<http://www.jb51.net/article/63835.htm>

<http://blog.csdn.net/zr339361504/article/details/52550209>

lua完全采用8位编码，lua字符串中的字符可以具有任何数值编码，包括数值0。也就是说，可以将任意二进制数据存储到一个字符串中。Lua的字符串是不可变的值，如果修改，实际上是新建一个字符串。

先看看TString的定义（lobject.h）

typedef union TString {

L\_Umaxalign dummy; /\* ensures maximum alignment for strings \*/

struct {

CommonHeader;

lu\_byte reserved;

unsigned int hash;

size\_t len;

} tsv;

} TString;

L\_Umaxalign dummy;

联合体TString中成员L\_Umaxalign dummy是用来保证与最大长度的C类型进行对齐，其定义如下：

typedef LUAI\_USER\_ALIGNMENT\_T L\_Umaxalign;

#define LUAI\_USER\_ALIGNMENT\_T union { double u; void \*s; long l; }

在其他可回收的对象（比如table）的实现中，也可看到这个联合体结构，这样做的目的是通过内存对齐，加快CPU访问内存的速度。

Reserved：这个变量用于标识这个字符串是否是lua虚拟机中的保留字符串，如果这个值为不为0，那么将不会再GC阶段被回收，而是一直保留在系统中。**只有lua语言中的关键字才会是保留字符串**

const char \*t1 = "nihao";

const char \*t2 = "else";

const char \*t3 = "function";

TString \*s1 = luaS\_newlstr(L, t1, strlen(t1));

TString \*s2 = luaS\_newlstr(L, t2, strlen(t2));

TString \*s3 = luaS\_newlstr(L, t3, strlen(t3));

cout << "[" << t1 << "]:reverse = " << (int)s1->tsv.reserved << endl;

cout << "[" << t2 << "]:reverse = " << (int)s2->tsv.reserved << endl;

cout << "[" << t3 << "]:reverse = " << (int)s3->tsv.reserved << endl;



原理：

lua\_State \*L = lua\_open();

#define lua\_open() luaL\_newstate()

LUALIB\_API lua\_State \*luaL\_newstate (void) {

lua\_State \*L = **lua\_newstate**(l\_alloc, NULL);

if (L) lua\_atpanic(L, &panic);

return L;

}

LUA\_API lua\_State \*lua\_newstate (lua\_Alloc f, void \*ud) {

int i;

lua\_State \*L;

global\_State \*g;

void \*l = (\*f)(ud, NULL, 0, state\_size(LG));

if (l == NULL) return NULL;

L = tostate(l);

g = &((LG \*)L)->g;

L->next = NULL;

L->tt = LUA\_TTHREAD;

g->currentwhite = bit2mask(WHITE0BIT, FIXEDBIT);

L->marked = luaC\_white(g);

set2bits(L->marked, FIXEDBIT, SFIXEDBIT);

preinit\_state(L, g);

g->frealloc = f;

g->ud = ud;

g->mainthread = L;

g->uvhead.u.l.prev = &g->uvhead;

g->uvhead.u.l.next = &g->uvhead;

g->GCthreshold = 0; /\* mark it as unfinished state \*/

g->strt.size = 0;

g->strt.nuse = 0;

g->strt.hash = NULL;

setnilvalue(registry(L));

luaZ\_initbuffer(L, &g->buff);

g->panic = NULL;

g->gcstate = GCSpause;

g->rootgc = obj2gco(L);

g->sweepstrgc = 0;

g->sweepgc = &g->rootgc;

g->gray = NULL;

g->grayagain = NULL;

g->weak = NULL;

g->tmudata = NULL;

g->totalbytes = sizeof(LG);

g->gcpause = LUAI\_GCPAUSE;

g->gcstepmul = LUAI\_GCMUL;

g->gcdept = 0;

for (i=0; i<NUM\_TAGS; i++) g->mt[i] = NULL;

if (luaD\_rawrunprotected(L, **f\_luaopen**, NULL) != 0) {

/\* memory allocation error: free partial state \*/

close\_state(L);

L = NULL;

}

else

luai\_userstateopen(L);

return L;

}

static void f\_luaopen (lua\_State \*L, void \*ud) {

global\_State \*g = G(L);

UNUSED(ud);

stack\_init(L, L); /\* init stack \*/

sethvalue(L, gt(L), luaH\_new(L, 0, 2)); /\* table of globals \*/

sethvalue(L, registry(L), luaH\_new(L, 0, 2)); /\* registry \*/

luaS\_resize(L, MINSTRTABSIZE); /\* initial size of string table \*/

luaT\_init(L);

**luaX\_init(L);**

luaS\_fix(luaS\_newliteral(L, MEMERRMSG));

g->GCthreshold = 4\*g->totalbytes;

}

void luaX\_init (lua\_State \*L) {

int i;

for (i=0; i<NUM\_RESERVED; i++) {

TString \*ts = luaS\_new(L, **luaX\_tokens[i]**);

luaS\_fix(ts); /\* reserved words are never collected \*/

lua\_assert(strlen(luaX\_tokens[i])+1 <= TOKEN\_LEN);

ts->tsv.reserved = **cast\_byte(i+1)**; /\* reserved word \*/

}

}

const char \*const luaX\_tokens [] = {

"and", "break", "do", "else", "elseif",

"end", "false", "for", "function", "if",

"in", "local", "nil", "not", "or", "repeat",

"return", "then", "true", "until", "while",

"..", "...", "==", ">=", "<=", "~=",

"<number>", "<name>", "<string>", "<eof>",

NULL

};

实际上也就是在open\_state的时候已经将保留字符串设置了

注意：CommonHeader中的next指针是用来存放冲突字符串

联合体的tsv是真正用来存储字符串的

lu\_byte reserved用来记录这个字符串是否为保留字，对于长字符串，可以用于惰性求hash值

unsigned int hash成员是字符串对应的hash值

size\_t len用来表示字符串的长度

在lua5.2.1之前，不区分字符串长和短的字符串，所有的字符串保存在一个全局的hash表中，对于lua虚拟机来说，相同的字符串只有一份数据，从lua5.2.1开始，只是把短的字符串（当前定义是长度小于40）放在全局hash表中，而字符串都是独立生成，同时在计算hash值时，引入一个随机种子，这样做的目的是防止hash dos—攻击者构造出非常多相同hash值的不同字符串，从而降低lua从外部压入字符串进去全局字符串hash表的效率

下面的源码基于5.2.3，区分了短字符串和长字符串

TString \*luaS\_newlstr (lua\_State \*L, const char \*str, size\_t l) {

if (l <= LUAI\_MAXSHORTLEN) /\* short string? \*/

return internshrstr(L, str, l);

else {

if (l + 1 > (MAX\_SIZET - sizeof(TString))/sizeof(char))

luaM\_toobig(L);

return createstrobj(L, str, l, LUA\_TLNGSTR, G(L)->seed, NULL);

}

}

/\* Variant tags for strings \*/

#define LUA\_TSHRSTR (LUA\_TSTRING | (0 << 4)) /\* short strings \*/

#define LUA\_TLNGSTR (LUA\_TSTRING | (1 << 4)) /\* long strings \*/

实际上会发现短字符串的internshrstr函数和下面的5.1.4的思想一样，放在全局hash表中，对于长字符串，实际上是直接放在了GC链表中，在新建的时候不计算hash值，也不保证唯一性，使用extra字段来表示是否计算hash

static TString \*createstrobj (lua\_State \*L, const char \*str, size\_t l,

int tag, unsigned int h, GCObject \*\*list) {

TString \*ts;

size\_t totalsize; /\* total size of TString object \*/

totalsize = sizeof(TString) + ((l + 1) \* sizeof(char));

ts = &luaC\_newobj(L, tag, totalsize, list, 0)->ts;

ts->tsv.len = l;

ts->tsv.hash = h;

ts->tsv.extra = 0;

memcpy(ts+1, str, l\*sizeof(char));

((char \*)(ts+1))[l] = '\0'; /\* ending 0 \*/

return ts;

}

（lgc.c）

GCObject \*luaC\_newobj (lua\_State \*L, int tt, size\_t sz, GCObject \*\*list,

int offset) {

global\_State \*g = G(L);

char \*raw = cast(char \*, luaM\_newobject(L, novariant(tt), sz));

GCObject \*o = obj2gco(raw + offset);

**if (list == NULL)**

**list = &g->allgc; /\* standard list for collectable objects \*/**

gch(o)->marked = luaC\_white(g);

gch(o)->tt = tt;

**gch(o)->next = \*list; // 这里相当于前向插入**

**\*list = o;**

return o;

}

对于字符串比较，首先比较类型，若是不同类型字符串，则肯定不相同，然后区分短字符串和长字符串，对于短字符串，若两者

针值相等，则相同，否则不相同；对应长字符串，则首先比较指针值，若不同，则比较长度值和内容逐字符比较。

int luaS\_eqstr (TString \*a, TString \*b) {

return (a->tsv.tt == b->tsv.tt) &&

(a->tsv.tt == LUA\_TSHRSTR ? eqshrstr(a, b) : luaS\_eqlngstr(a, b));

}

#define eqshrstr(a,b) check\_exp((a)->tsv.tt == LUA\_TSHRSTR, (a) == (b))

int luaS\_eqlngstr (TString \*a, TString \*b) {

size\_t len = a->tsv.len;

lua\_assert(a->tsv.tt == LUA\_TLNGSTR && b->tsv.tt == LUA\_TLNGSTR);

return (a == b) || /\* same instance or... \*/

((len == b->tsv.len) && /\* equal length and ... \*/

(memcmp(getstr(a), getstr(b), len) == 0)); /\* equal contents \*/

}

**Lua中字符串hash算法用的是JSHash，关于字符串的各种Hash算法，见：**

<https://www.byvoid.com/zhs/blog/string-hash-compare>

下面的源码都是基于5.1.4，没有区分短字符串和长字符串

创建新字符串：

TString \*luaS\_newlstr (lua\_State \*L, const char \*str, size\_t l) {

GCObject \*o;

unsigned int h = cast(unsigned int, l); /\* seed \*/

size\_t step = (l>>5)+1; /\* if string is too long, don't hash all its chars \*/

size\_t l1;

for (l1=l; l1>=step; l1-=step) /\* compute hash 这里保证不会计算很多次\*/

h = h ^ ((h<<5)+(h>>2)+cast(unsigned char, str[l1-1]));

for (o = G(L)->strt.hash[lmod(h, G(L)->strt.size)]; // 寻找是否存在该字符串#define G(L) (L->l\_G)

o != NULL;

o = o->gch.next) {

TString \*ts = rawgco2ts(o);

if (ts->tsv.len == l && (memcmp(str, getstr(ts), l) == 0)) {

/\* string may be dead \*/

if (isdead(G(L), o)) changewhite(o);

return ts;

}

}

return newlstr(L, str, l, h); /\* not found \*/

}

static TString \*newlstr (lua\_State \*L, const char \*str, size\_t l,

unsigned int h) {

TString \*ts;

stringtable \*tb;

if (l+1 > (MAX\_SIZET - sizeof(TString))/sizeof(char))

luaM\_toobig(L);

ts = cast(TString \*, luaM\_malloc(L, (l+1)\*sizeof(char)+sizeof(TString)));

ts->tsv.len = l;

ts->tsv.hash = h;

ts->tsv.marked = luaC\_white(G(L));

ts->tsv.tt = LUA\_TSTRING;

ts->tsv.reserved = 0;

memcpy(ts+1, str, l\*sizeof(char)); // TString+char[] 将字符串拷贝到TString之后

((char \*)(ts+1))[l] = '\0'; /\* ending 0 \*/

tb = &G(L)->strt;

h = lmod(h, tb->size);

ts->tsv.next = tb->hash[h]; /\* chain new entry \*/ // 新元素指向旧元素，即前向插入，实际上也就是TString中的CommonHeader中的next

tb->hash[h] = obj2gco(ts);

tb->nuse++; // 计数，存储当前全局hash表中有多少个字符串

if (tb->nuse > cast(lu\_int32, tb->size) && tb->size <= MAX\_INT/2)

luaS\_resize(L, tb->size\*2); /\* too crowded \*/

return ts;

}

void luaS\_resize (lua\_State \*L, int newsize) {

GCObject \*\*newhash;

stringtable \*tb;

int i;

if (G(L)->gcstate == GCSsweepstring)

return; /\* cannot resize during GC traverse \*/

newhash = luaM\_newvector(L, newsize, GCObject \*);

tb = &G(L)->strt;

for (i=0; i<newsize; i++) newhash[i] = NULL;

/\* rehash \*/

for (i=0; i<tb->size; i++) {

GCObject \*p = tb->hash[i];

while (p) { /\* for each node in the list \*/

GCObject \*next = p->gch.next; /\* save next \*/

unsigned int h = gco2ts(p)->hash;

int h1 = lmod(h, newsize); /\* new position \*/

lua\_assert(cast\_int(h%newsize) == lmod(h, newsize));

p->gch.next = newhash[h1]; /\* chain it \*/

newhash[h1] = p;

p = next;

}

}

luaM\_freearray(L, tb->hash, tb->size, TString \*);

tb->size = newsize;

tb->hash = newhash;

}

全局hash表stringtable

typedef struct stringtable {

GCObject \*\*hash;

lu\_int32 nuse; /\* number of elements \*/

int size; // hash表的大小，为2^n

} stringtable;

这个全局的stringtable存放在lua\_state的global\_State中：

struct lua\_State {

CommonHeader;

lu\_byte status;

StkId top; /\* first free slot in the stack \*/

StkId base; /\* base of current function \*/

global\_State \*l\_G;

CallInfo \*ci; /\* call info for current function \*/

const Instruction \*savedpc; /\* `savedpc' of current function \*/

StkId stack\_last; /\* last free slot in the stack \*/

StkId stack; /\* stack base \*/

CallInfo \*end\_ci; /\* points after end of ci array\*/

CallInfo \*base\_ci; /\* array of CallInfo's \*/

int stacksize;

int size\_ci; /\* size of array `base\_ci' \*/

unsigned short nCcalls; /\* number of nested C calls \*/

unsigned short baseCcalls; /\* nested C calls when resuming coroutine \*/

lu\_byte hookmask;

lu\_byte allowhook;

int basehookcount;

int hookcount;

lua\_Hook hook;

TValue l\_gt; /\* table of globals \*/

TValue env; /\* temporary place for environments \*/

GCObject \*openupval; /\* list of open upvalues in this stack \*/

GCObject \*gclist;

struct lua\_longjmp \*errorJmp; /\* current error recover point \*/

ptrdiff\_t errfunc; /\* current error handling function (stack index) \*/

};

在看看global\_State：

typedef struct global\_State {

stringtable strt; /\* hash table for strings \*/

lua\_Alloc frealloc; /\* function to reallocate memory \*/

void \*ud; /\* auxiliary data to `frealloc' \*/

lu\_byte currentwhite;

lu\_byte gcstate; /\* state of garbage collector \*/

int sweepstrgc; /\* position of sweep in `strt' \*/

GCObject \*rootgc; /\* list of all collectable objects \*/

GCObject \*\*sweepgc; /\* position of sweep in `rootgc' \*/

GCObject \*gray; /\* list of gray objects \*/

GCObject \*grayagain; /\* list of objects to be traversed atomically \*/

GCObject \*weak; /\* list of weak tables (to be cleared) \*/

GCObject \*tmudata; /\* last element of list of userdata to be GC \*/

Mbuffer buff; /\* temporary buffer for string concatentation \*/

lu\_mem GCthreshold;

lu\_mem totalbytes; /\* number of bytes currently allocated \*/

lu\_mem estimate; /\* an estimate of number of bytes actually in use \*/

lu\_mem gcdept; /\* how much GC is `behind schedule' \*/

int gcpause; /\* size of pause between successive GCs \*/

int gcstepmul; /\* GC `granularity' \*/

lua\_CFunction panic; /\* to be called in unprotected errors \*/

TValue l\_registry;

struct lua\_State \*mainthread;

UpVal uvhead; /\* head of double-linked list of all open upvalues \*/

struct Table \*mt[NUM\_TAGS]; /\* metatables for basic types \*/

TString \*tmname[TM\_N]; /\* array with tag-method names \*/

} global\_State;

Lua\_state ----> global\_state ----> strt(stringtable)



