实验一：Bigsize

vector create过程，创建vector数量N=1e4，每一vector分配的范围在[1，1e4]之间。

vector resize过程，每一vector进行resize的大小范围在[1, 1e4]之间。

|  |  |  |  |
| --- | --- | --- | --- |
| Allocator类型  测试项目 | Std::allocator | Malloc\_alloc | MemoryPool\_alloc |
| create, int | 0.169356 | 0.544501 | 0.527737 |
| create, Myclass | 0.327492 | 0.527253 | 0.039883 |
| resize, int | 0.007866 | 0.071805 | 0 |
| resize, Myclass | 0.064506 | 0.082872 | 0.011512 |

Std：：allocator必然是经过更加细致的优化的。在create的过程它的表现十分突出。MemoryPool的表现和Malloc接近，因为MemoryPool本身初始的时候free\_list中的可用区块很少，大多还是直接通过malloc分配出来，所以效率相差不大。

但是resize的时候MemoryPool的优势就开始体现出来。因为resize的时候vector长度有变大有变小，整体总内存占用基本保持不变，比较符合memorypool的高效场景。Memorypool因为管理内存的机制不同，最大程度地省略了先free（）再malloc（）的重复工作，提高效率的同时也能有效减少内存碎片的产生，可以看到MemoryPool在resize的时候表现是十分不错的

实验二：Smallsize

vector create过程，创建vector数量N=1e4，每一vector分配的范围在[1，1e4]之间。

vector resize过程，每一vector进行resize的大小范围在[1, 100]之间。

|  |  |  |  |
| --- | --- | --- | --- |
| Allocator类型  测试项目 | Std::allocator | Malloc\_alloc | MemoryPool\_alloc |
| resize, int | 0 | 0.030674 | 0.028324 |
| resize, Myclass | 0.019846 | 0.037222 | 0.03725 |

这些时候反而是std::allocator的表现更好。猜测std::allocator中也有内存池协助管理内存，只是支持的最大区块比较小，所以在小的resize的时候有比较好的效果。Malloc的效率也显著提高了，所幸memorypool还是表现稍微好一点点，费力的优化没有白做。

实验三：extreme scenario

|  |  |  |  |
| --- | --- | --- | --- |
| Allocator类型  测试项目 | Std::allocator | Malloc\_alloc | MemoryPool\_alloc |
| resize, int | 0.034345 | 0.182184 | 0.002022 |

Extreme scenario是专门为MemoryPool 的allocator机制特别打造的测试样例（TestExtreme.cpp）。通过合理地安排resize的顺序和大小，保证MemoryPool中及时回收出大量内存区块，每次需要分配内存的时候都可以直接从free\_list中调出内存而无需Malloc。可以最大程度地体现出内存池专门优化的特点。

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

std::vector<IntVec, MyAllocator<IntVec>> vecints1(TestSize);

for (int i = 0; i < TestSize; i++) vecints1[i].resize(128);

std::vector<IntVec, MyAllocator<IntVec>> vecints2(TestSize);

for (int i = 0; i < TestSize; i++) vecints2[i].resize(32);

for (int i = 0; i < PickSize; i++) {

// int idx = dis(gen) - 1;

int idx=i;

// int size = dis\_small(gen);

vecints1[idx].resize(64);

vecints2[idx].resize(64);

}

具体做法是，创建vector的时候均等划分成两大块，一块大小大些，占128个int，另一块只占32个int。选取这两个数是因为4byte的int放128个能低于memorypool管理内存的小区块上界（在这里设置成1024byte），resize的时候能由memorypool通过free\_list的机制处理。

Resize时先把较大的vector缩小为64个int，保证memorypool的free\_list回收了足够大小和数量的空余区块，再进行把较大的vector放大为64个int的操作。

这时刚刚回收的int区块正好能派上用场，memorypool直接把回收的内存重新分配给resize的vector，完全没有进行malloc和free的重复操作，从而保证效率大大提高。

这个测试案例下由于量身定做，memorypool的表现是压倒性的。Std：：allocator比直接调用malloc的版本运行时间小了1个数量级，而MemoryPool又比std::allocator小了1个数量级。从这里可以大致看出MemoryPool的运行机制。

实验四：类的构造函数包含多于一个参数

之前的allocator中MyClass一直都是使用两个参数的构造函数。

原先的写法只是assignment operator，只用接受一个引用参数，无法测试出construct方法的效果

// vecmyclass[idx1][idx2] = val;

我们改用这个方法进行赋值

/\*\*test whether the allocator supports ctors

\* that has more than 1 argument.

\* if it is replaced by HXZ::allocator,

\* the program cannot pass the compilation.

\*/

vecmyclass[idx1].get\_allocator().construct(&vecmyclass[idx1][idx2],11,15);

带有参数包的模板，可以正常编译并执行。

/\*...Args是模板参数包, Template parameter pack. \*/

template <typename T>

template <typename ObjectType,typename... ArgType>

void Allocator<T>::construct(ObjectType\* p,ArgType&& ...args){

new(p) ObjectType(std::forward<ArgType>(args)...);

return;

}

而简单的使用单一模板的构造函数就无法通过编译了。

template <typename T>

template <typename ObjectType, typename ValueType>

void Allocator<T>::construct(ObjectType\* p,ValueType value ){

new(p) ObjectType(value);

}//compile-time error