**Lua闭包与c闭包**

在lua中，只有闭包，函数只是upvalue数量为零的闭包。闭包是函数和upvalue的结合体，在lua中，闭包统一被称为函数，而函数就是这里所指的函数原型，函数原型Proto是不可以直接被调用，只有和upvalue绑定后才变成了Lua VM能够解释的函数对象。

typedef struct Proto {

CommonHeader;

TValue \*k; /\* constants used by the function \*/

Instruction \*code;

struct Proto \*\*p; /\* functions defined inside the function \*/

int \*lineinfo; /\* map from opcodes to source lines \*/

struct LocVar \*locvars; /\* information about local variables \*/

TString \*\*upvalues; /\* upvalue names \*/

TString \*source;

int sizeupvalues;

int sizek; /\* size of `k' \*/

int sizecode;

int sizelineinfo;

int sizep; /\* size of `p' \*/

int sizelocvars;

int linedefined;

int lastlinedefined;

GCObject \*gclist;

lu\_byte nups; /\* number of upvalues \*/

lu\_byte numparams;

lu\_byte is\_vararg;

lu\_byte maxstacksize;

} Proto;

Lua的函数是有估计的，允许嵌套，所以在Proto结构中，能看到对内层Proto的引用p。成成可用的闭包的过程就是讲Proto与upvalue绑定的过程，为了避免重复生成不必要的闭包，当生成一次闭包对象后，都将被cache引用，下次再通过这个Proto生成闭包时，通过比较upvalue是否一致来决定是否复用，cache是一个弱引用，一旦GC过程中法线引用的闭包已经不存在了，cache将被置空。

**UpValue**

UpValue指在lua闭包生成的那一刻，与函数原型Proto绑定在一起的那些外部变量，这些变量原本是上一层函数的局部变量或upvalue，可以在上层返回后继续被闭包引用。

当上层函数没有退出时，调用刚生成的闭包，这个时候闭包更像一个普通的内嵌函数，外层函数的局部变阿玲只是数据栈上的一个普通变量，Lua VM用一个数据栈上的引用映射局部变量，内嵌函数可以通过数据栈自由访问它，而一旦外层函数返回，数据栈空间收缩，原有的局部变量消失了，这个时候闭包需要用其他方式访问这些upvalue，如果将数据栈上的每个变量都实现成一个独立的对象是没有必要的，尤其是数字、布尔类型，没有必要生成复杂对象。Lua的Upvalue的实现更像是C语言中的指针，它引用了另外一个对象，多个闭包可以共享同一个upvalue，犹如c语言中，可以有多个指针指向同一个结构体。

typedef struct UpVal {

CommonHeader;

TValue \*v; /\* points to stack or to its own value \*/

union {

TValue value; /\* the value (when closed) \*/

struct { /\* double linked list (when open) \*/

struct UpVal \*prev;

struct UpVal \*next;

} l;

} u;

} UpVal;

Upval结构直接用一个TValue指针引用了一个lua值变量

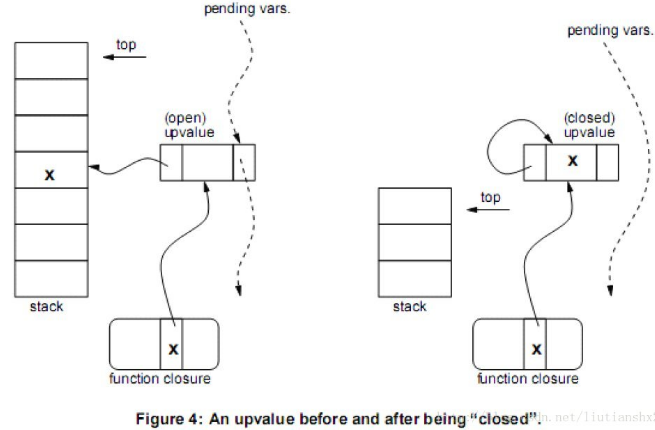
当被引用的变量还在数据栈上时，这个指针直接指向栈上的地址，这个upvalue被称为开放的

遍历当前所有open的upvalue利用的是当前lua\_State->openupval

所谓close状态upvalue是指：当upvalue应用的数据栈上的数据不再存在于数据栈时（通常是由申请局部变量的函数返回引起），需要把upvalue从lua\_State->openupval开放链表中剔除，并把其引用的数据栈上的变量存到另外一个安全的内存空间（UpVal结构中）

Upvalue的open和close状态在UpVal结构中，不需要用一个标记位区分，因为当upvalue close时，UpVal中的v指针一定指向结构体内部的value，所以可以使用下面的宏判断upvalue的open与close状态：

Define upisopen(up) ((up)->v!=&(up)->u.value)



**闭包**

Lua支持两种闭包：Lua语言实现和C语言实现。从外部数据类型来看，它们属于同一种类型（Lua\_TFUNCTION），从GC角度来看，它们同属于GCObject类型

typedef union Closure {

CClosure c;

LClosure l;

} Closure;

**Lua闭包**

Lua闭包仅仅是Proto和UpVal绑定的集合

typedef struct LClosure {

ClosureHeader;

struct Proto \*p;

UpVal \*upvals[1];

} LClosure;

Closure \*luaF\_newLclosure (lua\_State \*L, int nelems, Table \*e) {

Closure \*c = cast(Closure \*, luaM\_malloc(L, sizeLclosure(nelems)));

luaC\_link(L, obj2gco(c), LUA\_TFUNCTION);

c->l.isC = 0;

c->l.env = e;

c->l.nupvalues = cast\_byte(nelems);

while (nelems--) c->l.upvals[nelems] = NULL;

return c;

}

luaF\_newLclosure构造函数只绑定了Proto而没有初始化upvalue对象，这是因为构造LClosure有两种可能的途径

1. LClosure一般在Lua VM运行过程中被动态的构造出来，这时LClosure需要引用的upvalue都在当前的数据栈上，利用luaF\_findupvalue函数可以把数据栈上的TValue值转换为upvalue。luaF\_findupvalue的逻辑：现在当前的openvalue链表中寻找是否已经转换过，如果已存在则复用，反之就构造一个新的UpVal对象，并将它链接到openupval链表中。

线程的所有打开upvalue（在堆栈上的upvalue）都存放在L->openupvalue，当前函数的upvalue的位置由两种：

父函数的upvalue变量，此时只需要应用父函数的upvalue

父函数的local变量，那么现在L->openupvalue找，找到就引用，找不到就创建一个链接到L->openupvalue和g->uvhead，注意，这里创建的upvalue不会链接到allgc上