简单的Lua类模拟

本文要翻译自Simple Lua Classes，主要介绍一种在Lua中模拟类的简单方法。

A Simplified Way to Declare Lua Classes

本节介绍一种简化的声明类的方法。

本来Lua是没有类(class)的，Lua只有表(table)，但是Lua的强大的元编程工具(metaprogramming facilities)使得我们可以定义经典的对象，模拟类。实际上，有许多方法可以实现模拟类。

这里描述的方法是最常见、最灵活的方法，使用元表(metatables)。通过给table设置一个元表，可以定制该table的行为。例如，一个table有一个metatable，此metatable有一个`__index`键。如果此`__index`是一个函数，则任何在table中找不到的key都会被传递给`__index`，在`__index`函数中进行查找。当然metatable的`__index`本身也可以是一个table，则在原table中查找不到的东西会在`__index`表中进行查找。基本思路如下:

```
Account = {} -- 相当于类(也是一个表)
Account.__index = Account --类的__index指向自身(一个表)

function Account:create(balance) -- 创建对象的方法
   local acnt = {}             -- acnt是创建出来的对象
   -- 设置acnt的元素为类Account，则在acnt中找不到的东西
   -- 就在Account的__index指向的表(即Acctount本身)中查找
   setmetatable(acnt,Account)
   acnt.balance = balance      -- 初始化创建的对象
   return acnt
end

function Account:withdraw(amount) -- 定义一个方法withdraw
   self.balance = self.balance - amount
end

-- 创建一个Account的对象acc，并调用withdaw方法
acc = Account:create(1000)
acc:withdraw(100)
```

上面代码作用就是声明一个类`Account`，表示银行账户。`Account`有一个函数`create`，用于创建实例对象，其参数`balance`表示账户余额。`withdraw`函数表示取款操作。

这里`Account`的对象(`acc`)是用table表示的，只包含一个字段`balance`(余额)。Lua会尝试在`acc`中查找`withdraw`函数，但是找不到。因为`acc`的元表定义了`__index`，Lua会成元表中查找`withdraw`。因此，`acc:withdraw(100)`实际上相当于`Account.withdraw(acc,100)`。我们实际上可以把`withdraw`直接放进`acc`，但是这会很浪费、不灵活 - 每添加一个新方法都需要修改`create`函数。

A simplified way of creating classes

本节介绍一种创建类的简化的方法。

我将会定义一个函数`class`，它将会(transparently)完成一切，见下面的`class.lua`代码。

```lua
-- 使用class创建一个类Account
Account = class(function(acc,balance)
               acc.balance = balance
            end)

function Account:withdraw(amount)
   self.balance = self.balance - amount
end

-- 创建一个Account的实例对象，这里使用了call notation，
-- 即类名加圆括号Account(1000)，类似c++的call operator。
acc = Account(1000)
acc:withdraw(100)
```

这里，我们向新类提供了一个初始化函数，然后会自动生成一个构造器`constructor`。

也支持简单地继承。例如，下面的代码定义了一个基类`Animal`，可以声明几种具体的animals。所有通过`class`函数的类都有一个`is_a`方法，你可以用它找到类的运行时真正类型。

```lua
-- animal.lua

require 'class' -- class的定义见下文

-- 定义一个Animal
Animal = class(function(a,name)
   a.name = name
end)

function Animal:__tostring()
  return self.name..': '..self:speak()
end

-- 定义一个Dog类，继承自基类Animal
Dog = class(Animal)

function Dog:speak()
```

```lua
  return 'bark'
end

-- 定义一个Cat类，继承自基类Animal，并提供了一个init函数
Cat = class(Animal, function(c,name,breed)
         Animal.init(c,name)  -- must init base!
         c.breed = breed
       end)

function Cat:speak()
  return 'meow'
end

-- 定义一个Lion类，，继承自基类Cat
Lion = class(Cat)

function Lion:speak()
  return 'roar'
end

-- 创建Dog类的实例fido
fido = Dog('Fido')
-- 创建Cat类的实例felix
felix = Cat('Felix','Tabby')
-- 创建Lion类的实例leo
leo = Lion('Leo','African')
```

```
D:\Downloads\func>lua -i animal.lua
= fido,felix,leo
Fido: bark      Felix: meow     Leo: roar
= leo:is_a(Animal)
true
= leo:is_a(Dog)
false
= leo:is_a(Cat)
true
```

所有的Animal都定义了`__tostring`，无论何时需要一个对象的字符串表示时，Lua都会调用此函数。同时它又依赖`speak`，这在是没有定义的。这就是C++程序员所熟知的抽象基类，而具体的子类如`Dog`定义了`speak`。

请注意如果子类有它们自己的初始化函数，它们(initialization functions)必须手动显示地调用基类的`init`函数。

Implementation of class()

这一小节讲的是上面用到的`class`函数的实现。

`class()`使用了两个技巧。它允许你使用调用符号(call notation，如上面的`Dog('fido')`)构建一个类(即生成类的实例对象)。这是通过给类的设置一个定义了`__call`的元表实现的。继承是通过把基类的fields浅拷贝到子类实现的。这不是唯一实现继承的方法，比如，我们可以定义`__index`元方法，让其在基类中查找子类调用的函数。此处使用的方法会有更好的性能，代价是使得class objects看起来有点臃肿。每一个子类都持有一个`_base`字段，其代表基类，用于实现`is_a`函数。

注在运行时修改意基类不会影响子类。

```lua
-- class.lua
-- 与lua 5.1兼容 (不兼容 5.0).
function class(base, init)
   local c = {}    -- 要返回的新的class类型
   if not init and type(base) == 'function' then
      init = base
      base = nil
   elseif type(base) == 'table' then
    -- 新的类(c)是基类(base)的浅拷贝
      for i,v in pairs(base) do
         c[i] = v
      end
      c._base = base -- type(base) == 'table'
   end
   -- 类c将会是它所有实例对象的元表(metatable)，
   -- 并且实例对象会成c中查找方法.
   c.__index = c

   -- 暴露一个构造函数，调用方式: <classname>(<args>)
   local mt = {} -- 类c的元表
   mt.__call = function(class_tbl, ...) -- 1. 把类c设为对象元表 2. 初始化
       local obj = {}
       setmetatable(obj,c) --把类c设为对象ob的元表
       if init then
          init(obj,...) --这里没有调用基类的init，所以init不为空时，要在子类的init中手动调用基类的init
       else
          -- 确保基类的东西都初始化了
          if base and base.init then
            base.init(obj, ...)
          end
       end
       return obj
   end --mt.__call() 定义结束

   c.init = init
   c.is_a = function(self, klass)
      local m = getmetatable(self)
      while m do
         if m == klass then return true end
         m = m._base
```

```
        end
        return false
    end

    setmetatable(c, mt)
    return c
end
```

(可能未完待续)

References:

- Simple Lua Classes
- Lua元表
- Object Oriented Programming