Weak Tables Tutorial

It is helpful to understand *garbage collection* before reading this tutorial. The GarbageCollectionTutorial provides an introduction.

The weak link

In computer languages such as Lua that employ garbage collection, a reference to an object is said to be *weak* if it does not prevent collection of the object. Weak references are useful for determining when an object has been collected and for caching objects without impeding their collection.

Rather than provide an interface to individual weak references, Lua provides a higher -level construct called a *weak table*. In a weak table the keys and/or values are weak references. If a key or value of such a table is collected, that entry of the table will be removed.

Here is a complete, although very contrived, example of a weak table in action:

```
t = {}
setmetatable(t, { __mode = 'v' })

do
    local someval = {}
    t['foo'] = someval
end

collectgarbage()

for k, v in t do
    print(k, v)
end
```

Try this example with and without the `collectgarbage()` call commented out. With the call, the program will print nothing, as the value of the lone table entry will be collected.

A weak table is created by setting the `__mode` attribute in the metatable. In the above example we enable weak values with `v` (likewise `k` is for keys). The purpose of creating the test value `someval` as a local variable in a `do` block is so that we may force garbage collection of the value later with a call to `collectgarbage`. Note that using a literal such as a string or number for `someval` would not have worked, as literals are never garbage collected.

Since a table may already have a metatable, a safer way to enable the weak mode is with:

```
(getmetatable(t) or setmetatable(t, {})).__mode = 'v'
```

Weak tables are often used in situations where you wish to annotate values without altering them. For example, you might want to give objects a name which could be used when they were printed out. In this case, you would not want the fact that an object had been named to prevent it from being garbage collected, so you would want to use a table with weak keys (the objects):

```
local names = setmetatable({}, {__mode = "k"})

-- with the example below, this would be a local function
function name(obj, str)
  names[obj] = tostring(str)
  return obj
end

-- keep the original print function available
local _print = print
function print(...)
  for i = 1, arg.n do
    local name = names[arg[i]]
    if name then arg[i] = name end
  end
  _print(unpack(arg))
end
```

You might want to use this for debugging, by automatically naming global variables. You can do this by adding a simple metamethod (see MetamethodsTutorial) to the globals table:

```
local globalsmeta = {}
local nameable_type = {["function"] = true, userdata = true, thread = true, table = true}

function globalsmeta:__newindex(k, v)
  if nameable_type[type(v)] then name(v, k) end
  rawset(self, k, v)
end
setmetatable(_G, globalsmeta)
```

Note how we avoid doing a complex series of `if then elseif...` statements by using a constant table to do a single check on the type of the value.

For advanced tutorial readers, the text of the __newindex function could be written as follows:

```
    rawset(self, k, (nameable_type[type(v)] and name(v, k)) or v)
```

FindPage · RecentChanges · preferences
edit · history
Last edited June 25, 2003 4:27 pm PDT (diff)