个人简介

网名：codedump

《Lua设计与实现》作者

Agenda

1.接触Lua的历程，Lua在游戏行业的使用

2.Lua 5.1.4 GC分析

Lua在游戏行业的使用

纯使用编译类语言开发游戏

1.编码、编译

2.重启服务器

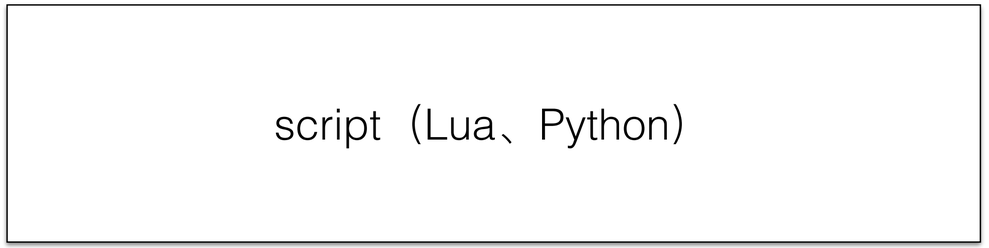
3.开发周期短，迭代速度快。

4.coredump、内存泄露 etc C\C++对个人要求高。

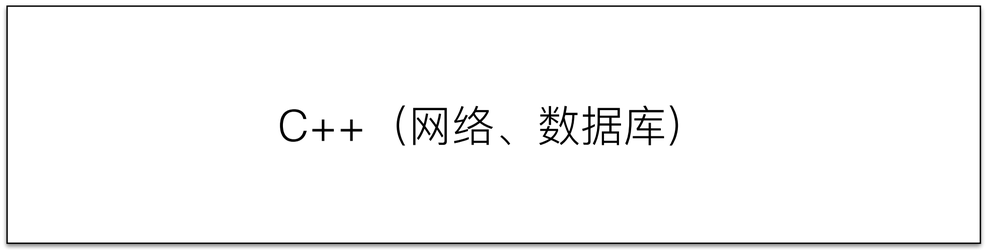
Lua在游戏行业的使用

C++搭配脚本语言

脚本层



核心层



基于引用计数的GC算法

优点：

没有特定的GC过程，开销均摊在过程中。

缺点：

循环引用

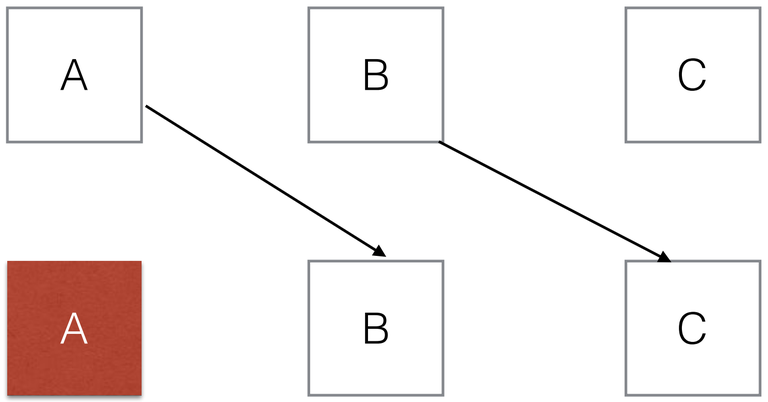
标记回收算法（Mark and Swap）

标记阶段：

遍历所有的对象，标记所有被其他对象所引用的对象

回收阶段：

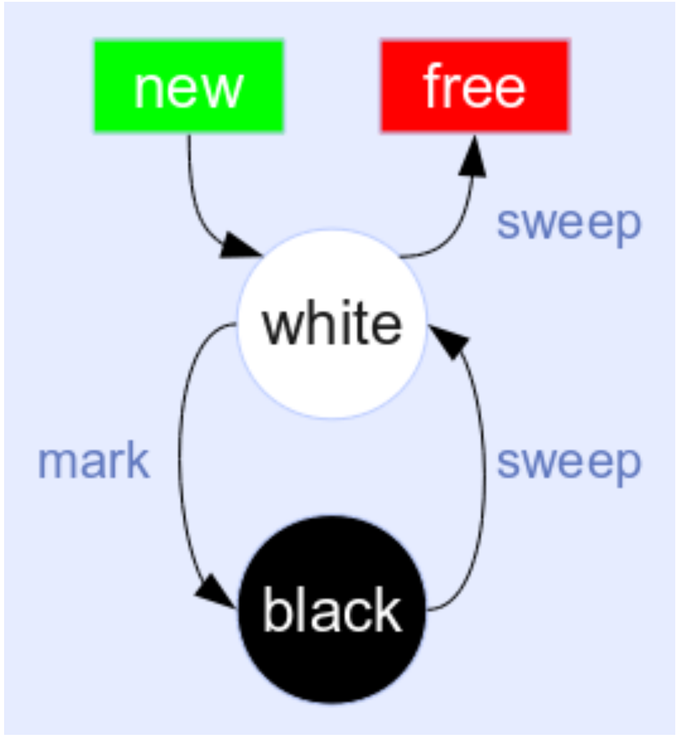
将所有不被其他对象引用的对象回收



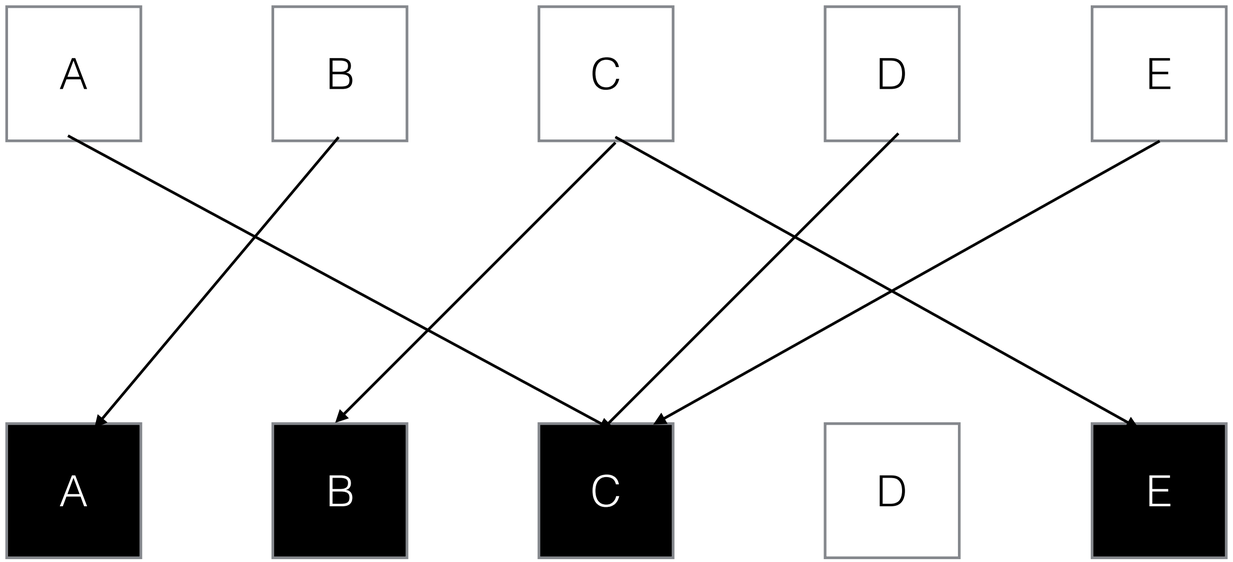
双色标记回收算法(Two-Color Mark and Sweep)

白色-每个新创建对象的颜色

黑色-被其他对象所引用的对象的颜色

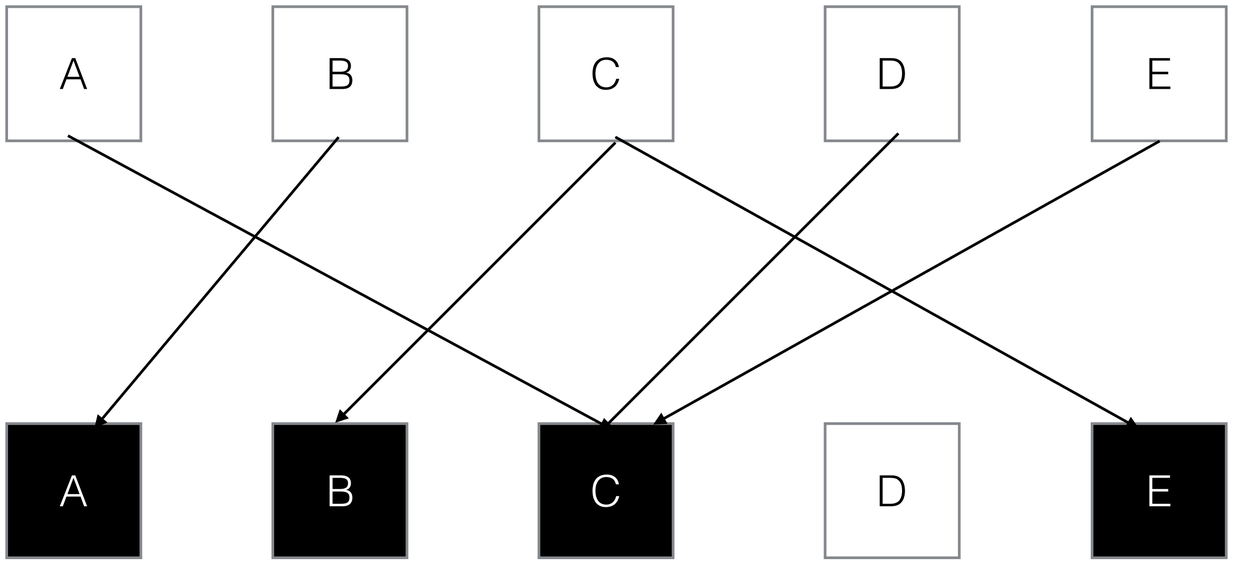


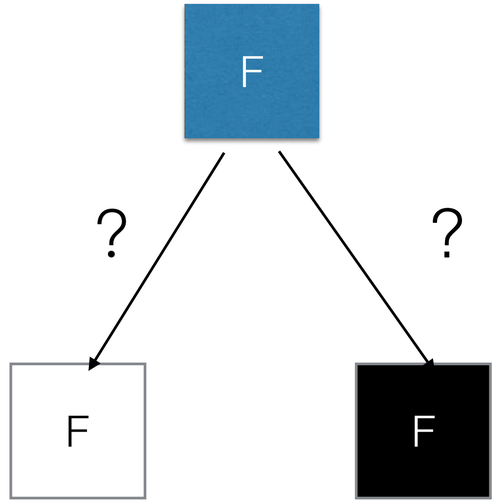
双色标记回收算法(Two-Color Mark and Sweep)



双色标记回收算法(Two-Color Mark and Sweep)

是否可以被中断?





双色标记回收算法(Two-Color Mark and Sweep)

缺陷

不可被中断，需要一次性完成GC所有流程

粗粒度的操作

对实时性要求高的系统影响大

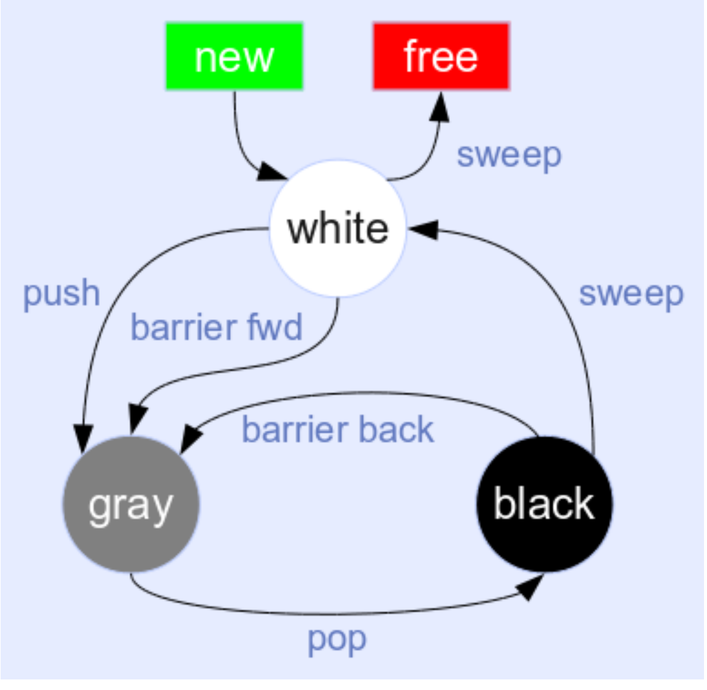
三色标记回收算法(Tri-Color Mark and Sweep)

Lua 5.1版本GC算法

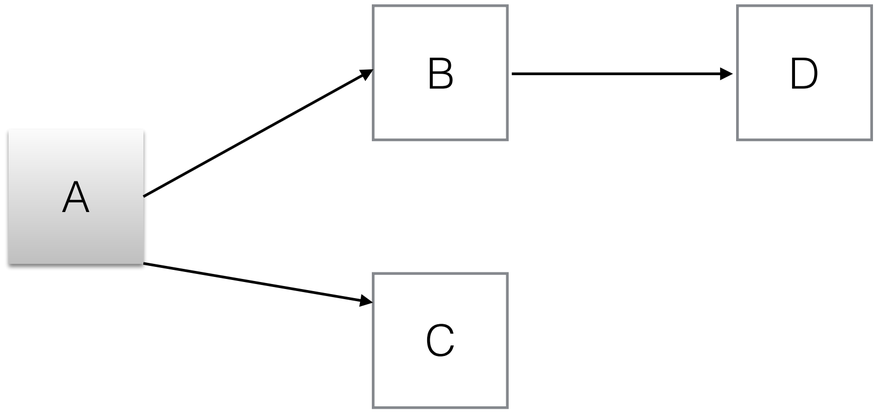
白色-每个新创建对象的颜色

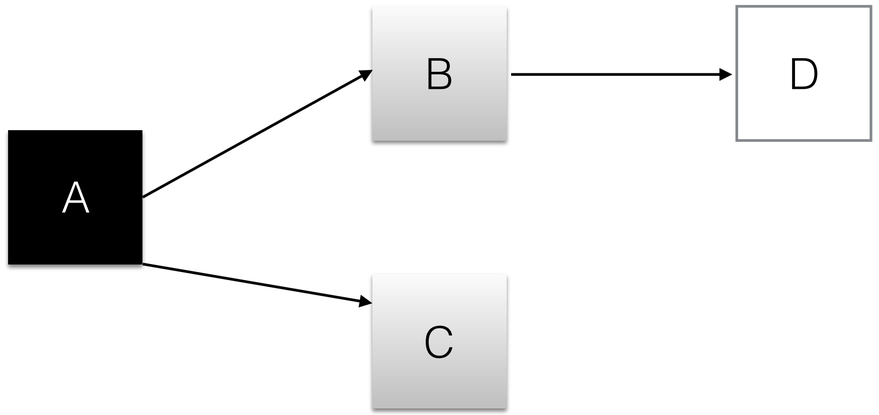
黑色-被其他对象所引用的对象的颜色

灰色-待扫描状态，对象已经被访问过，但是其引用的其他对象没有被访问过。



三色标记回收算法(Tri-Color Mark and Sweep)





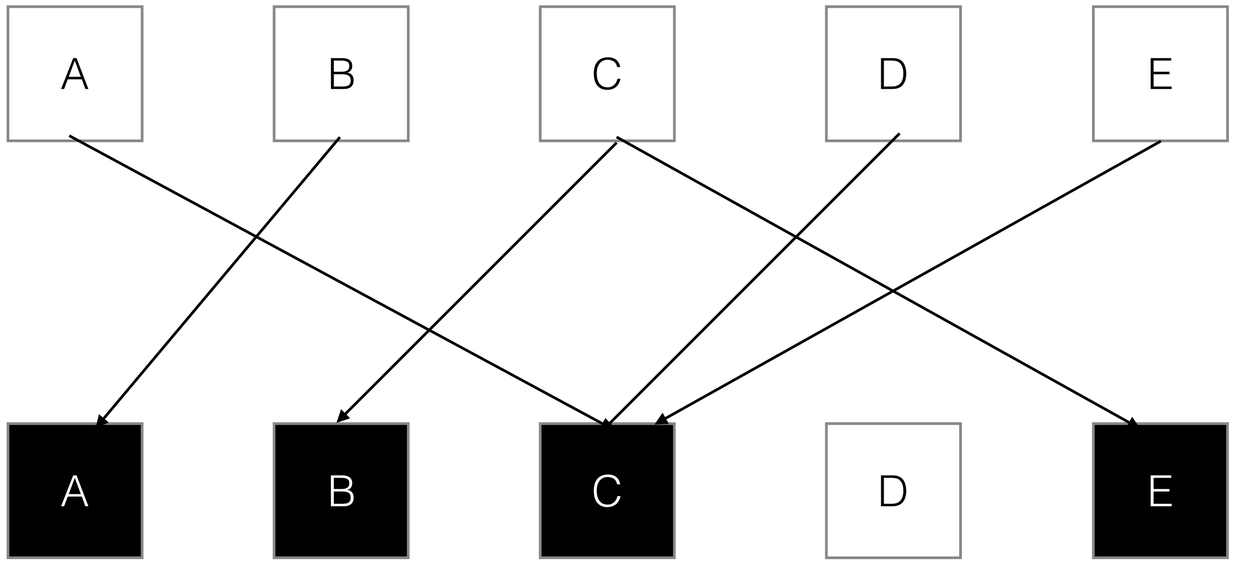
三色标记回收算法(Tri-Color Mark and Sweep)

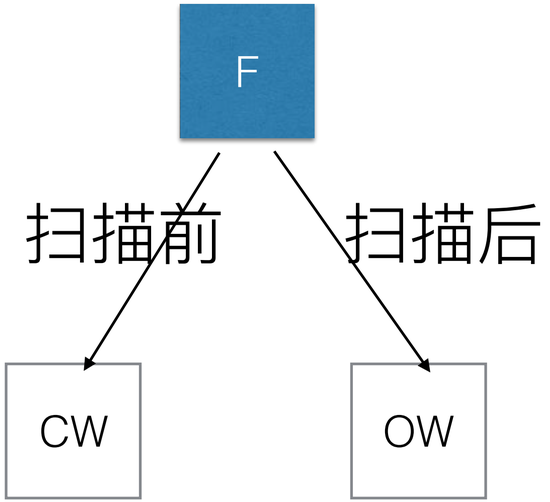
两种白色类型

当前白色（currentwhite）：当前GC的白色类型

其他白色（otherwhite）：下一次GC的白色类型

三色标记回收算法(Tri-Color Mark and Sweep)





三色标记回收算法(Tri-Color Mark and Sweep)

初始化阶段：（此时创建的对象标记为currentwhite）

将系统的mainthread、G表、registry表标记为灰色。

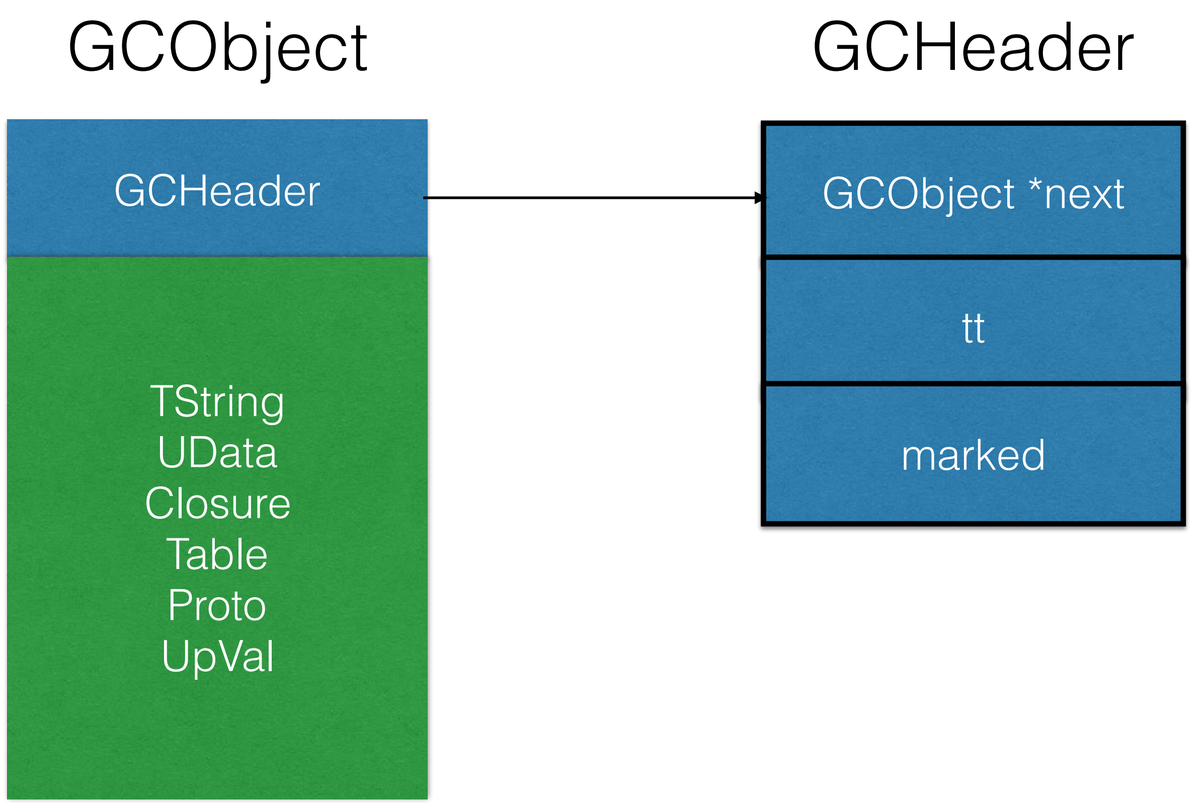
扫描阶段：（此时创建的对象标记为currentwhite）

当系统中还存在灰色节点的情况下，标记该对象以及其引用到的对象。

回收阶段：（此时创建的对象标记为otherwhite）

回收系统中所有颜色为currentwhite的对象。

Lua 5.1.4的GC实现 - 相关数据结构



next用于保存gc链表上下一个元素

tt用于保存数据类型

marked用于保存数据的颜色

Lua 5.1.4的GC实现 - 相关数据结构

Lua VM相关数据结构

GCObject *rootgc：对象刚创建时存放在该链表中。该链表的对象都是白色。

GCObject *gray：灰色对象存放在该链表中。

GCObject *grayagain：需要一次性不可被打断的灰色对象存放在该链表中。

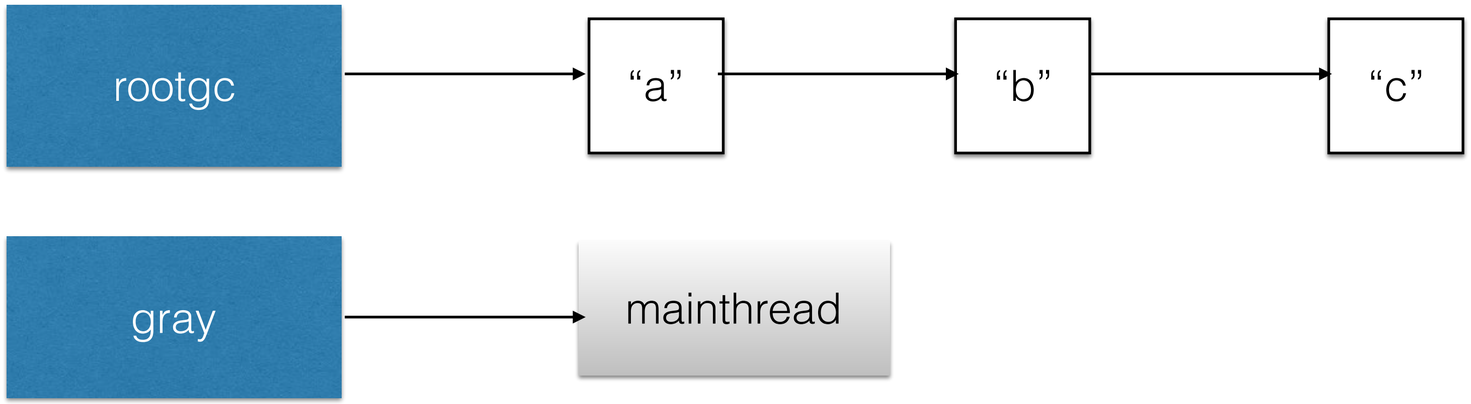
Lua 5.1.4的GC实现 - 例子

一段简单的代码：

```
a = “a”
b = “b”
c = “c”
c = nil
collectgarbage()
```

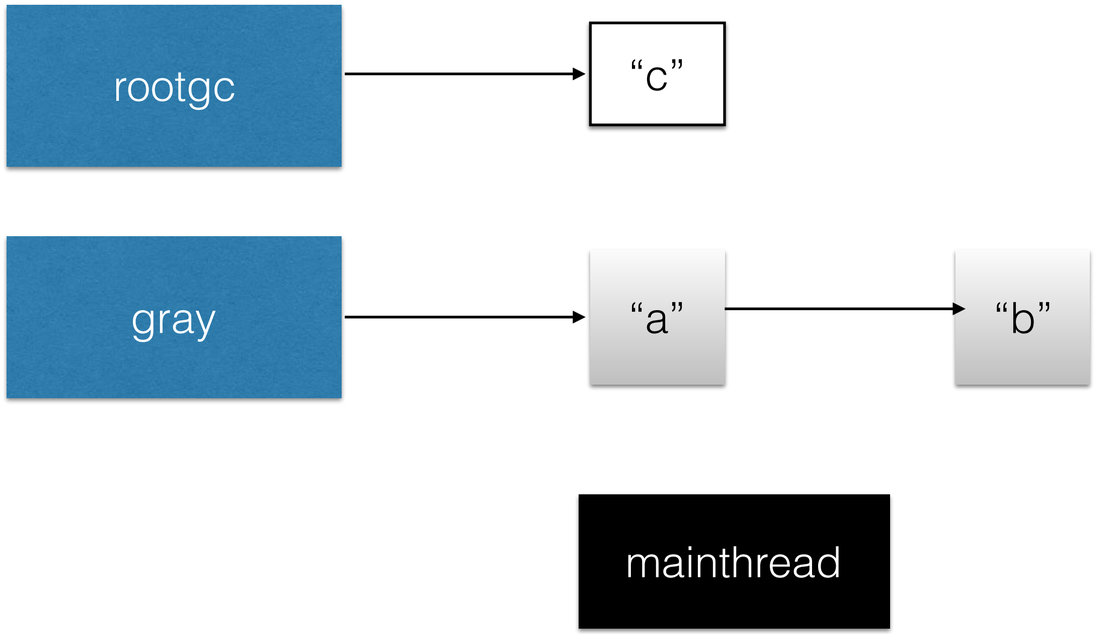
Lua 5.1.4的GC实现 - 例子

GC初始化



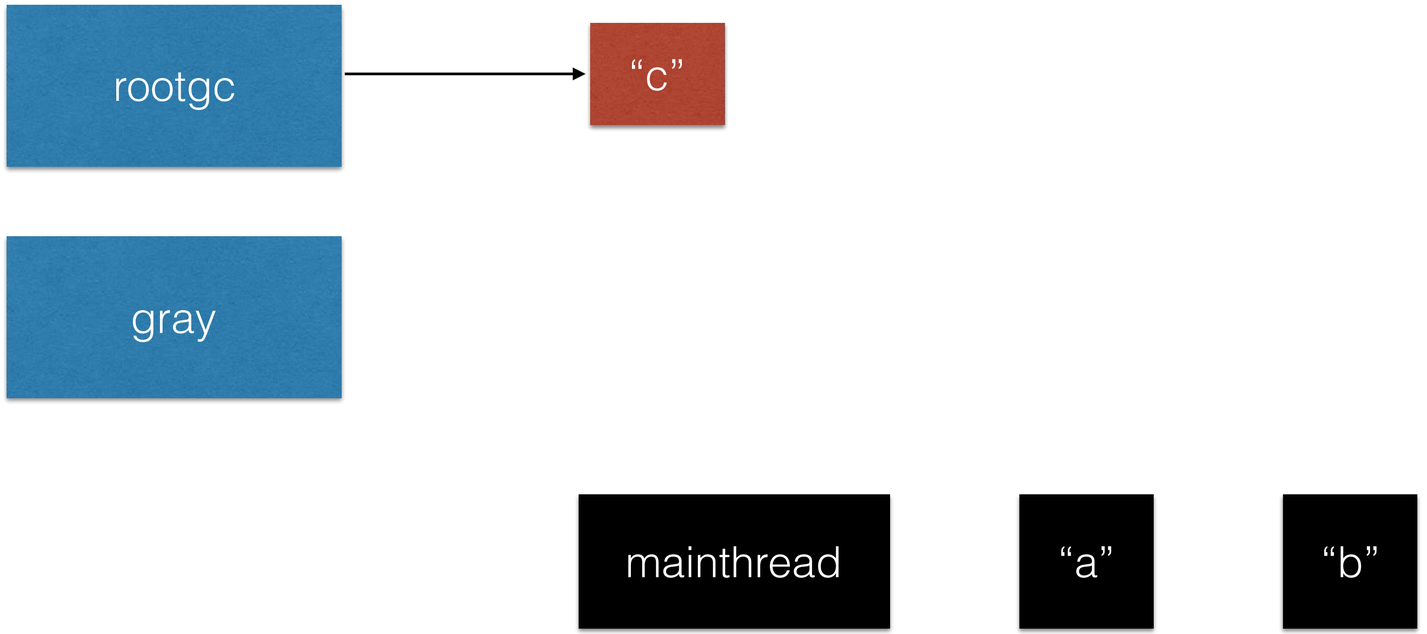
Lua 5.1.4的GC实现 - 例子

GC扫描阶段



Lua 5.1.4的GC实现 - 例子

GC清理阶段



Lua 5.1.4的GC实现 - 屏障（barrier）

引入屏障需要解决什么问题?

系统中某个对象a已经在标记为黑色的情况下，它新增了一个引用的对象，此时应该如何处理?

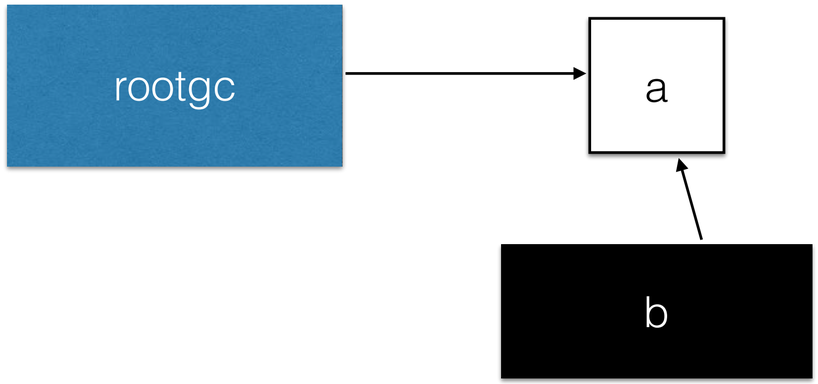
Lua 5.1.4的GC实现 - 屏障（barrier）

向前的屏障

对象a如果在变成黑色之后，又引用了新的对象，需要进行向前屏障操作。

Lua 5.1.4的GC实现 - 屏障（barrier）

向前屏障



两种情况：

1） 如果此时还在扫描阶段：对a进行标记操作,相当于a向前走了一步。

2）否则：将b重新置为白色。

Lua 5.1.4的GC实现 - 屏障（barrier）

向后的屏障

将原来已经是黑色的对象，重新加入到grayagain链表中，等待原子的一次性扫描操作。

三种在前面的颜色，回退到哪里？

Lua 5.1.4的GC实现 - 进度控制

singlestep函数的主逻辑：

初始化阶段：

将mainthread、G表、registry表加入gray链表。（不可打断）

扫描阶段：

1.扫描gray链表进行标记（可以打断）

2.扫描grayagain链表进行标记（不可打断）

回收阶段：

1.回收字符串（可以打断）

2.回收其他类型对象（可以打断）

Lua 5.1.4的GC实现 - 进度控制

Lua VM中涉及到GC进度控制的几个参数

totalbytes：当前分配的数据大小。
GCthreshold：开启GC的阈值，只要totalbytes大于这个值就开始GC。

estimate：预估的值，用于与gcpause配合设置GCthreshold
gcpause：用于设置GCthreshold为estimate的百分之几。

gcstepmul：用于进度控制，控制每次GC操作回收多少内存。

sweepstrgc: 保存上一次回收被打断时字符串回收的位置
sweepgc：保存上一次回收被打断时其他类型对象回收的位置

Lua 5.1.4的GC实现 - 进度控制

GC进度控制的伪代码：

根据参数计算这次GC回收的内存大小lim

循环：

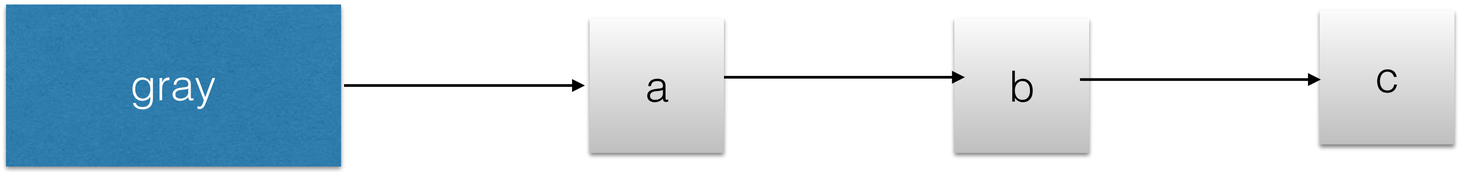
调用singlestep函数进行GC

根据singlestep函数的返回值来修改lim

如果 lim <= 0 或者 当前GC状态为初始化阶段，就终止循环

Lua 5.1.4的GC实现 - 进度控制

GC扫描阶段



GC扫描阶段