13 dict 对象，高效的关联式容器-慕课专栏

Python 中的 *dict* 对象是一种 **关联式容器** 对象，用于保存由 **键** (*key*)到 **值** (*value*)的映射关系。借助关联式容器，程序可快速定位到与指定 **键** 相关联的 **值** 。*dict* 对象在 *Python* 程序中使用频率非常高，如果应用不当将严重影响程序的执行效率。

本节，我们先从 *dict* 对象常用操作入手，回顾它的 **基本用法** ；接着结合其他内建容器对象，研究 *dict* 关键操作的 **执行效率** ；最后以 *dict* 对象 **内部结构** 收尾，详细讲解其内部 **哈希表** 的实现要点，以及其中几个关键 **性能考量** 。相信通过本节学习，读者将对 *dict* 实现原理了如指掌，这对用好 *dict* 对象至关重要。

基本用法

我们用一个 *dict* 对象来保存培训班学员的成绩，先创建一个空对象：

```
>>> scores = {}
>>> scores
{}
```

那么，一个什么都不放的 *dict* 对象需要占用多少内存呢？根据前面章节，我们知道对象头部字段是必不可少的。可我们很惊讶地发现，一个空的 dict 对象居然要占用 *240* 字节的内存！

```
>>> import sys
>>> sys.getsizeof(scores)
240
```

这是为什么呢？后续我们将从 *dict* 内部的哈希表中寻找答案。现在我们接着回顾 *dict* 的基本用法。

现在将 *jim* 的成绩保存保存到 *dict* 对象中：

```
>>> scores['jim'] = 70
>>> scores
{'jim': 70}
>>> sys.getsizeof(scores)
240
```

数据插入后，我们发现 *dict* 对象内存使用量保存不变。看来， *dict* 对象也有一种类似 *list* 对象的 **预分配机制** 。

现在，接着存入 *lily* 、*lucy* 以及 *tom* 的成绩。我们发现， *dict* 还没达到扩容条件，内存大小保存不变：

```
>>> scores['lily'] = 75
>>> scores['lucy'] = 80
>>> scores['tom'] = 90
>>> scores['alice'] = 95
>>> scores
{'jim': 70, 'lily': 75, 'lucy': 80, 'tom': 90, 'alice': 95}
>>> sys.getsizeof(scores)
240
```

借助 *dict* 对象，我们可以快速检索出某位学员的成绩。例如，获取 tom 的成绩：

```
>>> scores['tom']
90
```

"快速"不是一个精确的形容词，到底多快呢？这里先给出答案，由于 *dict* 对象底层由哈希表实现， 查找操作平均时间复杂度是 O(1)O(1)O(1) 。当然了，在哈希不均匀的情况下，最坏时间复杂度是 O(n)O(n)O(n) ，但一般情况下很难发生。

当然了，如果有某位学员(例如 *lily*)转学了，可通过 *pop* 方法将其剔除：

```
>>> scores.pop('lily')
75
>>> scores
{'jim': 70, 'lucy': 80, 'tom': 90, 'alice': 95}
```

哈希表结构决定了 *dict* 的删除操作也很快，平均时间复杂度也是 O(1)O(1)O(1) 。实际上， *dict* 插入、删除、查找的平均时间复杂度都是 ，O(1)O(1)O(1)最坏时间复杂度是 O(n)O(n)O(n) 。因此，哈希函数的选择就至关重要，一个好的哈希函数应该将键尽可能 **均匀** 地映射到哈希空间中，最大限度地避免 **哈希冲突** 。

执行效率

我们知道 *dict* 对象搜索操作时间复杂度为 O(1)O(1)O(1)，远远好于 *list* 对象的 O(n)O(n)O(n) 。这意味着什么了？为得到一个更准确、直观的感受，我们编写一个测试程序，分别测试不同规模 *dict* 、 *list* 对象完成 *1000* 次搜索所需的时间：

```
import random
import time


randint = lambda: random.randint(-2**30+1, 2**30-1)

def count_targets(items, targets):
    '''
    计算目标对象出现个数
        items: 待搜索容器
        targets: 待搜索目标元素列表
    '''

    found = 0
```

```
    for target in targets:
        if target in items:
            found += 1

    return found

def generate_random_dict(n):
    '''
    生成随机数字典
    '''

    dict_items = {}
    while len(dict_items) < n:
        dict_items[randint()] = True

    return dict_items

def generate_random_list(n):
    '''
    生成随机数列表
    '''

    return [
        randint()
        for _ in range(0, n)
    ]

def test_for_scale(scale, targets):
    '''
    执行一个样例
        scale: 测试容器规模
        targets: 待搜索元素列表
    '''


    dict_items = generate_random_dict(scale)
    list_items = generate_random_list(scale)


    start_ts = time.time()
    count_targets(dict_items, targets)
    dict_time = time.time() - start_ts


    start_ts = time.time()
    count_targets(list_items, targets)
    list_time = time.time() - start_ts


    print('Scale:', scale)
    print('Dict:', dict_time)
    print('List:', list_time)
    print()
```

```
def main():


    targets = generate_random_list(1000)


    for scale in [1000, 10000, 100000, 1000000]:
        test_for_scale(scale, targets)

if __name__ == '__main__':
    main()
```

测试程序代码逻辑并不复杂，请结合注释阅读理解，这里不再赘述。测试程序执行后，输出内容大致如下：

```
Scale: 1000
Dict: 0.00012683868408203125
List: 0.03683590888977051

Scale: 10000
Dict: 0.00017213821411132812
List: 0.3484950065612793

Scale: 100000
Dict: 0.00021696090698242188
List: 3.6795387268066406

Scale: 1000000
Dict: 0.0003829002380371094
List: 48.04447102546692
```

我们将测试结果制作表格， *dict* 和 *list* 的表现一目了然：

容器规模	增长系数	dict消耗时间	dict增长系数	list消耗时间	list增长系数
1000	1	0.000129s	1	0.036s	1
10000	10	0.000172s	1.33	0.348s	9.67
100000	100	0.000216s	1.67	3.679s	102.19
1000000	1000	0.000382s	2.96	48.044s	1335.56

从表格中，我们看到：当容器规模增长 *1000* 倍， *dict* 搜索时间几乎保持不变，但 *list* 搜索时间增长了差不多 *1000* 倍。当规模达到 *10* 万时，*1000* 次 *list* 搜索花了接近一分钟时间，而 *dict* 只需 382 微秒！*dict* 对象完成一次搜索只需 *0.382* 微秒，也就是说一秒钟可以完成 *200* 多万次搜索！

dict 对象到底用了什么黑科技呢？接下来，我们一起从它的内部结构中寻找答案。

内部结构

由于关联式容器使用场景非常广泛，几乎所有现代编程语言都提供某种关联式容器，而且特别关注键的 **搜索效率** 。例如， *C++* 标准模板库中的 *map* 就是一种关联式容器，内部基于 **红黑树** 实现。红黑树是一种 **平衡** 二叉树，能够提供良好的操作效率，插入、删除、搜索等关键操作的时间复杂度均为 O(log2n)O(log_2 n)O(log2n) 。

Python 虚拟机的运行重度依赖 *dict* 对象，包括 **名字空间** 以及 **对象属性空间** 等概念底层都是由 *dict* 对象实现的。因此， *Python* 对 *dict* 对象的效率要求更为苛刻。那么，操作效率优于 O(log2n)O(log_2n)O(log2n) 的数据结构有哪些呢？好吧，你可能已经猜到了， *Python* 中的 *dict* 对象就是基于 **散列表** 实现的。

现在，是时候揭开 dict 对象神秘的面纱了。 *dict* 对象在 *Python* 内部由结构体 *PyDictObject* 表示， *PyDictObject* 在头文件 *Include/dictobject.h* 中定义：

```
typedef struct {
   PyObject_HEAD


   Py_ssize_t ma_used;


   uint64_t ma_version_tag;

   PyDictKeysObject *ma_keys;


   PyObject **ma_values;
} PyDictObject;
```

dict 对象理论上应该是一种变长对象，但 *PyObject_HEAD* 头部告诉我们， *Python* 其实把它作为普通对象实现。除了对象公共头部外， *PyDictObject* 还包括以下几个字段：

- *ma_used* ，对象当前所保存的 **键值对个数** ；
- *ma_version_tag* ，对象当前 **版本号** ，每次修改时更新；
- *ma_keys* ，指向按键对象映射的 **哈希表** 结构；
- *ma_values* ， 分离模式下指向由所有 **值对象** 组成的数组。

到目前为止，我们还没找到哈希表的具体结构，但是已经发现了一些蛛丝马迹——*PyDictKeysObject* 。现在我们趁热打铁，扒开 *PyDictKeysObject* 的源码看一看，*Objects/dict-common.h* 头文件中 *_dictkeysobject* 是也：

```
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[];

};
```

_dictkeysobject 结构体包含 dict 对象哈希表实现的所有秘密，结合注释可以解读其中的关键字段：

- *dk_refcnt* ，引用计数，跟 **映射视图** 的实现有关，有点类似对象引用计数；
- *dk_size* ，哈希表大小，必须是 2n2^n2n ，这样可将模运算优化成 **按位与** 运算；
- *dk_lookup* ， **哈希查找函数** 指针，可根据 *dict* 当前状态选用最优函数版本；
- *dk_usable* ，键值对数组 **可用个数** ；
- *dk_nentries* ，键值对数组 **已用个数** ；
- *dk_indices* ，哈希表 **起始地址** ，哈希表后紧接着 **键值对数组** *dk_entries* 。

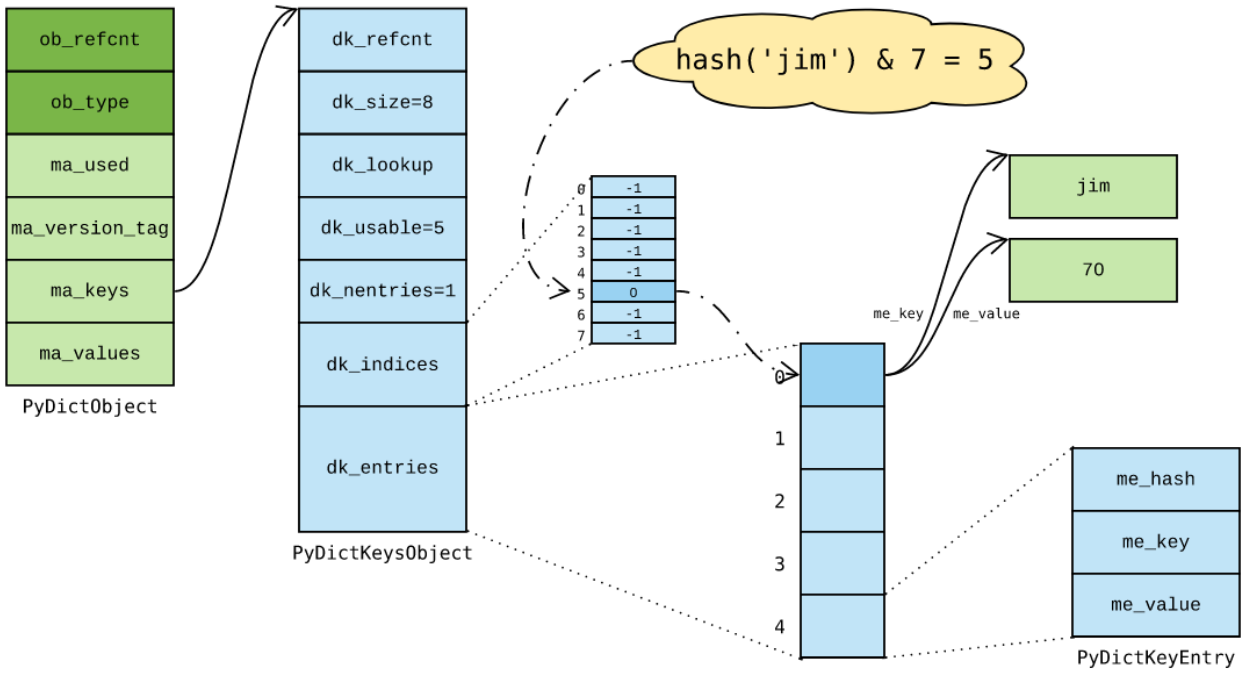
键值对结构体 *PyDictKeyEntry* 就非常直白了，除了保存键对象和值对象的指针外，缓存着键对象的哈希值：

```
typedef struct {

    Py_hash_t me_hash;
    PyObject *me_key;
    PyObject *me_value;
} PyDictKeyEntry;
```

- *me_hash* ，键对象的 **哈希值** ，避免重复调用 __hash__ 计算哈希值；
- *me_key* ，键对象指针；
- *me_value* ，值对象指针。

至此， *dict* 对象内部的哈希表结构已经非常清晰了：



dict 对象真正的实现藏身于 *PyDictKeysObject* 中，它内部包含两个关键数组，一个是 **键值对数组** *dk_entries* ，另一个是 **哈希索引数组** *dk_indices* 。*dict* 所维护的键值对，按照先来后到的顺序保存于键值对数组中；而哈希索引数组对应槽位则保存着键值对在数组中的位置。

如上图，当我们往空 *dict* 对象插入新键值对 *('jim', 70)* 时， *Python* 执行以下步骤：

1. 将键值对保存于 *dk_entries* 数组末尾，即下标为 *0* 的位置；
2. 计算键对象 *'jim'* 的哈希值并取右 *3* 位，得到该键在哈希索引数组中的下标 5 ；
3. 将键值对在数组中的下标 *0* ，保存于哈希索引数组中编号为 *5* 的槽位中。

这样一来，查找操作便可快速进行，分为以下几个步骤：

1. 计算键对象 *'jim'* 的哈希值并取右 *3* 位，得到该键在哈希索引数组中的下标 5 ；
2. 找到哈希索引数组下标为 *5* 的槽位，取出其中保存的下标 0 ；
3. 找到键值对数组第 0 个位置，并取出 **值对象** ；

由于 **哈希值计算** 以及 **数组定位** 均可在常数时间内完成，以上操作均可在常数时间内完成，也就是 O(1)O(1)O(1) 。为简化讨论，*Python* 应对 **哈希冲突** 的策略我们先按下不表，留在下节 **源码精讲** 中详细展开。

容量策略

根据行为我们断定， *dict* 对象也有一种类似 *list* 对象的 **预分配机制** 。那么， *dict* 对象容量管理策略是怎样的呢？

由 *Objects/dictobject.c* 源文件中的 *PyDict_MINSIZE* 宏定义，我们知道 *dict* 内部哈希表最小长度为 *8* ：

```
#define PyDict_MINSIZE 8
```

哈希表越密集，哈希冲突则越频繁，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。由于 **内存开销** 的制约，哈希表不可能无限度稀疏，需要在时间和空间上进行权衡。实践经验表明，一个 12\frac{1}{2}21到 23\frac{2}{3}32 满的哈希表，性能较为理想——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，进而控制哈希冲突频率， *Python* 通过 *USABLE_FRACTION* 宏将哈希表内元素控制在 23\frac{2}{3}32 以内。*USABLE_FRACTION* 宏根据哈希表规模 *n* ，计算哈希表可存储元素个数，也就是 **键值对数组** 的长度。以长度为 8 的哈希表为例，最多可以保持 *5* 个键值对，超出则需要扩容。USABLE_FRACTION 是一个非常重要的宏定义，位于源文件 *Objects/dictobject.c* 中：

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

哈希表规模一定是 2n2^n2n ，也就是说 *Python* 采用 **翻倍扩容** 策略。例如，长度为 *8* 的哈希表扩容后，长度变为 *16* 。

最后，我们来考察一个空的 dict 对象所占用的内存空间。*Python* 为空 dict 对象分配了一个长度为 *8* 的哈希表，因而也要占用相当多的内存，主要有以下几个部分组成：

- 可收集对象链表节点，共 24 字节，在此不再展开， **垃圾回收机制** 讲解中有详细介绍；
- *PyDictObject* 结构体，*6* 个字段，共 *48* 字节；
- *PyDictKeysObject* 结构体，除两个数组外有 *5* 个字段，共 *40* 字节；
- 哈希索引数组，长度为 *8* ，每个槽位 *1* 字节，共 *8* 字节；
- 键值对数组，长度为 *5* ，每个 *PyDictKeyEntry* 结构体 *24* 字节，共 *120* 字节。

```
>>> sys.getsizeof({})
240
```

内存优化

在 *Python* 早期，哈希表并没有分成两个数组实现，而是由一个键值对数组实现，这个数组也承担哈希索引的角色：

如上图，由一个键值对数组充当哈希表，哈希值直接在数组中定位到键值对，无须分成两个步骤，似乎更好。那么， *Python* 为啥多此一举将哈希表实现分为两个数组来实现呢？答案是处于 **内存** 考量。

由于哈希表必须保持 **稀疏** ，最多只有 23\frac{2}{3}32 满，这意味着要浪费至少 13\frac{1}{3}31 的内存空间。更雪上加霜的是，一个键值对条目 *PyDictKeyEntry* 大小达 *24* 字节，试想一个规模为 *65536* 的哈希表，将浪费高达 *512KB* 的内存空间：65536∗13∗24=524288 65536 * \frac{1}{3} * 24 = 524288 65536∗31∗24=524288

为了尽量节省内存， *Python* 将键值对数组压缩到原来的 23\frac{2}{3}32 ，只负责存储，索引由另一个数组负责。由于索引数组只需要保存 **键值对数组** 的下标，而整数占用的内存空间只是若干字节，因此可以节约大量内存。

索引数组 可根据哈希表规模，选择尽量小的整数类型。对于规模不超过 *256* 的哈希表，选择 *8* 位整数即可；对于长度不超过 *65536* 的哈希表， *16* 位整数足矣；其他以此类推。

以长度为 *8* 的哈希表为例，键值对数组只需分配 *5* 个单元，索引数组每个单元只需 1 字节即可：

这样一来，以区区 *8* 字节的代价，挽回了 *72* 字节的开销，还是节约了 *64* 字节！虽然 *64* 字节不足挂齿，但对于规模较大的哈希表，节约下来的内存还是相当可观的：

哈希表规模	条目表规模	旧方案	新方案	节约内存
8	8 * 2 / 3 = 5	24 * 8 = 192	1 * 8 + 24 * 5 = 128	64
256	256 * 2 / 3 = 170	24 * 256 = 6144	1 * 256 + 24 * 170 = 4336	1808
65536	65536 * 2 / 3 = 43690	24 * 65536 = 1572864	2 * 65536 + 24 * 43690 = 1179632	393232

由此可见，*Python* 作者们为节约内存可谓是殚精竭虑，以后在面试中又有一个不错的谈资！

小结

本节，我们考察了 *dict* 对象的搜索效率，并深入源码研究其内部 **哈希表** 实现，收获颇多：

1. *dict* 是一种高效的关联式容器，每秒完成高达 *200* 多万次搜索操作；
2. *dict* 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；
3. 为优化内存使用， *Python* 将 *dict* 哈希表分为 **哈希索引** 和 **键值对** 两个数组来实现；
4. 哈希表在 12\frac{1}{2}21 到 23\frac{2}{3}32 满时，性能较为理想，较好地平衡了 **内存开销** 与 **搜索效率** ；