

04 | 字典、集合，你真的了解吗？

2019-05-17 景霄

Python核心技术与实战

[进入课程 >](#)



讲述：冯永吉

时长 09:56 大小 9.10M



你好，我是景霄。

前面的课程，我们学习了 Python 中的列表和元组，了解了他们的基本操作和性能比较。这节课，我们再来学习两个同样很常见并且很有用的数据结构：字典（dict）和集合（set）。字典和集合在 Python 被广泛使用，并且性能进行了高度优化，其重要性不言而喻。

字典和集合基础


那究竟什么是字典，什么是集合呢？字典是一系列由键（key）和值（value）配对组成的元素的集合，在 Python3.7+，字典被确定为有序（注意：在 3.6 中，字典有序是一个

implementation detail，在 3.7 才正式成为语言特性，因此 3.6 中无法 100% 确保其有序性），而 3.6 之前是无序的，其长度大小可变，元素可以任意地删减和改变。

相比于列表和元组，字典的性能更优，特别是对于查找、添加和删除操作，字典都能在常数时间复杂度内完成。


而集合和字典基本相同，唯一的区别，就是集合没有键和值的配对，是一系列无序的、唯一的元素组合。

首先我们来看字典和集合的创建，通常有下面这几种方式：

 复制代码


```
1 d1 = {'name': 'jason', 'age': 20, 'gender': 'male'}
2 d2 = dict({'name': 'jason', 'age': 20, 'gender': 'male'})
3 d3 = dict([('name', 'jason'), ('age', 20), ('gender', 'male')])
4 d4 = dict(name='jason', age=20, gender='male')
5 d1 == d2 == d3 == d4
6 True
7
8 s1 = {1, 2, 3}
9 s2 = set([1, 2, 3])
10 s1 == s2
11 True
```

这里注意，Python 中字典和集合，无论是键还是值，都可以是混合类型。比如下面这个例子，我创建了一个元素为 1, 'hello', 5.0 的集合：

 复制代码

```
1 s = {1, 'hello', 5.0}
```


再来看元素访问的问题。字典访问可以直接索引键，如果不存在，就会抛出异常：

 复制代码

```
1 d = {'name': 'jason', 'age': 20}
2 d['name']
3 'jason'
```

```
4 d['location']
5 Traceback (most recent call last):
6   File "<stdin>", line 1, in <module>
7   KeyError: 'location'
```


也可以使用 `get(key, default)` 函数来进行索引。如果键不存在，调用 `get()` 函数可以返回一个默认值。比如下面这个示例，返回了 `'null'`。

 复制代码

```
1 d = {'name': 'jason', 'age': 20}
2 d.get('name')
3 'jason'
4 d.get('location', 'null')
5 'null'
```

说完了字典的访问，我们再来看集合。

首先我要强调的是，**集合并不支持索引操作，因为集合本质上是一个哈希表，和列表不一样**。所以，下面这样的操作是错误的，Python 会抛出异常：

 复制代码

```
1 s = {1, 2, 3}
2 s[0]
3 Traceback (most recent call last):
4   File "<stdin>", line 1, in <module>
5   TypeError: 'set' object does not support indexing
```


想要判断一个元素在不在字典或集合内，我们可以用 `value in dict/set` 来判断。

 复制代码

```
1 s = {1, 2, 3}
2 1 in s
3 True
4 10 in s
5 False
6
```

```
7 d = {'name': 'jason', 'age': 20}
8 'name' in d
9 True
10 'location' in d
11 False
```

当然，除了创建和访问，字典和集合也同样支持增加、删除、更新等操作。

 复制代码

```
1 d = {'name': 'jason', 'age': 20}
2 d['gender'] = 'male' # 增加元素对'gender': 'male'
3 d['dob'] = '1999-02-01' # 增加元素对'dob': '1999-02-01'
4 d
5 {'name': 'jason', 'age': 20, 'gender': 'male', 'dob': '1999-02-01'}
6 d['dob'] = '1998-01-01' # 更新键'dob'对应的值
7 d.pop('dob') # 删除键为'dob'的元素对
8 '1998-01-01'
9 d
10 {'name': 'jason', 'age': 20, 'gender': 'male'}
11
12 s = {1, 2, 3}
13 s.add(4) # 增加元素 4 到集合
14 s
15 {1, 2, 3, 4}
16 s.remove(4) # 从集合中删除元素 4
17 s
18 {1, 2, 3}
```

不过要注意，集合的 `pop()` 操作是删除集合中最后一个元素，可是集合本身是无序的，你无法知道会删除哪个元素，因此这个操作得谨慎使用。

实际应用中，很多情况下，我们需要对字典或集合进行排序，比如，取出值最大的 50 对。

对于字典，我们通常会根据键或值，进行升序或降序排序：


 复制代码

```
1 d = {'b': 1, 'a': 2, 'c': 10}
2 d_sorted_by_key = sorted(d.items(), key=lambda x: x[0]) # 根据字典键的升序排序
3 d_sorted_by_value = sorted(d.items(), key=lambda x: x[1]) # 根据字典值的升序排序
4 d_sorted_by_key
```

```
5 [('a', 2), ('b', 1), ('c', 10)]
6 d_sorted_by_value
7 [('b', 1), ('a', 2), ('c', 10)]
```

这里返回了一个列表。列表中的每个元素，是由原字典的键和值组成的元组。

而对于集合，其排序和前面讲过的列表、元组很类似，直接调用 `sorted(set)` 即可，结果会返回一个排好序的列表。

 复制代码

```
1 s = {3, 4, 2, 1}
2 sorted(s) # 对集合的元素进行升序排序
3 [1, 2, 3, 4]
```

字典和集合性能

文章开头我就说到了，字典和集合是进行过性能高度优化的数据结构，特别是对于查找、添加和删除操作。那接下来，我们就来看看，它们在具体场景下的性能表现，以及与列表等其他数据结构的对比。

比如电商企业的后台，存储了每件产品的 ID、名称和价格。现在的需求是，给定某件商品的 ID，我们要找出其价格。

如果我们用列表来存储这些数据结构，并进行查找，相应的代码如下：

 复制代码


```
1 def find_product_price(products, product_id):
2     for id, price in products:
3         if id == product_id:
4             return price
5     return None
6
7 products = [
8     (143121312, 100),
9     (432314553, 30),
10    (32421912367, 150)
11 ]
12
```

```
13 print('The price of product 432314553 is {}'.format(find_product_price(products, 432314553)))
14
15 # 输出
16 The price of product 432314553 is 30
```



假设列表有 n 个元素，而查找的过程要遍历列表，那么时间复杂度就为 $O(n)$ 。即使我们先对列表进行排序，然后使用二分查找，也会需要 $O(\log n)$ 的时间复杂度，更何况，列表的排序还需要 $O(n \log n)$ 的时间。

但如果我们用字典来存储这些数据，那么查找就会非常便捷高效，只需 $O(1)$ 的时间复杂度就可以完成。原因也很简单，刚刚提到过的，字典的内部组成是一张哈希表，你可以直接通过键的哈希值，找到其对应的值。


 复制代码

```
1 products = {
2     143121312: 100,
3     432314553: 30,
4     32421912367: 150
5 }
6 print('The price of product 432314553 is {}'.format(products[432314553]))
7
8 # 输出
9 The price of product 432314553 is 30
```



类似的，现在需求变成，要找出这些商品有多少种不同的价格。我们还用同样的方法来进行比较一下。

如果还是选择使用列表，对应的代码如下，其中，A 和 B 是两层循环。同样假设原始列表有 n 个元素，那么，在最差情况下，需要 $O(n^2)$ 的时间复杂度。

 复制代码


```
1 # list version
2 def find_unique_price_using_list(products):
3     unique_price_list = []
4     for _, price in products: # A
5         if price not in unique_price_list: #B
6             unique_price_list.append(price)
7     return len(unique_price_list)
```

```

8
9 products = [
10     (143121312, 100),
11     (432314553, 30),
12     (32421912367, 150),
13     (937153201, 30)
14 ]
15 print('number of unique price is: {}'.format(find_unique_price_using_list(products)))
16
17 # 输出
18 number of unique price is: 3

```

但如果我们选择使用集合这个数据结构，由于集合是高度优化的哈希表，里面元素不能重复，并且其添加和查找操作只需 $O(1)$ 的复杂度，那么，总的时间复杂度就只有 $O(n)$ 。

 复制代码


```

1 # set version
2 def find_unique_price_using_set(products):
3     unique_price_set = set()
4     for _, price in products:
5         unique_price_set.add(price)
6     return len(unique_price_set)
7
8 products = [
9     (143121312, 100),
10    (432314553, 30),
11    (32421912367, 150),
12    (937153201, 30)
13 ]
14 print('number of unique price is: {}'.format(find_unique_price_using_set(products)))
15
16 # 输出
17 number of unique price is: 3

```

可能你对这些时间复杂度没有直观的认识，我可以举一个实际工作场景中的例子，让你来感受一下。

下面的代码，初始化了含有 100,000 个元素的产品，并分别计算了使用列表和集合来统计产品价格数量的运行时间：

 复制代码

```

1 import time
2 id = [x for x in range(0, 100000)]
3 price = [x for x in range(200000, 300000)]
4 products = list(zip(id, price))
5
6 # 计算列表版本的时间
7 start_using_list = time.perf_counter()
8 find_unique_price_using_list(products)
9 end_using_list = time.perf_counter()
10 print("time elapse using list: {}".format(end_using_list - start_using_list))
11 ## 输出
12 time elapse using list: 41.61519479751587
13
14 # 计算集合版本的时间
15 start_using_set = time.perf_counter()
16 find_unique_price_using_set(products)
17 end_using_set = time.perf_counter()
18 print("time elapse using set: {}".format(end_using_set - start_using_set))
19 # 输出
20 time elapse using set: 0.008238077163696289

```

你可以看到，仅仅十万的数据量，两者的速度差异就如此之大。事实上，大型企业的后台数据往往有上亿乃至十亿数量级，如果使用了不合适的数据结构，就很容易造成服务器的崩溃，不但影响用户体验，并且会给公司带来巨大的财产损失。

字典和集合的工作原理

我们通过举例以及与列表的对比，看到了字典和集合操作的高效性。不过，字典和集合为什么能够如此高效，特别是查找、插入和删除操作？

这当然和字典、集合内部的数据结构密不可分。不同于其他数据结构，字典和集合的内部结构都是一张哈希表。

对于字典而言，这张表存储了哈希值（hash）、键和值这 3 个元素。

而对集合来说，区别就是哈希表内没有键和值的配对，只有单一的元素了。

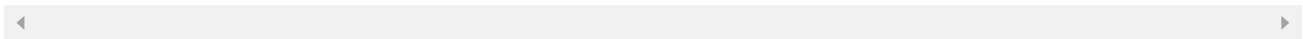
我们来看，老版本 Python 的哈希表结构如下所示：

 复制代码



```

2 | 哈希值 (hash)  键 (key)  值 (value)
3 --+-----+
4 0 |      hash0      key0      value0
5 --+-----+
6 1 |      hash1      key1      value1
7 --+-----+
8 2 |      hash2      key2      value2
9 --+-----+
10 . |              ...
11 _+-----+
12

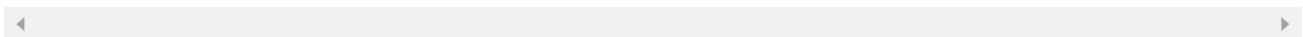
```




不难想象，随着哈希表的扩张，它会变得越来越稀疏。举个例子，比如我有这样一个字典：

 复制代码

```
1 {'name': 'mike', 'dob': '1999-01-01', 'gender': 'male'}
```



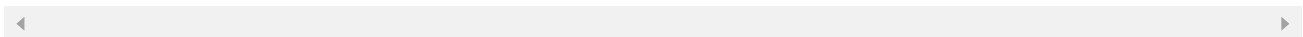
那么它会存储为类似下面的形式：

 复制代码


```

1 entries = [
2  ['--', '--', '--']
3  [-230273521, 'dob', '1999-01-01'],
4  ['--', '--', '--'],
5  ['--', '--', '--'],
6  [1231236123, 'name', 'mike'],
7  ['--', '--', '--'],
8  [9371539127, 'gender', 'male']
9  ]

```



这样的设计结构显然非常浪费存储空间。为了提高存储空间的利用率，现在的哈希表除了字典本身的结构，会把索引和哈希值、键、值单独分开，也就是下面这样新的结构：

 复制代码


```

1 Indices
2 -----
3 None | index | None | None | index | None | index ...
4 -----

```

```
5
6 Entries
7 -----
8 hash0    key0    value0
9 -----
10 hash1    key1    value1
11 -----
12 hash2    key2    value2
13 -----
14          ...
15 -----
```

那么，刚刚的这个例子，在新的哈希表结构下的存储形式，就会变成下面这样：

 复制代码

```
1 indices = [None, 1, None, None, 0, None, 2]
2 entries = [
3 [1231236123, 'name', 'mike'],
4 [-230273521, 'dob', '1999-01-01'],
5 [9371539127, 'gender', 'male']
6 ]
```

我们可以很清晰地看到，空间利用率得到很大的提高。

清楚了具体的设计结构，我们接着来看这几个操作的工作原理。

插入操作

每次向字典或集合插入一个元素时，Python 会首先计算键的哈希值（`hash(key)`），再和 `mask = PyDicMinSize - 1` 做与操作，计算这个元素应该插入哈希表的位置 `index = hash(key) & mask`。如果哈希表中此位置是空的，那么这个元素就会被插入其中。

而如果此位置已被占用，Python 便会比较两个元素的哈希值和键是否相等。

若两者都相等，则表明这个元素已经存在，如果值不同，则更新值。

若两者中有一个不相等，这种情况我们通常称为哈希冲突（hash collision），意思是两个元素的键不相等，但是哈希值相等。这种情况下，Python 便会继续寻找表中空余的位

置，直到找到位置为止。

值得一提的是，通常来说，遇到这种情况，最简单的方式是线性寻找，即从这个位置开始，挨个往后寻找空位。当然，Python 内部对此进行了优化（这一点无需深入了解，你有兴趣可以查看源码，我就不再赘述），让这个步骤更加高效。

查找操作

和前面的插入操作类似，Python 会根据哈希值，找到其应该处于的位置；然后，比较哈希表这个位置中元素的哈希值和键，与需要查找的元素是否相等。如果相等，则直接返回；如果不等，则继续查找，直到找到空位或者抛出异常为止。

删除操作

对于删除操作，Python 会暂时对这个位置的元素，赋予一个特殊的值，等到重新调整哈希表的大小时，再将其删除。

不难理解，哈希冲突的发生，往往会降低字典和集合操作的速度。因此，为了保证其高效性，字典和集合内的哈希表，通常会保证其至少留有 $1/3$ 的剩余空间。随着元素的不停插入，当剩余空间小于 $1/3$ 时，Python 会重新获取更大的内存空间，扩充哈希表。不过，这种情况下，表内所有的元素位置都会被重新排放。

虽然哈希冲突和哈希表大小的调整，都会导致速度减缓，但是这种情况发生的次数极少。所以，平均情况下，这仍能保证插入、查找和删除的时间复杂度为 $O(1)$ 。

总结

这节课，我们一起学习了字典和集合的基本操作，并对它们的高性能和内部存储结构进行了讲解。

字典在 Python3.7+ 是有序的数据结构，而集合是无序的，其内部的哈希表存储结构，保证了其查找、插入、删除操作的高效性。所以，字典和集合通常运用在对元素的高效查找、去重等场景。

思考题

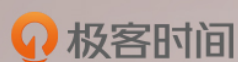
1. 下面初始化字典的方式，哪一种更高效？

```
1 # Option A
2 d = {'name': 'jason', 'age': 20, 'gender': 'male'}
3
4 # Option B
5 d = dict({'name': 'jason', 'age': 20, 'gender': 'male'})
```

2. 字典的键可以是一个列表吗？下面这段代码中，字典的初始化是否正确呢？如果不正确，可以说出你的原因吗？

```
1 d = {'name': 'jason', ['education']: ['Tsinghua University', 'Stanford University']}
```

欢迎留言和我分享，也欢迎你把这篇文章分享给你的同事、朋友。



Python 核心技术与实战

系统提升你的 Python 能力

景霄

Facebook 资深工程师



新版升级：点击「👉 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 03 | 列表和元组，到底用哪一个？

下一篇 04 | 字典和集合，到底用哪一个？

精选留言 (79)

写留言



pyhhou

2019-05-17

91

思考题 1:

第一种方法更快，原因感觉上是和之前一样，就是不需要去调用相关的函数，而且像老师说的那样 {} 应该是关键字，内部会去直接调用底层C写好的代码

思考题 2:...

展开

作者回复: 正解



燕儿衔泥

2019-05-17

21

- 1.直接 {} 的方式，更高效。可以使用dis分析其字节码
- 2.字典的键值，需要不可变，而列表是动态的，可变的。可以改为元组

作者回复: 使用dis分析其字节码很赞



随风の

2019-05-17

16

文中提到的新的哈希表结构有点不太明白 None 1 None None 0 None 2 是什么意思？index是索引的话 为什么中间会出现两个None

作者回复: 这只是一种表示。None表示indices这个array上对应的位置没有元素，index表示有元素，并且对应entries这个array index位置上的元素。你看那个具体的例子就能看懂了

Python 3.7 以后插入有序变为字典的特性。构造新字典的方式：

1. double star

```
>>> d1 = {'name': 'jason', 'age': 20, 'gender': 'male'}
```

```
>>> d2 = {'hobby': 'swim', **d1}
```

2. update 函数： ...

展开 ∨



charily

2019-05-18

6

老师，你好！有几个让我困惑的地方想跟您确认一下，问题有点多，希望不吝赐教！

1. 为了提高哈希表的空间利用率，于是使用了Indices、Entries结构分开存储（index）和（hashcode、key、value），这里的index是否就是Entries列表的下标？

2、如果问题1成立，通过hash(key) & (PyDicMinSize - 1)计算出来的是否为Indices列表的下标？ ...

展开 ∨



许山山

2019-05-17

6

```
>>> dis.dis(lambda : dict())
```

```
1 0 LOAD_GLOBAL 0 (dict)
```

```
3 CALL_FUNCTION 0 (0 positional, 0 keyword pair)
```

```
6 RETURN_VALUE
```

...

展开 ∨



许山山

2019-05-17

4

老师我明白了，(hash, key, val) 都是存在 entries 里面的，通过 indices[index] 找到 entry 再做比较就好了。



farFlight

2019-05-17

4

老师好，在王争老师的数据结构课程中提到哈希表常与链表一起使用，譬如用来解决哈希冲突。请问python底层对字典和集合的实现是否也是这样的呢？

作者回复: 这个就是文中所说的线性寻找了, 但是Python底层解决哈希冲突还有更好的方法, 线性寻找是最简单的, 但是不是最高效的



小狼

2019-05-17

👍 3

```
s2 = Set([1, 2, 3])
# Set 大写会报错:
NameError: name 'Set' is not defined
改成小写问题解决
```



鱼腐

2019-05-17

👍 3

Indices:none | one | none | index | none | index 是什么意思? 能补充讲解下吗

作者回复: 这只是一种表示。None表示indices这个array上对应的位置没有元素, index表示有元素, 并且对应entries这个array index位置上的元素。你看那个具体的例子就能看懂了



Hoo-Ah

2019-05-17

👍 3

1. 直接使用大括号更高效, 避免了使用类生成实例其他不必要的操作;
2. 列表不可以作为key, 因为列表是可变类型, 可变类型不可hash。

问题: 为什么在旧哈希表中元素会越来越稀?

展开 ▾

作者回复: 你比较一下旧哈希表和新哈希表的存储结构就会发现, 旧哈希表的空间利用率很低, 一个位置要同时分配哈希值, 键和值的空间, 但是新哈希表把indices和entries分开后, 空间利用率大大提高。

看文中的例子, 这是旧哈希表存储示意图

```
entries = [
    ['--', '--', '--']
    [-230273521, 'dob', '1999-01-01'],
    ['--', '--', '--'],
```

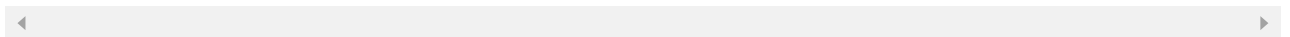
```
['--', '--', '--'],  
[1231236123, 'name', 'mike'],  
['--', '--', '--'],  
[9371539127, 'gender', 'male']  
]
```

VS

新哈希表存储示意图：

```
indices = [None, 1, None, None, 0, None, 2]  
entries = [  
    [1231236123, 'name', 'mike'],  
    [-230273521, 'dob', '1999-01-01'],  
    [9371539127, 'gender', 'male']  
]
```

你数一下两个版本中空着的元素的个数，就很清晰了。



Danpier

2019-05-18

👍 2

老师，对于集合插值有个疑问：

```
s={1, 2.0}
```

```
s.add(1.0)
```

```
s
```

```
# 输出...
```

展开 ▾



趁早

2019-05-18

👍 2

最后的例子很有代表性，举例很好

展开 ▾



刘朋

2019-05-17

👍 2

插入操作,

```
mask = PyDicMinSize - 1
```


$\text{index} = \text{hash}(\text{key}) \& \text{mask}$

能否有个例子,想详细了解一下细节

展开 ▾



山石尹口

2019-06-02

👍 1

list做key的问题有点疑惑,即使list是可变的,它在内存中的地址是不变的,对地址是可以hash的吧?不然所有引用类型都没法做key了。当然,用引用类型做key不一定是好的做法,可能会带来混淆。

展开 ▾



美美

2019-05-26

👍 1

1.文中说字典是无序的不对吧? Python3.6 后字典是有序的
2.还有示例代码用的是 format 方法,而 Python3.6 后的 f-string比原来的format方法易读多了

展开 ▾



William

2019-05-24

👍 1

老师请问, key、hash值、indice三者的联系是啥? 一直以为hash(key)就是内存地址



taoist

2019-05-18

👍 1

思考题:1

Option A: `python3 -m timeit -n 1000000 "d = {'name': 'jason', 'age': 20, 'gender': 'male'}"`

1000000 loops, best of 5: 76.2 nsec per loop

Option B: `python3 -m timeit -n 1000000 "d = dict({'name': 'jason', 'age': 20,...`

展开 ▾



许山山

2019-05-17

👍 1

每次向字典或集合插入一个元素时，Python 会首先计算键的哈希值 (`hash(key)`)，再和 `mask = PyDicMinSize - 1` 做与操作，计算这个元素应该插入哈希表的位置 `index = hash(key) & mask`。如果哈希表中此位置是空的，那么这个元素就会被插入其中。

而如果此位置已被占用，Python 便会比较两个元素的哈希值和键是否相等。...

展开 ∨



天凉好个秋

2019-05-17

👍 1

不难想象，随着哈希表的扩张，它会变得越来越稀疏。
后面例子中解释的原因没看懂，能详细说说吗？

作者回复: 哈希表为了保证其操作的有效性（查找，添加，删除等等），都会overallocate（保留至少1/3的剩余空间），但是很多空间其实都没有被利用，因此很稀疏

