std::flat_map 与 std::flat_multimap 的用法

褚一枫 2024013328 17612197488
zhuyf24@mails.tsinghua.edu.cn

2025 年 3 月 27 日

摘要

C++24 标准引入了两种新的容器：std::flat_map 与 std::flat_multimap，它们与传统容器 std::map 与 std::multimap 具有几乎完全相同的接口，但其底层实现的不同造成了性能上的差异。本文总结了 std::flat_map 与 std::flat_multimap 的相关用法，通过编写使用案例分析了使用时的注意事项，并通过设计实验与理论分析，对其性能进行了评估，从而分析了其优缺点与使用场景。

关键词：**C++24, std::flat_map, std::flat_multimap,** 性能分析

所有实验都在相同的硬件和编译器环境下运行，使用以下通用配置：

- 处理器：AMD Ryzen AI 9 HX 370
- 内存：32GB LPDDR5X
- 编译器：g++-15 (SUSE Linux) 15.0.1 20250317 (experimental)
- 操作系统：openSUSE Tumbleweed 20250316
- 编程语言：C++24
- 编译环境：Windows Subsystem for Linux (WSL)
- 编译参数：-O3 -std=c++23

参考文献电子版见 ref 文件夹，实验源代码见 src 文件夹。

目录

1 简介

C++ 作为一种广泛应用于系统开发、游戏引擎、高性能应用等领域的编程语言，其标准库的不断演进为开发者提供了越来越丰富的工具。C++24 标准作为最新的 C++ 标准，引入了多项重要更新，其中就包括两个新的关联容器：std::flat_map 与 std::flat_multimap。

在传统 C++ 中，关联容器 std::map 和 std::multimap 通常基于红黑树或其他平衡二叉搜索树实现，这种结构能够保证对数级别的查找、插入和删除时间复杂度。然而，红黑树结构内存分配次数多且数据在内存中分散存储，这可能导致缓存不友好，影响实际应用中的性能表现。

std::flat_map 和 std::flat_multimap 采用了不同的设计理念，它们使用两个平行数组（通常是 std::vector）分别存储键和值，一个用于所有键，另一个用于所有对应的值。这种扁平化的存储结构带来了几个显著特点：

- 内存连续性更好，提高了缓存友好性
- 在某些场景下可能提供更好的性能
- 提供了与传统 map 容器几乎相同的接口，便于开发者迁移现有代码

然而，这种实现方式在插入和删除操作时可能需要移动大量元素，从而导致性能下降。因此，了解这些新容器的行为特性、性能特点以及适用场景对于开发者来说至关重要。

本文将深入探讨 std::flat_map 与 std::flat_multimap 的基本用法、实现原理、性能特性以及适用场景。首先介绍其 API 及使用方式，然后通过具体案例展示其实际应用，进而通过实验和理论分析对比评估其性能特点，最后总结其优缺点并给出合适的使用场景建议。通过本文，读者将能够全面了解这两种新容器，并在实际开发中做出明智的容器选择。

2 相关用法

本节将详细介绍 std::flat_map 与 std::flat_multimap 的基本用法，包括头文件包含、构造方法、基本操作以及特有的成员函数等。

2.1 基本概念与头文件

std::flat_map 和 std::flat_multimap 定义在 C++23 标准库中，需要包含对应的头文件：

```
#include <flat_map>     // 对于std::flat_map
#include <flat_map>     // 对于std::flat_multimap（同一头文件）
```

这两个容器的基本概念如下：

- **std::flat_map**：不允许重复键的关联容器，使用两个连续存储的序列容器分别存储键和值
- **std::flat_multimap**：允许重复键的关联容器，底层实现与 flat_map 类似

2.2 模板参数

std::flat_map 与 std::flat_multimap 的模板参数定义如下：

```
template<
    class Key,                                  // 键类型
    class T,                                    // 值类型
    class Compare = std::less<Key>,             // 比较函数对象类型
    class KeyContainer = std::vector<Key>, // 存储键的容器类型
    class MappedContainer = std::vector<T> // 存储值的容器类型
> class flat_map;

template<
    class Key,
    class T,
    class Compare = std::less<Key>,
    class KeyContainer = std::vector<Key>,
```

```
    class MappedContainer = std::vector<T>
> class flat_multimap;
```

值得注意的是，与传统的 map 不同，这里允许用户指定用于存储键和值的底层容器类型，默认为 std::vector。

2.3 构造方法

flat_map 和 flat_multimap 提供了多种构造方法：

```
// 默认构造函数
flat_map();

// 使用比较器构造
explicit flat_map(const Compare& comp);

// 使用初始化列表构造
flat_map(std::initializer_list<std::pair<Key, T>> init);

// 从键值对范围构造
template<class InputIt>
flat_map(InputIt first, InputIt last);

// 从排序范围构造（sorted_unique_t标记表示输入已排序且唯
   一）
template<class InputIt>
flat_map(sorted_unique_t, InputIt first, InputIt last);

// 从键容器和值容器构造
flat_map(KeyContainer&& keys, MappedContainer&& maps);
```

flat_multimap 的构造方法与 flat_map 类似，但在接受已排序范围时使用的是 sorted_t 标记而非 sorted_unique_t。

2.4 基本操作

2.4.1 元素访问

```
// 访问指定键的元素，若不存在则插入默认值
T& operator[](const Key& key);
T& operator[](Key&& key);

// 访问指定键的元素，若不存在则抛出异常
T& at(const Key& key);
const T& at(const Key& key) const;
```

需要注意的是，std::flat_multimap 不提供 operator[] 操作，因为它允许存在重复键。

2.4.2 迭代器

```
// 返回指向容器第一个元素的迭代器
iterator begin() noexcept;
const_iterator begin() const noexcept;

// 返回指向容器尾部的迭代器
iterator end() noexcept;
const_iterator end() const noexcept;

// 返回反向迭代器
reverse_iterator rbegin() noexcept;
const_reverse_iterator rbegin() const noexcept;
reverse_iterator rend() noexcept;
const_reverse_iterator rend() const noexcept;
```

2.4.3 容量操作

```
// 检查容器是否为空
[[nodiscard]] bool empty() const noexcept;
```

```
// 返回容器中的元素数量
size_type size() const noexcept;

// 返回容器可容纳的最大元素数量
size_type max_size() const noexcept;
```

2.4.4 修改操作

```
// 清空容器
void clear() noexcept;

// 插入元素
std::pair<iterator, bool> insert(const value_type& value);
std::pair<iterator, bool> insert(value_type&& value);

// 原位构造元素
template<class... Args>
std::pair<iterator, bool> emplace(Args&&... args);

// 删除元素
iterator erase(const_iterator pos);
iterator erase(const_iterator first, const_iterator last);
size_type erase(const Key& key);

// 交换内容
void swap(flat_map& other) noexcept;
```

flat_multimap 的 insert 方法返回 iterator 而非 std::pair<iterator, bool>，因为它允许插入重复键。

2.4.5 查找操作

```
// 查找具有指定键的元素
```

```
iterator find(const Key& key);
const_iterator find(const Key& key) const;

// 检查容器是否包含具有特定键的元素
bool contains(const Key& key) const;

// 返回指定键的元素数量
size_type count(const Key& key) const;

// 返回第一个不小于给定键的元素的迭代器
iterator lower_bound(const Key& key);
const_iterator lower_bound(const Key& key) const;

// 返回第一个大于给定键的元素的迭代器
iterator upper_bound(const Key& key);
const_iterator upper_bound(const Key& key) const;

// 返回键值等于给定键的元素范围
std::pair<iterator, iterator> equal_range(const Key& key);
std::pair<const_iterator, const_iterator> equal_range(const
    Key& key) const;
```

2.5 特有操作

std::flat_map 和 std::flat_multimap 相比传统 map，还具有一些特有的操作：

```
// 获取键容器的引用
const key_container_type& keys() const noexcept;

// 获取值容器的引用
const mapped_container_type& values() const noexcept;

// 提取底层容器
```

```
std::pair<key_container_type, mapped_container_type>
    extract() &&;

// 替换底层容器
void replace(key_container_type&& key_cont,
    mapped_container_type&& mapped_cont);
```

这些特有操作反映了 flat_map 和 flat_multimap 的实现特点，允许用户直接操作底层容器，或执行特定的排序和去重操作。

2.6 自定义比较器

与 std::map 类似，std::flat_map 和 std::flat_multimap 也支持自定义比较器：

```
struct CustomCompare {
    bool operator()(const Key& a, const Key& b) const {
        // 自定义比较逻辑
        return a < b;
    }
};

std::flat_map<Key, Value, CustomCompare> myMap;
```

通过指定自定义比较器，用户可以控制元素在容器中的排序方式。

3 使用案例

本节将通过具体的代码示例展示 std::flat_map 和 std::flat_multimap 的实际应用，以及使用时的注意事项。

3.1 基本使用示例

以下是一个简单的 std::flat_map 使用示例，展示了其基本操作：

```
#include <flat_map>
```

```cpp
#include <iostream>
#include <string>

int main() {
    // 创建并初始化flat_map
    std::flat_map<int, std::string> studentIds = {
        {1001, "张三"},
        {1002, "李四"},
        {1003, "王五"}
    };

    // 使用operator[]添加新元素
    studentIds[1004] = "赵六";

    // 使用emplace添加新元素
    studentIds.emplace(1005, "钱七");

    // 访问元素
    std::cout << "学号1003对应学生: " << studentIds[1003]
       << std::endl;

    // 检查键是否存在
    if (studentIds.contains(1006)) {
        std::cout << "学号1006存在" << std::endl;
    } else {
        std::cout << "学号1006不存在" << std::endl;
    }

    // 遍历所有元素
    std::cout << "所有学生信息:" << std::endl;
    for (const auto& [id, name] : studentIds) {
        std::cout << "学号: " << id << ", 姓名: " << name
           << std::endl;
```

```
    }

    return 0;
}
```

运行结果如下：

```
学号1003对应学生： 王五
学号1006不存在
所有学生信息：
学号： 1001， 姓名： 张三
学号： 1002， 姓名： 李四
学号： 1003， 姓名： 王五
学号： 1004， 姓名： 赵六
学号： 1005， 姓名： 钱七
```

3.2 使用自定义键类型

以下示例展示了如何使用自定义类型作为键，以及如何提供自定义比较器：

```
#include <flat_map>
#include <iostream>
#include <string>

// 自定义坐标类型
struct Point {
    int x, y;

    // 便于输出的友元函数
    friend std::ostream& operator<<(std::ostream& os, const
        Point& p) {
        return os << "(" << p.x << ", " << p.y << ")";
    }
};
```

```cpp
// 自定义比较器，按点的曼哈顿距离排序
struct PointCompare {
    bool operator()(const Point& a, const Point& b) const {
        // 计算到原点的曼哈顿距离
        int distA = std::abs(a.x) + std::abs(a.y);
        int distB = std::abs(b.x) + std::abs(b.y);

        if (distA == distB) {
            // 距离相等时先比较x再比较y
            return (a.x == b.x) ? (a.y < b.y) : (a.x < b.x)
               ;
        }
        return distA < distB;
    }
};

int main() {
    // 使用自定义键类型和比较器
    std::flat_map<Point, std::string, PointCompare>
       locations;

    // 添加一些位置
    locations.insert({{3, 4}, "传感器A"});
    locations.insert({{1, 1}, "传感器B"});
    locations.insert({{5, 0}, "传感器C"});
    locations.insert({{-2, 3}, "传感器D"});

    // 遍历元素（将按自定义比较器排序）
    std::cout << "按到原点距离排序的位置:" << std::endl;
    for (const auto& [pos, name] : locations) {
        std::cout << name << " 位置: " << pos << std::endl;
    }
```

```
    return 0;
}
```

运行结果如下：

```
按到原点距离排序的位置:
传感器B 位置: (1, 1)
传感器D 位置: (-2, 3)
传感器C 位置: (5, 0)
传感器A 位置: (3, 4)
```

3.3 利用特有操作

以下示例展示了 flat_map 特有的操作，如直接访问底层容器功能：

```
#include <flat_map>
#include <iostream>
#include <vector>

int main() {
    // 创建flat_map
    std::flat_map<int, double> dataPoints = {
        {5, 10.5}, {3, 7.2}, {8, 12.3}, {1, 5.1}
    };

    // 直接访问键容器和值容器
    std::cout << "键容器内容:" << std::endl;
    for (const auto& key : dataPoints.keys()) {
        std::cout << key << " ";
    }
    std::cout << std::endl;

    std::cout << "值容器内容:" << std::endl;
    for (const auto& value : dataPoints.values()) {
```

```
        std::cout << value << " ";
    }
    std::cout << std::endl;

    // 提取底层容器
    auto [keys, values] = std::move(dataPoints).extract();

    // 手动创建新的容器
    std::vector<int> newKeys = {10, 20, 30};
    std::vector<double> newValues = {100.1, 200.2, 300.3};

    // 使用新容器创建flat_map
    std::flat_map<int, double> newDataPoints(std::move(
       newKeys), std::move(newValues));

    // 在插入无序元素后使用手动排序
    std::flat_map<int, std::string> events;
    events.emplace(50, "中点事件");
    events.emplace(10, "起始事件");
    events.emplace(90, "结束事件");
    events.emplace(30, "早期事件");
    events.emplace(70, "后期事件");

    // 遍历排序后的元素
    std::cout << "排序后的事件:" << std::endl;
    for (const auto& [time, desc] : events) {
        std::cout << "时间: " << time << ", 描述: " << desc
            << std::endl;
    }

    return 0;
}
```

运行结果如下：

```
键容器内容:
1 3 5 8
值容器内容:
5.1 7.2 10.5 12.3
排序后的事件:
时间: 10, 描述: 起始事件
时间: 30, 描述: 早期事件
时间: 50, 描述: 中点事件
时间: 70, 描述: 后期事件
时间: 90, 描述: 结束事件
```

3.4 大批量数据的预分配

当预先知道容器大小时，可以利用底层容器的预分配能力提高性能：

```
#include <flat_map>
#include <chrono>
#include <iostream>
#include <vector>
#include <random>

int main() {
    constexpr int DATA_SIZE = 10000;

    // 创建随机数生成器
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> keyDist(1, DATA_SIZE *
       10);

    // 准备数据
    std::vector<int> keys;
    std::vector<double> values;
```

```
    keys.reserve(DATA_SIZE);    // 提前分配空间
    values.reserve(DATA_SIZE);  // 提前分配空间

    // 生成随机数据
    for (int i = 0; i < DATA_SIZE; ++i) {
        keys.push_back(keyDist(gen));
        values.push_back(static_cast<double>(i) / 10.0);
    }

    // 使用预分配的容器构造flat_map
    auto start = std::chrono::high_resolution_clock::now();
    std::flat_map<int, double> dataMap(std::move(keys), std
       ::move(values));
    auto end = std::chrono::high_resolution_clock::now();

    std::chrono::duration<double, std::milli> duration =
       end - start;
    std::cout << "构造并排序包含 " << dataMap.size()
              << " 个元素的flat_map耗时: " << duration.
                 count()
              << " 毫秒" << std::endl;

    return 0;
}
```

运行结果如下：

```
构造并排序包含 9537 个元素的flat_map耗时: 0.408881 毫秒
```

3.5 使用 flat_multimap 管理重复键

以下示例展示了如何使用 flat_multimap 处理具有重复键的数据：

```
#include <flat_map>
#include <iostream>
```

```cpp
#include <string>

int main() {
    // 创建flat_multimap存储学生课程信息
    std::flat_multimap<std::string, std::string>
       studentCourses;

    // 插入数据（每个学生可以选多门课程）
    studentCourses.insert({"李明", "数学"});
    studentCourses.insert({"张华", "物理"});
    studentCourses.insert({"李明", "英语"});  // 李明的第二
       门课
    studentCourses.insert({"王芳", "化学"});
    studentCourses.insert({"张华", "生物"});  // 张华的第二
       门课
    studentCourses.insert({"李明", "历史"});  // 李明的第三
       门课

    // 按学生分组打印选课情况
    std::string currentStudent;
    std::cout << "学生选课情况：" << std::endl;

    for (const auto& [student, course] : studentCourses) {
        if (student != currentStudent) {
            currentStudent = student;
            std::cout << "\n" << student << " 选修的课程："
               ;
        }
        std::cout << " " << course;
    }
    std::cout << std::endl;

    // 查找特定学生的所有课程
```

```
    std::string targetStudent = "李明";
    auto [begin, end] = studentCourses.equal_range(
        targetStudent);

    std::cout << "\n使用equal_range查找 " << targetStudent
        << " 的所有课程：";
    for (auto it = begin; it != end; ++it) {
        std::cout << " " << it->second;
    }
    std::cout << std::endl;

    // 计算特定学生的课程数量
    std::cout << targetStudent << " 总共选修了 "
              << studentCourses.count(targetStudent) << "
                 门课程" << std::endl;

    return 0;
}
```

运行结果如下：

```
学生选课情况：

张华 选修的课程： 物理 生物
李明 选修的课程： 数学 英语 历史
王芳 选修的课程： 化学

使用equal_range查找 李明 的所有课程： 数学 英语 历史
李明 总共选修了 3 门课程
```

3.6 使用注意事项

在使用 std::flat_map 和 std::flat_multimap 时，需要注意以下几点：

- 插入性能：由于底层使用连续存储，在容器中间插入元素会导致后续

元素移动，对于频繁插入删除的场景，传统的 std::map 可能更适合

- 迭代器失效：任何修改容器的操作都可能导致迭代器失效，比传统 map 更需要注意
- 空间效率：对于频繁增删的场景可能导致内存碎片
- 预分配优化：对于已知大小的数据集，应考虑预先分配足够的空间
- 与排序算法配合：可以考虑先收集所有键值对再一次性构建容器，而不是逐个插入

通过了解这些使用注意事项，开发者可以更合理地选择使用 flat_map 或 flat_multimap 的场景，并在具体实现中作出合适的优化决策。

4 性能分析

本节将从理论和实践两个方面对 std::flat_map 与 std::flat_multimap 的性能进行分析，并与传统的 std::map 和 std::multimap 进行对比。

4.1 理论复杂度分析

首先，从底层实现角度分析 std::map/multimap 与 std::flat_map/flat_multimap 的根本差异：

- **std::map/multimap**: 通常基于红黑树（自平衡二叉搜索树）实现，每个键值对存储在独立的树节点中
- **std::flat_map/flat_multimap**: 使用两个平行数组（通常是 std::vector）分别存储键和值，保持有序状态

这种根本实现差异导致了它们在各种操作上的时间复杂度差异：

操作	**std::map/multimap**	**std::flat_map/flat_multimap**	底层原理
查找 (find, contains)	$O(\log n)$	$O(\log n)$	树的遍历 vs 二分查找
插入 (insert)	$O(\log n)$	$O(n)$	树的重平衡 vs 数组元素移动
删除 (erase)	$O(\log n)$	$O(n)$	树的重平衡 vs 数组元素移动
遍历 (iteration)	$O(n)$	$O(n)$	节点遍历 vs 连续内存访问

详细分析各操作的底层实现差异：

查找操作　虽然两种容器的查找操作理论复杂度都是 $O(\log n)$，但实现方式不同：

- std::map 通过遍历树节点进行查找，每次比较后向左或向右子树移动
- std::flat_map 在有序数组上使用二分查找算法
- 虽然复杂度相同，但 flat_map 的连续内存布局带来更好的缓存局部性，实际性能通常更优

插入操作　两种容器在插入操作上的复杂度差异显著：

- std::map 需要找到正确位置 $(O(\log n))$ 并可能执行树的重平衡 $(O(\log n))$，总体复杂度保持为 $O(\log n)$
- std::flat_map 需要找到正确位置 $(O(\log n))$，然后可能需要移动后续元素以维持有序性 $(O(n))$，总体复杂度为 $O(n)$

删除操作　删除操作也存在类似的复杂度差异：

- std::map 删除节点后可能需要重平衡树结构，但总体保持 $O(\log n)$ 复杂度
- std::flat_map 删除元素后需要移动后续元素填补空缺，这是一个 $O(n)$ 操作

遍历操作　虽然理论复杂度都是 $O(n)$，但性能特性差异明显：

- std::map 的遍历涉及在内存中分散的节点间跳转，缓存命中率较低
- std::flat_map 遍历连续内存区域，缓存友好性显著更高，实际性能通常优于 map

从理论复杂度分析可以看出：

- 两种容器的查找操作都是对数复杂度，但 flat_map 由于数据连续存储，缓存命中率更高
- flat_map 的插入和删除操作在最坏情况下需要线性时间，因为可能需要移动大量元素
- 遍历操作两者都是线性复杂度，但 flat_map 由于数据连续存储，通常会更快

4.2 实验设计

为了验证理论分析并获取更直观的性能对比，我们设计了以下实验：

- 实验 1：查找性能测试 - 测量在不同大小的容器中执行随机查找操作的平均时间
- 实验 2：插入性能测试 - 测量向已有容器中插入新元素的性能
- 实验 3：批量构造测试 - 测量一次性构造大容量容器的性能
- 实验 4：遍历性能测试 - 测量遍历容器中所有元素的时间

测试代码示例如下：

```
#include <chrono>
#include <iostream>
#include <map>
#include <flat_map>
#include <random>
#include <vector>
#include <iomanip>
#include <string>
#include <memory>

// 计时帮助函数
template<typename Func>
auto measure_time(Func&& func) {
    auto start = std::chrono::high_resolution_clock::now();
    func();
    auto end = std::chrono::high_resolution_clock::now();
    return std::chrono::duration<double, std::milli>(end -
       start).count();
}

template<typename Map, typename MultiMap>
void benchmark_containers(size_t size, size_t
   operation_count) {
```

```
std::cout << "\n
   ==========================================\n";
std::cout << "容器大小: " << size << ", 操作次数: " <<
   operation_count << std::endl;
std::cout << "
   ==========================================\n";

// 准备随机数据
std::random_device rd;
std::mt19937 gen(rd());
std::uniform_int_distribution<> dist(1, size * 10);

// 创建测试用的键值对
std::vector<std::pair<int, int>> data;
data.reserve(size);
for (size_t i = 0; i < size; ++i) {
    data.emplace_back(dist(gen), i);
}

// 准备查找和插入用的键
std::vector<int> testKeys;
testKeys.reserve(operation_count);
for (size_t i = 0; i < operation_count; ++i) {
    testKeys.push_back(dist(gen));
}

// 测试结果结构
struct TestResults {
    double lookup_time = 0;
    double insert_time = 0;
    double construction_time = 0;
    double iteration_time = 0;
    size_t memory_usage = 0;
```

```
};

TestResults map_results, flat_map_results;

std::cout << "测试标准 map/multimap...\n";

// 批量构造测试 - map
map_results.construction_time = measure_time([&]() {
    Map regularMap(data.begin(), data.end());
});

// 创建测试实例
Map regularMap;

// 插入测试 - map
map_results.insert_time = measure_time([&]() {
    for (size_t i = 0; i < operation_count; ++i) {
        regularMap.insert({testKeys[i], i});
    }
});

// 查找测试 - map
size_t found = 0;
map_results.lookup_time = measure_time([&]() {
    for (const auto& key : testKeys) {
        if (regularMap.find(key) != regularMap.end()) {
            ++found;
        }
    }
});

// 遍历测试 - map
volatile int sum = 0;
```

```
map_results.iteration_time = measure_time([&]() {
    for (const auto& [key, value] : regularMap) {
        sum += value;
    }
});

std::cout << "测试 flat_map/flat_multimap...\n";

// 批量构造测试 - flat_map
flat_map_results.construction_time = measure_time([&]()
    {
    MultiMap flatMap(data.begin(), data.end());
});

// 创建测试实例
MultiMap flatMap;

// 插入测试 - flat_map
flat_map_results.insert_time = measure_time([&]() {
    for (size_t i = 0; i < operation_count; ++i) {
        flatMap.insert({testKeys[i], i});
    }
});

// 查找测试 - flat_map
found = 0;
flat_map_results.lookup_time = measure_time([&]() {
    for (const auto& key : testKeys) {
        if (flatMap.find(key) != flatMap.end()) {
            ++found;
        }
    }
});
```

```
// 遍历测试 - flat_map
sum = 0;
flat_map_results.iteration_time = measure_time([&]() {
    for (const auto& [key, value] : flatMap) {
        sum += value;
    }
});

// 输出结果
std::cout << "----------- 性能对比 -----------\n";
std::cout << std::fixed << std::setprecision(2);
std::cout << std::left << std::setw(15) << "操作"
        << std::setw(18) << "标准容器 (ms)"
        << std::setw(18) << "flat容器 (ms)"
        << std::setw(18) << "性能比" << std::endl;

std::cout << std::left << std::setw(15) << "查找"
        << std::setw(15) << map_results.lookup_time
        << std::setw(15) << flat_map_results.
           lookup_time
        << std::setw(15) << map_results.lookup_time /
           flat_map_results.lookup_time << std::endl;

std::cout << std::left << std::setw(15) << "插入"
        << std::setw(15) << map_results.insert_time
        << std::setw(15) << flat_map_results.
           insert_time
        << std::setw(15) << map_results.insert_time /
           flat_map_results.insert_time << std::endl;

std::cout << std::left << std::setw(17) << "批量构造"
        << std::setw(15) << map_results.
```

```
                construction_time
            << std::setw(15) << flat_map_results.
                construction_time
            << std::setw(15) << map_results.
                construction_time / flat_map_results.
                construction_time << std::endl;

    std::cout << std::left << std::setw(15) << "遍历"
            << std::setw(15) << map_results.iteration_time
            << std::setw(15) << flat_map_results.
                iteration_time
            << std::setw(15) << map_results.iteration_time
                / flat_map_results.iteration_time << std::
                endl;
}

int main() {
    // 为了测试的完整性，测试不同大小的容器
    // 但较小的数据量以快速获得结果
    std::vector<size_t> sizes = {10000, 100000, 1000000};

    // 调整操作次数以平衡测试时间
    std::vector<size_t> operations = {10000, 100000,
        1000000};

    for (size_t i = 0; i < sizes.size(); ++i) {
        size_t size = sizes[i];
        size_t ops = operations[i];

        std::cout << "\n===== Map vs Flat_map =====\n";
        benchmark_containers<std::map<int, int>, std::
            flat_map<int, int>>(size, ops);
```

```
        std::cout << "\n===== Multimap vs Flat_multimap
            =====\n";
        benchmark_containers<std::multimap<int, int>, std::
            flat_multimap<int, int>>(size, ops);
    }

    return 0;
}
```

运行结果如下：

```
===== Map vs Flat_map =====

================================================
容器大小: 10000, 操作次数: 10000
================================================
测试标准 map/multimap...
测试 flat_map/flat_multimap...
----------- 性能对比 -----------
操作          标准容器 (ms) flat容器 (ms)    性能比
查找          0.62           0.53           1.17
插入          0.63           1.63           0.39
批量构造      0.82           0.48           1.71
遍历          0.07           0.00           32.40

===== Multimap vs Flat_multimap =====

================================================
容器大小: 10000, 操作次数: 10000
================================================
测试标准 map/multimap...
测试 flat_map/flat_multimap...
----------- 性能对比 -----------
操作          标准容器 (ms) flat容器 (ms)    性能比
查找          0.66           0.54           1.23
```

```
插入          0.56           1.78           0.32
批量构造      0.69           0.41           1.68
遍历          0.08           0.00           32.27

===== Map vs Flat_map =====

==========================================
容器大小: 100000, 操作次数: 100000
==========================================
测试标准 map/multimap...
测试 flat_map/flat_multimap...
----------- 性能对比 -----------
操作          标准容器 (ms) flat容器 (ms)   性能比
查找          13.46          7.12           1.89
插入          10.67          146.62         0.07
批量构造      11.73          5.33           2.20
遍历          1.54           0.02           69.04

===== Multimap vs Flat_multimap =====

==========================================
容器大小: 100000, 操作次数: 100000
==========================================
测试标准 map/multimap...
测试 flat_map/flat_multimap...
----------- 性能对比 -----------
操作          标准容器 (ms) flat容器 (ms)   性能比
查找          14.47          7.17           2.02
插入          9.74           160.01         0.06
批量构造      10.88          5.05           2.15
遍历          1.65           0.02           70.24

===== Map vs Flat_map =====
```

```
==========================================
容器大小: 1000000, 操作次数: 1000000
==========================================
测试标准 map/multimap...
测试 flat_map/flat_multimap...
----------- 性能对比 -----------
操作          标准容器 (ms) flat容器 (ms)   性能比
查找          824.18          104.21          7.91
插入          471.17          17398.70        0.03
批量构造      481.34          60.37           7.97
遍历          137.53          0.20            684.66

===== Multimap vs Flat_multimap =====

==========================================
容器大小: 1000000, 操作次数: 1000000
==========================================
测试标准 map/multimap...
测试 flat_map/flat_multimap...
----------- 性能对比 -----------
操作          标准容器 (ms) flat容器 (ms)   性能比
查找          803.13          98.35           8.17
插入          451.21          17175.69        0.03
批量构造      423.51          54.42           7.78
遍历          130.15          0.23            555.74
```

考虑到 map 组和 multimap 组的实验结果类似，下面只展开分析 map 组的实验结果。

4.3 实验结果

4.3.1 查找性能

在查找操作测试中，我们观察到以下结果：

容器大小	**std::map 耗时 (ms)**	**std::flat_map 耗时 (ms)**	性能比
10,000	0.62	0.53	1.17x
100,000	13.46	7.12	1.89x
1,000,000	824.18	104.21	7.91x

在查找操作中，std::flat_map 表现出明显的性能优势，尤其是在数据量较大时。这主要得益于数据的连续存储带来的缓存友好性，减少了缓存未命中的情况。

4.3.2 插入性能

对于插入操作，我们发现以下趋势：

容器大小	**std::map 耗时 (ms)**	**std::flat_map 耗时 (ms)**	性能比
10,000	0.63	1.63	0.39x
100,000	10.67	146.62	0.07x
1,000,000	471.17	17398.70	0.03x

在插入操作中，std::flat_map 的性能明显劣于 std::map，尤其是在数据量较大时。这主要是因为每次插入可能需要移动大量元素，导致性能下降。

4.3.3 批量构造性能

对于一次性构造大量数据的场景，结果如下：

数据量	**std::map 耗时 (ms)**	**std::flat_map 耗时 (ms)**	性能比
10,000	0.82	0.48	1.71x
100,000	11.73	5.33	2.20x
1,000,000	481.34	60.37	7.97x

在批量构造场景中，std::flat_map 表现出明显的性能优势，这主要是因为：

- 一次性分配内存减少了频繁分配的开销
- 减少了节点创建和链接的开销

4.3.4 遍历性能

遍历测试显示：

容器大小	**std::map 耗时 (ms)**	**std::flat_map 耗时 (ms)**	性能比
10,000	0.07	0.00	32.40x
100,000	1.54	0.02	69.04x
1,000,000	137.53	0.20	684.66x

在遍历操作中，std::flat_map 的性能优势非常明显，可达 std::map 的数十到上百倍。这主要得益于数据的连续存储带来的缓存友好性，减少了 CPU 等待时间。

4.4 分析与讨论

基于以上实验结果，我们可以得出以下结论：

1. 查找操作：std::flat_map 在查找操作上表现优异，这主要得益于数据连续存储带来的缓存友好性，减少了缓存未命中的情况。
2. 插入操作：在大数据量场景下，std::flat_map 的插入性能显著劣于 std::map。这与理论复杂度分析一致，因为每次插入可能需要移动大量元素。
3. 批量构造：当需要一次性构建容器时，std::flat_map 表现更好，因为它减少了频繁内存分配和节点创建的开销。
4. 遍历性能：std::flat_map 的遍历性能可达 std::map 的数十到上百倍，这是最显著的性能优势之一，非常适合频繁遍历的场景。

综合性能测试表明，std::flat_map 在查找、遍历和批量构造方面具有明显优势，但在频繁单次插入删除的场景下，std::map 仍然是更好的选择。这符合我们对其底层实现机制的理论分析。

5 优缺点与使用场景

基于前述的理论分析和实验结果，本节将总结 std::flat_map 和 std::flat_multimap 的优缺点，并探讨其最佳使用场景，帮助开发者在实际应用中做出明智的选择。

5.1 优点

std::flat_map 和 std::flat_multimap 相比传统的 map 容器具有以下优势：

- 卓越的查找性能：实验数据表明，flat_map 的查找操作比传统 map 快 7 倍，尤其在大型数据集上优势更为明显。
- 显著的遍历性能优势：得益于连续内存布局，flat_map 的遍历速度可达传统 map 的数十到上百倍，这是其最突出的性能优势之一。
- 高效的批量构造：在一次性构建大容器的场景中，flat_map 比传统 map 快约 7 倍。
- 优秀的缓存局部性：数据存储在连续内存区域，显著提高了缓存命中率，减少了 CPU 等待时间。
- 熟悉的 **API** 接口：与传统 map 保持高度相似的接口，降低了学习成本和代码迁移难度。
- 底层容器可访问性：提供了直接访问和操作底层容器的方法，增加了灵活性。

5.2 缺点

然而，flat_map 也存在一些明显的不足：

- 插入性能劣势：在大型容器中，单次插入操作的性能可能比传统 map 慢几百倍，实验中在 100 万元素的容器中，差距达到 30 倍。
- 删除操作效率低：删除元素后需要移动后续所有元素，在大数据集上开销显著。
- 迭代器易失效：任何修改容器的操作都可能导致迭代器、引用和指针失效，程度超过传统 map。
- 潜在的内存碎片：频繁的插入删除操作可能导致 vector 频繁重新分配，造成内存碎片。
- 实现相对新：作为 C++23 的新特性，部分编译器和库可能尚未完全支持或优化。

5.3 适用场景

基于以上分析，std::flat_map 和 std::flat_multimap 特别适合以下应用场景：

- 读操作远多于写操作的场景：如配置数据、常量表、字典等查询频繁但修改罕见的数据结构。
- 需要频繁遍历的应用：如游戏引擎中的实体系统、批处理操作、图形渲染管线等需要高效遍历所有元素的场景。
- 内存受限的环境：嵌入式系统、移动设备等对内存占用敏感的平台。
- 一次构建多次使用的静态数据：如预计算的查找表、静态资源索引等。
- 批量操作场景：当可以将多次修改合并为一次批量操作时，可以有效规避单次插入的性能劣势。
- 对缓存友好性要求高的性能关键代码：如高性能计算、实时系统等对延迟敏感的场景。

5.4 不适用场景

相反，在以下场景中应当避免使用 flat_map：

- 频繁单次插入或删除的动态集合：如不断添加和移除元素的事件队列、日志系统等。
- 大型数据集且需要频繁修改：元素数量庞大且经常变动的数据结构。
- 依赖迭代器稳定性的算法：需要在遍历过程中修改容器的场景，迭代器易失效会带来问题。
- 并发修改频繁的共享数据：在多线程环境中，线性时间的修改操作可能导致更长的锁定时间。

5.5 优化策略

在决定使用 flat_map 后，可以考虑以下优化策略以获得最佳性能：

- 预分配容量：对于可预知大小的数据集，应通过底层容器预分配足够的空间以减少重分配开销。

- 批量构建：尽可能收集所有元素后一次性构建容器，而非逐个添加。
- 利用特有操作：使用 extract() 和 replace() 等特有操作对底层容器进行批量修改，可以避免渐进式修改的开销。
- 考虑自定义容器：在特殊场景下，可以考虑为 KeyContainer 和 MappedContainer 使用 std::deque 或其他适合的容器类型，平衡不同操作的性能。

总之，std::flat_map 和 std::flat_multimap 为 C++ 开发者提供了在特定场景下性能优越的关联容器选择。通过深入了解其特性和适用场景，开发者可以在实际应用中充分利用其优势，同时规避其局限性，从而开发出更高效的软件系统。

6 总结

本文对 C++23 标准新引入的 std::flat_map 和 std::flat_multimap 两种容器进行了系统性探究，从基本用法、使用案例到性能分析，全面审视了这两种新型关联容器的特性与价值。通过对比研究，我们得出以下结论：

std::flat_map 和 std::flat_multimap 采用平行数组作为底层存储结构，与传统的基于树的 map 和 multimap 相比，在访问模式和性能特征上有显著差异。它们表现出连续内存布局的典型优势，在查找和遍历方面表现卓越，而在动态插入删除操作上则存在明显劣势。

性能测试表明，这两种新容器在查找和批量构造操作上比传统 map 快 7 倍，并在遍历操作上快数十到上百倍，。这种性能特点使其特别适合以下场景：读多写少的应用以及需要频繁遍历的系统。

然而，在频繁单次插入删除、大型数据集动态修改等场景中，传统 map 的对数复杂度优势仍然不可替代。这表明 C++ 标准库朝着“为不同场景提供专门工具”的方向发展，而非追求单一通用解决方案。

对于开发者而言，理解这些新容器的内部机制至关重要，这有助于：

- 在性能关键场景中做出正确的容器选择
- 针对所选容器的特性优化代码
- 更好地理解性能与内存使用之间的权衡

随着 C++23 标准的逐步普及，std::flat_map 和 std::flat_multimap 有望在特定应用领域获得广泛应用，特别是在游戏开发、嵌入式系统、高性能计算等对性能和内存效率有严格要求的场景。未来编译器和库实现的进一步优化，也有可能缓解这些容器在修改操作上的性能劣势。

总之，std::flat_map 和 std::flat_multimap 是 C++ 标准库的重要补充，它们不是用来取代传统 map 的工具，而是为开发者提供了更多元化的选择，使我们能够根据特定应用需求选择最合适的容器，从而开发出更高效的软件系统。正如本文分析所示，了解这些工具的优缺点和适用场景，对于充分发挥 C++ 语言的性能潜力至关重要。

参考文献

[1] ISO/IEC. *Programming Languages – C++*, ISO/IEC 14882:2023(E), 2023.

[2] Boost C++ Libraries. *Boost.Container: flat_map and flat_set*, 2023. https://www.boost.org/doc/libs/release/doc/html/container/non_standard_containers.html

[3] cppreference.com. *std::flat_map, std::flat_multimap*, 2023. https://en.cppreference.com/w/cpp/container/flat_map