12 list 源码解析：动态数组精讲-慕课专栏

list 对象是一种 **容量自适应** 的 **线性容器** ，底层由 **动态数组** 实现。动态数组结构决定了 *list* 对象具有优秀的尾部操作性能，但头部操作性能却很差劲。研发人员只有对底层数据结构有足够的认识，才能最大限度避免问题代码。

现成的动态数组实现很多，除了我们正在研究的 *list* 对象，*C++* 中的 vector 也是众所周知。虽然在实际项目中需要自行实现动态数组的场景已经很少很少了，但是源码还是有必要研究一番。源码研究不仅能加深对数据结构的理解，还能进一步提升编程水平，裨益颇多。

本节，我们开始 *list* 对象源码，深入学习 **动态数组** 实现的艺术。

容量调整

当我们调用 *append* 、*pop* 、*insert* 等方法时，列表长度随之发生变化。当列表长度超过底层数组容量时，便需要对底层数组进行 **扩容** ；当列表长度远低于底层数组容量时，便需要对底层数组进行 **缩容** 。

Objects/listobject.c 源码表明，*append* 等方法依赖 *list_resize* 函数调整列表长度，扩容缩容的秘密就藏在这里！*list_resize* 函数在调整列表长度前，先检查底层数组容量，并在必要时重新分配底层数组。接下来，我们一起来解读*list_resize* 函数，该函数同样位于源文件 *Objects/listobject.c* 中：

```
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;


    if (allocated >= newsize && newsize >= (allocated >> 1)) {
        assert(self->ob_item != NULL || newsize == 0);
        Py_SIZE(self) = newsize;
        return 0;
    }


    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;
}
```

在函数开头，有几个局部变量定义，对理解函数逻辑非常关键：

- *items* 指针，用于保存新数组；
- *new_allocated*，用于保存新数组容量；
- *num_allocated_bytes*，用于保存新数组内存大小，以字节为单位；
- *allocated*，用于保存旧数组容量。

然后，代码第 *12* 行，检查新长度与底层数组容量的关系。如果新长度不超过数组容量，且不小于数组容量的一半，则无需调整底层数组，直接更新 *ob_size* 字段。换句话讲， *list* 对象扩缩容的条件分别如下：

- **扩容条件**，新长度大于底层数组长度；
- **缩容条件**，新长度小于底层数组长度的一半；

扩容或缩容条件触发时，*list_resize* 函数根据新长度计算数组容量并重新分配底层数组（第 *27-44* 行）：

1. 第 *27* 行，新容量在长度加上 \frac{1}{8}81 的裕量，再加上 *3* 或 *6* 的裕量；

2. 第 *28-31* 行，如果新容量超过允许范围，返回错误；
3. 第 *33-34* 行，如果新长度为 *0* ，将新容量也设置为 *0* ，因此空列表底层数组亦为空；
4. 第 *36-40* 行，调用 *PyMem_Realloc* 函数重新分配底层数组；
5. 第 *41-44* 行，更新 *3* 个关键字段，依次设置为 **新底层数组** 、 **新长度** 以及 **新容量** 。

注意到代码第 *27* 行，新容量的计算公式有点令人费解。为什么还要加上 *3* 或者 *6* 的裕量呢？试想一下，如果新长度小于 *8* ，那么 \frac{1}{8}81 的裕量便是 *0* ！这意味着，当 *list* 对象长度从 *0* 开始增长时，需要频繁扩容！

为了解决这个问题，必须在 \frac{1}{8}81 裕量的基础上额外加上一定的固定裕量。而 *3* 和 *6* 这两个特殊数值的选择，使得列表容量按照 *0*、*4*、*8*、 16、*25*、*35*、*46*、*58*、*72*、*88*……这样的序列进行扩张。这样一来，当 *list* 对象长度较小时，容量翻倍扩展，扩容频率得到有效限制。

顺便提一下， *PyMem_Realloc* 函数是 *Python* 内部实现的内存管理函数之一，功能与 *C* 库函数 *realloc* 类似：

```
PyAPI_FUNC(void *) PyMem_Realloc(void *ptr, size_t new_size);
```

PyMem_Realloc 函数用于对动态内存进行扩容或者缩容，关键步骤如下：

1. 新申请一块尺寸为 *new_size* 的内存区域；
2. 将数据从旧内存区域 *ptr* 拷贝到新内存区域；
3. 释放旧内存区域 *ptr* ；
4. 返回新内存区域。

内存管理 是最考验研发人员编程功底的领域之一，鼓励大家到 *PyMem_Realloc* 源码（*./Objects/obmalloc.c*）中进一步研究内存管理的技巧。学有余力的童鞋，可模仿着自己实现一个 *realloc* 函数，假以时日编程内功将突飞猛进！

append 方法在 *Python* 内部由 *C* 函数 *list_append* 实现，而 *list_append* 进一步调用 *app1* 函数完成元素追加：

```
static int
app1(PyListObject *self, PyObject *v)
{
    Py_ssize_t n = PyList_GET_SIZE(self);

    assert (v != NULL);
    if (n == PY_SSIZE_T_MAX) {
        PyErr_SetString(PyExc_OverflowError,
            "cannot add more objects to list");
        return -1;
    }

    if (list_resize(self, n+1) < 0)
        return -1;

    Py_INCREF(v);
    PyList_SET_ITEM(self, n, v);
    return 0;
}
```

1. 第 *4* 行，调用 *PyList_GET_SIZE* 取出列表长度，即 *ob_size* 字段；
2. 第 *7-11* 行，判断列表当前长度，如果已经达到最大限制，则报错；
3. 第 *13-15* 行，调用 *list_resize* 更新列表长度，必要时 *list_resize* 对底层数组进行 **扩容**；
4. 第 *16* 行，自增元素对象 **引用计数** (元素对象新增一个来自列表对象的引用)；
5. 第 17 行，将元素对象指针保存到列表最后一个位置，列表新长度为 *n+1* ，最后一个位置下标为 *n* 。

我们看到，有了 *list_resize* 这个辅助函数后， *app1* 函数的实现就非常直白了。接下来，我们将看到 *insert*、*pop* 等方法的实现中也用到这个函数，从中可体会到程序逻辑 **划分** 、 **组合** 的巧妙之处。

insert 方法在 *Python* 内部由 *C* 函数 *list_insert_impl* 实现，而 *list_insert_impl* 则调用 *ins1* 函数完成元素插入：

```
static int
ins1(PyListObject *self, Py_ssize_t where, PyObject *v)
{
    Py_ssize_t i, n = Py_SIZE(self);
    PyObject **items;
    if (v == NULL) {
        PyErr_BadInternalCall();
        return -1;
    }
    if (n == PY_SSIZE_T_MAX) {
        PyErr_SetString(PyExc_OverflowError,
            "cannot add more objects to list");
        return -1;
    }

    if (list_resize(self, n+1) < 0)
        return -1;

    if (where < 0) {
        where += n;
        if (where < 0)
            where = 0;
    }
    if (where > n)
        where = n;
    items = self->ob_item;
    for (i = n; --i >= where; )
        items[i+1] = items[i];
    Py_INCREF(v);
    items[where] = v;
    return 0;
}
```

1. 第 *4* 行，调用 *PyList_GET_SIZE* 取出列表长度，即 *ob_size* 字段；
2. 第 *10-14* 行，判断列表当前长度，如果已经达到最大限制，则报错；
3. 第 *16-17* 行，调用 *list_resize* 更新列表长度，必要时 *list_resize* 对底层数组进行 **扩容**；

4. 第 *19-23* 行，检查插入位置下标，如果下标为负数，加上 *n* 将其转换为非负数；
5. 第 *21-22*、*24-25* 行，检查插入位置下标是否越界，如果越界则设为开头或结尾；
6. 第 *26-28* 行，将插入位置以后的所有元素逐一往后移一个位置，特别注意 *for* 循环必须 **从后往前** 迭代；
7. 第 *29* 行，自增元素对象 **引用计数** (元素对象新增一个来自列表对象的引用)；
8. 第 *30* 行，将元素对象指针保存到列表指定位置。

Python 序列 **下标很有特色**，除了支持 *0~n-1* 这样的惯例外，还支持 **倒数下标**。倒数下标为负数，从后往前数：最后一个元素为 *-1*，倒数第二个为 *-2*；以此类推，第一个元素下标为：*-n*。

倒数下标非常实用，可以很方便地取出序列最后几个元素，而不用关心序列的长度。*Python* 内部处理倒数下标时，自动为其加上长度序列 *n*，便转化成普通下标了。

pop 方法将指定下标的元素从列表中弹出，下标默认为 *-1*。换句话讲，如果未指定下标，*pop* 弹出最后一个元素：

```
>>> help(list.pop)
Help on method_descriptor:

pop(self, index=-1, /)
    Remove and return item at index (default last).

    Raises IndexError if list is empty or index is out of range.
```

pop 方法在 *Python* 内部由 *C* 函数 *list_pop_impl* 实现：

```
static PyObject *
list_pop_impl(PyListObject *self, Py_ssize_t index)
{
    PyObject *v;
    int status;

    if (Py_SIZE(self) == 0) {

        PyErr_SetString(PyExc_IndexError, "pop from empty list");
        return NULL;
    }
    if (index < 0)
        index += Py_SIZE(self);
    if (index < 0 || index >= Py_SIZE(self)) {
        PyErr_SetString(PyExc_IndexError, "pop index out of range");
        return NULL;
    }
    v = self->ob_item[index];
    if (index == Py_SIZE(self) - 1) {
        status = list_resize(self, Py_SIZE(self) - 1);
        if (status >= 0)
            return v;
        else
            return NULL;
    }
    Py_INCREF(v);
    status = list_ass_slice(self, index, index+1, (PyObject *)NULL);
    if (status < 0) {
        Py_DECREF(v);
        return NULL;
    }
    return v;
}
```

1. 第 *7-11* 行，如果列表为空，没有任何元素可弹出，抛出 *IndexError* 异常；
2. 第 *12-13* 行，如果给定下标为 **倒数下标** ，先加上列表长度，将其转换成普通下标；
3. 第 *14-16* 行，检查给定下标是否在合法范围内，超出合法范围同样抛出 *IndexError* 异常；
4. 第 *18* 行，从底层数组中取出待弹出元素；
5. 第 *19-25* 行，如果待弹出元素为列表最后一个，调用 *list_resize* 快速调整列表长度即可，无需移动其他元素；
6. 第 *26-31* 行，其他情况下调用 *list_ass_slice* 函数删除元素，调用前需要通过 *Py_INCREF* 增加元素引用计数，因为 *list_ass_slice* 函数内部将释放被删除元素；
7. 第 32 行，将待弹出元素返回。

list_ass_slice 函数其实有两种不同的语义，具体执行哪种语义由函数参数决定，函数接口如下：

```
static int
list_ass_slice(PyListObject *a, Py_ssize_t ilow, Py_ssize_t ihigh, PyObject *v);
```

- **删除语义** ，如果最后一个参数 *v* 值为 *NULL* ，执行删除语义，即：*del a[ilow:ihigh]* ；

- **替换语义**，如果最后一个参数 *v* 值不为 *NULL*，执行替换语义，即 *a[ilow:ihigh] = v* 。

因此，代码第 *27* 行中， *list_ass_slice* 函数执行删除语义，将 *[index, index+1)* 范围内的元素删除。由于半开半闭区间 *[index, index+1)* 中只包含 *index* 一个元素，效果等同于将下标为 *index* 的元素删除。

执行删除语义时， *list_ass_slice* 函数将被删元素后面的元素逐一往前移动，以便重新覆盖删除操作所造成的空隙。由此可见，*pop* 方法弹出元素，时间复杂度跟弹出位置有关：

- 最好时间复杂度(**尾部弹出**)，O(1)O(1)；
- 最坏时间复杂度(**头部弹出**)，O(n)O(n)；
- 平均时间复杂度， O(\frac{n}{2})O(2n)，亦即 O(n)O(n) 。

因此，调用 *pop* 方法弹出非尾部元素时，需要非常谨慎。

remove 方法将给定元素从列表中删除。与 *pop* 略微不同，*remove* 方法直接给定待删除元素，而不是元素下标。*remove* 方法在 *Python* 内部由 *C* 函数 *list_remove* 实现：

```
static PyObject *
list_remove(PyListObject *self, PyObject *value)

{
    Py_ssize_t i;

    for (i = 0; i < Py_SIZE(self); i++) {
        int cmp = PyObject_RichCompareBool(self->ob_item[i], value, Py_EQ);
        if (cmp > 0) {
            if (list_ass_slice(self, i, i+1,
                               (PyObject *)NULL) == 0)
                Py_RETURN_NONE;
            return NULL;
        }
        else if (cmp < 0)
            return NULL;
    }
    PyErr_SetString(PyExc_ValueError, "list.remove(x): x not in list");
    return NULL;
}
```

list_remove 函数先遍历列表中每个元素（第 *7* 行），检查元素是否为待删除元素 *value* （第 *8* 行），以此确定下标。然后， *list_remove* 函数调用 *list_ass_slice* 函数进行删除。注意到，如果给定元素不存在， *list_remove* 将抛出 *ValueError* 异常。

由此可见，*remove* 方法在删除前有一个时间复杂度为 O(n)O(n) 的查找过程，性能不甚理想，须谨慎使用。

list 对象是一种 **容量自适应** 的 **线性容器** ，底层由 **动态数组** 实现。*Python* 内部由函数 *list_resize* 调整列表长度， *list_resize* 自动为列表进行 **扩容** 或者 **缩容** ：

- 底层数组容量不够时，需要进行 **扩容** ；
- 扩容时， *Python* 额外分配大约 1/81/8 的容量裕量，以控制扩容频率；
- 底层数组空闲位置超过一半时，需要进行 **缩容** 。

动态数组的特性决定了 *list* 对象相关操作性能有好有坏，使用时须特别留意：

- *append* 向尾部追加元素，时间复杂度为 O(1)O(1)，放心使用；
- *insert* 往列表插入元素，最坏时间复杂度是 O(n)O(n)，平均时间复杂度也是 O(n)O(n)，须谨慎使用；
- *pop* 从列表中弹出元素，最好时间复杂度为 O(1)O(1)，平均时间复杂度为 O(n)O(n)，弹出非尾部元素时需谨慎；
- *remove* 从列表中删除元素，时间复杂度为 O(n)O(n)，同样须谨慎使用。