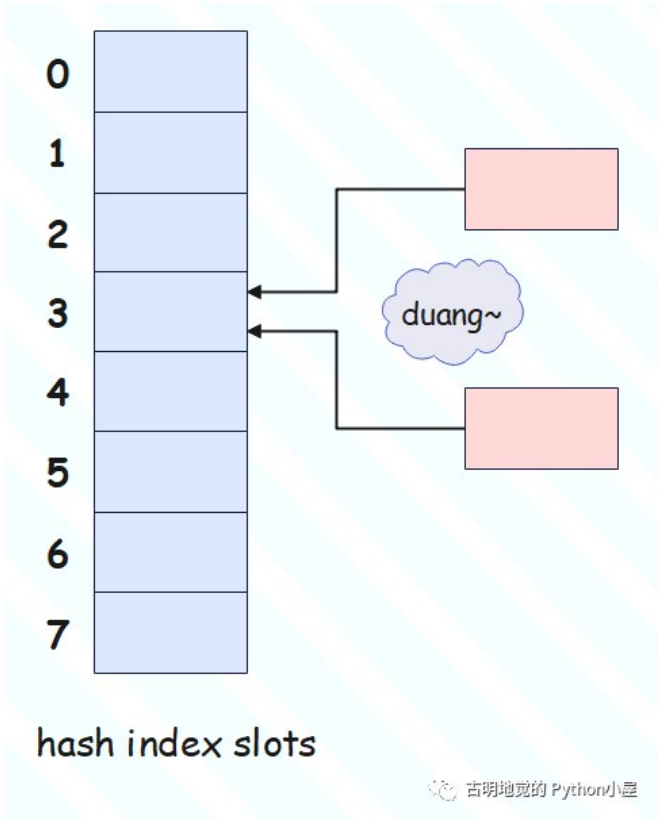
《源码探秘 CPython》35. 索引冲突与哈希攻击

原创 古明地觉 古明地觉的编程教室 2022-02-23 08:30

收录于合集
#CPython
97个 >

索引冲突

不同的对象，计算出的哈希值有可能相同，即使哈希值不同，生成的索引也可能相同。因为与哈希值空间相比，哈希表的槽位是非常有限的。如果不同的对象在经过映射之后，生成的索引相同，或者说它们被映射到了同一个槽，那么便发生了**索引冲突**。



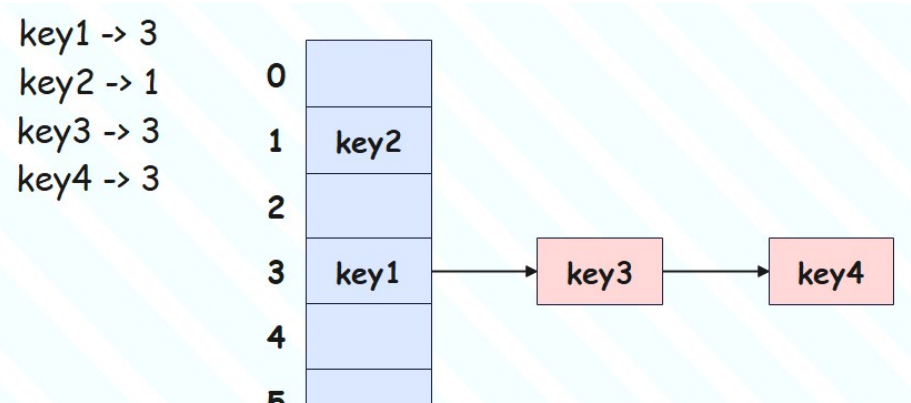
解决索引冲突的常用方法有两种：

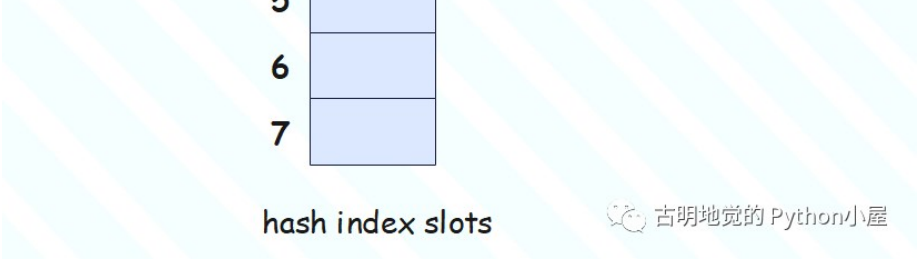
- 分离链接法(separate chaining)
- 开放寻址法(open addressing)

其中Python采用的便是开放寻址法，下面来看看这两种做法之间的区别。

分离链接法

分离链接法为每个哈希槽维护一个链表，所有哈希到同一槽位的键保存到对应的链表中：

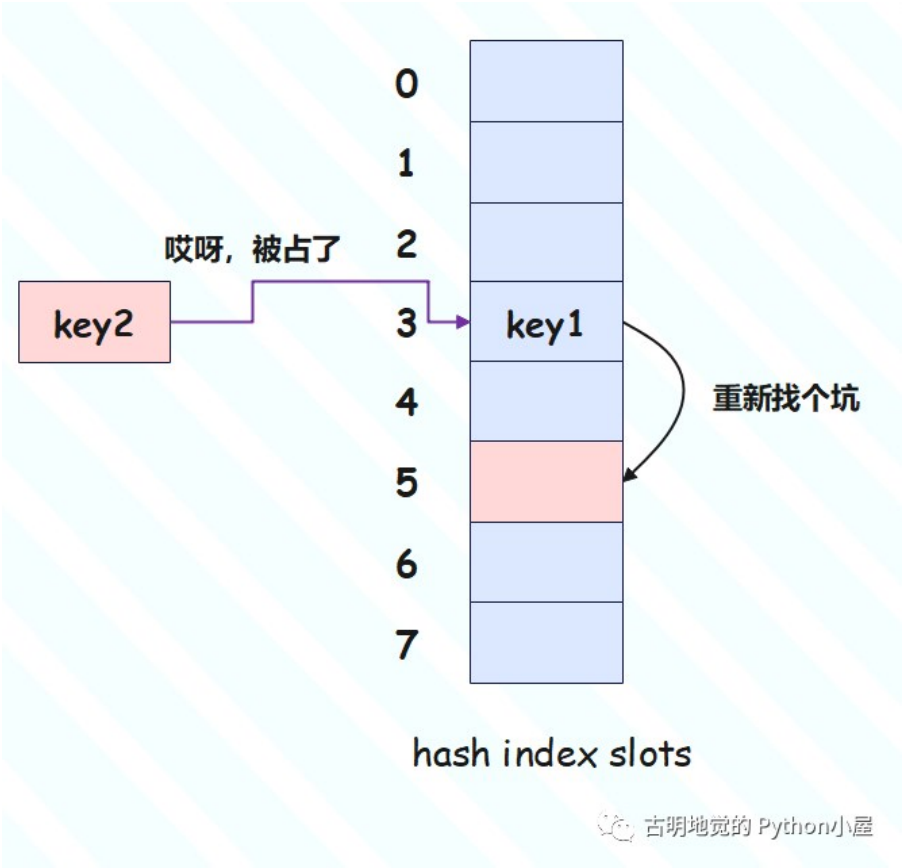




如上图所示，哈希索引数组的每一个槽都连接着一个链表，初始状态为空，映射到哈希表同一个槽的键则保存在对应的链表中。

开放寻址法

Python依旧是将key映射成索引，并存在哈希索引数组的槽中，若发现槽被占了，那么就尝试另一个。

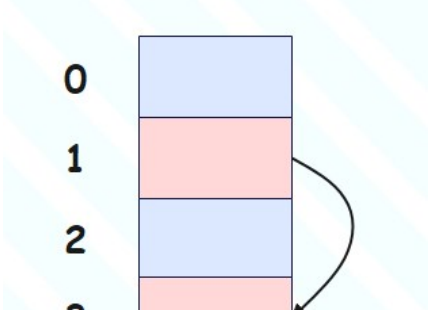


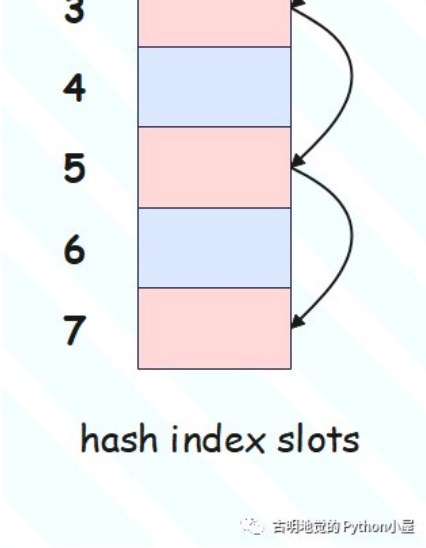
key2被映射到索引为3的槽位时，发现这个坑被key1给占了，所以只能重新找个坑了。但是为什么找到5呢？显然在解决索引冲突的时候是有策略的，一般而言，如果是第**i**次尝试，那么会在首槽的基础上加上一个偏移量**d(i)**。比如映射之后的索引是**n**，那么首槽就是**n**，然而索引为**n**的槽被占了，于是重新映射，而重新映射之后的索引就是**n+d(i)**。

所以可以看出探测方式因函数**d(i)**而异，而常见的探测函数也有两种：

- 线性探测(linear probing)
- 平方探测(quadratic probing)

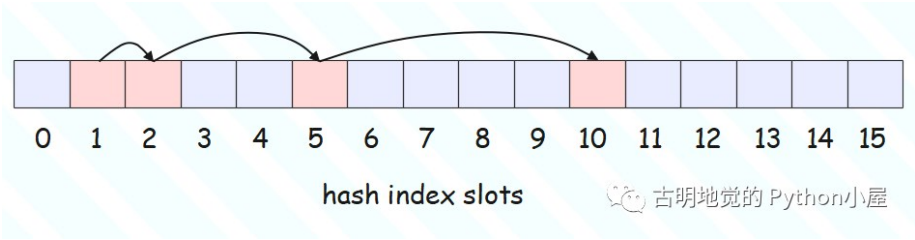
线性探测很好理解，**d(i)**是一个线性函数，例如**d(i)=2*i+1**



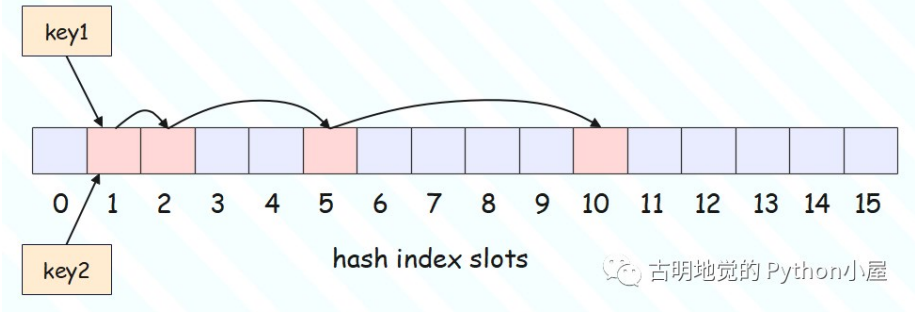


假设哈希之后对应的槽是1，但是被占了，这个时候会在首槽的基础上加一个偏移量**d(i)**。第1次尝试，偏移量是2；第2次尝试，偏移量是4；第3次尝试，偏移量是6。然后再加上首槽的1，所以尝试之后的位置分别是3、5、7。

平方探测也很好理解，**d(i)**是一个平方函数，比如**i**的平方。如果是平方探测，假设首槽还是1，那么冲突之后重试的槽就是1 + 1、1 + 4、1+ 9。

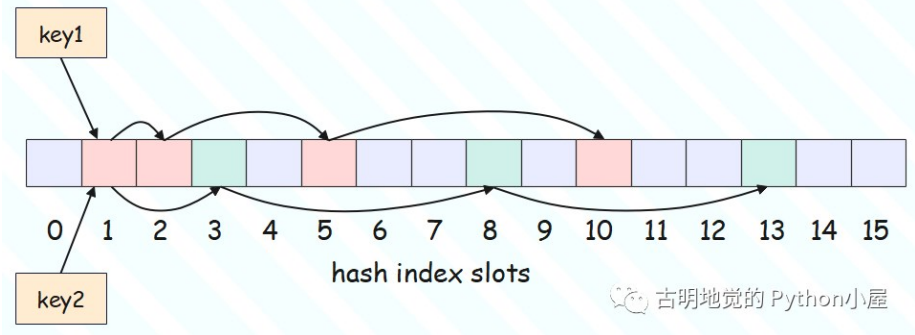


线性探测和平方探测都比较简单，但从效果上来看，平方探测似乎更胜一筹。如果哈希表存在局部热点，线性探测很难快速跳过热点区域，而平方探测则可以解决这一点。但其实这两种方法其实都不够好，因为固定的探测序列加大了冲突的概率。



key1和key2都映射到了索引为**1**的槽，而由于探测序列是相同的，因此后续可能出现多次冲突。

所以Python对此进行了优化，探测函数会参考对象哈希值，生成不同的探测序列，进一步降低索引冲突的可能性：



Python的这种做法被称为**迭代探测**，当然迭代探测也属于开放寻址法的一种。所以当出现索引冲突时，Python并不是简简单单地加上一个偏移量，而是使用专门设计的探测函数进行二次探查，也就是之前说的**改变规则、重新映射**，然后在函数内部会参考对象的哈希值来计算出一个新的索引。

探测函数

Python为哈希表搜索提供了多种探测函数，例如lookdict、lookdict_unicode、lookdict_index，一般通用的是lookdict。

lookdict_unicode是专门针对key为字符串的entry，lookdict_index针对key为整数的entry，可以把lookdict_unicode、lookdict_index看成lookdict的特殊实现，只不过key是整数和字符串的场景非常常见，因此为其单独实现了一个函数。

> 注意：我们对字典无论是设置值还是获取值，都需要进行搜索。要先找到entry在键值对数组中的索引，然后才能进行操作。

我们这里重点看一下**lookdict**的函数实现，它位于**Objects/dictobject.c**源文件内。关键代码如下：

```
static Py_ssize_t _Py_HOT_FUNCTION
lookdict(PyDictObject *mp, PyObject *key,
         Py_hash_t hash, PyObject **value_addr)
{
    size_t i, mask, perturb;
    //ma_keys
    PyDictKeysObject *dk;
    //ma_keys -> dk_entries
    PyDictKeyEntry *ep0;

top:
    dk = mp->ma_keys;
    ep0 = DK_ENTRIES(dk);
    mask = DK_MASK(dk);
    perturb = hash;
    //计算索引，也就是哈希索引数组中的哪一个槽
    //计算方式是hash值和mask按位与
    i = (size_t)hash & mask;

    for (;;) {
    //等价于dk->indecs[i]，也就是索引为i的槽里面存储的值
    //显然这个ix也是索引
    //只不过它表示某个"键值对"在"键值对数组"中的索引
        Py_ssize_t ix = dk_get_index(dk, i);
    //如果ix == DKIX_EMPTY，说明当前的槽是空的
    //该槽没有存储某个"键值对"在"键值对数组"中的索引
    //证明该槽是可用的
        if (ix == DKIX_EMPTY) {
            *value_addr = NULL;
            return ix;
        }
        if (ix >= 0) {
    //如果ix>=0，说明该槽被用了
    //那么根据该槽存储的索引，去键值对数组中查询
    //拿到指定的entry的指针
            PyDictKeyEntry *ep = &ep0[ix];
            assert(ep->me_key != NULL);
    //如果两个key一样，那么直接将值设置为ep->me_value
    //这里先比较地址是否一样，Python的变量在C里面就是一个指针
    //C里面的 == 相当于 Python里面的 is
            if (ep->me_key == key) {
                *value_addr = ep->me_value;
                return ix;
```

```
            }
        //如果不是同一个对象，那么就比较它们的哈希值是否相同
        //比如33和33是一个对象，都是小整数对象池里面的整数
        //但是3333和3333不是一个对象，但是它们的值是一样的
        //因此先判断id是否一致，如果不一致再比较哈希值是否一样
                if (ep->me_hash == hash) {
                    //哈希值一样的话， 那么获取me_key
                    PyObject *startkey = ep->me_key;
                    Py_INCREF(startkey);  //inc ref
                    //比较key是否相等
                    int cmp = PyObject_RichCompareBool(startkey, key, Py_EQ);
                    Py_DECREF(startkey);  //dec ref
                    if (cmp < 0) {
                        *value_addr = NULL;
                        return DKIX_ERROR;
                    }
                    if (dk == mp->ma_keys && ep->me_key == startkey) {
                        if (cmp > 0) {
                            *value_addr = ep->me_value;
                            return ix;
                        }
                    }
                    else {
                        /* The dict was mutated, restart */
                        goto top;
                    }
                }
            }
            //如果条件均不满足，调整姿势，进行下一次探索
            //具体做法是将perturb右移PERTURB_SHIFT个位
            //然后和i进行一系列运算之后，再和mask按位与
            //至于为什么这么做，可以认为是Python总结出的经验
            //这种做法在避免索引冲突时的表现比较好
            //总之会参考对象的哈希值，探测序列因哈希值而异
            perturb >>= PERTURB_SHIFT;
            i = (i*5 + perturb + 1) & mask;
        }
        Py_UNREACHABLE();
    }
```

以上就是lookdict函数的逻辑，但是还没结束，我们看到lookdict函数返回的是**变量ix**，而里面的**变量i**才是我们需要的。**变量i**表示**槽**在哈希索引数组中的索引，**变量ix**表示该**槽**存储的索引。由于我们是要寻找一个可用的槽，那么返回的应该是**槽**的位置、也就是**变量i**才对啊，为啥要返回**变量ix**呢？别急，往下看。

```
#define DKIX_EMPTY (-1)
#define DKIX_DUMMY (-2)  /* Used internally */
#define DKIX_ERROR (-3)
```

如果**ix**等于DKIX_EMPTY，证明当前找到的**槽**是可用的；如果ix大于0，会改变规则重新映射；

因此lookdict这个函数只是告诉我们当前哈希表能否找到一个可用的槽去存储，如果能，那么再由find_empty_slot函数将**槽**的索引返回，这个索引也就是lookdict里面的**变量i**。

```
static Py_ssize_t
find_empty_slot(PyDictKeysObject *keys, Py_hash_t hash)
{
    assert(keys != NULL);
    const size_t mask = DK_MASK(keys);
    size_t i = hash & mask;
    Py_ssize_t ix = dictkeys_get_index(keys, i);
```

```
    for (size_t perturb = hash; ix >= 0;) {
        perturb >>= PERTURB_SHIFT;
        i = (i*5 + perturb + 1) & mask;
        ix = dictkeys_get_index(keys, i);
    }
    return i;
}
```

查找逻辑是一模一样的，因此我们上面说的探测函数，应该是lookdict和find_empty_slot两者的组合，前者判断哈希表是否有**槽**可用，后者告诉我们具体要存储在哪一个**槽**。

说到这可能有人想到了，其实可以把lookdict和find_empty_slot放在一起实现。在lookdict函数里面，当ix==DKIX_EMPTY时，证明该槽可用，那么直接将**变量i**返回就行了。但Python没有这么做，它返回的是**变量ix**，如果后续发现它等于DKIX_EMPTY，那么再采用相同的规则将**变量i**重新算一遍，然后返回。

另外还有一点，我们之前说索引冲突时，会执行探测函数计算新的存储位置。其实不管有没有发生冲突，即使存储键值对的时候哈希表是空的，也要执行探测函数，毕竟探测函数的目的就是基于哈希值映射出一个合适的槽。如果探测函数执行的时候发现索引冲突了，也就是**变量ix>0**，并且key还不相等，那么会改变规则重新映射。

因此在存储某个键值对时，无论索引冲突多少次，探测函数只会执行一次。在探测函数里面，会不断尝试解决冲突，直到映射出一个可用的索引。

哈希攻击

Python在3.3以前，哈希算法只根据对象本身计算哈希值。因此只要解释器版本相同，对象哈希值也肯定相同。

如果一些别有用心的人构造出大量哈希值相同的key，并提交给服务器，会发生什么事情呢？例如，向一台使用Python2编写的web服务**post**一个json数据，数据包含大量哈希值都相同的key。这意味着哈希表将频繁发生索引冲突，性能也会由**O(1)**急剧下降为**O(N)**，这便是哈希攻击。

问题虽然很严重，但是好在应对方法比较简单，直接往对象身上撒把**盐(salt)**即可。具体做法如下：

- Python解释器进程启动后，产生一个随机数作为盐；
- 哈希函数同时参考对象本身以及随机数计算哈希值；

这样一来，攻击者由于无法获取解释器内部的随机数，也就无法构造出哈希值相同的对象了。Python自3.3以后，哈希函数均采用加盐模式，杜绝了哈希攻击的可能性。Python哈希算法在**Python/pyhash.c**源文件中实现，有兴趣可以自己去了解一下，我们这里就不展开了。

我们只需要知道索引冲突时，Python是怎么做的即可，至于如何设计一个哈希函数，以及Python底层使用的哈希函数的具体逻辑，则不在我们的展开范围之内。

小结

key 映射成索引的逻辑很好理解，就是先用哈希函数计算出它的哈希值，然后作为参数传递到探测函数中。在探测函数里面，会将哈希值和mask按位与，得到索引。如果索引冲突了，那么会改变规则，这里的规则如下：

```
//将当前哈希值右移PERTURB_SHIFT个位
perturb >>= PERTURB_SHIFT;
//然后将哈希值加上 i*5 + 1, 这个 i 就是当前冲突的索引
//运算之后的结果再和mask按位与, 得到一个新的 i
//然后判断变量 i 是否可用, 不可用重复当前逻辑
//直到出现一个可用的槽
i = (i*5 + perturb + 1) & mask;
```

所以这就是 Python 解决索引冲突的策略。

而哈希攻击也很简单，就是当解释器的版本相同时，同一个 key 的哈希值也相同，因为早期的Python解释器只根据对象本身计算哈希值。那么如果攻击者伪造大量哈希值相同的key，就会造成哈希表性能的急剧下降。

而Python从3.3开始，会加盐处理，由于攻击者不知道解释器内部的随机数，自然也就无法进行哈希攻击了。

收录于合集 #CPython 97

‹ 上一篇
《源码探秘 CPython》36. 哈希表能够直接删除元素吗?

下一篇 ›
《源码探秘 CPython》34. 对象的哈希值