深入理解并实现基本的哈希表数据结构

黄梓淳

Oct 22, 2025

1 导言

在日常生活中，我们经常遇到需要快速查找信息的场景，例如使用字典查询单词释义、翻阅电话簿查找联系人号码，或者通过图书馆索引定位书籍位置。这些场景都基于一个共同的概念 —— 「键-值」对，其中每个键都唯一对应一个值。如果使用简单的数组来存储这些键值对，查找特定元素时可能需要遍历整个数组，导致时间复杂度高达 $O(n)$，当数据量巨大时，这种线性搜索的效率将变得极低。哈希表（Hash Table）作为一种高效的数据结构，能够在平均 $O(1)$ 时间复杂度下完成插入、删除和查找操作，极大地提升了数据处理的性能。本文将深入解析哈希表的核心工作原理，并逐步指导读者使用 Python 语言实现一个基本的哈希表，帮助大家从理论过渡到实践。

2 第一部分：哈希表的核心思想与工作原理

2.1 1.1 什么是哈希表?

哈希表是一种基于哈希函数的数据结构，它通过将键（Key）映射到数组中的特定位置来存储和访问对应的值（Value）。这种设计巧妙地利用了「空间换时间」的策略，将查找操作的平均时间复杂度从 $O(n)$ 降低到 $O(1)$。具体来说，哈希表的核心在于使用一个数组作为底层存储，并通过哈希函数将任意键转换为固定范围内的整数索引，从而直接定位到数组中的位置进行数据操作。

2.2 1.2 哈希函数

哈希函数在哈希表中扮演着关键角色，它的主要作用是将任意大小的输入（即键）转换为一个固定范围的整数值，这个值作为数组的索引。一个理想的哈希函数应具备三个重要特性：确定性、高效性和均匀分布性。确定性确保相同的键总是产生相同的哈希值；高效性要求计算过程快速，不影响整体性能；均匀分布性则保证键的哈希值尽可能均匀地分布在数组空间中，从而最小化冲突的发生。例如，对于字符串键，常用的哈希函数可能基于字符的 ASCII 值进行加权求和，再对数组容量取模。

2.3 1.3 哈希冲突

哈希冲突是指两个或多个不同的键经过哈希函数计算后，得到相同的数组索引。根据鸽巢原理，由于哈希函数的输出范围有限，而输入键的数量可能无限，冲突是不可避免的。解决哈希冲突的方法有多种，其中链地址法和开

放定址法是最常见的两种。本文将重点介绍链地址法的实现，因为它简单易懂且能有效处理冲突。

3 第二部分：解决哈希冲突的关键技术

3.1 2.1 链地址法

链地址法的核心思想是将哈希表的每个数组位置（称为「桶」）视为一个链表的头节点。当多个键映射到同一个索引时，它们会被存储在同一个链表中。插入操作时，先计算键的哈希值找到对应桶，然后遍历链表：如果键已存在，则更新其值；否则，在链表末尾添加新节点。查找操作类似，通过哈希值定位桶后遍历链表比较键。删除操作则需要遍历链表并移除对应节点。链地址法的优点是实现简单，能有效处理冲突；缺点是需要额外空间存储指针，且如果链表过长，性能可能退化为 $O(n)$。

3.2 2.2 开放定址法

开放定址法是另一种解决冲突的方法，它要求所有元素都存储在数组本身中。当发生冲突时，系统会按照预定义的探测序列（如线性探测、二次探测或双重哈希）在数组中寻找下一个空闲位置。例如，线性探测会依次检查索引 $i+1$、$i+2$ 等，直到找到空位。与链地址法相比，开放定址法节省了指针空间，但可能面临聚集问题，影响性能。本文主要聚焦链地址法的实现，但了解开放定址法有助于拓宽对哈希表设计的认识。

4 第三部分：动手实现一个哈希表（以链地址法为例）

4.1 3.1 设计数据结构

在实现哈希表时，我们首先需要定义其底层数据结构。一个基本的哈希表类（例如 MyHashMap）应包含以下属性：size 表示当前键值对的数量；capacity 指定底层数组的容量；load_factor 为负载因子（默认 0.75），用于触发扩容；buckets 是一个数组，每个元素是一个链表的头节点，用于存储键值对。负载因子的计算公式为 $\frac{size}{capacity}$，当超过阈值时，哈希表会自动扩容以维持性能。

4.2 3.2 定义键值对节点类

为了在链表中存储键值对，我们需要定义一个节点类（例如 Node）。这个类包含三个属性：key 存储键，value 存储值，next 指向下一个节点的指针。在 Python 中，我们可以用一个简单的类来实现。

```
class Node:
    def __init__(self, key, value):
        self.key = key
        self.value = value
        self.next = None
```

这段代码定义了一个 Node 类，初始化方法 __init__ 接收 key 和 value 参数，并将 next 指针设为 None，表示链表末尾。每个节点都封装了一个键值对，并通过 next 指针连接成链表，从而支持链地址法的冲突解决。

4.3 3.3 实现核心方法

接下来，我们实现哈希表的核心方法，包括初始化、哈希函数、插入、查找、删除和扩容。

首先，定义 MyHashMap 类的构造函数 __init__。它初始化哈希表的容量、负载因子和桶数组。默认初始容量设为 16，每个桶初始化为空链表。

```
class MyHashMap:
    def __init__(self, initial_capacity=16, load_factor=0.75):
        self.capacity = initial_capacity
        self.load_factor = load_factor
        self.size = 0
        self.buckets = [None] * self.capacity
```

这段代码中，buckets 被初始化为一个长度为 capacity 的数组，每个元素为 None，表示空链表。size 记录当前元素数量，load_factor 用于后续扩容判断。

私有方法 _hash 负责计算键的哈希值。它使用 Python 内置的 hash() 函数，并对容量取模以确保索引在有效范围内。同时，处理负哈希值的情况。

```
    def _hash(self, key):
        if key is None:
            raise ValueError("Key cannot be None")
        return hash(key) % self.capacity
```

这里，hash(key) 生成一个整数哈希值，通过取模操作 % self.capacity 将其映射到 0 到 capacity-1 的范围内。如果键为 None，则抛出异常，避免无效输入。

put 方法用于插入或更新键值对。它先计算哈希索引，然后遍历对应链表：如果键存在，则更新值；否则，在链表末尾添加新节点。完成后，检查负载因子，必要时触发扩容。

```
    def put(self, key, value):
        index = self._hash(key)
        if self.buckets[index] is None:
            self.buckets[index] = Node(key, value)
        else:
            current = self.buckets[index]
            while current:
                if current.key == key:
                    current.value = value
                    return
                if current.next is None:
                    break
                current = current.next
            current.next = Node(key, value)
```

```
        self.size += 1
        if self.size / self.capacity > self.load_factor:
            self._resize()
```

在 put 方法中，如果桶为空，则直接创建新节点作为头节点；否则，遍历链表查找键。如果找到，更新值；如果未找到，在末尾添加新节点。最后，增加 size 并检查是否需要扩容，这有助于保持哈希表的高效性。

get 方法根据键查找值。它计算索引后遍历链表，返回匹配的值或 None。

```
    def get(self, key):
        index = self._hash(key)
        current = self.buckets[index]
        while current:
            if current.key == key:
                return current.value
            current = current.next
        return None
```

这段代码通过哈希索引定位桶，然后线性搜索链表。如果找到键，返回对应值；否则返回 None，表示键不存在。

remove 方法删除指定键的节点。它使用双指针技巧（prev 和 curr）遍历链表，便于删除操作。

```
    def remove(self, key):
        index = self._hash(key)
        current = self.buckets[index]
        prev = None
        while current:
            if current.key == key:
                if prev:
                    prev.next = current.next
                else:
                    self.buckets[index] = current.next
                self.size -= 1
                return
            prev = current
            current = current.next
```

在 remove 方法中，prev 指针记录前一个节点，curr 指向当前节点。如果找到键，则调整指针跳过该节点，实现删除。这确保了链表结构的正确性。

最后，_resize 方法负责动态扩容。当负载因子超标时，它创建一个新数组（容量翻倍），并将所有元素重新哈希到新数组中。

```
    def _resize(self):
        old_buckets = self.buckets
        self.capacity *= 2
```

```
        self.buckets = [None] * self.capacity
        self.size = 0
        for bucket in old_buckets:
            current = bucket
            while current:
                self.put(current.key, current.value)
                current = current.next
```

在 _resize 方法中，旧桶数组被保存，新数组容量翻倍。然后遍历旧数组中的每个节点，使用 put 方法重新插入到新数组中。这个过程称为「重哈希」，它通过减少链表长度来提升性能。

5 第四部分：复杂度分析与优化探讨

5.1 4.1 时间复杂度分析

在理想情况下，哈希表的插入、删除和查找操作的平均时间复杂度为 $O(1)$，这依赖于哈希函数的均匀分布性。然而，最坏情况下（例如所有键都冲突），性能可能退化为 $O(n)$，因为需要遍历长链表。因此，设计高质量的哈希函数至关重要。

5.2 4.2 空间复杂度

哈希表的空间复杂度为 $O(n+m)$，其中 n 是元素数量，m 是数组容量。这包括了存储元素和桶数组的空间。通过调整负载因子，可以在时间和空间之间取得平衡。

5.3 4.3 优化方向

为了进一步提升哈希表性能，可以考虑以下优化策略：选择合适的初始容量和负载因子，以减少扩容频率；设计更均匀的哈希函数，例如使用多项式滚动哈希；优化动态扩容策略，例如逐步扩容；以及在链表过长时转换为更高效的数据结构（如红黑树），正如 JDK 8 中的 HashMap 实现。

本文详细介绍了哈希表的核心思想，包括哈希函数的作用和冲突解决机制，并以链地址法为例实现了基本的哈希表。通过代码实现和复杂度分析，我们看到了哈希表在平均 $O(1)$ 时间复杂度下的高效性。哈希表在实际应用中广泛存在，例如 Python 的 dict、Java 的 HashMap，以及数据库索引和缓存系统。对于进阶学习，建议读者阅读标准库源码，了解并发哈希表（如 ConcurrentHashMap）和其他哈希算法，以深化对数据结构的理解。