



首先我们知道列表是会自动扩容的，那么什么时候会扩容呢？

之前说过，列表扩容的时候，是在添加元素时发现底层的指针数组已经满了的情况下才会扩容。换句话说，一个列表在添加元素的时候会扩容，那么说明在添加元素之前，其内部的元素个数和容量是相等的。下面就来看看底层是怎么实现的，扩容操作都位于 Objects/listobject.c 中。

```
1 static int
2 list_resize(PyListObject *self, Py_ssize_t newsize)
3 { //参数self就是列表, newsize指的元素在添加之后的ob_size
4   //比如列表的ob_size和容量都是5, append的时候发现容量不够
5   //所以会扩容, 那么这里的newsize就是6
6   //如果是extend添加3个元素, 那么这里的newsize就是8
7   //当然List_resize这个函数不仅可以扩容, 也可以缩容
8   //假设列表原来有1000个元素, 这个时候将列表清空了
9   //那么容量肯定缩小, 不然会浪费内存
10  //如果清空了列表, 那么这里的newsize显然就是0
11
12  //items是一个二级指针, 显然是用来指向指针数组的
13  PyObject **items;
14  //新的容量, 以及对应的内存大小
15  size_t new_allocated, num_allocated_bytes;
16  //获取原来的容量
17  Py_ssize_t allocated = self->allocated;
18
19  //如果newsize达到了容量的一半, 但还没有超过容量
20  //那么意味着newsize、或者新的ob_size和容量是匹配的
21  //所以不会变化
22  if (allocated >= newsize && newsize >= (allocated >> 1)) {
23    assert(self->ob_item != NULL || newsize == 0);
24    //只需要将列表的ob_size设置为newsize即可
25    Py_SIZE(self) = newsize;
26    return 0;
27  }
28
29  //走到这里说明容量和ob_size不匹配了, 所以要进行扩容或者缩容。
30  //因此要申请新的底层数组, 那么长度是多少呢?
31  //这里给出了公式, 一会儿我们可以通过Python进行测试
32  new_allocated = (size_t)newsize + (newsize >> 3) + (newsize < 9 ? 3
33 : 6);
34  //显然容量不可能无限大, 是有范围的
35  //当然这个范围基本上是达不到的
36  if (new_allocated > (size_t)PY_SSIZE_T_MAX / sizeof(PyObject *)) {
37    PyErr_NoMemory();
38    return -1;
39  }
40
41  //如果newsize为0, 那么容量也会变成0
42  //假设将列表全部清空了, 容量就会变成0
43  if (newsize == 0)
44    new_allocated = 0;
45
46  //我们说数组中存放的都是PyObject *, 所以要计算内存
```

```

47 num_allocated_bytes = new_allocated * sizeof(PyObject *);
48 //申请相应大小的内存, 将其指针交给items
49 items = (PyObject **)PyMem_Realloc(self->ob_item, num_allocated_byte
50 s);
51 if (items == NULL) {
52     //如果items是NULL, 代表申请失败
53     PyErr_NoMemory();
54     return -1;
55 }
56 //然后让ob_item = items, 也就是指向新的数组
57 //此时列表就发生了扩容或缩容
58 self->ob_item = items;
59 //将ob_size设置为newsize, 因为它维护列表内部元素的个数
60 Py_SIZE(self) = newsize;
61 //将原来的容量大小设置为新的容量大小
62 self->allocated = new_allocated;
    return 0;
}

```

我们看到还是很简单的，没有什么黑科技。然后是列表扩容的时候，容量和元素个数之间的规律。其实在list_resize函数中是有注释的，其种一行写着：

The growth pattern is: 0, 4, 8, 16, 25, 35, 46, 58, 72, 88, ...

说明我们往一个空列表中不断append元素的时候，容量会按照上面的规律进行变化，我们来试一下。

```

1 lst = []
2 allocated = 0
3 print("此时容量是: 0")
4
5 for item in range(100):
6     lst.append(item) # 添加元素
7
8     # 计算ob_size
9     ob_size = len(lst)
10
11     # 判断ob_size和当前的容量
12     if ob_size > allocated:
13         # lst的大小减去空列表的大小, 再除以8显然就是容量的大小
14         # 因为不管你有没有用, 容量已经分配了
15         allocated = (lst.__sizeof__() - [].__sizeof__()) // 8
16         print(f"列表扩容啦, 新的容量是: {allocated}")
17
18 """
19 此时容量是: 0
20 列表扩容啦, 新的容量是: 4
21 列表扩容啦, 新的容量是: 8
22 列表扩容啦, 新的容量是: 16
23 列表扩容啦, 新的容量是: 25
24 列表扩容啦, 新的容量是: 35
25 列表扩容啦, 新的容量是: 46
26 列表扩容啦, 新的容量是: 58
27 列表扩容啦, 新的容量是: 72
28 列表扩容啦, 新的容量是: 88
29 列表扩容啦, 新的容量是: 106
30 """

```

我们看到和官方给的结果是一样的，显然这是毫无疑问的，我们根据底层的公式也能算出来。

```

1 ob_size = 0
2 allocated = 0
3
4 print(allocated, end=" ")

```

```

5 for item in range(100):
6     ob_size += 1
7     if ob_size > allocated:
8         allocated = ob_size + (ob_size >> 3) + (3 if ob_size < 9 else 6)
9         print(allocated, end=" ")
10 # 0 4 8 16 25 35 46 58 72 88 106

```

这里再提一下，扩容是指解释器在添加元素时发现容量不够的时候才会扩容，如果我们直接通过`lst = []`这种形式创建列表的话，那么其长度和容量是一样的。

```

1 lst = [0] * 1000
2 # 长度和容量一致
3 print(
4     len(lst), (lst.__sizeof__() - [].__sizeof__()) // 8
5 ) # 1000 1000
6
7 # 但此时添加一个元素的话，那么ob_size会变成1001, 大于容量1000
8 # 所以此时列表就要扩容了，执行List_resize
9 # 里面的new_size就是1001, 然后是怎么分配容量来着
10 # new_allocated = (size_t)newsize + (newsize >> 3) + (newsize < 9 ? 3 :
11 6);
12 print(
13     "新容量:", 1001 + (1001 >> 3) + (3 if 1001 < 9 else 6)
14 ) # 新容量: 1132
15
16 # append一个元素, 列表扩容
17 lst.append(123)
18 # 计算容量
19 print((lst.__sizeof__() - [].__sizeof__()) // 8) # 1132

```

结果是一样的，因为底层就是这么实现的，所以结果必须一样。只不过我们通过这种测试的方式证明了这一点，也加深了对列表的认识。

需要注意的是，会影响列表元素个数的操作（append、extend、insert、pop等等），在执行前都会先执行一下list_resize进行容量检测。

如果计算之后的ob_size、也就是newsize和allocated之间的关系是匹配的，即 `allocated//2 <= newsize <= allocated`，那么只需要将ob_size的大小更新为newsize即可。如果不匹配，那么还要进行扩容，此时是一个 $O(n)$ 的操作。

介绍完扩容，再来介绍缩容，因为列表元素个数要是减少到和容量不匹配的话，也要进行缩容。

举个生活中的例子，假设你租了10间屋子用于办公，显然你要付10间屋子的房租，不管你有没有用，一旦租了肯定是要付钱的。同理底层数组也是一样，只要你申请了，不管有没有元素，内存已经占用了。

但有一天你用不到10间屋子了，假设要用8间或者9间，那么会让剩余的屋子闲下来。但由于退租比较麻烦，并且只闲下来一两间屋子，所以干脆就不退了，还是会付10间屋子的钱，这样没准哪天又要用的时候就不用重新租了。

对于列表也是如此，在删除元素(相当于屋子不用了)的时候，如果发现长度还没有低于容量的一半，那么也不会缩容。但反之就要缩容了，比如屋子闲了8间，也就是只需要两间屋子就足够了，那么此时肯定要退租了，闲了8间，可能会退掉6间。

```

1 lst = [0] * 1000
2 print(
3     len(lst), (lst.__sizeof__() - [].__sizeof__()) // 8
4 ) # 1000 1000
5
6 # 删除500个元素，此时长度或者说ob_size就为500
7 lst[500:] = []
8 # 但是ob_size还是达到了容量的一半，所以不会缩容
9 print(

```

```

10     len(lst), (lst.__sizeof__() - [].__sizeof__()) // 8
11 ) # 500 1000
12
13 #如果再删除一个元素的话, 那么不好意思, 显然就要进行扩容了
14 #因为ob_size变成了499, 小于1000 // 2
15 #扩容之后容量怎么算呢? 还是之前那个公式
16 print(499 + (499 >> 3) + (3 if 499 < 9 else 6)) # 567
17
18 #测试一下, 删除一个元素, 看看会不会按照我们期待的规则进行扩容
19 lst.pop()
20 print(
21     len(lst), (lst.__sizeof__() - [].__sizeof__()) // 8
22 ) # 499 567

```

一切都和我们想的是一样的, 另外在代码中我们还看到一个if语句, 就是如果newsize是0, 那么容量也是0, 我们来测试一下。

```

1 lst = [0] * 1000
2 print(
3     len(lst), (lst.__sizeof__() - [].__sizeof__()) // 8
4 ) # 1000 1000
5
6 lst[:] = []
7 print(
8     len(lst), (lst.__sizeof__() - [].__sizeof__()) // 8
9 ) # 0 0
10
11 # 如果按照之前的容量变化公式的话, 会发现结果应该是3
12 # 但是结果是0, 就是因为多了if判断
13 # 如果newsize是0, 就把容量也设置为0
14 print(0 + (0 >> 3) + (3 if 0 < 9 else 6)) # 3

```

为什么要这么做呢? 因为Python认为, 列表长度为0的话, 说明你不想用这个列表了, 所以多余的3个也没有必要申请了。

我们还以租房为栗, 如果你一间屋子都不用了, 说明你可能不用这里的屋子办公了, 因此直接全部退掉。

以上就是列表在改变容量时所采用的策略, 我们从头到尾全部分析了一遍。下一篇来介绍列表支持的操作, 我们会把大部分操作对应的源码都看一遍。

收录于合集 #CPython 97

[< 上一篇](#)

《源码探秘 CPython》27. 列表支持的操作 (上)

[下一篇 >](#)

《源码探秘 CPython》25. 列表是怎么实现的? 解密列表的数据结构

喜欢此内容的人还喜欢

MySQL全面瓦解28: 分库分表
架构与思维



Linux | tcpdump 数据抓包 (二)
小原的笔记



Linux内核基础-进程用户栈与内核栈
技术简说

