《源码探秘 CPython》7. 对象的引用计数，对象何时被回收？

原创 古明地觉 古明地觉的编程教室 2022-01-10 09:30

楔子

在 C 和 C++ 中，程序员被赋予了极大的自由，可以任意地申请内存。但权力的另一面对应着责任，程序员最后不使用的时候，必须负责将申请的内存释放掉，并把无效指针设置为空。可以说，这一点是万恶之源，大量内存泄漏、悬空指针、越界访问的 bug 由此产生。

而现代的开发语言都带有垃圾回收机制，语言本身负责内存的管理和维护，比如 C#、Java、Go。垃圾回收机制将开发人员从维护内存分配和清理的繁重工作中解放出来，但同时也剥夺了程序员和内存亲密接触的机会，并牺牲了一定的运行效率。不过好处就是提高了开发效率，并降低了 bug 发生的概率。

Python 里面同样具有垃圾回收机制，只不过它是为引用计数机制服务的。所以解释器通过内部的引用计数和垃圾回收，代替程序员进行繁重的内存管理工作，关于垃圾回收我们后面会详细说，先来看一下引用计数。

引用计数

Python 通过管理对象的引用计数来决定对象在内存中的存在与否，我们知道Python 中一切皆对象，所有对象都有一个**ob_refcnt**成员。这个成员维护着对象的引用计数，从而也最终决定着对象的存在与消亡。

我们在研究对象行为的时候说过：**比起类型对象，我们更关注实例对象的行为**。引用计数也是如此，只有实例对象，我们探讨引用计数才是有意义的。类型对象（内置）超越了引用计数规则，永远都不会被析构，或者销毁，因为它们在底层是被静态定义好的。同理，我们自定义的类，虽然可以被回收，但是探讨它的引用计数也是没有价值的。我们举个栗子：

```
class A:
    pass

del A
```

首先 del 关键字只能作用于变量，不可以作用于对象，比如：pi = 3.14，你可以 del pi，但是不可以 del 3.14，这是不符合语法规则的。因为 del 的作用是让变量指向的对象的引用计数减 1，所以我们只能 del 变量，不可以 del 对象。

我们使用 def、class 关键字定义出来也属于变量，比如上面代码中的 A，只要是变量，就可以被 del。但是**del 变量**只是删除了该变量，换言之就是让该变量无法再被使用，至于变量指向的对象是否会被回收，就看是否还有其它的引用也指向它。也就是说，对象是否被回收完全由解释器判断它的引用计数是否为 0 所决定。

因此当对象的所有引用都被删除之后，那么该对象也会被删除。以我们上面的代码为例，里面的 A 也是一个变量，引用了某个 PyTypeObject 结构体实例，并且它也只被变量 A 引用。所以当 del A 之后，底层的类对象也会被销毁。

另外，我们虽然说 int、str、tuple 这些是类型对象，但这只是从 Python 的层面。如果从解释器的角度来看，比如 int，它也是一个变量，指向对应的**数据结构(PyLong_Type)**，既然是变量，那么就可以被删除。

```
try:
```

```
    del int
except NameError as e:
    print(e)  # name 'int' is not defined
```

但是我们看到在删除的时候报错了，原因就在于 del 只能删除局部作用域和全局作用域内的变量(也就是局部变量和全局变量)，对于内置作用域里面的变量，del 是无法删除的。

如果想删除，则需要显式地指定内置作用域，举个栗子：

```
# 内置类型、内置函数都在 __builtins__ 里面
print(__builtins__)  # <module 'builtins' (built-in)>
# __builtins__ 等价于 import builtins
# 我们说变量查找默认按照 LEGB 规则
# 所以调用 __builtins__.int 比直接调用 int 要快一些

# 我们将 int 给删除掉，可以直接 del __builtins__.int
# 但这里获取了模块的属性字典
# 主要想表明操作模块本质上就是操作模块的属性字典
# 比如 module.attr 相当于 module.__dict__["attr"]
__builtins__.__dict__.pop("int")
try:
    print(int(3))
except NameError as e:
    print(e)  # name 'int' is not defined
```

我们看到此时 int 就不能使用了，因为它已经从内置作用域中被删除了。不过 int 虽然被删除了，但是 int 底层指向的 PyLong_Type 却没有被删除。

```
del __builtins__.int

try:
    print(int(3))
except NameError as e:
    print(e)  # name 'int' is not defined

# int 虽然被删除
# 但是 int 指向的类型对象没有被删除
n = 123
print(n.__class__)  # <class 'int'>
print(n.__class__("0xFF", base=16))  # 255

```

之所以会有这个现象，是因为底层的 PyLong_Type 并不仅仅被 int 这个变量所引用，我们来查看一下它的引用计数吧。

```
import sys

print(sys.getrefcount(int))  # 83
my_int = int
print(sys.getrefcount(int))  # 84
```

我们看到整形对象的引用计数非常多，至于它都被谁引用了，就无需我们关心了。总之，我们探讨类型对象的引用计数是没有太大意义的，而且内置类型超越了引用计数的规则，我们没必要太关注。我们重心是在实例对象上，相关操作也都是在实例对象上进行的。

```
>>> import sys
>>> e = 2.71
>>> sys.getrefcount(e)
2
```

创建一个新对象，显然此时的引用计数为 1，但为啥显示的是 2 呢？很简单，因为 e 这

个变量作为参数传到了 sys.getrefcount 这个函数里面，所以函数里面的参数也指向 2.71 这个 PyFloatObject，所以引用计数加1。当函数结束后，局部变量被销毁，再将引用计数减 1。

```
>>> e1 = e
>>> sys.getrefcount(e)
3
```

变量间的传递会传递指针，所以 e1 也会指向 2.71 这个浮点数，因此对象的引用计数加 1。注意：我们说变量只是个符号，引用计数是针对变量指向的对象而言的，变量本身没有所谓的引用计数。
此时变量e指向的对象的引用计数为3(**sys.getrefcount**函数的参数对**对象**的引用也算在内)。

```
>>> sys.getrefcount(e1)
3
```

我们说操作变量相当于操作变量指向的对象，由于 e 和 e1 都指向同一个对象，所以结果是一样的，也是 3。因为获取的是同一个对象的引用计数。

```
lst = [e, e1]
>>> sys.getrefcount(e)
5
```

放在容器里面，显然列表 lst 中多了两个指针，这两个指针也指向这里的**PyFloatObject对象**，因此结果为 5。

```
del lst
>>> sys.getrefcount(e)
3
```

将列表删除、或者将列表清空，那么里面的变量也就没了。而在删除变量之后，会将变量指向的对象的引用计数减去 1，所以又变成了 3。

```
# 再删除一个变量，引用计数再减 1
>>> del e1
>>> sys.getrefcount(e)
2
# 结果为 2，说明外部还有一个变量在引用它
# 所以这个浮点数不会被回收。
# 如果再次del，引用计数就会为0，这个浮点数就真的没了。
>>> del e
>>>
```

另外，引用计数什么时候会加1，什么时候会减1，我们在之前的文章中也说的很详细了，可以去看一下。

而在底层，解释器会通过**Py_INCREF(op)**和**Py_DECREF(op)**两个宏，来增加和减少一个对象的引用计数，当一个对象的引用计数减少到0后，**Py_DECREF**将调用该对象的析构函数来释放该对象所占有的内存和系统资源。这个析构函数由对象的类型对象(Py***_Type)中定义的函数指针来指定，也就是 tp_dealloc。

我们来看一下底层的这些宏：

```
#define _Py_NewReference(op) (                          \
    _Py_INC_TPALLOCS(op) _Py_COUNT_ALLOCS_COMMA         \
    _Py_INC_REFTOTAL  _Py_REF_DEBUG_COMMA               \
    Py_REFCNT(op) = 1)
    //对于新创建的对象，引用计数为 1

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

```
//引用计数为 0 时执行析构函数
//Py_TYPE(op)->tp_dealloc获取析构函数对应的函数指针
//再通过 * 获取指向的函数，函数接收 PyObject *
//所以将op转成PyObject *进行调用，最终回收对象


//增加引用计数
#define Py_INCREF(op) (                         \
    _Py_INC_REFTOTAL  _Py_REF_DEBUG_COMMA       \
    ((PyObject *)(op))->ob_refcnt++)
    //引用计数自增 1


//减少引用计数
#define Py_DECREF(op)                                   \
    do {                                                \
        PyObject *_py_decref_tmp = (PyObject *)(op);    \
        if (_Py_DEC_REFTOTAL  _Py_REF_DEBUG_COMMA       \
        --(_py_decref_tmp)->ob_refcnt != 0)             \
            _Py_CHECK_REFCNT(_py_decref_tmp)            \
        else                                            \
            _Py_Dealloc(_py_decref_tmp);                \
            //引用计数减1，如果减完1变成了0，则执行析构函数
    } while (0)

//注意:Py_INCREF和Py_DECREF不可以处理NULL指针，会报错
//所以又有两个宏，做了一层检测，会判断对象指针为NULL的情况
#define Py_XINCREF(op)                                  \
    do {                                                \
        PyObject *_py_xincref_tmp = (PyObject *)(op); \
        if (_py_xincref_tmp != NULL)                    \
            Py_INCREF(_py_xincref_tmp);                 \
    } while (0)

#define Py_XDECREF(op)                                  \
    do {                                                \
        PyObject *_py_xdecref_tmp = (PyObject *)(op); \
        if (_py_xdecref_tmp != NULL)                    \
            Py_DECREF(_py_xdecref_tmp);                 \
    } while (0)
//当然减少引用计数，除了Py_DECREF和Py_XDECREF之外
//还有一个Py_CLEAR，也可以处理空指针的情况
```

因此这几个宏作用如下：

- _Py_NewReference：接收一个对象，将其引用计数设置为1，用于新创建的对象。此外我们在定义里面还看到了一个宏**Py_REFCNT**，这是用来获取对象引用计数的，也就是获取 ob_refcnt 成员的值。当然除了**Py_REFCNT**之外，我们之前还见到了一个宏叫**Py_TYPE**，这是专门获取对象的类型的，得到的是一个指向 PyTypeObject 结构体实例的指针，也就是 ob_type 成员的值
- _Py_Dealloc：接收一个对象, 执行该对象的类型对象里面的析构函数, 来对该对象进行回收
- Py_INCREF：接收一个对象, 将该对象引用计数自增1，即ob_refcnt++
- Py_DECREF：接收一个对象, 将该对象引用计数自减1，即ob_refcnt--，如果自减之后发现为0，那么调用**_Py_Dealloc**
- Py_XINCREF：和**Py_INCREF**功能一致，但是可以处理空指针
- Py_XDECREF：和**Py_DECREF**功能一致，但是可以处理空指针
- Py_CLEAR：和**Py_XDECREF**类似，也可以处理空指针

在一个对象的引用计数为0时，与该对象对应的析构函数就会被调用，但是要特别注意的是，我们刚才一直说调用析构函数之后会回收对象，或者销毁对象、删除对象等等，意思都是将这个对象从内存中抹去，但是这并不意味着最终一定调用free释放空间。换句话说就是对象没了，但是对象占用的内存却有可能还在。

如果对象没了，占用的内存也要释放的话，那么频繁申请、释放内存空间会使Python的执行效率大打折扣，更何况Python已经背负了人们对其执行效率的不满这么多年。

所以Python底层大量采用了缓存池的技术，使用这种技术可以避免频繁地申请和释放内存空间。因此在析构的时候，只是将对象占用的空间归还到缓存池中，并没有真的释放。

这一点，在后面剖析内置实例对象(PyFloatObject，PyListObject等等)的实现中，将会看得一清二楚，因为大部分内置的实例对象都会有自己的缓存池。

对象的分类

这里再补充一下对象的分类，我们之前根据支持的操作，将Python对象分成了数值型、序列型、映射型，总共3类，但其实我们是可以分为5类的：

- Fundamental对象：类型对象，如int、float、bool、tuple
- Numeric对象：数值对象，如int实例、float实例、bool实例
- Sequence对象：序列对象，如str实例、list实例、tuple实例
- Mapping对象：关联对象(映射对象)，如dict实例
- Internal对象：虚拟机在运行时内部使用的对象，如**function实例(函数)**、**code实例(字节码)**、**frame实例(栈帧)**、**module实例(模块)**、**method实例(方法)**。没错，函数、字节码、栈帧、模块、方法等等它们在底层也都是一个类的实例对象。比如：函数的类型是**<class 'function'>**，在底层对应PyFunctionObject。那么问题来了，**<class 'function'>**的类型对象是什么呢？显然就是**<class 'type'>**啦。

小结

到此我们的基础概念就算说完了，从下一篇文章开始就要详细剖析内置对象的底层实现了，比如浮点数、整数、字符串、元组、列表等等都是如何实现的。

有了目前为止的这些基础，我们后面就会轻松很多，先把对象、变量等概念梳理清楚，然后再来搞这些数据结构的底层实现。

收录于合集 #CPython 97

‹ 上一篇
《源码探秘 CPython》8. 浮点数的创建与销毁

下一篇 ›
《源码探秘 CPython》6. 对象的多态性、和行为