14 dict 哈希表高级知识精讲-慕课专栏

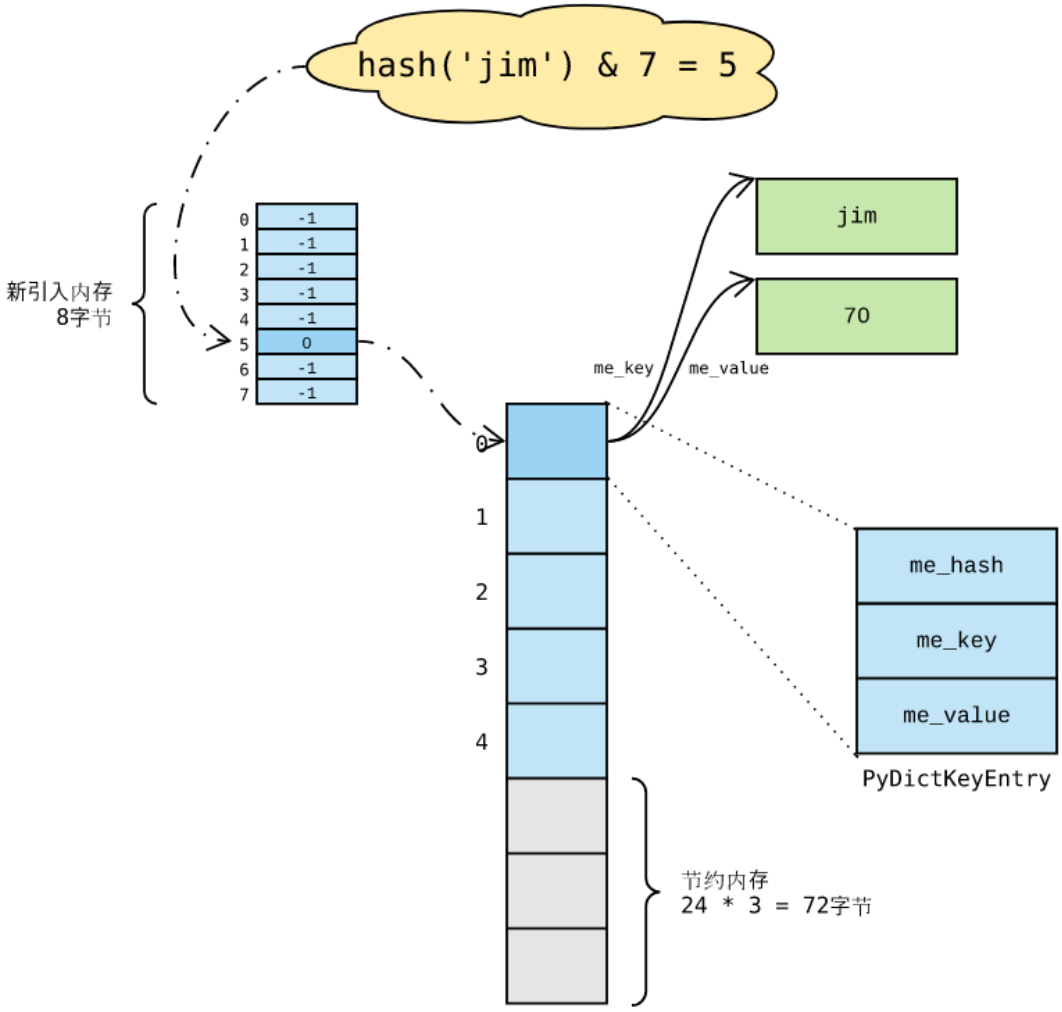
imooc.com/read/76/article/1910

上一小节，我们通过源码学习，研究了 *dict* 对象的内部结构，并找到隐藏其中的秘密—— **哈希表** 。**关联式容器** 一般由 **平衡搜索树** 或 **哈希表** 来实现，*dict* 选用 **哈希表** ，主要考虑 **搜索效率** 。但哈希表 **稀疏** 的特性，意味着巨大的内存开销。为优化内存使用，*Python* 别出心裁地将哈希表分成两部分来实现：**哈希索引** 以及 **键值对存储** 数组。

尽管如此，由于篇幅关系，很多细节我们还没来得及讨论。本节，我们再接再厉，继续研究 **哈希函数** 、**哈希冲突**、**哈希攻击** 以及 **删除操作** 等高级知识点，彻底掌握哈希表设计精髓。

哈希值

Python 内置函数 *hash* 返回对象 **哈希值** ，**哈希表** 依赖 **哈希值** 索引元素：



根据哈希表性质， **键对象** 必须满足以下两个条件，否则哈希表便不能正常工作：

- 哈希值在对象整个生命周期内不能改变；
- 可比较，且比较相等的对象哈希值必须相同；

满足这两个条件的对象便是 **可哈希** (*hashable*)对象，只有可哈希对象才可作为哈希表的键。因此，诸如 dict 、set等底层由哈希表实现的容器对象，其键对象必须是可哈希对象。

Python 内建对象中的 **不可变对象** (*immutable*)都是可哈希对象；而诸如 *list* 、*dict* 等 **可变对象** 则不是：

```
>>> hash([])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

不可哈希对象不能作为 *dict* 对象的键，显然 *list* 、 *dict* 等均不是合法的键对象：

```
>>> {
...     []: 'list is not hashable'
... }
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
TypeError: unhashable type: 'list'
>>>
>>> {
...     {}: 'dict is not hashable either'
... }
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
TypeError: unhashable type: 'dict'
```

而用户自定义的对象默认便是可哈希对象，对象哈希值由对象地址计算而来，且任意两个不同对象均不相等：

```
>>> class A:
...     pass
...
>>>
>>> a = A()
>>> b = A()
>>>
>>> hash(a), hash(b)
(-9223372036573452351, -9223372036573452365)
>>>
>>> a == b
False
```

那么，哈希值如何计算呢？答案是—— **哈希函数** 。在对象模型部分，我们知道对象行为由类型对象决定。 **哈希值** 计算作为对象行为中的一种，秘密也隐藏在类型对象中—— *tp_hash* 函数指针。而内置函数 *hash* 则依赖类型对象中的 *tp_hash* 函数，完成哈希值计算并返回。

以 *str* 对象为例，其哈希函数位于 *Objects/unicodeobject.c* 源文件，*unicode_hash* 是也：

```
PyTypeObject PyUnicode_Type = {
    PyVarObject_HEAD_INIT(&PyType_Type, 0)
    "str",
    sizeof(PyUnicodeObject),


    (hashfunc) unicode_hash,


    unicode_new,
    PyObject_Del,
};
```

对于用户自定义的对象，可以实现 ***hash*** 魔术方法，重写默认哈希值计算方法。举个例子，假设标签类 *Tag* 的实例对象由 *value* 字段唯一标识，便可以根据 value 字段实现 **哈希函数** 以及 **相等性** 判断：

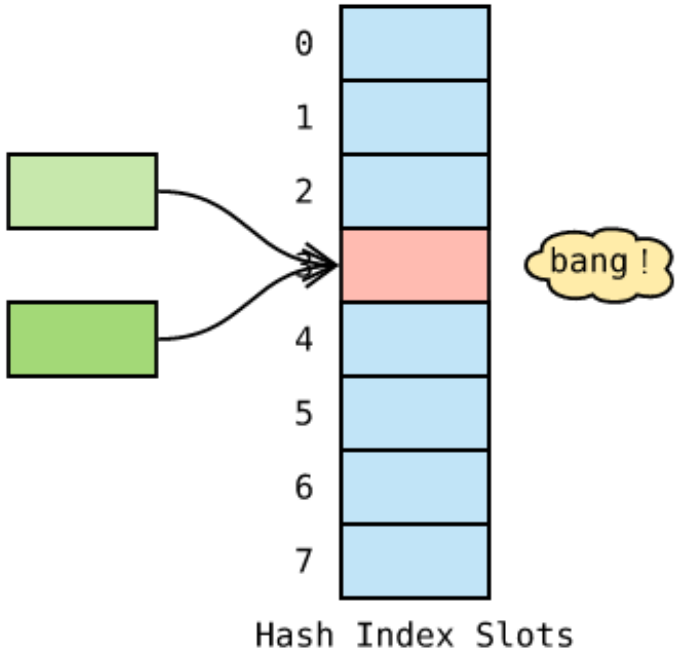
```
class Tag:

    def __init__(self, value, title):
        self.value = value
        self.title = title

    def __hash__(self):
        return hash(self.value)

    def __eq__(self, other):
        return self.value == other.value
```

哈希值 **使用频率** 较高，而且在对象生命周期内均不变。因此，可以在对象内部对哈希值进行缓存，避免重复计算。以 *str* 对象为例，内部结构中的 *hash* 字段便是用于保存哈希值的。

理想的哈希函数必须保证哈希值尽量均匀地分布于整个哈希空间，越是相近的值，其哈希值差别应该越大。

哈希冲突

一方面，不同的对象，哈希值有可能相同，另一方面，与哈希值空间相比，哈希表的槽位是非常有限的。因此，存在多个键被映射到哈希索引的同一槽位的可能性，这便是 **哈希冲突** ！

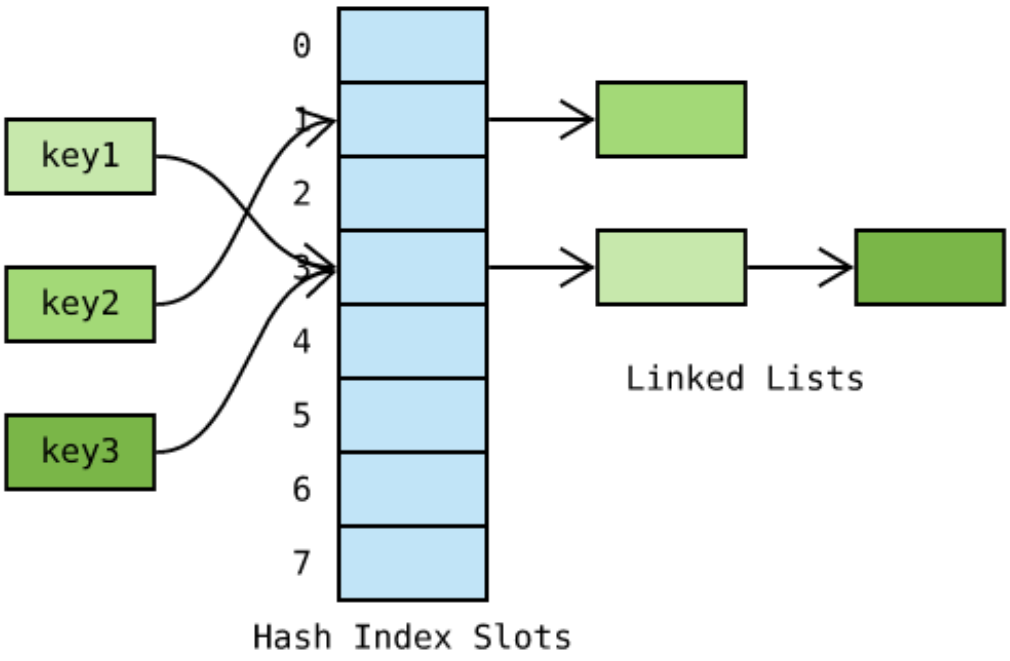


解决哈希冲突的常用方法有两种：

- **分离链接法** (*separate chaining*) ；
- **开放地址法** (*open addressing*)；

分离链接法

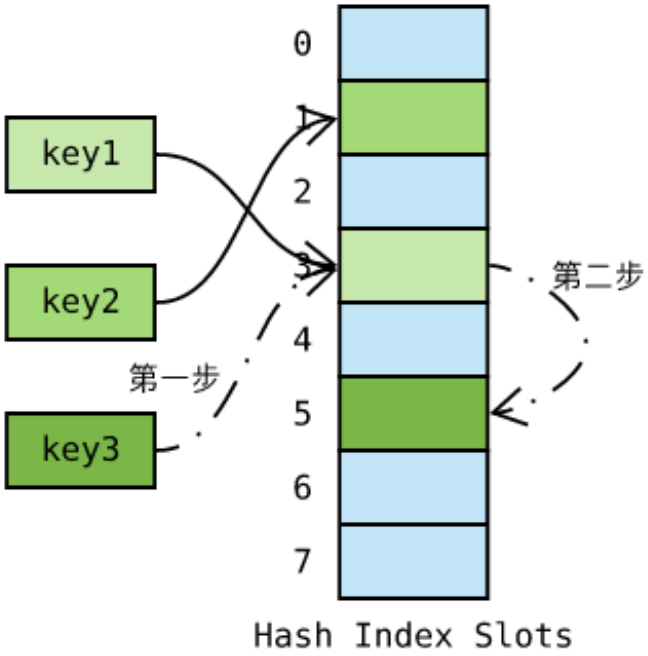
分离链接法 为每个哈希槽维护一个链表，所有哈希到同一槽位的键保存到对应的链表中：



如上图，**哈希索引** 每个槽位都接着一个 **链表** ，初始状态为空；哈希到某个槽位的 **键** 则保存于对应的链表中。例如，*key1* 和 *key3* 都哈希到下标为 3 的槽位，依次保存于槽位对应的链表中。

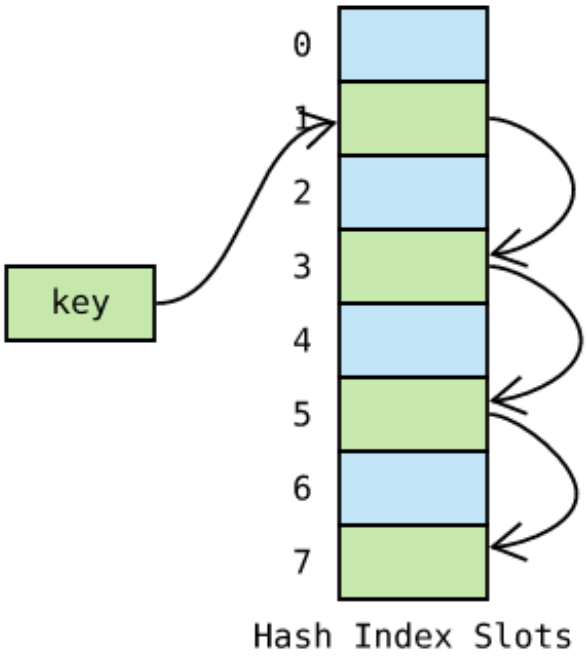
开放地址法

Python 采用 **开放地址法** (*open addressing*)，将数据直接保存于哈希槽位中，如果槽位已被占用，则尝试另一个。

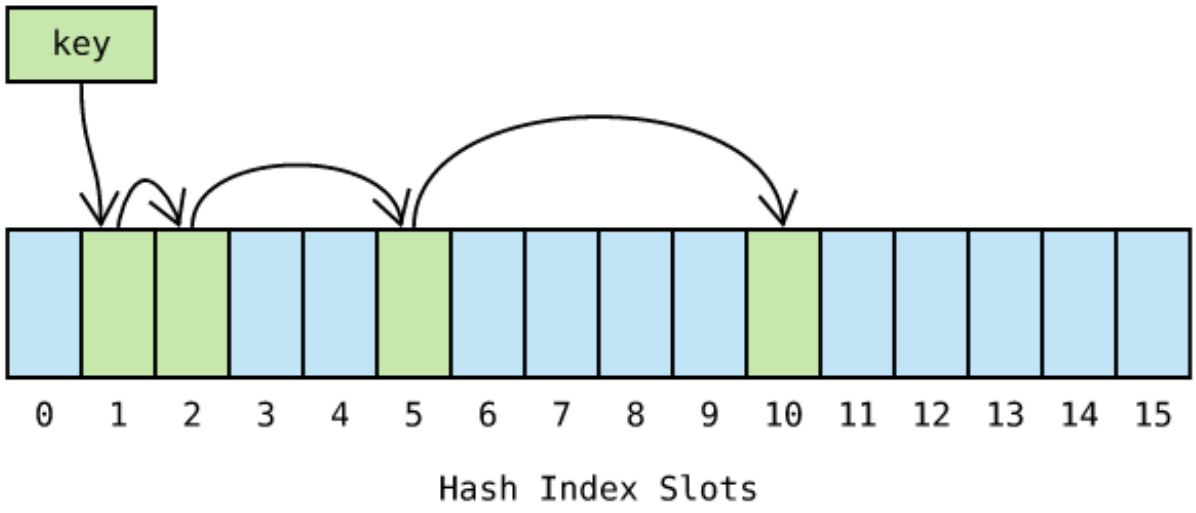


如上图，*key3* 哈希到槽位 *3* ，但已被 *key1* 占用了；接着尝试槽位 *5* 并成功保存。那么，槽位 *5* 是如何决定的呢？一般而言，第 iii 次尝试在首槽位基础上加上一定的偏移量 did_idi。因此，探测方式因函数 did_idi 而异。常见的方法有 **线性探测** (*linear probing*)以及 **平方探测** (*quadratic probing*)。

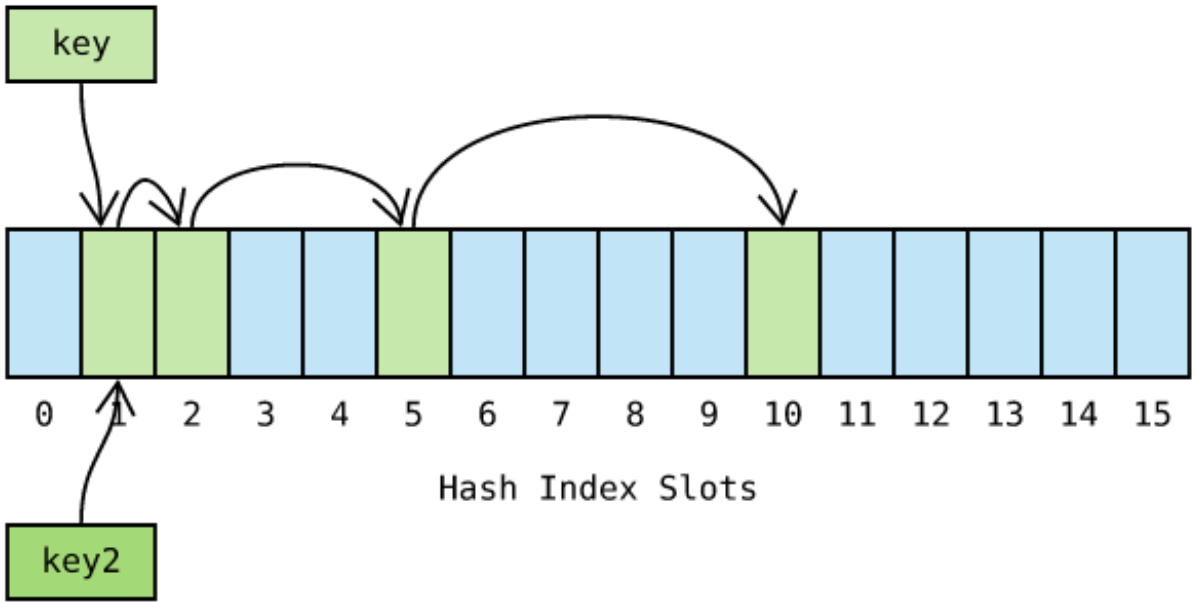
线性探测 ，顾名思义， did_idi 是一个线性函数，例如 di=2∗id_i = 2 * idi=2∗i：



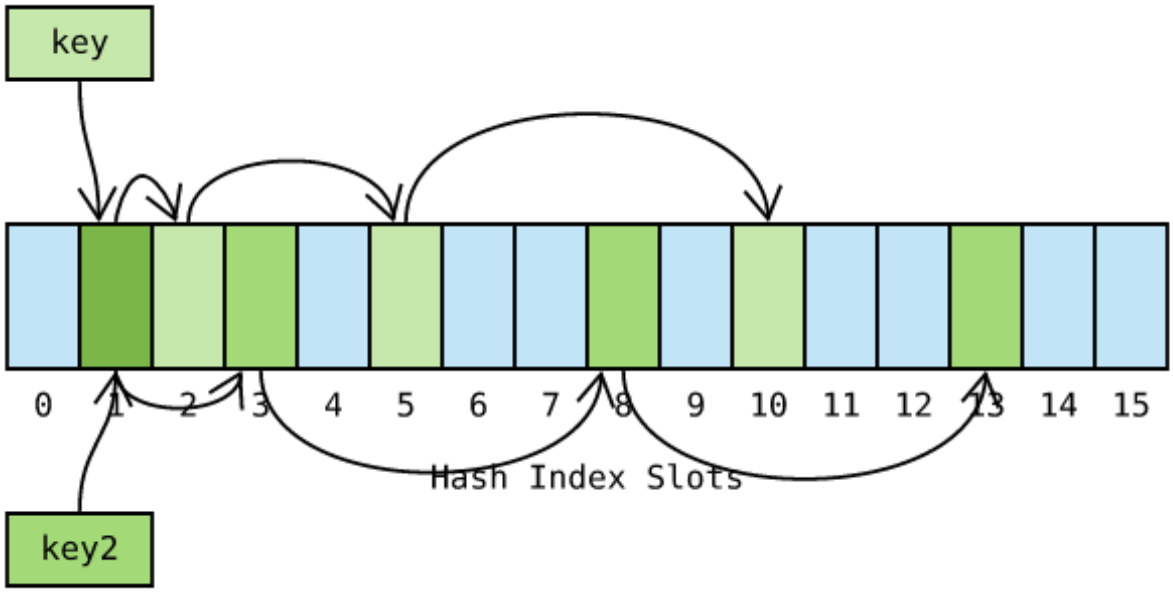
平方探测，顾名思义， did_idi 是一个平方函数，例如 di=i2d_i = i^2di=i2 ：



线性探测 和 **平方探测** 很简单，平方探测似乎更胜一筹。如果哈希表存在局部热点，探测很难快速跳过热点区域，而 **平方探测** 则好很多。然而，这两种方法都不够好——因为固定的探测序列加大了冲突的概率。



如图 *key* 和 *key2* 等都哈希到槽位 *1* ，由于探测序列式相同的，因此冲突概率很高。*Python* 对此进行了优化，探测函数参考对象哈希值，生成不同的探测序列，进一步降低哈希冲突的可能性：



Python 探测方法在 *lookdict* 函数中实现，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;
    PyDictKeysObject *dk;
    PyDictKeyEntry *ep0;

top:
    dk = mp->ma_keys;
    ep0 = DK_ENTRIES(dk);
    mask = DK_MASK(dk);
    perturb = hash;
    i = (size_t)hash & mask;

    for (;;) {
        Py_ssize_t ix = dk_get_index(dk, i);




        perturb >>= PERTURB_SHIFT;
        i = (i*5 + perturb + 1) & mask;
    }
    Py_UNREACHABLE();
}
```

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 *Python* 解释器相同，对象哈希值也肯定相同。我们执行 *Python 2* 解释器启动一个交互式终端，并计算字符串 *fasion* 的哈希值：

```
>>> import os
>>> os.getpid()
2878
>>> hash('fasion')
3629822619130952182
```

我们再次执行 *Python 2* 解释器启动另一个交互式终端，发现字符串 *fasion* 的哈希值保存不变：

```
>>> import os
>>> os.getpid()
2915
>>> hash('fasion')
3629822619130952182
```

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 *Python 2 Web* 服务器 *post* 一个 *json* 数据，数据包含大量的 *key* ，所有 *key* 的哈希值相同。这意味着哈希表将频繁发生哈希冲突，性能由 O(1)O(1)O(1) 急剧下降为 O(N)O(N)O(N)，被活生生打垮！这就是 **哈希攻击** 。

问题很严重，好在应对方法却很简单——为对象加把 **盐** (*salt*)。具体做法如下：

1. *Python* 解释器进程启动后，产生一个随机数作为 **盐** ；
2. 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，攻击者无法获悉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，有兴趣的童鞋可以学习一下，这里就不再展开了。

执行 *Python 3.7* 解释器，启动一个交互式终端，并计算字符串 *fasion* 的哈希值：

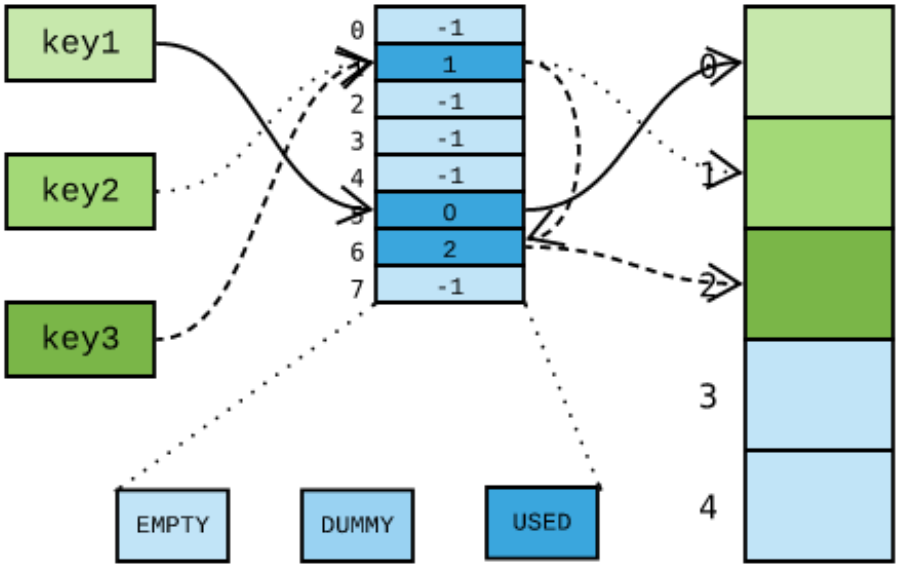
```
>>> hash('fasion')
7411353060704220518
```

再次执行 *Python 3.7* 解释器，启动另一个交互式终端，发现字符串 *fasion* 的哈希值已经变了：

```
>>> hash('fasion')
1784735826115825426
```

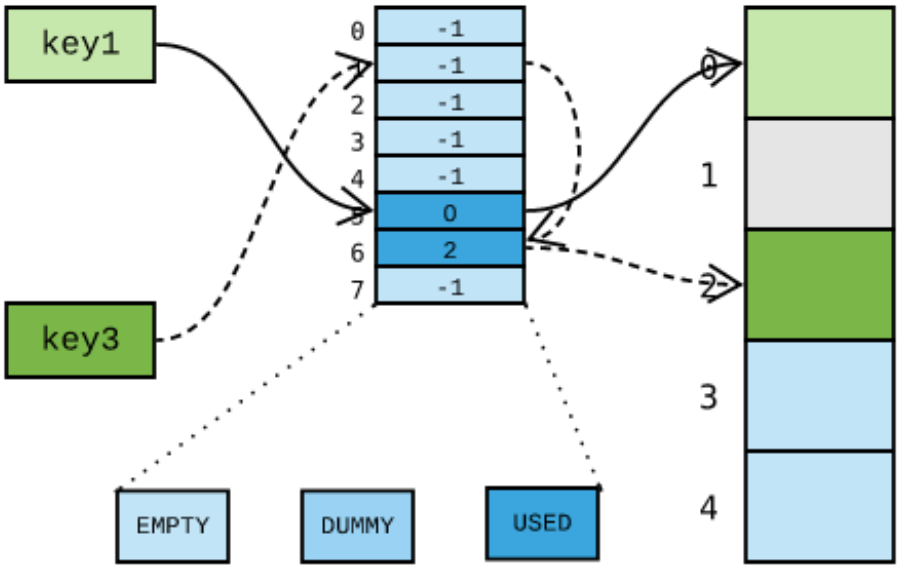
删除操作

现在回过头来讨论 *dict* 哈希表的 **删除** 操作，以下图这个场景为例：

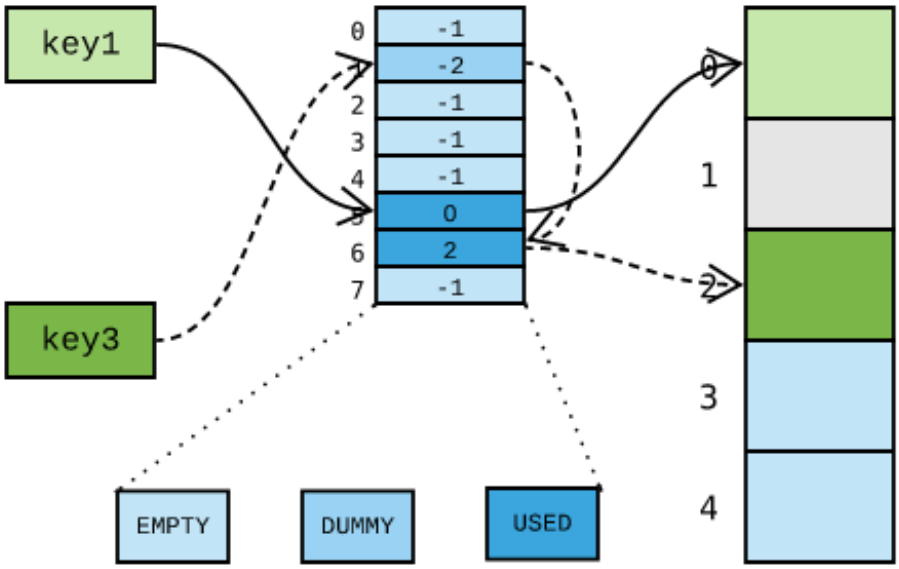


key1 最先插入，使用了哈希槽位 *5* 以及存储单元 *0* ；紧接着插入 *key2* ，使用了哈希槽位 *1* 以及存储单元 *1* ；最后插入 *key3* 时，由于哈希槽位被 *key2* 占用，改用槽位 *6* 。

如果需要删除 *key2* ，该如何操作呢？假设我们在将哈希槽位设置为 *EMPTY* ，并将存储单元标记为删除：



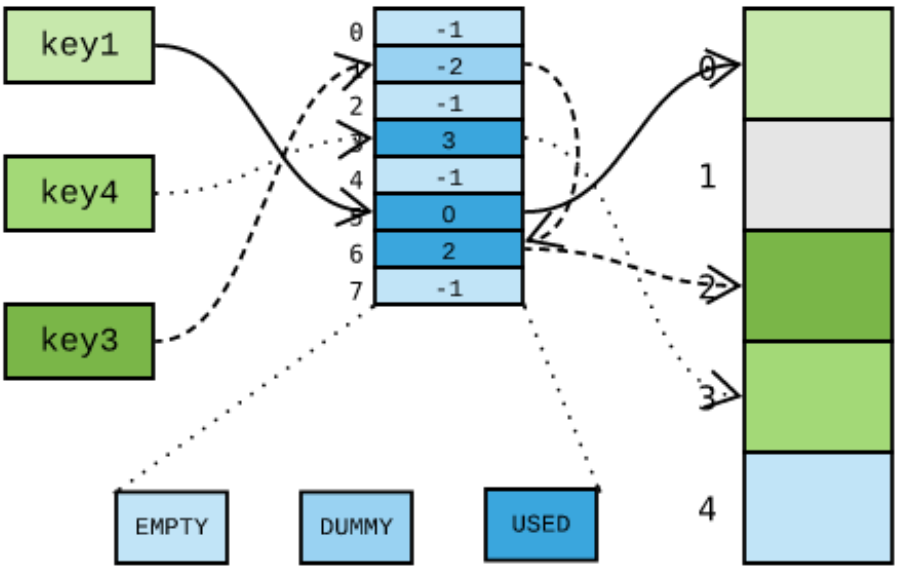
这样一来，由于 *key3* 哈希到的槽位 *1* 是空的，便误以为 *key3* 不存在。换句话讲，*key3* 不翼而飞了！因此，删除元素时，必须将对应的哈希槽设置为一个特殊的标识 *DUMMY* ，避免中断哈希探测链：



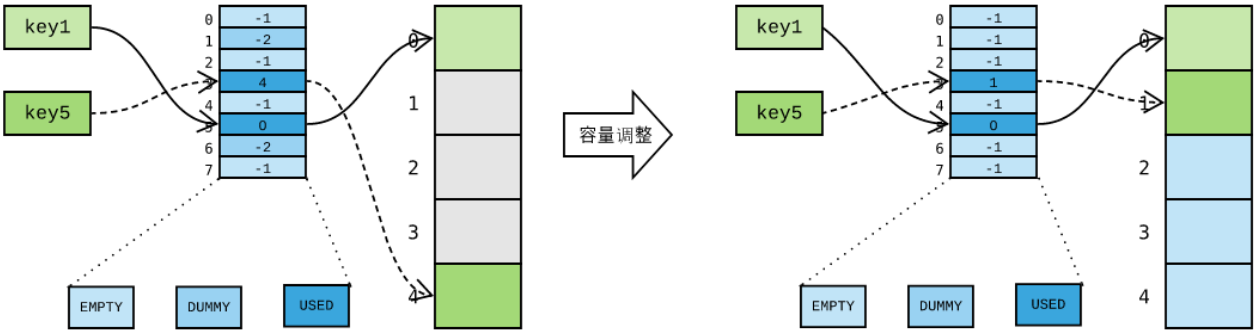
哈希槽位状态常量在 *Objects/dict-common.h* 头文件中定义：

```
#define DKIX_EMPTY (-1)
#define DKIX_DUMMY (-2)
#define DKIX_ERROR (-3)
```

那么，被删除的存储单元如何复用呢？*Python* 压根就没想费这个劲，直接使用新的不就好了吗？假设现在新插入 *key4* ，*Python* 并不理会已删除存储单元 *1* ，直接使用新的存储单元 *3* ：



是的，存储单元中可能有一些是浪费的，但却无伤大雅。如果存储单元已用完，*Python* 则执行一次容量调整操作，重新分配一个哈希表，并将所有元素搬过去，简单粗暴：



新哈希表规模由当前 *dict* 当前元素个数决定，因此容量调整有可能是 **扩容** 、**缩容** 或者 **保持不变** 。无论怎样，新哈希表创建后，便有新存储单元可用了！