杂项

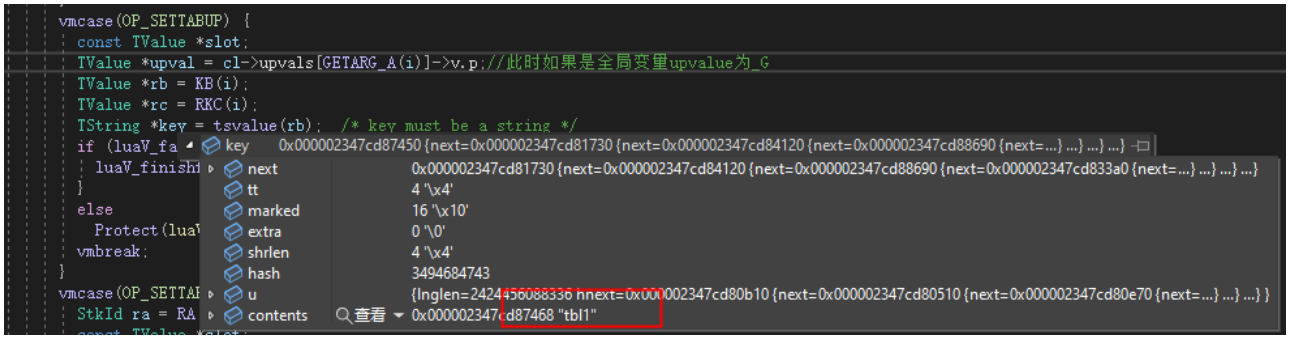
- lua解析器生成proto后，会创建一个LClosure(lua闭包)，然后把这个LClosure对象放到栈底（L->top），并且把_G设置为这个lua闭包的upvalue。如下图所示：

```
LUA_API int lua_load (lua_State *L, lua_Reader reader, void *data,
                      const char *chunkname, const char *mode) {
  ZIO z;
  int status;
  lua_lock(L);
  if (!chunkname) chunkname = "?";
  luaZ_init(L, &z, reader, data);
  status = luaD_protectedparser(L, &z, chunkname, mode);
  if (status == LUA_OK) {  /* no errors? */
    LClosure *f = clLvalue(s2v(L->top.p - 1));  /* get new function */ 已用时间 <= 2ms
    if (f->nupvalues >= 1) {  /* does it have an upvalue? */
      /* get global table from registry */
      const TValue *gt = getGtable(L);  //(&hvalue(&G(L)->l_registry)->array[LUA_RIDX_GLOBALS - 1])，l_registry即注册表（这是一个table）,LUA_RIDX_GLOBALS这也是一个table,
      //并且lua代码里的全局变量都放在这个table里
      /* set global table as 1st upvalue of 'f' (may be LUA_ENV) */
      setobj(L, f->upvals[0]->v.p, gt);
      luaC_barrier(L, f->upvals[0], gt);
    }
  }
  lua_unlock(L);
  return status;
}
```

此时只是解析了lua代码，lua的全局变量或者lua函数对象（LClosure）都没有创建，需要调用lua_pcall,如下图所示：

```
LUA_API int lua_pcallk (lua_State *L, int nargs, int nresults, int errfunc,
                        lua_KContext ctx, lua_KFunction k) {
  struct CallS c;
  int status;
  ptrdiff_t func;
  lua_lock(L);
  api_check(L, k == NULL || !isLua(L->ci),
    "cannot use continuations inside hooks");
  api_checknelems(L, nargs+1);
  api_check(L, L->status == LUA_OK, "cannot do calls on non-normal thread");
  checkresults(L, nargs, nresults);
  if (errfunc == 0)
    func = 0;
  else {
    StkId o = index2stack(L, errfunc);
    api_check(L, ttisfunction(s2v(o)), "error handler must be a function");
    func = savestack(L, o);
  }
  c.func = L->top.p - (nargs+1);  /* function to be called */   //此时的栈底存的是之前解析lua代码并创建的proto对象
  if (k == NULL || !yieldable(L)) {  /* no continuation or no yieldable? */
    c.nresults = nresults;  /* do a 'conventional' protected call */
    status = luaD_pcall(L, f_call, &c, savestack(L, c.func), func);
  }
  else {  /* prepare continuation (call is already protected by 'resume') */
    CallInfo *ci = L->ci;
    ci->u.c.k = k;  /* save continuation */
    ci->u.c.ctx = ctx;  /* save context */
    /* save information for error recovery */
    ci->u2.funcidx = cast_int(savestack(L, c.func));
    ci->u.c.old_errfunc = L->errfunc;
    L->errfunc = func;
    setoah(ci->callstatus, L->allowhook);  /* save value of 'allowhook' */
    ci->callstatus |= CIST_YPCALL;  /* function can do error recovery */
    luaD_call(L, c.func, nresults);  /* do the call */
    ci->callstatus &= ~CIST_YPCALL;
    L->errfunc = ci->u.c.old_errfunc;
    status = LUA_OK;  /* if it is here, there were no errors */
  }
  adjustresults(L, nresults);
  lua_unlock(L);
  return status;
}
```

此时全局对象以及lua函数对象会被创建并且以变量名为key放到Table（_G）中，如下图所示：



- 一个函数的upvalue解释在网上如下图所示，但其实时有问题的，如果一个函数不访问全局变量并且从未使用lua关键字（也就是不需要访问放在global_state的string数据）时，此时创建LClosure对象的upvaluesize为0。

  我们可以看到，每个lua函数，都有一个upvalue列表，并且他们首个upvalue，都是一个名为_ENV的upvalue，内层lua函数的_ENV指向外层lua函数的_ENV，而最外层的top-level函数，则将值指向了全局表_G。为什么lua要用这种组织方式？将_ENV作为每个lua函数的第0个upvalue呢？我认为，这是为了效率，同时也能是的逻辑更为清晰，lua函数去查找一个变量的方式，如下所示：

  - lua函数查找一个变量v，首先会在自己的local变量中查找，如果找到就直接获取它的值，找不到则进入下一步
  - 查找upvalue列表，有没有一个名为v的upvalue，有则获取它的值，没有则进入下一步
  - 到_ENV里去查找一个名为v的值

- lua闭包upvalue最大数量为255，因为闭包结构体中用了一个字节大小的变量来保存upvalue的数量。open upvalue是指该upvalue指向栈，当栈回缩后，open upvalue会变成close upvalue。如下图所示：

```c
/*
** Upvalues for Lua closures
*/
typedef struct UpVal {
  CommonHeader;
  union {
    TValue *p;  /* points to stack or to its own value *///指向L->stack时，该UpVal就是open upvalue，如果指向value时，该UpVal就是close upvalue
    ptrdiff_t offset;  /* used while the stack is being reallocated */
  } v;
  union {
    struct {  /* (when open) */
      struct UpVal *next;  /* linked list */
      struct UpVal **previous;
    } open;
    TValue value;  /* the value (when closed) */
  } u;
} UpVal;



#define ClosureHeader \
    CommonHeader; lu_byte nupvalues;/*1个字节 最大255*/ GCObject *gclist
```

```c
/*
** Close all upvalues up to the given stack level.
*/
void luaF_closeupval (lua_State *L, StkId level) {
  UpVal *uv;
  StkId upl;  /* stack index pointed by 'uv' */
  while ((uv = L->openupval) != NULL && (upl = uplevel(uv)) >= level) {
    //upvalue本身保存对象的地址
    TValue *slot = &uv->u.value;  /* new position for value */
    lua_assert(uplevel(uv) < L->top.p);
    luaF_unlinkupval(uv);  /* remove upvalue from 'openupval' list */
    //将upvalue->v.p指向本身（即upvalue->u.value）
    setobj(L, slot, uv->v.p);  /* move value to upvalue slot */
    uv->v.p = slot;  /* now current value lives here */
    if (!iswhite(uv)) {  /* neither white nor dead? */
      nw2black(uv);  /* closed upvalues cannot be gray */
      luaC_barrier(L, uv, slot);
    }
  }
}
```

TString

- 小于40的单独创建对象主要是为了性能考虑，如果全放在global_State中的strt中(strt的数据类型为stringtable)，长度过长时性能较差。strt扩容时是2倍扩容，在GC的GCSswpend步会判断是否需要收缩strt大小，收缩也是缩小一倍。
- ```c
  typedef struct stringtable {
    TString **hash;
    int nuse;  /* number of elements */
    int size;
  } stringtable;
  ```
  ```

```c
/*
** Header for a string value.
*/
typedef struct TString {
  CommonHeader;
  lu_byte extra;  /* reserved words for short strings; "has hash" for longs */
  lu_byte shrlen;  /* length for short strings */
  unsigned int hash;
  union {
    size_t lnglen;  /* length for long strings */
    struct TString *hnext;  /* linked list for hash table */
  } u;
  char contents[1];
} TString;
```

```c
/*
** new string (with explicit length)
*/
TString *luaS_newlstr (lua_State *L, const char *str, size_t l) {
  //长度小于等于40的放在g->strt
  if (l <= LUAI_MAXSHORTLEN)  /* short string? */
    return internshrstr(L, str, l);
  else {
    TString *ts;
    if (l_unlikely(l >= (MAX_SIZE - sizeof(TString))/sizeof(char)))
      luaM_toobig(L);
    ts = luaS_createlngstrobj(L, l);
    memcpy(getstr(ts), str, l * sizeof(char));
    return ts;
  }
}
```

ipairs和pairs差异

- ipairs:从1开始累加，只查找本身数组以及元表（元表得有index元方法）数组部分，当value为nil的时候退出。
- pairs:如果table的元表有pairs元方法则执行该函数做查找，否则遍历该表的数组以及hash。

table操作

- table新增元素：首先查找是否有元表，如果没有元表则直接往table里新增元素，如果有元表再查找是否有newindex元方法，如果没有则往元表中新增元素，如果有并且是函数则执行该函数，如果是一个表则重复执行上序步骤。
- table查找元素：首先在表中查找，如果没有则看该表是否有元表，没有则返回nil，如果有元表但没有index元方法则返回nil，如果index元方法是函数，则实现该函数，如果是个table，则重复上序步骤。

lua注册c函数

1. 编写 C 函数

首先，你需要一个 C 函数，它将作为 Lua 函数使用。例如：

```c
#include <lua.h>
#include <lauxlib.h>
#include <lualib.h>

// C 函数，它接收一个参数并返回其两倍
static int double_value(lua_State *L) {
    double value = luaL_checknumber(L, 1); // 获取第一个参数（索引为1）
    lua_pushnumber(L, value * 2); // 将结果推入堆栈
    return 1; // 返回推入堆栈的元素数量
}
```

2. 注册 C 函数

你可以通过静态或动态方式注册这个函数。

静态注册

在静态注册中，你通常在 Lua 的 C API 初始化函数中注册函数。这通常在包含 `luaL_openlib` 的地方完成，如下例所示：

```c
static const struct luaL_Reg mylib[] = {
    {"double", double_value},
    {NULL, NULL} /* 列表结束 */
};

LUALIB_API int luaopen_mylib(lua_State *L) {
    luaL_newlib(L, mylib);
    return 1;
}
```

然后在 Lua 脚本中，你可以通过 `require` 来加载和使用这个库：

```lua
local mylib = require("mylib")
print(mylib.double(5)) -- 输出 10
```

动态注册

在动态注册中，你可以在任何时候将函数注册到 Lua 状态机中：

```c
// 假设 lua_State *L 已经被初始化
lua_pushcfunction(L, double_value); // 将 C 函数推入堆栈
lua_setglobal(L, "double_value"); // 设置全局变量 "double_value"
```

现在，在 Lua 脚本中，你可以直接调用这个函数：

```lua
print(double_value(5)) -- 输出 10
```

LuaGC

- GC相关字段，其中某些字段分属于不同的GC模式

```
typedef struct global_State{
    ...
    l_mem totalbytes;  /* number of bytes currently allocated - GCdebt */

    l_mem GCdebt;  /* bytes allocated not yet compensated by the collector */GC债务，申请或释放需要gc的对象时会加减该字段的值。但当每次gc完成后，会重新设置这个值。当该字段大于0才会触发GC。

    lu_mem GCestimate;  /* an estimate of the non-garbage memory in use */

    lu_byte genminormul;  /* control for minor generational collections */部分分代gc内存百分比，默认为百分之20（GCestimate）

    lu_byte genmajormul;  /* control for major generational collections */全量分代gc内存百分比，默认为百分百（GCestimate）

    GCObject *allgc;  /* list of all collectable objects */ 所有需要GC的对象列表（自带终结器的对象不在该列表）

    GCObject *finobj;  /* list of collectable objects with finalizers */自带终结器的对象列表，一个新对象首先会放在allgc列表，
    当这个对象是table或者userdata并且给该对象设置元表时，如果有__gc函数则会把改对象从allgc移除并放到finobj列表中。

    GCObject *tobefnz;  /* list of userdata to be GC */自带终结器且需要被回收的对象列表

    GCObject *fixedgc;  /* list of objects not to be collected */自带终结器且不需要被回收的对象列表
```

```
    ...
} global_State;
```

- 增程式GC

```
typedef struct global_State{
    lu_byte gcstepmul;  /* GC "speed" */Step倍数，默认为100

    lu_byte gcstepsize;  /* (log2 of) GC granularity */Step大小，默认为8KB

    lu_byte gcpause;  /* size of pause between successive GCs */

    GCObject *gray;  /* list of gray objects */灰色链表

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

    GCObject *weak;  /* list of tables with weak values */

    GCObject *ephemeron;  /* list of ephemeron tables (weak keys) */

    GCObject *allweak;  /* list of all-weak tables */

    struct lua_State *mainthread;

    TValue l_registry;
}
```

```
/*
** Performs a basic incremental step. The debt and step size are
** converted from bytes to "units of work"; then the function loops
** running single steps until adding that many units of work or
** finishing a cycle (pause state). Finally, it sets the debt that
** controls when next step will be performed.
*/
static void incstep (lua_State *L, global_State *g) {
  int stepmul = (getgcparam(g->gcstepmul) | 1);  /* avoid division by 0 *///默认为101
  l_mem debt = (g->GCdebt / WORK2MEM) * stepmul;//当前负债的TValue个数
  l_mem stepsize = (g->gcstepsize <= log2maxs(l_mem))
                 ? ((cast(l_mem, 1) << g->gcstepsize) / WORK2MEM) * stepmul
                 : MAX_LMEM;  /* overflow; keep maximum value *///Step的个数，默认为8KB/sizeof(TValue)*101
  do {  /* repeat until pause or enough "credit" (negative debt) */
    lu_mem work = singlestep(L);  /* perform one single step */
    debt -= work;
  } while (debt > -stepsize && g->gcstate != GCSpause);
  if (g->gcstate == GCSpause)
    setpause(g);  /* pause until next cycle */
  else {
    debt = (debt / stepmul) * WORK2MEM;  /* convert 'work units' to bytes */
    luaE_setdebt(g, debt);
  }
}
```

```c
/*
** Set the "time" to wait before starting a new GC cycle; cycle will
** start when memory use hits the threshold of ('estimate' * pause /
** PAUSEADJ). (Division by 'estimate' should be OK: it cannot be zero,
** because Lua cannot even start with less than PAUSEADJ bytes).
*/
static void setpause (global_State *g) {
  l_mem threshold, debt;
  int pause = getgcparam(g->gcpause);
  l_mem estimate = g->GCestimate / PAUSEADJ;  /* adjust 'estimate' */
  lua_assert(estimate > 0);
  threshold = (pause < MAX_LMEM / estimate)  /* overflow? */
            ? estimate * pause  /* no overflow */
            : MAX_LMEM;  /* overflow; truncate to maximum */    //默认为当前所占内存的2倍
  debt = gettotalbytes(g) - threshold;
  if (debt > 0) debt = 0;
  luaE_setdebt(g, debt);
}
```

GCSpause（暂停）->GCSpropagate（扩散）->GCSenteratomic（原子步骤）->GCSswpallgc（sweep allgc列表）->GCSswpfinobj（sweep finobj列表）->GCSswptobefnz（sweep tobefnz列表）->GCSswpend（finish sweep）->GCScallfin（call finish）->GCSpause（暂停）

每次最少处理stepsize数量的TValue（不一定会回收），当执行完一次完整GC后，会调用setpause来设置下次触发GC的debt（默认为当前所占内存的2倍）。

GCSpause：清理灰色链表并且标记根节点（单步）。 GCSpropagate：从灰色链表中一个个取出处理（做标记或者放入灰色链表）（多步）。 GCSenteratomic： 1、首先确保gray链表为空,不为空则遍历gray上所有元素 2、置当前GC状态为GCSinsideatomic 3、标记当前运行的线程 4、标记注册表 5、标记基本类型的元表 6、标记上值: 1:若线程不为灰色或没有上值，则从有开放上值的线程链表(twups)中移除，并标记所有触碰过的上值 7、再次遍历gray上所有元素 8、遍历grayagain链接上所有元素 9、循环遍历浮游链表上弱key表(因为上面的步骤可能导致key变为黑色),直到没有需要标记的结点,最后浮游链表上的元素部分仍然是之前链表上的元素 --------- 至此所有可被访问的强对象都被标记了 ---------- 10、清理weak链表上弱表中可能需要被清理的值 11、清理allweak链表上弱表中的可能需要被清理的值 12、把finobj链表上没有被标记的对象移动到tobefnz链表上 13、标记tobefnz链表上的元素 14、再次遍历gray上所有元素 15、执行第9步 --------- 至此所有复活的对象都被标记了 ---------- 16、清理ephemeron链表上弱表中的可能需要被清理的key 17、清理allweak链表上弱表中的可能需要被清理的key 18、清理weak链表上12步之后新增弱表中的可能需要被清理的value 19、清理allweak链表上12步之后新增弱表中的可能需要被清理的value 20、清理字符串缓存 21、切换白色
22、进入清理阶段，用sweepgc记录下次该清理哪个元素 GCSswpallgc：处理sweepgc列表，每次数量GCSWEEPMAX（100）个对象，此时sweepgc指向allgc列表。 GCSswpfinobj：处理sweepgc列表，每次数量GCSWEEPMAX（100）个对象，此时sweepgc指向finobj列表，此时只是将 finobj列表的颜色改成另外一种白色。 GCSswptobefnz：处理sweepgc列表，每次数量GCSWEEPMAX（100）个对象，此时sweepgc指向tobefnz列表，此时只是将tobefnz列表的颜色改成另外一种白色。这是因为在atomic阶段中会把tobefnz列表中的userdata元素设置为黑色。 GCSswpend：调整string table。如下所示：

```c
/*
** If possible, shrink string table.
*/
static void checkSizes (lua_State *L, global_State *g) {
  if (!g->gcemergency) {
    if (g->strt.nuse < g->strt.size / 4) {  /* string table too big? */
      l_mem olddebt = g->GCdebt;
      luaS_resize(L, g->strt.size / 2);
      g->GCestimate += g->GCdebt - olddebt;  /* correct estimate */
    }
  }
}
```

GCScallfin：处理tobefnz列表，将tobefnz中的元素加到allgc列表，然后调用对象对应的gc函数，调用完后并清理gc函数，避免多次调用。

- 分代式GC

```c
** Does a generational "step".
** Usually, this means doing a minor collection and setting the debt to
** make another collection when memory grows 'genminormul'% larger.
**
** However, there are exceptions.  If memory grows 'genmajormul'%
** larger than it was at the end of the last major collection (kept
** in 'g->GCestimate'), the function does a major collection. At the
** end, it checks whether the major collection was able to free a
** decent amount of memory (at least half the growth in memory since
** previous major collection). If so, the collector keeps its state,
** and the next collection will probably be minor again. Otherwise,
** we have what we call a "bad collection". In that case, set the field
** 'g->lastatomic' to signal that fact, so that the next collection will
** go to 'stepgenfull'.
**
** 'GCdebt <= 0' means an explicit call to GC step with "size" zero;
** in that case, do a minor collection.
*/
static void genstep (lua_State *L, global_State *g) {
  if (g->lastatomic != 0)  /* last collection was a bad one? */
    stepgenfull(L, g);  /* do a full step */
  else {
    lu_mem majorbase = g->GCestimate;  /* memory after last major collection */
    lu_mem majorinc = (majorbase / 100) * getgcparam(g->genmajormul);
    if (g->GCdebt > 0 && gettotalbytes(g) > majorbase + majorinc) { //2倍majorbase
      lu_mem numobjs = fullgen(L, g);  /* do a major collection */
      //如果fullgen回收少于GCestimate的一半时，下次时bad collection，执行stepgenfull
      if (gettotalbytes(g) < majorbase + (majorinc / 2)) {
        /* collected at least half of memory growth since last major
           collection; keep doing minor collections. */
        lua_assert(g->lastatomic == 0);
      }
      else {  /* bad collection */
        g->lastatomic = numobjs;  /* signal that last collection was bad */
        setpause(g);  /* do a long wait for next (major) collection */
      }
    }
    else {  /* regular case; do a minor collection */
      youngcollection(L, g);
      setminordebt(g);
      g->GCestimate = majorbase;  /* preserve base value */
    }
  }
  lua_assert(isdecGCmodegen(g));
}
```

如果前一次是bad collection（即g->lastatomic（上次gc的对象个数）），则执行stepgenfull。否则去判断gcdebt是否大于0并且当前内存（gettotalbytes(g)）是否比上次major collection后的剩余内存两倍要大，

满足条件则执行fullgen（返回值为当前对象的个数），否则做young collection。fullgen源码如下所示：

```
/*
** Does a full collection in generational mode.
*/
static lu_mem fullgen (lua_State *L, global_State *g) {
  enterinc(g);  //将所有object设为白色并初始化相关列表以及初始化GC状态（复用增程式GC的代码）
  return entergen(L, g);
}
```

```
/*
** Enter incremental mode. Turn all objects white, make all
** intermediate lists point to NULL (to avoid invalid pointers),
** and go to the pause state.
*/
static void enterinc (global_State *g) {
  whitelist(g, g->allgc);
  g->reallyold = g->old1 = g->survival = NULL;
  whitelist(g, g->finobj);
  whitelist(g, g->tobefnz);
  g->finobjrold = g->finobjold1 = g->finobjsur = NULL;
  g->gcstate = GCSpause;
  g->gckind = KGC_INC;
  g->lastatomic = 0;
}
```

```
/*
** Enter generational mode. Must go until the end of an atomic cycle
** to ensure that all objects are correctly marked and weak tables
** are cleared. Then, turn all objects into old and finishes the
** collection.
*/
static lu_mem entergen (lua_State *L, global_State *g) {
  lu_mem numobjs;
  //执行增程式GC的GCSpause以及GCSatomic步骤
  luaC_runtilstate(L, bitmask(GCSpause));  /* prepare to start a new cycle */
  luaC_runtilstate(L, bitmask(GCSpropagate));  /* start new cycle */
  numobjs = atomic(L);  /* propagates all and then do the atomic stuff */

  //遍历所有对象如果是白色则free，否则设为old
  atomic2gen(L, g);

  //设置minor债务，当前所占内存的百分之20
  setminordebt(g);  /* set debt assuming next cycle will be minor */
  return numobjs;
}
```

```c
/*
** Clears all gray lists, sweeps objects, and prepare sublists to enter
** generational mode. The sweeps remove dead objects and turn all
** surviving objects to old. Threads go back to 'grayagain'; everything
** else is turned black (not in any gray list).
*/
static void atomic2gen (lua_State *L, global_State *g) {
  cleargraylists(g);
  /* sweep all elements making them old */
  g->gcstate = GCSswpallgc;
  sweep2old(L, &g->allgc);
  /* everything alive now is old */
  g->reallyold = g->old1 = g->survival = g->allgc;
  g->firstold1 = NULL;  /* there are no OLD1 objects anywhere */

  /* repeat for 'finobj' lists */
  sweep2old(L, &g->finobj);
  g->finobjrold = g->finobjold1 = g->finobjsur = g->finobj;

  sweep2old(L, &g->tobefnz);

  g->gckind = KGC_GEN;
  g->lastatomic = 0;
  g->GCestimate = gettotalbytes(g);  /* base for memory control */
  finishgencycle(L, g);
}
```

```c
/*
** Set debt for the next minor collection, which will happen when
** memory grows 'genminormul'%.
*/
static void setminordebt (global_State *g) {
  luaE_setdebt(g, -(cast(l_mem, (gettotalbytes(g) / 100)) * g->genminormul));//g->genminormul=20
}
```

stepgenfull源码如下所示：

```c
static void stepgenfull (lua_State *L, global_State *g) {
  lu_mem newatomic;  /* count of traversed objects */
  lu_mem lastatomic = g->lastatomic;  /* count from last collection */
  if (g->gckind == KGC_GEN)  /* still in generational mode? */
    enterinc(g);  /* enter incremental mode *///将所有object设为白色并初始化相关列表以及初始化GC状态（复用增程式GC的代码）
  luaC_runtilstate(L, bitmask(GCSpropagate));  /* start new cycle *///执行增程式GC的GCSpause步骤
  newatomic = atomic(L);  /* mark everybody *///类似执行增程式GC的GCSenteratomic步骤
  //新增的对象数量少于原存活下来的对象的 1/8
  if (newatomic < lastatomic + (lastatomic >> 3)) {  /* good collection? */
    //遍历所有对象如果是白色则free，否则设为old
    atomic2gen(L, g);  /* return to generational mode */
    //设置minor债务，当前所占内存的百分之20
    setminordebt(g);
  }
  else {  /* another bad collection; stay in incremental mode */
    //跑完一次完整的增程式GC
    g->GCestimate = gettotalbytes(g);  /* first estimate */;
    entersweep(L);
    luaC_runtilstate(L, bitmask(GCSpause));  /* finish collection */
    setpause(g);
    g->lastatomic = newatomic;
  }
}
```

youngcollection源码如下所示：

```
/*
** Does a young collection. First, mark 'OLD1' objects. Then does the
** atomic step. Then, sweep all lists and advance pointers. Finally,
** finish the collection.
*/
static void youngcollection (lua_State *L, global_State *g) {
  GCObject **psurvival;  /* to point to first non-dead survival object */
  GCObject *dummy;  /* dummy out parameter to 'sweepgen' */
  lua_assert(g->gcstate == GCSpropagate);
  if (g->firstold1) {  /* are there regular OLD1 objects? */
    markold(g, g->firstold1, g->reallyold);  /* mark them */
    g->firstold1 = NULL;  /* no more OLD1 objects (for now) */
  }
  markold(g, g->finobj, g->finobjrold);
  markold(g, g->tobefnz, NULL);
  atomic(L);

  /* sweep nursery and get a pointer to its last live element */
  g->gcstate = GCSswpallgc;
  psurvival = sweepgen(L, g, &g->allgc, g->survival, &g->firstold1);
  /* sweep 'survival' */
  sweepgen(L, g, psurvival, g->old1, &g->firstold1);
  g->reallyold = g->old1;
  g->old1 = *psurvival;  /* 'survival' survivals are old now */
  g->survival = g->allgc;  /* all news are survivals */

  /* repeat for 'finobj' lists */
  dummy = NULL;  /* no 'firstold1' optimization for 'finobj' lists */
  psurvival = sweepgen(L, g, &g->finobj, g->finobjsur, &dummy);
  /* sweep 'survival' */
  sweepgen(L, g, psurvival, g->finobjold1, &dummy);
  g->finobjrold = g->finobjold1;
  g->finobjold1 = *psurvival;  /* 'survival' survivals are old now */
  g->finobjsur = g->finobj;  /* all news are survivals */

  sweepgen(L, g, &g->tobefnz, NULL, &dummy);
  finishgencycle(L, g);
}
```