《源码探秘 CPython》34. 对象的哈希值

原创 古明地觉 古明地觉的编程教室 2022-02-22 08:30

楔子

通过字典的底层实现，我们找到了字典快速且高效的秘密，就是哈希表。对于映射型容器，一般会采用平衡搜索树或哈希表实现，而Python的字典选用了哈希表，主要是考虑到哈希表在搜索方面的效率更高。因为Python虚拟机重度依赖字典，所以对字典在搜索、设置元素方面的性能，要求的更加苛刻。

但是由于哈希表的稀疏特性，导致其会有巨大的内存牺牲，而为了优化，Python别出心裁地将哈希表分成两部分来实现，分别是：**哈希索引数组**和**键值对数组**。

但是显然这当中还有很多细节我们没有说，比如：key到底是如何映射成索引的？索引冲突了怎么办？哈希攻击又是什么？以及删除元素也会面临一些问题，又是如何解决的？

我们将用三篇文章来攻破这些难题，深入理解哈希表，下面先来看看对象的哈希值。

哈希值

Python内置函数hash可以计算对象的哈希值，哈希表依赖于哈希值。而根据哈希表的性质，我们知道键对象必须满足以下两个条件，否则它无法容纳在哈希表中。

- 哈希值在对象的整个生命周期内不可以改变；
- 可比较，如果两个对象相等，那么它们的哈希值一定相同；

满足这两个条件的对象便是**可哈希(hashable)**对象，只有可哈希对象才可以作为哈希表的**键(key)**。因此像字典、集合等底层由哈希表实现的数据结构，其元素必须是可哈希对象。

Python内置的不可变对象都是可哈希对象，比如：整数、浮点数、字符串、只包含不可变对象的元组等等；而像可变对象，比如列表、字典等等便不可作为哈希表的键。

```
#键是可哈希的就行，值是否可哈希则没有要求
>>> {1: 1, "xxx": [1, 2, 3], 3.14: 333}
{1: 1, 'xxx': [1, 2, 3], 3.14: 333}
>>>
# 列表是可变对象，因此无法哈希
>>> {[]: 123}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
>>>
# 元组也是可哈希的
>>> {(1, 2, 3): 123}
{(1, 2, 3): 123}
 #但如果元组里面包含了不可哈希的对象
 #那么整体也会变成不可哈希对象
>>> {(1, 2, 3, []): 123}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

```
20 >>>
```

而我们自定义类的实例对象也是可哈希的，并且哈希值是通过对象的地址计算得到的。

```
1 class A:
2     pass
3
4
5 a1 = A()
6 a2 = A()
7 print(hash(a1), hash(a2))  # 141215868971 141215869022
```

当然Python也支持我们重写哈希函数，比如：

```
1  class A:
2
3      def __hash__(self):
4          return 123
5
6
7  a1 = A()
8  a2 = A()
9  print(hash(a1), hash(a2))  # 123 123
10
11 print({a1: 1, a2: 2})
12 # {<__main__.A object at 0x000002A2842282B0>: 1,
13 # <__main__.A object at 0x000002A2842285E0>: 2}
```

我们看到虽然哈希值一样，但是在作为字典的键的时候，如果发生了冲突，会改变规则重新映射，因为类的实例对象之间默认是不相等的。

注意：我们自定义类的实例对象默认都是可哈希的，但如果类里面重写了__eq__方法，且没有重写__hash__方法的话，那么这个类的实例对象就不可哈希了。

```
1  class A:
2
3      def __eq__(self, other):
4          return True
5
6
7  a1 = A()
8  a2 = A()
9  try:
10     print(hash(a1), hash(a2))
11 except Exception as e:
12     print(e)  # unhashable type: 'A'
```

为什么会有这种现象呢？首先我们说，在没有重写**__hash__方法**的时候，哈希值默认是根据对象的地址计算得到的。而且对象如果相等，那么哈希值一定是一样的。

但是我们重写了**__eq__**，相当于控制了**==操作符**的比较结果，两个对象是否相等就是由我们来控制了，可哈希值却还是根据地址计算得到的。因此两个对象地址不同，哈希值不同，但是对象却可以相等、又可以不相等，这就导致了矛盾。所以在重写了__eq__、但是没有重写__hash__的情况下，其实例对象便不可哈希了。

但如果重写了**__hash__**，那么哈希值计算方式就不再通过地址计算了，因此此时是可以哈希的。

```
1  class A:
2
3      def __eq__(self, other):
4          return True
```

```
5
6       def __hash__(self):
7           return 123
8
9
10  a1 = A()
11  a2 = A()
12  print({a1: 1, a2: 2})
13  # {<__main__.A object at 0x000001CEC8D682B0>: 2}
```

我们看到字典里面只有一个元素，因为重写了__hash__方法之后，计算得到哈希值都是一样的。如果没有重写__eq__，实例对象之间默认是不相等的。因此哈希值一样，但是对象不相等，那么会重新映射。但是我们重写了__eq__，返回的结果是True，所以Python认为对象是相等的，由于key的不重复性，保留了后面的键值对。

但需要注意的是，在比较相等时，会先比较地址是否一样，如果地址一样，那么哈希表会直接认为相等。

```
1   class A:
2
3       def __eq__(self, other):
4           return False
5
6       def __hash__(self):
7           return 123
8
9       def __repr__(self):
10          return "A instance"
11
12
13  a1 = A()
14  # 我们看到 a1 == a1 为 False
15  print(a1 == a1)  # False
16  # 但是只保留了一个key，原因是地址一样
17  # 在比较是否相等之前，会先判断地址是否一样
18  # 如果地址一样，那么认为是同一个key
19  print({a1: 1, a1: 2})  # {A instance: 2}
20
21  a2 = A()
22  # 此时会保留两个key
23  # 因为 a1 和 a2 地址不同，a1 == a2 也为False
24  # 所以哈希表认为这是两个不同的 key
25  # 但由于哈希值一样，那么映射出来的索引也一样
26  # 因此写入 a2:2 时相当于发生了索引冲突，于是会重新映射
27  # 但总之这两个key都会被保留
28  print({a1: 1, a2: 2})  # {A instance: 1, A instance: 2}
```

同样的，我们再来看一个Python字典的例子

```
1  d = {1: 123}
2
3  d[1.0] = 234
4  print(d)  # {1: 234}
5
6  d[True] = 345
7  print(d)  # {1: 345}
```

天哪噜，这是咋回事？首先整数在计算哈希值的时候，得到结果就是其本身；而浮点数显然不是，但如果浮点数的小数点后面是0，那么它和整数是等价的。

因此**3**和**3.0**的哈希值一样，并且两者也是相等的，因此它们被视为同一个key，所以相当于是更新。同理True也一样，因为bool继承自int，所以它等价于1，比如：9 + True

= 10。因此True和1相等，并且哈希值也相等，那么索引**d[True] = 345**同样相当于更新。

但是问题来了，值更新了我们可以理解，字典里面只有一个元素也可以理解，可为什么key一直是**1**呢？理论上最终结果应该是**True**才对啊。

其实这算是Python偷了个懒吧(开个玩笑)，因为key的哈希值是一样的，并且也相等，所以Python不会对key进行替换。

从字典在设置元素的时候我们也知道，如果对key映射成索引之后，发现哈希索引数组的该槽没有人用，那么就按照先来后到的顺序将**键值对**存储在键值对数组中，再把它在键值对数组中的索引存在哈希索引数组的指定**槽**中。

但如果发现**槽**有人用了，那么根据槽里面存的索引，去键值对数组中查找指定的entry，然后比较两个key是否相等。如果对应的key不相等，则重新映射找一个新的槽；如果相等，则说明是同一个key，那么把value换掉即可。

所以在替换元素的整个过程中，根本没有涉及到对键的修改，因此上面那个例子的最终结果，value会变、但键依旧是1，而不是True。

总之理想的哈希函数必须保证哈希值尽量均匀地分布于整个哈希空间中，越是相近的值，其哈希值差别应该越大。还是那句话，哈希函数对哈希表的好坏起着至关重要的作用。

收录于合集 #CPython 97

‹ 上一篇
《源码探秘 CPython》35. 索引冲突与哈希攻击

下一篇 ›
《源码探秘 CPython》33. 字典是怎么实现的?