《源码探秘 CPython》8. 浮点数的创建与销毁

原创 古明地觉 古明地觉的编程教室 2022-01-11 09:30

楔子

从现在开始，我们就来分析Python常见的内置类型对象、以及对应的实例对象，看看它们在底层是如何实现的。但说实话，我们在前面几节中介绍对象的时候，已经说了不少了，不过从现在开始要进行更深入的分析。

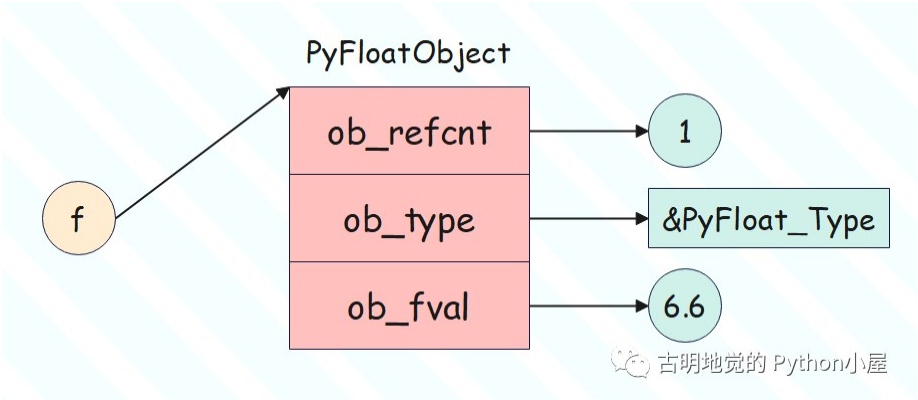
除了对象本身，还要看对象支持的操作在底层是如何实现的。我们首先以浮点数为例，因为它是最简单的，没错，浮点数比整数要简单。至于为什么，当我们分析整数的时候就知道了。

对象的结构

浮点数定义在**Include/floatobject.h**中，结构非常简单：

```
typedef struct {
    PyObject_HEAD
    double ob_fval;
} PyFloatObject;
```

除了**PyObject**这个公共的头部信息之外，只有一个额外的**ob_fval**，用于存储具体的值，并且使用的是 C 中的 double。以**f = 6.6**为例，底层结构如下：



还是很简单的，每个对象在底层都是由结构体表示的，这些结构体中有的成员负责维护对象的元信息，有的成员负责维护具体的值。比如这里的 6.6，总要有一个字段来存储 6.6 这个值，而这个字段就是**ob_fval**。所以浮点数的结构非常简单，直接使用一个 C 的 double 来维护。

假设我们要将两个浮点数相加，相信你已经知道解释器会如何做了？通过**PyFloat_AsDouble**，将两个PyFloatObject中的**ob_fval**抽出来，转成 C 的 double，然后进行相加，最后再根据相加的结果创建一个新的 PyFloatObject 即可。

浮点数（float 实例对象）的结构我们已经很清晰了，那么再来看看 float 类型对象在底层长什么样子。与实例对象不同，**float类型对象**全局唯一，底层对应定义好的静态全局变量PyFloat_Type，位于**Objects/floatobject.c**中。

```
PyTypeObject PyFloat_Type = {
    PyVarObject_HEAD_INIT(&PyType_Type, 0)
    "float",
    sizeof(PyFloatObject),
    0,
```

```
    (destructor)float_dealloc,                  /* tp_dealloc */
    0,                                          /* tp_print */
    0,                                          /* tp_getattr */
    0,                                          /* tp_setattr */
    0,                                          /* tp_reserved */
    (reprfunc)float_repr,                       /* tp_repr */
    &float_as_number,                           /* tp_as_number */
    0,                                          /* tp_as_sequence */
    0,                                          /* tp_as_mapping */
    (hashfunc)float_hash,                       /* tp_hash */
    0,                                          /* tp_call */
    (reprfunc)float_repr,                       /* tp_str */
    PyObject_GenericGetAttr,                    /* tp_getattro */
    0,                                          /* tp_setattro */
    0,                                          /* tp_as_buffer */
    Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE,   /* tp_flags */
    float_new__doc__,                           /* tp_doc */
    0,                                          /* tp_traverse */
    0,                                          /* tp_clear */
    float_richcompare,                          /* tp_richcompare */
    0,                                          /* tp_weaklistoffset */
    0,                                          /* tp_iter */
    0,                                          /* tp_iternext */
    float_methods,                              /* tp_methods */
    0,                                          /* tp_members */
    float_getset,                               /* tp_getset */
    0,                                          /* tp_base */
    0,                                          /* tp_dict */
    0,                                          /* tp_descr_get */
    0,                                          /* tp_descr_set */
    0,                                          /* tp_dictoffset */
    0,                                          /* tp_init */
    0,                                          /* tp_alloc */
    float_new,                                  /* tp_new */
};
```

PyFloat_Type保存了很多关于浮点数的元信息，关键字段包括：

- tp_name字段保存了类型名称，是一个**char ***，显然值为**"float"**
- tp_dealloc、tp_init、tp_alloc和tp_new字段是与对象创建销毁相关的函数
- tp_repr字段对应**__repr__**方法，生成语法字符串
- tp_str字段对应**__str__**方法，生成普通字符串
- tp_as_number字段对应数值对象支持的方法簇
- tp_hash字段是哈希值生成函数

PyFloat_Type很重要，作为浮点数的类型对象，它决定了浮点数的生死和行为。

浮点数的创建

下面我们来看看浮点数是如何创建的，在前面的文章中，我们初步了解了创建实例对象的一般过程。对于内置类型的实例对象，可以使用**Python/C API**创建，也可以通过**调用类型对象**创建。

调用**类型对象float**创建实例对象，解释器执行的是**类型对象type**中的tp_call函数。tp_call中会先调用类型对象（这里显然是float）的**tp_new**为其实例对象申请一份空间，申请完毕之后对象就已经创建好了。然后会再调用**tp_init**，并将实例对象作为参数传递进去，进行初始化，也就是设置属性。

但是对于float来说，它内部的**tp_init**成员是0，从PyFloat_Type的定义我们就可以看到。这就说明float没有**__init__**，原因是浮点数是一种很简单的对象，初始化操作只需要一个赋值语句，所以在tp_new中就可以完成。怎么理解这句话呢？我们举个栗子：

```python
class Girl1:

    def __init__(self, name, age):
        self.name = name
        self.age = age

# __new__   负责开辟空间、生成实例对象
# __init__  负责给实例对象绑定属性

# 但其实__init__所做的工作可以直接在__new__当中完成
# 换言之有 __new__ 就足够了，其实可以没有 __init__
# 我们将上面的例子改写一下
class Girl2:

    def __new__(cls, name, age):
        instance = object.__new__(cls)
        instance.name = name
        instance.age = age
        return instance


g1 = Girl1("古明地觉", 16)
g2 = Girl2("古明地觉", 16)
print(g1.__dict__ == g2.__dict__)  # True
```

我们看到效果是等价的，因为__init__里面是负责给 self 绑定属性的，这个 self 是__new__ 返回的。那么很明显，我们也可以在 __new__ 当中绑定属性，而不需要__init__。

但是按照规范，属性绑定应该放在 __init__ 中执行。只是对于浮点数而言，由于其结构非常简单，所以底层就没有给float实现__init__，所有都是在__new__当中完成的。

```python
print(float.__init__ is object.__init__)  # True
print(tuple.__init__ is object.__init__)  # True
print(list.__init__ is object.__init__)   # False
```

所以 float 没有 __init__，如果获取的话得到的是 object 的 __init__，因为 object 是 float 的基类。同理 tuple 也没有，但 list 是有的。

那么下面就来考察一下PyFloat_Type内部的tp_new成员，看看它是如何创建浮点数的。通过PyFloat_Type的定义我们可以看到，在创建的时候给成员tp_new设置是float_new，那么一切秘密就隐藏在float_new里面。

```c
static PyObject *
float_new_impl(PyTypeObject *type, PyObject *x)
{
    //如果type不是&PyFloat_Type，那么必须是它的子类
    //否则调用float_subtype_new会报错
    //但该条件很少触发，因为创建的是浮点数
    //所以type基本都是&PyFloat_Type
    if (type != &PyFloat_Type)
        return float_subtype_new(type, x);
    //然后检测这个 x 类型，如果它是一个字符串
    //那么就根据字符串创建浮点数，比如float("3.14")
    if (PyUnicode_CheckExact(x))
        return PyFloat_FromString(x);
    //不是字符串，则调用PyNumber_Float
    return PyNumber_Float(x);
}
```

所以核心就在于PyNumber_Float，该函数位于**Python/abstract.c**中。

```c
PyObject *
```

```
PyNumber_Float(PyObject *o)
{
    //方法簇
    PyNumberMethods *m;
    //传递的是NULL，直接返回错误
    if (o == NULL) {
        return null_error();
    }
    //如果传递过来的本来就是个浮点数
    //那么增加引用计数之后，直接返回
    if (PyFloat_CheckExact(o)) {
        Py_INCREF(o);
        return o;
    }
    //走到这里说明不是浮点数，那么它必须要能够转成浮点数
    //也就是类型对象的内部要有__float__这个魔法方法，即nb_float
    //这里拿到相应的方法簇
    m = o->ob_type->tp_as_number;
    //如果方法簇不为空，并且也实现了nb_float
    if (m && m->nb_float) {
        //那么调用nb_float，转成浮点数
        PyObject *res = m->nb_float(o);
        double val;
        //如果 res 不为空、并且是浮点数，直接返回
        //PyFloat_CheckExact检测一个对象的类型是否是float
        if (!res || PyFloat_CheckExact(res)) {
            return res;
        }
        //走到这里说明__float__返回的对象的类型不是一个float
        //如果不是float，那么float的子类目前也是可以的(会抛警告)
        //PyFloat_Check则检查对象的类型是否是float或者其子类
        if (!PyFloat_Check(res)) {
            //如果连子类也不是，那么就会引发TypeError
            //提示__float__返回的对象类型不是float
            PyErr_Format(PyExc_TypeError,
                         "%.50s.__float__ returned non-float (type %.50s)"
,
                         o->ob_type->tp_name, res->ob_type->tp_name);
            Py_DECREF(res);
            return NULL;
        }
        //到这里说明，res的类型是float的子类
        //那么获取ob_fval成员的值
        val = PyFloat_AS_DOUBLE(res);
        Py_DECREF(res);
        //构建浮点数，返回它的泛型指针 PyObject *
        return PyFloat_FromDouble(val);
    }
    //如果没有__float__，那么会去找__index__
    //这一点和我们之前说过的__int__类似
    if (m && m->nb_index) {
        PyObject *res = PyNumber_Index(o);
        if (!res) {
            return NULL;
        }
        //__index__返回的必须是整数
        //所以调用的是PyLong_AsDouble，而不是PyFloat_AsDouble
        double val = PyLong_AsDouble(res);
        Py_DECREF(res);
        if (val == -1.0 && PyErr_Occurred()) {
            return NULL;
        }
        //根据val构建PyFloatObject
        return PyFloat_FromDouble(val);
```

```
    }
    //如果类型不是float，并且内部也没有__float__和__index__
    //那么检测传递的对象的类型是不是float的子类
    //如果是，证明它的结构和浮点数是一致的
    //直接根据ob_fval构建PyFloatObject
    if (PyFloat_Check(o)) {
        return PyFloat_FromDouble(PyFloat_AS_DOUBLE(o));
    }
    //走到这里就真的没办法了，解释器实在不知道该咋办了
    //所以只能把它当成字符串来生成浮点数了
    return PyFloat_FromString(o);
}
```

所以一个 float 调用居然要走这么长的逻辑，当然了，这里面也存在很多快分支，总之解释器为我们考虑了很多。我们用 Python 来演绎一下：

```
# "3.14" 是个字符串
# 在tp_new里面直接调用PyFloat_FromString之后就返回了
print(float("3.14"))  # 3.14

class PI:
    def __float__(self):
        return 3.1415926

# 传递的参数是一个 PI 类型
# 所以会进入PyNumber_Float逻辑
# 由于对象实现了nb_float，所以会直接调用
print(float(PI()))  # 3.1415926
```

以上是通过类型对象创建，但我们说这种方式底层实际上也是调用了**Python/C API**。

```
PyObject *
PyFloat_FromDouble(double fval);

PyObject *
PyFloat_FromString(PyObject *v);
```

- PyFloat_FromDouble：通过 C 的浮点数创建Python浮点数
- PyFloat_FromString：通过Python字符串创建Python浮点数

如果我们是 f = 6.6 这种方式创建的话，那么解释器在编译的时候就知道这是一个浮点数，会调用 PyFloat_FromDouble 一步到胃，因为在该函数内部会直接根据 6.6 创建底层对应的PyFloatObject。而通过类型对象调用的话则会有一些额外的开销，因为这种方式最终也会调用相关的 Python/C API，但是在调用之前会干一些别的事情，比如类型检测等等。

所以 f = 6.6 比 float(6.6)、float("6.6") 都要高效。

还没结束，浮点数的实际创建过程我们还没有见到，因为最终还是调用 Python/C API 创建的。那么接下来就以PyFloat_FromDouble函数为例，看看浮点数在底层的创建过程，该函数同样位于**Objects/floatobject.c**中。

```
PyObject *
PyFloat_FromDouble(double fval)
{
    //在介绍引用计数时说过，引用计数为0，那么对象会被销毁
    //但是对象所占的内存则不一定回收，而是会缓存起来
    //所以从下面这行代码我们就看到了
    //创建浮点数对象的时候会优先从缓存池里面获取
    //而缓存池是使用链表实现的，每一个节点就是PyFloatObject实例
    //free_list(指针)指向链表的第一个节点
    PyFloatObject *op = free_list;
    //op不是NULL，说明缓存池中有对象，成功获取
```

```
    if (op != NULL) {
        //而一旦获取了，那么要维护free_list
        //要将free_list指向链表中的下一个节点
        //但问题来了，为啥获取下一个节点要通过Py_TYPE
        //Py_TYPE不是一个宏吗？用来获取的对象的ob_type
        //相信你已经猜到了，ob_type充当了链表中的next指针
        free_list = (PyFloatObject *) Py_TYPE(op);
        /*然后还要将缓存池(链表)的节点个数、
          也就是可以直接使用的浮点数对象的数量减去1*/
        //关于缓存池的具体实现，以及为什么要使用缓存池后续会细说
        //目前先知道Python在分配浮点数对象的时候
        //会先从缓存池里面获取就可以了
        numfree--;
    } else {
        //否则的话，说明缓存池里面已经没有可用对象了
        //那么会调用PyObject_MALLOC申请内存
        //PyObject_MALLOC是基于malloc的一个封装
        op = (PyFloatObject*) PyObject_MALLOC(sizeof(PyFloatObject));
        //申请失败的话，证明内存不够了
        if (!op)
            return PyErr_NoMemory();
    }

    //走到这里说明内存分配好了，PyFloatObject也创建了
    //但是不是还少了点啥呢？没错，显然内部的成员还没有初始化
    //还是那句话，内置类型的实例对象该分配多少空间，解释器了如指掌
    //因为通过PyFloatObject内部的成员一算就出来了。
    //所以虽然对象创建了，但是ob_refcnt、ob_type、以及ob_fval三个成员还没有被初始化
    //所以还要将其ob_refcnt设置为1(因为对于刚创建的对象来说，内部的引用计数显然为1)
    //并将ob_type设置为指向PyFloat_Type的指针，因为它的类型是float
    //而PyObject_INIT是一个宏，它就是专门用来设置ob_type以及ob_refcnt的
    //我们后面看这个宏的定义就知道了
    (void)PyObject_INIT(op, &PyFloat_Type);
    //将内部的ob_fval成员设置为fval，所以此时三个成员都已经初始化完毕
    op->ob_fval = fval;
    //将其转成PyObject *返回
    return (PyObject *) op;
}
```

所以整体流程如下：

- 1. 为实例对象分配内存空间，空间分配完了对象也就创建了，不过会优先使用缓存池
- 2. 初始化实例对象内部的引用计数和类型指针
- 3. 初始化ob_fval为指定的double值；

这里不知道你有没有发现一个现象，对于我们自定义的类而言，想要创建实例对象必须要借助于类型对象。

但使用Python/C API创建浮点数，却压根不需要类型对象float，而是直接就创建了。创建完之后再让其ob_type成员指向float，将类型和实例关联起来即可。

而之所以能够这么做的根本原因就在于内置类型的实例对象在底层都是静态定义好的，有多少成员已经写死了，所以创建的时候不需要类型对象。解释器知道创建这样的对象需要分配多少内存，所以会直接创建，创建完之后再对成员进行初始化，比如设置类型。

但是对于我们自定义的类而言，想要创建实例对象就必须要借助于类型对象了。

当然啦，这些内容之前也已经说过了，这里再啰嗦一遍，温故知新。这里我们不妨修改一下解释器源代码，根据输出内容猜猜我干了什么事情。

图片

最后看一下**PyObject_INIT**这个宏，它位于Include/objimpl.h中。

```
#define PyObject_INIT(op, typeobj) \
    ( Py_TYPE(op) = (typeobj), _Py_NewReference((PyObject *)(op)), (op) )
```

这个宏接收两个参数，分别是：实例对象的指针和对应的类型对象的指针，然后**Py_TYPE(op)**表示获取其内部的ob_type，将其设置为typeobj，而typeobj在源码中传入的是**&PyFloat_Type**。至于**_Py_NewReference**，这个宏我们在之前也说过了，它负责将对象的引用计数初始化为 1。

浮点数的销毁

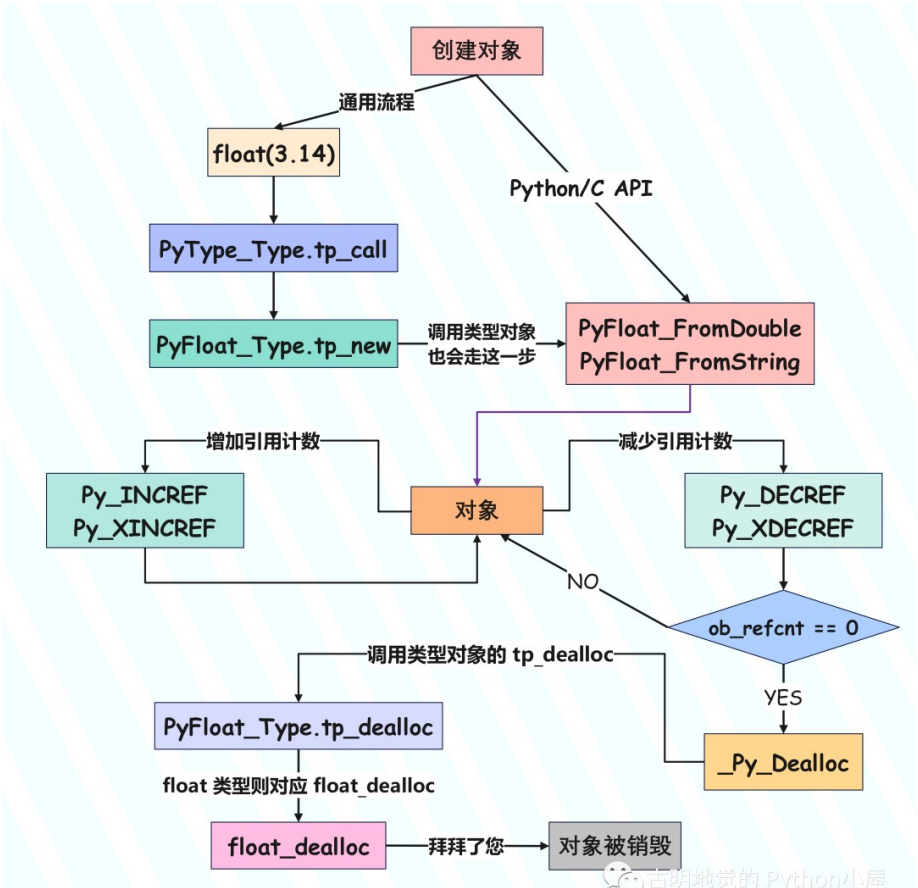
当删除一个变量时，Python会通过宏**Py_DECREF**或者**Py_XDECREF**来减少该变量指向的对象的引用计数；当引用计数为0时，就会调用其类型对象中的tp_dealloc指向的函数来回收该对象。当然啦，解释器依旧为回收对象这个过程提供了一个宏**_Py_Dealloc**，我们之前的文章中也说过了。

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

_Py_Dealloc(op)会调用op对应的类型对象中的析构函数，同时将op自身作为参数传递进去，表示将op指向的对象回收。

而PyFloat_Type中的**tp_dealloc**成员被初始化为**float_dealloc**，所以析构函数最终执行的是**float_dealloc**，在该函数内部我们会清晰地看到一个浮点数被销毁的全部过程。关于它的源代码，我们会在介绍缓存池的时候细说。

总结一下的话，浮点数对象从创建到销毁整个生命周期所涉及的关键函数、宏、调用关系可以如下图所示：



我们看到通过类型对象调用的方式来创建实例对象，最终也是要走**Python/C API**的，因此肯定没有直接通过**Python/C API**创建的方式快，因为前者多了几个步骤。

> 如果是float(3.14)，那么最终会调用PyFloat_FromDouble(3.14)；如果是float("3.14")，那么最终会调用PyFloat_FromString("3.14")。
> 所以调用类型对象的时候，会先兜个圈子再去使用Python/C API，肯定没有直接使用Python/C API的效率高，也就是说直接使用f=3.14这种方式是最快的。对于其它对象也是同理，当然大部分情况下我们也都是使用Python/C API来创建的。以列表为例，比起**list()**，我们更习惯使用**[]**

小结

以上就是浮点数在底层的创建和销毁，下一次我们就来聊一聊浮点数的缓存池，我们说一些占用内存小的对象在被销毁时，不会释放所占的内存，而是缓存起来。当下一次再使用的时候，直接从缓存池中获取，从而避免了频繁和内核打交道。

收录于合集 #CPython 97

‹ 上一篇
《源码探秘 CPython》9. 浮点数的缓存池机制

下一篇 ›
《源码探秘 CPython》7. 对象的引用计数，对象何时被回收?