《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?

原创 古明地觉 古明地觉的编程教室 2022-02-28 08:30

收录于合集
#CPython
97个 >

PyDictObject 的创建

解释器内部会通过PyDict_New来创建一个新的dict对象。

```
PyObject *
PyDict_New(void)
{
    //new_keys_object表示创建PyDictKeysObject*对象
    //里面传一个数值，表示哈希表的容量
    //#define PyDict_MINSIZE 8，从宏定义我们能看出来为8
    PyDictKeysObject *keys = new_keys_object(PyDict_MINSIZE);
    if (keys == NULL)
        return NULL;
    //这一步则是根据PyDictKeysObject *创建一个新字典
    return new_dict(keys, NULL);
}
```

所以整个过程分为两步，先创建`PyDictKeysObject`，然后再根据`PyDictKeysObject`创建`PyDictObject`。

因此核心逻辑就在`new_keys_object`和`new_dict`里面。

```
static PyDictKeysObject *new_keys_object(Py_ssize_t size)
{
    PyDictKeysObject *dk;
    Py_ssize_t es, usable;

    //检测，size是否>=PyDict_MINSIZE
    assert(size >= PyDict_MINSIZE);
    assert(IS_POWER_OF_2(size));

    usable = USABLE_FRACTION(size);
    //es:哈希表中的每个索引占多少字节
    //因为长度不同，哈希索引数组的元素大小也不同
    if (size <= 0xff) {
    //小于等于255，采用1字节存储
        es = 1;
    }
    else if (size <= 0xffff) {
    //小于等于65535，采用2字节存储
        es = 2;
    }
#if SIZEOF_VOID_P > 4
    else if (size <= 0xffffffff) {
    //否则采用4字节
        es = 4;
    }
#endif
    else {
        es = sizeof(Py_ssize_t);
    }

    //然后是创建PyDictKeysObject，这里会优先从缓存池中获取
```

```
    //当然，PyDictObject也有自己的缓存池
    //所以这两者都有缓存池，具体细节下一篇会详细说
    if (size == PyDict_MINSIZE && numfreekeys > 0) {
        dk = keys_free_list[--numfreekeys];
    }
    else {
        //否则malloc重新申请内存
        //注意这里申请的内存由三部分组成
        //1)PyDictKeysObject结构体本身的大小
        //2)哈希索引数组的长度乘以每个元素的大小，也就是es*size
        //3)键值对数组的长度乘上每个entry的大小
        dk = PyObject_MALLOC(sizeof(PyDictKeysObject)
                             + es * size
                             + sizeof(PyDictKeyEntry) * usable);
        if (dk == NULL) {
            PyErr_NoMemory();
            return NULL;
        }
    }
    //设置引用计数、可用的entry个数等信息
    DK_DEBUG_INCREF dk->dk_refcnt = 1;
    dk->dk_size = size;
    dk->dk_usable = usable;
    //dk_lookup很关键，它表示探测函数
    dk->dk_lookup = lookdict_unicode_nodummy;
    dk->dk_nentries = 0;
    //哈希表的初始化
    memset(&dk->dk_indices[0], 0xff, es * size);
    memset(DK_ENTRIES(dk), 0, sizeof(PyDictKeyEntry) * usable);
    return dk;
}
```

以上就是`PyDictKeysObject`的初始化过程，然后会再基于它创建`PyDictObject`，通过函数`new_dict`实现。

```
static PyObject *
new_dict(PyDictKeysObject *keys, PyObject **values)
{
    PyDictObject *mp;
    assert(keys != NULL);
    //PyDictObject的缓存池，具体细节下一篇说
    if (numfree) {
        mp = free_list[--numfree];
        assert (mp != NULL);
        assert (Py_TYPE(mp) == &PyDict_Type);
        _Py_NewReference((PyObject *)mp);
    }
    //系统堆中申请内存
    else {
        mp = PyObject_GC_New(PyDictObject, &PyDict_Type);
        if (mp == NULL) {
            DK_DECREF(keys);
            free_values(values);
            return NULL;
        }
    }
    //设置key、value等等
    mp->ma_keys = keys;
    mp->ma_values = values;
    mp->ma_used = 0;
    mp->ma_version_tag = DICT_NEXT_VERSION();
    assert(_PyDict_CheckConsistency(mp));
    return (PyObject *)mp;
}
```

以上就是字典的创建，过程应该不算复杂。下面我们再来看看，字典支持的操作是如何实现的。

给字典添加键值对

我们通过`d["name"] = "satori"`即可给字典添加一个键值对，如果键存在则修改value。

```
int
PyDict_SetItem(PyObject *op, PyObject *key, PyObject *value)
{
    PyDictObject *mp;  //字典
    Py_hash_t hash;    //哈希值
    if (!PyDict_Check(op)) {
    //不是字典则报错，该方法需要字典才可以调用
        PyErr_BadInternalCall();
        return -1;
    }
    assert(key);
    assert(value);
    mp = (PyDictObject *)op;
    //如果key不是字符串
    //或者哈希值还没有计算的话
    if (!PyUnicode_CheckExact(key) ||
        (hash = ((PyASCIIObject *) key)->hash) == -1)
    {
        //计算哈希值，PyObject_Hash是一个泛型API
        //会调用类型对象的tp_hash函数，因此等价于
        //Py_TYPE(key) -> tp_hash(key)
        hash = PyObject_Hash(key);
        if (hash == -1)
            return -1;
    }

    /* 调用insertdict，必要时调整元素 */
    return insertdict(mp, key, hash, value);
}
```

所以这一步相当于计算函数的哈希值，真正的设置键值对逻辑藏在`insertdict`里面，我们来看一下。

```
static int
insertdict(PyDictObject *mp, PyObject *key, Py_hash_t hash, PyObject *va
lue)
{
    //key对应的value
    PyObject *old_value;
    //entry
    PyDictKeyEntry *ep;

    //增加对key和value的引用计数
    Py_INCREF(key);
    Py_INCREF(value);
    //类型检查
    if (mp->ma_values != NULL && !PyUnicode_CheckExact(key)) {
        if (insertion_resize(mp) < 0)
            goto Fail;
    }
    //mp->ma_keys->dk_lookup表示获取探测函数
```

```
    //会基于传入的哈希值、key、判断哈希索引数组是否有可用的槽
    Py_ssize_t ix = mp->ma_keys->dk_lookup(mp, key, hash, &old_value);
    if (ix == DKIX_ERROR)
        //不存在，跳转至Fail
        goto Fail;

    assert(PyUnicode_CheckExact(key) || mp->ma_keys->dk_lookup == lookdi
ct);
    MAINTAIN_TRACKING(mp, key, value);
    //...
    if (ix == DKIX_EMPTY) {
    //如果ix==DKIX_EMPTY
    //说明哈希索引数组存在一个可用的槽
        assert(old_value == NULL);
        if (mp->ma_keys->dk_usable <= 0) {
            /* 判断是否需要resize */
            if (insertion_resize(mp) < 0)
                goto Fail;
        }
    //存在可用的槽，调用find_empty_slot
    //将可用槽的索引找到、并返回
        Py_ssize_t hashpos = find_empty_slot(mp->ma_keys, hash);
    //拿到PyDictKeyEntry *指针
        ep = &DK_ENTRIES(mp->ma_keys)[mp->ma_keys->dk_nentries];
    //将该entry在键值对数组中的索引存储在指定的槽里面
        dk_set_index(mp->ma_keys, hashpos, mp->ma_keys->dk_nentries);
        ep->me_key = key; //设置key
        ep->me_hash = hash;//设置哈希
    //但value还没有设置，因为还要判断哈希表的种类
    //如果ma_values数组不为空，说明是分离表
    //ma_keys只维护键
        if (mp->ma_values) {
            assert (mp->ma_values[mp->ma_keys->dk_nentries] == NULL);
        //要将value保存在ma_values中
            mp->ma_values[mp->ma_keys->dk_nentries] = value;
        }
        else {
        //否则是结合表
        //那么value就设置在PyDictKeyEntry对象的me_value里面
            ep->me_value = value;
        }

        mp->ma_used++;//使用个数+1
        mp->ma_version_tag = DICT_NEXT_VERSION();//版本数+1
        mp->ma_keys->dk_usable--;//可用数-1
        mp->ma_keys->dk_nentries++;//里面entry数量+1
        assert(mp->ma_keys->dk_usable >= 0);
        assert(_PyDict_CheckConsistency(mp));
        return 0;
    }
    //走到这里说明key已经存在了，那么此时相当于修改
    //将旧的value替换掉
    if (_PyDict_HasSplitTable(mp)) {
    //分离表，修改ma_values
        mp->ma_values[ix] = value;
        if (old_value == NULL) {
            /* pending state */
            assert(ix == mp->ma_used);
            mp->ma_used++;
        }
    }
    //结合表
    //修改ma_keys->dk_entries中指定entry的me_value
    else {
```

```
        assert(old_value != NULL);
        DK_ENTRIES(mp->ma_keys)[ix].me_value = value;
    }
    //增加版本号
    mp->ma_version_tag = DICT_NEXT_VERSION();
    Py_XDECREF(old_value);
    assert(_PyDict_CheckConsistency(mp));
    Py_DECREF(key);
    return 0;

Fail:
    Py_DECREF(value);
    Py_DECREF(key);
    return -1;
}
```

以上就是设置元素相关的逻辑，还是有点难度的，需要对着源码仔细理解一下。

根据key获取value

获取某个键对应的值，会执行PyDict_GetItem函数，但是核心逻辑是在dict_subscript函数里面，我们来看一下。

```
static PyObject *
dict_subscript(PyDictObject *mp, PyObject *key)
{
    Py_ssize_t ix;
    Py_hash_t hash;
    PyObject *value;
    //获取哈希值
    if (!PyUnicode_CheckExact(key) ||
        (hash = ((PyASCIIObject *) key)->hash) == -1) {
        hash = PyObject_Hash(key);
        if (hash == -1)
            return NULL;
    }
    //是否存在可用的槽
    //注意value传了一个指针进去
    //所以当entry存在时，会将 value 设置为指定的值
    ix = (mp->ma_keys->dk_lookup)(mp, key, hash, &value);
    if (ix == DKIX_ERROR)
        return NULL;
    //注意这里是获取元素，如果key被映射到了该槽
    //然后该槽还可用，这意味着什么呢？显然是不存在此key
    if (ix == DKIX_EMPTY || value == NULL) {
        if (!PyDict_CheckExact(mp)) {
            //如果其类型对象继承dict，那么在找不到key时
            //会执行__missing__方法
            PyObject *missing, *res;
            _Py_IDENTIFIER(__missing__);
            missing = _PyObject_LookupSpecial((PyObject *)mp, &PyId___mi
ssing__);
            //执行__missing__方法
            if (missing != NULL) {
                res = PyObject_CallFunctionObjArgs(missing,
                                                   key, NULL);
                Py_DECREF(missing);
                return res;
            }
```

```
            else if (PyErr_Occurred())
                return NULL;
        }
        //报错，KeyError
        _PyErr_SetKeyError(key);
        return NULL;
    }
    //否则就说明value获取到了
    //增加引用计数，返回value
    Py_INCREF(value);
    return value;
}
```

逻辑比较简单，重点是里面出现了__missing__方法，这个方法只有写在继承dict的类里面才有用，我们举个栗子：

```
class MyDict(dict):

    def __getitem__(self, item):
        # 执行 MyDict()["xx"]
        # 会走这里的魔法函数
        print("__getitem__")
        # 然后调用父类的__getitem__
        # 父类在执行__getitem__时发现key不存在
        # 会调用__missing__方法，并且会将key作为参数
        return super().__getitem__(item + " satori")

    def __missing__(self, key):
        print(key)
        return key.upper()


v = MyDict()["komeiji"]
"""
__getitem__
komeiji satori
"""
print(v)  # KOMEIJI SATORI
```

删除某个键值对

设置键值对如果明白了，删除键值对我觉得都不需要说了。还是根据key找到指定的槽，如果槽里面的索引是DKIX_EMPTY，那么说明根本不存在此key，KeyError；否则拿到指定的entry，将其设置为dummy。

因为删除元素不能真正的删除，所以它本质还是有点类似于修改一个键值对。

```
int
PyDict_DelItem(PyObject *op, PyObject *key)
{
    //先获取hash值
    Py_hash_t hash;
    assert(key);
    if (!PyUnicode_CheckExact(key) ||
        (hash = ((PyASCIIObject *) key)->hash) == -1) {
        hash = PyObject_Hash(key);
        if (hash == -1)
            return -1;
```

```
    }

    //真正来删除是下面这个函数
    return _PyDict_DelItem_KnownHash(op, key, hash);
}


int
_PyDict_DelItem_KnownHash(PyObject *op, PyObject *key, Py_hash_t hash)
{
    //......
    mp = (PyDictObject *)op;
    //获取对应entry的index
    ix = (mp->ma_keys->dk_lookup)(mp, key, hash, &old_value);
    if (ix == DKIX_ERROR)
        return -1;
    if (ix == DKIX_EMPTY || old_value == NULL) {
        _PyErr_SetKeyError(key);
        return -1;
    }
    //......
    //传入hash和ix，又调用了delitem_common
    return delitem_common(mp, hash, ix, old_value);
}

static int
delitem_common(PyDictObject *mp, Py_hash_t hash, Py_ssize_t ix,
               PyObject *old_value)
{
    PyObject *old_key;
    PyDictKeyEntry *ep;
    //找到指定的槽，拿到里面存储的索引
    Py_ssize_t hashpos = lookdict_index(mp->ma_keys, hash, ix);
    assert(hashpos >= 0);
    //已用的entries个数-1
    mp->ma_used--;
    //版本号增加
    mp->ma_version_tag = DICT_NEXT_VERSION();
    //拿到entry的指针
    ep = &DK_ENTRIES(mp->ma_keys)[ix];
    //先将dk_entries数组中指定的entry设置为dummy状态
    dk_set_index(mp->ma_keys, hashpos, DKIX_DUMMY);
    ENSURE_ALLOWS_DELETIONS(mp);
    old_key = ep->me_key;
    //将其key、value都设置为NULL
    ep->me_key = NULL;
    ep->me_value = NULL;
    //减少引用计数
    Py_DECREF(old_key);
    Py_DECREF(old_value);

    assert(_PyDict_CheckConsistency(mp));
    return 0;
}
```

流程非常清晰，也很简单。先计算hash值，再计算出索引，最后获取相应的entry，将me_key、me_value设置为NULL，并减少指向对象的引用计数。同时将entry从active态设置为dummy态，并调整ma_used**(已存在键值对)**的数量。

收录于合集 #CPython 97

‹ 上一篇

下一篇 ›

《源码探秘 CPython》38. 字典是如何扩容的?

《源码探秘 CPython》36. 哈希表能够直接删除元素吗?

喜欢此内容的人还喜欢

python 7天进阶之路-对象和json转换

缪斯之子

[系列]微服务·深入理解 gRPC - Part2

走向架构师的每一天

Abaqus python脚本开发 第三章 各类指令的方法对象变量 (3)

山石结构