Project

Memory Pool Allocator

Date:2022-05-30

1. 实验目的

本项目采用 C++11 标准，使用 g++进行编译运行，利用 VSCode 配置进行整体链接。

本项目旨在利用内存池来分配内存，以防止 C++默认的内存分配导致的内存块分散不连续。比如，若使用 new 或 malloc 分配内存，在分配一系列小空间的内存后，由于不连续的特性，致使可能出现无法分配大块内存的情况。内存池分配就可以解决这一问题，同时还可提高分配内存的性能。

本项目源文件有：test.cpp, normal_allocator.hpp, memory_pool_allocator.hpp。其中，normal_allocator.hpp 为使用 C++默认的 new 分配内存的模板类；memory_pool_allocator.hpp 为使用内存池分配内存的模板类；test.cpp 使用 tester 模板类对分配性能进行测试，同时还有 C++STL 的 allocator，以比较本项目实现的内存池分配器的性能。

2. 实验方案

normal_allocator.hpp

```
template <class T>
class Nallocator
{
public:
    typedef T value_type;
    typedef size_t size_type;
    typedef ptrdiff_t difference_type;

    Nallocator() = default;

    template <class U>
    Nallocator(const Nallocator<U> &other) noexcept;

    T *allocate(size_type n)
    {
        auto buf = (T *)(::operator new((size_type)(n * sizeof(T))));
        if (buf == 0)
            throw std::bad_alloc();
        return buf;
    }

    void deallocate(T *buf, size_type) { ::operator delete(buf); }
};
```

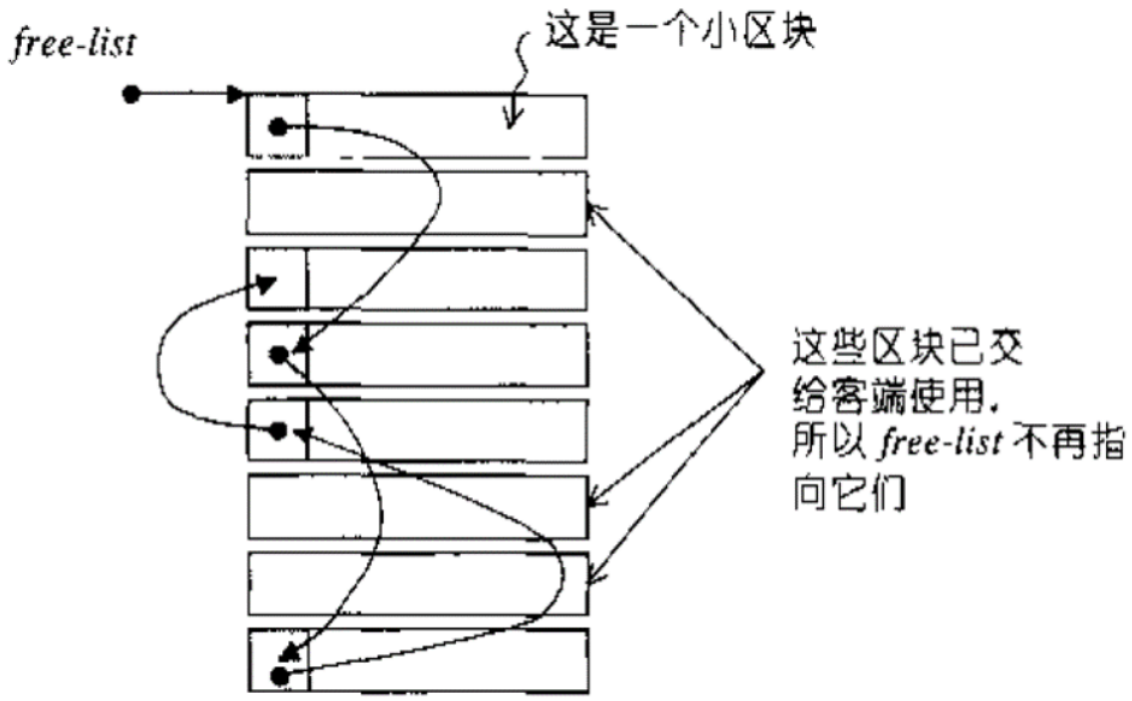
typedef T value_type，STL 的 vector 和 pair 要求
allocate 使用 new 分配内存，内存大小由类型大小和输入参数 n 决定，返回对应类型的指针
deallocate 使用 delete 删除 new 分配的内存
由此实现了使用 new 和 delete 的内存分配模板类

memory_pool_allocator.hpp

基类 pool_alloc_base

```
class pool_alloc_base
{
protected:
    typedef std::size_t size_t;
    // the max size of small block in memory pool
    enum
    {
        MAX_BYTES = 65536
    };
    // the align size of each block, i.e. the size could be divided by 64
    enum
    {
        ALIGN = 64
    };
    // the number of free lists
    enum
    {
        FREE_LIST_SIZE = (size_t)MAX_BYTES / (size_t)ALIGN
    };

    // the union object of free list
    union obj
    {
        union obj *free_list_link; // 下一个内存块节点
        char client_data[1];       // 用户看到的内存地址
    };
    static obj *volatile free_list[FREE_LIST_SIZE]; //链表头指针数组

    // 维护内存块状态
    static char *start_mem_pool; // 内存池中可用空间的起始地址
    static char *end_mem_pool;   //内存池中可用空间的结束地址
    static size_t heap_size;     //已经从堆中申请的总空间大小

    // align the block
    inline size_t round_up(size_t bytes);

    // get the header of the linked list according to bytes
    obj *volatile *get_free_list(size_t bytes) throw();

    // call refill when free_list has no enough resource
    void *refill(size_t n);

    /**
     * Allocate memory block from memory pool to avoid fragmentation
     * @param size size of each object
     * @param cnt_objs number of objects
     */
    char *allocate_chunk(size_t size, int &cnt_objs);
};
```

参照《STL 源码分析》所创建的基类，使用内存池链表实现分配内存的基础函数
在该基类中创建了联合类型 obj，包含指向链表下一个块的指针和自身内存地址，共同占用 4 个字节；
free_list 维护了一个链表头指针数组，其中每个成员都是指向一个链表的头指针。如图所示，图中的 free-list 就是 free_list 数组中的一个成员，为一个内存池链表的头指针。



refill 函数功能：当指定大小的内存池链表已全被分配时，调用 refill，分配额外内存以扩充链表，并返回用户所需的 n 个块

allocate_chunk 功能：根据指定参数分配内存，主要实现方式为根据大小找到指定链表，随后在链表中申请块，申请成功后将这些块从链表中删去并返回给调用者

子模板类 pool_alloc

```
template <class T>
class pool_alloc : public pool_alloc_base
{
public:
    typedef void _Not_user_specialized;
    typedef T value_type;
    typedef value_type *pointer;
    typedef const value_type *const_pointer;
    typedef value_type &reference;
    typedef const value_type &const_reference;
    typedef std::size_t size_type;
    typedef ptrdiff_t difference_type;
    typedef std::true_type propagate_on_container_move_assignment;
    typedef std::true_type is_always_equal;

    pool_alloc() = default;

    template <class U>
    pool_alloc(const pool_alloc<U> &other) noexcept;

    pointer address(reference _Val) const noexcept;

    const_pointer address(const_reference _Val) const noexcept;
```

```
    pointer allocate(size_type _Count);

    void deallocate(pointer _Ptr, size_type _Count);

    template <class _Uty>
    void destroy(_Uty *_Ptr);

    // Constructs an object of type T in allocated uninitialized storage pointed
to by _Ptr, using placement-new
    template <class _Objty, class... _Types>;

private:
    void *alloc(std::size_t n);

    void dealloc(void *p, std::size_t);

    void *_allocate(size_type n);

    void _deallocate(void *p, size_type n);
};
```

public 部分均为项目要求，参照 std::allocator 实现即可，其中**allocate**和**deallocate**调用 private 部分函数来实现分配与销毁。

private 部分，**alloc**和**dealloc**使用 malloc 和 free 进行内存操作，以应对超过最大字节限制的内存分配请求；**_allocate**和**_deallocate**则利用基类中的函数进行内存操作。

test.cpp

测试类 tester

为防止代码的冗余性，使用模板类 tester 封装测试函数,可放入各种不同的 allocator。tester 使用给定的测试代码对各种 allocator 进行调用测试。

测试主函数

```
int main()
{
    clock_t start;

    // test the normal allocator which uses malloc and free only
    tester<Nallocator> tester1;
    start = clock();
    tester1.main();
    std::cout << "Normal allocator without memory pool cost: "
              << (clock() - start) * 1.0 / CLOCKS_PER_SEC << " seconds"
              << std::endl
              << std::endl;

    // test my allocator, which uses memory pool
    tester<pool_alloc> tester2;
    start = clock();
    tester2.main();
    std::cout << "Allocator with memory pool designed by me cost: "
              << (clock() - start) * 1.0 / CLOCKS_PER_SEC << " seconds"
              << std::endl
```

```
              << std::endl;

    // test std::allocator
    tester<std::allocator> tester3;
    start = clock();
    tester3.main();
    std::cout << "std::allocator, allocator with memory pool in STL, cost: "
              << (clock() - start) * 1.0 / CLOCKS_PER_SEC << " seconds"
              << std::endl
              << std::endl;
    return 0;
}
```

使用时钟类型 clock_t 来记录每个 allocator 运行所花费的时间，从而得出性能分析结果

3. 实验结果

```
[Running] cd "e:\learn\computer_science\Cpp\hw7\" && g++ test.cpp -o test && "e:\learn\computer_science\Cpp\hw7\"test
correct assignment in vecints: 1766
correct assignment in vecpts: 4335
Normal allocator without memory pool cost: 4.599 seconds

correct assignment in vecints: 1766
correct assignment in vecpts: 4335
Allocator with memory pool designed by me cost: 3.844 seconds

correct assignment in vecints: 1766
correct assignment in vecpts: 4335
std::allocator, allocator with memory pool in STL, cost: 2.387 seconds


[Done] exited with code=0 in 12.504 seconds
```

运行结果如图。经过多次测试，基本运行时间关系式为：T(Normal) = T(Memory_pool) + 0.7s = T(stl::allocator) + 2.2s. 可见本项目实现的基于内存池的分配器性能良好，相较于普通的 new 有一定的性能提升，但还是不及 STL 里的 allocator，原因可能为 STL 底层代码的支持度不同。