说说数据结构中的栈和队列

在数据结构的基础体系中，栈和队列是两类特殊的线性数据结构——它们基于数组或链表封装而成，核心特点是**限制数据操作的顺序**，无法像数组、链表那样自由访问任意元素。这种“操作受限”的设计，让它们能精准适配特定业务场景，成为程序开发中高频使用的基础结构。比如编辑器的撤销功能依赖栈，消息队列的有序处理依赖队列，二者看似相似，实则操作规则、适用场景截然不同。本文将从定义、底层实现、核心特性、实战应用四个维度，理清栈和队列的本质与用法，帮大家吃透这两类基础结构。

一、先搞懂：栈和队列的核心定义

栈和队列均属于线性数据结构，元素之间呈一对一线性关联，与数组、链表共享“线性排列”的底层逻辑，但核心区别在于**数据插入和删除的位置限制**——这种限制造就了它们独特的操作规则，也是区分二者的关键。

1. 栈：先进后出（FILO）的“容器”

栈（Stack）的操作规则是**先进后出**，即 First In Last Out，简称 FILO。可以类比生活中的“叠盘子”：先叠上去的盘子放在最底层，最后才能取出；后叠上去的盘子放在最顶层，最先能取出。

栈的操作被严格限制在“栈顶”（最顶端的元素），核心操作只有两个：

- 入栈（Push）：将数据插入到栈顶，成为新的栈顶元素；
- 出栈（Pop）：将栈顶元素删除，并返回该元素，原栈顶的下一个元素成为新的栈顶。

补充：栈不支持在中间位置插入、删除元素，也不支持直接访问栈底或中间元素，若需访问，必须通过出栈操作依次取出栈顶元素，直到目标元素暴露。

2. 队列：先进先出（FIFO）的“队伍”

队列（Queue）的操作规则是**先进先出**，即 First In First Out，简称 FIFO。可以类比生活中的“排队买票”：先排队的人最先办理业务，后排队的人依次等待，遵循“先来后到”的规则。

队列的操作被限制在两个端点，核心操作同样有两个：

- 入队（Push）：将数据插入到队列的“队尾”，成为新的队尾元素；
- 出队（Pop）：将队列“队头”的元素删除，并返回该元素，原队头的下一个元素成为新的队头。

补充：与栈类似，队列也不支持在中间位置插入、删除元素，无法直接访问队尾以外的中间元素，访问中间元素需通过出队操作依次取出队头元素。

核心区别：操作规则决定用途

栈和队列的本质区别，不在于底层存储方式，而在于**操作顺序的限制**：栈是“一端进出”，队列是“两端进出”。这种差异让它们适配完全不同的场景——栈适合“回溯、撤销”类场景，队列适合“有序调度、排队处理”类场景。

二、底层实现：基于数组或链表的封装

栈和队列本身不具备独立的存储能力，底层需依赖数组或链表实现，两种实现方式各有优劣，需根据业务场景选择。C++标准库中，stack（栈）和queue（队列）均默认基于deque（双端队列）实现，兼顾数组和链表的优势。

1. 栈的两种底层实现

（1）基于数组实现（顺序栈）

以数组为底层容器，用一个指针（或变量）记录栈顶位置，入栈、出栈操作本质是修改栈顶指针并操作数组元素。

- 优势：数组下标访问高效，入栈、出栈操作时间复杂度均为O(1)，内存开销小；
- 局限：数组大小固定（原生数组），需手动处理扩容问题（动态数组vector可自动扩容，但扩容时会产生内存拷贝开销）；
- 适用场景：数据量固定、入栈出栈频繁，且内存敏感的场景。

（2）基于链表实现（链式栈）

以链表为底层容器，通常用单链表实现，将新元素作为新的头节点（栈顶），入栈即插入头节点，出栈即删除头节点。

- 优势：支持动态扩容，无需担心大小限制，插入、删除时无内存拷贝开销；
- 局限：链表节点需额外存储指针，内存开销略大，访问效率低于数组实现；
- 适用场景：数据量不固定、频繁扩容的场景。

2. 队列的两种底层实现

（1）基于数组实现（顺序队列）

以数组为底层容器，用两个指针分别记录队头和队尾位置，入队修改队尾指针，出队修改队头指针。为避免数组空间浪费，通常采用“循环数组”（环形队列）的优化方式。

- 优势：访问高效，入队、出队时间复杂度O(1)，内存开销小；
- 局限：需处理队列满、队列空的边界条件，环形队列实现逻辑略繁琐；
- 适用场景：数据量固定、排队处理频繁的场景（如打印机队列）。

（2）基于链表实现（链式队列）

以链表为底层容器，用双链表或单链表实现，入队即插入队尾节点，出队即删除队头节点，需额外记录队头和队尾指针。

- 优势：动态扩容，无需处理环形队列的边界问题，实现逻辑简单；
- 局限：节点指针占用额外内存，访问效率低于数组实现；
- 适用场景：数据量不固定、入队出队频繁的场景（如消息队列）。

三、核心操作与C++实战示例

C++标准库提供了stack和queue容器，封装了完整的入栈、出栈、入队、出队操作，无需手动实现底层逻辑，直接调用接口即可，适合快速开发。

1. 栈（stack）的核心操作与示例

stack容器的核心接口的简洁，仅暴露入栈、出栈、访问栈顶等必要操作，符合“操作受限”的设计原则。

```
#include <iostream>
#include <stack> // 需包含头文件
using namespace std;

int main() {
    // 1. 定义栈（默认存储int类型，底层基于deque实现）
    stack<int> st;

    // 2. 入栈（push）
    st.push(10);
    st.push(20);
    st.push(30); // 栈内元素：10（栈底）→20→30（栈顶）

    // 3. 访问栈顶元素（top），不删除
    cout << "栈顶元素：" << st.top() << endl; // 输出30

    // 4. 出栈（pop），删除栈顶元素，无返回值
    st.pop(); // 栈顶元素30被删除，新栈顶为20
    cout << "出栈后栈顶元素：" << st.top() << endl; // 输出20

    // 5. 其他常用接口
    cout << "栈的大小：" << st.size() << endl; // 输出2
    cout << "栈是否为空：" << (st.empty() ? "是" : "否") << endl; // 输出否

    // 清空栈（无直接清空接口，需循环出栈）
    while (!st.empty()) {
        st.pop();
    }
```

```
    cout << "清空后栈是否为空：" << (st.empty() ? "是" : "否") << endl; // 输出是
    return 0;
}

```

2. 队列（queue）的核心操作与示例

queue容器与stack类似，接口简洁，仅暴露入队、出队、访问队头/队尾等操作，无法访问中间元素。

```
#include <iostream>
#include <queue> // 需包含头文件
#include <string>
using namespace std;

int main() {
    // 1. 定义队列（存储string类型，底层基于deque实现）
    queue<string> q;

    // 2. 入队（push）
    q.push("消息1");
    q.push("消息2");
    q.push("消息3"); // 队列内元素：消息1（队头）→消息2→消息3（队尾）

    // 3. 访问队头、队尾元素（front/back），不删除
    cout << "队头元素：" << q.front() << endl; // 输出消息1
    cout << "队尾元素：" << q.back() << endl;  // 输出消息3

    // 4. 出队（pop），删除队头元素，无返回值
    q.pop(); // 队头元素消息1被删除，新队头为消息2
    cout << "出队后队头元素：" << q.front() << endl; // 输出消息2

    // 5. 其他常用接口
    cout << "队列大小：" << q.size() << endl; // 输出2
    cout << "队列是否为空：" << (q.empty() ? "是" : "否") << endl; // 输出否

    // 清空队列（无直接清空接口，需循环出队）
    while (!q.empty()) {
        q.pop();
    }
    cout << "清空后队列是否为空：" << (q.empty() ? "是" : "否") << endl; // 输出是
    return 0;
}

```

四、常见应用场景：精准匹配操作规则

栈和队列的“操作受限”特性，让它们在特定场景中具备不可替代的优势，以下是两类结构的高频应用场景，贴合实战开发需求。

1. 栈的典型应用场景

- 撤销与回溯操作：编辑器的Ctrl+Z撤销操作（每一步操作入栈，撤销时出栈恢复）、递归调用的函数栈（递归过程中，函数上下文入栈，递归返回时出栈）；
- 表达式求值：数学表达式的计算（如后缀表达式求值）、括号匹配校验（左括号入栈，右括号出栈匹配，判断是否合法）；
- 其他场景：浏览器的前进后退功能（两个栈分别存储前进、后退记录）、单调栈求解数组极值问题（如每日温度、接雨水）。

示例：括号匹配校验（栈的经典应用）

```
#include <iostream>
#include <stack>
#include <string>
using namespace std;

// 括号匹配校验函数
bool isValid(string s) {
    stack<char> st;
    for (char c : s) {
        // 左括号入栈
        if (c == '(' || c == '[' || c == '{') {
            st.push(c);
        } else {
            // 右括号，若栈空则不匹配
            if (st.empty()) return false;
            // 取出栈顶元素，判断是否匹配
            char top = st.top();
            st.pop();
            if ((c == ')' && top != '(') ||
                (c == ']' && top != '[') ||
                (c == '}' && top != '{')) {
                return false;
            }
        }
    }
    // 栈空说明所有括号都匹配，否则不匹配
    return st.empty();
}

```

```
int main() {
    string s1 = "({[]})";
    string s2 = "({[)]}";
    cout << s1 << " 是否合法: " << (isValid(s1) ? "是" : "否") << endl; // 是
    cout << s2 << " 是否合法: " << (isValid(s2) ? "是" : "否") << endl; // 否
    return 0;
}

```

2. 队列的典型应用场景

- 排队调度场景：打印机队列（按顺序处理打印任务）、任务调度队列（线程池按顺序执行任务）；
- 消息队列场景：即时通讯的消息推送（消息按发送顺序入队，接收方按顺序出队处理）、日志收集（日志按产生顺序入队，批量处理）；
- 算法应用：广度优先搜索（BFS）（遍历节点入队，按顺序出队遍历邻接节点）、滑动窗口问题（用队列维护窗口内元素）。

五、延伸知识点：特殊的栈和队列

在基础栈和队列的基础上，衍生出一些特殊结构，优化了特定场景的性能，是实战中的高频考点和常用工具。

- 单调栈：栈内元素保持严格的递增或递减顺序，用于高效求解数组中“下一个更大/更小元素”等问题，时间复杂度优化至O(n)；
- 双端队列（deque）：允许在两端插入、删除元素，兼顾栈和队列的优势，是C++中stack和queue的默认底层容器；
- 优先队列（priority_queue）：特殊的队列，入队时按优先级排序，出队时优先取出优先级最高的元素，底层基于堆实现，适合“优先级调度”场景（如急诊病人优先处理）。

六、总结

栈和队列是“操作受限的线性数据结构”，核心差异在于操作规则：栈遵循先进后出，适配回溯、撤销类场景；队列遵循先进先出，适配排队、调度类场景。二者均依赖数组或链表实现，数组实现高效省内存，链表实现灵活可扩容，C++标准库的stack和queue容器已封装好底层逻辑，可直接用于实战。

学习栈和队列的关键，不在于死记硬背接口，而在于理解“操作受限”的设计思想——它们放弃了自由访问的灵活性，换来了场景适配的高效性。在实际开发中，需根据核心操作（是否需要回溯、是否需要有序排队）选择合适的结构，同时可结合单调栈、优先队列等衍生结构，优化复杂问题的求解效率。

作为数据结构的基础，栈和队列也是后续学习堆、图等复杂结构及算法的基石，吃透二者的用法，能为后续进阶学习打下坚实基础。

（注：文档部分内容可能由 AI 生成）