2019 级

《物联网数据存储与管理》课程

报 告

姓　名　沈佳怡

学　号　U201915066

班　号　CS1904 班

日　期　2022.04.15

目　　录

一、 理论分析.....1
1.1 概况介绍.....错误！未定义书签。
1.2 冲突处理.....1
1.3 Cuckoo Hash 模型.....2
1.4 与 Bloom filter 对比.....5
1.5 Cuckoo Hash 问题.....5
1.6 Cuckoo Hash 优化.....5
二、代码实现.....6
2.1 数据结构说明.....6
三、测试.....10
四、总结.....11
参考文献.....13

一、理论分析

1.1 概况介绍

哈希表是一种常用的数据结构，例如 java 的 HashMap，python 的 dict 等。但哈希表结构也容易造成冲突问题。常见的哈希表（Hash Table 或者字典，dictionary）会通过链表、线性探测等方式来处理冲突。Cuckoo Hash 是一种解决 hash 冲突的方法，其目的是使用简单的 hash 函数来提高 hash table 的利用率，同时保证 O(1) 的查询时间。基本思想是使用 2 个 hash 函数来处理碰撞，从而每个 key 都对应到 2 个位置。

对于海量数据处理业务，我们通常需要一个索引数据结构，用来帮助查询，快速判断数据记录是否存在，这种数据结构通常又叫过滤器(filter)。考虑这样一个场景，上网的时候需要在浏览器上输入 URL，这时浏览器需要去判断这是否一个恶意的网站，它将对本地缓存的成千上万的 URL 索引进行过滤，如果不存在，就放行，如果（可能）存在，则向远程服务端发起验证请求，并回馈客户端给出警告。

索引的存储又分为有序和无序，前者使用关联式容器，比如 B 树，后者使用哈希算法。这两类算法各有优劣：比如，关联式容器时间复杂度稳定 O(logN)，且支持范围查询；又比如哈希算法的查询、增删都比较快 O(1)，但这是在理想状态下的情形，遇到碰撞严重的情况，哈希算法的时间复杂度会退化到 O(n)。因此，选择一个好的哈希算法是很重要的。

1.2 冲突处理

哈希表（HashTable）又叫做散列表，是根据关键码值（即键值对）而直接访问的数据结构。也就是说，它通过把关键码映射到表中一个位置来访问记录，以加快查找速度。这个映射函数就叫做散列（哈希）函数，存放记录的数组叫做散列表。

冲突，也叫做碰撞。对于两个数据元素的关键字 Ki 和 Kj(i!=j)，但是存在：Hash(Ki)==Hash(Kj),即：不同的关键字通过哈希函数计算出相同的哈希地址，这种现象就成为哈希冲突或哈希碰撞。冲突处理一般有两种方法：开放寻址（open addressing）和链表法（separate chaining）。

开放寻址在解决当前冲突的情况下同时可能会导致新的冲突，而链表法不会有这种问题。同时链表相比于开放寻址局部性较差，在程序运行过程中可能引起操作系统的缺页中断，从而导致系统颠簸。

1. 开放寻址

开放寻址的意思是当插入 key，value 发生冲突时，从当前位置向后按某种策略遍历哈希表。在开放寻址法中，所有元素都存在哈希表里。也就是说，每个表项或包含动态集合的一个元素，或包含空。当查找某个元素时，要系统地检查所有的表项，直到找到所需的元素或者最终查明该元素不在表中。在开放寻址法中，哈希表可能被填满，以至于不能插入任何新的元素，该方法导致的一个结果便是装载因子绝对不会超过 1。

开放寻址法的好处是其不使用指针，直接使用哈希函数来算出存取的位置，这样也可以加快存取速度。但开放寻址法会导致之后的 key 被占用。在开放寻址中有三种最常用的技术，其对应的哈希函数形式分别为：

1. 线性探测法

$$h(k,i) = (h'(k) + i) \ mod \ m, \ i = 0, 1, \cdots, m-1$$

2. 二次探测法

$$h(k,i) = (h'(k) + c_1 i + c_2 i^2) \ mod \ m, \ i = 0, 1, \cdots, m-1$$

3. 双重哈希法

$$h(k,i) = (h_1(k) + i h_2(k)) \ mod \ m, \ i = 0, 1, \cdots, m-1$$

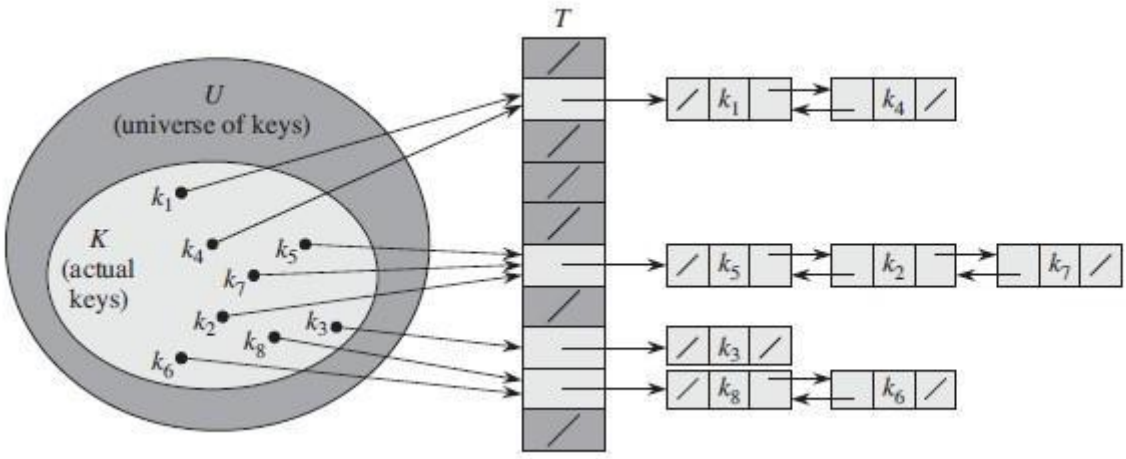

图 1.1 开放寻址法

2. 链表法

链表法的意思是哈希表中的每个元素都是一个类似链表或者其他数据结构的 head。当出现冲突时，在链表后面添加元素。这也就意味着，如果某一个位置冲突过多的话，插入的时间复杂段将退化为 O(N)。

1.3 Cuckoo Hash 模型

CuckooHash(布谷鸟散列),最早于 2001 年由 Rasmus Pagh 和 Flemming Friche Rodler 提出,是为了解决哈希冲突问题,利用较少的计算换取较大的空间。Cuckoo 是布谷鸟的意思。和普通的哈希表不同,在 Cuckoo Hash 表中,每个元素在 bucket list 中有两个位置可以存放。如果两个位置都被其他元素占了，则随机选择一个位置将其踢走。被踢走的元素被移动到它的备用位置，如果也被其他元素占了则也将其踢走，如此反复。

分别使用 hash1、hash2 计算对应的 key 位置：

1. 两个位置均为空，则任选一个插入；
2. 两个位置中一个为空，则插入到空的那个位置
3. 两个位置均不为空，则随机踢出一个位置上的 keyx，被踢出的 keyx 再执行该算法找其另一个位置，循环直到插入成功。
4. 如果被踢出的次数达到一定的阈值，则认为 hash 表已满，并进行重新哈希 rehash

下图 1.2 是 Cuckoo Hash 的示例。

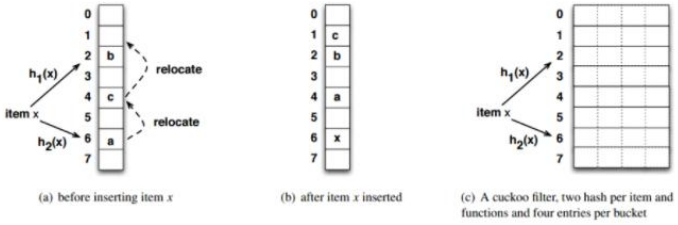

Figure 1: Illustration of cuckoo hashing

图 1.2 cuckoo hash

图中(a)表示插入元素 x 之前已经存在的元素 a，b，c。通过计算 x 的两个可用位置分别被 a，b 占用了，随机选择一个位置，也就是元素 a 存放的位置。然后元素 a 被迁移到它的备用位置，也就是元素 c 现在的位置。类似的 c 再做迁移。(c) 表示 Cuckoo Hash 的一种实现，每个 bucket 有四个 slot。对于 (a) 的情形，则直接把 x 插入到元素 a 或 b 所在的 bucket 的空的 slot 里面即可。

其基本思想为：

1. 使用两个哈希函数 h1（x）、 h2（x）和两个哈希桶 T1、T2 。

2. 插入元素 x：

如果 T1［h1（x)］、T2［h2（x)］有一个为空，则插入；两者都空，随便选一个插入。

如果 T1［h1（x)］、T2［h2（x)］都满，则随便选择其中一个（设为 y ），将其踢出，插入 x。

重复上述过程，插入元素 y。

如果插入时，踢出次数过多，则说明哈希桶满了。则进行扩容、ReHash 后，再次插入。

3. 查询元素 x：

读取 T1［h1（x)］、T2［h2（x)］和 x 比对即可

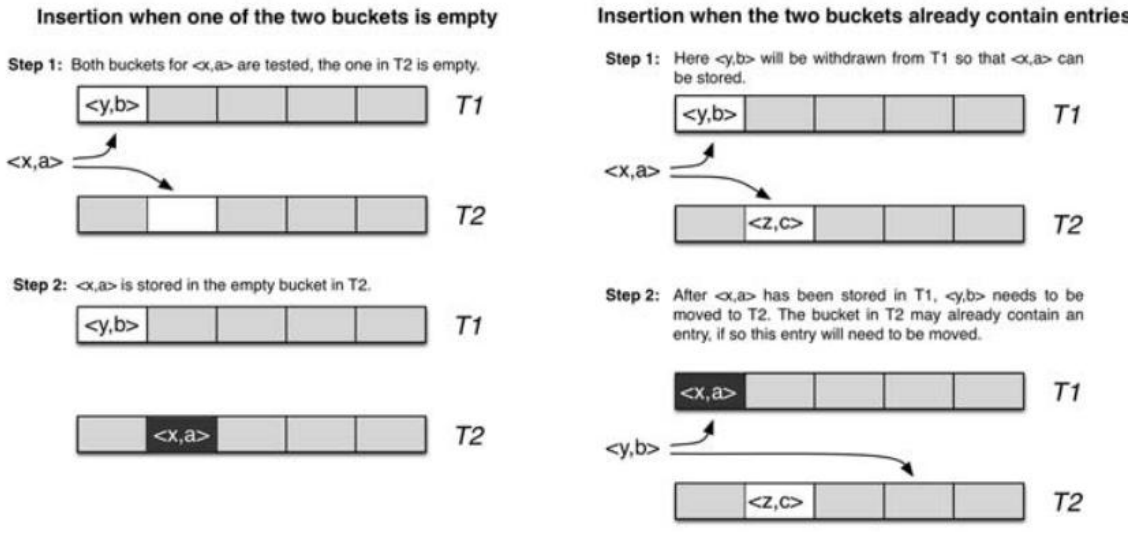

图 1.3 cuckoo hash 操作

1. Cockoo hash 有两种变形。一种通过增加哈希函数进一步提高空间利用率；另一种是增加哈希表，每个哈希函数对应一个哈希表，每次选择多个张表中空余位置进行放置。三个哈希表可以达到 80% 的空间利用率。

2. Cockoo hash 的过程可能因为反复踢出无限循环下去，这时候就需要进行一次循环踢出的限制，超过限制则认为需要添加新的哈希函数。

3. Cockoo hash 的应用，Cuckoo Filter。

1.4 与 Bloom filter 对比

标准 Bloom filters 的一个限制是不能删除现有项而不重新生成整个筛选器(或可能会引入一般不太理想的 false negatives)。

现在有几种方法可以扩展标准的 Bloom 过滤器来支持删除，但有很大的空间或性能开销。如 Counting Bloom filters 已被建议为多个应用，但它们通常使用 3-4×空间用于保证和 space-optimized Bloom filter 相同的 false negatives。其他变体包括 d-left counting Bloom filters，仍要使用 1.5 倍的空间，此外 quotient filters 可以在和 Bloom 过滤器相当的空间开销提供显著降低的查找性能。

与标准 Bloom 过滤器相当的空间或性能，现在提出了一种实用的数据结构——cuckoo filter，包含以下四个主要优点。

1. 它支持动态添加和删除项;

2. 即使接近装满它也能提供比传统的 bloom fliter 更高的查找性能(例如，95%的空间利用);

3.它比其他方法更容易实现商过滤器;

4. 在许多实际应用中，如果目标误报率小于 3%，它比 Bloom 过滤器占用更少的空间应用程序。

filter type	space cost	cache misses per lookup	deletion support
Bloom	1	k	no
blocked Bloom	1x	1	no
counting Bloom	3x ~ 4x	k	yes
d-left counting Bloom	1.5x ~ 2x	d	yes
quotient	1x ~ 1.2x	≥ 1	yes
cuckoo	$\leq$1x	2	yes

图 1.4 假设标准和计数 bloom 使用 K 哈希函数，d-left 计数 bloom 有 d 个分区时对比图

	bits per item	load factor α	avg. # memory references / lookup positive query	avg. # memory references / lookup negative query
space-optimized Bloom filter	$1.44\log_2(1/\epsilon)$	–	$\log_2(1/\epsilon)$	2
cuckoo filter	$(\log_2(1/\epsilon)+3)/\alpha$	95.5%	2	2
cuckoo filter w/ semi-sort	$(\log_2(1/\epsilon)+2)/\alpha$	95.5%	2	2

ϵ	target false positive rate
f	fingerprint length in bits
α	load factor ($0 \leq \alpha \leq 1$)
b	number of entries per bucket
m	number of buckets
n	number of items
C	average bits per item

图 1.5 Bloomfilter 和三个 cuckoo filters 的空间和查找成本

1.5 Cuckoo Hash 问题

Cuckoo Hash 主要有以下两个问题：

1. 写入性能差.数据项写入 Cuckoo 哈希表时,如果发生哈希冲突,则将候选桶中任意一个数据项迁移到该数据项的其他候选桶中,并且可能迁移另一个数据

项,直到找到一个空槽或者达到最大搜索阈值。本质上,深度优先搜索的随机替换策略导致 Cuckoo 哈希的 Cuckoo 路径很长,频繁的“踢出”操作导致其在多个桶之间进行密集的数据迁移,使得哈希表写入性能低下,同时细粒度锁的大量使用存在死锁和活锁的风险.基于随机替换策略的 Cuckoo 哈希表的桶之间的负载不均衡,导致在高负载率下易出现高延迟插入和无限循环.

2. 正向查询性能差.Cuckoo 哈希表通常使用 D(D＞1)个散列函数,查询时最少只需要访问 1 个桶就能得到查询结果,最多时也只需要访问 D 个桶.在读负载远大于写负载的情况下,哈希表的吞吐量取决于读负载的吞吐量.

1.6 Cuckoo Hash 优化

常用手段有使用多路哈希桶、使用多个 Hash 函数和使用广度优先搜索缩短深度优先搜索路径长度。

大多数 Cuckoo 哈希表使用随机替换策略搜索空槽,如果数据项 x 的候选桶已满,则随机选择桶中的一个数据项 y 并将其迁移到候选桶中,这可能导致循环迁移数据项,直到找到一个空槽或者达到最大搜索阈值.这个过程中所有相关的桶都是 Cuckoo 路径的一部分.随着哈希表负载率的增加,Cuckoo 路径的长度也会增加,大量数据项的迁移导致哈希表的性能低下.Cuckoo 哈希表可看成一个无向图,每个桶为一个顶点,每个数据项为一条边,用以连接两个候选桶(顶点).用广度优先方法搜索空槽,桶中的每个数据项都被视为可能的路径,并以相同的方式将路径拓 展到其他候选桶.广度优先搜索能够以对数的形式缩短深度优先搜索 Cuckoo。

二、代码实现

cuckoo hash 采用 python 语言实现，下面将对代码层面进行说明。

2.1 数据结构说明

Cuckoo hash 的 python 类定义如下：

```
# -*- coding: utf-8 -*-
"""This module contains the CuckooHash class."""

from numbers import Number
from random import randint
class CuckooHash(object):
    """Custom hash map which handles collisions using Cuckoo Hashing."""
    def __init__(self, size):
        """Initialize hash table of given size."""
        self._assert_valid_size(size)
        self.size = size
        self.nitems = 0
        self.array = [(None, None)] * int(size)
        self._num_hashes = 8
        self._max_path_size = size
        self._random_nums = self._get_new_random_nums()
    @staticmethod
    def _assert_valid_size(size):
        """Raise appropriate error if input is not a natural number."""
        if not isinstance(size, Number):
            raise TypeError("Size must be a valid integer")
        elif size < 0 or int(size) != size:
            raise ValueError("Size must be a natural number")
    def set(self, key, value):
        """Add key value pair and increment number of items in dictionary."""
        return self._set(key, value, increment_nitems=True)
    def _set(self, key, value, increment_nitems):
        """Add key value pair and return True on success, False on failure."""
        self._assert_valid_key(key)
        self._assert_valid_value(value)
        if self._is_full():
            return False
        set_result = self._set_helper(key, value, 0)
        if set_result is not True:
            # the set did not succeed, so we rehash the table with new hashes
            try:
                self._rehash(*set_result)
            except RuntimeError: # recursive stack limit exceeded
                return False
```

```
            self.nitems += int(increment_nitems)
            return True
        @staticmethod
        def _assert_valid_key(key):
            """Raise TypeError if the input is not a valid key."""
            if not isinstance(key, str):
                raise TypeError("Keys must be strings.")
        @staticmethod
        def _assert_valid_value(value):
            """Raise TypeError if value is None."""
            if value is None:
                raise TypeError("dictionary values can't be None.")
        def _is_full(self):
            """Return True if the hash map is full, False otherwise."""
            return self.nitems == self.size
        def _set_helper(self, key, value, num_iters):
            """
            Recursively try to add key value pair to dictionary in under
            _max_path_size number of steps, and return True on success and the
            unset key and value on failure.
            """
            if num_iters > self._max_path_size:
                return key, value
            else:
                array_indices = self._get_hashes(key)
                if self._add_to_free_slot(key, value, array_indices) == "success":
                    return True
                else:
                    # set the array at the first hash of the key to key value pair
                    # and recursively re set the pair that was bumped out
                    (slot_key, slot_val), self.array[array_indices[0]] = \
                                            self.array[array_indices[0]], (key, value)
                    return self._set_helper(slot_key, slot_val, num_iters + 1)
        def _add_to_free_slot(self, key, value, array_indices):
            """
            Attempt to add key value pair and return "success" on sucess and
            "failure" if all hashes hashed to an occupied slot."""
            for i in array_indices:
                slot_key, _ = self.array[i]
                # Check if slot is empty or if given key has already been set
                # to allow overwriting of keys
                if slot_key is None or slot_key == key:
                    self.array[i] = key, value
                    return "success"
```

```
        return "failure"
    def _rehash(self, unset_key, unset_value):
        """
        Take a key value pair not yet in the dictionary and generate new
        hash functions until all key value pairs including the input pair can be
        added to the table without collisions.
        """
        # generate new hash functions
        self._random_nums = self._get_new_random_nums()
        self._set(unset_key, unset_value, increment_nitems=False)
        for index, (key, value) in enumerate(self.array):
            if key is not None and index not in self._get_hashes(key):
                self.array[index] = (None, None)
                self._set(key, value, increment_nitems=False)
    def _get_new_random_nums(self):
        """Generate new random numbers to be used in the hash functions."""
        return [randint(-1000, 1000) for _ in range(self._num_hashes)]
    def get(self, key):
        """
        Return value associated with input key if it has been set, None
        otherwise.
        """
        array_index = self._find_array_index(key)
        return None if array_index == "not found" else self.array[array_index][1]
    def delete(self, key):
        """
        Delete key from hash map and return the associated value on success,
        and None on failure.
        """
        array_index = self._find_array_index(key)
        if array_index == "not found":
            return None
        else:
            self.nitems -= 1
            (_, val), self.array[array_index] = \
                                        self.array[array_index], (None, None)
            return val
    def _find_array_index(self, key):
        """Return the index of key in the array or "not found" if not found."""
        indices = self._get_hashes(key)
        matches = [i for i in indices if self.array[i][0] == key]
        return matches[0] if matches else "not found"
```

```
    def _get_hashes(self, string):
```

```
        """Return a list of each hash applied to the input string."""
        return [self._get_hash(string + str(i)) for i in self._random_nums]
    def _get_hash(self, string):
        """Return the hash of the input string."""
        return hash(string) % self.size
    def load(self):
        """Return the current load of the hash map."""
        return float(self.nitems) / self.size
```

三、测试

本次测试使用一个文本 alice.txt 作为数据集放入 cuckoo hash，随后进行 12 次基准测试观察多少错误率。

测试代码如下：

```
# -*- coding: utf-8 -*-
import unittest
from my_hash import CuckooHash


class TestCuckooHashConstruction(unittest.TestCase):
    def test_construction_valid_arg(self):
        self.assertEqual(3, CuckooHash(3.0).size)
        self.assertEqual(200, CuckooHash(200).size)
        self.assertEqual(0, CuckooHash(0).size)
        self.assertEqual(1000, CuckooHash(1000).size)
    def test_construction_bad_type(self):
        with self.assertRaises(TypeError):
            CuckooHash('string')
    def test_construction_bad_value(self):
        with self.assertRaises(ValueError):
            CuckooHash(-1)
        with self.assertRaises(ValueError):
            CuckooHash(-1.0)
        with self.assertRaises(ValueError):
            CuckooHash(2.2)
class TestCuckooHashSetGet(unittest.TestCase):
    def setUp(self):
        self.small_hash_map = CuckooHash(20)
        self.big_hash_map = CuckooHash(1000)
        self.alice_text = open('C:/Users/马世拓/Desktop/bloom-filter-main/alice.txt',
'r').read()
    def test_set_get_full(self):
        bulk_set_results = self._bulk_set(self.small_hash_map, 20)
        self.assertTrue(all(bulk_set_results))
        bulk_get_results = self._bulk_get(self.small_hash_map, 20)
        self.assertEqual(range(20), bulk_get_results)
    def test_good_keys(self):
        self.assertTrue(self.big_hash_map.set("", "somevalue"))
        self.assertTrue(self.big_hash_map.set("somekey", (1, 2, 3)))
        self.assertTrue(self.big_hash_map.set("ʕ•ᴥ•ʔ", "(•ω•)"))
        self.assertTrue(self.big_hash_map.set(self.alice_text, 'alice'))
        self.assertEqual(self.big_hash_map.get(""), "somevalue")
        self.assertEqual(self.big_hash_map.get("somekey"), (1, 2, 3))
        self.assertEqual(self.big_hash_map.get("ʕ•ᴥ•ʔ"), "(•ω•)")
```

```
            self.assertEqual(self.big_hash_map.get(self.alice_text), 'alice')
        def test_set_by_assignment(self):
            value = [1, 2, 3]
            self.big_hash_map.set("list", value)
            value.append(4)
            self.assertEqual(value, self.big_hash_map.get("list"))
        def test_bad_keys(self):
            with self.assertRaises(TypeError):
                self.small_hash_map.set([1], 0)
                self.small_hash_map.set(10, 20)
        def test_failed_set(self):
            self._bulk_set(self.small_hash_map, 20)
            self.assertFalse(self.small_hash_map.set('20th item', 5))
        def test_set_overwrite(self):
            self.small_hash_map.set('somekey', 0)
            self.assertTrue(self.small_hash_map.set('somekey', 1))
            self.assertEqual(1, self.small_hash_map.get('somekey'))
        def _bulk_set(self, hash_map, nitems):
            return [hash_map.set("key" + str(i), i) for i in range(nitems)]
        def _bulk_get(self, hash_map, nitems):
            return [hash_map.get("key" + str(i)) for i in range(nitems)]
    class TestCuckooHashDelete(unittest.TestCase):
        def setUp(self):
            self.hash_map = CuckooHash(50)
        def test_delete_existing(self):
            self.hash_map.set("somekey", 20)
            self.assertEqual(self.hash_map.delete("somekey"), 20)
            self.assertIsNone(self.hash_map.get("somekey"))
        def test_delete_nonexisting(self):
            self.assertIsNone(self.hash_map.delete("somekey"))
    class TestCuckooHashLoad(unittest.TestCase):
        def setUp(self):
            self.hash_map = CuckooHash(5)
        def test_load(self):
            self.assertEqual(self.hash_map.load(), 0)
            self.hash_map.set('key', 0)
            self.assertAlmostEqual(self.hash_map.load(), 0.2)
            self.hash_map.set('otherkey', 0)
            self.assertAlmostEqual(self.hash_map.load(), 0.4)
            self.hash_map.delete('key')
            self.assertAlmostEqual(self.hash_map.load(), 0.2)
            self.hash_map.delete('otherkey')
            self.assertEqual(self.hash_map.load(), 0)
    if __name__ == '__main__':
```

```
    unittest.main()
```

最终测试结果如图 3.1 所示

```
Python 3.10.4 (tags/v3.10.4:9d38120, Mar 23 2022, 23:13:41) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.

= RESTART: C:\Users\sx\Documents\Tencent Files\825442900\FileRecv\cuckoohashing-master\test_my_hash.py
......
Warning (from warnings module):
  File "C:\Users\sx\Documents\Tencent Files\825442900\FileRecv\cuckoohashing-master\test_my_hash.py", line 30
    self.alice_text = open('Alice.txt', 'r').read()
ResourceWarning: unclosed file <_io.TextIOWrapper name='Alice.txt' mode='r' encoding='cp936'>
....F.
======================================================================
FAIL: test_set_get_full (__main__.TestCuckooHashSetGet)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\Users\sx\Documents\Tencent Files\825442900\FileRecv\cuckoohashing-master\test_my_hash.py", line 36, in test_set_get_full
    self.assertEqual(range(20), bulk_get_results)
AssertionError: range(0, 20) != [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]

----------------------------------------------------------------------
Ran 12 tests in 0.204s

FAILED (failures=1)
```

图 3.1 实验结果

12 次基准测试中仅有一次失败，大致效果较为理想，可认为实验成功。

四、总结

本次报告针对 Cuckoo Hash 的基本原理和其他过滤器对比进行了探简单探讨，并对目前存在的问题进行分析和解决，并在最后通过 Python 实现了一个简单的 cuckoo Hash 并加以测试。

通过对这次课题的深入研究，让我对目前的大数据存储有了更深刻的了解，让我了解到在现在高速发展的社会，海量的数据需要记录处理和查询，对数据的管理要求也就更加严格，增强了我对计算机科学和大数据分析等领域的兴趣。

也十分感谢两位老师的辛勤付出和帮助，让我在这门课程的学习中受益匪浅。

参考文献

• R. Pagh and F. Rodler, “Cuckoo hashing,” Proc. ESA, pp. 121 – 133, 2001.

• Yu Hua, Hong Jiang, Dan Feng, "FAST: Near Real-time Searchable Data Analytics for the Cloud", Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC), November 2014, Pages: 754-765.

• Yu Hua, Bin Xiao, Xue Liu, "NEST: Locality-aware Approximate Query Service for Cloud Computing", Proceedings of the 32nd IEEE International Conference on Computer Communications (INFOCOM), April 2013, pages: 1327-1335.

• Qiuyu Li, Yu Hua, Wenbo He, Dan Feng, Zhenhua Nie, Yuanyuan Sun, "Necklace: An Efficient Cuckoo Hashing Scheme for Cloud Storage Services", Proceedings of IEEE/ACM International Symposium on Quality of Service (IWQoS), 2014.

• B. Fan, D. G. Andersen, and M. Kaminsky, “MemC3: Compact and concurrent memcache with dumber caching and smarter hashing,” Proc. USENIX NSDI, 2013.

• B. Debnath, S. Sengupta, and J. Li, “ChunkStash: speeding up inline storage deduplication using flash memory,” Proc. USENIX ATC, 2010.