Lesson 04: Function 函式

歡迎來到最後一課！我們要要學習程式設計中最強大的概念之一：Function（函式）。如果把程式比作一本食譜書，那麼函式就像是書中的各個料理食譜——每個食譜都有明確的材料（輸入）、步驟（處理）和成品（輸出）。

什麼是函式？

函式的概念

想像你在廚房裡：

製作果汁的步驟：

1. 準備水果（蘋果、橘子等）
2. 清洗水果
3. 榨汁
4. 加糖調味
5. 倒入杯子

每次想喝果汁時，你不需要重新想這些步驟，因為你已經有了一個「製作果汁」的食譜。函式就是這樣的概念，可以把一系列步驟打包起來，給它一個名字，之後就可以重複使用。

```
-- 不使用函式，每次都要重寫
print("蘋果汁製作中...")
print("清洗 → 榨汁 → 調味 → 完成")

print("橘子汁製作中...")
print("清洗 → 榨汁 → 調味 → 完成")

-- 使用函式，只需要定義一次
function make_juice(fruit)
    print(fruit .. "汁製作中...")
    print("清洗 → 榨汁 → 調味 → 完成")
end

-- 重複使用
make_juice(\"蘋果\")
make_juice(\"橘子\")
```

函式的好處

1. **避免重複**：寫一次，用很多次
2. **易於維護**：修改一個地方，全部地方都更新
3. **可讀性高**：程式邏輯更清晰
4. **模組化**：把複雜問題分解成小問題

定義和呼叫函式

基本語法

```
function 函式名稱(參數1, 參數2, ...)
    -- 函式內容
    return 回傳值  -- 可選
end
```

簡單範例

```
-- 定義一個打招呼的函式
function say_hello()
    print("哈囉！歡迎來到 Lua 的世界！")
end

-- 呼叫函式
say_hello()  -- 輸出：哈囉！歡迎來到 Lua 的世界！
```

帶參數的函式

```
-- 個人化的問候
function greet(name)
    print("哈囉，" .. name .. "！很高興見到你！")
end

-- 呼叫函式並傳入參數
greet("小明")    -- 哈囉，小明！很高興見到你！
greet("小華")    -- 哈囉，小華！很高興見到你！

-- 多個參數
function introduce(name, age, city)
    print("我是 " .. name .. "，今年 " .. age .. " 歲，住在 " .. city)
end
```

```lua
introduce("小明", 20, "台北")
-- 輸出：我是 小明，今年 20 歲，住在 台北
```

回傳值

函式可以回傳計算結果：

```lua
-- 計算圓面積
function calculate_circle_area(radius)
    local area = 3.14159 * radius * radius
    return area
end

-- 使用回傳值
local my_area = calculate_circle_area(5)
print("半徑 5 的圓面積是：" .. my_area)

-- 直接在其他計算中使用
local total_area = calculate_circle_area(3) + calculate_circle_area(4)
print("兩個圓的總面積：" .. total_area)
```

多個回傳值

Lua 的函式可以同時回傳多個值：

```lua
-- 計算矩形的面積和周長
function rectangle_info(length, width)
    local area = length * width
    local perimeter = 2 * (length + width)
    return area, perimeter
end

-- 接收多個回傳值
local area, perimeter = rectangle_info(5, 3)
print("面積：" .. area .. ", 周長：" .. perimeter)

-- 也可以只接收部分回傳值
local area = rectangle_info(5, 3)  -- 只接收第一個回傳值
print("面積：" .. area)
```

參數處理

預設參數

Lua 沒有內建的預設參數，但可以這樣實作：

```lua
function greet_with_default(name, greeting)
    greeting = greeting or "哈囉"  -- 如果沒有提供 greeting，使用 "哈囉"
    print(greeting .. "，" .. name .. "！")
end

greet_with_default("小明")              -- 哈囉，小明！
greet_with_default("小華", "早安")      -- 早安，小華！
```

變數長度參數

使用 `...` 可以接受任意數量的參數：

```lua
-- 計算多個數字的總和
function sum(...)
    local numbers = {...}  -- 把所有參數打包成 Table
    local total = 0

    for _, number in ipairs(numbers) do
        total = total + number
    end

    return total
end

print(sum(1, 2, 3))           -- 6
print(sum(1, 2, 3, 4, 5))     -- 15
print(sum(10))                -- 10

-- 找出最大值
function find_max(...)
    local numbers = {...}

    if #numbers == 0 then
        return nil
    end

    local max = numbers[1]
    for i = 2, #numbers do
        if numbers[i] > max then
            max = numbers[i]
```

```
        end
    end

    return max
end

print(find_max(3, 1, 4, 1, 5, 9, 2, 6))  -- 9
```

變數作用域

local 和 global 變數

```
-- 全域變數：在任何地方都可以存取
global_counter = 0

function increment_global()
    global_counter = global_counter + 1
end

-- 區域變數：只在特定範圍內有效
function demo_local()
    local local_var = "我是區域變數"
    print(local_var)  -- 可以使用
end

demo_local()
-- print(local_var)  -- 錯誤！在這裡無法存取
```

函式內的變數作用域

```
-- 好的範例：使用 local 變數
function calculate_bmi(weight, height)
    local bmi = weight / (height * height)  -- local 變數
    local status                            -- 宣告 local 變數

    if bmi < 18.5 then
        status = "體重過輕"
    elseif bmi < 24 then
        status = "體重正常"
    else
        status = "體重過重"
    end
```

```
    return bmi, status
end

-- 函式外無法存取 bmi 和 status
local my_bmi, my_status = calculate_bmi(70, 1.75)
print("BMI: " .. string.format("%.1f", my_bmi) .. ", 狀態: " ..
my_status)
```

為什麼要使用 local？

```
-- 壞的範例：使用全域變數
function bad_calculate(a, b)
    temp = a * 2       -- 全域變數，可能會影響其他程式
    result = temp + b  -- 全域變數
    return result
end

-- 好的範例：使用區域變數
function good_calculate(a, b)
    local temp = a * 2     -- 只在函式內有效
    local result = temp + b -- 不會影響其他程式
    return result
end

-- 為什麼 local 比較好？
-- 1. 避免命名衝突
-- 2. 記憶體使用更有效率
-- 3. 程式邏輯更清晰
-- 4. 除錯更容易
```

匿名函式

匿名函式

```
-- 一般的函式定義
function square(x)
    return x * x
end

-- 匿名函式：沒有名稱的函式
local square_anonymous = function(x)
    return x * x
end
```

```lua
-- 在 Table 中使用匿名函式
local operations = {
    add = function(a, b) return a + b end,
    subtract = function(a, b) return a - b end,
    multiply = function(a, b) return a * b end,
    divide = function(a, b) return a / b end
}

print(operations.add(5, 3))       -- 8
print(operations.multiply(4, 7))  -- 28
```

函式作為參數

```lua
-- 接受函式作為參數的函式
function apply_operation(numbers, operation)
    local result = {}
    for i, num in ipairs(numbers) do
        result[i] = operation(num)
    end
    return result
end

-- 使用範例
local numbers = {1, 2, 3, 4, 5}

-- 每個數字平方
local squares = apply_operation(numbers, function(x) return x * x end)
print("平方：", table.concat(squares, ", "))  -- 1, 4, 9, 16, 25

-- 每個數字加倍
local doubles = apply_operation(numbers, function(x) return x * 2 end)
print("加倍：", table.concat(doubles, ", "))  -- 2, 4, 6, 8, 10
```

遞迴函式

遞迴就是函式呼叫自己：

```lua
-- 計算費波那契數列（遞迴版本）
function fibonacci(n)
    if n <= 0 then
        return 0
    elseif n == 1 then
        return 1
    else
        return fibonacci(n - 1) + fibonacci(n - 2)
    end
end

print(fibonacci(7))  -- 13
```

常見錯誤和注意事項

1. 函式名稱的作用域

```lua
-- 錯誤示範：函式還沒定義就使用
print(my_function())  -- 錯誤！my_function 還不存在

function my_function()
    return "Hello"
end

-- 正確做法：先定義再使用
function my_function()
    return "Hello"
end

print(my_function())  -- 正確！
```

2. 參數數量不匹配

```lua
function greet(name, age)
    print("哈囉 " .. name .. "，你 " .. age .. " 歲")
end

greet("小明")         -- 錯誤！age 是 nil
greet("小明", 25, "額外參數")  -- 正常運作，額外參數被忽略

-- 安全的做法
function safe_greet(name, age)
    name = name or "訪客"
    age = age or "未知"
    print("哈囉 " .. name .. "，你 " .. tostring(age) .. " 歲")
end
```

3. 全域變數污染

```lua
-- 錯誤示範：忘記使用 local
function calculate_something()
    result = 42  -- 意外建立了全域變數
    return result
end

-- 正確做法：明確使用 local
function calculate_something()
    local result = 42  -- 區域變數，不會污染全域
    return result
end
```

grandMA2 Plugin 開發實戰應用

函式是 grandMA2 plugin 的靈魂！讓我們學習如何建立專業的燈光控制 plugin：

Plugin 必要函式：Start 和 Cleanup

```lua
-- grandMA2 Plugin 基礎架構

-- 全域配置
local config = {
    default_fade_time = 2,
    max_brightness = 100,
    min_brightness = 0,
    effect_speed = 0.5
}
```

```lua
-- Plugin 進入點：當 plugin 被載入時執行
function Start()
    gma.echo("=== Professional Lighting Controller 啟動 ===")

    -- 初始化系統
    InitializeSystem()

    -- 顯示主選單
    ShowMainMenu()

    gma.echo("Plugin 執行完成")
end

-- 清理函式：當 plugin 終止時執行
function Cleanup()
    gma.echo("=== 清理系統資源 ===")

    -- 關閉所有進度條
    CloseAllProgressBars()

    -- 恢復預設狀態
    RestoreDefaultState()

    gma.echo("Professional Lighting Controller 已安全結束")
end

-- 系統初始化函式
function InitializeSystem()
    gma.echo("初始化燈光控制系統...")

    -- 檢查系統狀態
    local system_ready = CheckSystemHealth()
    if not system_ready then
        gma.echo("警告：系統狀態異常")
        return false
    end

    -- 載入預設設定
    LoadDefaultSettings()

    gma.echo("系統初始化完成")
    return true
end
```

```
-- 系統健康檢查
function CheckSystemHealth()
    gma.echo("檢查系統健康狀態...")

    -- 檢查重要燈具是否存在
    local critical_fixtures = {1, 2, 3, 4, 5}
    for _, fixture_id in ipairs(critical_fixtures) do
        local handle = gma.show.getobj.handle("Fixture " .. fixture_id)
        if not handle then
            gma.echo("錯誤：關鍵燈具 " .. fixture_id .. " 不存在")
            return false
        end
    end

    gma.echo("系統健康狀態：正常")
    return true
end

-- 必須返回這兩個函式
return Start, Cleanup
```

打造燈光效果工具箱

```
-- grandMA2 Plugin 實戰：專業燈光效果庫
-- Plugin 名稱：Lighting Effects Toolkit

local LightingEffects = {}

-- 呼吸燈效果
function LightingEffects.Breathing(fixtures, cycles, min_brightness,
max_brightness, speed)
    gma.echo("執行呼吸燈效果 - " .. cycles .. " 個循環")

    local progress = gma.gui.progress.start("呼吸燈效果")
    gma.gui.progress.setrange(progress, 1, cycles * 2)

    local step = 0
    for cycle = 1, cycles do
        -- 漸亮階段
        for brightness = min_brightness, max_brightness, 5 do
            gma.cmd("Fixture " .. fixtures .. " at " .. brightness .. "
fade " .. speed)
            gma.sleep(speed + 0.1)
            step = step + 1
```

```
            gma.gui.progress.set(progress, math.floor(step / 2))
        end

        -- 漸暗階段
        for brightness = max_brightness, min_brightness, -5 do
            gma.cmd("Fixture " .. fixtures .. " at " .. brightness .. "
fade " .. speed)
            gma.sleep(speed + 0.1)
            step = step + 1
            gma.gui.progress.set(progress, math.floor(step / 2))
        end
    end

    gma.gui.progress.stop(progress)
    gma.echo("呼吸燈效果完成")
end

-- 彩虹效果
function LightingEffects.Rainbow(fixtures, duration, smooth)
    local colors = {"red", "orange", "yellow", "green", "cyan", "blue",
"magenta"}
    local fade_time = smooth and (duration / #colors) or 0

    gma.echo("執行彩虹效果 - 持續時間：" .. duration .. "秒")

    for _, color in ipairs(colors) do
        gma.cmd("Fixture " .. fixtures .. " at 80 " .. color .. " fade "
.. fade_time)
        gma.sleep(duration / #colors)
    end

    gma.echo("彩虹效果完成")
end

-- 閃爍效果
function LightingEffects.Strobe(fixtures, times, on_time, off_time,
intensity)
    gma.echo("執行閃爍效果 - " .. times .. " 次")

    local progress = gma.gui.progress.start("閃爍效果")
    gma.gui.progress.setrange(progress, 1, times)

    for i = 1, times do
        -- 亮起
        gma.cmd("Fixture " .. fixtures .. " at " .. intensity)
```

```
            gma.sleep(on_time)

            -- 關閉
            gma.cmd("Fixture " .. fixtures .. " at 0")
            gma.sleep(off_time)

            gma.gui.progress.set(progress, i)
        end

        gma.gui.progress.stop(progress)
        gma.echo("閃爍效果完成")
    end

    function Start()
        gma.echo("=== 燈光效果工具箱啟動 ===")

        -- 示範各種效果
        LightingEffects.Breathing("1 thru 5", 2, 10, 100, 0.5)
        gma.sleep(1)

        LightingEffects.Rainbow("6 thru 10", 7, true)
        gma.sleep(1)

        LightingEffects.Strobe("1 thru 3", 5, 0.1, 0.2, 100)
    end

    function Cleanup()
        gma.echo("燈光效果工具箱已結束")
    end

    return Start, Cleanup
```

為什麼函式在 grandMA2 Plugin 開發中如此重要？

函式類型	Plugin 應用	實際用途
Start()	Plugin 進入點	系統初始化、主程式邏輯
Cleanup()	資源清理	關閉進度條、恢復狀態
工具函式	效果庫	可重複使用的燈光效果
輔助函式	使用者介面	輸入驗證、選單顯示
管理函式	系統管理	場景管理、設定載入

練習時間

基礎練習

1. **溫度轉換器**:
   寫函式轉換攝氏和華氏溫度
2. **成績評等**:
   寫函式根據分數回傳等級（A、B、C、D、F）
3. **grandMA2 專案練習**:
   - 建立一個包含 5 種不同效果的燈光 Plugin

小結

函式是程式設計中最強大的工具，在 grandMA2 plugin 開發中更是不可或缺:

1. **模組化設計**：每個函式負責特定功能，程式易讀易維護
2. **可重複使用**：寫一次，用很多次的燈光效果
3. **專業架構**：Start 和 Cleanup 是 plugin 的標準架構
4. **錯誤處理**：用函式包裝複雜邏輯，更容易處理錯誤
5. **實際應用**：建立專業級的燈光控制系統

掌握函式，你就掌握了建構複雜 grandMA2 plugin 的核心技能。現在你不只是會寫程式，更是會建立專業燈光控制工具的開發者!