《源码探秘 CPython》32. 初识哈希表

原创 古明地觉 古明地觉的编程教室 2022-02-18 08:30

楔子

Python的字典是一种映射型容器对象，保存了**键(key)**到**值(value)**的映射关系。通过字典，我们可以快速的实现查找，json这种数据结构也是借鉴了Python的字典。而且字典是经过高度优化的，因为Python底层也在大量地使用字典。

在Python里面我们要如何创建一个字典呢？

```
# 创建一个字典
d = {"a": 1, "b": 2}
print(d)  # {'a': 1, 'b': 2}

# 或者我们还可以通过dict，传入关键字参数即可
d = dict(a=1, b=2, c=3, d=4)
print(d)  # {'a': 1, 'b': 2, 'c': 3, 'd': 4}

# 当然dict里面还可以接收位置参数，但是最多接收一个
d1 = dict({"a": 1, "b": 2}, c=3, d=4)
d2 = dict([("a", 1), ("b", 2)], c=3, d=4)
print(d1)  # {'a': 1, 'b': 2, 'c': 3, 'd': 4}
print(d2)  # {'a': 1, 'b': 2, 'c': 3, 'd': 4}


# 还可以根据已有字典创建新的字典
d = {**{"a": 1, "b": 2}, "c": 3, **{"d": 4}}
print(d)  # {'a': 1, 'b': 2, 'c': 3, 'd': 4}

#当然通过dict也是可以的
#但是注意：**这种方式本质上是把字典变成多个关键字参数
#所以里面的key一定要符合Python的变量规范
d = dict(**{"a": 1, "b": 2}, c=3, **{"d": 4})
print(d)  # {'a': 1, 'b': 2, 'c': 3, 'd': 4}

try:
    # 这种是不合法的，因为**{1: 1}等价于1=1
    d = dict(**{1: 1})
except Exception as e:
    print(e)  # keywords must be strings
# 但下面是合法的
d = {**{1: 1, 2: 2}, **{(1, 2, 3): "嘿嘿"}}
print(d)  # {1: 1, 2: 2, (1, 2, 3): '嘿嘿'}
```

字典的底层是借助哈希表实现的，什么是哈希表我们一会儿说，总之字典添加元素、删除元素、查找元素等操作的平均时间复杂度是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)，但是这种情况很少发生。

我们来测试一下字典的执行效率吧，看看它和列表之间的区别。

```
import time
import numpy as np


def test(count: int, value: int):
    """
```

```
    :param count: 循环次数
    :param value: 查询的元素
    :return:
    """
    # 有一千万个随机数的列表
    lst = list(np.random.randint(0, 2 ** 30, size=1000))
    # 根据这个列表构造出含有一千万个键值对的字典
    d = dict.fromkeys(lst)

    # 查询元素value是否在列表中，循环count次，并统计时间
    t1 = time.perf_counter()
    for _ in range(count):
        value in lst
    t2 = time.perf_counter()
    print("列表查询耗时:", round(t2 - t1, 2))

    # 查询元素value是否在字典中，循环count次，并统计时间
    t1 = time.perf_counter()
    for _ in range(count):
        value in d
    t2 = time.perf_counter()
    print("字典查询耗时:", round(t2 - t1, 2))


# 分别查询一千次、一万次、十万次、二十万次
test(10 ** 3, 22333)
"""
列表查询耗时: 0.13
字典查询耗时: 0.0
"""
test(10 ** 4, 22333)
"""
列表查询耗时: 1.22
字典查询耗时: 0.0
"""
test(10 ** 5, 22333)
"""
列表查询耗时: 12.68
字典查询耗时: 0.01
"""
test(10 ** 5 * 2, 22333)
"""
列表查询耗时: 25.72
字典查询耗时: 0.01
"""
```

字典的查询速度非常快，从测试中我们看到，随着循环次数越来越多，列表所花费的总时间越来越长。但是字典由于查询所花费的时间极少，查询速度非常快，所以即便循环50万次，花费的总时间也不过才0.01秒左右。

此外字典还有一个特点，就是它的**快**不会受到数据量的影响，从含有一万个键值对的字典中查找，和从含有一千万个键值对的字典中查找，两者花费的时间几乎是没有区别的。

那么哈希表到底是什么样的数据结构，为什么能这么快呢？下面来分析一下。

什么是哈希表

由于映射型容器的使用场景非常广泛，几乎所有现代语言都支持映射型容器，而且特别关注**键**的搜索效率。例如：C++标准模板库中的 map 就是一种映射型容器，内部基于

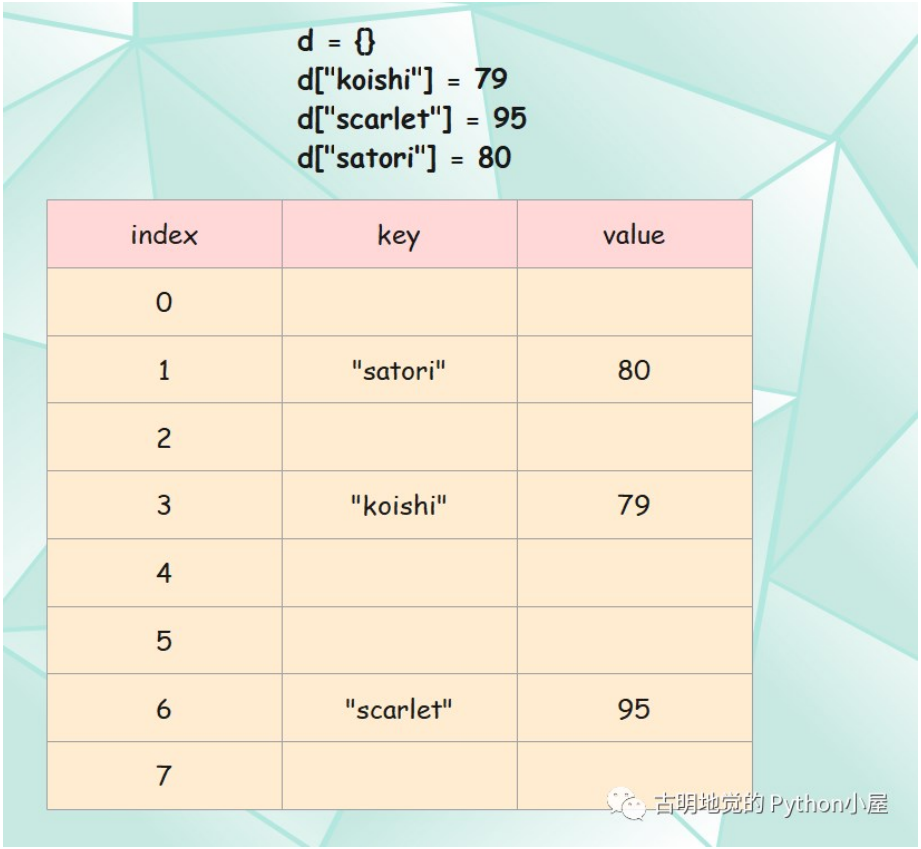
红黑树实现。红黑树是一种平衡二叉树，能够提供良好的操作效率，插入、删除、搜索等关键操作的时间复杂度均为**O(logN)**，另外Linux的epoll也使用了红黑树。

而对于Python来讲，映射型容器指的就是**字典**，我们说字典在Python内部是被高度优化的。因为不光我们在用，Python虚拟机在运行时也重度依赖字典，比如：自定义类、以及其实例对象都有自己的属性字典，还有命名空间本质上也是一个字典，因此Python对字典的性能要求会更加苛刻。

所以Python在实现字典时采用的数据结构，在添加、删除、查询元素等方面肯定是要优于红黑树的，没错，就是**哈希表**、也称**散列表**。

我们在介绍元组的时候，说元组可以作为字典的key，但是列表不可以，就是因为列表是不可哈希的。而哈希表的原理是将key通过哈希函数进行运算，得到一个哈希值，再将这个哈希值映射成索引。因此这就有一个前提，就是你的key不可以变，而列表是个可变对象，因此它不可以作为字典的key。

直接这么说的话，很难解释清楚，我们画一张图。



我们发现除了key、value之外，还有一个index，其实哈希表本质上也是使用了索引。我们知道虽然数组在遍历的时候是个时间复杂度为**O(n)**的操作，但是通过索引定位元素则是一个时间复杂度为**O(1)**的操作，不管数组有多长，通过索引总是能瞬间定位到指定元素。

所以哈希表实际上也是使用了**数组**的思想，会将**key**映射成一个数值，作为索引。至于它是怎么映射的，我们后面再谈，现在我们就假设是按照我们接下来说的方法映射的。

比如我们这里有一个能容纳8个元素的字典，如上图所示。我们先设置**d["koishi"]=79**，那么会对**koishi**这个字符串进行哈希运算，得到一个哈希值，然后再让哈希值对当前的总容量进行取模，这样的话是不是能够得到一个小于8的数呢？假设是3，那么就存在索引为3的位置。

然后**d["scarlet"]=83**，那么按照同样的规则运算得到6，那么就存在索引为6的位置；同理第三次设置**d["satori"]=80**，对字符串**satori**进行哈希、取模，得到1，那么存储在索引为1的位置。

同理当我们根据键来获取值的时候，比如：**d["satori"]**，那么同样会对字符串**satori**进行哈希、取模，得到索引发现是1，直接把索引为1的value给取出来。

当然这种方式肯定存在缺陷，比如：

- 不同的key进行哈希、取模运算之后得到的结果一定是不同的吗？
- 在运算之后得到索引的时候，发现这个位置已经有人占了怎么办？
- 取值的时候，索引为1，可如果索引为1对应的key和我们指定的key不一致怎么办？

所以哈希运算是会冲突的，如果冲突，那么Python底层会改变策略重新映射，直到映射出来的索引没有人用。比如我们设置一个新的键值对**d["tomoyo"]=88**，可是**tomoyo**这个key映射之后得到的结果也是1，而索引为1的地方已经被key为**satori**的键给占了，那么Python就会改变规则来对**tomoyo**重新进行运算，找到一个空位置进行添加。但如果我们再次设置**d["satori"]=100**，那么对**satori**映射得到的结果也是1，而key是一致的，那么就会把对应的值进行修改。

同理，当我们获取值的时候，比如**d["tomoyo"]**，那么对key进行映射，得到索引。但是发现该索引位置对应的key不是**tomoyo**而是**satori**，于是改变规则(这个规则跟设置key冲突时，采用的规则是一样的)，重新映射，得到新的索引，然后发现key是一致的，于是将值取出来。

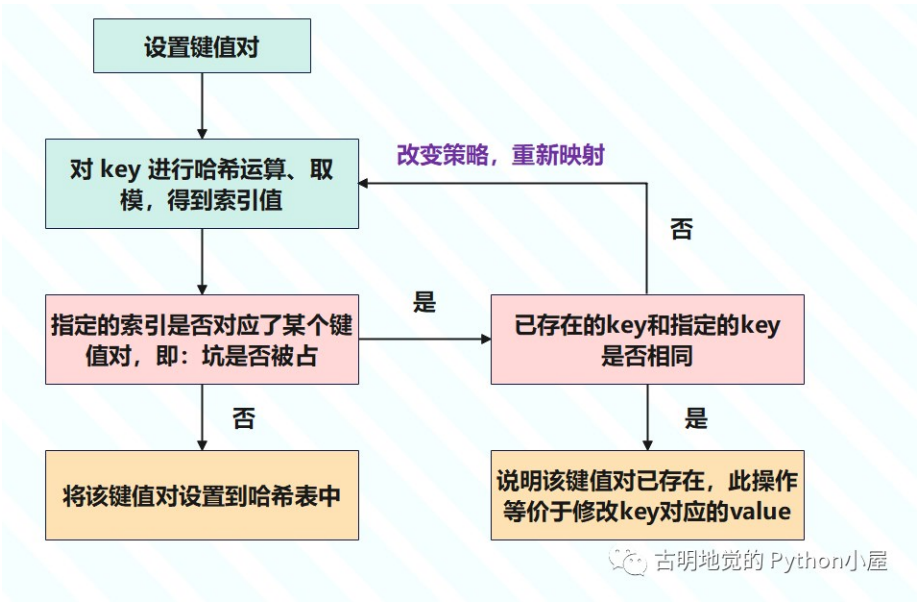
所以从这里就已经能说明问题了，就是把key转换成类似数组的索引。可能有人问，这些值貌似不是连续的啊。对的，肯定不是连续的。并不是说你先存，你的索引就小、就在前面，这是由key进行哈希运算之后的结果决定的。而且哈希表、或者说字典也会扩容，并且它还不是像列表那样，容量不够才扩容，而当元素个数达到容量的三分之二的时候就会扩容。

因为字典不可能会像列表那样，元素之间是连续、一个一个挨在一起。既然是哈希运算，得到的哈希值肯定是随机的，再根据哈希值映射出的索引也是随机的。那么在元素个数达到容量三分之二的时候，计算出来的索引发生碰撞的概率会非常大，不可能等到容量不够了再去扩容，而是在元素个数达到容量的三分之二时就要扩容，也就是申请一个更大的哈希表。

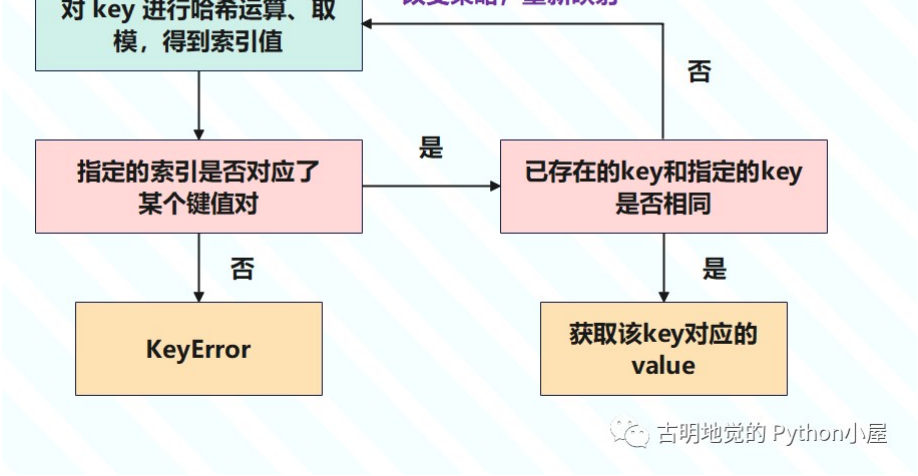
一句话总结：哈希表就是一种空间换时间的方法

假设容量为1024，那么就相当于有1024个位置，每个key都会映射成索引，找到自己的位置。各自的位置是不固定的，肯定会空出来很多，但是无所谓，只要保证这些键值对在1024个位置上是相对有序，通过索引可以在相应的位置找到它即可。

大量的文字会有些枯燥，我们来用两张图来解释一下设置元组和获取元素的整个过程。



以上是设置元素，还是比较清晰的，果然图像是个好东西。再来看看获取元素：



小结

以上就是哈希表的基本原理，这也是Python早期所采用的哈希表，但是它有一个严重的问题，就是内存浪费严重。那么后续我们来看看Python是如何进行优化的，下一篇文章就来介绍字典的底层结构。

收录于合集 #CPython 97

‹ 上一篇
《源码探秘 CPython》33. 字典是怎么实现的?

下一篇 ›
《源码探秘 CPython》31. 元组是怎么实现的?