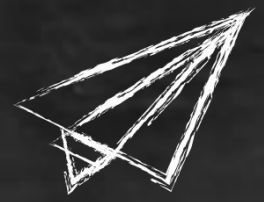


2025/10/19

双向队列

汇报人: 倪中阳

CONTENTS

目录

01

双向队列-引言

为什么需要双向队列?

已有结构的局限

栈（Stack）
后进先出-LIFO

队列（Queue）
先进先出-FIFO

如果我们希望在两端都能进行插入与删除呢?

比如:
同时处理
“新任务加入”和“旧任务移除”;

现实中的动机场景

场景1：浏览器的“前进 / 后退”

当用户访问网页时，我们希望可以从头部回退，也能从尾部前进;

栈无法做到两端移动，普通队列也不行;

双向队列正好支持在两端进行操作。

算法问题

场景2：滑动窗口算法

在“滑动窗口最大值”问题中，需要不断从右侧加入新元素、从左侧移除旧元素;

如果用普通队列删除左端会很低效;

双向队列能在 O(1) 时间完成两端操作，是最佳选择。。

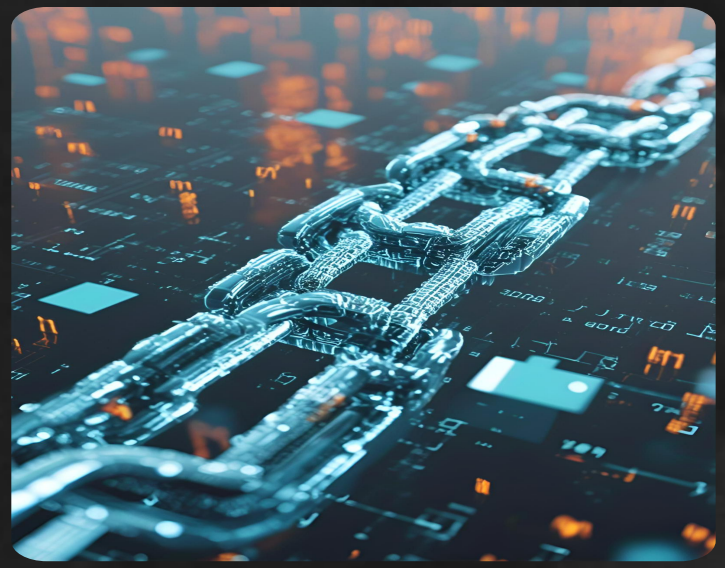
02

双向队列-定义与特点

定义(Definition)

双向队列（Deque，Double Ended Queue）
是一种 允许在两端进行插入和删除操作 的线性数据结构。

Front <—— [1] [2] [3] [4] ——> Back
↑删/插 ↑删/插



C++ STL(Standard Template Library)中的定义

```
#include <deque>
std::deque<int> dq;  // 定义一个整型双向队列
```

STL中 deque 提供了：

高效的双端插入/删除；

支持随机访问（dq[i]）；

动态扩容，不需要手动循环数组。

03

双向队列-实现原理

顺序存储实现（基于循环数组）

思想：

使用一个循环数组存储元素，通过 - 两个索引变量（front, rear）-控制队首和队尾的位置。

索引： 0 1 2 3 4
数据： [ ] [A] [B] [C] [ ]
front → 1
rear → 3

优点	缺点
内存连续，随机访问快（O(1)）	容量固定，扩容成本高
无需指针，空间开销小	插入删除需移动指针，逻辑复杂

插入/删除规则：

push_front()：front 向左移动，存入新元素；

push_back()：rear 向右移动，存入新元素；

pop_front()：front 向右移动；

pop_back()：rear 向左移动；

当指针越界时，通过取模 %size 实现“循环”。

链式存储实现（基于双向链表）

思想：

每个节点有两个指针：

prev 指向前驱节点；

next 指向后继节点。

可在头尾两端常数时间插入/删除。

双向操作：

头插：新节点连接到头结点前；

尾插：新节点连接到尾结点后；

删除操作只需修改相邻节点指针。

结构示意图

NULL ← [A] ⇄ [B] ⇄ [C] → NULL

优点	缺点
动态扩容，无需预设长度	每个节点额外存两个指针，空间开销较大
插入/删除效率高（O(1)）	随机访问需遍历（O(n)）

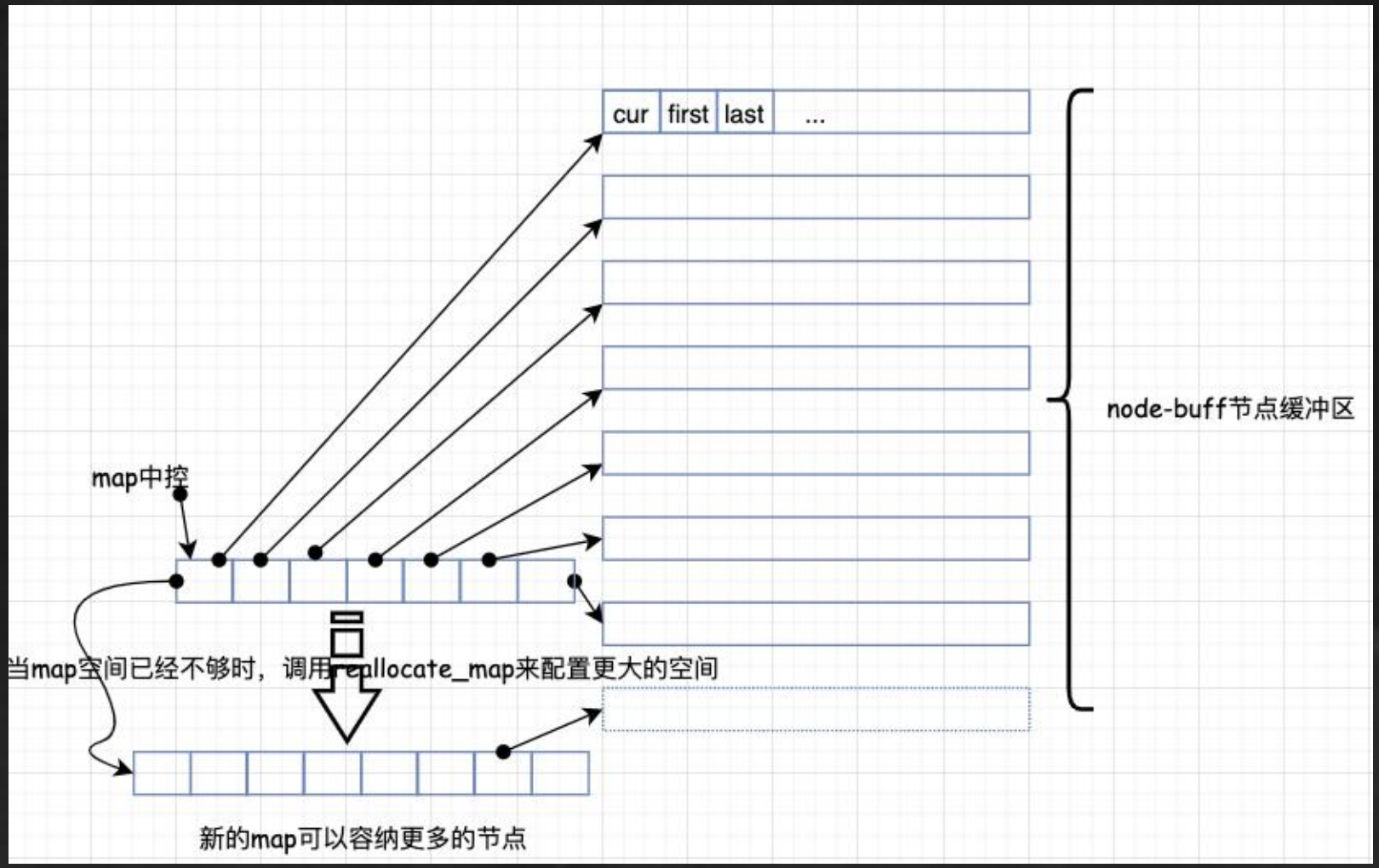
C++ STL中的实现 - 打破 “连续内存” 的迷思：分段存储架构

在 C++ STL 中

std::deque 的底层实现其实是 “中控数组 + 指针映射表（分段缓冲区）”（既非单纯数组，也非链表）

具体来说，deque 有缓冲区（Buffer），这是实际存储数据的地方，每个缓冲区是一块固定大小的连续内存块。在 SGI（Silicon Graphics Inc） STL 中，其默认大小通常是 512 字节。当一个缓冲区满了之后，就会新建一个缓冲区，然后通过中控数组将这些缓冲区关联起来。这种方式避免了像 vector 那样一次性分配大块内存可能失败的问题，也减少了内存碎片的产生。

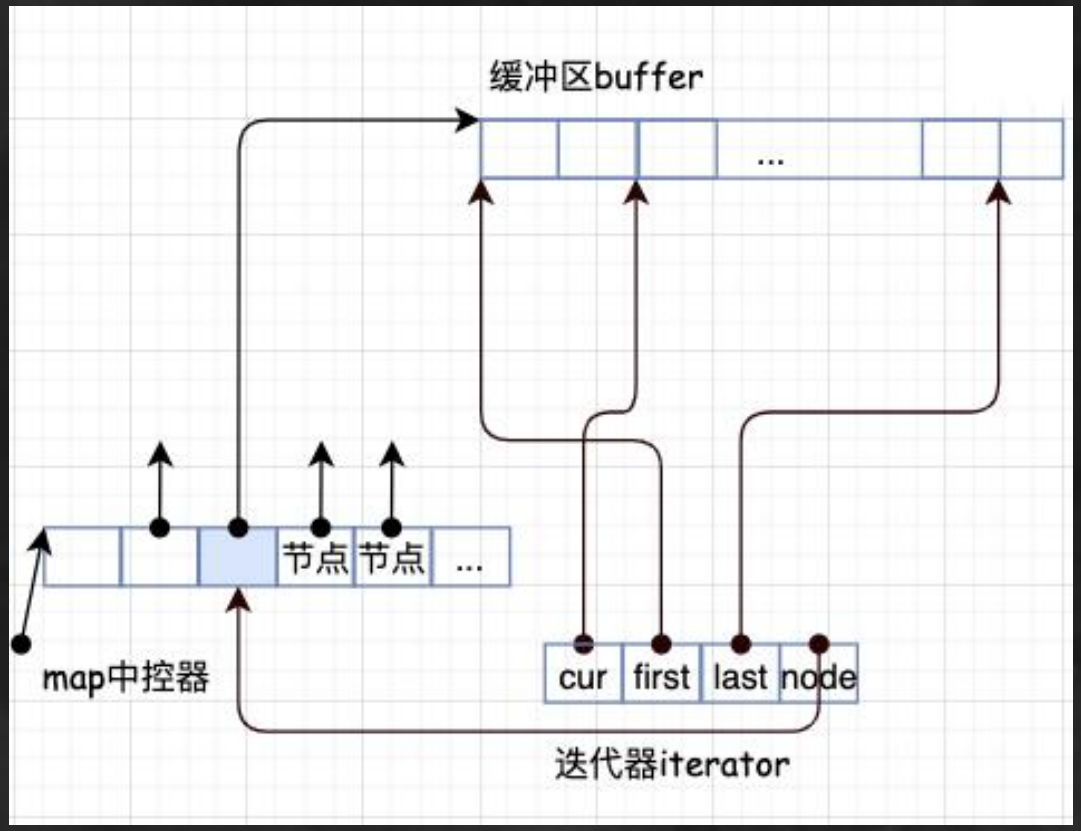
中控数组（Map）则是管理这些缓冲区的关键。它本质上是一个指针数组，数组中的每个元素都指向一个缓冲区的起始地址。比如，中控数组的第 i 个元素，就指向了第 i 个缓冲区。通过这种方式，deque 在逻辑上把这些分段的缓冲区连接成了一个连续的序列，尽管它们在物理内存上可能并不连续。


cur
first
last
...
node-buff节点缓冲区
map中控
当map空间已经不够时，调用reallocate_map来配置更大的空间
新的map可以容纳更多的节点

中控数组
+
分段缓冲区

Deque中的迭代器设计（Iterator）

由于 deque 的分段存储特性，其迭代器不能像普通指针那样简单地进行自增或自减操作。deque 的迭代器需要记录当前元素指针 cur、缓冲区起始指针 first、缓冲区末尾指针 last，以及中控数组指针 node。通过这些指针，迭代器可以在不同的缓冲区之间跳转，从而实现对 deque 中所有元素的无缝遍历。



迭代器的 “跨界” 魔法：如何实现逻辑连续访问？

deque 迭代器的设计是实现其逻辑连续访问的关键，当迭代器到达当前缓冲区末尾（cur == last）时，它会通过 node 指针找到中控数组中下一个缓冲区的指针，然后将 cur 重置为新缓冲区的 first 位置，从而实现跨缓冲区的无缝迭代 。这一过程对用户是完全透明的，用户在使用 deque 时，无需关心其内部复杂的存储结构和迭代器实现细节，就能够像使用连续数组一样，通过迭代器自由地遍历 deque 中的元素，充分体现了 deque 在设计上的精妙之处。

用过 deque 的都知道，它既能像 vector 一样支持随机访问，又能像 list 一样高效完成双端插入 —— 但很少有人注意到，这背后藏着迭代器的 “跨界” 绝技：明明底层是分段内存（中控数组管理的零散缓冲区），迭代器却能让用户感知到 “逻辑连续” 的访问体验，就像在一块完整内存里遍历一样。

案例：向后迭代（++it）的 “跨界”

假设迭代器当前在 Buffer1 的最后一个元素（cur == last - 1），此时执行++it：

边界检测：迭代器发现 cur + 1 == last（已经到 Buffer1 的末尾），需要 “跨界”；

找下一个缓冲区：通过map_ptr（指向中控数组第 0 个元素），移动到下一个位置map_ptr++（现在指向中控数组第 1 个元素，即 Buffer2 的地址）；

切换缓冲区：将first更新为 Buffer2 的起始地址，last更新为 Buffer2 的末尾地址；

定位新元素：将cur设为first（Buffer2 的第一个元素），完成 “跨界”。

举个实际场景：用 deque 存储日志数据，需要频繁在头部插入时间戳、尾部追加日志，同时还要遍历打印所有日志。如果用 list，遍历效率低；用 vector，头部插入会移动大量数据；而 deque 的迭代器让 “遍历” 和 “双端插入” 都高效 —— 这就是 “跨界” 魔法的价值。

04

核心操作细节

双端插入 / 删除：O (1) 效率的实现密码

在 deque 的双端插入和删除操作中，以尾插（push_back）为例，当我们执行 push_back 操作时，deque 会首先检查尾部缓冲区是否还有空余空间 。如果有，直接将元素插入到尾部缓冲区的空闲位置，这一步操作的时间复杂度为 O (1)。但如果尾部缓冲区已满，，deque 需要新建一个缓冲区，接着，更新中控数组，将新缓冲区的指针添加到中控数组的相应位置，最后把元素插入到新缓冲区的起始位置。虽然这个过程涉及到新缓冲区的分配和中控数组的更新，但由于这些操作并不频繁，所以均摊下来，尾插操作的时间复杂度仍然是 O (1)。

头部插入（push_front）的逻辑与尾插类似，只是方向相反。当头部缓冲区有空间时，直接插入元素；若头部缓冲区已满，则新建缓冲区，在中控数组的头部添加新缓冲区的指针，然后插入元素。这种双端插入的高效性，使得 deque 在需要频繁在两端添加元素的场景中表现出色。

随机访问：分段结构下的 O (1) 如何实现？

在 deque 中，虽然其元素存储在分段的缓冲区中，但通过数学计算，依然能够实现 O (1) 的随机访问。假设我们要访问 deque 中的第 n 个元素 。首先，计算缓冲区索引，即 buf_idx = n / 缓冲区大小，然后，计算缓冲区内偏移量，offset = n % 缓冲区大小。最后，通过中控数组获取缓冲区指针，访问缓冲区起始地址 + offset，这样就能准确地定位到目标元素。

尽管这个过程比在连续内存的数组中直接通过索引访问多了一次指针跳转，即先通过中控数组找到缓冲区指针，再在缓冲区内定位元素，但由于每一步操作都是简单的数学计算和指针操作，所以整体的时间复杂度仍然保持在 O (1)。在实际应用中，当我们需要快速获取 deque 中某个位置的元素时，这种随机访问机制能够快速响应，虽然性能略逊于 vector 的直接内存访问，但远优于 list 的链表式遍历 。

中间操作：为何 deque 不擅长 “中间地带”？

deque 在中间位置进行插入和删除操作时，效率并不高。当在中间位置插入元素时，如果插入位置在当前缓冲区内部，那么需要将插入位置之后的元素依次向后移动一个位置，为新元素腾出空间 。如果插入位置跨缓冲区，不仅要移动元素，还需要调整中控数组的指针，以保证逻辑上的连续性。删除操作同理，需要将删除位置之后的元素向前移动，若跨缓冲区同样要调整中控数组。这种操作涉及到大量元素的移动以及中控数组的调整，时间复杂度为 O (n)，与 list 在中间位置通过指针操作实现 O (1) 的插入删除效率相比，deque 在中间操作上明显处于劣势。

所以，在实际使用 deque 时，应尽量避免在中间位置进行频繁操作，而将其应用场景主要聚焦在双端操作上。

容量管理：无 capacity () 的背后设计

deque 与 vector 不同，它没有 capacity () 的概念。vector 为了避免频繁的内存重新分配，会预先分配一定的内存空间，当元素数量超过当前容量时，再进行扩容操作，这可能涉及到大量元素的拷贝和旧内存的释放 。而 deque 采用分段存储的方式，天然支持动态扩展。当需要存储更多元素时，只需添加新的缓冲区，并在中控数组中添加对应的指针即可，无需一次性预分配大量内存。这种方式使得 deque 的内存利用率更高，不会因为预分配过多内存而造成浪费。

然而，这种设计也意味着 deque 无法通过预分配内存来优化批量插入性能。在某些需要批量插入大量元素的场景中，vector 可以通过 reserve () 函数预先分配足够的内存，从而减少插入过程中的内存重新分配次数，提高插入效率 。但 deque 由于其独特的存储结构，无法采用类似的优化方式。不过，在大多数情况下，deque 在双端操作上的高效性以及良好的内存管理特性，使其在许多场景中依然是一种非常实用的数据结构。

05

与其他结构对比

在 C++ 的标准库中，deque 与 vector 和 list 一同构成了顺序容器的重要成员，但它们各自有着独特的设计理念和适用场景 。

vector 是连续内存存储的代表，就像一整条紧密排列的书架，所有元素在内存中依次紧密相连。这使得它在随机访问时性能卓越，通过索引直接定位元素就如同在书架上按编号取书一样迅速，时间复杂度为 O (1)。vector 在尾端插入和删除元素时也有不错的效率，平均时间复杂度为 O (1) 。但在中间位置插入或删除元素时，vector 就需要移动大量元素，效率较低，时间复杂度为 O (n)，比如在已经摆满书的书架中间插入一本新书，就需要将后面的书依次挪动。

list 则是离散存储的典型，采用双向链表结构，每个元素都像一个独立的小房间，通过指针与前后元素相连。这让 list 在插入和删除操作上极为灵活，无论在链表的任何位置，只需修改指针指向，就能完成元素的插入或删除，时间复杂度为 O (1) 。但由于这种离散的结构，list 的随机访问需要从链表头开始遍历，就像在一排不相邻的房间中找特定房间，效率低下，时间复杂度为 O (n)。

再看 deque，它结合了 vector 和 list 的部分优点，采用分段连续内存存储。在双端操作上，deque 和 list 一样高效，能在 O (1) 时间内完成插入和删除 。在随机访问方面，虽然不如 vector 直接，但通过中控数组和数学计算，也能实现 O (1) 的时间复杂度，只是多了一次指针跳转。在中间位置操作时，deque 的效率与 vector 类似，时间复杂度为 O (n) 。

选择数据结构

具体选择哪种容器，需要根据实际需求来判断。如果你的场景需要频繁进行双端操作，比如实现任务队列、滑动窗口等，且偶尔有随机访问的需求，那么 deque 是首选 。像在实现一个实时数据处理系统的任务队列时，任务不断从队头取出、从队尾插入，同时可能需要偶尔查看队列中间某个任务的状态，deque 就能很好地胜任。但如果追求极致的随机访问性能，比如实现一个频繁读取数据的数组型数据存储结构，vector 无疑是更好的选择 。而当需要在容器中间频繁进行插入和删除操作时，list 则更胜一筹，比如在实现一个频繁修改节点的双向链表结构时，list 的优势就会充分体现出来 。

典型案例：

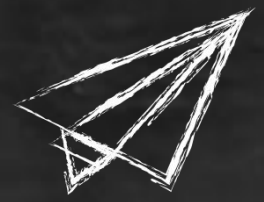
实现双端队列（如任务调度队列，支持高优先级任务头插、普通任务尾插）。
滑动窗口算法（需频繁在两端增删，同时随机访问窗口内元素）。
缓存系统（需高效更新首尾元素，同时支持随机访问缓存项）。

```
=== Double-ended operation test (100000 operations) ===
deque:  0.003226 s
vector: 0.326935 s
list:   0.010975 s
vector is 101.356399x slower than deque
list   is 3.402437x slower than deque

=== Random access test (50000 elements) ===
deque:  0.003646 s
vector: 0.001213 s
list:   3.212677 s
deque is 3.007175x slower than vector
list  is 881.102792x slower than deque

=== Middle insertion/deletion test (10000 operations) ===
deque:  0.025605 s
vector: 0.005436 s
list:   0.001102 s
deque is 23.233282x slower than list
vector is 0.212303x slower than deque

=== Approximate memory usage (50000 elements) ===
deque:  ~4503599627370691 KB
vector: ~195 KB
list:   ~976 KB
```



THE END

谢谢