前言

元表对应的英文是metatable，元方法是metamethod。我们都知道，在C++中，两个类是无法直接相加的，但是，如果你重载了“+”符号，就可以进行类的加法运算。在Lua中也有这个道理，两个table类型的变量，你是无法直接进行“+”操作的，如果你定义了一个指定的函数，就可以进行了。那这篇博文就是主要讲的如何定义这个指定的函数，这个指定的函数是什么？希望对学习Lua的朋友有帮助。

Lua是怎么做的？

通常，Lua中的每个值都有一套预定义的操作集合，比如数字是可以相加的，字符串是可以连接的，但是对于两个table类型，则不能直接进行“+”操作。这需要我们进行一些操作。在Lua中有一个元表，也就是上面说的metatable，我们可以通过元表来修改一个值得行为，使其在面对一个非预定义的操作时执行一个指定的操作。比如，现在有两个table类型的变量a和b，我们可以通过metatable定义如何计算表达式a+b，具体的在Lua中是按照以下步骤进行的：

1. 先判断a和b两者之一是否有元表；
2. 检查该元表中是否有一个叫__add的字段；
3. 如果找到了该字段，就调用该字段对应的值，这个值对应的是一个metamethod；（Lua中方法是可以放在一个字段中的，还记得？？？忘了点这里 (http://www.jellythink.com/archives/495)）
4. 调用__add对应的metamethod计算a和b的值。

上述四个步骤就是计算table类型变量a+b的过程。在Lua中，每个值都有一个元表，table和userdata类型的每个变量都可以有各自独立的元表，而其他类型的值则共享其类型所属的单一元表。

告别metatable小白

现在就说说最基本的metatable内容。Lua在创建新的table时不会创建元表，比如以下代码就可以演示：

```
local t = {1, 2}
print(getmetatable(t))     -- nil
```

我们是使用getmetatable来获取一个table或userdata类型变量的元表，当创建新的table变量时，使用getmetatable去获得元表，将返回nil；同理，我们也可以使用setmetatable去设置一个table或userdata类型变量的元表，例如以下代码：

```
local t = {}
print(getmetatable(t))     -->nil

local t1 = {}
setmetatable(t, t1)
assert(getmetatable(t) == t1)
```

任何table都可以作为任何值得元表，而一组相关的table有可以共享一个通用的元表，此元表描述了它们共同的行为。一个table甚至可以作为它自己的元表，用于描述其特有的行为。总之，任何搭配形式都是合法的。

在Lua代码中，只能设置table的元表。若要设置其它类型的值得元表，则必须通过C代码来完成。还存在一个特例，对于字符串，标准的字符串程序库为所有的字符串都设置了一个元表，而其它类型在默认情况下都没有元表。查看两句代码的打印值，就可以看出来：

```
print(getmetatable("Hello World"))
print(getmetatable(10))
```

在table中，我可以重新定义的元方法有以下几个：

```
__add(a, b) --加法
__sub(a, b) --减法
__mul(a, b) --乘法
__div(a, b) --除法
__mod(a, b) --取模
__pow(a, b) --乘幂
__unm(a) --相反数
__concat(a, b) --连接
__len(a) --长度
__eq(a, b) --相等
__lt(a, b) --小于
__le(a, b) --小于等于
__index(a, b) --索引查询
__newindex(a, b, c) --索引更新（PS：不懂的话，后面会有讲）
__call(a, ...) --执行方法调用
__tostring(a) --字符串输出
__metatable --保护元表
```

接下来就介绍介绍如果去重新定义这些方法。

算术类的元方法

现在我使用完整的实例代码来详细的说明算术类元方法的使用。我准备定义一些对集合的操作方法，所有的方法都放入Set这个table中，至于为什么table中可以存放函数，可以参考《Lua中的函数》(http://www.jellythink.com/archives/495)这篇文章。下面的代码是我模拟的一个集合的操作：

```
Set = {}
local mt = {} -- 集合的元表

-- 根据参数列表中的值创建一个新的集合
function Set.new(l)
    local set = {}
     setmetatable(set, mt)
    for _, v in pairs(l) do set[v] = true end
     return set
end

-- 并集操作
function Set.union(a, b)
    local retSet = Set.new{} -- 此处相当于Set.new({})
    for v in pairs(a) do retSet[v] = true end
    for v in pairs(b) do retSet[v] = true end
    return retSet
end

-- 交集操作
function Set.intersection(a, b)
    local retSet = Set.new{}
    for v in pairs(a) do retSet[v] = b[v] end
    return retSet
end

-- 打印集合的操作
function Set.toString(set)
     local tb = {}
     for e in pairs(set) do
          tb[#tb + 1] = e
     end
     return "{" .. table.concat(tb, ", ") .. "}"
end

function Set.print(s)
     print(Set.toString(s))
end
```

现在，我定义“+”来计算两个集合的并集，那么就需要让所有用于表示集合的table共享一个元表，并且在该元表中定义如何执行一个加法操作。首先创建一个常规的table，准备用作集合的元表，然后修改Set.new函数，在每次创建集合的时候，都为新的集合设置一个元表。代码如下：

```
Set = {}
local mt = {} -- 集合的元表

-- 根据参数列表中的值创建一个新的集合
function Set.new(l)
    local set = {}
     setmetatable(set, mt)
    for _, v in pairs(l) do set[v] = true end
     return set
end
```

在此之后，所有由Set.new创建的集合都具有一个相同的元表，例如：

```
local set1 = Set.new({10, 20, 30})
local set2 = Set.new({1, 2})
print(getmetatable(set1))
print(getmetatable(set2))
assert(getmetatable(set1) == getmetatable(set2))
```

最后，我们需要把元方法加入元表中，代码如下：

```
mt.__add = Set.union
```

这以后，只要我们使用“+”符号求两个集合的并集，它就会自动的调用Set.union函数，并将两个操作数作为参数传入。比如以下代码：

```
local set1 = Set.new({10, 20, 30})
local set2 = Set.new({1, 2})
local set3 = set1 + set2
Set.print(set3)
```

在上面列举的那些可以重定义的元方法都可以使用上面的方法进行重定义。现在就出现了一个新的问题，set1和set2都有元表，那我们要用谁的元表啊？虽然我们这里的示例代码使用的都是一个元表，但是实际coding中，会遇到我这里说的问题，对于这种问题，Lua是按照以下步骤进行解决的：

1. 对于二元操作符，如果第一个操作数有元表，并且元表中有所需要的字段定义，比如我们这里的__add元方法定义，那么Lua就以这个字段为元方法，而与第二个值无关；
2. 对于二元操作符，如果第一个操作数有元表，但是元表中没有所需要的字段定义，比如我们这里的__add元方法定义，那么Lua就去查找第二个操作数的元表；
3. 如果两个操作数都没有元表，或者都没有对应的元方法定义，Lua就引发一个错误。

以上就是Lua处理这个问题的规则，那么我们在实际编程中该如何做呢？比如set3 = set1 + 8这样的代码，就会打印出以下的错误提示：

```
lua: test.lua:16: bad argument #1 to 'pairs' (table expected, got number)
```

但是，我们在实际编码中，可以按照以下方法，弹出我们定义的错误消息，代码如下：

```
function Set.union(a, b)
     if getmetatable(a) ~= mt or getmetatable(b) ~= mt then
          error("metatable error.")
     end

    local retSet = Set.new{} -- 此处相当于Set.new({})
    for v in pairs(a) do retSet[v] = true end
    for v in pairs(b) do retSet[v] = true end
    return retSet
end
```

当两个操作数的元表不是同一个元表时，就表示二者进行并集操作时就会出现问题，那么我们就可以打印出我们需要的错误消息。

上面总结了算术类的元方法的定义，关系类的元方法和算术类的元方法的定义是类似的，这里不做累述。

__tostring元方法

写过Java或者C#的人都知道，Object类中都有一个tostring的方法，程序员可以重写该方法，以实现自己的需求。在Lua中，也是这样的，当我们直接print(a)（a是一个table）时，是不可以的。那怎么办，这个时候，我们就需要自己重新定义__tostring元方法，让print可以格式化打印出table类型的数据。

函数print总是调用tostring来进行格式化输出，当格式化任意值时，tostring会检查该值是否有一个__tostring的元方法，如果有这个元方法，tostring就用该值作为参数来调用这个元方法，剩下实际的格式化操作就由__tostring元方法引用的函数去完成，该函数最终返回一个格式化完成的字符串。例如以下代码：

```
mt.__tostring = Set.toString
```

如何保护我们的“奶酪”——元表

我们会发现，使用getmetatable就可以很轻易的得到元表，使用setmetatable就可以很容易的修改元表，那这样做的风险是不是太大了，那么如何保护我们的元表不被篡改呢？

在Lua中，函数setmetatable和getmetatable函数会用到元表中的一个字段，用于保护元表，该字段是__metatable。当我们想要保护集合的元表，是用户既不能看也不能修改集合的元表，那么就需要使用__metatable字段了；当设置了该字段时，getmetatable就会返回这个字段的值，而setmetatable则会引发一个错误；如以下演示代码：

```
function Set.new(l)
    local set = {}
     setmetatable(set, mt)
    for _, v in pairs(l) do set[v] = true end
     mt.__metatable = "You cannot get the metatable" -- 设置完我的元表以后，不让其他人再设置
     return set
end

local tb = Set.new({1, 2})
print(tb)

print(getmetatable(tb))
setmetatable(tb, {})
```

上述代码就会打印以下内容：

```
{1, 2}
You cannot get the metatable
lua: test.lua:56: cannot change a protected metatable
```

__index元方法

是否还记得当我们访问一个table中不存在的字段时，会返回什么值？默认情况下，当我们访问一个table中不存在的字段时，得到的结果是nil。但是这种状况很容易被改变；Lua是按照以下的步骤决定是返回nil还是其它值得：

1. 当访问一个table的字段时，如果table有这个字段，则直接返回对应的值；
2. 当table没有这个字段，则会促使解释器去查找一个叫__index的元方法，接下来就就会调用对应的元方法，返回元方法返回的值；
3. 如果没有这个元方法，那么就返回nil结果。

下面通过一个实际的例子来说明__index的使用。假设要创建一些描述窗口，每个table中都必须描述一些窗口参数，例如颜色，位置和大小等，这些参数都是有默认值得，因此，我们在创建窗口对象时可以指定那些不同于默认值得参数。

```
Windows = {} -- 创建一个命名空间

-- 创建默认值表
Windows.default = {x = 0, y = 0, width = 100, height = 100, color = {r = 255, g = 255, b = 255}}

Windows.mt = {} -- 创建元表

-- 声明构造函数
function Windows.new(o)
     setmetatable(o, Windows.mt)
     return o
end

-- 定义__index元方法
Windows.mt.__index = function (table, key)
     return Windows.default[key]
end

local win = Windows.new({x = 10, y = 10})
print(win.x)                -- >10 访问自身已经拥有的值
print(win.width)           -- >100 访问default表中的值
print(win.color.r)           -- >255 访问default表中的值
```

根据上面代码的输出，结合上面说的那三步，我们再来看看，print(win.x)时，由于win变量本身就拥有x字段，所以就直接打印了其自身拥有的字段的值；print(win.width)，由于win变量本身没有width字段，那么就去查找是否拥有元表，元表中是否有__index对应的元方法，由于存在__index元方法，返回了default表中的width字段的值，print(win.color.r)也是同样的道理。

在实际编程中，__index元方法不必一定是一个函数，它还可以是一个table。当它是一个函数时，Lua以table和不存在key作为参数来调用该函数，这就和上面的代码一样；当它是一个table时，Lua就以相同的方式来重新访问这个table，所以上面的代码也可以是这样的：

```
-- 定义__index元方法
Windows.mt.__index = Windows.default
```

__newindex元方法

__newindex元方法与__index类似，__newindex用于更新table中的数据，而__index用于查询table中的数据。当对一个table中不存在的索引赋值时，在Lua中是按照以下步骤进行的：

1. Lua解释器先判断这个table是否有元表；
2. 如果有了元表，就查找元表中是否有__newindex元方法；如果没有元表，就直接添加这个索引，然后对应的赋值；

3. 如果有这个__newindex元方法，Lua解释器就执行它，而不是执行赋值；
4. 如果这个__newindex对应的不是一个函数，而是一个table时，Lua解释器就在这个table中执行赋值，而不是对原来的table。

那么这里就出现了一个问题，看以下代码：

```
local tb1 = {}
local tb2 = {}

tb1.__newindex = tb2
tb2.__newindex = tb1

setmetatable(tb1, tb2)
setmetatable(tb2, tb1)

tb1.x = 10
```

发现什么问题了么？是不是循环了，在Lua解释器中，对这个问题，就会弹出错误消息，错误消息如下：

```
loop in settable
```

丢掉那该死的元表

有的时候，我们就不想从__index对应的元方法中查询值，我们也不想更新table时，也不想执行__newindex对应的方法，或者__newindex对应的table。那怎么办？在Lua中，当我们查询table中的值，或者更新table中的值时，不想理那该死的元表，我们可以使用rawget函数，调用rawget(tb, i)就是对table tb进行了一次“原始的（raw）”访问，也就是一次不考虑元表的简单访问；你可能会想，一次原始的访问，没有访问__index对应的元方法，可能有性能的提升，其实一次原始访问并不会加速代码执行的速度。对于__newindex元方法，可以调用rawset(t, k, v)函数，它可以不涉及任何元方法而直接设置table t中与key k相关联的value v。

总结

这篇博文具体的总结了Lua中的元表和元方法，可以说Lua中的元表和元方法是很多内容的基础，所以我在这里总结的很详细，并结合了很多代码。如果你有幸看到了这篇文章，希望你也花点时间认真的读一读，想要理解Lua，玩转Lua，当然了，不能只是会一些语法，掌握元表和元方法是必不可少的。最后，也希望这篇文章对大家有用。下一篇博文，我会结合__index和__newindex说一些实例代码。

还有两个月，游戏就要上线了，希望能准时上线，好忙，好忙！

2014年7月17日 于深圳。