《源码探秘 CPython》29. 列表支持的操作（下）

原创 古明地觉 古明地觉的编程教室 2022-02-15 09:00

两个列表相加

两个列表之间可以进行加法运算，结果是一个新的列表，那么源码里面是如何实现的呢？

```c
static PyObject *
list_concat(PyListObject *a, PyObject *bb)
{
    //相加之后的列表长度
    Py_ssize_t size;
    //循环变量
    Py_ssize_t i;
    //两个二级指针，指向相应的ob_item
    PyObject **src, **dest;
    //相加之后创建的新列表
    PyListObject *np;
    //类型检测，注意这里的bb是一个PyObject *
    //所以它可以不是列表，但它的类型对象必须继承list
    //因此这里检测函数是PyList_Check，相当于isinstance(obj, list)
    //如果是PyList_CheckExactly，相当于type(obj) is list
    if (!PyList_Check(bb)) {
        PyErr_Format(PyExc_TypeError,
                     "can only concatenate list (not \"%.200s\") to list",
                     bb->ob_type->tp_name);
        return NULL;
    }
#define b ((PyListObject *)bb)
    //注意上面的宏，这里会将 bb 转成 PyListObject *

    //判断长度是否溢出
    if (Py_SIZE(a) > PY_SSIZE_T_MAX - Py_SIZE(b))
        return PyErr_NoMemory();
    //计算新列表的长度
    size = Py_SIZE(a) + Py_SIZE(b);
    //申请空间，其中指针数组的长度为size
    np = (PyListObject *) list_new_prealloc(size);
    //申请失败返回NULL
    if (np == NULL) {
        return NULL;
    }
    //先将 + 左边的列表里面的元素拷贝过去
    //获取a -> ob_item和np -> ob_item
    src = a->ob_item;
    dest = np->ob_item;
    //将元素依次拷贝过去，并增加引用计数
    for (i = 0; i < Py_SIZE(a); i++) {
        PyObject *v = src[i];
        Py_INCREF(v);
        dest[i] = v;
    }

    //再将 + 右边的列表里面的元素拷贝过去
    //获取b->ob_item和np->ob_item + Py_SIZE(a)
    //要从Py_SIZE(a)的位置开始设置，否则就把之前的元素覆盖掉了
```

```
    src = b->ob_item;
    dest = np->ob_item + Py_SIZE(a);
    //将元素依次拷贝过去，增加引用计数
    for (i = 0; i < Py_SIZE(b); i++) {
        PyObject *v = src[i];
        Py_INCREF(v);
        dest[i] = v;
    }
    //设置ob_size
    Py_SIZE(np) = size;
    //返回np
    return (PyObject *)np;
#undef b
}

```

以上就是列表的加法运算，逻辑就是创建一个新列表，然后将相加的两个列表里面的元素依次拷贝过去。

判断某个元素是否在列表中

对于一个序列来说，可以使用in操作符，等价于调用__contains__魔法方法。

```
static int
list_contains(PyListObject *a, PyObject *el)
{
    PyObject *item;
    Py_ssize_t i;
    int cmp;
    //挨个循环，比较是否相等，如果存在相等元素，cmp会等于1
    //因此cmp == 0 && i < Py_SIZE(a)不满足，直接返回
    //不相等则为0，会一直比完列表中所有的元素
    for (i = 0, cmp = 0 ; cmp == 0 && i < Py_SIZE(a); ++i) {
        item = PyList_GET_ITEM(a, i);
        Py_INCREF(item);
        cmp = PyObject_RichCompareBool(el, item, Py_EQ);
        Py_DECREF(item);
    }
    return cmp;
}
```

真的非常简单，没有什么好说的。

列表的深浅拷贝

列表的深浅拷贝也是初学者容易犯的错误之一，我们看一个Python的例子。

```
lst = [[]]

# 默认是浅拷贝，这个过程会创建一个新列表
# 但我们说列表l里面都是指针，因此只会将里面的指针拷贝一份
# 但是指针指向的内存并没有拷贝
lst_cp = lst.copy()

# 两个对象的地址是一样的
print(id(lst[0]), id(lst_cp[0]))  # 2207105155392 2207105155392

# 操作lst[0]，会改变lst_cp[0]
```

```
lst[0].append(123)
print(lst, lst_cp)  # [[123]] [[123]]

# 操作lst_cp[0], 会改变lst[0]
lst_cp[0].append(456)
print(lst, lst_cp)  # [[123, 456]] [[123, 456]]
```

我们通过索引或者切片也是一样的道理：

```
lst = [[], 1, 2, 3]
val = lst[0]
lst_cp = lst[0: 1]

print(lst[0] is val is lst_cp[0])  # True
# 此外, lst[:]等价于lst.copy()
```

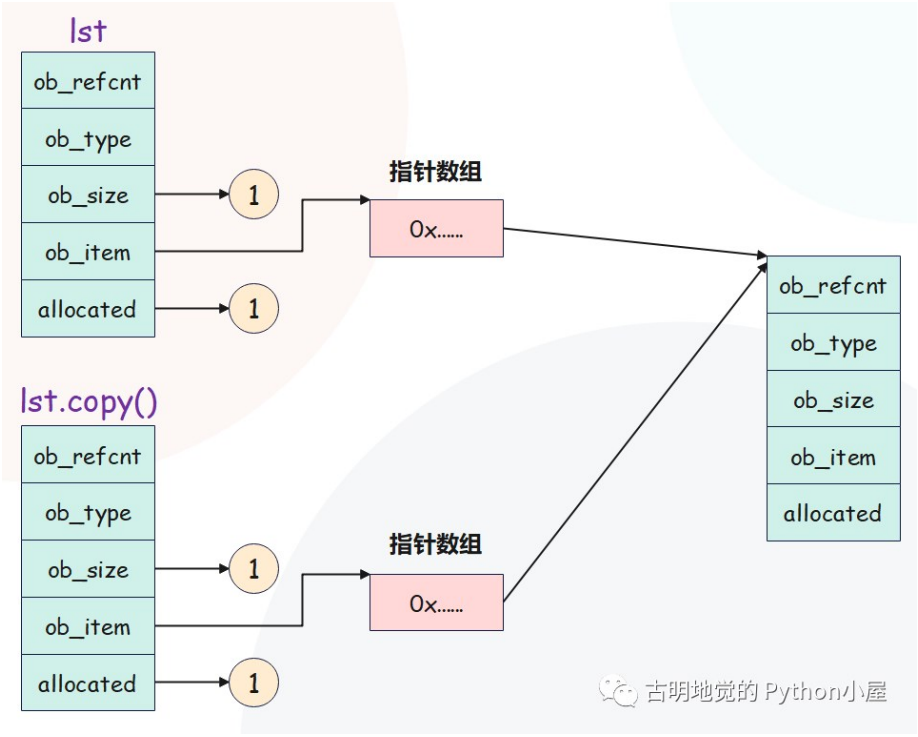
之所以会有这样现象，是因为Python的变量、容器里面的元素都是一个泛型指针 **PyObject ***，在传递的时候会传递指针， 但是在操作的时候会操作指针指向的内存。

所以lst.copy()就是创建了一个新列表，然后把元素拷贝了过去，只不过元素都是指针。因为只是拷贝指针，没有拷贝指针指向的对象(内存)，所以它们指向的是同一个对象。

但如果我们就想在拷贝指针的同时也拷贝指针指向的对象呢？答案是使用一个叫copy的模块。

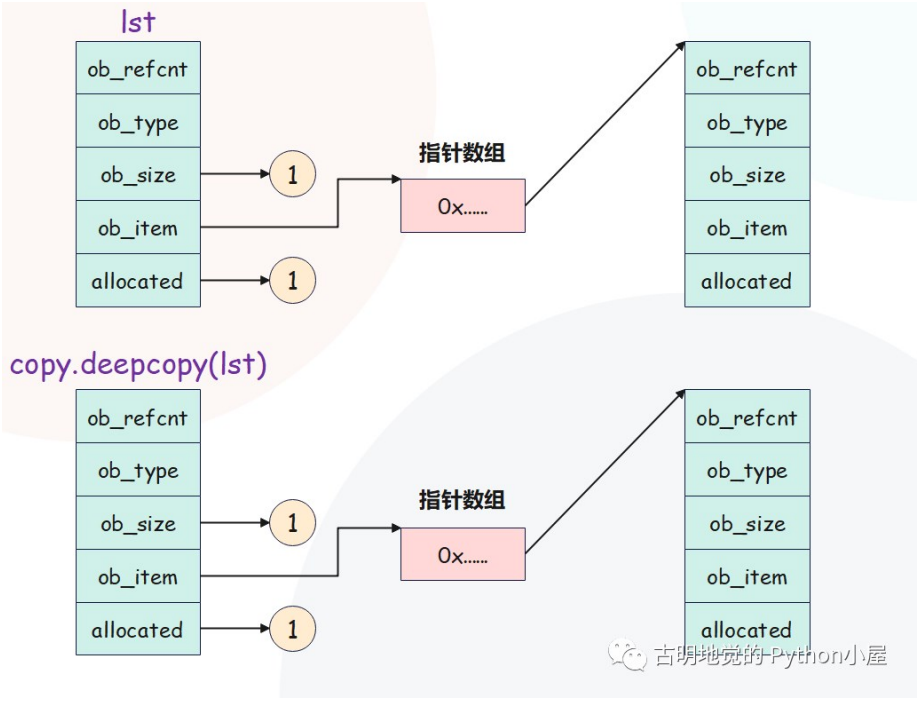
```
import copy

lst = [[]]
# 此时拷贝的时候, 就会把指针指向的对象也给拷贝一份
lst_cp1 = copy.deepcopy(lst)
lst_cp2 = lst[:]

lst_cp2[0].append(123)
print(lst)  # [[123]]
print(lst_cp1)  # [[]]

# lst[:]这种方式也是浅拷贝, 所以修改lst_cp2[0], 也会影响lst[0]
# 但是没有影响lst_cp1[0], 证明它们是相互独立的, 因为指向的是不同的对象
```

浅拷贝示意图如下：



里面的两个指针数组存储的元素是一样的，都是同一个对象的地址。

深拷贝示意图如下：



里面的两个指针数组存储的元素是不一样的，因为是不同对象的地址。

注意：copy.deepcopy虽然在拷贝指针的同时会将指针指向的对象也拷贝一份，但这仅仅是针对于**可变对象**，对于**不可变对象**是不会拷贝的。

```
import copy

lst = [[], "古明地觉"]
lst_cp = copy.deepcopy(lst)

print(lst[0] is lst_cp[0])  # False
print(lst[1] is lst_cp[1])  # True
```

为什么会这样，其实原因很简单。因为不可变对象是不支持本地修改的，你若想修改只能创建新的对象并指向它。但是这对其它的变量则没有影响，其它变量该指向谁就还指向谁。

因为**b = a**只是将**a存储的对象的指针**拷贝一份给b，然后a和b都指向了同一个对象，至于a和b本身则是没有任何关系的。如果此时a指向了新的对象，是完全不会影响b的，b还是指向原来的对象。

因此，如果一个指针指向的对象不支持本地修改，那么深拷贝不会拷贝对象本身，因为指向的是不可变对象，所以不会有修改一个影响另一个的情况出现。

关于列表还有一些陷阱：

```
lst = [[]] * 5
lst[0].append(1)
print(lst)  # [[1], [1], [1], [1], [1]]
# 列表乘上一个n，等于把列表里面的元素重复n次
# 但列表里面存储的是指针，也就是将指针重复n次
# 所以上面的列表里面的5个指针存储的地址是相同的
# 也就是说，它们都指向了同一个列表


# 这种方式创建的话，里面的指针都指向了不同的列表
lst = [[], [], [], [], []]
```

```
lst[0].append(1)
print(lst)  # [[1], [], [], [], []]


# 再比如字典，在后续系列中会说
d = dict.fromkeys([1, 2, 3, 4], [])
print(d)  # {1: [], 2: [], 3: [], 4: []}
d[1].append(123)
print(d)  # {1: [123], 2: [123], 3: [123], 4: [123]}
# 它们都指向了同一个列表
```

类似的陷阱还有很多，因此在工作中要注意，否则一不小心就会出现大问题。

总之记住三句话：**虽然Python中一切皆对象，但我们拿到的其实是指向对象的指针**；**变量在传递的时候本质上是将对象的指针拷贝一份，所以Python是变量的赋值传递、对象的引用传递**；**而在操作变量(指针)的时候，会操作变量(指针)指向的内存**。

小结

到此，关于列表的操作就介绍完了，我们用了三篇文章来介绍，因为列表用的非常频繁。在使用时容易掉入陷阱中，所以多花了一些笔墨去介绍它。

下一篇我们来介绍列表的创建和销毁，以及列表的缓存池是如何实现的。

收录于合集 #CPython 97

‹ 上一篇
《源码探秘 CPython》30. 列表的创建与销毁，以及缓存池机制

下一篇 ›
《源码探秘 CPython》28. 列表支持的操作（中）