CPython 中的列表扩容策略

Python 中的列表允许不断插入元素，且这个过程中列表对象的标识不变，因为它采用了分离式结构，即列表对象只保存与整个表有关的信息（容量和元素个数），实际元素（的引用）存放在另一个独立的元素存储区对象里。

在不断向一个列表里加入元素的过程中，元素存储区一定会被填满。这时候的操作过程如下：

1. 另外申请一块更大的元素存储区
2. 把表中已有的元素复制到新存储区
3. 用新的元素存储区替换原来的元素存储区（改变表对象的元素区链接）
4. 实际加入新元素

经过这几步操作，还是原来那个列表对象，但其元素存储区可以容纳更多元素。

假设随着操作的进行，列表的大小从 0 逐渐扩大到 n。考虑后端插入的情况。由于不需要移动元素，一次操作本身的复杂度是 $O(1)$。但在这里还有一个情况需要考虑：在连续加入了一些数据项后，列表的当前元素存储区终将被填满，这时就需要换一块存储区。更换存储区时需要复制表中的元素，整个复制需要 $O(m)$ 时间（其中 m 是当时的元素个数）。这样就出现了一个问题，即应该怎样选择新存储区的大小呢？这件事牵涉到空闲存储单元的量和替换存储的频度问题。

考虑一种简单策略：每次替换存储时增加 10 个元素存储位置。这种策略可称为*线性增长*，10 是增长的参数。假设表长（表中元素个数）从 0 不断增长到 1000，每加入 10 个元素就要换一次存储，复制当时的所有元素。总的元素复制次数是：
$10 + 20 + 30 + \ldots + 990 = 10 \times \sum_{i=1}^{99} i = 49500$
对一般的 n，总的元素复制次数大约是$10 + 20 + \ldots + (n-10) = 10 \times \sum_{i=1}^{\frac{n-10}{10}} i = \frac{n(n-10)}{20} \approx \frac{n^2}{20}$。也就是说，虽然每次做尾端插入的代价是 $O(1)$，加上元素复制之后，一般而言，执行一次插入操作的平均代价还是达到了 $O(n)$。

这里总操作开销比较高，是因为在不断插入元素的过程中频繁替换元素存储区，而且每次替换需要复制的元素越来越多。修改替换时增加的空位数，例如把 10 改成 100，只能减小 n^2 的常量系数，不能带来本质的改进。应该考虑一种策略，其中随着元素数量的增加，替换存储区的频率不断降低。下面看看 cpython 中是怎么做的：

列表的实现在 cpython 项目的`Objects/listobject.c`文件中，其中调整列表大小的函数为`list_resize`，摘录如下：

```
static int
list_resize(PyListObject *self, Py_ssize_t newsize)
{
    PyObject **items;
    size_t new_allocated, num_allocated_bytes;
    Py_ssize_t allocated = self->allocated;

    /* Bypass realloc() when a previous overallocation is large enough
       to accommodate the newsize.  If the newsize falls lower than half
       the allocated size, then proceed with the realloc() to shrink the list.
    */
    if (allocated >= newsize && newsize >= (allocated >> 1)) {
        assert(self->ob_item != NULL || newsize == 0);
        Py_SIZE(self) = newsize;
        return 0;
    }
```

```
    /* This over-allocates proportional to the list size, making room
     * for additional growth.  The over-allocation is mild, but is
     * enough to give linear-time amortized behavior over a long
     * sequence of appends() in the presence of a poorly-performing
     * system realloc().
     * The growth pattern is:  0, 4, 8, 16, 25, 35, 46, 58, 72, 88, ...
     * Note: new_allocated won't overflow because the largest possible value
     *       is PY_SSIZE_T_MAX * (9 / 8) + 6 which always fits in a size_t.
     */
    new_allocated = (size_t)newsize + (newsize >> 3) + (newsize < 9 ? 3 : 6);
    if (new_allocated > (size_t)PY_SSIZE_T_MAX / sizeof(PyObject *)) {
        PyErr_NoMemory();
        return -1;
    }

    if (newsize == 0)
        new_allocated = 0;
    num_allocated_bytes = new_allocated * sizeof(PyObject *);
    items = (PyObject **)PyMem_Realloc(self->ob_item, num_allocated_bytes);
    if (items == NULL) {
        PyErr_NoMemory();
        return -1;
    }
    self->ob_item = items;
    Py_SIZE(self) = newsize;
    self->allocated = new_allocated;
    return 0;
}
```

根据元素数量计算需要分配存储空间的语句是`new_allocated = (size_t)newsize + (newsize >> 3) + (newsize < 9 ? 3 : 6);`，由前面的条件语句可知只有当元素数量超过当前已分配的存储空间时才会重新进行分配，可以算出在不断后端插入过程中分配的元素存储区空间分别为`0, 1+1>>3+3=4, 5+5>>3+3=8, 9+9>>3+6=16, 17+17>>3+6=25, 26+26>>3+6=35, ...`，这边每次计算时的`newsize`都要比前一次计算得到的存储空间大1。

代码中的`newsize >> 3`，也就是`newsize // 8`，保证了每次扩容都和当时的存储区大小成比例关系，这样平摊下来每次操作的平均复杂度还是能达到$O(1)$，并且也没有双倍扩容那么浪费空间。

下面我们在 cpython 中验证一下，`sys.getsizeof`函数可以获取一个对象占用的内存大小（位），我们首先初始化一个空列表，并检查与整个表有关的信息（容量和元素个数）所占空间大小：

```
>>> import sys
>>> a = []
>>> sys.getsizeof(a)
64
```

所以当存储区为空时整个表占用 8 字节，接下来，只要将新获取的值减去 64 再除以 8 就可以得到当时的存储区的大小了，下面是完整的代码：

```
import sys

def get_storage_byte(li):
    return (sys.getsizeof(li) - 64) // 8

a = []
start = 0
```

```
for i in range(1000):
    a.append(i)
    storage = get_storage_byte(a)
    if storage != start:
        print(start, end=' ')
        start = storage
```

以下为输出结果`0 4 8 16 25 35 46 58 72 88 106 126 148 173 201 233 269 309 354 405 462 526 598 679 771 874 990`，可以看到与`list_resize`函数中的注释 *The growth pattern is: 0, 4, 8, 16, 25, 35, 46, 58, 72, 88, ...* 保持一致。