

# 哈希表数据结构

黄京

Nov 01, 2025

在日常生活中，我们经常需要快速查找信息，例如在字典中查询单词、在电话簿中寻找号码，或者从缓存中获取数据。这些场景都要求一种能实现快速查找、插入和删除操作的数据结构。如果使用数组，虽然插入操作快速，但查找需要遍历整个数组，时间复杂度为  $O(n)$ ；链表同样存在查找效率低下的问题。因此，我们迫切需要一种能接近  $O(1)$  时间复杂度的数据结构。哈希表就是这样一种“魔法”数据结构，它通过巧妙的设计，将平均时间复杂度降至  $O(1)$ 。本文的目标是彻底解析哈希表的工作原理，掌握处理哈希冲突的核心方法，从头开始用代码实现一个功能完整的哈希表，并对其性能进行分析和优化。

## 1 哈希表的核心思想 —— 为何它能如此之快？

哈希表的快速性能源于数组的随机访问能力。数组通过下标索引，可以在  $O(1)$  时间内访问任何元素，这为哈希表的高效性奠定了基石。然而，问题在于我们如何将任意类型的键，例如字符串或对象，转换为一个数组下标？这就需要引入哈希函数的概念。

哈希函数是一座从“键”到“地址”的魔法桥梁。它接受一个键作为输入，返回一个整数哈希值，然后将这个整数映射到数组的固定范围内，即  $index = \text{hash}(key) \bmod \text{array\_size}$ 。一个理想的哈希函数应具有确定性，即相同的键必须始终产生相同的哈希值；高效性，计算速度要快；以及均匀性，哈希值应尽可能均匀分布，以减少冲突。例如，假设我们有一个存储员工信息的哈希表，键是员工 ID。如果哈希函数计算  $\text{hash}(101)$  返回 1， $\text{hash}(102)$  返回 2，那么我们可以直接将数据存入数组的对应位置，实现近乎即时的访问。

## 2 无法避免的挑战 —— 哈希冲突

哈希冲突是指两个不同的键经过哈希函数计算后，得到了相同的数组索引。例如， $\text{hash}(\text{John})$  和  $\text{hash}(\text{Jane})$  都计算出索引 5。由于键空间远大于数组空间，冲突是必然发生的。因此，一个好的哈希表设计不在于避免冲突，而在于高效地解决冲突。解决哈希冲突有两种主流方法：链地址法和开放地址法。

链地址法的思想是让数组的每个位置都指向一个链表，所有哈希到同一索引的键值对都存储在这个链表中。操作时，插入需要计算索引并将键值对添加到对应链表的末尾；查找需要遍历链表比对键是否相等；删除则需要找到并移除对应节点。链地址法的优点是简单有效，链表可以无限扩展，适合不知道数据量的情况。开放地址法则是在发生冲突时，按照某种探测序列在数组中寻找下一个空闲的位置。常见的探测方法包括线性探测，即  $index = (\text{original\_index} + i) \bmod \text{size}$ ，其中  $i$  从 1 开始递增；二次探测，即  $index = (\text{original\_index} + i^2) \bmod \text{size}$ ；以及双重哈希，使用第二个哈希函数来计算步长。开放地址法的优点是所有数据都存储在数组中，缓存友好，但删除操作复杂，需要标记删除，且容易产生聚集现象。

### 3 动手实现 —— 构建我们自己的哈希表（采用链地址法）

我们将使用链地址法来实现一个基本的哈希表。首先，设计数据结构。定义键值对节点类 `HashNode`，它包含 `key`、`value` 和 `next` 指针，用于构建链表结构。然后定义哈希表类 `MyHashMap`，核心字段包括一个存储链表的数组 `table`、当前键值对数量 `size`、数组容量 `capacity` 和负载因子 `loadFactor`。负载因子定义为  $loadFactor = size / capacity$ ，用于在后续操作中触发动态扩容。

接下来实现核心方法。构造函数初始化一个固定容量的数组，例如 16，并设置默认负载因子为 0.75。哈希函数 `_hash` 是一个私有方法，对于整数键，直接取模，例如 `return key % capacity`；对于字符串键，使用多项式哈希码并取模以确保均匀性，例如通过遍历字符串字符计算哈希值： $hash = s[0] \times 31^{(n-1)} + s[1] \times 31^{(n-2)} + \dots + s[n - 1]$ ，其中  $n$  是字符串长度，然后对 `capacity` 取模。

`put` 方法用于插入键值对。首先计算索引 `index = _hash(key)`，然后遍历 `table[index]` 对应的链表。如果找到相同的 `key`，则更新其 `value`；否则，在链表头部插入新节点，以保持  $O(1)$  的插入时间。插入后增加 `size`，并检查负载因子是否超过阈值（例如 0.75），如果超过，则调用 `_resize` 方法进行扩容。这段代码的关键在于处理链表遍历和节点插入，确保在冲突时正确维护数据结构。

`get` 方法用于查找键对应的值。计算索引后，遍历链表，查找并返回对应 `key` 的 `value`，若未找到则返回 `null`。这体现了链地址法的查找逻辑，依赖于链表的线性搜索，但在平均情况下，链表长度短，性能接近  $O(1)$ 。

`remove` 方法用于删除键值对。计算索引，遍历链表找到节点并删除，同时减少 `size`。删除操作需要小心处理链表指针，例如如果删除的是头节点，需要更新数组引用；否则，调整前驱节点的 `next` 指针。

动态扩容是保证性能的关键。当元素过多时，链表会变长，性能从  $O(1)$  退化为  $O(n)$ 。`_resize` 方法创建一个新数组，容量通常是原容量的两倍，然后遍历旧表中的每一个节点，根据新的容量重新哈希所有键，并将它们放入新数组的正确位置。最后将 `table` 引用指向新数组。尽管扩容是一个耗时操作，时间复杂度为  $O(n)$ ，但通过摊还分析，其平均成本依然是  $O(1)$ ，这确保了哈希表在长期运行中的高效性。

### 4 分析与优化

哈希表的时间复杂度在最佳情况下，当哈希函数均匀且无冲突时，所有操作均为  $O(1)$ 。平均情况下，在合理负载因子下，通过链表平均长度分析，操作依然是  $O(1)$ 。最坏情况下，所有键都冲突，退化为一个链表，操作  $O(n)$ 。因此，设计一个好的哈希函数至关重要，例如 Java 中 `String.hashCode()` 的实现使用多项式哈希码，充分利用键的所有信息，让结果的每一位都影响最终哈希值，以确保均匀分布。

进阶优化思路包括将链表转换为红黑树，当链表过长时，将查找性能从  $O(n)$  提升至  $O(\log n)$ ，如 Java 8+ 的 `HashMap` 所做的那样。此外，选择优质的初始容量和负载因子也能优化性能，例如根据预期数据量设置初始容量，避免频繁扩容。

本文全面回顾了哈希表的核心思想：通过哈希函数和数组实现快速访问，使用链地址法或开放地址法解决冲突，并通过动态扩容维持性能。哈希表广泛应用于数据库索引、缓存如 Redis、集合以及对象表示等场景。鼓励读者动手实现一遍，并尝试不同的哈希函数或冲突解决策略，以加深理解。下期可能介绍更高级的数据结构，如 `ConcurrentHashMap`，或深入探讨红黑树的原理与应用。

附录：完整代码实现可参考 GitHub 仓库（例如，使用 Java 编写），其中包含了上述所有方法的详细实现和测试用例。