

微信扫一扫  
关注该公众号

收录于合集

#CPython

97个 &gt;

## 对象的多态性

Python 创建一个对象，比如 `PyFloatObject`，会分配内存并进行初始化。然后内部统一使用泛型指针 `PyObject*` 来保存和维护这个对象，而不是 `PyFloatObject *`。通过 `PyObject *` 保存和维护对象，可以实现更加抽象的上层逻辑，而不用关心对象的实际类型和实现细节。比如：哈希计算。

```
1 Py_hash_t
2 PyObject_Hash(PyObject *v);
```

该函数可以计算任意对象的哈希值，而不用关心对象的类型是啥，它们都可以使用这个函数。

但是不同类型的对象，其行为也千差万别，哈希值计算的方式亦是如此，那么 `PyObject_Hash` 函数是如何解决这个问题的呢？不用想，因为元信息存储在对应的类型对象之中，所以肯定会通过其 `ob_type` 拿到指向的类型对象。而类型对象中有一个成员叫做 `tp_hash`，它是一个函数指针，指向的函数专门用来计算其实例对象的哈希值。所以我们看一下 `PyObject_Hash` 的函数定义吧，看看它内部都做了什么，该函数位于 `Object/Object.c` 中。

```
1 Py_hash_t
2 PyObject_Hash(PyObject *v)
3 {
4     //Py_TYPE是一个宏, 用来获取PyObject *内部的ob_type
5     PyTypeObject *tp = Py_TYPE(v);
6     //获取对应的类型对象内部的tp_hash
7     //tp_hash是一个函数指针, 对应 __hash__
8     if (tp->tp_hash != NULL)
9         //如果tp_hash不为空, 证明确实指向了具体的hash函数
10        //那么拿到函数指针之后, 通过* 获取对应的函数
11        //然后将PyObject *传进去计算哈希值, 返回。
12        return (*tp->tp_hash)(v);
13
14    //走到这里说明tp_hash为空, 但这存在两种可能。
15    //1. 说明该类型对象可能还未完全初始化, 导致tp_hash暂时为空
16    //2. 说明该类型本身就不支持其 "实例对象" 被哈希
17    //如果是第 1 种情况, 那么它的 tp_dict、也就是属性字典一定为空
18    //tp_dict是动态设置的, 它为空, 是类型对象没有完全初始化的重要特征
19    //但如果tp_dict不为空, 说明类型对象一定已经被完全初始化了
20    //所以此时tp_hash要是还为空, 就真的说明该类型不支持实例对象被哈希
21    if (tp->tp_dict == NULL) {
22        //属性字典为空, 那么先进行类型的初始化
23        if (PyType_Ready(tp) < 0)
24            return -1;
25        //然后再看是否tp_hash是否为空, 为空的话, 说明不支持哈希
26        //不为空则调用对应的哈希函数
27        if (tp->tp_hash != NULL)
28            return (*tp->tp_hash)(v);
29    }
30    // 走到这里代表以上条件都不满足, 说明该对象不可以被hash
31    return PyObject_HashNotImplemented(v);
32 }
```

函数先通过 `ob_type` 指针找到对象的类型，然后通过类型对象的 `tp_hash` 函数指针调用

对应的哈希计算函数。所以PyObject\_Hash根据对象的类型不同，然后调用不同的哈希函数，这不正是实现了多态吗？我们再以Python为例：

```
1 # 计算 v 的哈希值
2 hash(v)
3 # 而 hash(v) 等价于
4 v.__class__.__hash__(v)
5 # 如果 v 是一个列表, 那么就是 List.__hash__(v)
6 # 如果 v 是一个字符串, 那么就是 str.__hash__(v)
```

如果一个对象支持哈希操作，那么它的类型对象当中一定定义了\_\_hash\_\_方法，通过v.\_\_class\_\_就可以获取它的类型对象，然后将v作为参数调用\_\_hash\_\_即可。

所以通过ob\_type字段，Python在C语言的层面实现了对对象的多态特性，思路跟C++中的虚表指针有着异曲同工之妙。

另外可能有人觉得PyObject\_Hash函数的源码写的不是很精简，比如一开始已经判断过内部的tp\_hash是否为NULL，然后在下面又判断了一次。那么可不可以先判断tp\_dict是否为NULL，为NULL进行初始化，然后再判断tp\_hash是否NULL，不为NULL的话执行tp\_hash。这样的话，代码会变得精简很多。

答案是可以的，而且这种方式似乎更直观，但是效率上不如源码。因为我们这种方式的话，无论是什么对象，都需要判断其类型对象中tp\_dict和tp\_hash是否为NULL。而源码中先判断tp\_hash是否为NULL，不为NULL的话就不需要再判断tp\_dict了。所以对于已经初始化（tp\_hash不为NULL）的类型对象，源码中少了一次对tp\_dict是否为NULL的判断，效率会更高。

而这种行为叫做CPython的快分支，并且CPython中还有很多其它的快分支，快分支的特点就是命中率极高，可以尽早做出判断、尽早处理。回到当前这个场景，只有当类型对象未被初始化的时候，才会不走快分支；而一旦初始化完毕，那么后续就都走快分支。

也就是说，快分支只有在第一次调用的时候才可能不会命中，其余情况都是命中，因此没有必要每次都对tp\_dict进行判断。因此源码的设计是非常合理的，我们在后面分析函数调用的时候，也会看到很多类似于这样的快分支。

为了更好地理解快分支，我们举一个生活中的栗子：好比你去见心上人，但是心上人说你今天没有打扮，于是你又跑回去打扮一番，然后再去见心上人。那么问题来了，为什么不能先打扮呢。

答案是在绝大部分情况下，即使你不打扮，心上人也不会介意，只有在极少数情况下，比如心情不好，才会让你回去打扮之后再过来。所以不打扮直接去见心上人就能牵手便属于快分支，它的特点就是命中率极高，绝大部分都会走这个情况。因此没必要每次都因为打扮而耽误时间，因为快分支只有在极少数情况下才不会命中。

## 对象的行为

了解完对象的多态性，我们再来说说对象的行为。虽然Python的类型对象和实例对象都属于对象，但我们更关注的是实例对象的行为。

而不同对象的行为不同，比如哈希值的计算方式，它是由类型对象的tp\_hash成员决定的。但除了tp\_hash，PyTypeObject中还定义了很多其它的函数指针，这些指针最终都会指向某个函数，或者为空表示不支持该操作。

这些函数指针可以看做是**类型对象**所定义的操作，这些操作决定了其**实例对象**在运行时的**行为**。虽然所有类型对象在底层都是由结构体PyTypeObject实例化得到的，但内部成员接收的值不同，得到的类型对象就不同；类型对象不同，导致其实例对象的行为就不同，这也正是一种对象区别于另一种对象的关键所在。

比如int和str内部都有\_\_hash\_\_，但它们是不同的类型，因此哈希值的计算方式也不同。

而根据支持的操作不同，Python 中可以将对象进行以下分类：

- 数值型操作：比如整数、浮点数的加减乘除
- 序列型操作：比如字符串、列表、元组的通过索引、切片取值行为
- 映射型操作：比如字典通过 key 映射出 value

这三种操作，在 PyTypeObject 中分别对应三个指针。每个指针指向一个结构体实例，这个结构体实例中有大量的成员，成员也是函数指针，指向了具体的函数。我们回顾一下 PyTypeObject 的定义：

```
1  typedef struct _typeobject {
2      PyObject_VAR_HEAD
3      const char *tp_name;
4
5      // .....
6      PyNumberMethods *tp_as_number; // 数值型相关操作
7      PySequenceMethods *tp_as_sequence; // 序列型相关操作
8      PyMappingMethods *tp_as_mapping; // 映射型相关操作
9      // .....
10 } PyTypeObject;
```

PyNumberMethods、PySequenceMethods、PyMappingMethods 都是结构体，里面每一个成员也都是函数指针类型，指针指向的函数就是相应的操作。我们以 PyNumberMethods 为例，看看它是怎么定义的？

```
1  //object.h
2  typedef struct {
3      binaryfunc nb_add;
4      binaryfunc nb_subtract;
5      binaryfunc nb_multiply;
6      binaryfunc nb_remainder;
7      binaryfunc nb_divmod;
8      ternaryfunc nb_power;
9      unaryfunc nb_negative;
10     unaryfunc nb_positive;
11     unaryfunc nb_absolute;
12     inquiry nb_bool;
13     unaryfunc nb_invert;
14     binaryfunc nb_lshift;
15     binaryfunc nb_rshift;
16     //.....
17     //.....
18     binaryfunc nb_inplace_matrix_multiply;
19 } PyNumberMethods;
```

你看到了什么？是不是想到了Python里面的魔法方法，所以它们也被称为**方法簇**。

在**PyNumberMethods**这个方法簇里面定义了作为一个数值应该支持的操作，如果一个对象能被视为数值，比如整数，那么在其对应的类型对象 PyLong\_Type中，tp\_as\_number -> nb\_add 就指定了该对象进行加法操作时的具体行为。

同样，PySequenceMethods 和 PyMappingMethods中分别定义了作为一个序列对象和映射对象应该支持的行为，这两种对象的典型例子就是 list和 dict。

所以，只要**类型对象**提供**相关操作**，**实例对象**便具备**对应的行为**，因为实例对象对象所调用的方法都是由类型对象提供的。

```
1  class Girl:
2
3      def __init__(self, name, age):
4          self.name = name
5          self.age = age
6
7      def say(self):
```

```

8         pass
9
10    def cry(self):
11        pass
12
13
14 g = Girl("古明地觉", 16)
15 print(g.__dict__) # {'name': '古明地觉', 'age': 16}
16 print("say" in Girl.__dict__) # True
17 print("cry" in Girl.__dict__) # True

```

我们看到实例对象的属性字典里面只有在 `__init__` 里面设置的一些属性而已，而实例能够调用的 `say`、`cry` 都是定义在类型对象中的。

因此一定要记住：**类型对象定义的操作，决定了实例对象的行为。**

```

1 class Int(int):
2
3     def __getitem__(self, item):
4         return item
5
6
7 a = Int(1)
8 b = Int(2)
9
10 print(a + b) # 3
11 print(a["你好"]) # 你好

```

继承自 `int` 的 `Int` 在实例化之后自然是一个数值对象，但看上去 `a[""]` 这种操作是一个类似于字典才支持的操作，为什么可以实现呢？

原因就是 我们重写了 `__getitem__` 这个魔法方法，该方法在底层对应 `PyMappingMethods` 中的 `mp_subscript` 操作。最终 `Int` 实例对象表现的像一个字典一样。

归根结底就在于这几个方法簇都只是 `PyTypeObject` 的一个成员罢了，默认使用 `PyTypeObject` 结构体创建的 `PyLong_Type` 所生成的实例对象是不具备列表和字典的属性特征的。但是我们继承 `PyLong_Type`，同时指定 `__getitem__`，使得我们自己构建出来的类型对象所生成的实例对象，同时具备多种属性特征，就是因为解释器支持这种做法。

我们自定义的类在底层也是 `PyTypeObject` 结构体实例，而在继承 `int` 的时候，将其内部定义的 `PyNumberMethods` 方法簇也继承了下来，而我们又单独实现了 `PyMappingMethods` 中的 `mp_subscript`。所以自定义类 `Int` 的实例对象具备了整数的全部行为，以及字典的部分行为（因为我们只实现了 `__getitem__`）。

我们再通过 `PyFloat_Type` 实际考察一下：

```

1 //Object/floatobject.c
2 PyTypeObject PyFloat_Type = {
3     PyVarObject_HEAD_INIT(&PyType_Type, 0)
4     "float",
5     sizeof(PyFloatObject),
6     //.....
7     &float_as_number,          /* tp_as_number */
8     0,                         /* tp_as_sequence */
9     0,                         /* tp_as_mapping */
10    //.....
11 };

```

我们看到了该类型对象 `float` 在创建时，给成员 `tp_as_number` 传入了一个 `float_as_number` 指针。那么这个 `float_as_number` 就是 `PyNumberMethods` 结构

体实例，而其内部的每一个成员都是指向了浮点数运算函数的指针。

```
1 static PyNumberMethods float_as_number = {
2     float_add,          /* nb_add */
3     float_sub,          /* nb_subtract */
4     float_mul,          /* nb_multiply */
5     float_rem,          /* nb_remainder */
6     float_divmod,       /* nb_divmod */
7     float_pow,          /* nb_power */
8     // ...
9 };
```

里面的 `float_add`、`float_sub`、`float_mul` 等等显然都是已经定义好的函数指针，然后创建 `PyNumberMethods` 结构体实例 `float_as_number` 的时候，分别赋值给了成员 `nb_add`、`nb_subtract`、`nb_multiply` 等等。

而创建完浮点数相关操作的 `PyNumberMethods` 结构体实例 `float_as_number` 之后，将其指针交给 `PyFloat_Type` 中的 `tp_as_number` 成员。而浮点数相加的时候，会先通过 `变量 -> ob_type -> tp_as_number -> nb_add` 获取该操作对应的函数指针，其中浮点类型对象的 `tp_as_number` 成员的值是 `&float_as_number`，因此再获取其成员 `nb_add` 的时候，拿到的就是 `float_add` 指针，然后调用 `float_add` 函数。

整个过程还是不难理解的，另外我们在 `PyFloat_Type` 中看到 `tp_as_sequence` 和 `tp_as_mapping` 这两个成员接收到的值则不是一个函数指针，而是 0（相当于空）。

因此浮点数不支持序列型操作和映射型操作，比如：`pi = 3.14`，我们无法使用 `len` 计算长度、无法通过索引或者切片获取指定位置的值、无法通过 `key` 获取 `value`，这和我们使用 Python 时候的表现是一致的。

## 小结

以上就是对象的多态性和行为，多态比较简单，是通过泛型指针 `PyObject *` 和 `ob_type` 实现的。

而对象的行为是由其类型对象内部定义的操作所决定的，比如一个对象可以计算长度，那么它的类型对象内部要实现 `__len__`；一个对象可以转成整数，那么它的类型对象内部要实现 `__int__` 或 `__index__`。

```
1 class A:
2
3     def __len__(self):
4         return 123
5
6     def __int__(self):
7         return 456
8
9
10 a = A()
11 print(len(a)) # 123
12 print(int(a)) # 456
13 # 而 len(a) 在底层会调用 A.__len__(a)
14 # int(a) 在底层会调用 A.__int__(a)
15 print(A.__len__(a)) # 123
16 print(A.__int__(a)) # 456
17
18 # 注意: len(a) 在底层调用的是 A.__len__(a), 而不是 a.__len__()
19 # 举个例子
20 print(a.__len__(), len(a)) # 123 123
21 a.__dict__["__len__"] = "哼哼哼"
22 print(a.__len__, len(a)) # 哼哼哼 123
```

```
23 其它内置函数同理
24 # 而且实例调用类型对象中定义的方法时, 事实上也是通过类型对象调用的
25 # a.some_method(*args) 只是 A.some_method(a, *args) 的一个语法糖
```

总之核心就是一句话：**类型对象**定义了**哪些操作**，决定了**实例对象**具备**哪些行为**。

收录于合集 #CPython 97

< 上一篇

《源码探秘 CPython》7. 对象的引用计数，对象何时被回收？

下一篇 >

《源码探秘 CPython》5. 对象是如何被调用的

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

