

## 《源码探秘 CPython》69. 给类型对象设置类型和基类信息

原创 古明地觉 古明地觉的编程教室 2022-04-15 08:30

收录于合集

#CPython

97个 >



微信扫一扫  
关注该公众号

在上一篇文章中我们说道，内置类对象虽然在底层静态定义好了，但是还不够完善。解释器在启动之后还要再打磨一下，然后才能得到我们平时使用的类型对象，而这个过程被称为类型对象的初始化。

类型对象的初始化，是通过 `PyType_Ready` 函数实现的，我们来看一下，它位于 `Objects/typeobject.c` 中。另外由于初始化这部分内容比较多，接下来我们准备用三篇文章去介绍它。

```
1 int
2 PyType_Ready(PyTypeObject *type)
3 {
4     //这里的参数显然是类型对象
5     //以 <class 'type'> 为例
6
7     //dict:属性字典
8     //bases:继承的所有基类, 即 __bases__
9     PyObject *dict, *bases;
10
11     //base:继承的第一个基类, 即 __base__
12     PyTypeObject *base;
13     Py_ssize_t i, n;
14
15     //.....
16     //.....
17
18     //获取类型对象中 tp_base 成员指定的基类
19     base = type->tp_base;
20     if (base == NULL && type != &PyBaseObject_Type) {
21         //如果基类为空、并且该类本身不是object
22         //那么将该类的基类设置为 object、即 &PyBaseObject_Type
23         //所以一些类型对象在底层定义的时候, tp_base 成员为空
24         //因为tp_base是在这里、也就是初始化的时候进行设置的
25         base = type->tp_base = &PyBaseObject_Type;
26         Py_INCREF(base);
27     }
28
29     //如果基类不是NULL, 也就是指定了基类
30     //但是基类的属性字典是NULL
31     if (base != NULL && base->tp_dict == NULL) {
32         //说明该类的基类尚未初始化, 那么会先对基类进行初始化
33         //注意这里的 tp_dict, 它表示每个类都会有的属性字典
34         //而属性字典是否为 NULL, 是类型对象是否初始化完成的重要标志
35         if (PyType_Ready(base) < 0)
36             goto error;
37     }
38
39     //如果该类型对象的 ob_type 为空, 但是基类不为空
40     //那么将该类型对象的 ob_type 设置为基类的 ob_type
41     //为什么要做这一步, 我们后面会详细说
42     if (Py_TYPE(type) == NULL && base != NULL)
43         Py_TYPE(type) = Py_TYPE(base);
44
45     //获取 __bases__, 检测是否为空
46     bases = type->tp_bases;
47     //如果为空, 则根据 __base__ 进行设置
```

```

48     if (bases == NULL) {
49         //如果 base 也为空, 那么 bases 就是空元祖
50         //而base如果为空了, 说明当前的类对象一定是object
51         if (base == NULL)
52             bases = PyTuple_New(0);
53         //如果 base 不为空, 那么 bases 就是 (base,)
54         else
55             bases = PyTuple_Pack(1, base);
56         if (bases == NULL)
57             goto error;
58         //设置 tp_bases
59         type->tp_bases = bases;
60     }
61
62     //设置属性字典, 后续再聊
63     dict = type->tp_dict;
64     if (dict == NULL) {
65         dict = PyDict_New();
66         if (dict == NULL)
67             goto error;
68         type->tp_dict = dict;
69     }
70     //.....
71 }

```

对于指定了tb\_base的类对象，当然就使用指定的基类，而对于没有指定tp\_base的类对象，虚拟机将为其指定一个默认的基类：&PyBaseObject\_Type，也就是 Python 的 object。

现在我们看到 PyType\_Type 的 tp\_base 指向了 PyBaseObject\_Type，这在Python中体现的就是 type 继承自 object、或者说 object 是 type 的父类。但是所有的类的 ob\_type 又都指向了 PyType\_Type，包括 object，因此我们又说 type 是包括 object 在内的所有类的类(元类)。

而在获得了基类之后，会判断基类是否被初始化，如果没有，则需要先对基类进行初始化。可以看到，判断初始化是否完成的条件是tp\_dict是否为NULL，这符合之前的描述。对于内置类对象来说，在解释器启动的时候，就已经作为全局对象存在了，所以它们的初始化不需要做太多工作，只需小小的完善一下即可，比如设置基类、类型、以及对 tp\_dict 进行填充。

在基类设置完毕后，会继续设置 ob\_type，这个ob\_type就是 \_\_class\_\_ 返回的类型对象。

首先 PyType\_Ready 函数里面接收的是一个 PyTypeObject 对象，我们知道这个在 Python 中就是类对象。因此这里是设置这些类对象的 ob\_type，那么对应的显然就是元类 (metaclass)，我们自然会想象到Python的type。

而Py\_TYPE(type) = Py\_TYPE(base)这一行代码是把父类的ob\_type设置成了当前类的ob\_type，那么这一步的意义何在呢？我们使用Python来演示一下。

```

1  class MyType(type):
2      pass
3
4  class A(metaclass=MyType):
5      pass
6
7  class B(A):
8      pass
9
10 print(type(A)) # <class '__main__.MyType'>
11 print(type(B)) # <class '__main__.MyType'>

```

我们看到B继承了A，而A的类型是MyType，那么B的类型也成了MyType。也就是说 A 是由 XX 生成的，那么B在继承A之后，B 也会由 XX 生成，所以源码中的那一步就是用来做这件事情的。另外，这里之所以用 XX 代替，是因为 Python 里面不仅仅只有 type 是元类，那些继承了 type 的子类也可以是元类。

而且如果你熟悉 flask 的话，你会发现 flask 源码里面就有类似于这样的操作：

```
1 class MyType(type):
2
3     def __new__(mcs, name, bases, attrs):
4         # 关于第一个参数我们需要说一下
5         # 对于一般的类来说这里应该是cls
6         # 但我们这里是元类，所以应该用mcs，意思就是metaclass
7         # 我们额外设置一些属性吧，关于元类我们后续会介绍
8         # 虽然目前还没有看底层实现，但至少使用方法应该知道
9         attrs.update({"name": "古明地觉"})
10        return super().__new__(mcs, name, bases, attrs)
11
12 def with_metaclass(meta, bases=(object, )):
13     return meta("", bases, {})
14
15 class Girl(with_metaclass(MyType, (int,))):
16     pass
17
18 print(type(Girl)) # <class '__main__.MyType'>
19 print(getattr(Girl, "name")) # 古明地觉
20 print(Girl("123")) # 123
```

所以逻辑很清晰了，虚拟机就是将子类的metaclass设置为基类的metaclass。对于当前的 PyType\_Type来说，其metaclass就是object的metaclass，也是它自己。而在源码的 PyBaseObject\_Type中也可以看到，其ob\_type被设置成了 [&PyType\\_Type](#)。

```
1 if (Py_TYPE(type) == NULL && base != NULL)
2     Py_TYPE(type) = Py_TYPE(base);
```

tb\_base 和 ob\_type 设置完毕之后，会设置 tb\_bases。tb\_base 对应 \_\_base\_\_，tb\_bases 对应 \_\_bases\_\_，我们用 Python 演示一下，这两者的区别。

```
1 class A:
2     pass
3
4 class B(A):
5     pass
6
7 class C:
8     pass
9
10 class D(B, C):
11     pass
12
13 print(D.__base__) # <class '__main__.B'>
14 print(D.__bases__) # (<class '__main__.B'>, <class '__main__.C'>)
15
16 print(C.__base__) # <class 'object'>
17 print(C.__bases__) # (<class 'object'>,)
18
19 print(B.__base__) # <class '__main__.A'>
20 print(B.__bases__) # (<class '__main__.A'>,)

```

我们看到 D 同时继承多个类，那么 tp\_base 就是先出现的那个基类。而 tp\_bases 则是继承的所有基类，但是基类的基类是不会出现的，比如 object。对于 B 而言也是一样的。

然后我们看看 C，因为 C 没有显式地继承任何类，那么 tp\_bases 就是NULL。但是Python3 里面所有的类都默认继承了object，所以tp\_base就是object。而 tp\_bases，显然是 [\(object,\)](#)。

以上就是 tp\_base、ob\_type、tp\_bases 的设置，还是比较简单的，它们在设置完毕之后，就要对 tp\_dict 进行填充了。而填充 tp\_dict 是一个极其繁复的过程，我们下一篇文章再说。

收录于合集 #CPython 97

< 上一篇

《源码探秘 CPython》70. 类对象属性字典的填充

下一篇 >

《源码探秘 CPython》68. 深入 class

文章已于2022-06-25修改

喜欢此内容的人还喜欢

python 7天进阶之路-对象和json转换  
缪斯之子



[系列]微服务·深入理解 gRPC - Part2  
走向架构师的每一天



Abaqus python脚本开发 第三章 各类指令的方法对象变量 (3)  
山石结构

