《源码探秘 CPython》27. 列表支持的操作（上）

原创 古明地觉 古明地觉的编程教室 2022-02-11 09:00

楔子

分析完列表的扩容之后，我们来看看它支持的操作，显然我们要通过类型对象PyList_Type来查看。

```
PyTypeObject PyList_Type = {
    PyVarObject_HEAD_INIT(&PyType_Type, 0)
    "list",
    sizeof(PyListObject),
    0,
    (destructor)list_dealloc,                   /* tp_dealloc */
    0,                                          /* tp_vectorcall_offset
*/
    0,                                          /* tp_getattr */
    0,                                          /* tp_setattr */
    0,                                          /* tp_as_async */
    (reprfunc)list_repr,                        /* tp_repr */
    0,                                          /* tp_as_number */
    &list_as_sequence,                          /* tp_as_sequence */
    &list_as_mapping,                           /* tp_as_mapping */
    //......
};
```

我们看到，列表支持序列型操作和映射型操作，下面我们就来分析一下这些操作在底层是如何实现的。

不过在介绍之前，先来看几个宏。

```
//根据索引获取元素
#define PyList_GET_ITEM(op, i) (((PyListObject *)(op))->ob_item[i])
//根据索引设置元素
#define PyList_SET_ITEM(op, i, v) (((PyListObject *)(op))->ob_item[i] = (
v))
//获取列表长度
#define PyList_GET_SIZE(op)    (assert(PyList_Check(op)),Py_SIZE(op))
```

其实从名字也可以看出，这些宏的作用是啥，一目了然。

append 追加元素

append方法用于向尾部追加一个元素，我们看看底层实现。

```
static PyObject *
list_append(PyListObject *self, PyObject *object)
{
    //显然调用的app1是核心，它里面实现了添加元素的逻辑
    //Py_RETURN_NONE是一个宏，表示返回Python中的None
    //因为list.append返回的就是None
    if (app1(self, object) == 0)
        Py_RETURN_NONE;
    return NULL;
}
```

```
11
12 static int
13 app1(PyListObject *self, PyObject *v)
14 {
15     //参数self是列表，v是要添加的元素
16     //获取列表的长度
17     Py_ssize_t n = PyList_GET_SIZE(self);
18     assert (v != NULL);
19     //如果长度已经达到了限制，那么无法再添加了
20     //会抛出OverflowError
21     if (n == PY_SSIZE_T_MAX) {
22         PyErr_SetString(PyExc_OverflowError,
23             "cannot add more objects to list");
24         return -1;
25     }
26
27     //还记得这个list_resize吗?
28     //self就是列表， n+1就是newsize、或者说新的ob_size
29     //会自动判断是否要进行扩容， 当然里面还有重要的一步
30     //就是将列表的ob_size设置成newsize、也就是这里的n + 1
31     //因为append之后列表长度大小会变化，而ob_size则要时刻维护这个大小
32     if (list_resize(self, n+1) < 0)
33         return -1;
34
35     //因为v作为了列表的一个元素，所以其指向的对象的引用计数要加1
36     Py_INCREF(v);
37     //设置元素，原来的列表长度为n，最大索引是n - 1
38     //那么追加的话就等于将元素设置在索引为n的地方
39     PyList_SET_ITEM(self, n, v);
40     return 0;
41 }
```

以上就是 append 的逻辑，所谓插入、追加本质上都是通过索引设置元素。

获取元素

我们在使用列表的时候，可以通过**val = lst[1]**这种方式获取元素，那么底层是如何实现的呢?

```
1  static PyObject *
2  list_subscript(PyListObject* self, PyObject* item)
3  {
4      //先看item是不是一个整型
5      //显然这个item除了整型之外，也可以是切片
6      if (PyIndex_Check(item)) {
7          Py_ssize_t i;
8          //这里检测i是否合法，因为Python的整数是没有限制的
9          //但是列表的长度和容量都由一个有具体类型的变量维护
10         //因此个数肯定是有范围的
11         //所以我们输入一个lst[2 ** 100000]显然不行
12         //在Python中会报错IndexError: cannot fit 'int' into an index-sized
13  integer
14         i = PyNumber_AsSsize_t(item, PyExc_IndexError);
15
16         //出现错误，直接return NULL
17         if (i == -1 && PyErr_Occurred())
18             return NULL;
19
20         //如果i小于0， 那么加上列表的长度， 变成正数索引
21         if (i < 0)
22             i += PyList_GET_SIZE(self);
```

```
        //然后调用list_item
        return list_item(self, i);
    }
    else if (PySlice_Check(item)) {
    //......
    }
    else {
        //......
    }
}


static PyObject *
list_item(PyListObject *a, Py_ssize_t i)
{
    //检测索引i的合法性，如果i > 列表的长度
    //那么会报出索引越界的错误。
    if (!valid_index(i, Py_SIZE(a))) {
        //如果索引为负数也会报出索引越界错误
        //因为上面已经对负数索引做了处理了
        //但如果负数索引加上长度之后还是个负数，那么同样报错
        //假设列表长度是5，你的索引是-100
        //加上长度之后是-95，结果还是个负数，所以也会报错
        if (indexerr == NULL) {
            indexerr = PyUnicode_FromString(
                "list index out of range");
            if (indexerr == NULL)
                return NULL;
        }
        PyErr_SetObject(PyExc_IndexError, indexerr);
        return NULL;
    }
    //通过ob_item获取第i个元素
    Py_INCREF(a->ob_item[i]);
    //返回
    return a->ob_item[i];
}
```

而获取元素的时候不光可以通过索引，还可以通过切片的方式。

```
static PyObject *
list_subscript(PyListObject* self, PyObject* item)
{
    if (PyIndex_Check(item)) {
        //.......
    }
    // 如果传入的 item 是切片
    else if (PySlice_Check(item)) {
    /*
    start: 切片的起始位置
    end: 切片的结束位置
    step: 切片的步长
    slicelength: 获取的元素个数，比如[1:5:2]，显然slicelength就是2
    cur: 底层数组中元素的索引
    i: 循环变量
    */
        Py_ssize_t start, stop, step, slicelength, cur, i;
        //返回的结果
        PyObject* result;

        //下面代码中会有所体现
        PyObject* it;
        PyObject **src, **dest;

```

```
        //对切片item进行解包，得到起始位置、结束位置、步长
        if (PySlice_Unpack(item, &start, &stop, &step) < 0) {
            return NULL;
        }
        //计算出slicelength，这里需要结合列表来计算
        //比如我们指定的切片虽然是[1:3:5]
        //但如果列表只有3个元素， 那么slicelength也只能是1
        slicelength = PySlice_AdjustIndices(Py_SIZE(self), &start, &stop,
                                            step);

        //如果slicelength为0， 那么不好意思
        //表示没有元素可以获取， 因此直接返回一个空列表即可
        if (slicelength <= 0) {
            //PyList_New表示创建一个PyListObject
            //里面的参数表示底层数组的长度
            return PyList_New(0);
        }
        //如果步长为1，那么会调用list_slice
        //这个函数内部的逻辑很简单
        //首先接收一个PyListObject *和两个整型(ilow, ihigh)
        //然后在内部会创建一个PyListObject *np， 申请相应的底层数组，设置allocat
ed
        //然后将参数列表中索引为ilow的元素到索引为ihigh的元素依次拷贝到np -> ob_
item里面
        //再设置ob_size并返回
        else if (step == 1) {
            return list_slice(self, start, stop);
        }
        else {
            //走到这里说明步长不为1，那么只能逐个获取指定元素
            //所以这一步负责申请底层数组、设置容量
            //容量就是这里的slicelength， 上面的list_slice中也调用了这一步
            result = list_new_prealloc(slicelength);
            if (!result) return NULL;

            //src是一个二级指针， 也就是self -> ob_item
            src = self->ob_item;
            //同理dest是result -> ob_item
            dest = ((PyListObject *)result)->ob_item;
            //进行循环， cur从start开始遍历， 每次加上step步长
            for (cur = start, i = 0; i < slicelength;
                 cur += (size_t)step, i++) {
                //it就是self -> ob_item中的元素
                it = src[cur];
                //增加指向的对象的引用计数
                Py_INCREF(it);
                //将其设置到dest中
                dest[i] = it;
            }
            //将大小设置为slicelength，说明通过切片创建新列表
            //其长度和容量也是一致的
            Py_SIZE(result) = slicelength;
            //返回结果
            return result;
        }
    }
    else {
        //此时说明item不合法
        PyErr_Format(PyExc_TypeError,
                     "list indices must be integers or slices, not %.200s"
,
                     item->ob_type->tp_name);
        return NULL;
    }
```

```
    }
```

我们发现这个和字符串类似啊，因为通过字符串也支持切片的方式获取。所以随着源码的分析，我们也渐渐明朗这些操作在底层是如何实现的了，真的一点不神秘，实现的逻辑非常简单。

虽然代码量还是有一些的，但是逻辑不难理解，比我们想象中的单纯很多。

设置元素

获取元素知道了，设置元素也不难了。

```
static int
list_ass_subscript(PyListObject* self, PyObject* item, PyObject* value)
{  //在list_subscript的基础上多了一个value参数

    if (PyIndex_Check(item)) {
        //依旧是进行检测i是否合法
        Py_ssize_t i = PyNumber_AsSsize_t(item, PyExc_IndexError);
        if (i == -1 && PyErr_Occurred())
            return -1;
        //索引小于0，则加上列表的长度
        if (i < 0)
            i += PyList_GET_SIZE(self);
        //调用list_ass_item进行设置
        //我们之前见到了list_item，是基于索引获取元素的
        //这里的list_ass_item是基于索引进行元素设置的
        return list_ass_item(self, i, value);
    }
    else if (PySlice_Check(item)) {
      //......
    }
}


static int
list_ass_item(PyListObject *a, Py_ssize_t i, PyObject *v)
{
    //判断索引是否越界
    if (!valid_index(i, Py_SIZE(a))) {
        PyErr_SetString(PyExc_IndexError,
                        "list assignment index out of range");
        return -1;
    }
    //这里的list_ass_slice后面会说
    if (v == NULL)
        return list_ass_slice(a, i, i+1, v);
    //增加v指向对象的引用计数，因为指向它的指针被传到了列表中
    Py_INCREF(v);
    //将第i个元素设置成v
    Py_SETREF(a->ob_item[i], v);
    return 0;
}
```

通过索引设置元素，逻辑很容易。但很明显除了索引，还可以指定切片，而通过切片设置元素就比较复杂了。复杂的原因就在于步长，我们通过Python来演示一下。

```
lst = [1, 2, 3, 4, 5, 6, 7, 8]

#首先通过切片进行设置的话
#右值一定要是一个可迭代对象
```

```python
lst[0: 3] = [11, 22, 33]
# 会将lst[0]设置为11、lst[1]设置为22、lst[2]设置为33
print(lst)  # [11, 22, 33, 4, 5, 6, 7, 8]


#而且它们的长度是可以不相等的
#这里表示将[0: 3]的元素设置为[1, 2], lst[0]设置成1, lst[1]设置成2
#问题来了, lst[2]咋办?
#由于右值中已经没有元素与之匹配了, 那么lst[2]就会被删掉
lst[0: 3] = [1, 2]
print(lst)  # [1, 2, 4, 5, 6, 7, 8]

#所以如果想删除[0: 3]的元素, 那么只需要执行lst[0: 3] = []即可
#因为[]里面没有元素能与之匹配, 所以lst中[0: 3]的位置由于匹配不到
#那么相当于执行了删除操作。当然由于Python的动态特性,
#lst[0: 3] = []、lst[0: 3] = ()、lst[0: 3] = ""等等都是可以的
lst[0: 3] = ""
print(lst)  # [5, 6, 7, 8]
#实际上我们del lst[0]的时候, 就是执行了lst[0: 1] = []


# 当然如果右值元素多的话也是可以的
lst[0: 1] = [1, 2, 3, 4]
print(lst)  # [1, 2, 3, 4, 6, 7, 8]
#lst[0]匹配1很好理解, 但是此时左边已经结束了
#所以剩余的元素会依次插在后面


#然后重点来了, 如果切片有步长的话, 那么两边一定要匹配
#由于此时lst中有8个元素, lst[:: 2]会得到4个元素
#那么右边的可迭代对象的长度也是4
lst[:: 2] = ['a', 'b', 'c', 'd']
print(lst)  # ['a', 2, 'b', 4, 'c', 7, 'd']

# 但是, 如果长度不一致
try:
    lst[:: 2] = ['a', 'b', 'c']
except Exception as e:
    # 显然会报错
    print(e)  # attempt to assign sequence of size 3 to extended slice of size 4
```

总结一下就是：list_subscript用于获取元素，list_ass_subscript用于设置元素。调用这两个函数，我们即可传入索引，也可以传入切片。

- 获取元素时传入的是索引，那么list_subscript内部会调用list_item；传入的是切片，那么会调用list_slice。
- 设置元素时传入的是索引，那么list_ass_subscript内部会调用list_ass_item；传入的是切片，那么会调用list_ass_slice。并且list_ass_slice虽然是设置元素，但删除元素也是调用的它，比如通过 lst[n:n+1]=[] 便可删除索引为n的元素。事实上remove、pop方法都只是计算出待删除元素的索引，真正的删除操作还是通过list_ass_slice来执行的。
- 另外，当传入切片时，只有步长为 1，才会调用list_slice和list_ass_slice；如果步长不为1，那么就采用循环的方式逐个遍历。

设置元素我们只看了基于索引的方式，至于如何通过切片设置元素我们就不看了，主要是考虑的情况比较多，但是核心逻辑并不复杂，有兴趣可以自己去阅读一下。

插入元素

我们调用 insert 可以在任意位置插入元素，那么它的逻辑又是怎么样的呢？

```
int
PyList_Insert(PyObject *op, Py_ssize_t where, PyObject *newitem)
```

```
{
    //类型检查
    if (!PyList_Check(op)) {
        PyErr_BadInternalCall();
        return -1;
    }
    //底层又调用ins1
    return ins1((PyListObject *)op, where, newitem);
}


static int
ins1(PyListObject *self, Py_ssize_t where, PyObject *v)
{
    /*参数self:PyListObject *
    参数where:索引
    参数v:插入的值，这是一个PyObject *指针，因为列表里面存的都是指针
    */

    //i是循环变量，n则是当前列表的元素个数
    Py_ssize_t i, n = Py_SIZE(self);
    //指向指针数组的二级指针
    PyObject **items;
    //如果v是NULL，错误的内部调用
    if (v == NULL) {
        PyErr_BadInternalCall();
        return -1;
    }
    //列表的元素个数不可能无限增大
    //当达到这个PY_SSIZE_T_MAX时，会报出内存溢出错误
    if (n == PY_SSIZE_T_MAX) {
        PyErr_SetString(PyExc_OverflowError,
            "cannot add more objects to list");
        return -1;
    }

    //调整列表容量，既然要inert，那么就势必要多出一个元素
    //这个元素还没有设置进去，但是先把这个坑给留出来
    //当然如果容量够的话，是不会扩容的，只有当容量不够的时候才会扩容
    if (list_resize(self, n+1) < 0)
        return -1;

    //确定插入点
    if (where < 0) {
        //这里可以看到如果where小于0，那么我们就加上n
        //比如有6个元素，我们where=-1，那么会加上6，得到5
        //显然就是insert在索引为5的位置上
        where += n;
        //如果吃撑了，写个-100，加上元素的个数还是小于0
        if (where < 0)
            //那么where=0，就在开头插入
            where = 0;
    }
    //如果where > n，那么就索引为n的位置插入，
    //可元素个数为n，最大索引是n-1啊
    //对，所以此时就相当于append
    if (where > n)
        where = n;
    //走到这，索引就确定完了，然后是设置元素
    //拿到原来的二级指针，指向一个指针数组
    items = self->ob_item;
    //从where开始向后遍历，把索引为i的值赋值给索引为i+1
    //相当于元素后移
```

```
    //既然是在where处插入，那么where之前的就不需要动了
    //所以到where处就停止了
    for (i = n; --i >= where; )
        items[i+1] = items[i];
    //增加v指向的对象的引用计数，因为要被放到列表里
    Py_INCREF(v);
    //将索引为where的值设置成v
    items[where] = v;
    return 0;
}
```

所以可以看到，列表在插入数据的时候是非常灵活的，不管你在什么位置插入，都是合法的。因为它会自己调整位置，在确定位置之后，会将当前位置以及之后的所有元素都向后挪动一个位置，空出来的地方设置为插入的值。

并且这是一个时间复杂度为O(n)的操作，因为插入位置以及后面的所有元素都要向后移动。

pop弹出元素

pop默认是从尾部弹出元素的，因为如果不指定索引的话，默认是-1。当然我们也可以指定索引，弹出指定索引对应的元素。

```
static PyObject *
list_pop_impl(PyListObject *self, Py_ssize_t index)
{
    //弹出的对象的指针
    //因为弹出一个元素实际上是先用某个变量保存起来
    //然后再从列表中删掉
    PyObject *v;

    //下面代码中体现
    int status;

    //如果列表长度为0，显然没有元素可以弹，因此会报错
    if (Py_SIZE(self) == 0) {
        PyErr_SetString(PyExc_IndexError, "pop from empty list");
        return NULL;
    }
    //索引小于0，那么加上列表的长度得到正数索引
    if (index < 0)
        index += Py_SIZE(self);
    //依旧是调用valid_index，判断是否越界。
    //如果索引小于0或者大于等于列长度，抛出 IndexError
    //显然pop没有insert那么智能
    if (!valid_index(index, Py_SIZE(self))) {
        PyErr_SetString(PyExc_IndexError, "pop index out of range");
        return NULL;
    }
    //根据索引获取指定位置的元素
    v = self->ob_item[index];

    //这里同样是一个快分支，如果index是最后一个元素
    //那么直接删除即可，此时不影响其它元素
    if (index == Py_SIZE(self) - 1) {
        //那么直接调用list_resize即可
        //我们说只要涉及元素增删，都要执行list_resize
        //这里会将ob_size减去1
        //至于容量是否变化，就看newsize和allocated之间的关系是否合理
        //如果allocated//2 <= newsize <= allocated，那么容量就不变

```

```
39          status = list_resize(self, Py_SIZE(self) - 1);
40          //list_resize执行成功会返回0
41          if (status >= 0)
42              //直接将对象的指针返回
43              return v;
44          else
45              return NULL;
46      }
47      //否则说明不是最后一个元素
48      //那么该元素被删除之后，它后面所有的元素要向前移动一个位置
49      Py_INCREF(v);
50      //调用list_ass_slice
51      //这一步等价于self[index: index + 1] = []
52      //里面会将元素删掉，并将剩余元素的位置进行调整
53      status = list_ass_slice(self, index, index+1, (PyObject *)NULL);
54      if (status < 0) {
55          //设置失败，减少引用计数
56          Py_DECREF(v);
57          return NULL;
58      }
59      //返回指针
60      return v;
61  }
```

所以pop本质上也是调用了list_ass_slice，都是先计算出索引，然后通过索引来进行操作。

小结

列表用的非常广泛，关于它的操作打算介绍的详细一些，因此会分为**上中下**来用三篇文章进行介绍。

收录于合集 #CPython 97

‹ 上一篇
《源码探秘 CPython》28. 列表支持的操作（中）

下一篇 ›
《源码探秘 CPython》26. 列表是怎么实现扩容的?