C++的算法有哪些

C++的算法体系以标准模板库（STL）为核心，STL将算法与容器、迭代器深度绑定，提供了大量通用、高效的算法接口，无需开发者重复造轮子，且能适配多种数据结构，是C++编程中提升开发效率、保证代码规范性的核心工具。多数C++算法集中在<algorithm>头文件中，部分数值算法位于<numeric>头文件，按功能可分为排序、查找、修改、遍历、数值计算等几大类。本文将逐一拆解各类核心算法，搭配实战示例，帮你理清C++算法的全貌，掌握不同场景下的算法选型与使用方法。

一、前置基础：STL算法的核心依赖

C++的STL算法并非独立存在，其设计依赖两大核心组件，这也是理解和使用C++算法的前提：

- 迭代器（Iterator）：算法与容器的“桥梁”，算法通过迭代器访问容器中的数据，无需直接操作容器本身，实现了“算法与容器解耦”。常用迭代器类型包括输入迭代器、输出迭代器、正向迭代器、双向迭代器、随机访问迭代器，不同算法对迭代器的要求不同。
- 函数对象（Functor）：可作为算法的参数，用于自定义算法的排序、判断等逻辑，灵活扩展算法功能，常见的有谓词（返回bool值的函数/函数对象，用于条件判断）、仿函数、lambda表达式。

核心原则：STL算法是“通用的”，同一算法可适配vector、list、deque等多种容器（需满足迭代器要求），无需针对不同容器单独编写算法。

二、核心算法分类及实战示例

按功能划分，C++ STL算法可分为六大类，涵盖绝大多数开发场景，以下重点拆解高频算法的用法与适用场景。

2.1 排序算法：最常用的基础算法

排序算法用于对容器中的数据按指定规则重新排列，核心依赖随机访问迭代器（如vector、deque适配，list不适用），高频算法包括sort、stable_sort、partial_sort等。

（1）sort：快速排序（不稳定排序）

核心功能：对指定区间的元素进行升序排序，默认使用小于号（<）作为比较规则，可自定义谓词实现降序或自定义排序。底层采用“快速排序+插入排序+堆排序”的混合实现（Introsort），时间复杂度平均O(nlogn)，最坏$O(n^2)$，空间复杂度O(logn)。

实战示例：

```
#include <iostream>
#include <vector>
#include <algorithm> // sort所在头文件
```

```cpp
using namespace std;

int main() {
    vector<int> arr = {3, 1, 4, 1, 5, 9, 2, 6};

    // 1. 默认升序排序
    sort(arr.begin(), arr.end());
    cout << "升序排序结果: ";
    for (int num : arr) cout << num << " "; // 输出: 1 1 2 3 4 5 6 9

    // 2. 自定义降序排序（lambda表达式作为谓词）
    sort(arr.begin(), arr.end(), [](int a, int b) {
        return a > b;
    });
    cout << "\n降序排序结果: ";
    for (int num : arr) cout << num << " "; // 输出: 9 6 5 4 3 2 1 1
    return 0;
}

```

（2）stable_sort：稳定排序

核心功能：与sort功能一致，但保证“相等元素的相对位置不变”（稳定排序），底层采用归并排序，时间复杂度O(nlogn)，空间复杂度O(n)，适用于需保留相等元素原始顺序的场景（如多字段排序）。

（3）partial_sort：部分排序

核心功能：对指定区间的元素排序，仅保证前k个元素是升序的（且为整个区间的前k小元素），后续元素无序，时间复杂度O(nlogk)，适用于只需获取前k个有序元素的场景（如获取Top10）。

2.2 查找算法：定位元素的高效工具

查找算法用于在容器中定位目标元素或满足条件的元素，按查找逻辑可分为线性查找、二分查找等，适配不同数据场景。

（1）线性查找：find、find_if

适用于无序容器（如vector、list），时间复杂度O(n)，遍历整个区间查找目标。

- find：查找与目标值相等的第一个元素，返回指向该元素的迭代器，未找到则返回end()；
- find_if：查找满足自定义条件的第一个元素，接收谓词作为参数，灵活性更强。

实战示例：

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

```cpp
#include <vector>
#include <algorithm>
using namespace std;

int main() {
    vector<int> arr = {3, 1, 4, 1, 5, 9};

    // 1. find：查找值为5的元素
    auto it1 = find(arr.begin(), arr.end(), 5);
    if (it1 != arr.end()) {
        cout << "找到元素5，位置：" << it1 - arr.begin() << endl; // 输出：4
    } else {
        cout << "未找到元素5" << endl;
    }

    // 2. find_if：查找第一个大于4的元素
    auto it2 = find_if(arr.begin(), arr.end(), [](int num) {
        return num > 4;
    });
    if (it2 != arr.end()) {
        cout << "第一个大于4的元素：" << *it2 << endl; // 输出：5
    }
    return 0;
}

```

（2）二分查找：binary_search、lower_bound、upper_bound

仅适用于**已排序的容器**，时间复杂度O(logn)，效率远高于线性查找，底层基于二分法实现。

- binary_search：判断目标元素是否存在，返回bool值（true存在，false不存在）；
- lower_bound：查找第一个大于等于目标值的元素，返回迭代器；
- upper_bound：查找第一个大于目标值的元素，返回迭代器，常与lower_bound配合获取区间中等于目标值的元素范围。

（3）其他查找算法

find_end：查找某个子序列在目标区间中最后一次出现的位置；count、count_if：统计目标元素或满足条件的元素个数；search：查找子序列第一次出现的位置。

2.3 修改算法：操作容器元素的核心手段

修改算法用于对容器中的元素进行增删、替换、反转、去重等操作，多数算法会直接修改容器内容（部分算法需配合容器的erase方法完成删除）。

（1）去重：unique

核心功能：移除容器中连续的重复元素，仅保留第一个，返回指向“去重后最后一个元素”的迭代器。注意：unique不会删除元素，仅将重复元素移至容器尾部，需搭配erase完成真正删除，且建议先排序（确保重复元素连续）。

实战示例：

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main() {
    vector<int> arr = {3, 1, 1, 4, 5, 5, 5, 9};

    // 先排序，确保重复元素连续
    sort(arr.begin(), arr.end());
    // 去重，获取去重后尾部迭代器
    auto it = unique(arr.begin(), arr.end());
    // 真正删除重复元素
    arr.erase(it, arr.end());

    cout << "去重后结果：";
    for (int num : arr) cout << num << " "; // 输出：1 3 4 5 9
    return 0;
}

```

（2）其他常用修改算法

- reverse：反转容器中指定区间的元素，适配双向迭代器（如vector、list），时间复杂度O(n)；
- replace、replace_if：替换元素，replace替换与目标值相等的元素，replace_if替换满足条件的元素；
- swap：交换两个容器的内容，或交换两个元素的值，高效无额外内存开销；
- fill：将容器指定区间的元素填充为目标值（如将vector全部填充为0）。

2.4 遍历算法：批量处理容器元素

遍历算法用于对容器中的每个元素执行相同或自定义操作，核心是“批量处理”，减少重复遍历代码。

（1）for_each：通用遍历

最常用的遍历算法，接收迭代器区间和函数对象（或lambda表达式），对每个元素执行函数操作，支持修改元素值，灵活性极强。

实战示例：

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

// 自定义函数：打印元素
void printNum(int num) {
    cout << num << " ";
}

int main() {
    vector<int> arr = {3, 1, 4, 1, 5, 9};

    // 1. 传入函数名，打印所有元素
    cout << "遍历打印：";
    for_each(arr.begin(), arr.end(), printNum); // 输出：3 1 4 1 5 9

    // 2. 传入lambda表达式，修改元素（每个元素加2）
    for_each(arr.begin(), arr.end(), [](int& num) {
        num += 2;
    });

    cout << "\n修改后遍历：";
    for_each(arr.begin(), arr.end(), printNum); // 输出：5 3 6 3 7 11
    return 0;
}

```

（2）transform：遍历并转换元素

遍历目标区间的元素，通过自定义逻辑转换元素值，并将结果存储到另一个容器中，适用于批量元素转换（如将所有元素转为绝对值、平方等）。

2.5 数值算法：用于数值计算（<numeric>头文件）

数值算法集中在<numeric>头文件，专门用于处理数值类型的元素（int、float、double等），实现求和、求积、累加等计算功能。

- accumulate：对指定区间的元素求和（默认），可自定义累加规则（如求积、累加平方和）；
- inner_product：计算两个区间元素的内积（对应元素相乘后求和）；
- adjacent_difference：计算相邻元素的差值，存储到目标容器中。

实战示例（accumulate求和）：

```cpp
#include <iostream>
#include <vector>
#include <numeric> // accumulate所在头文件
using namespace std;

int main() {
    vector<int> arr = {1, 2, 3, 4, 5};

    // 1. 默认求和（初始值为0）
    int sum = accumulate(arr.begin(), arr.end(), 0);
    cout << "元素总和：" << sum << endl; // 输出：15

    // 2. 自定义累加规则：求所有元素的平方和（初始值为0）
    int squareSum = accumulate(arr.begin(), arr.end(), 0, [](int total, int num) {
        return total + num * num;
    });
    cout << "元素平方和：" << squareSum << endl; // 输出：55（1+4+9+16+25）
    return 0;
}
```

2.6 集合算法：用于两个容器的集合操作

集合算法用于对两个**已排序的容器**执行集合操作（交集、并集、差集等），操作后结果仍保持有序，适用于批量数据的集合运算。

- set_intersection：求两个容器的交集（同时存在于两个容器中的元素）；
- set_union：求两个容器的并集（去重后合并两个容器的元素）；
- set_difference：求两个容器的差集（存在于第一个容器但不存在于第二个容器的元素）。

三、C++算法的使用注意事项

- 迭代器兼容性：不同算法对迭代器类型有要求（如sort需随机访问迭代器，list无法使用sort，需用list自带的成员函数sort），误用会导致编译错误；
- 容器有效性：修改算法（如unique、erase）会改变容器的元素数量或位置，操作后需注意迭代器失效问题（避免使用失效的迭代器访问元素）；
- 自定义谓词：编写自定义谓词时，需保证逻辑稳定（如排序谓词需满足严格弱序），否则会导致算法运行异常；
- 头文件引入：不同算法位于不同头文件，需正确引入（<algorithm>为通用算法，<numeric>为数值算法）。

四、总结：C++算法的核心价值与学习建议

C++的算法体系以STL为核心，覆盖排序、查找、修改、遍历、数值计算、集合操作等全场景，其核心价值是“通用、高效、可扩展”——通过迭代器与容器解耦，实现一份算法适配多种数据结构，通过函数对象扩展自定义逻辑，兼顾规范性与灵活性。

对于初学者，学习C++算法的重点的：

1. 优先掌握高频算法（sort、find、for_each、accumulate、unique），明确其适用场景与时间复杂度；
2. 理解迭代器与算法的关联，分清不同算法对迭代器的要求；
3. 熟练使用lambda表达式作为谓词，灵活扩展算法功能；
4. 结合容器特性选型（如有序容器用二分查找，无序容器用线性查找，list用自带排序函数）。

进阶学习可深入算法底层实现（如sort的混合排序逻辑、二分查找的原理），结合数据结构优化算法选型，真正发挥C++算法的高效性，写出简洁、高性能的代码。