Python内存管理机制及优化简析

Jun 20, 2015

准备工作

为了方便解释Python的内存管理机制, 本文使用了`gc`模块来辅助展示内存中的Python对象以及Python垃圾回收器的工作情况. 本文中具体使用到的接口包括:

```
gc.disable()  # 暂停自动垃圾回收.
gc.collect()  # 执行一次完整的垃圾回收, 返回垃圾回收所找到无法到达的对象的数量.
gc.set_threshold()  # 设置Python垃圾回收的阈值.
gc.set_debug()  # 设置垃圾回收的调试标记. 调试信息会被写入std.err.
```

完整的`gc`模块文档可以参看这里.

同时我们还使用了`objgraph`Python库, 本文中具体使用到的接口包括:

```
objgraph.count(typename)  # 对于给定类型typename, 返回Python垃圾回收器正在跟踪的对象个数.
```

`objgraph`可以通过命令`pip install objgraph`安装. 完整的文档可以参看这里.

Python内存管理机制

Python有两种共存的内存管理机制: *引用计数*和*垃圾回收*. 引用计数是一种非常高效的内存管理手段, 当一个Python对象被引 用时其引用计数增加1, 当其不再被一个变量引用时则计数减1. 当引用计数等于0时对象被删除.

```
import gc

import objgraph

gc.disable()


class A(object):
        pass

class B(object):
        pass

def test1():
        a = A()
        b = B()

test1()
print objgraph.count('A')
print objgraph.count('B')
```

上面程序的执行结果为:

```
Object count of A: 0
Object count of B: 0
```

在`test1`中, 我们分别创建了类`A`和类`B`的对象, 并用变量`a`, `b`引用起来. 当`test1`调用结束后`objgraph.count('A')`返回0, 意味着内存中`A`的对象数量 没有增长. 同理`B`的对象数量也没有增长. 注意我们通过`gc.disable()`关闭了 Python的垃圾回收, 因此`test1`中生产的对象是在函数调用结束引用计数为0时被自 动删除的.

引用计数的一个主要缺点是无法自动处理循环引用. 继续上面的代码:

```
def test2():
    a = A()
    b = B()
    a.child = b
    b.parent = a

test2()
print 'Object count of A:', objgraph.count('A')
print 'Object count of B:', objgraph.count('B')
gc.collect()
print 'Object count of A:', objgraph.count('A')
print 'Object count of B:', objgraph.count('B')
```

在上面的代码的执行结果为:

```
Object count of A: 1
Object count of B: 1
Object count of A: 0
Object count of B: 0
```

`test1` 相比 `test2` 的改变是将 `A` 和 `B` 的对象通过 `child` 和 `parent` 相互引用 起来. 这就形成了一个循环引用. 当 `test2` 调用结束后, 表面上我们不再引用两个对象, 但由于两个对象相互引用着对方, 因此引用计数不为0, 则不会被自动回收. 更糟糕的是由于现在没有任何变量引用他们, 我们无法再找到这两个变量并清除. Python使用垃圾回收机制来处理这样的情况. 执行 `gc.collect()` , Python垃圾 回收器回收了两个相互引用的对象, 之后 `A` 和 `B` 的对象数又变为0.

垃圾回收机制

本节将简单介绍Python的垃圾回收机制. Garbage Collection for Python 以及Python垃圾回收源码 中的注释进行了更详细的解释.

在Python中, 所有能够引用其他对象的对象都被称为容器(container). 因此只有容器之间才可能形成循环引用. Python的垃圾回收机制利用了这个特点来寻找需要被释放的对象. 为了记录下所有的容器对象, Python将每一个 容器都链到了一个双向链表中, 之所以使用双向链表是为了方便快速的在容器集合中插入和删除对象. 有了这个 维护了所有容器对象的双向链表以后, Python在垃圾回收时使用如下步骤来寻找需要释放的对象:

1. 对于每一个容器对象, 设置一个 `gc_refs` 值, 并将其初始化为该对象的引用计数值.
2. 对于每一个容器对象, 找到所有其引用的对象, 将被引用对象的 `gc_refs` 值减1.
3. 执行完步骤2以后所有 `gc_refs` 值还大于0的对象都被非容器对象引用着, 至少存在一个非循环引用. 因此 不能释放这些对象, 将他们放入另一个集合.
4. 在步骤3中不能被释放的对象, 如果他们引用着某个对象, 被引用的对象也是不能被释放的, 因此将这些对象也放入另一个集合中.
5. 此时还剩下的对象都是无法到达的对象. 现在可以释放这些对象了.

值得注意的是, 如果一个Python对象含有 `__del__` 这个方法, Python的垃圾回收机制即使发现该对象不可到达 也不会释放他. 原因是 `__del__` 这个方式是当一个Python对象引用计数为0即将被删除前调用用来做清理工作的. 由于垃圾回收找到的需要释放的对象中往往存在循环引用的情况, 对于循环引用的对象 `a` 和 `b` , 应该先调用哪 一个对象的 `__del__` 是无法决定的, 因此Python垃圾回收机制就放弃释放这些对象, 转而将这些对象保存起来, 通过 `gc.garbage` 这个变量访问. 程序员可以通过 `gc.garbage` 手动释放对象, 但是更好的方法是避免在代码中 定义 `__del__` 这个方法.

除此之外, Python还将所有对象根据'生存时间'分为3代, 从0到2. 所有新创建的对象都分配为第0代. 当这些对象 经过一次垃圾回收仍然存在则会被放入第1代中. 如果第1代中的对象在一次垃圾回收之后仍然存货则被放入第2代. 对于不同代的对象Python的回收的频率也不一样. 可以通过 `gc.set_threshold(threshold0[, threshold1[, threshold2]])` 来定义. 当Python的垃圾回收器中新增的对象数量减去删除的对象数量大于threshold0时, Python会对第0代对象 执行一次垃圾回收. 每当第0代被检查的次数超过了threshold1时, 第1代对象就会被执行一次垃圾回收. 同理每当 第1代被检查的次数超过了threshold2时, 第2代对象也会被执行一次垃圾回收.

由于Python的垃圾回收需要检查所有的容器对象, 因此当一个Python程序生产了大量的对象时, 执行一次垃圾回收将 带来可观的开销. 因此可以通过一些手段来尽量避免垃圾回收以提高程序的效率.

调优手段

手动垃圾回收

对Python的垃圾回收进行调优的一个最简单的手段便是关闭自动回收, 根据情况手动触发. 例如在用Python开发游戏时, 可以在一局游戏的开始关闭GC, 然后在该局游戏结束后手动调用一次GC清理内存. 这样能完全避免在游戏过程中因此 GC造成卡顿. 但是缺点是在游戏过程中可能因为内存溢出导致游戏崩溃.

调高垃圾回收阈值

相比完全手动的垃圾回收, 一个更温和的方法是调高垃圾回收的阈值. 例如一个游戏可能在某个时刻产生大量的子弹对象(假如是2000个). 而此时Python的垃圾回收的threshold0为1000. 则一次垃圾回收会被触发, 但这2000个子弹对象并不需要被回收. 如果此时 Python的垃圾回收的threshold0为10000, 则不会触发垃圾回收. 若干秒后, 这些子弹命中目标被删除, 内存被引用计数机制 自动释放, 一次(可能很耗时的)垃圾回收被完全的避免了.

调高阈值的方法能在一定程度上避免内存溢出的问题(但不能完全避免), 同时可能减少可观的垃圾回收开销. 根据具体项目 的不同, 甚至是程序输入的不同, 合适的阈值也不同. 因此需要反复测试找到一个合适的阈值, 这也算调高阈值这种手段 的一个缺点.

避免循环引用

一个可能更好的方法是使用良好的编程习惯尽可能的避免循环引用. 两种常见的手段包括: 手动解循环引用和使用弱引用.

手动解循环引用

手动解循环引用指在编写代码时写好解开循环引用的代码, 在一个对象使用结束不再需要时调用. 例如:

```
class A(object):
    def __init__(self):
        self.child = None

    def destroy(self):
        self.child = None

class B(object):
    def __init__(self):
        self.parent = None

    def destroy(self):
        self.parent = None

def test3():
    a = A()
    b = B()
    a.child = b
    b.parent = a
    a.destroy()
    b.destroy()

test3()
print 'Object count of A:', objgraph.count('A')
print 'Object count of B:', objgraph.count('B')
```

上面代码的运行结果为:

```
Object count of A: 0
Object count of B: 0
```

使用弱引用

弱引用指当引用一个对象时, 不增加该对象的引用计数, 当需要使用到该对象的时候需要首先检查该对象是否还存在. 弱引用的实现方式有多种, Python自带一个弱引用库`weakref`, 其详细文档参加这里. 使用`weakref`改写我们的代码:

```
def test4():
    a = A()
    b = B()
    a.child = weakref.ref(b)
    b.parent = weakref.ref(a)
```

```
test4()
print 'Object count of A:', objgraph.count('A')
print 'Object count of B:', objgraph.count('B')
```

上面代码的运行结果为:

```
Object count of A: 0
Object count of B: 0
```

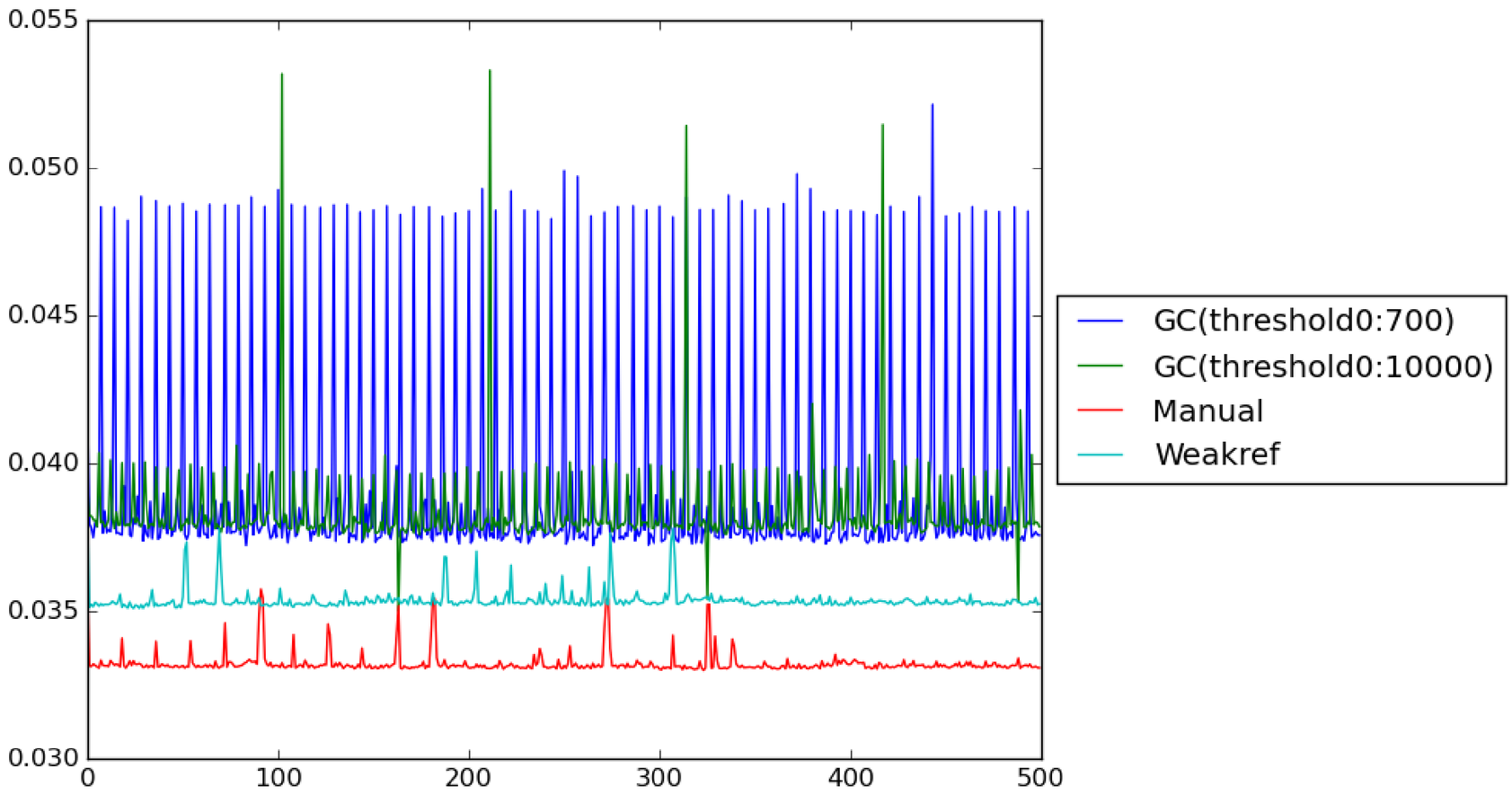
除了使用Python自带的`weakref`库以外, 通常我们也可以根据自己项目的业务逻辑实现弱引用. 例如在游戏开发中, 通常很多对象都是有 其唯一的ID的. 在引用一个对象时我们可以保存其ID而不是直接引用该对象. 在需要使用该对象的时候首先根据ID去检查该对象是否存在.

效率测试

为了测试各种调优手段对Python运行效率的实际影响, 本文使用了如下代码进行效率测试:

```
class Dungeon(object):
        def __init__(self):
                self._monster_list = []

        def add_monster(self, monster):
                self._monster_list.append(monster)
                monster.on_add_to_dungeon(self)

        def destroy(self):
                self._monster_list = []

class Monster(object):
        ATTRIBUTE_NAME_LIST = None
        def __init__(self, attribute_count):
                self._dungeon = None
                for i in xrange(attribute_count):
                        self.__dict__[Monster.ATTRIBUTE_NAME_LIST[i]] = i

        def on_add_to_dungeon(self, dungeon):
                self._dungeon = dungeon


class MonsterWithWeakref(Monster):
        def on_add_to_dungeon(self, dungeon):
                self._dungeon = weakref.ref(dungeon)


def run_logic(
                max_iteration, dungeon_per_iter, monster_per_dungeon, attribute_count,
                unloop_model):
        iter_timer_list = []
        for i in xrange(max_iteration):
                begin_at = timeit.default_timer()
                for dc in xrange(dungeon_per_iter):
                        dungeon = Dungeon()
                        for mc in xrange(monster_per_dungeon):
                                if unloop_model == 2:
                                        monster = MonsterWithWeakref(attribute_count)
                                else:
                                        monster = Monster(attribute_count)
                                dungeon.add_monster(monster)
                        if unloop_model == 1:
                                dungeon.destroy()
                end_at = timeit.default_timer()
                iter_timer_list.append(end_at-begin_at)
        return iter_timer_list
```

在上面的测试代码中, 我们模拟一个*非常*简单的游戏场景, 在每一帧(在上面的测试代码中为一个iteration)会创建若干个副本(Dungeon)对象, 对每一个副本对象创建并加入若干个怪物(Monster)对象. 当一个怪物加入 副本时便会形成一个循环引用. 当每一帧结束时, 新创建的副本和怪物都不再使用, 根据调优的方式不同分别 进行不同的处理: 1. 不进行任何处理; 2. 通过手动解环的方式解除循环引用; 3. 在创建怪物时使用`weakref`引用副本. 然后测试会记录下每一帧的运行时间然后返回.



上图是对每一种调优方法进行500帧测试的结果. 其中`GC(threshold0:700)`指只使用Python的GC手段, 700为Python默认的`threshold0`值. `GC(threshold0:10000)`为将`threshold0`的值调高到`10000`. `Manual`为手动解除循环引用. `Weakref`为使用`weakref`避免循环引用. 从图中可以看到`GC(threshold0:700)`的平均每帧耗时最高, 且每隔一段时间会出现一次较高的费时, 原因是此时GC在 工作. 而`GC(threshold:10000)`的平均每帧耗时则更低, 且出现因GC造成的高费时的次数也更少, 然而 由于调高`threshold0`值以后每次需要回收的对象数量大大增加, 因此GC耗时的峰值是最高的. 使用`weakref` 的每帧耗时则低很多且平稳度更高. 而表现最为出色的则是手动解除循环引用.

所以在使用Python时, 一种好的习惯是将Python的垃圾回收作为一种保护机制, 用来回收编码中泄露的循环 引用对象, 而在实际编程中则尽量解开所有的循环引用以节省大量的GC开销.

查找循环引用

从上一节的测试中可以看到如果能在编码时解开所有的循环引用对于程序运行的效率会有不小的提升, 尤其 是对那些需要长时间运行的, 时间敏感的Python程序(例如Web服务器). 但是在实际项目中很难保证所有的循环引用都被解开. 因此常常需要先查找运行的程序中存在哪些循环 引用然后再解开. Python的`gc`模块提供了`gc.set_debug`接口来设置一些辅助的调试信息. 如果我们 调用`gc.set_debug(gc.DEBUG_COLLECTABLE | gc.DEBUG_OBJECTS)`则每当Python进行垃圾回收时都会将 其发现的无法到达需要回收的对象的信息打印出来. 例如:

```
gc.collect()
gc.set_debug(gc.DEBUG_COLLECTABLE | gc.DEBUG_OBJECTS)
test2()
gc.collect()
```

上面的程序会输出:

```
gc: collectable <T 0x7fb6cacf6090>
gc: collectable <T 0x7fb6cace0fd0>
gc: collectable <dict 0x7fb6cac974b0>
gc: collectable <dict 0x7fb6cac97280>
```

而`gc.set_debug(gc.DEBUG_UNCOLLECTABLE | gc.DEBUG_OBJECTS)`则会输出所有无法到达且 垃圾回收器无法回收的对象, 例如:

```
class T2(object):
        def __del__(self):
                pass

def test_uncollectable():
        a = T2()
        b = T2()
        a.child = b
        b.parent = a

gc.collect()
gc.set_debug(gc.DEBUG_UNCOLLECTABLE | gc.DEBUG_OBJECTS)
test_uncollectable()
gc.collect()
```

上面的程序会输出:

```
gc: uncollectable <T2 0x7fb91fd2a110>
gc: uncollectable <T2 0x7fb91fd2a150>
gc: uncollectable <dict 0x7fb91fccf4b0>
gc: uncollectable <dict 0x7fb91fccf168>
```

设置`gc.set_debug(gc.DEBUG_COLLECTABLE | gc.DEBUG_UNCOLLECTABLE | gc.DEBUG_OBJECTS)`则能同时捕获两种情况. 当在测试环境运行程序时, 通过上面的调用打开debug信息, 将 debug信息保存到log文件中, 用`grep`等工具找到所有循环引用对象中那些属于我们自己编写的. 然后在代码中解除循环引用即可.

Kai Zhang

Kai Zhang
kylerzhang11@gmail.com

Kai Zhang's Blog.