

《源码探秘 CPython》12. 小整数对象池

原创 古明地觉 古明地觉的编程教室 2022-01-17 09:30



微信扫一扫
关注该公众号

收录于合集

#CPython

97个 >

由于分析过了浮点数以及浮点类型对象，因此int类型对象的实现以及int实例对象的创建啥的就不说了，可以自己去源码中查看，我们后面会着重介绍它的一些操作。

还是那句话，Python底层的API设计的很优美，都非常的相似，比如创建浮点数可以使用 `PyFloat_FromDouble`、`PyFloat_FromString` 等等，那么创建整数也可以使用 `PyLong_FromLong`、`PyLong_FromDouble`、`PyLong_FromString` 等等，直接去Objects中对应的源文件中查看即可。

这里说一下Python的小整数对象池，我们知道Python的整数属于不可变对象，运算之后会创建新的对象。

```
1 >>> a = 666
2 >>> id(a)
3 2431274354736
4 >>> a += 1
5 >>> id(a)
6 2431274355024
7 >>>
```

所以这种做法就势必会有性能缺陷，因为程序运行时会有大量对象的创建和销毁。根据浮点数的经验，我们猜测Python应该也对整数使用了缓存池吧。答案是差不多，只不过不是缓存池，而是小整数对象池。

Python将那些使用频率高的整数预先创建好，而且都是单例模式，这些预先创建好的整数会放在一个静态数组里面，我们称为小整数对象池。如果需要使用的会直接拿来用，而不用重新创建。注意：这些整数在Python解释器启动的时候，就已经创建了。

小整数对象池的实现位于 `longobject.c` 中。

```
1 #ifndef NSMALLPOSINTS
2 #define NSMALLPOSINTS      257
3 #endif
4 #ifndef NSMALLNEGINTS
5 #define NSMALLNEGINTS      5
6 #endif
7
8 static PyLongObject small_ints[NSMALLNEGINTS + NSMALLPOSINTS];
```

- **NSMALLPOSINTS**宏规定了对象池中正数的个数(从0开始，包括0)，默认257个；
- **NSMALLNEGINTS**宏规定了对象池中负数的个数，默认5个；
- **small_ints**是一个整数对象数组，保存预先创建好的小整数对象；

以默认配置为例，解释器在启动的时候就会预先创建一个可以容纳262个整数的数组，并会依次初始化 -5 到 256(包括两端)之间的262个PyLongObject。所以小整数对象池的结构如下：

-5	-4	-3	-2	-1	0	1	255	256
----	----	----	----	----	---	---	-------	-----	-----

古明地觉的Python小屋

但是为什么要实现缓存从-5到256之间的整数呢？因为Python认为这个范围内的整数使

用频率最高，而缓存这些整数的内存相对可控。因此这只是某种权衡，很多程序的开发场景都没有固定的正确答案，需要根据实际情况来权衡利弊。

```
1 >>> a = 256
2 >>> b = 256
3 >>> id(a), id(b)
4 (140714000246400, 140714000246400)
5 >>>
6 >>> a = 257
7 >>> b = 257
8 >>> id(a), id(b)
9 (2431274355184, 2431274354896)
10 >>>
```

256位于小整数对象池内，所以全局唯一，需要使用的话直接去取即可，因此它们的地址是一样的。但是257不在小整数对象池内，所以它们的地址不一样。

我们上面是在交互式下演示的，但如果有小伙伴不是通过交互式的话，那么会得到出乎意料的结果。

```
1 a = 257
2 b = 257
3 print(id(a) == id(b)) # True
```

可能有人会好奇，为什么地址又是一样的了，257明明不在小整数对象池中啊。虽然涉及到了后面的内容，但是提前解释一下也是可以的。主要区别就在于一个是在交互式下执行的，另一个是通过 python3 xxx.py的方式执行的。

首先Python的编译单元是函数，每个函数都有自己的作用域，在这个作用域中出现的所有常量都是唯一的，并且都位于常量池中，由co_consts指向。虽然我们上面的不是函数，而是在全局作用域中，但是全局你也可以看成是一个函数，它也是一个独立的编译单元。同一个编译单元中，常量只会出现一次。

当a = 257的时候，会创建257这个整数、并放入常量池中；所以b = 257的时候就不会再创建了，因为常量池中已经有了，所以会直接从常量池中获取，因此它们的地址是一样的，因为是同一个PyLongObject。

```
1 # Python3.6下执行，注意：该系列的所有代码都是基于Python3.8
2 # 但是这里先使用Python3.6，至于原因，后面会说
3 def f1():
4     a = 256
5     b = 257
6     return id(a), id(b)
7
8
9 def f2():
10    a = 256
11    b = 257
12    return id(a), id(b)
13
14
15 print(f1()) # (140042202371968, 140042204149712)
16 print(f2()) # (140042202371968, 140042204255024)
```

此时f1和f2显然是两个独立的编译单元，但256属于小整数对象池中的整数、全局唯一。因此即便不在同一个编译单元的常量池中，它的地址也是唯一的，因为是预先定义好的，所以会直接拿来用。但是257显然不是小整数对象池中的整数，而且不在同一个编译单元的常量池中，所以地址是不一样的。

而对于交互式环境来说，因为我们输入一行代码就会立即执行一行，所以任何一行可独立执行的代码都是一个独立的编译单元。注意：是可独立执行的代码，比如变量赋值、函数、方法调用等等；

但如果是if、for、while、def等等需要多行表示的话，比如：if 2 > 1:，显然这就不是

一行可独立执行的代码，它还依赖你输入的下面的内容。

```
1 # 此时按下回车, 我们看到不再是>>>, 而是...
2 # 这代表还没有结束, 还需要你下面的内容
3 >>> if 2 > 1:
4 ...     print("2 > 1")
5 ... # 此时这个if语句整体才是一个独立的编译单元
6 2 > 1
7 >>>
```

但是像 `a = 1`、`foo()`、`lst.append(123)` 这些显然它们是一行可独立执行的代码，因此在交互式它们是独立的编译单元。

```
1 # 此时这行代码已经执行了, 它是一个独立的编译单元
2 >>> a = 257
3 # 这行代码也是独立的编译单元, 所以它里面的常量池为空, 因此要重新创建
4 >>> b = 257
5 # 由于它们是不同的常量池内的整数, 所以id是不一样的。
6 >>> id(a), id(b)
7 (2431274355184, 2431274354896)
```

但是问题来了，看看下面的代码，`a`和`b`指向的对象的地址为啥又一样了？666和777明显也不在常量池中啊。

```
1 >>> a = 666; b=666
2 >>> id(a), id(b)
3 (2431274354896, 2431274354896)
4 >>> a, b = 777, 777
5 >>> id(a), id(b)
6 (2431274354800, 2431274354800)
7 >>>
```

显然此时应该已经猜到原因了，因为上面两种方式无论哪一种，都是在一行，因此整体会作为一个编译单元，所以地址是一样的。

然后将上面那个在 Python3.6 下执行的代码，拿到 Python3.8 执行一遍。

```
1 def f1():
2     a = 256
3     b = 2 ** 30
4     return id(a), id(b)
5
6
7 def f2():
8     a = 256
9     b = 2 ** 30
10    return id(a), id(b)
11
12
13 print(f1()) # (140714000246400, 2355781138896)
14 print(f2()) # (140714000246400, 2355781138896)
```

我们看到在 Python3.8 中，如果是通过 `python xxx.py` 的方式执行的话，即便是大整数、并且不在同一个编译单元的常量池中，它们的地址也是一样的，说明 Python 在 3.8 版本的时候做了优化。

注意：我们之前说创建完常量之后会放入常量池中，其实不够准确，因为常量池里面存储的并不是常量，而是指向常量的指针。

在 Python3.6 的时候，如果是大整数、并且不在同一个编译单元，那么两个编译单元中，常量池存储的指针指向的不是同一个对象。也就是说，`2**30` 会存在两份，每个指针指向不同的 `PyLongObject`（值为 `2**30`），即使它们的值相同。换言之，就是每个编译单元内都会创建 `2**30`。

而在 Python3.8 的时候做了优化，对于那些在编译期就能确定的常量，即使不在同一个编译单元中，那么也会只有一份。因此对于 3.8 而言，上面的 `2**30` 只会存在一份，两个指针指向的都是同一个 PyLongObject。

为了更好的理解，我们再反过来验证一下，不是说编译期就能确定的常量会只有一份吗？那如果编译期间无法确定的常量呢，会不会就不止一份了？我们测试一下：

```
1 def f1():
2     a = 256
3     b = 2 ** 30
4     return id(a), id(b)
5
6 def f2():
7     # eval("255 + 1") 等价于 256
8     # eval("2 ** 30") 等价于 2 ** 30
9     # 但它们在编译期间是无法确定的
10    a = eval("255 + 1")
11    b = eval("2 ** 30")
12    return id(a), id(b)
13
14
15 print(f1()) # (140729634596496, 2671447665136)
16 print(f2()) # (140729634596496, 2670821643760)
```

由于编译期间无法确定，那么只能在运行时动态创建。256 仍然只有一份，因为创建之后发现它位于小整数对象池中。但是 `2**30` 就不一样了，从打印的地址来看，它是存在两份的。因为动态创建的时候，发现当前的常量池中并没有，那么只能选择再创建一份。至于不同的编译单元共享常量（对象），必须是编译阶段就能确定的常量。

所以这就是 Python 在 3.8 的时候引入的一个优化机制，针对于编译期间就能够确定的常量（解释执行的时候不可以）。因为常量池中存储的实际上也是一个指针，指针指向堆区的某个对象，如果在创建的时候，发现其它的编译单元中已经在堆区创建了该对象，那么在当前的编译单元中就不会再创建了，在常量池中会直接保存已创建的对象指针。

但是在 3.6 的时候，则没有这个优化机制，它在某一个编译单元内创建常量的时候，不会参考其它的编译单元。换句话说，3.6 版本的解释器不关注你在其它的编译单元内创建了哪些常量，只要当前的编译单元内尚未创建该对象，那么就会重新创建，所以会存在值相同的对象在堆区被创建了两次。

因此 3.8 引入的这个机制还是比较有意义的，节省了内存的使用。

当然啦，这里还要再补充一点：我们上面虽然一直说的是 `2**30`，但 CPython 内部有一个 **常量折叠** 机制，在编译之后会将表达式替换为计算之后的值，所以在编译之后 `2**30` 会变成 `1073741824`。

注意：如果没有特殊说明，我们这个系列的所有代码都是在 Python3.8 下执行的，源码也是 3.8 版本。

说实话，我就是因为发现在 Python3.8 中，发现不同编译单元的大整数打印的地址都是一样的，才在上面试了一下 Python3.6。最终发现了，原来 CPython 在 3.8 的时候引入了这个优化机制。

以上就是小整数对象池相关的内容，比较简单，下一篇文章我们分析整数的运算，这也是最关键的地方。

收录于合集 [#CPython 97](#)

[< 上一篇](#)

《源码探秘 CPython》13. 整数在底层是如何进行大小比较的？

[下一篇 >](#)

《源码探秘 CPython》11. 整数是怎么设计的，为什么它不会溢出？

喜欢此内容的人还喜欢

浅谈Kotlin协程及首页弹窗中的应用
洋钱罐技术团队



20行Python代码破解了网站登入
Red Teams



DBPack 赋能 python 微服务协调分布式事务
DBMesh 技术

