CodeFlow

Live Application Development Environment for iOS, tvOS & macOS
CodeFlow

Live Application Development Environment for iOS, tvOS & macOS

- instant feedback on real devices
- native OS SDK
- Lua language
- native project APIs
- live code
- live assets
- true debugger
- live storyboards
A Bridge? What for?
Transparent development of iOS code in Lua
Goals of the iOS bridge

- Enable the development of iOS apps in Lua using the native OS SDK
- Make the use of the native SDK feel natural in Lua
- Make it easy for a Swift or ObjC developer to move to Lua

⇒ **Transparent integration between Lua and iOS**

- Not the same objective as some other bridges
  - Exposing Lua-specific features to the iOS native world was not in the scope, nor was the definition of a Swift / ObjC version of the Lua C API.
  - Low-level aspects of the native world had to be hidden from the Lua code
The foundations

Dealing with type conversions, memory management, and threads
Mixing Lua and native types

- Different typing systems
  - Lua: typed values; untyped function parameters
  - C world: typed variables and parameters; ABI
- Calling native from Lua: convert parameters to the expected types
  - Easy for base types, more complex for structured types, objects, collections...
  - Doing this conversion is the first role of a bridge
- Example: expose a struct to Lua
  - Pseudo-object with constructor, accessors, ... and methods
  - Automatic Lua table $\rightarrow$ struct conversion in function calls

```lua
local CGPoint = struct.CGPoint:_structInterface { x = 0.0, y = 0.0 }

struct CGPoint {
  CGFloat x;
  CGFloat y;
};

local aPoint = struct.CGPoint (100, 50)
aPoint.x = 200
self.view.center = aPoint
--- ...
self.view.center = { x = 150, y = aPoint.y + 20 }
```
Making Memory Models Coexist

- Different memory models
  - Lua: garbage collector
  - ObjC runtime: automatic reference counting
Making Memory Models Coexist

- Different memory models
  - Lua: garbage collector
  - ObjC runtime: automatic reference counting

Diagram:
- Object A
  - Create
  - Retain
  - Release
- Object B
  - Create: +1
  - Retain: +2
  - Release: +1
- Object C
  - Create
  - Retain
  - Release

Note: Object B is marked as disposed of.
Making Memory Models Coexist

- Different memory models
  - Lua: garbage collector
  - ObjC runtime: automatic reference counting
- Managing objects lifecycle
  - A native object passed to the Lua runtime is retained until GC-ed, and released by its finalizer metamethod
  - A Lua value passed to the native world maintains a Lua reference to prevent GC (luaL_ref) and remove this reference when not used anymore. (luaL_unref)
Making Memory Models Coexist

- Different memory models
  - Lua: garbage collector
  - ObjC runtime: automatic reference counting
- Managing objects lifecycle
  - A native object passed to the Lua runtime is retained until GC-ed, and released by its finalizer metamethod
  - A Lua value passed to the native world maintains a Lua reference to prevent GC (luaL_ref) and remove this reference when not used anymore. (luaL_unref)
- The retain cycle problem
  - It is possible from Lua, to create a retain cycle between native objects
Making Memory Models Coexist

- Different memory models
  - Lua: garbage collector
  - ObjC runtime: automatic reference counting
- Managing objects lifecycle
  - A native object passed to the Lua runtime is retained until GC-ed, and released by its finalizer metamethod
  - A Lua value passed to the native world maintains a Lua reference to prevent (luaL_ref) and remove this reference when not used anymore. (luaL_unref)
- The retain cycle problem
  - It is possible from Lua, to create a retain cycle between native objects

![Diagram showing two objects, Object A and Object B, each with a reference count of +1, forming a retain cycle.]
Making Memory Models Coexist

- Different memory models
  - Lua: garbage collector
  - ObjC runtime: automatic reference counting
- Managing objects lifecycle
  - A native object passed to the Lua runtime is retained until GC-ed, and released by its finalizer metamethod
  - A Lua value passed to the native world maintains a Lua reference to prevent GC (luaL_ref) and remove this reference when not used anymore. (luaL_unref)
- The retain cycle problem
  - It is possible from Lua, to create a retain cycle between native objects
    \[ \Rightarrow \text{memory leak!} \]
  - Weak object references are the solution
    - Object reference getters: weakRef and strongRef
      ```lua
      local weakSelf = self.weakRef
      ```
    - A weak reference become an *all-nil* object when the referenced object is deallocated
Running Lua in a Threaded World

- Lua runs as a single thread, while the host OS is heavily multi-threaded.
- In an iOS app, code execution is triggered by user or external events.
  - We cannot control in which thread our Lua methods are called!
- The iOS bridge has to make Lua work in a multi-threaded environment.
Running Lua in a Threaded World

- Lua runs as a single thread, while the host OS is heavily multi-threaded
- In an iOS app, code execution is triggered by user or external events
  - We can not control in which thread our Lua methods are called!
- The iOS bridge has to make Lua work in a multi-threaded environment

- Our solution:
  - Every top-level Lua code invocation runs in its own Lua thread (i.e. lua_State)
  - A simple scheduler allows to execute only one Lua thread at a given time, with well-defined deschedule points
- Looks simple but works great in practice!
Design Patterns Translation

Making native design patterns feel natural in Lua
About Native Design Patterns

• An API is not just about types and function: how to use it is even more important.
• Typical design patterns define the expected way to use the APIs.
• The iOS / macOS SDKs rely on strong design patterns and conventions: MVC, delegation, observing, target-action…
• Making these design patterns feel natural in Lua is key for the bridge usability!

Now, a few examples of design patterns adaptation to Lua:
Pattern 1: Subclass to Customize

This is how Controllers work in iOS.

⇒ We need the possibility to subclass native classes in Lua!

```lua
local ViewController = class.createClass("ViewController", objc.UIViewController)

function ViewController:loadView ()
    -- Create a view programmatically.
    self.view = objc.UIView:new()
end

function ViewController:viewDidLoad ()
    self[ViewController.superclass]:viewDidLoad()
    self:configureView()
    self:addMessageHandler (ViewController, "refreshView")
end

function ViewController:configureView ()
    -- Put here the code configuring the controller's view
    self.view.backgroundColor = objc.UIColor.whiteColor
end

function ViewController:refreshView ()
    -- Update the controller's view
    self:configureView()
    -- Other refresh actions
    -- ...
end

return ViewController
```

This creates a Lua subclass of native UIViewController

Two native methods overridden in Lua

Two Lua methods not visible from the native code
Pattern 2: delegation

• A *delegate* object is used to customize or control the actions of a SDK object, by implementing a well-defined API contract declared as a *protocol*. A delegate object can be of any class, provided it implements the expected protocol.

• A Lua object can be declared as the delegate of a native object.

• *Publishing* a protocol makes the protocol’s methods defined by a Lua class callable from the native code.

```
local TableDataSource = class.createClass("TableDataSource")

function TableDataSource:setTableView (tableView)
  self.tableView = tableView
  tableView.datasource = self
end

TableDataSource:publishObjcProtocols "UITableViewDataSource"

function TableDataSource:tableView_numberOfRowsInSection (tableView, section)
  local objects = self.objects
  return objects and #objects or 0
end

function TableDataSource:tableView_cellForRowAtIndexPath (tableView, indexPath)
  local cell = tableView:dequeueReusableCellWithIdentifier_forIndexPath("Cell", indexPath)
  local object = self.objects [indexPath.row + 1]
  cell.textLabel.text = object.description
  return cell
end
```
Pattern 3: closure parameters

- Closure (aka ObjC blocks) parameters are used for synchronous or asynchronous callback in many places of the iOS / macOS SDKs
- Lua functions are a perfect match for closure parameters!

```lua
function CollectionController:setCollectionText(text)
    local words = {}
    local wordsCount = 0
    text:enumerateSubstringsInRange_options_usingBlock (
        NSRange(0, text.length),
        NsString.Enumeration.ByWords,
        function(word, range, effectiveRange)
            wordsCount = wordsCount + 1
            words[wordsCount] = word
        end)

    self.textWords = words
    self.collectionView:reloadData()
end
```

This native NSString method takes a closure parameter.

You simply pass a Lua function for this closure parameter.
Bindings Generation

Supporting large OS SDKs thanks to automation
Two main components in the bridge

- **Generic bridge library**: memory & threads management, OO framework, generic type conversion and function call bridging
- **Bindings**: the specific code that makes the bridge work for a given SDK or API

- iOS / macOS SDKs are quite big (~1900 header files for iOS, 2300 for macOS)
  - Bindings generation has to be automated
- Use clang (llvm) for parsing C / Objective-C headers
- Bindings generation is based on the AST generated by clang
SDK Bindings Generation

- **Bindings Libraries**
  - Mix of generated code and declarative typing information
  - Linked with the target application
  - Include: constants, enums, structs, C functions, classes with methods and properties, protocols ...
  - Loaded as Lua modules
    ```lua
    local UIGestureRecognizer = require "UIKit.UIGestureRecognizer"
    ```

- **Bindings Metadata**
  - Used by the IDE

- **Bindings Lua Interface**
  - A user-readable Lua version of the SDK
IDE Integration

Supporting native SDKs in the IDE for a better coding experience
Bridge - IDE Integration

• Goal: help the developer to use the native SDK(s) in Lua
• In the Lua source code editor
  • auto-completion of SDK symbols defined in Bindings Libraries
Bridge - IDE Integration

- Goal: help the developer to use the native SDK(s) in Lua
- In the Lua source code editor
  - auto-completion of SDK symbols defined in Bindings Libraries
- For build configuration of target app
  - by computing bindings-related dependencies in Lua modules
- In the Lua debugger
  - inspect native types in the Variables Inspector
  - interrupt on error in case of failed type conversion or wrong nullability … and continue execution after fixing the issue!
Bridge - IDE Integration

• Goal: help the developer to use the native SDK(s) in Lua

• In the Lua source code editor
  • auto-completion of SDK symbols defined in Bindings Libraries

• For build configuration of target app
  • by computing bindings-related dependencies in Lua modules

• In the Lua debugger
  • inspect native types in the Variables Inspector
  • interrupt on error in case of failed type conversion or wrong nullability … and continue execution after fixing the issue!
Tour completed

What have we seen?
Recap

Needed for this bridge:

• A well-defined goal for the iOS bridge.
• Solid low-level foundations: types, memory and threads.
• Careful transposition of the SDK’s main design patterns.
• Bindings generation tools to support large SDKs.
• IDE integration to brings additional value to the user.
For More Information

- About CodeFlow and live-coding on iOS
  - Explore [https://www.celedev.com](https://www.celedev.com)
  - Follow the project: @celedev

- About the iOS bridge
  - Part 2: CodeFlow object framework
  - Part 3: CodeFlow native bridge
Thank You!

Questions?

Jean-Luc Jumpertz  @JLJump

www.celedev.com  @celedev