《源码探秘 CPython》2. 解密 PyObject、PyVarObject、PyTypeObject

原创 古明地觉 古明地觉的编程教室 2022-01-03 16:34

楔子

上一篇文章说到，面向对象理论中的**类**和**对象**这两个概念在 Python 内部都是通过**对象**实现的。类是一种对象，称为**类型对象**，类实例化得到的也是对象，称为**实例对象**。

并且根据对象的不同特点还可以进一步分类：

- 可变对象：对象创建之后可以本地修改
- 不可变对象：对象创建之后不可以本地修改
- 定长对象：对象所占用的内存大小固定
- 不定长对象：对象所占用的内存大小不固定

但是对象在 Python 的底层是如何实现的呢？我们知道标准的 Python 解释器是 C 语言实现的 CPython（我们这里所研究的），但 C 并不是一个面向对象的语言，那么它是如何实现 Python 的面向对象的呢？

首先对于人的思维来说，对象是一个比较形象的概念，但对于计算机来说，对象却是一个抽象的概念。它并不能理解这是一个整数，那是一个字符串，计算机所知道的一切都是字节。通常的说法是：对象是**数据以及基于这些数据所能进行的操作的集合**。在计算机中，一个对象实际上就是一片被分配的内存空间，这些内存可能是连续的，也可能是离散的。

而 Python 的任何对象在 C 中都对应一个结构体实例，在 Python 中创建一个对象，等价于在 C 中创建一个结构体实例。所以 Python 的对象，其本质上就是 C 的 malloc 函数为结构体实例在堆区申请的一块内存。

下面我们就来分析一下 Python 的对象在 C 中是如何实现的，究竟生得一副什么模样，是三头六臂还是烈焰红唇。而第一步，就是下面要介绍的 PyObject。

实现对象机制的基石：PyObject

Python 中一切皆对象，而所有的对象都拥有一些共同的信息（也叫头部信息），这些信息就在 PyObject 中，PyObject 是 Python 整个对象机制的核心，我们来看看它的定义（我们整个系列的 CPython 都是 3.8 .0 版本的）：

```
//Include/object.h
typedef struct _object {
    _PyObject_HEAD_EXTRA
    Py_ssize_t ob_refcnt;
    struct _typeobject *ob_type;
} PyObject;
```

以上便是 PyObject 的内部信息，我们先来看看 `_PyObject_HEAD_EXTRA`，这是一个宏，如果将其展开的话：

```
#ifdef Py_TRACE_REFS

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

#
```

```
#define _PyObject_EXTRA_INIT 0, 0,

#else
#define _PyObject_HEAD_EXTRA
#define _PyObject_EXTRA_INIT
#endif
```

那么这个宏是做什么的呢？这个宏是用来实现一个名叫 refchain 的"双向链表"的，Python 会将程序中创建的所有对象都放入到这个双向链表中，用于跟踪所有活跃的堆对像。每一个对象都指向了它的前一个对象和后一个对象，如果是第一个对象，那么它的前继节点为 NULL；如果是最后一个节点，那么它的后继节点为 NULL。不过这个宏仅仅是在 debug 下有用，所以我们目前不需要管这个宏。

我们的重心是：ob_refcnt 和 ob_type。

ob_refcnt：引用计数

ob_refcnt 表示对象的引用计数，当一个对象被引用时，那么 ob_refcnt 会自增 1；引用解除时，ob_refcnt 自减1。而一旦对象的引用计数为 0 时，那么这个对象就会被回收。

那么在哪些情况下，引用计数会加 1 呢？哪些情况下，引用计数会减1呢？

导致引用计数加 1 的情况：

- 对象被创建：比如 `name = "古明地觉"`，此时对象就是 `"古明地觉"` 这个字符串，创建成功时它的引用计数为 1
- 变量传递使得对象被新的变量引用：比如 `name2 = name`
- 引用该对象的某个变量作为参数传到一个函数或者类中：比如`func(name)`
- 引用该对象的某个变量作为元组、列表、集合等容器的一个元素：比如`lst = [name]`

导致引用计数减1的情况：

- 引用该对象的变量被显示的销毁：`del name`
- 对象的引用指向了别的对象：`name = ""`
- 引用该对象的变量离开了它的作用域，比如函数的局部变量在函数执行完毕的时候会被销毁
- 引用该对象的变量所在的容器被销毁，或者被从容器里面删除

所以我们使用 **del 变量** 的时候，并不是删除变量指向的对象，我们没有这个权力。del 只是使对象的引用计数减一，至于对象到底删不删是解释器判断引用计数是否为 0 决定的。为 0 就删，不为 0 就不删，就这么简单。

另外 ob_refcnt 的类型是 Py_ssize_t，在 64 位机器上直接把这个类型看成 long 即可，因此一个对象的引用计数不能超过 long 所表示的最大范围。但是显然，如果不是吃饱了撑的写恶意代码，是不可能超过这个范围的。

ob_type：类型指针

我们说一个对象是有类型的，类型对象描述实例对象的数据和行为，而 ob_type 存储的便是对应类型对象的指针，所以类型对象在底层对应的是 struct _typeobject 实例。从这里我们可以看出，所有的类型对象在底层都是由同一个结构体实例化得到的，因为 PyObject 是所有的对象共有的，它们的 ob_type 指向的都是 struct _typeobject。

> 所以不同的实例对象对应不同的结构体，但是类型对象对应的都是同一个结构体。

因此我们看到 PyObject 的定义非常简单，就是一个引用计数和一个类型指针，所以 Python 中的任意对象都必有**引用计数**和**类型**这两个属性。

实现变长对象机制的基石：PyVarObject

PyObject 是所有对象的核心，它包含了所有对象都共有的信息，但是还有那么一个属性虽然不是每个对象都有，但至少有一大半的对象会有，能猜到是什么吗？

之前说过，Python 中的对象根据所占的内存是否固定，可以分为定长对象和变长对象，而变长对象显然有一个长度的概念，比如字符串、列表、元组等等。即便是相同的实例对象，但是长度不同，所占的内存也是不同的。比如：字符串内部有多少个字符、元组、列表内部有多少个元素，显然这里的**多少**也是 Python 中很多对象的共有特征，虽然不像引用计数和类型那样是每个对象都必有的，但也是绝大部分对象所具有的。

所以针对变长对象，Python 底层也提供了一个结构体，因为 Python 里面很多都是变长对象。

```
//Include/object.h
typedef struct {
    PyObject ob_base;
    Py_ssize_t ob_size;
} PyVarObject;
```

所以我们看到 PyVarObject 实际上是 PyObject 的一个扩展，它在 PyObject 的基础上提供了一个 ob_size 字段，用于记录内部的元素个数。比如列表，列表（PyListObject 实例）中的 ob_size 维护的就是列表的元素个数，插入一个元素，ob_size 会加1，删除一个元素，ob_size 会减1。

我们使用 len 获取列表的元素个数是一个时间复杂度为 O(1) 的操作，因为 ob_size 始终和内部的元素个数保持一致，使用 len 获取元素个数的时候会直接访问 ob_size。

因此在 Python 中，所有的变长对象都拥有 PyVarObject，而所有的对象都拥有 PyObject，这就使得在 Python 中，对**"对象"**的引用变得非常统一。我们只需要一个 PyObject * 就可以引用任意一个对象，而不需要管这个对象实际是一个什么样的对象。所以在 Python 中，所有的变量、以及容器内部的元素，本质上都是一个 PyObject *。而在操作变量的时候，也要先根据 ob_type 成员判断对象的类型，然后再寻找该对象具有的方法，这也是 Python 效率慢的原因之一。

由于**PyObject**和**PyVarObject**要经常被使用，所以 Python 提供了两个宏，方便定义。

```
#define PyObject_HEAD           PyObject ob_base;
#define PyObject_VAR_HEAD      PyVarObject ob_base;
```

比如定长对象浮点数，在底层对应的结构体为 PyFloatObject，只需在头部 PyObject 的基础上再加上一个 double 即可。

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

而对于变长对象列表，在底层对应的结构体是 PyListObject，所以它需要在 PyVarObject 的基础上再加上一个指向数组的二级指针和一个容量。

```
typedef struct {
    PyObject_VAR_HEAD
    PyObject **ob_item;
    Py_ssize_t allocated;
} PyListObject;
```

这上面的每一个成员都代表什么，我们之前已经分析过了。ob_item 就是指向指针数组的二级指针，而 allocated 表示已经分配的容量，一旦添加元素的时候发现 ob_size 自增 1 之后会大于 allocated，那么解释器就会对 ob_item 指向的指针数组进行扩容了。更准确的说，是申请一个容量更大的数组，然后将原来指向的指针数组内部的元素按照顺序一个一个地拷贝到新的数组里面去，并让 ob_item 指向新的数组，这一点在分析 PyListObject 的时候会细说。所以我们看到列表在添加元素的时候，地址是不会改变的，即使容量不够了也没有关系，直接让 ob_item 指向新的数组就好了，至于 PyListObject 对象本身的地址是不会变化的。

实现类型对象机制的基石：PyTypeObject

通过 PyObject 和 PyVarObject，我们看到了 Python 中所有对象的共有信息以及变长对象的共有信息。对于任何一个对象，不管它是什么类型，内部必有引用计数（ob_refcnt）和类型指针（ob_type）；对于任意一个变长对象，不管它是什么类型，除了引用计数和类型指针之外，内部还有一个表示元素个数的 ob_size。

显然目前是没有什么问题，一切都是符合我们的预期的，但是当我们顺着时间轴回溯的话，就会发现端倪。比如：

- 当在内存中创建对象、分配空间的时候，解释器要给该对象分配多大的空间？显然不能随便分配，那么该对象的内存信息在什么地方？
- 一个对象是支持相应的操作的，解释器怎么判断该对象支持哪些操作呢？再比如一个整数可以和一个整数相乘，但是一个列表也可以和一个整数相乘，即使是相同的操作，但由不同类型的对象执行也会有不同的结果，那么此时解释器又是如何进行区分的？

想都不用想，这些信息肯定都在对象所对应的类型对象中。而且占用的空间大小实际上是对象的一个元信息，这样的元信息和其所属类型是密切相关的，因此它一定会出现在与之对应的类型对象当中。至于支持的操作就更不用说了，我们平时自定义类的时候，方法都写在什么地方，显然都是写在类里面，因此一个对象支持的操作也定义在类型对象当中。

而将一个对象和其类型对象关联起来的，毫无疑问正是该对象内部的**PyObject**中的**ob_type**，也就是类型指针。我们通过对象的**ob_type**成员即可获取类型对象的指针，通过该指针可以获取存储在类型对象中的某些元信息。

下面我们来看看类型对象在底层是怎么定义的：

```
// 类型对象对应的结构体
typedef struct _typeobject {
    PyObject_VAR_HEAD
    const char *tp_name;
    Py_ssize_t tp_basicsize, tp_itemsize;
    destructor tp_dealloc;
    printfunc tp_print;
    getattrfunc tp_getattr;
    setattrfunc tp_setattr;
    PyAsyncMethods *tp_as_async; /* formerly known as tp_compare (Python
 2)
                                    or tp_reserved (Python 3) */
    reprfunc tp_repr;
    PyNumberMethods *tp_as_number;
    PySequenceMethods *tp_as_sequence;
    PyMappingMethods *tp_as_mapping;
    hashfunc tp_hash;
    ternaryfunc tp_call;
    reprfunc tp_str;
    getattrofunc tp_getattro;
    setattrofunc tp_setattro;
    PyBufferProcs *tp_as_buffer;
    unsigned long tp_flags;

    const char *tp_doc; /* Documentation string */


    traverseproc tp_traverse;

    inquiry tp_clear;
    richcmpfunc tp_richcompare;

    Py_ssize_t tp_weaklistoffset;
```

```
34
35      getiterfunc tp_iter;
36      iternextfunc tp_iternext;
37      struct PyMethodDef *tp_methods;
38      struct PyMemberDef *tp_members;
39      struct PyGetSetDef *tp_getset;
40      struct _typeobject *tp_base;
41      PyObject *tp_dict;
42      descrgetfunc tp_descr_get;
43      descrsetfunc tp_descr_set;
44      Py_ssize_t tp_dictoffset;
45      initproc tp_init;
46      allocfunc tp_alloc;
47      newfunc tp_new;
48      freefunc tp_free; /* Low-level free-memory routine */
49      inquiry tp_is_gc; /* For PyObject_IS_GC */
50      PyObject *tp_bases;
51      PyObject *tp_mro; /* method resolution order */
52      PyObject *tp_cache;
53      PyObject *tp_subclasses;
54      PyObject *tp_weaklist;
55      destructor tp_del;
56      unsigned int tp_version_tag;
57
58      destructor tp_finalize;
59
60  #ifdef COUNT_ALLOCS
61      Py_ssize_t tp_allocs;
62      Py_ssize_t tp_frees;
63      Py_ssize_t tp_maxalloc;
64      struct _typeobject *tp_prev;
65      struct _typeobject *tp_next;
66  #endif
67  } PyTypeObject;
    #endif
```

类型对象在底层对应的是 `struct _typeobject`，当然还有一个别名叫 **PyTypeObject**，它里面的成员非常非常多，我们一会挑几个重要的说，因为有一部分成员并不是那么重要，我们在后续会慢慢说。

目前我们了解到 Python 中的类型对象在底层就是一个**PyTypeObject**实例，它保存了实例对象的元信息，描述对象的类型。

> Python 中的实例对象在底层对应不同的结构体实例，而类型对象则是对应同一个结构体实例，换句话说无论是 int、str、dict 等等等等，它们在 C 的层面都是由 PyTypeObject 这个结构体实例化得到的，只不过成员的值不同，PyTypeObject 这个结构体在实例化之后得到的类型对象也不同。

我们看一下 PyTypeObject 内部几个非常关键的成员：

- `PyObject_VAR_HEAD`：我们说这是一个宏，对应一个PyVarObject，所以类型对象是一个变长对象。而且类型对象也有引用计数和类型，这与我们前面分析的是一致的
- `tp_name`：类型的名称，而这是一个 `char *`，显然它可以是 `int、str、dict` 之类的
- `tp_basicsize，tp_itemsize`：创建对应实例对象时所需要的内存信息
- `tp_dealloc`：其实例对象执行析构函数时所作的操作
- `tp_print`：其实例对象被打印时所作的操作
- `tp_as_number`：其实例对象为数值时，所支持的操作。这是一个结构体指针，指向的结构体中的每一个成员都是一个函数指针，其函数就是整型对象可以执行的操作，比如：四则运算、左移、右移、取模等等
- `tp_as_sequence`：其实例对象为序列时，所支持的操作。同样是一个结构体指针
- `tp_as_mapping`：其实例对象为映射时，所支持的操作。也是一个结构体指针
- `tp_base`：继承的基类

我们暂时就挑这么几个，事实上从名字上你也能看出来这每一个成员代表的含义。而且

这里面的成员虽然多，但并非每一个类型对象都具备，比如 int 类型它就没有 tp_as_sequence 和 tp_as_mapping，所以 int 类型的这两个成员的值都是 0。

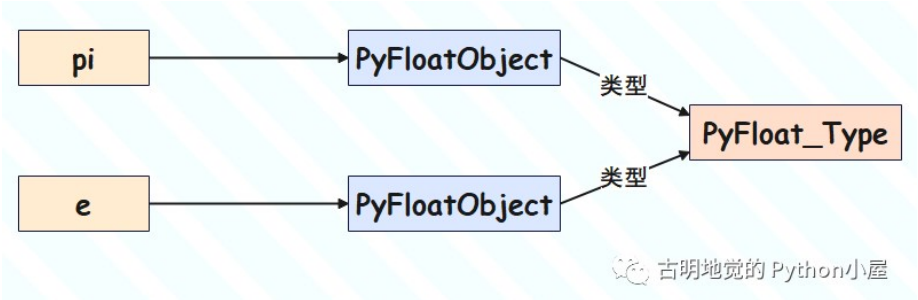
具体的我们就在分析具体的类型对象的时候再说吧，然后先来看看 Python 对象在底层都叫什么名字吧。

- 整数 -> PyLongObject结构体实例，int -> PyLong_Type(PyTypeObject结构体实例)
- 字符串 -> PyUnicodeObject结构体实例，str -> PyUnicode_Type(PyTypeObject结构体实例)
- 浮点数 -> PyFloatObject结构体实例，float -> PyFloat_Type(PyTypeObject结构体实例)
- 复数 -> PyComplexObject结构体实例，complex -> PyComplex_Type(PyTypeObject结构体实例)
- 元组 -> PyTupleObject结构体实例，tuple -> PyTuple_Type(PyTypeObject结构体实例)
- 列表 -> PyListObject结构体实例，list -> PyList_Type(PyTypeObject结构体实例)
- 字典 -> PyDictObject结构体实例，dict -> PyDict_Type(PyTypeObject结构体实例)
- 集合 -> PySetObject结构体实例，set -> PySet_Type(PyTypeObject结构体实例)
- 不可变集合 -> PyFrozenSetObject结构体实例，frozenset -> PyFrozenSet_Type(PyTypeObject结构体实例)
- 元类：PyType_Type(PyTypeObject结构体实例)

所以 Python 中的对象在底层的名字都遵循一定的标准，包括解释器提供的 Python/C API 也是如此。

下面以浮点数为例，考察一下类型对象和实例对象之间的关系。浮点类型我们说底层对应的是 PyTypeObject 的实例 PyFloat_Type，并且浮点类型是全局唯一的；而浮点数则是 PyFloatObject 实例，浮点数可以有任意个，比如：圆周率 pi 是一个、自然对数 e 又是一个。

```
>>> float
<class 'float'>
>>> pi = 3.14
>>> e = 2.71
>>>
>>> type(pi) is type(e) is float
True
>>>
```



两个变量均指向了浮点数（PyFloatObject 结构体实例），除了公共头部字段 ob_refcnt 和 ob_type，专有字段 ob_fval 保存了对应的数值；浮点类型 float 则对应 PyTypeObject 结构体实例（PyFloat_Type），保存了类型名、浮点数的内存分配信息以及相关操作。而将这两者关联起来的就是 ob_type 这个类型指针，它位于 PyObject 中，是所有对象共有的，而 Python 便是根据这个 ob_type 来判断该对象的类型，进而获取该对象的元信息。

> 我们说变量只是一个指针，那么 int、float、dict 这些是不是变量，显然是的，函数和类也是一个变量，所以它们在底层也是一个指针。只不过这些变量是内置的，直接指向了具体的

> PyTypeObject 实例。并且为了方便，有时我们用 int、float 等等，来代指指向的对象。比如：float 指向了底层的 PyFloat_Type，所以它其实是 PyFloat_Type 的指针，但为了表述方便我们会直接用 float 来代指 PyFloat_Type。
>
> 而且类型对象在解释器启动的时候就已经是创建好了的，不然的话我们怎么能够直接用呢？

我们来看一下 float 对应的类型对象在底层是怎么定义的吧。

```
1  // Object/floatobject.c
2  PyTypeObject PyFloat_Type = {
3      PyVarObject_HEAD_INIT(&PyType_Type, 0)
4      "float",
5      sizeof(PyFloatObject),
6      0,
7      (destructor)float_dealloc,                  /* tp_dealloc */
8
9      // ...
10     (reprfunc)float_repr,                       /* tp_repr */
11
12     // ...
13 };
```

我们看到 PyFloat_Type 在源码中就直接被创建了，这是必须的，否则我们就没有办法直接访问 float 这个变量了，然后先看第 4 行，我们看到 tp_name 被初始化成了 "float"，也就是类名；第 5 行表示实例对象所占的字节数，我们看到就是一个 PyFloatObject 实例所占的内存大小，并且显然这个值是不会变的，说明无论创建多少个实例对象，它们的大小都是不变的，这也符合我们之前的结论，都是 24 字节。

再往下就是一些各种操作对应的函数指针，最后我们来看一下第 3 行，显然它接收的是一个 PyVarObject，PyVarObject_HEAD_INIT 是一个宏，在底层长这样。

```
1  // Include/object.h
2  #define PyObject_HEAD_INIT(type)        \
3      { _PyObject_EXTRA_INIT              \
4      1, type },
5
6
7  #define PyVarObject_HEAD_INIT(type, size)       \
8      { PyObject_HEAD_INIT(type) size },
```

先看 PyObject_HEAD_INIT，里面的 _PyObject_EXTRA_INIT 是用来实现 refchain 这个双向链表的，我们不需要管。里面的 1 指的是引用计数，我们看到刚创建的时候默认是设置为 1 的，至于 type 就是该类型对象的类型（ob_type）了，这个是作为宏的参数传进来的；而 PyVarObject_HEAD_INIT，则是在 PyObject_HEAD_INIT 的基础之上，增加了一个 size，显然我们从名字也能看出来这个 size 是什么，就是 PyVarObject 内部的 ob_size。

然后我们回过头再看一下 `PyFloat_Type` 的定义，我们看到它的 ob_type 被设置成了 &PyType_Type，说明 float 的类型是 type。

```
1  >>> float.__class__
2  <class 'type'>
3  >>> # 显然这是符合我们的预期的
```

而且所有的类型对象（还有元类）在底层都被定义成了静态的全局变量，因为它们的生命周期是伴随着整个解释器的，并且在任意地方都可以访问。

小结

PyObject 是 Python 对象的核心，因为 Python 对象在 C 的层面就是一个结构体，所有的结构体都嵌套了 PyObject 这个结构体，而 PyObject 内部有引用计数和类型这两

个成员，因此我们可以肯定的说 Python 的任何一个对象都有引用计数和类型这两个属性。

而大部分的对象都具有长度的概念，所以 PyObject 再加上长度就诞生出了 PyVarObject，它在 PyObject 的基础上多了一个 ob_size 成员，用于描述对象的长度。比如字符串内部的 ob_size 维护的是字符串的字符个数，元组、列表、字典等等，其内部的 ob_size 维护的是存储的元素个数，所以使用 len 函数获取对象长度是一个 O(1) 的操作。

最后我们提到了 PyTypeObject，它实例化之后就得到了类型对象。我们说不同的实例对象在底层对应不同的结构体实例（比如浮点数对应PyFloatObject、列表对应 PyListObject）；但所有的类型对象在底层对应的都是相同的结构体（PyTypeObject）实例，比如 float、str、dict 等等都是由 PyTypeObject 实例化得到的。

而将实例对象和类型对象关联起来的，就是实例对象的 ob_type 成员，我们后面在分析具体的内置对象的时候就会看到这一点。

收录于合集 #CPython 97

‹ 上一篇
《源码探秘 CPython》3. type 和 object 的恩怨纠葛

下一篇 ›
《源码探秘 CPython》1. Python中一切皆对象，这里的对象究竟是什么？解密Pytho...

文章已于2022-03-29修改