would be to pass the manager object to the user's function as an argument. Besides eliminating the complexity of making `on_exit` and the other registration methods appear implicitly within the function, this would allow the manager to be passed to utility functions. For example, the `allocate_canvas` function could take a scope manager as an optional argument, and in that case register the canvas cleanup code for us. On the other hand, the explicit manager variable makes the user's code more verbose in the simple case, and opens the door for confusion should someone try to operate on the manager of an already expired scope.

This pattern to assist with exception safety is the final component in our bag of exception tools. Combined with custom error objects, which allow discern-

ing between errors, and a try–catch construct implemented in pure Lua, programmers can explore richer use of exceptions in their programs and libraries. Limitations and rough spots exist for sure, but hopefully this situation is temporary — the authors of Lua have a good track record of improving the flexibility of the language and its implementation over time.