Lua 源码欣赏

云风

2012 年 7 月 17 日

注

这是这本书中其中两章的草稿。我不打算按顺序来编写这本书，而打算以独立章节的形式分开完成，到最后再做统一的调整。由于业余时间不多，所以产出也不固定。

目录

第一章　String 的实现

Lua 中的字符串可以包含任何 8 位字符，包括了 C 语言中标示字符串结束的 \0 。它以带长度的内存块的形式在内部保存字符串，同时在和 C 语言做交互时，又能保证在每个内部储存的字符串末尾添加 \0 以兼容 C 库函数。这使得 Lua 的字符串应用范围相当广。Lua 管理及操作字符串的方式和 C 语言不太相同，通过阅读其实现代码，可以加深对 Lua 字符串的理解，能更为高效的使用它。

1.1　数据结构

从 Lua 5.2.1 开始，字符串保存在 Lua 状态机内有两种内部形式，短字符串及长字符串。

源代码 1.1: lobject.h: variant string

```
55 /*
56 ** LUA_TSTRING variants */
57 #define LUA_TSHRSTR      (LUA_TSTRING | (0 << 4))   /*
       short strings */
58 #define LUA_TLNGSTR      (LUA_TSTRING | (1 << 4))   /*
       long strings */
```

这个小类型区分放在类型字节的高四位，所以为外部 API 所不可见。对于最终用户来说，他们只见到 LUA_TSTRING 一种类型。区分长短字符串的界限由定义在 luaconf.h 中的宏 LUAI_MAXSHORTLEN 来决定。其默认设置为 40 字节。由 Lua 的实现决定了，LUAI_MAXSHORTLEN 不

可以设置少于 10 字节[1]。

字符串一旦创建，则不可被改写。Lua 的值对象若为字符串类型，则以引用方式存在。属于需被垃圾收集器管理的对象。也就是说，一个字符串一旦没有被任何地方引用就可以回收它。

字符串类型 TString 定义在 lobject.h 里。

源代码 1.2: lobject.h: tstring

```
413 typedef union TString {
414   L_Umaxalign dummy;  /* ensures maximum alignment for
        strings */
415   struct {
416     CommonHeader;
417     lu_byte extra;  /* reserved words for short
          strings; "has hash" for longs */
418     unsigned int hash;
419     size_t len;  /* number of characters in string */
420   } tsv;
421 } TString;
```

除了用于 GC 的 CommonHeader 外，有 extra hash 和 len 三个域。extra 用来记录辅助信息[2] 记录字符串的 hash 可以用来加快字符串的匹配和查找；由于 Lua 并不以 \0 结尾来识别字符串的长度，故需要一个 len 域来记录其长度。

字符串的数据内容并没有被分配独立一块内存来保存，而是直接最加在 TString 结构的后面。用 getstr 这个宏就可以取到实际的 C 字符串指针。

源代码 1.3: lobject.h: getstr

```
425 #define getstr(ts)  cast(const char *, (ts) + 1)
```

所有短字符串均被存放在全局表（global_State）的 strt 域中。strt 是 string table 的简写，它是一个哈希表。

[1]元方法名和保留字必须是短字符串，所以短字符串长度不得短于最长的元方法名 __newindex 和保留字 function 。

[2]对于短字符串 extra 用来记录这个字符串是否为保留字，这个标记用于词法分析器对保留字的快速判断；对于长字符串，可以用于惰性求哈希值。

源代码 1.4: lstate.h: stringtable

```
59 typedef struct stringtable {
60   GCObject **hash;
61   lu_int32 nuse;  /* number of elements */
62   int size;
63 } stringtable;
```

相同的短字符串在同一个 Lua State 中只存在唯一一份，这被称为字符串的内部化[3]。

长字符串则独立存放，从外部压入一个长字符串时，简单复制一遍字符串，并不立刻计算其 hash 值，而是标记一下 extra 域。直到需要对字符串做键匹配时，才惰性计算 hash 值，加快以后的键比较过程[4]。

1.1.1 Hash DoS

在 Lua 5.2.0 之前，字符串是不分长短一律内部化后放在字符串表中的。

对于长字符串，为了加快内部化的过程，计算长字符串哈希值是跳跃进行的。下面是 Lua 5.2.0 中的字符串哈希值计算代码：

```
1   unsigned int h = cast(unsigned int, l);  /* seed */
2   size_t step = (l>>5)+1;  /* if string is too long,
        don't hash all its chars */
3   size_t l1;
4   for (l1=l; l1>=step; l1-=step)  /* compute hash */
5     h = h ^ ((h<<5)+(h>>2)+cast(unsigned char, str[l1
          -1]));
```

Lua 5.2.0 发布后不久，有人在邮件列表中提出，Lua 的这个设计有可能对其给于 Hash DoS 攻击的机会[5]。攻击者可以轻易构造出上千万拥有相同哈希值的不同字符串，以此数十倍的降低 Lua 从外部压入字符串进入内

[3]合并相同的字符串可以大量减少内存占用，缩短比较字符串的时间。因为相同的字符串只需要保存一份在内存中，当用这个字符串做键匹配时，比较字符串只需要比较地址是否相同就够了，而不必逐字节比较。

[4]关于长字符串惰性求哈希值的过程，参见 2.2.1 节。

[5]Real-World Impact of Hash DoS in Lua. http://lua-users.org/lists/lua-l/2012-01/msg00497.html

部字符串表的效率[4]。当 Lua 用于大量依赖字符串处理的诸如 HTTP 服务的处理时，输入的字符串不可控制，很容易被人恶意利用。

Lua 5.2.1 为了解决这个问题，把长字符串独立出来。大量文本处理中的输入的字符串不再通过哈希内部化进入全局字符串表。同时，使用了一个随机种子用于哈希值的计算，使攻击者无法轻易构造出拥有相同哈希值的不同字符串。

源代码 1.5: lstring.c: stringhash

```
51 unsigned int luaS_hash (const char *str, size_t l,
       unsigned int seed) {
52   unsigned int h = seed ^ l;
53   size_t l1;
54   size_t step = (l >> LUAI_HASHLIMIT) + 1;
55   for (l1 = l; l1 >= step; l1 -= step)
56     h = h ^ ((h<<5) + (h>>2) + cast_byte(str[l1 - 1]))
         ;
57   return h;
58 }
```

这个随机种子是在 Lua State 创建时放在全局表中的，它利用了构造状态机的内存地址随机性以及用户可配置的一个随机量同时来决定[6]。

源代码 1.6: lstate.c: makeseed

```
80 /*
81 ** Compute an initial seed as random as possible. In
       ANSI, rely on
82 ** Address Space Layout Randomization (if present) to
       increase
83 ** randomness..
84 */
85 #define addbuff(b,p,e) \
```

[6]用户可以在 luaconf.h 中配置 luai_makeseed 定义自己的随机方法，默认是利用 time 函数获取时间构造种子。值得注意的是，使用系统当前时间来构造随机种子这种行为，有可能给调试带来一些困扰。因为字符串 hash 值的不同，会让程序每次运行过程中的内部布局有一些细微变化。好在字符串池使用的是开散列算法（参见1.2.2），这个影响非常的小。但如果你希望让嵌入 lua 的程序，每次运行都严格一致，最好自己定义 luai_makeseed 函数。

```
86    { size_t t = cast(size_t, e); \
87      memcpy(buff + p, &t, sizeof(t)); p += sizeof(t); }
88
89  static unsigned int makeseed (lua_State *L) {
90    char buff[4 * sizeof(size_t)];
91    unsigned int h = luai_makeseed();
92    int p = 0;
93    addbuff(buff, p, L);  /* heap variable */
94    addbuff(buff, p, &h);  /* local variable */
95    addbuff(buff, p, luaO_nilobject);  /* global
          variable */
96    addbuff(buff, p, &lua_newstate);  /* public function
          */
97    lua_assert(p == sizeof(buff));
98    return luaS_hash(buff, p, h);
99  }
```

1.2 实现

1.2.1 字符串比较

比较两个字符串是否相同，需要区分长短字符串。子类型不同自然不是相同的字符串。

源代码 1.7: lstring.c: eqstr

```
45  int luaS_eqstr (TString *a, TString *b) {
46    return (a->tsv.tt == b->tsv.tt) &&
47            (a->tsv.tt == LUA_TSHRSTR ? eqshrstr(a, b) :
                luaS_eqlngstr(a, b));
48  }
```

长字符串比较，当长度不同时，自然是不同的字符串。而长度相同时，则需要逐字节比较：

源代码 1.8: lstring.c: eqlngstr

```
33 int luaS_eqlngstr (TString *a, TString *b) {
34   size_t len = a->tsv.len;
35   lua_assert(a->tsv.tt == LUA_TLNGSTR && b->tsv.tt ==
       LUA_TLNGSTR);
36   return (a == b) ||  /* same instance or... */
37     ((len == b->tsv.len) &&  /* equal length and ...
         */
38      (memcmp(getstr(a), getstr(b), len) == 0));  /*
          equal contents */
39 }
```

短字符串因为经过的内部化，所以不必比较字符串内容，而仅需要比较对象地址即可。Lua 用一个宏来高效的实现它：

源代码 1.9: lstring.h: eqshrstr

```
34 #define eqshrstr(a,b)    check_exp((a)->tsv.tt ==
     LUA_TSHRSTR, (a) == (b))
```

1.2.2 短字符串的内部化

所有的短字符串都被内部化放在全局的字符串表中。这张表是用一个哈希表来实现[7]。

源代码 1.10: lstring.c: internshrstr

```
133 static TString *internshrstr (lua_State *L, const char
      *str, size_t l) {
134   GCObject *o;
135   global_State *g = G(L);
136   unsigned int h = luaS_hash(str, l, g->seed);
137   for (o = g->strt.hash[lmod(h, g->strt.size)];
138        o != NULL;
```

[7]字符串表和 Lua 表中的哈希部分（源代码2.4）不同，需求更简单。它不必迭代。全局只有一个，不用太考虑空间利用率。所以这里使用的是开散列算法。开散列也被称为 Separate chaining [6]。即，将哈系值相同的对象串在分别独立的链表中，实现起来更为简单。

```
139         o = gch(o)->next) {
140       TString *ts = rawgco2ts(o);
141       if (h == ts->tsv.hash &&
142           ts->tsv.len == l &&
143           (memcmp(str, getstr(ts), l * sizeof(char)) ==
                0)) {
144         if (isdead(G(L), o))  /* string is dead (but was
                not collected yet)? */
145           changewhite(o);  /* resurrect it */
146         return ts;
147       }
148     }
149     return newshrstr(L, str, l, h);  /* not found;
          create a new string */
150   }
```

这是一个开散列的哈希表实现。一个字符串被放入字符串表的时候，先检查一下表中有没有相同的字符串。如果有，则复用已有的字符串；没有则创建一个新的。碰到哈希值相同的字符串，简单的串在同一个哈希位的链表上即可。

注意 144-145 行，这里需要检查表中的字符串是否是死掉的字符串。这是因为 Lua 的垃圾收集过程是分步完成的。而向字符串池添加新字符串在任何步骤之间都可能发生。有可能在标记完字符串后发现有些字符串没有任何引用，但在下个步骤中又产生了相同的字符串导致这个字符串复活。

当哈希表中字符串的数量 (nuse 域) 超过预定容量 (size 域) 时。可以预计 hash 冲突必然发生。这个时候就调用 luaS_resize 方法把字符串表的哈希链表数组扩大，重新排列所有字符串的位置。这个过程和 Lua 表（参见源代码2.5）的处理类似，不过里面涉及垃圾收集的一些细节，不在本节分析。

源代码 1.11: lstring.c: resize

```
64  void luaS_resize (lua_State *L, int newsize) {
65    int i;
```

```
66    stringtable *tb = &G(L)->strt;
67    /* cannot resize while GC is traversing strings */
68    luaC_runtilstate(L, ~bitmask(GCSsweepstring));
69    if (newsize > tb->size) {
70      luaM_reallocvector(L, tb->hash, tb->size, newsize,
            GCObject *);
71      for (i = tb->size; i < newsize; i++) tb->hash[i] =
            NULL;
72    }
73    /* rehash */
74    for (i=0; i<tb->size; i++) {
75      GCObject *p = tb->hash[i];
76      tb->hash[i] = NULL;
77      while (p) {  /* for each node in the list */
78        GCObject *next = gch(p)->next;  /* save next */
79        unsigned int h = lmod(gco2ts(p)->hash, newsize);
              /* new position */
80        gch(p)->next = tb->hash[h];  /* chain it */
81        tb->hash[h] = p;
82        resetoldbit(p);  /* see MOVE OLD rule */
83        p = next;
84      }
85    }
86    if (newsize < tb->size) {
87      /* shrinking slice must be empty */
88      lua_assert(tb->hash[newsize] == NULL && tb->hash[
            tb->size - 1] == NULL);
89      luaM_reallocvector(L, tb->hash, tb->size, newsize,
            GCObject *);
90    }
91    tb->size = newsize;
92  }
```

每在 Lua 状态机内部创建一个字符串，都会按 C 风格字符串存放，以兼容 C 接口。即在字符串的末尾加上一个 \0 ，这在 lstring.c 的 108 行可以见到。这样不违背 Lua 自己用内存块加长度的方式储存字符串的规则，在把 Lua 字符串传递出去和 C 语言做交互时，又不必做额外的转换。

源代码 1.12: lstring.c: createstrobj

```
98  static TString *createstrobj (lua_State *L, const char
        *str, size_t l,
99                                int tag, unsigned int h,
                                      GCObject **list) {
100   TString *ts;
101   size_t totalsize;  /* total size of TString object
        */
102   totalsize = sizeof(TString) + ((l + 1) * sizeof(char
        ));
103   ts = &luaC_newobj(L, tag, totalsize, list, 0)->ts;
104   ts->tsv.len = l;
105   ts->tsv.hash = h;
106   ts->tsv.extra = 0;
107   memcpy(ts+1, str, l*sizeof(char));
108   ((char *)(ts+1))[l] = '\0';  /* ending 0 */
109   return ts;
110 }
```

1.3 Userdata 的构造

Userdata 在 Lua 中并没有太特别的地方，在储存形式上和字符串相同。可以看成是拥有独立元表，不被内部化处理，也不需要追加 \0 的字符串。在实现上，只是对象结构从 TString 换成了 UData 。所以实现代码也被放在 lstring.c 中，其 api 也以 luaS 开头。

源代码 1.13: lobject.h: udata

```
434 typedef union Udata {
```

```
435   L_Umaxalign dummy;  /* ensures maximum alignment for
          'local' udata */
436   struct {
437     CommonHeader;
438     struct Table *metatable;
439     struct Table *env;
440     size_t len;  /* number of bytes */
441   } uv;
442 } Udata;
```

Userdata 的数据部分和字符串一样，紧接在这个结构后面，并没有单独分配内存。Userdata 的构造函数如下：

源代码 1.14: lstring.c: newudata

```
175 Udata *luaS_newudata (lua_State *L, size_t s, Table *e
      ) {
176   Udata *u;
177   if (s > MAX_SIZET - sizeof(Udata))
178     luaM_toobig(L);
179   u = &luaC_newobj(L, LUA_TUSERDATA, sizeof(Udata) + s
        , NULL, 0)->u;
180   u->uv.len = s;
181   u->uv.metatable = NULL;
182   u->uv.env = e;
183   return u;
184 }
```

第二章　Table 的实现

Lua 使用 table 作为统一的数据结构。在一次对 Lua 作者的采访中[1]，他们这样说道：

> ***Roberto***：从我的角度，灵感来自于 *VDM*（一个主要用于软件规范的形式化方法），当我们开始创建 *Lua* 时，有一些东西引起了我的兴趣。*VDM* 提供三种数据聚合的方式：*set*、*sequence* 和 *map*。不过，*set* 和 *sequence* 都很容易用 *map* 来表达，因此我有了用 *map* 作为统一结构的想法。*Luiz* 也有他自己的原因。
>
> ***Luiz***：没错，我非常喜欢 *AWK*，特别是它的联合数组。

2.1　数据结构

用 table 来表示 Lua 中的一切数据结构是 Lua 语言的一大特色。为了效率，Lua 的官方实现，又把 table 的储存分为数组部分和哈希表部分。数组部分从 1 开始作整数数字索引。这可以提供紧凑且高效的随机访问。而不能被储存在数组部分的数据全部放在哈希表中，唯一不能做哈希键值的是 nil，这个限制可以帮助我们发现许多运行期错误。Lua 的哈希表有一个高效的实现，几乎可以认为操作哈希表的时间复杂度为 O(1)。

这样分开的储存方案，对使用者是完全透明的，并没有强求 Lua 语言的实现一定要这样分开。但在 lua 的基础库中，提供了 pairs 和 ipairs 两个不同的 api 来实现两种不同方式的 table 遍历方案；用 # 对 table 取长度时，也被定义成和整数下标有关，而非整个 table 的尺寸。在 Lua 的 C

[1] 见《Masterminds of Programming: Conversations with the Creators of Major Programming Languages》的第 7 章，中译名《编程之魂》。笔者曾做过这一章的翻译：http://blog.codingnow.com/2010/06/masterminds_of_programming_7_lua.html

API 中，有独立的 api 来操作数组的整数下标部分。也就是说，Lua 有大量的特性，提供了对整数下标的高效访问途径。

Table 的内部数据结构被定义在 lobject.h 中，

源代码 2.1: lobject.h: table

```
544 /*
545 ** Tables
546 */
547
548 typedef union TKey {
549   struct {
550     TValuefields;
551     struct Node *next;  /* for chaining */
552   } nk;
553   TValue tvk;
554 } TKey;
555
556
557 typedef struct Node {
558   TValue i_val;
559   TKey i_key;
560 } Node;
561
562
563 typedef struct Table {
564   CommonHeader;
565   lu_byte flags;  /* 1<<p means tagmethod(p) is not
        present */
566   lu_byte lsizenode;  /* log2 of size of 'node' array
        */
567   struct Table *metatable;
568   TValue *array;  /* array part */
569   Node *node;
570   Node *lastfree;  /* any free position is before this
```

```
        position */
571     GCObject *gclist;
572     int sizearray;  /* size of 'array' array */
573   } Table;
574
575
576
577   /*
578   ** 'module' operation for hashing (size is always a
        power of 2)
579   */
580   #define lmod(s,size) \
581           (check_exp((size&(size-1))==0, (cast(int, (s)
              & ((size)-1)))))
582
583
584   #define twoto(x)          (1<<(x))
585   #define sizenode(t)       (twoto((t)->lsizenode))
```

Table 的数组部分被储存在 TValue *array 中，其长度信息存于 int sizearray 。哈希表储存在 Node *node ，哈希表的大小用 lu_byte lsizenode 表示，由于哈希表的大小一定为 2 的整数次幂，所以这里的 lsizenode 表示的是幂次，而不是实际大小。

每个 table 结构，最多会由三块连续内存构成。一个 Table 结构，一块存放了连续整数索引的数组，和一块大小为 2 的整数次幂的哈希表。哈希表的最小尺寸为 2 的 0 次幂，也就是 1 。为了减少空表的维护成本，Lua 在这里做了一点优化。它定义了一个不可改写的空哈希表：dummynode 。让空表被初始化时，node 域指向这个 dummy 节点。它虽然是一个全局变量，但因为对其访问是只读的，所以不会引起线程安全问题。[2]

源代码 2.2: ltable.c: dummynode

[2]不当的链接 lua 库，有可能造成错误。如果你错误的链接了两份相同的 Lua 库实现到你的进程中，大多数情况下，代码可以安全的运行。但在清除空 table 时，会因为无法正确的识别 dummynode 而程序崩溃。建议在 Lua 扩展库的实现时调用 luaL_checkversion 做一下检查。更详细的分析参见笔者的一篇 blog http://blog.codingnow.com/2012/01/lua_link_bug.html 。

```
67 #define dummynode                    (&dummynode_)
68
69 #define isdummy(n)                   ((n) == dummynode)
70
71 static const Node dummynode_ = {
72   {NILCONSTANT},  /* value */
73   {{NILCONSTANT, NULL}}  /* key */
74 };
```

阅读 luaH_new 和 luaH_free 两个 api 的实现，可以了解这一层次的数据结构。

源代码 2.3: ltable.c: new

```
368 Table *luaH_new (lua_State *L) {
369   Table *t = &luaC_newobj(L, LUA_TTABLE, sizeof(Table)
          , NULL, 0)->h;
370   t->metatable = NULL;
371   t->flags = cast_byte(~0);
372   t->array = NULL;
373   t->sizearray = 0;
374   setnodevector(L, t, 0);
375   return t;
376 }
377
378
379 void luaH_free (lua_State *L, Table *t) {
380   if (!isdummy(t->node))
381     luaM_freearray(L, t->node, cast(size_t, sizenode(t
          )));
382   luaM_freearray(L, t->array, t->sizearray);
383   luaM_free(L, t);
384 }
```

其中 setnodevector 用来初始化哈希表部分。内存管理部分则使用了 luaM 相关 api 。

2.2 算法

Table 按照 lua 语言的定义，需要实现四种基本操作：读、写、迭代和获取长度。lua 中并没有删除操作，而仅仅是把对应键位的值设置为 nil 。

写操作被实现为查询已有键位，若不存在则创建新键。得到键位后，写入操作就是一次赋值。所以，在 table 模块中，实际实现的基本操作为：创建、查询、迭代和获取长度。

创建操作的 api 为 luaH_newkey ，阅读它的实现就能对整个 table 有一个全面的认识。它只负责在哈希表中创建出一个不存在的键，而不关数组部分的工作。因为 table 的数组部分操作和 C 语言数组没有什么不同，不需要特别处理。

源代码 2.4: ltable.c: newkey

```
405 TValue *luaH_newkey (lua_State *L, Table *t, const
      TValue *key) {
406   Node *mp;
407   if (ttisnil(key)) luaG_runerror(L, "table index is
        nil");
408   else if (ttisnumber(key) && luai_numisnan(L, nvalue(
        key)))
409     luaG_runerror(L, "table index is NaN");
410   mp = mainposition(t, key);
411   if (!ttisnil(gval(mp)) || isdummy(mp)) {  /* main
        position is taken? */
412     Node *othern;
413     Node *n = getfreepos(t);  /* get a free place */
414     if (n == NULL) {  /* cannot find a free place? */
415       rehash(L, t, key);  /* grow table */
416       /* whatever called 'newkey' take care of TM
            cache and GC barrier */
417       return luaH_set(L, t, key);  /* insert key into
            grown table */
418     }
419     lua_assert(!isdummy(n));
```

```
420     othern = mainposition(t, gkey(mp));
421     if (othern != mp) {  /* is colliding node out of
            its main position? */
422       /* yes; move colliding node into free position
            */
423       while (gnext(othern) != mp) othern = gnext(
            othern);  /* find previous */
424       gnext(othern) = n;  /* redo the chain with 'n'
            in place of 'mp' */
425       *n = *mp;  /* copy colliding node into free pos.
             (mp->next also goes) */
426       gnext(mp) = NULL;  /* now 'mp' is free */
427       setnilvalue(gval(mp));
428     }
429     else {  /* colliding node is in its own main
            position */
430       /* new node will go into free position */
431       gnext(n) = gnext(mp);  /* chain new position */
432       gnext(mp) = n;
433       mp = n;
434     }
435   }
436   setobj2t(L, gkey(mp), key);
437   luaC_barrierback(L, obj2gco(t), key);
438   lua_assert(ttisnil(gval(mp)));
439   return gval(mp);
440 }
```

lua 的哈希表以闭散列[3]方式实现。每个可能的键值，在哈希表中都有一个主要位置，称作 mainposition 。创建一个新键时，检查 mainposition ，若无人使用，则可以直接设置为这个新键。若之前有其它键占据了这个位置，则检查占据此位置的键的主位置是不是这里。若两者位置冲突，则

[3]用闭散列方法解决哈希表的键冲突，往往可以让哈希表内数据更为紧凑，有更高的空间利用率。关于其算法，可以参考：http://en.wikipedia.org/wiki/Open_addressing

利用 Node 结构中的 next 域，以一个单向链表的形式把它们链起来；否则，新键占据这个位置，而老键更换到新位置并根据它的主键找到属于它的链的那条单向链表中上一个结点，重新链入。

无论是哪种冲突情况，都需要在哈希表中找到一个空闲可用的结点。这里是在 getfreepos 函数中，递减 lastfree 域来实现的。lua 也不会在设置键位的值为 nil 时而回收空间，而是在预先准备好的哈希空间使用完后惰性回收。即在 lastfree 递减到哈希空间头时，做一次 rehash 操作。

源代码 2.5: ltable.c: rehash

```
343 static void rehash (lua_State *L, Table *t, const
      TValue *ek) {
344   int nasize, na;
345   int nums[MAXBITS+1];  /* nums[i] = number of keys
        with 2^(i-1) < k <= 2^i */
346   int i;
347   int totaluse;
348   for (i=0; i<=MAXBITS; i++) nums[i] = 0;  /* reset
        counts */
349   nasize = numusearray(t, nums);  /* count keys in
        array part */
350   totaluse = nasize;  /* all those keys are integer
        keys */
351   totaluse += numusehash(t, nums, &nasize);  /* count
        keys in hash part */
352   /* count extra key */
353   nasize += countint(ek, nums);
354   totaluse++;
355   /* compute new size for array part */
356   na = computesizes(nums, &nasize);
357   /* resize the table to new computed sizes */
358   luaH_resize(L, t, nasize, totaluse - na);
359 }
```

rehash 的主要工作是统计当前 table 中到底有多少有效键值对，以及决定数组部分需要开辟多少空间。其原则是最终数组部分的利用率需要超过 50% 。

lua 使用一个 rehash 函数中定义在栈上的 nums 数组来做这个整数键统计工作。这个数组按 2 的整数幂次来分开统计各个区段间的整数键个数。统计过程的实现见 numusearray 和 numusehash 函数。

最终，computesizes 函数计算出不低于 50% 利用率下，数组该维持多少空间。同时，还可以得到有多少有效键将被储存在哈希表里。

根据这些统计数据，rehash 函数调用 luaH_resize 这个 api 来重新调整数组部分和哈希部分的大小，并把不能放在数组里的键值对重新塞入哈希表。

查询操作 luaH_get 的实现要简单的多。当查询键为整数键且在数组范围内时，在数组部分查询；否则，根据键的哈希值去哈希表中查询。拥有相同哈希值的冲突键值对，在哈希表中由 Node 的 next 域单向链起来，所以遍历这个链表就可以了。

2.2.1 短字符串优化

以短字符串为键相当常见，lua 对此做了一点优化。

Lua 5.2.1 对短于 LUAI_MAXSHORTLEN （默认设置为 40 字节）的短字符串会做内部唯一化处理。相同的短字符串在同一个 State 中只会存在一份。这可以简化字符串的比较操作。在 Lua 5.2.0 之前，则会对所有的字符串不论长短做此处理。但在大多数应用场合，长字符串都是文本处理的对象，而不会做比较操作，内部唯一化处理将带来额外开销。而且对长字符串的部分哈希可能被用于 DoS 攻击，参见1.1.1。

在 Lua 5.2.1 中，长字符串并不做内部唯一化，且其哈希值也是惰性计算的。我们在 mainposition 函数中的 101-106 行可以看到这段惰性计算的代码。

源代码 2.6: ltable.c: mainposition

```
97 static Node *mainposition (const Table *t, const
       TValue *key) {
98   switch (ttype(key)) {
99     case LUA_TNUMBER:
```

```
100         return hashnum(t, nvalue(key));
101       case LUA_TLNGSTR: {
102         TString *s = rawtsvalue(key);
103         if (s->tsv.extra == 0) {  /* no hash? */
104           s->tsv.hash = luaS_hash(getstr(s), s->tsv.len,
                  s->tsv.hash);
105           s->tsv.extra = 1;  /* now it has its hash */
106         }
107         return hashstr(t, rawtsvalue(key));
108       }
109       case LUA_TSHRSTR:
110         return hashstr(t, rawtsvalue(key));
111       case LUA_TBOOLEAN:
112         return hashboolean(t, bvalue(key));
113       case LUA_TLIGHTUSERDATA:
114         return hashpointer(t, pvalue(key));
115       case LUA_TLCF:
116         return hashpointer(t, fvalue(key));
117       default:
118         return hashpointer(t, gcvalue(key));
119     }
120   }
```

从 luaH_get 的实现中可以看到，遇到短字符串查询时，就会去调用 luaH_getstr 回避逐字节的字符串比较操作。

源代码 2.7: ltable.c: get

```
463 /*
464 ** search function for short strings
465 */
466 const TValue *luaH_getstr (Table *t, TString *key) {
467   Node *n = hashstr(t, key);
468   lua_assert(key->tsv.tt == LUA_TSHRSTR);
469   do {  /* check whether 'key' is somewhere in the
```

```
      chain */
470     if (ttisshrstring(gkey(n)) && eqshrstr(rawtsvalue(
          gkey(n)), key))
471       return gval(n);  /* that's it */
472     else n = gnext(n);
473   } while (n);
474   return luaO_nilobject;
475 }
476
477
478 /*
479 ** main search function
480 */
481 const TValue *luaH_get (Table *t, const TValue *key) {
482   switch (ttype(key)) {
483     case LUA_TNIL: return luaO_nilobject;
484     case LUA_TSHRSTR: return luaH_getstr(t, rawtsvalue
          (key));
485     case LUA_TNUMBER: {
486       int k;
487       lua_Number n = nvalue(key);
488       lua_number2int(k, n);
489       if (luai_numeq(cast_num(k), nvalue(key))) /*
            index is int? */
490         return luaH_getint(t, k);  /* use specialized
              version */
491       /* else go through */
492     }
493     default: {
494       Node *n = mainposition(t, key);
495       do {  /* check whether 'key' is somewhere in the
            chain */
496         if (luaV_rawequalobj(gkey(n), key))
```

```
497             return gval(n);  /* that's it */
498           else n = gnext(n);
499         } while (n);
500         return luaO_nilobject;
501       }
502     }
503 }
```

2.2.2 数字类型的哈希值

当数字类型为键，且没有置入数组部分时，我们需要对它们取哈希值，便于放进哈希表内。

在 Lua 5.1 之前，对数字类型的哈希运算非常简单。就是对其占用的内存块的数据，按整数形式相加，代码如下：

```
1  #define numints          cast_int(sizeof(lua_Number)/
       sizeof(int))
2
3  static Node *hashnum (const Table *t, lua_Number n) {
4    unsigned int a[numints];
5    int i;
6    if (luai_numeq(n, 0))  /* avoid problems with -0 */
7      return gnode(t, 0);
8    memcpy(a, &n, sizeof(a));
9    for (i = 1; i < numints; i++) a[0] += a[i];
10   return hashmod(t, a[0]);
11 }
```

由于 Lua 的数字类型是允许用户配置的，有人为了让它可以处理 64 位整数，将其配置成 long double 。这使得上面这段代码引发了一个 bug ，他将这个这个 bug 报告到 lua 邮件列表中[4]。LuaJIT 的作者 Mike Pall 指出，在 64 位系统下，long double 被认为是 16 字节而不是 32 位系统下的 12 字节，以保持对齐。但其数值本身还是 12 字节，另 4 字节被填入随机

[4]见 2009 年 10 月，围绕 Crash in luaL_loadfile in 64-bit x86_64 的讨论。http://lua-users.org/lists/lua-l/2009-10/msg00642.html

的垃圾数据。如果依旧用上面的算法计算数字的哈希值，则有可能导致相同的数字拥有不同的哈希值[5]。

从严谨角度上来说，上面的 hashnum 算法也是值得商榷的。所以到了 Lua 5.2 以后，这个函数被修改为如下的实现：

源代码 2.8: ltable.c: hashnum

```
80 static Node *hashnum (const Table *t, lua_Number n) {
81   int i;
82   luai_hashnum(i, n);
83   if (i < 0) {
84     if (cast(unsigned int, i) == 0u - i)  /* use
         unsigned to avoid overflows */
85       i = 0;  /* handle INT_MIN */
86     i = -i;  /* must be a positive value */
87   }
88   return hashmod(t, i);
89 }
```

这里把数字哈希算法函数提取出来，变成了用户可配置的函数 luai_hashnum 。这个函数默认定义在 llimits.h 中。

这里有两个预定义的版本，当用户维持数字类型为 double ，且目标机器使用 IEEE754 标准的浮点数时，使用这样一个版本：

源代码 2.9: llimits.h: hashnum1

```
232 #define luai_hashnum(i,n)  \
233   { volatile union luai_Cast u; u.l_d = (n) + 1.0;  /*
        avoid -0 */ \
234     (i) = u.l_p[0]; (i) += u.l_p[1]; }  /* add double
        bits for his hash */
```

它使用一个联合体来取出浮点数所占内存中的数据。

```
1 union luai_Cast { double l_d; LUA_INT32 l_p[2]; };
```

[5]在 64 位系统下，用 lightuserdata 来处理 64 位整数是更好的选择，笔者实现了一个简单的扩展库：https://github.com/cloudwu/lua-int64

如果无法这种技巧来快速计算数字的哈希值，则改用一个性能不高，但更为通用，符合标准的算法：

源代码 2.10: llimits.h: hashnum2

```
281 #include <float.h>
282 #include <math.h>
283
284 #define luai_hashnum(i,n) { int e;  \
285   n = frexp(n, &e) * (lua_Number)(INT_MAX -
        DBL_MAX_EXP);  \
286   lua_number2int(i, n); i += e; }
287
288 #endif
```

2.3 表的迭代

在 Lua 中，并没有提供一个自维护状态的迭代器。而是给出了一个 next 方法。传入上一个键，返回下一个键值对。这就是 luaH_next 所要实现的。

源代码 2.11: ltable.c: next

```
169 int luaH_next (lua_State *L, Table *t, StkId key) {
170   int i = findindex(L, t, key);  /* find original
        element */
171   for (i++; i < t->sizearray; i++) {  /* try first
        array part */
172     if (!ttisnil(&t->array[i])) {  /* a non-nil value?
          */
173       setnvalue(key, cast_num(i+1));
174       setobj2s(L, key+1, &t->array[i]);
175       return 1;
176     }
177   }
```

```
178   for (i -= t->sizearray; i < sizenode(t); i++) {  /*
          then hash part */
179     if (!ttisnil(gval(gnode(t, i)))) {  /* a non-nil
            value? */
180       setobj2s(L, key, gkey(gnode(t, i)));
181       setobj2s(L, key+1, gval(gnode(t, i)));
182       return 1;
183     }
184   }
185   return 0;  /* no more elements */
186 }
```

它尝试返回传入的 key 在数组部分中的下一个非空值。当超出数组部分后，则检索哈希表中的对应位置，并返回哈希表中对应节点在储存空间分布上的下一个节点处的键值对。

在大多数其它语言中，遍历一个无序集合的过程中，通常不允许对这个集合做任何修改。即使允许，也可能产生未定义的结果。在 lua 中也一样，遍历一个 table 的过程中，向这个 table 插入一个新键这个行为，将无法预测后续的遍历行为[6]但是，lua 却允许在遍历过程中，修改 table 中已存在的键对应的值[7]。由于 lua 没有显式的从 table 中删除键的操作，只能对不需要的键设为空。

一旦在迭代过程中发生垃圾收集，对键值赋值为空的操作就有可能导致垃圾收集过程中把这个键值对标记为死键[8]。所以，在 next 操作中，从上一个键定位下一个键时，需要支持检索一个死键，查询这个死键的下一个键位。具体代码见 findindex 的实现：

源代码 2.12: ltable.c: findindex

[6]从实现上看，在遍历过程中插入一个不存在的键，并不会让程序崩溃，但有可能在当次遍历过程中，无法遍历到这个新插入的键。更严重的后果是，新插入的键触发了 rehash 过程，很有可能遍历到曾经遍历过的节点。

[7]在 Lua 手册[3] 关于 next 的描述中，这样写道：The behavior of next is undefined if, during the traversal, you assign any value to a non-existent field in the table. You may however modify existing fields. In particular, you may clear existing fields.

[8]死键在 rehash 后会从哈希表中清除，而不添加新键就不会 rehash 表而导致死键消失。在这个前提下，遍历 table 是安全的。

```
144 static int findindex (lua_State *L, Table *t, StkId
      key) {
145   int i;
146   if (ttisnil(key)) return -1;  /* first iteration */
147   i = arrayindex(key);
148   if (0 < i && i <= t->sizearray)  /* is 'key' inside
        array part? */
149     return i-1;  /* yes; that's the index (corrected
          to C) */
150   else {
151     Node *n = mainposition(t, key);
152     for (;;) {  /* check whether 'key' is somewhere in
          the chain */
153       /* key may be dead already, but it is ok to use
           it in 'next' */
154       if (luaV_rawequalobj(gkey(n), key) ||
155             (ttisdeadkey(gkey(n)) && iscollectable(key
                ) &&
156              deadvalue(gkey(n)) == gcvalue(key))) {
157         i = cast_int(n - gnode(t, 0));  /* key index
              in hash table */
158         /* hash elements are numbered after array ones
             */
159         return i + t->sizearray;
160       }
161       else n = gnext(n);
162       if (n == NULL)
163         luaG_runerror(L, "invalid key to " LUA_QL("
              next"));  /* key not found */
164     }
165   }
166 }
```

lua 的 table 的长度定义只对序列表有效[9]。所以，在实现的时候，仅需要遍历 table 的数组部分。只有当数组部分填满时才需要进一步的去检索哈希表。它使用二分法，来快速在哈希表中快速定位一个非空的整数键的位置。

源代码 2.13: ltable.c: getn

```
532 static int unbound_search (Table *t, unsigned int j) {
533   unsigned int i = j;  /* i is zero or a present index
          */
534   j++;
535   /* find 'i' and 'j' such that i is present and j is
          not */
536   while (!ttisnil(luaH_getint(t, j))) {
537     i = j;
538     j *= 2;
539     if (j > cast(unsigned int, MAX_INT)) {  /*
            overflow? */
540       /* table was built with bad purposes: resort to
            linear search */
541       i = 1;
542       while (!ttisnil(luaH_getint(t, i))) i++;
543       return i - 1;
544     }
545   }
546   /* now do a binary search between them */
547   while (j - i > 1) {
548     unsigned int m = (i+j)/2;
549     if (ttisnil(luaH_getint(t, m))) j = m;
550     else i = m;
551   }
552   return i;
```

[9]在 Lua 5.1 时，对 table 的获取长度定义更为严格一些。一个 table t 的长度 n ，要保证 t[n] 不为空，且 t[n+1] 一定为空。t[1] 为空的 table 的长度可定义为零。到了 lua 5.2 后，在手册[3] 的 3.4.6 节简化了定义。在循序序列中出现空洞（即有空值）有可能影响长度计算，但并非空值一定会截断长度统计。

```
553 }
554
555
556 /*
557 ** Try to find a boundary in table 't'. A 'boundary'
        is an integer index
558 ** such that t[i] is non-nil and t[i+1] is nil (and 0
        if t[1] is nil).
559 */
560 int luaH_getn (Table *t) {
561   unsigned int j = t->sizearray;
562   if (j > 0 && ttisnil(&t->array[j - 1])) {
563     /* there is a boundary in the array part: (binary)
          search for it */
564     unsigned int i = 0;
565     while (j - i > 1) {
566       unsigned int m = (i+j)/2;
567       if (ttisnil(&t->array[m - 1])) j = m;
568       else i = m;
569     }
570     return i;
571   }
572   /* else must find a boundary in hash part */
573   else if (isdummy(t->node))  /* hash part is empty?
        */
574     return j;  /* that is easy... */
575   else return unbound_search(t, j);
576 }
```

2.4 对元方法的优化

Lua 实现复杂数据结构，大量依赖给 table 附加一个元表（ metatable ）来实现。故而 table 本身的一大作用就是作为元表存在。查询元表中是

否存在一个特定的元方法就很容易成为运行期效率的热点。如果不能高效的解决这个热点，每次对带有元表的 table 的操作，都需要至少多作一次 hash 查询。但是，并非所有元表都提供了所有元方法的，对于不存在的元方法查询就是一个浪费了。

在 源代码2.1 中，我们可以看到，每个 Table 结构中都有一个 flags 域。它记录了那些元方法不存在。ltable.h 里定义了一个宏：

```
1 #define invalidateTMcache(t)      ((t)->flags = 0)
```

这个宏用来在 table 被修改时，清空这组标记位，强迫重新做元方法查询。只要充当元表[10]的 table 没有被修改，缺失元方法这样的查询结果就可以缓存在这组标记位中了。

源代码 2.14: ltm.h: fasttm

```
41 #define gfasttm(g,et,e) ((et) == NULL ? NULL : \
42   ((et)->flags & (1u<<(e))) ? NULL : luaT_gettm(et, e,
         (g)->tmname[e]))
43
44 #define fasttm(l,et,e)  gfasttm(G(l), et, e)
```

我们可以看到 fasttm 这个宏能够快速的剔除不存在的元方法。

另一个优化点是，不必在每次做元方法查询的时候都压入元方法的名字。在 state 初始化时，lua 对这些元方法生成了字符串对象：

源代码 2.15: ltm.c: init

```
32 void luaT_init (lua_State *L) {
33   static const char *const luaT_eventname[] = {  /*
       ORDER TM */
34     "__index", "__newindex",
35     "__gc", "__mode", "__len", "__eq",
36     "__add", "__sub", "__mul", "__div", "__mod",
37     "__pow", "__unm", "__lt", "__le",
38     "__concat", "__call"
```

[10]lua 5.2 仅对 table 的元表做了这个优化，而没有理会其它类型的元表的元方法查询。这大概是因为，只有 table 容易缺失一些诸如 __index 这样的元方法，而使用 table 的默认行为。当 lua 代码把这些操作作用于其它类型如 userdata 时，它没有 table 那样的默认行为，故对应的元方法通常存在。

```
39    };
40    int i;
41    for (i=0; i<TM_N; i++) {
42      G(L)->tmname[i] = luaS_new(L, luaT_eventname[i]);
43      luaS_fix(G(L)->tmname[i]);  /* never collect these
            names */
44    }
45  }
```

这样，在 table 查询这些字符串要比我们使用 lua_getfield 这样的外部 api 要快的多[11]。通过调用 luaT_gettmbyobj 可以获得需要的元方法。

源代码 2.16: ltm.c: gettm

```
63  const TValue *luaT_gettmbyobj (lua_State *L, const
      TValue *o, TMS event) {
64    Table *mt;
65    switch (ttypenv(o)) {
66      case LUA_TTABLE:
67        mt = hvalue(o)->metatable;
68        break;
69      case LUA_TUSERDATA:
70        mt = uvalue(o)->metatable;
71        break;
72      default:
73        mt = G(L)->mt[ttypenv(o)];
74    }
75    return (mt ? luaH_getstr(mt, G(L)->tmname[event]) :
        luaO_nilobject);
76  }
```

[11]使用 lua_getfield 做字符串检索，需要先将字符串压入 state 。这意味着需要把外部字符串在短字符串表池中做一次哈希查询。

2.4.1 类型名字

最后，有一段和元方法不太相关的代码也放在 ltm 模块中[12]。在 ltm.h / ltm.c 中还为每个 lua 类型提供了字符串描述。它用于输出调试信息以及作为 lua_typename 的返回值。这个字符串并未在其它场合用到，所以也没有为其预生成 string 对象。

源代码 2.17: ltm.c: typename

```
22 static const char udatatypename[] = "userdata";
23
24 LUAI_DDEF const char *const luaT_typenames_[
     LUA_TOTALTAGS] = {
25   "no value",
26   "nil", "boolean", udatatypename, "number",
27   "string", "table", "function", udatatypename, "
       thread",
28   "proto", "upval"  /* these last two cases are used
       for tests only */
29 };
```

udatatypename 在这里单独并定义出来，多半出于严谨的考虑。让 userdata 和 lightuserdata 返回的 "userdata" 字符串指针保持一致。

[12]把这组字符串常量定义在 ltm 中，大概是因为这里同时定义了元方法的名字常量，实现比较类似罢了。

参考文献

[1] Roberto Ierusalimschy. The novelties of lua 5.2. 2011. http://www.inf.puc-rio.br/~roberto/talks/novelties-5.2.pdf.

[2] Waldemar Celes Roberto Ierusalimschy, Luiz Henrique de Figueiredo. The evolution of lua. *Proceedings of ACM HOPL III (2007) 2-1-2-26*. http://www.lua.org/doc/hopl.pdf.

[3] Waldemar Celes Roberto Ierusalimschy, Luiz Henrique de Figueiredo. *Lua 5.2 Reference Manual*, 2011. http://www.lua.org/manual/5.2/manual.html.

[4] Lua Wiki. Hash dos. http://lua-users.org/wiki/HashDos.

[5] Wikipedia. Indent style. http://en.wikipedia.org/wiki/Indent_style.

[6] Wikipedia. Separate chaining. http://en.wikipedia.org/wiki/Hash_table#Separate_chaining.

源代码目录