哈希表的重新哈希与C++标准库中的哈希表实现

1 哈希表的重新哈希（Rehashing）

如果哈希表变得太满，操作的运行时间将开始变得太长，插入可能会因为二次探测开放寻址哈希表而失败。这可能会发生在插入和删除太多混合的情况下。一个解决方案是构建另一个大约是原来两倍大的表（以及一个相关的新哈希函数），然后扫描整个原始哈希表，为每个（未被删除的）元素计算新的哈希值，并将其插入到新表中。

作为一个例子，假设元素13、15、24和6被插入到一个使用线性探测的哈希表中。随着时间的推移，表变得太满，我们需要进行重新哈希。我们将构建一个更大的新表，并将所有元素重新哈希到这个新表中。这个过程需要扫描整个原始哈希表，对于每个元素，我们使用新的哈希函数计算其在新表中的位置，并将元素插入到那里。

这个过程可以确保哈希表在插入和删除操作后仍然保持较高的性能。重新哈希是一个昂贵的操作，因为它需要遍历整个哈希表，但对于保持哈希表性能来说这是必要的。在实际应用中，我们通常会监控哈希表的负载因子，并在它超过某个阈值时进行重新哈希，以避免性能下降。

2 C++标准库中的哈希表实现

在C++11标准中引入了unordered_map和unordered_set这两个类模板，它们分别提供了基于哈希表的map和set的实现。这些容器提供了快速的插入、删除和搜索操作，这些操作的平均时间复杂度为O(1)。这些容器的实现通常使用哈希表，但具体的实现细节对用户是透明的。

2.1 unordered_map和unordered_set

unordered_map和unordered_set容器使用哈希表来存储元素，但它们并不保证元素的顺序。这些容器的关键在于哈希函数的设计和冲突解决策略。在C++标准库中，这些容器的哈希函数通常是可配置的，用户可以提供自定义的哈希函数来满足特定的需求。

2.2 性能考虑

虽然unordered_map和unordered_set提供了快速的操作，但是它们在最坏情况下的性能可能会下降到O(n)，这通常发生在哈希函数不能均匀分布元素时。因此，选择合适的哈希函数对于这些容器的性能至关重要。

2.3 使用示例

使用unordered_map和unordered_set非常简单，用户可以像使用其他标准库容器一样使用它们。例如，unordered_map可以用于存储键值对，而unordered_set可以用于存储唯一的元素。这些容器提供了插入、删除和查找元素的操作，以及一些其他的实用功能，如size()和empty()

2.4 注意事项

使用unordered_map和unordered_set时需要注意的是，由于它们基于哈希表，所以对元素的哈希值的计算可能会比较昂贵，特别是对于复杂的数据类型。此外，这些容器不保证元素的顺序，如果需要有序的元素，应该使用map和set。

总的来说，unordered_map和unordered_set为C++提供了一种高效的键值存储和集合操作的方式，但用户需要注意选择合适的哈希函数，并考虑到它们在最坏情况下的性能。