为C 标准库容器写自己的内存分配程序 PDF转换可能丢失图片或格式，建议阅读原文

https://www.100test.com/kao_ti2020/253/2021_2022__E4_B8_BAC___E6_A0_87_E5_c97_253816.htm 根据sgi 的STL源码的二级分配算法改写的内存池分配程序，只要稍微修改就可以实现共享内存方式管理，使用C 标准库容器中的map，set，multimap，multiset测试通过，vector测试通不过，原因是在内存回收的时候考虑的比较简单，vector每次分配内存个数不固定，回收也不固定，这样的话，程序还需要继续完善。 内存池管理程序源码如下：以下是引用片段：#ifndef MY_ALLOCATOR_H_ #define MY_ALLOCATOR_H_ #include "stdafx.h" #include #include namespace happyever { enum { NODENUMS = 2 }. union _Obj { union _Obj* M_free_list_link. char M_client_data[1]. } . typedef union _Obj Obj. struct _Cookie { int iShmKey. /* 共享内存键值 */ int iShmID. /* iShmKey对应的shmid */ int iSemKey. /* 锁信号键值 */ int iSemID. /* 锁信号标识 */ int iTotalsize. /* 容器总容量 */ void* pStartall. /* 共享内存自身地址 */ char* pStartfree. /* 自由空间的开始地址*/ char* pEndfree. /* 自由空间的结束地址*/ int iUseNum[NODENUMS]. /*用来存放free_list中节点的size*/ short sFreelistIndex[NODENUMS]. /*存放分配内存节点的链表*/ Obj* uFreelist[NODENUMS]. }. typedef struct _Cookie Cookie. //Obj. //Cookie. static Cookie *pHead = NULL. template class MyAlloc { private: static const int ALIGN = sizeof(Obj). int round_up(int bytes). int freelist_index(int bytes). int freelist_getindex(int bytes). char* chunk_alloc(int size, int *nobjs). void* refill(int num,int n). public: // type definitions typedef T value_type. typedef T* pointer. typedef const T* const_pointer. typedef Tamp. const_reference. typedef std::size_t size_type. typedef std::ptrdiff_t difference_type. template struct rebind { typedef MyAlloc other. }. pointer address (reference value) const { return amp.value. } MyAlloc() throw() { std::cout } MyAlloc(const MyAllocamp. x) throw() { std::cout" } ~MyAlloc() throw() { std::cout } size_type max_size () const throw() { return std::numeric_limits::max() / sizeof(T). } //void PrintFreelistAndCookie(). pointer allocate (size_type num, const void* = 0) { pointer ret = 0. Obj** my_free_list. Obj* result. int index. // print message and allocate memory with global new std::cerr index = freelist_index(sizeof(T)). if(index >= NODENUMS) { return NULL. } my_free_list = pHead->uFreelist index. //Lock(semid,LOCK_NUM). result = *my_free_list. if (result == 0) { ret = (pointer)refill((int)num, round_up(sizeof(T))). } else { *my_free_list = result->M_free_list_link. ret = (pointer)result. } //UnLock(semid,LOCK_NUM). pHead->iUseNum[index] = pHead->iUseNum[index] (int)num. if(0 == ret) { std::cerr exit(1). } std::cerr PrintFreelistAndCookie(). return ret. } void construct (pointer p, const T&amp. value) { // initialize memory with placement new new((void*)p)T(value). } void destroy (pointer p) { // destroy objects by calling their destructor p->~T(). } void deallocate (pointer p, size_type num) { Obj** my_free_list. Obj* q . int index. index = freelist_getindex(sizeof(T)). if(index >= NODENUMS) { std::cerr exit(1). } my_free_list = pHead->uFreelist index. q = (Obj*) p. //Lock(semid,LOCK_NUM). /*这个地方可

能会有问题*/ //for(int i=0 .i { q->M_free_list_link = *my_free_list. *my_free_list = q. } //UnLock(semid,LOCK_NUM). pHead->iUseNum[index] = pHead->iUseNum[index] - (int)num. std::cerr PrintFreelistAndCookie(). } }. template int MyAlloc::round_up(int bytes) { int i. i = bytes. if(bytes { i = ALIGN. } std::cout return i. }. template int MyAlloc::freelist_index(int bytes) { int i. for(i=0 . i { if(pHead->sFreelistIndex[i] == bytes) break.