# Lua5.3.3 GC运行分析

CaseZheng 2016-12-26

Lua GC是增量式GC，不会因为单次GC时间过长导致主逻辑卡顿。

Lua GC使用了标记清除算法，Lua的所有对象引用都在虚拟机的准确管理下，因此可以准确的处理对象的引用情况。

标记清除算法：包含标记和清除两个主要阶段。标记阶段查找所有被对象集合直接或间接引用的对象。清除阶段释放所有没有被标记的对象，将被标记的对象的标记清除。

增量式GC：Lua的增量式GC主要体现在标记和各个清除阶段是可以分多次执行的，每次执行后都暂停GC，之后再继续向下处理。

GC在标记阶段使用三种颜色来标记对象

白色：未被引用

灰色：被引用，切本身包含未进一步处理的对其它对象的引用

黑色：被引用，且无更多对其它对象的引用

GC的基本阶段：

GCSpause: 处于两次完整 GC 流程中间的休息状态，GCSpause 到 GCSpropagate ，一次性标记 root set

GCSpropagate: 可分多次执行，直到 gray 链表处理完，进入GCSatomic

GCSatomic: 一次性的处理所有需要回顾一遍的地方, 保证一致性, 然后进入清理阶段 GCSswpallgc

GCSswpallgc: 清理 allgc,可以多次执行,清理完完进入 GCSswpfinobj

GCSswpfinobj: 清理 finobj,可分多次执行,清理完进入 GCSswptobefnz

GCSswptobefnz: 清理 tobefnz,可分多次执行,清理完后进入GCSswpend

GCSswpend: sweep main thread 然后进入 GCScallfin

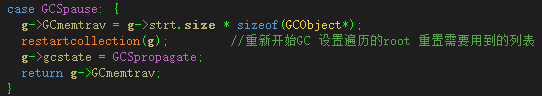
GCScallfin: 执行一些 finalizer (\_\_gc) ，可分多次执行，然后进入 GCSpause,完成循环

Lua增量式GC在一些阶段不可暂停，包括：GCSpause到GCSpropagate阶段转化时需要一次性标记根对象集合（root set）。GCSpropagate到GCSswpallgc阶段有个GCSatomic阶段，GCSatomic阶段保持一致性不可暂停。GCSswpend阶段清除主线程，不可暂停。

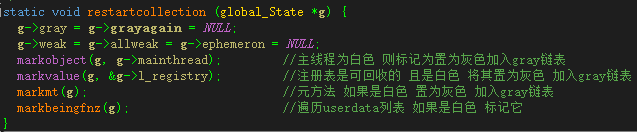
由于GC是增量式，那么在GC暂停时，有可能出现创建新对象或者改变对象的引用的情况。如果创建新对象，不需要特殊处理，GCSatomic阶段保证遍历所有thread的stack进行标记。如果是改变了引用的情况，则需要在改变引用的地方通过设置write barrier来保证不会有活着的对象被漏标记的情况。( 可以允许暂时错误标记没有被引用的对象, 这样没有错, 可以等下一次正常 gc 流程清理掉, 只是这种情况效率较低 )

Lua每执行一步最终都会到singlestep函数中去。

1.GCSpause -> GCSpropagate

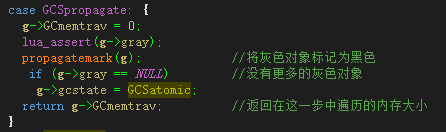


可以看到从GCSPause到GCSpropagate阶段一步执行完毕，不能分多次执行。



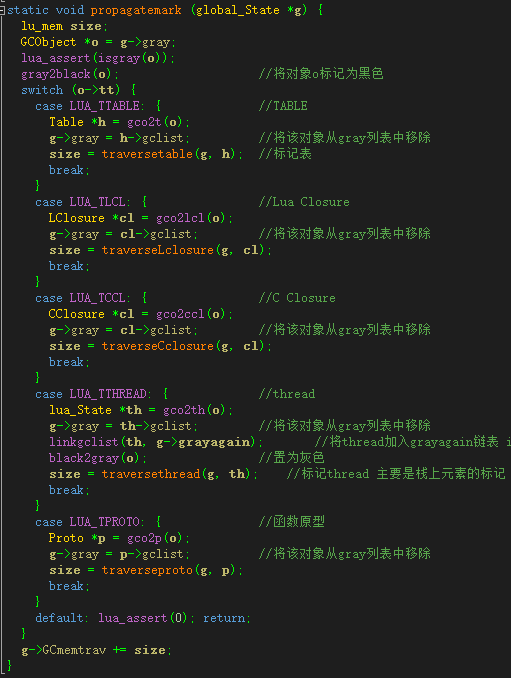
进入restartcollection可以看到其主要是重置了GC将用到了几个列表，将主线程、注册表、基本类型的元方法、userdata对象全部置为灰色，即设置标记时出发的root set。

2.GCSpropagate -> GCSatomic

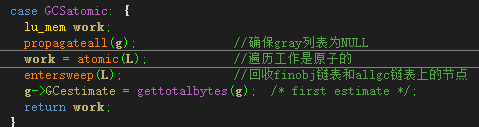


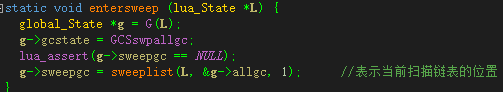
在这里可以看到只有当g->gray为NULL才会进入GCSatomic阶段，即GCSpropagate到GCSatomic阶段可以分多次执行，知道g->gray为NULL才进入GCSatimic阶段。

进入propagatemark可以看到其对table、lua closure、C closure、thread、proto分别进行不同的操作，但共同点是找到被引用对象，标记它，根据不同情况将其加入不同链表。此处可以看到thread始终是灰色的。

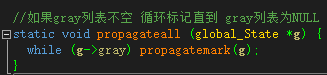


3.GCSatomic -> GCSswpallgc





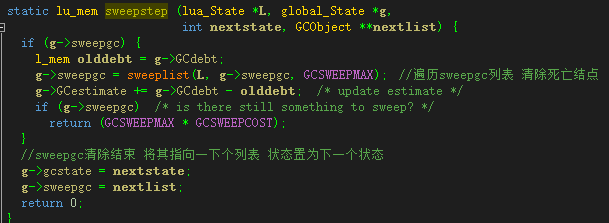
可以看到，从GCSatomic到GCSswpallgc是不可分步执行的。



先看propagateall主要是遍历g->gray调用propagatemark清空g->gray列表。然后进入一个不可分割的原子操作，处理需要回顾的地方，保持一致性。经过这次处理，Lua虚拟机里面就不再有灰色对象了，只剩下黑色和白色。对于atomic后面有时间再分析。再然后会进入GCSswpallgc阶段，并将当前需要清理的列表g->allgc赋给g->sweepgc。

4.GCSswpallgc -> GCSswpfinobj





在这一步我们可以看到清理了g->allgc列表 清理的数量为GCSWEEPMAX 80个。如果清理g->allgc列表为空切换到要清理的下一个状态，设置要清理的列表，否则直接返回，下次GC还会清理g->allgc列表。



sweeplist根据当前Lua虚拟机的白色bit位与对象的颜色位如果是dead 从链表删除，释放内存，否则，将对象的颜色设为与当前Lua虚拟机相同。

5.GCSswpfinobj -> GCSswptobefnz



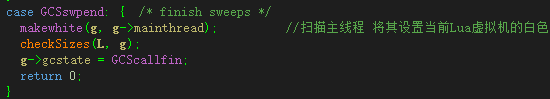
与GCSswpallgc -> GCSwpfinobj类似，清理的列表是g->finobj

6.GCSswptobefnz -> GCSsWpend

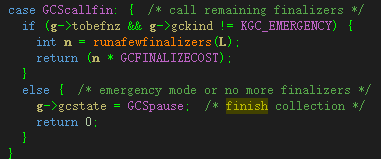


与GCSswpallgc -> GCSwpfinobj类似，清理的列表是g->tobefnz

7.GCSswpend -> GCScallfin



8.GCScallfin -> GCSpause



GCScallfin -> GCSpause可以看出可多次执行，执行finalizer(\_\_gc)然后进入GCSpause完成GC的一个循环。

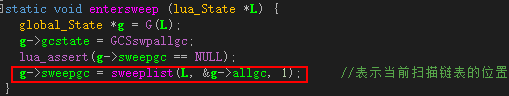
sweep阶段创建新的对象，会不会有问题？

即sweep 的工作是干掉白色对象，把黑色对象涂回白色，新创建的对象是挂在allgc头部的白色对象。那么新创建的对象会不会又被gc干掉的可能？

这种情况不会发生。

1.进入 sweep 阶段之前 white bit 进行了 flip，此时判断死亡用的是 other white bit, 跟新创建对象用的 white bit 不一样。

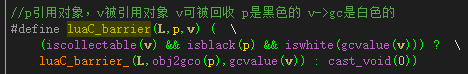
2. 进入 sweep 阶段前 entersweep 函数会线进行1次单步sweep,让 sweepgc 指针指向 allgc 链表内部而不是头部. 这样的目的是避免正式开始 sweep 阶段时, 还需要花时间跳过链表头部现在到正式开始 sweep 阶段之间新创建的对象们.

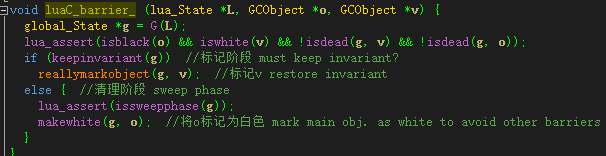


改变对象的引用，gc会不会有问题？

write barrier 用来保证增量式GC在GC流程中暂停时，对象引用状态的改变不会引起GC流程产生错误的结果

1. luaC\_barrier



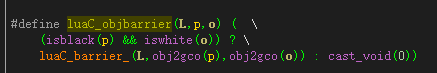


对象p指向对象v，对象v可被回收，p是黑色，v->gc是白色

标记阶段，标记v

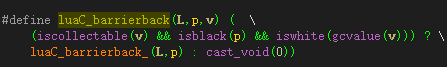
清理阶段，将v直接标记为白色

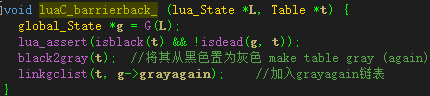
2. luaC\_objbarrier



和luaC\_barrier雷同，不同的是luaC\_barrier针对的是TValue，luaC\_objbarrier针对GCObject。

3. luaC\_barrierback





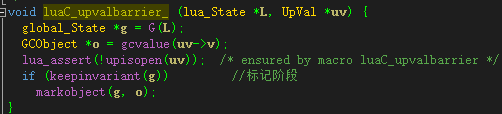
针对给对象设置元表时元表的状态设置，将t从黑色置为灰色，加入grayagain列表，这样可以进一步处理t中引用的进一步变化。如果使用luaC\_barrier，会导致t可能被错误标记

如local u = new\_userdata()  
setmetatable(u, mt) //使用luaC\_barrier mt被标记为白色  
setmetatable(u, nil)  
会导致 mt 被错误标记, 要下一次 gc 循环才能正确清理

如果直接将u标记为灰色，那么就可以处理这种变化。

4. luaC\_upvalbarrier





针对close upvalue，upvalue的值可被回收，且upvalue是close的，标记它即可。

thread的特殊之处是什么？

1.thread是由lua\_newthread创建的，而别的对象由luaC\_newobj创建。

2.thread刚创建时是白色，在标记阶段将一直是灰色。在清理阶段再标记为白色。因为Lua在执行时其栈一直在变化，其引用关系也在变化，所以thread一直是灰色的。

为什么要有grayagain列表，和gary列表的不同？

当 propagatemark 处理所有 gray 对象时, thread 对象也会被从 gray 链表中拿出来, thread 总是灰色的, 被处理完后也不会变成黑色, 而是保持灰色, 但是也不能这样直接放回 gray 链表, 否则 gray 链表就永远处理不完了. 所以另外放到一个叫 grayagain 的链表中。另外在 barrierback 系列函数中, 对象被涂成灰色, 也不是放到 gray 链表, 而是放到 grayagain 链表。在 traverseweakvalue 和 traverseephemoron 中也有把对象挂到 grayagain 上。grayagain 最后是在 atomic 阶段的 retraversegrays 中处理

Lua GC运行可以由外部通过collectgarbage直接调用。也可以由程序内部触发。

1.外部调用，可以完成一次完整循环，也可以执行一个单步。

collectgarbage ([opt [, arg]])

垃圾收集器的通用接口，通过参数opt提供一组不同的功能：

* **"**collect**":** 做一次完整的垃圾收集循环。 这是默认选项。
* **"**stop**":** 停止垃圾收集器的运行。 在调用重启前，收集器只会因显式的调用运行。
* **"**restart**":** 重启垃圾收集器的自动运行。
* **"**count**":** 以 K 字节数为单位返回 Lua 使用的总内存数。 这个值有小数部分，所以只需要乘上 1024 就能得到 Lua 使用的准确字节数（除非溢出）。
* **"**step**":** 单步运行垃圾收集器。 步长“大小”由 arg 控制。 传入 0 时，收集器步进（不可分割的）一步。 传入非 0 值， 收集器收集相当于 Lua 分配这些多（K 字节）内存的工作。如果收集器结束一个循环将返回 **true** 。
* **"**setpause**":** 将 arg 设为收集器的 间歇率。返回 间歇率 的前一个值。
* **"**setstepmul**":** 将 arg 设为收集器的 步进倍率。返回 步进倍率 的前一个值。
* **"**isrunning**":** 返回表示收集器是否在工作的布尔值 （即未被停止）。

collectgarbage在源代码中对应的C API是luaB\_collectgarbage

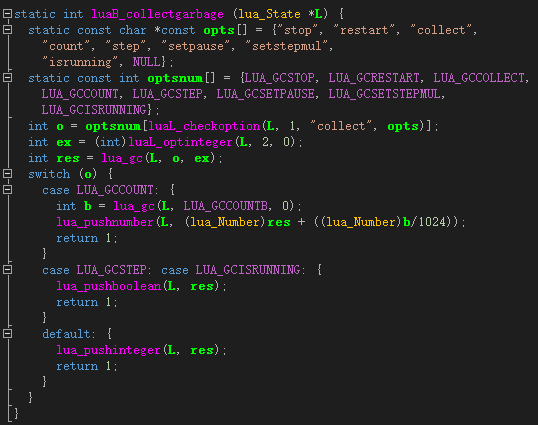
static const luaL\_Reg **base\_funcs**[] = {

...

{"collectgarbage", luaB\_collectgarbage},

...

}

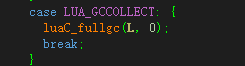


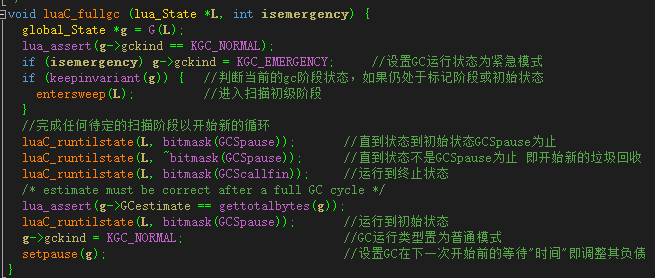
通过collectgarbage可以调用GC运行，清理垃圾内存。

lua\_gc函数，会根据传入参数进入不同函数处理。

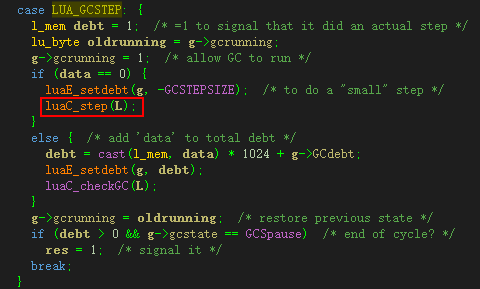


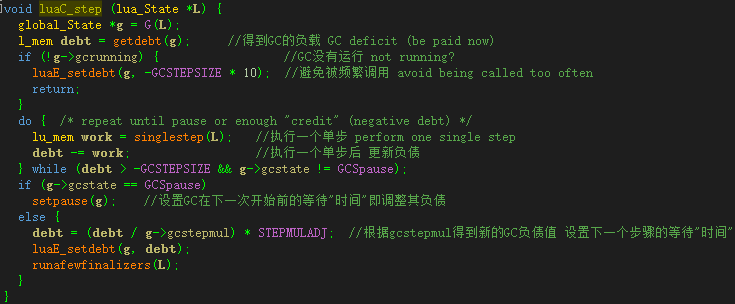
完整循环一次。





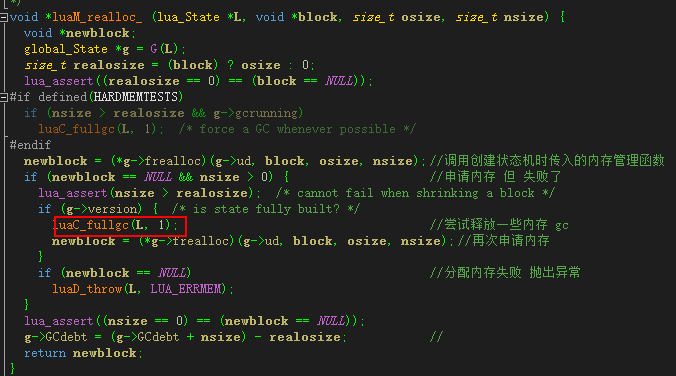
单步执行一次。





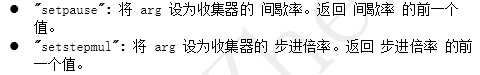
2.分配内存不足时自动调用，完成一次完整循环。但此时设置紧急模式。

此时，标记thread不会标记栈，不会收缩栈，不会去清理tobefnz列表，



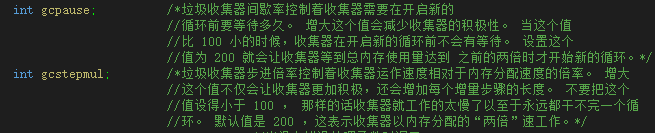
3.Lua在运行时检查GC发现其负债超过限定，会单步执行GC

上面3点即LuaGC运行的时机，第三点是Lua自动单步运行GC，那么这个自动运行的间隙如何控制呢？

这个可以看上面的collectgarbage ([opt [, arg]])函数。其中的

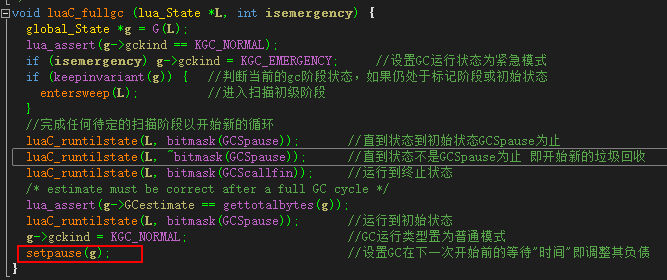
用来控制Lua自动清理内存的时机。

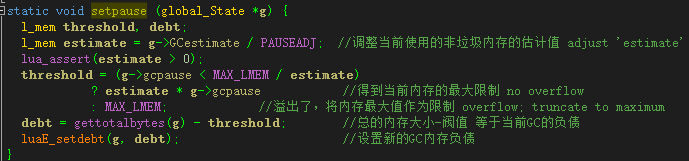
其值保存在global\_State中

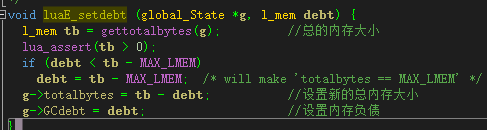


通俗的讲pause是一次Lua GC完成循环后的等待“时间”。而stepmul是GC在一次循环过程中，一步结束后，下一步清理的间隔“时间”。

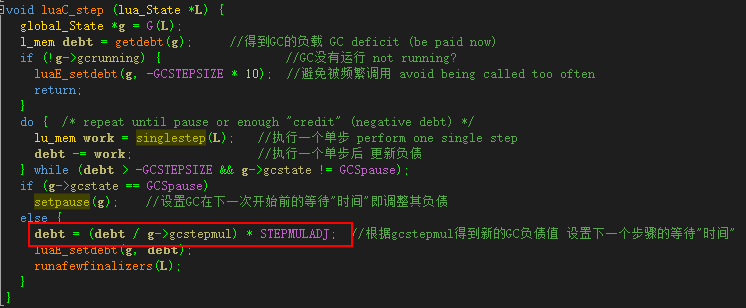
所以pause用于GC一次完成循环后







而stepmul用于一步执行完后



参考文档：

1.http://www.android100.org/html/201502/05/112408.html

2.https://zhuanlan.zhihu.com/p/22403251

3.Lua程序设计中文第二版

4.lua 源码鉴赏

5.Lua5.3.3参考文档

6.Lua5.3.3源代码

7.网上其它博客