(http://twitter.com/kenlistdev) (http://github.com/kenlist)

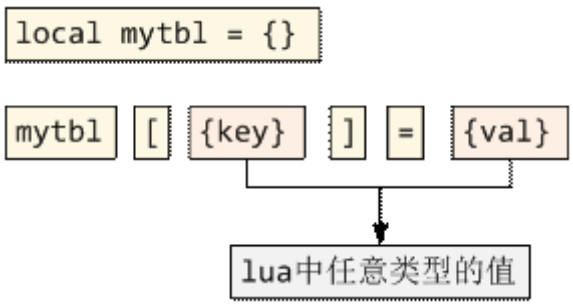
(http://geekluo.com/) (mailto:rijian.lrj@alibaba-inc.com)

Lua数据结构 -- Table（三）

2014/04/11

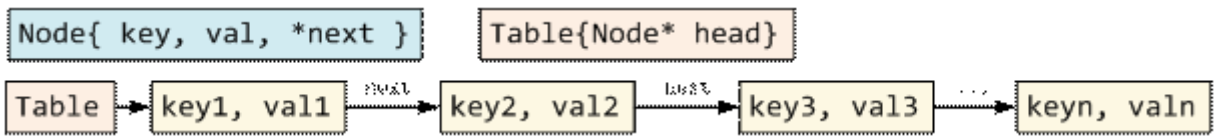
前面（一）、（二）里面其实已经把一些常用的数据类型（数值、布尔、字符串）说明了，这次要描述的是Table，Table在Lua里是一种常用的数据类型，是Lua里的精髓之一，其效率必须得到保证，而实现这种支持任意类型key和value的Table也是较为复杂的。

##一， Table的设计思想： 1， 首先，讲一下Lua要设计的Table是怎么样子的：



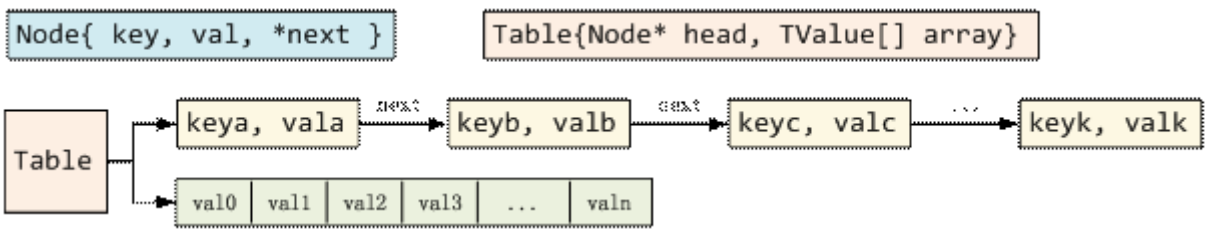
Lua就是想做这种支持任意类型的key和任意类型val的table，并且要**高效和节约内存。**

2， 基本的实现（基于链表的实现）：

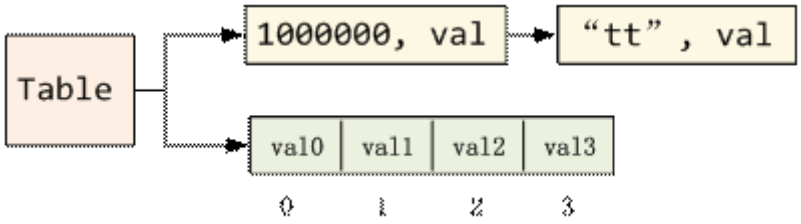


基于链表的实现是最简单的，其实map<TValue*, TValue*>就可以了，这样实现是最容易的。但当遇到很多key的数组（如t[0]、t[1]、t[2]。。这种数值索引大数组）时，明明**可以用O(1)查找的，却要O(n)去查找。**

3， 区分数字key和其它类型的key

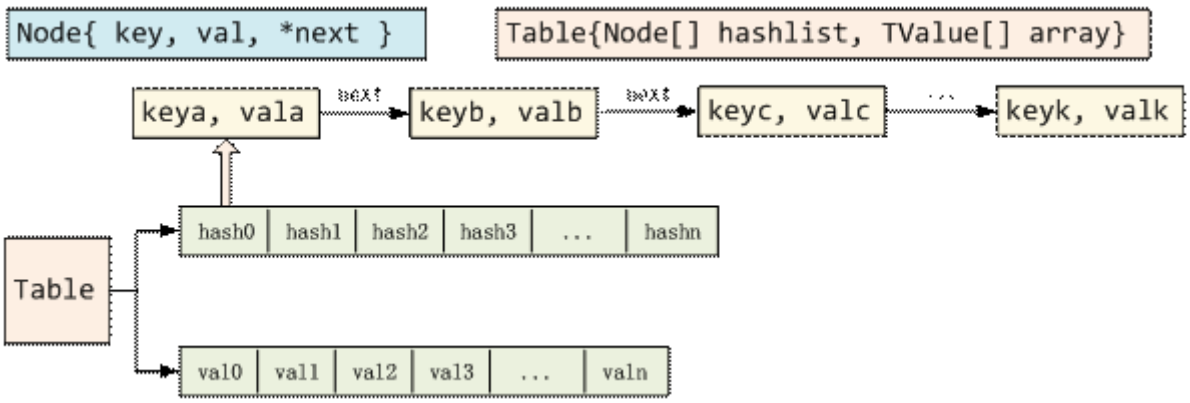


经过改良的Table，除了有key链表之外，还有一个**数组array专门存放key为数值的val。**但是这种情况下，要保证数值部分是连续且从0开始的，如果出现**t[100000000] = 1**，则把这个离散的数据放到链表中：



4， 利用哈希表再度优化

区分了array和head之后，始终有个问题，对于链表部分的数据，查找始终是O(n)的，有没有办法优化这部分代码呢，在Lua里，利用哈希表再对这部分的Node进行查找。

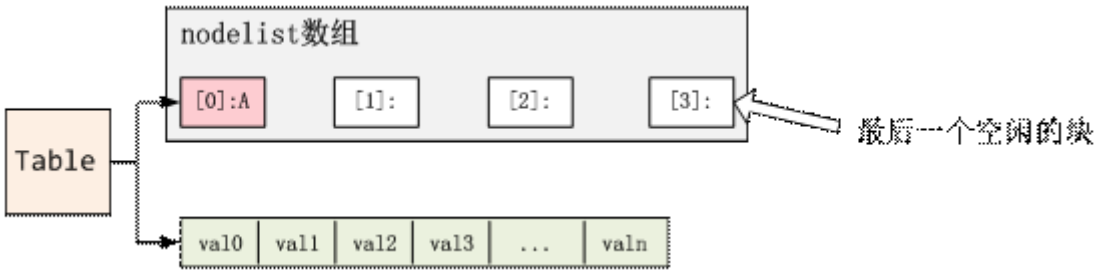


每次计算一个key的哈希值是非常快的，哈希后直接映射到hashlist的某个位置。这里已经很接近Lua Table的最终设计，但是这种方法仍然有个弊端，哈希表的大小无法较好地估计，hashlist的长度可能是一个固定的长度，无法动态扩容。

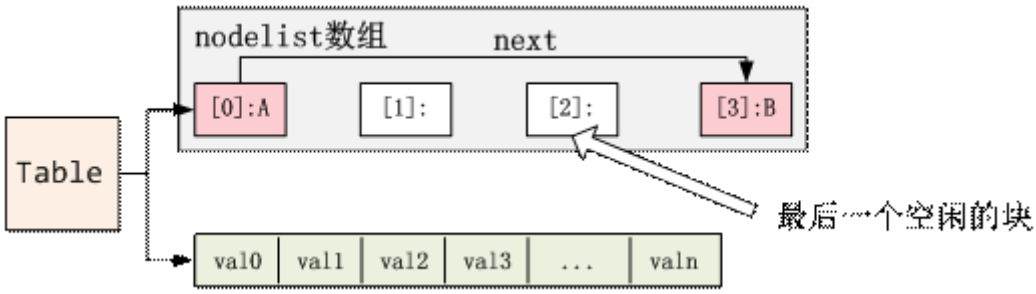
5， 动态扩容的Table设计

下面用例子展示一下动态扩容的Table设计

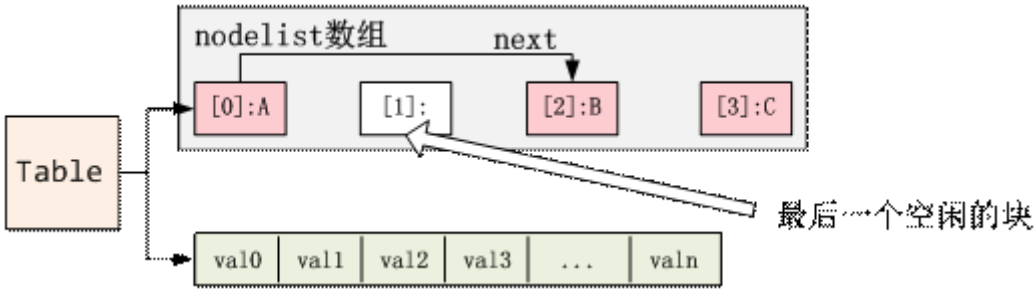
1） 如下图，现在初始状态下，只有[0]被使用了，里面放着A，其它都是空：



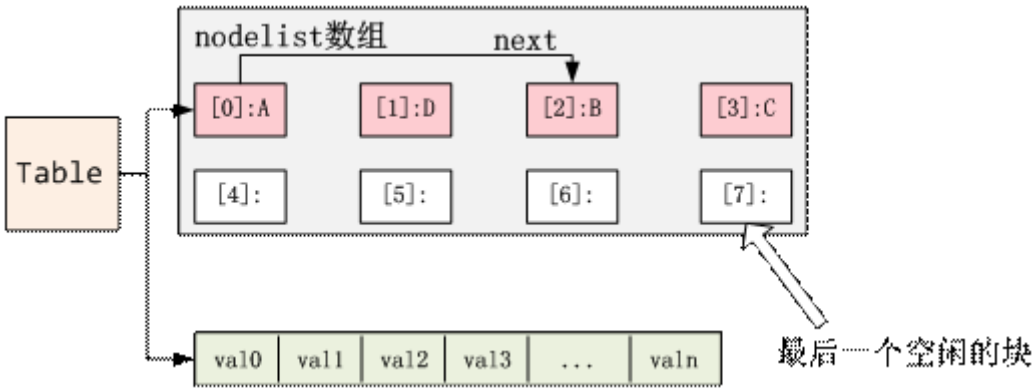
2） 现在要**插入一个新的元素B**，计算出其**哈希值是0**，即是说应该插入到节点[0]。这个时候发现**节点[0]已经被使用**，则会分配最后一个空闲块lastfree给这个元素B，然后node[0]的next指向node[3]，即：



3） 然后再**插入一个新的nodeC**，计算出其哈希值是3，即是说应该插入到node[3]。这个时候发现**node[3]已经被使用**，但是**元素B的哈希值是0**，即本来应该插入到node[0]的，于是**把node[3]的内容移到lastfree**，然后**再在node[3]插入新的nodeC**，即：



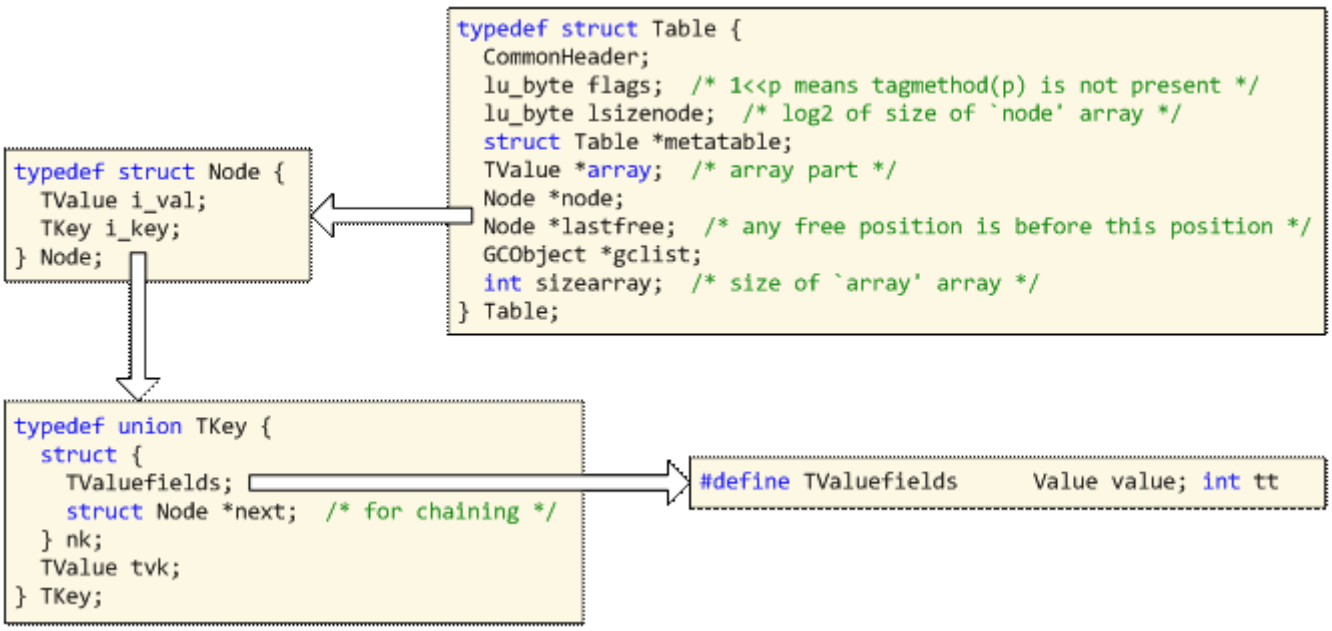
4） 这是如果再往Table插入一个元素D，那么必然最后一个空闲块会被使用完，那么就会把nodelist的大小扩大一倍，并且重新计算每个元素的哈希值并重新插值，可能的结果如下：



在最后一步的重新计算哈希值，不仅重新计算nodelist的哈希值，也会重新计算arraylist的哈希值，arraylist也是会动态扩大的，这就是lua中table的设计。

##二， Lua里面的实现：

Table相关数据结构关系图如下：



上图中有Table、Node、TKey这3个数据结构，不用急，我们先从简单的入手，看看**Node数据结构**： （lobject.h 332-335）

```
typedef struct Node {
  TValue i_val;
  TKey i_key;
} Node;
```

Node就是设计思想里的key、value数据结构，包含i_key和i_val两个成员，这2个成员很好理解，一个就是table的key，另一个就是这个key的value。i_val是一般值的TValue类型，而i_key的TKey类型的。可以看到Node并没有next指针，**其实它把next指针藏在TKey下面了**，请看TKey数据结构：（lobject.h 319-329）

```
typedef union TKey {
  struct {
    TValuefields;
    struct Node *next;  /* for chaining */
  } nk;
  TValue tvk;
} TKey;
```

```
#define TValuefields	Value value; int tt
```

可以看到TKey其实是一个支持TValue的数据结构外，还**多了一个next指针**。这个next指针就是用作**同一个hash值下冲突时的链表指针**。明白了Node结构之后，接下来看看Table数据结构：（lobject.h 338-348）

```
typedef struct Table {
  CommonHeader;
  lu_byte flags;  /* 1<<p means tagmethod(p) is not present */
  lu_byte lsizenode;  /* log2 of size of `node' array */
  struct Table *metatable;
  TValue *array;  /* array part */
  Node *node;
  Node *lastfree;  /* any free position is before this position */
  GCObject *gclist;
  int sizearray;  /* size of `array' array */
} Table;
```

每个字段意义如下：

- CommonHeader：与TValue中的GCHeader能对应起来的部分
- flags：用于元表元方法的一些优化手段，一共有8位用于标记是否没有某个元方法
- lsizenode：用于表示node的长度，如下图所示

node成员其实是上面讨论的hashlist成员，是一个固定长度大小的数组，但是lsizenode的数据类型是lu_byte，只有一个字节长，**表示范围只有0~255**，一般数组大小都会很大，所以这里lsizenode用于**表示整体长度的log2值**，同时也表明了，**hashlist的长度是2的幂，每次增长都会×2**.

- metatable：元表指针
- array：这个成员就是上面讨论的array，用于给数值的索引

- node：上面提到的hashlist成员
- lastfree：lastfree就是链表的最后一个空元素
- gclist：用于gc的，以后会有专门对GC的详细讨论
- sizearray：array数组的大小

##离散数值key存储的实现： 在luaH_getnum（ltable.c 432-449）函数中，实现了对数值key的获取，可以看到第一个判断：

```
if (cast(unsigned int, key-1) < cast(unsigned int, t->sizearray))
        return &t->array[key-1];
```

即**如果key在sizearray的范围内，则直接用t->array成员来存储，如果不是则计算key的哈希值，然后放到node里。**

还有一种情况，就是如果对某个连续数值的table赋值：t[2] = nil，那是否从2到后面的key都会马上放到哈希表里呢？答案是否定的，不会马上做，等到做完gc后，会进行table的resize。

##Table的Rehash（重新计算大小）： 1） **rehash的时机：**

在newkey(ltable.c 399-429)函数中可以看到

```
  Node *n = getfreepos(t);  /* get a free place */
  if (n == NULL) {  /* cannot find a free place? */
    rehash(L, t, key);  /* grow table */
    return luaH_set(L, t, key);  /* re-insert key into grown table */
  }
```

n是hashlist中的一个没使用的节点，**当没有空余节点的时候**，就会调用rehash进行grow table，这个可以参考本文上面说到的动态扩容章节。

2） **rehash函数(ltable.c 333-349)**

table的这个rehash操作，代码不多，但是却十分复杂，接下来我们分解一下它所做的事：

a． 计算使用数值作为key的元素数量na、计算实际为数组申请的空间大小nasize、计算hashlist的元素数量nhsize。 这里有点模糊，na和nasize的关系，下面写个例子更好说明一下：

```
na = 3          nasize = 4
na = 5          nasize = 8
na = 12         nasize = 16
na = 14         nasize = 16
...
```

没错，nasize一定要是2的幂，computesizes(ltable.c 189-208)通过特定算法，高效地计算出实际要使用的数组大小，举下面例子说明一下：

```
tbl[2]=0
table.nasize = 0    //数组部分大小
table.nhsize = 1    //哈希链表部分大小
```

```
tbl[2]=0    tbl[3]=0
table.nasize = 0    //数组部分大小
table.nhsize = 2    //哈希链表部分大小
```

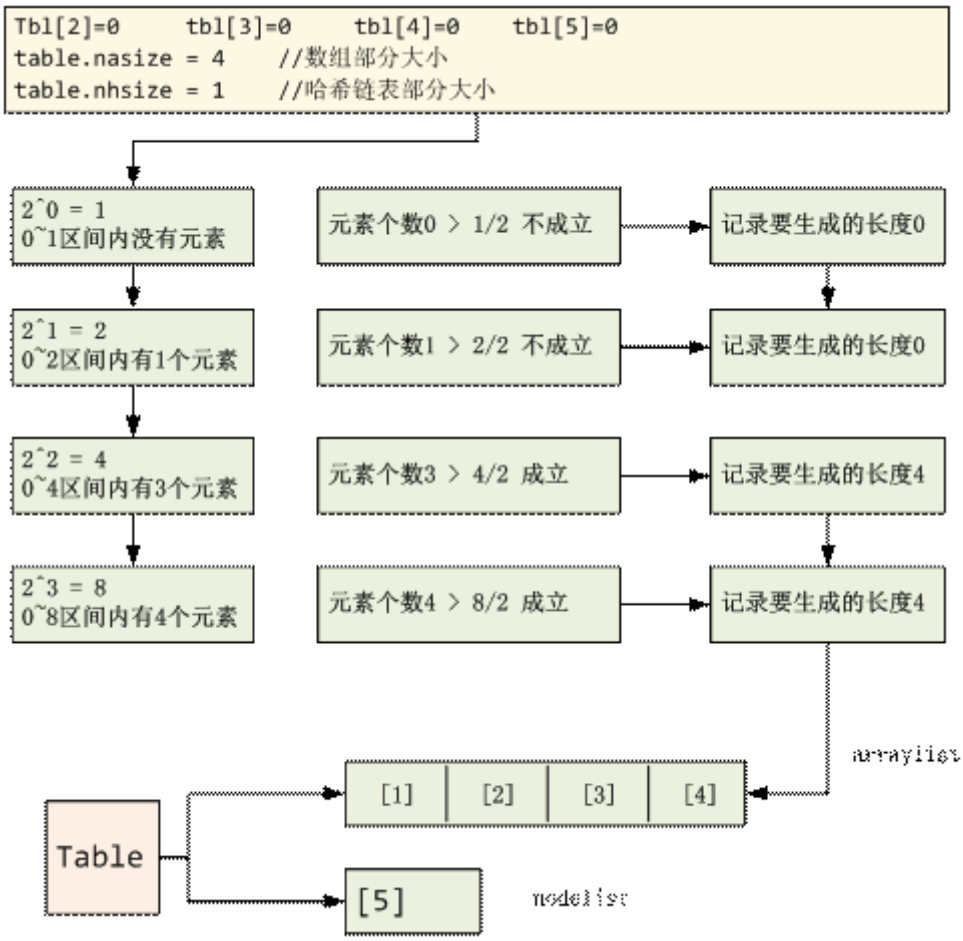
```
Tbl[2]=0    tbl[3]=0    tbl[4]=0
table.nasize = 4    //数组部分大小
table.nhsize = 0    //哈希链表部分大小
```

```
Tbl[2]=0    tbl[3]=0    tbl[4]=0    tbl[5]=0
table.nasize = 4    //数组部分大小
table.nhsize = 1    //哈希链表部分大小
```

lua其实是用了一个条件来决定数组部分大小的：

如果数值key的元素个数大于对应个数幂大小的一半，则生成对应幂长度的数组链表。

很抽象，还是拿上面的例子来说明：



整体算法如上图所示，还是挺精致的，不太懂用语言描述，可以想象一个元素如果拥有tbl[10]到tbl[50]，那么这个arraylist的长度是64，中间可能会多**生成1~10和50~64这个区间的数组**，但是这种方法既能动态扩容，又能提升效率，牺牲一点点还是值得的。

b． resize(ltable.c 300-327)函数，根据前面计算出来的nasize和nhsize，realloc对应数组的大小，并对其中的元素重新计算哈希值和赋值。

哈希的实现：

主要可以看到mainposition（ltable.c 96-113）函数，用于计算哈希然后快速定位到某个Node上面，可以看到它根据不同类型有不同的哈希计算：

类型	哈希方法
LUA_TNUMBER	hashnum（ltable.c 81-92） 很简单的哈希方法，把 lua_Number(double)的内存 memcpy 到一个 uint 上，然后加起来，再取 mod
LUA_TSTRING	用回字符串的哈希方法，详情请看“数据结构（二） -- TString”
LUA_TBOOLEAN	hashboolean（ltable.c 53） 非常简单，直接取其 boolean 值，然后取模，可以预想到其值不是 0 就是 1
其它类型	hashpointer（ltable.c 63） 也是很简单，直接去其指针值，然后取模

###元表的实现： 元表是metatable，可以绑定metatable的对象在lua中只有table和userdata。这里讨论的是table中的metatable，在userdata中的其实也一样。我们看到Table数据结构里的struct Table* metatable指针，下面以index操作为例，其它的话其实也一样：

看luaV_gettable(lvm.c 108-131)，我们可以看到在取一个对应key后会有判断：

```
if (!ttisnil(res) ||  /* result is no nil? */
(tm = fasttm(L, h->metatable, TM_INDEX)) == NULL) { /* or no TM? */
```

这个判断其实就是看看返回结果如果是空，就会去**取元表的__index对象**，取回来之后，下次循环就再次用这个tm来取key，如果在tm上找不到对应key，而且tm又有metatable，**就会一直循环下去。**

这里fasttm做了一些优化，其实就是先用h->metatable的**flags成员去判断是否存在__index元方法**，如果不存在马上返回。flags只有8位，用于存储常用的元操作，可以在ltm.h 18-37看到，快速操作的常用元方法是__index 、__newindex、__gc、__mode、__eq，说明flags还有3位没用到。

循环有个MAXTAGLOOP，这里其实限制了元表的深度不能超过100（其实超过5个深度的元表已经很恐怖了）。元操作对象的获取方法其实是luaT_gettm（ltm.c 50-58）和luaT_gettmbyobj（ltm.c 61-74）

###总结：

对于Table，还有个弱表的特性，这个留待在说gc的时候再详细讨论。其实Table的实现还是挺多细节的，不过主要的思想和处理都说了（除了gc）。

lua (6) (http://geekluo.com/tags.html#lua-ref)

← Previous (http://geekluo.com/contents/2014/04/11/2-lua-data-structure.html)

Next → (http://geekluo.com/contents/2014/04/11/4-lua-tstring-structure.html)