Object-Oriented Programming Project

STL allocator + memory pool

Date:2020-06-25

CONTENT

1 内存池的结构设计
- 1.1 设计思路与总体框架
- 1.2 数据结构
  - 1.2.1 内存池
  - 1.2.2 STL 内存分配器
- 1.3 函数说明
  - 1.2.1 内存池
  - 1.2.2 STL 内存分配器

2 测试结果及分析
- 2.1 测试结果
- 2.2 结果分析

3 小组分工及心得体会
- 3.1 小组分工
- 3.2 心得体会

1 内存池的结构设计

1.1 设计思路与总体框架

按照题目要求，我们采用内存池(Memory Pool)来实现Allocator。而内存池是通过彼此连接的块结构(block)来实现的。每一个块包含一个指向前一个块的指针和一个指向后一个块的指针。内存池包含一定数目的块，并对每一块进行空间分配。

不同的内存池之间彼此相连，每一个内存池包含一个指向前一个内存池的指针和指向后一个内存池的指针。这些内存池构成了内存池链表(Memory List)。

为了定位到不同的内存池和不同的块，内存池和内存池链表中都有指示空闲空间和已被分配空间的指针。

总体框架如图1所示。

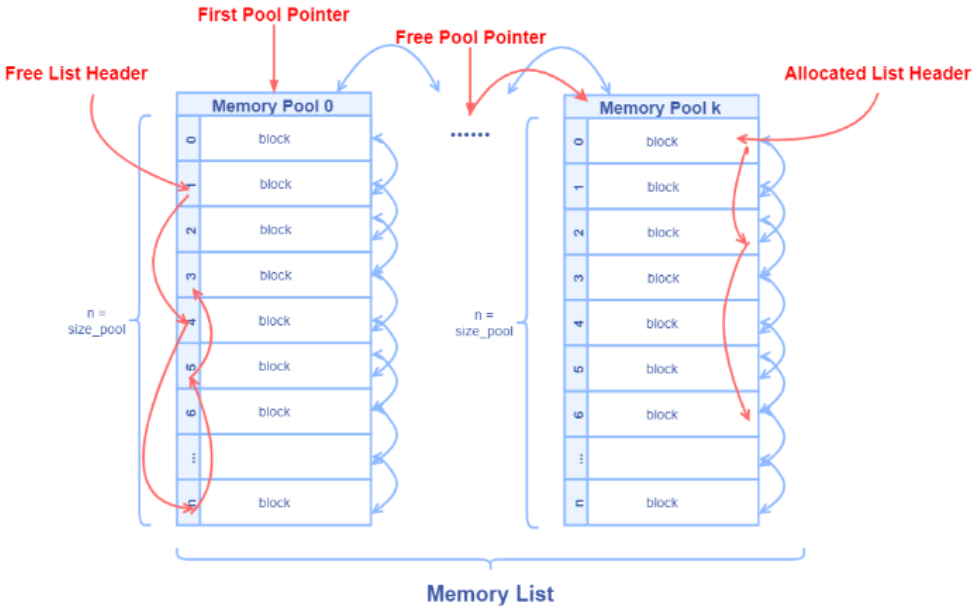

图1-1 内存池及内存池链表实现框架

内存分配的步骤如下：

1. Allocator通过内存池链表分配内存，调用内存池链表内的分配内存函数。
2. 内存池链表首先定位到当前的有空闲空间内存池，然后调用内存池内分配内存函数。若没有有空闲的内存池，则新建。
3. 内存池定位到空闲块，随后进行空间分配。

1.2 数据结构

1.2.1 内存池

- 块结构(block)

```
typedef struct block
{
    block* pre = nullptr;
    block* nex = nullptr;
}block;
```

- 内存池类(Mem_Pool)

```
class Mem_Pool {
public:
    block* allocH = nullptr; //header pointer for allocation
```

```
    block* freeH = nullptr; //header pointer for free block
    void* poolH = nullptr; //header pointer of the memory pool
    size_t num_block;   //number of blocks
    size_t size_block;   //size of blocks
    size_t size_pool;   //size of pool
    int id = 0;    //the sequence of block
    Mem_Pool* mpre;   //the previous pool
    Mem_Pool* mnex;   //the next pool

    Mem_Pool();    //initialize a space with default size
    Mem_Pool(size_t num, size_t size); //initialize a space with given blocks and size
    void* alloc(size_t size); //allocate a space of size "size"
    Mem_Pool* dealloc(Mem_Pool* cur, Mem_Pool* end, void* ptr); //deallocate the space ptr points at
    ~Mem_Pool(); //destroy the memory pool
};
```

- 内存池链表类(Mem_List)

```
class Mem_List {
public:
    Mem_Pool* first_pool;   //the first pool in the list
    Mem_Pool* free_pool;   //the first free pool in the list
    Mem_List(); //initialize a list
    void* alloc(size_t size);   //allocate a list with size "size"
    void dealloc(void* ptr);   //deallocate a space that ptr points at
};
```

1.2.2 STL 内存分配器

- 内存分配器类(Allocator)

```
template <typename T>
class Allocator
{
public:
    typedef T value_type;
    typedef T* pointer;
    typedef T& reference;
    typedef const T* const_pointer;
    typedef const T& const_reference;
    typedef std::size_t size_type;
    typedef std::ptrdiff_t difference_type;
    //convert an allocator<T> to allocator<U>
    template <typename U> struct rebind
    {
        typedef Allocator<U> Other;
    };

    inline Allocator() = default;
    inline ~Allocator() = default;
    inline Allocator(Allocator const&) = default;
    template<typename U>
    inline explicit Allocator(Allocator<U> const&) {}

    //address info
    inline pointer address(reference x) { return &x; }
    inline const_pointer address(const_reference x) { return &x; }
    //allocate and deallocate
    inline pointer allocate(size_type cnt, typename std::allocator<void>::const_pointer = 0) {
        return reinterpret_cast<pointer>(Mem_Pool.alloc(cnt * sizeof(T)));
    }
    inline void deallocate(pointer p, size_type n) {
        Mem_Pool.dealloc(p);
    }
    //return the size
    inline size_type max_size() const {
        return std::numeric_limits<size_type>::max() / sizeof(T);
    }
    //construction and destruction function
    inline void construct(pointer p, const T & t) { new(p) T(t); }
    inline void destroy(pointer p) { p->~T(); }
    //bool operator
    inline bool operator==(Allocator const&) { return true; }
    inline bool operator!=(Allocator const& a) { return !operator==(a); }

private:
    static Mem_List Mem_Pool;
};
```

标准库内存分配器类(allocator)

```
// TEMPLATE CLASS allocator
template class allocator
    : public _Allocator_base
{ // generic allocator for objects of class _Ty
public:
    typedef _Allocator_base _Mybase;
    typedef typename _Mybase::value_type value_type;
    typedef value_type _FARQ *pointer;
    typedef value_type _FARQ &reference;
    typedef const value_type _FARQ *const_pointer;
    typedef const value_type _FARQ &const_reference;

    typedef _SIZT size_type;
    typedef _PDFT difference_type;

    template struct rebind
    { // convert an allocator to an allocator
        typedef allocator other;
    };
    pointer address(reference _Val) const
    { // return address of mutable _Val
        return (&_Val);
    }
    const_pointer address(const_reference _Val) const
    { // return address of nonmutable _Val
        return (&_Val);
    }
    allocator() _THROW0()
    { // construct default allocator (do nothing)
    }
    allocator(const allocator &) _THROW0()
    { // construct by copying (do nothing)
    }
    template allocator(const allocator &) _THROW0()
    { // construct from a related allocator (do nothing)
    }
    template allocator &operator=(const allocator &)
    { // assign from a related allocator (do nothing)
        return (*this);
    }
    void deallocate(pointer _Ptr, size_type)
    { // deallocate object at _Ptr, ignore size
        ::operator delete(_Ptr);
    }
    pointer allocate(size_type _Count)
    { // allocate array of _Count elements
        return (_Allocate(_Count, (pointer)0));
    }
    pointer allocate(size_type _Count, const void _FARQ *)
    { // allocate array of _Count elements, ignore hint
        return (allocate(_Count));
    }
    void construct(pointer _Ptr, const _Ty &_Val)
    { // construct object at _Ptr with value _Val
        _Construct(_Ptr, _Val);
    }
    void destroy(pointer _Ptr)
    { // destroy object at _Ptr
        _Destroy(_Ptr);
    }
    _SIZT max_size() const _THROW0()
    { // estimate maximum array size
        _SIZT _Count = (_SIZT)(-1) / sizeof(_Ty);
        return (0 < _Count ? _Count : 1);
    }
};

// allocator TEMPLATE OPERATORS
template inline bool operator==(const allocator &, const allocator &) _THROW0()
{ // test for allocator equality (always true)
    return (true);
}

template inline bool operator!=(const allocator &, const allocator &) _THROW0()
{ // test for allocator inequality (always false)
    return (false);
}

```

- 内存池实现的内存分配器接口

```
template <typename T> Mem_List Allocator<T>::Mem_Pool;
```

标准库内存分配器模板接口

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

1.3 函数说明

1.2.1 内存池

内存池类(Mem_Pool)

- 构造函数(Mem_Pool(size_t num, size_t size);)

```
Mem_Pool::Mem_Pool(size_t num, size_t size)
{
    if (num < block_num)
        num_block = block_num;
    else
        num_block = num;
    if (size < block_size)
        size_block = block_size;
    else
        size_block = size;
    size_pool = (sizeof(block) + size_block) * num_block;

    poolH = ::operator new(size_pool); //create a pool and now poolH point to the pool
    mpre = mnex = nullptr;

    //block[i+1]->pre=block[i+2]
    //block[i+1]->nex=block[i]
    //freeH always points to block with largest index that is not allocated
    for (int i = 0; i < num_block; i++)
    {
        block* x = reinterpret_cast<block*>(static_cast<char*>(poolH) + i * (sizeof(block) + size_block));
        x->pre = nullptr;
        x->nex = freeH;
        if (freeH != nullptr)
            freeH->pre = x;
        freeH = x;
    }
}
```

- 默认构造函数(Mem_Pool();)

```
Mem_Pool::Mem_Pool()
{
    Mem_Pool(block_num, block_size);
}
```

- 析构函数(~Mem_Pool();)

```
Mem_Pool::~Mem_Pool()
{
    if (poolH != nullptr)
        ::operator delete(poolH);
}
```

- 分配内存函数(void* alloc(size_t size);)

  负责空间配置，返回一个t对象大小的空间。

```
void* Mem_List::alloc(size_t size)
{
    //if the current memory pool is full, then allocate in the next pool
    if (free_pool->freeH == nullptr)
    {
        //if there is no next free pool, initialize a new pool
        if (free_pool->mnex == nullptr)
        {
            free_pool->mnex = new Mem_Pool();
            free_pool->mnex->id = free_pool->id + 1;
        }
        //link the pools
        free_pool->mnex->mpre = free_pool;
        free_pool->mnex->mnex = nullptr;
```

```
        free_pool = free_pool->mnex;
    }
    return free_pool->alloc(size);
}
```

- 回收内存函数(Mem_Pool* dealloc(Mem_Pool* cur, Mem_Pool* end, void* ptr);)
  负责空间释放

```
Mem_Pool* Mem_Pool::dealloc(Mem_Pool* cur, Mem_Pool* end, void* ptr)
{
    if (cur == nullptr)
        return nullptr;
    else
    {
        void* tmpx = cur->poolH;
        //if ptr is in the current pool
        if (ptr > tmpx && ptr < (void*)((char*)tmpx + cur->size_pool))
        {
            //get to the current block
            block* now = reinterpret_cast<block*>(static_cast<char*>(ptr) - sizeof(block));
            if (ptr == cur->allocH)
            {
                allocH = allocH->nex;
                if (now->pre != nullptr)
                    now->pre->nex = now->nex; //relink the blocks
            }
            else
            {
                //relink the blocks
                if (now->nex != nullptr)
                    now->nex->pre = now->pre;
                if (now->pre != nullptr)
                    now->pre->nex = now->nex;
            }

            now->nex = cur->freeH; //add the current block to the free list
            if (cur->freeH != nullptr)
                cur->freeH->pre = now;
            return cur;
        }
        //if ptr is not in the current pool, seek it in the next pool
        else if (cur != end && cur->mnex != nullptr)
            return dealloc(cur->mnex, end, ptr);
        else
            return nullptr;
    }
}
```

内存池链表类(Mem_List)

- 构造函数(Mem_List();)

```
Mem_List::Mem_List()
{
    first_pool = new Mem_Pool();
    free_pool = first_pool;
}
```

- 分配内存函数(void* alloc(size_t size);)

```
void* Mem_List::alloc(size_t size)
{
    //if the current memory pool is full, then allocate in the next pool
    if (free_pool->freeH == nullptr)
    {
        //if there is no next free pool, initialize a new pool
        if (free_pool->mnex == nullptr)
        {
            free_pool->mnex = new Mem_Pool();
            free_pool->mnex->id = free_pool->id + 1;
        }
        //link the pools
        free_pool->mnex->mpre = free_pool;
        free_pool->mnex->mnex = nullptr;
        free_pool = free_pool->mnex;
    }
    return free_pool->alloc(size);
}
```

- 回收内存函数(void dealloc(void* ptr);)

```
void Mem_List::dealloc(void* ptr)
{
    //find ptr in memory pools
    Mem_Pool* cur = first_pool->dealloc(first_pool, free_pool, ptr);
    if (cur == nullptr)
    {
        ::operator delete(ptr);
        return;
    }
    if (cur->allocH != nullptr && cur != free_pool)
    {
        if (cur != first_pool) //squeeze current pool
        {
            cur->mpre->mnex = cur->mnex;
            cur->mnex->mpre = cur->mpre;
        }
        else
        {
            cur->mnex->mpre = nullptr;
            first_pool = first_pool->mnex;
        }
        cur->mnex = free_pool->mnex;
        if (free_pool->mnex != nullptr)
            free_pool->mnex->mpre = cur;
        cur->mpre = free_pool;
        free_pool = cur;
    }
}
```

1.2.2 STL 内存分配器

地址函数address

- 函数原型：

```
pointer address(reference _Val) const
{ // return address of mutable _Val
    return (&_Val);
}
const_pointer address(const_reference _Val) const
{ // return address of nonmutable _Val
    return (&_Val);
}
```

- 函数实现

```
//address info
inline pointer address(reference x) { return &x; }
inline const_pointer address(const_reference x) { return &x; }
```

- 函数功能

  返回指针地址信息

内存分配函数allocate

- 函数原型

```
pointer allocate(size_type _Count)
{ // allocate array of _Count elements
    return (_Allocate(_Count, (pointer)0));
}

pointer allocate(size_type _Count, const void _FARQ *)
{ // allocate array of _Count elements, ignore hint
    return (allocate(_Count));
}
```

- 函数实现

```
//allocate and deallocate
inline pointer allocate(size_type cnt, typename std::allocator<void>::const_pointer = 0)
{
    return reinterpret_cast<pointer>(Mem_Pool.alloc(cnt * sizeof(T)));
}
```

- 函数功能

  分配存储块，尝试分配n个T类型的存储空间，然后返回第一个元素的起始地址，只是分配空间，不构造对象。在标准默认allocator，存储块是使用一次或多次 ::operator new 进行分配，如果他不能分配请求的存储空间，则抛出bad_alloc异常。

内存回收函数deallocate

- 函数原型

```
void deallocate(pointer _Ptr, size_type)
{ // deallocate object at _Ptr, ignore size
    ::operator delete(_Ptr);
}
```

- 函数实现

```
inline void deallocate(pointer p, size_type n)
{
    Mem_Pool.dealloc(p);
}
```

- 函数功能

  释放先前allocate分配的且没有被释放的存储空间，Ptr指向以前使用allocator :: allocate分配的存储块的指针，size是在调用allocator :: allocate时为这个存储块分配的元素数量。在默认的allocator中，使用 ::operator delete进行释放。

构造函数construct

- 函数原型

```
void construct(pointer __p, const _Tp& __val)
{ ::new(__p) _Tp(__val); }
```

- 函数实现

```
inline void construct(pointer p, const T &t) { new (p) T(t); }
```

- 函数功能

  在Ptr指向的位置构建对象U，此时该函数不分配空间，pointer Ptr是allocate分配后的起始地址，constructor将其参数转发给相应的构造函数构造U类型的对象。

销毁函数destroy

- 函数原型

```
void destroy(pointer __p) { __p->~_Tp(); }
};
```

- 函数实现

```
inline void destroy(pointer p) { p->~T(); }
//bool operator
```

- 函数功能

  销毁p指向的对象，但是不会释放空间，也就意味着，这段空间依然可以使用，该函数使用T的析构函数，p->~T();

最大值函数max _size

- 函数原型

```
_SIZT max_size() const _THROW0()
{ // estimate maximum array size
    _SIZT _Count = (_SIZT)(-1) / sizeof(_Ty);
    return (0 < _Count ? _Count : 1);
}
```

- 函数实现

```
//return the size
inline size_type max_size() const
{
    return std::numeric_limits<size_type>::max() / sizeof(T);
}
```

- 函数功能

  返回最大可能分配的大小

布尔运算器bool operator

- 函数原型

```
// allocator TEMPLATE OPERATORS
template inline bool operator==(const allocator &, const allocator &) _THROW0()
{ // test for allocator equality (always true)
    return (true);
}

template inline bool operator!=(const allocator &, const allocator &) _THROW0()
{ // test for allocator inequality (always false)
    return (false);
}
```

- 函数实现

```
//bool operator
inline bool operator==(Allocator const &) { return true; }
inline bool operator!=(Allocator const &a) { return !operator==(a); }
```

- 函数功能

  运用布尔函数，一个返回真一个返回假，为了测试和输出运算结果的正确性。

rebind模板

- 模板原型

```
template struct rebind
{ // convert an allocator to an allocator
    typedef allocator other;
};
```

- 模板实现

```
//convert an allocator<T> to allocator<U>
template <typename U>
struct rebind
{
    typedef Allocator<U> Other;
};
```

- 模板功能

  定义了一种结构，它使一种类型的对象的分配器能够为另一种类型的对象分配存储空间。给定了类型T的分配器Allocator=allocator<T>，现在想根据相同的策略得到另外一个类型U的分配器allocator<U>，那么allocator<U>=allocator<T>::Rebind<U>::other。

2 测试结果及分析

2.1 测试结果

我们分别选取了10000，12000，14000，16000，18000个vector，从中随机选取了1000个随机的vector并对其用随机的大小resize，最后释放所有的vector，记录时间以及vecints和vecpts的结果。

因为是随机的所以重复了五次系统在10000个vector时的表现。

表2-1 vector=10000时std::allocator的5次表现

次数	1	2	3	4	5
vecints	6394	2849	5462	2347	1587
vecpts	2061	2653	2673	6661	690
time/s	8.87646	8.19436	8.90245	8.89644	9.04124

取平均值得vecints = 3727.8，vecpts = 2947.6，time = 8.78219s。

分别用系统的std::allocator和内存池实现的Allocator进行测试得出结果如下表，其中在vector数量为18000的时候由于可能系统内存等限制，Allocator已经无法返回结果。

表2-2 std:allocator在不同数量vector下的表现

std::allocator	10000	12000	14000	16000	18000
vecints	3728	4596	13821	13924	9371
vecpts	2948	5974	12195	934	17826
time/s	8.7822	13.0877	17.5072	24.9772	29.5988

表2-3 Allocator在不同数量vector下的表现

Allocator	10000	12000	14000	16000	18000
vecints	7364	10765	4306	10437	\
vecpts	1277	11621	3196	454	\
time/s	13.7185	16.8264	24.8769	31.8043	\

2.2 结果分析

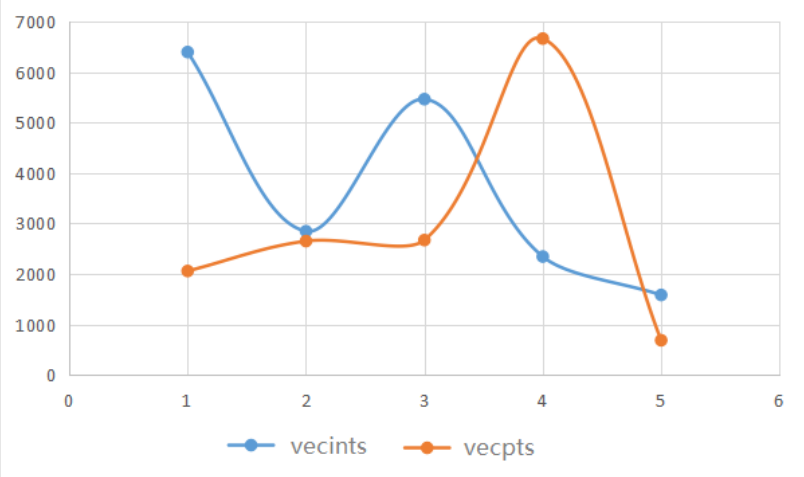

图2-1 vector=10000时5次std:allocator的vecints和vecpts

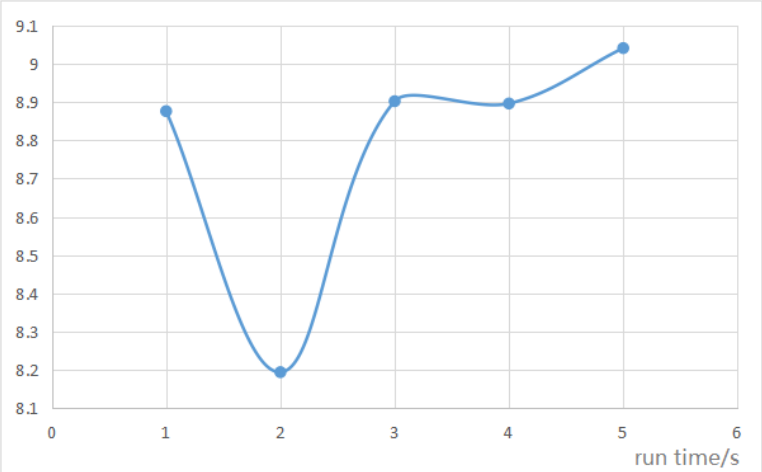

图2-2 vector=10000时5次std:allocator的run time

使用random函数随机resize所以记录5次结果作为采样，可以看到run time、vecints、vecpts都是在一个比较稳定的范围内变化的，run time最大和最小值之间的变化不超过1s，数据整体比较稳定。而vecints和vecpts之间没有确定的大小关系，最大最小值之间的差异较大，充分满足了随机性。

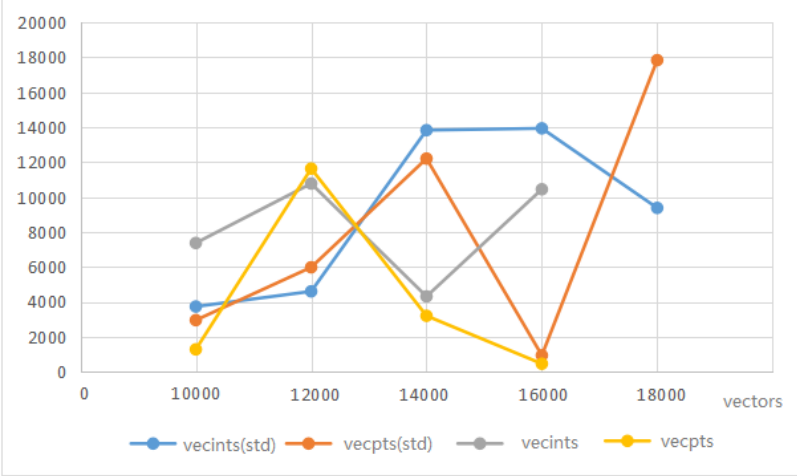

图2-3 不同vector数量下两种内存分配器的vecints和vecpts

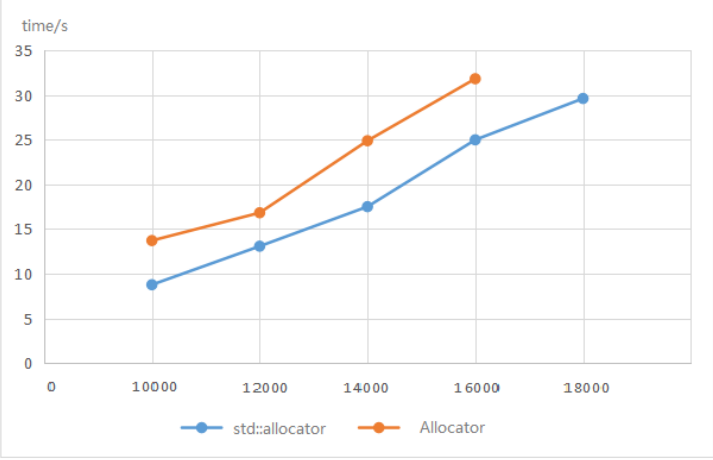

图2-4 不同vector数量下两种内存分配器的run time

在不同的vector数量下用系统的std::allocator和内存池实现的Allocator性能比较如上图所示，Allocator整体的运行速度不如std::allocator，而且vector数量越大，它们之间的差距越大，因此可以得出结论：内存池的优势主要在于分配小对象的时候提高代码运行速度，在对象数量比较大的时候表现不如系统自带的内存分配器，我们可以选取数量更少的vector比如8000来进一步验证这一结论。而且在vector数量过大到18000时，Allocator就难以运算出结果，其稳定性不如系统的std:allocator。

从vecints、vecpts可以看出，与系统管理内存相比，在vector数量适中的时候内存池占用的空间相对来说是比较小的，这也体现了它在性能优化方面的优点：

1. 针对特殊情况，例如需要频繁分配释放固定大小的内存对象时，不需要复杂的分配算法和多线程保护。也不需要维护内存空闲表的额外开销，从而获得较高的性能。
2. 由于开辟一定数量的连续内存空间作为内存池块，因而一定程度上提高了程序局部性，提升了程序性能。
3. 比较容易控制页边界对齐和内存字节对齐，没有内存碎片的问题。

3 小组分工及心得体会

3.1 小组分工

Memory Pool & List & 报告

STL Allocator & Test & 报告

3.2 心得体会

本次作业让我们了解了C++内存分配的原理，并且运用原理涉及了自己的内存分配机制。在完成作业的同时，我们对模板、容器的概念和使用规范更加熟练，对函数接口的定制有了更深的理解，运用C++语言的技巧有所提高。