11 list 对象，容量自适应的数组式容器-慕课专栏

笔者经常在面试中与候选人探讨 *Python* 内置容器对象， *list* 作为最常用容器中的一员，肯定少不了它：

- 你用过 *list* 对象吗？
- *list* 对象都支持哪些操作？时间复杂度、空间复杂度分别是多少？
- 试分析 *append* 和 *insert* 这两个方法的时间复杂度？
- 头部追加性能较差，如何解决？

list 对象的基本用法和技术原理，每个 *Python* 开发工程师都必须掌握，但不少候选人对此却只是一知半解。本节，我们一起查缺补漏，力求彻底掌握 *list* 对象，完美解答面试官针对 *list* 对象的灵魂之问。

基本用法

我们先来回顾一下 *list* 对象的基本操作：

```
>>> l = [1, 2, 3]
>>> l
[1, 2, 3]


>>> l.append(4)
>>> l
[1, 2, 3, 4]


>>> l.pop()
4
>>> l
[1, 2, 3]


>>> l.insert(0, 4)
>>> l
[4, 1, 2, 3]


>>> l.pop(0)
4
>>> l
[1, 2, 3]


>>> l.index(2)
1


>>> l.extend([1, 2])
>>> l
[1, 2, 3, 1, 2]


>>> l.count(1)
2
>>> l.count(3)
1


>>> l.reverse()
>>> l
[2, 1, 3, 2, 1]


>>> l.clear()
>>> l
[]
```

一个合格的 *Python* 开发工程师，除了必须熟练掌握 *list* 对象的基本操作，还需要对每个操作的 **实现原理** 及对应的 **时间复杂度** 、 **空间复杂度** 有准确的认识。列表操作总体比较简单，但有个操作特别容易被误用：

- *insert* 方法向头部追加元素时需要挪动整个列表，时间复杂度是 O(n)O(n)O(n) ，性能极差，需谨慎使用；
- *append* 方法向尾部追加元素时，无需挪动任何元素，时间复杂度 O(1)O(1)O(1) ；
- *pop* 方法从头部弹出元素时也需要挪动整个列表，时间复杂度是 O(n)O(n)O(n) ，同样需谨慎使用；
- *pop* 方法从尾部弹出元素时，无需挪动任何元素，时间复杂度是 O(1)O(1)O(1)

由此可见，对列表头部和尾部进行操作，性能有天壤之别。后续我们将一起探索 *list* 对象内部结构，从中寻找造成这种现象的原因。此外， *list* 对象还可根据元素个数 **自动扩缩容** ，其中秘密也将一一揭晓。

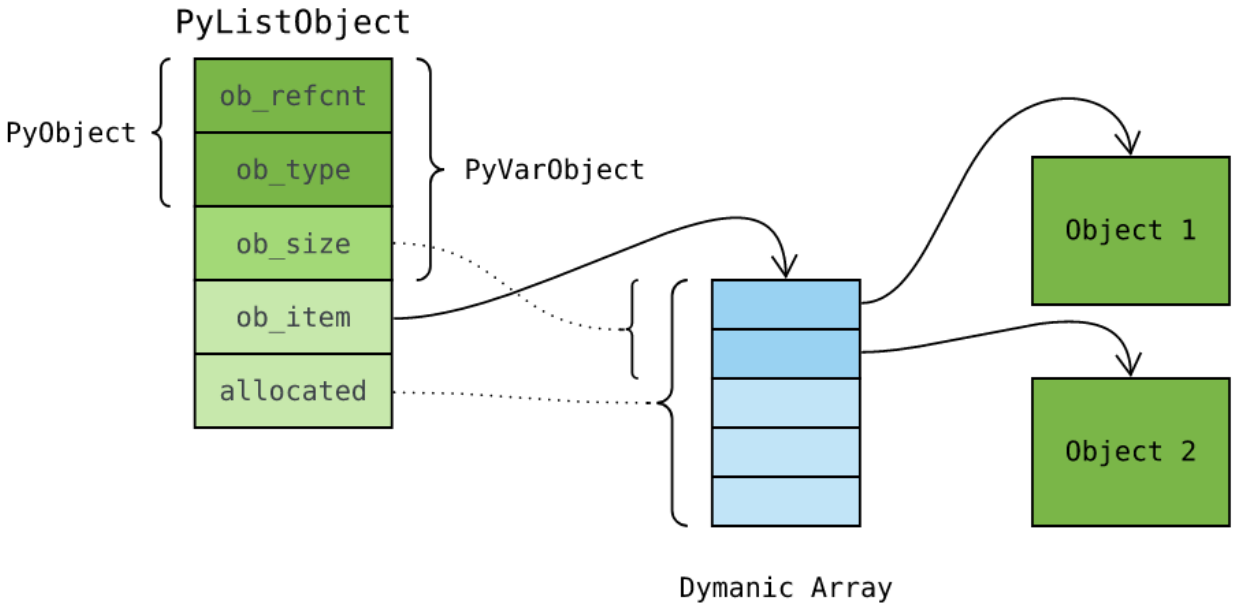
内部结构

list 对象在 *Python* 内部，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;
```

毫无疑问， *list* 对象是一种 **变长对象** ，因此包含变长对象公共头部。除了公共头部， *list* 内部维护了一个动态数组，而数组则依次保存元素对象的指针：

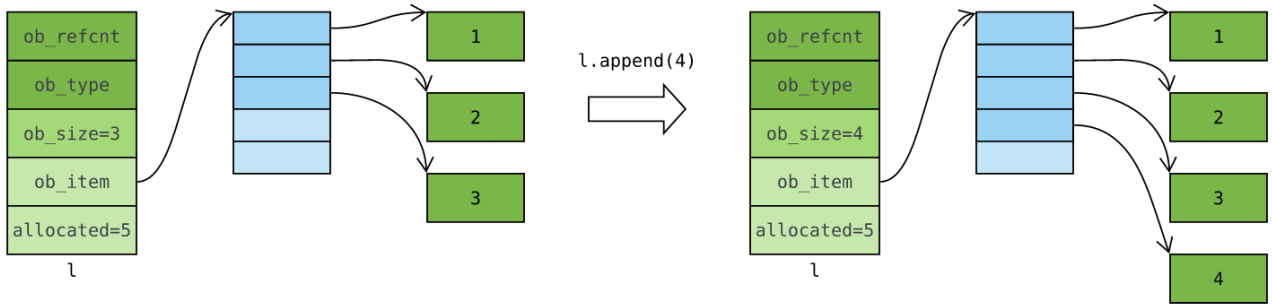
- ***ob_item*** ，指向动态数组的指针，动态数组保存元素对象的指针；
- ***allocated*** ，动态数组长度，即列表 **容量** ；
- ***ob_size*** ，动态数组当前保存元素个数，即列表 **长度** 。



尾部操作

在列表对象尾部增删元素，可快速完成，无须挪动其他元素。

假设列表元素 *l* 内部数组长度为 *5* ，以及保存 *3* 个元素，分别是： *1* 、 *2* 、 *3* 。当我们调用 *append* 方法向尾部追加元素时，由于内部数组还有未用满，只需将新元素保存于数组下一可用位置并更新 *ob_size* 字段：

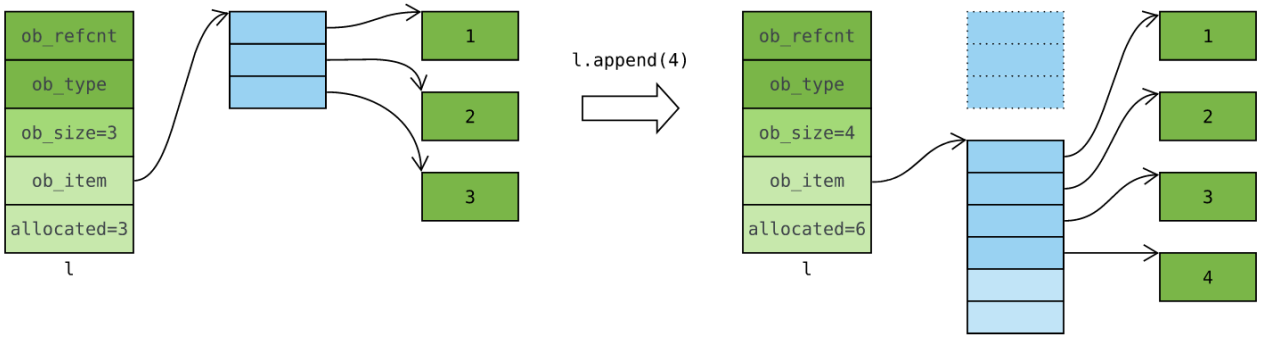


因此，大多数情况下， *append* 方法性能都足够好，时间复杂度是 O(1)O(1)O(1)。

动态扩容

如果 *list* 对象内部数组已用满，再添加元素时则需要进行扩容。 *append* 等方法在操作时都会对内部数组进行检查，如需扩容则调用 *list_resize* 函数。在 *list_resize* 函数， *Python* 重新分配一个长度更大的数组并替换旧数组。为避免频繁扩容， *Python* 每次都会为内部数组预留一定的裕量。

假设列表元素 *l* 保存 *3* 个元素，内部数组长度为 *3* ，已满。当我们调用 *append* 方法向列表尾部追加元素时，需要对内部数组进行扩容。扩容步骤如下：



1. 分配一个更大的数组，假设长度为 *6* ，预留一定裕量避免频繁扩容；
2. 将列表元素从旧数组逐一转移到新数组；
3. 以新数组替换旧数组，并更新 *allocated* 字段；
4. 回收旧数组。

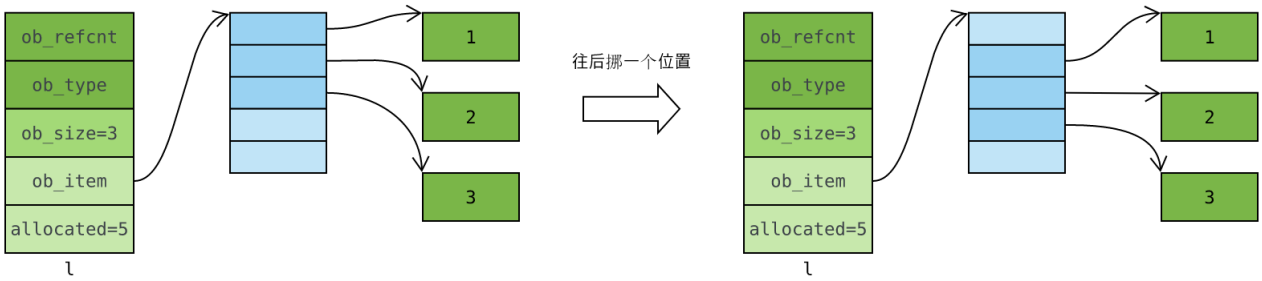
由于内部数组扩容时，需要将列表元素从旧数组拷贝到新数组，时间复杂度为 O(n)O(n)O(n) ，开销较大，需要尽量避免。为此， *Python* 在为内部数组扩容时，会预留一定裕量，一般是 1/81/81/8 左右。假设为长度为 *1000* 的列表对象扩容， *Python* 会预留大约 *125* 个空闲位置，分配一个长度 *1125* 的新数组。

由于扩容操作的存在， *append* 方法最坏情况下时间复杂度为 O(n)O(n)O(n) 。由于扩容操作不会频繁发生，将扩容操作时的元素拷贝开销平摊到多个 *append* 操作中，平均时间复杂度还是 O(1)O(1)O(1) 。

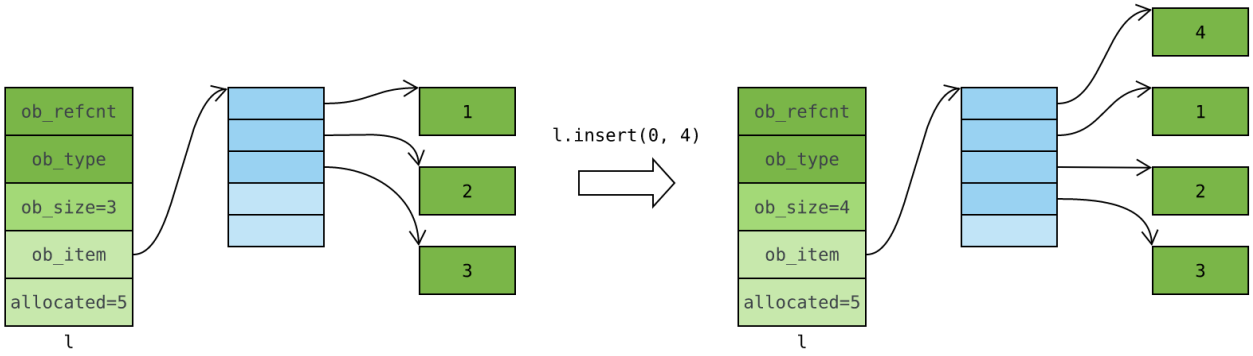
头部操作

与尾部相比，由于在列表头部增删元素需要挪动其他列表元素，性能有天地之别。

假设列表元素 *l* 内部数组长度为 *5* ，以及保存 *3* 个元素，分别是： *1* 、 *2* 、 *3* 。当我们调用 *insert* 方法向头部插入元素时，需要先将当前所有元素往后挪一位，以便为新元素腾出一个空闲位置：



然后， *insert* 方法将新元素存入挪出来的空闲位置，更新新 *ob_size* 字段，便完成了插入操作：



因此， *insert* 在头部插入元素的时间复杂度是 O(n)O(n)O(n) ，必须谨慎使用，尽量避免。调用 *pop* 方法从头部弹出元素，性能也是很差，时间复杂度同样是 O(n)O(n)O(n)。笔者见过有人将 *list* 对象当成一个 **队列** 来用，真想把他抓起来打一顿：

```
q = []


q.append(job)


job = q.pop(0)
```

这个队列实现，出队操作需要将整个队列挪动一遍，性能很差。如果队列规模很大，这将成为拖垮程序的关键因素。如果队列规模很小，这种写法虽说问题不大，但也不建议——你最好不要在代码中埋一颗不知什么时候会爆炸的雷。

如果你需要频繁操作列表头部，可以考虑使用标准库里的 *deque* ，这是一种 **双端队列** 结构。 *deque* 头部和尾部操作性能都很优秀，时间复杂度都是 O(1)O(1)O(1) 。如果你需要一个 **先进先出** (*FIFO*)队列，可以这么写：

```
from collections import deque

q = deque()


q.append(job)


q.popleft()
```

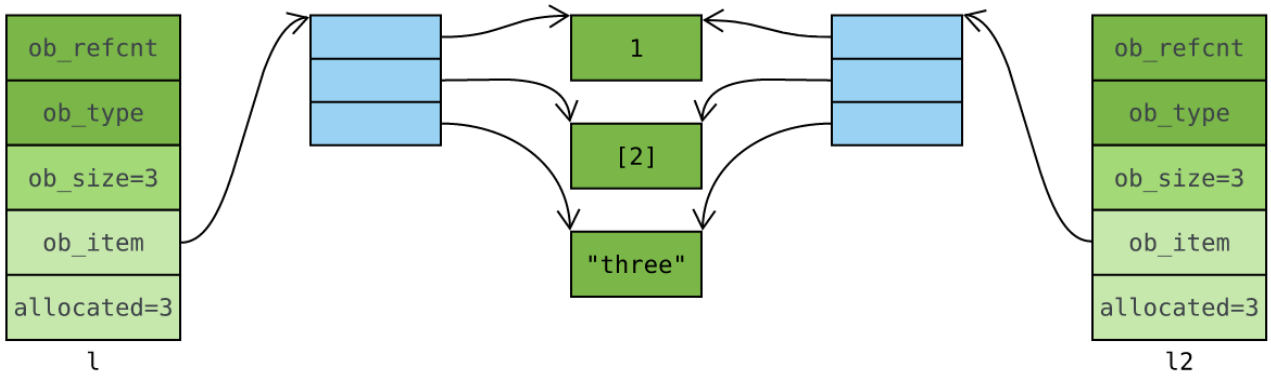
浅拷贝

调用 *list* 对象 *copy* 方法，可将列表拷贝一份，生成一个全新的列表：

```
>>> l = [1, [2], 'three']
>>> l
[1, [2], 'three']


>>> l2 = l.copy()
>>> l2
[1, [2], 'three']
>>> id(l), id(l2)
(4417331976, 4420505736)


>>> l2[0] = 'one'
>>> l2
['one', [2], 'three']
>>> l
[1, [2], 'three']


>>> l2[1][0] = 'two'
>>> l2
['one', ['two'], 'three']
>>> l
[1, ['two'], 'three']
```

由于 *copy* 方法只是对列表对象进行 **浅拷贝** ，对新列表可变元素的修改对旧列表可见！

如何理解浅拷贝呢？我们知道，列表对象内部数组保存元素对象的 **指针** ； *copy* 方法复制内部数组时，拷贝的也是元素对象的指针，而不是将元素对象拷贝一遍。因此，新列表对象与旧列表保存的都是同一组对象：



由此可见，*l* 和 *l2* 内嵌的列表对象其实是同一个，一旦对其进行修改，对 *l* 和 *l2* 都可见。

copy 方法实现的浅拷贝行为，可能不是你想要的。这时，可以通过 *copy* 模块里的 *deepcopy* 函数进行 **深拷贝** ：

```
>>> l = [1, [2], 'three']
>>> l
[1, [2], 'three']


>>> from copy import deepcopy
>>> l2 = deepcopy(l)
>>> l2
[1, [2], 'three']


>>> l2[1][0] = 'two'
>>> l2
[1, ['two'], 'three']
>>> l
[1, [2], 'three']
```

deepcopy 函数将递归复制所有容器对象，确保新旧列表不会包含同一个容器对象。这样一来，代码第 *13* 行处的修改，便对原列表不可见了。深拷贝的行为跟浅拷贝恰好相反。

浅拷贝 和 **深拷贝** 是 *Python* 面试中频繁考察的概念，必须完全掌握。

小结

本节我们一起回顾了 *list* 对象的典型用法并在此基础上研究其内部结构。 *list* 对象是一种 **变长对象** ，内部结构除了变长对象 **公共头部** 外，维护着一个 **动态数组** ，用于保存元素对象指针。其中，关键字段包括：

- *ob_item* ， **动态数组** 指针，数组保存元素对象指针；
- *allocated* ，动态数组长度，即列表 **容量** ；
- *ob_size* ，动态数组已保存元素个数，即列表 **长度** 。

Python 内部负责管理 *list* 对象的容量，在必要时 **自动扩缩容** ，极大降低开发人员的负担。

列表头部操作与尾部操作的性能差距非常大，而 *collections.deque* 作为替代品可解决列表头部操作的性能问题。列表 *copy* 方法只实现了 **浅拷贝** ，想要 **深拷贝** 只能借助 *copy.deepcopy* 函数。