EBook Gratuito

APPRENDIMENTO Lua

Free unaffiliated eBook created from
Stack Overflow contributors.

#lua

Sommario

Di

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: lua

It is an unofficial and free Lua ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Lua.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Capitolo 1: Iniziare con Lua

Osservazioni

Lua è un linguaggio di scripting minimalista, leggero e integrabile. È stato progettato, implementato e gestito da un team della PUC-Rio , la Pontificia Università Cattolica di Rio de Janeiro in Brasile. La mailing list è aperta per essere coinvolta.

Casi di utilizzo comuni per Lua includono la creazione di script di videogiochi, l'estensione di applicazioni con plug-in e config, l'involuzione di alcune logiche di business di alto livello o semplicemente l'incorporamento in dispositivi come TV, automobili, ecc.

Per attività ad alte prestazioni esiste un'implementazione indipendente che utilizza il compilatore just-in-time disponibile chiamato LuaJIT .

Versioni

Versione	Gli appunti	Data di rilascio
1.0	Versione iniziale, non pubblica.	1993/07/28
1.1	Prima versione pubblica. Documento della Conferenza che lo descrive.	1994/07/08
2.1	A partire da Lua 2.1, Lua è diventata disponibile gratuitamente per tutti gli scopi, compresi gli usi commerciali. Documento ufficiale che lo descrive.	1995/02/07
2.2	Stringhe lunghe, l'interfaccia di debug, migliori stackback di stack	1995/11/28
2.4	Compilatore `luac` esterno	1996/05/14
2.5	Funzioni di corrispondenza del modello e vararg.	1996/11/19
3.0	Introdotto auxlib, una libreria per aiutare a scrivere le librerie Lua	1997/07/01
3.1	Funzioni anonime e chiusure di funzioni tramite "upvalues".	1998/07/11
3.2	Libreria di debug e nuove funzioni di tabella	1999/07/08

Versione	Gli appunti	Data di rilascio
3.2.2		2000/02/22
4.0	Stati multipli, istruzioni "for", aggiornamento API.	2000/11/06
4.0.1		2002-07-04
5.0	Coroutine, metatables, scoping lessicale completo, chiamate di coda, booleans passano alla licenza MIT.	2003/04/11
5.0.3		2006-06-26
5.1	Revamp del sistema di moduli, garbage collector incrementale, metatables per tutti i tipi, luaconf.h revamp, parser completamente rientranti, argomenti variadici.	2006-02-21
5.1.5		2012-02-17
5.2	Garbage collector di emergenza, goto, finalizzatori per tavoli.	2011-12-16
5.2.4		2015/03/07
5.3	Supporto UTF-8 di base, operazioni bit a bit, interi 32/64 bit.	2015/01/12
5.3.4	Ultima versione.	2017/01/12

Examples

Installazione

Binari

I binari Lua sono forniti dalla maggior parte delle distribuzioni GNU / Linux come un pacchetto.

Ad esempio, su Debian, Ubuntu e le loro derivate può essere acquisito eseguendo questo:

```
sudo apt-get install lua50
```

```
sudo apt-get install lua51
```

```
sudo apt-get install lua52
```

Ci sono alcune build semi-ufficiali fornite per Windows, MacOS e alcuni altri sistemi operativi ospitati su SourceForge .

Gli utenti Apple possono anche installare facilmente Lua usando Homebrew :

```
brew install lua
```

(Attualmente Homebrew ha 5.2.4, per 5.3 vedere Homebrew / versioni .)

fonte

La fonte è disponibile nella pagina ufficiale . L'acquisizione di fonti e la costruzione di se stessa dovrebbero essere banali. Sui sistemi Linux dovrebbe essere sufficiente quanto segue:

```
$ wget http://lua.org/ftp/lua-5.3.3.tar.gz
$ echo "a0341bc3d1415b814cc738b2ec01ae56045d64ef ./lua-5.3.3.tar.gz" | sha1sum -c -
$ tar -xvf ./lua-5.3.3.tar.gz
$ make -C ./lua-5.3.3/ linux
```

Nell'esempio sopra stiamo fondamentalmente scaricando un `tarball` sorgente dal sito ufficiale, verificandone il checksum ed estraendo ed eseguendo `make` . (Controlla il checksum nella pagina ufficiale).

Nota: è necessario specificare quale target di build si desidera. Nell'esempio, abbiamo specificato `linux` . Altri obiettivi di costruzione disponibili includono `solaris` , `aix` , `bsd` , `freebsd` , `macosx` , `mingw` , ecc. Per ulteriori dettagli, `mingw doc/readme.html` , che è incluso nella fonte. (Puoi anche trovare l'ultima versione del README online .)

moduli

Le librerie standard sono limitate alle primitive:

- `coroutine` - funzionalità di gestione di coroutine
- `debug` - ganci e strumenti di debug
- `io` - I / O primitivi di base
- `package` - funzionalità di gestione dei moduli
- `string` e funzionalità di corrispondenza specifica del modello Lua
- `table` - primitive per trattare un tipo di Lua essenziale ma complesso - tabelle
- `os` - operazioni di base del sistema operativo
- `utf8` - primitive UTF-8 di base (dal Lua 5.3)

Tutte queste librerie possono essere disabilitate per una build specifica o caricate in fase di esecuzione.

Librerie e infrastrutture Lua di terze parti per la distribuzione di moduli sono scarse, ma migliorano. Progetti come LuaRocks , Lua Toolbox e LuaDist stanno migliorando la situazione. Molte informazioni e molti suggerimenti possono essere trovati sul vecchio Lua Wiki , ma sappiate che alcune di queste informazioni sono piuttosto vecchie e obsolete.

Commenti

Commenti a riga singola in Lua iniziano con `--` e continuano fino alla fine della riga:

```
-- this is single line comment
```

```
-- need another line
-- huh?
```

Blocca i commenti che iniziano con `--[[` e termina con `]]` :

```
--[[
    This is block comment.
    So, it can go on...
    and on...
    and on....
]]
```

I commenti di blocco utilizzano lo stesso stile di delimitatore di stringhe lunghe; qualsiasi numero di segni di uguale può essere aggiunto tra parentesi per delimitare un commento:

```
--[=[
    This is also a block comment
    We can include "]]" inside this comment
--]=]

--[==[
    This is also a block comment
    We can include "]=]" inside this comment
--]==]
```

Un trucco per commentare blocchi di codice è circondarlo con `--[[` e `--]]` :

```
--[[
    print'Lua is lovely'
--]]
```

Per riattivare il blocco, aggiungi semplicemente una `-` alla sequenza di apertura del commento:

```
---[[
    print'Lua is lovely'
--]]
```

In questo modo, la sequenza `--` nella prima riga inizia un commento a riga singola, proprio come l'ultima riga, e l'istruzione di `print` non è commentata.

Facendo un ulteriore passo avanti, due blocchi di codice possono essere impostati in modo tale che se il primo blocco viene commentato, il secondo non lo sarà, e viceversa:

```
---[[
  print 'Lua is love'
--[=[]]
  print 'Lua is life'
--]=]
```

Per attivare il secondo blocco mentre disabiliti il primo blocco, elimina l'iniziale `-` nella prima riga:

```
--[[
```

```
  print 'Lua is love'
--[=[]]
  print 'Lua is life'
--]=]
```

Esecuzione di programmi Lua

Di solito, Lua viene spedito con due file binari:

- `lua` - interprete indipendente e shell interattiva
- compilatore `luac` -bytecode

Diciamo che abbiamo un programma di esempio (`bottles_of_mate.lua`) come questo:

```
local string = require "string"

function bottle_take(bottles_available)

    local count_str = "%d bottles of mate on the wall."
    local take_str = "Take one down, pass it around, " .. count_str
    local end_str = "Oh noes, " .. count_str
    local buy_str = "Get some from the store, " .. count_str
    local bottles_left = 0

    if bottles_available > 0 then
         print(string.format(count_str, bottles_available))
         bottles_left = bottles_available - 1
         print(string.format(take_str, bottles_left))
    else
        print(string.format(end_str, bottles_available))
        bottles_left = 99
        print(string.format(buy_str, bottles_left))
    end

    return bottles_left
end

local bottle_count = 99

while true do
    bottle_count = bottle_take(bottle_count)
end
```

Il programma stesso può essere eseguito eseguendo in seguito sulla tua shell:

```
$ lua bottles_of_mate.lua
```

L'output dovrebbe assomigliare a questo, correndo nel ciclo infinito:

```
Get some from the store, 99 bottles of mate on the wall.
99 bottles of mate on the wall.
Take one down, pass it around, 98 bottles of mate on the wall.
98 bottles of mate on the wall.
Take one down, pass it around, 97 bottles of mate on the wall.
97 bottles of mate on the wall.
```

```
...
...
3 bottles of mate on the wall.
Take one down, pass it around, 2 bottles of mate on the wall.
2 bottles of mate on the wall.
Take one down, pass it around, 1 bottles of mate on the wall.
1 bottles of mate on the wall.
Take one down, pass it around, 0 bottles of mate on the wall.
Oh noes, 0 bottles of mate on the wall.
Get some from the store, 99 bottles of mate on the wall.
99 bottles of mate on the wall.
Take one down, pass it around, 98 bottles of mate on the wall.
...
```

Puoi compilare il programma nel bytecode di Lua eseguendo il seguente comando sulla tua shell:

```
$ luac bottles_of_mate.lua -o bottles_of_mate.luac
```

Anche l'elenco bytecode è disponibile eseguendo quanto segue:

```
$ luac -l bottles_of_mate.lua


main <./bottles.lua:0,0> (13 instructions, 52 bytes at 0x101d530)
0+ params, 4 slots, 0 upvalues, 2 locals, 4 constants, 1 function
    1    [1]    GETGLOBAL    0 -1    ; require
    2    [1]    LOADK        1 -2    ; "string"
    3    [1]    CALL         0 2 2
    4    [22]    CLOSURE      1 0    ; 0x101d710
    5    [22]    MOVE         0 0
    6    [3]    SETGLOBAL    1 -3    ; bottle_take
    7    [24]    LOADK        1 -4    ; 99
    8    [27]    GETGLOBAL    2 -3    ; bottle_take
    9    [27]    MOVE         3 1
    10    [27]    CALL         2 2 2
    11    [27]    MOVE         1 2
    12    [27]    JMP          -5    ; to 8
    13    [28]    RETURN       0 1

function <./bottles.lua:3,22> (46 instructions, 184 bytes at 0x101d710)
1 param, 10 slots, 1 upvalue, 6 locals, 9 constants, 0 functions
    1    [5]    LOADK        1 -1    ; "%d bottles of mate on the wall."
    2    [6]    LOADK        2 -2    ; "Take one down, pass it around, "
    3    [6]    MOVE         3 1
    4    [6]    CONCAT       2 2 3
    5    [7]    LOADK        3 -3    ; "Oh noes, "
    6    [7]    MOVE         4 1
    7    [7]    CONCAT       3 3 4
    8    [8]    LOADK        4 -4    ; "Get some from the store, "
    9    [8]    MOVE         5 1
    10    [8]    CONCAT       4 4 5
    11    [9]    LOADK        5 -5    ; 0
    12    [11]    EQ           1 0 -5    ; - 0
    13    [11]    JMP          16    ; to 30
    14    [12]    GETGLOBAL    6 -6    ; print
    15    [12]    GETUPVAL     7 0    ; string
    16    [12]    GETTABLE     7 7 -7    ; "format"
    17    [12]    MOVE         8 1
    18    [12]    MOVE         9 0
```

```
    19    [12]    CALL         7 3 0
    20    [12]    CALL         6 0 1
    21    [13]    SUB          5 0 -8    ; - 1
    22    [14]    GETGLOBAL    6 -6    ; print
    23    [14]    GETUPVAL     7 0    ; string
    24    [14]    GETTABLE     7 7 -7    ; "format"
    25    [14]    MOVE         8 2
    26    [14]    MOVE         9 5
    27    [14]    CALL         7 3 0
    28    [14]    CALL         6 0 1
    29    [14]    JMP          15    ; to 45
    30    [16]    GETGLOBAL    6 -6    ; print
    31    [16]    GETUPVAL     7 0    ; string
    32    [16]    GETTABLE     7 7 -7    ; "format"
    33    [16]    MOVE         8 3
    34    [16]    MOVE         9 0
    35    [16]    CALL         7 3 0
    36    [16]    CALL         6 0 1
    37    [17]    LOADK        5 -9    ; 99
    38    [18]    GETGLOBAL    6 -6    ; print
    39    [18]    GETUPVAL     7 0    ; string
    40    [18]    GETTABLE     7 7 -7    ; "format"
    41    [18]    MOVE         8 4
    42    [18]    MOVE         9 5
    43    [18]    CALL         7 3 0
    44    [18]    CALL         6 0 1
    45    [21]    RETURN       5 2
    46    [22]    RETURN       0 1
```

Iniziare

variabili

```
var = 50 -- a global variable
print(var) --> 50
do
  local var = 100 -- a local variable
  print(var) --> 100
end
print(var) --> 50
-- The global var (50) still exists
-- The local var (100) has gone out of scope and can't be accessed any longer.
```

tipi

```
num = 20 -- a number
num = 20.001 -- still a number
str = "zaldrizes buzdari iksos daor" -- a string
tab = {1, 2, 3} -- a table (these have their own category)
bool = true -- a boolean value
bool = false -- the only other boolean value
print(type(num)) --> 'number'
print(type(str)) --> 'string'
print(type(bool)) --> 'boolean'
type(type(num)) --> 'string'
```

```
-- Functions are a type too, and first-class values in Lua.
print(type(print)) --> prints 'function'
old_print = print
print = function (x) old_print "I'm ignoring the param you passed me!" end
old_print(type(print)) --> Still prints 'function' since it's still a function.
-- But we've (unhelpfully) redefined the behavior of print.
print("Hello, world!") --> prints "I'm ignoring the param you passed me!"
```

Il tipo speciale `nil`

Un altro tipo in Lua è `nil` . L'unico valore nel `nil` tipo è `nil` . `nil` esiste per essere diverso da tutti gli altri valori in Lua. È un tipo di valore non valore.

```
print(foo) -- This prints nil since there's nothing stored in the variable 'foo'.
foo = 20
print(foo) -- Now this prints 20 since we've assigned 'foo' a value of 20.

-- We can also use `nil` to undefine a variable
foo = nil -- Here we set 'foo' to nil so that it can be garbage-collected.

if nil then print "nil" end --> (prints nothing)
-- Only false and nil are considered false; every other value is true.
if 0 then print "0" end --> 0
if "" then print "Empty string!" --> Empty string!
```

espressioni

```
a = 3
b = a + 20 a = 2 print(b, a) -- hard to read, can also be written as
b = a + 20; a = 2; print(a, b) -- easier to read, ; are optional though
true and true --> returns true
true and 20 --> 20
false and 20 --> false
false or 20 --> 20
true or 20 --> true
tab or {}
  --> returns tab if it is defined
  --> returns {} if tab is undefined
  -- This is useful when we don't know if a variable exists
tab = tab or {} -- tab stays unchanged if it exists; tab becomes {} if it was previously nil.

a, b = 20, 30 -- this also works
a, b = b, a -- switches values
```

Definizione di funzioni

```
function name(parameter)
    return parameter
end
print(name(20)) --> 20
-- see function category for more information
name = function(parameter) return parameter end -- Same as above
```

booleani

Solo `false` e `nil` valutano come false, tutto il resto, incluso `0` e la stringa vuota vengono valutate come vere.

raccolta dei rifiuti

```
tab = {"lots", "of", "data"}
tab = nil; collectgarbage()
-- tab does no longer exist, and doesn't take up memory anymore.
```

tavoli

```
tab1 = {"a", "b", "c"}
tab2 = tab1
tab2[1] = "d"
print(tab1[1]) --> 'd' -- table values only store references.
--> assigning tables does not copy its content, only the reference.

tab2 = nil; collectgarbage()
print(tab1) --> (prints table address) -- tab1 still exists; it didn't get garbage-collected.

tab1 = nil; collectgarbage()
-- No more references. Now it should actually be gone from memory.
```

Queste sono le basi, ma c'è una sezione sulle tabelle con maggiori informazioni.

condizioni

```
if (condition) then
    -- do something
elseif (other_condition) then
    -- do something else
else
    -- do something
end
```

per loops

Ci sono due tipi di loop `for` in Lua: un ciclo numerico `for` loop e un ciclo `for` generico.

- Un ciclo numerico `for` ha la seguente forma:

  ```
  for a=1, 10, 2 do -- for a starting at 1, ending at 10, in steps of 2
    print(a) --> 1, 3, 5, 7, 9
  end
  ```

  La terza espressione in un ciclo numerico `for` è il passo in base al quale il ciclo aumenterà.

Ciò facilita i cicli di inversione:

```
for a=10, 1, -1 do
  print(a) --> 10, 9, 8, 7, 6, etc.
end
```

Se l'espressione del passo viene omessa, Lua assume un passo predefinito di 1.

```
for a=1, 10 do
  print(a) --> 1, 2, 3, 4, 5, etc.
end
```

Si noti inoltre che la variabile loop è locale al ciclo `for` . Non esisterà dopo che il ciclo è finito.

- Generico `for` cicli lavoro attraverso tutti i valori che una funzione iteratore restituisce:

```
for key, value in pairs({"some", "table"}) do
  print(key, value)
  --> 1 some
  --> 2 table
end
```

Lua fornisce diverse costruita nel iteratori (ad esempio, `pairs` , `ipairs`), e gli utenti possono definire i propri iteratori ordinazione pure da utilizzare con generico `for` cicli.

fare blocchi

```
local a = 10
do
    print(a) --> 10
    local a = 20
    print(a) --> 20
end
print(a) --> 10
```

Alcune cose complicate

A volte Lua non si comporta come si potrebbe pensare dopo aver letto la documentazione. Alcuni di questi casi sono:

Nil e Nothing non sono gli stessi (COMUNE PITFALL!)

Come previsto, `table.insert(my_table, 20)` aggiunge il valore `20` alla tabella e `table.insert(my_table, 5, 20)` aggiunge il valore 20 alla 5a posizione. Cosa fanno invece `table.insert(my_table, 5, nil)` ? Ci si potrebbe aspettare che trattino `nil` come nessun argomento, e inserisca il valore `5` alla fine della tabella, ma in realtà aggiunge il valore `nil` alla 5a

posizione della tabella. Quando è un problema?

```
(function(tab, value, position)
    table.insert(tab, position or value, position and value)
end)({}, 20)
-- This ends up calling table.insert({}, 20, nil)
-- and this doesn't do what it should (insert 20 at the end)
```

Una cosa simile accade con il `tostring()` :

```
print (tostring(nil)) -- this prints "nil"
table.insert({}, 20) -- this returns nothing
-- (not nil, but actually nothing (yes, I know, in lua those two SHOULD
-- be the same thing, but they aren't))

-- wrong:
print (tostring( table.insert({}, 20) ))
-- throws error because nothing ~= nil

--right:
local _tmp = table.insert({}, 20) -- after this _tmp contains nil
print(tostring(_tmp)) -- prints "nil" because suddenly nothing == nil
```

Ciò potrebbe anche causare errori durante l'utilizzo di codice di terze parti. Se, ad esempio, la documentazione di alcuni stati di funzione "restituisce ciambelle se fortunati, altrimenti nulla", l'implementazione *potrebbe* sembrare un po 'come questa

```
function func(lucky)
    if lucky then
        return "donuts"
    end
end
```

questa implementazione potrebbe sembrare ragionevole all'inizio; restituisce le ciambelle quando deve, e quando si digita `result = func(false)` result conterrà il valore `nil` .

Tuttavia, se si scrivesse la `print(tostring(func(false)))` lua genererebbe un errore simile a questo `stdin:1: bad argument #1 to 'tostring' (value expected)`

Perché? `tostring` ottiene chiaramente una discussione, anche se è `nil` . Sbagliato. func non restituisce nulla, quindi `tostring(func(false))` è lo stesso di `tostring()` e NOT lo stesso di `tostring(nil)` .

Errori che dicono "valore atteso" sono una forte indicazione che questa potrebbe essere la fonte del problema.

Lasciare spazi vuoti negli array

Questa è una trappola enorme se sei nuovo a lua e ci sono molte informazioni nella categoria delle tabelle

Ciao mondo

Questo è ciao codice mondiale:

```
print("Hello World!")
```

Come funziona? È semplice! Lua esegue la funzione `print()` e usa la stringa `"Hello World"` come argomento.

Leggi Iniziare con Lua online: https://riptutorial.com/it/lua/topic/659/iniziare-con-lua

Capitolo 2: Argomenti Variadici

introduzione

I vararg , come sono comunemente noti, consentono alle funzioni di assumere un numero arbitrario di argomenti senza specifica. Tutti gli argomenti dati a tale funzione sono *raggruppati* in un'unica struttura nota come *lista vararg* ; che è scritto come `...` in Lua. Esistono metodi di base per estrarre il numero di argomenti dati e il valore di tali argomenti usando la funzione `select()` , ma schemi di utilizzo più avanzati possono sfruttare la struttura per la sua piena utilità.

Sintassi

- *...* - Crea la funzione i cui argomenti elencano in cui appare una funzione variadica
- *select (what, ...)* - Se 'what' è un numero compreso nell'intervallo 1 al numero di elementi nel vararg, restituisce 'what'th all'ultimo elemento nel vararg. Il ritorno sarà nullo se l'indice è fuori limite. Se 'what' è la stringa '#', restituisce il numero di elementi nel vararg.

Osservazioni

Efficienza

L'elenco vararg è implementato come elenco collegato nell'implementazione PUC-Rio della lingua, questo significa che gli indici sono O (n). Ciò significa che l'iterazione degli elementi in un vararg usando `select()` , come nell'esempio seguente, è un'operazione O (n ^ 2).

```
for i = 1, select('#', ...) do
    print(select(i, ...))
end
```

Se si pianifica di iterare sugli elementi in una lista vararg, prima imballare l'elenco in una tabella. Gli accessi alle tabelle sono O (1), quindi l'iterazione è O (n) in totale. Oppure, se sei così inclinato, vedi l'esempio di `foldr()` dalla sezione di utilizzo avanzato; usa ricorsione per scorrere su una lista di vararg in O (n).

Definizione della lunghezza della sequenza

Il vararg è utile in quanto la lunghezza del vararg rispetta qualsiasi nils esplicitamente passato (o calcolato). Per esempio.

```
function test(...)
    return select('#', ...)
end

test()             --> 0
test(nil, 1, nil)  --> 3
```

Questo comportamento è in conflitto con il comportamento delle tabelle, tuttavia, in cui l'operatore `#` lunghezza non funziona con "buchi" (nils incorporati) in sequenze. Calcolare la lunghezza di una tabella con buchi non è definito e non può essere considerato affidabile. Quindi, a seconda dei valori in `...` , prendere la lunghezza di `{...}` potrebbe non dare la risposta *'corretta'* . In Lua 5.2+ è stato introdotto `table.pack()` per gestire questa deficienza (esiste una funzione nell'esempio che implementa questa funzione in puro Lua).

Uso idiomatico

Poiché le varici trasportano la loro lunghezza, le persone le usano come sequenze per evitare il problema con i buchi nelle tabelle. Questo non era il loro uso previsto e l'implementazione di riferimento di Lua non è ottimizzata per. Sebbene tale uso sia esplorato negli esempi, generalmente è disapprovato.

Examples

Nozioni di base

Le funzioni Variadic vengono create utilizzando la sintassi `...` ellissi nella lista degli argomenti della definizione della funzione.

```
function id(...)
    return
end
```

Se hai chiamato questa funzione come `id(1, 2, 3, 4, 5)` allora `...` (AKA la lista vararg) conterrebbe i valori `1, 2, 3, 4, 5` .

Le funzioni possono accettare argomenti richiesti e `...`

```
function head(x, ...)
    return x
end
```

Il modo più semplice per estrarre elementi dalla lista vararg è semplicemente assegnargli delle variabili.

```
function head3(...)
    local a, b, c = ...
    return a, b, c
end
```

`select()` può anche essere usato per trovare il numero di elementi ed estrarre elementi da `...` indirettamente.

```
function my_print(...)
    for i = 1, select('#', ...) do
        io.write(tostring(select(i, ...)) .. '\t')
    end
```

```
    io.write '\n'
end
```

`...` può essere inserito in una tabella per facilità d'uso, usando `{...}` . Questo pone tutti gli argomenti nella parte sequenziale della tabella.

5.2

`table.pack(...)` può anche essere utilizzato per impacchettare la lista vararg in una tabella. Il vantaggio di `table.pack(...)` è che imposta il campo `n` della tabella restituita sul valore di `select('#', ...)` . Questo è importante se il tuo elenco di argomenti può contenere nils (vedi la sezione commenti sotto).

```
function my_tablepack(...)
    local t = {...}
    t.n = select('#', ...)
    return t
end
```

La lista vararg può anche essere restituita da funzioni. Il risultato è più ritorni.

```
function all_or_none(...)
    local t = table.pack(...)
    for i = 1, t.n do
        if not t[i] then
            return    -- return none
        end
    end
    return ...    -- return all
end
```

Uso avanzato

Come affermato negli esempi di base, è possibile avere argomenti con limiti variabili e l'elenco di argomenti variabili (`...`). Puoi usare questo fatto per separare ricorsivamente una lista come faresti in altre lingue (come Haskell). Di seguito è riportata un'implementazione di `foldr()` che sfrutta questo. Ogni chiamata ricorsiva lega la testa dell'elenco vararg a `x` e passa il resto dell'elenco a una chiamata ricorsiva. Questo distrugge l'elenco finché non c'è un solo argomento (`select('#', ...) == 0`). Successivamente, ciascun valore viene applicato all'argomento della funzione `f` con il risultato precedentemente calcolato.

```
function foldr(f, ...)
    if select('#', ...) < 2 then return ... end
    local function helper(x, ...)
        if select('#', ...) == 0 then
          return x
        end
        return f(x, helper(...))
    end
    return helper(...)
end
```

```
function sum(a, b)
    return a + b
end

foldr(sum, 1, 2, 3, 4)
--> 10
```

Puoi trovare altre definizioni di funzioni che sfruttano questo stile di programmazione qui nel numero 3 attraverso il numero 8.

L'unica struttura dati idiomatica di Lua è la tabella. L'operatore della lunghezza della tabella non è definito se vi sono `nil` in una sequenza. A differenza delle tabelle, la lista vararg rispetta esplicitamente `nil` s come indicato negli esempi di base e nella sezione dei commenti (si prega di leggere quella sezione se non l'hai ancora). Con poco lavoro la lista vararg può eseguire ogni operazione che una tabella può oltre alla mutazione. Ciò rende la lista dei vararg un buon candidato per l'implementazione di tuple immutabili.

```
function tuple(...)
    -- packages a vararg list into an easily passable value
    local co = coroutine.wrap(function(...)
        coroutine.yield()
        while true do
            coroutine.yield(...)
        end
    end)
    co(...)
    return co
end

local t = tuple((function() return 1, 2, nil, 4, 5 end)())

print(t())                  --> 1    2    nil    4    5    | easily unpack for multiple args
local a, b, d = t()         --> a = 1, b = 2, c = nil      | destructure the tuple
print((select(4, t())))     --> 4                          | index the tuple
print(select('#', t()))     --> 5                          | find the tuple arity (nil
respecting)

local function change_index(tpl, i, v)
    -- sets a value at an index in a tuple (non-mutating)
    local function helper(n, x, ...)
        if select('#', ...) == 0 then
            if n == i then
                return v
            else
                return x
            end
        else
            if n == i then
                return v, helper(n+1, ...)
            else
                return x, helper(n+1, ...)
            end
        end
    end
    return tuple(helper(1, tpl()))
end

local n = change_index(t, 3, 3)
```

```
print(t())                  --> 1    2    nil    4    5
print(n())                  --> 1    2    3    4    5
```

La principale differenza tra ciò che è sopra e le tabelle è che le tabelle sono mutabili e hanno la semantica del puntatore, dove la tupla non ha quelle proprietà. Inoltre, le tuple possono contenere esplicitamente `nil` e hanno un'operazione di lunghezza mai definita.

Leggi Argomenti Variadici online: https://riptutorial.com/it/lua/topic/4475/argomenti-variadici

Capitolo 3: Booleani a Lua

Osservazioni

Booleans, verità e falsità sono semplici in Lua. Revisionare:

1. Esiste un tipo booleano con esattamente due valori: `true` e `false` .
2. In un contesto condizionale (`if` , `elseif` , `while` , `until`), non è richiesto un booleano. Qualsiasi espressione può essere utilizzata.
3. In un contesto condizionale, il `false` e il `nil` falsi e tutto il resto è vero.
4. Anche se 3 implica già questo: se provieni da altre lingue, ricorda che `0` e il conteggio delle stringhe vuoto come vero nei contesti condizionali in Lua.

Examples

Il tipo booleano

Booleans e altri valori

Quando si ha a che fare con lua è importante distinguere tra i valori booleani `true` e `false` e quelli che valutano true o false.

Ci sono solo due valori in lua che valutano in falso: `nil` e `false` , mentre tutto il resto, incluso il valore numerico `0` valutato su vero.

Alcuni esempi di cosa significa:

```
if 0 then print("0 is true") end --> this will print "true"
if (2 == 3) then print("true") else print("false") end --> this prints "false"
if (2 == 3) == false then print("true") end --> this prints "true"
if (2 == 3) == nil then else print("false") end
--> prints false, because even if nil and false both evaluate to false,
--> they are still different things.
```

Operazioni logiche

Gli operatori logici in lua non restituiscono necessariamente valori booleani:

`and` restituirà il secondo valore se il primo valore è true;

`or` restituisce il secondo valore se il primo valore restituisce false;

Questo rende possibile simulare l'operatore ternario, proprio come in altre lingue:

```
local var = false and 20 or 30 --> returns 30
local var = true and 20 or 30 --> returns 20
-- in C: false ? 20 : 30
```

Questo può anche essere usato per inizializzare le tabelle se non esistono

```
tab = tab or {} -- if tab already exists, nothing happens
```

o per evitare di utilizzare le istruzioni if, rendendo il codice più facile da leggere

```
print(debug and "there has been an error") -- prints "false" line if debug is false
debug and print("there has been an error") -- does nothing if debug is false
-- as you can see, the second way is preferable, because it does not output
-- anything if the condition is not met, but it is still possible.
-- also, note that the second expression returns false if debug is false,
-- and whatever print() returns if debug is true (in this case, print returns nil)
```

Verifica se le variabili sono definite

Si può anche facilmente verificare se esiste una variabile (se è definita), poiché le variabili inesistenti restituiscono `nil` , che restituisce false.

```
local tab_1, tab_2 = {}
if tab_1 then print("table 1 exists") end --> prints "table 1 exists"
if tab_2 then print("table 2 exists") end --> prints nothing
```

L'unico caso in cui questo non si applica è quando una variabile memorizza il valore `false` , nel qual caso esiste tecnicamente ma continua a essere considerato falso. Per questo motivo, è un cattivo progetto creare funzioni che restituiscono `false` e `nil` seconda dello stato o dell'input. Possiamo comunque verificare se abbiamo un valore `nil` o `false` :

```
if nil == nil then print("A nil is present") else print("A nil is not present") end
if false == nil then print("A nil is present") else print("A nil is not present") end
-- The output of these calls are:
-- A nil is present!
-- A nil is not present
```

Contesti condizionali

I contesti condizionali in Lua (`if` , `elseif` , `while` , `until`) non richiedono un valore booleano. Come molte lingue, qualsiasi valore Lua può apparire in una condizione. Le regole per la valutazione sono semplici:

1. `false` e `nil` contano come false.
2. Tutto il resto conta come vero.

```
if 1 then
```

```
  print("Numbers work.")
end
if 0 then
  print("Even 0 is true")
end

if "strings work" then
  print("Strings work.")
end
if "" then
  print("Even the empty string is true.")
end
```

Operatori logici

In Lua, i booleani possono essere manipolati tramite *operatori logici* . Questi operatori includono `not` , `and` , and `or` .

Nelle espressioni semplici, i risultati sono abbastanza semplici:

```
print(not true) --> false
print(not false) --> true
print(true or false) --> true
print(false and true) --> false
```

Ordine di precedenza

L'ordine di precedenza è simile agli operatori matematici unary `-` , `*` e `+` :

- `not`
- allora `and`
- allora `or`

Questo può portare a espressioni complesse:

```
print(true and false or not false and not true)
print( (true and false) or ((not false) and (not true)) )
    --> these are equivalent, and both evaluate to false
```

Scorciatoia valutazione

Gli operatori `and` e `or` potrebbero essere valutati solo utilizzando il primo operando, a condizione che il secondo non sia necessario:

```
function a()
    print("a() was called")
    return true
end
```

```
function b()
    print("b() was called")
    return false
end

print(a() or b())
    --> a() was called
    --> true
    --  nothing else
print(b() and a())
    --> b() was called
    --> false
    --  nothing else
print(a() and b())
    --> a() was called
    --> b() was called
    --> false
```

Operatore condizionale idiomatico

A causa della precedenza degli operatori logici, della capacità di valutazione della scorciatoia e della valutazione dei valori non `false` e non `nil` come `true` , un operatore condizionale idiomatico è disponibile in Lua:

```
function a()
    print("a() was called")
    return false
end
function b()
    print("b() was called")
    return true
end
function c()
    print("c() was called")
    return 7
end

print(a() and b() or c())
    --> a() was called
    --> c() was called
    --> 7

print(b() and c() or a())
    --> b() was called
    --> c() was called
    --> 7
```

Inoltre, a causa della natura della struttura `x and a or b` , `a` non verrà mai *restituito* se viene valutato come `false` , questo condizionale restituirà sempre `b` indipendentemente da cosa sia `x` .

```
print(true and false or 1)  -- outputs 1
```

Tabelle di verità

Gli operatori logici in Lua non "restituiscono" booleano, ma uno dei loro argomenti. Usando `nil` per false e numeri per true, ecco come si comportano.

```
print(nil and nil)       -- nil
print(nil and 2)         -- nil
print(1 and nil)         -- nil
print(1 and 2)           -- 2

print(nil or nil)        -- nil
print(nil or 2)          -- 2
print(1 or nil)          -- 1
print(1 or 2)            -- 1
```

Come puoi vedere, Lua restituirà sempre il primo valore che fa *fallire* o *avere successo* . Ecco le tabelle di verità che mostrano questo.

```
  x   |  y   || and           x   |  y   || or
----------------------     ----------------------
false|false||  x           false|false||  y
false|true ||  x           false|true ||  y
true |false||  y           true |false||  x
true |true ||  y           true |true ||  x
```

Per chi ne ha bisogno, ecco due funzioni che rappresentano questi operatori logici.

```
function exampleAnd(value1, value2)
  if value1 then
    return value2
  end
  return value1
end

function exampleOr(value1, value2)
  if value1 then
    return value1
  end
  return value2
end
```

Emulazione dell'operatore ternario con "e" o "operatori logici".

In lua, gli operatori logici `and` e `or` restituiscono uno degli operandi come risultato invece di un risultato booleano. Di conseguenza, questo meccanismo può essere sfruttato per emulare il comportamento dell'operatore ternario nonostante non abbia un operatore ternario "reale" nella lingua.

Sintassi

condizione **e** *truthy_expr* **o** *falsey_expr*

Utilizzare nell'assegnazione / inizializzazione variabile

```
local drink = (fruit == "apple") and "apple juice" or "water"
```

Utilizzare nel costruttore di tabelle

```
local menu =
{
  meal  = vegan and "carrot" or "steak",
  drink = vegan and "tea"    or "chicken soup"
}
```

Usa come argomento di funzione

```
print(age > 18 and "beer" or "fruit punch")
```

Utilizzare nella dichiarazione di ritorno

```
function get_gradestring(student)
  return student.grade > 60 and "pass" or "fail"
end
```

Avvertimento

Ci sono situazioni in cui questo meccanismo non ha il comportamento desiderato. Considera questo caso

```
local var = true and false or "should not happen"
```

In un operatore ternario "reale", il valore previsto di `var` è `false` . In lua, tuttavia, la valutazione `and` "cade attraverso" perché il secondo operando è falso. Di conseguenza, `var should not happen` .

Due possibili soluzioni a questo problema, refactoring questa espressione in modo che l'operando medio non è falso. per esempio.

```
local var = not true and "should not happen" or false
```

o in alternativa, usa il classico `if then else` costruisci.

Leggi Booleani a Lua online: https://riptutorial.com/it/lua/topic/3101/booleani-a-lua

Capitolo 4: coroutine

Sintassi

- coroutine.create (funzione) restituisce una coroutine (tipo (coroutine) == 'thread') contenente la funzione.
- coroutine.resume (co, ...) riprendi o avvia la coroutine. Qualsiasi argomento addizionale dato per riprendere viene restituito dal coroutine.yield () che precedentemente ha sospeso la coroutine. Se non è stata avviata la coroutine, gli argomenti aggiuntivi diventano gli argomenti della funzione.
- coroutine.yield (...) restituisce la coroutine attualmente in esecuzione. L'esecuzione riprende dopo la chiamata a coroutine.resume () che ha avviato quella coroutine. Qualsiasi argomento fornito per la resa viene restituito dalla corrispondente coroutine.resume () che ha avviato la coroutine.
- coroutine.status (co) restituisce lo stato della coroutine, che può essere:
    - "dead": la funzione nella coroutine ha raggiunto la sua fine e la coroutine non può più essere ripresa
    - "in esecuzione": la coroutine è stata ripresa ed è in esecuzione
    - "normale": la coroutine ha ripreso un'altra coroutine
    - "sospeso": la coroutine ha ceduto ed è in attesa di ripresa
- coroutine.wrap (funzione) restituisce una funzione che quando viene richiamata riprende la coroutine che sarebbe stata creata da coroutine.create (funzione).

Osservazioni

Il sistema di coroutine è stato implementato in lua per emulare il multithreading esistente in altre lingue. Funziona passando ad una velocità estremamente alta tra le diverse funzioni in modo che l'utente umano pensi di essere eseguito nello stesso momento.

Examples

Crea e usa una coroutine

Tutte le funzioni per interagire con le coroutine sono disponibili nella tabella delle **coroutine** . Viene creata una nuova coroutine utilizzando la funzione **coroutine.create** con un singolo argomento: una funzione con il codice da eseguire:

```
thread1 = coroutine.create(function()
            print("honk")
        end)
```

```
print(thread1)
-->> thread: 6b028b8c
```

Un oggetto di coroutine restituisce il valore di tipo **thread** , che rappresenta una nuova coroutine. Quando viene creata una nuova coroutine, il suo stato iniziale viene sospeso:

```
print(coroutine.status(thread1))
-->> suspended
```

Per riprendere o avviare una coroutine, viene utilizzata la funzione **coroutine.resume** , il primo argomento fornito è l'oggetto thread:

```
coroutine.resume(thread1)
-->> honk
```

Ora la coroutine esegue il codice e termina, cambiando il suo stato in **dead** , che non può essere ripreso.

```
print(coroutine.status(thread1))
-->> dead
```

Coroutine possono sospendere la sua esecuzione e riprenderla in seguito grazie alla funzione **coroutine.yield** :

```
thread2 = coroutine.create(function()
        for n = 1, 5 do
            print("honk "..n)
            coroutine.yield()
        end
    end)
```

Come puoi vedere, **coroutine.yield ()** è presente all'interno del ciclo for, ora quando riprendiamo la coroutine, eseguirà il codice fino a raggiungere una coroutine.yield:

```
coroutine.resume(thread2)
-->> honk 1
coroutine.resume(thread2)
-->> honk 2
```

Dopo aver terminato il ciclo, lo stato del thread diventa **morto** e non può essere ripreso. Coroutine consente anche lo scambio tra i dati:

```
thread3 = coroutine.create(function(complement)
    print("honk "..complement)
    coroutine.yield()
    print("honk again "..complement)
end)
coroutine.resume(thread3, "stackoverflow")
-->> honk stackoverflow
```

Se la coroutine viene eseguita di nuovo senza argomenti aggiuntivi, il *complemento* rimarrà

comunque l'argomento dal primo curriculum, in questo caso "stackoverflow":

```
coroutine.resume(thread3)
-->> honk again stackoverflow
```

Infine, quando termina una coroutine, qualsiasi valore restituito dalla sua funzione va al curriculum corrispondente:

```
thread4 = coroutine.create(function(a, b)
    local c = a+b
    coroutine.yield()
    return c
end)
coroutine.resume(thread4, 1, 2)
print(coroutine.resume(thread4))
-->> true, 3
```

Le Coroutine vengono utilizzate in questa funzione per trasferire i valori a un thread chiamante da una chiamata ricorsiva profonda.

```
local function Combinations(l, r)
    local ll = #l
    r = r or ll
    local sel = {}
    local function rhelper(depth, last)
        depth = depth or 1
        last = last or 1
        if depth > r then
            coroutine.yield(sel)
        else
            for i = last, ll - (r - depth) do
                sel[depth] = l[i]
                rhelper(depth+1, i+1)
            end
        end
    end
    return coroutine.wrap(rhelper)
end

for v in Combinations({1, 2, 3}, 2) do
    print("{"..table.concat(v, ", ")..."}")
end
--> {1, 2}
--> {1, 3}
--> {2, 3}
```

Le coroutine possono anche essere usate per la valutazione pigra.

```
-- slices a generator 'c' taking every 'step'th output from the generator
-- starting at the 'start'th output to the 'stop'th output
function slice(c, start, step, stop)
    local _
    return coroutine.wrap(function()
        for i = 1, start-1 do
            _ = c()
        end
```

```
        for i = start, stop do
            if (i - start) % step == 0 then
                coroutine.yield(c())
            else
                _ = c()
            end
        end
    end)
end


local alphabet = {}
for c = string.byte('a'), string.byte('z') do
    alphabet[#alphabet+1] = string.char(c)
end
-- only yields combinations 100 through 102
-- requires evaluating the first 100 combinations, but not the next 5311633
local s = slice(Combinations(alphabet, 10), 100, 1, 102)
for i in s do
    print(table.concat(i))
end
--> abcdefghpr
--> abcdefghps
--> abcdefghpt
```

Le coroutine possono essere utilizzate per i costrutti di tubazioni come descritto in Programmazione in Lua . L'autore di PiL, Roberto Ierusalimschy, ha anche pubblicato un articolo sull'uso delle coroutine per implementare meccanismi di controllo del flusso più avanzati e generali come le continuazioni.

Leggi coroutine online: https://riptutorial.com/it/lua/topic/3410/coroutine

Capitolo 5: funzioni

Sintassi

- *funcname* = function (paramA, paramB, ...) corpo; return exprlist end - una funzione semplice
- function *functionname* (paramA, paramB, ...) corpo; return exprlist end - stenografia per sopra
- *nome di* funzione locale = funzione (paramA, paramB, ...); return exprlist end - a lambda
- *funcname* locale; *funcname* = function (paramA, paramB, ...) corpo; return exprlist end - lambda che può effettuare chiamate ricorsive
- *corpo della* funzione funzionale locale (paramA, paramB, ...); return exprlist end - stenografia per sopra
- *funcname* (paramA, paramB, ...) - chiama una funzione
- local *var* = *var* o "Default" - un parametro predefinito
- return nil, "messaggi di errore" - metodo standard per annullare un errore

Osservazioni

Le funzioni sono generalmente impostate con la `function a(b,c) ... end` e raramente con l'impostazione di una variabile su una funzione anonima (`a = function(a,b) ... end`). È vero il contrario quando si passano le funzioni come parametri, si utilizzano principalmente le funzioni anonime e le funzioni normali non vengono utilizzate più spesso.

Examples

Definire una funzione

```
function add(a, b)
    return a + b
end
-- creates a function called add, which returns the sum of it's two arguments
```

Diamo un'occhiata alla sintassi. Innanzitutto, vediamo una parola chiave `function` . Bene, è abbastanza descrittivo. Successivamente vediamo l'identificatore di `add` ; il nome. Vediamo quindi gli argomenti `(a, b)` questi possono essere qualsiasi cosa, e sono locali. Solo all'interno del corpo della funzione possiamo accedervi. Saltiamo fino alla fine, vediamo ... beh, la `end` ! E tutto ciò che c'è in mezzo è il corpo della funzione; il codice che viene eseguito quando viene chiamato. La parola chiave `return` è ciò che rende la funzione effettivamente utile. Senza di esso, la funzione non restituisce nulla, che equivale a restituire nil. Questo può naturalmente essere utile per cose che interagiscono con IO, ad esempio:

```
function printHello(name)
    print("Hello, " .. name .. "!");
end
```

In quella funzione, non abbiamo usato la dichiarazione di ritorno.

Le funzioni possono anche restituire i valori in modo condizionale, ovvero una funzione ha la possibilità di restituire nulla (nil) o un valore. Questo è dimostrato nel seguente esempio.

```
function add(a, b)
    if (a + b <= 100) then
        return a + b -- Returns a value
    else
        print("This function doesn't return values over 100!") -- Returns nil
    end
end
```

È anche possibile che una funzione restituisca più valori separati da virgole, come mostrato:

```
function doOperations(a, b)
    return a+b, a-b, a*b
end

added, subbed, multiplied = doOperations(4,2)
```

Le funzioni possono anche essere dichiarate locali

```
do
    local function add(a, b) return a+b end
    print(add(1,2)) --> prints 3
end
print(add(2, 2)) --> exits with error, because 'add' is not defined here
```

Possono essere salvati anche nelle tabelle:

```
tab = {function(a,b) return a+b end}
(tab[1])(1, 2) --> returns 3
```

Chiamare una funzione.

Le funzioni sono utili solo se possiamo chiamarle. Per chiamare una funzione viene utilizzata la seguente sintassi:

```
print("Hello, World!")
```

Stiamo chiamando la funzione di `print` . Utilizzando l'argomento `"Hello, World"` . Come è ovvio, questo stamperà `Hello, World` sul flusso di output. Il valore restituito è accessibile, proprio come qualsiasi altra variabile.

```
local added = add(10, 50) -- 60
```

Le variabili sono anche accettate nei parametri di una funzione.

```
local a = 10
```

```
local b = 60

local c = add(a, b)

print(c)
```

Le funzioni che prevedono una tabella o una stringa possono essere chiamate con uno zucchero sintattico preciso: le parentesi che circondano la chiamata possono essere omesse.

```
print"Hello, world!"
for k, v in pairs{"Hello, world!"} do print(k, v) end
```

Funzioni anonime

Creazione di funzioni anonime

Le funzioni anonime sono come normali funzioni Lua, tranne che non hanno un nome.

```
doThrice(function()
    print("Hello!")
end)
```

Come puoi vedere, la funzione non è assegnata a nessun nome come `print` o `add` . Per creare una funzione anonima, tutto ciò che devi fare è omettere il nome. Queste funzioni possono anche prendere argomenti.

Capire lo zucchero sintattico

È importante capire che il seguente codice

```
function double(x)
    return x * 2
end
```

è in realtà solo una scorciatoia per

```
double = function(x)
    return x * 2
end
```

Tuttavia, la funzione di cui sopra **non** è anonima in quanto la funzione è direttamente assegnata a una variabile!

Le funzioni sono valori di prima classe

Ciò significa che una funzione è un valore con gli stessi diritti dei valori convenzionali come numeri e stringhe. Le funzioni possono essere memorizzate in variabili, in tabelle, possono essere passate come argomenti e possono essere restituite da altre funzioni.

Per dimostrarlo, creeremo anche una funzione "metà":

```
half = function(x)
    return x / 2
end
```

Quindi, ora abbiamo due variabili, `half` e `double` , entrambe contenenti una funzione come valore. E se volessimo creare una funzione che alimenterebbe il numero 4 in due funzioni date e calcolerebbe la somma di entrambi i risultati?

Vorremmo chiamare questa funzione come `sumOfTwoFunctions(double, half, 4)` . Questo alimenterà la `double` funzione, la `half` funzione e l'intero `4` nella nostra funzione.

```
function sumOfTwoFunctions(firstFunction, secondFunction, input)
    return firstFunction(input) + secondFunction(input)
end
```

La funzione `sumOfTwoFunctions` precedente mostra come le funzioni possono essere passate all'interno degli argomenti e accessibili da un altro nome.

Parametri di default

```
function sayHello(name)
    print("Hello, " .. name .. "!")
end
```

Quella funzione è una funzione semplice e funziona bene. Ma cosa succederebbe se chiamassimo semplicemente `sayHello()` ?

```
stdin:2: attempt to concatenate local 'name' (a nil value)
stack traceback:
    stdin:2: in function 'sayHello'
    stdin:1: in main chunk
    [C]: in ?
```

Non è esattamente fantastico. Ci sono due modi per risolvere questo problema:

1. Si ritorna immediatamente dalla funzione:

   ```
   function sayHello(name)
     if not (type(name) == "string") then
       return nil, "argument #1: expected string, got " .. type(name)
     end -- Bail out if there's no name.
     -- in lua it is a convention to return nil followed by an error message on error

     print("Hello, " .. name .. "!") -- Normal behavior if name exists.
   end
   ```

2. Hai impostato un parametro *predefinito* .

   Per fare ciò, usa semplicemente questa semplice espressione

```
function sayHello(name)
    name = name or "Jack" -- Jack is the default,
                          -- but if the parameter name is given,
                          -- name will be used instead
    print("Hello, " .. name .. "!")
end
```

Il `name = name or "Jack"` idioma `name = name or "Jack"` funziona perché `or` nei cortocircuiti di Lua. Se l'elemento sul lato sinistro di un `or` è diverso da `nil` o `false` , il lato destro non viene mai valutato. D'altra parte, se `sayHello` viene chiamato con nessun parametro, il `name` sarà `nil` , quindi la stringa `"Jack"` verrà assegnata al `name` . (Nota che questo idioma, quindi, non funzionerà se il valore booleano `false` è un valore ragionevole per il parametro in questione.)

Più risultati

Le funzioni in Lua possono restituire più risultati.

Per esempio:

```
function triple(x)
    return x, x, x
end
```

Quando si chiama una funzione, per salvare questi valori, è necessario utilizzare la seguente sintassi:

```
local a, b, c = triple(5)
```

Quale risultato in `a = b = c = 5` in questo caso. È anche possibile ignorare i valori restituiti utilizzando la variabile throwaway _ nella posizione desiderata in un elenco di variabili:

```
local a, _, c = triple(5)
```

In questo caso, il secondo valore restituito verrà ignorato. È anche possibile ignorare i valori restituiti non assegnandoli a nessuna variabile:

```
local a = triple(5)
```

Alla variabile `a` verrà assegnato il primo valore di ritorno e gli altri due verranno scartati.

Quando una funzione restituisce una quantità variabile di risultati, è possibile memorizzarli tutti in una tabella, eseguendo la funzione al suo interno:

```
local results = {triple(5)}
```

In questo modo, si può scorrere sulla tabella dei `results` per vedere cosa ha restituito la funzione.

Nota

Questo può essere una sorpresa in alcuni casi, ad esempio:

```
local t = {}
table.insert(t, string.gsub("  hi", "^%s*(.*)$", "%1")) --> bad argument #2 to 'insert'
(number expected, got string)
```

Ciò accade perché `string.gsub` restituisce 2 valori: la stringa data, con le occorrenze del modello sostituito e il numero totale di corrispondenze verificatesi.

Per risolvere questo, utilizzare una variabile intermedia o put `()` attorno alla chiamata, in questo modo:

```
table.insert(t, (string.gsub("  hi", "^%s*(.*)$", "%1"))) --> works. t = {"hi"}
```

Questo afferra solo il primo risultato della chiamata e ignora il resto.

Numero variabile di argomenti

Argomenti Variadici

Argomenti nominati

```
local function A(name, age, hobby)
    print(name .. "is " .. age .. " years old and likes " .. hobby)
end
A("john", "eating", 23) --> prints 'john is eating years old and likes 23'
-- oops, seems we got the order of the arguments wrong...
-- this happens a lot, specially with long functions that take a lot of arguments
-- and where the order doesn't follow any particular logic

local function B(tab)
    print(tab.name .. "is " .. tab.age .. " years old and likes " .. tab.hobby)
end
local john = {name="john", hobby="golf", age="over 9000", comment="plays too much golf"}
B(john)
--> will print 'John is over 9000 years old and likes golf'
-- I also added a 'comment' argument just to show that excess arguments are ignored by the
function

B({name = "tim"}) -- can also be written as
B{name = "tim"} -- to avoid cluttering the code
--> both will print 'tim is nil years old and likes nil'
-- remember to check for missing arguments and deal with them

function C(tab)
    if not tab.age then return nil, "age not defined" end
    tab.hobby = tab.hobby or "nothing"
    -- print stuff
end

-- note that if we later decide to do a 'person' class
-- we just need to make sure that this class has the three fields
-- age, hobby and name, and it will be compatible with these functions

-- example:
```

```
local john = ClassPerson.new("John", 20, "golf") -- some sort of constructor
john.address = "some place" -- modify the object
john:do_something("information") -- call some function of the object
C(john) -- this works because objects are *usually* implemented as tables
```

Controllo dei tipi di argomenti

Alcune funzioni funzionano solo su un determinato tipo di argomento:

```
function foo(tab)
    return tab.bar
end
--> returns nil if tab has no field bar, which is acceptable
--> returns 'attempt to index a number value' if tab is, for example, 3
--> which is unacceptable

function kungfoo(tab)
    if type(tab) ~= "table" then
        return nil, "take your useless " .. type(tab) .." somewhere else!"
    end

    return tab.bar
end
```

questo ha diverse implicazioni:

```
print(kungfoo(20)) --> prints 'nil, take your useless number somewhere else!'

if kungfoo(20) then print "good" else print "bad" end --> prints bad

foo = kungfoo(20) or "bar" --> sets foo to "bar"
```

ora possiamo chiamare la funzione con qualsiasi parametro che vogliamo, e non manderà in crash il programma.

```
-- if we actually WANT to abort execution on error, we can still do
result = assert(kungfoo({bar=20})) --> this will return 20
result = assert(kungfoo(20)) --> this will throw an error
```

Quindi, cosa succede se abbiamo una funzione che fa qualcosa con un'istanza di una classe specifica? Questo è difficile, perché le classi e gli oggetti sono in genere tabelle, quindi la funzione `type` restituirà `'table'` .

```
local Class = {data="important"}
local meta = {__index=Class}

function Class.new()
    return setmetatable({}, meta)
end
-- this is just a very basic implementation of an object class in lua

object = Class.new()
fake = {}
```

```
print(type(object)), print(type(fake)) --> prints 'table' twice
```

Soluzione: confronta i metabli

```
-- continuation of previous code snippet
Class.is_instance(tab)
    return getmetatable(tab) == meta
end

Class.is_instance(object) --> returns true
Class.is_instance(fake) --> returns false
Class.is_instance(Class) --> returns false
Class.is_instance("a string") --> returns false, doesn't crash the program
Class.is_instance(nil) --> also returns false, doesn't crash either
```

chiusure

```
do
    local tab = {1, 2, 3}
    function closure()
        for key, value in ipairs(tab) do
            print(key, "I can still see you")
        end
    end
    closure()
    --> 1 I can still see you
    --> 2 I can still see you
    --> 3 I can still see you
end

print(tab) --> nil
-- tab is out of scope

closure()
--> 1 I can still see you
--> 2 I can still see you
--> 3 I can still see you
-- the function can still see tab
```

esempio di utilizzo tipico

```
function new_adder(number)
    return function(input)
        return input + number
    end
end
add_3 = new_adder(3)
print(add_3(2)) --> prints 5
```

esempio di utilizzo più avanzato

```
function base64.newDecoder(str) -- Decoder factory
    if #str ~= 64 then return nil, "string must be 64 characters long!" end
```

```
    local tab = {}
    local counter = 0
    for c in str:gmatch"." do
        tab[string.byte(c)] = counter
        counter = counter + 1
    end

    return function(str)
        local result = ""

        for abcd in str:gmatch"..?.?.?" do
            local a, b, c, d = string.byte(abcd,1,-1)
            a, b, c, d = tab[a], tab[b] or 0, tab[c] or 0, tab[d] or 0
            result = result .. (
                string.char( ((a<<2)+(b>>4))%256 ) ..
                string.char( ((b<<4)+(c>>2))%256 ) ..
                string.char( ((c<<6)+d)%256 )
            )
        end
        return result
    end
end
```

Leggi funzioni online: https://riptutorial.com/it/lua/topic/1250/funzioni

Capitolo 6: Garbage collector e tavoli deboli

Sintassi

1. collectgarbage (gcrule [, gcdata]): raccoglie i rifiuti usando gcrule
2. setmetatable (tab, {__mode = weakmode}) - imposta la modalità debole della scheda su weakmode

Parametri

parametro	dettagli
gcrule e gcdata	Azione su gc (garbage collector): `"stop"` (interruzione della raccolta), `"restart"` (ricomincia a raccogliere), `"collect"` o `nil` (raccogli tutti i rifiuti), `"step"` (esegui un passo di raccolta), `"count"` (conteggio di ritorno della memoria utilizzata in KB), `"setpause"` e dati sono numeri da `0` % a `100` % (imposta il parametro di pausa di gc), `"setstepmul"` ei dati sono numeri da `0` % a `100` (imposta `"stepmul"` per gc) .
weakmode	Tipo di tabella debole: `"k"` (solo chiavi deboli), `"v"` (solo valori deboli), `"vk"` (chiavi e valori deboli)

Examples

Tavoli deboli

```
local t1, t2, t3, t4 = {}, {}, {}, {} -- Create 4 tables
local maintab = {t1, t2} -- Regular table, strong references to t1 and t2
local weaktab = setmetatable({t1, t2, t3, t4}, {__mode = 'v'}) -- table with weak references.

t1, t2, t3, t4 = nil, nil, nil, nil -- No more "strong" references to t3 and t4
print(#maintab, #weaktab) --> 2 4

collectgarbage() -- Destroy t3 and t4 and delete weak links to them.
print(#maintab, #weaktab) --> 2 2
```

Leggi Garbage collector e tavoli deboli online: https://riptutorial.com/it/lua/topic/5769/garbage-collector-e-tavoli-deboli

Capitolo 7: Gestione degli errori

Examples

Usando il pcall

`pcall` significa "chiamata protetta". È usato per aggiungere la gestione degli errori alle funzioni. `pcall` funziona come `try-catch` in altre lingue. Il vantaggio di `pcall` è che l'intera esecuzione dello script non viene interrotta se si verificano errori nelle funzioni chiamate con `pcall` . Se si verifica un errore in una funzione chiamata con `pcall` viene generato un errore e il resto del codice continua l'esecuzione.

Sintassi:

```
pcall( f , arg1,···)
```

Valori di ritorno:

Restituisce due valori

1. stato (booleano)

- Restituisce **true** se la funzione è stata eseguita senza errori.
- Restituisce **false** se si è verificato un errore all'interno della funzione.

2. valore di ritorno della funzione **o** messaggio di errore se si è verificato un errore all'interno del blocco funzione.

---

pcall può essere usato per vari casi, tuttavia uno comune è quello di catturare errori dalla funzione che è stata data alla tua funzione. Ad esempio, diciamo che abbiamo questa funzione:

```
local function executeFunction(funcArg, times) then
    for i = 1, times do
        local ran, errorMsg = pcall( funcArg )
        if not ran then
            error("Function errored on run " .. tostring(i) .. "\n" .. errorMsg)
        end
    end
end
```

Quando gli errori di funzione dati durante l'esecuzione 3, il messaggio di errore sarà chiaro all'utente che non proviene dalla funzione, ma dalla funzione che è stata assegnata alla nostra funzione. Inoltre, con questo in mente, è possibile inviare una nota BSoD all'utente. Tuttavia, ciò dipende dall'applicazione che implementa questa funzione, poiché un'API probabilmente non lo farà.

Esempio A - *Esecuzione senza pcall*

```
function square(a)
  return a * "a"    --This will stop the execution of the code and throws an error, because of
the attempt to perform arithmetic on a string value
end

square(10);

print ("Hello World")    -- This is not being executed because the script was interrupted due
to the error
```

Esempio B - *Esecuzione con pcall*

```
function square(a)
  return a * "a"
end

local status, retval = pcall(square,10);

print ("Status: ", status)        -- will print "false" because an error was thrown.
print ("Return Value: ", retval)  -- will print "input:2: attempt to perform arithmetic on a
string value"
print ("Hello World")    -- Prints "Hello World"
```

Esempio - *Esecuzione di codice impeccabile*

```
function square(a)
  return a * a
end

local status, retval = pcall(square,10);

print ("Status: ", status)        -- will print "true" because no errors were thrown
print ("Return Value: ", retval)  -- will print "100"
print ("Hello World")    -- Prints "Hello World"
```

Gestione degli errori in Lua

Supponendo che abbiamo la seguente funzione:

```
function foo(tab)
  return tab.a
end -- Script execution errors out w/ a stacktrace when tab is not a table
```

Miglioriamolo un po '

```
function foo(tab)
  if type(tab) ~= "table" then
    error("Argument 1 is not a table!", 2)
  end
  return tab.a
end -- This gives us more information, but script will still error out
```

Se non vogliamo che una funzione blocchi un programma anche in caso di errore, è normale che lua esegua le seguenti operazioni:

```
function foo(tab)
    if type(tab) ~= "table" then return nil, "Argument 1 is not a table!" end
    return tab.a
end -- This never crashes the program, but simply returns nil and an error message
```

Ora abbiamo una funzione che si comporta in questo modo, possiamo fare cose del genere:

```
if foo(20) then print(foo(20)) end -- prints nothing
result, error = foo(20)
if result then print(result) else log(error) end
```

E se vogliamo che il programma si arresti in modo anomalo se qualcosa va storto, possiamo ancora farlo:

```
result, error = foo(20)
if not result then error(error) end
```

Fortunatamente non abbiamo nemmeno bisogno di scrivere tutto questo ogni volta; lua ha una funzione che fa esattamente questo

```
result = assert(foo(20))
```

Leggi Gestione degli errori online: https://riptutorial.com/it/lua/topic/4561/gestione-degli-errori

Capitolo 8: Imposta

Examples

Cerca un elemento in un elenco

Non esiste un modo integrato per cercare un elenco per un particolare oggetto. Tuttavia, la programmazione in Lua mostra come potresti creare un set che può aiutare:

```
function Set (list)
  local set = {}
  for _, l in ipairs(list) do set[l] = true end
  return set
end
```

Quindi puoi mettere la tua lista nel Set e testare l'iscrizione:

```
local items = Set { "apple", "orange", "pear", "banana" }

if items["orange"] then
  -- do something
end
```

Usare una tabella come un insieme

Crea un set

```
 local set = {} -- empty set
```

Crea un set con elementi impostando il loro valore su `true` :

```
 local set = {pear=true, plum=true}

 -- or initialize by adding the value of a variable:
 local fruit = 'orange'
 local other_set = {[fruit] = true} -- adds 'orange'
```

Aggiungi un membro al set

aggiungi un membro impostando il suo valore su `true`

```
  set.peach = true
  set.apple = true
  -- alternatively
  set['banana'] = true
  set['strawberry'] = true
```

Rimuovi un membro dal set

```
set.apple = nil
```

È preferibile utilizzare `nil` anziché `false` per rimuovere "apple" dalla tabella poiché renderà più semplici gli elementi iterating. `nil` cancella la voce dalla tabella mentre imposta su `false` suo valore.

Test di appartenenza

```
if set.strawberry then
    print "We've got strawberries"
end
```

Scorrere gli elementi in un set

```
for element in pairs(set) do
    print(element)
end
```

Leggi Imposta online: https://riptutorial.com/it/lua/topic/3875/imposta

Capitolo 9: Introduzione all'API Lua C

Sintassi

- lua_State * L = lua_open (); // Crea un nuovo stato VM; Lua 5.0
- lua_State * L = luaL_newstate (); // Crea un nuovo stato VM; Lua 5.1+
- int luaL_dofile (lua_State * L, const char * *nomefile*); // Esegui uno script lua con il *nome file* specificato usando lua_State specificato
- void luaL_openlibs (lua_State * L); // Carica tutte le librerie standard nel lua_State specificato
- void lua_close (lua_State * L); // Chiudi lo stato della VM e rilascia tutte le risorse al suo interno
- void lua_call (lua_State * L, int nargs, int nresults); // Chiama il luavalue all'indice - (nargs + 1)

Osservazioni

Anche Lua fornisce un'API C adeguata alla sua macchina virtuale. Al contrario della VM stessa, l'interfaccia API C è basata sullo stack. Quindi, la maggior parte delle funzioni destinate a essere utilizzate con i dati consiste nell'aggiungere alcune cose in cima allo stack virtuale o rimuoverle. Inoltre, tutte le chiamate API devono essere utilizzate con attenzione all'interno dello stack e le sue limitazioni.

In generale, tutto ciò che è disponibile sulla lingua Lua può essere fatto usando la sua API C. Inoltre, ci sono alcune funzionalità di aggiunta come l'accesso diretto al registro interno, il comportamento delle modifiche di allocatore di memoria standard o garbage collector.

Puoi compilare gli esempi di API Lua C eseguendo il seguente comando sul tuo terminale:

```
$ gcc -Wall ./example.c -llua -ldl -lm
```

Examples

Creazione di Lua Virtual Machine

```
#include <lua.h>
#include <lauxlib.h>
#include <lualib.h>

int main(void)
{
```

5.1

```
  /* Start by creating a new VM state */
  lua_State *L = luaL_newstate();
```

```
  /* Load standard Lua libraries: */
  luaL_openlibs(L);
```

5.1

```
  /* For older version of Lua use lua_open instead */
  lua_State *L = lua_open();

  /* Load standard libraries*/
  luaopen_base(L);
  luaopen_io(L);
  luaopen_math(L);
  luaopen_string(L);
  luaopen_table(L);
```

```
    /* do stuff with Lua VM. In this case just load and execute a file: */
    luaL_dofile(L, "some_input_file.lua");

    /* done? Close it then and exit. */
    lua_close(L);

    return EXIT_SUCCESS;
}
```

Chiamando le funzioni Lua

```
#include <stdlib.h>

#include <lauxlib.h>
#include <lua.h>
#include <lualib.h>

int main(void)
{
    lua_State *lvm_hnd = lua_open();
    luaL_openlibs(lvm_hnd);

    /* Load a standard Lua function from global table: */
    lua_getglobal(lvm_hnd, "print");

    /* Push an argument onto Lua C API stack: */
    lua_pushstring(lvm_hnd, "Hello C API!");

    /* Call Lua function with 1 argument and 0 results: */
    lua_call(lvm_hnd, 1, 0);

    lua_close(lvm_hnd);

    return EXIT_SUCCESS;
 }
```

Nell'esempio sopra stiamo facendo queste cose:

- creando e configurando Lua VM come mostrato nel primo esempio
- ottenere e spingere una funzione Lua dalla tabella Lua globale allo stack virtuale
- spingendo la stringa `"Hello C API"` come argomento di input sullo stack virtuale

- istruire VM per chiamare una funzione con un argomento che è già in pila
- chiusura e pulizia

NOTA:

Ricordatevi che `lua_call()` apre la funzione e gli argomenti dalla pila lasciano solo il risultato.

Inoltre, sarebbe più sicuro usare invece Lua protected call - `lua_pcall()` .

Interprete Lua integrato con API personalizzata e personalizzazione Lua

Dimostrare come incorporare un interprete lua nel codice C, esporre una funzione definita da C allo script Lua, valutare uno script Lua, chiamare una funzione definita da C da Lua e chiamare una funzione definita da Lua da C (l'host).

In questo esempio, vogliamo che lo stato d'animo venga impostato da uno script Lua. Ecco mood.lua:

```
-- Get version information from host
major, minor, build = hostgetversion()
print( "The host version is ", major, minor, build)
print("The Lua interpreter version is ", _VERSION)

-- Define a function for host to call
function mood( b )

    -- return a mood conditional on parameter
    if (b and major > 0) then
        return 'mood-happy'
    elseif (major == 0) then
        return 'mood-confused'
    else
        return 'mood-sad'
    end
end
```

Si noti che `mood()` non viene chiamato nello script. È appena definito per l'applicazione host da chiamare. Si noti inoltre che lo script chiama una funzione chiamata `hostgetversion()` che non è definita nello script.

Successivamente, definiamo un'applicazione host che utilizza 'mood.lua'. Ecco l'hostlua.c:

```
#include <stdio.h>
#include <lua.h>
#include <lualib.h>
#include <lauxlib.h>

/*
 * define a function that returns version information to lua scripts
 */
static int hostgetversion(lua_State *l)
{
    /* Push the return values */
    lua_pushnumber(l, 0);
```

```
    lua_pushnumber(l, 99);
    lua_pushnumber(l, 32);

    /* Return the count of return values */
    return 3;
}

int main (void)
{
    lua_State *l = luaL_newstate();
    luaL_openlibs(l);

    /* register host API for script */
    lua_register(l, "hostgetversion", hostgetversion);

    /* load script */
    luaL_dofile(l, "mood.lua");

    /* call mood() provided by script */
    lua_getglobal(l, "mood");
    lua_pushboolean(l, 1);
    lua_call(l, 1, 1);

    /* print the mood */
    printf("The mood is %s\n", lua_tostring(l, -1));
    lua_pop(l, 1);

    lua_close(l);
    return 0;
}
```

Ed ecco l'output:

```
The host version is     0    99    32
Lua interpreter version is     Lua 5.2
The mood is mood-confused
```

Anche dopo aver compilato 'hostlua.c', siamo ancora liberi di modificare 'mood.lua' per modificare l'output del nostro programma!

Manipolazione della tabella

Per accedere o modificare un indice su un tavolo, è necessario in qualche modo posizionare la tabella nello stack.
Supponiamo, per questo esempio, che la tua tabella sia una variabile globale chiamata tbl.

Ottenere il contenuto in un indice particolare:

```
int getkey_index(lua_State *L)
{
  lua_getglobal(L, "tbl");    // this put the table in the stack
  lua_pushstring(L, "index"); // push the key to access
  lua_gettable(L, -2);        // retrieve the corresponding value; eg. tbl["index"]

  return 1;                   // return value to caller
```

```
}
```

Come abbiamo visto, tutto ciò che devi fare è spingere il tavolo nello stack, premere l'indice e chiamare lua_gettable. l'argomento -2 indica che la tabella è il secondo elemento dalla cima dello stack.
lua_gettable attiva metamethods. Se non vuoi attivare metamethod, usa invece lua_rawget. Usa gli stessi argomenti.

Impostazione del contenuto in un indice particolare:

```
int setkey_index(lua_State *L)
{
  // setup the stack
  lua_getglobal(L, "tbl");
  lua_pushstring(L, "index");
  lua_pushstring(L, "value");
  // finally assign the value to table; eg. tbl.index = "value"
  lua_settable(L, -3);

  return 0;
}
```

Lo stesso trapano come ottenere il contenuto. Devi spingere la pila, spingere l'indice e quindi inserire il valore nella pila. dopodiché chiami lua_settable. l'argomento -3 è la posizione della tabella nello stack. Per evitare di innescare metamethods, usa lua_rawset invece di lua_settable. Usa gli stessi argomenti.

Trasferimento del contenuto da una tabella a un'altra:

```
int copy_tableindex(lua_State *L)
{
    lua_getglobal(L, "tbl1"); // (tbl1)
    lua_getglobal(L, "tbl2");// (tbl1)(tbl2)
    lua_pushstring(L, "index1");// (tbl1)(tbl2)("index1")
    lua_gettable(L, -3);// (tbl1)(tbl2)(tbl1.index1)
    lua_pushstring(L, "index2");// (tbl1)(tbl2)(tbl1.index1)("index2")
    lua_pushvalue(L, -2); // (tbl1)(tbl2)(tbl1.index1)("index2")(tbl1.index1)
    lua_settable(L, -4);// (tbl1)(tbl2)(tbl1.index1)
    lua_pop(L, 1);

    return 0;
}
```

Ora stiamo mettendo insieme tutto ciò che abbiamo imparato qui. Inserisco il contenuto dello stack nei commenti in modo da non perderlo.

`tbl1.index1` entrambe le tabelle nello stack, inseriamo l'indice della tabella 1 nello stack e otteniamo il valore su `tbl1.index1` . Notare l'argomento `-3` su gettable. Sto guardando il primo tavolo (terzo dall'alto) e non il secondo. Quindi inseriamo l'indice della seconda tabella, copiamo il file `tbl1.index1` in cima allo stack e quindi richiamiamo `lua_settable` , sul quarto elemento dall'alto.

Per l'amor di pulizia, ho eliminato l'elemento superiore, quindi solo i due tavoli rimangono in pila.

Leggi Introduzione all'API Lua C online: https://riptutorial.com/it/lua/topic/671/introduzione-all-api-lua-c

Capitolo 10: iteratori

Examples

Ciclo generico

Gli iteratori utilizzano una forma del ciclo `for` noto come loop generico .

La forma generica del ciclo `for` utilizza tre parametri:

1. Una **funzione iteratrice** che viene chiamata quando è necessario il valore successivo. Riceve sia lo stato invariante che la variabile di controllo come parametri. Restituzione della terminazione dei segnali `nil` .
2. Lo **stato invariante** è un valore che non cambia durante l'iterazione. In genere è l'oggetto dell'iteratore, ad esempio una tabella, una stringa o un dato utente.
3. La **variabile control** rappresenta un valore iniziale per l'iterazione.

Possiamo scrivere un ciclo `for` per iterare tutte le coppie chiave-valore in una tabella usando la funzione successiva .

```
local t = {a=1, b=2, c=3, d=4, e=5}

-- next is the iterator function
-- t is the invariant state
-- nil is the control variable (calling next with a nil gets the first key)
for key, value in next, t, nil do
  -- key is the new value for the control variable
  print(key, value)
  -- Lua calls: next(t, key)
end
```

Iteratori standard

La libreria standard Lua fornisce due funzioni di iteratore che possono essere utilizzate con un ciclo `for` per attraversare coppie di valori-chiave all'interno di tabelle.

Per scorrere su una tabella di sequenze possiamo usare la funzione di libreria ipairs .

```
for index, value in ipairs {'a', 'b', 'c', 'd', 'e'} do
  print(index, value)  --> 1 a, 2 b, 3 c, 4 d, 5 e
end
```

Per iteratore su tutte le chiavi e i valori in qualsiasi tabella possiamo usare le coppie di funzioni della libreria.

```
for key, value in pairs {a=1, b=2, c=3, d=4, e=5} do
  print(key, value)  --> e 5, c 3, a 1, b 2, d 4  (order not specified)
end
```

Iteratori apolidi

Sia le coppie che gli ipers rappresentano gli iteratori senza stato. Un iteratore senza stato utilizza solo il generico per la variabile di controllo del ciclo e lo stato invariante per calcolare il valore di iterazione.

Coppie Iterator

Possiamo implementare l'iteratore di `pairs` senza stato utilizzando la funzione `next` .

```
-- generator function which initializes the generic for loop
local function pairs(t)
  -- next is the iterator function
  -- t is the invariant state
  -- control variable is nil
  return next, t, nil
end
```

Ipatori Iterator

Siamo in grado di implementare l'iteratore stateless `ipairs` in due funzioni separate.

```
-- function which performs the actual iteration
local function ipairs_iter(t, i)
  local i = i + 1  -- next index in the sequence (i is the control variable)
  local v = t[i]   -- next value (t is the invariant state)
  if v ~= nil then
    return i, v    -- index, value
  end
  return nil       -- no more values (termination)
end

-- generator function which initializes the generic for loop
local function ipairs(t)
  -- ipairs_iter is the iterator function
  -- t is the invariant state (table to be iterated)
  -- 0 is the control variable (first index)
  return ipairs_iter, t, 0
end
```

Iteratore di caratteri

Siamo in grado di creare nuovi iteratori apolidi adempiendo il contratto del generico `for` ciclo.

```
-- function which performs the actual iteration
local function chars_iter(s, i)
  if i < #s then
    i = i + 1
    return i, s:sub(i, i)
  end
end
```

```
-- generator function which initializes the generic for loop
local function chars(s)
  return chars_iter, s, 0
end

-- used like pairs and ipairs
for i, c in chars 'abcde' do
    print(i, c) --> 1 a, 2 b, 3 c, 4 f, 5 e
end
```

Prime Number Iterator

Questo è un esempio più semplice di un iteratore senza stato.

```
-- prime numbers iterator
local incr = {4, 1, 2, 0, 2}
function primes(s, p, d)
   s, p, d = s or math.huge, p and p + incr[p % 6] or 2, 1
   while p <= s do
      repeat
         d = d + incr[d % 6]
         if d*d > p then return p end
      until p % d == 0
      p, d = p + incr[p % 6], 1
   end
end

-- print all prime numbers <= 100
for p in primes, 100 do  -- passing in the iterator (do not call the iterator here)
   print(p)  -->  2  3  5  7  11 ... 97
end

-- print all primes in endless loop
for p in primes do  -- please note: "in primes", not "in primes()"
   print(p)
end
```

Iteratori di stato

Gli iteratori di stato contengono alcune informazioni aggiuntive sullo stato corrente dell'iteratore.

Utilizzo delle tabelle

Lo stato di aggiunta può essere impacchettato nello stato invariante del ciclo generico .

```
  local function chars_iter(t, i)
    local i = i + 1
    if i <= t.len then
      return i, t.s:sub(i, i)
    end
  end

  local function chars(s)
    -- the iterators state
```

```
  local t = {
    s = s,    -- the subject
    len = #s  -- cached length
  }
  return chars_iter, t, 0
end

for i, c in chars 'abcde' do
  print(i, c) --> 1 a, 2 b, 3 c, 4 d, 5 e
end
```

Utilizzando le chiusure

Lo stato aggiuntivo può essere avvolto all'interno di una chiusura di funzione. Poiché lo stato è completamente contenuto nello scopo della chiusura, lo stato invariante e la variabile di controllo non sono necessari.

```
local function chars(s)
  local i, len = 0, #s
  return function() -- iterator function
    i = i + 1
    if i <= len then
      return i, s:sub(i, i)
    end
  end
end

for i, c in chars 'abcde' do
  print(i, c) --> 1 a, 2 b, 3 c, 4 d, 5 e
end
```

Usando le coroutine

Uno stato addizionale può essere contenuto all'interno di una coroutine, di nuovo lo stato invariante e la variabile di controllo non sono necessari.

```
local function chars(s)
  return coroutine.wrap(function()
    for i = 1, #s do
      coroutine.yield(s:sub(i, i))
    end
  end)
end

for c in chars 'abcde' do
  print(c) --> a, b, c, d, e
end
```

Leggi iteratori online: https://riptutorial.com/it/lua/topic/4165/iteratori

Capitolo 11: Metatables

Sintassi

- [[local] *mt* =] getmetatable (*t*) -> recupera la metatable associata per ' *t* '
- [[local] *t* =] setmetatable (*t* , *mt*) -> imposta il metatable per ' *t* ' su ' *mt* ' e restituisce ' *t* '

Parametri

Parametro	Dettagli
t	Variabile che si riferisce a un tavolo lua; può anche essere un letterale da tavolo.
mt	Tabella da utilizzare come metatable; può avere zero o più campi metamethod impostati.

Osservazioni

Ci sono alcuni metamethods non menzionati qui. Per l'elenco completo e il loro utilizzo, consultare la voce corrispondente nel manuale di lua .

Examples

Creazione e utilizzo di metabli

Un metatable definisce un insieme di operazioni che alterano il comportamento di un oggetto lua. Una metatable è solo una tabella ordinaria, che viene utilizzata in un modo speciale.

```
local meta = { } -- create a table for use as metatable

-- a metatable can change the behaviour of many things
-- here we modify the 'tostring' operation:
-- this fields should be a function with one argument.
-- it gets called with the respective object and should return a string
meta.__tostring = function (object)
    return string.format("{ %d, %d }", object.x, object.y)
end

-- create an object
local point = { x = 13, y = -2 }
-- set the metatable
setmetatable(point, meta)

-- since 'print' calls 'tostring', we can use it directly:
print(point) -- prints '{ 13, -2 }'
```

Utilizzo di tabelle come metamethods

Alcuni metametodi non devono essere funzioni. Per esempio più importante per questo è il `__index` __index. Può anche essere una tabella, che viene quindi utilizzata come ricerca. Questo è abbastanza comunemente usato nella creazione di classi in lua. Qui, una tabella (spesso la stessa metatable) viene utilizzata per contenere tutte le operazioni (metodi) della classe:

```
local meta = {}
-- set the __index method to the metatable.
-- Note that this can't be done in the constructor!
meta.__index = meta

function create_new(name)
    local self = { name = name }
    setmetatable(self, meta)
    return self
end

-- define a print function, which is stored in the metatable
function meta.print(self)
    print(self.name)
end

local obj = create_new("Hello from object")
obj:print()
```

Garbage collector: metamethod __gc

5.2

Gli oggetti in lua sono raccolti. A volte, è necessario liberare alcune risorse, stampare un messaggio o fare qualcos'altro quando un oggetto viene distrutto (raccolto). Per questo, è possibile utilizzare il metametodo `__gc` , che viene chiamato con l'oggetto come argomento quando l'oggetto viene distrutto. Potresti vedere questo metametodo come una sorta di distruttore.

Questo esempio mostra il metametodo `__gc` in azione. Quando il tavolo interno assegnato a `t` ottiene la garbage collection, stampa un messaggio prima di essere raccolto. Allo stesso modo per il tavolo esterno quando si raggiunge la fine della sceneggiatura:

```
local meta =
{
    __gc = function(self)
        print("destroying self: " .. self.name)
    end
}

local t = setmetatable({ name = "outer" }, meta)
do
    local t = { name = "inner" }
    setmetatable(t, meta)
end
```

Più metamethods

Ci sono molti più metametodi, alcuni di essi sono aritmetici (ad esempio addizione, sottrazione, moltiplicazione), ci sono operazioni bit a bit (e, o, xor, shift), confronto (<,>) e anche operazioni di tipo base come == e # (uguaglianza e lunghezza). Costruiamo una classe che supporta molte di queste operazioni: una chiamata per l'aritmetica razionale. Mentre questo è molto semplice, mostra l'idea.

```
local meta = {
    -- string representation
    __tostring = function(self)
        return string.format("%s/%s", self.num, self.den)
    end,
    -- addition of two rationals
    __add = function(self, rhs)
        local num = self.num * rhs.den + rhs.num * self.den
        local den = self.den * rhs.den
        return new_rational(num, den)
    end,
    -- equality
    __eq = function(self, rhs)
        return self.num == rhs.num and self.den == rhs.den
    end
}

-- a function for the creation of new rationals
function new_rational(num, den)
    local self = { num = num, den = den }
    setmetatable(self, meta)

    return self
end

local r1 = new_rational(1, 2)
print(r1) -- 1/2

local r2 = new_rational(1, 3)
print(r1 + r2) -- 5/6

local r3 = new_rational(1, 2)
print(r1 == r3) -- true
-- this would be the behaviour if we hadn't implemented the __eq metamethod.
-- this compares the actual tables, which are different
print(rawequal(r1, r3)) -- false
```

Rendi i tavoli chiamabili

Esiste un metametodo chiamato `__call` , che definisce il bevahiour dell'oggetto dopo essere stato usato come funzione, ad esempio `object()` . Questo può essere usato per creare oggetti funzione:

```
-- create the metatable with a __call metamethod
local meta = {
    __call = function(self)
        self.i = self.i + 1
    end,
    -- to view the results
    __tostring = function(self)
        return tostring(self.i)
    end
```

```
}

function new_counter(start)
    local self = { i = start }
    setmetatable(self, meta)
    return self
end

-- create a counter
local c = new_counter(1)
print(c) --> 1
-- call -> count up
c()
print(c) --> 2
```

Il metodo metamethod viene chiamato con l'oggetto corrispondente, tutti gli argomenti rimanenti vengono passati alla funzione dopo che:

```
local meta = {
    __call = function(self, ...)
        print(self.prepend, ...)
    end
}

local self = { prepend = "printer:" }
setmetatable(self, meta)

self("foo", "bar", "baz")
```

Indicizzazione delle tabelle

Forse l'uso più importante dei metatables è la possibilità di modificare l'indicizzazione delle tabelle. Per questo, ci sono due azioni da considerare: *leggere* il contenuto e *scrivere* il contenuto della tabella. Si noti che entrambe le azioni vengono attivate solo se la chiave corrispondente non è presente nella tabella.

Lettura

```
local meta = {}

-- to change the reading action, we need to set the '__index' method
-- it gets called with the corresponding table and the used key
-- this means that table[key] translates into meta.__index(table, key)
meta.__index = function(object, index)
    -- print a warning and return a dummy object
    print(string.format("the key '%s' is not present in object '%s'", index, object))
    return -1
end

-- create a testobject
local t = {}

-- set the metatable
setmetatable(t, meta)
```

```
print(t["foo"]) -- read a non-existent key, prints the message and returns -1
```

Questo potrebbe essere usato per generare un errore durante la lettura di una chiave inesistente:

```
-- raise an error upon reading a non-existent key
meta.__index = function(object, index)
    error(string.format("the key '%s' is not present in object '%s'", index, object))
end
```

scrittura

```
local meta = {}

-- to change the writing action, we need to set the '__newindex' method
-- it gets called with the corresponding table, the used key and the value
-- this means that table[key] = value translates into meta.__newindex(table, key, value)
meta.__newindex = function(object, index, value)
    print(string.format("writing the value '%s' to the object '%s' at the key '%s'",
                        value, object, index))
    --object[index] = value -- we can't do this, see below
end

-- create a testobject
local t = { }

-- set the metatable
setmetatable(t, meta)

-- write a key (this triggers the method)
t.foo = 42
```

Ora puoi chiederti come è scritto il valore reale nella tabella. In questo caso, non lo è. Il problema qui è che i metametodi possono innescare metamethods, il che risulterebbe in un loop infinito o, più precisamente, in un overflow dello stack. Quindi, come possiamo risolvere questo? La soluzione per questo è chiamata *accesso a tabelle non elaborate* .

Accesso alla tabella grezza

A volte, non si desidera attivare metamethods, ma in realtà scrivere o leggere esattamente la chiave data, senza alcune funzioni intelligenti avvolte attorno all'accesso. Per questo, lua ti fornisce i metodi di accesso alle tabelle raw:

```
-- first, set up a metatable that allows no read/write access
local meta = {
    __index = function(object, index)
        -- raise an error
        error(string.format("the key '%s' is not present in object '%s'", index, object))
    end,
    __newindex = function(object, index, value)
        -- raise an error, this prevents any write access to the table
        error(string.format("you are not allowed to write the object '%s'", object))
    end
}
```

```
local t = { foo = "bar" }
setmetatable(t, meta)

-- both lines raise an error:
--print(t[1])
--t[1] = 42

-- we can now circumvent this problem by using raw access:
print(rawget(t, 1)) -- prints nil
rawset(t, 1, 42) -- ok

-- since the key 1 is now valid, we can use it in a normal manner:
print(t[1])
```

Con questo, possiamo ora riscrivere il vecchio metodo `__newindex` per scrivere effettivamente il valore nella tabella:

```
meta.__newindex = function(object, index, value)
    print(string.format("writing the value '%s' to the object '%s' at the key '%s'",
                        value, object, index))
    rawset(object, index, value)
end
```

Simulazione OOP

```
local Class = {} -- objects and classes will be tables
local __meta = {__index = Class}
-- ^ if an instance doesn't have a field, try indexing the class
function Class.new()
    -- return setmetatable({}, __meta) -- this is shorter and equivalent to:
    local new_instance = {}
    setmetatable(new_instance, __meta)
    return new_instance
end
function Class.print()
    print "I am an instance of 'class'"
end

local object = Class.new()
object.print() --> will print "I am an instance of 'class'"
```

I metodi di istanza possono essere scritti passando l'oggetto come primo argomento.

```
-- append to the above example
function Class.sayhello(self)
    print("hello, I am ", self)
end
object.sayhello(object) --> will print "hello, I am <table ID>"
object.sayhello() --> will print "hello, I am nil"
```

C'è dello zucchero sintattico per questo.

```
function Class:saybye(phrase)
    print("I am " .. self .. "\n" .. phrase)
```

```
end
object:saybye("c ya") --> will print "I am <table ID>
                      -->              c ya"
```

Possiamo anche aggiungere campi predefiniti a una classe.

```
local Class = {health = 100}
local __meta = {__index = Class}

function Class.new() return setmetatable({}, __meta) end
local object = Class.new()
print(object.health) --> prints 100
Class.health = 50; print(object.health) --> prints 50
-- this should not be done, but it illustrates lua indexes "Class"
-- when "object" doesn't have a certain field
object.health = 200 -- This does NOT index Class
print(object.health) --> prints 200
```

Leggi Metatables online: https://riptutorial.com/it/lua/topic/2444/metatables

Capitolo 12: Object-Orientamento

introduzione

Lua stessa non offre un sistema di classe. È tuttavia possibile implementare classi e oggetti come tabelle con pochi trucchi.

Sintassi

- `function <class>.new() return setmetatable({}, {__index=<class>}) end`

Examples

Orientamento semplice degli oggetti

Ecco un esempio di base su come fare un sistema di classi molto semplice

```
Class = {}
local __instance = {__index=Class} -- Metatable for instances
function Class.new()
    local instance = {}
    setmetatable(instance, __instance)
    return instance
-- equivalent to: return setmetatable({}, __instance)
end
```

Per aggiungere variabili e / o metodi, aggiungili alla classe. Entrambi possono essere sovrascritti per ogni istanza.

```
Class.x = 0
Class.y = 0
Class:getPosition()
    return {self.x, self.y}
end
```

E per creare un'istanza della classe:

```
object = Class.new()
```

o

```
setmetatable(Class, {__call = Class.new}
    -- Allow the class itself to be called like a function
object = Class()
```

E per usarlo:

```
object.x = 20
-- This adds the variable x to the object without changing the x of
-- the class or any other instance. Now that the object has an x, it
-- will override the x that is inherited from the class
print(object.x)
-- This prints 20 as one would expect.
print(object.y)
-- Object has no member y, therefore the metatable redirects to the
-- class table, which has y=0; therefore this prints 0
object:getPosition() -- returns {20, 0}
```

Modifica dei metametodi di un oggetto

avere

```
local Class = {}
Class.__meta = {__index=Class}
function Class.new() return setmetatable({}, Class.__meta)
```

Supponendo di voler cambiare il comportamento di un `object = Class.new()` istanza singola `object = Class.new()` utilizzando una metatable,

ci sono alcuni errori da evitare:

```
setmetatable(object, {__call = table.concat}) -- WRONG
```

Ciò scambia il vecchio metatable con quello nuovo, rompendo quindi l'ereditarietà della classe

```
getmetatable(object).__call = table.concat -- WRONG AGAIN
```

Tieni presente che i "valori" della tabella sono solo di riferimento; c'è, infatti, solo una tabella reale per tutte le istanze di un oggetto a meno che il costruttore non sia definito come in [1] , così facendo si modifica il comportamento di *tutte le* istanze della classe.

Un modo corretto per farlo:

Senza cambiare la classe:

```
setmetatable(
    object,
    setmetatable(
        {__call=table.concat},
        {__index=getmetatable(object)}
    )
)
```

Come funziona? - Creiamo una nuova metatable come nell'errore numero 1, ma invece di lasciarla vuota, creiamo una copia soft per il metatable originale. Si potrebbe dire che la nuova metatabola "eredita" da quella originale come se fosse un'istanza di classe stessa. Ora possiamo sovrascrivere i valori del metatable originale senza modificarli.

Cambiando la classe:

1o (consigliato):

```
local __instance_meta = {__index = Class.__meta}
-- metatable for the metatable
-- As you can see, lua can get very meta very fast
function Class.new()
    return setmetatable({}, setmetatable({}, __instance_meta))
end
```

2 ° (meno consigliato): vedi [1]

---

[1] `function Class.new() return setmetatable({}, {__index=Class}) end`

Leggi Object-Orientamento online: https://riptutorial.com/it/lua/topic/8908/object-orientamento

Capitolo 13: Pattern matching

Sintassi

- string.find (str, pattern [, init [, plain]]) - Restituisce l'indice di inizio e di fine della corrispondenza in str
- string.match (str, pattern [, index]) - Corrisponde a un pattern una volta (iniziando dall'indice)
- string.gmatch (str, pattern) - Restituisce una funzione che scorre attraverso tutte le corrispondenze in str
- string.gsub (str, pattern, repl [, n]) - Sostituisce sottostringhe (fino ad un massimo di n volte)
- `.` rappresenta tutti i personaggi
- `%a` rappresenta tutte le lettere
- `%l` rappresenta tutte le lettere minuscole
- `%u` rappresenta tutte le lettere maiuscole
- `%d` rappresenta tutte le cifre
- `%x` rappresenta tutte le cifre esadecimali
- `%s` rappresenta tutti i caratteri degli spazi bianchi
- `%p` rappresenta tutti i caratteri di punteggiatura
- `%g` rappresenta tutti i caratteri *stampabili* tranne lo spazio
- `%c` rappresenta tutti i caratteri di controllo
- `[set]` rappresenta la classe che è l'unione di tutti i caratteri nel set.
- `[^set]` rappresenta il complemento del set
- `*` partita avida 0 o più occorrenze della precedente classe di caratteri
- `+` partita avida 1 o più occorrenze della precedente classe di personaggi
- `-` Lazy match 0 o più occorrenze della precedente classe di caratteri
- `?` corrisponde esattamente a 0 o 1 occorrenza della precedente classe di caratteri

Osservazioni

In alcuni esempi, viene usata la notazione `(<string literal>):function <string literal>` , che è

equivalente a `string.function(<string literal>, <string literal>)` perché tutte le stringhe hanno una metatable con il `__index` campi `__index` alla tabella delle `string` .

Examples

Abbinamento modello Lua

Invece di usare espressioni regolari, la libreria di stringhe Lua ha un set speciale di caratteri usati nelle corrispondenze di sintassi. Entrambi possono essere molto simili, ma la corrispondenza del modello Lua è più limitata e ha una sintassi diversa. Ad esempio, la sequenza di caratteri `%a` corrisponde a qualsiasi lettera, mentre la sua versione maiuscola rappresenta *tutti i caratteri non lettere* , tutte le classi di caratteri (una sequenza di caratteri che, come modello, può corrispondere a un insieme di elementi) sono elencate di seguito.

Classe di carattere	Sezione corrispondente
%un	lettere (AZ, az)
% c	caratteri di controllo (\ n, \ t, \ r, ...)
% d	cifre (0-9)
l%	lettera minuscola (az)
% p	caratteri di punteggiatura (!,?, &, ...)
%S	personaggi spaziali
% u	lettere maiuscole
% w	caratteri alfanumerici (AZ, az, 0-9)
%X	cifre esadecimali (\ 3, \ 4, ...)
% z	il personaggio con rappresentazione 0
.	Corrisponde a qualsiasi personaggio

Come accennato in precedenza, qualsiasi versione maiuscola di quelle classi rappresenta il complemento della classe. Ad esempio, `%D` corrisponderà a qualsiasi sequenza di caratteri non a cifre:

```
string.match("f123", "%D")          --> f
```

Oltre alle classi di caratteri, alcuni personaggi hanno funzioni speciali come modelli:

```
( ) % . + - * [ ? ^ $
```

Il carattere `%` rappresenta un carattere di escape, facendo `%?` corrisponde a un interrogatorio e `%%` corrisponde al simbolo percentuale. È possibile utilizzare il carattere `%` con qualsiasi altro carattere non alfanumerico, pertanto, se è necessario sfuggire, ad esempio, un preventivo, è necessario utilizzare `\\` before it, che sfugge a qualsiasi carattere da una stringa lua.

Un set di caratteri, rappresentato all'interno di parentesi quadre (`[]`), ti consente di creare una speciale classe di caratteri, combinando diverse classi e singoli caratteri:

```
local foo = "bar123bar2341"
print(foo:match "[arb]")            --> b
```

Puoi ottenere il complemento del set di caratteri avviandolo con `^` :

```
local foo = "bar123bar2341"
print(string.match(foo, "[^bar]"))  --> 1
```

In questo esempio, `string.match` troverà la prima volta in cui non è **B, una** o **r.**

I pattern possono essere più utili con l'aiuto della ripetizione / modificatori opzionali, i pattern in lua offrono questi quattro caratteri:

Personaggio	Modificatore
+	Una o più ripetizioni
*	Zero o più ripetizioni
-	Anche zero o più ripetizioni
?	Opzionale (zero o una ricorrenza)

Il carattere `+` rappresenta uno o più caratteri corrispondenti nella sequenza e restituirà sempre la sequenza abbinata più lunga:

```
local foo = "12345678bar123"
print(foo:match "%d+")  --> 12345678
```

Come puoi vedere, `*` è simile a `+` , ma accetta zero occorrenze di caratteri ed è comunemente usato per abbinare spazi opzionali tra diversi modelli.

Il carattere `-` è simile a `*` , ma invece di restituire la sequenza abbinata più lunga, corrisponde a quella più corta.

Il modificatore `?` corrisponde a un carattere opzionale, consentendo di associare, ad esempio, una cifra negativa:

```
local foo = "-20"
print(foo:match "[+-]?%d+")
```

Il motore di corrispondenza del modello Lua fornisce alcuni elementi di corrispondenza del modello aggiuntivi:

Oggetto personaggio	Descrizione
`%n`	per n tra 1 e 9 corrisponde una sottostringa uguale alla n-esima stringa catturata
`%bxy`	confronta la sottostringa tra due caratteri distinti (coppia bilanciata di `x y`)
`%f[set]`	motivo di frontiera: corrisponde a una stringa vuota in qualsiasi posizione in modo tale che il carattere successivo appartiene all'insieme e il personaggio precedente non appartiene all'insieme

string.find (Introduzione)

La funzione di `find`

Per prima cosa diamo un'occhiata alla funzione `string.find` in generale:

La funzione `string.find (s, substr [, init [, plain]])` restituisce l'indice iniziale e finale di una sottostringa, se trovata, e nil altrimenti, a partire dall'indice `init` se viene fornito (il valore predefinito è 1).

```
("Hello, I am a string"):find "am" --> returns 10 11
-- equivalent to string.find("Hello, I am a string", "am") -- see remarks
```

Presentazione dei modelli

```
("hello world"):find ".- " -- will match characters until it finds a space
    --> so it will return 1, 6
```

Tutti **tranne** i seguenti caratteri si rappresentano `^$()%.[]*+-?)` . Ognuno di questi personaggi può essere rappresentato da una `%` segue il carattere stesso.

```
("137'5 m47ch s0m3 d1g175"):find "m%d%d" -- will match an m followed by 2 digit
    --> this will match m47 and return 7, 9

("stack overflow"):find "[abc]" -- will match an 'a', a 'b' or a 'c'
    --> this will return 3 (the A in stAck)

("stack overflow"):find "[^stack ]"
    -- will match all EXCEPT the letters s, t, a, c and k and the space character
```

```
    --> this will match the o in overflow

("hello"):find "o%d?" --> matches o, returns 5, 5
("hello20"):find "o%d?" --> matches o2, returns 5, 6
    -- the ? means the character is optional

("hellllllo"):find "el+" --> will match llllll
("heo"):find "el+" --> won't match anything

("hellllllo"):find "el*" --> will match llllll
("heo"):find "el*" --> will match e

("helelo"):find "h.+l" -- + will match as much as it gets
    --> this matches "helel"
("helelo"):find "h.-l" -- - will match as few as it can
    --> this wil only match "hel"

("hello"):match "o%d*"
    --> like ?, this matches the "o", because %d is optional
("hello20"):match "o%d*"
    --> unlike ?, it maches as many %d as it gets, "o20"
("hello"):match "o%d"
    --> wouldn't find anything, because + looks for 1 or more characters
```

La funzione `gmatch`

Come funziona

La funzione `string.gmatch` richiede una stringa di input e un pattern. Questo schema descrive su cosa effettivamente tornare indietro. Questa funzione restituirà una funzione che in realtà è un iteratore. Il risultato di questo iteratore corrisponderà al modello.

```
type(("abc"):gmatch ".") --> returns "function"

for char in ("abc"):gmatch "." do
    print char -- this prints:
    --> a
    --> b
    --> c
end

for match in ("#afdde6"):gmatch "%x%x" do
    print("#" .. match) -- prints:
    --> #af
    --> #dd
    --> #e6
end
```

Presentazione di catture:

Questo è molto simile alla funzione normale, tuttavia restituirà solo le acquisizioni invece della corrispondenza completa.

```
for key, value in ("foo = bar, bar=foo"):gmatch "(%w+)%s*=%s*(%w+)" do
    print("key: " .. key .. ", value: " .. value)
    --> key: foo, value: bar
    --> key: bar, value: foo
end
```

La funzione di gsub

non confondere con la funzione string.sub, che restituisce una sottostringa!

Come funziona

argomento stringa

```
("hello world"):gsub("o", "0")
    --> returns "hell0 w0rld", 2
    -- the 2 means that 2 substrings have been replaced (the 2 Os)

("hello world, how are you?"):gsub("[^%s]+", "word")
    --> returns "word word, word word word?", 5

("hello world"):gsub("([^%s])([^%s]*)", "%2%1")
    --> returns "elloh orldw", 2
```

argomento della funzione

```
local word = "[^%s]+"

function func(str)
    if str:sub(1,1):lower()=="h" then
        return str
    else
        return "no_h"
    end
end
("hello world"):gsub(word, func)
    --> returns "hello no_h", 2
```

argomento della tabella

```
local word = "[^%s]+"

sub = {}
sub["hello"] = "g'day"
sub["world"] = "m8"

("hello world"):gsub(word, sub)
    --> returns "g'day m8"

("hello world, how are you?"):gsub(word, sub)
```

```
--> returns "g'day m8, how are you?"
-- words that are not in the table are simply ignored
```

Leggi Pattern matching online: https://riptutorial.com/it/lua/topic/5829/pattern-matching

Capitolo 14: PICO-8

introduzione

PICO-8 è una console di fantasia programmata in Lua incorporato. Ha già una buona documentazione . Utilizzare questo argomento per dimostrare funzionalità non documentate o sotto-documentate.

Examples

Ciclo di gioco

È possibile utilizzare PICO-8 come shell interattiva , ma probabilmente vorrai attingere al ciclo di gioco. Per fare ciò, è necessario creare almeno una di queste funzioni di callback:

- `_update()`
- `_update60()` (dopo v0.1.8)
- `_draw()`

Un "gioco" minimo potrebbe semplicemente disegnare qualcosa sullo schermo:

```
function _draw()
  cls()
  print("a winner is you")
end
```

Se si definisce `_update60()` , il ciclo di gioco tenta di eseguire a 60fps e ignora `update()` (che viene eseguito a 30fps). Una funzione di aggiornamento viene chiamata prima di `_draw()` . Se il sistema rileva fotogrammi persi, salterà la funzione di disegno ogni altro fotogramma, quindi è meglio mantenere la logica di gioco e l'input del giocatore nella funzione di aggiornamento:

```
function _init()
  x = 63
  y = 63

  cls()
end

function _update()
  local dx = 0 dy = 0

  if (btn(0)) dx-=1
  if (btn(1)) dx+=1
  if (btn(2)) dy-=1
  if (btn(3)) dy+=1

  x+=dx
  y+=dy
  x%=128
  y%=128
```

```
end

function _draw()
  pset(x,y)
end
```

La funzione `_init()` è, in senso stretto, opzionale in quanto i comandi esterni a qualsiasi funzione vengono eseguiti all'avvio. Ma è un modo pratico per ripristinare il gioco alle condizioni iniziali senza riavviare la cartuccia:

```
if (btn(4)) _init()
```

Input del mouse

Sebbene non sia ufficialmente supportato, puoi utilizzare l' input del mouse nei tuoi giochi:

```
function _update60()
  x = stat(32)
  y = stat(33)

  if (x>0 and x<=128 and
      y>0 and y<=128)
  then

    -- left button
    if (band(stat(34),1)==1) then
      ball_x=x
      ball_y=y
    end
  end

  -- right button
  if (band(stat(34),2)==2) then
    ball_c+=1
    ball_c%=16
  end

  -- middle button
  if (band(stat(34),4)==4) then
    ball_r+=1
    ball_r%=64
  end
end

function _init()
  ball_x=63
  ball_y=63
  ball_c=10
  ball_r=1
end

function _draw()
  cls()
  print(stat(34),1,1)
  circ(ball_x,ball_y,ball_r,ball_c)
  pset(x,y,7) -- white
end
```

Modalità di gioco

Se si desidera una schermata del titolo o una schermata di fine partita, si consideri l'impostazione di un meccanismo di cambio modalità:

```
function _init()
  mode = 1
end

function _update()
  if (mode == 1) then
    if (btnp(5)) mode = 2
  elseif (mode == 2) then
    if (btnp(5)) mode = 3
  end
end

function _draw()
  cls()
  if (mode == 1) then
    title()
  elseif (mode == 2) then
    print("press 'x' to win")
  else
    end_screen()
  end
end

function title()
  print("press 'x' to start game")
end

function end_screen()
  print("a winner is you")
end
```

Leggi PICO-8 online: https://riptutorial.com/it/lua/topic/8715/pico-8

Capitolo 15: Scrivere e usare i moduli

Osservazioni

Lo schema di base per scrivere un modulo è riempire una tabella con chiavi che sono nomi di funzioni e valori che sono le stesse funzioni. Il modulo restituisce quindi questa funzione per chiamare il codice da `require` e utilizzare. (Le funzioni sono valori di prima classe in Lua, quindi l'archiviazione di una funzione in una tabella è semplice e comune.) La tabella può contenere anche costanti importanti sotto forma di, per esempio, stringhe o numeri.

Examples

Scrivere il modulo

```
--- trim: a string-trimming module for Lua
-- Author, date, perhaps a nice license too
--
-- The code here is taken or adapted from  material in
-- Programming in Lua, 3rd ed., Roberto Ierusalimschy

-- trim_all(string) => return string with white space trimmed on both sides
local trim_all = function (s)
  return (string.gsub(s, "^%s*(.-)%s*$", "%1"))
end

-- trim_left(string) => return string with white space trimmed on left side only
local trim_left = function (s)
  return (string.gsub(s, "^%s*(.*)$", "%1"))
end

-- trim_right(string) => return string with white space trimmed on right side only
local trim_right = function (s)
  return (string.gsub(s, "^(.-)%s*$", "%1"))
end

-- Return a table containing the functions created by this module
return {
  trim_all = trim_all,
  trim_left = trim_left,
  trim_right = trim_right
}
```

Un approccio alternativo a quello sopra è quello di creare una tabella di livello superiore e quindi memorizzare le funzioni direttamente in essa. In quell'idioma, il nostro modulo in alto sarebbe simile a questo:

```
-- A conventional name for the table that will hold our functions
local M = {}

-- M.trim_all(string) => return string with white space trimmed on both sides
function M.trim_all(s)
```

```
  return (string.gsub(s, "^%s*(.-)%s*$", "%1"))
end

-- M.trim_left(string) => return string with white space trimmed on left side only
function M.trim_left(s)
  return (string.gsub(s, "^%s*(.*)$", "%1"))
end

-- trim_right(string) => return string with white space trimmed on right side only
function M.trim_right(s)
  return (string.gsub(s, "^(.-)%s*$", "%1"))
end


return M
```

Dal punto di vista del chiamante, c'è poca differenza tra i due stili. (Una differenza che vale la pena menzionare è che il primo stile rende più difficile per gli utenti la monkeypatch del modulo.Questo è un pro o un contro, a seconda del tuo punto di vista.Per maggiori dettagli su questo, vedi questo post del blog di Enrique García Cota.)

Usando il modulo

```
-- The following assumes that trim module is installed or in the caller's package.path,
-- which is a built-in variable that Lua uses to determine where to look for modules.
local trim = require "trim"

local msg = "    Hello, world!      "
local cleaned = trim.trim_all(msg)
local cleaned_right = trim.trim_right(msg)
local cleaned_left = trim.trim_left(msg)

-- It's also easy to alias functions to shorter names.
local trimr = trim.trim_right
local triml = trim.trim_left
```

Leggi Scrivere e usare i moduli online: https://riptutorial.com/it/lua/topic/1148/scrivere-e-usare-i-moduli

Capitolo 16: tabelle

Sintassi

- ipairs (numeric_table) - Tabella di Lua con iteratore di indici numerici
- pair (input_table) - generico iteratore di tabella Lua
- key, value = next (input_table, input_key) - Selettore del valore della tabella Lua
- table.insert (input_table, [position], value) - inserisce il valore specificato nella tabella di input
- removed_value = table.remove (input_table, [position]): popout o rimozione del valore specificato dalla posizione

Osservazioni

Le tabelle sono l'unica struttura dati incorporata disponibile in Lua. Questa è elegante semplicità o confusa, a seconda di come la si guarda.

Una tabella Lua è una raccolta di coppie chiave-valore in cui le chiavi sono univoche e né la chiave né il valore sono `nil` . In quanto tale, una tabella Lua può assomigliare ad un dizionario, hashmap o array associativo proveniente da altre lingue. Molti schemi strutturali possono essere creati con tabelle: stack, code, set, elenchi, grafici, ecc. Infine, le tabelle possono essere utilizzate per creare *classi* in Lua e per creare un sistema di *moduli* .

Lua non applica regole particolari su come vengono utilizzate le tabelle. Gli oggetti contenuti in una tabella possono essere una miscela di tipi Lua. Ad esempio, una tabella può contenere stringhe, funzioni, valori booleani, numeri *e persino altre tabelle* come valori o chiavi.

Una tabella Lua con tasti interi positivi consecutivi che iniziano con 1 è detta avere una sequenza. Le coppie chiave-valore con chiavi intere positive sono gli elementi della sequenza. Altre lingue chiamano questo array basato su 1. Alcune operazioni e funzioni standard funzionano solo sulla sequenza di una tabella e alcune hanno un comportamento non deterministico quando vengono applicate a una tabella senza una sequenza.

L'impostazione di un valore in una tabella su `nil` rimuove dalla tabella. Gli iteratori non vedrebbero più la chiave correlata. Quando si codifica per una tabella con una sequenza, è importante evitare di interrompere la sequenza; Rimuovere solo l'ultimo elemento o utilizzare una funzione, come la `table.remove` standard.remove, che sposta gli elementi verso il basso per colmare il divario.

Examples

Creare tabelle

La creazione di una tabella vuota è semplice come questa:

```
local empty_table = {}
```

Puoi anche creare una tabella sotto forma di un semplice array:

```
local numeric_table = {
    "Eve", "Jim", "Peter"
}
-- numeric_table[1] is automatically "Eve", numeric_table[2] is "Jim", etc.
```

Tieni presente che, per impostazione predefinita, l'indicizzazione della tabella inizia da 1.

È anche possibile creare una tabella con elementi associativi:

```
local conf_table = {
    hostname = "localhost",
    port     = 22,
    flags    = "-Wall -Wextra"
    clients  = {                -- nested table
        "Eve", "Jim", "Peter"
    }
}
```

L'uso sopra è lo zucchero di sintassi per ciò che è sotto. Le chiavi in questa istanza sono del tipo, stringa. La sintassi precedente è stata aggiunta per far apparire le tabelle come record. Questa sintassi in stile record è parallela alla sintassi per l'indicizzazione delle tabelle con le chiavi stringa, come mostrato nell'esercitazione sull'uso di base.

Come spiegato nella sezione commenti, la sintassi stile record non funziona per ogni chiave possibile. Inoltre una chiave può essere qualsiasi valore di qualsiasi tipo, e gli esempi precedenti riguardavano solo stringhe e numeri sequenziali. In altri casi dovrai utilizzare la sintassi esplicita:

```
local unique_key = {}
local ops_table = {
    [unique_key] = "I'm unique!"
    ["^"]  = "power",
    [true] = true
}
```

Tabella di iterazione

La libreria standard Lua fornisce una funzione di `pairs` che itera su chiavi e valori di una tabella. Durante l'iterazione con `pairs` non esiste un ordine specificato per attraversare, *anche se i tasti della tabella sono numerici* .

```
for key, value in pairs(input_table) do
    print(key, " -- ", value)
end
```

Per le tabelle che utilizzano **i tasti numerici** , Lua fornisce una funzione di `ipairs` . La funzione `ipairs` eseguirà sempre iterazioni dalla `table[1]` , dalla `table[2]` , ecc. Fino a trovare il primo valore `nil` .

```
for index, value in ipairs(numeric_table) do
```

```
    print(index, ". ", value)
end
```

Tieni presente che l'iterazione usando `ipairs()` non funzionerà come potresti volere in poche occasioni:

- `input_table` ha "buchi" in esso. (Per ulteriori informazioni, consultare la sezione "Come evitare gli spazi vuoti nelle tabelle utilizzate come matrici".) Ad esempio:

  ```
  table_with_holes = {[1] = "value_1", [3] = "value_3"}
  ```

- le chiavi non erano tutte numeriche. Per esempio:

  ```
  mixed_table = {[1] = "value_1", ["not_numeric_index"] = "value_2"}
  ```

Naturalmente, quanto segue funziona anche per una tabella che è una sequenza corretta:

```
for i = 1, #numeric_table do
    print(i, ". ", numeric_table[i])
end
```

L'iterazione di una tabella numerica in ordine inverso è semplice:

```
for i = #numeric_table, 1, -1 do
    print(i, ". ", numeric_table[i])
end
```

Un ultimo modo per scorrere le tabelle è usare il `next` selettore in un ciclo `for` generico . Come `pairs` non esiste un ordine specifico per attraversare. (Il metodo delle `pairs` usa il `next` internamente, quindi usare il `next` è essenzialmente una versione più manuale delle `pairs` `pairs` nel manuale di riferimento di Lua e `next` nel manuale di riferimento di Lua per maggiori dettagli.)

```
for key, value in next, input_table do
    print(key, value)
end
```

Uso di base

L'utilizzo della tabella di base include l'accesso e l'assegnazione di elementi di tabella, l'aggiunta di contenuto di tabella e la rimozione del contenuto della tabella. Questi esempi presumono che tu sappia come creare tabelle.

Accesso agli elementi

Data la seguente tabella,

```
local example_table = {"Nausea", "Heartburn", "Indigestion", "Upset Stomach",
                       "Diarrhea", cure = "Pepto Bismol"}
```

Si può indicizzare la parte sequenziale della tabella usando la sintassi dell'indice, l'argomento della sintassi dell'indice è la chiave della coppia chiave-valore desiderata. Come spiegato nel tutorial di creazione, la maggior parte della sintassi delle dichiarazioni è zucchero sintattico per la dichiarazione delle coppie chiave-valore. Gli elementi inclusi sequenzialmente, come i primi cinque valori in `example_table` , usano valori interi crescenti come chiavi; la sintassi del record usa il nome del campo come una stringa.

```
print(example_table[2])        --> Heartburn
print(example_table["cure"])   --> Pepto Bismol
```

Per le chiavi di stringa c'è lo zucchero di sintassi per mettere in parallelo la sintassi stile record per le chiavi stringa nella creazione della tabella. Le due righe seguenti sono equivalenti.

```
print(example_table.cure)      --> Pepto Bismol
print(example_table["cure"])   --> Pepto Bismol
```

Puoi accedere alle tabelle usando le chiavi che non hai mai usato prima, non è un errore come in altre lingue. In questo modo restituisce il valore predefinito `nil` .

Assegnazione di elementi

È possibile modificare elementi di tabella esistenti assegnandoli a una tabella utilizzando la sintassi dell'indice. Inoltre, la sintassi di indicizzazione in stile record è disponibile anche per l'impostazione dei valori

```
example_table.cure = "Lots of water, the toilet, and time"
print(example_table.cure)    --> Lots of water, the toilet, and time

example_table[2] = "Constipation"
print(example_table[2])      --> Constipation
```

Puoi anche aggiungere nuovi elementi a una tabella esistente usando il compito.

```
example_table.copyright_holder = "Procter & Gamble"
example_table[100] = "Emergency source of water"
```

Nota speciale: alcune stringhe non sono supportate con la sintassi del record. Vedere la sezione commenti per i dettagli.

Rimozione di elementi

Come affermato in precedenza, il valore predefinito per una chiave senza valore assegnato è `nil` . Rimuovere un elemento da una tabella è semplice come reimpostare il valore di una chiave sul valore predefinito.

```
example_table[100] = "Face Mask"
```

Gli elementi sono ora indistinguibili da un elemento non impostato.

Lunghezza della tabella

Le tabelle sono semplicemente array associativi (vedi osservazioni), ma quando si usano chiavi intere contigue a partire da 1 la tabella si dice abbia una *sequenza* .

La ricerca della lunghezza della parte di sequenza di una tabella viene eseguita utilizzando `#` :

```
local example_table = {'a', 'l', 'p', 'h', 'a', 'b', 'e', 't'}
print(#example_table)    --> 8
```

È possibile utilizzare l'operazione di lunghezza per aggiungere facilmente elementi a una tabella di sequenze.

```
example_table[#example_table+1] = 'a'
print(#example_table)    --> 9
```

Nell'esempio precedente, il valore precedente di `#example_table` è `8` , l'aggiunta `1` ti dà la successiva chiave intera valida nella sequenza, `9` , quindi ... `example_table[9] = 'a'` . Questo funziona per qualsiasi lunghezza del tavolo.

Nota speciale: l' utilizzo di chiavi intere non contigue e a partire da 1 interrompe la sequenza trasformando la tabella in una *tabella sparsa* . In questo caso il risultato dell'operazione di lunghezza non è definito. Vedi la sezione commenti.

Utilizzo delle funzioni della libreria di tabella per aggiungere / rimuovere elementi

Un altro modo per aggiungere elementi a una tabella è la funzione `table.insert()` . La funzione di inserimento funziona solo sulle tabelle di sequenza. Esistono due modi per chiamare la funzione. Il primo esempio mostra il primo utilizzo, in cui si specifica l'indice per inserire l'elemento (il secondo argomento). Ciò spinge tutti gli elementi dall'indice dato a `#table` su una posizione. Il secondo esempio mostra l'altro utilizzo di `table.insert()` , in cui l'indice non è specificato e il valore dato viene aggiunto alla fine della tabella (indice `#table + 1`).

```
local t = {"a", "b", "d", "e"}
table.insert(t, 3, "c")         --> t = {"a", "b", "c", "d", "e"}

t = {"a", "b", "c", "d"}
table.insert(t, "e")            --> t = {"a", "b", "c", "d", "e"}
```

Per `table.insert()` parallelo per la rimozione di elementi è `table.remove()` . Allo stesso modo ha due semantiche di chiamata: una per rimuovere elementi in una determinata posizione e un'altra per rimuovere dalla fine della sequenza. Quando si rimuove dal centro di una sequenza, tutti gli elementi seguenti vengono spostati di un indice.

```
local t = {"a", "b", "c", "d", "e"}
local r = table.remove(t, 3)        --> t = {"a", "b", "d", "e"}, r = "c"

t = {"a", "b", "c", "d", "e"}
r = table.remove(t)                 --> t = {"a", "b", "c", "d"}, r = "e"
```

Queste due funzioni mutano la tabella data. Come si potrebbe essere in grado di dire il secondo metodo di chiamata `table.insert()` e `table.remove()` fornisce semantica dello stack alle tabelle. Facendo leva su questo, puoi scrivere il codice come nell'esempio qui sotto.

```
function shuffle(t)
    for i = 0, #t-1 do
        table.insert(t, table.remove(t, math.random(#t-i)))
    end
end
```

Implementa la Fisher-Yates Shuffle, forse in modo inefficiente. Usa `table.insert()` per aggiungere l'elemento estratto a caso alla fine della stessa tabella e `table.remove()` per estrarre in modo casuale un elemento dalla parte rimanente della tabella non mescolata.

Evitare gli spazi vuoti nelle tabelle utilizzate come array

Definire i nostri termini

Per *array* qui intendiamo una tabella Lua usata come sequenza. Per esempio:

```
-- Create a table to store the types of pets we like.
local pets = {"dogs", "cats", "birds"}
```

Stiamo usando questa tabella come una sequenza: un gruppo di elementi immessi da numeri interi. Molte lingue chiamano questo array, e anche noi. Ma a rigor di termini, non c'è niente come un array a Lua. Ci sono solo tabelle, alcune delle quali sono simili ad array, alcune delle quali sono simili a hash (o simili a un dizionario, se preferite), e alcune sono miste.

Un punto importante sul nostro array di `pets` è che non ha lacune. Il primo oggetto, `pets[1]` , è la stringa "cani", il secondo oggetto, `pets[2]` , è la stringa "gatti", e l'ultimo oggetto, `pets[3]` , è "uccelli". La libreria standard di Lua e la maggior parte dei moduli scritti per Lua assumono 1 come primo indice per le sequenze. Un array gapless ha quindi elementi da `1..n` senza perdere alcun numero nella sequenza. (Nel caso limite, `n = 1` , e l'array ha solo un elemento in esso.)

Lua fornisce gli `ipairs` funzione `ipairs` per iterare su tali tabelle.

```
-- Iterate over our pet types.
for idx, pet in ipairs(pets) do
  print("Item at position " .. idx .. " is " .. pet .. ".")
end
```

Questo stamperebbe "L'oggetto in posizione 1 è un cane.", "L'oggetto in posizione 2 è un gatto.", "L'oggetto in posizione 3 è un uccello."

Ma cosa succede se facciamo quanto segue?

```
local pets = {"dogs", "cats", "birds"}
pets[12] = "goldfish"
for idx, pet in ipairs(pets) do
```

```
  print("Item at position " .. idx .. " is " .. pet .. ".")
end
```

Una matrice come questo secondo esempio è una matrice sparsa. Ci sono lacune nella sequenza. Questo array è simile al seguente:

```
{"dogs", "cats", "birds", nil, nil, nil, nil, nil, nil, nil, nil, "goldfish"}
-- 1        2       3      4    5    6    7    8    9    10   11       12
```

I valori nulli non occupano alcuna memoria aggiuntiva; internamente lua salva solo i valori `[1] = "dogs"` , `[2] = "cats"` , `[3] = "birtds"` e `[12] = "goldfish"`

Per rispondere alla domanda immediata, gli `ipairs` si fermeranno dopo gli uccelli; "goldfish" negli `pets[12]` non sarà mai raggiunto a meno che non modifichiamo il nostro codice. Questo perché gli `ipairs` `1..n-1` da `1..n-1` dove `n` è la posizione del primo `nil` trovato. Lua definisce la `table[length-of-table + 1]` come `nil` . Quindi, in una sequenza corretta, l'iterazione si interrompe quando Lua cerca di ottenere, per esempio, il quarto elemento in un array di tre elementi.

Quando?

Le due posizioni più comuni per i problemi che sorgono con gli array sparsi sono (i) quando si tenta di determinare la lunghezza dell'array e (ii) quando si tenta di ripetere l'array. In particolare:

- Quando si utilizza l'operatore di lunghezza `#` poiché l'operatore di lunghezza smette di contare al primo `nil` trovato.
- Quando si utilizza la funzione `ipairs()` poiché come detto sopra, si interrompe l'iterazione al primo `nil` trovato.
- Quando si utilizza la funzione `table.unpack()` poiché questo metodo interrompe la decompressione al primo `nil` trovato.
- Quando si utilizzano altre funzioni che (direttamente o indirettamente) accedono a uno dei precedenti.

Per evitare questo problema, è importante scrivere il codice in modo che se si prevede che una tabella sia un array, non si introducono spazi vuoti. Gli spazi possono essere introdotti in diversi modi:

- Se si aggiunge qualcosa a un array nella posizione sbagliata.
- Se si inserisce un valore `nil` in una matrice.
- Se rimuovi i valori da una matrice.

Potresti pensare: "Ma non farei mai nessuna di quelle cose". Bene, non intenzionalmente, ma ecco un esempio concreto di come le cose potrebbero andare storte. Immagina di voler scrivere un metodo di filtro per Lua come Ruby's `select` e Perl's `grep` . Il metodo accetterà una funzione di test e un array. Itera su tutta la serie, chiamando a turno il metodo di prova su ciascun oggetto. Se l'oggetto passa, quell'elemento viene aggiunto ad una matrice di risultati che viene restituita alla fine del metodo. Quanto segue è un'implementazione bacata:

```
local filter = function (fun, t)
```

```
  local res = {}
  for idx, item in ipairs(t) do
    if fun(item) then
      res[idx] = item
    end
  end

  return res
end
```

Il problema è che quando la funzione restituisce `false` , saltiamo un numero nella sequenza. Immagina di chiamare il `filter(isodd, {1,2,3,4,5,6,7,8,9,10})` : ci saranno degli spazi nella tabella restituita ogni volta che c'è un numero pari nell'array passato al `filter` .

Ecco un'implementazione fissa:

```
local filter = function (fun, t)
  local res = {}
  for _, item in ipairs(t) do
    if fun(item) then
      res[#res + 1] = item
    end
  end

  return res
end
```

Suggerimenti

1. Usa le funzioni standard: `table.insert(<table>, <value>)` si aggiunge sempre alla fine dell'array. `table[#table + 1] = value` è una mano breve per questo. `table.remove(<table>, <index>)` sposterà tutti i seguenti valori per riempire il gap (che può anche rallentare).
2. Controlla i valori `nil` **prima di** inserirli, evitando cose come `table.pack(function_call())` , che potrebbe introdurre valori `nil` nella nostra tabella.
3. Verificare i valori `nil` **dopo l'** inserimento e, se necessario, riempiendo lo spazio spostando tutti i valori consecutivi.
4. Se possibile, utilizza valori segnaposto. Ad esempio, cambia `nil` per `0` o un altro valore di segnaposto.
5. Se lasciare spazi vuoti è inevitabile, questo dovrebbe essere documentato in modo appropriato (commentato).
6. Scrivi un `__len()` e usa l'operatore `#` .

Esempio per 6 .:

```
tab = {"john", "sansa", "daenerys", [10] = "the imp"}
print(#tab) --> prints 3
setmetatable(tab, {__len = function() return 10 end})
-- __len needs to be a function, otherwise it could just be 10
print(#tab) --> prints 10
for i=1, #tab do print(i, tab[i]) end
--> prints:
-- 1 john
```

```
-- 2 sansa
-- 3 daenerys
-- 4 nil
-- ...
-- 10 the imp

for key, value in ipairs(tab) do print(key, value) end
--> this only prints '1 john \n 2 sansa \n 3 daenerys'
```

Un'altra alternativa è usare la funzione `pairs()` e filtrare gli indici non interi:

```
for key in pairs(tab) do
    if type(key) == "number" then
        print(key, tab[key]
    end
end
-- note: this does not remove float indices
-- does not iterate in order
```

Leggi tabelle online: https://riptutorial.com/it/lua/topic/676/tabelle

Titoli di coda

S. No	Capitoli	Contributors
1	Iniziare con Lua	1971chevycamaro, Allan Burleson, Community, DarkWiiPlayer, Darryl L Johnson, elektron, greatwolf, Guilherme Salazar, hjpotter92, hugomg, Kamiccolo, lhf, Nikola Geneshki, SoniEx2, Telemachus
2	Argomenti Variadici	greatwolf, Kamiccolo, ktb, RamenChef, SoniEx2
3	Booleani a Lua	DarkWiiPlayer, engineercoding, greatwolf, Kamiccolo, Katenkyo , Samuel McKay, Telemachus
4	coroutine	010110110101, Bjornir, Eshkation, Kamiccolo, ktb, SoniEx2
5	funzioni	Art C, Basilio German, DarkWiiPlayer, Firas Moalla, greatwolf, Guilherme Salazar, Jon Ericson, Katenkyo, ktb, MBorsch, Necktrox, qaisjp, RBerteig, Romário, SoniEx2, Telemachus, Unheilig, WolfgangTS
6	Garbage collector e tavoli deboli	greatwolf, Kamiccolo, val
7	Gestione degli errori	Black, DarkWiiPlayer, engineercoding, greatwolf
8	Imposta	Egor Skriptunoff, Jon Ericson, ryanpattison
9	Introduzione all'API Lua C	greatwolf, Jeremy Thien, Kamiccolo, Luiz Menezes, RBerteig, tversteeg
10	iteratori	Adam, Egor Skriptunoff, greatwolf
11	Metatables	DarkWiiPlayer, greatwolf, Kamiccolo, pschulz, Telemachus
12	Object-Orientamento	DarkWiiPlayer, Kamiccolo
13	Pattern matching	DarkWiiPlayer, engineercoding, Eshkation, greatwolf, Kamiccolo , Stephen Leppik
14	PICO-8	Jon Ericson
15	Scrivere e usare i moduli	SoniEx2, Telemachus
16	tabelle	DarkWiiPlayer, greatwolf, Hastumer, Kamiccolo, ktb, mjanicek, SoniEx2, Telemachus, Tom Blodget