《源码探秘 CPython》33. 字典是怎么实现的?

原创 古明地觉 古明地觉的编程教室 2022-02-21 08:30

字典的底层结构

下面我们来看看字典的底层实现，它对应的结构体是**PyDictObject**，位于Include/cpython/dictobject.h中，实现还是有点复杂的。

```
typedef struct {
    PyObject_HEAD
    Py_ssize_t ma_used;
    uint64_t ma_version_tag;
    PyDictKeysObject *ma_keys;
    PyObject **ma_values;
} PyDictObject;
```

解释一下里面的字段的含义：

- PyObject_HEAD：定长对象的头部信息，问题来了，字典明明变长对象啊，它的头部应该是PyObject_VAR_HEAD才对啊。因此肯定有别的成员来维护字典的长度。
- ma_used：字典里面键值对的个数，它充当了ob_size。
- ma_version_tag：字典的版本号，对字典的每一次修改都会导致其改变；除此之外还有一个全局的字典版本计数器pydict_global_version，任何一个字典的修改都会影响它；并且pydict_global_version会和最后操作的字典内部的ma_version_tag保持一致，当然这个成员我们没必要关注，没太大意义。
- ma_keys：从定义上来看它是一个指针，指向了PyDictKeysObject。而Python里面的哈希表分为两种，分别是combined table和split table，即结合表和分离表。如果是结合表，那么键值对全部由ma_keys维护，此时ma_values为NULL。
- ma_values：如果是分离表，那么键由ma_keys维护，值由ma_values维护。而ma_values是一个二级指针，指向PyObject * 类型的指针数组；

这里先解释一下结合表和分离表的由来。结合表的话，键和值就存在一起；分离表的话，键和值就存在不同的地方。那么问题来了，为什么要将哈希表分为两种呢？事实上，早期的哈希表只有结合表这一种，并且现在创建一个字典使用的也是结合表。

```
from ctypes import *

class PyObject(Structure):
    _fields_ = [("ob_refcnt", c_ssize_t),
                ("ob_type", c_void_p)]

class PyDictObject(PyObject):

    _fields_ = [("ma_used", c_ssize_t),
                ("ma_version_tag", c_uint64),
                ("ma_keys", c_void_p),
                ("ma_values", c_void_p)]


d = {"a": 1, "b": 2}
print(
    PyDictObject.from_address(id(d)).ma_values
)  # None
```

我们看到**ma_values**打印的结果是一个None，证明是**结合表**，值不是由**ma_values**维护，而是和键一起都由**ma_keys**负责维护。

而分离表是在PEP-0412中被引入的，主要是为了提高内存使用率，也就是让不同的字典

共享相同的一组key。比如我们自定义类的实例对象，它们默认都有自己的属性字典，如果对某个类多次实例化，那么改成分离表会更有效率。因为它们的属性名称是相同的，完全可以共享同一组key；如果是结合表，那么每个实例的属性字典都要保存相同的key，这显然是一种浪费。

```
from ctypes import *

class PyObject(Structure):
    _fields_ = [("ob_refcnt", c_ssize_t),
                ("ob_type", c_void_p)]

class PyDictObject(PyObject):

    _fields_ = [("ma_used", c_ssize_t),
                ("ma_version_tag", c_uint64),
                ("ma_keys", c_void_p),
                ("ma_values", c_void_p)]

class A:
    pass

a1 = A()
a2 = A()

#因为类型我们指定的是 void *
#所以打印的就是一串地址
#我们看到输出不为None，说明采用的确实是分离表
print(
    PyDictObject.from_address(id(a1.__dict__)).ma_values,
    PyDictObject.from_address(id(a2.__dict__)).ma_values
)  # 2885727436352 2885727434960
#然后再查看ma_keys，既然是共享同一组key
#那么它们的地址应该是一样的
print(
    PyDictObject.from_address(id(a1.__dict__)).ma_keys,
    PyDictObject.from_address(id(a2.__dict__)).ma_keys
)  # 2886174469264 2886174469264

#结果确实是一样的
#不同实例对象的属性字典里面的key是共享的
#因为是同一个类的实例对象，属性字典的key是相同的
#所以没必要将同一组key保存多次
```

以上就是结合表和分离表之间的区别，只需要知道分离表是Python为了提高内存使用率而专门引入的即可。我们平时自己创建的字典，使用的都是结合表，因此我们的重点也将会放在结合表身上。

而结合表的话，键值都由ma_keys维护，它是一个指向PyDictKeysObject的指针，因此玄机就隐藏在这个结构体里面。

```
typedef struct _dictkeysobject PyDictKeysObject;

struct _dictkeysobject {
    Py_ssize_t dk_refcnt;
    Py_ssize_t dk_size;
    dict_lookup_func dk_lookup;
    Py_ssize_t dk_usable;
    Py_ssize_t dk_nentries;
    char dk_indices[];
};
```

字段含义如下：

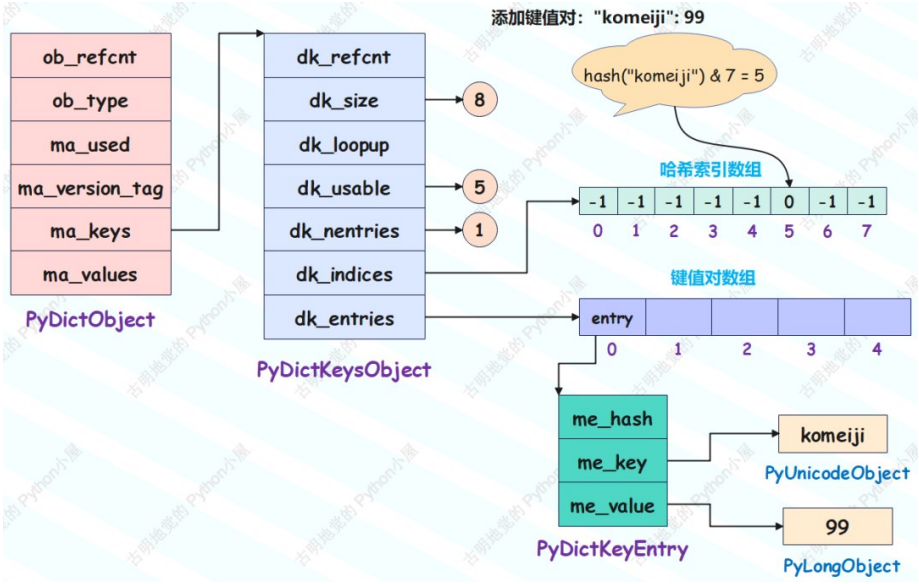
- dk_refcnt：key的引用计数，也就是key被多少个字典所使用。如果是结合表，那么该成员始终是1，因为结合表独占一组key；如果是分离表，那么该成员大于等于1，因为分离表可以共享一组key；
- dk_size：哈希表大小，并且大小是2的n次方，这样可将模运算优化成按位与运算；
- dk_lookup：哈希函数，用于计算key的哈希值，然后映射成索引。一个好的哈希函数应该能尽量少的避免冲突，并且哈希函数对哈希表的性能起着至关重要的作用。所以底层的哈希函数有很多种，会根据对象的种类选择最合适的一个。
- dk_usable：键值对数组的长度，关于什么是键值对数组下面会解释。
- dk_nentries：哈希表中已使用的entry数量，这个entry你可以理解为键值对，一个entry就是一个键值对。
- dk_indices：哈希索引数组，后面会解释。
- dk_entries：键值对数组，虽然结构体里面没有写，但它确实存在。其类型是一个PyDictKeyEntry类型的数组，用于存储键值对、也就是上面说的entry。所以这也说明了，Python的一个键值对，在底层就是一个PyDictKeyEntry结构体实例。

然后再来看看PyDictKeyEntry对象、也就是Python的**键值对**长什么样子。

```
typedef struct {
    Py_hash_t me_hash;
    PyObject *me_key;
    PyObject *me_value;
} PyDictKeyEntry;
```

显然**me_key**和**me_value**指向了键和值，我们之前说Python的变量、以及容器内部的元素都是泛型指针PyObject *，这里也得到了证明。但是我们看到entry除了有键和值之外，还有一个me_hash，它表示键对应的哈希值，这样可以避免重复计算。

至此，字典的整个底层结构就非常清晰了。



字典的真正实现藏在**PyDictKeysObject**中，它的内部包含两个关键数组：一个是哈希索引数组dk_indices，另一个是键值对数组dk_entries。

字典所维护的键值对(entry)会按照先来后到的顺序保存在键值对数组中，而哈希索引数组则保存**键值对**在**键值对数组**中的索引。另外，哈希索引数组中的一个位置我们称之为一个**槽**，比如图中的哈希索引数组便有8个槽，其数量由**dk_size**维护。

比如我们创建一个空字典，注意：虽然字典是空的，但是容量已经有了，然后往里面插入键值对**"komeiji":99**的时候，Python会执行以下步骤：

1. 将键值对保存在dk_entries中，由于初始字典是空的，所以会保存在dk_entries数组中索引为0的位置。
2. 通过哈希函数计算出"komeiji"的哈希值，然后将哈希值映射成索引，假设是5。
3. 将 "键值对" 在 "键值对数组"中的索引0，保存在哈希索引数组中索引为5的槽里面。

然后当我们在查找键**"komeiji"**对应的值的时候，便可瞬间定位。过程如下：

1. 通过哈希函数计算出"komeiji"的哈希值，然后映射成索引。因为在设置的时候索引是5，所以在获取时，映射出来的索引肯定也是5。
2. 找到哈希索引数组中索引为5的槽，得到其保存的0，这里的0对应键值对数组的索引
3. 找到键值对数组中索引为0的位置存储的entry，先比较key、也就是entry->me_key是否一致，不一致则重新映射。如果一致，则取出me_value，然后返回。

由于**哈希值计算**以及**数组索引查找**均是O(1)的时间复杂度，所以字典的查询速度才会这么快。当然我们上面没有涉及到索引冲突，关于索引冲突我们会在后面详细说，但是键值对的存储和获取就是上面那个流程。

当然我们在上一篇文章中，为了避免牵扯太多，所以说的相对简化了。比如：**"xxx": 80**，假设**"xxx"**映射出来的索引是2，那么键值对就直接存在索引为2的地方。这实际上是简化了，因为这相当于把**哈希索引数组**和**键值对数组**合在一块了。而早期的Python，也确实是这么做的。

但是从上面字典的结构图中我们看到，实际上是先将**键值对**按照先来后到的顺序存在一个数组**(键值对数组)**中，然后再将它在键值对数组中的索引存放在另一个数组**(哈希索引数组)**的某个槽里面，因为**"xxx"**映射出来的是2，所以就存在索引为2的槽里面。

而在查找的时候，映射出来的**索引2**其实是哈希索引数组中的索引。然后索引为**2**的槽又存储了一个**索引**，这个**索引**是键值对数组中的索引，会再根据该**索引**从键值对数组里面获取指定的entry。最后比较key是否相同、如果相同则返回指定的value。

所以能看出两者整体思想是基本类似的，理解起来区别不大，甚至第一种方式实现起来还更简单一些。但为什么采用后者这种实现方式，以及这两者之间的区别，我们在后面还会专门分析，之所以采用后者主要是基于内存的考量。

容量策略

根据字典的行为我们断定，字典肯定和列表一样有着**预分配机制**。因为可以扩容，那么为了避免频繁申请内存，所以在扩容的是时候会将容量申请的比键值对个数要多一些。那么字典的容量策略是怎么样的呢？

在Object/dictobject.c源文件中我们可以看到一个宏定义：

```
#define PyDict_MINSIZE 8
```

从这个宏定义中我们可以得知，一个字典的最小容量是8，或者说内部哈希索引数组的长度最小是8。

哈希表越密集，索引冲突则越频繁，性能也就越差。因此，哈希表必须是一种**稀疏**的表结构，越稀疏则性能越好。但由于**内存开销**的制约，哈希表不可能无限地稀疏，所以需要在时间和空间上进行权衡。

而实践经验表明，一个**1/2**到**2/3**满的哈希表，性能较为理想，能以相对合理的**内存**换取相对高效的**执行性能**。

为保证哈希表的稀疏程度，进而控制索引冲突的频率，Python通过宏**USABLE_FRACTION**将哈希表的元素控制在容量的2/3以内。

宏**USABLE_FRACTION**会根据哈希表的长度，计算哈希表可存储的元素个数，也就是**键值对数组**的长度。以长度为**8**的哈希表为例，最多可以保存**5**个键值对，超出则需要扩容。

```
#define USABLE_FRACTION(n) (((n) << 1)/3)
```

哈希表规模一定是2的n次方，也就是说Python采用**翻倍扩容**的策略。例如，长度为8的哈希表扩容后，长度会变为16。另外，这里的哈希表长度和哈希索引数组的长度是等价的。

空字典占用的内存大小

然后我们来考察一下空字典所占用的内存空间，Python为空字典分配了一个长度为8的哈希表，因而也要占用相当多的内存，主要由以下几个部分组成：

- PyDictObject中有6个成员，一个8字节，加起来共48字节；
- PyDictKeysObject中有7个成员，除了两个数组之外，剩余的每个成员也是一个8字节，所以加起来40字节；
- 而剩余的两个数组，一个是char类型的数组dk_indices，里面1个元素占1字节；还有一个PyDictKeyEntry类型的数组dk_entries，里面一个元素占24字节，因为PyDictKeyEntry里面有三个成员，一个8字节。但是注意：字典容量为8，说明哈希索引数组dk_indices长度为8，但是键值对数组dk_entries长度是5，至于原因我们上面分析的很透彻了。因此这两个数组加起来总共是 8 + 24 * 5 = 128字节；

所以一个空字典占用的内存是：**48 + 40 + 128 = 216字节**，我们来测试一下：

```
>>> d = dict()
>>> d.__sizeof__()
216
>>>
```

注意：我们说空字典的容量为8，但前提它不是通过Python/C API创建的。如果是**d = {}**这种方式，那么初始容量就是0，显然此时只有48字节，因为ma_keys此时是NULL。

```
>>> d = {}
>>> d.__sizeof__()
48
>>>
```

另外多提一句，我们在计算内存的时候使用的不是**sys.getsizeof**，而是对象的**__sizeof__**方法，那么这两者有什么区别呢？答案是使用**sys.getsizeof**会比调用对象的**__sizeof__**方法计算出来的内存大小多16个字节。

原因是字典是一个可以发生循环引用的对象，而对于可以发生循环引用的对象，都将参与GC。因此它们除了PyObject_Head之外，还会额外有一个16字节的PyGC_Head。

而使用**sys.getsizeof**计算的大小会将PyGC_Head也算在内，但是对象的**__sizeof__**方法则不会，因此两者差了16字节，这一点要注意。

不过整数、浮点数、字符串等等，它们使用**sys.getsizeof**和**__sizeof__**计算出来的结果是一样的。

```
>>> sys.getsizeof(123), (123).__sizeof__()
(28, 28)
>>>
>>> sys.getsizeof("matsuri"), "matsuri".__sizeof__()
(56, 56)
```

至于为什么一样，想必你已经猜到了，因为整数、字符串这种对象不可能发生循环引用。而Python的GC是专门针对可能发生循环引用的对象的，而不会发生循环引用的对象则不会参与GC，一个引用计数足够了，所以它们使用两种方式计算出的结果是一样的。

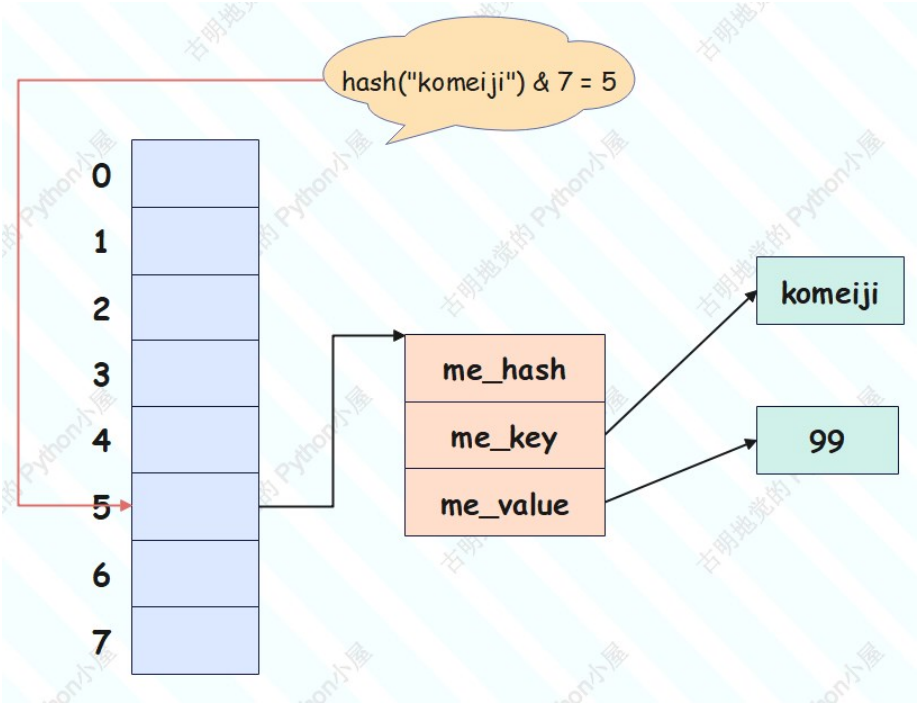
> 关于垃圾回收，是一门很复杂的学问，这里简单提一下。在该系列的后续，我们会详细的探讨Python中的垃圾回收。

另外我们这里计算的是空字典，而包含任意个元素的字典的大小也能计算出来，只要知

道哈希索引数组和键值对数组的长度即可，有兴趣可以自己试一下。

内存优化

在早期，哈希表并没有分成两个数组实现，而是只由一个键值对数组实现，这个数组也承担哈希索引数组的角色：



我们看到这种结构不正是我们在介绍哈希表时说的吗？一个键值对数组既用来存储，又用来充当索引，无需分成两个步骤，而且这种方式也似乎更简单、更直观。没错，Python在早期确实是通过这种方式实现的哈希表，只是这种实现方式有一个弊端，就是太耗费内存了。

因为哈希表必须保持一定程度的稀疏，最多只有2/3满，这意味着至少要浪费1/3的空间。

所以Python为了尽量节省内存，将键值对数组压缩到原来的2/3，只用来存储，而对key进行映射得到的索引由另一个数组**(哈希索引数组)**来体现。假设映射的索引是4，那么就去找哈希索引数组中索引为4的槽，该槽存储的便是键值对在**键值对数组**中的索引。

之所以这么设计，是因为键值对数组里面一个元素要占用24字节，而哈希索引数组在容量不超过255的时候，里面一个元素只占一个字节；容量不超过65535的时候，里面一个元素只占两个字节，其它以此类推。

所以**哈希索引数组**里面的元素大小比**键值对数组**要小很多，将哈希表分成两个数组**(避免键值对数组的浪费)**来实现会更加的节省内存。我们可以举个栗子计算一下，假设有一个容量为65535的哈希表。

如果是通过第一种方式，只用一个数组来存储的话：

```
# 总共需要1572840字节
>>> 65535 * 24
1572840
# 除以3，会浪费524280字节
>>> 65535  * 24 // 3
524280
>>>
```

如果是通过第二种方式，使用两个数组来存储的话：

```
#容量虽然是65535
#但键值对数组是容量的2 / 3
#然后加上哈希索引数组的大小
>>> 65535 * 24 * 2 // 3 + 65535 * 2
1179630
>>>
```

所以一个数组存储比两个数组存储要多用393210字节的内存，因此Python选择使用两个数组来存储。

最后再来提一下字典的顺序问题，Python从3.6开始，字典的遍历是有序的，那么这是怎么实现的呢？

很简单，在存储时，虽然映射之后的索引是随机的，但键值对本身始终是按照先来后到的顺序被添加进键值对数组中。而字典在for循环时，会直接遍历键值对数组，所以遍历的结果是有序的。但即便如此，我们也不应该依赖此特性。

小结

我们通过考察字典的搜索效率，并深入源码研究其内部哈希表的实现，得到以下结论：

- 字典是一种高效的映射型容器，每秒可完成高达 200 多万次的搜索操作
- 字典内部由哈希表实现，哈希表的稀疏特性意味着昂贵的内存开销
- 为优化内存使用，Python通过两个数组来实现哈希表
- 哈希表在 1/2 到 2/3 满时，性能较为理想，较好地平衡了 内存开销 与 搜索效率

以上就是字典的底层实现，但是还没有结束，哈希表的背后还隐藏了很多细节，我们就下一篇文章再聊吧。

收录于合集 #CPython 97

‹ 上一篇
《源码探秘 CPython》34. 对象的哈希值

下一篇 ›
《源码探秘 CPython》32. 初识哈希表