3 垃圾回收

- 引用计数
- gc

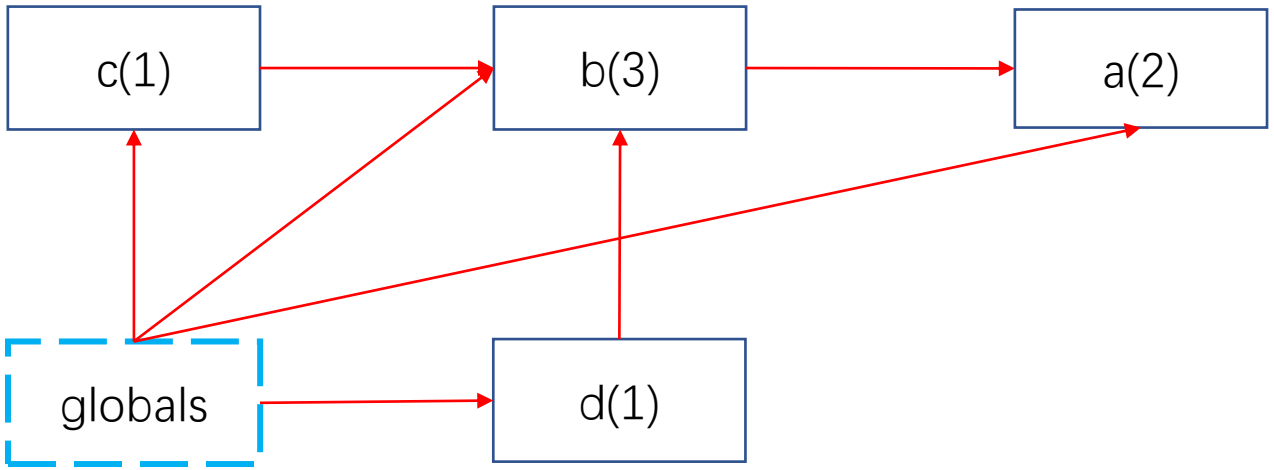
3.1 引用计数

引用计数是python主要使用的垃圾回收算法。垃圾回收(garbage collection)是我们可以不用像c/c++一样手动管理内存的申请、释放。

引用计数算法的主要规则

- 当一个对象被其他对象引用时，计数器 +1
- 当这个引用被释放掉时，计数器 -1
- 当计数器为0时，这个对象被析构
- 当这个对象被析构时，它所引用的其他对象的计数会被减1

3.1 引用计数

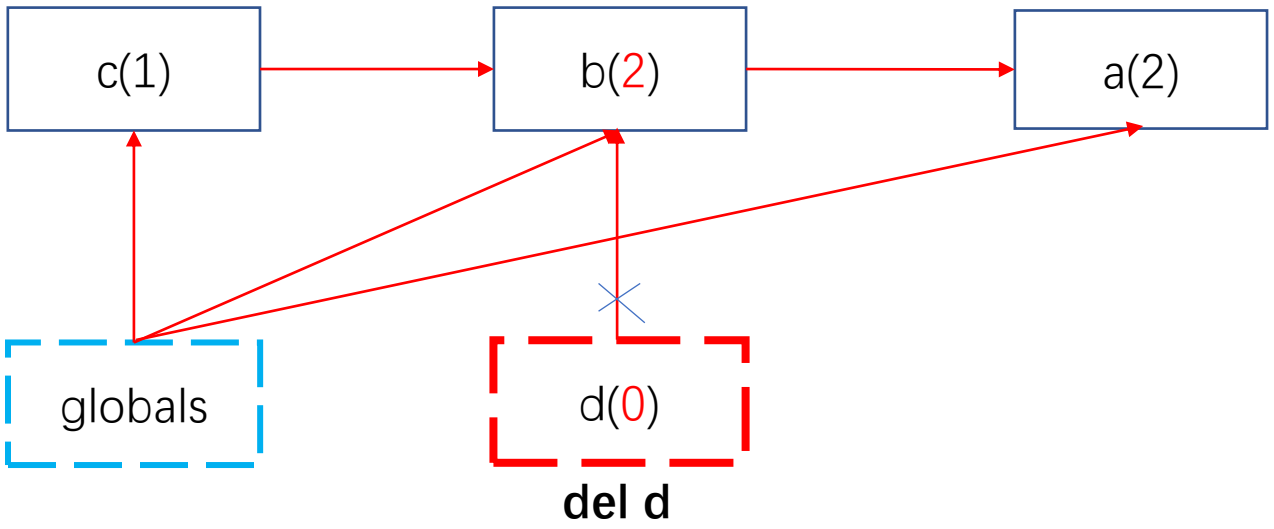



```python
class A:
    pass

a = A()
b = A()
c = A()
d = A()

b.v = a
c.v = b
d.v = b

print(globals())
```

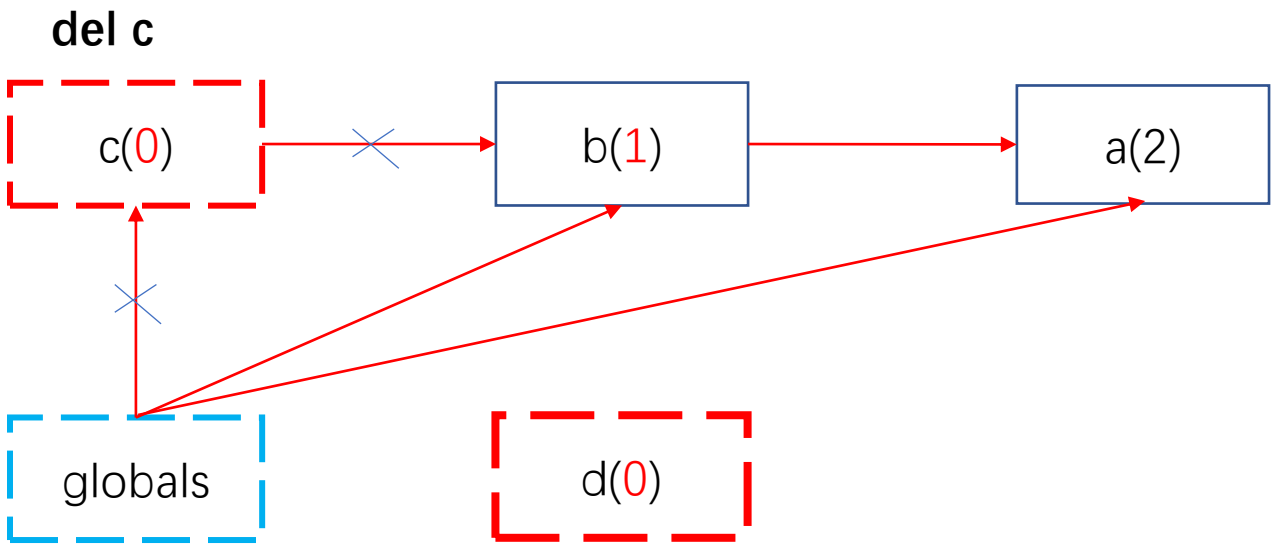
gc2.py

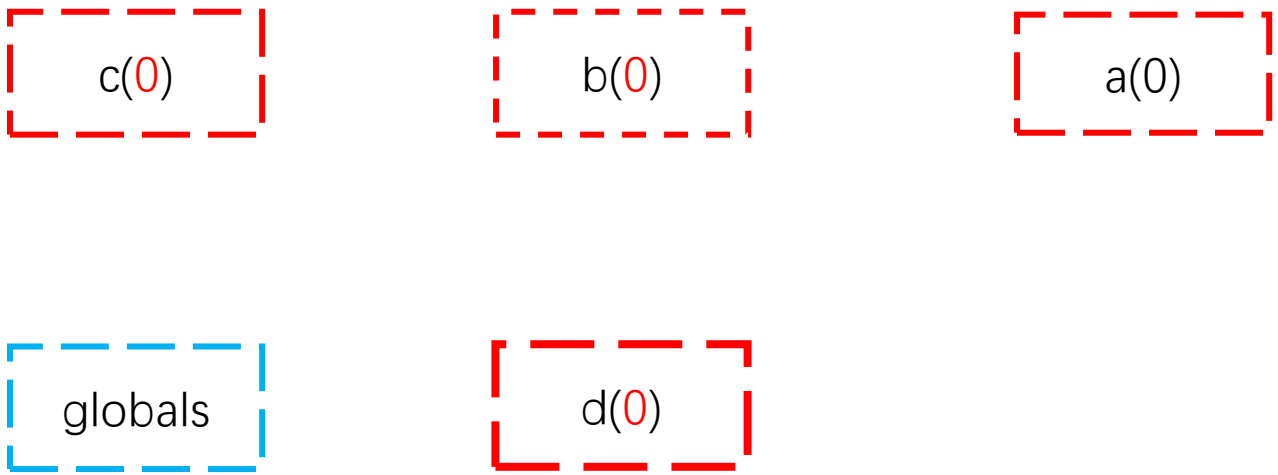
{'__name__': '__main__', '__doc__': None, '__package__': None, '__loader__': <_frozen_importlib_external.SourceFileLoa
object at 0x0000021DE91647C0>, '__spec__': None, '__annotations__': {}, '__builtins__': <module 'builtins' (built-in)>,
'g:\\python\\meta\\garbage_collection\\gc1.py', '__cached__': None, 'sys': <module 'sys' (built-in)>,
'A': <class '__main__.A'>, 'a': <__main__.A object at 0x0000021DE91ABAF0>, 'b': <__main__.A object at
0x0000021DE9047C40>, 'c': <__main__.A object at 0x0000021DE91ABAC0>, 'd': <__main__.A object at
0x0000021DE91ABA90>}

3.1 引用计数



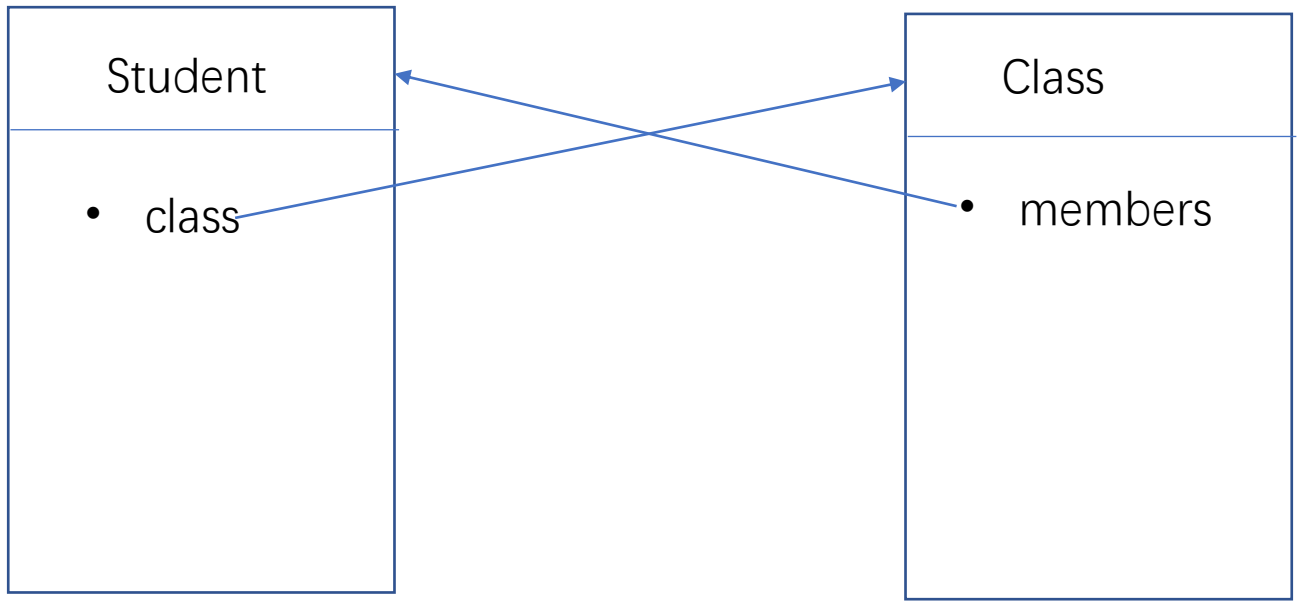
3.1 引用计数




c(0)
b(0)
a(0)
globals
d(0)

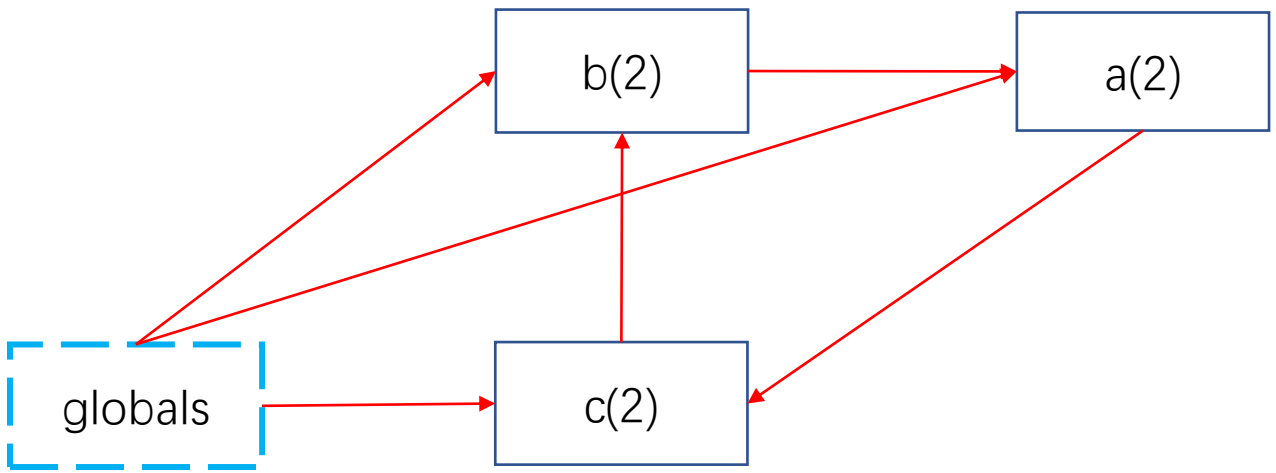
3.1 引用计数

循环引用: 引用关系变成一个环



3.1 引用计数

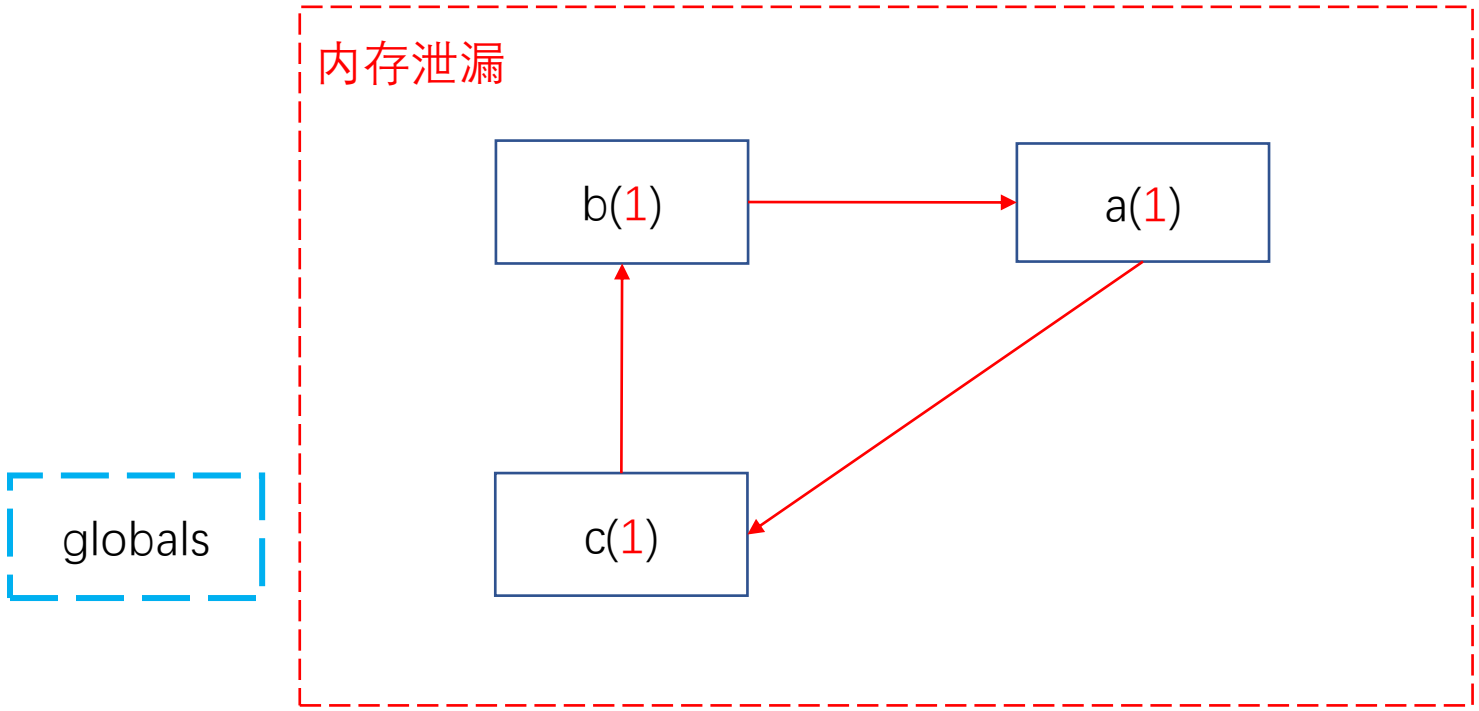
循环引用: 引用关系变成一个环




```python
class A:
    pass

a = A()
b = A()
c = A()

a.v = b
b.v = c
c.v = a
```

3.1 引用计数

循环引用: 引用关系变成一个环



3 gc

- 引用计数
- gc

3.2 gc

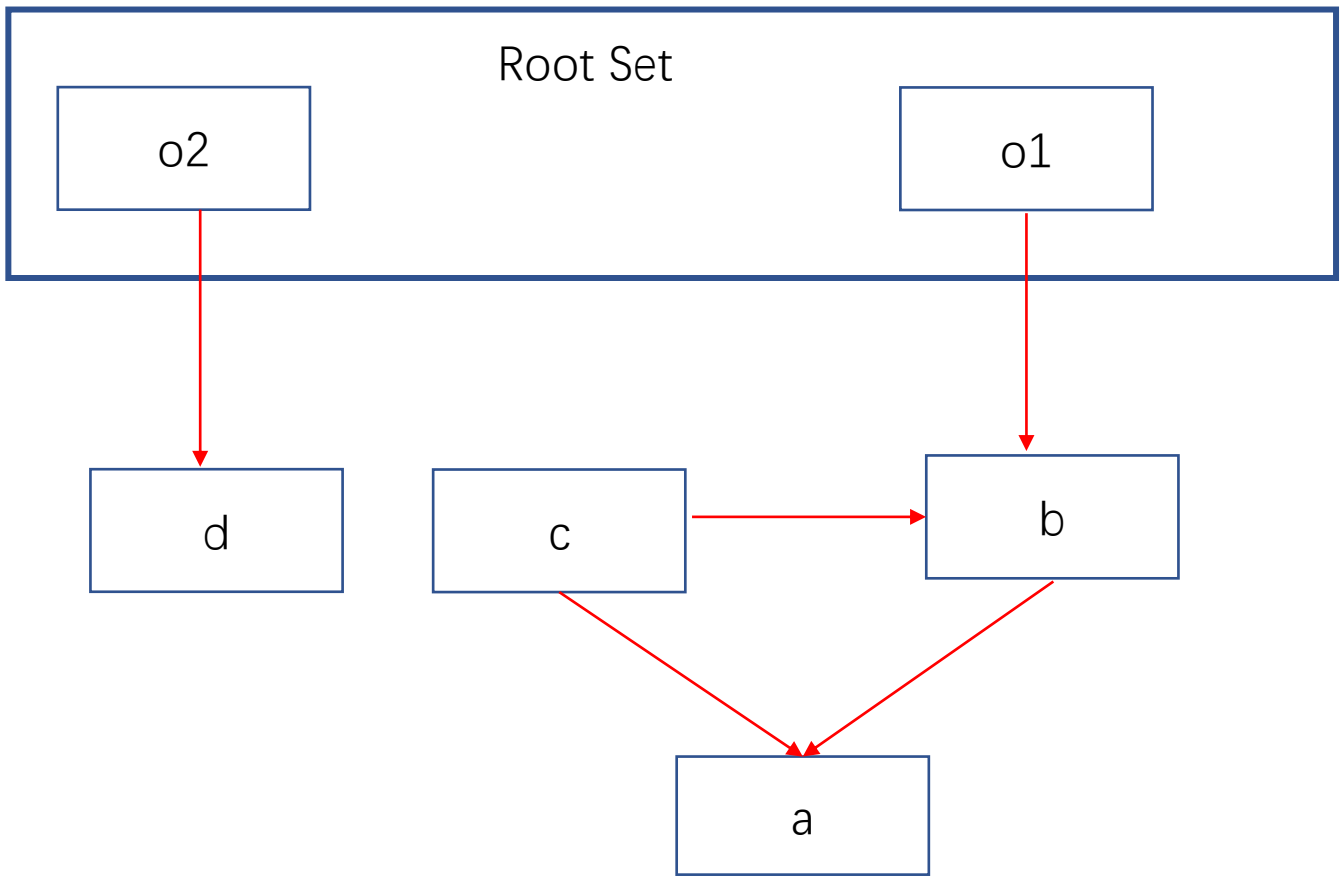
python为了解决循环引用的问题，引用了gc模块，在里面实现了标记-清除（mark-and-sweep)以及世代回收这2个垃圾回收算法。

标记-清除（mark-and-sweep)

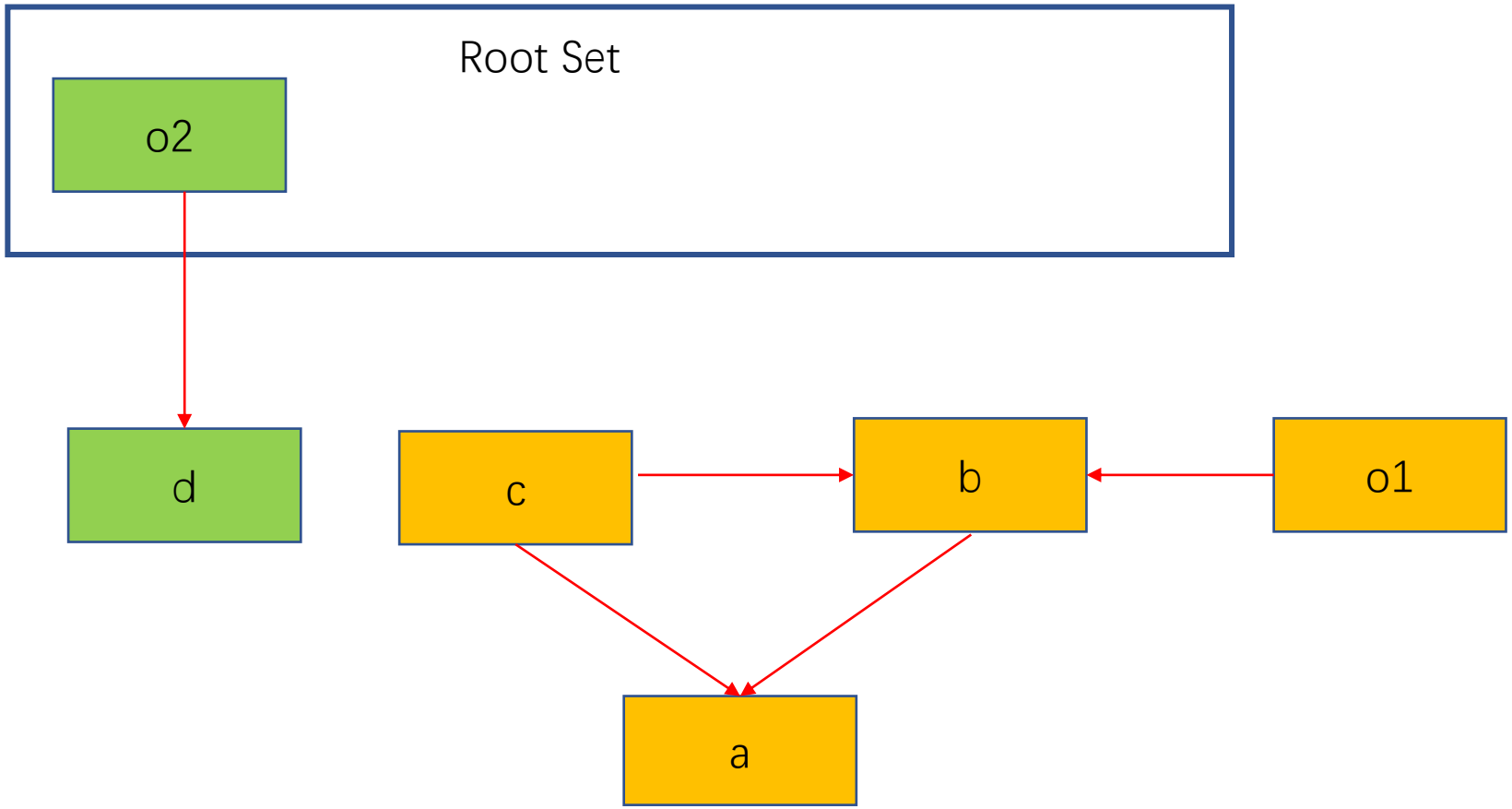
1. **Step 标记(mark)**
   - 设置2个集合 可达集合(reachable)、不可达集合（unreachable)
   - 将根对象标记为可达，并放入可达集合中。从可达集合里的对象开始，遍历所有对象，所有能够访问到对象认为都标记为可达，并放入可达集合中，直至遍历完成

2. **清除(sweep)**
   - 将其他所有对象标记为不可达
   - 并删除不可达对象

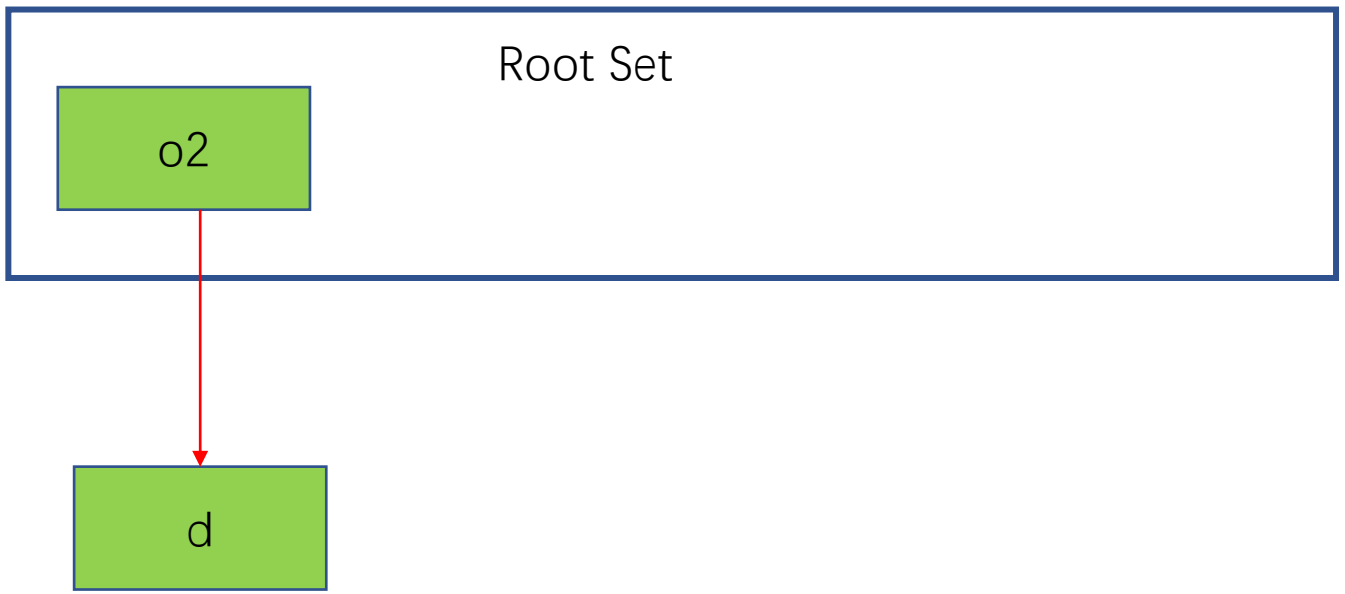
3.2 计数



3.2 计数



3.2 计数



标记-清除（mark-and-sweep)

1. **Step 标记(mark)**
   - 设置2个集合 可达集合(reachable)、不可达集合（unreachable)
   - 将根对象标记为可达，并放入可达集合中。从可达集合里的对象开始，遍历所有对象，所有能够访问到对象认为都标记为可达，并放入可达集合中，直至遍历完成

2. **清除(sweep)**
   - 将其他所有对象标记为不可达
   - 并删除不可达对象

世代回收器（Generation Collector):建立在标记清楚法之上，分代回收解决标记清除时间过长的问题

- 80% - 98% 分配的对象在生成后很短时间内就会被销毁
- 将堆分成多个区
- 所有刚生成的对象都分配到1号区
- 如果1号区满了，将当前所有的对象移动到2号区，并以此类推。
- 区号越小，对象的分配和释放就越活跃

gc module

- **gc.enable**() 启动自动垃圾回收
- **gc.collect**(*[generation]*) 手动进行一次垃圾回收,并返回不可达对象的数量
- **gc.disable()** 关闭自动垃圾回收
- **gc.set_threshold**(*t0, [t1, [t2]]*) 设置自动回收相关参数
  - *t0:* 从上次回收后，如果 分配对象数 – 释放对象数 > t0, 则进行一次世代0的垃圾回收
  - *t1:* 自从上一次世代1检查后，世代0被检查的次数超过t1,则进行一次世代1的垃圾回收
  - *t2:*自从上一次世代2检查后，世代1被检查的次数超过t2,则进行一次世代2的垃圾回收

gc4.py

如何在python中解决内存泄漏

- 什么情况下需要手动解决内存泄漏，而不用gc
- 如何找到循环引用

如何在python中解决内存泄漏

1.确定哪些对象发生了循环引用

- **gc.set_debug**(*flags*) :
  - *gc.DEBUG_COLLECTABLE*: 输出所有可达对象信息
  - *gc.DEBUG_UNCOLLECTABLE* : 输出所有不可达对象信息，如果该对象不能被释放，则放入garbage列表
  - *gc.DEBUG_SAVEALL*: 将所有不可达对象都放入garbage列表，而不进行释放
  - *gc.DEBUG_LEAK = DEBUG_COLLECTABLE | DEBUG_UNCOLLECTABLE | DEBUG_SAVEALL*

如何在python中解决内存泄漏

2.确定泄漏对象的引用关系

- pip3 install objgraph
- 安装graphviz,并将bin目录加入环境变量path中
- 使用objgraph.show_refs获取该对象引用了谁
- 使用objgraph.show_backrefs获取该对象被谁引用

gc5.py