

# Python 源码学习

KenshinLiu

Last Update: 2022 年 4 月 11 日

## 编译 Python

Listing 1: 编译使用的操作系统和 Python 版本信息

```
1 $ sw_vers
2 ProductName:    macOS
3 ProductVersion: 12.3
4 BuildVersion:   21E230
5
6 $ git branch
7 * main
8
9 $ git log -n 1
10 commit df9f7597559b6256924fcd3a1c3dc24cd5c5edaf (HEAD -> main, origin/main, origin/HEAD)
11 Author: Inada Naoki <songofacandy@gmail.com>
12 Date:   Tue Mar 1 10:27:20 2022 +0900
13     compiler: Merge except_table and cnotab (GH-31614)
```

Listing 2: 编译命令

```
1 # debug 模式编译
2 $ ./configure --with-pydebug
3 $ make
```

Listing 3: doxygen 生成调用关系

```
1 $ doxygen -g doxygen.config
2 $ doxygen doxygen.config
```

## Object

Python 所有对象的基础都是 Object. Object 在堆上进行分配。每个 Object 上都有一个表示引用计数的变量，当引用计数变为 0 之后，Object 会被回收。Object 上有一个表示类型的变量，每个 Object 的类型在 Object 被创建出来的时候就被固定，之后不会在改变。这个类型也是使用一个对象来进行表示，对象内部包含一个指针，指针指向被称为类型对象。这个类型对象包含一个表示类型的指针，指向自身。

Object 一旦分配之后，分配的内存大小以及内存地址都不会再发生改变。如果 Object 需要保存可变大小的数据，那么 Object 可以持有指向可变大小数据的一个指针。

Python 中的对象总是通过 PyObject\* 类型的指针进行访问。PyObject 结构体只包含引用计数和一个类型指针。Python 中的对象的内存地址的前部都是一个 PyObject 结构体，通过这个 PyObject 结构体，就可以知道一个对象的引用计数和真实的类型。

Listing 4: PyObject 定义

```
1 struct _object {
2     PyObject_HEAD_EXTRA // 这个字段用来编译一个 debug 版本，不用关注
3     Py_ssize_t ob_refcnt;
4     PyTypeObject *ob_type;
5 };
6 typedef struct _object PyObject;
```

PyObject 只能表示单个的对象，如果需要表示含有多个对象的结构，需要用到另一个结构体 PyVarObject。其中的 ob\_size 字段表示对象的数量。

Listing 5: PyVarObject 定义

```
1 typedef struct {
2     PyObject ob_base;
3     Py_ssize_t ob_size; /* Number of items in variable part */
4 } PyVarObject;
```

上面提到了 Object 上有一个表示类型的变量。这个变量就是 PyObject 中的 ob\_type 字段，类型为 PyTypeObject。Object 真正的类型就存储在这个结构体中。

Listing 6: PyTypeObject 的定义

```
1 struct _typeobject {
2     PyObject_VAR_HEAD
3     const char *tp_name; /* For printing, in format "<module>.<name>" */
4     // ... ..
5 };
6
7 typedef struct _typeobject PyTypeObject;
```

定义中可以看到有一个 tp\_name 字段，这个字段以字符串形式存储了类型名。

在定义的开始位置还可以看到一个 PyObject\_VAR\_HEAD，也就是上面提到的 PyVarObject，这说明 PyTypeObject 对象自身也有一个指针指向一个 PyTypeObject 类型的对象。那么这里就会有一个疑问：其他的 Object 的类型名存储在 PyTypeObject 的实例中，那么 PyTypeObject 实例自身的类型又是什么？存储在什么位置？要解答这个问题需要看一下 PyTypeObject 的实例化过程，下面以 Python 中最基础的类型类 class type 来举例子。

Listing 7: PyTypeObject 实例化

```

1 PyTypeObject PyType_Type = {
2     PyVarObject_HEAD_INIT(&PyType_Type, 0) // 对象的指针指向自身
3     "type",                                /* tp_name */
4     // ... ...
5 };

```

PyType\_Type 就是 class type 的实例，在实例化的代码中可以看到 PyType\_Type 对象的 PyObject\_VAR\_HEAD 字段设置为了它自身，tp\_name 字段设置为"type"。这样就成功的让类的继承体系中的每个类型都有了一个类型定义。

使用如下代码可以进行验证。

Listing 8: 修改 typeobject.c

```

1 PyTypeObject PyType_Type = {
2     PyVarObject_HEAD_INIT(&PyType_Type, 0) // 对象的指针指向自身
3     "PyTypeObject", // tp_name 从 "type" 修改为 "PyTypeObject"
4     // ... ...
5 };

```

Listing 9: 创建 test.py

```

1 class A(object):
2     pass
3
4 class B(A):
5     pass
6
7 a = A()
8 b = B()
9
10 print(A.__class__)
11 print(B.__class__)
12
13 print(a.__class__)
14 print(b.__class__)

```

重新编译修改后的 Python，并使用生成的 Python 解释器执行 test.py，可以看到如下输出。

Listing 10: test.py 运行输出

```

1 # 修改 Python 源码前, 命令 python test.py 的输出
2 <class 'type'>
3 <class 'type'>
4 <class '__main__.A'>
5 <class '__main__.B'>
6
7 # 修改 Python 源码后, 命令 python test.py 的输出
8 <class 'PyTypeObject'>
9 <class 'PyTypeObject'>
10 <class '__main__.A'>
11 <class '__main__.B'>

```

从输出中可以看出, 类型 A 和类型 B 的类型都变成了修改后的 “PyTypeObject”, 但是对象 a 和对象 b 的类型不变。

再看一下 Python 中 int 类型的实例化。

Listing 11: int 类型类的实例化

```

1 PyTypeObject PyLong_Type = {
2     PyVarObject_HEAD_INIT(&PyType_Type, 0) // class int 的类型也是 class type
3     "int",                                     /* tp_name */
4     offsetof(PyLongObject, ob_digit),         /* tp_basicsize */
5     sizeof(digit),
6 }

```

## 对象的析构

PyObject 是根据 ob\_refcnt 来判断是否需要释放一个对象, 当对象的引用增加 1, ob\_refcnt 增加 1, 当对象的引用减少 1, ob\_refcnt 减少 1, 在 ob\_refcnt 变为 0 时, 对象被释放。但是存在一个特例, 所有的类型对象也有一个 ob\_refcnt 变量, 但是永远也不会释放。

至于不会释放的原理, 是因为它的释放函数被设置为了空。

一个 PyObject 对象在 ob\_refcnt 到达 0 的时候, 会被调用 Py\_DECREF 宏进行对象的释放, 具体代码如下

Listing 12: PyObject 释放过程

```

1 static inline void Py_DECREF(PyObject *op) {
2     // ... ..
3     if (--op->ob_refcnt != 0) {

```

```

4         // ... ...
5     }
6     else {
7         _Py_Dealloc(op);
8     }
9 #endif
10 }
11
12 void
13 _Py_Dealloc(PyObject *op)
14 {
15     destructor dealloc = Py_TYPE(op)->tp_dealloc;
16     (*dealloc)(op); // 在此处调用对象释放函数
17 }

```

但是对于类型对象来说，tp\_dealloc 函数指针都被设置为了 0，所以类型对象永远不会被释放。虽然 tp\_dealloc 被设置为 0，并不代表类型类型的释放函数不会被调用，相反，这个函数的调用其实很频繁。可以修改如下位置的源码进行验证。

Listing 13: 修改 PyLong\_Type 源码

```

1 #include <stdio.h>
2 int long_tp_dealloc(PyObject* obj) {
3     printf("this is PyLong_Type tp_dealloc function");
4 }
5
6 PyTypeObject PyLong_Type = {
7     PyVarObject_HEAD_INIT(&PyType_Type, 0)
8     "int", /* tp_name */
9     offsetof(PyLongObject, ob_digit), /* tp_basicsize */
10    sizeof(digit), /* tp_itemsize */
11    // 修改此处源码，从 0 改为上面添加的 long_tp_dealloc 函数指针
12    long_tp_dealloc, /* tp_dealloc */
13    0, /* tp_vectorcall_offset */
14 }

```

修改完成重新编译之后，使用如下代码进行验证。

Listing 14: test.py

```

1 a = 10

```

使用编译出的新 Python 解释器执行 test.py 即可发现有大量的 “this is PyLong\_Type tp\_dealloc function” 句子输出，说明此函数被频繁调用。

## 备注

上面的代码分析步骤主要是跟着《Python 源码剖析》，对照着 Python 3.11 代码进行学习，引用了部分书中内容。