upvalue 相关操作

registry 实现了全局的值，upvalue 机制实现了与**C static** 变量等价的东东，**这种变量只能在特定的函数内可见**。每当你在 Lua 中创建一个新的 C 函数，你可以将这个函数与任意多个 upvalues 联系起来，每一个 upvalue 可以持有一个单独的 Lua 值。下面**当函数被调用的时候，可以通过假索引自由的访问任何一个 upvalues。**

我们称这种一个**C 函数和她的 upvalues 的组合为闭包（closure）**。记住：在 Lua 代码中，一个闭包是一个从外部函数访问局部变量的函数。一个 C 闭包与一个 Lua 闭包相近。关于闭包的一个有趣的事实是，你可以使用相同的函数代码创建不同的闭包，带有不同的 upvalues。

看一个简单的例子，我们在 C 中创建一个 newCounter 函数。（我们已经在 6.1 节部分在 Lua 中定义过同样的函数）。这个函数是个函数工厂：每次调用他都返回一个新的 counter 函数。尽管所有的 counters 共享相同的 C 代码，但是每个都保留独立的 counter 变量，工厂函数如下：

```
/* forward declaration */
static int counter (lua_State *L);

int newCounter (lua_State *L) {
  lua_pushnumber(L, 0);
  lua_pushcclosure(L, &counter, 1);
  return 1;
}
```

这里的关键函数是 **lua_pushcclosure**，她的**第二个参数是一个基本函数**（例子中卫 counter），**第三个参数是 upvalues 的个数**（例子中为 1）。**在创建新的闭包之前，我们必须将 upvalues 的初始值入栈**，在我们的例子中，我们将数字 0 作为唯一的 upvalue 的初始值入栈。如预期的一样，lua_pushcclosure 将新的闭包放到栈内，因此闭包已经作为 newCounter 的结果被返回。

现在，我们看看 counter 的定义：

```
static int counter (lua_State *L) {
  double val = lua_tonumber(L, lua_upvalueindex(1));
  lua_pushnumber(L, ++val);  /* new value */
  lua_pushvalue(L, -1);    /* duplicate it */
  lua_replace(L, lua_upvalueindex(1)); /* update upvalue */
  return 1; /* return new value */
}
```

这里的关键函数是 **lua_upvalueindex**（实际是一个宏），**用来产生一个 upvalue 的假索引**。这个假索引除了不在栈中之外，和其他的索引一样。表达式 lua_upvalueindex(1)函数第一个 upvalue 的索引。因此，在函数 counter 中的 **lua_tonumber 获取第一个(仅有的)upvalue 的当前值**，转换为数字型。然后，函数 counter 将新的值++val 入栈，并将这个值的一个拷贝使用新的值替换 upvalue。最后，返回其他的拷贝。

与 Lua 闭包不同的是，C 闭包不能共享 upvalues：每一个闭包都有自己独立的变量集。然而，我们可以设置不同函数的 upvalues 指向同一个表，这样这个表就变成了一个所有函数共享数据的地方。