

浙江大学



课程名称：____面向对象程序设计____实验类型：____综合____

实验项目名称：____STL allocator + memory pool____

学生姓名：____董佳昕____专业：____计算机科学与技术____学号：____3200105446____

同组学生姓名：____无____指导老师：____许威威____

实验地点：____紫金港机房____实验日期：____2022____年____6____月____1____日

目录

目录

一、实验目的

二、实验方案

- 2.1 STL allocator的定义
- 2.2 基于malloc与free函数实现的BasicAllocator模板类
 - 2.2.1 各个成员函数的实现
 - 2.2.2 BasicAllocator的大致表现
- 2.3 基于简易内存池策略实现的BlockBufferPool模板类
 - 2.3.1 分配内存: allocate
 - 2.3.2 释放内存: deallocate
 - 2.3.3 具体实现
 - 2.3.4 BlockBufferPool的大致表现
- 2.4 在BlockBufferPool基础上的改进尝试: DiverseBlockBufferPool模板类
- 2.5 基于FreeList实现的FreeListBufferPool
 - 2.5.1 对申请到的大内存进行分割
 - 2.5.2 分配内存: allocate
 - 2.5.3 释放内存: deallocate
 - 2.5.4 申请空间填充Free_List链表: refill
 - 2.5.5 从内存池中分配size*n大小的空间: chunk_alloc
 - 2.5.6 具体实现
 - 2.5.7 FreeListBufferPool的大致表现

三、实验结果

- 3.1 std::allocator
- 3.2 BasicAllocator
- 3.3 BlockBufferPool
- 3.4 BlockBufferPool without deallocate
- 3.5 DiverseBlockBufferPool
- 3.6 FreeListBufferPool

一、实验目的

1. 借助内存池策略，实现自定义的STL allocator
2. 提高小内存的分配效率

使用标准：C++98

使用说明：

1. 请使用VS2022打开此工程
2. 由于使用的是模板类编程，模板类的实现函数必须放在头文件中，因此，整个程序只有 **testallocator.cpp** 一个cpp文件，每种方法的实现放在了不同的.h文件中
3. 测试对应的实现方式时，要将对应的代码一并取消注释
4. 为了防止冲突，每次只能测试一种实现策略，不同实现策略所需要取消注释的代码如下图所示

1. std::allocator

```
#define MyAllocator std::allocator
const string NowAllocator = "std::allocator";
void FreeAll(){}
```

2. BasicAllocator

```
#include "BasicAllocator.h"
#define MyAllocator BasicAllocator
const string NowAllocator = "BasicAllocator";
void FreeAll(){}
```

3. BlockBufferPool

```
#include "BlockBufferPool.h"
#define MyAllocator BlockBufferPool
const string NowAllocator = "BlockBufferPool";
```

4. DiverseBlockBufferPool

```
#include "DiverseBlockBufferPool.h"
#define MyAllocator DiverseBlockBufferPool
const string NowAllocator = "DiverseBlockBufferPool";
```

5. FreeListBufferPool

```
#include "FreeListBufferPool.h"
#define MyAllocator FreeListBufferPool
const string NowAllocator = "FreeListBufferPool";
void FreeAll(){}
```

二、实验方案

2.1 STL allocator的定义

STL allocator是一个模板类，要求具有以下成员类型、成员变量、成员函数

```

1  template <typename T>
2  class allocator{
3  public:
4      typedef T                value_type;
5      typedef value_type*      pointer;
6      typedef value_type&      reference;
7      typedef value_type const* const_pointer;
8      typedef value_type const& const_reference;
9      typedef size_t           size_type;
10     typedef ptrdiff_t        difference_type;
11
12     template <typename _other> struct rebind {
13         typedef allocator<_other> other;
14     }
15 public:
16     pointer address(reference val) const;
17     const_pointer address(const_reference val) const;
18     pointer allocate(size_type cnt, char* pHint = 0);
19     void deallocate(pointer p, size_type n);
20     size_type max_size() const throw();
21     void construct(pointer p, const_reference val);
22     void destory(pointer p);
23 public:
24     allocator() throw() {}
25     allocator(const_reference val) throw() {}
26     template<typename _other>
27     allocator(allocator<_other> const &val) throw() {}
28     ~allocator() noexcept {}
29 public:
30     bool operator==(const allocator& A) const;
31     bool operator!=(const allocator& A) const;
32 };

```

其中，由于**allocator**本身并没有成员变量，且根据C++标准的定义，任意两个同类的**allocator**实例永远视作相等的，因此，我们的重点就在于实现以下7个成员函数：

```

1  pointer address(reference val) const;
2  const_pointer address(const_reference val) const;
3  pointer allocate(size_type cnt, char* pHint = 0);
4  void deallocate(pointer p, size_type n);
5  size_type max_size() const throw();
6  void construct(pointer p, const_reference val);
7  void destory(pointer p);

```

接下来，我将先使用**malloc()**与**free()**函数，实现一个基础的**allocator**模板类，以加强我们对**allocator**的理解

2.2 基于malloc与free函数实现的BasicAllocator模板类

2.2.1 各个成员函数的实现

根据**allocator**的各种成员函数的用途，我们可以很轻松的实现7个成员函数

1. **address**函数：要求返回传入数据的地址

- 我们只需要直接借助**&**操作符即可

```

1 // 返回val的地址
2 pointer address(reference val) const {
3     return &val;
4 }
5 const_pointer address(const_reference val) const {
6     return &val;
7 }

```

2. max_size函数：要求返回当前类最多可容纳的元素个数

- 由于容器本身能够容纳的元素个数是根据当前能够申请到的内存决定的，因此这个函数其实并不是那么重要
- 在实现上，我选择了直接返回0xFFFFFFFF / 当前单个元素的大小

```

1 // 可容纳的最多元素
2 size_type max_size() const throw() {
3     return static_cast<size_type>(-1) / sizeof(value_type);
4 }

```

3. construct函数：要求将val的值填入p指向的空间

- 这里我们需要使用到**placement new**，以保证调用到构造函数
- **placement new**：在用户指定的内存位置上构建新的对象，构建过程中不需要额外分配内存，只需要调用对象的构造函数

```

1 // 在地址p所指向的空间，使用val进行填充
2 void construct(pointer p, const_reference val) {
3     new (p) value_type(val);
4 }

```

4. destory函数：析构p指向的内存块中的内容

- 我们只需要调用对象的析构函数即可

```

1 void destory(pointer p) {
2     p->~T();
3 }

```

5. allocate函数：分配cnt个对象的内存地址，并返回这块内存的起始地址

- 直接调用**malloc()**函数，分配cnt * sizeof(T) 个字节的内存即可

```

1 pointer allocate(size_type cnt, char* pHint = 0){
2     if (cnt <= 0) return nullptr;
3     if (max_size() < cnt) throw std::bad_alloc();
4     void* pMem = malloc(cnt * sizeof(value_type));
5     if (pMem == nullptr) throw std::bad_alloc();
6     return static_cast<pointer>(pMem);
7 }

```

6. deallocate函数：分配指针p指向一块内存，内存大小为n个对象

- 直接调用**free()**函数，释放内存

```
1 void deallocate(pointer p, size_type n) {
2     free(p);
3 }
```

在上述的实现过程中，我们发现，不同分配的策略的区别仅仅在与`allocate`函数和`deallocate`函数的不同，其他函数都是一样的。因此，在后面的实现方式中，我们仅讨论这两个函数的实现方式，其余5个成员函数的实现方式与`BasicAllocator`的实现方式相同

2.2.2 BasicAllocator的大致表现

由于`BasicAllocator`与系统默认的`allocator`的实现思路大致相同，最后的表现也与系统默认的`allocator`不相上下

2.3 基于简易内存池策略实现的BlockBufferPool模板类

在编写`BasicAllocator`模板类的时候，我们可以发现，由于每一次执行`allocate`和`deallocate`函数，我们都需要调用`malloc()`和`free()`函数申请/释放内存，这样会导致一个弊端：如果我们要连续申请多次小内存，我们会多次调用这两个系统函数访问内存，导致我们访问内存的时间变得非常多。

如果我们能够提前申请一大块到缓冲区，当我们需要分配小块内存的时候，我们将缓冲区的一部分直接分配给调用者，这样我们就可以通过减少对内存的申请，来提高分配内存的效率。

具体来说，我们维护了一个链表`current_Block`，指向我们现在申请下来，并且还没有被分配给调用者的空间

2.3.1 分配内存：allocate

1. 首先，判断当前需要申请的内存是否超出一大块`BlockSize`的大小，如果超出了，我们直接调用`malloc`函数进行分配。如果没有超出，则执行以下操作
2. 查看现在的内存池`current_Block`中是否还有剩下的空间，如果有，则直接分配给调用者，并将当前块减小相应的大小
3. 如果没有足够的空间，则调用`malloc`函数，申请一大块内存放入`BlockSize`中。由于在第1步中，我们已经解决了待分配内存 $> \text{BlockSize}$ 的情况，此时的待分配内存必然 $\leq \text{BlockSize}$ ，我们只需要再执行一遍第2步，即可将内存分配出去

2.3.2 释放内存：deallocate

1. 根据函数的定义，我们需要释放的是 $n * \text{sizeof}(T)$ 个字节的内存
2. 为了减小调用`malloc`和`free`的次数，我们此时并不会真正将内存还给系统，而是选择将这块内存视为能够被分配的内存，放入`current_Block`中
3. 在程序结束的时候，我们再通过调用`FreeAll()`函数，将从系统中申请到的内存全部还给系统

2.3.3 具体实现

1. Block结构体

```
1 // 内存池，以Block为单位
2 struct Block {
3     char* start_address; // 当前Block的起始地址
4     bool isAllocate;     // 表明当前Block是否是通过allocate分配的
5     char* now_address;   // 当前Block剩余空间的地址
6     int last_size;       // 当前Block剩余的空间
7     Block* next;         // Block链表的下一个元素
8 };
9 Block* current_Block; // 内存块链表的头指针;
```

2. 分配内存

```
1  template <typename T, size_t BlockSize>
2  typename BlockBufferPool<T, BlockSize>::pointer
3  BlockBufferPool<T, BlockSize>::allocate(size_type cnt, const_pointer
4  pHint) {
5      if (cnt <= 0) return nullptr;
6      size_t num = cnt * sizeof(value_type);
7      pointer ans = nullptr;
8      Block* now = nullptr;
9
10     // 待分配的内存比一个Block大，则直接调用malloc分配内存
11     if (num > BlockSize) {
12         now = allocateBlock(nullptr, num);
13         ans = reinterpret_cast<pointer>(now->start_address);
14         now->now_address = nullptr;
15         now->last_size = 0;
16         return ans;
17     }
18
19     // 遍历current_Block，判断是否有Block块能够放入数据
20     now = current_Block;
21     while (now != nullptr) {
22         if (now->last_size < num) { now = now->next; continue; }
23         else break;
24     }
25     // 现有的Block都放不进去，则重新申请一块内存
26     if (now == nullptr) now = allocateBlock(nullptr, BlockSize);
27
28     // 将当前元素放入now指向的Block中
29     ans = reinterpret_cast<pointer>(now->now_address);
30     now->now_address += num;
31     now->last_size -= num;
32     return ans;
33 }
```

3. 释放内存

```
1  template <typename T, size_t BlockSize>
2  void BlockBufferPool<T, BlockSize>::deallocate(pointer p, size_type size{
3      if (p == nullptr) return;
4      // 将p指向的size大小的内存，作为新的Block块放入current_Block中
5      allocateBlock(reinterpret_cast<char*>(p), size);
6  }
```

4. 维护current_Block的链表结构，这个函数需要实现的功能有以下几点

1. 申请一块新的内存，放到链表中
2. 由于调用了deallocate函数，会有一块新的内存可供分配，我们将其作为一个新的块放入链表之中

```
1  template <typename T, size_t BlockSize>
2  Block* BlockBufferPool<T, BlockSize>::allocateBlock(char* p, int size) {
3      Block* now = new Block();
4      if (now == nullptr) {
5          cout << "new Block() error\n";
6      }
```

```

6         throw std::bad_alloc();
7     }
8     // 申请一块新的内存
9     if (p == nullptr) {
10         p = reinterpret_cast<char*>(malloc(size));
11         now->isAllocate = true;
12     }
13     // deallocate释放掉的内存
14     else {
15         now->isAllocate = false;
16     }
17     now->start_address = p;
18     now->now_address = p;
19     now->last_size = size;
20     now->next = current_Block;
21     current_Block = now;
22     return now;
23 }

```

5. 程序结束，将所有的内存还给系统

```

1 void FreeAll() {
2     Block* now = current_Block;
3     while (now != nullptr) {
4         Block* next = now->next;
5         if (now->isAllocate) { free(now->start_address);}
6         delete now;
7         now = next;
8     }
9     current_Block = nullptr;
10 }

```

2.3.4 BlockBufferPool的大致表现

从理论上来说，由于减少了malloc和free的次数，实际的表现应该会比BasicAllocator好一点。但从实际运行结果来看，两者的运行时间差距并不大。

2.4 在BlockBufferPool基础上的改进尝试： DiverseBlockBufferPool模板类

这种实现方式的提出，是因为我在测试BlockBufferPool类的时候，发现如果我们在deallocate的时候，不去将p对应的内存放入current_Block中，时间就会少很多，因此就想到了这一种实现方法。但是由于最后测试的时候，发现运行时间反而更长了，并且又找到了一个更优的实现方法，因此这里只做简述。

具体来说，这个方法就是维护了三种链表：

1. **BigBlock**：表示我向系统直接申请的内存块，用于最后将内存全部返还给系统
2. **used_DiverseBlock**：表示当前已经使用了的内存块
3. **free_DiverseBlock**：表示当前没有被使用的内存块

当我们从系统中申请到一块内存(大小为512B)时，我们会将其切分为2个32B，1个64B，1个128B，1个256B的内存块，存入free_DiverseBlock中，并将这整个512B的内存放入BigBlock中

当分配内存的时候，我们直接去free_DiverseBlock中寻找到对应的内存块，并分配给调用者，并将这个内存块放入used_DiverseBlock中

当释放内存的时候，我们将p对应的Block块从used_DiverseBlock转移到free_DiverseBlock中

当程序结束时，我们遍历BigBlock，执行free操作

2.5 基于FreeList实现的FreeListBufferPool

分析2.3的代码，我们可以看到，由于内存页是通过新建的Block类及逆行维护的，实际上每一次allocate和deallocate，我们都需要通过额外次数的new操作维护current_Block类，这大大增加了我们申请内存的次数。

除此之外，由于申请的一块大内存的大小是固定的，这就导致我们每一次分配再释放内存，会出现很多的内存碎片，大大降低了我们对内存的使用效率。

通过以上分析，我们可以得出以下两个改进措施

1. 维护空余内存的链表，不是通过额外的空间来维护，而是直接将链表信息写入到申请到的空内存，这样可以减少对内存的使用
2. 将申请到的大内存分解为有梯度的一系列小块，在allocate的时候，我们根据需要分配的内存大小，有选择的分配一个小块返回，这样可以将内存碎片保证在一定的大小范围之内

2.5.1 对申请到的大内存进行分割

1. 维护16个Free_List链表，分别指向内存大小为8B, 16B, 24B, 32B,...128B的内存块
2. 通过union结构体union Block维护链表结构，将链表的next指针直接写入到申请到的内存的首地址
3. 在全局保存16个Block *的指针，保存着16个内存块的头指针
4. 在全局记录两个char *类型的指针start_free, end_free，表示当前从系统申请到的，还没有被分配的堆内存，同时记录一个size_t类型的变量heap_size表示其内存大小

2.5.2 分配内存：allocate

1. 首先判断待申请内存大小size = n * sizeof(value_type)*是否大于MAX_MEMORY = 128B
2. 如果大于，直接调用malloc()进行分配
3. 如果不大于，则首先根据size的大小，判断其是属于哪个内存块等级的，找到对应的Free_List链表头指针，记为now
4. 如果now不为空，代表当前还有空闲空间，直接将now对应的空间分配个调用者
5. 如果now为空，代表当前种类的内存块没有空闲空间了，则调用refill函数申请一块内存，然后分配给调用者
6. 对于refill函数，我们会在2.5.4进行讲解

2.5.3 释放内存：deallocate

1. 首先判断待申请内存大小size = n * sizeof(value_type)*是否大于MAX_MEMORY = 128B
2. 如果大于，直接调用free()进行释放
3. 如果不大于，则首先根据size的大小，判断其是属于哪个内存块等级的，找到对应的Free_List链表头指针，记为now
4. 由于分配的时候是按照内存块进行分配的，当前指针p对应的数据占用的实际大小一定与now指向的内存大小相同，因此我们可以将p插入到Free_List链表之中即可实现dellocate操作

2.5.4 申请空间填充Free_List链表：refill

1. 首先，我们调用chunk_alloc函数，尝试申请n * size的内存
 1. n是自定义的一个值，为了减少refill的次数，在本程序中设置为20，表示我们一次性申请至多20个内存块
 2. size是待填充的Free_List链表对应的内存块大小
 3. chunk_alloc函数会修改n的值，表示我们实际上申请了几个内存块

2. 如果我们只能申请到1个内存块，则直接将这个内存块分配给调用者
3. 如果我们申请到 ≥ 2 个的内存块，则
 1. 将第0块内存块分配给调用者
 2. 将第1~n-1块内存块放入**Free_List**中
 3. 具体来说，就是借助**union**结构体，将当前空闲内存块的首地址填充为下一个内存块的指针，以维护链表结构
 4. 最后，将第0块内存块的地址返回
4. 对于**chunk_alloc**函数，我们会在2.5.5中进行讲解

2.5.5 从内存池中分配size*n大小的空间：chunk_alloc

1. 首先计算当前内存池中剩余的内存空间大小：**bytes_left = end_free - start_free**；待分配的内存空间大小：**total_size = n * size**
2. 如果内存池剩余空间能够完全满足要求，即**bytes_left >= total_size**
 - 则直接从内存池中分配**total_size**的空间给调用者
3. 如果内存池剩余空间不能完全满足需求，但至少能分配一个块，即**size < bytes_left < total_size**
 - 则尽可能多的满足需求
 - 同时修改**n**的值，以告诉调用者实际分配了多少空间
4. 如果内存池剩余空间无法分配一个区块
 - 先将内存池中剩余的空间分配给适当的**Free_List**
 - 然后调用**malloc()**函数，申请一大块内存
 - 将申请到的内存放入内存池中，递归调用自己，分配申请到的内存
 - 如果**malloc()**失败，则检查当前的**Free_List**，看是否有比**size**大的剩余空间。如果有，则将其还给内存池，同时递归调用自己，分配这一块内存
 - 如果**Free_List()**里面也没有剩余空间，则可以判定内存分配失败，抛出异常

2.5.6 具体实现

1. 辅助函数及变量

```
1  const int ALIGN = 8;           //小型区块对应的大小为8的倍数
2  const int MAX_MEMORY = 128;    // 小型区块最大为128B
3  const int CNT_OF_LIST = MAX_MEMORY / ALIGN; //free-lists一共有16个
4
5  //将size上调为ALIGN的倍数
6  static size_t convert_size_to_size(size_t size) {
7      return (size + ALIGN - 1) & ~(ALIGN - 1);
8  }
9
10 //根据size大小判断当前区块属于哪个FreeList
11 static size_t convert_size_to_index(size_t size) {
12     return (size + ALIGN - 1) / ALIGN - 1;
13 }
14
15 union Block {
16     union Block* next; // Free_List链表指向的下一块内存
17     char data;         // 当前内存存放的数据
18 };
19 Block* Free_List[CNT_OF_LIST];
20 char* start_free = nullptr; // 内存池开始地址
21 char* end_free = nullptr;   // 内存池结束地址
22 size_t heap_size;           // 内存池的大小
```

2. 分配内存

```
1  template <typename T>
2  typename FreeListBufferPool<T>::pointer
3  FreeListBufferPool<T>::allocate(size_type cnt, char* pHint) {
4      int size = cnt * sizeof(value_type);
5      // 大于MAX_MEMORY, 直接用malloc分配内存
6      if (size > MAX_MEMORY) {
7          return reinterpret_cast<pointer>(malloc(size));
8      }
9
10     int index = convert_size_to_index(size);
11     Block* now = Free_List[index];
12     // 链表中无空闲空间, 调用refill申请一块内存
13     if (now == nullptr) {
14         void* ans = refill(convert_size_to_size(size));
15         return reinterpret_cast<pointer>(ans);
16     }
17     // 链表中有空闲空间, 直接返回
18     Free_List[index] = Free_List[index]->next;
19     return reinterpret_cast<pointer>(now);
20 }
```

3. 释放内存

```
1  template <typename T>
2  void FreeListBufferPool<T>::deallocate(pointer p, size_type n) {
3      int size = n * sizeof(value_type);
4      // 大于MAX_MEMORY, 直接用free释放内存
5      if (size > MAX_MEMORY) {
6          free(p);
7          return;
8      }
9      // 将p对应的内存放到Free_List中
10     int index = convert_size_to_index(size);
11     Block* now = reinterpret_cast<Block*>(p);
12     now->next = Free_List[index];
13     Free_List[index] = now;
14 }
15
```

4. 申请空间填充Free_List链表

```
1  void* refill(size_t size) {
2      int n = 20;
3      // 尝试申请n*size的空间
4      char* chunk = chunk_alloc(size, n);
5
6      // 如果只能申请到1*size的空间, 则直接分配给调用者, Free_List没有新节点
7      if (n == 1) return reinterpret_cast<void*>(chunk);
8
9      // 申请到>=2*size的空间
10     int index = convert_size_to_index(size);
11     // 将第0块分配给调用者
12     Block* ans = reinterpret_cast<Block*>(chunk);
13     // 将第1~n-1块空间放到Free_List中
```

```

14     Block* now_Block = nullptr;
15     Block* next_Block = reinterpret_cast<Block*>(chunk + size);
16     Free_List[index] = next_Block;
17     for (int i = 1;; i++) {
18         now_Block = next_Block;
19         next_Block = reinterpret_cast<Block*>((char*)next_Block + size);
20         if (i == n - 1) {
21             now_Block->next = nullptr;
22             break;
23         }
24         else {
25             now_Block->next = next_Block;
26         }
27     }
28     return reinterpret_cast<void*>(ans);
29 }

```

5. 从内存池中分配size*n大小的空间

```

1  char* chunk_alloc(size_t size, int& n) {
2      size_t total_size = n * size;
3      size_t bytes_left = end_free - start_free;
4
5      // 内存池剩余空间完全满足需求
6      if (bytes_left >= total_size) {
7          char* ans = start_free;
8          start_free += total_size;
9          return ans;
10     }
11
12     // 内存池剩余空间不能完全满足需求，但至少能分配一个块
13     else if (bytes_left >= size) {
14         n = bytes_left / size;
15         total_size = n * size;
16         char* ans = start_free;
17         start_free += total_size;
18         return ans;
19     }
20
21     // 内存池剩余空间无法分配一个区块
22     else {
23         // 先将内存池中剩余的空间分配给适当的Free_List
24         if (bytes_left > 0) {
25             int index = convert_size_to_index(bytes_left);
26             reinterpret_cast<Block*>(start_free)->next =
Free_List[index];
27             Free_List[index] = reinterpret_cast<Block*>(start_free);
28         }
29
30         size_t bytes_to_get = 2 * total_size +
convert_size_to_size(heap_size >> 4);
31         start_free = (char*)malloc(bytes_to_get);
32
33         // 堆空间不足，malloc失败
34         if (start_free == nullptr) {
35             // 检查当前Free_List，看是否有比size大的空闲空间，并将其分裂
36             for (int i = size; i <= MAX_MEMORY; i += ALIGN) {

```

```

37         int index = convert_size_to_index(i);
38         Block* now = Free_List[index];
39         if (now != nullptr) {
40             Free_List[index] = now->next;
41             // 将当前的空间放入内存池
42             start_free = reinterpret_cast<char*>(now);
43             end_free = start_free + i;
44             // 递归调用chunk_alloc, 进行重新分配
45             return chunk_alloc(size, n);
46         }
47     }
48     // Free_List中也没有空余空间, 可以放弃治疗了
49     throw bad_alloc();
50 }
51
52 // malloc成功
53 heap_size += bytes_to_get;
54 end_free = start_free + bytes_to_get;
55 // 递归调用自己, 分配malloc到的内存, 修改n
56 return chunk_alloc(size, n);
57 }
58 }

```

2.5.7 FreeListBufferPool的大致表现

实际运行上看, **FreeListBufferPool**确实会比前面两种实现方式快上不少, 与**std::allocate**相比, 也是快一点的

三、实验结果


分配策略	1次	5次	10次	20次	30次	40次	50次	平均时长
std::allocator	19	90	176	348	523	704	883	17.813
BasicAllocator	19	89	177	355	522	684	848	17.673
BlockBufferPool	19	93	186	371	570	754	953	18.809
BlockBufferPool without deallocate	14	67	131	266	393	522	649	13.276
DiverseBlockBufferPool	682	3443	×	×	×	×	×	685.30
FreeListBufferPool	17	75	147	290	431	582	732	14.965

从结果上看, 可以得出以下结论

1. 由于各种因素的影响, **std::allocator**、**BasicAllocator**、**BlockBufferPool**的时间相差不大
2. **BlockBufferPool without deallocate**的运行速度最快, 但是会产生大量的内存泄露, 大量的内存会被闲置
3. **FreeListBufferPool**的运行速度与**std::allocator**、**BasicAllocator**、**BlockBufferPool**相比提升明显, 且不会产生太多的内存泄漏, 是目前的最优选择
4. **DiverseBlockBufferPool**不知道为啥, 与预期差距极大, 甚至产生了大量的越界的错误.....


下面是运行结果的截图

3.1 std::allocator

 Microsoft Visual Studio 调试控制台


```
use std::allocator for 1 tests used 19 ms
use std::allocator for 5 tests used 90 ms
use std::allocator for 10 tests used 176 ms
use std::allocator for 20 tests used 348 ms
use std::allocator for 30 tests used 523 ms
use std::allocator for 40 tests used 704 ms
use std::allocator for 50 tests used 883 ms
```

3.2 BasicAllocator

 Microsoft Visual Studio 调试控制台


```
use BasicAllocator for 1 tests used 19 ms
use BasicAllocator for 5 tests used 89 ms
use BasicAllocator for 10 tests used 177 ms
use BasicAllocator for 20 tests used 355 ms
use BasicAllocator for 30 tests used 522 ms
use BasicAllocator for 40 tests used 684 ms
use BasicAllocator for 50 tests used 848 ms
```

3.3 BlockBufferPool

 Microsoft Visual Studio 调试控制台

```
use BlockBufferPool for 1 tests used 19 ms
use BlockBufferPool for 5 tests used 93 ms
use BlockBufferPool for 10 tests used 186 ms
use BlockBufferPool for 20 tests used 371 ms
use BlockBufferPool for 30 tests used 570 ms
use BlockBufferPool for 40 tests used 754 ms
use BlockBufferPool for 50 tests used 953 ms
```

3.4 BlockBufferPool without deallocate

 Microsoft Visual Studio 调试控制台

```
use BlockBufferPool without deallocate for 1 tests used 14 ms
use BlockBufferPool without deallocate for 5 tests used 67 ms
use BlockBufferPool without deallocate for 10 tests used 131 ms
use BlockBufferPool without deallocate for 20 tests used 266 ms
use BlockBufferPool without deallocate for 30 tests used 393 ms
use BlockBufferPool without deallocate for 40 tests used 522 ms
use BlockBufferPool without deallocate for 50 tests used 649 ms
```

3.5 DiverseBlockBufferPool

```
use DiverseBlockBufferPool for 1 tests used 682 ms  
use DiverseBlockBufferPool for 5 tests used 3443 ms
```

3.6 FreeListBufferPool

```
use FreeListBufferPool for 1 tests used 17 ms  
use FreeListBufferPool for 5 tests used 75 ms  
use FreeListBufferPool for 10 tests used 147 ms  
use FreeListBufferPool for 20 tests used 290 ms  
use FreeListBufferPool for 30 tests used 431 ms  
use FreeListBufferPool for 40 tests used 582 ms  
use FreeListBufferPool for 50 tests used 732 ms
```