《源码探秘 CPython》36. 哈希表能够直接删除元素吗？

原创 古明地觉 古明地觉的编程教室 2022-02-24 08:30

关于键值对的增删改查等操作，我们会单独介绍，这里先来探讨一下哈希表在删除元素的时候能否直接删除？

通过前面的学习，我们现在已经知道哈希表是先通过哈希函数计算出**键**的哈希值，然后将哈希值传递到探测函数中，再将哈希值映射为一个索引，最终通过索引去访问连续的内存区域。而哈希表这种数据结构，最终目的就是加速键的搜索过程。

并且我们知道，当键值对数量越多，在映射成索引之后就越容易出现冲突。而我们之前说如果冲突了，就改变规则重新映射。事实上，Python也确实是这么做的，这种方法叫做**开放寻址法**。当发生冲突时，在探测函数内部会参考哈希值以及冲突的索引，计算下一个候选位置，如果可用就设置进去。如果不可用，会继续重复此过程，直到找到一个可用的位置。

通过多次探测，可以从一个位置到达多个位置，我们认为这些位置就形成了一个**冲突探测链(探测序列)**。比如当我们插入一个**key="satori"**的键值对时，在**位置a**发现不行，又走**位置b**，发现也被人占了，于是到达**位置c**，而**位置c**没有key，于是就存在了这里。

那么经过以上流程，**a -> b -> c**便形成了一条冲突探测链，同理我们查找的时候也会按照这个顺序进行查找。

显然上面这些东西，现在理解起来已经没什么难度了，但是问题来了。

如果我此时把上面**位置b**的entry给删掉的话，会引发什么后果？首先我们知道，**位置b**上的key和我们指定的**satori**这个key在映射之后的索引是一样的，不然它们也不会映射到同一个槽。

当我们直接获取**d["satori"]**，肯定会先到达**位置a**，发现存在entry、但是key不是字符串**satori**，于是重新映射；然后走到**位置b**，发现还不对，再走到**位置c**，发现key是**satori**，于是就把值取出来了。显然这符合我们的预期，但是，我要说但是了。

如果我们把位置b上的entry删掉呢？那么老规矩，映射成索引，先走到位置a，但是发现坑被占；于是又走到位置b，结果发现居然没有entry，那么直接就报出了一个KeyError。

所以继续寻找的前提是，这个地方要存储了entry，并且存在的**entry -> me_key**和指定的key不相同；但如果没有的话，就说明根本没有这个key，直接KeyError。

然而**satori**这个key确实是存在的，因此这种情况我们称之为**探测链断裂**。本来应该走到位置c的，但是由于位置b没有元素，导致探测函数在位置b就停止了。

因此我们发现，当一个元素只要位于任何一条探测链当中，在删除元素时都不能真正意义上的删除，而是一种**伪删除**操作。

```
//一个键值对就是一个entry
//在底层就是一个 PyDictKeyEntry 实例
typedef struct {
    Py_hash_t me_hash;
    PyObject *me_key;
    PyObject *me_value;
} PyDictKeyEntry;
```

而当一个PyDictObject对象发生变化时，其中的entry会在三种不同的状态之间进行切换：**unused态**、**active态**、**dummy态**。

unused态

当一个entry的me_key和me_value都是NULL的时候，entry处于unused态。unused态表明该entry中并没有存储key、value，并且在此之前也没有存储过它们，每一个entry在初始化的时候都会处于这个状态。

不过me_value的话，即使不是unused态也可能为NULL，更准确的说不管何时它都可能会为NULL，这取决于哈希表到底是结合表、还是分离表。

如果是分离表的话，value是不存在这里的，只有key存在这里，因此me_value永远是NULL。而如果是结合表，那么key和value都存在这里面。所以对于me_key，只可能在unused态的时候才可能为NULL。

active态

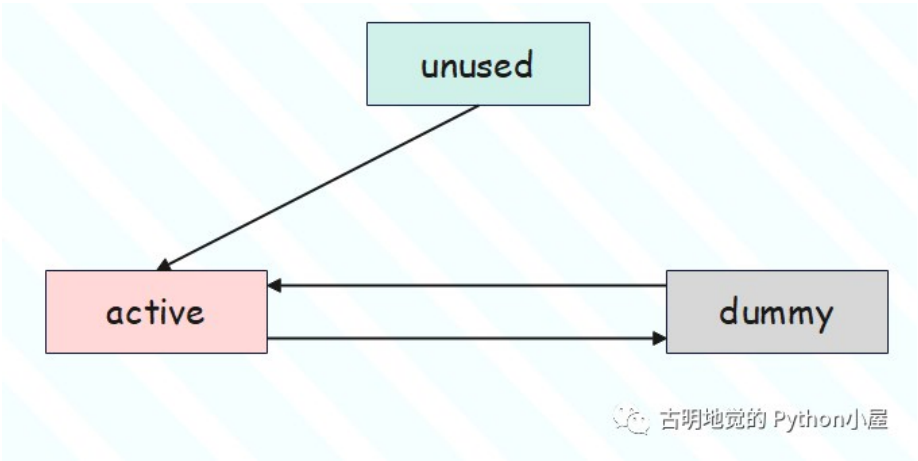
当entry存储了key时，那么此时entry便从unused态变成了active态。

dummy态

当某个key被删除时，它所在的entry便从active态变成dummy态，否则就会发生我们之前说的探测链断裂。至于这个dummy到底是啥，我们后面说。总之entry进入dummy态，就是我们刚才提到的伪删除技术。

当Python沿着某条探测链搜索时，如果发现一个entry处于dummy态，就会明白虽然当前的entry是无效的，但是后面的entry可能是有效的，所以不会报错，而是会继续搜索，这样就保证了探测链的连续性。

至于报错，是在找到了unused态的entry时才会报错，因为这里确实一直都没有存储过key。但索引又是当前这个位置，因此指定的key就真的不存在哈希表中，此时才会报错。



以上是三种状态之间的转换，unused态只能转换为active态；active态只能转换为dummy态；dummy态只能转化为active态。

当entry被使用时，它便由unused态转为active态，此时me_key由**NULL**变成**非NULL**；当删除某个key时，它所在的entry便由active态转为dummy态。

那么问题来了，dummy态转为active态，你能猜到会在什么时候发生吗？

很容易想到，如果新来了一个key，这个key在存储的时候发生冲突，也就是找到的entry存储了别的key，那么会沿着冲突探测链继续查找。在查找的时候要是遇到了处于dummy态的entry，那么该entry就会从dummy态变成active态。

换句话说，对于处于dummy态的entry，Python压根不会主动理会，只是说这个元素被标记为删除了，但是内存还会继续占用。如果新来的key，没有发生冲突，一上来就有位置可以存储，那么不会理会dummy态的entry。

只有当发生冲突的时候，正好撞上了dummy态的entry，才会将dummy态的entry给替换掉。此时entry就变成了active态，然后内部维护的就是新的键值对。

另外当哈希表满了，会申请新的存储单元，然后将所有的active态的entry都搬过去，而处于dummy态的entry则直接丢弃。

之所以可以丢弃，是因为dummy态的entry存在是为了保证探测链不断裂，但是现在所有active态的entry都拷贝到新的内存当中了，它们会形成一条新的探测链，因此也就不需要这些dummy态的entry了。

至于到底是扩容、还是缩容，则取决于当前哈希表的entry个数。但是无论怎么样，当新的哈希表创建之后，便又有新的存储单元可用了。

收录于合集 #CPython 97

‹ 上一篇
《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?

下一篇 ›
《源码探秘 CPython》35. 索引冲突与哈希攻击