# 概述

所谓的队列是一种先入先出（FIFO，First In First Out）的数学模型。

# 分类

## 静态队列

静态队列对于队列的长度有一个限制，如果内存不能继续分配时，则队尾不能继续向后增加。

静态队列有两种处理方式：

1. 队头出去以后后面的元素依次向前移动一个位置，这样元素移动多处理效率低；

说明：这种涉及到数据的移动，基本不会用，还是用指针移动比较好。

1. 队列元素不移动，只移动队头指针，这样比较浪费空间资源。

注：**静态队列即用数组实现的，链式队列是用链表实现的**。

**静态队列通常都是循环队列（否则空间不足）**。为了弥补普通队列的缺点，引入了环形队列。

环形/循环队列的排队是有顺时针和逆时针之说的。

在元素出队后，队头指针是不断向后移动的，队尾也是可以向后移动的，这样处理速度快，而且也可以充分利用空间。

## 循环队列

解决假溢出的办法就是如果后面满了，就再从头开始，也就是头尾相接的循环。

### 定义

**循环队列它的容量是固定的**，并且它的队头和队尾指针都可以随着元素入出队列而发生改变，这样循环队列逻辑上就好像是一个环形存储空间。

注：在实际内存当中，不可能有真正的环形存储区，我们只是用顺序表模拟出来的逻辑上的循环。

循环队列只需要改变front和rear指针，也就是让front或rear指针不断加1，即使超出了地址范围，也会自动从头开始，我们可以采取取模运算处理：

(rear+1)%QueueSize

(front+1)%QueueSize

取模就是取余数的意思，他取到的值永远不会大于除数。

#define MAXSIIZE 100

typedef struct

{

ElemType \*base; //用于存放内存分配基地址,也可用数组存放

int front;

int rear;

}

### 参数

循环队列需要2个参数front和rear确定，2个参数不同场合有不同意义。

1. 队列初始化

front和rear的值都是零

1. 队列非空

front代表的是队列的第一个元素

rear代表的是队列的最后一个有效元素的下一个元素

1. 队列空

front和rear的值相等，但不一定是零

### 初始化

initQueue(cycleQueue \*q)

{

q->base = (ElemType \*)malloc(MAXSIZE \* sizeof(ElemType));

if( !q->base )

exit(0);

q->front = q->next = 0;

}

### 入队

入队主要两步完成：

1. 将值存入rear代表的位置
2. 增加长度

错误的写法：q->rear = q->rear+1; // 这个是静态队列的写法

正确的写法：q->rear = (q->rear+1) % MAXSIZE;

代码：

InsertQueue(cycleQueue \*q, ElemType e)

{

if( (q->rear+1)%MAXSIZE == q->front ) //队列满

return;

q->base[q->rear] = e;

q->rear = (q->rear+1) % MAXSIZE;

}

### 出队

循环队列出队的步骤：

错误写法：q->front = (q->front+1) // 这个是静态队列移动指针的实现

q->front = (q->front+1) %数组长度。

代码：

DeleteQueue(cycleQueue \*q, ElemType \*e)

{

if( q->front == q->rear ) //队列空

return;

\*e = q->base[q->front];

q->front = (q->front+1) % MAXSIZE;

}

### 队列判空

如果front和rear的值相等，则该队列就一定为空。

### 队列已满

判断循环队列是否已满的方法：

1. 增加一个标识符参数（一般不用该方法）；
2. 少用一个元素（通常使用这种方法）

**如果front和rear的值紧挨着，则表示循环队列已满**

C语言伪算法：

if((q->rear+1)%数组长度 == q->front)

{

已满

}

else

{

不满

}

## 链式队列

## 映射队列

参考：

映射队列：<http://mysql.taobao.org/monthly/2020/09/08/>

## 优先队列

优先队列即堆。

参考：

<https://blog.51cto.com/u_15939722/6007363>

## 单调队列

单调队列是一种特殊的队列数据结构，其主要特点是队列中的元素保持特定的单调性（递增或递减）。这种队列通常用于解决一些特定类型的问题，比如在滑动窗口中寻找最大值或最小值等。

单调队列通常支持以下两种操作：

入队操作：将元素加入队尾，但要维持队列的单调性，可能会删除队尾元素以满足单调性要求。

出队操作：从队头移除元素，但是仅当队头元素等于要出队的元素时才执行。

实现单调队列时，需要考虑保持队列的单调性。如果需要一个递增的单调队列，每次入队操作时，将队尾小于当前元素的元素出队，以保持队列递增的性质。相应地，如果需要一个递减的单调队列，每次入队操作时，将队尾大于当前元素的元素出队。

// 递增单调队列的入队操作

void insertIncreasing(deque<int>& dq, int x) {

// 删除队尾小于等于当前元素的元素，保持递增性质

while (!dq.empty() && dq.back() <= x) {

dq.pop\_back();

}

dq.push\_back(x); // 插入当前元素

}

// 单调队列的出队操作（出队首元素）

void dequeueOperation(deque<int>& dq, int x) {

if (!dq.empty() && dq.front() == x) {

dq.pop\_front(); // 如果队头元素等于要出队的元素，则出队

}

}

# 操作

## 入队

## 出队

## 存储

### 顺序存储/数组

#### 定义

typedef struct SeqQueue{

int data[MAXSIZE]; //也可以采用int \*data动态数组的形式存储

int front; // front头指针

int rear; // rear尾指针，若队列不为空，指向队列尾元素的下一个位置

}SeqQueue;

这里的定义是假设存储的是int类型数据，最理想的方式使用：

typedef int ElementType;

#### 初始化

int SqQueueInit(SqQueue \*q){

q->front=0;

q->rear=0;

return 0;

}

#### 销毁

void destroySeqQueue(SeqQueue \*q) {

// 如果data是动态分配的内存空间，释放它

// 假设data是静态数组，不需要释放内存

// 释放动态分配的内存空间

// 如果是静态数组，这一步可以忽略不做

// 如果是动态数组，需要释放内存

// 示例代码：

// free(q->data);

// 将队列指针置为NULL，表示队列已经被销毁

q = NULL;

}

#### 队列长度

int SqQueueLength(SqQueue \*q){

return (q->rear-q->front+MAX\_SIZE)%MAX\_SIZE;

}

当计算队列长度时，使用 `(q->rear - q->front + MAX\_SIZE) % MAX\_SIZE` 这个表达式，其含义和计算方法如下：

1、Rear 和 Front 之间的距离：

- (q->rear - q->front): 这部分计算了队列中最后一个元素和第一个元素之间的距离。

2、加上数组长度：

- + MAX\_SIZE：这个步骤是为了确保即使 rear 已经经过循环回到数组的开头，也能得到一个非负的值。

3、取模运算：

- % MAX\_SIZE：这个运算确保了最后的结果在数组大小范围内。它消除了任何可能超出数组大小的情况，将结果限制在 0 到 MAX\_SIZE - 1 之间。

综合来说，这个计算方法保证了即使 rear 已经经过循环回到数组开头，也能正确计算出队列的长度。例如，如果 rear 在数组的末尾，而 front 在数组的开头，则 (rear - front) 会给出一个负数，加上 MAX\_SIZE 后变成一个非负数，最后再进行取模运算，确保了计算出的长度是准确的，并且处于合理的范围内（0 到 MAX\_SIZE - 1）。

注意：

上述方法使用了取模运算 (q->rear - q->front + MAX\_SIZE) % MAX\_SIZE，这是因为队列的循环性质。在循环队列中，当 rear 指针超过数组的末尾时，它会回到数组的开头。所以 (q->rear - q->front + MAX\_SIZE) % MAX\_SIZE 的计算可以确保即使 rear 被循环到数组开头，也能正确计算出队列的长度。

如下的方法：

int SqQueueLength(SqQueue \*q){

return (q->rear-q->front+1);

}

上述方法 return (q->rear - q->front + 1); 并未考虑到队列的循环性质。这种方式只计算了 rear 和 front 之间的距离，但没有处理队列循环的情况，可能导致错误的长度计算。如果 rear 已经循环到数组开头，而 front 还在数组中间，这种简单相减的方法就会得到错误的队列长度。

因此，第一个方法通过取模运算考虑了队列循环的情况，可以更准确地计算出循环队列的长度。

#### 获取元素

int SeqQueueGetElement(SeqQueue \*q,int \*e){

\*e=q->data[q->front];

q->front--;

}

#### 入队

思路：插入元素需要更新rear变量

代码：

int SqQueueEnQueue(SqQueue \*q, int \*element){

if((q->rear+1)%MAX\_SIZE == q->front){

return -1;

}

q->data[q->rear]=\*element;

q->rear=(q->rear+1)%MAX\_SIZE;

return 0;

}

#### 出队

思路：删除元素需要判断是否为空（即front是都等于rear），然后更新front成员变量

代码：

int SqQueueDeQueue(SqQueue \*q, int \*element){

if(q->front == q->rear){

//队列为空（队列为空使用空余一个元素的方式，不使用flag）

return -1;

}

\*element = q->data[q->front];//将队头元素赋值给element

q->front = (q->front+1)%MAX\_SIZE;

//将front指针向后移动一个位置，如到最后则转到数组头部

// 这里没有直接将q->front++也是考虑了循环队列的情况

return 0;

}

### 链式存储/链表

注：队列重点掌握链表存储。

#### 定义

typedef struct LinkQueueNode{

int data; //存储节点的数据信息

struct LinkQueueNode \*next;

}LinkQueueNode, \*LinkQueuePtr;

注：上述是定义一个数据节点，下面才是真正定义数据结构特有属性。

typedef struct LinkQueue{

LinkQueuePtr front,rear;

}LinkQueue;

说明：LinkQueuePtr与LinkQueue的定义不同，LinkQueuePtr p表示指针，而LinkQueuePtr \*q才表示指针！

或者：

typedef struct Node {

int data; // 节点中存储的数据

struct Node \*next; // 指向下一个节点的指针

} Node;

typedef struct LinkedQueue {

Node \*front; // 指向队列头部的指针

Node \*rear; // 指向队列尾部的指针

} LinkedQueue;

#### 初始化

initQueue(LinkQueue \*q)

{

q->front = q->rear = (LinkQueuePtr)malloc(sizeof(LinkQueueNode));

if( !q->front)

exit(0);

q->front->next = NULL; //头结点指针域设置为NULL

}

#### 获取元素

思路：获取元素只能获取队尾和队首的

#### 入队

思路：插入新元素，首先申请一个新节点，把新节点（此时还是孤立的节点）的成员变量（即data和next），然后更新原来节点rear与新节点的连接关系以及队列的rear。

代码：

int LinkQueueEnQueue(LinkQueue \*q, int \*element){

LinkQueuePtr p = (LinkQueuePtr)malloc(sizeof(LinkQueueNode));

if(NULL == p) exit(0);

p->data=\*element;

p->next=NULL;//新节点的下一个节点为NULL

//更新新节点的参数，接下来更新整个链表的成员变量，即front和rear

q->rear->next=p;

//操作队尾，原来的rear指向现在的新节点

q->rear=p;

//更新rear节点（前面是更新节点之间的指向关系）

return 0；

}

注：注意定义LinkQueuePtr p而不是LinkQueuePtr \*p，因为LinkQueuePtr本身就是指针类型。

或：

int LinkQueueEnQueue(LinkQueue \*q, int \*element) {

// 创建新节点

Node \*newNode = (Node \*)malloc(sizeof(Node));

if (newNode == NULL) {

// 内存分配失败

return 0; // 返回 0 表示入队失败

}

// 将数据存储到新节点中

newNode->data = \*element;

newNode->next = NULL; // 新节点将成为队列尾部，因此其next指针为NULL

if (q->rear == NULL) {

// 如果队列为空，新节点即是队列头部又是队列尾部

q->front = newNode;

q->rear = newNode;

} else {

// 否则将新节点连接到队列尾部，并更新rear指针

q->rear->next = newNode;

q->rear = newNode;

}

return 1; // 返回 1 表示入队成功

}

#### 出队

int LinkQueueDeQueue(LinkQueue \*q, int \*element){

LinkQueuePtr p;//只需要声明变量而不需要实际分配内存空间

if(q->front==q->rear){

return -1;

}

//更新新节点的成员变量

p=q->front->next;//该节点成了最新的头结点

\*element = p->data;//头结点的数据

q->front->next=p->next;

if(q->rear==p){

q->rear=q->front;

}

free(p);

return 0;

}

或：

int LinkQueueDeQueue(LinkQueue \*q, int \*element) {

if (q->front == NULL) {

// 队列为空，无法执行出队操作

return 0; // 返回 0 表示出队失败

}

// 将头部节点数据传出

\*element = q->front->data;

Node \*temp = q->front;

q->front = q->front->next; // 移动front指针到下一个节点（这里不能是temp，后面会释放temp）

// 如果队列只有一个节点，出队后队列为空，需要更新rear指针为NULL

if (q->front == NULL) {

q->rear = NULL;

}

free(temp); // 释放出队节点的内存

return 1; // 返回 1 表示出队成功

}

#### 销毁

DestroyQueue(LinkQueue \*q)

{

while(q->front){

q->rear = q->front->next;

free(q->front);

q->front = q->rear;

}

}

或：

void DestroyQueue(LinkQueue \*q) {

Node \*current = q->front;

while (current != NULL) {

Node \*temp = current;

current = current->next;

free(temp); // 释放节点内存

}

q->front = NULL;

q->rear = NULL;

}

# 实现

与栈实现类似，队列的实现方式包括三种：

1. 用简单的循环数组实现
2. 用动态循环数据实现
3. 用链表实现

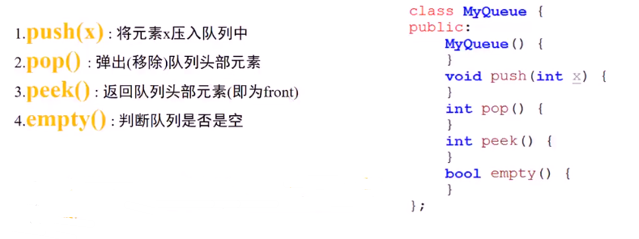
## 简单数组

## 动态数组

## 链表

## 栈

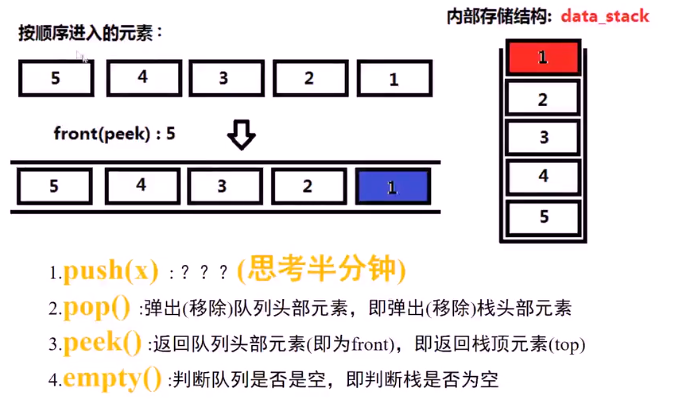
设计一个队列，支持基本的队列操作，这个队列的内部存储数据的结构为栈，栈的方法只能包括push、top、pop、size、empty等标准的栈方法。

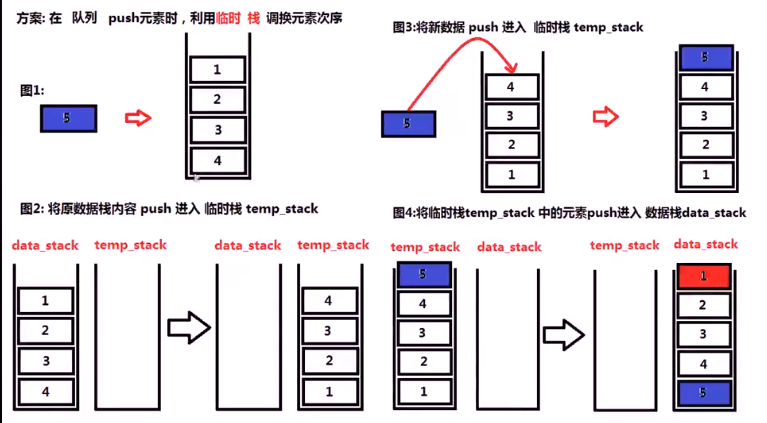


注：Leetcode 232

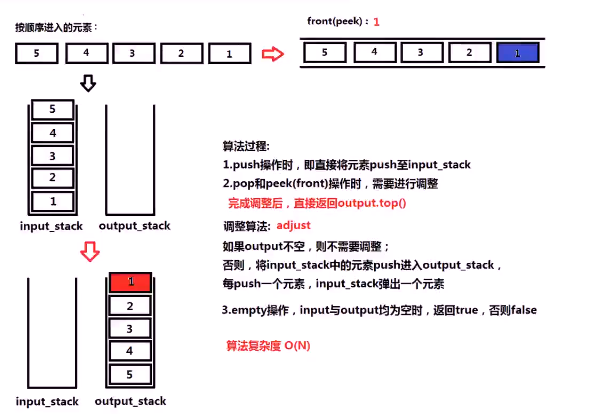
**分析：**

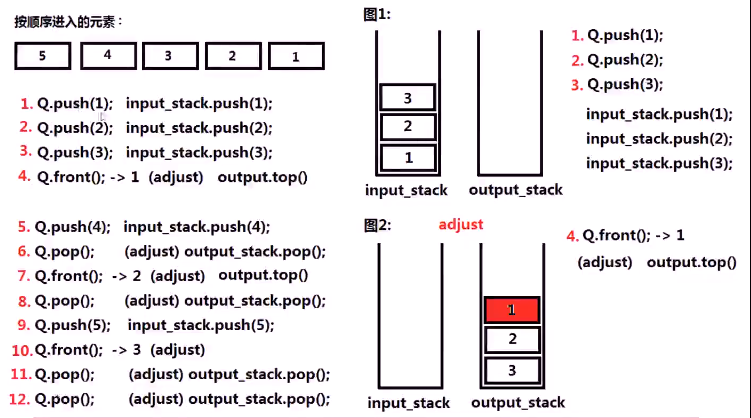
**方法一：**

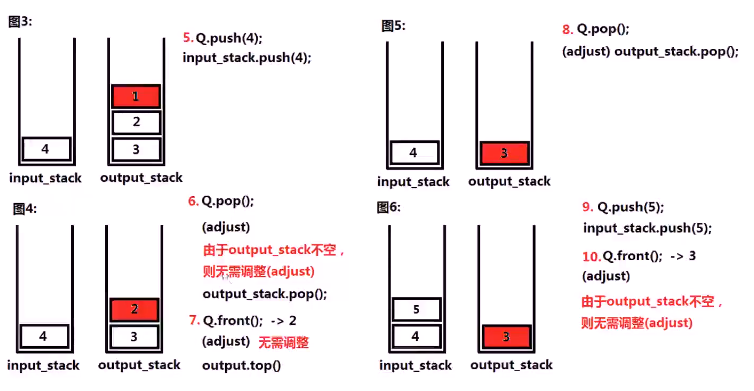




**方法二：**



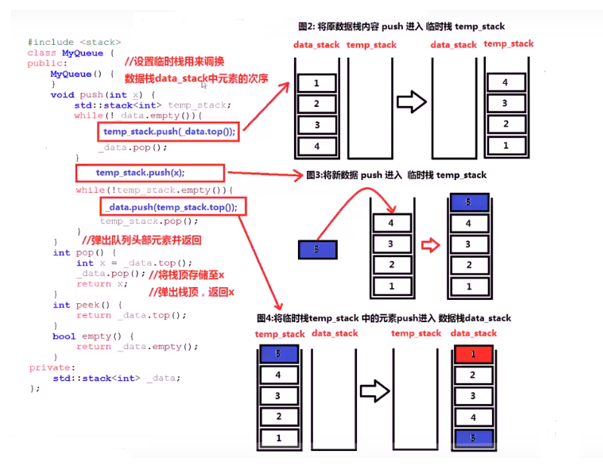






**代码：**

**方法一：**



**方法二：**



**测试：**

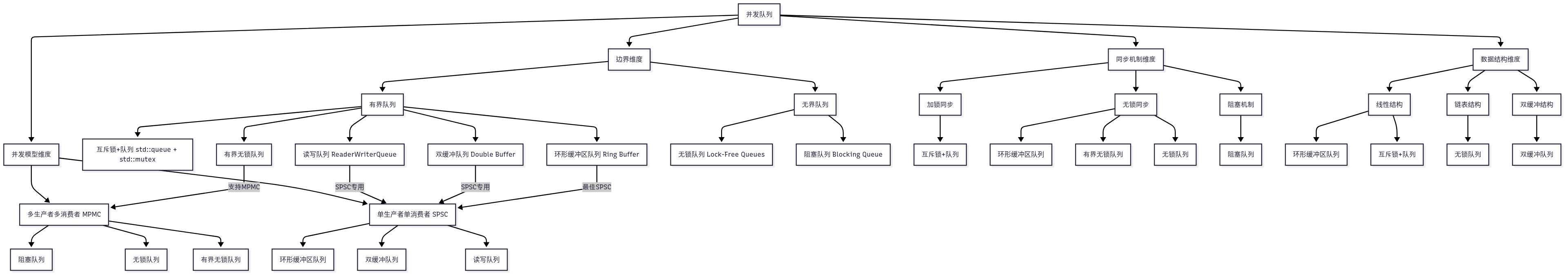


# 并发队列

## 概述



## 分类



### 按边界维度

#### 有界队列 (Bounded Queues)

固定容量

需处理满队列情况

代表类型：

- 互斥锁+队列 (std::queue + std::mutex)

- 环形缓冲区队列 (Ring Buffer)

- 有界无锁队列

- 双缓冲队列 (Double Buffer)

- 读写队列 (ReaderWriterQueue)

#### 无界队列 (Unbounded Queues)

动态扩容

无满队列限制（除内存耗尽）

代表类型：

- 无锁队列 (Lock-Free Queues)

- 阻塞队列 (Blocking Queue)

### 按并发模型维度

#### SPSC (单生产者单消费者)

专为点对点通信优化

代表类型：

- 读写队列

- 双缓冲队列

- 环形缓冲区队列

#### MPMC (多生产者多消费者)

支持多对多通信

代表类型：

- 有界无锁队列

- 无锁队列

- 阻塞队列

- 互斥锁+队列

### 按同步机制维度

#### 加锁同步

使用互斥锁保护临界区

代表：std::queue + std::mutex

#### 无锁同步

基于原子操作/CAS

代表：

- 环形缓冲区队列

- 有界无锁队列

- 无锁队列

- 读写队列

- 双缓冲队列

#### 阻塞机制

基于条件变量等待

代表：阻塞队列

### 按数据结构维度

#### 线性结构 (数组)

内存连续

缓存友好

代表：

- 环形缓冲区队列

- 互斥锁+队列（当使用std::vector）

#### 链表结构

动态节点

灵活但缓存不友好

代表：无锁队列（链表实现）

#### 双缓冲结构

双数组交替使用

代表：双缓冲队列

## 互斥锁+队列/std::queue + std::mutex

### 原理

原理：操作队列时，先获取锁，保证操作的原子性和线程安全。

### 特点

优点：实现简单，几乎支持所有数据类型。

缺点：高并发时锁竞争严重，性能下降，且线程阻塞。

### 应用场景

场景：低并发或对性能要求不高的场景。

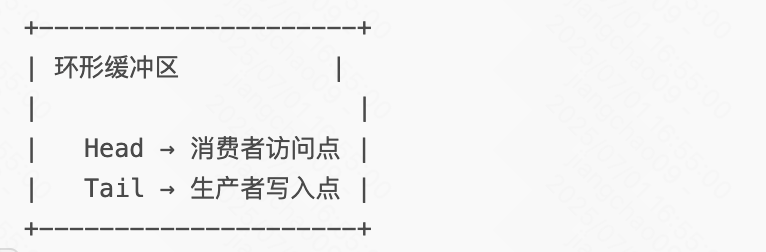
### 经典实现

## 环形缓冲区队列/Ring Buffer/Circular Queue

### 原理

原理：用固定大小数组实现环形缓冲，通过移动头尾指针操作队列。

环形缓冲区队列一般适用于单生产者-单消费者模式：



**在SPSC模型中：**

- 生产者独占修改tail指针

- 消费者独占修改head指针

- 无竞争：两个指针永远不会被同时修改

**在MPMC模型中：**

- 多个生产者同时竞争修改tail

- 多个消费者同时竞争修改head

- 需要复杂同步机制解决冲突

**技术挑战分析：**

1、队列状态判断冲突

判断队列空/满需要原子访问两个指针：

// 判断队列满的经典条件

bool is\_full() {

return (tail + 1) % size == head; // 非原子操作

}

在MPMC中，这个判断需要原子保护，否则会产生竞态条件。

2、指针更新原子性

cpp

// SPSC的简单更新

tail = (tail + 1) % size; // 单线程操作安全

// MPMC需要原子更新

std::atomic<size\_t> tail;

...

size\_t current = tail.load();

while(!tail.compare\_exchange\_weak(current, (current+1)%size)) {

// 循环直到CAS成功

}

3、缓存行伪共享

struct RingBuffer {

size\_t head; // 消费者使用

size\_t tail; // 生产者使用

// ... // 两者在同一缓存行

};

在MPMC中：

生产者修改tail → 使消费者缓存中的head失效

消费者修改head → 使生产者缓存中的tail失效

导致缓存一致性风暴（Cache Coherence Storm）

### 特点

优点：极低延迟，缓存友好，适合单生产者单消费者场景。

缺点：不支持多生产者或多消费者，容量固定。

优势与局限：



### 应用场景

场景：日志系统、实时数据流处理。

### 经典实现

template<typename T, size\_t Capacity>

class RingBuffer {

std::array<T, Capacity> buffer;

alignas(64) std::atomic<size\_t> head{0};

alignas(64) std::atomic<size\_t> tail{0};

public:

bool try\_push(T item) {

size\_t currentTail = tail.load(std::memory\_order\_relaxed);

size\_t nextTail = (currentTail + 1) % Capacity;

if(nextTail == head.load(std::memory\_order\_acquire))

return false; // 满

buffer[currentTail] = std::move(item);

tail.store(nextTail, std::memory\_order\_release);

return true;

}

bool try\_pop(T& item) {

size\_t currentHead = head.load(std::memory\_order\_relaxed);

if(currentHead == tail.load(std::memory\_order\_acquire))

return false; // 空

item = std::move(buffer[currentHead]);

head.store((currentHead + 1) % Capacity, std::memory\_order\_release);

return true;

}

};

### 工业级实现

#### SPSC Queue

#### Linux kfifo

Linux内核kfifo：SPSC环形缓冲区

#### DPDK rte\_ring

DPDK rte\_ring：支持SPSC/MPMC模式

#### Java Disruptor

Java Disruptor：SPSC为基础的多阶段流水线

## 有界无锁队列

### 原理

原理：无锁环形缓冲区，内存固定，支持多生产多消费。

有界无锁队列能够实现多生产者多消费者（MPMC）模型的核心在于其通过原子操作替代锁、分离竞争域以及优化内存访问来解决并发冲突。以下是其关键技术原理及实现机制：

* 核心问题：多线程竞争的本质

在MPMC场景下，传统队列的瓶颈在于：

1、指针竞争：多个生产者同时竞争尾指针（tail），多个消费者同时竞争头指针（head）。

2、状态判断冲突：判空/判满需原子读取两个指针（head和tail），易引发竞态条件。

3、伪共享（False Sharing）：频繁修改的指针位于同一缓存行，导致缓存一致性风暴。

* 关键技术：解决MPMC冲突的四大设计

1、指针分离与区域划分

- 生产区域与消费区域分离：

- 生产者独占操作：p\_head（生产头指针）和p\_tail（生产尾指针）。

- 消费者独占操作：c\_head（消费头指针）和c\_tail（消费尾指针）。

- 通过四指针隔离生产者与消费者的竞争域，避免enq-deq竞争。

- 示例（DPDK方案）：

- 生产者通过CAS竞争p\_head分配槽位，按序提交后收缩p\_tail。

- 消费者类似操作c\_head和c\_tail。

2、高效原子指令替换锁

- CAS（Compare-And-Swap）：传统方案使用CAS更新指针，但开销大。

- FAA（Fetch-And-Add）优化：

- 西北工业大学专利方案使用FAA指令批量分配槽位（如local p\_next = p\_head + M），减少CAS争用。

- 引入版本号机制（Versioning）检测指针越界，通过回滚（Rollback）处理冲突。

- 性能对比：

| 指令类型 | 开销（时钟周期） | 适用场景 |

|----------|------------------|------------------|

| CAS | 20-100 | 精细竞争控制 |

| FAA | 5-20 | 批量槽位分配 |

3、乱序提交与批量处理

- 乱序Catchup机制：

- 传统DPDK需生产者按序收缩p\_tail，限制并发性。

- 改进方案允许生产者乱序更新p\_tail，仅当p\_tail版本号匹配时提交，提升吞吐量。

- 批量操作：

- 支持单元素（Single）、固定数量（Fixed）、批量（Bulk）入队/出队。

- 示例：生产者一次性申请连续槽位（如`M`个元素），减少原子操作次数。

4、内存布局优化

- 缓存行对齐：

- 四指针（p\_head, p\_tail, c\_head, c\_tail）按CPU缓存行（通常64字节）对齐，消除伪共享。

- 示例：使用alignas(64)修饰指针。

- 环形数组预分配：

- 队列容量固定为2的整数幂（如1024），通过位运算index & (size-1)替代取模，加速索引计算。

* 性能优势对比



* 典型实现流程（以入队为例）

1、申请槽位：

- 线程本地拷贝c\_tail和p\_head，计算可用空间：

avail\_count = capacity + c\_tail - p\_head。

2、冲突检测：

- 若`p\_head`越界（超过c\_tail），触发回退（Rollback）机制。

3、更新生产头指针：

- 使用FAA更新p\_head：local p\_next = p\_head + M。

- CAS竞争更新全局p\_head。

4、写入数据：

- 将数据填入已分配的连续槽位。

5、提交生产尾指针：

- 当p\_tail == local p\_next时，CAS更新p\_tail，否则等待匹配。

* 总结

有界无锁队列实现MPMC的核心是通过分离竞争域（生产/消费指针）、高效原子指令（FAA替代CAS）、批量处理及内存优化（缓存行对齐）解决多线程竞争问题。西北工业大学等方案进一步引入版本号机制和乱序提交，将吞吐量提升至千万级操作/秒，成为高并发系统的底层基础设施首选。

### 特点

优点：高吞吐，低延迟，避免频繁内存分配。

缺点：容量固定，溢出时需要特殊处理。

- 容量固定：无法动态扩容，需预估峰值流量。

- 实现复杂度：需处理回滚、版本号等边界条件。

与环形缓冲区队列区别：



与环形缓冲区队列联系：

- 两者都是有界队列（固定容量）

- 都可以实现无锁并发

- 都使用原子操作进行同步

- 都适用于生产者-消费者模式

### 应用场景

场景：高性能通信，网络IO缓冲。

### 开源实现

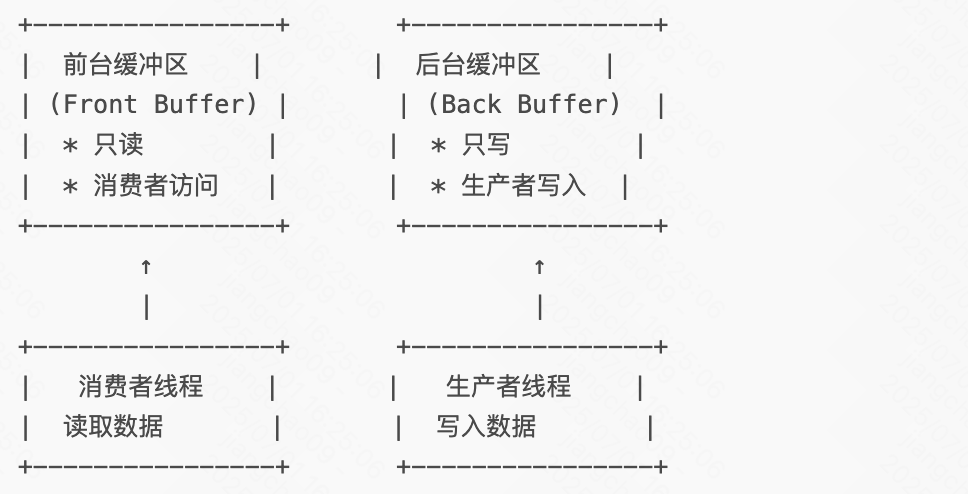
#### folly::MPMCQueue

#### Boost.Lockfree::queue

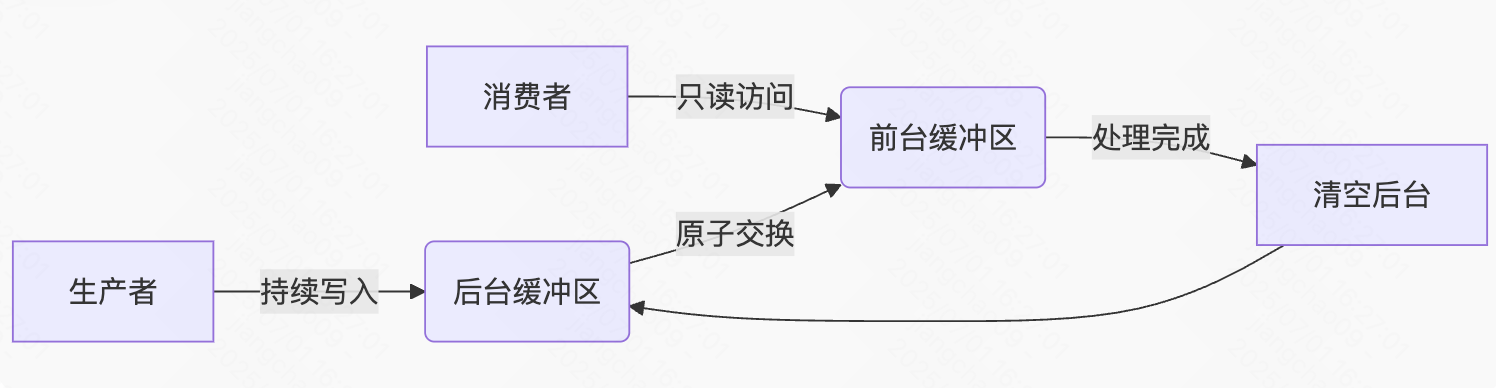
## 双缓冲队列/Double Buffer

### 原理

双缓冲队列是一种生产者-消费者分离的并发数据结构，通过维护两个独立缓冲区实现高效数据交换：



工作流程：



1、生产者阶段：生产者持续写入后台缓冲区

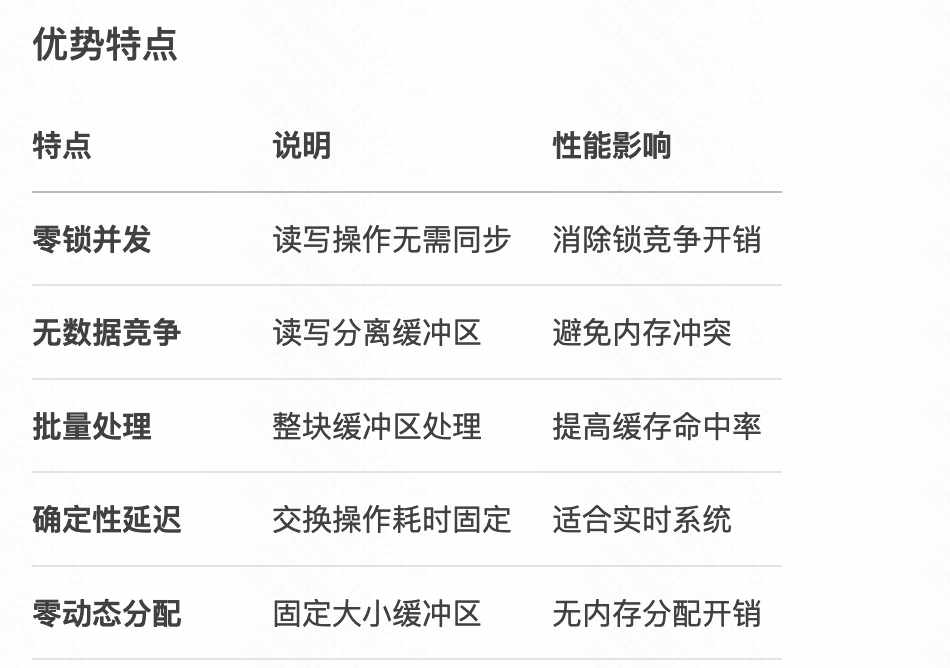
2、消费者阶段：消费者从前台缓冲区读取数据

3、交换阶段：当消费者完成处理时，原子性地交换前后台缓冲区

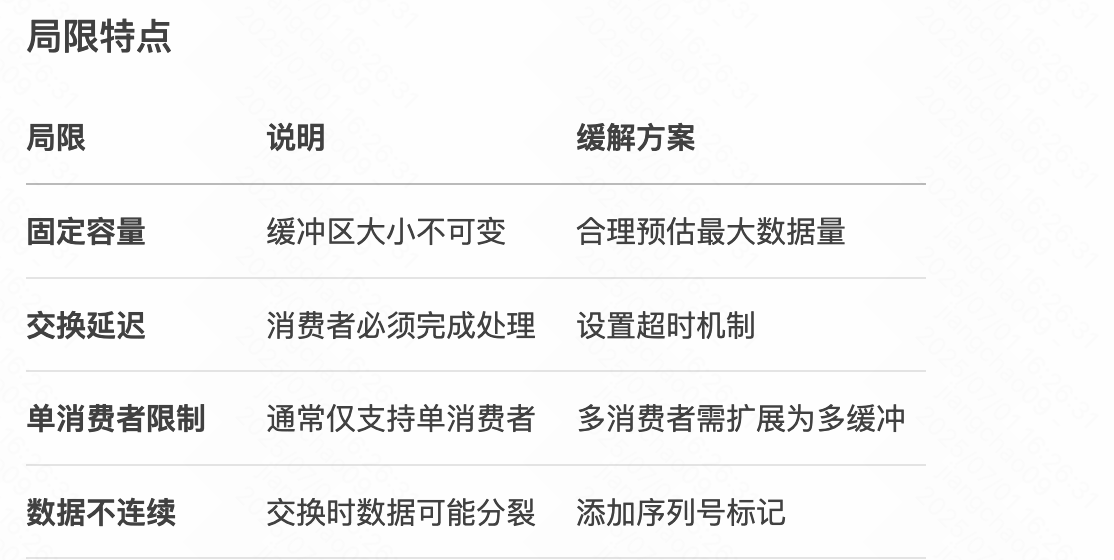
swap(frontBuffer, backBuffer);

### 特点

优势：



缺点：



### 性能优化

1、三重缓冲：解决交换时的卡顿问题

class TripleBuffer {

std::vector<T> buffers[3];

std::atomic<int> front{0}; // 消费者访问

std::atomic<int> middle{1}; // 空闲缓冲区

std::atomic<int> back{2}; // 生产者写入

public:

void swap() {

// 1. 将back与middle交换

int newMid = back.exchange(middle);

// 2. 将middle与front交换

front.store(middle.exchange(front));

// 3. 更新back为原空闲缓冲区

back.store(newMid);

}

};

2、无锁交换优化

void lockfree\_swap() {

int currentFront = frontIndex.load(std::memory\_order\_relaxed);

int newFront = 1 - currentFront;

// 原子比较交换

while(!frontIndex.compare\_exchange\_weak(

currentFront, newFront,

std::memory\_order\_release,

std::memory\_order\_relaxed))

{

// CAS失败时重试

}

}

3、部分交换机制

void partial\_swap(size\_t maxItems) {

auto& front = get\_front();

auto& back = get\_back();

// 仅交换部分数据

size\_t swapCount = std::min(maxItems, back.size());

front.assign(back.begin(), back.begin() + swapCount);

back.erase(back.begin(), back.begin() + swapCount);

}

4、时间戳同步

struct TimestampedBuffer {

std::vector<T> data;

uint64\_t timestamp; // 最后更新时间

};

// 消费者选择最新缓冲区

const auto& read() const {

return (buffers[0].timestamp > buffers[1].timestamp)

? buffers[0] : buffers[1];

}

### 应用场景

图形渲染（前缓冲显示，后缓冲绘制）

实时数据采集

数据生产消费速率差异大、需要批量处理、实时性要求高的系统

### 经典实现

class DoubleBuffer {

std::vector<Data> buffers[2];

std::atomic<int> frontIndex{0};

public:

// 生产者写入后台缓冲区

void write(const Data& data) {

buffers[1 - frontIndex].push\_back(data);

}

// 消费者交换缓冲区

void swap() {

frontIndex.store(1 - frontIndex, std::memory\_order\_release);

buffers[1 - frontIndex].clear();

}

// 读取前台缓冲区

const std::vector<Data>& read() const {

return buffers[frontIndex.load(std::memory\_order\_acquire)];

}

};

### 开源实现

#### C++11

#include <vector>

#include <atomic>

#include <mutex>

#include <condition\_variable>

template<typename T>

class DoubleBuffer {

std::vector<T> buffers[2]; // 双缓冲区

std::atomic<int> frontIndex{0}; // 前台索引

std::mutex swapMutex; // 交换锁（可选）

std::condition\_variable swapCond;

bool swapping{false}; // 交换标志

public:

// 写入后台缓冲区

void write(const T& item) {

// 无需锁 - 后台缓冲区独占写入

buffers[1 - frontIndex].push\_back(item);

}

// 交换缓冲区（原子操作）

void swap() {

// 可选：等待当前交换完成

std::unique\_lock<std::mutex> lock(swapMutex);

swapCond.wait(lock, [this]{ return !swapping; });

swapping = true;

// 原子交换前台索引

frontIndex.store(1 - frontIndex, std::memory\_order\_release);

// 清空新的后台缓冲区

buffers[1 - frontIndex].clear();

swapping = false;

swapCond.notify\_all();

}

// 读取前台缓冲区（只读）

const std::vector<T>& read() const {

return buffers[frontIndex.load(std::memory\_order\_acquire)];

}

// 批量获取数据（避免复制）

const std::vector<T>& consume() {

swap();

return read();

}

};

## 无锁队列/Lock-Free Queues

### 原理

原理：链表结构，使用**CAS操作无锁**更新头尾指针，实现并发安全。

- 使用原子操作（CAS, Fetch-And-Add）替代锁

- 通过内存屏障保证顺序一致性

- 基于链表或环形缓冲区实现

有界无锁队列与无锁队列（Lock-Free Queues）均属于并发编程中的高性能数据结构，但其设计目标、实现机制和适用场景存在显著差异。以下是二者的核心区别与联系分析：

* 核心联系

1、无锁特性

两者均通过原子操作（如CAS）替代传统锁，避免线程阻塞，提升并发吞吐量。

2、线程安全

均支持多线程环境下的安全操作，依赖内存屏障或缓存行对齐保证数据一致性。

3、适用生产者-消费者模型

用于解耦生产与消费流程，提升系统并行效率。

* 核心区别



* 关键机制对比

1、有界无锁队列的优化技术

- 指针分离：生产者操作p\_head/p\_tail，消费者操作c\_head/c\_tail，隔离竞争域。

- 批量处理：支持单次申请多个槽位，减少原子操作次数（如Disruptor）。

- 内存布局：缓存行对齐（alignas(64)）消除伪共享。

2、无锁队列的弱一致性挑战

- 遍历不确定性：迭代时修改不抛异常，但可能读到过期数据（弱一致性）。

- 动态扩容：无界设计避免满队列阻塞，但可能引发OOM（需监控资源）。

* 性能场景示例

- 有界队列：环形缓冲区（如Disruptor）在单生产者单消费者（SPSC）模式下延迟可低于50ns，但扩展到MPMC时性能下降10倍。

- 无锁队列：ConcurrentLinkedQueue在多生产者多消费者（MPMC）场景下吞吐量高，但99%延迟可能达微秒级。

* 选型建议

1、优先有界无锁队列：

需确定性延迟（实时系统）、内存受限（嵌入式）或批量处理（网络包处理）的场景。

2、优先无锁队列：

需动态扩容（任务队列）、弱一致性可接受（日志采集）或写密集型（高并发提交）的场景。

* **总结**

- 本质联系：两者均以原子操作实现无锁并发，但有界队列通过空间预换时间，无界队列通过动态性换灵活性。

- 趋势结合：现代系统常混合使用（如流水线：SPSC有界队列 + 无界任务队列）。

### 特点

优点：无阻塞，性能优异，支持多生产者多消费者。

缺点：实现复杂，ABA问题、内存回收（如引用计数、Hazard Pointer）处理难。

性能特点：

- 吞吐量：百万级操作/秒

- 延迟：<100ns (单生产者单消费者)

- 适用场景：高频交易、网络数据包处理

### 应用场景

场景：高性能系统，如数据库内部、通信中间件。

### 经典实现

template<typename T>

class LockFreeQueue {

struct Node {

std::atomic<Node\*> next;

T data;

};

std::atomic<Node\*> head;

std::atomic<Node\*> tail;

public:

void enqueue(T value) {

Node\* newNode = new Node{nullptr, std::move(value)};

Node\* oldTail = tail.load(std::memory\_order\_relaxed);

while(!tail.compare\_exchange\_weak(

oldTail, newNode,

std::memory\_order\_release,

std::memory\_order\_relaxed))

{

// CAS失败时重试

}

oldTail->next.store(newNode, std::memory\_order\_release);

}

bool dequeue(T& result) {

Node\* oldHead = head.load(std::memory\_order\_relaxed);

Node\* nextNode;

do {

if(!oldHead) return false; // 空队列

nextNode = oldHead->next.load(std::memory\_order\_acquire);

} while(!head.compare\_exchange\_weak(

oldHead, nextNode,

std::memory\_order\_release,

std::memory\_order\_relaxed));

result = std::move(nextNode->data);

delete oldHead;

return true;

}

};

### 开源实现

#### MSQueue/Michael-Scott Queue

## 阻塞队列/BlockingConcurrentQueue

### 原理

原理：无锁队列基础上结合条件变量，实现空时阻塞等待，满时阻塞生产。

### 特点

优点：满足生产者消费者平衡，避免忙等待浪费CPU。

缺点：线程切换带来开销，响应时间不如纯无锁队列。

**核心特性：**

- 多生产者多消费者（MPMC）支持

- 阻塞和非阻塞API混合

- 批量操作优化

- 动态扩容

### 性能优化技术

1、细粒度锁设计：

- 每个生产者有独立子队列

- 消费者轮询所有子队列

2、缓存行对齐：

struct alignas(64) CacheLineAlignedCounter {

std::atomic<size\_t> count;

};

3、指数退避策略：

unsigned retryCount = 0;

while(!try\_op()) {

if(retryCount++ > MAX\_RETRY) {

std::this\_thread::yield();

retryCount = 0;

}

}

4、批量操作：

- 减少原子操作次数

- 提高缓存利用率

### 应用场景

场景：任务调度器、线程池任务队列。

### 开源实现

#### BlockingConcurrentQueue

#include "blockingconcurrentqueue.h"

// 创建队列（可指定初始容量）

moodycamel::BlockingConcurrentQueue<int> queue(1024);

// 生产者线程

void producer() {

for(int i = 0; i < 1000; ++i) {

queue.enqueue(i); // 非阻塞入队

// 批量入队

int items[10] = {...};

queue.enqueue\_bulk(items, 10);

}

}

// 消费者线程

void consumer() {

int item;

while(true) {

// 阻塞等待

queue.wait\_dequeue(item);

// 非阻塞尝试

if(queue.try\_dequeue(item)) {

// 处理item

}

// 批量出队

int items[32];

size\_t count = queue.try\_dequeue\_bulk(items, 32);

if(count > 0) {

// 批量处理

}

}

}

#### TBB concurrent\_bounded\_queue

## 读写队列

### 原理

1、SPSC模型（Single Producer, Single Consumer）

- 严格限定仅一个线程写入、仅一个线程读取，消除多线程竞争。

- 写入顺序与读取顺序完全一致（严格FIFO），无需复杂同步机制。

2、无锁设计（Lock-Free）

- 通过原子操作（CAS/内存屏障）替代锁，避免线程阻塞。

- 示例：写入时原子更新tail指针，读取时原子更新head指针，二者互不干扰。

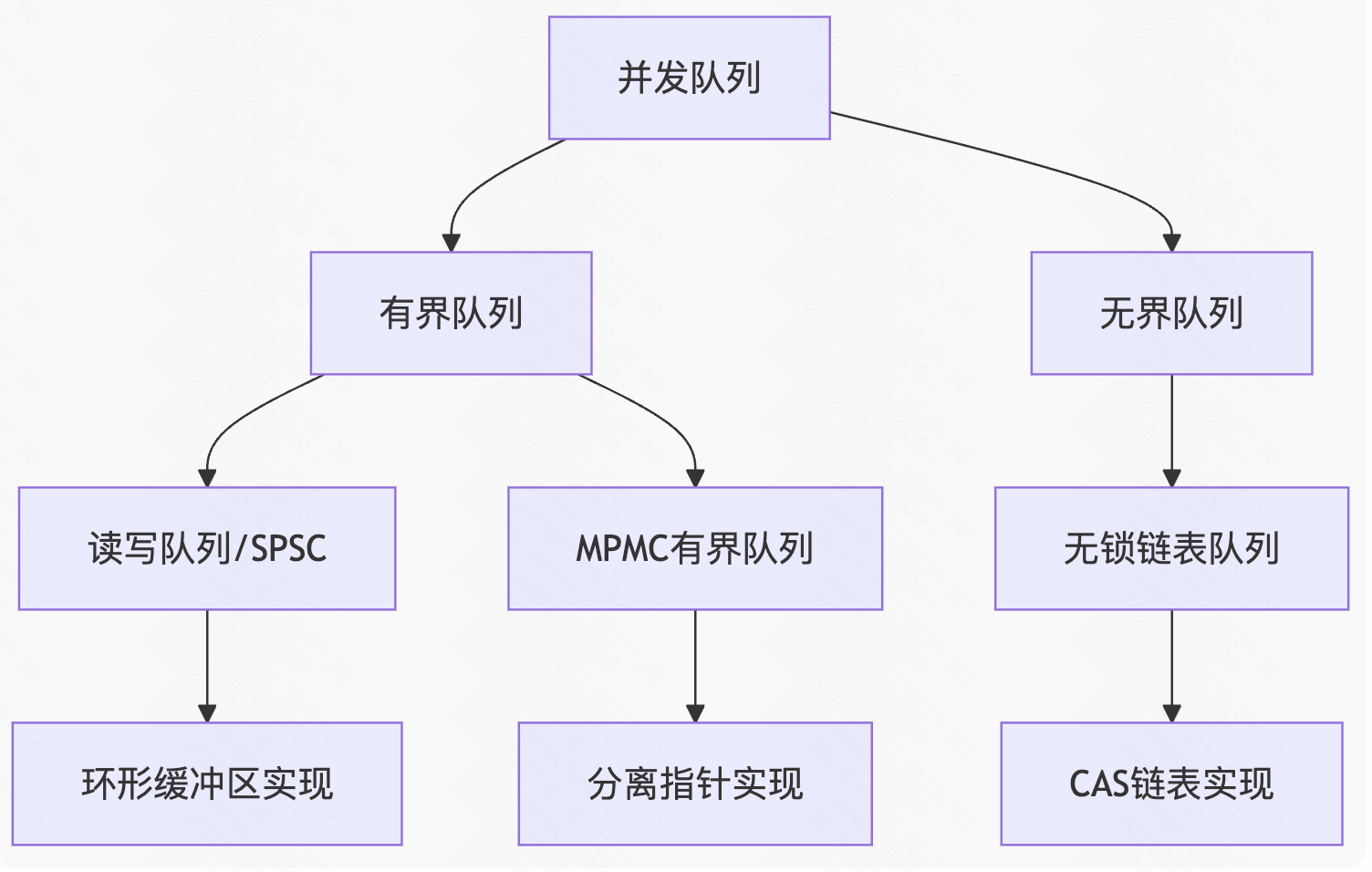
3、环形缓冲区（Ring Buffer）

- 底层使用预分配的连续数组，通过模运算实现循环复用。

- 优势：零动态内存分配，缓存局部性优（CPU缓存命中率高）。

读写队列（如ReaderWriterQueue）与其他并发队列的关系解析

读写队列（特指SPSC队列）是并发队列家族中的一个高度优化的特例，与其他并发队列既有联系又有区别。以下是其关系全景图：



* 核心关系详解

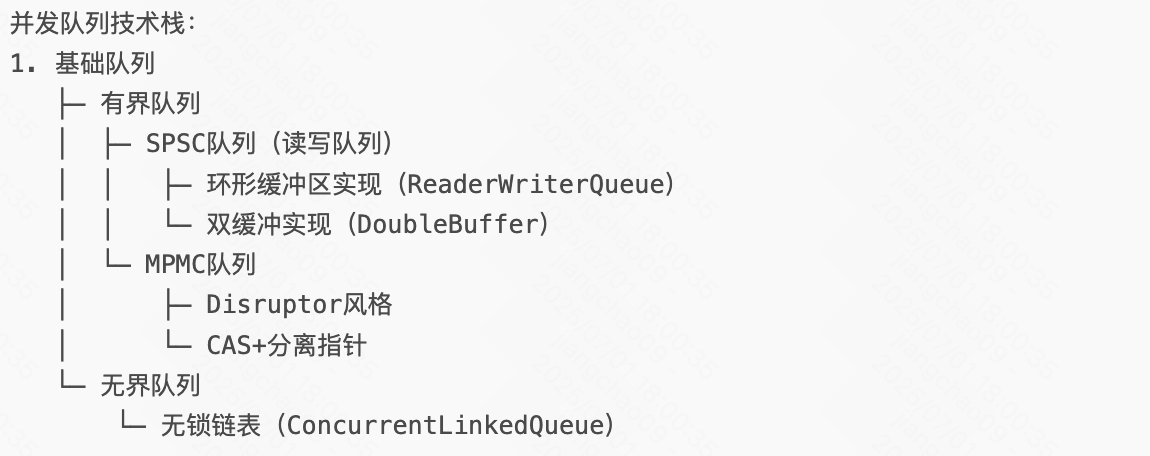
1、与通用并发队列的共性



2、作为SPSC队列的特殊性\*\*



3、在技术栈中的位置



* 核心区别对比

1. 竞争处理机制



2、内存访问模式

// SPSC队列（缓存友好）

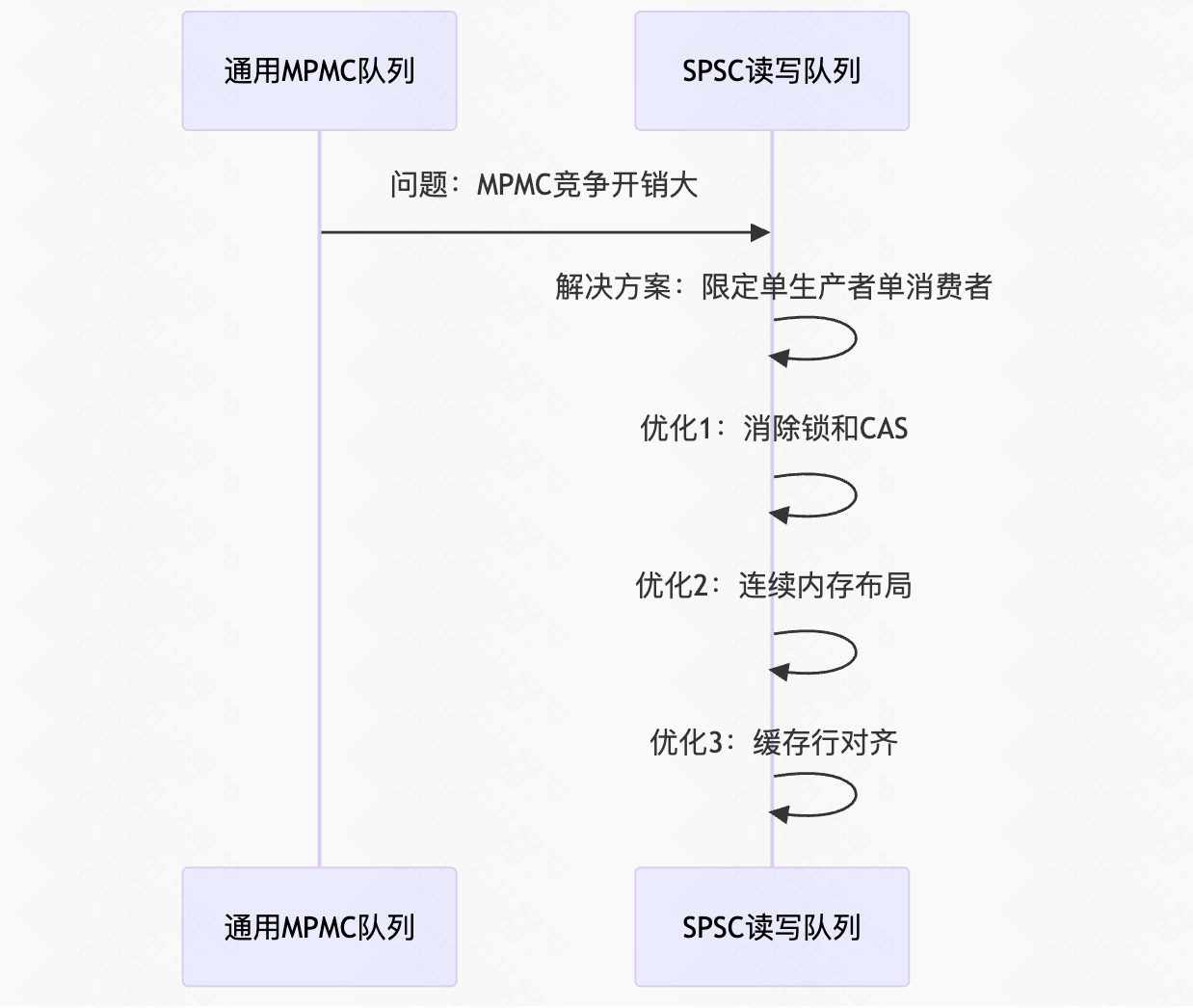
[元素0][元素1][元素2]... // 连续内存访问

// 链表队列（缓存不友好）

元素0 → 元素1 → 元素2 // 指针跳转访问

* 设计演进关系

1、从通用队列到SPSC优化

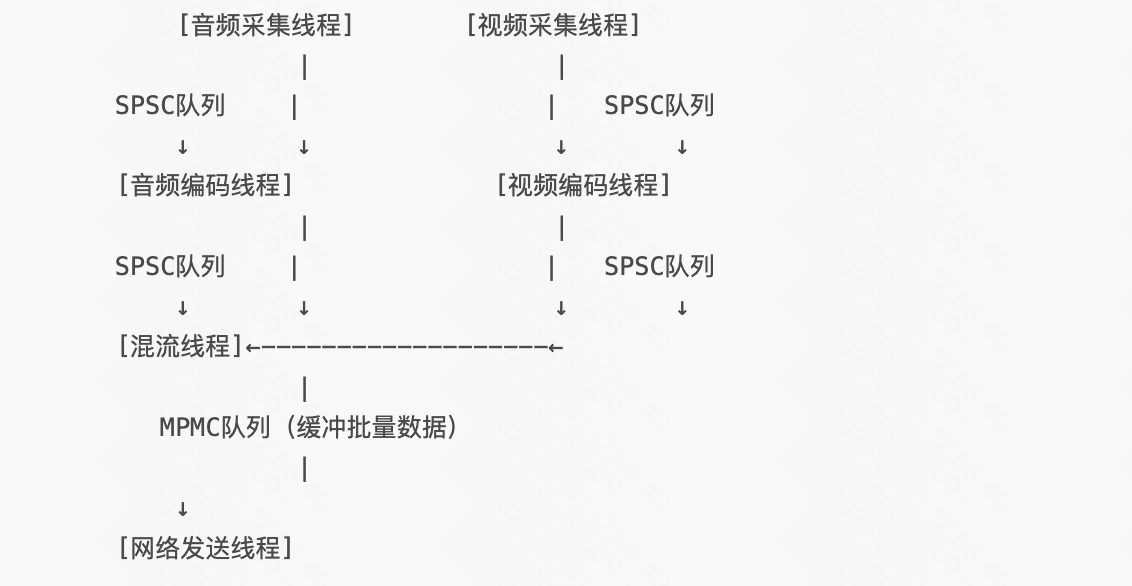


2、与其他SPSC实现的关系



* 工业系统中的协同应用

1、音视频处理流水线



2、金融交易系统优化

// 低延迟路径：SPSC队列

SPSCQueue<Order> realtimeQueue; // 生产者：风控线程 → 消费者：交易引擎

// 批量处理路径：MPMC队列

ConcurrentQueue<Report> batchQueue; // 多生产者 → 单消费者（报表生成）

3、游戏引擎数据流

[物理线程] → SPSC队列 → [动画线程] → SPSC队列 → [渲染线程]

↑ ↓

[输入线程] → SPSC队列 → [逻辑线程] → MPMC队列 → [AI线程]

* 总结：定位与价值

1、定位

SPSC读写队列是并发队列家族中的高性能特化分支，专为单生产者单消费者场景极致优化。

2、选型指南

| 场景 | 推荐队列类型 |

|------|--------------|

| 线程间点对点实时传输 | SPSC读写队列 |

| 多源数据聚合 | MPMC队列 |

| 大规模任务调度 | 无界队列 |

| 批量数据交换 | 双缓冲队列 |

### 特点



### 应用场景

1、实时数据处理

- 音视频流处理：生产者捕获音频帧，消费者实时编码（如FFmpeg流水线）。

- 高频交易：订单生成线程→风控线程，延迟要求纳秒级。

2、资源受限系统

- 嵌入式设备：无人机传感器数据采集（内存有限，预分配缓冲区）。

- 游戏引擎：物理计算线程→渲染线程，避免帧率波动。

3、高吞吐日志

- 日志写入线程→网络发送线程，批量提交减少I/O次数。

不适用场景：多生产者/多消费者（MPMC）需改用ConcurrentQueue或组合SPSC队列。

### 实现细节

// 简化版SPSC队列实现（基于环形缓冲区）

template<typename T, size\_t Capacity>

class SPSCQueue {

alignas(64) std::atomic<size\_t> head{0}; // 消费者独占缓存行

alignas(64) std::atomic<size\_t> tail{0}; // 生产者独占缓存行

T buffer[Capacity];

public:

bool try\_push(T item) {

size\_t current\_tail = tail.load(std::memory\_order\_relaxed);

size\_t next\_tail = (current\_tail + 1) % Capacity;

if (next\_tail == head.load(std::memory\_order\_acquire)) // 判满

return false;

buffer[current\_tail] = std::move(item);

tail.store(next\_tail, std::memory\_order\_release); // 释放语义

return true;

}

bool try\_pop(T& item) {

size\_t current\_head = head.load(std::memory\_order\_relaxed);

if (current\_head == tail.load(std::memory\_order\_acquire)) // 判空

return false;

item = std::move(buffer[current\_head]);

head.store((current\_head + 1) % Capacity, std::memory\_order\_release);

return true;

}

};

### 关键优化技术

1、缓存行对齐

alignas(64)隔离head与tail，避免伪共享（False Sharing）。

2、内存序控制

- 生产者：store(tail, std::memory\_order\_release)`确保写入先于指针更新。

- 消费者：load(head, std::memory\_order\_acquire)`确保读取后于指针加载。

3、动态扩容

enqueue()在容量不足时自动扩容（加倍策略），try\_enqueue保持固定容量。

## 对比





## 选型

1、超低延迟交易系统：

无锁队列（如Disruptor模式）

环形缓冲区（固定消息大小）

2、Web服务器任务队列：

// Nginx风格事件队列

moodycamel::BlockingConcurrentQueue<HttpRequest> requestQueue;

// 工作线程

void worker\_thread() {

HttpRequest req;

while(requestQueue.wait\_dequeue\_timed(req, 100ms)) {

process\_request(req);

}

}

3、实时音视频处理：

// 音频处理流水线

DoubleBuffer<AudioFrame> audioBuffer;

// 采集线程

void capture\_thread() {

while(running) {

auto frame = capture\_audio();

audioBuffer.write(frame);

}

}

// 处理线程

void process\_thread() {

while(running) {

audioBuffer.swap();

auto& frames = audioBuffer.read();

batch\_process(frames);

}

}

4、游戏引擎：

// 实体组件系统（ECS）

RingBuffer<EntityEvent, 1024> eventQueue;

// 物理系统 → 渲染系统

void physics\_step() {

for(auto& entity : entities) {

if(entity.moved) {

eventQueue.try\_push({MOVE\_EVENT, entity.id});

}

}

}

void render\_step() {

EntityEvent event;

while(eventQueue.try\_pop(event)) {

update\_render\_state(event);

}

}

## 优化

1、预分配内存池：

ObjectPool<Node> nodePool;

void enqueue(T value) {

Node\* newNode = nodePool.allocate();

newNode->data = std::move(value);

// ...入队操作...

}

2、线程本地缓存：

thread\_local std::vector<T> localBuffer;

void flush\_buffer() {

if(!localBuffer.empty()) {

queue.enqueue\_bulk(localBuffer.data(), localBuffer.size());

localBuffer.clear();

}

}

3、硬件亲和性优化：

void pin\_thread(int cpu\_id) {

cpu\_set\_t cpuset;

CPU\_ZERO(&cpuset);

CPU\_SET(cpu\_id, &cpuset);

pthread\_setaffinity\_np(pthread\_self(), sizeof(cpuset), &cpuset);

}

4、SIMD批处理：

void process\_bulk(T\* items, size\_t count) {

for(size\_t i = 0; i < count; i += 4) {

\_\_m128i data = \_mm\_load\_si128((\_\_m128i\*)&items[i]);

// SIMD处理...

}

}

最佳实践：根据应用场景选择队列类型，结合内存池和批处理技术可进一步提升性能2-5倍。对于99%的通用场景，BlockingConcurrentQueue提供了最佳平衡点。

## 总结

简单、易用：使用基于互斥锁的阻塞队列足够。

极限性能需求：无锁队列、环形缓冲区（尤其SPSC）更合适。

需要等待机制：使用带阻塞的并发队列。

内存管理复杂：无锁链表实现要慎重，需考虑ABA和内存回收。

# 应用

操作系统把优先级相等的任务按照到达的先后顺序安排在队列中

输入输出区接受键盘的输入就是按队列的形式输入和输出的

异步数据传输（文件IO、管道、socket）

用来决定每一位求助于客服中心的用户所需等待的时间

所有和时间有关的操作都有队列的影子

## 设计双端循环队列

**题目：**设计实现双端循环队列。

你的实现需要支持以下操作：

MyCircularDeque(k)：构造函数,双端队列的大小为k。

insertFront()：将一个元素添加到双端队列头部。如果操作成功返回true。

insertLast()：将一个元素添加到双端队列尾部。如果操作成功返回true。

deleteFront()：从双端队列头部删除一个元素。如果操作成功返回true。

deleteLast()：从双端队列尾部删除一个元素。如果操作成功返回true。

getFront()：从双端队列头部获得一个元素。如果双端队列为空，返回-1。

getRear()：获得双端队列的最后一个元素。如果双端队列为空，返回-1。

isEmpty()：检查双端队列是否为空。

isFull()：检查双端队列是否满了。

示例：

MyCircularDeque circularDeque = new MycircularDeque(3); // 设置容量大小为3

circularDeque.insertLast(1); // 返回 true

circularDeque.insertLast(2); // 返回 true

circularDeque.insertFront(3); // 返回 true

circularDeque.insertFront(4); // 已经满了，返回 false

circularDeque.getRear(); // 返回 2

circularDeque.isFull(); // 返回 true

circularDeque.deleteLast(); // 返回 true

circularDeque.insertFront(4); // 返回 true

circularDeque.getFront(); // 返回 4

提示：

所有值的范围为[1, 1000]

操作次数的范围为[1, 1000]

请不要使用内置的双端队列库。

**分析：**

head是头部待出队位置（获取头元素，即直接访问arr[head]）

从头插入，head是逆时针旋转（需要先减1）

从头删除，head是顺时针旋转

tail是尾部待入队位置，（获取尾元素，需要先将tail逆时针旋转1）

从尾插入，tail是顺时针旋转

从尾删除，tail是逆时针旋转

**代码：**

class MyCircularDeque {

private:

int \*m\_data;

int m\_head;

int m\_tail;

int m\_size;

int m\_capacity;

public:

/\*\* Initialize your data structure here. Set the size of the deque to be k. \*/

MyCircularDeque(int k) {

m\_data = new int[k];

m\_head = m\_tail = 0;

m\_size = 0;

m\_capacity = k;

}

/\*\* Adds an item at the front of Deque. Return true if the operation is successful. \*/

bool insertFront(int value) {

if(isFull())

return false;

m\_head--;

if(m\_head < 0)

{

m\_head = m\_capacity -1;

}

m\_data[m\_head] = value;

m\_size++;

return true;

}

/\*\* Adds an item at the rear of Deque. Return true if the operation is successful. \*/

bool insertLast(int value) {

if(isFull())

return false;

m\_data[m\_tail] = value; //先赋值，然后m\_tail++

m\_tail++;

if(m\_tail == m\_capacity)

{

m\_tail = 0;

}

m\_size++;

return true;

}

/\*\* Deletes an item from the front of Deque. Return true if the operation is successful. \*/

bool deleteFront() {

if(isEmpty())

return false;

m\_head++;

if(m\_head == m\_capacity)

{

m\_head = 0;

}

m\_size--;

return true;

}

/\*\* Deletes an item from the rear of Deque. Return true if the operation is successful. \*/

bool deleteLast() {

if(isEmpty())

return false;

m\_tail--;

if(m\_tail < 0)

{

m\_tail = m\_capacity-1;

}

m\_size--;

return true;

}

/\*\* Get the front item from the deque. \*/

int getFront() {

if(isEmpty())

return -1;

return m\_data[m\_head];

}

/\*\* Get the last item from the deque. \*/

int getRear() {

if(isEmpty())

return -1;

int index = m\_tail - 1;

if(index < 0)

{

index = m\_capacity - 1;

}

return m\_data[index];

}

/\*\* Checks whether the circular deque is empty or not. \*/

bool isEmpty() {

return (0 == m\_size);

}

/\*\* Checks whether the circular deque is full or not. \*/

bool isFull() {

return (m\_size == m\_capacity);

}

};

/\*\*

\* Your MyCircularDeque object will be instantiated and called as such:

\* MyCircularDeque\* obj = new MyCircularDeque(k);

\* bool param\_1 = obj->insertFront(value);

\* bool param\_2 = obj->insertLast(value);

\* bool param\_3 = obj->deleteFront();

\* bool param\_4 = obj->deleteLast();

\* int param\_5 = obj->getFront();

\* int param\_6 = obj->getRear();

\* bool param\_7 = obj->isEmpty();

\* bool param\_8 = obj->isFull();

\*/

注：Leetcode 641

## 设计循环队列

**题目：**

你的实现应该支持如下操作：

MyCircularQueue(k): 构造器，设置队列长度为 k 。

Front: 从队首获取元素。如果队列为空，返回 -1 。

Rear: 获取队尾元素。如果队列为空，返回 -1 。

enQueue(value): 向循环队列插入一个元素。如果成功插入则返回真。

deQueue(): 从循环队列中删除一个元素。如果成功删除则返回真。

isEmpty(): 检查循环队列是否为空。

isFull(): 检查循环队列是否已满。

**分析：**

**代码：**

class MyCircularQueue {

public:

/\*\* Initialize your data structure here. Set the size of the queue to be k. \*/

MyCircularQueue(int k) {

m\_head = 0;

m\_tail = 0;

m\_capacity = k;

m\_size = 0;

m\_arr = new int[k];

}

/\*Insert an element into the circular queue. Return true if the operation is successful. \*/

bool enQueue(int value) {

if (isFull())

return false;

m\_arr[m\_tail] = value;

++m\_tail;

if (m\_tail == m\_capacity)

{

m\_tail = 0;

}

++m\_size;

return true;

}

/\*Delete an element from the circular queue. Return true if the operation is successful. \*/

bool deQueue() {

if (isEmpty()) **//出队判空，入堆判满**

return false;

++m\_head; //出队要先++m\_head后判断是否越界

if (m\_head == m\_capacity)

{

m\_head = 0;

}

--m\_size;

return true;

}

/\*\* Get the front item from the queue. \*/

int Front() {

if (isEmpty())

return -1;

return m\_arr[m\_head];

}

/\*\* Get the last item from the queue. \*/

int Rear() {

if (isEmpty())

return -1;

//注意：最后元素的索引是m\_tail的上一个元素

int index = m\_tail - 1;

if (index < 0)

{

index = m\_capacity - 1;

}

return m\_arr[index];

}

/\*\* Checks whether the circular queue is empty or not. \*/

bool isEmpty() {

return m\_size == 0;

}

/\*\* Checks whether the circular queue is full or not. \*/

bool isFull() {

return m\_size == m\_capacity;

}

int size() {

return m\_size;

}

private:

int\* m\_arr;

int m\_head; //待出队位置

int m\_tail; //待入队位置

int m\_size; //元素个数

int m\_capacity; //容量固定长度

};

另外一种写法：

class MyCircularQueue {

private:

int \*m\_data; //数据域

int m\_front; //指针域

int m\_rear;

int m\_size; //用于判空和溢出，这样比较方便

int m\_capacity;

public:

/\*\* Initialize your data structure here. Set the size of the queue to be k. \*/

MyCircularQueue(int k) {

m\_data = new int[k];

m\_front = m\_rear = 0;

m\_size = 0;

m\_capacity = k;

}

/\*\* Insert an element into the circular queue. Return true if the operation is successful. \*/

bool enQueue(int value) {

if(isFull())

return false;

m\_data[m\_rear] = value;

m\_rear++;

if(m\_rear == m\_capacity)

m\_rear = 0;

m\_size++;

return true;

}

/\*\* Delete an element from the circular queue. Return true if the operation is successful. \*/

bool deQueue() {

if(isEmpty())

return false;

m\_front++;

if(m\_front == m\_capacity)

m\_front = 0;

m\_size--;

return true;

}

/\*\* Get the front item from the queue. \*/

int Front() {

if(isEmpty())

return -1;

return m\_data[m\_front];

}

/\*\* Get the last item from the queue. \*/

int Rear() {

if(isEmpty())

return -1;

int index = m\_rear -1;

if(index < 0)

index = m\_capacity - 1;

return m\_data[index];

}

/\*\* Checks whether the circular queue is empty or not. \*/

bool isEmpty() {

return (0 == m\_size);

}

/\*\* Checks whether the circular queue is full or not. \*/

bool isFull() {

return (m\_size == m\_capacity);

}

};

/\*\*

\* Your MyCircularQueue object will be instantiated and called as such:

\* MyCircularQueue\* obj = new MyCircularQueue(k);

\* bool param\_1 = obj->enQueue(value);

\* bool param\_2 = obj->deQueue();

\* int param\_3 = obj->Front();

\* int param\_4 = obj->Rear();

\* bool param\_5 = obj->isEmpty();

\* bool param\_6 = obj->isFull();

\*/

注：Leetcode 622

## 用两个栈实现队列

题目要求：用两个栈实现一个队列。队列的声明如下，请实现它的两个函数appendTail和deleteHead，分别完成在队列尾部插入结点和队列头部删除结点的功能。

注：剑指offer P59

## 近请求次数

**题目：**写一个RecentCounter类来计算最近的请求。

它只有一个方法：ping(int t)，其中t代表以毫秒为单位的某个时间。

返回从3000毫秒前到现在的ping数。

任何处于[t-3000, t]时间范围之内的 ping都将会被计算在内，包括当前（指 t 时刻）的ping。

保证每次对ping的调用都使用比之前更大的t值。

注：Leetcode 933

**分析：**

int ping(int t)

{

q.push(t); // 存储ping的时间点

q\_len ++;

while (q.front() < t - 3000)

{

q.pop();

q\_len --;

}

return q\_len;

}

## 滑动窗口的最大值

**题目：**给定一个数组和滑动窗口的大小，请找出所有滑动窗口里的最大值。例如，如果输入数组{2,3,4,2,6,2,5,1}及滑动窗口的大小3，那么一共存在6个滑动窗口，它们的最大值分别为{4,4,6,6,6,5}。

**分析：**

这是使用单调队列的经典题目。

难点是如何求一个区间里的最大值呢？（这好像是废话），暴力一下不就得了。

暴力方法，遍历一遍的过程中每次从窗口中再找到最大的数值，这样很明显是O(n × k)的算法。

有的同学可能会想用一个大顶堆（优先级队列）来存放这个窗口里的k个数字，这样就可以知道最大的最大值是多少了，但是问题是这个窗口是移动的，而大顶堆每次只能弹出最大值，我们无法移除其他数值，这样就造成大顶堆维护的不是滑动窗口里面的数值了。所以不能用大顶堆。

此时我们需要一个队列，这个队列呢，放进去窗口里的元素，然后随着窗口的移动，队列也一进一出，每次移动之后，队列告诉我们里面的最大值是什么。

这个队列应该长这个样子：

class MyQueue {

public:

void pop(int value) {

}

void push(int value) {

}

int front() {

return que.front();

}

};

每次窗口移动的时候，调用que.pop(滑动窗口中移除元素的数值)，que.push(滑动窗口添加元素的数值)，然后que.front()就返回我们要的最大值。

这么个队列香不香，要是有现成的这种数据结构是不是更香了！

其实在C++中，可以使用 multiset 来模拟这个过程，文末提供这个解法仅针对C++，以下讲解我们还是靠自己来实现这个单调队列。

然后再分析一下，队列里的元素一定是要排序的，而且要最大值放在出队口，要不然怎么知道最大值呢。

但如果把窗口里的元素都放进队列里，窗口移动的时候，队列需要弹出元素。

那么问题来了，已经排序之后的队列怎么能把窗口要移除的元素（这个元素可不一定是最大值）弹出呢。

其实队列没有必要维护窗口里的所有元素，只需要维护有可能成为窗口里最大值的元素就可以了，同时保证队列里的元素数值是由大到小的。

那么这个维护元素单调递减的队列就叫做单调队列，即单调递减或单调递增的队列。C++中没有直接支持单调队列，需要我们自己来实现一个单调队列

不要以为实现的单调队列就是对窗口里面的数进行排序，如果排序的话，那和优先级队列又有什么区别了呢。

来看一下单调队列如何维护队列里的元素。

动画如下：

图示

AI 生成的内容可能不正确。

对于窗口里的元素{2, 3, 5, 1 ,4}，单调队列里只维护{5, 4} 就够了，保持单调队列里单调递减，此时队列出口元素就是窗口里最大元素。

此时大家应该怀疑单调队列里维护着{5, 4} 怎么配合窗口进行滑动呢？

设计单调队列的时候，pop，和push操作要保持如下规则：

1、pop(value)：如果窗口移除的元素value等于单调队列的出口元素，那么队列弹出元素，否则不用任何操作

2、push(value)：如果push的元素value大于入口元素的数值，那么就将队列入口的元素弹出，直到push元素的数值小于等于队列入口元素的数值为止

保持如上规则，每次窗口移动的时候，只要问que.front()就可以返回当前窗口的最大值。

为了更直观的感受到单调队列的工作过程，以题目示例为例，输入: nums = [1,3,-1,-3,5,3,6,7], 和 k = 3，动画如下：

图片包含 图形用户界面

AI 生成的内容可能不正确。

那么我们用什么数据结构来实现这个单调队列呢？

使用deque最为合适，常用的queue在没有指定容器的情况下，deque就是默认底层容器。

基于刚刚说过的单调队列pop和push的规则，代码不难实现，如下：

class MyQueue { //单调队列（从大到小）

public:

deque<int> que; // 使用deque来实现单调队列

// 每次弹出的时候，比较当前要弹出的数值是否等于队列出口元素的数值，如果相等则弹出。

// 同时pop之前判断队列当前是否为空。

void pop(int value) {

if (!que.empty() && value == que.front()) {

que.pop\_front();

}

}

// 如果push的数值大于入口元素的数值，那么就将队列后端的数值弹出，直到push的数值小于等于队列入口元素的数值为止。

// 这样就保持了队列里的数值是单调从大到小的了。

void push(int value) {

while (!que.empty() && value > que.back()) {

que.pop\_back();

}

que.push\_back(value);

}

// 查询当前队列里的最大值 直接返回队列前端也就是front就可以了。

int front() {

return que.front();

}

};

这样我们就用deque实现了一个单调队列，接下来解决滑动窗口最大值的问题就很简单了，直接看代码吧。

**代码：**

class Solution {

private:

class MyQueue { //单调队列（从大到小）

public:

deque<int> que; // 使用deque来实现单调队列

// 每次弹出的时候，比较当前要弹出的数值是否等于队列出口元素的数值，如果相等则弹出。

// 同时pop之前判断队列当前是否为空。

void pop(int value) {

if (!que.empty() && value == que.front()) {

que.pop\_front();

}

}

// 如果push的数值大于入口元素的数值，那么就将队列后端的数值弹出，直到push的数值小于等于队列入口元素的数值为止。

// 这样就保持了队列里的数值是单调从大到小的了。

void push(int value) {

while (!que.empty() && value > que.back()) {

que.pop\_back();

}

que.push\_back(value);

}

// 查询当前队列里的最大值 直接返回队列前端也就是front就可以了。

int front() {

return que.front();

}

};

public:

vector<int> maxSlidingWindow(vector<int>& nums, int k) {

MyQueue que;

vector<int> result;

for (int i = 0; i < k; i++) { // 先将前k的元素放进队列（初始化）

que.push(nums[i]);

}

result.push\_back(que.front()); // result 记录前k的元素的最大值

for (int i = k; i < nums.size(); i++) {

que.pop(nums[i - k]); // 滑动窗口移除最前面元素

que.push(nums[i]); // 滑动窗口前加入最后面的元素

result.push\_back(que.front()); // 记录对应的最大值

}

return result;

}

};

注：剑指offer P290

## 第k个数

## 前K个高频元素

给定一个非空的整数数组，返回其中出现频率前 k 高的元素。

示例 1:

输入: nums = [1,1,1,2,2,3], k = 2

输出: [1,2]

示例 2:

输入: nums = [1], k = 1

输出: [1]

提示：

你可以假设给定的 k 总是合理的，且 1 ≤ k ≤ 数组中不相同的元素的个数。

你的算法的时间复杂度必须优于 $O(n \log n)$ , n 是数组的大小。

题目数据保证答案唯一，换句话说，数组中前 k 个高频元素的集合是唯一的。

你可以按任意顺序返回答案。

思路：

这道题目主要涉及到如下三块内容：

1、要统计元素出现频率

2、对频率排序

3、找出前K个高频元素

首先统计元素出现的频率，这一类的问题可以使用map来进行统计。

然后是对频率进行排序，这里我们可以使用一种 容器适配器就是优先级队列。

什么是优先级队列呢？

其实就是一个披着队列外衣的堆，因为优先级队列对外接口只是从队头取元素，从队尾添加元素，再无其他取元素的方式，看起来就是一个队列。

而且优先级队列内部元素是自动依照元素的权值排列。那么它是如何有序排列的呢？

缺省情况下priority\_queue利用max-heap（大顶堆）完成对元素的排序，这个大顶堆是以vector为表现形式的complete binary tree（完全二叉树）。

什么是堆呢？

堆是一棵完全二叉树，树中每个结点的值都不小于（或不大于）其左右孩子的值。如果父亲结点是大于等于左右孩子就是大顶堆，小于等于左右孩子就是小顶堆。

所以大家经常说的大顶堆（堆头是最大元素），小顶堆（堆头是最小元素），如果懒得自己实现的话，就直接用priority\_queue（优先级队列）就可以了，底层实现都是一样的，从小到大排就是小顶堆，从大到小排就是大顶堆。

本题我们就要使用优先级队列来对部分频率进行排序。

为什么不用快排呢，使用快排要将map转换为vector的结构，然后对整个数组进行排序，而这种场景下，我们其实只需要维护k个有序的序列就可以了，所以使用优先级队列是最优的。

此时要思考一下，是使用小顶堆呢，还是大顶堆？

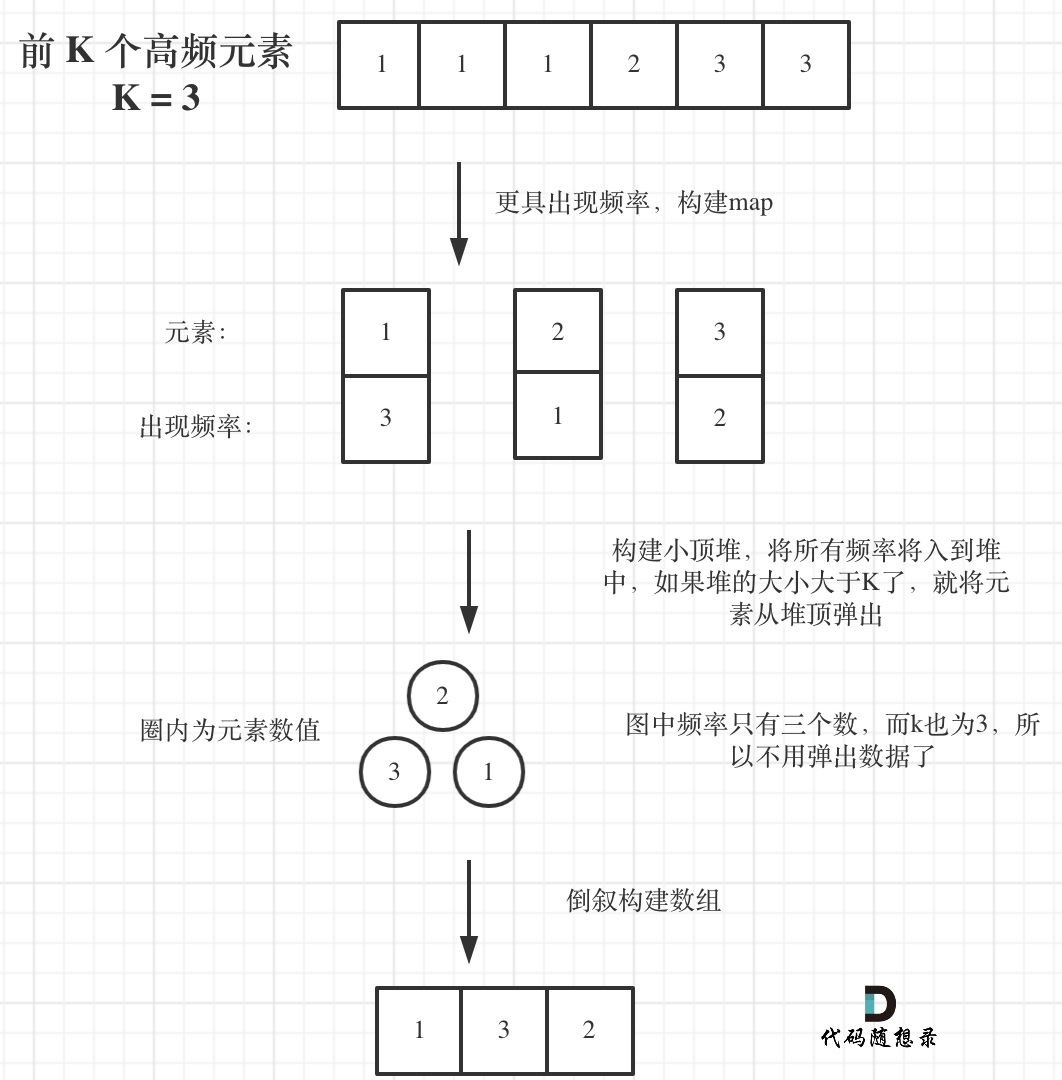
有的同学一想，题目要求前 K 个高频元素，那么果断用大顶堆啊。

那么问题来了，定义一个大小为k的大顶堆，在每次移动更新大顶堆的时候，每次弹出都把最大的元素弹出去了，那么怎么保留下来前K个高频元素呢。

而且使用大顶堆就要把所有元素都进行排序，那能不能只排序k个元素呢？

所以我们要用小顶堆，因为要统计最大前k个元素，只有小顶堆每次将最小的元素弹出，最后小顶堆里积累的才是前k个最大元素。

寻找前k个最大元素流程如图所示：（图中的频率只有三个，所以正好构成一个大小为3的小顶堆，如果频率更多一些，则用这个小顶堆进行扫描）



代码：

class Solution {

public:

// 小顶堆

class mycomparison {

public:

bool operator()(const pair<int, int>& lhs, const pair<int, int>& rhs) {

return lhs.second > rhs.second;

}

};

vector<int> topKFrequent(vector<int>& nums, int k) {

// 要统计元素出现频率

unordered\_map<int, int> map; // map<nums[i],对应出现的次数>

for (int i = 0; i < nums.size(); i++) {

map[nums[i]]++;

}

// 对频率排序

// 定义一个小顶堆，大小为k

priority\_queue<pair<int, int>, vector<pair<int, int>>, mycomparison> pri\_que;

// 用固定大小为k的小顶堆，扫面所有频率的数值

for (unordered\_map<int, int>::iterator it = map.begin(); it != map.end(); it++) {

pri\_que.push(\*it);

if (pri\_que.size() > k) { // 如果堆的大小大于了K，则队列弹出，保证堆的大小一直为k

pri\_que.pop();

}

}

// 找出前K个高频元素，因为小顶堆先弹出的是最小的，所以倒序来输出到数组

vector<int> result(k);

for (int i = k - 1; i >= 0; i--) {

result[i] = pri\_que.top().first;

pri\_que.pop();

}

return result;

}

};

## 和至少为 K 的最短子数组

注：Leetcode 862