Lua GC

原理

一个GC算法,其原理大体就是,遍历系统中的所有对象,看哪些对象没有被引用,没有引用关系的就认为是可以回收的对象进行删除操作.

这里的关键在于,如何找出没有"引用"的对象.

使用引用计数的GC算法,会在一个对象被引用的情况下将该对象的引用计数加1;反之减1.如果引用计数为0,那么就是没有被引用的对象.引用计数算法的优点是不需要扫描每一个对象,对象本身的引用计数只需要减到0自然就会被回收.缺点是会有循环引用问题.

另一种算法是"Mark & Sweep",标记及回收算法.它的原理是每一次做GC的时候首先扫描并且标记系统中的所有对象,被扫描并且标记到的对象认为是可达的(reachable),这些对象在这些GC中并不会回收,反之如果在这次扫描中都没有被标记的对象认为是可以回收的.Lua采用的就是这种算法.

早期的Lua5.0使用的是"Two-Color Mark & Sweep",两色标记及回收算法,其算法的原理是,系统中的每个对象非黑即白,也就是要么被引用要么没有被引用.我们来简单看看这个算法的伪代码:

```
每个新创建的对象颜色为白色

// 初始化阶段
遍历在root链表中的对象,加入到对象链表

// 标记阶段
当对象链表中还有未扫描的元素:
    从中取出一个对象,标记为黑色
    遍历这个对象关联的其他所有对象:
        标记为黑色

// 回收阶段
遍历所有对象:
    如果为白色:
        这些对象都是没有被引用的对象,逐个回收
    否则:
        重新加入对象链表中等待下一轮的GC检查
```

这个算法的缺陷在于,如前面所言,它是"二元"的,每个对象只可能有一种状态,不能有其他中间的状态,那么就要求这个算法每次做GC操作时不可被打断的一次性扫描并清除完毕所有的对象.

来看看这个过程不能被打断的原因.在遍历对象链表标记每个对象颜色的过程中被打断,新增了一个对象,那么

应该将这个对象标记为白色还是黑色?如果标记为白色,假如GC已经到了回收阶段,那么这个对象就会在没有遍历其关联对象的情况下被回收;如果标记为黑色,假如GC已经到了回收阶段,那么这个对象在本轮GC中并没有被扫描过就认为是不必回收的.可以看到,在双色标记扫描算法中,标记阶段和回收阶段必须合在一起完成.不能被打断,也就意味着每次GC操作的代价极大.

Lua5.1采用了在该算法基础上改进的"Tri-Color Incremental Mark & Sweep",三色增量标记及回收算法,比之前面而言,每个对象的颜色多了一种(实际在Lua中是四种,后面展开讨论),多了一种颜色状态的好处在于,它不必再要求一次GC要一次性扫描完毕所有的对象,这个GC过程可以是增量的,可以被中断再恢复继续进行的.其三种颜色的分类如下:

- White:表示当前对象为待访问状态,用于表示对象还没有被GC的标记过,这也是任何一个Lua对象在创建之后的初始状态,换言之,如果一个对象,在一个GC扫描过程完毕之后,仍然是白色的,那么说明该对象没有被系统中任何一个对象所引用,可以回收其空间了.
- Gray:表示当前对象为带扫描状态,用于表示对象已经被GC访问过,但是该对象引用的其他对象还没有被访问到.
- Black:表示当前对象为已扫描状态,用于表示对象已经被GC访问过,并且该对象引用的其他对象也已经被访问过了.

下面给出LuaGC的伪代码:

```
每个新创建的对象颜色为白色

// 初始化阶段
遍历在root节点中引用的对象从白色置为灰色,并且放入到灰色节点列表中.

// 标记阶段
当灰色链表中还有未扫描的元素:
    从中取出一个对象,标记为黑色
    遍历这个对象关联的其他所有对象:
        如果是白色:
            标记为灰色,加入灰色链表

// 回收阶段
遍历所有对象:
    如果为白色:
        这些对象都是没有被引用的对象,逐个回收
    否则:
        重新加入对象链表中等待下一轮的GC检查
```

可以看到,引入了新的灰色节点的概念之后,算法不再要求一次性完整地执行完毕,而是可以把已经扫描但是其引用的对象还未被扫描的对象置为灰色,第二部的标记阶段中,只要灰色节点集合中还有元素在,那么这个标记过程就会继续下去,即使在中间该阶段被打断转而执行其它的操作了也没有关系.

然而,即使是这样,却还有另一个没有解决的问题.从上面的算法可以看出,没有被引用的对象其颜色在一个扫描过程中始终保持不变为白色,那么假如一个对象在一个GC过程的标记阶段之后被创建,根据前面对颜色的描述,

它应该是白色的,这样在紧跟着的回收阶段,这个对象就会在没有被扫描标记的情况下被认为是没有被引用的对象而删除.因此,Lua除了前面的三色概念之外,又细分出来一个"双白色"的概念.简单的说,Lua中的白色分为"当前白色(currentwhite)"和"非当前白色(otherwhite)",这两种白色的状态交替使用,第N次GC使用的是第一种白色,那么下一次就是另外一种,以此类推,代码在回收时候会做判断,如果某个对象的白色不是此次GC回收使用的白色状态将不会被认为是没有被引用的对象而回收,这样的白色对象将留在下一次GC中进行扫描,因为在下一次GC中上一次幸免的白色将成为这次的回收颜色.

数据结构

来看Lua代码中针对GC的数据结构.

```
(lobject.h)
56 /*
57 ** Union of all Lua values
58 */
59 typedef union {
60   GCObject *gc;
61   void *p;
62   lua_Number n;
63   int b;
64 } Value;

133 /*
134 ** Union of all collectable objects
135 */
136 union GCObject {
137   GCheader gch;
138   union TString ts;
139   union Udata u;
140   union Closure cl;
141   struct Table h;
142   struct Proto p;
143   struct UpVal uv;
144   struct lua_State th;  /* thread */
145 };
```

可以看到,所有Lua中的对象都有一个GCheader的公共头部,这里面包含的数据有:

```
(lobject.h)
39 /*
40 ** Common Header for all collectable objects (in macro form, to be
41 ** included in other objects)
42 */
43 #define CommonHeader  GCObject *next; lu_byte tt; lu_byte marked
44
45
46 /*
47 ** Common header in struct form
48 */
49 typedef struct GCheader {
50   CommonHeader;
51 } GCheader;
```

其中有三个元素:

- next: GCObject链表指针,这个指针将所有GC对象都链接在一起形成链表.
- tt:数据类型.
- marked:标记字段,用于存储前面提到的几种颜色.其具体值定义如下:

```
(lgc.h)
 41 /*
 42 ** Layout for bit use in `marked' field:
 43 ** bit 0 - object is white (type 0)
 44 ** bit 1 - object is white (type 1)
 45 ** bit 2 - object is black
 46 ** bit 3 - for userdata: has been finalized
 47 ** bit 3 - for tables: has weak keys
 48 ** bit 4 - for tables: has weak values
 49 ** bit 5 - object is fixed (should not be collected)
 50 ** bit 6 - object is "super" fixed (only the main thread)
 51 */
 52
 53
 54 #define WHITE0BIT 0
 55 #define WHITE1BIT 1
 56 #define BLACKBIT  2
 57 #define FINALIZEDBIT  3
 58 #define KEYWEAKBIT  3
 59 #define VALUEWEAKBIT  4
 60 #define FIXEDBIT  5
 61 #define SFIXEDBIT 6
 62 #define WHITEBITS bit2mask(WHITE0BIT, WHITE1BIT)
```

在这里, WHITE0BIT和WHITE1BIT就是前面提到的两种白色状态.

来看看lua虚拟机的数据结构与GC相关的数据对象.

```
68 typedef struct global_State {
69   stringtable strt;  /* hash table for strings */
70   lua_Alloc frealloc;  /* function to reallocate memory */
71   void *ud;         /* auxiliary data to `frealloc' */
72   lu_byte currentwhite;
73   lu_byte gcstate;  /* state of garbage collector */
74   int sweepstrgc;  /* position of sweep in `strt' */
75   GCObject *rootgc;  /* list of all collectable objects */
76   GCObject **sweepgc;  /* position of sweep in `rootgc' */
77   GCObject *gray;  /* list of gray objects */
78   GCObject *grayagain;  /* list of objects to be traversed atomically */
79   GCObject *weak;  /* list of weak tables (to be cleared) */
80   GCObject *tmudata;  /* last element of list of userdata to be GC */
81   Mbuffer buff;  /* temporary buffer for string concatentation */
82   lu_mem GCthreshold;
83   lu_mem totalbytes;  /* number of bytes currently allocated */
84   lu_mem estimate;  /* an estimate of number of bytes actually in use */
85   lu_mem gcdept;  /* how much GC is `behind schedule' */
86   int gcpause;  /* size of pause between successive GCs */
87   int gcstepmul;  /* GC `granularity' */
88   lua_CFunction panic;  /* to be called in unprotected errors */
89   TValue l_registry;
90   struct lua_State *mainthread;
91   UpVal uvhead;  /* head of double-linked list of all open upvalues */
92   struct Table *mt[NUM_TAGS];  /* metatables for basic types */
93   TString *tmname[TM_N];  /* array with tag-method names */
94 } global_State;
```

具体流程

新创建对象

从前面的分析可以知道,每一个新创建的对象,最基本的操作,就是将对象的颜色设置为白色,意指本次GC还未扫描到的对象,同时将对象挂载到扫描过程开始会遍历的链表上.基本的思想就是如此,但是针对不同的数据类型,还有不同的处理.

一般的数据类型调用的是luaC_link函数:

```
(lgc.c)
686 void luaC_link (lua_State *L, GCObject *o, lu_byte tt) {
687   global_State *g = G(L);
688   o->gch.next = g->rootgc;
689   g->rootgc = o;
690   o->gch.marked = luaC_white(g);
691   o->gch.tt = tt;
692 }
```

这个函数做的事情很简单:

- 将对象挂载到rootgc链表上.
- 设置颜色为白色.
- 设置数据的类型.

但是Upvalue和udata类型的数据,创建过程有些不一样,首先来看Upvalue,新建一个Upvalue调用的是luaC_linkupval函数:

```
(lgc.c)
695 void luaC_linkupval (lua_State *L, UpVal *uv) {
696   global_State *g = G(L);
697   GCObject *o = obj2gco(uv);
698   o->gch.next = g->rootgc;  /* link upvalue into `rootgc' list */
699   g->rootgc = o;
700   if (isgray(o)) {
701     if (g->gcstate == GCSpropagate) {
702       gray2black(o);  /* closed upvalues need barrier */
703       luaC_barrier(L, uv, uv->v);
704     }
705     else {  /* sweep phase: sweep it (turning it into white) */
706       makewhite(g, o);
707       lua_assert(g->gcstate != GCSfinalize && g->gcstate != GCSpause);
708     }
709   }
710 }
```

这里的第一个疑问是,前面其它的数据类型,在最开始的时候,都是将颜色设置为白色的,而针对UpValue,则是根据颜色是不是灰色来做后面的一些操作.原因在于,UpValue在整个Lua虚拟机中,是存放在全局管理的数据结构中的,其中处于open状态的存放在lua_State结构体的openupval链表中:

```
(lstring.c)
 96 Udata *luaS_newudata (lua_State *L, size_t s, Table *e) {
 97   Udata *u;
 98   if (s > MAX_SIZET - sizeof(Udata))
 99     luaM_toobig(L);
100   u = cast(Udata *, luaM_malloc(L, s + sizeof(Udata)));
101   u->uv.marked = luaC_white(G(L));  /* is not finalized */
102   u->uv.tt = LUA_TUSERDATA;
103   u->uv.len = s;
104   u->uv.metatable = NULL;
105   u->uv.env = e;
106   /* chain it on udata list (after main thread) */
107   u->uv.next = G(L)->mainthread->next;
108   G(L)->mainthread->next = obj2gco(u);
109   return u;
110 }
```

初始化阶段

```
(lgc.c)
559   switch (g->gcstate) {
560     case GCSpause: {
561       markroot(L);  /* start a new collection */
562       return 0;
563     }
```

初始化阶段,将从root节点出发,遍历所有root链表上的节点,将它们的颜色从白色变成灰色,加入到gray链表中.
初始化阶段的入口是markroot函数:

```
(lgc.c)
500 /* mark root set */
501 static void markroot (lua_State *L) {
502   global_State *g = G(L);
503   g->gray = NULL;
504   g->grayagain = NULL;
505   g->weak = NULL;
506   markobject(g, g->mainthread);
507   /* make global table be traversed before main stack */
508   markvalue(g, gt(g->mainthread));
509   markvalue(g, registry(L));
510   markmt(g);
511   g->gcstate = GCSpropagate;
512 }
```

其中的markobject和markvalue函数都是用于标记对象颜色为灰色,所不同的是前者是针对object而后者是针对TValue,它们最终都会调用reallymarkobject函数:

```
(lgc.c)
69 static void reallymarkobject (global_State *g, GCObject *o) {
70   lua_assert(iswhite(o) && !isdead(g, o));
71   white2gray(o);
72   switch (o->gch.tt) {
73     case LUA_TSTRING: {
74       return;
75     }
76     case LUA_TUSERDATA: {
77       Table *mt = gco2u(o)->metatable;
78       gray2black(o);  /* udata are never gray */
79       if (mt) markobject(g, mt);
80       markobject(g, gco2u(o)->env);
81       return;
82     }
83     case LUA_TUPVAL: {
84       UpVal *uv = gco2uv(o);
85       markvalue(g, uv->v);
86       if (uv->v == &uv->u.value)  /* closed? */
87         gray2black(o);  /* open upvalues are never black */
88       return;
89     }
90     case LUA_TFUNCTION: {
91       gco2cl(o)->c.gclist = g->gray;
92       g->gray = o;
93       break;
94     }
95     case LUA_TTABLE: {
96       gco2h(o)->gclist = g->gray;
97       g->gray = o;
98       break;
99     }
100     case LUA_TTHREAD: {
101       gco2th(o)->gclist = g->gray;
102       g->gray = o;
103       break;
104     }
105     case LUA_TPROTO: {
106       gco2p(o)->gclist = g->gray;
107       g->gray = o;
108       break;
109     }
110     default: lua_assert(0);
111   }
112 }
```

可以看到,绝大部分类型的对象,在这个函数中的操作只是简单的将其改变颜色为灰色加入到gray链表中.但是

有几个类型是做区别处理的.

- 对于字符串类型数据,由于这种类型没有引用其他的数据,所以略过将其改变颜色灰色的流程,后面只要不是黑色的字符串对象直接进行回收.
- 对于udata类型数据,因为这种类型永远也不会引用其它的数据,所以这里也是一步到位直接标记为黑色,另外对于这种类型,还需要标记对应的metatable和env表.
- 对于Upvalue类型数据,如果当前是close状态的话,那么该UpValue已经没有与其它数据的引用关系了,可以直接标记为黑色.至于open状态的Upvalue,由于open状态的Upvalue,其引用状态可能会发生频繁的变动,所以留待后面的remarkupvals函数中进行原子性的标记,这在后面会做解释.

另外需要注意的是,这里并没有针对这个对象所引用的对象递归调用reallymarkobject函数进行递归调用,比如Table类型的对象就没有遍历它的Key和Value数据进行mark,而在udata中要标记metable和env表,是因为除了这里之外并没有直接能访问到它的其它地方了.并没有递归去标记引用对象的原因,是想让这个标记过程尽量的快.

扫描标记阶段

```
(lgc.c)
564     case GCSpropagate: {
565       if (g->gray)
566         return propagatemark(g);
567       else {  /* no more `gray' objects */
568         atomic(L);  /* finish mark phase */
569         return 0;
570       }
571     }
```

在这一步,将扫描所有gray链表中的对象,将它们以及所引用到的对象标记成黑色.需要注意的是,前面的初始化阶段是一次性到位的,而这一步却是可以多次进行,每次扫描之后会返回本次扫描标记的对象大小之和,其入口函数是propagatemark,下一次再次扫描时,只要gray链表中还有待扫描的对象,就继续执行这个函数进行标记.

这个函数与前面的reallymarkobject函数,做的事情其实差不多,都是对一个对象进行标记颜色的动作,区别在于,这里将对象从灰色标记成黑色,表示这个对象及其所引用的对象都已经被标记过,另一个区别在于,前面的流程不会递归对一个对象所引用的对象进行标记,而这里会根据不同的类型调用对应的traverse*函数遍历引用到的对象进行标记.实际工作中,里面的每种类型的对象,处理还不太一样,下面来看看:

- 如果是表对象,并且是一个弱表,那么需要将颜色从黑色回退到灰色去,还需要再做扫描.
- 如果是线程对象,也是从黑色回退到灰色去,同时是加入到grayagain链表中,需要在后面原子性的做扫描标记,这个过程不可被打断.
- 其他的对象都是正常的扫描标记流程.

当gray链表中的所有对象都被处理完毕,此时还不能立即进入下一个流程进行回收操作,因为在前面的流程中,

到了这里可以谈谈barrier操作了.

从前面的描述可以知道,分步增量式的扫描标记算法,由于中间可以被打断执行其他操作,那么就会出现新增加的对象与已经被扫描过的对象之间会有引用关系的变化,而算法中需要保证不会出现有黑色的对象引用的对象中有白色对象的情况,于是需要两种不同的处理:

- 标记过程向前走一步,这种情况指的是,如果有一个新创建的对象,其颜色是白色,而它被一个黑色对象引用了,那么将这个对象的颜色从白色变成灰色,也就是在这个GC过程中的进度向前走了一步.
- 标记过程向后走一步,与前面的情况一样,但是此时是将黑色的对象回退到灰色,也就是这个原先已经被标记为黑色的对象需要重新被扫描,这相当于在GC过程中向后走了一步.

在代码中,最终调用luaC_barrierf函数的,都是向前走的操作;反之,调用luaC_barrierback的操作则是向后走的操作:

```
(lgc.h)
86 #define luaC_barrier(L,p,v) { if (valiswhite(v) && isblack(obj2gco(p)))  \
87     luaC_barrierf(L,obj2gco(p),gcvalue(v)); }
88
89 #define luaC_barriert(L,t,v) { if (valiswhite(v) && isblack(obj2gco(t)))  \
90     luaC_barrierback(L,t); }
91
92 #define luaC_objbarrier(L,p,o)  \
93     { if (iswhite(obj2gco(o)) && isblack(obj2gco(p))) \
94         luaC_barrierf(L,obj2gco(p),obj2gco(o)); }
95
96 #define luaC_objbarriert(L,t,o)  \
97    { if (iswhite(obj2gco(o)) && isblack(obj2gco(t))) luaC_barrierback(L,t); }
```

可以看到的是,回退操作仅针对的Table类型的对象,而对于其他类型的对象都是向前操作,我们来看看这么做的原因.

Table是Lua中最常见的数据结构,而且一个Table与其关联的Key,Value之间是1比N的对应关系.如果针对Table对象做的是向前的标记操作,那么就意味着,但凡一个Table,只要有新增的对象,都需要将这个新的对象标记为灰色加入gray链表中等待扫描,实际上这样会有不必要的开销.所以针对Table类型的对象,使用的是针对该Table对象本身要做的向后的操作,这样不论有多少个对象新增到这个Table中,只要改变了一次就将这个Table对象回退到灰色状态,等待重新的扫描.但是这里需要注意的是,对Table对象进行回退操作时,并不是将它放入gray链表中,因为这样子做实际上还是会出现前面提到的多次反复标记的问题,针对Table对象,对它执行回退操作时是将它加入到grayagain链表中,用于在扫描完毕gray链表之后再一次性的原子扫描:

```
(lgc.c)
675 void luaC_barrierback (lua_State *L, Table *t) {
676   global_State *g = G(L);
677   GCObject *o = obj2gco(t);
678   lua_assert(isblack(o) && !isdead(g, o));
679   lua_assert(g->gcstate != GCSfinalize && g->gcstate != GCSpause);
680   black2gray(o);  /* make table gray (again) */
681   t->gclist = g->grayagain;
682   g->grayagain = o;
683 }
```

可以看到的是,需要进行barrierback操作的对象,最后并没有如新建对象那样加入gray链表中,而是加入grayagain列表中,避免一个对象频繁的被回退-扫描,grayagain链表下面将进行说明.

而相对的,向前的操作就简单多了:

```
(lgc.c)
662 void luaC_barrierf (lua_State *L, GCObject *o, GCObject *v) {
663   global_State *g = G(L);
664   lua_assert(isblack(o) && iswhite(v) && !isdead(g, v) && !isdead(g, o));
665   lua_assert(g->gcstate != GCSfinalize && g->gcstate != GCSpause);
666   lua_assert(ttype(&o->gch) != LUA_TTABLE);
667   /* must keep invariant? */
668   if (g->gcstate == GCSpropagate)
669     reallymarkobject(g, v);  /* restore invariant */
670   else  /* don't mind */
671     makewhite(g, o);  /* mark as white just to avoid other barriers */
672 }
```

这里只要当前的GC没有在扫描标记阶段,那么就标记这个对象,否则将对象标记为白色,等待下一次的GC.

当gray链表中没有对象的时候,还不能马上进入下一个阶段,因为前面还有未处理的数据,这一步需要原子不被中断的完成,其入口是atomic函数.前面提到Lua的增量式GC算法分为多个阶段,可以被中断,然而这一步则例外,这一步将处理弱表链表和前面提到的grayagain链表,是扫描阶段的最后一步,不可被中断:

```
(lgc.c)
525 static void atomic (lua_State *L) {
526   global_State *g = G(L);
527   size_t udsize;  /* total size of userdata to be finalized */
528   /* remark occasional upvalues of (maybe) dead threads */
529   remarkupvals(g);
530   /* traverse objects cautch by write barrier and by 'remarkupvals' */
531   propagateall(g);
532   /* remark weak tables */
533   g->gray = g->weak;
534   g->weak = NULL;
535   lua_assert(!iswhite(obj2gco(g->mainthread)));
536   markobject(g, L);  /* mark running thread */
537   markmt(g);  /* mark basic metatables (again) */
538   propagateall(g);
539   /* remark gray again */
540   g->gray = g->grayagain;
541   g->grayagain = NULL;
542   propagateall(g);
543   udsize = luaC_separateudata(L, 0);  /* separate userdata to be finalized */
544   marktmu(g);  /* mark `preserved' userdata */
545   udsize += propagateall(g);  /* remark, to propagate `preserveness' */
546   cleartable(g->weak);  /* remove collected objects from weak tables */
547   /* flip current white */
548   g->currentwhite = cast_byte(otherwhite(g));
549   g->sweepstrgc = 0;
550   g->sweepgc = &g->rootgc;
551   g->gcstate = GCSsweepstring;
552   g->estimate = g->totalbytes - udsize;  /* first estimate */
553 }
```

这个函数中,分别做了以下的几个操作:

- 调用remarkupvals函数去标记open状态的Upvalue,这一步完毕之后gray链表又会有新的对象,于是需要调用propagateall再次将gray链表中的对象标记以下.
- 修改gray链表指针指向管理弱表的weak指针,同时标记当前的Lua_State指针,以及基本的meta表.
- 修改gray链表指针指向grayagain指针,同样是调用propagateall函数进行遍历扫描操作.
- 修改状态到下个回收阶段.

```
(lgc.c)
128 size_t luaC_separateudata (lua_State *L, int all) {
129   global_State *g = G(L);
130   size_t deadmem = 0;
131   GCObject **p = &g->mainthread->next;
132   GCObject *curr;
133   while ((curr = *p) != NULL) {
134     if (!(iswhite(curr) || all) || isfinalized(gco2u(curr)))
135       p = &curr->gch.next;  /* don't bother with them */
136     else if (fasttm(L, gco2u(curr)->metatable, TM_GC) == NULL) {
137       markfinalized(gco2u(curr));  /* don't need finalization */
138       p = &curr->gch.next;
139     }
140     else {  /* must call its gc method */
141       deadmem += sizeudata(gco2u(curr));
142       markfinalized(gco2u(curr));
143       *p = curr->gch.next;
144       /* link `curr' at the end of `tmudata' list */
145       if (g->tmudata == NULL)  /* list is empty? */
146         g->tmudata = curr->gch.next = curr;  /* creates a circular list */
147       else {
148         curr->gch.next = g->tmudata->gch.next;
149         g->tmudata->gch.next = curr;
150         g->tmudata = curr;
151       }
152     }
153   }
154   return deadmem;
155 }
```

回收阶段

回收阶段又分了两步,一个是针对字符串类型的回收,一个则是针对其他类型对象的回收:

```
(lgc.c)
572     case GCSsweepstring: {
573       lu_mem old = g->totalbytes;
574       sweepwholelist(L, &g->strt.hash[g->sweepstrgc++]);
575       if (g->sweepstrgc >= g->strt.size)  /* nothing more to sweep? */
576         g->gcstate = GCSsweep;  /* end sweep-string phase */
577       lua_assert(old >= g->totalbytes);
578       g->estimate -= old - g->totalbytes;
579       return GCSWEEPCOST;
580     }
581     case GCSsweep: {
582       lu_mem old = g->totalbytes;
583       g->sweepgc = sweeplist(L, g->sweepgc, GCSWEEPMAX);
584       if (*g->sweepgc == NULL) {  /* nothing more to sweep? */
585         checkSizes(L);
586         g->gcstate = GCSfinalize;  /* end sweep phase */
587       }
588       lua_assert(old >= g->totalbytes);
589       g->estimate -= old - g->totalbytes;
590       return GCSWEEPMAX*GCSWEEPCOST;
591     }
```

针对字符串类型数据,每次调用sweepwholelist函数回收字符串hash桶数组中的一个字符串链表,当所有字符串hash桶数据全部遍历完毕,切换到下一个状态GCSsweep进行其他数据的回收;针对其他数据的回收,则是调用sweeplist函数进行.

```
(lgc.c)
407 static GCObject **sweeplist (lua_State *L, GCObject **p, lu_mem count) {
408   GCObject *curr;
409   global_State *g = G(L);
410   int deadmask = otherwhite(g);
411   while ((curr = *p) != NULL && count-- > 0) {
412     if (curr->gch.tt == LUA_TTHREAD)  /* sweep open upvalues of each thread */
413       sweepwholelist(L, &gco2th(curr)->openupval);
414     if ((curr->gch.marked ^ WHITEBITS) & deadmask) {  /* not dead? */
415       lua_assert(!isdead(g, curr) || testbit(curr->gch.marked, FIXEDBIT));
416       makewhite(g, curr);  /* make it white (for next cycle) */
417       p = &curr->gch.next;
418     }
419     else {  /* must erase `curr' */
420       lua_assert(isdead(g, curr) || deadmask == bitmask(SFIXEDBIT));
421       *p = curr->gch.next;
422       if (curr == g->rootgc)  /* is the first element of the list? */
423         g->rootgc = curr->gch.next;  /* adjust first */
424       freeobj(L, curr);
425     }
426   }
427   return p;
428 }
```

可以看到,在sweeplist中,首先拿到otherwhite,这个表示本次GC操作不可以被回收的白色类型,后面就是依次遍历链表中的数据,判断每个对象的白色是否满足被回收的颜色条件.

结束阶段

```
(lgc.c)
592     case GCSfinalize: {
593       if (g->tmudata) {
594         GCTM(L);
595         if (g->estimate > GCFINALIZECOST)
596           g->estimate -= GCFINALIZECOST;
597         return GCFINALIZECOST;
598       }
599       else {
600         g->gcstate = GCSpause;  /* end collection */
601         g->gcdept = 0;
602         return 0;
603       }
604     }
```