《源码探秘 CPython》9. 浮点数的缓存池机制

原创 古明地觉 古明地觉的编程教室 2022-01-12 09:30

缓存池

我们说浮点数这种对象是经常容易被创建和销毁的，如果每次创建都借助操作系统分配内存、每次销毁都借助操作系统回收内存的话，那效率会低到什么程度，可想而知。

因此Python解释器在操作系统之上封装了一个内存池，在内存管理的时候会详细介绍内存池，目前可以认为内存池就是预先向操作系统申请的一部分内存，专门用于小内存对象的快速创建和销毁，这便是Python的内存池机制。

但浮点数使用的频率很高，我们有时会创建和销毁大量的临时对象，所以如果每一次对象的创建和销毁都伴随着内存相关的操作的话，这个时候即便是有内存池机制，效率也是不高的。

考虑如下代码：

```
>>> pi = 3.14
>>> r = 2.0
>>> s = pi * r ** 2
>>> s
12.56
>>>
```

这个语句首先计算半径r的平方，然后根据结果创建一个临时对象，假设是t；然后再将pi和t进行相乘，得到最终结果并赋值给s；最终销毁临时变量t，所以这背后是隐藏着一个临时对象的创建和删除的。

当然这里一行代码可能感觉不到啥，假设我们要计算很多很多个半径对应的面积呢？显然需要写for循环，如果循环一万次就意味着要创建和销毁临时对象各一万次。

因此，如果每一次创建对象都需要分配内存，销毁对象时需要回收内存的话，那么大量临时对象的创建和销毁就意味着也要伴随大量的**内存分配以及回收操作**，这显然是无法忍受的，更何况Python本身就已经够慢了。

因此Python在浮点数对象被销毁后，并不急着回收对象所占用的内存，换句话说其实对象还在，只是将该对象放入一个空闲的链表中。

之前我们说对象可以理解为一片内存空间，对象如果被销毁，那么理论上内存空间要归还给操作系统，或者回到内存池中。但Python考虑到效率，并没有真正的销毁对象，而是将对象放入到链表中，占用的内存还在。

后续如果再需要创建新的浮点数对象时，那么从链表中直接取出之前放入的对象(我们认为被回收的对象)，然后根据新的浮点数对象重新初始化对应的成员即可，这样就避免了内存分配造成的开销。而这个链表就是我们说的**缓存池**，当然不光浮点数对象有**缓存池**，Python中的很多其它对象也有对应的**缓存池**，比如列表。

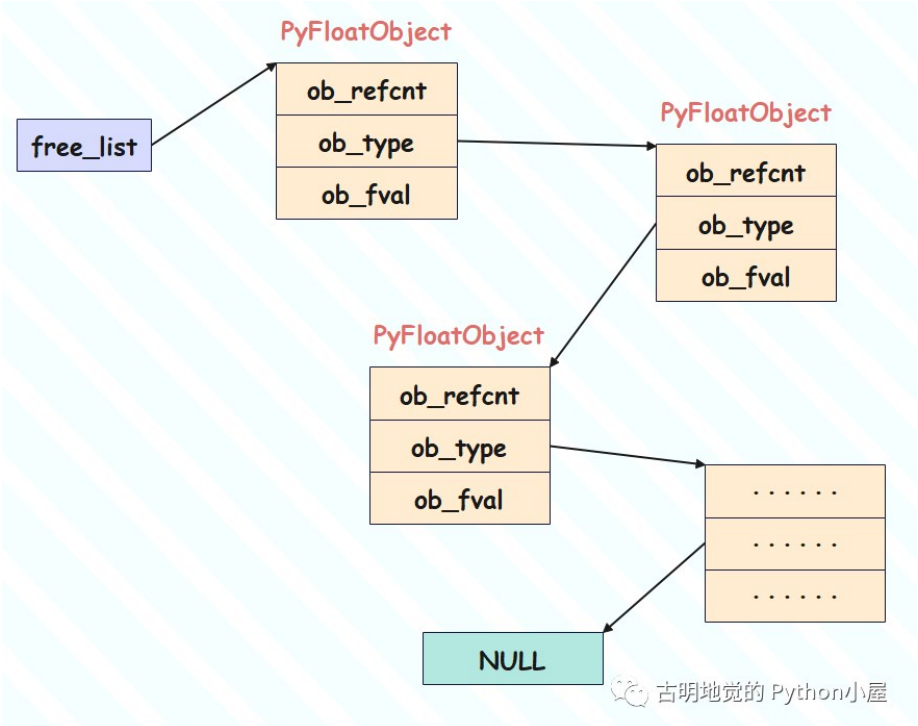
而浮点对象的缓存池(链表)同样在 Objects/floatobject.c中定义：

```
#ifndef PyFloat_MAXFREELIST
#define PyFloat_MAXFREELIST    100
#endif
static int numfree = 0;
static PyFloatObject *free_list = NULL;
```

- PyFloat_MAXFREELIST：缓存池(链表)中能容纳的浮点数的最大数量，说白了就是链表的最大长度，这里是100个，因为不可能将所有要销毁的PyFloatObject实例都放入到缓存池中
- numfree：表示当前缓存池(链表)中已经存在的浮点数的个数, 初始为0
- free_list: 指向链表头结点的指针, 链表里面存储的都是PyFloatObject, 所以头节点的指针就是**PyFloatObject ***

但是问题来了，如果是通过链表来存储的话，那么对象肯定要有一个指针，来指向下一个对象，但是浮点数对象内部似乎没有这样的指针啊。

是的，因为解释器是使用内部的ob_type来指向下一个对象，本来ob_type指向的应该是PyFloat_Type，但在缓存池中指向的是下一个PyFloatObject。



以上就是浮点数的缓存池，说白了就是一个链表，free_list指向链表的头结点，节点之间通过**ob_type**充当**next**指针。

所以**PyFloat_FromDouble**这个API，我们再来回顾一下：

```
PyObject *
PyFloat_FromDouble(double fval)
{
    //显然op是缓存池中第一个PyFloatObject的指针
    PyFloatObject *op = free_list;
    if (op != NULL) {
        // 上一篇文章中此处没有细说
        // 所以下面就来展开一下
        free_list = (PyFloatObject *) Py_TYPE(op);
        numfree--;
    } else {
        op = (PyFloatObject*) PyObject_MALLOC(sizeof(PyFloatObject));
        if (!op)
            return PyErr_NoMemory();
    }
    //.......
    return (PyObject *) op;
}
```

当op不为NULL时，说明缓存池中有缓存好的对象，于是会将链表的头结点取出来重新分配。但是还要维护free_list，因此要获取下一个节点(PyFloatObject实例)，然后让free_list指向它。

在链表中，ob_type被用于指向下一个**PyFloatObject**，换言之ob_type保存的是下一个**PyFloatObject的地址**。不过话虽如此，可它的类型仍是struct _typeobject *，或者说PyTypeObject *，因此在存储的时候，下一个**PyFloatObject ***一定是先转成了**PyTypeObject ***，之后再交给的ob_type，因为对于指针来说，是可以任意转化的，我们一会在看 float_dealloc 的时候就知道了。

那么同理，这里的**Py_TYPE(op)**在获取下一个对象的指针之后，还要再转成PyFloatObject *，然后才能交给free_list保存。如果没有下一个对象了，那么**free_list**就是NULL。在下一次分配的时候，上面的if条件**(op!=NULL)**就会不成立，从而走下面的else，使用**PyObject_MALLOC**重新分配内存。

以上就是缓存池在浮点数在的创建过程中起到的作用，也就是**对象创建时，会先从缓存池中获取**。

既然创建时可以从缓存池获取，那么销毁的时候，肯定要放入到缓存池中。而销毁对象时，会调用类型对象的析构函数tp_dealloc，对于浮点数而言就是float_dealloc，我们看一下源代码，同样位于Objects/floatobject.c中。

```
static void
float_dealloc(PyFloatObject *op)
{
    if (PyFloat_CheckExact(op)) {
     //numfree就是当前缓存池已容纳的PyFloatObject实例的数量
     //如果达到了缓存池的最大容量
        if (numfree >= PyFloat_MAXFREELIST)  {
     //那么调用PyObject_FREE回收对象所占内存
     //因为缓存池的容量不是无限的，这里是100个
     //当然我们可以修改解释器源代码改变这一点
     //另外注意这里的PyObject_FREE
     //我们说Python/C API分为两种
     //显然这种格式的属于"泛型 API"
            PyObject_FREE(op);
            return;
        }
     //否则的话，说明没有达到最大容量限制
     //显然此时不会真的销毁对象，而是将其放入缓存池中
     //然后将numfree加1
        numfree++;
     //我们说free_list指向链表的第一个节点
     //而这里是获取了op的ob_type，让其等于free_list
     //说明该对象内部的ob_type指向了链表中的头结点
     //那么显然该对象就成了链表的新的头结点
     //因此可以看出，对象在插入链表的时候，采用的头插法
     //但ob_type的类型是struct _typeobject *
     //所以交给ob_type保存的时候，还要将free_list的类型转化一下
     //而在获取的时候，再转成PyFloatObject *
     //这在上面的PyFloat_FromDouble中我们已经看到了
        Py_TYPE(op) = (struct _typeobject *)free_list;
     //free_list始终指向链表中的头结点，但现在头结点变了
     //所以最后再让free_list = op，指向新添加的PyFloatObject，
     //因为它被插入到了链表的第一个位置上
        free_list = op;
    }
    //否则的话，说明PyFloat_CheckExact(op)为假
    //PyFloat_CheckExact(op)用于检测op的类型是不是float
    //为假的话，说明此时op的类型不是float
    //那么通过Py_TYPE(op)->tp_free直接获取对应的类型对象的tp_free
    //然后释放掉op指向的对象所占的内存
    else
        Py_TYPE(op)->tp_free((PyObject *)op);
}
```

这便是Python浮点数缓存池的全部秘密，由于缓存池在提高对象分配效率方面发挥着至

关重要的作用，所以Python很多其它的内置实例对象也都实现了缓存池，我们后续在分析的时候会经常看到它的身影。

说白了缓存池的作用只有一个，就是在对象被销毁的时候不释放所占的内存，下次创建新的对象时能够直接拿来用。因为内存没有被释放，因此创建起来就快很多。

看一个思考题：

```
>>> a = 1.414
>>> id(a)
2431274355248
>>>
>>> del a
>>>
>>> b = 1.732
>>> id(b)
2431274355248
>>>
```

我们看到两个对象的id是一样的，相信你肯定知道原因。因为a在**del**之后，对象被放入到缓存池中，然后创建b的时候会从缓存池中获取。所以a指向的对象被重新利用了，内存还是原来的那一块内存，只不过将**ob_fval**的值从1.414改成了1.732，所以前后地址没有变化。

这就是缓存池，不需要任何内存分配，一个对象就出来了。

修改解释器、验证缓存池

最后我们修改一下源码：当对象放入到缓冲池中，我们打印一下放入的浮点数对象的地址；当对象从缓存池中取出时，我们打印一下取出的浮点数对象的地址。

```
>>> e = 2.71
从缓存池中获取，地址：0x7f5c4a46e790
>>>
>>> hex(id(e))
'0x7f5c4a46e790'
>>>
>>> del e
对象放入缓存池中，放入的对象的地址：0x7f5c4a46e790
>>>
```


我们第一次创建对象的时候，居然是从缓存池里面获取的，说明在解释器启动之后，链表中就已经有空闲对象了。因为解释器启动时，会做大量的初始化工作。

然后我们使用Python获取它的id，这里转成了16进制，发现地址是一样的。然后放入到缓存池中，放入的对象的地址也是相同的，这和我们得到结论是一致的。

```
>>> a = 22.33
从缓存池中获取，地址：0x7f5c4a46e790
>>> b = 6.66
从缓存池中获取，地址：0x7f5c4a3dc7f0
>>>
>>> del a, b
对象放入缓存池中，放入的对象的地址：0x7f5c4a46e790
对象放入缓存池中，放入的对象的地址：0x7f5c4a3dc7f0
>>>
>>> a = 22.33
从缓存池中获取，地址：0x7f5c4a3dc7f0
>>> b = 6.66
从缓存池中获取，地址：0x7f5c4a46e790
>>>
```


我们看到a指向的对象的地址，和上面变量e指向的对象的地址是一样，说明内存被重新

利用了，然后我们再来看看 a、b 之间的关系。

我们创建新的变量a、b并打印地址，然后删除a、b变量，再重新创建a、b变量并打印地址，结果发现它们存储的对象的地址在删除前后正好是相反的。至于原因，如果思考一下将对象放入缓存池、以及从缓存池获取对象的时候所采取的策略，那么很容易就明白了。

因为**del a, b**的时候会先删除a，再删除b。删除a的时候，会将a指向的对象作为链表中的头结点，然后删除b的时候，会将b指向的对象作为链表中的新的头结点，所以之前a指向的对象就变成了链表中的第二个节点。

而获取的时候，也会从链表的头部开始获取，所以当重新创建变量a的时候，其指向的对象实际上使用的是之前变量b指向的对象所占的内存，而一旦获取，那么free_list指针会向后移动。

因此创建变量b的时候，其指向的对象使用的就是之前变量a指向的对象所占的内存。因此前后打印的地址是相反的，所以我们算是通过实践从另一个角度印证了之前分析的结论。

通过ctypes模拟底层数据结构

有时我们想观察底层数据结构的表现行为时，不一定非要修改解释器，因为那样太麻烦，还要重新编译。Python 在上层给我们提供了一种方式，可以让我们通过Python的类轻松地模拟C的结构体。

```
from ctypes import *

class PyObject(Structure):
    """
    我们继承 ctypes.Structure
    此时就得到 C 的结构体
    然后通过 _fields_ 指定结构体成员
    """
    _fields_ = [
    # _fields_ 是一个列表
    # 内部的元组对应结构体的成员
        ("ob_refcnt", c_ssize_t),
        ("ob_type", c_void_p)
    ]
    # ob_refcnt 是 Py_ssize_t 类型
    # 等价于 c_ssize_t
    # 至于 ob_type，我们就用 void *

class PyFloatObject(PyObject):
    """
    继承PyObject，相当于结构体的嵌套
    """
    _fields_ = [
        ("ob_fval", c_double)
    ]

e = 2.71
# 创建PyFloatObject实例，返回它的指针
# from_address表示根据对象的地址创建
f = PyFloatObject.from_address(id(e))
# 此时 e 和 f 都指向了 2.71 这个浮点数

# 注意接下来会发生神奇的一幕
print(
    e, hex(id(e))
)  # 2.71 0x1f9bf763810
```

```
37
38 # f 等价于底层的 PyFloatObject *
39 # 修改 ob_fval 成员
40 f.ob_fval = 3.14
41 # 再次打印
42 print(
43     e, hex(id(e))
44 )  # 3.14 0x1f9bf763810
```

我们看到**id(e)**在前后并没有发生改变，证明 e 指向的始终是同一个对象，但是它的值却变了。咦，不是说浮点数是不可变对象吗？**如果想变的话只能创建一个新的浮点数**，这样一来前后打印的地址应该会变才对啊。

首先说明结论是没错的，可这是从Python的角度而言。如果是从解释器的角度来看的话，没有什么可变不可变，只要我们想让它可变，那么它就是可变的。

为了更好的观察底层数据结构的表现，我们后面会经常使用这种方式，而且会介绍更多的**骚操作**，但是切记这种动态修改解释器的做法不可用于生产环境。

小结

以上就是浮点数的缓存池机制，简单来说是一种**空间换时间**的做法。

为了避免频繁地和内核打交道，CPython引入了内存池机制，事先会向操作系统申请一部分内存，然后根据大小分成不同的单元，按需分配。这样就无需频繁和操作系统的内核打交道了，因为系统调用是代价昂贵的操作。

但有了内存池还不够，我们知道Python所有的对象都是申请在堆上的，而在堆上分配内存，效率要比栈差很多。所以又引入了缓存池，对象在被销毁后不释放所占内存，而是通过一个链表串起来，留着下次备用。

收录于合集 #CPython 97

‹ 上一篇
《源码探秘 CPython》10. 浮点数的行为

下一篇 ›
《源码探秘 CPython》8. 浮点数的创建与销毁