

Lua源码粗浅解析(5.4.7)

A、常见数据结构

1、TString

- TString

```

/*
** Header for a string value.
*/
typedef struct TString {
  CommonHeader;
  lu_byte extra; /* reserved words for short strings: "has hash" for longs */ //关键字, 0为非关键字, 关键字从不参与GC
  ls_byte shrlen; /* length for short strings, negative for long strings */ //字符串长度, 如果是长字符串(大于40, 长字符串不会放到string hash table中) 该字段值为LSTRREG (-1)
  unsigned int hash; //hash值
  union {
    size_t lnglen; /* length for long strings */ //长字符串的长度
    struct TString *hnext; /* linked list for hash table */ //短字符串
  } u;
  char *contents; /* pointer to content in long strings */ //指向具体字符串内容的指针
  lua_Alloc falloc; /* deallocation function for external strings */ //
  void *ud; /* user data for external strings */ //外部参数
} TString;

```

- luaS_newlstr

```

/*
** new string (with explicit length)
*/
TString *luaS_newlstr(lua_State *L, const char *str, size_t l) {
  if (l <= LUAI_MAXSHORTLEN) /* short string? */ //小于40属于short string会放在global_State的stringtable中
    return internshrstr(L, str, l);
  else {
    //大于40的每次都会创建新对象
    TString *ts;
    if (l_unlikely(l * sizeof(char) >= (MAX_SIZE - sizeof(TString))))
      luaM_toobig(L);
    //创建TString
    ts = luaS_createlngstrobj(L, l);
    //赋值
    memcpy(getlngstr(ts), str, l * sizeof(char));
    return ts;
  }
}

```

PS:过长的字符串(长度大于40)每次都会创建一个新对象,大概是为了性能考虑,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(计算hash值时会遍历整个字符串)。stringtable缩小扩容都是2倍。

- internshrstr

```

/*
** Checks whether short string exists and reuses it or creates a new one.
*/
static TString *internshrstr (lua_State *L, const char *str, size_t l) {
    TString *ts;
    global_State *g = G(L);
    stringtable *tb = &g->strt;
    unsigned int h = luaS_hash(str, l, g->seed);
    TString **list = &tb->hash[lmod(h, tb->size)];
    lua_assert(str != NULL); /* otherwise 'memcmp'/'memcpy' are undefined */
    for (ts = *list; ts != NULL; ts = ts->u.hnext) {
        if (l == cast_uint(ts->shrlen) &&
            (memcmp(str, getshrstr(ts), l * sizeof(char)) == 0)) {
            /* found! */
            if (isdead(g, ts)) /* dead (but not collected yet)? */
                changewhite(ts); /* resurrect it */
            return ts;
        }
    }
    /* else must create a new string */
    if (tb->nuse >= tb->size) { /* need to grow string table? */
        growstrtab(L, tb);
        list = &tb->hash[lmod(h, tb->size)]; /* rehash with new size */
    }
    ts = createstrobj(L, sizestrshr(l), LUA_VSHRSTR, h);
    ts->shrlen = cast(ls_byte, l);
    getshrstr(ts)[l] = '\0'; /* ending 0 */
    memcpy(getshrstr(ts), str, l * sizeof(char));
    ts->u.hnext = *list;
    *list = ts;
    tb->nuse++;
    return ts;
}

```

- stringtable

```

typedef struct stringtable {
    TString **hash; /* array of buckets (linked lists of strings) */ //二维数组
    int nuse; /* number of elements */ //元素个数
    int size; /* number of buckets */ //桶的大小
} stringtable;

```

2、Table

- Table

```

typedef struct Table {
    CommonHeader;
    lu_byte flags; /* 1<<p means tagmethod(p) is not present */ //原表标记
    lu_byte lsizenode; /* log2 of size of 'node' array */ //hash表大小
    unsigned int alimit; /* "limit" of 'array' array */ //数组大小
    Value *array; /* array part */ //数组
    Node *node; /* element list */ //元素列表
    struct Table *metatable; /* metatable */ //原表
    GCObject *gclist; /* gc-related, points to position in gc list */ //gc相关，指向所处gc列表中的位置
} Table;

```

- table相关操作指令：

- OP_NEWTABLE :

```

vmcase(OP_NEWTABLE) {
    StkId ra = RA(i);
    unsigned b = cast_uint(GETARG_vB(i)); /* log2(hash size) + 1 */
    unsigned c = cast_uint(GETARG_vC(i)); /* array size */
    Table *t;
    if (b > 0)
        b = 1u << (b - 1); /* hash size is 2^(b - 1) */
    if (TESTARG_k(i)) { /* non-zero extra argument? */
        lua_assert(GETARG_Ax(*pc) != 0);
        /* add it to array size */
        c += cast_uint(GETARG_Ax(*pc)) * (MAXARG_vC + 1);
    }
    pc++; /* skip extra argument */
    L->top.p = ra + 1; /* correct top in case of emergency GC */
    t = luaH_new(L); /* memory allocation */
    sethvalue2s(L, ra, t);
    if (b != 0 || c != 0)
        luaH_resize(L, t, c, b); /* idem */ //设置大小
    checkGC(L, ra + 1);
    vmbreak;
}

```

- OP_SETLIST :

```

//设置table数组部分元素
vmcase(OP_SETLIST) {
    StkId ra = RA(i);
    unsigned n = cast_uint(GETARG_vB(i));
    unsigned int last = cast_uint(GETARG_vC(i));
    Table *h = hvalue(s2v(ra));
    if (n == 0)
        n = cast_uint(L->top.p - ra) - 1; /* get up to the top */
    else
        L->top.p = ci->top.p; /* correct top in case of emergency GC */
    last += n;
    if (TESTARG_k(i)) {
        last += cast_uint(GETARG_Ax(*pc)) * (MAXARG_vC + 1);
        pc++;
    }
    /* when 'n' is known, table should have proper size */
    if (last > luaH_reallocsize(h)) { /* needs more space? */
        /* fixed-size sets should have space preallocated */
        lua_assert(GETARG_vB(i) == 0);
        luaH_resizearray(L, h, last); /* preallocate it at once */
    }
    for (; n > 0; n--) {
        TValue *val = s2v(ra + n);
        obj2arr(h, last - 1, val);
        last--;
        luaC_barrierback(L, obj2gco(h), val);
    }
    vmbreak;
}

```

- OP_SETI :

```
//设置key为int类型的元素，可能放在数组部分也可能放到hash部分
//具体逻辑在luaV_fastseti
vmcase(OP_SETI) {
    StkId ra = RA(i);
    int hres;
    int b = GETARG_B(i);
    TValue *rc = RKC(i);
    luaV_fastseti(s2v(ra), b, rc, hres);
    if (hres == HOK)
        luaV_finishfastset(L, s2v(ra), rc);
    else {
        TValue key;
        setivalue(&key, b);
        Protect(luaV_finishset(L, s2v(ra), &key, rc, hres));
    }
    vmbreak;
}
```

- OP_SETFIELD :

```
//设置key为short string类型的元素
vmcase(OP_SETFIELD) {
    StkId ra = RA(i);
    int hres;
    TValue *rb = KB(i);
    TValue *rc = RKC(i);
    TString *key = tsvalue(rb); /* key must be a short string */
    char* content = key->contents;
    luaV_fastset(s2v(ra), key, rc, hres, luaH_psetshortstr);
    if (hres == HOK)
        luaV_finishfastset(L, s2v(ra), rc);
    else
        Protect(luaV_finishset(L, s2v(ra), rb, rc, hres));
    vmbreak;
}
```

- OP_SETTABLE :

```
//设置单个元素
vmcase(OP_SETTABLE) {
    StkId ra = RA(i);
    int hres;
    TValue *rb = vRB(i); /* key (table is in 'ra') */
    TValue *rc = RKC(i); /* value */
    //判断key是否为int类型
    if (ttisinteger(rb)) { /* fast track for integers? */
        luaV_fastseti(s2v(ra), ivalue(rb), rc, hres);
    }
    else {
        //非int类型，hres可能在数组部分也可能在hash部分
        luaV_fastset(s2v(ra), rb, rc, hres, luaH_pset);
    }
    if (hres == HOK)
        luaV_finishfastset(L, s2v(ra), rc);
    else
        Protect(luaV_finishset(L, s2v(ra), rb, rc, hres));
    vmbreak;
}
```

- `local tbl = {1, ["test1"] = 4, 2, ["test2"] = 5, 3, ["test3"] = 6}`; 该代码的指令调用：首先调用OP_NEWTABLE，并且b(hash size)、c(array size)不为0，再调用OP_SETFIELD，一个个设置hash部分数据以及OP_SETLIST一次性设置array部分数据。
`local tbl = {}; tbl[1] = 1; tbl["test1"] = 4; tbl[2] = 2`; 该代码的指令调用：首先

调用OP_NEWTABLE，并且b(hash size)、c(array size)为0，再按顺序调用OP_SETI或者OP_SETFIELD，赋值过程中会调整table的数组和hash大小。

OP_SETTABLE在5.4.7貌似没用了。被拆分为OP_SETI和OP_SETFIELD。

- luaV_fastseti (OP_SETI查找位置)：

```
#define luaV_fastseti(t,k,val,hres) \
if (!ttistable(t)) hres = HNOTATABLE; \
else { luaH_fastseti(hvalue(t), k, val, hres); }

#define luaH_fastseti(t,k,val,hres) \
{ Table *h = t; lua_Unsigned u = l_castS2U(k) - 1u; \
  //未超过数组大小，此时该下标为空则直接赋值 \
  //否则则替换 \
  if ((u < h->alimit)) { \
    lu_byte *tag = getArrTag(h, u); \
    if (tagisempty(*tag)) hres = ~cast_int(u); \
    else { fval2arr(h, u, tag, val); hres = HOK; }} \
  //超过数组大小，则会放到hash表，对hash表长度取模 \
  else { hres = luaH_psetint(h, k, val); }}
```

- luaV_finishset，如果luaV_fastseti和luaV_fastset没有找到位置，则调用luaV_finishset：

```
void luaV_finishset (lua_State *L, const TValue *t, TValue *key, TValue
*val, int hres) {
  int loop; /* counter to avoid infinite loops */
  for (loop = 0; loop < MAXTAGLOOP; loop++) {
    const TValue *tm; /* '__newindex' metamethod */
    if (hres != HNOTATABLE) { /* is 't' a table? */
      Table *h = hvalue(t); /* save 't' table */
      tm = fasttm(L, h->metatable, TM_NEWINDEX); /* get metamethod */
      //没有newindex的元方法
      if (tm == NULL) { /* no metamethod? */
        luaH_finishset(L, h, key, val, hres); /* set new value */
        invalidateTMcache(h);
        luaC_barrierback(L, obj2gco(h), val);
        return;
      }
      /* else will try the metamethod */
    }
    else { /* not a table; check metamethod */
      tm = luaT_gettmbyobj(L, t, TM_NEWINDEX);
      if (l_unlikely(notm(tm)))
        luaG_typeerror(L, t, "index");
    }
    /* try the metamethod */
    //元方法是个函数则调用对应的函数
    if (ttisfunction(tm)) {
      luaT_callTM(L, tm, t, key, val);
      return;
    }
  }
}
```

```

    //元方法不是函数则把元方法当作一个table继续
    t = tm; /* else repeat assignment over 'tm' */
    luaV_fastset(t, key, val, hres, luaH_pset);
    if (hres == HOK)
        return; /* done */
    /* else 'return luaV_finishset(L, t, key, val, slot)' (loop) */
}
luaG_runerror(L, "'__newindex' chain too long; possible loop");
}

void luaH_finishset (lua_State *L, Table *t, const TValue *key, TValue
*value, int hres) {
    lua_assert(hres != HOK);
    if (hres == HNOTFOUND) {
        luaH_newkey(L, t, key, value);
    }
    else if (hres > 0) { /* regular Node? */
        setobj2t(L, gval(gnode(t, hres - HFIRSTNODE)), value);
    }
    else { /* array entry */
        hres = ~hres; /* real index */
        obj2arr(t, hres, value);
    }
}
}

```

◦ luaH_newkey :

```

//如果key是int类型，且在数组大小限制内，则在这之前赋值已经完成，到这一步则是因
为
//key为非int类型或者key为int类型但超过数组限制从而被分配到hash部分。
static void luaH_newkey (lua_State *L, Table *t, const TValue
*key, TValue *value)
{
    Node *mp;
    TValue aux;
    if (l_unlikely(ttisnil(key)))
        luaG_runerror(L, "table index is nil");
    else if (ttisfloat(key))
    {
        //key为浮点数，首先尝试转成int类型
        lua_Number f = fltvalue(key);
        lua_Integer k;
        if (luaV_flttointeger(f, &k, F2Ieq)) { /* does key fit in an
integer? */
            setivalue(&aux, k);
            key = &aux; /* insert it as an integer */
        }
        else if (l_unlikely(luai_numisnan(f)))
            luaG_runerror(L, "table index is NaN");
    }
    if (ttisnil(value))
        return; /* do not insert nil values */
}

```

```

//获取key对应的node
mp = mainpositionTV(t, key);
if (!isempty(gval(mp)) || isdummy(t)) { /* main position is taken?
*/
    Node *othern;
    Node *f = getfreepos(t); /* get a free place */
    if (f == NULL) { /* cannot find a free place? */
        rehash(L, t, key); /* grow table */
        /* whatever called 'newkey' takes care of TM cache */
        luaH_set(L, t, key, value); /* insert key into grown table */
        return;
    }
    lua_assert(!isdummy(t));
    //获取当前位置的node的key本该所在的位置
    othern = mainpositionfromnode(t, mp);
    //如果不相等那就是其他key之前就已经发生过碰撞被分配到这里
    if (othern != mp) { /* is colliding node out of its main position?
*/
        /* yes; move colliding node into free position */
        //将原来的挪到空闲位置，并将传入的key放入所对应的node里面
        while (othern + gnext(othern) != mp) /* find previous */
            othern += gnext(othern);
        gnext(othern) = cast_int(f - othern); /* rechain to point to 'f'
*/
        *f = *mp; /* copy colliding node into free pos. (mp->next also
goes) */
        if (gnext(mp) != 0) {
            gnext(f) += cast_int(mp - f); /* correct 'next' */
            gnext(mp) = 0; /* now 'mp' is free */
        }
        setempty(gval(mp));
    }
    else { /* colliding node is in its own main position */
        /* new node will go into free position */
        //赋值到找到的空闲位置并且将空闲位置连接起来
        if (gnext(mp) != 0)
            gnext(f) = cast_int((mp + gnext(mp)) - f); /* chain new
position */
        else lua_assert(gnext(f) == 0);
        gnext(mp) = cast_int(f - mp);
        mp = f;
    }
}
//没有发生冲突且table的lsizenode不为0，则找到对应位置直接赋值
setnodekey(L, mp, key);
luaC_barrierback(L, obj2gco(t), key);
lua_assert(isempty(gval(mp)));
setobj2t(L, gval(mp), value);
}

```

- rehash :

```

static void rehash (lua_State *L, Table *t, const TValue *ek)
{
    unsigned int asize; /* optimal size for array part */
    unsigned int na; /* number of keys in the array part */
    //nums[i]记录的值是key的大小在2^(i - 1)到2^i的数量
    unsigned int nums[MAXABITS + 1];
    int i;
    unsigned totaluse;
    for (i = 0; i <= MAXABITS; i++) nums[i] = 0; /* reset counts */
    setlimittosize(t);
    //计算数组部分数据分布
    na = numusearray(t, nums); /* count keys in array part */
    totaluse = na; /* all those keys are integer keys */
    //计算hash部分数据分布
    totaluse += numusehash(t, nums, &na); /* count keys in hash part */
    /* count extra key */
    if (ttisinteger(ek))
        na += countint(ivalue(ek), nums);
    totaluse++;
    /* compute new size for array part */
    //计算新的数组部分大小
    asize = computesizes(nums, &na);
    /* resize the table to new computed sizes */
    //重新分配数组以及hash表的大小
    luaH_resize(L, t, asize, totaluse - na);
}

//计算数组部分大小，规则是找到最大且数量超过一半的
static unsigned computesizes (unsigned nums[], unsigned *pna)
{
    int i;
    unsigned int twotoi; /* 2^i (candidate for optimal size) */
    unsigned int a = 0; /* number of elements smaller than 2^i */
    unsigned int na = 0; /* number of elements to go to array part */
    unsigned int optimal = 0; /* optimal size for array part */
    /* loop while keys can fill more than half of total size */
    for (i = 0, twotoi = 1;
         twotoi > 0 && *pna > twotoi / 2;
         i++, twotoi *= 2) {
        a += nums[i];
        if (a > twotoi/2) { /* more than half elements present? */
            optimal = twotoi; /* optimal size (till now) */
            na = a; /* all elements up to 'optimal' will go to array part */
        }
    }
    lua_assert((optimal == 0 || optimal / 2 < na) && na <= optimal);
    *pna = na;
    return optimal;
}

```

- 因此定义一个table时最好是定义即初始化，否则一个个添加元素会导致多次rehash。或者用table.create(数组大小，hash大小)api来创建table。table.create如下所示：


```

static int tcreate (lua_State *L)
{
    lua_Unsigned sizeseq = (lua_Unsigned)luaL_checkinteger(L, 1);
    lua_Unsigned sizereq = (lua_Unsigned)luaL_optinteger(L, 2, 0);
    luaL_argcheck(L, sizeseq <= UINT_MAX, 1, "out of range");
    luaL_argcheck(L, sizereq <= UINT_MAX, 2, "out of range");
    lua_createtable(L, (unsigned)sizeseq, (unsigned)sizereq);
    return 1;
}

LUA_API void lua_createtable (lua_State *L, unsigned narray, unsigned nrec)
{
    Table *t;
    lua_lock(L);
    t = luaH_new(L);
    sethvalue2s(L, L->top.p, t);
    api_incr_top(L);
    if (narray > 0 || nrec > 0)
        luaH_resize(L, t, narray, nrec);
    luaC_checkGC(L);
    lua_unlock(L);
}

```

- table其他相关API：
 - ipairs和pairs：
 - ipairs源码如下所示：

```

static int luaB_ipairs (lua_State *L)
{
    luaL_checkany(L, 1);
    lua_pushcfunction(L, ipairsaux); /* iteration function */
    lua_pushvalue(L, 1); /* state */
    lua_pushinteger(L, 0); /* initial value */
    return 3;
}

static int ipairsaux (lua_State *L)
{
    lua_Integer i = luaL_checkinteger(L, 2);
    //累加index
    i = luaL_intop(+, i, 1);
    lua_pushinteger(L, i);
    return (lua_geti(L, 1, i) == LUA_TNIL) ? 1 : 2;
}

LUA_API int lua_geti (lua_State *L, int idx, lua_Integer n)
{
    TValue *t;
    lu_byte tag;
    lua_lock(L);

```

```

    t = index2value(L, idx);
    //查找key为int类型的元素，这一步需要说明的是，该key不一定会在数组中。
    hash表也会找，只要key满足条件。
    luaV_fastgeti(t, n, s2v(L->top.p), tag);
    //当原表中该key为空时，会调用luaV_finishget，该函数会调用元表的index
    元方法，
    //如果元表的index元方法是个表则重复该操作，如果是个函数则调用该函数
    if (tagisempty(tag)) {
        TValue key;
        setivalue(&key, n);
        tag = luaV_finishget(L, t, &key, L->top.p, tag);
    }
    api_incr_top(L);
    lua_unlock(L);
    return novariant(tag);
}

```

ipairs遍历table时是从下标1开始查找，并且原表没有会去元表中查找。当value为nil时退出。

- pairs源码如下所示：

```

int luaH_next (lua_State *L, Table *t, StkId key)
{
    unsigned int asize = luaH_reallocsize(t);
    unsigned int i = findindex(L, t, s2v(key), asize); /* find
original key */
    //遍历数组部分
    for (; i < asize; i++) { /* try first array part */
        lu_byte tag = *getArrTag(t, i);
        if (!tagisempty(tag)) { /* a non-empty entry? */
            setivalue(s2v(key), cast_int(i) + 1);
            farr2val(t, i, tag, s2v(key + 1));
            return 1;
        }
    }
    for (i -= asize; i < sizenode(t); i++) { /* hash part */
        // #define gnode(t,i) (&(t)->node[i])
        //遍历table的hash部分,gnode是直接将i当成下标，而不是key
        if (!isempty(gval(gnode(t, i)))) { /* a non-empty entry? */
            Node *n = gnode(t, i);
            getnodekey(L, s2v(key), n);
            setobj2s(L, key + 1, gval(n));
            return 1;
        }
    }
    return 0; /* no more elements */
}

```

pairs会遍历数组和hash。

- 几个特殊的table：

- registry表，key为LUA_REGISTRYINDEX，返回的是global_State的l_registry字段。
- global表，key为LUA_RIDX_GLOBALS (2)，保存在l_registry表，字面意思全局变量会放在这个表里。
- loaded表，key为LUA_LOADED_TABLE (loaded)，保存在package表(保存在l_registry表，key为package)，调用require加载解析过的lua文件或代码会保存在这个表里。调用load加载代码不会保存在该表中。require和load都会调用lua_load来解析代码。

```
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 */
        //load什么样的代码upvalue size会小于1?
        //require("return 1")这样的代码，upvalue size都是1
        if (f->nupvalues >= 1) { /* does it have an upvalue? */
            /* get global table from registry */
            TValue gt;
            getGlobalTable(L, &gt;);
            /* 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;
}
```

从上面代码可以看出，会将解析生成的LClosure的第一个upvalue指向global表，而当定义非local变量的时候，会调用OP_SETTABUP，给第一个upvalue赋值，此时第一个upvalue指向的是global表。这也就是说全局变量会放在global表。

3、CClosure和LClosure

- 1、结构体定义

```
#define ClosureHeader \
CommonHeader; lu_byte nupvalues; GCObject *gclist

typedef struct CClosure {
    ClosureHeader;          //nupvalues upvalue的数量、gclist gc相关
    lua_CFunction f;        //函数指针
    TValue upvalue[1];      /* list of upvalues */
} CClosure;
```

```
typedef struct LClosure {
    ClosureHeader;    //nupvalues upvalue的数量、gclist gc相关
    struct Proto *p;
    UpVal *upvals[1]; /* list of upvalues */
} LClosure;
```

- 2、UpVal结构体的定义如下：

```
typedef struct UpVal {
    CommonHeader;
    union {
        TValue *p; /* points to stack or to its own value */
        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;
```

LClosure的upvalue分为open和close两种状态，当v.p不是指向u.value的时候就是open，当v.p指向u.value就是close的。下面通过一个实例来解释。

- 3、示例：

```
function test()
    local a = 1
    return function()
        a = a + 1
    end
end

local testfunc = test()
testfunc()
testfunc()
```

当调用函数test生成一个LClosure时（pushclosure），此时a的作用域还未结束，此时这个upvalue就是open的，v.p指向的是a在lua栈上的地址，当函数test执行结束时，a的作用域结束，lua栈回收，此时upvalue会变成close（luaF_closeupval），会将原来的值赋值给u.value，并且v.p也会

指向u.value。当upvalue是open的时候，会记录在lua_State的openupval字段。当变成close的时候，会从openupvalue双向列表里删除。

- 4、close upvalue

```
void luaF_closeupval (lua_State *L, StkId level) {
    UpVal *uv;
    StkId upl;
    //open状态下，upvalue记录在openipval列表上
    while ((uv = L->openupval) != NULL && (upl = uplevel(uv)) >= level) {
        TValue *slot = &uv->u.value;
        lua_assert(uplevel(uv) < L->top.p);
        //从openupval列表中删除
        luaF_unlinkupval(uv);
        //v.p指向u.value
        setobj(L, slot, uv->v.p);
        uv->v.p = slot;
        if (!iswhite(uv)) {
            nw2black(uv);
            luaC_barrier(L, uv, slot);
        }
    }
}
```

B、协程

1、创建协程luaB_cocreate

```
static int luaB_cocreate (lua_State *L) {
    lua_State *NL;
    //此时栈顶必须是一个函数。如果执行local co = coroutine.create(counter),
    //首先会执行OP_CLOSURE，创建一个LClosure对象放在栈顶
    luaL_checktype(L, 1, LUA_TFUNCTION);
    //创建一个新的lua_State对象放在栈顶
    NL = lua_newthread(L);
    //将LClosure对象再次压入栈顶
    lua_pushvalue(L, 1); /* move function to top */
    //将栈顶的1个数据（也就是LClosure对象）复制到NL的栈顶，并且L的栈顶收缩
    lua_xmove(L, NL, 1); /* move function from L to NL */
    return 1;
}
```

2、唤起协程luaB_coresume

```
static int luaB_coresume (lua_State *L) {
    //此时需要切换的协程处于栈顶
```

```

lua_State *co = getco(L);
int r;
//lua_gettop(L) - 1的作用是获取参数个数
r = auxresume(L, co, lua_gettop(L) - 1);
if (l_unlikely(r < 0)) {
    lua_pushboolean(L, 0);
    lua_insert(L, -2);
    return 2; /* return false + error message */
}
else {
    lua_pushboolean(L, 1);
    lua_insert(L, -(r + 1));
    return r + 1; /* return true + 'resume' returns */
}
}

static int auxresume (lua_State *L, lua_State *co, int narg) {
    int status, nres;
    if (l_unlikely(!lua_checkstack(co, narg))) {
        lua_pushliteral(L, "too many arguments to resume");
        return -1; /* error flag */
    }
    //将参数复制到协程co
    lua_xmove(L, co, narg);
    //就不再进一步展开了，在这一步中会调用setjmp，将当前的执行环境（包括寄存器、堆栈指针等）
    //保存到协程co的errorJmp字段上，然后执行协程co的指令
    status = lua_resume(co, L, narg, &nres);
    if (l_likely(status == LUA_OK || status == LUA_YIELD)) {
        if (l_unlikely(!lua_checkstack(L, nres + 1))) {
            lua_pop(co, nres); /* remove results anyway */
            lua_pushliteral(L, "too many results to resume");
            return -1; /* error flag */
        }
        //将返回值复制到L的栈上
        lua_xmove(co, L, nres); /* move yielded values */
        return nres;
    }
    else {
        lua_xmove(co, L, 1); /* move error message */
        return -1; /* error flag */
    }
}

```

3、挂起协程luaB_yield

```

static int luaB_yield (lua_State *L) {
    return lua_yield(L, lua_gettop(L));
}

LUA_API int lua_yieldk (lua_State *L, int nresults, lua_KContext ctx,

```

```

lua_KFunction k) {
    CallInfo *ci;
    luai_userstateyield(L, nresults);
    lua_lock(L);
    ci = L->ci;
    api_checkpop(L, nresults);
    if (l_unlikely(!yieldable(L))) {
        if (L != G(L)->mainthread)
            luaG_runerror(L, "attempt to yield across a C-call boundary");
        else
            luaG_runerror(L, "attempt to yield from outside a coroutine");
    }
    //此时的L是之前调用resume唤起的协程
    //将状态设为挂起，记录返回值个数
    L->status = LUA_YIELD;
    ci->u2.nyield = nresults; /* save number of results */
    if (isLua(ci)) { /* inside a hook? */
        lua_assert(!isLuacode(ci));
        api_check(L, nresults == 0, "hooks cannot yield values");
        api_check(L, k == NULL, "hooks cannot continue after yielding");
    }
    else {
        if ((ci->u.c.k = k) != NULL) /* is there a continuation? */
            ci->u.c.ctx = ctx; /* save context */
        //调用longjmp，跳转到之前resume保存的地方
        luaD_throw(L, LUA_YIELD);
    }
    lua_assert(ci->callstatus & CIST_HOOKED); /* must be inside a hook */
    lua_unlock(L);
    return 0; /* return to 'luaD_hook' */
}

```

C、GC

```

#define luaC_condGC(L, pre, pos) \
{ if (G(L)->GCdebt <= 0) { pre: luaC_step(L): pos:}; \
  condchangemem(L, pre, pos): }

```

只有当GCdebt小于等于0时，才会触发GC，这个GCdebt的单位是对象的个数而不是实际的内存大小。当申请一个新的需要GC的对象时，该变量做-1操作。totalbytes字段才是保存当前lua虚拟机申请过的内存字节数。需要注意的是并不是每次new一个GC对象都会去check gc。

1、增量式GC

```

static void incstep (lua_State *L, global_State *g) {
    //STEPSIZE默认值为250
    l_obj stepsize = applygcparam(g, STEPSIZE, 100);
    //work2do 默认值为500 也就是说最多一次处理500个元素
    l_obj work2do = applygcparam(g, STEPMUL, stepsize);
    int fast = 0;
    if (work2do == 0) { /* special case: do a full collection */

```

```

    work2do = MAX_LOBJ; /* do unlimited work */
    fast = 1;
}
do { /* repeat until pause or enough work */
    l_obj work = singlestep(L, fast); /* perform one single step */
    if (g->gckind == KGC_GENMINOR) /* returned to minor collections? */
        return; /* nothing else to be done here */
    work2do -= work;
} while (work2do > 0 && g->gcstate != GCSpause);
if (g->gcstate == GCSpause)
    setpause(g); /* pause until next cycle */
else
    luaE_setdebt(g, stepsize);
}

```

GCSpause

```

//清理灰色链表并且标记根节点 (单步)
static void restartcollection (global_State *g) {
    cleargraylists(g);
    g->marked = NFIXED;
    markobject(g, g->mainthread);
    markvalue(g, &g->l_registry);
    markmt(g);
    markbeingfnz(g); /* mark any finalizing object left from previous cycle */
}

```

GCSpropagate

```

static void propagatemark (global_State *g) {
    GCObject *o = g->gray;
    //设置为黑色，这里要跟luaC_barrierback对应看，假如在GCSpropagate阶段，global表已经被扫描过了，global表被标记为黑色。
    //此时又定义一个全局变量，那么在luaC_barrierback里会将global表又塞回gray列表。
    nw2black(o);
    //从灰色链表里删除
    g->gray = *getgcclist(o); /* remove from 'gray' list */
    switch (o->tt) {
        case LUA_VTABLE: traversetable(g, gco2t(o)); break;
        case LUA_VUSERDATA: traverseudata(g, gco2u(o)); break;
        case LUA_VLCL: traverseLclosures(g, gco2lcl(o)); break;
        case LUA_VCCL: traverseCclosures(g, gco2ccl(o)); break;
        case LUA_VPROTO: traverseproto(g, gco2p(o)); break;
        case LUA_VTHREAD: traversethread(g, gco2th(o)); break;
        default: lua_assert(0);
    }
}

```


luaC_barrierback和**luaC_barrier**，关于这两个网上解释一大堆，但没有我特别信服的文章。在这里写下此时的思考，之后如果发现理解不对再改。

luaC_barrierback(L,父节点,子节点)如果父节点是黑色并且子节点是白色，则将父节点改为灰色，放入**grayagain**列表。**luaC_barrier**(L,父节点,子节点)，当GC步骤小于等于**GCSenteratomic**时，如果父节点是黑色并且子节点是白色，直接将子节点往前推。否则，将子节点设置为新的白色，等待下次GC。按我的理解，这两种其实可以互相替换，或者只保留一个。这两种的主要分别是作用的对象类型上，**table**的赋值用**luaC_barrierback**，这是因为**table**赋值是经常性的操作，用**luaC_barrierback**放到**grayagain**列表，在之后一次性做标记。而**upvalue**的改变用**luaC_barrier**是因为**upvalue**的变动不是经常性的，这样能减少遍历的对象，提升性能。

GCSenteratomic

这个阶段也是单步的，需要在这一步明确所有对象的颜色（此时如果main thread、register表等是白色则加入gray表），并且在最后将global_State的currentwhite设置为新白色。这一步是GC能跟上新增元素的兜底机制，所以这一步有可能会比较重。

GCSswpallgc、GCSswpfinobj、GCSswptobefnz

```
case GCSswpallgc: { /* sweep "regular" objects */
    sweepstep(L, g, GCSswpfinobj, &g->finobj, fast);
    work = GCSWEEPMAX;
    break;
}
case GCSswpfinobj: { /* sweep objects with finalizers */
    sweepstep(L, g, GCSswptobefnz, &g->tobefnz, fast);
    work = GCSWEEPMAX;
    break;
}
case GCSswptobefnz: { /* sweep objects to be finalized */
    sweepstep(L, g, GCSswpend, NULL, fast);
    work = GCSWEEPMAX;
    break;
}
```

这三个阶段做的是相同的工作，只是对应的列表不一样而已（GCSswpallgc阶段遍历的是global_State的allgc列表、GCSswpfinobj阶段遍历的是global_State的finobj列表、GCSswptobefnz遍历的是global_State的tobefnz列表），各自检查对应的列表，如果需要回收就回收，不需要则改变对象marked字段，设置为新的白色。每次最多检查GCSWEEPMAX（20）个对象。如果对象的类型是table或者userdata，当设置元表并且有“__gc”元方法时，会将该对象从allgc列表中移除，并放入finobj列表。在GCSenteratomic阶段，会调用separatetobefnz函数，这个函数会将finobj列表里白色的移动到tobefnz列表。所以在GCSswpfinobj阶段处理finobj列表时，finobj列表里的元素全是不需要回收的，所以这阶段的作用是将finobj列表里的元素的颜色改成当前白色。

GCSswpend

在非紧急状态下，如果常驻的string table太空闲，则会回收global_State的strt字段。如果string table的size太大或者申请内存时第一次失败后或者lua脚本调用collectgarbage("collect")，会设为紧急状态，并做一次完整步骤的GC，并再次申请内存。

```

static void checkSizes (lua_State *L, global_State *g) {
    if (!g->gcemergency) {
        if (g->strt.nuse < g->strt.size / 4) /* string table too big? */
            luaS_resize(L, g->strt.size / 2);
    }
}

//首次内存申请失败，则会执行这个函数
static void *tryagain (lua_State *L, void *block, size_t osize, size_t nsize) {
    global_State *g = G(L);
    if (cantryagain(g)) {
        luaC_fullgc(L, 1); /* try to free some memory... */
        return callfrealloc(g, block, osize, nsize); /* try again */
    }
    else return NULL; /* cannot run an emergency collection */
}

static void growstrtab (lua_State *L, stringtable *tb) {
    //global的strt太大
    if (l_unlikely(tb->nuse == INT_MAX)) { /* too many strings? */
        luaC_fullgc(L, 1); /* try to free some... */
        if (tb->nuse == INT_MAX) /* still too many? */
            luaM_error(L); /* cannot even create a message... */
    }
    if (tb->size <= MAXSTRTB / 2) /* can grow string table? */
        luaS_resize(L, tb->size * 2);
}

void luaC_fullgc (lua_State *L, int isemergency) {
    global_State *g = G(L);
    lua_assert(!g->gcemergency);
    //设为紧急状态
    g->gcemergency = cast_byte(isemergency); /* set flag */
    //阻塞执行一次完整步骤的GC，遍历所有的对象
    switch (g->gckind) {
        case KGC_GENMINOR: fullgen(L, g); break;
        case KGC_INC: fullinc(L, g); break;
        case KGC_GENMAJOR:
            g->gckind = KGC_INC;
            fullinc(L, g);
            g->gckind = KGC_GENMAJOR;
            break;
    }
    g->gcemergency = 0;
}

//执行完整的GC，遍历所有对象做标记或回收，最后将gcstate设为GCSpause阶段。
static void fullinc (lua_State *L, global_State *g) {
    if (keepinvariant(g)) /* black objects? */
        entersweep(L); /* sweep everything to turn them back to white */
    /* finish any pending sweep phase to start a new cycle */
    luaC_runtilstate(L, GCSpause, 1);
    luaC_runtilstate(L, GCScallfin, 1); /* run up to finalizers */
}

```

```

/* 'marked' must be correct after a full GC cycle */
lua_assert(g->marked == gettotalobjs(g));
luaC_runtillstate(L, GCSpause, 1); /* finish collection */
setpause(g);
}

```

GCScallfin

每次从tobefnz列表中取出一个对象，并调用对象的“_gc”方法。

2、分代式GC

- G_NEW 0 //新创建的对象
- G_SURVIVAL 1 //G_NEW对象存活一次GC变成G_SURVIVAL
- G_OLD0 2 //luaC_barrier时对象变成G_OLD0
- G_OLD1 3 //G_SURVIVAL和G_OLD0状态的对象存活一次GC变成G_OLD1
- G_OLD 4 //G_OLD1或者G_TOUCHED2对象存活过一次GC变成G_OLD
- G_TOUCHED1 5 //luaC_barrierback时对象变成G_TOUCHED1
- G_TOUCHED2 6 //G_TOUCHED1对象存活一次GC变成G_TOUCHED2

```
tbl = {1,2,3}
```

上述代码minor gc步骤介绍：

1、新建tbl对象时，tbl的mark字段为(白色|G_NEW)，因为tbl是全局变量，那么会放在global表，会调用luaC_barrierback，如果global表此时是黑色的，那global表会变成(灰色|G_TOUCHED1)并且放入grayagain列表。

2、第一次执行minor gc：在youngcollection函数中调用atomic(L)时，当清空grayagain列表时，(此时假设grayagain只有一个global表一个元素)，首先将global表改成(黑色|G_TOUCHED1)，由于此时是G_TOUCHED1，那么又会改成(灰色|G_TOUCHED1)重新加到grayagain列表中。而属于global表的tbl对象，由于也是一个table，在处理global表时会把tbl加入gray列表，并且最终会被设置为(黑色|G_NEW)并从gray列表里删除。

调用sweepgen时，会将tbl对象设置为(白色|G_SURVIVAL)。G_NEW到G_SURVIVAL会重新设置为白色。

对于global表还在grayagain列表，执行finishgencycle时，会将global表设置为(黑色|G_TOUCHED2)并且保留在grayagain列表。

3、第二次执行minor gc：在youngcollection函数中调用atomic(L)时，当清空grayagain列表时，(此时假设grayagain只有一个global表一个元素)，由于此时是G_TOUCHED2，会将global表改成(黑色|G_OLD)。而属于global表的tbl对象改成(黑色|G_SURVIVAL)。调用sweepgen时，会将tbl对象设置为(黑色|G_OLD1)。

4、第三次执行minor gc：调用sweepgen时，会将tbl对象设置为(黑色|G_OLD)。

```

local function create_closure()
    local upvalue = {1,2,3}
    return function(new_value)
        upvalue = new_value
    end
end
local closure = create_closure()

```

```
local new_table = {4,5,6}
closure(new_table)
```

操作码为OP_SETUPVAL 上述代码minor gc步骤介绍：

- 1、首先会调用luaC_barrier，由于upvalue数据类型为table，upvalue会变为（灰色|G_OLD0），并且加入gray列表。
- 2、第一次执行minor gc：调用atomic时，该元素会被设置为黑色|G_OLD0，调用sweepgen时，该元素会被设置为黑色|G_OLD1。
- 3、第二次执行minor gc：调用sweepgen时，该元素会被设置为黑色|G_OLD。

Minor Collection

- **minor mode**假设的是变成Old的元素基本不会死亡，或者说Old元素死亡也不重要，死亡了在minor mode下也不会去处理。所以该模式下在触发GC的时候只会去处理年轻一代的元素。这样GC需要处理的元素就会大量减少，以提升GC性能。但这也有一个隐藏的问题，因为面向C层次的有些接口新增元素并不会去检查GC，那么有可能导致某一次GC的时候需要处理大量元素，造成CPU突刺，当然大量业务逻辑在lua层的时候，这种情况应该不会出现。
- **youngcollection**：

```
static void youngcollection (lua_State *L, global_State *g) {
    l_obj addedold1 = 0;
    l_obj marked = g->marked; /* preserve 'g->marked' */
    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); /* will lose 'g->marked' */

    /* sweep nursery and get a pointer to its last live element */
    g->gcstate = GCSswpallgc;
    psurvival = sweepgen(L, g, &g->allgc, g->survival, &g->firstold1,
&addedold1);
    /* sweep 'survival' */
    sweepgen(L, g, psurvival, g->old1, &g->firstold1, &addedold1);
    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, &addedold1);
    /* sweep 'survival' */
    sweepgen(L, g, psurvival, g->finobjold1, &dummy, &addedold1);
}
```

```

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, &addedold1);

/* keep total number of added old1 objects */
g->marked = marked + addedold1;

/* decide whether to shift to major mode */
//本次新增的old1元素个数超过minor gc前元素百分之25的一半
//或者minor gc新增的old元素个数超过minor gc前的元素个数
//切换为major mode
if (checkminormajor(g, addedold1)) {
    minor2inc(L, g, KGC_GENMAJOR); /* go to major mode */
    g->marked = 0; /* avoid pause in first major cycle */
}
else
    finishgencycle(L, g); /* still in minor mode; finish it */
}

```

- checkminormajor :

```

static int checkminormajor (global_State *g, l_obj addedold1) {
    //MINORMUL默认值是25%
    l_obj step = applygcparam(g, MINORMUL, g->GCmajorminor);
    //MINORMAJOR默认值是100%
    l_obj limit = applygcparam(g, MINORMAJOR, g->GCmajorminor);
    //本次新增的old1元素个数超过minor gc前元素百分之25的一半
    //或者minor gc新增的old元素个数超过minor gc前的元素个数
    return (addedold1 >= (step >> 1) || g->marked >= limit);
}

```

- 相关参数 :
 - g->marked参数 : minor模式下, 该参数表示累计变为G_OLD1的元素个数, GC模式切换时该字段会置为0。这个精度并不准确, 分代式GC处于minor gc时的global_State的marked字段的精度并不准确是因为某个元素调用luaC_objbarrier时, marked字段会加1, 并且该元素被设置为G_OLD0, 而G_OLD0变为G_OLD1是, marked字段又会加1, 一个元素变为G_OLD1, 而marked字段被累加了两次。
 - g->GCmajorminor参数 : 执行youngcollection前的元素个数。

Major Collection

- major mode复用增量式GC的代码, 当从minor gc切换到major gc的时候, 设置一些参数并将gcstate设置为GCSswpallgc, 之后流程跟增量式GC是一样的, 只是执行完一步后不会重新设置debt, 也就是说下次新增对象也会进入gc, 而且执行完整套增量式GC后, 也就是到GCsenteratomic, 会检查是否可以切回minor gc。

- minor2inc :

```
static void minor2inc (lua_State *L, global_State *g, lu_byte kind) {
    //GCmajorminor设置为本次minor gc变为G_OLD1的元素累计个数
    g->GCmajorminor = g->marked; /* number of live objects */
    g->gckind = kind;
    g->reallyold = g->old1 = g->survival = NULL;
    g->finobjrold = g->finobjold1 = g->finobjsur = NULL;
    //将gcstate设置为GCSswpallgc
    entersweep(L); /* continue as an incremental cycle */
    /* set a debt equal to the step size */
    //设置debt, 100的LUA_GCPSTEPSIZE ( 百分比 · 默认值为250% ) · 即debt=250
    luaE_setdebt(g, applygcparam(g, STEPSIZE, 100));
}
```

- major gc:

```
...
case GCSEnteratomic: {
    work = atomic(L);
    if (checkmajorminor(L, g))
        entersweep(L);
    break;
}
...
```

- checkmajorminor:

```
static int checkmajorminor (lua_State *L, global_State *g) {
    if (g->gckind == KGC_GENMAJOR) { /* generational mode? */
        //当前元素总数
        l_obj numobjs = gettotalobjs(g);
        //相较于上次minor gc增加的元素个数
        l_obj addedobjs = numobjs - g->GCmajorminor;
        //新增元素的LUAJ_MAJORMINOR ( 百分比 · 默认值为50%)
        l_obj limit = applygcparam(g, MAJORMINOR, addedobjs);
        //会被GC的元素个数
        l_obj tobecollected = numobjs - g->marked;
        if (tobecollected > limit) {
            //切换到minor gc
            //如果是白色则直接释放 · 否则直接设置为G_OLD。
            atomic2gen(L, g); /* return to generational mode */
            setminordebt(g);
            return 0; /* exit incremental collection */
        }
    }
    g->GCmajorminor = g->marked; /* prepare for next collection */
}
```

```
return 1; /* stay doing incremental collections */
}
```

3、两种GC模式实际应用的思考

当业务逻辑用lua来做开发时，当服务器执行一段时间后，可以将GC模式从增量式GC切换为分代式GC，这样能提升服务器性能。但是需要将那种一直会销毁变动的数据区分清楚，这种数据在缓存中时最好不要用lua对象来保存，比如玩家数据，玩家经常性的会有登陆登出操作，而且玩家数据也是常变动的，这会导致分代式GC会有较大可能触发major gc，major gc虽然复用了增量式GC的代码，分步执行，但它每步执行完不会重新设置debt，下次新增元素又会进gc，那么就有可能导致CPU突刺，这种结果是与使用分代式GC的初衷相违背的。

D、热更

- 跟热更相关的表：
 - loaded表，l_registry表的key为LUA_LOADED_TABLE（_LOADED）。
 - searchers表，l_registry表的key为searchers。
- require相关伪代码如下所示：

```
static int ll_require (lua_State *L) {
    //获取传入的文件名
    const char *name = luaL_checkstring(L, 1);
    lua_settop(L, 1); /* LOADED table will be at index 2 */
    //查找_LOADED表里是否已经加载过对应模块
    lua_getfield(L, LUA_REGISTRYINDEX, LUA_LOADED_TABLE);
    lua_getfield(L, 2, name); /* LOADED[name] */
    if (lua_toboolean(L, -1)) /* is it there? */
        return 1; /* package is already loaded */

    ...

    findloader(L, name);

    ...

    //执行findloader生成的对象
    lua_call(L, 2, 1); /* run loader to load module */

    ...

    return 2; /* return module result and loader data */
}

static void findloader (lua_State *L, const char *name) {
    ...
    //searchers表在createsearcherstable函数初始化，
    //searchers表保存的是几个函数指针，分别是searcher_preload、searcher_Lua、searcher_C、
    searcher_Croot
```

```
if (l_unlikely(lua_getfield(L, lua_upvalueindex(1), "searchers")
                != LUA_TTABLE))
    luaL_error(L, "'package.searchers' must be a table");

//首先在preload表中查找
//调用searcher_Lua，最终会执行lua_load加载解析lua文件，生成LClosure对象。成功则返回
//searcher_C、searcher_Croot加载的是C库
...
}
```

- 因此lua脚本热更新只需要将对应的package.loaded["文件名"]=nil，然后再require("文件名")就可以了。但需要注意的是重新加载的文件中定义的变量的继承问题，这个是热更新的重点。

一个简单的例子：

老的A.lua的代码如下：

```
local ATest = {1,2,3}
return ATest
```

B.lua的代码如下所示：

```
BTest = require("A")
```

新的A.lua的代码如下：

```
local ATest = {4,5,6}
return ATest
```

当新的A.lua重新require后，BTest的数据还是1，2，3。