2018 级

《物联网数据存储与管理》课程

报 告

姓　名　徐 屹

学　号　U201814727

班　号　物联网 1801 班

日　期　2021.06.21

目　录

一、理论分析 1
1.1 解决哈希冲突的办法和比较（时间和空间的权衡） 1
1.2 Cuckoo Hashing 算法描述 1
1.3 Cuckoo Hashing 图例 2
1.4 Cuckoo Hashing 优缺点 3
1.5 Cuckoo Hashing 优化（减少哈希碰撞） 4
1.6 Cuckoo Hashing 优化——使用桶（bucket）的 4 路槽位（slot） 4
1.7 Cuckoo Hashing 优化——增加哈希表 4
二、数据结构的设计（Node.js 下 Cuckoo Hashing 的实现） 6
2.1 目标 6
2.2 index.js 中建立 Cuckoo Hashing 哈希表 6
2.3 Cuckoo Hashing 哈希表的属性（read-only） 6
2.4 定义相关的操作 6
三、操作流程分析 8
四、实验测试的性能 9
4.1 比较 cuckoo hash table、集合(Set)和对象(Object)插入数据的性能 9
4.2 测试 HashTable 的建立 10
4.3 测试不同 key 和 value 下 HashTable 的功能 14
4.4 压力测试 19
五、总结 21

一、理论分析

1.1 解决哈希冲突的办法和比较（时间和空间的权衡）

（1）Dense Hash Table

Dense Hash Table 就是普通的用线性探查解决冲突的哈希表，如图所示。当插入 F 时，通过哈希函数将 F 映射到表中所指位置，发现该位置已存有 D 了，向后扫描探查，直到碰到一个空位，再将 F 插入。而查询一个 G 时，则需要从对应的位置开始向后找，直到找到 G（命中），或找到空位（G 不在表中）。这种结构对内存的利用不佳，如果想让哈希表尽量存满(k, v)对，那么插入/查询的性能将会严重下降。

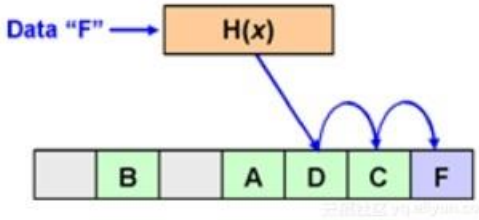


图 1 Dense Hash Table

期望查询的时间复杂度：O (1)；

（2）Chain Hash Table

Chain Hash Table 是利用拉链法解决冲突的哈希表，它的特点是空间利用率比较高，除了顺序存下所有(k, v)对之外仅需要一个索引来记录链表头，但由于链表的空间不连续，导致查询性能一般。

期望查询的时间复杂度：O (1+α)；

（3）Cuckoo Hashing

Cuckoo Hash Table 使用了两个哈希函数来解决冲突。Cuckoo 查询操作的理论复杂度为最差 O(1)，而 Cuckoo 的插入复杂度为均摊 O(1)。

特点：占用空间少，查询速度快。

引入 Cuckoo 是希望它在实际应用中，能够在较高的空间利用率下，仍然维持不错的查询性能。

1.2 Cuckoo Hashing 算法描述

使用 hashA、hashB 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，则踢出一个位置后插入，被踢出的对调用该算法，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

1.3 Cuckoo Hashing 图例

（1）插入 key1 两个位置均为空,则插入任意位置；

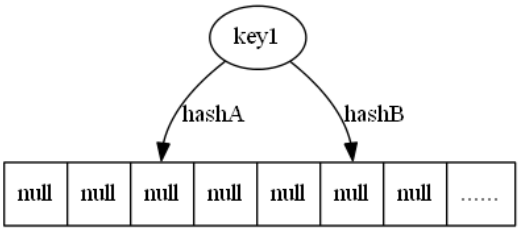


图 2 Cuckoo Hashing

（2）插入后；

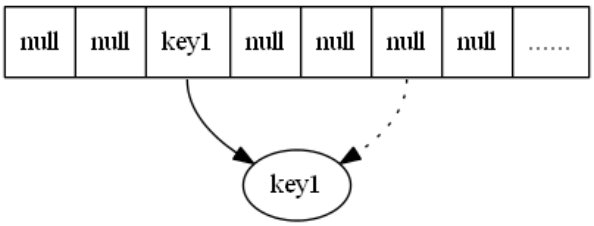


图 3 Cuckoo Hashing

（3）插入 key2 两个位置有一个位置为空,则插入空的位置中；

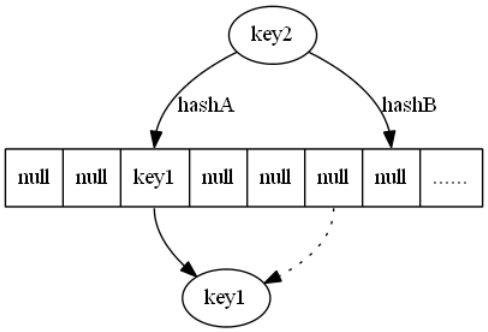


图 4 Cuckoo Hashing

（4）插入后效果；

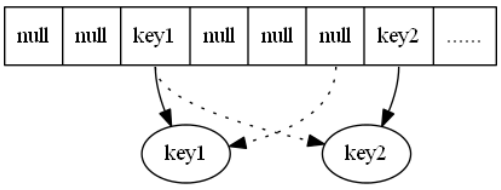


图 5 Cuckoo Hashing

（5）新插入 keyi 发现对应两个位置均被占据；

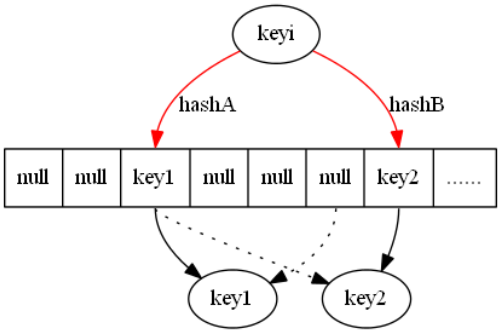


图 6 Cuckoo Hashing

（6）随机选择一个位置提出所在位置的 key（key1），将踢出的 key 放置在另一个哈希结果对应的位置上；

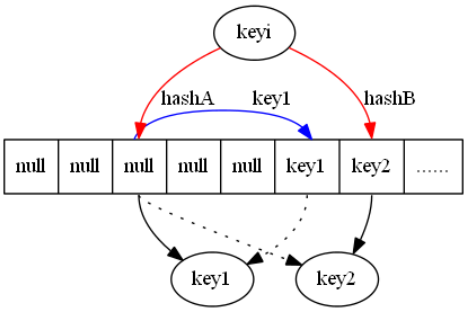


图 7 Cuckoo Hashing

（7）如果踢出的 key（key1）又占据/踢出了其他 key（keyj）的位置，则反复执行上面的过程直到结束；

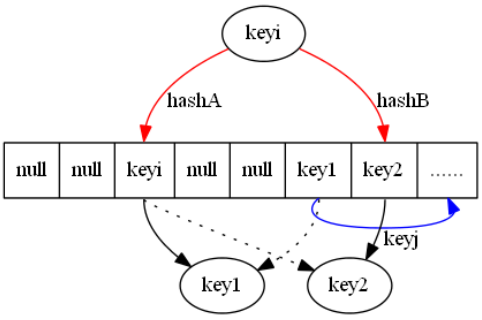


图 8 Cuckoo Hashing

1.4 Cuckoo Hashing 优缺点

cuckoo hashing 适合空间需求量大，对读性能要求高，对写性能相对低，操作比例读为主写为辅的场景。理由基于 Cuckoo hashing 的优点和缺点。

（1）优点

① 哈希表本身的空间利用率高；

② 查询可以使用两次读完成；

（2）缺点

① 插入操作的复杂度大；
② 读写的高并发算法复杂。

1.5 Cuckoo Hashing 优化（减少哈希碰撞）

（1）将一维改成多维，使用桶（bucket）的 4 路槽位（slot）；
（2）一个 key 对应多个 value；
（3）增加哈希函数，从两个增加到多个；
（4）增加哈希表，类似于第一种

1.6 Cuckoo Hashing 优化——使用桶（bucket）的 4 路槽位（slot）

一个改进的哈希表如下图所示，每个桶（bucket）有 4 路槽位（slot）。当哈希函数映射到同一个 bucket 中，在其它三路 slot 未被填满之前，是不会有元素被踢的，这大大缓冲了碰撞的几率。

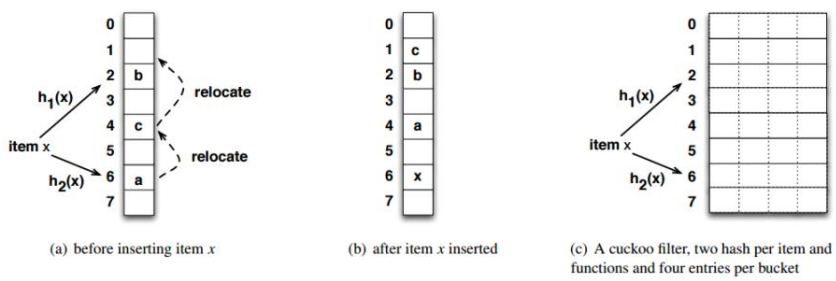


Figure 1: Illustration of cuckoo hashing

图 9 Cuckoo Hashing

1.7 Cuckoo Hashing 优化——增加哈希表

（1）当新插入一个 key hashA 在上面哈希表位置和 hashB 在下面哈希表的位置分别被 key1 和 keyx 占据，任选一个 key 提出（这里选择 key1）；

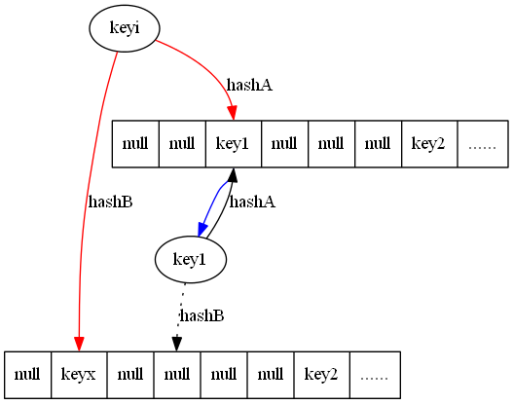


图 10 Cuckoo Hashing

（2）计算 key1 hashB 的值然后插入到下面的 hashB 对应的哈希表中。

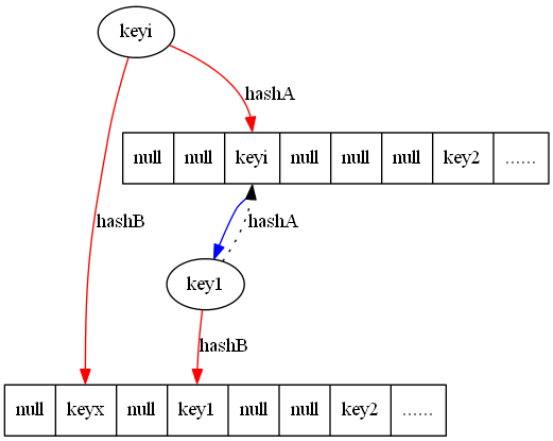


图 11 Cuckoo Hashing

二、数据结构的设计（Node.js 下 Cuckoo Hashing 的实现）

2.1 目标

在 Node.js 下建立快速、可靠的 Cuckoo Hashing 哈希表，并对该表的性能、不同 key 和 value 下的表现进行测试和分析。

2.2 index.js 中建立 Cuckoo Hashing 哈希表

var hashTable = new HashTable(keySize, valueSize, [elementsMin],

[elementsMax])

参数：

keySize：整数，必须是 4 字节的倍数，最多 64 字节（HashTable.KEY_MAX）；

valueSize：整数，从 0 字节到最多 1 MB （HashTable.VALUE_MAX）；

elementsMin：整数，提示预计插入的最小元素数量，以避免在短期内不必要的调整大小；

elementsMax：整数，提示预期插入的最大元素数量，以确保长期有足够的容量。

2.3 Cuckoo Hashing 哈希表的属性（read-only）

hashTable.capacity

作用：返回一个整数，读取哈希表当前总容量，即哈希表可容纳 100% 负载的元素数量，这将通过自动调整哈希表缓冲器的大小而增加。

hashTable.length

作用：返回一个整数，读取哈希表中实际存在的元素数量。

hashTable.load

作用：返回一个 0 和 1 之间的分数，读取哈希表的长度除以哈希表的容量的结果。

hashTable.size

作用：返回一个整数，按字节读取所有哈希表缓冲器的总大小。

2.4 定义相关的操作

hashTable.set(key, keyOffset, value, valueOffset)

作用：在哈希表中插入或更新元素；

参数：

key：缓冲区，包含要插入或更新的 key；

keyOffset：整数，key 的偏移；

value：缓冲区，包含要插入或更新的 value；

valueOffset：整数，value 的偏移；

返回：返回一个整数，如果元素已插入则返回 0，如果元素已更新则返回 1。

hashTable.get(key, keyOffset, value, valueOffset)

作用：从哈希表中检索元素的价值；
参数：
key：缓冲区，包含要检索的密钥；
keyOffset：整数，key 的偏移；
value：缓冲区中，如果元素存在，元素值将被复制到此缓冲区中；
valueOffset：整数，value 的偏移；
返回：返回一个整数，如果找不到元素则返回 0，如果发现元素则返回 1。

hashTable.exist(key, keyOffset)

作用：测试哈希表中是否存在元素；
参数：
key：缓冲区，包含要测试的密钥；
keyOffset：整数，key 的偏移；
返回：返回一个整数，如果找不到元素则返回 0，如果发现元素则返回 1。

hashTable.unset(key, keyOffset)

作用：从哈希表中删除元素。
参数：
key：缓冲区，包含要删除的密钥；
keyOffset：整数，key 的偏移；
返回：返回一个整数，如果未找到元素则返回 0，如果元素被移除则返回 1。

hashTable.cache(key, keyOffset, value, valueOffset)

作用：类似于 set()，但插入驱逐最近使用最少的元素，而不是调整哈希表的大小；
参数：
key：缓冲区,包含要插入或更新的密钥；
keyOffset：整数，key 的偏移；
value：缓冲区，包含要插入或更新的 value；
valueOffset：整数，value 的偏移;
返回：返回一个整数，如果元素已插入则返回 0，如果元素已更新则返回 1，如果元素通过驱逐其他元素插入，则返 2。

三、操作流程分析

（1）安装
在 VS code 的终端输入：
```
npm install @ronomon/hash-table
```

（2）设计数据结构
index.js 中建立 Cuckoo Hashing 哈希表，并定义相关的操作

（3）测试
设计测试文件，引入 index.js 中的数据结构，输入：
node 测试文件的名字

本次进行了三次测试，测试文件分别为 vanilla.js、test.js、benchmark.js、stress.js。

四、实验测试的性能

4.1 比较 cuckoo hash table、集合(Set)和对象(Object)插入数据的性能

测试数据如下：

```
var keySize = 16;
var valueSize = 0;
var element = keySize + valueSize;
var elements = 4000000;
var buffer = Node.crypto.randomBytes(element * elements);
```

（1）cuckoo hash table

用 index.js 中定义的操作，首先初始化表：

```
var table = new HashTable(keySize, valueSize, elements, elements);
```

然后依次循环地插入数据：

```
table.set(buffer, offset, buffer, offset + keySize);
```

（2）集合（Set）

数据类型 Set 可以直接使用，首先进行初始化：

```
var set = new Set();
```

然后依次循环地插入数据（不能存放重复的元素）：

```
set.add(buffer.slice(offset, offset + keySize));
```

（3）对象（Object）

数据类型 Object 可以直接使用，首先进行初始化：

```
var object = {};
```

然后依次循环地插入数据（因为依次划分 value 就更慢了，未测试前就可以推断使用 Object 就是最慢的，所以这里干脆简化一下，直接让 value 为 1）：

```
var key = buffer.toString('base64', offset, offset + keySize);
object[key] = 1;
```

测试结果如下图所示：

```
PS C:\Users\19686\Desktop\hash-table-master\hash-table-master> node vanilla.js

                      key=16 bytes, value=0 bytes

  @ronomon/hash-table: Inserting 4000000 elements...
  @ronomon/hash-table: 1314ms

            new Set(): Inserting 4000000 elements...
            new Set(): 2630ms

       vanilla object: Inserting 4000000 elements...
       vanilla object: 6871ms
```

图 12 测试一的结果

分析：从图中可以看出，处理速度上，cuckoo hash table > 集合（Set）> 对象（Object），体现 cuckoo hash table 插入数据要更加快速、可靠。

4.2 测试 HashTable 的建立

进行测试的 keySize 和 valueSize 的值有：

```
var KEY_SIZES = [4, 8, 12, 16, 20, 24, 28, 32, 36, 40, 44, 48, 52, 56, 60
, 64];
var VALUE_SIZES = [
  0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11,
  12, 13, 14, 16, 18, 20, 22, 23,
  24, 28, 32, 36, 40, 44, 48, 52, 56, 60, 61,
  64, 128, 256, 1024, 4096, 65536, 65537
];
```

对每一组进行测试：

```
KEY_SIZES.forEach(
  function(keySize) {
    var manyIndex = Math.floor(Math.random() * VALUE_SIZES.length);
    var cacheIndex = Math.floor(Math.random() * VALUE_SIZES.length);
    VALUE_SIZES.forEach(
      function(valueSize, valueSizeIndex) {
        var many = valueSizeIndex === manyIndex && Math.random() < 0.5;
        var cache = valueSizeIndex === cacheIndex;
        Test(keySize, valueSize, many, cache);
      }
    );
  }
);
```

上面 Test 的具体定义（代码较多，这里只取关键的，删去报错的语句）：

```
var table = new HashTable(keySize, valueSize, elementsMin, elementsMax);
```

//定义函数 Mutate，当 id < elements 循环 Mutate 函数

```
function Mutate(id, reset) {
    var keyOffset = id * keySize;
    var valueOffset = id * valueSize;
    function Cache() {
      if (!cache) return;
      var source = ENTROPY;
      var sourceOffset = (ENTROPY_OFFSET += valueSize);
      if (sourceOffset + valueSize > ENTROPY.length) {
        sourceOffset = ENTROPY_OFFSET = 0;
      }
      var sourceHash1 = Hash(source, sourceOffset, valueSize);
```

```
      var result = table.cache(key, keyOffset, source, sourceOffset);
      var sourceHash2 = Hash(source, sourceOffset, valueSize);

      if (result === 1) {
        // Updated
        Assert('result', result, state[id]);
      } else if (result === 2) {
        // Evicted
        state[id] = 1;
      } else {
        // Inserted
        Assert('result', result, 0);
        state[id] = 1;
        tableLength++;
      }
    }
    function Exist(cached) {
      var result = table.exist(key, keyOffset);
    }
    function Get(cached) {
      var target = TARGET;
      var targetOffset = Math.floor(
        Math.random() * (TARGET.length - valueSize)
      );
      if (!state[id] || cache) {
        var targetHex = target.toString(
          'hex',
          targetOffset,
          targetOffset + valueSize
        );
      }
      var result = table.get(key, keyOffset, target, targetOffset);
      if (result === 1) {
        var targetSlice = target.slice(targetOffset, targetOffset + value
Size);
        var valueSlice = value.slice(valueOffset, valueOffset + valueSize
);
      } else {
        Assert(
          'target',
          target.toString('hex', targetOffset, targetOffset + valueSize),
          targetHex
        );
      }
```

```
    }
    function Set() {
      if (cache) return;
      var source = ENTROPY;
      var sourceOffset = (ENTROPY_OFFSET += valueSize);
      if (sourceOffset + valueSize > ENTROPY.length) {
        sourceOffset = ENTROPY_OFFSET = 0;
      }
      var result = table.set(key, keyOffset, source, sourceOffset);
      if (!state[id]) {
        state[id] = 1;
        tableLength++;
      }
    }
    function Unset() {
      var result = table.unset(key, keyOffset);
      if (cache) {
        if (result === 1) {
          state[id] = 0;
          tableLength--;
        }
      } else {
        if (state[id]) {
          state[id] = 0;
          tableLength--;
        }
      }
    }
    if (leader) {
      leader = false;
    }
    if (reset) {
      Get(0);
      Exist(0);
      Unset();
      Get(0);
      Exist(0);
    } else {
      Get(0);
      Exist(0);
      if (Random() < 0.50) {
        Unset();
        Get(0);
        Exist(0);
```

```
      }
      if (Random() < 0.50) {
        Cache();
        Set();
        Get(1);
        Exist(1);
      }
      if (Random() < 0.25) {
        Unset();
        Get(0);
        Exist(0);
      }
      if (Random() < 0.25) {
        Cache();
        Set();
        Get(1);
        Exist(1);
      }
    }
    var stats = { capacity: 0, size: 0 };
    for (var index = 0, length = table.tables.length; index < length; ind
ex++) {
      var tableSize = table.tables[index].buffer.length;
      stats.capacity += (tableSize / table.bucket) * 8;
      stats.size += tableSize;
    }
  }
```

//输出

```
Log(
    'HashTable:' +
    ' keySize=' + keySize.toString().padEnd(3, ' ') +
    ' valueSize=' + valueSize.toString().padEnd(6, ' ') +
    ' buffers=' + table.tables.length.toString().padEnd(5, ' ') +
    ' elements=' + elements
  );
```

输出结果如下图所示（共 607 组，这里只展示部分）：

```
PS C:\Users\19686\Desktop\hash-table-master\hash-table-master> node test.js
HashTable: keySize=4   valueSize=0      buffers=1     elements=33004
HashTable: keySize=4   valueSize=1      buffers=1     elements=4
HashTable: keySize=4   valueSize=2      buffers=16    elements=29
HashTable: keySize=4   valueSize=3      buffers=16    elements=45
HashTable: keySize=4   valueSize=4      buffers=1     elements=13
HashTable: keySize=4   valueSize=5      buffers=1     elements=21
HashTable: keySize=4   valueSize=6      buffers=16    elements=15
HashTable: keySize=4   valueSize=7      buffers=1     elements=2
HashTable: keySize=4   valueSize=8      buffers=16    elements=16
HashTable: keySize=4   valueSize=9      buffers=16    elements=6
HashTable: keySize=4   valueSize=10     buffers=1     elements=17
```

图 13 测试二的结果

```
HashTable: keySize=64  valueSize=61     buffers=1     elements=56
HashTable: keySize=64  valueSize=64     buffers=1     elements=14
HashTable: keySize=64  valueSize=128    buffers=16    elements=8
HashTable: keySize=64  valueSize=256    buffers=16    elements=61
HashTable: keySize=64  valueSize=1024   buffers=1     elements=55
HashTable: keySize=64  valueSize=4096   buffers=1     elements=47
HashTable: keySize=64  valueSize=65536  buffers=1     elements=32
HashTable: keySize=64  valueSize=65537  buffers=1     elements=5
================================================================================
PASSED ALL TESTS
================================================================================
```

图 14 测试二的结果

分析：从图中最后一行可以看出，通过了所有的测试。

4.3 测试不同 key 和 value 下 HashTable 的功能

进行测试的 keySize 和 valueSize 的值有：

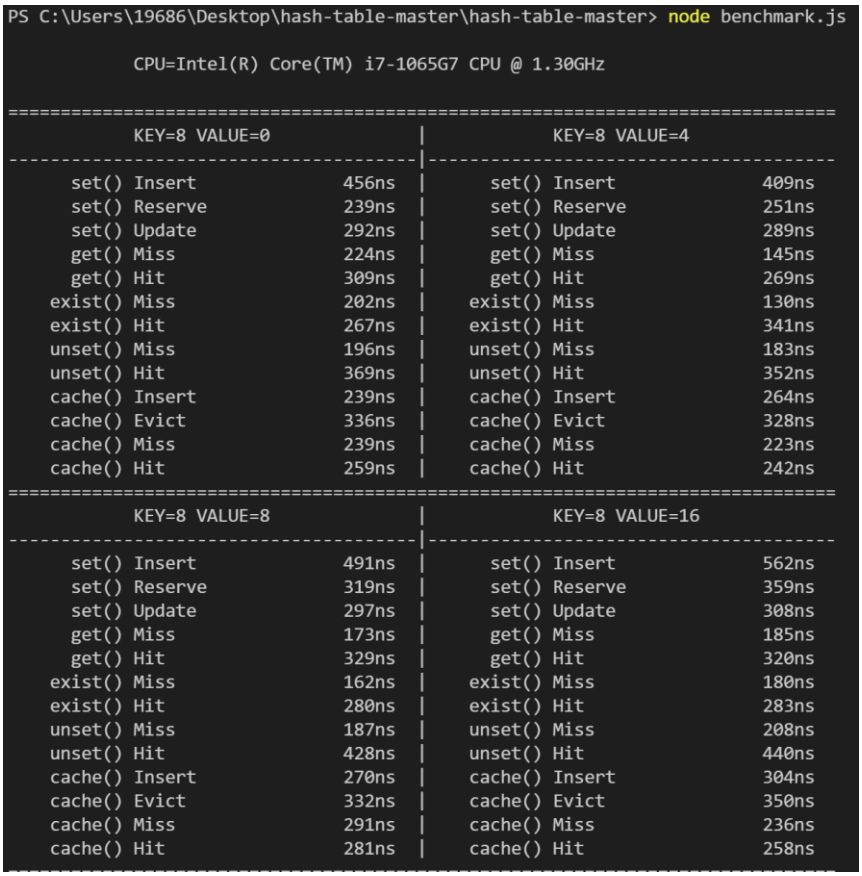
```
var KEY_SIZES = [8, 16, 32, 64];
var VALUE_SIZES = [0, 4, 8, 16, 32, 64, 4096, 65536];
```

对每一组进行测试：

```
KEY_SIZES.forEach(
  function(keySize) {
    VALUE_SIZES.forEach(
      function(valueSize) {
        benchmark(keySize, valueSize);
        var results = benchmark(keySize, valueSize);
        display(keySize, valueSize, results);
      }
    );
  }
);
```

上面的 **benchmark** 函数是对 **HashTable** 的功能（见 **2.4**）依次的进行测试，**display** 函数会将测试结果在终端展示。

输出结果如下图所示：



```
PS C:\Users\19686\Desktop\hash-table-master\hash-table-master> node benchmark.js

           CPU=Intel(R) Core(TM) i7-1065G7 CPU @ 1.30GHz

================================================================================
           KEY=8 VALUE=0            |           KEY=8 VALUE=4
------------------------------------|------------------------------------
     set() Insert             456ns |     set() Insert             409ns
     set() Reserve            239ns |     set() Reserve            251ns
     set() Update             292ns |     set() Update             289ns
     get() Miss               224ns |     get() Miss               145ns
     get() Hit                309ns |     get() Hit                269ns
   exist() Miss               202ns |   exist() Miss               130ns
   exist() Hit                267ns |   exist() Hit                341ns
   unset() Miss               196ns |   unset() Miss               183ns
   unset() Hit                369ns |   unset() Hit                352ns
   cache() Insert             239ns |   cache() Insert             264ns
   cache() Evict              336ns |   cache() Evict              328ns
   cache() Miss               239ns |   cache() Miss               223ns
   cache() Hit                259ns |   cache() Hit                242ns
================================================================================
           KEY=8 VALUE=8            |           KEY=8 VALUE=16
------------------------------------|------------------------------------
     set() Insert             491ns |     set() Insert             562ns
     set() Reserve            319ns |     set() Reserve            359ns
     set() Update             297ns |     set() Update             308ns
     get() Miss               173ns |     get() Miss               185ns
     get() Hit                329ns |     get() Hit                320ns
   exist() Miss               162ns |   exist() Miss               180ns
   exist() Hit                280ns |   exist() Hit                283ns
   unset() Miss               187ns |   unset() Miss               208ns
   unset() Hit                428ns |   unset() Hit                440ns
   cache() Insert             270ns |   cache() Insert             304ns
   cache() Evict              332ns |   cache() Evict              350ns
   cache() Miss               291ns |   cache() Miss               236ns
   cache() Hit                281ns |   cache() Hit                258ns
--------------------------------------------------------------------------------
```

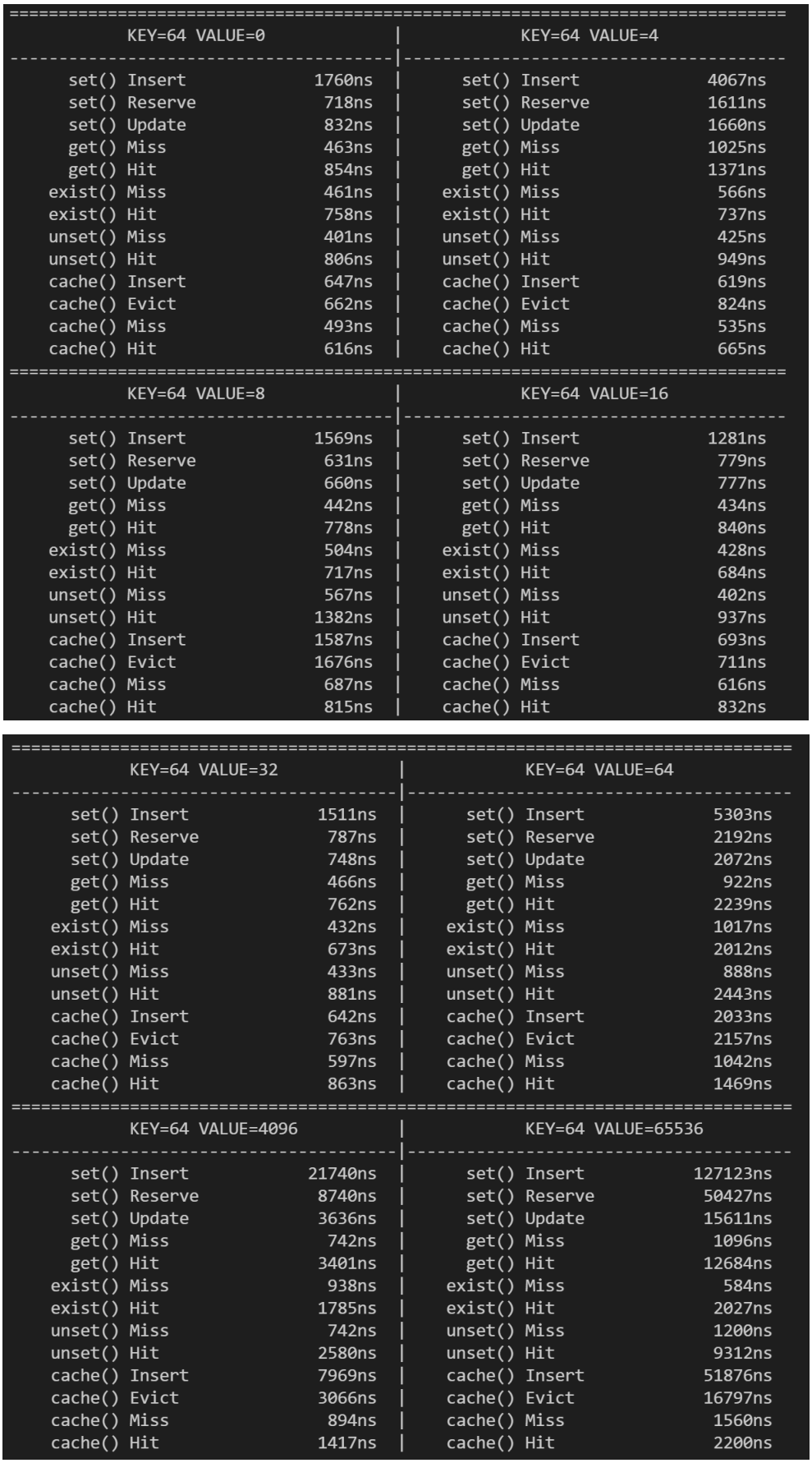
```
================================================================================
           KEY=8 VALUE=32           |           KEY=8 VALUE=64
------------------------------------|------------------------------------
     set() Insert            1965ns |     set() Insert            2760ns
     set() Reserve            826ns |     set() Reserve           1111ns
     set() Update             848ns |     set() Update            1113ns
     get() Miss               346ns |     get() Miss               371ns
     get() Hit                882ns |     get() Hit               1130ns
   exist() Miss               388ns |   exist() Miss               405ns
   exist() Hit                585ns |   exist() Hit                586ns
   unset() Miss               402ns |   unset() Miss               379ns
   unset() Hit               1069ns |   unset() Hit               1232ns
   cache() Insert             733ns |   cache() Insert            1033ns
   cache() Evict              958ns |   cache() Evict             1205ns
   cache() Miss               607ns |   cache() Miss               701ns
   cache() Hit                574ns |   cache() Hit                628ns
================================================================================
           KEY=8 VALUE=4096         |           KEY=8 VALUE=65536
------------------------------------|------------------------------------
     set() Insert           14009ns |     set() Insert          186962ns
     set() Reserve           5829ns |     set() Reserve          92331ns
     set() Update            1979ns |     set() Update           15524ns
     get() Miss               410ns |     get() Miss               488ns
     get() Hit               1493ns |     get() Hit              14590ns
   exist() Miss               360ns |   exist() Miss               285ns
   exist() Hit                611ns |   exist() Hit                605ns
   unset() Miss               322ns |   unset() Miss               217ns
   unset() Hit               1558ns |   unset() Hit               8047ns
   cache() Insert            5643ns |   cache() Insert           78144ns
   cache() Evict             2213ns |   cache() Evict            16799ns
   cache() Miss               343ns |   cache() Miss               386ns
   cache() Hit                594ns |   cache() Hit                581ns
```

KEY=16 VALUE=0		KEY=16 VALUE=4	
set() Insert	1458ns	set() Insert	1579ns
set() Reserve	776ns	set() Reserve	734ns
set() Update	735ns	set() Update	840ns
get() Miss	405ns	get() Miss	451ns
get() Hit	708ns	get() Hit	764ns
exist() Miss	440ns	exist() Miss	432ns
exist() Hit	723ns	exist() Hit	681ns
unset() Miss	444ns	unset() Miss	438ns
unset() Hit	1078ns	unset() Hit	1160ns
cache() Insert	679ns	cache() Insert	694ns
cache() Evict	884ns	cache() Evict	913ns
cache() Miss	640ns	cache() Miss	620ns
cache() Hit	684ns	cache() Hit	699ns
KEY=16 VALUE=8		KEY=16 VALUE=16	
set() Insert	757ns	set() Insert	791ns
set() Reserve	416ns	set() Reserve	401ns
set() Update	347ns	set() Update	434ns
get() Miss	236ns	get() Miss	272ns
get() Hit	386ns	get() Hit	416ns
exist() Miss	248ns	exist() Miss	271ns
exist() Hit	413ns	exist() Hit	372ns
unset() Miss	297ns	unset() Miss	281ns
unset() Hit	532ns	unset() Hit	518ns
cache() Insert	367ns	cache() Insert	373ns
cache() Evict	904ns	cache() Evict	420ns
cache() Miss	691ns	cache() Miss	365ns
cache() Hit	832ns	cache() Hit	348ns

KEY=16 VALUE=32		KEY=16 VALUE=64	
set() Insert	2231ns	set() Insert	2703ns
set() Reserve	943ns	set() Reserve	1543ns
set() Update	1049ns	set() Update	1451ns
get() Miss	517ns	get() Miss	447ns
get() Hit	966ns	get() Hit	1289ns
exist() Miss	463ns	exist() Miss	462ns
exist() Hit	749ns	exist() Hit	803ns
unset() Miss	450ns	unset() Miss	468ns
unset() Hit	1368ns	unset() Hit	1563ns
cache() Insert	1001ns	cache() Insert	1180ns
cache() Evict	1092ns	cache() Evict	1360ns
cache() Miss	617ns	cache() Miss	695ns
cache() Hit	686ns	cache() Hit	778ns
KEY=16 VALUE=4096		KEY=16 VALUE=65536	
set() Insert	18044ns	set() Insert	159941ns
set() Reserve	7402ns	set() Reserve	64537ns
set() Update	2151ns	set() Update	15336ns
get() Miss	545ns	get() Miss	489ns
get() Hit	2404ns	get() Hit	15840ns
exist() Miss	479ns	exist() Miss	502ns
exist() Hit	1068ns	exist() Hit	1557ns
unset() Miss	344ns	unset() Miss	649ns
unset() Hit	2329ns	unset() Hit	8916ns
cache() Insert	6756ns	cache() Insert	69300ns
cache() Evict	2414ns	cache() Evict	16879ns
cache() Miss	485ns	cache() Miss	751ns
cache() Hit	666ns	cache() Hit	1104ns

KEY=32 VALUE=0		KEY=32 VALUE=4	
set() Insert	1991ns	set() Insert	818ns
set() Reserve	1092ns	set() Reserve	421ns
set() Update	1044ns	set() Update	447ns
get() Miss	576ns	get() Miss	292ns
get() Hit	1014ns	get() Hit	479ns
exist() Miss	625ns	exist() Miss	276ns
exist() Hit	983ns	exist() Hit	425ns
unset() Miss	580ns	unset() Miss	276ns
unset() Hit	1372ns	unset() Hit	545ns
cache() Insert	861ns	cache() Insert	388ns
cache() Evict	654ns	cache() Evict	454ns
cache() Miss	349ns	cache() Miss	413ns
cache() Hit	449ns	cache() Hit	420ns
KEY=32 VALUE=8		KEY=32 VALUE=16	
set() Insert	1047ns	set() Insert	1178ns
set() Reserve	416ns	set() Reserve	442ns
set() Update	456ns	set() Update	487ns
get() Miss	283ns	get() Miss	292ns
get() Hit	525ns	get() Hit	491ns
exist() Miss	380ns	exist() Miss	331ns
exist() Hit	513ns	exist() Hit	516ns
unset() Miss	289ns	unset() Miss	324ns
unset() Hit	585ns	unset() Hit	651ns
cache() Insert	427ns	cache() Insert	413ns
cache() Evict	480ns	cache() Evict	488ns
cache() Miss	377ns	cache() Miss	377ns
cache() Hit	461ns	cache() Hit	499ns

KEY=32 VALUE=32		KEY=32 VALUE=64	
set() Insert	1527ns	set() Insert	2531ns
set() Reserve	1251ns	set() Reserve	1638ns
set() Update	1339ns	set() Update	1678ns
get() Miss	699ns	get() Miss	645ns
get() Hit	1293ns	get() Hit	1629ns
exist() Miss	601ns	exist() Miss	583ns
exist() Hit	1038ns	exist() Hit	1148ns
unset() Miss	647ns	unset() Miss	719ns
unset() Hit	1559ns	unset() Hit	1690ns
cache() Insert	1072ns	cache() Insert	1358ns
cache() Evict	1301ns	cache() Evict	1575ns
cache() Miss	860ns	cache() Miss	920ns
cache() Hit	1151ns	cache() Hit	1019ns
KEY=32 VALUE=4096		KEY=32 VALUE=65536	
set() Insert	19703ns	set() Insert	166452ns
set() Reserve	8140ns	set() Reserve	71402ns
set() Update	3029ns	set() Update	16023ns
get() Miss	507ns	get() Miss	563ns
get() Hit	2500ns	get() Hit	14689ns
exist() Miss	550ns	exist() Miss	459ns
exist() Hit	1194ns	exist() Hit	953ns
unset() Miss	729ns	unset() Miss	692ns
unset() Hit	3039ns	unset() Hit	9037ns
cache() Insert	6709ns	cache() Insert	64554ns
cache() Evict	2635ns	cache() Evict	17283ns
cache() Miss	625ns	cache() Miss	1309ns
cache() Hit	967ns	cache() Hit	1337ns



KEY=64 VALUE=0		KEY=64 VALUE=4	
set() Insert	1760ns	set() Insert	4067ns
set() Reserve	718ns	set() Reserve	1611ns
set() Update	832ns	set() Update	1660ns
get() Miss	463ns	get() Miss	1025ns
get() Hit	854ns	get() Hit	1371ns
exist() Miss	461ns	exist() Miss	566ns
exist() Hit	758ns	exist() Hit	737ns
unset() Miss	401ns	unset() Miss	425ns
unset() Hit	806ns	unset() Hit	949ns
cache() Insert	647ns	cache() Insert	619ns
cache() Evict	662ns	cache() Evict	824ns
cache() Miss	493ns	cache() Miss	535ns
cache() Hit	616ns	cache() Hit	665ns
KEY=64 VALUE=8		KEY=64 VALUE=16	
set() Insert	1569ns	set() Insert	1281ns
set() Reserve	631ns	set() Reserve	779ns
set() Update	660ns	set() Update	777ns
get() Miss	442ns	get() Miss	434ns
get() Hit	778ns	get() Hit	840ns
exist() Miss	504ns	exist() Miss	428ns
exist() Hit	717ns	exist() Hit	684ns
unset() Miss	567ns	unset() Miss	402ns
unset() Hit	1382ns	unset() Hit	937ns
cache() Insert	1587ns	cache() Insert	693ns
cache() Evict	1676ns	cache() Evict	711ns
cache() Miss	687ns	cache() Miss	616ns
cache() Hit	815ns	cache() Hit	832ns

KEY=64 VALUE=32		KEY=64 VALUE=64	
set() Insert	1511ns	set() Insert	5303ns
set() Reserve	787ns	set() Reserve	2192ns
set() Update	748ns	set() Update	2072ns
get() Miss	466ns	get() Miss	922ns
get() Hit	762ns	get() Hit	2239ns
exist() Miss	432ns	exist() Miss	1017ns
exist() Hit	673ns	exist() Hit	2012ns
unset() Miss	433ns	unset() Miss	888ns
unset() Hit	881ns	unset() Hit	2443ns
cache() Insert	642ns	cache() Insert	2033ns
cache() Evict	763ns	cache() Evict	2157ns
cache() Miss	597ns	cache() Miss	1042ns
cache() Hit	863ns	cache() Hit	1469ns
KEY=64 VALUE=4096		KEY=64 VALUE=65536	
set() Insert	21740ns	set() Insert	127123ns
set() Reserve	8740ns	set() Reserve	50427ns
set() Update	3636ns	set() Update	15611ns
get() Miss	742ns	get() Miss	1096ns
get() Hit	3401ns	get() Hit	12684ns
exist() Miss	938ns	exist() Miss	584ns
exist() Hit	1785ns	exist() Hit	2027ns
unset() Miss	742ns	unset() Miss	1200ns
unset() Hit	2580ns	unset() Hit	9312ns
cache() Insert	7969ns	cache() Insert	51876ns
cache() Evict	3066ns	cache() Evict	16797ns
cache() Miss	894ns	cache() Miss	1560ns
cache() Hit	1417ns	cache() Hit	2200ns

图 15 测试三的结果

4.4 压力测试

一个简单的压力测试，使用尽可能多的可用内存。

计算内存占用：

```
var size = Math.round(Node.os.freemem() * 0.8);
```

测试：

```
for (var index = 0; index < elements; index++) {
  var result = hashTable.cache(key, keyOffset, value, valueOffset);
  if (result === 0) stats.inserts++;
  if (result === 1) stats.updates++;
  if (result === 2) stats.evictions++;
  if (hashTable.get(key, keyOffset, temp, 0) !== 1) {
    throw new Error('get() after cache() !== 1');
  }
  if (!temp.equals(value.slice(valueOffset, valueOffset + valueSize))) {
    throw new Error('get() after cache() received a different value');
  }
  keyOffset += keySize;
  valueOffset += valueSize;
  if (valueOffset + valueSize > value.length) valueOffset = 0;
  if (keyOffset === key.length) {
    console.log(pad(index + 1) + '/' + pad(elements));
    if (index !== elements - 1) throw new Error('key wrapped unexpectedly
');
  } else if (index % 1000 === 0) {
    console.log(pad(index) + '/' + pad(elements));
  }
}
```

测试结果如下图所示：

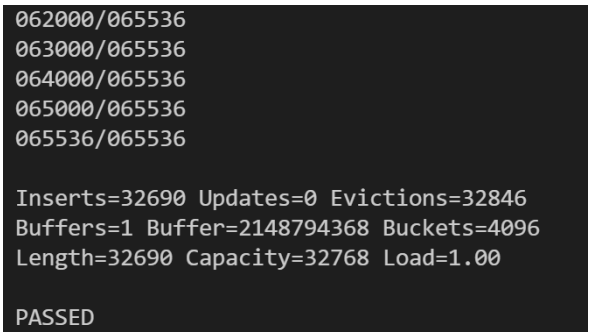
```
PS C:\Users\19686\Desktop\hash-table-master\hash-table-master> node stress.js

System has 2487667917 bytes of free memory available.
keySize=32 valueSize=65536 elementsMin=18969 capacity=32768 size=2148794368

Inserting 65536 elements...

000000/065536
001000/065536
002000/065536
003000/065536
004000/065536
```

图 16 测试四的结果


062000/065536
063000/065536
064000/065536
065000/065536
065536/065536
Inserts=32690 Updates=0 Evictions=32846
Buffers=1 Buffer=2148794368 Buckets=4096
Length=32690 Capacity=32768 Load=1.00
PASSED
图 17 测试四的结果

五、总结

本次报告深入探索了解决哈希冲突的办法，重点研究了 Cuckoo Hashing 的原理、优化以及使用，其优化比如有老师课上讲的每个桶（bucket）可以有 4 路槽位、还可以两个桶共享一个桶（那么扩容时只要对共享桶进行操作，则只操作了 1/3 的桶），但后来写这份报告时，想再仔细回顾和确认一下时，很可惜的是没在课件中找到这部分。

不过令我很惊喜的是发现 Node.js 中的@ronomon/hash-table 建立了快速、可靠的 Cuckoo Hashing 哈希表，并且对该表的性能、不同 key 和 value 下的表现进行测试和分析。

因为在前端中有接触过 javascript，所以安装和使用非常的容易上手，我自己也进行了测试，从测试的结果也可以看出使用 Cuckoo Hashing 的优越性：具有多个设计决策和优化功能；每个元素，一个 key 和相应的 value，最多可以位于 2 个可能的存储桶中的 1 个，保证在最坏的情况下持续查找时间；每个存储桶包含多达 8 个元素，以支持 80% 或更高的哈希表负载因数，与支持最多 50% 的负载因子和浪费其他 50% 内存的线性（Dense Hash Table）或链式哈希表（Chain Hash Table）不同。更高效的负载系数意味着表大小的频率较低；每个哈希表实例在多个缓冲区分开，并以线性最高 4，294，967，296 个元素或 16 TB 内存进行分隔……

现代的海量数据规模巨大且呈指数增长，还面临着降低延迟、访问瓶颈、数据迁移、查询效率等等众多艰巨的挑战，而通过学习这门课，我受益匪浅，能够大概地了解了一下数据存储和管理的原理和方式，包括这次深入解决哈希冲突的办法，之前学数据结构、做区块链报告时就有遇到过哈希冲突，这次查资料时又发现区块链中有的用的是 cuckoo 的思想，能在学习的过程中不断地扩展知识面、甚至知识与知识之间关联起来，也是学习的一种乐趣。