Sep 29, 2015

Python 源码阅读 - 垃圾回收机制

概述

无论何种垃圾收集机制，一般都是两阶段：垃圾检测和垃圾回收.

在Python中，大多数对象的生命周期都是通过对象的引用计数来管理的.

问题：但是存在循环引用的问题：a 引用 b，b 引用 a，导致每一个对象的引用计数都不为0，所占用的内存永远不会被回收

要解决循环引用：必需引入其他垃圾收集技术来打破循环引用. Python中使用了 标记-清除 以及 分代收集

即，Python 中垃圾回收机制：引用计数(主要)，标记清除，分代收集(辅助)

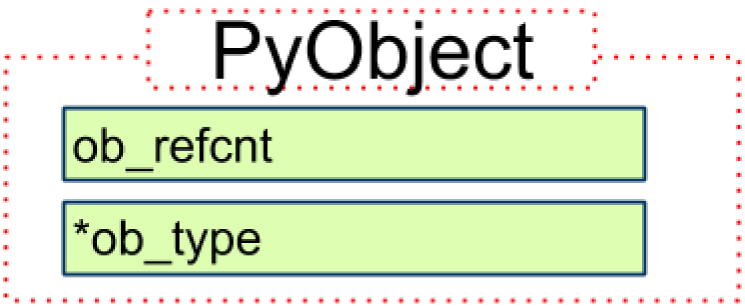
引用计数

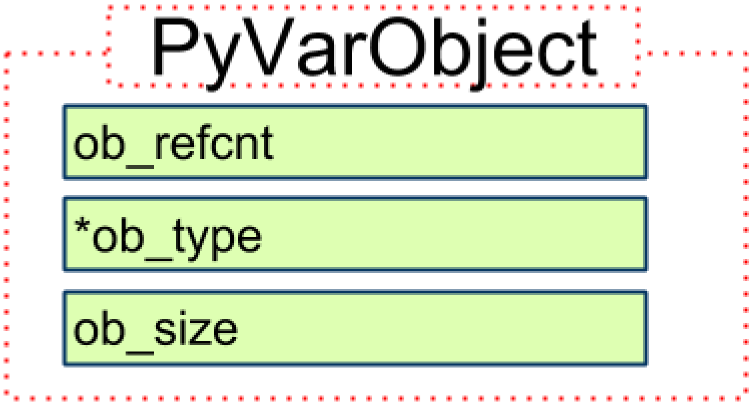
引用计数，意味着必须在每次分配和释放内存的时候，加入管理引用计数的动作

引用计数的优点：最直观最简单，实时性，任何内存，一旦没有指向它的引用，就会立即被回收

计数存储

回顾 Python 的对象





e.g. 引用计数增加以及减少

```
>>> from sys import getrefcount
>>>
>>> a = [1, 2, 3]
>>> getrefcount(a)
2
>>> b = a
>>> getrefcount(a)
3
>>> del b
>>> getrefcount(a)
2
```

计数增加

增加对象引用计数, refcnt incr

```
#define Py_INCREF(op) (                         \
        _Py_INC_REFTOTAL  _Py_REF_DEBUG_COMMA        \
        ((PyObject*)(op))->ob_refcnt++)
```

计数减少

减少对象引用计数, refcnt desc

```
#define _Py_DEC_REFTOTAL        _Py_RefTotal--
#define _Py_REF_DEBUG_COMMA     ,
```

```
        do {                                                  \
                if (_Py_DEC_REFTOTAL  _Py_REF_DEBUG_COMMA       \
                --((PyObject*)(op))->ob_refcnt != 0)            \
                        _Py_CHECK_REFCNT(op)                    \
                else                                            \
                _Py_Dealloc((PyObject *)(op));                  \
        } while (0)
```

即，发现refcnt变成0的时候，会调用 `_Py_Dealloc`

```
PyAPI_FUNC(void) _Py_Dealloc(PyObject *);
#define _Py_REF_DEBUG_COMMA     ,

#define _Py_Dealloc(op) (                               \
        _Py_INC_TPFREES(op) _Py_COUNT_ALLOCS_COMMA       \
        (*Py_TYPE(op)->tp_dealloc)((PyObject *)(op)))
#endif /* !Py_TRACE_REFS */
```

会调用各自类型的 `tp_dealloc`

例如dict

```
PyTypeObject PyDict_Type = {
    PyVarObject_HEAD_INIT(&PyType_Type, 0)
    "dict",
    sizeof(PyDictObject),
    0,
    (destructor)dict_dealloc,                   /* tp_dealloc */
    ....
}


static void
dict_dealloc(register PyDictObject *mp)
{
    .....
    // 如果满足条件，放入到缓冲池freelist中
    if (numfree < PyDict_MAXFREELIST && Py_TYPE(mp) == &PyDict_Type)
        free_list[numfree++] = mp;
    // 否则，调用tp_free
    else
        Py_TYPE(mp)->tp_free((PyObject *)mp);
    Py_TRASHCAN_SAFE_END(mp)
}
```

WKLKEN BUILDING

Python基本类型的 `tp_dealloc` ，通常都会与各自的缓冲池机制相关，释放会优先放入缓冲池中(对应的分配会优先从缓冲池取). 这个内存分配与回收同缓冲池机制相关

当无法放入缓冲池时，会调用各自类型的 `tp_free`

int，比较特殊

```
// int，通用整数对象缓冲池机制
      (freefunc)int_free,                          /* tp_free */
```

string

```
// string
    PyObject_Del,                                  /* tp_free */
```

dict/tuple/list

```
    PyObject_GC_Del,                               /* tp_free */
```

然后，我们再回头看，自定义对象的 `tp_free`

```
PyTypeObject PyType_Type = {
    PyVarObject_HEAD_INIT(&PyType_Type, 0)
    "type",                                        /* tp_name */
    ...
    PyObject_GC_Del,                               /* tp_free */
};
```

即，最终，当计数变为0，触发内存回收动作. 涉及函数 `PyObject_Del` 和 `PyObject_GC_Del` ，并且，自定义类以及容器类型(dict/list/tuple/set等)使用的都是后者 `PyObject_GC_Del` .

内存回收 PyObject_Del / PyObject_GC_Del

如果引用计数=0:

```
1. 放入缓冲池
2. 真正销毁，PyObject_Del/PyObject_GC_Del内存操作
```

这两个操作都是进行内存级别的操作

- PyObject_Del

> *PyObject_Del(op) releases the memory allocated for an object. It does not run a destructor – it only frees the memory. PyObject_Free is identical.*

这块删除， `PyObject_Free` 涉及到了Python底层内存的分配和管理机制，具体见前面的博文

- PyObject_GC_Del

```
void
PyObject_GC_Del(void *op)
{
        PyGC_Head *g = AS_GC(op);

        // Returns true if a given object is tracked
        if (IS_TRACKED(op))
                // 从跟踪链表中移除
                gc_list_remove(g);
        if (generations[0].count > 0) {
                generations[0].count--;
        }
        PyObject_FREE(g);
}
```

`IS_TRACKED` 涉及到标记-清除的机制

`generations` 涉及到了分代回收

`PyObject_FREE`，则和Python底层内存池机制相关

标记-清除

问题：什么对象可能产生循环引用?

只需要关注关注可能产生循环引用的对象

PyIntObject/PyStringObject等不可能

Python中的循环引用总是发生在container对象之间, 所谓containser对象即是内部可持有对其他对象的引用: list/dict/class/instance等等

垃圾收集带来的开销依赖于container对象的数量, 必需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.

可收集对象链表

可收集对象链表: 将需要被收集和跟踪的container, 放到可收集的链表中

任何一个python对象都分为两部分: PyObject_HEAD + 对象本身数据

```
/* PyObject_HEAD defines the initial segment of every PyObject. */
#define PyObject_HEAD                   \
    _PyObject_HEAD_EXTRA                \
    Py_ssize_t ob_refcnt;               \
    struct _typeobject *ob_type;

//------------------------------------------------------

  #define _PyObject_HEAD_EXTRA            \
      struct _object *_ob_next;           \
      struct _object *_ob_prev;

// 双向链表结构, 垃圾回收
```

可收集对象链表

```
Modules/gcmodule.c

/* GC information is stored BEFORE the object structure. */
typedef union _gc_head {
    struct {
        // 建立链表需要的前后指针
        union _gc_head *gc_next;
        union _gc_head *gc_prev;
        // 在初始化时会被初始化为 GC_UNTRACED
        Py_ssize_t gc_refs;
    } gc;
    long double dummy;  /* force worst-case alignment */
} PyGC_Head;
```

创建container的过程: `container对象 = pyGC_Head | PyObject_HEAD | Container Object`

```
PyObject *
_PyObject_GC_New(PyTypeObject *tp)
{
    PyObject *op = _PyObject_GC_Malloc(_PyObject_SIZE(tp));
    if (op != NULL)
        op = PyObject_INIT(op, tp);
    return op;
}

=> _PyObject_GC_Malloc

#define _PyGC_REFS_UNTRACKED                    (-2)
#define GC_UNTRACKED                    _PyGC_REFS_UNTRACKED

PyObject *
_PyObject_GC_Malloc(size_t basicsize)
{
    PyObject *op;
    PyGC_Head *g;
    if (basicsize > PY_SSIZE_T_MAX - sizeof(PyGC_Head))
        return PyErr_NoMemory();

    // 为 对象本身+PyGC_Head申请内存, 注意分配的size
    g = (PyGC_Head *)PyObject_MALLOC(
        sizeof(PyGC_Head) + basicsize);
    if (g == NULL)
        return PyErr_NoMemory();

    // 初始化 GC_UNTRACED
    g->gc.gc_refs = GC_UNTRACKED;
    generations[0].count++; /* number of allocated GC objects */

    // 如果大于阈值, 执行分代回收
    if (generations[0].count > generations[0].threshold &&
        enabled &&
        generations[0].threshold &&
        !collecting &&
        !PyErr_Occurred()) {

        collecting = 1;
        collect_generations();
        collecting = 0;
    }
```

```
    return op;
}
```

PyObject_HEAD and PyGC_HEAD

注意, FROM_GC 和 AS_GC 用于 PyObject_HEAD <=> PyGC_HEAD 地址相互转换

```
// => Modules/gcmodule.c

/* Get an object's GC head */
#define AS_GC(o) ((PyGC_Head *)(o)-1)

/* Get the object given the GC head */
#define FROM_GC(g) ((PyObject *)(((PyGC_Head *)g)+1))

// => objimpl.h

#define _Py_AS_GC(o) ((PyGC_Head *)(o)-1)
```

问题: 什么时候将container放到这个对象链表中

e.g list

```
// => listobject.c

PyObject *
PyList_New(Py_ssize_t size)
{
    PyListObject *op;
    op = PyObject_GC_New(PyListObject, &PyList_Type);
    _PyObject_GC_TRACK(op);
    return (PyObject *) op;
}

// =>  _PyObject_GC_TRACK

// objimpl.h
// 加入到可收集对象链表中

#define _PyObject_GC_TRACK(o) do { \
    PyGC_Head *g = _Py_AS_GC(o); \
    if (g->gc.gc_refs != _PyGC_REFS_UNTRACKED) \
        Py_FatalError("GC object already tracked"); \
    g->gc.gc_refs = _PyGC_REFS_REACHABLE; \
```

WKLKEN BUILDING

```
    g->gc.gc_prev->gc.gc_next = g; \
    _PyGC_generation0->gc.gc_prev = g; \
    } while (0);
```

问题: 什么时候将container从这个对象链表中摘除

```
// Objects/listobject.c

static void
list_dealloc(PyListObject *op)
{
    Py_ssize_t i;
    PyObject_GC_UnTrack(op);
    .....
}

// => PyObject_GC_UnTrack => _PyObject_GC_UNTRACK

// 对象销毁的时候
#define _PyObject_GC_UNTRACK(o) do { \
    PyGC_Head *g = _Py_AS_GC(o); \
    assert(g->gc.gc_refs != _PyGC_REFS_UNTRACKED); \
    g->gc.gc_refs = _PyGC_REFS_UNTRACKED; \
    g->gc.gc_prev->gc.gc_next = g->gc.gc_next; \
    g->gc.gc_next->gc.gc_prev = g->gc.gc_prev; \
    g->gc.gc_next = NULL; \
    } while (0);
```

问题: 如何进行标记-清除

现在, 我们得到了一个链表

Python将自己的垃圾收集限制在这个链表上, 循环引用一定发生在这个链表的一群独享之间.

0. 概览

`_PyObject_GC_Malloc` 分配内存时, 发现超过阈值, 此时, 会触发gc, `collect_generations` 然后调用 `collect` , `collect` 包含标记-清除逻辑

```
gcmodule.c
```

```c
static Py_ssize_t
collect(int generation)
{
  // 第1步: 将所有比 当前代 年轻的代中的对象 都放到 当前代 的对象链表中
  /* merge younger generations with one we are currently collecting */
  for (i = 0; i < generation; i++) {
      gc_list_merge(GEN_HEAD(i), GEN_HEAD(generation));
  }


  // 第2步
  update_refs(young);
  // 第3步
  subtract_refs(young);

  // 第4步
  gc_list_init(&unreachable);
  move_unreachable(young, &unreachable);

  // 第5步
    /* Move reachable objects to next generation. */
    if (young != old) {
        if (generation == NUM_GENERATIONS - 2) {
            long_lived_pending += gc_list_size(young);
        }
        gc_list_merge(young, old);
    }
    else {
        /* We only untrack dicts in full collections, to avoid quadratic
           dict build-up. See issue #14775. */
        untrack_dicts(young);
        long_lived_pending = 0;
        long_lived_total = gc_list_size(young);
    }

  // 第6步
    delete_garbage(&unreachable, old);

}
```

1. 第一步: gc_list_merge

将所有比 当前代 年轻的代中的对象 都放到 当前代 的对象链表中

```c
// => gc_list_merge
```

```
/* append list `from` onto list `to`; `from` becomes an empty list */
static void
gc_list_merge(PyGC_Head *from, PyGC_Head *to)
{
    PyGC_Head *tail;
    assert(from != to);
    if (!gc_list_is_empty(from)) {
        tail = to->gc.gc_prev;
        tail->gc.gc_next = from->gc.gc_next;
        tail->gc.gc_next->gc.gc_prev = tail;
        to->gc.gc_prev = from->gc.gc_prev;
        to->gc.gc_prev->gc.gc_next = to;
    }
    // 清空
    gc_list_init(from);
}

=>

static void
gc_list_init(PyGC_Head *list)
{
    list->gc.gc_prev = list;
    list->gc.gc_next = list;
}
```

即，此刻，所有待进行处理的对象都集中在同一个链表中

处理，

其逻辑是，要去除循环引用，得到有效引用计数

有效引用计数：将循环引用的计数去除，最终得到的 => 将环从引用中摘除，各自引用计数数值-1

实际操作，并不要直接修改对象的 ob_refcnt，而是修改其副本， `PyGC_Head` 中的 `gc.gc_ref`

2. 第二步：update_refs

遍历对象链表，将每个对象的gc.gc_ref值设置为ob_refcnt

```
static void
update_refs(PyGC_Head *containers)
{
    PyGC_Head *gc = containers->gc.gc_next;
    for (; gc != containers; gc = gc->gc.gc_next) {
        assert(gc->gc.gc_refs == GC_REACHABLE);
        gc->gc.gc_refs = Py_REFCNT(FROM_GC(gc));
        /* Python's cyclic gc should never see an incoming refcount
         * of 0:  if something decref'ed to 0, it should have been
         * deallocated immediately at that time.
         * Possible cause (if the assert triggers):  a tp_dealloc
         * routine left a gc-aware object tracked during its teardown
         * phase, and did something-- or allowed something to happen --
         * that called back into Python.  gc can trigger then, and may
         * see the still-tracked dying object.  Before this assert
         * was added, such mistakes went on to allow gc to try to
         * delete the object again.  In a debug build, that caused
         * a mysterious segfault, when _Py_ForgetReference tried
         * to remove the object from the doubly-linked list of all
         * objects a second time.  In a release build, an actual
         * double deallocation occurred, which leads to corruption
         * of the allocator's internal bookkeeping pointers.  That's
         * so serious that maybe this should be a release-build
         * check instead of an assert?
         */
        assert(gc->gc.gc_refs != 0);
    }
}
```

3. 第三步: 计算有效引用计数

```
/* A traversal callback for subtract_refs. */
static int
visit_decref(PyObject *op, void *data)
{
    assert(op != NULL);
    // 判断op指向的对象是否是被垃圾收集监控的, 对象的type对象中有Py_TPFLAGS_HAVE_GC符号
    if (PyObject_IS_GC(op)) {
        PyGC_Head *gc = AS_GC(op);
        /* We're only interested in gc_refs for objects in the
         * generation being collected, which can be recognized
         * because only they have positive gc_refs.
         */
        assert(gc->gc.gc_refs != 0); /* else refcount was too small */
        if (gc->gc.gc_refs > 0)
            gc->gc.gc_refs--;  // -1
```

```c
}


/* Subtract internal references from gc_refs.  After this, gc_refs is >= 0
 * for all objects in containers, and is GC_REACHABLE for all tracked gc
 * objects not in containers.  The ones with gc_refs > 0 are directly
 * reachable from outside containers, and so can't be collected.
 */
static void
subtract_refs(PyGC_Head *containers)
{
    traverseproc traverse;
    PyGC_Head *gc = containers->gc.gc_next;
    // 遍历链表
    for (; gc != containers; gc=gc->gc.gc_next) {
        // 与特定的类型相关, 得到类型对应的traverse函数
        traverse = Py_TYPE(FROM_GC(gc))->tp_traverse;
        // 调用
        (void) traverse(FROM_GC(gc),
                       (visitproc)visit_decref, // 回调形式传入
                       NULL);
    }
}
```

我们可以看看dictobject的traverse函数

```c
static int
dict_traverse(PyObject *op, visitproc visit, void *arg)
{
    Py_ssize_t i = 0;
    PyObject *pk;
    PyObject *pv;

    // 遍历所有键和值
    while (PyDict_Next(op, &i, &pk, &pv)) {
        Py_VISIT(pk);
        Py_VISIT(pv);
    }
    return 0;
}
```

逻辑大概是: 遍历容器对象里面的所有对象, 通过 `visit_decref` 将这些对象的引用计数都-1,

最终，遍历完链表之后，整个可收集对象链表中所有container对象之间的循环引用都被去

4. 第四步：垃圾标记

`move_unreachable` ，将可收集对象链表中，根据有效引用计数 不等于0(root对象) 和 等于0(非root对象，垃圾，可回收)，一分为二

```
/* Move the unreachable objects from young to unreachable.  After this,
 * all objects in young have gc_refs = GC_REACHABLE, and all objects in
 * unreachable have gc_refs = GC_TENTATIVELY_UNREACHABLE.  All tracked
 * gc objects not in young or unreachable still have gc_refs = GC_REACHABLE.
 * All objects in young after this are directly or indirectly reachable
 * from outside the original young; and all objects in unreachable are
 * not.
 */
static void
move_unreachable(PyGC_Head *young, PyGC_Head *unreachable)
{
    PyGC_Head *gc = young->gc.gc_next;

    /* Invariants:  all objects "to the left" of us in young have gc_refs
     * = GC_REACHABLE, and are indeed reachable (directly or indirectly)
     * from outside the young list as it was at entry.  All other objects
     * from the original young "to the left" of us are in unreachable now,
     * and have gc_refs = GC_TENTATIVELY_UNREACHABLE.  All objects to the
     * left of us in 'young' now have been scanned, and no objects here
     * or to the right have been scanned yet.
     */

    while (gc != young) {
        PyGC_Head *next;

        // 对于root object,
        if (gc->gc.gc_refs) {
            /* gc is definitely reachable from outside the
             * original 'young'.  Mark it as such, and traverse
             * its pointers to find any other objects that may
             * be directly reachable from it.  Note that the
             * call to tp_traverse may append objects to young,
             * so we have to wait until it returns to determine
             * the next object to visit.
             */
            PyObject *op = FROM_GC(gc);
            traverseproc traverse = Py_TYPE(op)->tp_traverse;
            assert(gc->gc.gc_refs > 0);
            // 设置其gc->gc.gc_refs = GC_REACHABLE
            gc->gc.gc_refs = GC_REACHABLE;
```

```
            (void) traverse(op,
                            (visitproc)visit_reachable,
                            (void *)young);
            next = gc->gc.gc_next;
            if (PyTuple_CheckExact(op)) {
                _PyTuple_MaybeUntrack(op);
            }
        }
        // 有效引用计数=0, 非root对象, 移动到unreachable链表中
        else {
            /* This *may* be unreachable.  To make progress,
             * assume it is.  gc isn't directly reachable from
             * any object we've already traversed, but may be
             * reachable from an object we haven't gotten to yet.
             * visit_reachable will eventually move gc back into
             * young if that's so, and we'll see it again.
             */
            next = gc->gc.gc_next;
            gc_list_move(gc, unreachable);
            gc->gc.gc_refs = GC_TENTATIVELY_UNREACHABLE;
        }
        gc = next;
    }
}
```

5. 第五步: 将存活对象放入下一代

```
    /* Move reachable objects to next generation. */
    if (young != old) {
        if (generation == NUM_GENERATIONS - 2) {
            long_lived_pending += gc_list_size(young);
        }
        gc_list_merge(young, old);
    }
    else {
        /* We only untrack dicts in full collections, to avoid quadratic
           dict build-up. See issue #14775. */
        untrack_dicts(young);
        long_lived_pending = 0;
        long_lived_total = gc_list_size(young);
    }
```

6. 第六步: 执行回收

```
gcmoudle.c
```

```
static int
gc_list_is_empty(PyGC_Head *list)
{
    return (list->gc.gc_next == list);
}


/* Break reference cycles by clearing the containers involved.  This is
 * tricky business as the lists can be changing and we don't know which
 * objects may be freed.  It is possible I screwed something up here.
 */
static void
delete_garbage(PyGC_Head *collectable, PyGC_Head *old)
{
    inquiry clear;

    // 遍历
    while (!gc_list_is_empty(collectable)) {
        PyGC_Head *gc = collectable->gc.gc_next;
        // 得到对象
        PyObject *op = FROM_GC(gc);

        assert(IS_TENTATIVELY_UNREACHABLE(op));
        if (debug & DEBUG_SAVEALL) {
            PyList_Append(garbage, op);
        }
        else {
            // 清引用
            if ((clear = Py_TYPE(op)->tp_clear) != NULL) {
                Py_INCREF(op);
                // 这个操作会调整container对象中每个引用所有对象的引用计数，从而完成打破
                clear(op);
                Py_DECREF(op);
            }
        }

        // 重新送回到reachable链表.
        // 原因：在进行clear动作，如果成功，会把自己从垃圾收集机制维护的链表中摘除，由于
        if (collectable->gc.gc_next == gc) {
            /* object is still alive, move it, it may die later */
            gc_list_move(gc, old);
            gc->gc.gc_refs = GC_REACHABLE;
        }
    }
}
```

=> 来看下，list的clear

```
list_clear(PyListObject *a)
{
    Py_ssize_t i;
    PyObject **item = a->ob_item;
    if (item != NULL) {
        /* Because XDECREF can recursively invoke operations on
           this list, we make it empty first. */
        i = Py_SIZE(a);
        Py_SIZE(a) = 0;
        a->ob_item = NULL;
        a->allocated = 0;
        while (--i >= 0) {
            // 减引用
            Py_XDECREF(item[i]);
        }
        PyMem_FREE(item);
    }
    /* Never fails; the return value can be ignored.
       Note that there is no guarantee that the list is actually empty
       at this point, because XDECREF may have populated it again! */
    return 0;
}


// e.g. 处理list3, 调用其list_clear, 减少list4的引用计数, list4.ob_refcnt=0, 引发对象销


static void
list_dealloc(PyListObject *op)
{
    Py_ssize_t i;
    PyObject_GC_UnTrack(op);  //  从可收集对象链表中去除, 会影响到list4所引用所有对象的引

    Py_TRASHCAN_SAFE_BEGIN(op)
    if (op->ob_item != NULL) {
        /* Do it backwards, for Christian Tismer.
           There's a simple test case where somehow this reduces
           thrashing when a *very* large list is created and
           immediately deleted. */
        i = Py_SIZE(op);
        while (--i >= 0) {
            Py_XDECREF(op->ob_item[i]);
        }
        PyMem_FREE(op->ob_item);
    }
    if (numfree < PyList_MAXFREELIST && PyList_CheckExact(op))
        free_list[numfree++] = op;
```

```
    Py_TRASHCAN_SAFE_END(op)
}
```

7. gc逻辑

```
分配内存
-> 发现超过阈值了
-> 触发垃圾回收
-> 将所有可收集对象链表放到一起
-> 遍历, 计算有效引用计数
-> 分成 有效引用计数=0 和 有效引用计数 > 0 两个集合
-> 大于0的, 放入到更老一代
-> =0的, 执行回收
-> 回收遍历容器内的各个元素, 减掉对应元素引用计数(破掉循环引用)
-> 执行-1的逻辑, 若发现对象引用计数=0, 触发内存回收
-> python底层内存管理机制回收内存
```

分代回收

分代收集: 以空间换时间

思想: 将系统中的所有内存块根据其存货的时间划分为不同的集合, 每个集合就成为一个"代", 垃圾收集的频率随着"代"的存活时间的增大而减小(活得越长的对象, 就越不可能是垃圾, 就应该减少去收集的频率)

Python中, 引入了分代收集, 总共三个"代". Python 中, 一个代就是一个链表, 所有属于同一"代"的内存块都链接在同一个链表中

表头数据结构

```
gcmodule.c


  struct gc_generation {
      PyGC_Head head;
      int threshold; /* collection threshold */  // 阈值
      int count; /* count of allocations or collections of younger
                    generations */    // 实时个数
  };
```

三个代的定义

```
#define NUM_GENERATIONS 3
#define GEN_HEAD(n) (&generations[n].head)

//  三代都放到这个数组中
/* linked lists of container objects */
static struct gc_generation generations[NUM_GENERATIONS] = {
    /* PyGC_Head,                               threshold,      count */
    {{{GEN_HEAD(0), GEN_HEAD(0), 0}},           700,            0},     //700个cor
    {{{GEN_HEAD(1), GEN_HEAD(1), 0}},           10,             0},     // 10个
    {{{GEN_HEAD(2), GEN_HEAD(2), 0}},           10,             0},     // 10个
};

PyGC_Head *_PyGC_generation0 = GEN_HEAD(0);
```

超过阈值，触发垃圾回收

```
PyObject *
 _PyObject_GC_Malloc(size_t basicsize)
 {
     // 执行分配
     ....
     generations[0].count++; /* number of allocated GC objects */  //增加一个
     if (generations[0].count > generations[0].threshold && // 发现大于预支了
         enabled &&
         generations[0].threshold &&
         !collecting &&
         !PyErr_Occurred())
         {
             collecting = 1;
             collect_generations();  //  执行收集
             collecting = 0;
         }
     op = FROM_GC(g);
     return op;
 }
```

```
=> collect_generations
```

```
static Py_ssize_t
collect_generations(void)
{
    int i;
    Py_ssize_t n = 0;

    /* Find the oldest generation (highest numbered) where the count
```

```
        // 从最老的一代, 开始回收
        for (i = NUM_GENERATIONS-1; i >= 0; i--) {  // 遍历所有generation
            if (generations[i].count > generations[i].threshold) {  // 如果超过了阈值
                /* Avoid quadratic performance degradation in number
                   of tracked objects. See comments at the beginning
                   of this file, and issue #4074.
                */
                if (i == NUM_GENERATIONS - 1
                    && long_lived_pending < long_lived_total / 4)
                    continue;
                n = collect(i); // 执行收集
                break;  // notice: break了
            }
        }
        return n;
    }
```

Python 中的gc模块

gc模块, 提供了观察和手动使用gc的接口

```
import gc

gc.set_debug(gc.DEBUG_STATS | gc.DEBUG_LEAK)

gc.collect()
```

注意 `__del__` 给gc带来的影响

#python

5015 Words

2015-09-29 08:00 +0800

← NEWER
这段时间的一些想法

OLDER →
我为什么要写博客

comments powered by Disqus

WKLKEN BUILDING