



微信扫一扫
关注该公众号

收录于合集

#CPython

97个 >

楔子

在上一篇文章中，我们知道了对象是如何被创建的，主要有两种方式，一种是通过 **Python/C API**，另一种是通过**调用类型对象**。对于内置类型的实例对象而言，这两种方式都是支持的，比如列表，我们即可以通过 `[]` 创建，也可以通过 `list()`，前者是 **Python/C API**，后者是**调用类型对象**。

但对于自定义类的实例对象而言，我们只能通过**调用类型对象**的方式来创建。而一个对象如果可以被调用，那么这个对象就是 **callable**，否则就不是 **callable**。

而决定一个对象是不是 **callable**，就取决于其对应的类型对象中是否定义了某个方法。如果从 Python 的角度看的话，这个方法就是 `__call__`，从解释器角度看的话，这个方法就是 `tp_call`。

从 Python 的角度看对象的调用

调用 `int`、`str`、`tuple` 可以创建一个整数、字符串、元组，调用自定义的类也可以创建出相应的实例对象，说明类型对象是可调用的，也就是 **callable**。那么这些类型对象 (`int`、`str`、`tuple`、`class` 等等) 的类型对象 (`type`) 内部一定有 `__call__` 方法。

```
1 # int可以调用
2 # 那么它的类型对象、也就是元类(type)，内部一定有__call__方法
3 print(hasattr(type, "__call__")) # True
4
5 # 而调用一个对象，等价于调用其类型对象的 __call__ 方法
6 # 所以 int(3.14)实际就等价于如下
7 print(type.__call__(int, 3.14)) # 3
```

注意：这里描述的可能有一些绕，我们说 `int`、`str`、`float` 这些都是类型对象（简单来说就是**类**），而 `123`、`"你好"`、`3.14` 是其对应的实例对象，这些都没问题。但 `type` 不是类型对象，显然是的，虽然我们称呼它为**元类**，但它也是类型对象，如果 `print(type)` 显示的也是一个类。

那么相对 `type` 而言，`int`、`str`、`float` 是不是又成了实例对象呢？因为它们的类型是 `type`。

所以 `class` 具有**二象性**：

- 如果站在实例对象（如：`123`、`"satori"`、`[]`、`3.14`）的角度上，它是类型对象
- 如果站在 `type` 的角度上，它是实例对象

同理 `type` 的类型也是 `type`，那么 `type` 既是 `type` 的类型对象，`type` 也是 `type` 的实例对象。虽然这里描述的会有一些绕，但应该不难理解，并且为了避免后续的描述出现歧义，这里我们做一个申明：

- 整数、浮点数、字符串等等，我们称之为**实例对象**
- `int`、`float`、`str`、`dict`，以及我们自定义的类，我们称之为**类型对象**
- `type` 虽然也是类型对象，但我们称它为**元类**

所以 `type` 的内部有 `__call__` 方法，那么说明类型对象都是可调用的，因为调用类型对象就是调用 `type` 的 `__call__` 方法。而实例对象能否调用就不一定了，这取决于它的类型对象中是否定义了 `__call__` 方法，因为调用一个对象，本质上是执行其类型对象内部的 `__call__` 方法。

```

1 class A:
2     pass
3
4 a = A()
5 # 因为我们自定义的类 A 里面没有 __call__
6 # 所以 a 是不可以被调用的
7 try:
8     a()
9 except Exception as e:
10     # 告诉我们 A 的实例对象不可以被调用
11     print(e) # 'A' object is not callable
12
13 # 如果我们给 A 设置了一个 __call__
14 type.__setattr__(A, "__call__", lambda self: "这是__call__")
15 # 发现可以调用了
16 print(a()) # 这是__call__

```

我们看到这就是动态语言的特性，即便在类创建完毕之后，依旧可以通过`type`进行动态设置，而这在静态语言中是不支持的。所以`type`是所有类的元类，它控制了我们自定义类的生成过程，`type`这个古老而又强大的类可以让我们玩出很多新花样。

但是对于内置的类，`type`是不可以对其动态增加、删除或者修改属性的，因为内置的类在底层是静态定义好的。因为从源码中我们看到，这些内置的类、包括元类，它们都是`PyTypeObject`对象，在底层已经被声明为全局变量了，或者说它们已经作为静态类存在了。所以`type`虽然是所有类型对象的元类，但是只有在面对我们自定义的类，`type`才具有增删改的能力。

而且在上一篇文章中我们也解释过，Python 的动态性是解释器将字节码翻译成 C 代码的时候动态赋予的，因此给类动态设置属性或方法只适用于动态类，也就是在 py 文件中使用 class 关键字定义的类。

而对于静态类、或者编写扩展模块时定义的扩展类（两者是等价的），它们在编译之后已经是指向 C 一级的数据结构了，不需要再被解释器解释了，因此解释器自然也就无法在它们身上动手脚，毕竟彪悍的人生不需要解释。

```

1 try:
2     type.__setattr__(dict, "__call__", lambda self: "这是__call__")
3 except Exception as e:
4     print(e) # can't set attributes of built-in/extension type 'dict'

```

我们看到抛异常了，提示我们**不可以给内置/扩展类型dict设置属性**，因为它们绕过了解释器解释执行这一步，所以其属性不能被动态设置。

同理其实实例对象亦是如此，静态类的实例对象也不可以动态设置属性：

```

1 class Girl:
2     pass
3
4 g = Girl()
5 g.name = "古明地觉"
6 # 实例对象我们也可以手动设置属性
7 print(g.name) # 古明地觉
8
9 lst = list()
10 try:
11     lst.name = "古明地觉"
12 except Exception as e:
13     # 但是内置类型的实例对象是不可以的
14     print(e) # 'List' object has no attribute 'name'

```

可能有人奇怪了，为什么列表不行呢？答案是内置类型的实例对象没有`__dict__`属性字典，因为相关属性或方法底层已经定义好了，不可以动态添加。如果我们自定义类的时候

候，设置了 `__slots__`，那么效果和内置的类是相同的。

当然了，我们后面会介绍如何通过动态修改解释器来改变这一点，举个栗子，不是说静态类无法动态设置属性吗？下面我就来打自己脸：

```
1 import gc
2
3 try:
4     type.__setattr__(list, "ping", "pong")
5 except TypeError as e:
6     print(e) # can't set attributes of built-in/extension type 'list'
7
8 # 我们看到无法设置, 那么我们就来改变这一点
9 attrs = gc.get_referents(tuple.__dict__)[0]
10 attrs["ping"] = "pong"
11 print().ping) # pong
12
13 attrs["append"] = lambda self, item: self + (item,)
14 print(
15     ().append(1).append(2).append(3)
16 ) # (1, 2, 3)
```

我脸肿了。好吧，其实这只是我们玩的一个小把戏，当我们介绍整个 CPython 的时候，会来专门聊一聊如何动态修改解释器。比如：让元组变得可修改，让 Python 真正利用多核等等。

从解释器的角度看对象的调用

我们以内置类型 `float` 为例，我们说创建一个 `PyFloatObject`，可以通过 [3.14](#) 或者 `float(3.14)` 的方式。前者使用 [Python/C API](#) 创建，3.14 直接被解析为 C 一级数据结构，也就是 `PyFloatObject` 实例；后者使用类型对象创建，通过对 `float` 进行一个调用、将 3.14 作为参数，最终也得到指向 C 一级数据结构 `PyFloatObject` 实例。

Python/C API 的创建方式我们已经很清晰了，就是根据值来推断在底层应该对应哪一种数据结构，然后直接创建即可。我们重点看一下通过类型调用来创建实例对象的方式。

如果一个对象可以被调用，它的类型对象中一定要有 `tp_call` (更准确的说成员 `tp_call` 的值是一个函数指针，不可以是 0)，而 `PyFloat_Type` 是可以调用的，这就说明 `PyType_Type` 内部的 `tp_call` 是一个函数指针，这在 Python 的层面上我们已经验证过了，下面我们再来通过源码看一下。

```
1 //typeobject.c
2 PyTypeObject PyType_Type = {
3     PyVarObject_HEAD_INIT(&PyType_Type, 0)
4     "type", /* tp_name */
5     sizeof(PyHeapTypeObject), /* tp_basicsize */
6     sizeof(PyMemberDef), /* tp_itemsize */
7     (destructor)type_dealloc, /* tp_dealloc */
8     //... /* tp_hash */
9     (ternaryfunc)type_call, /* tp_call */
10    //...
11 }
```

我们看到在实例化 `PyType_Type` 的时候 `PyTypeObject` 内部的成员 `tp_call` 被设置成了 `type_call`。这是一个函数指针，当我们调用 `PyFloat_Type` 的时候，会触发这个 `type_call` 指向的函数。

因此 `float(3.14)` 在 C 的层面上等价于：

```
1 (&PyFloat_Type) -> ob_type -> tp_call(&PyFloat_Type, args, kwargs);
2 // 即：
```

```

3 (&PyType_Type) -> tp_call(&PyFloat_Type, args, kwargs);
4 // 而在创建 PyType_Type 的时候, 给 tp_call 成员传递的是 type_call
5 // 因此最终相当于
6 type_call(&PyFloat_Type, args, kwargs)

```

如果用 Python 来演示这一过程的话:

```

1 # float(3.14), 等价于
2 f1 = float.__class__.__call__(float, 3.14)
3 # 等价于
4 f2 = type.__call__(float, 3.14)
5
6 print(f1, f2) # 3.14 3.14

```

这就是 float(3.14) 的秘密, 相信list、dict在实例化的时候是怎么做的, 你已经猜到了, 做法是相同的。

```

1 # lst = list("abcd")
2 lst = list.__class__.__call__(list, "abcd")
3 print(lst) # ['a', 'b', 'c', 'd']
4
5 # dct = dict([("name", "古明地觉"), ("age", 17)])
6 dct = dict.__class__.__call__(dict, [("name", "古明地觉"), ("age", 17)])
7 print(dct) # {'name': '古明地觉', 'age': 17}

```

最后我们来围观一下 type_call 函数, 我们说 type 的 __call__ 方法, 在底层对应的是 type_call 函数, 它位于Object/typeobject.c中。

```

1 static PyObject *
2 type_call(PyTypeObject *type, PyObject *args, PyObject *kwargs)
3 {
4     // 如果我们调用的是 float
5     // 那么显然这里的 type 就是 &PyFloat_Type
6
7     // 这里是声明一个PyObject *
8     // 显然它是要返回的实例对象的指针
9     PyObject *obj;
10
11     // 这里会检测 tp_new 是否为空, tp_new 是什么估计有人已经猜到了
12     // 我们说__call__对应底层的tp_call
13     // 显然__new__对应底层的tp_new, 这里是为实例对象分配空间
14     if (type->tp_new == NULL) {
15         // tp_new 是一个函数指针, 指向具体的构造函数
16         // 如果 tp_new 为空, 说明它没有构造函数
17         // 因此会报错, 表示无法创建其实例
18         PyErr_Format(PyExc_TypeError,
19                     "cannot create '%.100s' instances",
20                     type->tp_name);
21         return NULL;
22     }
23
24     //通过tp_new分配空间
25     //此时实例对象就已经创建完毕了, 这里会返回其指针
26     obj = type->tp_new(type, args, kwargs);
27     //类型检测, 暂时不用管
28     obj = _Py_CheckFunctionResult((PyObject*)type, obj, NULL);
29     if (obj == NULL)
30         return NULL;
31
32     //我们说这里的参数type是类型对象, 但也可以是元类
33     //元类也是由PyTypeObject结构体实例化得到的
34     //元类在调用的时候执行的依旧是type_call
35     //所以这里是检测type指向的是不是PyType_Type
36     //如果是的话, 那么实例化得到的obj就不是实例对象了, 而是类型对象

```

```

37 //要单独检测一下
38 if (type == &PyType_Type &&
39     PyTuple_Check(args) && PyTuple_GET_SIZE(args) == 1 &&
40     (kwds == NULL ||
41      (PyDict_Check(kwds) && PyDict_GET_SIZE(kwds) == 0)))
42     return obj;
43
44 //tp_new应该返回相应类型对象的实例对象(的指针)
45 //但如果不是, 就直接将这里的obj返回
46 //此处这么做可能有点难理解, 我们一会细说
47 if (!PyType_IsSubtype(Py_TYPE(obj), type))
48     return obj;
49
50 //拿到obj的类型
51 type = Py_TYPE(obj);
52 //执行 tp_init
53 //显然这个tp_init就是__init__函数
54 //这与Python中类的实例化过程是一致的。
55 if (type->tp_init != NULL) {
56     //将tp_new返回的对象作为self, 执行 tp_init
57     int res = type->tp_init(obj, args, kwds);
58     if (res < 0) {
59         //执行失败, 将引用计数减1, 然后将obj设置为NULL
60         assert(PyErr_Occurred());
61         Py_DECREF(obj);
62         obj = NULL;
63     }
64     else {
65         assert(!PyErr_Occurred());
66     }
67 }
68 //返回obj
69 return obj;
70 }

```

因此从上面我们可以看到关键的部分有两个：

- 调用类型对象的 `tp_new` 指向的函数为实例对象申请内存
- 调用 `tp_init` 指向的函数为实例对象进行初始化，也就是设置属性

所以这对应Python中的 `__new__` 和 `__init__`，我们说 `__new__` 是为实例对象开辟一份内存，然后返回指向这片内存(对象)的指针，并且该指针会自动传递给 `__init__` 中的 `self`。

```

1 class Girl:
2
3     def __new__(cls, name, age):
4         print("__new__方法执行啦")
5         # 写法非常固定
6         # 调用object.__new__(cls)就会创建Girl的实例对象
7         # 因此这里的cls指的就是这里的Girl, 注意:一定要返回
8         # 因为__new__会将自己的返回值交给__init__中的self
9         return object.__new__(cls)
10
11     def __init__(self, name, age):
12         print("__init__方法执行啦")
13         self.name = name
14         self.age = age
15
16
17 g = Girl("古明地觉", 16)
18 print(g.name, g.age)
19 """
20 __new__方法执行啦
21 __init__方法执行啦

```

```
22 古明地觉 16
23 """
```

`__new__` 里面的参数要和 `__init__` 里面的参数保持一致，因为我们会先执行 `__new__`，然后解释器会将 `__new__` 的返回值和我们传递的参数组合起来一起传递给 `__init__`。因此 `__new__` 里面的参数除了 `cls` 之外，一般都会写 `*args` 和 `**kwargs`。

然后再回过头来看一下 `type_call` 中的这几行代码：

```
1  static PyObject *
2  type_call(PyTypeObject *type, PyObject *args, PyObject *kwargs)
3  {
4      //.....
5      //.....
6      if (!PyType_IsSubtype(Py_TYPE(obj), type))
7          return obj;
8
9      //.....
10     //.....
11 }
```

我们说 `tp_new` 应该返回该类型对象的实例对象，而且一般情况下我们是不写 `__new__` 的，会默认执行。但是我们一旦重写了，那么必须要手动返回 `object.__new__(cls)`。可如果我们不返回，或者返回其它的话，会怎么样呢？

```
1  class Girl:
2
3      def __new__(cls, *args, **kwargs):
4          print("__new__方法执行啦")
5          instance = object.__new__(cls)
6          # 打印看看instance到底是个什么东西
7          print("instance:", instance)
8          print("type(instance):", type(instance))
9
10         # 正确做法是将instance返回
11         # 但是我们不返回，而是返回个 123
12         return 123
13
14     def __init__(self, name, age):
15         print("__init__方法执行啦")
16
17
18 g = Girl()
19 """
20 __new__方法执行啦
21 instance: <__main__.Girl object at 0x000002C0F16FA1F0>
22 type(instance): <class '__main__.Girl'>
23 """
```

这里面有很多可以说的点，首先就是 `__init__` 里面需要两个参数，但是我们没有传，却还不报错。原因就在于这个 `__init__` 压根就没有执行，因为 `__new__` 返回的不是 `Girl` 的实例对象。

通过打印 `instance`，我们知道了 `object.__new__(cls)` 返回的就是 `cls` 的实例对象，而这里的 `cls` 就是 `Girl` 这个类本身。我们必须返回 `instance`，才会执行对应的 `__init__`，否则 `__new__` 直接就返回了。我们在外部来打印一下创建的实例对象吧，看看结果：

```
1  class Girl:
2
3      def __new__(cls, *args, **kwargs):
4          return 123
5
6      def __init__(self, name, age):
```

```

7     print("__init__方法执行啦")
8
9
10  g = Girl()
11  print(g, type(g)) # 123 <class 'int'>

```

我们看到打印的是123，所以再次总结一些`tp_new`和`tp_init`之间的区别，当然也对应`__new__`和`__init__`的区别：

- `tp_new`：为该类型对象的实例对象申请内存，在Python的`__new__`方法中通过`object.__new__(cls)`的方式申请，然后将其返回
- `tp_init`：`tp_new`的返回值会自动传递给`self`，然后为`self`绑定相应的属性，也就是进行实例对象的初始化

但如果`tp_new`返回的不是对应类型的实例对象的指针，比如`type_call`中第一个参数接收的`&PyFloat_Type`，但是`tp_new`中返回的却是`PyLongObject *`，所以此时就不会执行`tp_init`。

以上面的代码为例，我们`Girl`中的`__new__`应该返回`Girl`的实例对象才对，但实际上返回了整型，因此类型不一致，所以不会执行`__init__`。

下面我们可以做总结了，通过类型对象去创建实例对象的整体流程如下：

- 第一步：获取类型对象的类型对象，说白了就是元类，执行元类的 `tp_call` 指向的函数，即 `type_call`
- 第二步：`type_call` 会调用该类型对象的 `tp_new` 指向的函数，如果 `tp_new` 为 `NULL`，那么会到 `tp_base` 指定的父类里面去寻找 `tp_new`。在新式类当中，所有的类都继承自 `object`，因此最终会执行 `object` 的 `__new__`。然后通过访问对应类型对象中的 `tp_basicsize` 信息，这个信息记录着该对象的实例对象需要占用多大的内存，继而完成申请内存的操作
- 调用`type_new`创建完对象之后，就会进行实例对象的初始化，会将指向这片空间的指针交给 `tp_init`，但前提是 `tp_new` 返回的实例对象的类型要一致。

所以都说 Python 在实例化的时候会先调用 `__new__` 方法，再调用 `__init__` 方法，相信你应该知道原因了，因为在源码中先调用 `tp_new`、再调用的 `tp_init`。

```

1  static PyObject *
2  type_call(PyTypeObject *type, PyObject *args, PyObject *kwargs)
3  {
4      //调用__new__方法，拿到其返回值
5      obj = type->tp_new(type, args, kwargs);
6
7      if (type->tp_init != NULL) {
8          //将__new__返回的实例obj，和args、kwargs组合起来
9          //一起传给 __init__
10         //其中 obj 会传给 self，
11         int res = type->tp_init(obj, args, kwargs);
12         //.....
13         return obj;
14     }

```

所以源码层面表现出来的，和我们在 Python 层面看到的是一样的。

小结

到此，我们就从 Python 和解释器两个层面了解了对象是如何调用的，更准确的说我们是从解释器的角度对 Python 层面的知识进行了验证，通过 `tp_new` 和 `tp_init` 的关系，来了解 `__new__` 和 `__init__` 的关系。

另外，对象调用远不止我们目前说的这么简单，更多的细节隐藏在了幕后，只不过现在没办法将其一次性全部挖掘出来。后续我们会循序渐进，一点点揭开它什么面纱，并且

在这个过程中还会不断地学习到新的东西。比如说，实例对象在调用方法的时候会自动将实例本身作为参数传递给 self，那么它为什么传递呢？解释器在背后又做了什么工作呢？这些我们就以后慢慢说吧。

收录于合集 [#CPython 97](#)

[< 上一篇](#)

[《源码探秘 CPython》6. 对象的多态性、和行为](#)

[下一篇 >](#)

[《源码探秘 CPython》4. 对象是怎么被创建的？](#)

喜欢此内容的人还喜欢

[【Python】五种格式化输出字符串的方法](#)
[AI算法之道](#)



[【Python系列】为啥老问装饰器、迭代器、生成器？](#)
[嘎嘎软件测试](#)



[Golang序列化与反序列化操作](#)
[程序员khaos](#)

