《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Python中的对象模型

原创 古明地觉 古明地觉的编程教室 2022-01-02 23:39

收录于合集
#CPython
97个 >

Python 中一切皆对象

关于 Python，你肯定听过这么一句话："Python 中一切皆对象"。没错，在 Python 的世界里，一切都是对象。整数是一个对象、字符串是一个对象、字典是一个对象，甚至 int、str、list 等等，再加上我们使用 class 自定义的类，它们也是对象。

像 int、str、list 等基本类型，以及我们自定义的类，由于它们可以表示类型，因此我们称之为**类型对象**；类型对象实例化得到的对象，我们称之为**实例对象**。但不管是哪种对象，它们都属于对象。

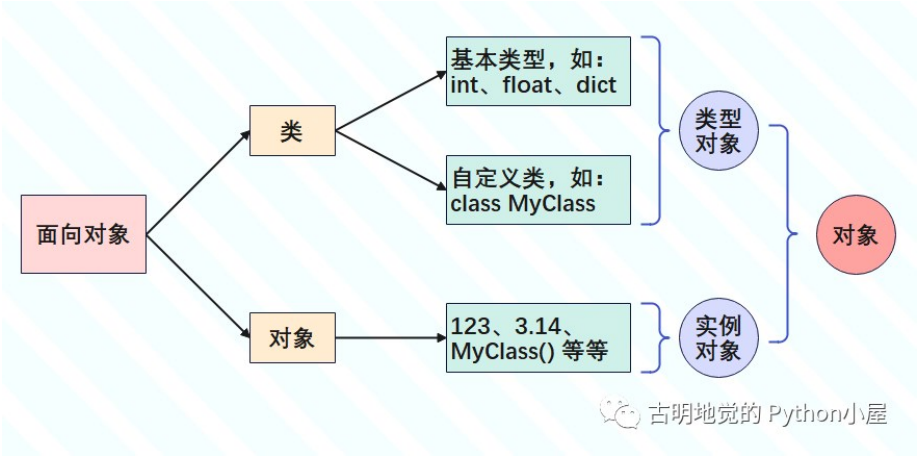
因此 Python 中面向对象的理念贯彻的非常彻底，面向对象中的"类"和"对象"在 Python 中都是通过"对象"实现的。

> 在面向对象理论中，存在着"类"和"对象"两个概念，像 int、dict、tuple、以及使用 class 关键字自定义的类型对象实现了面向对象理论中"类"的概念，而 123、(1, 2, 3)，"xxx" 等等这些实例对象则实现了面向对象理论中"对象"的概念。但是在 Python 中，面向对象的"类"和"对象"都是通过对象实现的。

我们举个栗子：

```
>>> # int它是一个类，因此它属于类型对象，类型对象实例化得到的对象属于实例对象
>>> int
<class 'int'>
>>> int('0123')
123
>>>
```

因此可以用一张图来描述面向对象在Python中的体现：



类型、对象体系

a 是一个整数（实例对象），其类型是 int（类型对象）。

```
>>> a = 123
```

```
>>> a
123
>>> type(a)
<class 'int'>
>>> isinstance(a, int)
True
>>>
```

但是问题来了，按照面向对象的理论来说，对象是由类实例化得到的，这在 Python 中也是适用的。既然是对象，那么就必定有一个类来实例化它，换句话说对象一定要有类型。至于一个对象的类型是什么，就看这个对象是被谁实例化的，被谁实例化那么类型就是谁。而我们说 Python 中一切皆对象，所以像 int、str、tuple 这些内置的类型也是具有相应的类型的，那么它们的类型又是谁呢?

我们使用type函数查看一下就好了。

```
>>> type(int)
<class 'type'>
>>> type(str)
<class 'type'>
>>> type(dict)
<class 'type'>
>>> type(type)
<class 'type'>
>>>
```

我们看到类型对象的类型，无一例外都是 **type**。type 应该是初学 Python 的时候就接触了，当时使用 type 都是为了查看一个对象的类型，然而 type 的作用远没有这么简单，我们后面会说，总之我们目前看到类型对象的类型是 type。

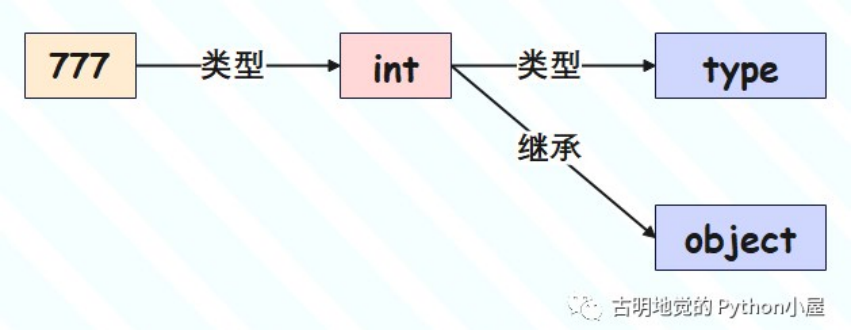
所以 int、str 等类型对象是 type 的对象，而 type 我们也称其为元类，表示类型对象的类型。至于 type 本身，它的类型还是 type，所以它连自己都没放过，把自己都变成自己的对象了。

> 因此在 Python 中，你能看到的任何对象都是有类型的，我们可以使用 type 函数查看，也可以获取该对象的 __class__ 属性查看。所以：实例对象、类型对象、元类，Python 中任何一个对象都逃不过这三种身份。

Python 中还有一个特殊的类型（对象），叫做 object，它是所有类型对象的基类。不管是什么类，内置的类也好，我们自定义的类也罢，它们都继承自 object。因此，object 是所有类型对象的"基类"、或者说"父类"。

```
>>> issubclass(int, object)
True
>>>
```

因此，综合以上关系，我们可以得到下面这张关系图：

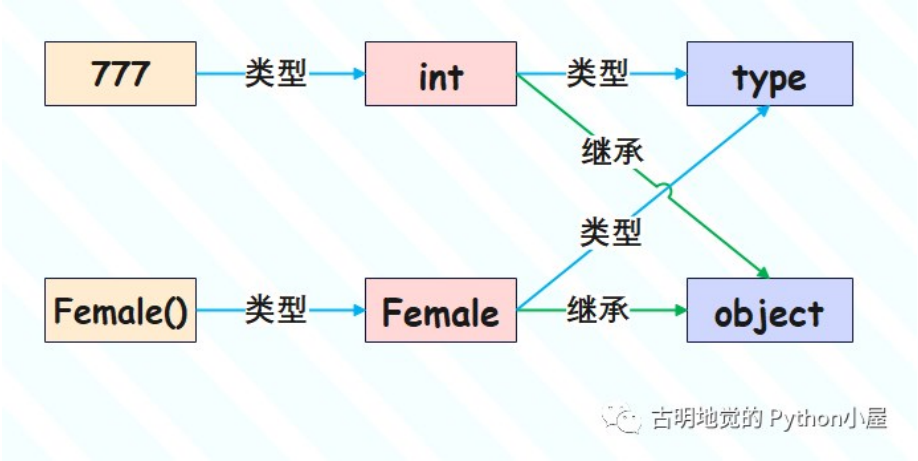


我们自定义的类型也是如此，举个栗子：

```
class Female:
    pass
```

```
3
4
5 print(type(Female))  # <class 'type'>
6 print(issubclass(Female, object))  # True
```

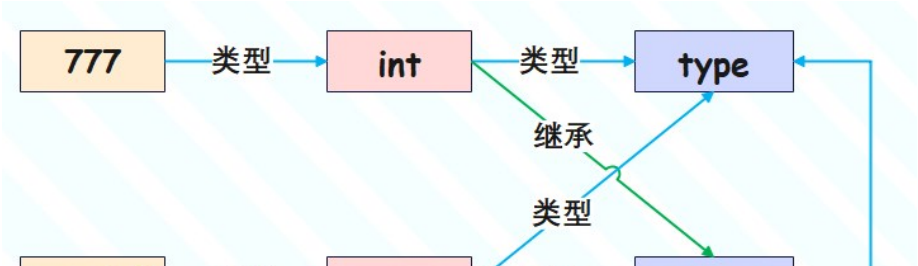
在 Python3 中，自定义的类即使不显式的继承 object，也会默认继承自 object。

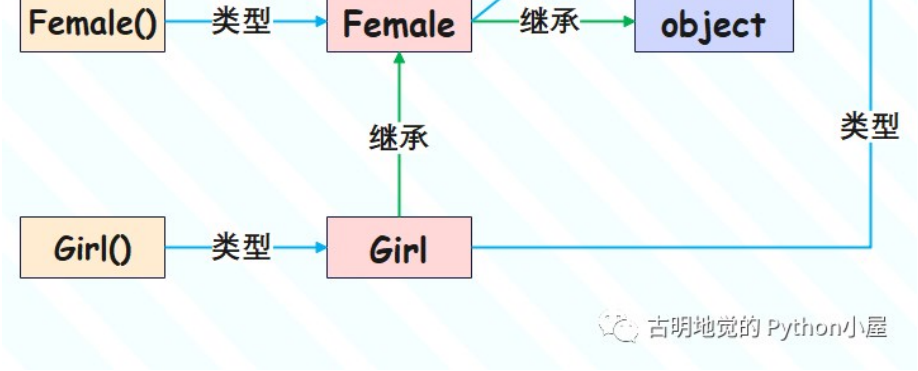


那么我们自定义再自定义一个子类，继承自 Female 呢?

```
1  class Female:
2      pass
3
4  class Girl(Female):
5      pass
6  # 自定义类的类型都是 type
7  print(type(Girl))  # <class 'type'>
8
9  # 但 Girl 继承自 Female，所以它是 Female 的子类
10 print(issubclass(Girl, Female))  # True
11 # 而 Female 继承自 object，所以 Girl 也是 object 的子类
12 print(issubclass(Girl, object))  # True
13
14
15 # 这里需要额外多提一句实例对象，我们之前使用 type 得到的都是该类的类型对象
16 # 换句话说谁实例化得到的它，那么对它使用 type 得到的就是谁
17 print(type(Girl()))  # <class '__main__.Girl'>
18 print(type(Female()))  # <class '__main__.Female'>
19
20 # 但是我们说 Girl 的父类是 Female，Female 的父类是 object
21 # 所以 Girl 的实例对象也是 Female 和 object 的实例对象
22 # Female 的实例对象也是 object 的实例对象
23 print(isinstance(Girl(), Female))  # True
24 print(isinstance(Girl(), object))  # True
```

因此上面那张关系图就可以变成下面这样:





我们说可以使用 type 和 __class__ 查看一个对象的类型，并且还可以通过 isinstance 来判断该对象是不是某个已知类型的实例对象；那如果想查看一个类型对象都继承了哪些类该怎么做呢？我们目前都是使用 issubclass 来判断某个类型对象是不是另一个已知类型对象的子类，那么可不可以直接获取某个类型对象都继承了哪些类呢？

答案是可以的，方法有三种，我们分别来看一下：

```
class A: pass

class B: pass

class C(A): pass

class D(B, C): pass

# 首先 D 继承自 B 和 C，C 又继承 A，我们现在要来查看 D 继承的父类
# 方法一：使用 __base__
print(D.__base__)  # <class '__main__.B'>

# 方法二：使用 __bases__
print(D.__bases__)  # (<class '__main__.B'>, <class '__main__.C'>)

# 方法三：使用 __mro__
print(D.__mro__)
# (<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <cl
ass '__main__.A'>, <class 'object'>)
```

- `__base__`：如果继承了多个类，那么只显示继承的第一个类，没有显示继承则返回一个 `<class 'object'>`
- `__bases__`：返回一个元组，会显示所有直接继承的父类，如果没有显示的继承，则返回 `(<class 'object'>,)`
- `__mro__`：mro(Method Resolution Order)表示方法查找顺序，会从自身出发，找到最顶层的父类，因此返回自身、继承的基类、以及基类继承的基类，一直找到 object

最后我们来看一下 **type** 和 **object**，估计这两个老铁之间的关系会让很多人感到困惑。我们说 type 是所有类的元类，而 object 是所有的基类，这就说明 type 是要继承自 object 的，而 object 的类型是 type。

```
>>> type.__base__
<class 'object'>
>>> object.__class__
<class 'type'>
>>>
```

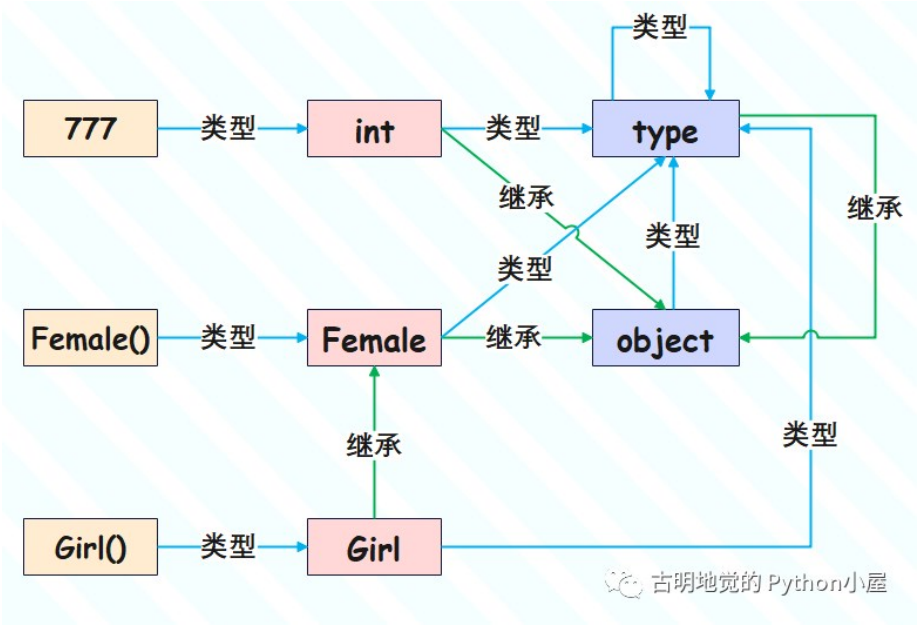
这就怪了，这难道不是一个先有鸡还是先有蛋的问题吗？其实不是的，这两个对象是共存的，它们之间的定义其实是互相依赖的。至于到底是怎么肥事，我们后面在看解释器源码的时候就会很清晰了。

总之目前先记住两点：

- `type站在类型金字塔的最顶端， 任何的对象按照类型追根溯源， 最终得到的都是type`
- `object站在继承金字塔的最顶端，任何的类型对象按照继承追根溯源，最终得到的都是object`

我们说 **type** 的类型还是 **type**，但是 **object** 的基类则不再是 **object**，而是一个 **None**。这是为什么呢？其实答案很简单，Python 在查找属性或方法的时候，会回溯继承链，自身如果没有的话，就会按照 __mro__ 指定的顺序去基类中查找。所以继承链一定会有一个终点，否则就会像没有出口的递归一样出现死循环了。

最后将上面那张关系图再完善一下的话：



因此上面这种图才算是完整，其实只看这张图我们就能解读出很多信息。
比如：
实例对象的类型是类型对象，类型对象的类型是元类；
所有的类型对象的基类都收敛于 object，所有对象的类型都收敛于 type。
因此 Python 算是将一切皆对象的理念贯彻到了极致，也正因为如此，Python 才具有如此优秀的动态特性。

> 事实上，目前介绍的有些基础了，但 Python 的对象的概念确实非常重要。为了后面再分析源码的时候能够更轻松，因此我们有必要系统地回顾一下，并且上面的关系图会使我们在后面的学习变得轻松。

Python 中的变量只是个名字

Python 中的变量只是个名字，站在 C 语言的角度来说的话，Python 中的变量存储的只是对象的内存地址，或者说指针，这个指针指向的内存存储的才是对象。

所以在 Python 中，我们都说变量指向了某个对象。在其它静态语言中，变量相当于是为某块内存起的别名，获取变量等于获取这块内存所存储的值。而 Python 中变量代表的内存所存储的不是对象，只是对象的指针（或者说引用）。

我们用两段代码，一段 C 语言的代码，一段 Python 的代码，来看一下差别。

```c
#include <stdio.h>

```

```c
void main()
{
    int a = 123;
    printf("address of a = %p\n", &a);

    a = 456
    printf("address of a = %p\n", &a);
}
//输出结果
/*
address of a = 0x7fffa94de03c
address of a = 0x7fffa94de03c
*
```

我们看到前后输出的地址是一样的，再来看看 Python 的。

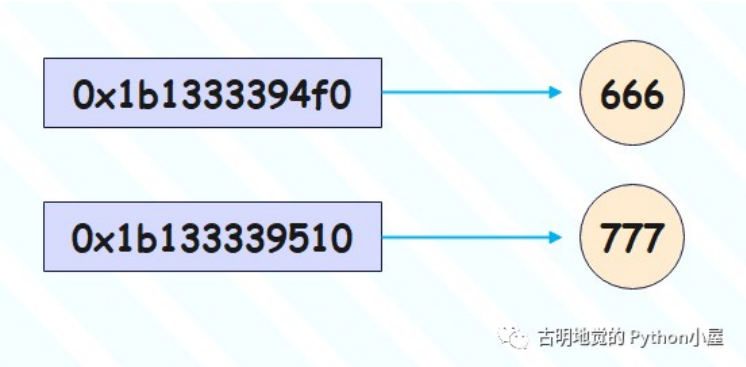
```python
a = 666
print(hex(id(a)))  # 0x1b1333394f0

a = 667
print(hex(id(a)))  # 0x1b133339510
```

然而我们看到 Python 中变量 a 的地址前后发生了变化，我们分析一下原因。

首先在 C 中，创建一个变量的时候必须规定好类型，比如 int a = 666，那么变量 a 就是 int 类型，以后在所处的作用域中就不可以变了。如果这时候，再设置 a = 777，那么等于是把内存中存储的 666 换成 777，a 的地址和类型是不会变化的。

而在 Python 中，a = 666 等于是先开辟一块内存，存储的值为 666，然后让变量 a 指向这片内存，或者说让变量 a 存储这块内存的指针。然后 a = 777 的时候，再开辟一块内存，然后让 a 指向存储 777 的内存，由于是两块不同的内存，所以它们的地址是不一样的。



> 所以 Python 中的变量只是一个和对象关联的名字罢了，它代表的是对象的指针。换句话说 Python 中的变量就是个便利贴，可以贴在任何对象上，一旦贴上去了，就代表这个对象被引用了。

我们再来看看变量之间的传递，在 Python 中是如何体现的。

```python
a = 666
print(hex(id(a)))  # 0x1e6c51e3cf0

b = a
print(hex(id(b)))  # 0x1e6c51e3cf0
```

我们看到打印的地址是一样的，我们再用一张图解释一下。

我们说 a = 666 的时候，先开辟一份内存，再让 a 存储对应内存的指针；然后 b = a 的时候，会把 a 的地址拷贝一份给 b，所以 b 存储了和 a 相同的地址，它们都指向了同一个对象。

因此说 Python 是值传递、或者引用传递都是不准确的，准确的说 Python 是变量之间的赋值传递，对象之间的引用传递。因为 Python 中的变量本质上就是一个指针，所以在 b = a 的时候，等于把 a 指向的对象的地址（a 本身）拷贝一份给 b，所以对于变量来说是赋值传递；然后 a 和 b 又都是指向对象的指针，因此对于对象来说是引用传递。

另外还有最关键的一点，我们说 Python 中的变量是一个指针，当传递一个变量的时候，传递的是指针；但是在操作一个变量的时候，会操作变量指向的内存。所以 id(a) 获取的不是 a 的地址，而是 a 指向的内存的地址（在底层其实就是 a 本身），同理 b = a，是将 a 本身，或者说将 a 存储的、指向某个具体的对象的地址传递给了 b。

> 在 C 的层面上，显然 a 和 b属于指针变量，那么 a 和 b有没有地址呢？显然是有的，只不过在 Python 中你是看不到的，Python 解释器只允许你看到对象的地址。

最后提一下变量的类型

我们说变量的类型其实不是很准确，应该是变量指向（引用）的对象的类型，因为 Python 中变量是个指针，操作指针会操作指针指向的内存，所以我们使用 type(a) 查看的其实是变量 a 指向的对象的类型，当然为了方便也会直接说变量的类型，理解就行。那么问题来了，我们在创建一个变量的时候，并没有显示的指定类型啊，但 Python 显然是有类型的，那么 Python 是如何判断一个变量指向的是什么类型的数据呢？

答案是：解释器是通过靠猜的方式，通过你赋的值（或者说变量引用的值）来推断类型。所以在 Python 中，如果你想创建一个变量，那么必须在创建变量的时候同时赋值，否则解释器就不知道这个变量指向的数据是什么类型。所以 Python 是先创建相应的值，这个值在 C 中对应一个结构体，结构体里面有一个成员专门用来存储该值对应的类型，因此在 Python 中，类型是和对象绑定的，而不是和变量。当创建完值之后，再让这个变量指向它，所以 Python 中是先有值后有变量。但显然 C 中不是这样的，因为 C 中变量代表的内存所存储的就是具体的值，所以 C 中可以直接声明一个变量的同时不赋值。因为 C 要求声明变量的同时必须指定类型，所以声明变量的同时，其类型和内存大小就已经固定了。而 Python 中变量代表的内存是个指针，它只是指向了某个对象，所以由于其便利贴的特性，可以贴在任意对象上面，但是不管贴在哪个对象，你都必须先有对象才可以，不然变量贴谁去？

另外，尽管 Python 在创建变量的时候不需要指定类型，但 Python 是强类型语言，强类型语言，强类型语言，重要的事情说三遍。而且是动态强类型，因为类型的强弱和是否需要显示声明类型之间没有关系。

可变对象与不可变对象

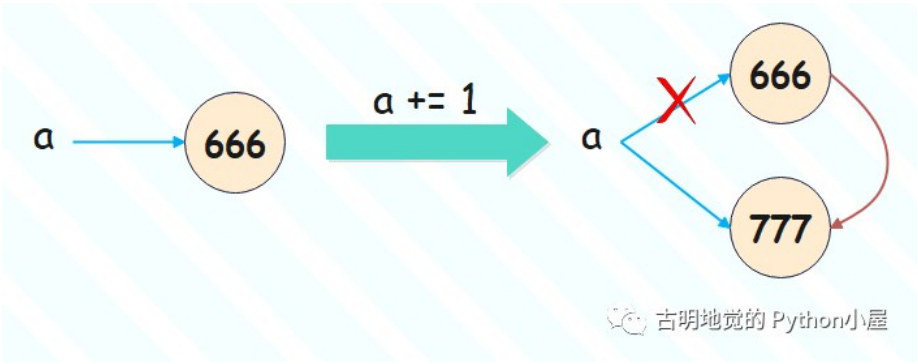
我们说一个对象其实就是一片被分配的内存空间，内存中存储了相应的值，不过这些空间可以是连续的，也可以是不连续的。

不可变对象一旦创建，其内存中存储的值就不可以再修改了。如果想修改，只能创建一个新的对象，然后让变量指向新的对象，所以前后的地址会发生改变。而可变对象在创建之后，其存储的值可以动态修改。

像整数就是一个不可变对象。

```
>>> a = 666
>>> id(a)
1365442984464
>>> a += 1
>>> id(a)
1365444032848
>>>
```

我们看到在对 a 执行 +1 操作时，前后地址发生了变化，所以整数不支持本地修改，因此是一个不可变对象；



原来 a = 666，而我们说操作一个变量等于操作这个变量指向的内存，所以 a+=1，会将 a 指向的整型对象 666 和 1 进行加法运算，得到 667。所以会开辟新的空间来存储这个 667，然后让 a 指向这片新的空间，至于原来的 666 所占的空间怎么办，Python 解释器会看它的引用计数，如果不为 0 代表还有变量引用（指向）它，如果为 0 证明没有变量引用了，所以会被回收。

关于引用计数，我们后面会详细说，目前只需要知道当一个对象被一个变量引用的时候，那么该对象的引用计数就会加 1。有几个变量引用，那么它的引用计数就是几。

可能有人觉得，每次都要创建新对象，销毁旧对象，效率肯定会很低吧。事实上确实如此，但是后面我们会从源码的角度上来看 Python 如何通过小整数对象池等手段进行优化。

而列表是一个可变对象，它是可以修改的。

> 这里先多提一句，Python 中的对象本质上就是 C 中 malloc 函数为结构体实例在堆区申请的一块内存。Python 中的任何对象在 C 中都会对应一个结构体，这个结构体除了存放具体的值之外，还存放了一些额外的信息，这个我们在剖析 Python 中的内建类型的实例对象的时候会细说。

首先 Python 中列表，当然不光是列表，还有元组、集合，这些容器它们的内部存储的也不是具体的对象，而是对象的指针。比如：lst = [1, 2, 3]，你以为 lst 存储的是三个整型对象吗？其实不是的，lst 存储的是三个整型对象的指针，当我们使用 lst[0] 的时候，拿到的是一个指针，但是操作（比如print）的时候会自动操作（print）指针指向的内存。

不知道你是否思考过，Python 底层是 C 来实现的，所以 Python 中的列表的实现必然要借助 C 中的数组。可我们知道 C 中的数组里面的所有元素的类型必须一致，但列表却可以存放任意的元素，因此从这个角度来讲，列表里面的元素它就不可能是对象，因为不同的对象在底层对应的结构体是不同的，所以这个元素只能是指针。

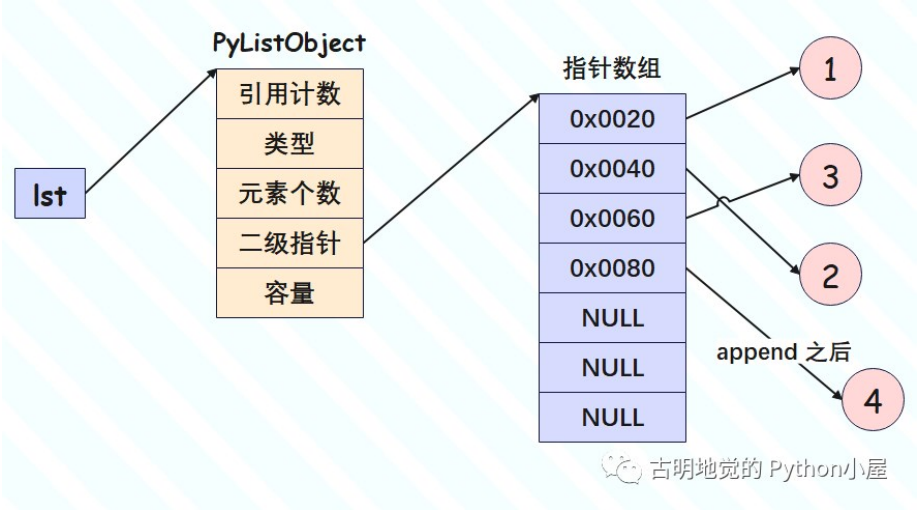
可能有人又好奇了，不同对象的指针也是不同的啊，是的，但 C 中的指针是可以转化的。Python 底层将所有对象的指针，都转成了 PyObject 类型的指针，这样不就是同一种类型的指针了吗？关于这个 PyObject，它是我们后面要剖析的重中之重，这个 PyObject 贯穿了我们的整个系列。目前只需要知道列表（还有其

它容器）存储的元素、以及 Python 中的变量，它们都是一个泛型指针 PyObject *。

```
>>> lst = [1, 2, 3]
>>> id(lst)
1365442893952
>>> lst.append(4)
>>> lst
[1, 2, 3, 4]
>>> id(lst)
1365442893952
>>>
```

我们看到列表在添加元素的时候，前后地址并没有改变。列表在 C 中是通过 PyListObject 实现的，我们在介绍列表的时候会细说。这个 PyListObject 内部除了一些基本信息之外，还有一个成员叫 ob_item，它是一个 PyObject 的二级指针，指向了我们刚才说的 PyObject * 类型的数组的首个元素的地址。

结构图如下：



显然图中的指针数组是用来存储具体的对象的指针的，每一个指针都指向了相应的对象（这里是整型对象）。可能有人注意到，整型对象的顺序有点怪，其实我是故意这么画的。因为 PyObject * 数组内部的元素是连续且有顺序的，但是指向的整型对象则是存储在堆区的，它们的位置是任意性的。但是不管这些整型对象存储在堆区的什么位置，它们和数组中的指针都是一一对应的，我们通过索引是可以正确获取到指向的对象的。

另外我们还可以看到一个现象，那就是 Python 中的列表在底层是分开存储的，因为 PyListObject 结构体实例并没有存储相应的指针数组，而是存储了指向这个指针数组的二级指针。显然我们添加、删除、修改元素等操作，都是通过这个二级指针来间接操作这个指针数组。

为什么要这么做？

因为在一个对象一旦被创建（任何语言都是如此），那么它在内存中的大小就不可以变了。所以这就意味着那些可以容纳可变长度数据的可变对象，要在内部维护一个指针，指针指向一个内存区域，该区域存放具体的数据。如果空间不够了，那么就申请一片更大的内存区域，然后将元素依次拷贝过去，再让指针指向新的内存区域。而我们看到 PyListObject 正是这么做的，其内部并没有直接存储具体的指针数组，而是存储了指向它的二级指针。但是 Python 在计算内存大小的时候是会将这个指针数组也算进去的，所以 Python 中列表的大小是可变的，但是底层对应的 PyListObject 实例的大小是不变的，因为指针数组没有存在PyListObject里面。但为什么要这么设计呢？

这么做的原因就在于，遵循这样的规则可以使通过指针维护对象的工作变得非常简单。一旦允许对象的大小可在运行期改变，那么我们就可以考虑如下场景。

在内存中有对象 A，并且其后面紧跟着对象 B。如果运行的某个时候，A 的大小增大了，这就意味着必须将 A 整个移动到内存中的其他位置，否则 A 增大的部分会覆盖掉原本属于 B 的数据。但要将 A 移动到内存的其他位置，那么所有指向 A 的指针就必须立即得到更新。可想而知这样的工作是多么的繁琐，因此通过在可变对象的内部维护一个指针就变得简单多了。

定长对象与变长对象

Python 中一个对象占用的内存有多大呢？相同类型的实例对象的大小是否相同呢？试一下就知道了，我们可以通过 sys 模块中 getsizeof 函数查看一个对象所占的内存。

```
import sys

print(sys.getsizeof(0))  # 24
print(sys.getsizeof(1))  # 28
print(sys.getsizeof(2 << 33))  # 32


print(sys.getsizeof(0.))  # 24
print(sys.getsizeof(3.14))  # 24
print(sys.getsizeof((2 << 33) + 3.14))  # 24
```

我们看到整型对象的大小不同，所占的内存也不同，像这种内存大小不固定的对象，我们称之为**变长对象**；而浮点数所占的内存都是一样的，像这种内存大小固定的对象，我们称之为**定长对象**

> 至于Python是如何计算对象所占的内存，我们在剖析具体对象的时候会说，因为这要涉及到底层对应的结构体。

而且我们知道 Python 中的整数是不会溢出的，而 C 中的整数显然是有最大范围的，那么 Python 是如何做到的呢？答案是 Python 在底层是通过 C 的 32 位整型数组来存储自身的整数的，通过多个 32 位整型组合起来，以支持存储更大的数值，所以整型对象越大，就需要越多的 32 位整数。而 32 位整数是 4 字节，所以我们上面代码中的那些整型对象，都是 4 字节、4 字节的增长。

当然 Python 中的对象在底层都是一个结构体，这个结构体中除了维护具体的值之外，还有其它的成员信息，在计算内存大小的时候，它们也是要考虑在内的，当然这些我们后面会说。

而浮点数的大小是不变的，因为 Python 的浮点数的值在 C 中是通过一个 double 来维护的。而 C 中的值的类型一旦确定，大小就不变了，所以 Python 的 float 对象的大小也是不变的。但既然是固定的类型，肯定范围是有限的，所以当浮点数不断增大，会牺牲精度来进行存储。如果实在过大，那么会抛出 OverFlowError。

```
>>> int(100000000000000000000000000000000.)  # 牺牲了精度
99999999999999994557523098704281 6
>>> 10 ** 1000  # 不会溢出
1000000000000000.....
>>>
>>> 10. ** 1000  # 报错了
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OverflowError: (34, 'Result too large')
>>>
```

还有字符串，字符串毫无疑问肯定是变长对象，因为长度不同大小不同。

```
import sys

```

```
print(sys.getsizeof("a"))  # 50
print(sys.getsizeof("abc"))  # 52
```

我们看到多了两个字符，多了两个字节，这很好理解。但是这些说明了一个空字符串要占 49 个字节，我们来看一下。

```
import sys

print(sys.getsizeof(""))  # 49
```

显然这 49 个字节是用来维护其它成员信息的，因为底层的结构体除了维护具体的值之外，还要维护其它的信息，比如：引用计数等等，这些在分析源码的时候会详细说。

小结

本次介绍了 Python 中的对象体系，我们说 Python 中一切皆对象，类型对象和实例对象都属于对象；还说了对象的种类，根据是否支持本地修改可以分为可变对象和不可变对象，根据占用的内存是否变化可以分为定长对象和变长对象；还说了 Python 中变量的本质，Python 中的变量本质上是一个指针（PyObject *），而变量则存储在对应的名字空间（或者说命名空间）中，当然名字空间我们没有说，是因为这些在后续系列会详细说，不过这里可以先补充一下。

名字空间分为：全局名字空间（存储全局变量）、局部名字空间（存储局部变量）、闭包名字空间（存储闭包变量）、内建名字空间（存储内置变量，比如int、str，它们都在这里），而名字空间又分为静态名字空间和动态名字空间：比如局部名字空间就是静态的，因为函数中的局部变量在编译的时候就可以确定，所以函数对应的局部名字空间是使用一个数组存储的；而全局变量是动态的，因为在运行时可以进行动态添加、删除，因此全局名字空间使用的是一个字典来保存，字典的 key 就是变量的名字（依旧是个指针，底层是指向字符串的指针），字典的 value 就是变量指向的对象（的指针）。

```
a = 123
b = "xxx"

# 通过 globals() 即可获取全局名字空间
print(globals())  #{..., 'a': 123, 'b': 'xxx'}

# 我们看到虽然显示的是变量名和变量指向的值
# 但是在底层，字典存储的键值对也是指向具体对象的指针
# 只不过我们说操作指针会操作指向的内存
# 所以这里 print 打印之后，显示的也是具体的值，但是存储的是指针
# 至于对象本身，则存储在堆区，并且被指针指向



#  此外，我们往全局名字空间中设置一个键值对，也等价于创建了一个全局变量
globals()["c"] = "hello"
print(c)  # hello


# 此外这个全局名字空间是唯一的，即使你把它放在函数中也是一样
def foo():
    globals()["d"] = "古明地觉"


# foo一旦执行，{"d": "古明地觉"}就设置进了全局名字空间中
foo()
print(d)  # 古明地觉
```

怎么样，是不是有点神奇呢？所以名字空间是 Python 作用域的灵魂，它严格限制了变量的活动范围，当然这些后面都会慢慢的说。目前算是回顾基础吧，虽说是基础但是其实也涉及到了一些解释器的知识，不过这一关我们迟早是要过的，所以就提前接触一下

吧。

收录于合集 #CPython 97

上一篇 · 《源码探秘 CPython》2. 解密 PyObject、PyVarObject、PyTypeObject

文章已于2022-01-03修改