My Allocator

Date: June-4th,2022

author:韩恺荣

Chapter One:介绍

在这个实验中，我们要自定义一个内存池，代替标准模板库中的std::allocator,并对他们的性能进行比较和分析。

一个使用 allocator 的简单实例:

```
template < class T, class Alloc = allocator<T> > class vector;
template < class T, class Alloc = allocator<T> > class list;
```

此外，为保证自定义的内存池能够正确的运行，我们还需要为我们的内存池保留如下的接口:

```
typedef void _Not_user_specialized;
typedef _Ty value_type;
typedef value_type *pointer;
typedef const value_type *const_pointer;
typedef value_type& reference;
typedef const value_type& const_reference;
typedef size_t size_type;
typedef ptrdiff_t difference_type;
typedef true_type propagate_on_container_move_assignment;
typedef true_type is_always_equal;

pointer address(reference _Val) const _NOEXCEPT
const_pointer address(const_reference _Val) const _NOEXCEPT
void deallocate(pointer _Ptr, size_type _Count)
_DECLSPEC_ALLOCATOR pointer allocate(size_type _Count)
template<class _Uty> void destroy(_Uty *_Ptr)
template<class _Objty, class _Types>
void construct(_Objty *_Ptr, _Types&&... _Args)
```

Chapter 2: 实验环境:

C++标准：C++20（C++17 也可以）

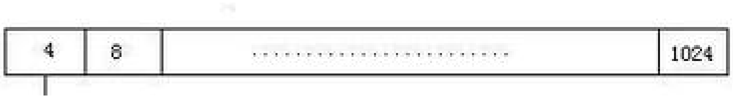
IDE:Visual Studio2022

系统：windows10

内存大小：16GB

Chapter 3: 算法描述:

我实现内存池的方式综述如下:



对于一个内存池，我们使用 freelist 来对该内存池的内容进行管理, 具体的管理方式为, 我们将我们将要处理的内存对象分为两大类, 一类的大小大于 1024 字节（代码中定义为 MAX_SIZE），另一类的大小小于 1024 字节, 对于大于 1024 字节的对象, 我们采用 malloc 的方式直接申请内存，而对于小于 1024 字节的对象，我们将这部分对象以 8 字节为步长分为 1024/8 个块区，即 128 块区，分别是 8、16、32……1016、1024，每一个块区都有一个 freelist 指向他们的内存地址，，当我要处理一个大小为 x（x 不大于 1024)的对象时，我们就将他放在距离他最近且比他大的 8 的整数倍大小的内存块上, 如一个对象大小为 10, 那我们将他最终放置于内存大小为 16 的块区中，而寻找内存大小为 16 的块区是通过 freelist 实现的，初始化时将所有的 freelist 链表头部设置为 NULL，之后按需扩张我们的内存池即可。

具体的函数描述如下:

（1）变量说明:

```
const size_t SEG_SIZE = 8;//step between different block in memory
const size_t MAX_SIZE = 1024;//maxsize for memory block
const size_t CHUNK_NUM = 8;//memory blocks' number of every allocating
const size_t FREELIST_SIZE = MAX_SIZE / SEG_SIZE;//freelist's size
```

SEG_SIZE:步长，表示内存大小的间隔

MAX_SIZE：放置于 freelist 指向内存的最大对象大小

CHUNK_NUM：每次分配的内存块数

FREELIST_SIZE：freelist 大小

```
struct Freelist_node {//freelist's node,store address mainly
    struct Freelist_node* next;
};
```

freelist 的节点内容如上图

（2）宏定义说明：

```
#define INDEX_SEG(e) (((((e)+7)&~(SEG_SIZE-1))>>3)-1)//index calculator
```

计算距离 e 最近且不小于 e 的 8 的整数倍数字再除以 8 并减一，如 10 经过处理变成 1，表示他将连接在 freelist[1]这个节点指向的内存

（3）重要函数说明：

a）回收函数：

大于 maxsize 的对象直接调用 free 方法，否则将 freelist 对应的节点指向下一个即可。

```
static inline void deallocate(pointer ptr, size_type count) {//deallocate function
    size_t size_sum = count * sizeof(value_type);//calculate size of block we will deallocate
    if (size_sum > MAX_SIZE) {//if size is bigger than maxsize, free directly
        free(ptr);
        return;
    }
    else {//put freelist's head to this block
        ((Freelist_node*)ptr)->next = free_list[INDEX_SEG(size_sum)];
        free_list[INDEX_SEG(size_sum)] = (Freelist_node*)ptr;
    }
}
```

b）分配函数：

根据对象的大小检测和分类，大于 1024 直接调用 malloc 函数，否则计算对象大小对应的 freelist 的下标，然后使用 freelist 指向的区域放置该对象即可。

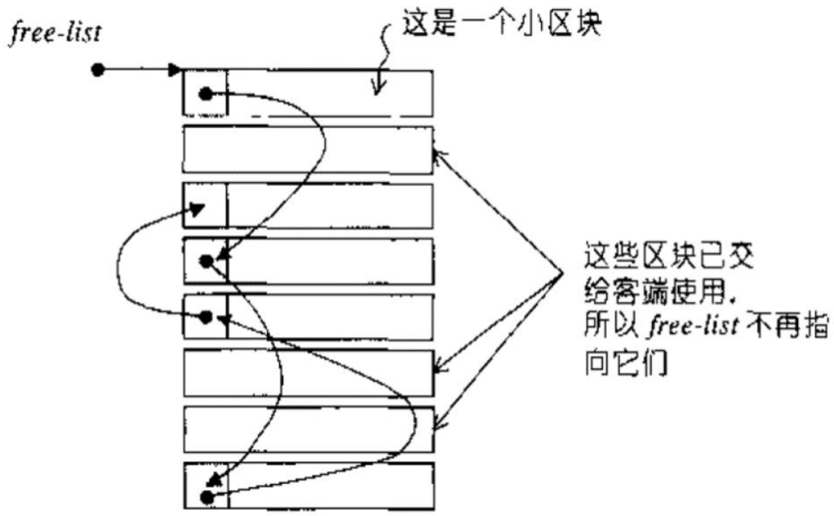
但如果 freelist 为 null，则调用 CallForMem 函数向堆栈申请内存空间。

```
    static  pointer allocate(size_type count) {//allocate function
        size_t size_sum = count * sizeof(value_type);//calculate size of block
        if (size_sum > MAX_SIZE) {//if size is bigger than maxsize, malloc directly
            char *buf = (char*)std::malloc(size_sum);//use a buffer to store block
            if (!buf) {//allocate fail
                std::cout << "memory overflow" << std::endl;
                throw "error";//throw exception
            }
            return (pointer)buf;//return address of the buf
        }
        Freelist_node* cur;
        Freelist_node* des;
        des = free_list[INDEX_SEG(size_sum)];
        if (des == nullptr) {
#if DEBUG_STATE
            count1[INDEX_SEG(size_sum)][0]++;//use in debug,record freelist's message
            count1[INDEX_SEG(size_sum)][1]=0;
            std::cout << "申请内存空间的大小:" << size_sum << " freelist 下标:" << INDEX_SEG(size_sum) << std::endl;
#endif
            return (pointer)CallForMem(size_sum, INDEX_SEG(size_sum));//freelist is null, call for memory
        }
#if DEBUG_STATE
        count1[INDEX_SEG(size_sum)][1]++;
#endif
```

```
            free_list[INDEX_SEG(size_sum)] = free_list[INDEX_SEG(size_sum)]->next;//update freelist
            return (pointer)des;
        }
```

c）内存申请函数：

当 freelist 是 null 时，证明内存池中无内存存放该对象，即可调用该函数，该函数向内存空间申请一段内存，大小为 CHUNK_NUM * align_to_eight，其中 CHUNK_NUM 是内存的块数，align_to_eight 是对象大小距离最近但不小于的 8 的整数倍的数值，然后并用 freelist 将这段内存串联起来。




```
inline static char* CallForMem(size_t size_to_call, int index) {//call for memory by this function
    Freelist_node* temp = nullptr;
    size_t all = 7;
    size_t align_to_eight = ((size_to_call + 7) & ~all);//find the nearest number of times of 8
    char* buf = (char*)std::malloc(CHUNK_NUM * align_to_eight);//allocate
    if (!buf) {
        std::cout << "out of memory" << std::endl;//allocate fail
        throw "error";
    }
    char* record = buf + align_to_eight;
    for (int i = 0; i < CHUNK_NUM - 1; i++) {//link node one by one
        if (i == 0) {
            free_list[index] = (Freelist_node*)record;
            temp = (Freelist_node*)record;
            temp->next = nullptr;
            record += align_to_eight;
        }
        else {
            temp->next = (Freelist_node*)record;
            temp = temp->next;
            temp->next = nullptr;
            record += align_to_eight;
        }
    }
    return buf;
}
```

d）其他函数:

构造和摧毁函数直接调用对象类型的构造和析构函数即可

```
template<class Ut>
static inline void destroy(Ut* p) {//destory, use destructor function directly
    p->~Ut();
}
template<class Ut, class Pt>
static inline void construct(Ut* p, Pt argv) {//constructor , use constructor function directly
    new(p) Ut(argv);
}
```

内存池的构造函数和 max_size()、address()方法都简单仿照 std::allocator 的函数调用方式即可:

```
MyAllocator() {}//constructor

template<class T>
MyAllocator(const MyAllocator<T>& a) {}//constructor

~MyAllocator() {}//destructor

inline size_type max_size() {//retunrn maxsize of a valuetype
    return size_type(UINT_MAX / sizeof(value_type));
}

inline pointer address(reference _Val) {//call std::addressof to get address of val
    return std::addressof(_Val);
}
inline const_pointer address(const_reference _Val) {//call std::addressof to get address of val
    return std::addressof(_Val);
}
```

Chapter 3: Testing Results

（1）测试一：直接用 pta 的代码，将 std::allocator 替换为我的 allocator，验证结果的正确性：（为方便测试，我将 pta 和钉钉群里的代码拷贝在同一个 main.cpp 文件，并定义宏#define TEST_TYPE 0，当这个数值设置为 1 时，对应 pta 的测试程序，当设置为 0 时，对应课程钉钉群上传的测试程序）

```
Microsoft Visual Studio 调试控制台
correct assignment in vecints: 3633
correct assignment in vecpts: 1112
```

使用 std::allocator 和我的 allocator 结果一致

（2）测试二：运行钉钉群中的测试：

```
D:\0学业\大二下\oop\Mem pool\ConsoleApplication1\x64\Debug\ConsoleApplication1.exe
incorrect assignment in vector 9999 for object (13,20)
请按任意键继续. . .
```

和 std::allocator 的结果一致

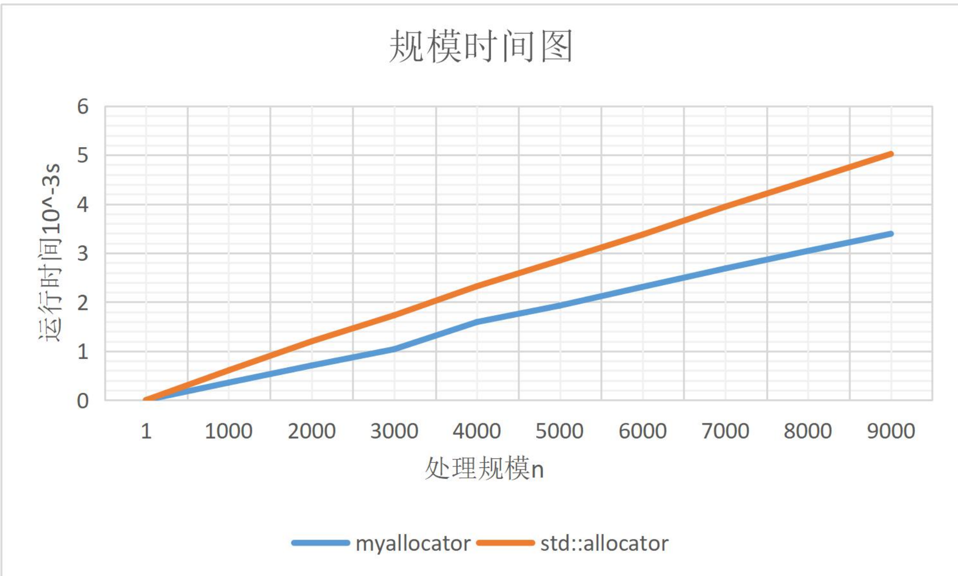
（3）测试和 std::allocator 相比的时间优化程度(resize 因素)：

我们定义 resize 的次数是处理规模 n，而运行时 resize 的范围由

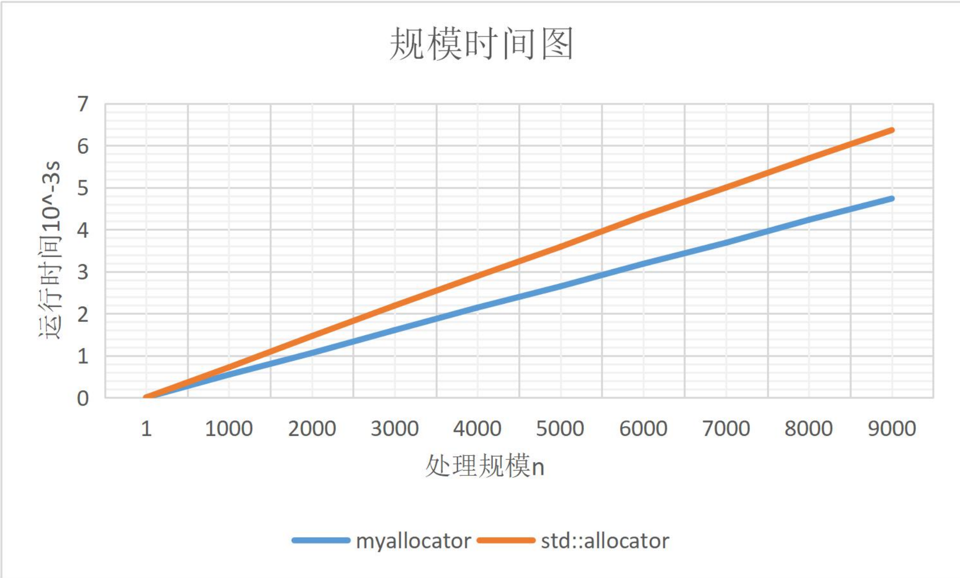
```
std::uniform_int_distribution<> dis(1, distr[j]);
```

函数生成，所以改变生成的规模范围，我们可以发现：

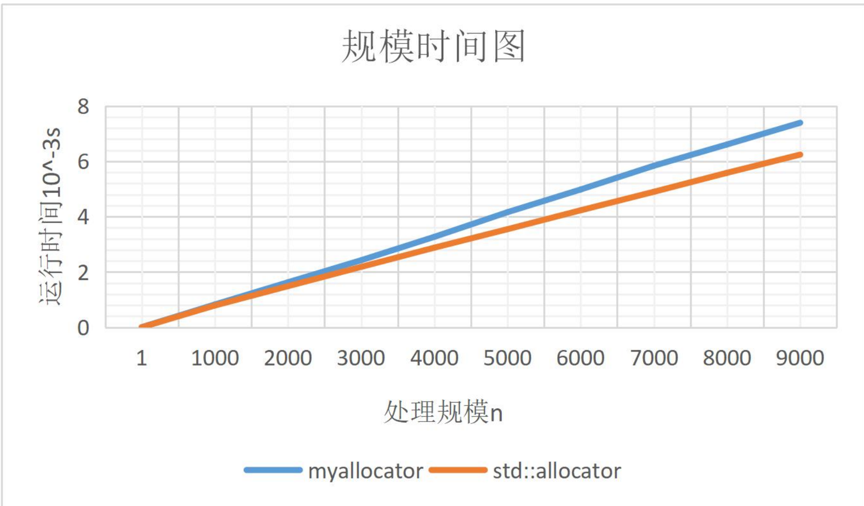
a)当 resize 的数值范围在 1~8 时：



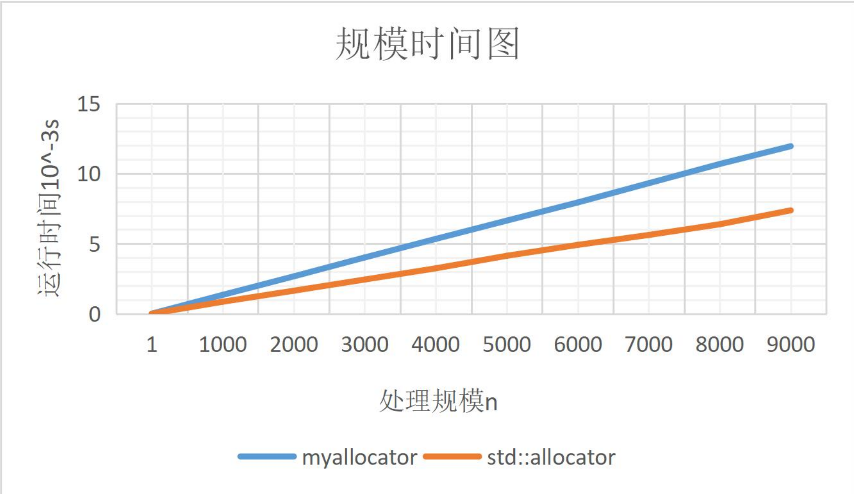
b)当 resize 的数值范围在 1~32 时：



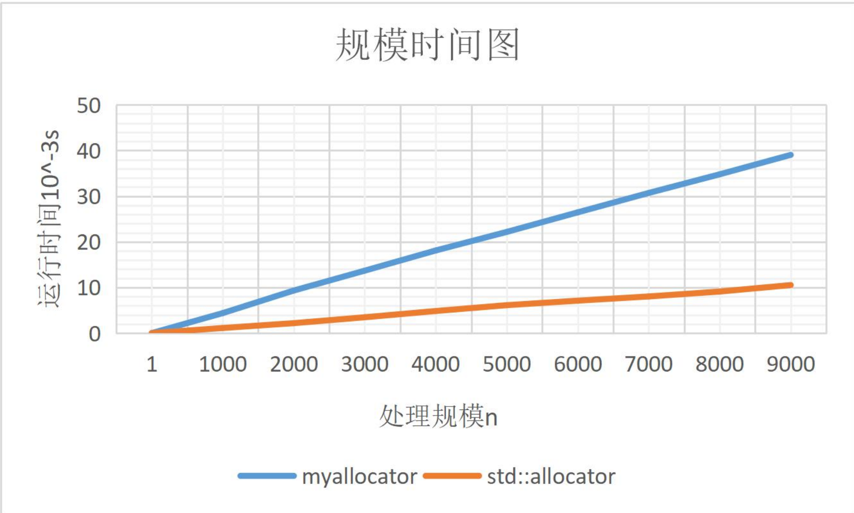
c)当 resize 的数值范围在 1~64 时:



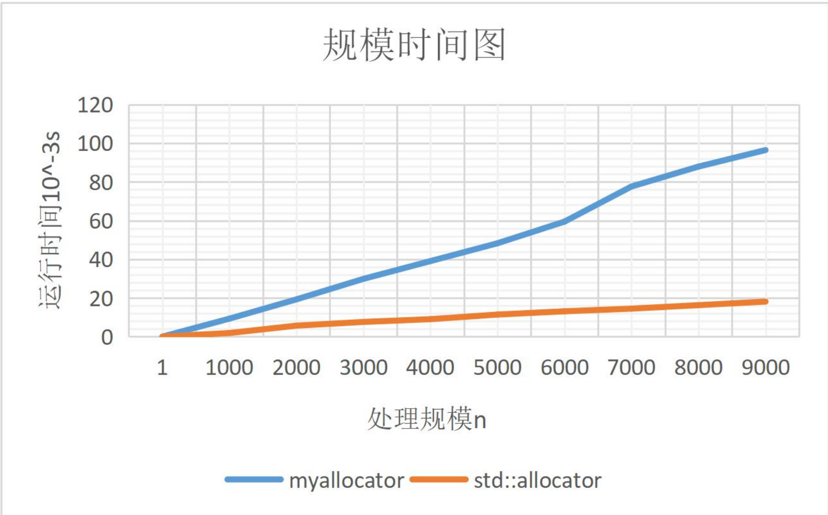
d)当 resize 的数值范围在 1~128 时:



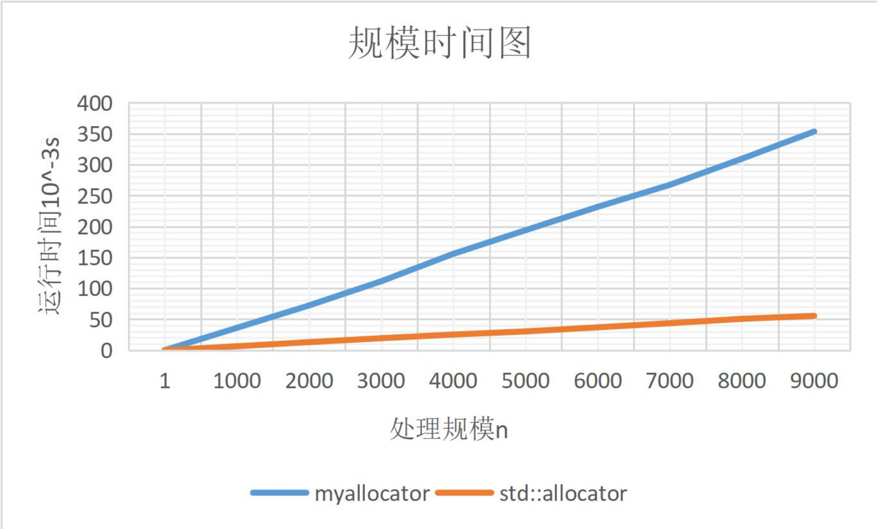
e)当 resize 的数值范围在 1~512 时:



f)当 resize 的数值范围在 1~1024 时:



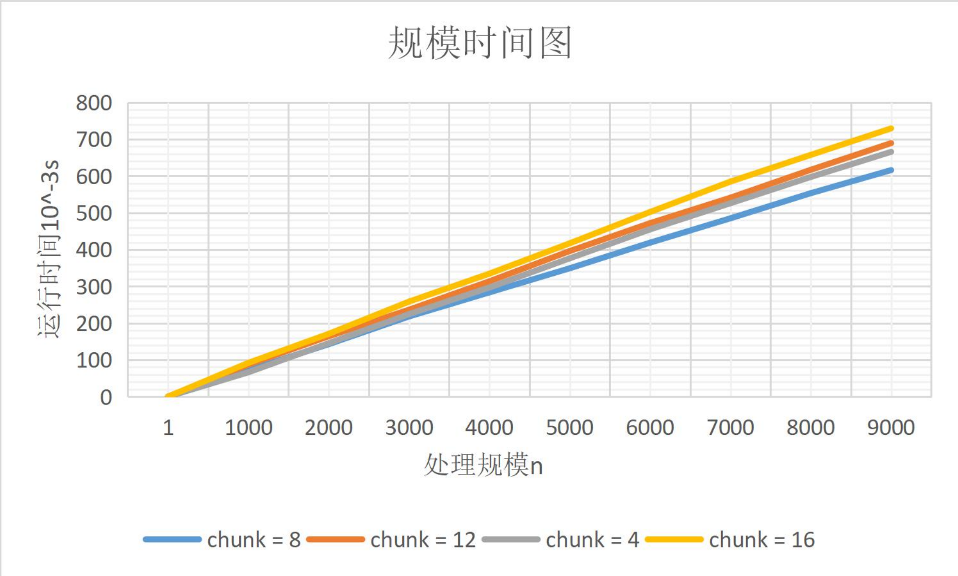
g)当 resize 的数值范围在 1~2000 时：



结论：在面对大量小内存对象时，我们的 myallocator 有着更强大且优越的性能，这是因为每次申请内存块都申请了 CHUNK_NUM 块，因此当再一次插入相同大小区间的对象时，可以直接用内存池取出内存地址直接插入从而加速，但当面对大量分布均匀但范围很广的对象时，我们的结果会慢于标准模板库，这是因为当均匀分布时，每次插一次都有可能引发连续分 CHUNK_NUM 块内存的动作，而分配 CHUNK_NUM 块时串联 freelist 需要花费额外的时间。

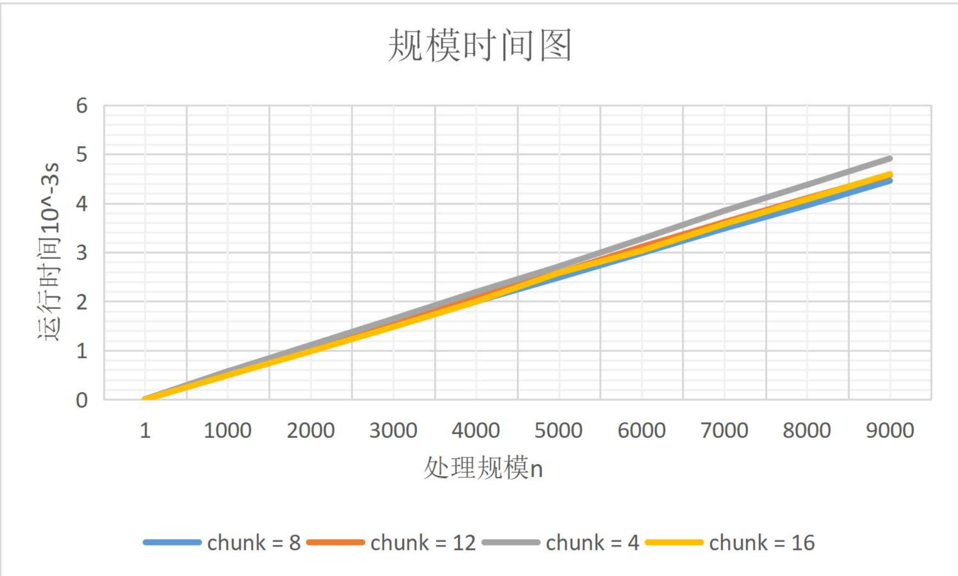
（4）测试和 std::allocator 相比的时间优化程度(CHUNK_NUM 因素)：

a)当resize范围是0~10000时，chunk=8表现更出色，而chunk=16最慢，这是因为 resize 范围太广，导致再次落入 0~1024 的某个特定块的概率变小，所以 chunk 太大会造成大量的建立 freelist 时的时间冗余，且会浪费大量空间



b）当 resize 范围是 0~32 时

此时 chunk=4 表现最慢，这是因为这个时候，大量落在相同区块的重复会让增加 chunk 的数量更占时间优势



Chapter 4:评价与反思

总结来看，我们的 myallocator 在大量的小对象内存分配时，展现出较大优势，但当 resize 范围扩大时，甚至时间会比 std::allocator 更慢，此外 myallocator 由于每次都申请 CHUNK_NUM 个内存块

来为下一次相同大小的内存分配加速，所以会产生一定的内存消耗，并且每次都将对象放在不小于它的内存块中，导致内存会有浪费，所以我们是用空间来换取时间的优化。

Appendix:source code

```
#pragma once

#include<iostream>

#include<cassert>

#define DEBUG_STATE 0



#if DEBUG_STATE

    int count1[1024/8][2] = {0};

#endif

namespace Alloc {

    const size_t SEG_SIZE = 8;//step between different block in memory

    const size_t MAX_SIZE = 1024;//maxsize for memory block

    const size_t CHUNK_NUM = 4;//memory blocks' number of every allocating

    const size_t FREELIST_SIZE = MAX_SIZE / SEG_SIZE;//freelist's size

#define INDEX_SEG(e) (((((e)+SEG_SIZE-1)&~(SEG_SIZE-1))>>3)-1)//index calculator



    struct Freelist_node {//freelist's node,store address mainly

        struct Freelist_node* next;

    };
```

```
template <class _Ty>
class MyAllocator
{
public:
	/*---value definition and typedef here---------*/
	typedef void _Not_user_specialized;
	typedef _Ty value_type;
	typedef value_type* pointer;
	typedef const value_type* const_pointer;
	typedef value_type& reference;
	typedef const value_type& const_reference;
	typedef size_t size_type;
	static inline struct Freelist_node* free_list[FREELIST_SIZE] = { nullptr };//freelist
	template<class T>
	struct rebind { typedef MyAllocator<T> other; };
	/*-----------function is defined here ------------*/
	MyAllocator() {}//constructor

	template<class T>
	MyAllocator(const MyAllocator<T>& a) {}//constructor
```

```
~MyAllocator() {}//destructor

inline size_type max_size() {//retunrn maxsize of a valuetype
    return size_type(UINT_MAX / sizeof(value_type));
}

inline pointer address(reference _Val) {//call std::addressof to get address of val
    return std::addressof(_Val);
}
inline const_pointer address(const_reference _Val) {//call std::addressof to get
address of val
    return std::addressof(_Val);
}
static inline void deallocate(pointer ptr, size_type count) {//deallocate function
    size_t size_sum = count * sizeof(value_type);//calculate size of block we will
deallocate
    if (size_sum > MAX_SIZE) {//if size is bigger than maxsize, free directly
        free(ptr);
        return;
    }
    else {//put freelist's head to this block
        ((Freelist_node*)ptr)->next = free_list[INDEX_SEG(size_sum)];
```

```
                free_list[INDEX_SEG(size_sum)] = (Freelist_node*)ptr;
            }
        }
        static  pointer allocate(size_type count) {//allocate function
            size_t size_sum = count * sizeof(value_type);//calculate size of block
            if (size_sum > MAX_SIZE) {//if size is bigger than maxsize, malloc directly
                char *buf = (char*)std::malloc(size_sum);//use a buffer to store block
                if (!buf) {//allocate fail
                    std::cout << "memory overflow" << std::endl;
                    throw "error";//throw exception
                }
                return (pointer)buf;//return address of the buf
            }
            Freelist_node* cur;
            Freelist_node* des;
            des = free_list[INDEX_SEG(size_sum)];
            if (des == nullptr) {
#if DEBUG_STATE
                count1[INDEX_SEG(size_sum)][0]++;//use in debug,record freelist's
message
                count1[INDEX_SEG(size_sum)][1]=0;
                std::cout << "申请内存空间的大小:" << size_sum << " freelist  下标:" <<
```

```
INDEX_SEG(size_sum) << std::endl;
#endif
                return (pointer)CallForMem(size_sum, INDEX_SEG(size_sum));//freelist is
null, call for memory
            }
#if DEBUG_STATE
            count1[INDEX_SEG(size_sum)][1]++;
#endif
            free_list[INDEX_SEG(size_sum)] =
free_list[INDEX_SEG(size_sum)]->next;//update freelist
            return (pointer)des;
        }
        template<class Ut>
        static inline void destroy(Ut* p) {//destory, use destructor function directly
            p->~Ut();
        }
        template<class Ut, class Pt>
        static inline void construct(Ut* p, Pt argv) {//constructor , use constructor function
directly
            new(p) Ut(argv);
        }
        private:
```

```
inline static char* CallForMem(size_t size_to_call, int index) {//call for memory
by this function
        Freelist_node* temp = nullptr;
        size_t all = 7;
        size_t align_to_eight = ((size_to_call + 7) & ~all);//find the nearest
number of times of 8
        char* buf = (char*)std::malloc(CHUNK_NUM * align_to_eight);//allocate
        if (!buf) {
            std::cout << "out of memory" << std::endl;//allocate fail
            throw "error";
        }
        char* record = buf + align_to_eight;
        for (int i = 0; i < CHUNK_NUM - 1; i++) {//link node one by one
            if (i == 0) {
                free_list[index] = (Freelist_node*)record;
                temp = (Freelist_node*)record;
                temp->next = nullptr;
                record += align_to_eight;
            }
            else {
                temp->next = (Freelist_node*)record;
                temp = temp->next;
```

```
                        temp->next = nullptr;
                        record += align_to_eight;
                    }
                }
                return buf;
            }
    };
}
```

Appendix2:test code

```
#include <iostream>
#include <random>
#include <vector>
#include<Windows.h>
#include<iomanip>
#include<list>
#include "MyAllocator.h"

template<class T>
using MyAllocator = Alloc::MyAllocator<T>; // replace the std::allocator with your allocator
template<class T>
using IniAllocator = std::allocator<T>;
using Point2D = std::pair<int, int>;
```

```
using namespace std;

const int TestSize = 10000;

const int PickSize = 1000;

const int test_size = 10;

double time_counter[2][10][test_size];

double time_counter_list[2][10][test_size];

int main()

{

    std::random_device rd;

    std::mt19937 gen(rd());

    //std::uniform_int_distribution<> dis(1, TestSize);

    int distr[10] = { 8,32,64,128,256,512,1024,2000,5000,10000 };

    LARGE_INTEGER t1, t2, t3,t4,tc;//high precision clock

    double time_t1=0,time_t2=0;

    QueryPerformanceFrequency(&tc);//record frequency of the clock in cpu

    /*test for vector*/

    using IntVec = std::vector<int, MyAllocator<int>>;

    using IntVec_t = std::vector<int>;

    std::vector<IntVec, MyAllocator<IntVec>> vecints(TestSize);

    std::vector<IntVec_t> vecints_t(TestSize);



    for (int j = 0; j < 10; j++) {
```

```
//实验组，使用自定义内存分配器
    std::uniform_int_distribution<> dis(1, distr[j]);
    time_t1 = 0;
    time_t2 = 0;
    for (int i = 0; i < TestSize; i++) {
        size_t size = dis(gen);
        QueryPerformanceCounter(&t1);
        vecints[i].resize(size);
        QueryPerformanceCounter(&t2);
        time_t1 += (double)((t2.QuadPart - t1.QuadPart) * 1000.0 / tc.QuadPart);
        if (i % 1000 == 0) {
            time_counter[0][j][i / 1000] = time_t1;
        }
    }
    //对照组，使用 std::allocator
    for (int i = 0; i < TestSize; i++) {
        size_t size = dis(gen);
        QueryPerformanceCounter(&t3);
        vecints_t[i].resize(size);
        QueryPerformanceCounter(&t4);
        time_t2 += (double)((t4.QuadPart - t3.QuadPart) * 1000.0 / tc.QuadPart);
        if (i % 1000 == 0) {
```

```
                time_counter[1][j][i / 1000] = time_t2;
            }
        }
    }

    cout << "vector test result" << endl;
    for (int j = 0; j < 10; j++) {
        cout << "when resize is between 0 and " << distr[j] << endl<<endl;
        cout << setw(15) << setiosflags(ios::left) << "input scale:";
        for (int i = 0; i < 10000; i += 1000)cout << setw(10) << i;
        cout << endl;
        cout << setw(15) << setiosflags(ios::left) << "myallocator:";
        for (int i = 0; i < test_size; i++) {
            cout << setw(10) << time_counter[0][j][i] ;
        }
        cout << endl;
        cout << setw(15) << setiosflags(ios::left) << "std::allocator:";
        for (int i = 0; i < test_size; i++) {
            cout << setw(10) << time_counter[1][j][i] ;
        }
        cout << endl;
    }
```

```
}
```