首先简单的例子：数字数组

一般情况下，lua中并不需要外部的数组，因为哈希表很好的实现了数组，但是对于非常大的数组而言，哈希表可能导致内存不足。因为对于每一个元素必须保存一个范性值，一个链接地址，加上一些以备将来增长的额外空间。在C中直接存储数字值不需要额外的空间，将比哈希表的实现方式节省50%的内存空间（个人认为估计还不止，每个table中数组元素的大小也比c中的大）。

Lua为这种情况专门提供了一个基本的类型：userdata。userdata提供了一个在lua中没有预定于操作的raw内存区域。

LUA\_API void \*lua\_newuserdata (lua\_State \*L, size\_t size)

Lua\_newuserdata函数按照指定的大小分配一块内存，将对应的Udata放到栈内，并返回内存块的地址。

LUA\_API void \*lua\_newuserdata (lua\_State \*L, size\_t size) {

Udata \*u;

lua\_lock(L);

luaC\_checkGC(L);

**u = luaS\_newudata(L, size, getcurrenv(L));**

setuvalue(L, L->top, u);

api\_incr\_top(L);

lua\_unlock(L);

return u + 1; **//这里返回u+1，也就是说将luaS\_newudata中分配内存的前半部分存放Udata，后半部分存放真正的userdata数据**

}

Udata \*luaS\_newudata (lua\_State \*L, size\_t s, Table \*e) {

Udata \*u;

if (s > MAX\_SIZET - sizeof(Udata))

luaM\_toobig(L);

printf("%d", sizeof(Udata));

u = cast(Udata \*, luaM\_malloc(L, s + sizeof(Udata))); **//分配本身userdata的内存+Udata内存**

u->uv.marked = luaC\_white(G(L)); /\* is not finalized \*/

u->uv.tt = LUA\_TUSERDATA;

u->uv.len = s;

u->uv.metatable = NULL;

u->uv.env = e;

/\* chain it on udata list (after main thread) \*/

u->uv.next = G(L)->mainthread->next;

G(L)->mainthread->next = obj2gco(u); **//将所有的userdata放在lua\_state之后（前插的方式）**

return u;

}

例子代码：

static int newArray(lua\_State \*L) {

int n = luaL\_checkint(L, 1);

size\_t nbytes = sizeof(NumArray)+(n - 1) \* sizeof(double);

NumArray \*a = (NumArray \*)lua\_newuserdata(L, nbytes);

a->size = n;

return 1;

}

static int setArray(lua\_State \*L) {

NumArray \*a = (NumArray \*)lua\_touserdata(L, 1);

int index = luaL\_checkint(L, 2);

double value = luaL\_checknumber(L, 3);

a->values[index - 1] = value;

return 0;

}

static int getArray(lua\_State \*L) {

NumArray \*a = (NumArray \*)lua\_touserdata(L, 1);

int index = luaL\_checkint(L, 2);

lua\_pushnumber(L, a->values[index - 1]);

return 1;

}

static luaL\_Reg array[] = {

{ "new", newArray },

{ "set", setArray },

{ "get", getArray },

};

luaL\_openlib(L, "array", array, 0);

lua代码：

local t = array.new(100)

array.set(t, 3, 100)

local num = array.get(t, 3)

print("num = ", num)

在上面的例子中，有一个很严重的问题，就是如果在array.set函数中，如果传递的userdata并不是NumArray，而是另外一个userdata，那么就会出问题，解决方案是给当前的userdata添加metetable，添加一个新函数：

const char \*array\_metatable = "array\_metatable";

**static NumArray \*checkArray(lua\_State \*L) {**

**void \*ud = luaL\_checkudata(L, 1, array\_metatable);**

**luaL\_argcheck(L, ud != NULL, 1, "array expected");**

**return (NumArray \*)ud;**

**}**

static int newArray(lua\_State \*L) {

int n = luaL\_checkint(L, 1);

size\_t nbytes = sizeof(NumArray)+(n - 1) \* sizeof(double);

NumArray \*a = (NumArray \*)lua\_newuserdata(L, nbytes);

**luaL\_getmetatable(L, array\_metatable);**

**lua\_setmetatable(L, -2);**

a->size = n;

return 1;

}

static int setArray(lua\_State \*L) {

**NumArray \*a = checkArray(L);**

int index = luaL\_checkint(L, 2);

double value = luaL\_checknumber(L, 3);

a->values[index - 1] = value;

return 0;

}

static int getArray(lua\_State \*L) {

**NumArray \*a = checkArray(L);**

int index = luaL\_checkint(L, 2);

lua\_pushnumber(L, a->values[index - 1]);

return 1;

}

**luaL\_newmetatable(L, array\_metatable);**

luaL\_openlib(L, "array", array, 0);

**访问面向对象的数据**

对于表来说，不管什么时候只要找不到给定的key，\_\_index原方法就会被调用，对于userdata而言，每次被访问的时候原方法都会被调用。因为userdata根本就没有任何key，注意a.size(a)等价于a:size()

local t = array.new(100)

local metatable = getmetatable(t)

metatable.\_\_index = metatable

metatable.set = array.set

metatable.get = array.get

metatable.size = array.size

t:set(3, 45)

local num = t:get(3)

local size = t:size()

print("num = ", num)

print("size = ", size)

**注意：我们不能从lua中设置userdata的metatable，但是我们在lua中可以访问metatable**

当我们在计算t:size的时候，lua对象在t中找不到size这个键值，因为对象是一个userdata，所以，lua试着从对象t的metatable的\_\_index域获取这个值

测试一下通过lua在c中创建一个userdata，然后c中保存了该userdata的指针，然后再c中释放该userdata内存，会导致lua后续崩溃

NumArray \*aa = NULL;

static int newArray(lua\_State \*L) {

int n = luaL\_checkint(L, 1);

size\_t nbytes = sizeof(NumArray) + (n - 1) \* sizeof(double);

aa = (NumArray \*)lua\_newuserdata(L, nbytes);

luaL\_getmetatable(L, array\_metatable);

lua\_setmetatable(L, -2);

aa->size = n;

return 1;

}

static int deleteArray(lua\_State \*L) {

NumArray \*a = checkArray(L);

**void \*p = (char \*)aa - sizeof(Udata);**

**printf("aa = %d\n", aa);**

**printf("apa = %d\n", p);**

**if (aa != NULL) free(p);**

return 0;

}

local t = array.new(5)

local metatable = getmetatable(t)

metatable.\_\_index = metatable

metatable.set = array.set

metatable.get = array.get

metatable.size = array.size

t:set(3, 45)

**array.delete(t)**

local num = t:get(3)

local size = t:size()

print("num = ", num)

print("size = ", size)



注意上面的释放，返回的aa实际上是在整个userdata起始位置+sizeof(Udata)，因此释放的时候应该找到起始位置

**LightUserdata**

一个lightuserdata是一个表示c指针的值（也就是一个void\*类型的值）。由于它是一个值，我们不能创建他们