《源码探秘 CPython》95. Python 的分代收集技术

原创 古明地觉 古明地觉的编程教室 2022-05-20 08:30 发表于北京

在这里先给已经找到人生另一半的大家送上一句祝福:520 节日快乐，愿每一个人都能和自己的伴侣幸福美满。
如果还没有找到另一半，那么在这个特殊的日子里，也请多给自己一些温柔。顺便告诉自己："我是最棒的"。

楔子

通过上一篇文章我们知道，Python 主要的内存管理手段是引用计数，而标记-清除和分代收集只是为了打破循环引用而引入的补充技术。

这一事实意味着垃圾回收只关注可能会产生循环引用的对象，而像整数、字符串这些对象是绝对不可能产生循环引用的，因为它们内部不可能持有对其他对象的引用，所以这些直接通过引用计数机制就可以实现，另外后面我们说的垃圾回收也专指那些可能产生循环引用的对象。

而循环引用只会发生在 container 对象之间，所谓 container 对象就是指内部可持有对其它对象的引用的对象，比如字典、列表、元组、自定义类对象、自定义类对象的实例对象等等。

所以当垃圾回收机制开始运行时，只需要检查这些 container 对象即可，对于整数、字符串、浮点数等对象则不需要理会，这使得垃圾回收带来的开销只依赖于 container 对象的数量，而非所有对象的数量。

为了达到这一点，Python 就必须跟踪所创建的每一个 container 对象，并将这些对象组织到一个集合中，只有这样，才能将垃圾回收的动作限制在这些对象上。而 Python 的做法是维护一条双向链表（实际上 3 条），我们称之为可收集对象链表，所有的 container 对象在创建之后，都会被插入到这条链表当中。

当然，除了维护用于 container 对象的链表之外，还维护一个名为 refchain 的链表，这个链表也是双向的。程序中产生的所有对象都会挂到这个链表上，注意是所有对象。然后当对象要被回收时，就将它从 refchain 里面摘除，比较简单。

所以任何一个对象在创建之后都会加入到 refchain 里面，而 container 对象由于要参与垃圾回收，所以它还必须加入到可收集对象链表里面。

可收集对象链表

在分析Python对象机制的时候我们看到，任何一个Python对象都可以分为两部分，一部分是PyObject_HEAD，另一部分是对象自身的数据。然而对于一个需要被垃圾回收机制跟踪的 container 对象来说还不够，因为这个对象还必须链入到 Python 内部的可收集对象链表中。而一个 container 对象要想成为一个可收集的对象，则必须加入额外的信息，这个信息位于 PyObject_HEAD 之前，称为 PyGC_Head。

```
//Include/objimpl.h

typedef union _gc_head {
    struct {
        //链表后继指针，指向后一个被跟踪的对象
        union _gc_head *gc_next;
        //链表前继指针，指向前一个被跟踪的对象
        union _gc_head *gc_prev;
        //对象引用计数副本，在标记清除算法中使用
        Py_ssize_t gc_refs;
    } gc;
```

```
    //内存对齐用，确保 _gc_head 结构体大小至少是 16 字节
    long double dummy;
} PyGC_Head;
```

所以，对于Python创建的可收集 container 对象，其内存布局与我们之前所了解的内存布局是不同的，我们可以从可收集 container 对象的创建过程中进行窥探。

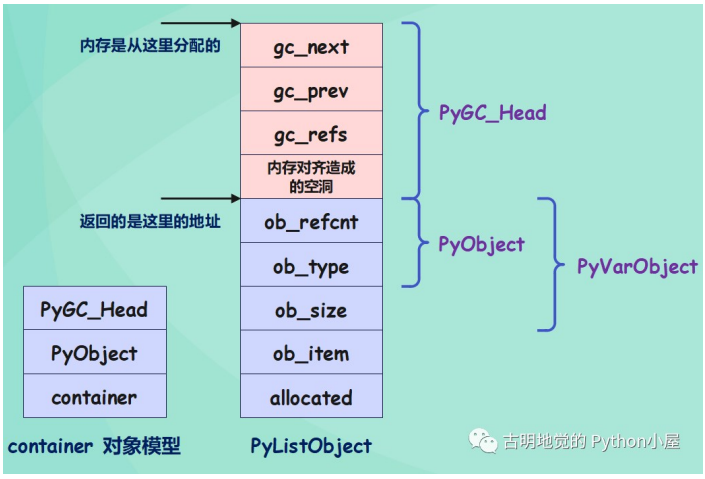
```
//Modules/gcmodule.c
PyObject *
_PyObject_GC_New(PyTypeObject *tp)
{
    //对于container对象
    //会调用_PyObject_GC_New申请内存，之前见过的
    PyObject *op = _PyObject_GC_Malloc(_PyObject_SIZE(tp));
    if (op != NULL)
        op = PyObject_INIT(op, tp);
    return op;
}

PyObject *
_PyObject_GC_Malloc(size_t basicsize)
{
    return _PyObject_GC_Alloc(0, basicsize);
}

static PyObject *
_PyObject_GC_Alloc(int use_calloc, size_t basicsize)
{
    struct _gc_runtime_state *state = &_PyRuntime.gc;
    PyObject *op;
    PyGC_Head *g;
    size_t size;
    if (basicsize > PY_SSIZE_T_MAX - sizeof(PyGC_Head))
        return PyErr_NoMemory();
    //将对象和PyGC_Head所需内存加起来
    size = sizeof(PyGC_Head) + basicsize;
    //为对象本身和PyGC_Head申请内存
    if (use_calloc)
        g = (PyGC_Head *)PyObject_Calloc(1, size);
    else
        g = (PyGC_Head *)PyObject_Malloc(size);
    if (g == NULL)
        return PyErr_NoMemory();
    //......
    //根据 PyGC_Head 的地址得到 PyObject 的地址
    op = FROM_GC(g);
    return op;
}
```

因此可以很清晰地看到，当Python为可收集的 container 对象申请内存空间时，还额外地为PyGC_Head申请了空间，并且位置位于 container 对象之前。但是返回的时候又调用了FROM_GC，也就是说返回的仍是 container 对象的地址。

所以像列表、字典等 container 对象的内存分布就应该变成这样。



在可收集 container 对象的内存分布中，分为三个部分，首先第一块用于垃圾回收机制，然后紧跟着的是 Python 所有对象都会有的 PyObject，最后才是 container 自身的数据。这里的container对象，既可以是 PyDictObject、也可以是 PyListObject 等等。

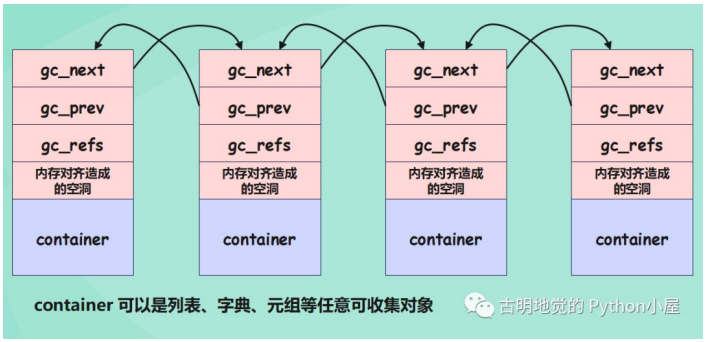
根据PyGC_Head，我们知道里面除了两个建立链表结构的前继指针和后继指针外，还有一个gc_ref，这个成员对垃圾回收的运行直观重要，我们后面会说。

另外当垃圾回收机制运行期间，我们需要在一个可收集 container 对象的PyGC_Head部分和PyObject部分之间来回切换。更清楚的说，某些时候，我们持有一个对象 A 的PyObject的地址

（或者说 A 的地址），但是我们需要根据这个地址来获得PyGC_Head的地址；而且某些时候，我们又需要反过来进行逆运算。所以 Python 提供了两个地址之间的转换算法：

```
//Modules/gcmodule.c
//根据PyObject得到PyGC_Head
#define AS_GC(o) ((PyGC_Head *)(o)-1)
//根据PyGC_Head得到PyObject
#define FROM_GC(g) ((PyObject *)(((PyGC_Head *)g)+1))
```

在PyGC_Head中，出现了用于建立链表的两个指针，只有将创建的可收集container对象链接到Python内部维护的可收集对象链表中，Python的垃圾回收机制才能跟踪和处理这个container对象。



但是我们发现，在创建可收集 container 对象之时，并没有立刻将这个对象链入到链表中。实际上，这个动作是发生在创建某个 container 对象的最后一步，以 PyListObject 的创建为例。

```
//listobject.c
PyObject *
PyList_New(Py_ssize_t size)
{
    PyListObject *op;
    //...
    Py_SIZE(op) = size;
    op->allocated = size;
    //创建PyListObject对象
    //并在设置完属性之后、返回之前执行了_PyObject_GC_TRACK
    _PyObject_GC_TRACK(op);
    return (PyObject *) op;
}
```

这个_PyObject_GC_TRACK我们见过很多回了，之前的解释是开启 GC 跟踪，但是现在我们明白了，这一步本质上就是将所创建的container对象链接到了Python的可收集对象链表中。

那么这一步是怎么实现的呢？

```
//Include/internal/pycore_object.h
#define _PyObject_GC_TRACK(op) \
    _PyObject_GC_TRACK_impl(__FILE__, __LINE__, _PyObject_CAST(op))

static inline void _PyObject_GC_UNTRACK_impl(const char *filename, int l
ineno,
                                             PyObject *op)
{
    _PyObject_ASSERT_FROM(op, _PyObject_GC_IS_TRACKED(op),
                          "object not tracked by the garbage collector",
                          filename, lineno, "_PyObject_GC_UNTRACK");

    PyGC_Head *gc = _Py_AS_GC(op);
    PyGC_Head *prev = _PyGCHead_PREV(gc);
    PyGC_Head *next = _PyGCHead_NEXT(gc);
    _PyGCHead_SET_NEXT(prev, next);
    _PyGCHead_SET_PREV(next, prev);
    gc->_gc_next = 0;
    gc->_gc_prev &= _PyGC_PREV_MASK_FINALIZED;
}
```

前面我们说过，Python 会将自己的垃圾回收机制限制在其维护的可收集对象链表上，因为所有的循环引用一定是在这个链表的对象里面发生的。而在_PyObject_GC_TRACK之后，我们创建的container对象也就置身于Python垃圾回收机制的掌控当中了，也就是我们之前所说的开启 GC 跟踪。

同样的，Python还提供将一个container对象从链表中摘除的方法，显然这个方法应该会在对象被销毁的时候调用。

```
//Include/internal/pycore_object.h

#define _PyObject_GC_UNTRACK(op) \
    _PyObject_GC_UNTRACK_impl(__FILE__, __LINE__, _PyObject_CAST(op))

static inline void _PyObject_GC_UNTRACK_impl(const char *filename, int l
```

```
ineno,
                                        PyObject *op)
{
    _PyObject_ASSERT_FROM(op, _PyObject_GC_IS_TRACKED(op),
                          "object not tracked by the garbage collector",
                          filename, lineno, "_PyObject_GC_UNTRACK");

    PyGC_Head *gc = _Py_AS_GC(op);
    PyGC_Head *prev = _PyGCHead_PREV(gc);
    PyGC_Head *next = _PyGCHead_NEXT(gc);
    _PyGCHead_SET_NEXT(prev, next);
    _PyGCHead_SET_PREV(next, prev);
    gc->_gc_next = 0;
    gc->_gc_prev &= _PyGC_PREV_MASK_FINALIZED;
}
```

很明显，_PyObject_GC_UNTRACK只是_PyObject_GC_TRACK的逆运算而已。就这样，借助 gc_next 和 gc_prev 指针，Python 将需要跟踪的对象一个接一个地组织成双向链表。

分代收集技术

无论什么语言，写出来的程序都有共同之处，那就是不同对象的生命周期会存在不同。有的对象所占的内存块的生命周期很短，而有的内存块的生命周期则很长，甚至可能从程序的开始持续到程序结束，这两者的比例大概在80~90%。

这对于垃圾回收机制有着重要的意义，因为我们已经知道，像标记-清除这样的算法所带来的额外开销实际上是和系统中内存块的数量相关，当需要回收的内存块越多的时候，垃圾检测带来的额外开销就越多，相反则越少。

因此我们可以采用一种空间换时间的策略，因为目前所有对象（container 对象）都在一个链子上，每当进行垃圾回收的时候，都要把所有对象全部检查一遍。而其实有不少比较稳定的对象(在多次垃圾回收的洗礼下能活下来)，我们完全没有必要每次都检查，或者说检查的频率可以降低一些。

于是聪明如你已经猜到了，我们再来一条链子不就可以了，把那些认为比较稳定的对象移到另外一条链子上，而新的链子进行垃圾回收的频率会低一些。

所以这种思想就是：将系统中的所有内存块根据其存活时间划分为不同的集合，每一个集合就称为一个"代"，垃圾回收的频率随着"代"的存活时间的增大而减小。也就是说，存活的时间越长的对象就越可能不是垃圾，就越可能是程序中需要一直存在的对象，就应该少去检测它。反正不是垃圾，检测也是白检测。

那么关键的问题来了，这个存活时间是如何被衡量的呢？或者说当对象比较稳定的时候的这个稳定是如何衡量的呢？没错，我们上面已经暴露了，就是通过经历了几次垃圾回收动作来评判，如果一个对象经历的垃圾回收次数越多，那么显然其存活时间就越长。

而Python的垃圾回收器，每当条件满足时(至于什么条件我们后面会说)，就会进行一次垃圾回收(注意：不同的代的垃圾回收频率是不同的)，而每次扫黄的时候你都不在，吭，每次垃圾回收的时候你都能活下来，这就说明你存活的时间更长，或者像我们上面说的更稳定，那么就不应该再把你放在这条链子上了，而是会移动到新的链子上。而在新的链子上，进行垃圾回收的频率会降低，因为既然稳定了，检测就不必那么频繁了，或者说新的链子上触发垃圾回收所需要的时间更长了。

对象存活时间	释放概率	回收频率
长	低	低
短	高	高

古明地觉的 Python小屋

"代"似乎是一个比较抽象的概念，但在Python中，你就把"代"想象成多个对象组成的集合，或者你把"代"想象成链表也可以。因为这些对象都串在链表上面，并且属于同一"代"的内存块都被链接在同一个链表中。

而在Python中总共存在三条链表，说明Python中所有的对象总共可以分为三代：零代、一代、二代，一个"代"就是一条我们上面提到的可收集对象链表。而在前面所介绍的链表的基础之上，为了支持分代机制，我们需要的仅仅是一个额外的表头而已。

```
//Include/internal/mem.h

#define NUM_GENERATIONS 3
struct gc_generation {
    PyGC_Head head;
    //"代"不同, 这两个字段的含义也不同
    //一会再聊
```

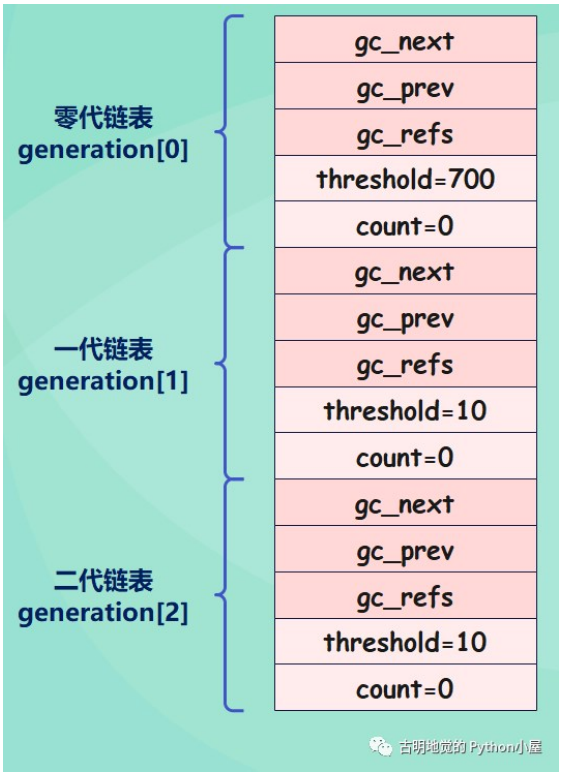
```
    int threshold;
    int count;
};

//Modules/gcmodule.c
#define _GEN_HEAD(n) GEN_HEAD(state, n)
    struct gc_generation generations[NUM_GENERATIONS] = {
        /* PyGC_Head,                                    threshold,    c
ount */
        {{(uintptr_t)_GEN_HEAD(0), (uintptr_t)_GEN_HEAD(0)},   700,
   0},
        {{(uintptr_t)_GEN_HEAD(1), (uintptr_t)_GEN_HEAD(1)},   10,
   0},
        {{(uintptr_t)_GEN_HEAD(2), (uintptr_t)_GEN_HEAD(2)},   10,
   0},
    };
    for (int i = 0; i < NUM_GENERATIONS; i++) {
        state->generations[i] = generations[i];
    };
    state->generation0 = GEN_HEAD(state, 0);
    struct gc_generation permanent_generation = {
          {(uintptr_t)&state->permanent_generation.head,
           (uintptr_t)&state->permanent_generation.head}, 0, 0
    };
    state->permanent_generation = permanent_generation;
}
```

每一个"代"就是一个 gc_generation结构体实例，而维护了三个 gc_generation 结构体实例的数组，控制了三条可收集对象链表，这就是 Python 用于分代垃圾收集的三个"代"。

对于每一个gc_generation，其中的count记录了当前这条可收集对象链表中一共有多少个container对象。而在_PyObject_GC_Alloc中我们可以看到每当分配了内存，就会进行_PyRuntime.gc.generations[0].count++动作，将第 0 代链表中所维护的对象数量加 1。这预示着所有新创建的container对象实际上都会被加入到 0 代链表当中，而这一点也确实如此，已经被_PyObject_GC_TRACK证明了。

而当三个"代"初始化完毕之后，对应的 gc_generation 数组大概是这样的。



我们看到，"代"不同，那么对应的threshold也不同。对于零代链表而言，threshold字段表示该条可收集对象链表中最多可以容纳多少个新创建的 container 对象，从源码我们得到是 700 个。而一旦零代链表中新创建的 container 对象超过了 700 个，那么会立刻触发垃圾回收机制。

```
//Modules/gcmodule.c
static Py_ssize_t
collect_generations(struct _gc_runtime_state *state)
{
    Py_ssize_t n = 0;
    for (int i = NUM_GENERATIONS-1; i >= 0; i--) {
        //当count大于threshold的时候
        if (state->generations[i].count > state->generations[i].threshol
d) {
            if (i == NUM_GENERATIONS - 1
```

```
                && state->long_lived_pending < state->long_lived_total / 4
)
                continue;
            //执行此函数对链表进行清理
            n = collect_with_callback(state, i);
            break;
        }
    }
    return n;
}
```

当调用_PyObject_GC_Alloc为container对象分配内存时，零代链表的 count 字段自增 1，并将该对象接入零代链表；当调用PyObject_GC_Del释放container对象的内存时，零代链表的 count 字段自减 1。

当 count 大于 threshold 时，也就是新创建的container对象（准确的说，还要减去已回收的container对象）超过了 700 个，那么清理一次零代链表。

清理完零代链表之后，将 count 重置为 0，然后继续重新统计新创建的 container 对象的个数。如果它减去已回收的 container 对象的个数又超过了 threshold（默认 700），那么继续清理零代链表。

然后零代链表每清理一次，一代链表的 count 字段自增 1；一代链表每清理一次，二代链表的 count 字段自增 1。所以：

- 零代链表的 count 字段维护的是新创建的 container 对象的数量减去已回收的 container 对象的数量。如果 count 超过了 700，那么清理零代链表，并将自身的 count 字段清零，一代链表的 count 字段自增 1；
- 一代链表的 count 字段维护的是零代链表的清理次数，它的 threshold 是 10。所以当零代链表的清理次数达到 11 次时，清理一次一代链表，注意：清理一代链表的同时还会顺带清理零代链表。然后将自身的 count 字段和零代链表的 count 字段清零，二代链表的 count 字段自增 1；
- 二代链表的 count 字段维护的是一代链表的清理次数，它的 threshold 是 10。所以当一代链表的清理次数达到 11 次时，清理一次二代链表，注意：清理二代链表的同时还会清理一代链表和零代链表。然后将自身的 count 字段和一代链表、零代链表的 count 字段都清零，注意：没有三代链表；

而上面用于清理链表的collect_generations函数，内部是一个 for 循环，所以无论是清理哪一代链表，都是执行的这个函数。当然，真正用来清理的其实是内部调用的collect_with_callback函数，清理零代链表，里面的参数 i 指的就是 0；清理一代链表，里面的参数 i 指的就是 1；清理二代链表，里面的参数 i 指的就是 2。

所以 collect_generations 里面的 for 循环是从二代链表开始遍历的，该函数内部调用了 collect_with_callback 函数，实际上 collect_with_callback 内部又会调用 collect 函数。

因为清理一个"代"时，会将该"代"以及前面的"代"一块清理，所以在 collect 函数内部，会将该"代"的后一个"代"（如果有的话）的 count 字段自增 1，并将该"代"以及前面的"代"（如果有的话）的 count 字段清零。然后调用 gc_list_merge 函数将该"代"前面的所有"代"都合并到该"代"，执行标记-清除算法，一起清理。

清理完毕之后，就可以区分出可达和不可达的对象。可达的对象会被移入到下一"代"，因为它们是稳定的，所以检测频率应该降低；而不可达的对象，显然会被引用计数机制干掉。

所以以上就分代收集的秘密，总结一下就是：

- 零代链表中新创建的 container 对象减去已回收的 container 对象的个数达到了 701，触发一次零代链表的清理；
- 零代链表的清理次数达到 11 次，触发一次一代链表的清理；
- 一代链表的清理次数达到 11 次，触发一次二代链表的清理；
- 清理某个"代"时，会将比它年轻的"代"都合并到该"代"，然后一起清理；
- 可达的对象移入更老的代，不可达的对象由于循环引用的影响被消除，那么会被引用计数机制干掉。

再透过源码验证一下我们的结论：

```
//Modules/gcmodule.c
static Py_ssize_t
collect(struct _gc_runtime_state *state, int generation,
        Py_ssize_t *n_collected, Py_ssize_t *n_uncollectable, int nofail)
{

    int i;
    //可达的对象个数
    Py_ssize_t m = 0;
    //不可达的对象个数
    Py_ssize_t n = 0;
    //正在检测的代
    PyGC_Head *young;
    //下一个代
    PyGC_Head *old;

```

```
    //......
    //因为没有三代链表，所以当清理的是零代或一代链表时
    //将下一代链表的 count 字段自增 1
    if (generation+1 < NUM_GENERATIONS)
        state->generations[generation+1].count += 1;
    //同时将它前面的代的 count 字段清零
    for (i = 0; i <= generation; i++)
        state->generations[i].count = 0;

    //将前面的代都合并到该代
    for (i = 0; i < generation; i++) {
        gc_list_merge(GEN_HEAD(state, i), GEN_HEAD(state, generation));
    }

    //......
    //将可达的对象移入下一个代
    if (young != old) {
        if (generation == NUM_GENERATIONS - 2) {
            state->long_lived_pending += gc_list_size(young);
        }
        gc_list_merge(young, old);
    }
    else {
        untrack_dicts(young);
        state->long_lived_pending = 0;
        state->long_lived_total = gc_list_size(young);
    }

    //所有不可达的对象都是垃圾，要被清理
    gc_list_init(&finalizers);
    move_legacy_finalizers(&unreachable, &finalizers);
    move_legacy_finalizer_reachable(&finalizers);

    //......
    assert(!PyErr_Occurred());
    //返回可达对象个数 + 不可达对象的个数
    return n+m;
}
```

以上就是分代收集技术。

到目前为止，我们算是理解 Python 的垃圾回收到底是怎么一回事了。总结一下就是引用计数机制为主，标记-清除和分代收集为辅，后者主要是为了弥补前者的致命缺点而存在的。因为单有引用计数不够的，严格意义上讲它也算不上垃圾回收，所以还需要依赖标记-清除和分代收集为其兜底（这两者才是真正的垃圾回收机制）。

那么下一篇，我们就要深入源码，来剖析垃圾回收是怎么实现的，下一篇文章也是本系列的最后一篇文章了。

收录于合集 #CPython 97

‹ 上一篇
《源码探秘 CPython》96. 深入源码，探究垃圾回收的秘密

下一篇 ›
《源码探秘 CPython》94. Python 的引用计数与标记-清除