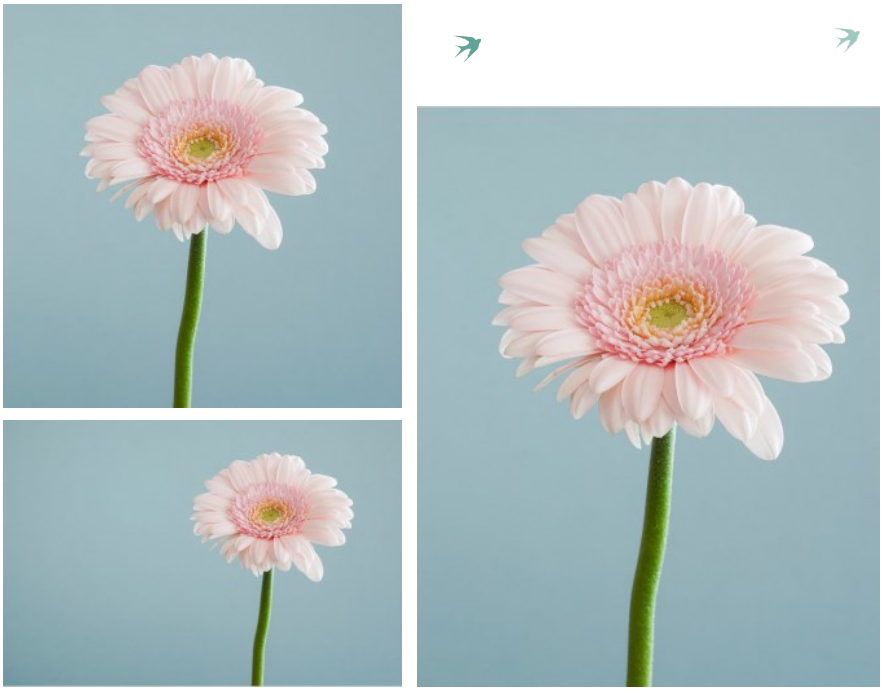
《源码探秘 CPython》41. 集合支持的操作是怎么实现的？

原创 古明地觉 古明地觉的编程教室 2022-03-05 09:00

收录于合集
#CPython
97个 >



楔子

下面我们来分析一下集合的操作是怎么实现的，比如元素的添加、删除，以及集合的扩容等等。

并且集合还支持交集、并集、差集等运算，它们又是如何实现的呢？那么就一起来看一看吧。

添加元素

添加元素，会调用PySet_Add函数。

```
int
PySet_Add(PyObject *anyset, PyObject *key)
{
    //参数是两个指针
    //类型检测
    if (!PySet_Check(anyset) &&
        (!PyFrozenSet_Check(anyset) || Py_REFCNT(anyset) != 1)) {
        PyErr_BadInternalCall();
        return -1;
    }
    //本质上调用了set_add_key
    return set_add_key((PySetObject *)anyset, key);
}
```

在进行了参数检测之后，又调用了`set_add_key`。

```c
static int
set_add_key(PySetObject *so, PyObject *key)
{
    //声明一个变量，用于保存哈希值
    Py_hash_t hash;

    //类型检测，看看是否是ASCII字符串
    if (!PyUnicode_CheckExact(key) ||
        (hash = ((PyASCIIObject *) key)->hash) == -1) {
        //如果不是ASCII字符串
        //那么计算哈希值
        hash = PyObject_Hash(key);
        //如果计算之后的哈希值为-1
        //在表示该对象不可被哈希，Python层面显然会报错
        if (hash == -1)
            return -1;
    }
    //底层又调用了set_add_entry，并把hash也作为参数传了进去
    return set_add_entry(so, key, hash);
}
```

和字典类似，这一步也不是添加元素的真正逻辑，只是计算了哈希值。显然下面的`set_add_entry`就是具体的逻辑了。

```c
static int
set_add_entry(PySetObject *so, PyObject *key, Py_hash_t hash)
{
  //...
  restart:
    //获取mask
    mask = so->mask;
    //hash和mask进行按位与，得到一个索引
    i = (size_t)hash & mask;
    //获取对应的entry指针
    entry = &so->table[i];
    if (entry->key == NULL)
        //如果entry->key == NULL
        //表示当前位置没有被使用
        //直接跳到found_unused标签
        goto found_unused;

    //否则说明该位置已经存储entry
    freeslot = NULL;
    perturb = hash; // 将perturb设置为hash

    //接下来就要改变规则，重新映射了
    while (1) {
      //获取已存在entry的hash字段的值
      //如果和我们当前的哈希值一样的话
        if (entry->hash == hash) {
        //获取已存在entry的key
            PyObject *startkey = entry->key;
        //entry里面的key不可以为dummy态
        //因为这相当于删除(伪删除)了，那么hash应该为-1
            assert(startkey != dummy);
        //如果startkey和key相等，说明指向了同一个对象
        //那么两者视为相等，而集合内的元素不允许重复
            if (startkey == key)
            //直接跳转到found_active标签
                goto found_active;
        //如果不是同一个对象，再比较维护的值是否相等
        //快分支，假设两者都是字符串，然后进行比较
            if (PyUnicode_CheckExact(startkey)
```

```
                && PyUnicode_CheckExact(key)
                && _PyUnicode_EQ(startkey, key))
            //如果一样，跳转到found_active标签
                goto found_active;

        //到这里说明两者不是同一个对象，也不都是字符串
        //那么只能走通用的比较逻辑了
            table = so->table;
        //增加startkey的引用计数
            Py_INCREF(startkey);
        //比较两个对象维护的值是否一致
            cmp = PyObject_RichCompareBool(startkey, key, Py_EQ);
        //减少startkey的引用计数
            Py_DECREF(startkey);
        //如果cmp大于0，比较成功
            if (cmp > 0)
            //说明两个值是相同的
            //跳转到found_active标签
                goto found_active;
            if (cmp < 0)
             //小于0说明比较失败
             //跳转到comparison_error标签
                goto comparison_error;
            //拿到当前的mask
            mask = so->mask;
        }
        //如果不能hash
        else if (entry->hash == -1)
            //则设置为freeslot
            freeslot = entry;

        //如果当前索引值加上9小于等于当前的mask
        //#define LINEAR_PROBES 9
        if (i + LINEAR_PROBES <= mask) {
            //循环9次，这里逻辑我们一会单独说
            for (j = 0 ; j < LINEAR_PROBES ; j++) {
                // ......
            }
        }

        //程序走到这里说明索引冲突了
        //改变规则，重新计算索引值
        perturb >>= PERTURB_SHIFT;
        //我们看到计算规则和字典是一样的
        i = (i * 5 + 1 + perturb) & mask;
        //获取新索引对应的entry
        entry = &so->table[i];
        //如果对应的key为NULL，说明重新计算索引之后找到了可以存储的地方
        if (entry->key == NULL)
            //跳转到found_unused_or_dummy
            goto found_unused_or_dummy;
        //否则说明比较倒霉，改变规则重新映射之后，索引依旧冲突
        //那么继续循环，比较key是否一致等等
    }

found_unused_or_dummy:
    //如果这个freeslot为NULL，说明是可用的
    if (freeslot == NULL)
        //跳转
        goto found_unused;
    //否则，说明为dummy态
    //那么我们依旧可以使用，正好废物利用
    //将used数量加一
    so->used++;
```

```
104     //设置key和hash值
105     freeslot->key = key;
106     freeslot->hash = hash;
107     return 0;
108 
109   //发现未使用的
110   found_unused:
111     //将fill和used个数+1
112     so->fill++;
113     so->used++;
114     //设置key和hash值
115     entry->key = key;
116     entry->hash = hash;
117     //检查active态+dummy的entry个数是否小于mask的3/5
118     if ((size_t)so->fill*5 < mask*3)
119         //是的话，表示无需扩容
120         return 0;
121     //否则要进行扩容
122     //如果active态的entry大于50000，那么两倍扩容，否则四倍扩容
123     return set_table_resize(so, so->used>50000 ? so->used*2 : so->used*4
124 );
125 
126   //如果是found_active，表示key重复了
127   //直接减少一个引用计数即可
128   found_active:
129     Py_DECREF(key);
130     return 0;
131 
132   //比较失败，同样减少引用计数，返回-1
133   comparison_error:
134     Py_DECREF(key);
135     return -1;
    }
```

代码很多，我们还删除了一部分，整个流程总结一下就是：

- 传入hash值，计算出索引值，通过索引值找到对应的entry；
- 如果entry->key=NULL，那么将hash和key存到对应的entry；
- 如果entry->key != NULL，那么就比较两个key是否相同；
- 如果相同，则不插入，直接减少引用计数。因为不是字典，不存在更新一说；
- 如果不相同，那么从该索引往后遍历9个entry，如果存在key为NULL的entry，那么设置进去；
- 如果以上条件都不满足，则改变策略重新计算索引值，直到找到一个满足key为NULL的entry；
- 判断容量问题，如果active态+dummy态的entry个数不小于3/5*mask，那么扩容，扩容的规则是active态的entry个数是否大于50000，是的话就二倍扩容，否则4倍扩容；

最后是`if (i + LINEAR_PROBES <= mask)`，这一部分代码我们省略了，那它是做什么的呢？首先哈希值相同但是key不同时，按照学习字典的思路，肯定是映射一个新的索引。

但是问题来了，这样是不能有效地利用CPU缓存的，L1 Cache加载数据会一次性加载64字节，称为一个cache line。如果两个位置间隔比较远，因为映射出来的索引是随机的，对应的entry可能不在cache中，从而导致CPU下一次需要重新读取。

所以Python中引入了**LINEAR_PROBES**，从当前的entry开始，查找前面的9个entry。如果还找不到可用位置，然后才重新计算，从而提高cache的稳定性。

所以集合和字典在解决哈希冲突的时候采取的策略是一样的，只不过集合多考虑了CPU的cache。

删除元素

删除元素会调用set_remove函数，但是删除的核心逻辑位于set_discard_entry函数中。

```
static int
set_discard_entry(PySetObject *so, PyObject *key, Py_hash_t hash)
{   //传入集合、key、以及计算的哈希值

    setentry *entry;
    PyObject *old_key;
    //通过传入的key和hash找到该entry
    //并且entry->key要和传入的key是一样的
    entry = set_lookkey(so, key, hash);
    //如果entry为NULL，说明不存在此key
    //直接返回-1
    if (entry == NULL)
        return -1;
    //如果entry不为NULL，但是对应的key为NULL
    //返回DISCARD_NOTFOUND
    if (entry->key == NULL)
        return DISCARD_NOTFOUND;
    //获取要删除的key
    old_key = entry->key;
    //并将entry设置为dummy
    entry->key = dummy;
    //hash值设置为-1
    entry->hash = -1;
    //减少使用数量
    so->used--;
    //减少引用计数
    Py_DECREF(old_key);
    //返回DISCARD_FOUND
    return DISCARD_FOUND;
}
```

如果找到了指定的key，在set_remove函数里面会返回None，否则报出KeyError。

可以看到集合添加、删除元素和字典是有些相似的，毕竟底层都是使用了哈希表嘛。

集合的扩容

当集合的容量不够时，会自动扩容，具体的逻辑位于set_table_resize函数中。

```
static int
set_table_resize(PySetObject *so, Py_ssize_t minused)
{   //显然参数是:PySetObject *指针以及容量大小

    //三个setentry *指针
    setentry *oldtable, *newtable, *entry;
    //oldmask
    Py_ssize_t oldmask = so->mask;
    //newmask
    size_t newmask;

    //是否为其申请过内存
    int is_oldtable_malloced;
    //将PySet_MINSIZE个entry直接copy过来
```

```
    setentry small_copy[PySet_MINSIZE];
    //minused必须大于等于0
    assert(minused >= 0);
    //newsize不断扩大二倍，直到大于minused
    //所以我们刚才说的大于50000，二倍扩容，否则四倍扩容
    //实际上是最终的newsize是比二倍或者四倍扩容的结果要大的
    size_t newsize = PySet_MINSIZE;
    while (newsize <= (size_t)minused) {
        //newsize最大顶多也就是PY_SSIZE_T_MAX+1
        //但一般不可能有这么多元素
        newsize <<= 1;
    }
    //为新的table申请空间
    oldtable = so->table;
    assert(oldtable != NULL);
    is_oldtable_malloced = oldtable != so->smalltable;

    //如果newsize和PySet_MINSIZE(这里的8)相等
    if (newsize == PySet_MINSIZE) {

        //拿到smalltable，就是默认初始化8个entry数组的那个成员
        newtable = so->smalltable;
        //如果oldtable和newtable一样
        if (newtable == oldtable) {
            //并且没有dummy态的entry
            if (so->fill == so->used) {
                //那么无需做任何事情
                return 0;
            }
            //否则的话，dummy的个数一定大于0
            assert(so->fill > so->used);
            //扔掉dummy态，只把oldtable中active态的entry拷贝过来
            memcpy(small_copy, oldtable, sizeof(small_copy));
            //将small_copy重新设置为oldtable
            oldtable = small_copy;
        }
    }
    else {
        //否则的话，肯定大于8，申请newsize个setentry所需要的空间
        newtable = PyMem_NEW(setentry, newsize);
        //如果newtable为NULL，那么申请内存失败，返回-1
        if (newtable == NULL) {
            PyErr_NoMemory();
            return -1;
        }
    }
    //newtable肯定不等于oldtable
    assert(newtable != oldtable);
    //创建一个能容纳newsize个entry的空set
    memset(newtable, 0, sizeof(setentry) * newsize);
    //将mask设置为newsize-1
    //将table设置为newtable
    so->mask = newsize - 1;
    so->table = newtable;
    //获取newmask
    newmask = (size_t)so->mask;
    //遍历旧table的setentry数组
    //将setentry的key和hash全部设置到新的table里面
    //如果fill==used，说明没有dummy态的entry
    if (so->fill == so->used) {
        for (entry = oldtable; entry <= oldtable + oldmask; entry++) {
            if (entry->key != NULL) {
            //设置元素的逻辑在此函数中
                set_insert_clean(newtable, newmask, entry->key, entry->h
```

```
ash);
            }
        }
    } else {
    //逻辑和上面一样，但是存在dummy态的entry
    //判断时需要多一个条件:entry->key != dummy
    //由于会丢弃dummy态的entry，因此扩容后fill和used相等
    //所以这里将used赋值给fill
        so->fill = so->used;
    //另外估计有人觉得这里的代码有点啰嗦
    //代码是类似的，没必要分成两个分支
    //其实这是Python为了性能考虑的
    //如果fill==used，说明不存在dummy太的entry
    //那么遍历时就无需加上entry->key != dummy这个条件了
        for (entry = oldtable; entry <= oldtable + oldmask; entry++) {
            if (entry->key != NULL && entry->key != dummy) {
                set_insert_clean(newtable, newmask, entry->key, entry->h
ash);
            }
        }
    }

    //如果已经为旧的table申请了内存，那么要将其归还给系统堆
    if (is_oldtable_malloced)
        PyMem_DEL(oldtable);
    return 0;
}
```

整个逻辑还是不难理解的，该函数内部负责申请内存，初始化成员。但是设置元素的核心逻辑位于`set_insert_clean`中，我们看一下。

```
static void
set_insert_clean(setentry *table, size_t mask, PyObject *key, Py_hash_t
hash)
{
    setentry *entry;
    //perturb初始值为hash
    size_t perturb = hash;
    //计算索引
    size_t i = (size_t)hash & mask;
    size_t j;

    while (1) {
        //获取当前entry
        entry = &table[i];
        if (entry->key == NULL)
        //如果为空则跳转found_null设置key与hash
            goto found_null;
        if (i + LINEAR_PROBES <= mask) {
            //否则还是老规矩，遍历之后的9个entry
            for (j = 0; j < LINEAR_PROBES; j++) {
                entry++;
            //找到空的entry，那么跳转到found_null设置key与hash
                if (entry->key == NULL)
                    goto found_null;
            }
        }
        // 没有找到，那么改变规则，重新计算索引
        perturb >>= PERTURB_SHIFT;
        i = (i * 5 + 1 + perturb) & mask;
    }
 found_null:
    //设置key与hash
    entry->key = key;
```

```
    entry->hash = hash;
}
```

以上就是集合的扩容，我们又看到了字典的影子。

集合的交集运算

我们在使用集合的时候，可以取两个集合的交集、并集、差集、对称差集等等。这里介绍一下交集，其余的可以自己参考源码研究一下，源码位于setobject.c中。

```c
static PyObject *
set_intersection(PySetObject *so, PyObject *other)
{
    //result，集合运算之后会产生新的集合
    PySetObject *result;
    PyObject *key, *it, *tmp;
    Py_hash_t hash;
    int rv;

    //如果两个对象相同
    if ((PyObject *)so == other)
        //直接返回其中一个的拷贝即可
        return set_copy(so);

    //这行代码表示创建一个空的PySetObject *
    result = (PySetObject *)make_new_set_basetype(Py_TYPE(so), NULL);
    //如果result == NULL，说明创建失败
    if (result == NULL)
        return NULL;

    //检测other是不是PySetObject *
    if (PyAnySet_Check(other)) {
        //初始索引为0
        Py_ssize_t pos = 0;
        //setentry *
        setentry *entry;

        //如果other元素的个数大于so
        if (PySet_GET_SIZE(other) > PySet_GET_SIZE(so)) {
            //就把so和other进行交换
            tmp = (PyObject *)so;
            so = (PySetObject *)other;
            other = tmp;
        }

        //从少的那一方的开始遍历
        while (set_next((PySetObject *)other, &pos, &entry)) {
            //拿到key和hash
            key = entry->key;
            hash = entry->hash;
            //传入other的key和hash，在so中去找
            rv = set_contains_entry(so, key, hash);
            if (rv < 0) {
                //如果rv<0，说明不存在
                Py_DECREF(result);
                return NULL;
            }
            if (rv) {
```

```
                //存在的话设置进result里面
                if (set_add_entry(result, key, hash)) {
                    Py_DECREF(result);
                    return NULL;
                }
            }
        }
        //直接返回
        return (PyObject *)result;
    }
    //...
}
```

逻辑比我们想象中的要单纯，假设有两个集合S1和S2，遍历元素少的集合，然后判断元素在另一个集合中是否存在。如果存在，则添加进要返回的集合中，否则遍历下一个。

小结

以上就是集合相关的内容，它的效率也是非常高的，能够以O(1)的复杂度去查找某个元素。最关键的是，它用起来也特别的方便。

此外Python里面还有一个frozenset，也就是不可变的集合。但frozenset对象和set对象都是同一个结构体，只有PySetObject，没有PyFrozenSetObject。

我们在看PySetObject的时候，发现该对象里面也有一个hash成员，如果是不可变集合，那么hash值是不为-1的，因为它不可以添加、删除元素，是不可变对象。由于比较相似，因此frozenset就不再说了，可以自己对着源码简单看一下，源码还是setobject.c。

收录于合集 #CPython 97

‹ 上一篇
《源码探秘 CPython》42. 迭代器的实现原理

下一篇 ›
《源码探秘 CPython》40. 集合是怎么实现的?