Blog My Minds

Python源码剖析—Set容器

2016-10-24 | python源码剖析

Python的Set容器

set 与 List 对象相似，均为可变异构容器。但是其实现却和 Dict 类似，均为哈希表。具体的数据结构代码如下。

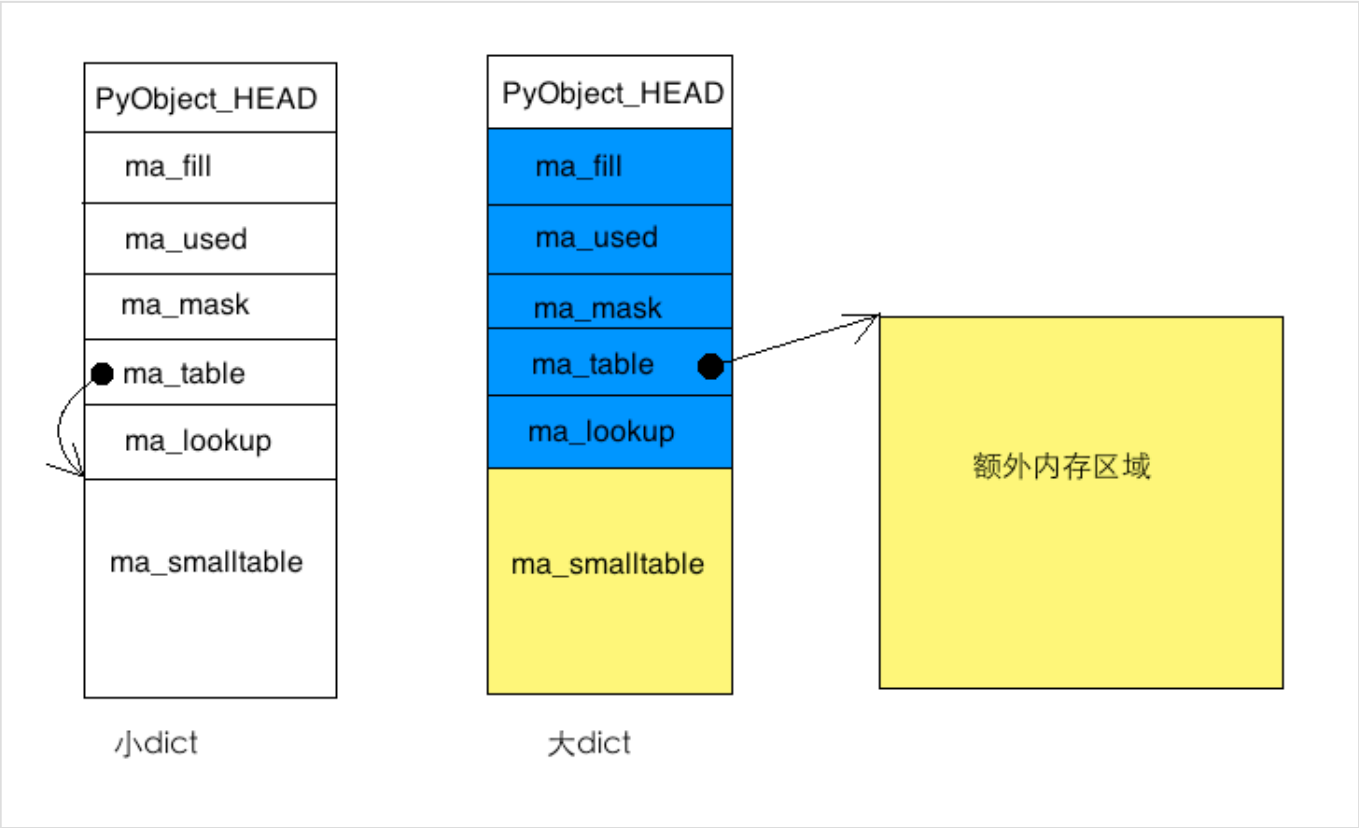
```
typedef struct {
    long hash;      /* cached hash code for the entry key */
    PyObject *key;
} setentry;


/*
This data structure is shared by set and frozenset objects.
*/

typedef struct _setobject PySetObject;
struct _setobject {
    PyObject_HEAD

    Py_ssize_t fill;  /* # Active + # Dummy */
    Py_ssize_t used;  /* # Active */

    /* The table contains mask + 1 slots, and that's a power of 2.
     * We store the mask instead of the size because the mask is more
     * frequently needed.
     */
    Py_ssize_t mask;

    /* table points to smalltable for small tables, else to
     * additional malloc'ed memory.  table is never NULL!  This rule
     * saves repeated runtime null-tests.
     */
    setentry *table;
    setentry *(*lookup)(PySetObject *so, PyObject *key, long hash);
    setentry smalltable[PySet_MINSIZE];
```

```
31
32      long hash;                  /* only used by frozenset objects */
33      PyObject *weakreflist;      /* List of weak references */
34  };
```

`setentry` 是哈希表中的元素，记录插入元素的哈希值以及对应的Python对象。 `PySetObject` 是哈希表的具体结构：

- `fill` 被填充的键的个数，包括Active和dummy，稍后解释具体意思
- `used` 被填充的键中有效的个数，即集合中的元素个数
- `mask` 哈希表的长度的掩码，数值为容量值减一
- `table` 存放元素的数组的指针
- `smalltable` 默认的存放元素的数组

当元素较少时，所有元素只存放在 `smalltable` 数组中，此时 `table` 指向 `smalltable` 。当元素增多，会从新分配内存存放所有的元素，此时 `smalltable` 没有用， `table` 指向新分配的内存。



哈希表中的元素有三种状态：

1. active 元素有效，此时setentry.key != null && != dummy
2. dummy 元素无效key=dummy，此插槽(slot)存放的元素已经被删除
3. NULL 无元素，此插槽从来没有被使用过

dummy是为了表明当前位置存放过元素，需要继续查找。假设a和b元素具有相同的哈希值，所以b只能放在冲撞函数指向的第二个位置。先删除a，再去查找b。如果a被设置为NULL，那么无法确定b是不存在还是应该继续探查第二个位置，所以a只能被设置为dummy。查找b的过程中，第一个位置为dummy所以继续探查，直到找到b；或者直到NULL，证明b确实不存在。

Set中的缓存

set 中会存在缓存系统，缓存数量为80个 _setobject 结构。

```
/* Reuse scheme to save calls to malloc, free, and memset */
#ifndef PySet_MAXFREELIST
#define PySet_MAXFREELIST 80
#endif
static PySetObject *free_list[PySet_MAXFREELIST];
static int numfree = 0;

static void
set_dealloc(PySetObject *so)
{
    register setentry *entry;
    Py_ssize_t fill = so->fill;
    PyObject_GC_UnTrack(so);
    Py_TRASHCAN_SAFE_BEGIN(so)
    if (so->weakreflist != NULL)
        PyObject_ClearWeakRefs((PyObject *) so);
    // 释放每个setentry
    for (entry = so->table; fill > 0; entry++) {
        if (entry->key) {
            --fill;
            Py_DECREF(entry->key);
        }
    }
    // 如果分配了内存存放setentry，则释放掉
    if (so->table != so->smalltable)
        PyMem_DEL(so->table);
    // 缓存_setobject
    if (numfree < PySet_MAXFREELIST && PyAnySet_CheckExact(so))
        free_list[numfree++] = so;
    else
        Py_TYPE(so)->tp_free(so);
    Py_TRASHCAN_SAFE_END(so)
}

}
```

freelist 缓存只会对 _setobject 结构本身起效，会释放掉额外分配的存储键的内存。

Set中查找元素

set 中元素查找有两个函数，在默认情况下的查找函数为 set_lookkey_string 。当发现查找的元素不是 string 类型时，会将对应的 lookup 函数设置为 set_lookkey ，然后调用该函数。

```
1  static setentry *
2  set_lookkey_string(PySetObject *so, PyObject *key, register long hash)
3  {
4      register Py_ssize_t i;
5      register size_t perturb;
6      register setentry *freeslot;
7      register size_t mask = so->mask;
8      setentry *table = so->table;
9      register setentry *entry;
10
11     /* Make sure this function doesn't have to handle non-string keys,
12        including subclasses of str; e.g., one reason to subclass
13        strings is to override __eq__, and for speed we don't cater to
14        that here. */
15
16     /*
17     * 元素不是string，设置lookup = set_lookkey并调用
18     */
19     if (!PyString_CheckExact(key)) {
20         so->lookup = set_lookkey;
21         return set_lookkey(so, key, hash);
22     }
23     // 元素是字符串
24     i = hash & mask;
25     entry = &table[i];
26     // 插槽为空，或者插槽上的key的内存地址与被查找一致
27     if (entry->key == NULL || entry->key == key)
28         return entry;
29     // 第一个插槽为dummy，需要继续调用冲撞函数查找
30     if (entry->key == dummy)
31         freeslot = entry;
32     // 第一个插槽为其他元素，检查是否相等
33     else {
34         if (entry->hash == hash && _PyString_Eq(entry->key, key))
35             return entry;
36         freeslot = NULL;
37     }
38
39     /* In the loop, key == dummy is by far (factor of 100s) the
40        least likely outcome, so test for that last. */
41     /* 第一个插槽为dummy，继续查找 */
```

```
    for (perturb = hash; ; perturb >>= PERTURB_SHIFT) {

        // 冲撞函数
        i = (i << 2) + i + perturb + 1;
        entry = &table[i & mask];
        if (entry->key == NULL)
            return freeslot == NULL ? entry : freeslot;
        if (entry->key == key
            || (entry->hash == hash
            && entry->key != dummy
            && _PyString_Eq(entry->key, key)))
            return entry;
        // 记录第一个为dummy的插槽，当key不存在是返回该插槽
        if (entry->key == dummy && freeslot == NULL)
            freeslot = entry;
    }
    assert(0);          /* NOT REACHED */
    return 0;
}
```

查找函数最后返回的插槽有三种情况：

1. key存在，返回此插槽
2. key不存在，对应的插槽为NULL，返回此插槽
3. key不存在，对应的插槽有dummy，返回第一个dummy的插槽

`set_lookkey` 与此类似，只不过比较元素时需要调用对应的比较函数。

set的重新散列

为了减少哈希冲撞，当哈希表中的元素数量太多时需要扩大桶的长度以减少冲撞。Python中当填充的元素大于总的2/3时开始重新散列，会重新分配一个有效元素个数的两倍或者四倍的新的散列表。

```
static int
set_add_key(register PySetObject *so, PyObject *key)
{
    register long hash;
    register Py_ssize_t n_used;

    if (!PyString_CheckExact(key) ||
        (hash = ((PyStringObject *) key)->ob_shash) == -1) {
        hash = PyObject_Hash(key);
        if (hash == -1)
            return -1;
    }
```

```
13      assert(so->fill <= so->mask);  /* at least one empty slot */
14      n_used = so->used;
15      Py_INCREF(key);
16      if (set_insert_key(so, key, hash) == -1) {
17          Py_DECREF(key);
18          return -1;
19      }
20      // 填充的元素 > 2/3 总数量
21      if (!(so->used > n_used && so->fill*3 >= (so->mask+1)*2))
22          return 0;
23      // 新分配的内存为2倍或者4倍有效元素的个数。
24      // 可以知道一般情况下，有效元素占新分配元素的 1/6
25      // 再占满一半才需要再次分配(2/3 - 1/6 = 1/2)
26      return set_table_resize(so, so->used>50000 ? so->used*2 : so->used*4);
27  }
```

(完)

‹ Python源码剖析—信号处理机制 如何反驳-HowToDisagree ›