《源码探秘 CPython》39. 字典的缓存池

原创 古明地觉 古明地觉的编程教室 2022-03-03 08:30

世界上哪来那么多一见如故，不过是因为我喜欢你，所以眼里都是你。

楔子

先来简单回顾一下，我们知道字典里面有一个ma_keys和ma_values，其中ma_keys是一个指向PyDictKeysObject的指针，ma_values是一个指向PyObject *数组的二级指针。当哈希表为分离表时，键由ma_keys维护，值由ma_values维护；当哈希表为结合表时，键和值均由ma_keys维护。

那么当我们在销毁一个PyDictObject时，也肯定是要先释放ma_keys和ma_values。

如果是分离表，会将每个value的引用计数减1，然后释放ma_values；再将每个key的引用计数减1，然后释放ma_keys。最后再释放PyDictObject本身。

如果是结合表，由于key、value都在ma_keys中，将每个key、value的引用计数减1之后，只需要再释放ma_keys即可。最后再释放PyDictObject本身。

整个过程还是很清晰的，只不过这里面遗漏了点什么东西，没错，就是缓存池。在介绍浮点数的时候，我们说不同的对象都有自己的缓存池，当然字典也不例外。并且除了PyDictObject之外，PyDictKeysObject也有相应的缓存池，毕竟它负责存储具体的键值对。

那么下面我们就来研究一下这两者的缓存池。

PyDictObject缓存池

字典的缓存池和列表的缓存池高度相似，都是采用数组实现的，并且容量也是80个。

```
#ifndef PyDict_MAXFREELIST
#define PyDict_MAXFREELIST 80
#endif
static PyDictObject *free_list[PyDict_MAXFEELIST];
static int numfree = 0;  //缓存池当前存储的元素个数
```

开始时，这个缓存池什么也没有，直到第一个PyDictObject对象被销毁时，缓存池里面才开始接纳被销毁的PyDictObject对象。

```
static void
dict_dealloc(PyDictObject *mp)
{
    //获取ma_values指针
    PyObject **values = mp->ma_values;
    //获取ma_keys指针
    PyDictKeysObject *keys = mp->ma_keys;
    Py_ssize_t i, n;

    //因为要被销毁，所以让GC不再跟踪
    PyObject_GC_UnTrack(mp);
    //用于延迟释放
    Py_TRASHCAN_SAFE_BEGIN(mp)

    //调整引用计数
    //如果values不为NULL，说明是分离表
    if (values != NULL) {
    //将指向的value、key的引用计数减1
    //然后释放ma_values和ma_keys
        if (values != empty_values) {
            for (i = 0, n = mp->ma_keys->dk_nentries; i < n; i++) {
                Py_XDECREF(values[i]);
            }
            free_values(values);
        }
        DK_DECREF(keys);
    }
    //否则说明是结合表
    else if (keys != NULL) {
    //结合表的话，dk_refcnt一定是1
    //此时只需要释放ma_keys，因为键值对全部由它来维护
    //在DK_DECREF里面，会将每个key、value的引用计数减1
    //然后释放ma_keys
        assert(keys->dk_refcnt == 1);
        DK_DECREF(keys);
    }
    //将被销毁的对象放到缓存池当中
    if (numfree < PyDict_MAXFREELIST && Py_TYPE(mp) == &PyDict_Type)
        free_list[numfree++] = mp;
    else
    //如果缓存池已满，则将释放内存
        Py_TYPE(mp)->tp_free((PyObject *)mp);
    Py_TRASHCAN_SAFE_END(mp)
}
```

同理，当创建字典时，也会优先从缓存池里面获取。

```
static PyObject *
new_dict(PyDictKeysObject *keys, PyObject **values)
{
    //...
    if (numfree) {
        mp = free_list[--numfree];
    }
    //...
}
```

因此在缓存池的实现上，字典和列表有着很高的相似性。不仅都是由数组实现，在销毁的时候也都会放在数组的尾部，创建的时候也会从数组的尾部获取。当然啦，因为这么做符合数组的特性，如果销毁和创建都是在数组的头部操作，那么时间复杂度就从O(1)

变成了O(n)。

我们用Python来测试一下：

```
d1 = {k: 1 for k in "abcdef"}
d2 = {k: 1 for k in "abcdef"}
print("id(d1):", id(d1))
print("id(d2):", id(d2))
# 放到缓存池的尾部
del d1
del d2
# 缓存池:[d1, d2]

# 从缓存池的尾部获取
# 显然id(d3)和上面的id(d2)是相等的
d3 = {k: 1 for k in "abcdefghijk"}
# id(d4)和上面的id(d1)是相等的
d4 = {k: 1 for k in "abcdefghijk"}
print("id(d3):", id(d3))
print("id(d4):", id(d4))
# 输出结果
"""
id(d1): 1363335780736
id(d2): 1363335780800
id(d3): 1363335780800
id(d4): 1363335780736
"""
```

输出结果和我们的预期是相符合的，以上就是PyDictObject的缓存池。

PyDictKeysObject缓存池

PyDictKeysObject也有自己的缓存池，同样基于数组实现，大小是80。

```
//PyDictObject的缓存池叫 free_list
//PyDictKeysObject的缓存池叫 keys_free_list
//两者不要搞混了
static PyDictKeysObject *keys_free_list[PyDict_MAXFREELIST];
static int numfreekeys = 0;  //缓存池当前存储的元素个数
```

我们先来看看它的销毁过程：

```
static void
free_keys_object(PyDictKeysObject *keys)
{
    //将每个entry的me_key、me_value的引用计数减1
    for (i = 0, n = keys->dk_nentries; i < n; i++) {
        Py_XDECREF(entries[i].me_key);
        Py_XDECREF(entries[i].me_value);
    }
#if PyDict_MAXFREELIST > 0
    //将其放在缓存池当中
    //当缓存池未满、并且dk_size为8的时候被缓存
    if (keys->dk_size == PyDict_MINSIZE && numfreekeys < PyDict_MAXFREEL
IST) {
        keys_free_list[numfreekeys++] = keys;
        return;
    }
#endif
    PyObject_FREE(keys);
}
```

销毁的时候，也是放在了缓存池的尾部，那么创建的时候肯定也是先从缓存池的尾部获取。

```c
static PyDictKeysObject *new_keys_object(Py_ssize_t size)
{
    PyDictKeysObject *dk;
    Py_ssize_t es, usable;
    //...
    //创建 ma_keys，如果缓存池有可用对象、并且size等于8，
    //那么会从 keys_free_list 中获取
    if (size == PyDict_MINSIZE && numfreekeys > 0) {
        dk = keys_free_list[--numfreekeys];
    }
    else {
        // 否则malloc重新申请
        dk = PyObject_MALLOC(sizeof(PyDictKeysObject)
                             + es * size
                             + sizeof(PyDictKeyEntry) * usable);
        }
    }
    //...
    return dk;
}
```

所以PyDictKeysObject的缓存池和列表同样是高度相似的，只不过它想要被缓存，还需要满足一个额外的条件，那就是dk_size必须等于8。很明显，这个限制是出于对内存方面的考量。

我们还是来验证一下。

```c
import ctypes


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


class PyDictObject(PyObject):
    _fields_ = [("ma_used", ctypes.c_ssize_t),
                ("ma_version_tag", ctypes.c_uint64),
                ("ma_keys", ctypes.c_void_p),
                ("ma_values", ctypes.c_void_p)]


d1 = {_: 1 for _ in "mnuvwxyz12345"}
print(
    PyDictObject.from_address(id(d1)).ma_keys
)  # 1962690551536
# 键值对个数超过了8，dk_size必然也超过了 8
# 那么当销毁d1的时候，d1.ma_keys不会被缓存
# 而是会直接释放掉
del d1

d2 = {_: 1 for _ in "a"}
print(
    PyDictObject.from_address(id(d2)).ma_keys
)  # 1962387670624

# d2 的 dk_size 显然等于 8
# 因此它的 ma_keys 是会被缓存的
del d2

```

```
d3 = {_: 1 for _ in "abcdefg"}
print(
    PyDictObject.from_address(id(d3)).ma_keys
)  # 1962699215808
# 尽管 d2 的 ma_keys 被缓存起来了
# 但是 d3 的 dk_size 大于 8
# 因此它不会从缓存池中获取，而是重新创建


# d4 的 dk_size 等于 8
# 因此它会获取 d2 被销毁的 ma_keys
d4 = {_: 1 for _ in "abc"}
print(
    PyDictObject.from_address(id(d4)).ma_keys
)  # 1962387670624
```

所以从打印的结果来看，由于**d4.ma_keys**和**d2.ma_keys**是相同的，因此证实了我们的结论。不像列表和字典，它们是只要被销毁，就会放到缓存池里面，因为它们没有存储具体的数据，大小是固定的。

但是PyDictKeysObject不同，它存储了entry，每个entry占24字节。如果内部的entry非常多，那么缓存起来会有额外的内存开销。因此Python的策略是，只有在dk_size等于8的时候，才会缓存。当然这三者在缓存池的实现上，是基本一致的。

小结

到此，字典相关的内容我们就全部介绍完了。总的来说，Python的字典是一个被高度优化的数据结构，因为解释器在运行的时候也重度依赖字典，这就决定了它的效率会非常高。

当然，我们没有涉及字典的全部内容，比如字典有很多方法，比如keys、values、items方法等等，我们并没有说。这些有兴趣的话，可以对着源码看一遍，不是很难。

总之我们平时，也可以尽量多使用字典。

收录于合集 #CPython 97

‹ 上一篇
《源码探秘 CPython》40. 集合是怎么实现的?

下一篇 ›
《源码探秘 CPython》38. 字典是如何扩容的?