《源码探秘 CPython》25. 列表是怎么实现的？解密列表的数据结构

原创 古明地觉 古明地觉的编程教室 2022-02-09 09:30

楔子

列表可以说使用的非常广泛了，在初学列表的时候，可能书上会告诉你列表就是一个大仓库，什么都可以存放。

但是在最开始的几个章节中，我们花了很大的笔墨介绍了Python的对象，并明白了Python变量的本质。我们知道列表中存放的元素其实都是泛型指针PyObject *，所以到现在列表已经没有什么好神秘的了。

并且根据我们使用列表的经验，可以得出以下两个结论：

- 每个列表的元素个数可以不一样，所以这是一个变长对象
- 可以对列表的元素进行添加、删除、修改等操作，所以这是一个可变对象

在分析列表对应的底层结构之前，我们先来回顾一下列表的使用。

```
# 创建一个列表，这里是通过Python/C API创建的
>>> lst = [1, 2, 3, 4]
>>> lst
[1, 2, 3, 4]

# 往列表尾部追加一个元素，此时是在本地操作的，返回值为None
# 但是列表被改变了
>>> lst.append(5)
>>> lst
[1, 2, 3, 4, 5]

# 从尾部弹出一个元素，会返回弹出的元素
>>> lst.pop()
5
# 此时列表也会被修改
>>> lst
[1, 2, 3, 4]
# 另外在pop的时候还可以指定索引，弹出指定位置的元素
>>> lst.pop(0)
1
>>> lst
[2, 3, 4]

# 也可以在指定位置插入一个元素
>>> lst.insert(0, 'x')
>>> lst
['x', 2, 3, 4]

# 通过extend在尾部追加多个元素
>>> lst.extend([7, 8, 9])
>>> lst
['x', 2, 3, 4, 7, 8, 9]

# 查找指定元素第一次出现的位置
>>> lst.index(3)
2

# 计算某个元素在列表中出现的次数
>>> lst.count(3)
```

```
2

# 翻转列表
>>> lst.reverse()
>>> lst
[9, 8, 7, 4, 3, 2, 'x']

# 根据元素的值删除第一个出现的元素
>>> lst.remove(4)
[9, 8, 7, 3, 2, 'x']

# 清空列表
>>> lst.clear()
>>> lst
[]
>>>
```

上面的一些操作是列表经常使用的，但是在分析它的实现之前，我们肯定要了解它们的时间复杂度如何。这些东西即使不看源码，也是必须要知道的，尤其想要成为一名优秀的Python工程师。

- append：会向尾部追加元素，所以时间复杂度为O(1)
- pop：默认从尾部弹出元素，所以时间复杂度为O(1);如果不是尾部，而是从其它的位置弹出元素的话，那么该位置后面所有的元素都要向前移动，此时时间复杂度为O(n)
- insert：向指定位置插入元素，该位置后面的所有元素都要向后移动，所以时间复杂度为O(n)

注意：由于列表里面的元素个数是可以自由变化的，所以列表有一个容量的概念，我们后面会说。当添加元素时，列表可能会扩容；同理当删除元素时，列表可能会缩容。

而容量一旦发生了改变，那么像append之类的时间复杂度为O(1)的操作，就会退化成O(n)，这是最坏情况。但是这种情况发生的频率不高，因此平均来算的话，还是O(1)。

列表的底层结构：PyListObject

列表在底层由**PyListObject**结构体表示，定义于头文件 Include/listobject.h 中：

```
typedef struct {
    PyObject_VAR_HEAD
    PyObject **ob_item;
    Py_ssize_t allocated;
} PyListObject;
```

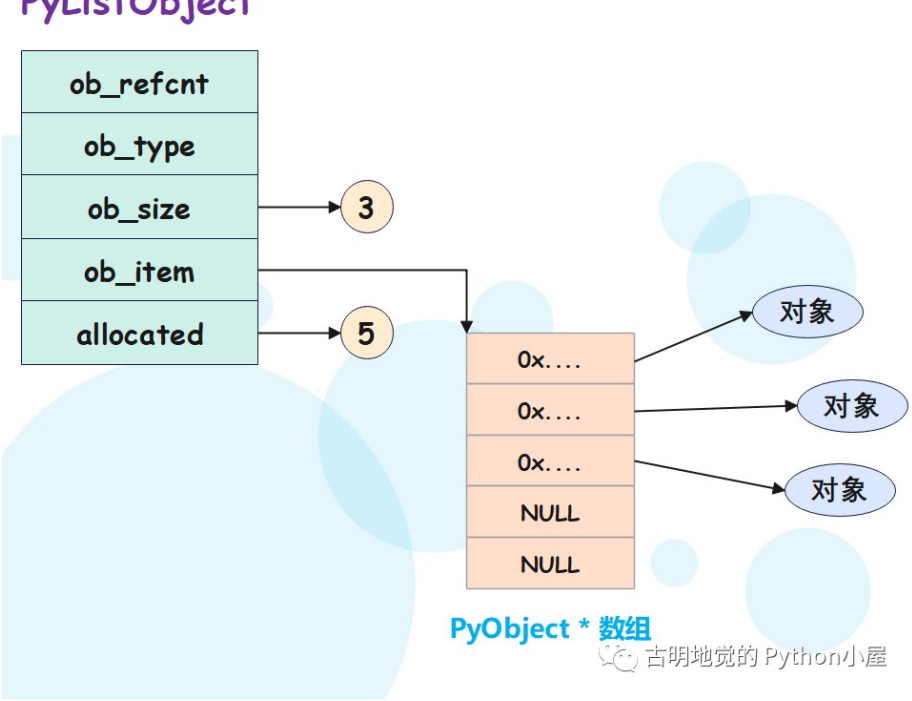
我们看到里面有如下成员：

- PyObject_VAR_HEAD：变长对象的公共头部信息
- ob_item：一个二级指针，指向PyObject *类型的指针数组，这个指针数组保存的便是对象的指针，而操作底层数组都是通过ob_item来进行操作的
- allocated：容量，我们知道列表底层是使用了C的数组，而底层数组的长度就是列表的容量

列表之所以要有容量的概念，是因为列表可以动态添加元素，但是底层的数组在创建完毕之后，其长度却是固定的。所以一旦添加新元素的时候，发现数组已经满了，这个时候只能申请一个更长的数组，同时把原来数组中的元素依次拷贝到新的数组里面(这一过程就是列表的扩容)，然后再将新元素添加进去。

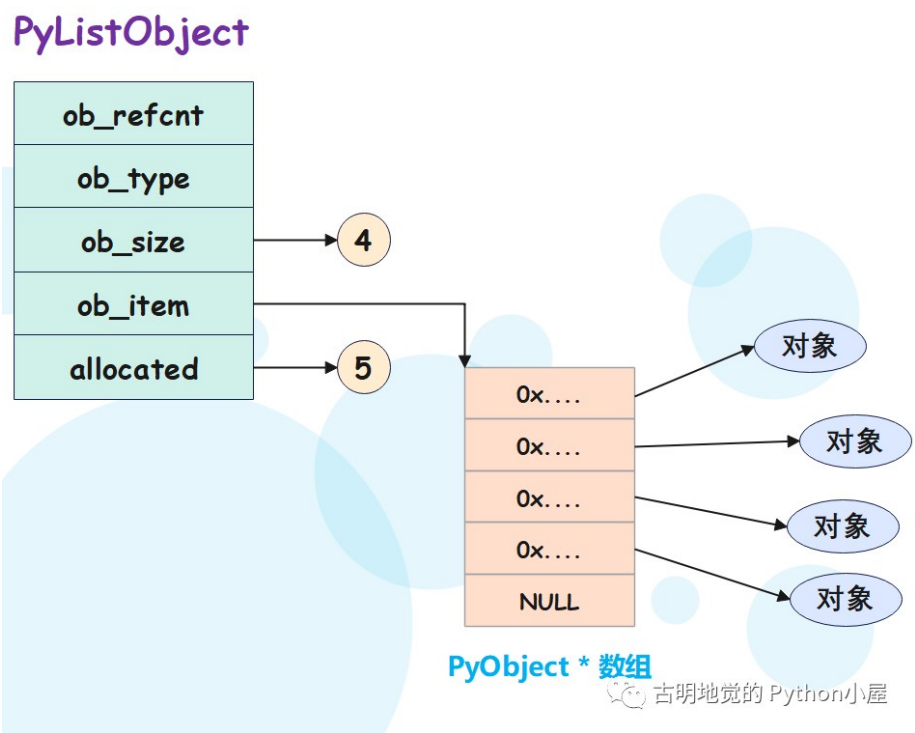
但是问题来了，总不可能每添加一个元素，就申请一次数组、将所有元素都拷贝一次吧。所以Python在列表扩容的时候，会将数组申请的长一些，可以在添加元素的时候不用每次都申请新的数组。

PyListObject

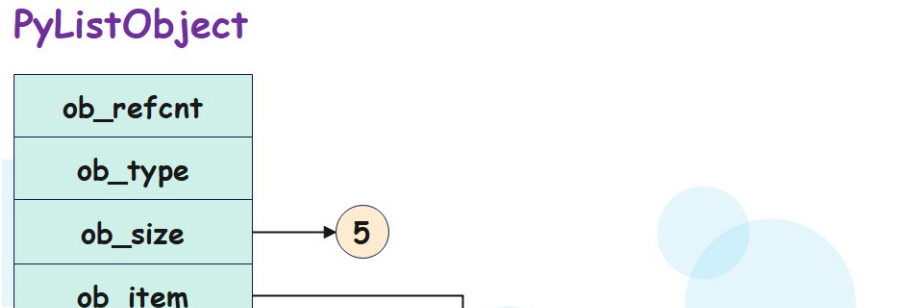


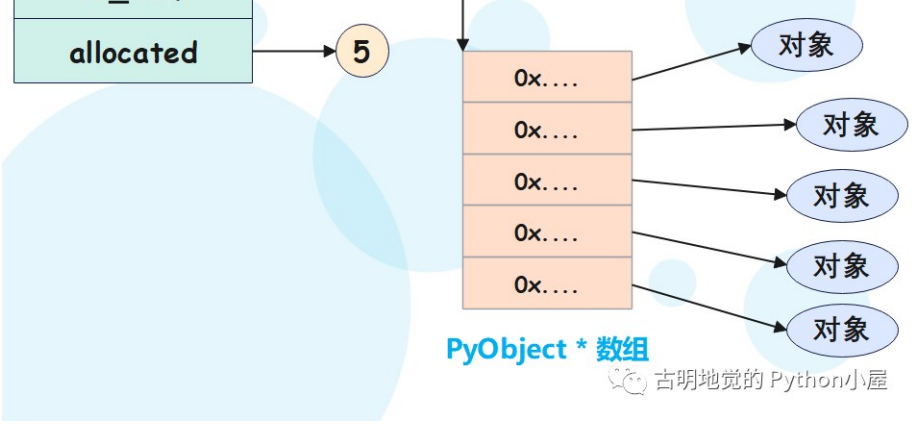
这便是列表的底层结构示意图，我们看到底层数组的长度为5，说明此时列表的容量为5，但是里面只有3个PyObject *指针，说明列表的ob_size是3，或者说列表里面此时有3个元素。

如果这个时候我们往列表中append一个元素，那么会将这个新元素设置在数组索引为ob_size的位置、或者说第四个位置。一旦设置完，ob_size会自动加1，因为ob_size要和列表的长度保持一致。

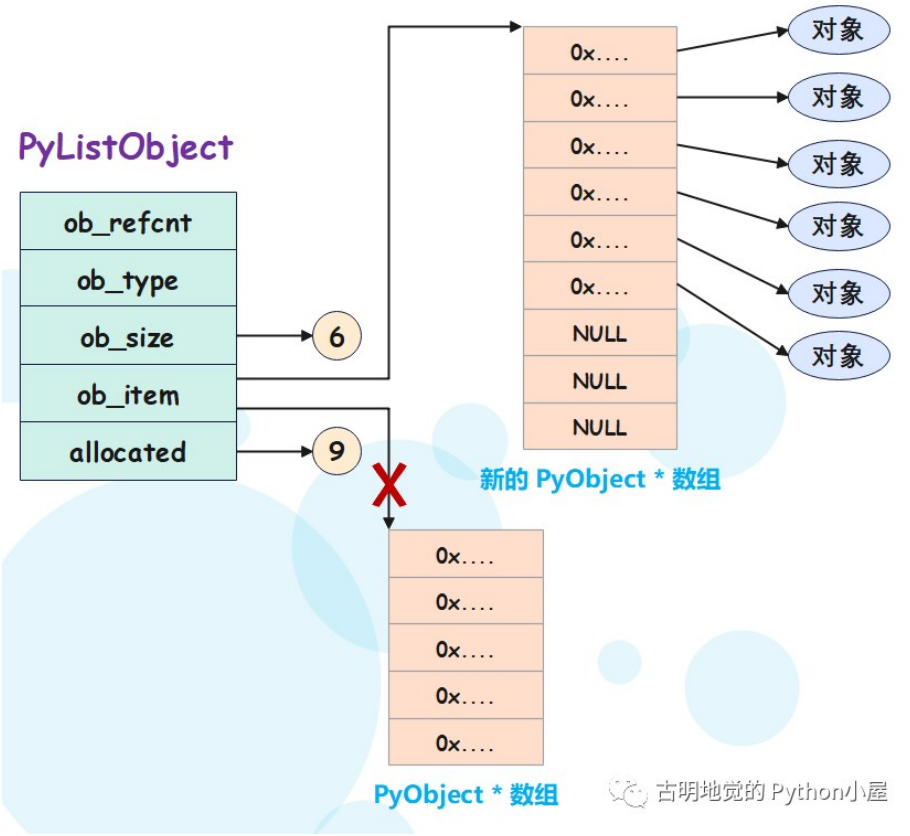


如果此时再往列表中append一个元素的话，那么还是将新元素设置在索引为ob_size的位置，此时也就是第5个位置。





列表的容量是5，但此时长度也达到了5，这说明当下一次append的时候，已经没有办法再容纳新的元素了。因为此时列表的长度、或者说元素个数已经达到了容量。当然最直观的还是这里的底层数组，很明显全都占满了。那这个时候如果想再接收新的元素的话，要怎么办呢？显然只能扩容了。



原来的容量是5个，长度也是5个，当再来一个新元素的时候由于没有位置了，所以要扩容。但是扩容的时候肯定会将容量申请的大一些、即底层数组申请的长一些，具体申请多长，Python内部有一个公式，我们后面会说。

总之申请的新的底层数组长度是9，那么说明列表的容量就变成了9。然后将原来数组中的**PyObject ***按照顺序依次拷贝到新的数组里面，再让ob_item指向新的数组。最后将要添加的新元素设置在新的数组中索引为ob_size的位置、即第6个位置，然后将ob_size加1即可，此时ob_size变成了6。

以上便是列表底层在扩容的时候所经历的过程。

> 由于扩容会申请新的数组，然后将旧数组的元素拷贝到新数组中，所以这是一个时间复杂度为O(n)的操作。而append可能会导致列表扩容，因此append最坏情况下也是一个O(n)的操作，只不过扩容不会频繁发生，所以append方法的平均时间复杂度还是O(1)。

另外我们还可以看到一个现象，那就是Python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ob_item这个二级指针来间接操作这个指针数组。

至于原因，我们在介绍Python对象模型的时候就说过了，不记得了的话，可以回去翻一翻。

所以底层对应的PyListObject实例的大小其实是不变的，因为指针数组没有存在PyListObject里面。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。

而且我们知道，列表在append之后地址是不变的，至于原因上面的几张图已经解释的很清楚了。

如果长度没有达到容量，那么append其实就是往底层数组中设置了一个新元素；如果达到容量了，那么会扩容，但扩容只是申请一个新的指针数组，然后让ob_item重新指向罢了。

所以底层的指针数组会变，但是PyListObject结构体实例本身是没有变化的。因此列表无论是append、extend、pop、insert等等，只要是在本地操作，那么它的地址是不会变化的。

下面我们再来看看列表所占的内存大小是怎么算的，首先PyListObject里面的PyObject_VAR_HEAD总共24字节，ob_item占8字节，allocated占8字节，总共40字节。

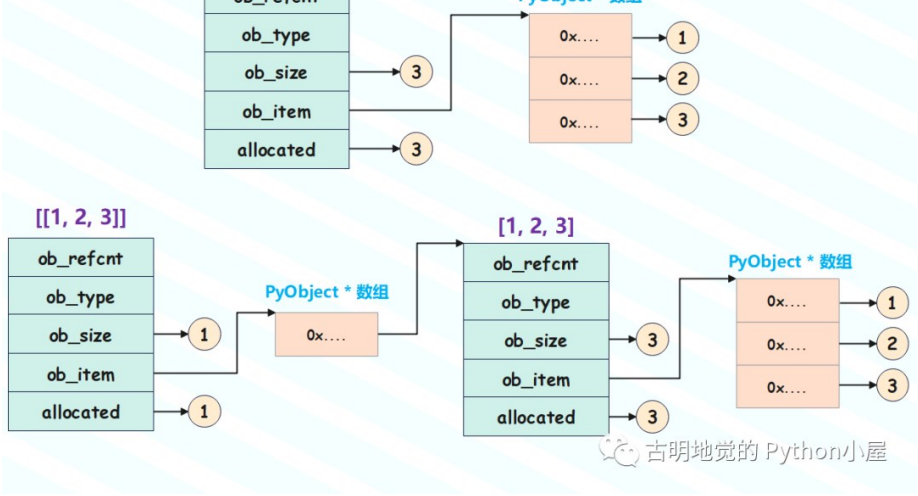
但是不要忘记，在计算列表大小的时候，ob_item指向的指针数组也要算在内。所以：**列表的大小 = 40 + 8 * 指针数组长度(或者列表容量)**。注意是指针数组长度、或者列表容量，可不是列表长度，因为数组一旦申请了，不管你用没用，大小就摆在那里了。就好比你租了间房子，就算不住，房租该交还是得交。

```
# 显然一个空数组占40个字节
print([].__sizeof__())  # 40

# 40 + 3 * 8 = 64
print([1, 2, "x" * 1000].__sizeof__())  # 64
#虽然里面有一个长度为1000的字符串
#但我们说列表存放的都是指针，所以大小都是8字节

#注意：我们通过lst = [1, 2, 3]这种方式创建列表的话
#不管内部元素有多少个，其ob_size和allocated都是一样的
#只有当列表在添加元素的时候发现容量不够了才会扩容
lst = list(range(10))
# 40 + 10 * 8 = 120
print(lst.__sizeof__())  # 120
# 这个时候append一个元素
lst.append(123)
print(lst.__sizeof__())  # 184
#我们发现大小达到了184，(184 - 40) // 8 = 18
#说明扩容之后申请的数组的长度为18
```

所以列表的大小我们就知道是怎么来的了，而且为什么列表在通过索引定位元素的时候，时间复杂度是O(1)。因为列表存储的都是对象的指针，不管对象有多大，其指针大小是固定的，都是8字节。通过索引可以瞬间计算出偏移量，从而找到对应元素的指针，而操作指针会自动操作指针所指向的内存。

```
print([1, 2, 3].__sizeof__())  # 64
print([[1, 2, 3]].__sizeof__())  # 48
```

相信上面这个结果，你肯定能分析出原因。因为第一个列表中有3个指针，所以大小是**40 + 24 = 64**；而第二个列表中有一个指针，所以是**40 + 8 = 48**。用一张图来展示一下**[1, 2, 3]**和**[[1, 2, 3]]**的底层结构，看看它们之间的区别：



到此相信你已经彻底掌握列表的结构了，那么下一篇我们来介绍列表支持的操作，看看这些操作在 C 里面是如何实现的。

收录于合集 #CPython 97

‹ 上一篇
《源码探秘 CPython》26. 列表是怎么实现扩容的?

下一篇 ›
《源码探秘 CPython》24. 字符串的intern机制