《源码探秘 CPython》38. 字典是如何扩容的?

原创 古明地觉 古明地觉的编程教室 2022-03-01 08:30

收录于合集
#CPython
97个 >

微信扫一扫
关注该公众号

楔子

这里先来补充一个之前没有说的点，PyDictObject里面有一个ma_used字段，它维护的是键值对的数量，充当ob_size；而在PyDictKeysObject里面有一个dk_nentries，它维护键值对数组中已使用的entry数量，而这个entry又可以理解为键值对。那么问题来了，这两者有什么区别呢？

如果不涉及元素的删除，那么两者的值会是一样的。而一旦删除某个已存在的key，那么ma_used会**减1**，而dk_nentries则保持不变。

首先ma_used**减1**表示键值对数量相比之前少了一个，这显然符合我们在Python里面使用字典时的表现；但我们知道元素的删除其实是伪删除，会将对应的entry从active态变成dummy态，然而entry的总数量并没有改变。

也就是说，ma_used其实等于active态的entry总数；如果将dk_nentries减去dummy态的entry总数，那么得到的就是ma_used。

所以这就是两者的区别，我们对一个字典使用**len函数**，获取的也是ma_used，而不是dk_nentries。

```
static Py_ssize_t
dict_length(PyDictObject *mp)
{
    return mp->ma_used;
}
```

这算是一个遗漏的点，这里补充一下，然后来看看字典是如何扩容的。

字典的扩容

当已使用entry的数量达到了总容量的2/3时，会发生扩容。但解释器要怎么判断entry数量是否达到了总容量的2/3呢？

我们说Python在早期只有一个键值对数组，这个键值对数组不仅要存储具体的entry，还要完成哈希索引数组的功能。本来这个方式很简单，但是内存浪费严重，于是后面Python官方就将一个数组拆成两个数组来实现。

不是说只能用2/3吗？那我给键值对数组就申请容量的2/3，并且只负责存储键值对。至于索引，则由哈希索引数组来体现。通过将key映射成索引，可以找到哈希索引数组中指定的**槽**，再根据**槽**里面存储的值，可以在键值对数组中找到指定entry。

> 因此减少内存开销的核心就在于，减少键值对数组的浪费。

所以哈希索引数组的长度就可以看成是哈希表的容量，而键值对数组的长度本身就是哈希索引数组的2/3、或者说容量的2/3。那么很明显，当键值对数组满了，就说明当前的哈希表要扩容了。

```
//增长率
```

```
#define GROWTH_RATE(d) ((d)->ma_used*3)
```

并且扩容的时候，新哈希表的容量为**大于等于ma_used*3**的**最小2的幂次方**。假设当前**ma_used*3**等于63，那么扩容之后的容量就是64，也就是2的8次方。

总而言之新哈希表的容量不能小于**ma_used*3**，并且等于**2的幂次方**，基于这两个限制条件，去取最小值。

并且注意是**ma_used*3**，不是dk_nentries。因为dk_nentries还包含了dummy态的entry，但是哈希表在扩容的时候会将其丢弃，只保留active态的entry。所以扩容时，新哈希表的长度取决于ma_used。

然后我们来看看扩容对应的具体逻辑。

```
static int
insertion_resize(PyDictObject *mp)
{
    //本质上调用了dictresize
    //传入PyDictObject * 和增长率
    return dictresize(mp, GROWTH_RATE(mp));
}
```

所以核心藏在`dictresize`函数里面。

```
static int
dictresize(PyDictObject *mp, Py_ssize_t minsize)
{
    //新的哈希表容量，以及当前老哈希表的键值对个数
    Py_ssize_t newsize, numentries;
    //老哈希表的ma_keys
    PyDictKeysObject *oldkeys;
    //老哈希表的ma_values
    PyObject **oldvalues;
    //老哈希表的dk_entries，新哈希表的dk_entries
    PyDictKeyEntry *oldentries, *newentries;

    /* 确定哈希表的大小*/
    //PyDict_MINSIZE等于8，所以哈希表的容量最少是8
    //然后不断左移一位，也就是乘上2
    //因为哈希表的容量必须是2的幂次方
    for (newsize = PyDict_MINSIZE;
    //直到newsize大于等于minsize为止
    //这个minsize就是我们传递的参数，等于ma_used*3
         newsize < minsize && newsize > 0;
         newsize <<= 1)
        ;
    if (newsize <= 0) {
        PyErr_NoMemory();
        return -1;
    }

    //获取老哈希表的ma_keys
    oldkeys = mp->ma_keys;

    /* 创建能够容纳newsize个entry的内存空间 */
    mp->ma_keys = new_keys_object(newsize);
    if (mp->ma_keys == NULL) {
        //把老哈希表的key拷贝过去
        mp->ma_keys = oldkeys;
        return -1;
    }
    assert(mp->ma_keys->dk_usable >= mp->ma_used);
    //如果之前设置了探测函数
```

```
    //那么也作为新哈希表的探测函数
    if (oldkeys->dk_lookup == lookdict)
        mp->ma_keys->dk_lookup = lookdict;

    //获取当前键值对的个数
    numentries = mp->ma_used;
    //获取老哈希表的dk_entries
    oldentries = DK_ENTRIES(oldkeys);
    //获取新哈希表的dk_entries
    newentries = DK_ENTRIES(mp->ma_keys);
    //获取新哈希表的ma_values
    oldvalues = mp->ma_values;
    //如果oldvalues不为NULL，说明是一个split table
    //分离表的特点是key是字符串
    //并且分离表不支持扩容，如果想扩容
    //那么需要把split table转换成combined table
    if (oldvalues != NULL) {
        for (Py_ssize_t i = 0; i < numentries; i++) {
            assert(oldvalues[i] != NULL);
        //获取ma_values数组里面的元素
        //依次设置到PyDictKeyEntry对象里面去
            PyDictKeyEntry *ep = &oldentries[i];
            PyObject *key = ep->me_key;
            Py_INCREF(key);
            newentries[i].me_key = key;
            newentries[i].me_hash = ep->me_hash;
            newentries[i].me_value = oldvalues[i];
        }

        //减少原来对oldkeys的引用计数
        DK_DECREF(oldkeys);
        //将ma_values设置为NULL
        //因为所有的value都存在了PyDictKeyEntry对象的me_value里面
        mp->ma_values = NULL;
        if (oldvalues != empty_values) {
            free_values(oldvalues);
        }
    }
    // 否则的话说明这本身就是一个combined table
    else {
      //numentries等于mp->ma_used，也就是键值对的个数
      //如果等于oldkeys->dk_nentries
      //证明没有dummy态的entry
        if (oldkeys->dk_nentries == numentries) {
        //那么直接将旧的entries拷贝到新的entries里面去
            memcpy(newentries, oldentries, numentries * sizeof(PyDictKe
yEntry));
        }
      //否则说明存在dummy态的entry
        else {
        //active态的entry搬到新table中
        //dummy态的entry则被丢弃
            PyDictKeyEntry *ep = oldentries;
            for (Py_ssize_t i = 0; i < numentries; i++) {
                while (ep->me_value == NULL)
                    ep++;
                newentries[i] = *ep++;
            }
        }

        //字典的缓存池操作，后面介绍
        assert(oldkeys->dk_lookup != lookdict_split);
        assert(oldkeys->dk_refcnt == 1);
        if (oldkeys->dk_size == PyDict_MINSIZE &&
```

```
            numfreekeys < PyDict_MAXFREELIST) {
            DK_DEBUG_DECREF keys_free_list[numfreekeys++] = oldkeys;
        }
        else {
            DK_DEBUG_DECREF PyObject_FREE(oldkeys);
        }
    }

    //建立哈希表索引
    build_indices(mp->ma_keys, newentries, numentries);
    mp->ma_keys->dk_usable -= numentries;
    mp->ma_keys->dk_nentries = numentries;
    return 0;
}
```

代码虽然虽然有点长，但是逻辑很好理解：

- 首先要确定哈希表的大小，很显然这个大小一定要大于minsize。这个minsize我们已经看到了，是通过宏定义的，等于ma_used的3倍；
- 根据新的table，重新申请内存；
- 将原来的处于active态的entry拷贝到新的内存当中，而对于处于dummy态的entry则直接丢弃。可以丢弃的原因我们前面也说过了。因为哈希表扩容会申请的一个新的数组，直接将原来的active态的entry组成一条新的探测链即可，因此也就不需要这些dummy态的entry了。

以上就是哈希表的扩容，或者说字典的扩容，我们就介绍到这儿，下一篇来介绍字典的缓存池，这也是关于字典的最后一篇文章。

收录于合集 #CPython 97

‹ 上一篇
《源码探秘 CPython》39. 字典的缓存池

下一篇 ›
《源码探秘 CPython》37. 字典是怎么创建的，支持的操作又是如何实现的？

喜欢此内容的人还喜欢

一文剖析MySQL主从复制异常错误代码13114
TtrOpsStack

力扣 428. 序列化和反序列化 N 叉树 DFS
钰娘娘知识汇总

MySQL · 参数故事 · timed_mutexes
夜雨成诗