# 《源码探秘 CPython》40. 集合是怎么实现的?

原创 古明地觉 古明地觉的编程教室 2022-03-04 08:30






春风微凉，春雨清爽，春色尽荡漾；春笋渐壮，春蕾朝阳，春光满庭芳。

## 集合的数据结构

了解完字典之后，我们再来看看集合，这两者底层都使用了哈希表。而且事实上，集合就类似于没有value的字典。

集合的数据结构定义在setobject.h中，我们来看一下。

```
typedef struct {
    PyObject_HEAD
    Py_ssize_t fill;
    Py_ssize_t used;
    Py_ssize_t mask;
    setentry *table;
    Py_hash_t hash;
    Py_ssize_t finger;
    setentry smalltable[PySet_MINSIZE];
    PyObject *weakreflist;
} PySetObject;

```

- PyObject_HEAD：定长对象的头部信息，但集合显然是一个变长对象。所以和字典一样，肯定有其它字段充当ob_size。
- fill：等于active态的entry数量加上dummy态的entry数量。和字典类似，一个entry就是集合里面的一个元素，类型为setentry。因此在集合里面，一个entry就是一个setentry结构体实例。
- used：等于active态的entry数量，显然这个used充当了ob_size。
- mask：在看字典源码的时候，我们也见到了mask，它用于和哈希值进行按位与、计算索引，并且这个mask等于哈希表的容量减1。为什么呢？假设哈希值等于 v，哈希表容量是 n，那么通过v对n取模即可得到一个位于0到n-1之间的数。但是取模运算的效率不高，而 v&(n-1) 的作用等价于 v%n，并且速度更快，所以mask的值要等于哈希表的容量减1。但是注意，只有在 n 为 2 的幂次方的时候， v&(n-1) 和 v%n 才是完全等价的，所以哈希表的容量要求是 2 的幂次方，就是为了将"取模运算"优化成"按位与运算"。
- table：指向setentry数组的指针，而这个setentry数组可以是下面的smalltable，也可以是单独申请的一块内存。
- hash：集合的哈希值，只适用于frozenset。
- finger：用于pop一个元素，search finger就是我们从包含某个元素的节点开始，找到我们希望的元素。
- smalltable：smalltable是一个setentry类型的数组，集合的元素就存在里面。但是记得我

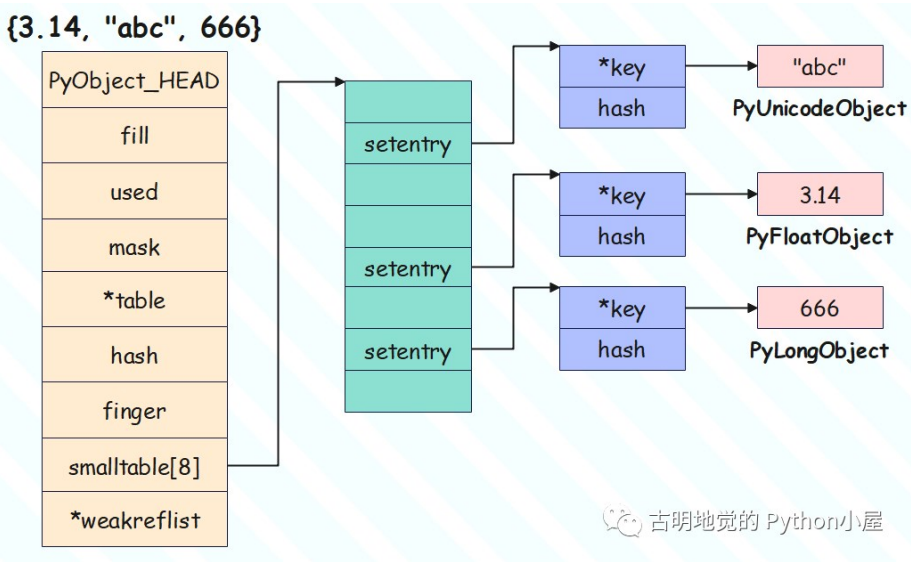
们之前说过，变长对象的内部不会存储具体元素，而是会存储一个指针，该指针指向的内存区域才是用来存储的。这样当扩容的时候，只需要让指针指向新的内存区域即可，从而方便维护。没错，对于集合而言，只有在容量不超过8的时候，元素才会存在里面；而一旦超过了8，那么会使用malloc单独申请内存。PySet_MINSIZE是一个宏，值为8。

- weakreflist：弱引用列表，不做深入讨论。

有了字典的经验，再看集合会简单很多。然后是setentry，用于承载集合内的元素，那么它的结构长什么样呢？相信你能够猜到。

```
typedef struct {
    PyObject *key;
    Py_hash_t hash;
} setentry;
```

相比字典的entry，它少了一个value，这是显而易见的。因此集合的结构很清晰了，假设有一个集合**{3.14,"abc",666}**，那么它的结构如下：




由于集合只有三个元素，所以会存在smalltable数组里面，那么怎么证明这一点呢？我们通过ctypes来测试一下。

```
from ctypes import *

class PyObject(Structure):
    _fields_ = [
        ("ob_refcnt", c_ssize_t),
        ("ob_type", c_void_p),
    ]

class SetEntry(Structure):
    _fields_ = [
        ("key", POINTER(PyObject)),
        ("hash", c_longlong)
    ]

class PySetObject(PyObject):
    _fields_ = [
        ("fill", c_ssize_t),
        ("used", c_ssize_t),
        ("mask", c_ssize_t),
        ("table", POINTER(SetEntry)),
        ("hash", c_long),
        ("finger", c_ssize_t),
        ("smalltable", (SetEntry * 8)),
        ("weakreflist", POINTER(PyObject)),
    ]

```

```
27
28 s = {3.14, "abc", 666}
29 # 先来打印一下哈希值
30 print('hash(3.14) =', hash(3.14))
31 print('hash("abc") =', hash("abc"))
32 print('hash(666) =', hash(666))
33 """
34 hash(3.14) = 322818021289917443
35 hash("abc") = 8036038346376407734
36 hash(666) = 666
37 """
38
39 # 获取PySetObject结构体实例
40 py_set_obj = PySetObject.from_address(id(s))
41 # 遍历smalltable，打印索引、和哈希值
42 for index, entry in enumerate(py_set_obj.smalltable):
43     print(index, entry.hash)
44 """
45 0 0
46 1 0
47 2 666
48 3 322818021289917443
49 4 0
50 5 0
51 6 8036038346376407734
52 7 0
53 """
```

根据输出的哈希值我们可以断定，这三个元素确实存在了smalltable数组里面，并且**666**存在了数组索引为2的位置、**3.14**存在了数组索引为3的位置、**"abc"**存在了数组索引为6的位置。

当然，由于哈希值是随机的，所以每次执行之后打印的结果都会不一样，但是整数除外，它的哈希值就是它本身。既然哈希值不一样，那么每次映射出来的索引也可能不同，但总之，这三个元素是存在smalltable数组里面。

然后我们再考察一下其它的字段：

```
1  s = {3.14, "abc", 666}
2  py_set_obj = PySetObject.from_address(id(s))
3  #集合里面有3个元素，所以fill和used都是3
4  print(py_set_obj.fill)  # 3
5  print(py_set_obj.used)  # 3
6
7  #将集合元素全部删除
8  #这里不能用s.clear()，原因一会儿说
9  for _ in range(len(s)):
10     s.pop()
11
12 # 但我们说哈希表在删除元素的时候是伪删除
13 # 所以fill不变，但是used每次会减1
14 print(py_set_obj.fill)  # 3
15 print(py_set_obj.used)  # 0
```

fill成员维护的是**active态的entry数量加上dummy态的entry数量**，所以删除元素时它的大小是不变的；但used成员的值每次会减1，因为它维护的是**active态的entry的数量**。所以只要不涉及元素的删除，那么这两者的大小是相等的。

然后我们上面说不能用s.clear()，因为该方法表示清空集合，此时会重置为初始状态，然后fill和used都会是0，我们就观察不到想要的现象了。

清空集合之后，我们再往里面添加元素，看看是什么效果：

```
s = {3.14, "abc", 666}
py_set_obj = PySetObject.from_address(id(s))
for _ in range(len(s)):
    s.pop()

# 添加一个元素
s.add(0)
print(py_set_obj.fill)  # 3
print(py_set_obj.used)  # 1
```

多次执行的话，会发现打印的结果可能是3、1，也有可能是4、1。至于原因，有了字典的经验，相信你肯定能猜到。

首先添加元素之后，used肯定为1。至于fill，如果添加的时候，正好撞上了一个dummy态的entry，那么将其替换掉，此时fill不变，仍然是3；如果没有撞上dummy态的entry，而是添加在了新的位置，那么fill就是4。

```
for i in range(1, 10):
    s.add(i)
print(py_set_obj.fill)  # 10
print(py_set_obj.used)  # 10
s.pop()
print(py_set_obj.fill)  # 10
print(py_set_obj.used)  # 9
```

在上面的基础上，继续添加9个元素，然后used变成了10，这很好理解，因为此时集合有10个元素。但fill也是10，这是为什么？很简单，因为哈希表扩容了，扩容时会删除dummy态的entry，所以fill和used是相等的。同理，如果再继续pop，那么fill和used就又变得不相等了。

## 集合的创建

底层提供了PySet_New函数用于创建一个集合，我们来看一下：

```
PyObject *
PySet_New(PyObject *iterable)
{
    //底层调用了make_new_set
    return make_new_set(&PySet_Type, iterable);
}
```

接收一个可迭代对象，真正用来创建的是`make_new_set`。

```
static PyObject *
make_new_set(PyTypeObject *type, PyObject *iterable)
{
    //PySetObject *指针
    PySetObject *so;

    //申请该元素所需要的内存
    so = (PySetObject *)type->tp_alloc(type, 0);
    //申请失败，返回NULL
    if (so == NULL)
        return NULL;

    //初始化都为0
    so->fill = 0;
    so->used = 0;
    //PySet_MINSIZE默认为8
```

```
    //而mask等于哈希表容量减1，所以初始值是7
    so->mask = PySet_MINSIZE - 1;
    //初始化的时候，setentry数组显然是smalltable
    //所以让table指向smalltable数组
    so->table = so->smalltable;
    //初始化hash值为-1
    so->hash = -1;
    //finger为0
    so->finger = 0;
    //弱引用列表为NULL
    so->weakreflist = NULL;
    //以上只是初始化，如果可迭代对象不为NULL
    //那么把元素依次设置到集合中
    if (iterable != NULL) {
    //该过程是通过set_update_internal函数实现的
    //该函数内部会遍历iterable，将迭代出的元素依次添加到集合里面
        if (set_update_internal(so, iterable)) {
            Py_DECREF(so);
            return NULL;
        }
    }
    //返回初始化完成的set
    return (PyObject *)so;
}
```


以上就是集合的底层结构以及创建方式，并且在创建的时候我们居然没有看到缓存池。对的，集合没有自己的缓存池。

下一篇文章，我们来介绍集合的相关操作。

收录于合集 #CPython 97

‹ 上一篇
《源码探秘 CPython》41. 集合支持的操作是怎么实现的?

下一篇 ›
《源码探秘 CPython》39. 字典的缓存池