1.table的特性

1. 在Lua中table是个非常重要的类型，通过使用table的一些特性可以实现许多数据结构，例如map,array queue,stack等。
2. 通过使用者角度来讲，table既可以当作array使用也可以当作map使用，那么对于设计者来讲，那么需要保证table的高效率的查找、插入、遍历。
3. 当然，table的设计者还提出了metatable(元表)的概念，以供使用者可以用来实现继承、操作符重载等设计，不过metatable暂时不在这边文章进行讨论。

2.table的定义

```
typedef union TKey {
  struct {
    TValuefields;
    int next;  /* 用于标记链表下一个节点 */
  } nk;
  TValue tvk;
} TKey;

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


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

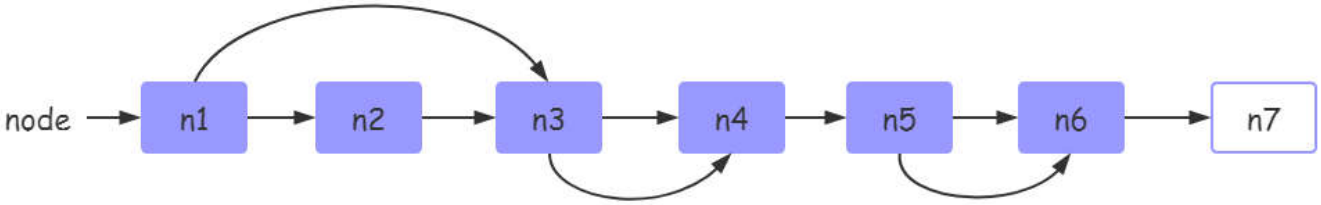
- **flags** ： 元方法的标记，用于查询table是否包含某个类别的元方法
- **lsizenode** : (1 < <lsizenode)表示table的hash部分大小
- **sizearray** : table的数组部分大小
- **array** : table的array数组首节点
- **node** ：table的hash表首节点
- **lastfree** : 表示table的hash表空闲节点的游标
- **metatable** : 元表
- **gclist** ： table gc相关的参数

为了提高table的插入查找效率，在table的设计上，采用了array数组和hashtable(哈希表)两种数据的结合。

> 所以table会将**部分**整形key作为下标放在数组中, 其余的整形key和其他类型的key都放在hash表中。

3.hash表结构

在table中的实现中，hash表占绝大部分比重，下面是table中hash表的结构示意简图：



hash表在解决冲突有两个常用的方法：

- **开放定址法**：当冲突发生时，使用某种探查(亦称探测)技术在散列表中形成一个探查(测)序列。沿此序列逐个单元地查找，直到找到给定 的关键字，或者碰到一个开放的地址(即该地址单元为空)为止（若要插入，在探查到开放的地址，则可将待插入的新结点存人该地址单元）。查找时探查到开放的 地址则表明表中无待查的关键字，即查找失败。
- **链地址法**：又叫拉链法，所有关键字为同义词的结点链接在同一个单链表中。若选定的散列表长度为m，则可将散列表定义为一个由m个头指针组成的指针数 组T[0...m-1]。凡是散列地址为i的结点，均插入到以T[i]为头指针的单链表中。T中各分量的初值均应为空指针。在拉链法中，装填因子α可以大于 1，但一般均取α≤1。

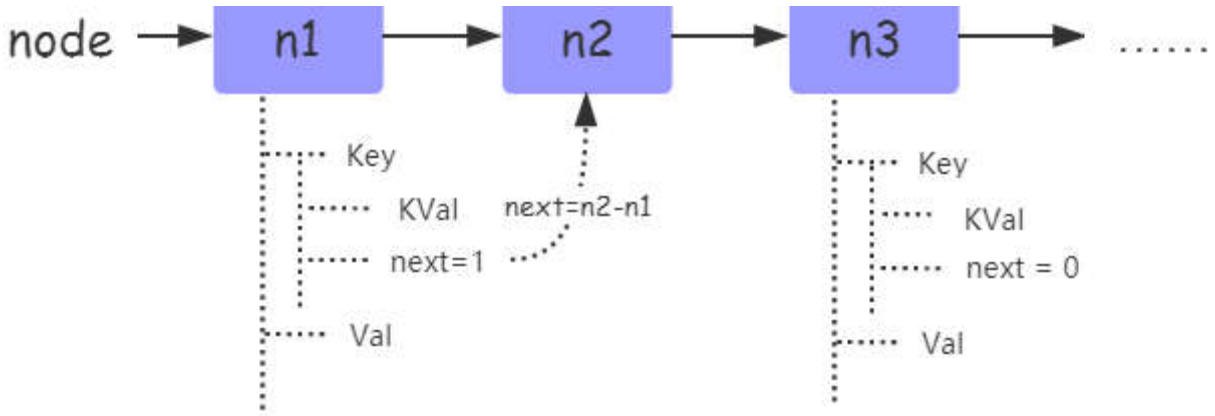
简单对比可以发现以上两种方法的优缺点：

开放定址法相比链地址法节省更多的内存，但在插入和查找的时候拥有更高的复杂度。

但是table中的hash表的实现结合了以上两种方法的一些特性：

- 查找和插入等同链地址法复杂度。
- 内存开销近似等同于开放定址法。

原因是table中的hash表虽然采用链地址法的形式处理冲突，但是链表中的额外的节点是hash表中的节点，并不需要额外开辟链表节点；下面是TKey结构的介绍：



那么如何将hash表的空余节点利用起来作为链表的节点呢？这个算法的实现得益于lastfree这个指针的作用，后面会详细介绍。

4.table的创建

lua中通过luaH_new来创建一个新table：

```
Table *luaH_new (lua_State *L) {
  GCObject *o = luaC_newobj(L, LUA_TTABLE, sizeof(Table));
  Table *t = gco2t(o);
  t->metatable = NULL;
  t->flags = cast_byte(~0);
  t->array = NULL;
  t->sizearray = 0;
  setnodevector(L, t, 0);
  return t;
}
```

此时，table中的array部分和hash部分都为空。

5.luaH_get分析

table中通过这个函数来从表中查找key对应的值；可以看到它会通过key的类型不同，从而进行不同的处理。

```
const TValue *luaH_get (Table *t, const TValue *key) {
  switch (ttype(key)) {
    case LUA_TSHRSTR: return luaH_getshortstr(t, tsvalue(key));
```

```
      case LUA_TSHRSTR: return luaH_getshortstr(t, tsvalue(key));
      case LUA_TNUMINT: return luaH_getint(t, ivalue(key));
      case LUA_TNIL: return luaO_nilobject;
      case LUA_TNUMFLT: {
        lua_Integer k;
        if (luaV_tointeger(key, &k, 0)) /* index is int? */
          return luaH_getint(t, k);  /* use specialized version */
        /* else... */
      }  /* FALLTHROUGH */
      default:
        return getgeneric(t, key);
    }
  }
```

- 如果key是nil，则直接返回nil。
- 如果key是整数类形，则调用luaH_getint来处理，因为整形key可能会在array中取值。
- 如果key是浮点数类型，则首先判断key是否能转化为整数，如果是则调用luaH_getint，否则调用getgeneric。
- 如果key是短字符串类型，则调用luaH_getshortstr来处理。（其实这个case有点不理解，短字符串也可以交给getgeneric来处理）
- 其他类型的key，都使用getgeneric来处理。

下面着重分析luaH_getint、getgeneric这两个函数的流程。

luaH_getint

```
const TValue *luaH_getint (Table *t, lua_Integer key) {
  /* (1 <= key && key <= t->sizearray) */
  if (l_castS2U(key) - 1 < t->sizearray)
    return &t->array[key - 1];
  else {
    Node *n = hashint(t, key);
    for (;;) {  /* check whether 'key' is somewhere in the chain */
      if (ttisinteger(gkey(n)) && ivalue(gkey(n)) == key)
        return gval(n);  /* that's it */
      else {
        int nx = gnext(n);
        if (nx == 0) break;
        n += nx;
      }
    }
    return luaO_nilobject;
  }
}
```

前面说过，table由array和hashtable组成，所以对于整形的key需要先去数组范围内找：

- 如果key的大小在数组大小范围内，那么就直接在数组中查找值并返回。
- 否则，获取int的hash值对应的hashslot，然后在slot-link上找到key对应的值并返回。（和链地址法的查找是一样的）
- 如果找不到，则返回nil。

getgeneric

```
static const TValue *getgeneric (Table *t, const TValue *key) {
  Node *n = mainposition(t, key);
  for (;;) {  /* check whether 'key' is somewhere in the chain */
    if (luaV_rawequalobj(gkey(n), key))
      return gval(n);  /* that's it */
    else {
      int nx = gnext(n);
      if (nx == 0)
        return luaO_nilobject;  /* not found */
      n += nx;
    }
  }
}
```

其实getgeneric流程就是传统的链地址法查找流程，不过值得注意的是mainposition函数，在这里面区分了lua对于各种类型的hash方式：

```
static Node *mainposition (const Table *t, const TValue *key) {
  switch (ttype(key)) {
    case LUA_TNUMINT:
      return hashint(t, ivalue(key));
    case LUA_TNUMFLT:
      return hashmod(t, l_hashfloat(fltvalue(key)));
    case LUA_TSHRSTR:
      return hashstr(t, tsvalue(key));
    case LUA_TLNGSTR:
      return hashpow2(t, luaS_hashlongstr(tsvalue(key)));
    case LUA_TBOOLEAN:
      return hashboolean(t, bvalue(key));
    case LUA_TLIGHTUSERDATA:
      return hashpointer(t, pvalue(key));
    case LUA_TLCF:
      return hashpointer(t, fvalue(key));
    default:
      lua_assert(!ttisdeadkey(key));
      return hashpointer(t, gcvalue(key));
  }
}
```

mainpostion为hash值%hash表的大小。
值得注意的是对于字符串的hash处理，lua区分了长字符串和短字符串(5.3之后对字符串按照长短做了区分处理）

- 对于短字符串，lua都存放在stringtable中，所以对于短字符串只有一个实体。可以直接使用string在stringtable中的hash值。
- 对于长字符串，lua中可能会存在多个实例，所以需要通过luaS_hash来计算其hash值。

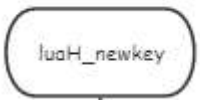
6.luaH_set分析

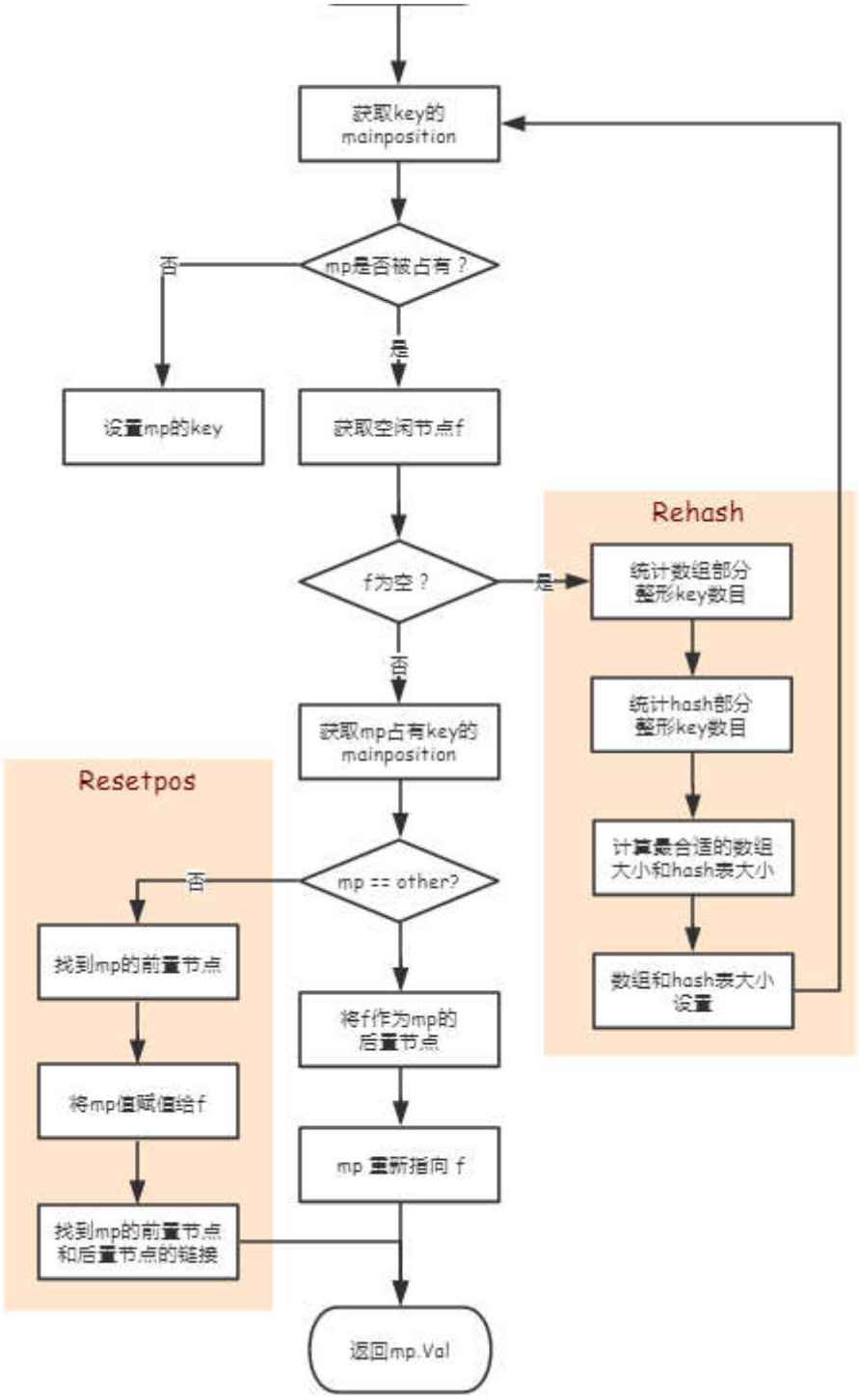
```
TValue *luaH_set (lua_State *L, Table *t, const TValue *key) {
  const TValue *p = luaH_get(t, key);
  if (p != luaO_nilobject)
    return cast(TValue *, p);
  else return luaH_newkey(L, t, key);
}
```

luaH_set 不是传统意义上的set，也就是直接传入key和value然后设置，而是传入key会返回这个key对应的TValue，然后再通过setobj2t对这个TValue进行设置。所以这个set函数就很简单了：

- 首先调用luaH_get查找table是否已经存在这个key了，有则直接返回。
- 否则调用luaH_newkey创建key，并返回对应的TValue。（注意此时key一定不在数组部分内）

那么luaH_set的分析就转化为luaH_newkey的分析：

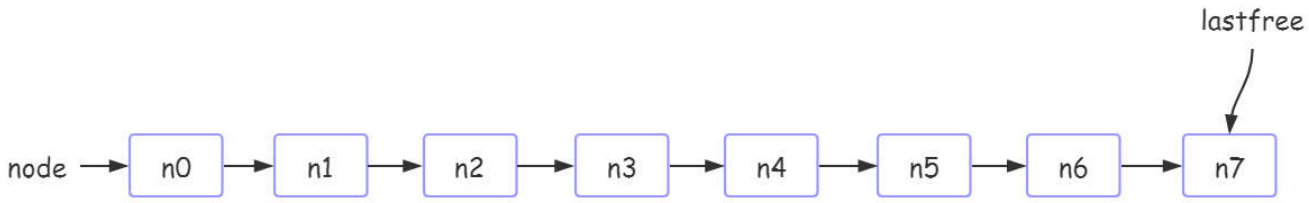




看流程图有点复杂，下面给出实际的例子来举例（先不考虑**Rehash部分**）：

```
//假设tb的hash表默认大小为8个元素
local tb = {}
tb[3] = 'a'
tb[11] = 'b'
tb[19] = 'c'
tb[6] = 'd'
tb[14] = 'e'
```

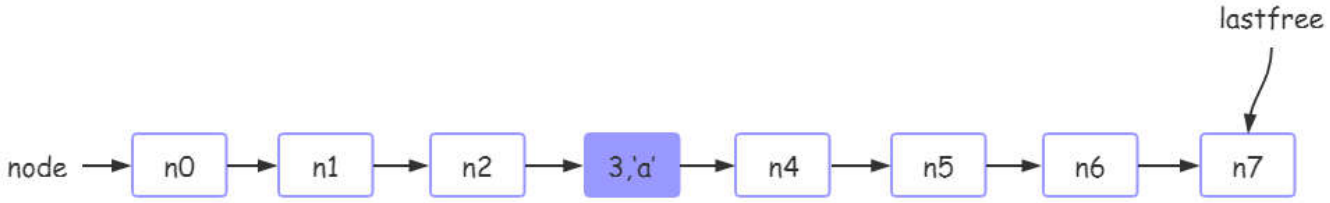
1. 执行完local tb = {}之后，tb中的hash表状态是这样：



空余节点指针lastfree指向了最后一个node。

> 注意：table创建默认hash表大小为0，这里为了方便描述假设初始大小为8，这样就不用管rehash部分了

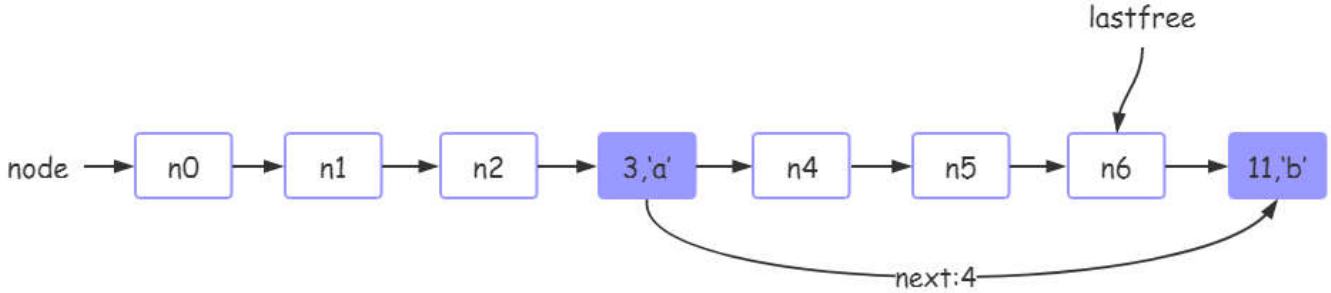
2. 执行完tb[3] = 'a' 之后，tb中的hash表状态是这样：



因为3的mainposition为3，所以放在了n3位置。

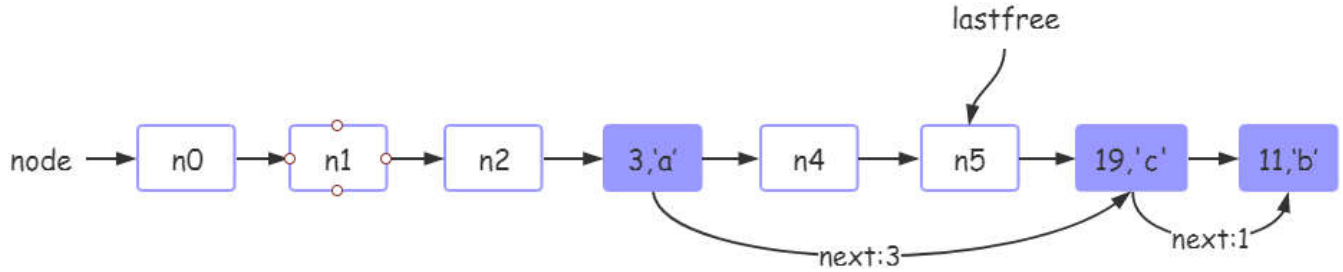
> 注意：key的manposition等于hash(k)%table_size, 所以mainposition(3)=3%8

3. 执行完tb[11] = 'b' 之后，tb中的hash表状态是这样：



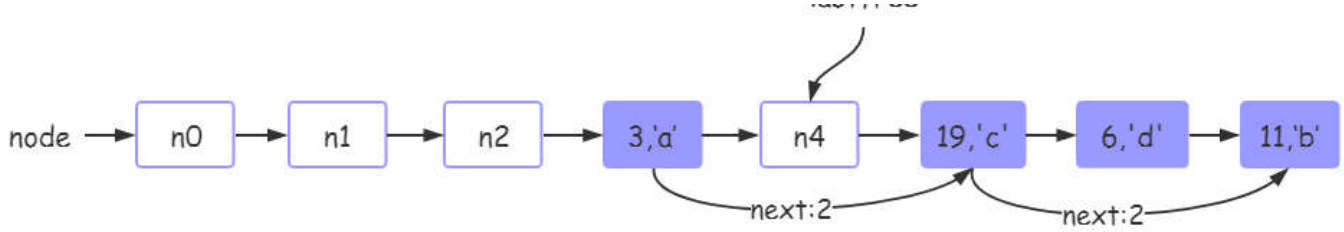
因为11的mainposition也为3，然而3位置已经被占用了，所以此时使用lastfree获取一个空节点n7，将当前key存储在n7位置上，并且使用头插法将n7节点插入在mainposition节点n3之后，所以这里的next = n7 - n3 = 4。

4. 执行完tb[19] = 'c' 之后，tb中的hash表状态是这样：



原因和上面类似，只不过注意的是，19是插入在mainposition节点n3和mp的next节点n7之间，所以需要重新维护n3的next值。

5. 执行完tb[6] = 'd' 之后，tb中的hash表状态是这样：



在这一步中有些不一样的处理，首先还是算出6的mainposition为6，然后发现n6已经被key:19占用了。但是此时我们不能直接使用lastfree来存储key:6，因为19和6不是同一个链表上的，也就是说key:19抢了key:6的位置：

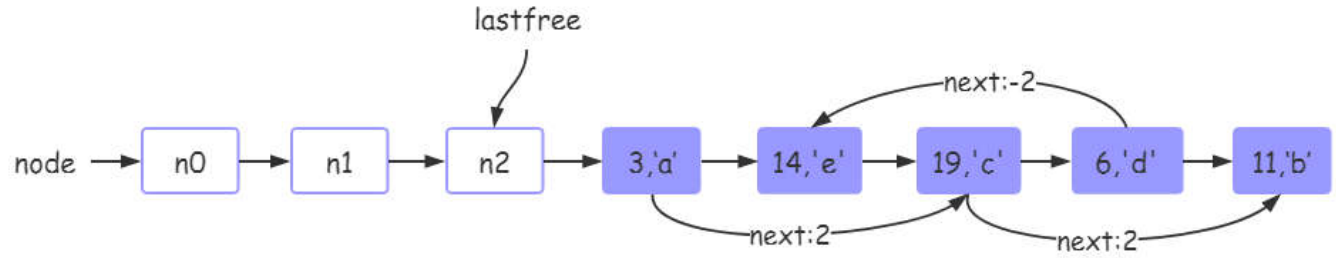
$$mainpositon(19) = 19$$

$$mainpositon(6) = 6\%8 = 6$$

对于这种情况，我们需要key:19让出位置，通过lastfree申请一个空节点n5，然后将19的位置换到n5上（注意维护next节点）。然后将key:6放在n6节点上。

> 这部分操作就是流程图上Resetpos部分

6. 执行完tb[14] = 'e’ 之后，tb中的hash表状态是这样：

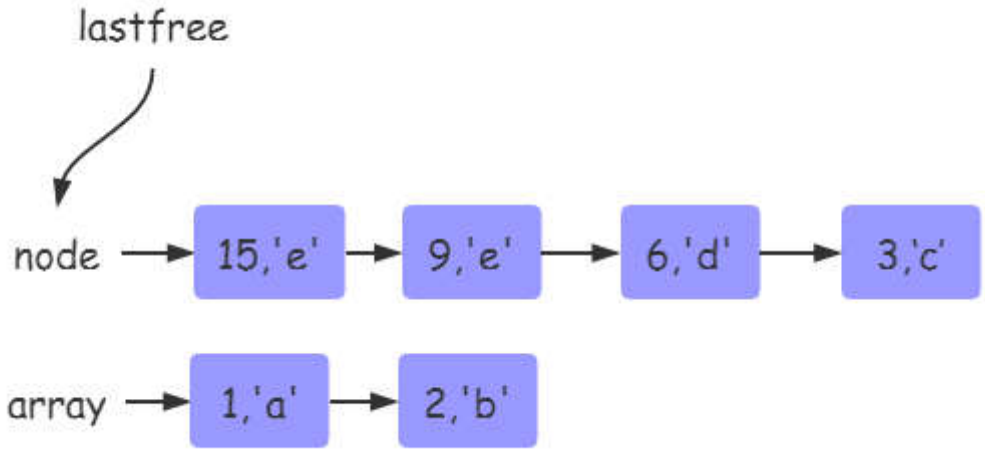


这里和步骤3类似。

下面再来分析一下Rehash的部分：

> Rehash并不一定代表hash表的扩容，而是根据table里面的key的个数和类型，重新更合适的分配array的大小和hash表的大小，可能会扩容、可能不变、也有可能缩小。

假如此时table里面的array和hash表状态如下：



此时再插入一个key:10，因为lastfree已经无法获取空节点了，所以触发了rehash。

- 首先通过numusearray计算数组部分val不为nil的所有整形数目，和nums[]。对于nums[i] = j，其意义表

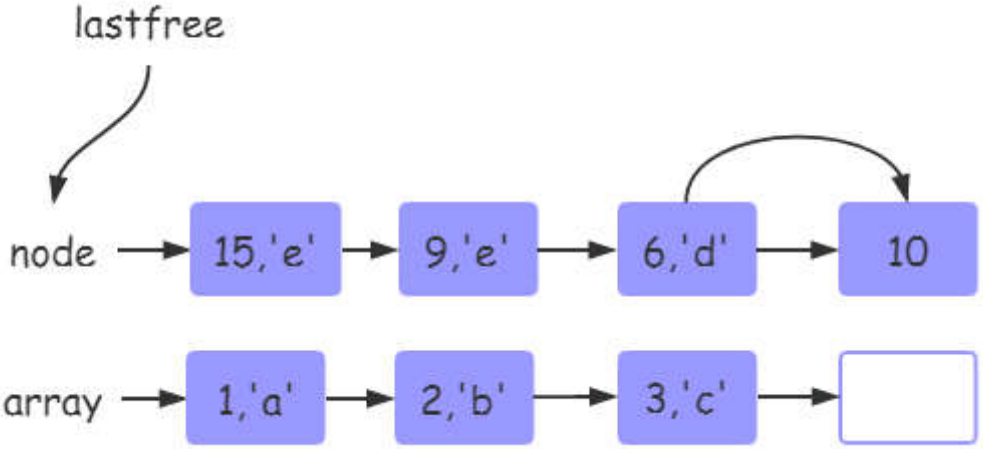
示key在2^i-1 到2^i之间其整形key的个数有j个。

- 然后再通过numusehash计算hash部分Val不为nil的所有整形数目，和nums[]。
- 通过整形key的个数确定array的大小，computesizes，这里确定array的大小有个规则就是要满足2的幂size，并且整形key数目num > arraySize / 2，还要保证放入整形key的数目高于arraySize / 2。
- 最后根据array大小和总key个数，确定hash表的大小。（ps：hash表的大小也只能是2的幂，如果不是则向上对齐）

通过上面的规则可以计算得到array部分的大小为4，hash表大小为7 - 3 = 4。(7是指Key的总数，3是指能放入数组的Key的个数)

> 无论是array的rehash还是hash表的rehash都是先开辟新的内存，然后将原来的元素重新插入。

插入key:10后的状态为：



> 值得注意的是：table元素的删除是通过table[key] = nil来实现的，然而通过我们上面对luaH_set介绍我们可以知道，仅仅是把把key对应的val设置为nil而已，并没有真正的从table中删除这个key，只有当插入新的key或者rehash的时候才可能会被覆盖或者清除。

8. C#中实现一个LuaTable

参照lua5.3LuaTable的源码，在c#中实现了一个LuaTable.
地址： https://github.com/YzlCoder/LuaTable