哈希表入门&STL容器：unordered_map

1.哈希函数与哈希表

哈希表是什么?

`std::unordered_map` 是基于哈希表（hash table）实现的。哈希表是一种数据结构，用来高效地存储和查找数据。

哈希表的基本构成

1. **哈希函数（Hash Function）**：
   - 哈希函数是一个把键（key）转换成整数（哈希值，hash value）的函数。这个整数决定了元素存储在哪个“桶”（bucket）里。
   - 举个例子，假设我们有一个键 `42`，哈希函数可能会把它转换成整数 `5`，那么这个键值对就会存储在第 `5` 个桶里。
2. **桶（Bucket）**：
   - 桶是哈希表中的一个位置，用来存储键值对。
   - 哈希表内部有一个桶数组（bucket array），每个桶可以存储一个或多个键值对。

哈希表如何工作?

1. **插入元素**：
   - 当你插入一个键值对时，哈希函数会计算键的哈希值，然后决定把这个键值对放到哪个桶里。
   - 如果那个桶已经有其他元素了（即发生了哈希冲突），新的键值对会被添加到这个桶的链表（linked list）中。
2. **查找元素**：
   - 当你查找一个键时，哈希函数会计算这个键的哈希值，然后找到对应的桶。
   - 在桶里，哈希表会遍历链表，找到对应的键值对。
3. **删除元素**：
   - 删除操作也是类似的，先找到对应的桶，然后在桶的链表中找到并删除指定的键值对。

负载因子和哈希冲突

1. **负载因子（Load Factor）**：
   - 负载因子是哈希表中元素数量与桶数量的比值。
   - 负载因子过高（即元素数量远多于桶数量）会导致更多的哈希冲突，从而降低性能。
   - 当负载因子超过某个阈值时，哈希表会自动扩展桶数组（rehash），以保持较低的负载因子。
2. **哈希冲突（Hash Collision）**：
   - 当多个键的哈希值相同时，哈希冲突就发生了。
   - `std::unordered_map` 使用链地址法（separate chaining）来处理冲突，即每个桶存储一个链表，所有映射到该桶的元素都存储在这个链表中。

举个例子

假设我们有一个 `std::unordered_map<int, std::string>`，并且插入了以下键值对：

```
myMap[1] = "one";
myMap[2] = "two";
myMap[3] = "three";
```

1. **插入元素**：
   - 假设哈希函数把键 `1` 转换成哈希值 `1`，键 `2` 转换成哈希值 `2`，键 `3` 转换成哈希值 `0`。
   - 结果是：
     - 键 `1` 和值 `"one"` 存储在桶 `1`。
     - 键 `2` 和值 `"two"` 存储在桶 `2`。
     - 键 `3` 和值 `"three"` 存储在桶 `0`。
2. **查找元素**：
   - 查找键 `2` 时，哈希函数计算出哈希值 `2`，然后哈希表会在桶 `2` 中找到键 `2` 对应的值 `"two"`。
3. **删除元素**：
   - 删除键 `1` 时，哈希函数计算出哈希值 `1`，然后哈希表会在桶 `1` 中找到并删除键 `1` 对应的键值对。

主要操作的时间复杂度

- **查找**：平均 O(1)，最坏 O(n)（当所有元素都映射到同一个桶时）。
- **插入**：平均 O(1)，最坏 O(n)（当发生重新哈希时）。
- **删除**：平均 O(1)，最坏 O(n)（当所有元素都映射到同一个桶时）。

总结

- **哈希表** 是一种高效的数据结构，用来存储和查找数据。
- **哈希函数** 把键转换成哈希值，决定元素存储在哪个桶里。
- **桶** 存储键值对，可能包含一个链表来处理哈希冲突。
- **负载因子** 和 **哈希冲突** 是影响哈希表性能的重要因素。

2.以哈希表为基础的STL容器

在 C++ 标准库中，以哈希表为基础实现的容器主要包括以下几种：

1. `std::unordered_map`：
   - 用于存储键值对（key-value pairs）。
   - 提供快速的查找、插入和删除操作，平均时间复杂度为 O(1)。
   - 不保证元素的顺序。
2. `std::unordered_multimap`：
   - 类似于 `std::unordered_map`，但允许多个元素拥有相同的键（即键可以重复）。

- 适用于需要存储多个相同键的场景。
- 同样提供 O(1) 的快速操作，且不保证元素的顺序。

3. `std::unordered_set`：
   - 用于存储唯一的元素（没有键值对，只有键）。
   - 提供快速的查找、插入和删除操作，平均时间复杂度为 O(1)。
   - 不保证元素的顺序。
4. `std::unordered_multiset`：
   - 类似于 `std::unordered_set`，但允许存储多个相同的元素（即元素可以重复）。
   - 适用于需要存储重复元素的场景。
   - 同样提供 O(1) 的快速操作，且不保证元素的顺序。

简单示例

以下是每种容器的简单使用示例：

`std::unordered_map`

```
#include <iostream>
#include <unordered_map>
#include <string>

int main() {
    std::unordered_map<int, std::string> myMap;
    myMap[1] = "one";
    myMap[2] = "two";
    myMap[3] = "three";

    for (const auto& pair : myMap) {
        std::cout << "Key: " << pair.first << ", Value: " << pair.second <<
std::endl;
    }

    return 0;
}
```

`std::unordered_multimap`

```
#include <iostream>
#include <unordered_multimap>
#include <string>

int main() {
    std::unordered_multimap<int, std::string> myMultiMap;
    myMultiMap.insert({1, "one"});
    myMultiMap.insert({2, "two"});
    myMultiMap.insert({1, "uno"}); // 键 1 重复

    for (const auto& pair : myMultiMap) {
        std::cout << "Key: " << pair.first << ", Value: " << pair.second <<
std::endl;
    }
```

```
    return 0;
}
```

`std::unordered_set`

```
#include <iostream>
#include <unordered_set>

int main() {
    std::unordered_set<int> mySet;
    mySet.insert(1);
    mySet.insert(2);
    mySet.insert(3);

    for (const auto& elem : mySet) {
        std::cout << "Element: " << elem << std::endl;
    }

    return 0;
}
```

`std::unordered_multiset`

```
#include <iostream>
#include <unordered_multiset>

int main() {
    std::unordered_multiset<int> myMultiSet;
    myMultiSet.insert(1);
    myMultiSet.insert(2);
    myMultiSet.insert(1); // 元素 1 重复

    for (const auto& elem : myMultiSet) {
        std::cout << "Element: " << elem << std::endl;
    }

    return 0;
}
```

总结

- 以上四种容器都是基于哈希表实现的，提供 O(1) 的快速操作。
- `std::unordered_map` 和 `std::unordered_multimap` 用于存储键值对，区别在于是否允许重复键。
- `std::unordered_set` 和 `std::unordered_multiset` 用于存储唯一元素，区别在于是否允许重复元素。
- 这些容器**都不保证元素的顺序**（元素的顺序可以储存在数组等线性容器中），但在大多数情况下能提供高效的查找、插入和删除操作。

unordered_map相关的函数接口

常见成员函数

1. `operator[]`：访问或插入元素。
2. `at`：访问元素，不存在时抛出异常。
3. `insert`：插入元素。
4. `emplace`：原地构造并插入元素。
5. `erase`：删除元素。
6. `find`：查找元素。
7. `size`：返回容器中元素的数量。
8. `empty`：检查容器是否为空。
9. `clear`：清空容器。
10. `begin` **和** `end`：返回指向容器首元素和尾后元素的迭代器。

```
#include <iostream>
#include <unordered_map>
#include <string>

int main() {
    // 创建一个 unordered_map 容器
    std::unordered_map<int, std::string> myMap;

    // 使用 operator[] 插入元素
    myMap[1] = "one";
    myMap[2] = "two";

    // 使用 at 访问元素
    //try，catch是C++专门用于异常处理的关键词
    try {
        std::cout << "Element at key 1: " << myMap.at(1) << std::endl;
    } catch (const std::out_of_range& e) {
        std::cout << "Key not found!" << std::endl;
    }

    // 使用 insert 插入元素
    auto insertResult = myMap.insert({3, "three"});
    if (insertResult.second) {
        std::cout << "Insert succeeded for key 3." << std::endl;
    } else {
        std::cout << "Insert failed for key 3." << std::endl;
    }

    // 使用 emplace 插入元素
    auto emplaceResult = myMap.emplace(4, "four");
    if (emplaceResult.second) {
        std::cout << "Emplace succeeded for key 4." << std::endl;
    } else {
        std::cout << "Emplace failed for key 4." << std::endl;
    }
```

```
    // 使用 find 查找元素
    auto findResult = myMap.find(2);
    if (findResult != myMap.end()) {
        std::cout << "Found key 2 with value: " << findResult->second <<
std::endl;
    } else {
        std::cout << "Key 2 not found." << std::endl;
    }

    // 使用 erase 删除元素
    myMap.erase(2);
    std::cout << "Key 2 erased." << std::endl;

    // 使用 size 获取元素数量
    std::cout << "Size of map: " << myMap.size() << std::endl;

    // 使用 empty 检查容器是否为空
    if (myMap.empty()) {
        std::cout << "Map is empty." << std::endl;
    } else {
        std::cout << "Map is not empty." << std::endl;
    }

    // 使用 clear 清空容器
    myMap.clear();
    std::cout << "Map cleared." << std::endl;

    // 再次检查容器是否为空
    if (myMap.empty()) {
        std::cout << "Map is empty after clear." << std::endl;
    } else {
        std::cout << "Map is not empty after clear." << std::endl;
    }

    // 使用 begin 和 end 迭代访问元素
    myMap[5] = "five";
    myMap[6] = "six";
    std::cout << "Map contents:" << std::endl;
    for (auto it = myMap.begin(); it != myMap.end(); ++it) {
        std::cout << "Key: " << it->first << ", Value: " << it->second <<
std::endl;
    }

    return 0;
}
```

1. `operator[]`：用于插入或访问元素。如果键不存在，会插入一个默认值。
2. `at`：用于访问元素。如果键不存在，会抛出 `std::out_of_range` 异常。
3. `insert`：用于插入元素。如果键已存在，插入操作会失败。
4. `emplace`：用于原地构造并插入元素。如果键已存在，插入操作会失败。
5. `erase`：用于删除元素。

6. `find`：用于查找元素。如果找到，返回指向该元素的迭代器；否则，返回 `end` 迭代器。
7. `size`：返回容器中的元素数量。
8. `empty`：检查容器是否为空。
9. `clear`：清空容器，删除所有元素。
10. `begin` **和** `end`：返回指向容器首元素和尾后元素的迭代器，用于迭代访问容器中的元素。

深入：成员函数的返回值

构造函数和析构函数

1. **构造函数**
   - `unordered_map()`: 创建一个空的 `unordered_map`。
   - `unordered_map(size_type n)`: 创建一个空的 `unordered_map`，并预留 `n` 个桶。
   - `unordered_map(size_type n, const hasher& hf)`: 创建一个空的 `unordered_map`，预留 `n` 个桶，并使用 `hf` 作为哈希函数。
   - `unordered_map(size_type n, const hasher& hf, const key_equal& ke)`: 创建一个空的 `unordered_map`，预留 `n` 个桶，使用 `hf` 作为哈希函数，并使用 `ke` 作为键比较器。
   - `unordered_map(const unordered_map& um)`: 拷贝构造函数，创建一个与 `um` 相同的 `unordered_map`。
   - `unordered_map(unordered_map&& um) noexcept`: 移动构造函数，创建一个从 `um` 移动的 `unordered_map`。
2. **析构函数**
   - `~unordered_map()`: 销毁 `unordered_map`，释放所有资源。

迭代器

3. **begin()**
   - `iterator begin() noexcept;`
   - `const_iterator begin() const noexcept;`
   - 返回指向容器中第一个元素的迭代器。
4. **end()**
   - `iterator end() noexcept;`
   - `const_iterator end() const noexcept;`
   - 返回指向容器中最后一个元素之后的迭代器。

容量

5. **empty()**
   - `bool empty() const noexcept;`
   - 检查容器是否为空，若为空则返回 `true`，否则返回 `false`。
6. **size()**
   - `size_type size() const noexcept;`
   - 返回容器中元素的数量。

7. **max_size()**
    - `size_type max_size() const noexcept;`
    - 返回容器能够容纳的最大元素数量。

元素访问

8. **operator[]**
    - `mapped_type& operator[](const key_type& k);`
    - `mapped_type& operator[](key_type&& k);`
    - 访问指定键的元素，如果键不存在则插入一个新的元素。
9. **at()**
    - `mapped_type& at(const key_type& k);`
    - `const mapped_type& at(const key_type& k) const;`
    - 访问指定键的元素，如果键不存在则抛出 `std::out_of_range` 异常。

修改器

10. **insert()**
    - `pair<iterator, bool> insert(const value_type& val);`
    - `iterator insert(const_iterator hint, const value_type& val);`
    - 插入元素，如果元素已存在则不插入。
11. **erase()**
    - `iterator erase(const_iterator position);`
    - `size_type erase(const key_type& k);`
    - `iterator erase(const_iterator first, const_iterator last);`
    - 移除指定位置或键的元素。
12. **clear()**
    - `void clear() noexcept;`
    - 移除所有元素。

查找

13. **find()**
    - `iterator find(const key_type& k);`
    - `const_iterator find(const key_type& k) const;`
    - 查找指定键的元素，返回指向该元素的迭代器，如果找不到则返回 `end()`。
14. **count()**
    - `size_type count(const key_type& k) const;`
    - 返回指定键的元素数量（对于 `unordered_map`，结果要么是 0 要么是 1）。
15. **equal_range()**
    - `pair<iterator, iterator> equal_range(const key_type& k);`
    - `pair<const_iterator, const_iterator> equal_range(const key_type& k) const;`

- 返回一个范围，包含所有等于指定键的元素。

桶接口

16. **bucket_count()**
    - `size_type bucket_count() const noexcept;`
    - 返回桶的数量。
17. **max_bucket_count()**
    - `size_type max_bucket_count() const noexcept;`
    - 返回可以容纳的最大桶数量。
18. **bucket_size()**
    - `size_type bucket_size(size_type n) const;`
    - 返回指定桶中的元素数量。
19. **bucket()**
    - `size_type bucket(const key_type& k) const;`
    - 返回指定键的桶编号。

哈希策略

20. **load_factor()**
    - `float load_factor() const noexcept;`
    - 返回当前负载因子。
21. **max_load_factor()**
    - `float max_load_factor() const noexcept;`
    - `void max_load_factor(float ml);`
    - 返回或设置最大负载因子。
22. **rehash()**
    - `void rehash(size_type n);`
    - 重新组织桶，使得桶的数量至少为 `n`。
23. **reserve()**
    - `void reserve(size_type n);`
    - 预留空间，使得可以容纳至少 `n` 个元素而不需要重新哈希。

这些是 `std::unordered_map` 中常用的成员函数及其返回值的简要说明。通过这些函数，你可以方便地管理和操作哈希表数据结构。