## 引言

本书主要针对的是 C++ 程序的性能优化，深入介绍 C++ 程序性能优化的方法和实例。全书由 4 个篇组成，第 1 篇介绍 C++ 语言的对象模型，该篇是优化 C++ 程序的基础；第 2 篇主要针对如何优化 C++ 程序的内存使用；第 3 篇介绍如何优化程序的启动性能；第 4 篇介绍了三类性能优化工具，即内存分析工具、性能分析工具和 I/O 检测工具，它们是测量程序性能的利器。

本章首先简单介绍自定义内存池性能优化的原理，然后列举软件开发中常用的内存池的不同类型，并给出具体实现的实例。

# 6.1 自定义内存池性能优化的原理

如前所述，读者已经了解到"堆"和"栈"的区别。而在编程实践中，不可避免地要大量用到堆上的内存。例如在程序中维护一个链表的数据结构时，每次新增或者删除一个链表的节点，都需要从内存堆上分配或者释放一定的内存；在维护一个动态数组时，如果动态数组的大小不能满足程序需要时，也要在内存堆上分配新的内存空间。

## 6.1.1 默认内存管理函数的不足

利用默认的内存管理函数new/delete或malloc/free在堆上分配和释放内存会有一些额外的开销。

系统在接收到分配一定大小内存的请求时，首先查找内部维护的内存空闲块表，并且需要根据一定的算法（例如分配最先找到的不小于申请大小的内存块给请求者，或者分配最适于申请大小的内存块，或者分配最大空闲的内存块等）找到合适大小的空闲内存块。如果该空闲内存块过大，还需要切割成已分配的部分和较小的空闲块。然后系统更新内存空闲块表，完成一次内存分配。类似地，在释放内存时，系统把释放的内存块重新加入到空闲内存块表中。如果有可能的话，可以把相邻的空闲块合并成较大的空闲块。

默认的内存管理函数还考虑到多线程的应用，需要在每次分配和释放内存时加锁，同样增加了开销。

可见，如果应用程序频繁地在堆上分配和释放内存，则会导致性能的损失。并且会使系统中出现大量的内存碎片，降低内存的利用率。

默认的分配和释放内存算法自然也考虑了性能，然而这些内存管理算法的通用版本为了应付更复杂、更广泛的情况，需要做更多的额外工作。而对于某一个具体的应用程序来说，适合自身特定的内存分配释放模式的自定义内存池则可以获得更好的性能。

### 6.1.2 内存池的定义和分类

自定义内存池的思想通过这个"池"字表露无疑，应用程序可以通过系统的内存分配调用预先一次性申请适当大小的内存作为一个内存池，之后应用程序自己对内存的分配和释放则可以通过这个内存池来完成。只有当内存池大小需要动态扩展时，才需要再调用系统的内存分配函数，其他时间对内存的一切操作都在应用程序的掌控之中。

应用程序自定义的内存池根据不同的适用场景又有不同的类型。

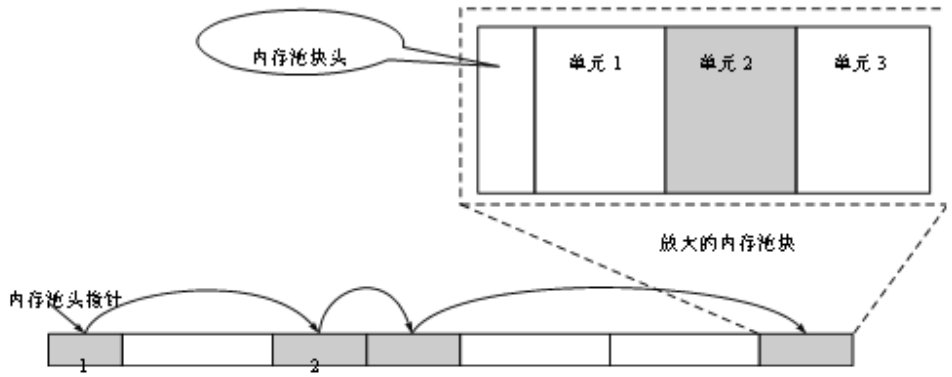
从线程安全的角度来分，内存池可以分为单线程内存池和多线程内存池。单线程内存池整个生命周期只被一个线程使用，因而不需要考虑互斥访问的问题；多线程内存池有可能被多个线程共享，因此则需要在每次分配和释放内存时加锁。相对而言，单线程内存池性能更高，而多线程内存池适用范围更广。

从内存池可分配内存单元大小来分，可以分为固定内存池和可变内存池。所谓固定内存池是指应用程序每次从内存池中分配出来的内存单元大小事先已经确定，是固定不变的；而可变内存池则每次分配的内存单元大小可以按需变化，应用范围更广，而性能比固定内存池要低。

6.1.3 内存池工作原理示例

下面以固定内存池为例说明内存池的工作原理，如图 6-1 所示。

**图 6-1 固定内存池**



固定内存池由一系列固定大小的内存块组成，每一个内存块又包含了固定数量和大小的内存单元。

如图 6-1 所示，该内存池一共包含 4 个内存块。在内存池初次生成时，只向系统申请了一个内存块，返回的指针作为整个内存池的头指针。之后随着应用程序对内存的不断需求，内存池判断需要动态扩大时，才再次向系统申请新的内存块，并把所有这些内存块通过指针链接起来。对于操作系统来说，它已经为该应用程序分配了 4 个等大小的内存块。由于是大小固定的，所以分配的速度比较快；而对于应用程序来说，其内存池开辟了一定大小，内存池内部却还有剩余的空间。

例如放大来看第 4 个内存块，其中包含一部分内存池块头信息和 3 个大小相等的内存池单元。单元 1 和单元 3 是空闲的，单元 2 已经分配。当应用程序需要通过该内存池分配一个单元大小的内存时，只需要简单遍历所有的内存池块头信息，快速定位到还有空闲单元的那个内存池块。然后根据该块的块头信息直接定位到第 1 个空闲的单元地址，把这个地址返回，并且标记下一个空闲单元即可；当应用程序释放某一个内存池单元时，直接在对应的内存池块头信息中标记该内存单元为空闲单元即可。

可见与系统管理内存相比，内存池的操作非常迅速，它在性能优化方面的优点主要如下。

（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。

回页首

## 6.2 一个内存池的实现实例

本节分析在某个大型应用程序实际应用到的一个内存池实现，并详细讲解其使用方法与工作原理。这是一个应用于单线程环境且分配单元大小固定的内存池，一般用来为执行时会动态频繁地创建且可能会被多次创建的类对象或者结构体分配内存。

本节首先讲解该内存池的数据结构声明及图示，接着描述其原理及行为特征。然后逐一讲解实现细节，最后介绍如何在实际程序中应用此内存池，并与使用普通内存函数申请内存的程序性能作比较。

6.2.1 内部构造

内存池类 MemoryPool 的声明如下：

```
class MemoryPool
{
private:
    MemoryBlock*   pBlock;
    USHORT          nUnitSize;
    USHORT          nInitSize;
    USHORT          nGrowSize;

public:
                     MemoryPool( USHORT nUnitSize,
                                 USHORT nInitSize = 1024,
                                 USHORT nGrowSize = 256 );
                    ~MemoryPool();

    void*           Alloc();
    void            Free( void* p );
};
```

MemoryBlock 为内存池中附着在真正用来为内存请求分配内存的内存块头部的结构体，它描述了与之联系的内存块的使用信息：

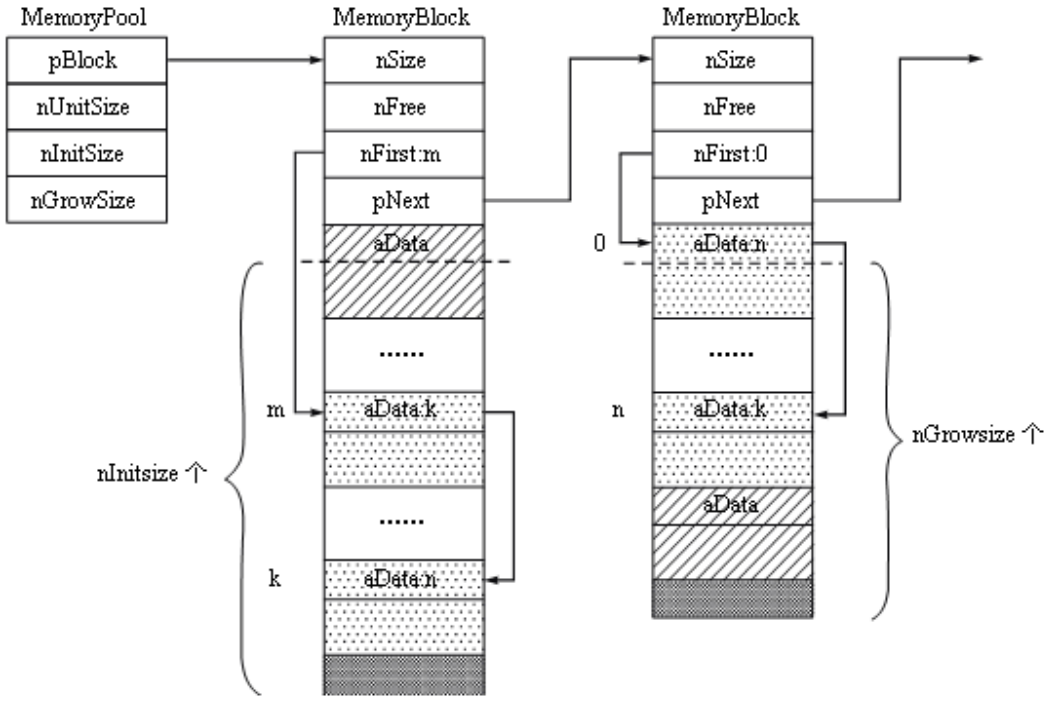
```
struct MemoryBlock
{
    USHORT          nSize;
    USHORT          nFree;
    USHORT          nFirst;
    USHORT          nDummyAlign1;
    MemoryBlock*  pNext;
    char            aData[1];
```

```
        static void* operator new(size_t, USHORT nTypes, USHORT
nUnitSize)
        {
                return ::operator new(sizeof(MemoryBlock) + nTypes *
nUnitSize);
        }
        static void  operator delete(void *p, size_t)
        {
                ::operator delete (p);
        }

        MemoryBlock (USHORT nTypes = 1, USHORT nUnitSize = 0);
        ~MemoryBlock() {}
};
```

此内存池的数据结构如图 6-2 所示。

**图 6-2 内存池的数据结构**




6.2.2 总体机制

此内存池的总体机制如下。

（1）在运行过程中，MemoryPool 内存池可能会有多个用来满足内存申请请求的内存块，这些内存块是从进程堆中开辟的一个较大的连续内存区域，它由一个 MemoryBlock 结构体和多个可供分配的内存单元组成，所有内存块组成了一个内存块链表，MemoryPool 的 pBlock 是这个链表的头。对每个内存块，都可以通过其头部的 MemoryBlock 结构体的 pNext 成员访问紧跟在其后面的那个内存块。

（2）每个内存块由两部分组成，即一个 MemoryBlock 结构体和多个内存分配单元。这些内存分配单元大小固定（由 MemoryPool 的 nUnitSize 表示），MemoryBlock 结构体并不维护那些已经分配的单元的信息；相反，它只维护没有分配的自由分配单元的信息。它有两个成员比较重要：nFree 和 nFirst。nFree 记录这个内存块中还有多少个自由分配单元，而 nFirst 则记录下一个可供分配的单元的编号。每一个自由分配单元的头两个字节（即一个 USHORT 型值）记录了紧跟它之后的下一个自由分配单元的编号，这样，通过利用每个自由分配单元的头两个字节，一个 MemoryBlock 中的所有自由分配单元被链接起来。

（3）当有新的内存请求到来时，MemoryPool 会通过 pBlock 遍历 MemoryBlock 链表，直到找到某个 MemoryBlock 所在的内存块，其中还有自由分配单元（通过检测 MemoryBlock 结构体的 nFree 成员是否大于 0）。如果找到这样的内存块，取得其 MemoryBlock 的 nFirst 值（此为该内存块中第 1 个可供分配的自由单元的编号）。然后根据这个编号定位到该自由分配单元的起始位置（因为所有分配单元大小固定，因此每个分配单元的起始位置都可以通过编号分配单元大小来偏移定位），这个位置就是用来满足此次内存申请请求的内存的起始地址。但在返回这个地址前，需要首先将该位置开始的头两个字节的值（这两个字节值记录其之后的下一个自由分配单元的编号）赋给本内存块的 MemoryBlock 的 nFirst 成员。这样下一次的请求就会用这个编号对应的内存单元来满足，同时将此内存块的 MemoryBlock 的 nFree 递减 1，然后才将刚才定位到的内存单元的起始位置作为此次内存请求的返回地址返回给调用者。

（4）如果从现有的内存块中找不到一个自由的内存分配单元（当第 1 次请求内存，以及现有的所有内存块中的所有内存分配单元都已经被分配时会发生这种情形），MemoryPool 就会从进程堆中申请一个内存块（这个内存块包括一个 MemoryBlock 结构体，及紧邻其后的多个内存分配单元，假设内存分配单元的个数为 n，n 可以取值 MemoryPool 中的 nInitSize 或者 nGrowSize），申请完后，并不会立刻将其中的一个分配单元分配出去，而是需要首先初始化这个内存块。初始化的操作包括设置 MemoryBlock 的 nSize 为所有内存分配单元的大小（注意，并不包括 MemoryBlock 结构体的大小）、nFree 为 n-1（注意，这里是 n-1 而不是 n，因为此次新内存块就是为了满足一次新的内存请求而申请的，马上就会分配一块自由存储单元出去，如果设为 n-1，分配一个自由存储单元后无须再将 n 递减 1），nFirst 为 1（已经知道 nFirst 为下一个可以分配的自由存储单元的编号。为 1 的原因与 nFree 为 n-1 相同，

即立即会将编号为 0 的自由分配单元分配出去。现在设为 1，其后不用修改 nFirst 的值），MemoryBlock 的构造需要做更重要的事情，即将编号为 0 的分配单元之后的所有自由分配单元链接起来。如前所述，每个自由分配单元的头两个字节用来存储下一个自由分配单元的编号。另外，因为每个分配单元大小固定，所以可以通过其编号和单元大小（MemoryPool 的 nUnitSize 成员）的乘积作为偏移值进行定位。现在唯一的问题是定位从哪个地址开始？答案是 MemoryBlock 的 aData[1]成员开始。因为 aData[1]实际上是属于 MemoryBlock 结构体的（MemoryBlock 结构体的最后一个字节），所以实质上，MemoryBlock 结构体的最后一个字节也用做被分配出去的分配单元的一部分。因为整个内存块由 MemoryBlock 结构体和整数个分配单元组成，这意味着内存块的最后一个字节会被浪费，这个字节在图 6-2 中用位于两个内存的最后部分的浓黑背景的小块标识。确定了分配单元的起始位置后，将自由分配单元链接起来的工作就很容易了。即从 aData 位置开始，每隔 nUnitSize 大小取其头两个字节，记录其之后的自由分配单元的编号。因为刚开始所有分配单元都是自由的，所以这个编号就是自身编号加 1，即位置上紧跟其后的单元的编号。初始化后,将此内存块的第 1 个分配单元的起始地址返回,已经知道这个地址就是 aData。

（5）当某个被分配的单元因为 delete 需要回收时，该单元并不会返回给进程堆，而是返回给 MemoryPool。返回时，MemoryPool 能够知道该单元的起始地址。这时，MemoryPool 开始遍历其所维护的内存块链表，判断该单元的起始地址是否落在某个内存块的地址范围内。如果不在所有内存地址范围内，则这个被回收的单元不属于这个 MemoryPool；如果在某个内存块的地址范围内，那么它会将这个刚刚回收的分配单元加到这个内存块的 MemoryBlock 所维护的自由分配单元链表的头部，同时将其 nFree 值递增 1。回收后，考虑到资源的有效利用及后续操作的性能，内存池的操作会继续判断：如果此内存块的所有分配单元都是自由的，那么这个内存块就会从 MemoryPool 中被移出并作为一个整体返回给进程堆;如果该内存块中还有非自由分配单元，这时不能将此内存块返回给进程堆。但是因为刚刚有一个分配单元返回给了这个内存块，即这个内存块有自由分配单元可供下次分配，因此它会被移到 MemoryPool 维护的内存块的头部。这样下次的内存请求到来，MemoryPool 遍历其内存块链表以寻找自由分配单元时，第 1 次寻找就会找到这个内存块。因为这个内存块确实有自由分配单元，这样可以减少 MemoryPool 的遍历次数。

综上所述，每个内存池（MemoryPool）维护一个内存块链表（单链表），每个内存块由一个维护该内存块信息的块头结构（MemoryBlock）和多个分配单元组成，块头结构 MemoryBlock 则进一步维护一个该内存块的所有自由分配单元组成的"链表"。这个链表不是通过"指向下一个自由分配单元的指针"链接起来的,而是通过"下一个自由分配单元的编号"链接起来，这个编号值存储在该自由分配单元的头两个字节中。另外，第 1 个自由分配单元的起始位置并不是 MemoryBlock 结构体"后面的"第 1 个地址位置，而是 MemoryBlock 结构体"内部"的最后一个字节 aData（也可能不是最后一个，因为考虑到字节对齐的问题），即分配单元实际上往前面错了一位。又因为 MemoryBlock 结构体后面的空间刚好是分配单元的整数倍,这样依次错位下去，内存块的最后一个字节实际没有被利用。这么做的一个原因也是考虑到不同平台的

移植问题，因为不同平台的对齐方式可能不尽相同。即当申请 MemoryBlock 大小内存时，可能会返回比其所有成员大小总和还要大一些的内存。最后的几个字节是为了"补齐"，而使得 aData 成为第 1 个分配单元的起始位置，这样在对齐方式不同的各种平台上都可以工作。

6.2.3 细节剖析

有了上述的总体印象后，本节来仔细剖析其实现细节。

（1）MemoryPool 的构造如下：

```
MemoryPool::MemoryPool( USHORT _nUnitSize,
                        USHORT _nInitSize, USHORT _nGrowSize )
{
    pBlock      = NULL;                    ①
    nInitSize   = _nInitSize;       ②
    nGrowSize   = _nGrowSize;       ③

    if ( _nUnitSize > 4 )
        nUnitSize = (_nUnitSize + (MEMPOOL_ALIGNMENT-1)) &
~(MEMPOOL_ALIGNMENT-1); ④
    else if ( _nUnitSize <= 2 )
        nUnitSize = 2;              ⑤
    else
        nUnitSize = 4;
}
```

从①处可以看出，MemoryPool 创建时，并没有立刻创建真正用来满足内存申请的内存块，即内存块链表刚开始时为空。

②处和③处分别设置"第 1 次创建的内存块所包含的分配单元的个数"，及"随后创建的内存块所包含的分配单元的个数"，这两个值在 MemoryPool 创建时通过参数指定，其后在该 MemoryPool 对象生命周期中一直不变。

后面的代码用来设置 nUnitSize，这个值参考传入的_nUnitSize 参数。但是还需要考虑两个因素。如前所述，每个分配单元在自由状态时，其头两个字节用来存放"其下一个自由分配单元的编号"。即每个分配单元"最少"有"两个字节"，这就是⑤处赋值的原因。④处是将大于 4 个字节的大小_nUnitSize 往上"取整到"大于_nUnitSize 的最小的 MEMPOOL_ ALIGNMENT 的倍数（前提是 MEMPOOL_ALIGNMENT 为 2

的倍数）。如_nUnitSize 为 11 时，MEMPOOL_ALIGNMENT 为 8，nUnitSize 为 16；MEMPOOL_ALIGNMENT 为 4，nUnitSize 为 12；MEMPOOL_ALIGNMENT 为 2，nUnitSize 为 12，依次类推。

（2）当向 MemoryPool 提出内存请求时：

```
void* MemoryPool::Alloc()
{
    if ( !pBlock )           ①
    {
                      ……
    }

    MemoryBlock* pMyBlock = pBlock;
    while (pMyBlock && !pMyBlock->nFree )②
        pMyBlock = pMyBlock->pNext;

    if ( pMyBlock )             ③
    {
        char* pFree = pMyBlock->aData+(pMyBlock->nFirst*nUnitSize);
        pMyBlock->nFirst = *((USHORT*)pFree);

        pMyBlock->nFree--;
        return (void*)pFree;
    }
    else                    ④
    {
        if ( !nGrowSize )
            return NULL;

                pMyBlock = new(nGrowSize, nUnitSize)
FixedMemBlock(nGrowSize, nUnitSize);
        if ( !pMyBlock )
            return NULL;

        pMyBlock->pNext = pBlock;
        pBlock = pMyBlock;

        return (void*)(pMyBlock->aData);
```

```
    }
}
```

MemoryPool 满足内存请求的步骤主要由四步组成。

①处首先判断内存池当前内存块链表是否为空，如果为空，则意味着这是第 1 次内存申请请求。这时，从进程堆中申请一个分配单元个数为 nInitSize 的内存块，并初始化该内存块（主要初始化 MemoryBlock 结构体成员，以及创建初始的自由分配单元链表，下面会详细分析其代码）。如果该内存块申请成功，并初始化完毕，返回第 1 个分配单元给调用函数。第 1 个分配单元以 MemoryBlock 结构体内的最后一个字节为起始地址。

②处的作用是当内存池中已有内存块（即内存块链表不为空）时遍历该内存块链表，寻找还有"自由分配单元"的内存块。

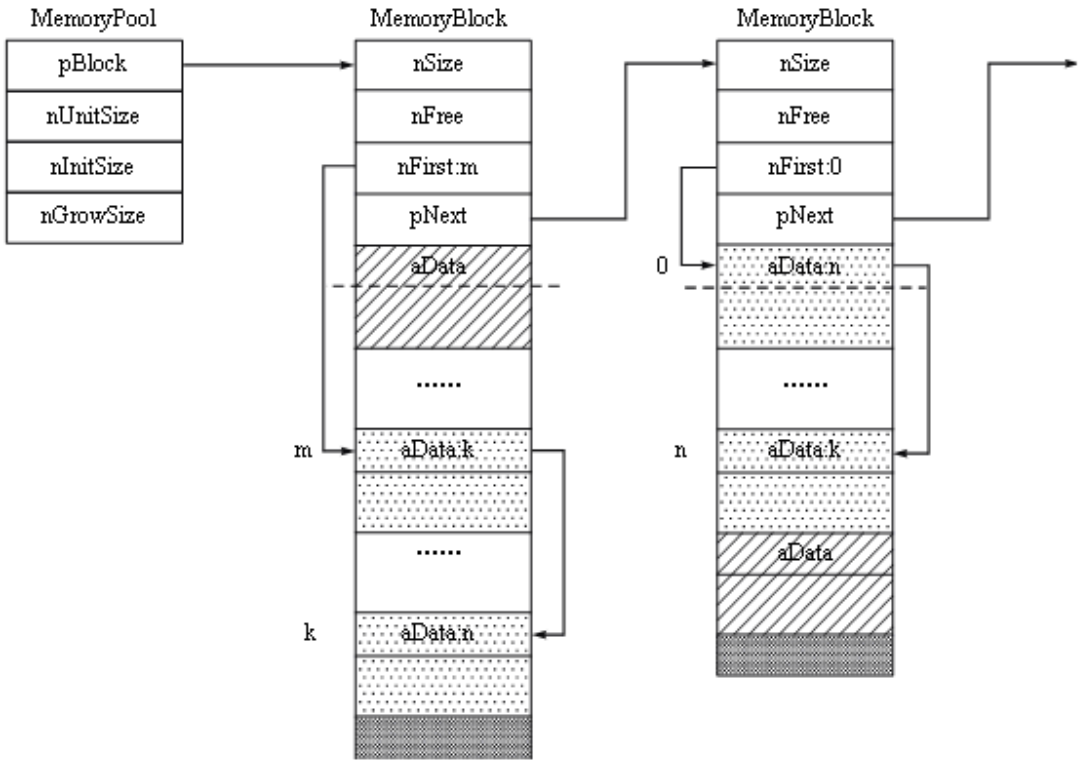
③处检查如果找到还有自由分配单元的内存块，则"定位"到该内存块现在可以用的自由分配单元处。"定位"以 MemoryBlock 结构体内的最后一个字节位置 aData 为起始位置，以 MemoryPool 的 nUnitSize 为步长来进行。找到后，需要修改 MemoryBlock 的 nFree 信息（剩下来的自由分配单元比原来减少了一个），以及修改此内存块的自由存储单元链表的信息。在找到的内存块中，pMyBlock->nFirst 为该内存块中自由存储单元链表的表头，其下一个自由存储单元的编号存放在 pMyBlock->nFirst 指示的自由存储单元（亦即刚才定位到的自由存储单元）的头两个字节。通过刚才定位到的位置，取其头两个字节的值，赋给 pMyBlock->nFirst，这就是此内存块的自由存储单元链表的新的表头，即下一次分配出去的自由分配单元的编号（如果 nFree 大于零的话）。修改维护信息后，就可以将刚才定位到的自由分配单元的地址返回给此次申请的调用函数。注意，因为这个分配单元已经被分配，而内存块无须维护已分配的分配单元，因此该分配单元的头两个字节的信息已经没有用处。换个角度看，这个自由分配单元返回给调用函数后，调用函数如何处置这块内存，内存池无从知晓，也无须知晓。此分配单元在返回给调用函数时，其内容对于调用函数来说是无意义的。因此几乎可以肯定调用函数在用这个单元的内存时会覆盖其原来的内容，即头两个字节的内容也会被抹去。因此每个存储单元并没有因为需要链接而引入多余的维护信息，而是直接利用单元内的头两个字节，当其分配后，头两个字节也可以被调用函数利用。而在自由状态时，则用来存放维护信息，即下一个自由分配单元的编号，这是一个有效利用内存的好例子。

④处表示在②处遍历时，没有找到还有自由分配单元的内存块，这时，需要重新向进程堆申请一个内存块。因为不是第一次申请内存块，所以申请的内存块包含的分配单元个数为 nGrowSize，而不再是 nInitSize。与①处相同，先做这个新申请内存块的初始化工作，然后将此内存块插入 MemoryPool 的内存块链表的头部，再将此内存块的第 1 个分配单元返回给调用函数。将此新内存块插入内存块链表的头部的原

因是该内存块还有很多可供分配的自由分配单元（除非 nGrowSize 等于 1，这应该不太可能。因为内存池的含义就是一次性地从进程堆中申请一大块内存，以供后续的多次申请），放在头部可以使得在下次收到内存申请时，减少②处对内存块的遍历时间。

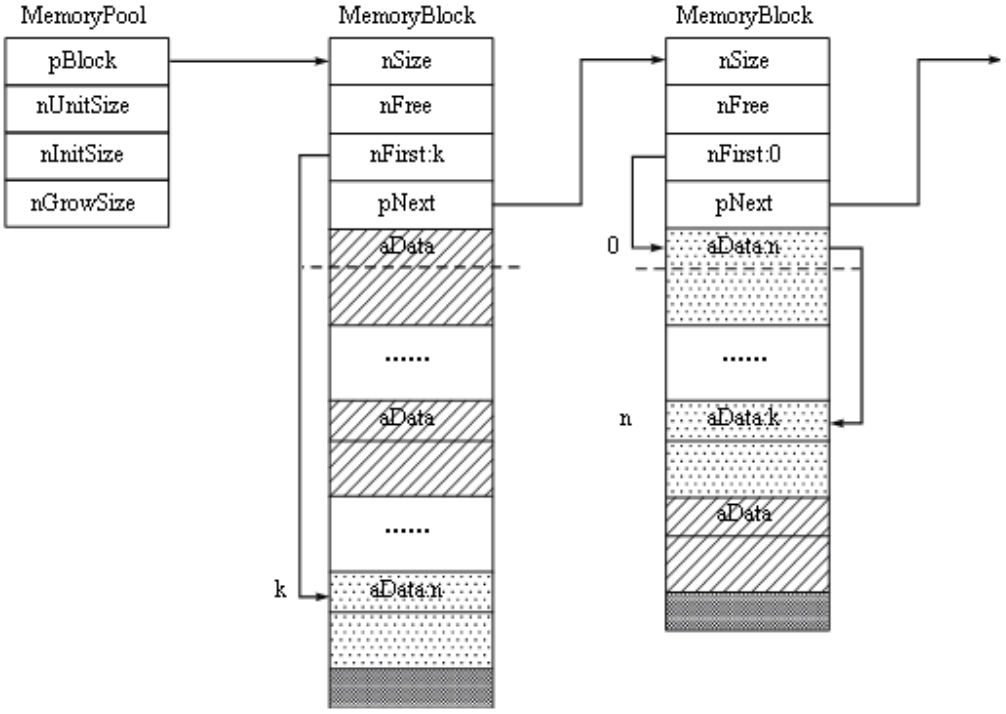
可以用图 6-2 的 MemoryPool 来展示 MemoryPool::Alloc 的过程。图 6-3 是某个时刻 MemoryPool 的内部状态。

**图 6-3 某个时刻 MemoryPool 的内部状态**




因为 MemoryPool 的内存块链表不为空，因此会遍历其内存块链表。又因为第 1 个内存块里有自由的分配单元，所以会从第 1 个内存块中分配。检查 nFirst，其值为 m，这时 pBlock->aData+(pBlock->nFirst*nUnitSize)定位到编号为 m 的自由分配单元的起始位置（用 pFree 表示）。在返回 pFree 之前，需要修改此内存块的维护信息。首先将 nFree 递减 1，然后取得 pFree 处开始的头两个字节的值（需要说明的是，这里 aData 处值为 k。其实不是这一个字节。而是以 aData 和紧跟其后的另外一个字节合在一起构成的一个 USHORT 的值，不可误会）。发现为 k，这时修改 pBlock 的 nFirst 为 k。然后，返回 pFree。此时 MemoryPool 的结构如图 6-4 所示。

图 6-4 MemoryPool 的结构




可以看到，原来的第 1 个可供分配的单元（m 编号处）已经显示为被分配的状态。而 pBlock 的 nFirst 已经指向原来 m 单元下一个自由分配单元的编号，即 k。

（3）MemoryPool 回收内存时：

```
void MemoryPool::Free( void* pFree )
{
    ……

    MemoryBlock* pMyBlock = pBlock;

    while ( ((ULONG)pMyBlock->aData > (ULONG)pFree) ||
         ((ULONG)pFree >= ((ULONG)pMyBlock->aData +
pMyBlock->nSize)) )①
    {
         ……
    }
```

```
    pMyBlock->nFree++;                          ②
    *((USHORT*)pFree) = pMyBlock->nFirst;  ③
    pMyBlock->nFirst = (USHORT)(((ULONG)pFree-(ULONG)(pBlock->aData))
/ nUnitSize);④

    if (pMyBlock->nFree*nUnitSize == pMyBlock->nSize )⑤
    {
        ……
    }
    else
    {
        ……
    }
}
```

如前所述，回收分配单元时，可能会将整个内存块返回给进程堆，也可能将被回收分配单元所属的内存块移至内存池的内存块链表的头部。这两个操作都需要修改链表结构。这时需要知道该内存块在链表中前一个位置的内存块。

①处遍历内存池的内存块链表，确定该待回收分配单元（pFree）落在哪一个内存块的指针范围内，通过比较指针值来确定。

运行到②处，pMyBlock 即找到的包含 pFree 所指向的待回收分配单元的内存块（当然，这时应该还需要检查 pMyBlock 为 NULL 时的情形，即 pFree 不属于此内存池的范围，因此不能返回给此内存池，读者可以自行加上）。这时将 pMyBlock 的 nFree 递增 1，表示此内存块的自由分配单元多了一个。

③处用来修改该内存块的自由分配单元链表的信息，它将这个待回收分配单元的头两个字节的值指向该内存块原来的第一个可分配的自由分配单元的编号。

④处将 pMyBlock 的 nFirst 值改变为指向这个待回收分配单元的编号，其编号通过计算此单元的起始位置相对 pMyBlock 的 aData 位置的差值，然后除以步长（nUnitSize）得到。

实质上，③和④两步的作用就是将此待回收分配单元"真正回收"。值得注意的是，这两步实际上是使得此回收单元成为此内存块的下一个可分配的自由分配单元，即将它放在了自由分配单元链表的头部。注意，其内存地址并没有发生改变。实际上，一个分配单元的内存地址无论是在分配后，还是处于自由状态时，一直都不会变化。变化的只是其状态（已分配/自由），以及当其处于自由状态时在自由分配单元链表中的位置。

⑤处检查当回收完毕后,包含此回收单元的内存块的所有单元是否都处于自由状态，且此内存是否处于内存块链表的头部。如果是，将此内存块整个的返回给进程堆，同时修改内存块链表结构。

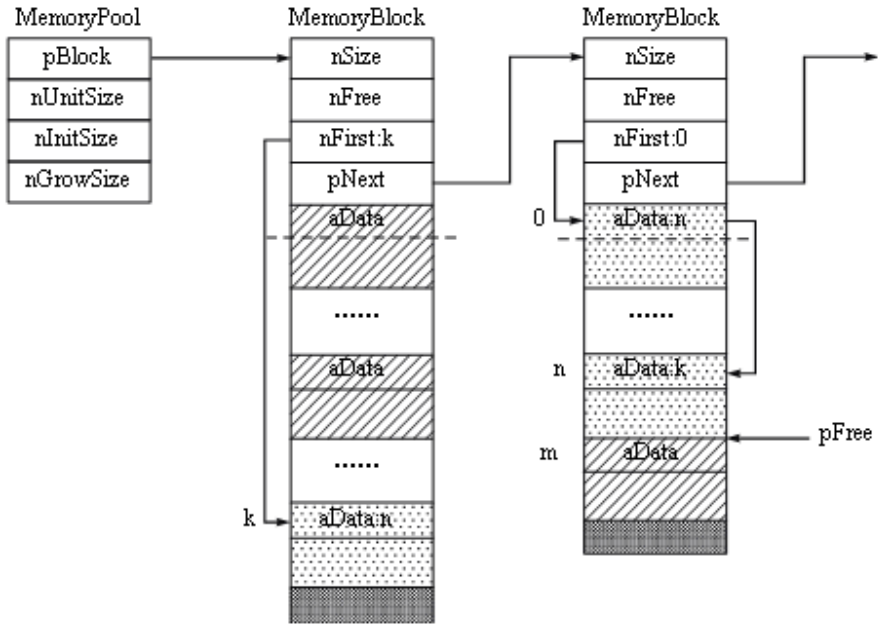
注意，这里在判断一个内存块的所有单元是否都处于自由状态时，并没有遍历其所有单元，而是判断 nFree 乘以 nUnitSize 是否等于 nSize。nSize 是内存块中所有分配单元的大小，而不包括头部 MemoryBlock 结构体的大小。这里可以看到其用意，即用来快速检查某个内存块中所有分配单元是否全部处于自由状态。因为只需结合 nFree 和 nUnitSize 来计算得出结论，而无须遍历和计算所有自由状态的分配单元的个数。

另外还需注意的是，这里并不能比较 nFree 与 nInitSize 或 nGrowSize 的大小来判断某个内存块中所有分配单元都为自由状态，这是因为第 1 次分配的内存块（分配单元个数为 nInitSize）可能被移到链表的后面，甚至可能在移到链表后面后，因为某个时间其所有单元都处于自由状态而被整个返回给进程堆。即在回收分配单元时，无法判定某个内存块中的分配单元个数到底是 nInitSize 还是 nGrowSize，也就无法通过比较 nFree 与 nInitSize 或 nGrowSize 的大小来判断一个内存块的所有分配单元是否都为自由状态。

以上面分配后的内存池状态作为例子，假设这时第 2 个内存块中的最后一个单元需要回收（已被分配，假设其编号为 m，pFree 指针指向它），如图 6-5 所示。

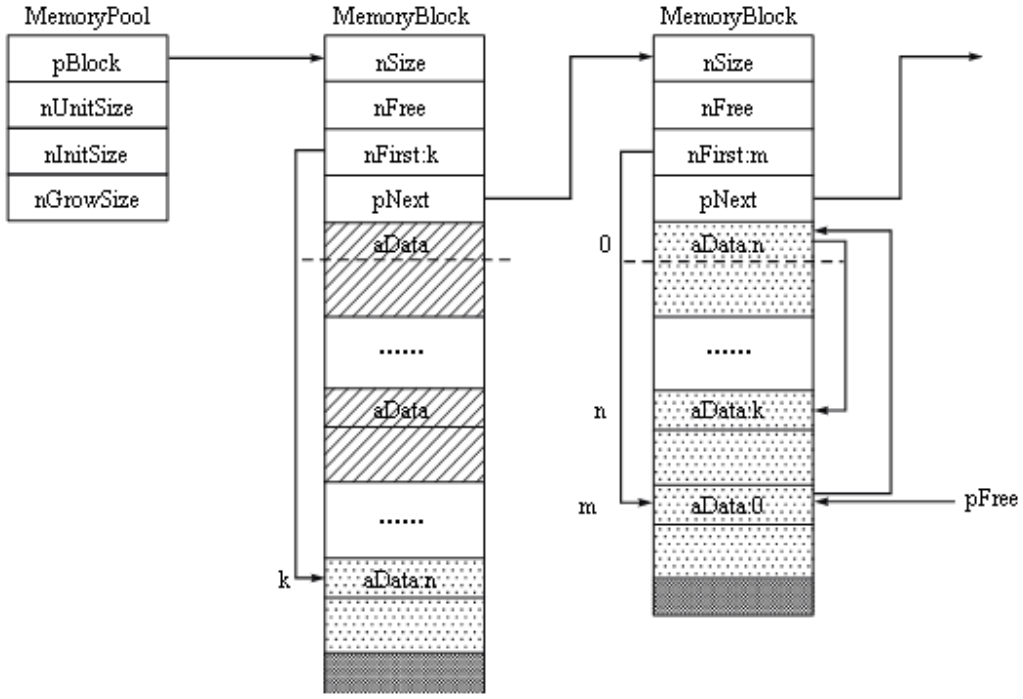
不难发现，这时 nFirst 的值由原来的 0 变为 m。即此内存块下一个被分配的单元是 m 编号的单元，而不是 0 编号的单元（最先分配的是最新回收的单元，从这一点看，这个过程与栈的原理类似，即先进后出。只不过这里的"进"意味着"回收"，而"出"则意味着"分配"）。相应地，m 的"下一个自由单元"标记为 0，即内存块原来的"下一个将被分配出去的单元"，这也表明最近回收的分配单元被插到了内存块的"自由分配单元链表"的头部。当然，nFree 递增 1。

图 6-5 分配后的内存池状态




处理至⑥处之前，其状态如图 6-6 所示。

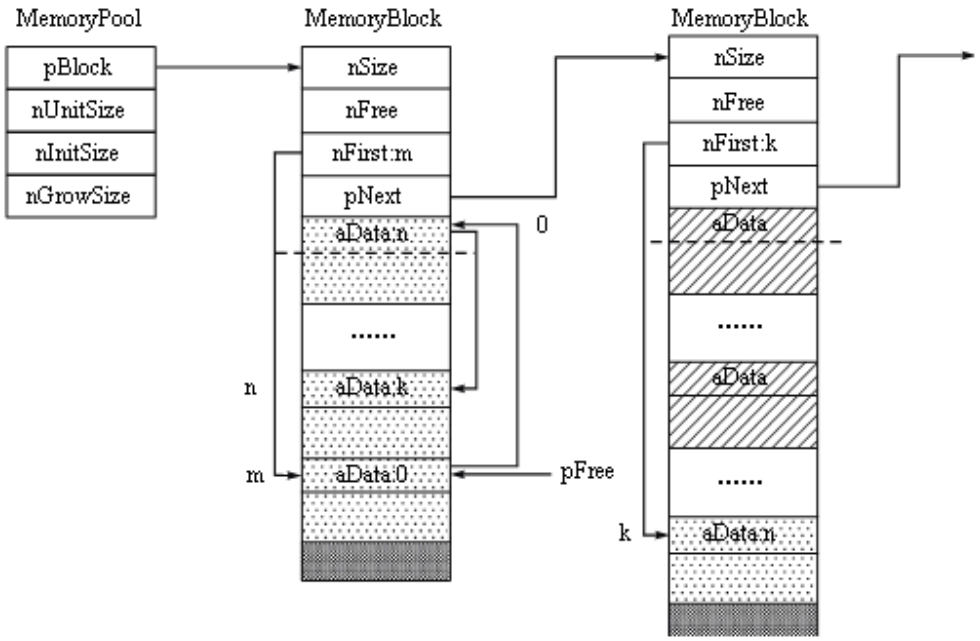
图 6-6 处理至⑥处之前的内存池状态



这里需要注意的是，虽然 pFree 被"回收"，但是 pFree 仍然指向 m 编号的单元，这个单元在回收过程中，其头两个字节被覆写，但其他部分的内容并没有改变。而且从整个进程的内存使用角度来看，这个 m 编号的单元的状态仍然是"有效的"。因为这里的"回收"只是回收给了内存池，而并没有回收给进程堆，因此程序仍然可以通过 pFree 访问此单元。但是这是一个很危险的操作，因为首先该单元在回收过程中头两个字节已被覆写，并且该单元可能很快就会被内存池重新分配。因此回收后通过 pFree 指针对这个单元的访问都是错误的，读操作会读到错误的数据，写操作则可能会破坏程序中其他地方的数据，因此需要格外小心。

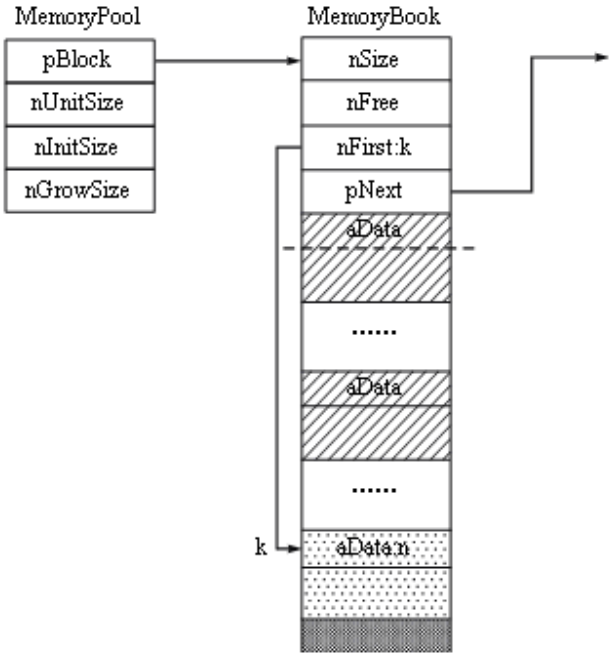
接着，需要判断该内存块的内部使用情况，及其在内存块链表中的位置。如果该内存块中省略号"……"所表示的其他部分中还有被分配的单元，即 nFree 乘以 nUnitSize 不等于 nSize。因为此内存块不在链表头，因此还需要将其移到链表头部，如图 6-7 所示。

**图 6-7 因回收引起的 MemoryBlock 移动**




如果该内存块中省略号"……"表示的其他部分中全部都是自由分配单元，即 nFree 乘以 nUnitSize 等于 nSize。因为此内存块不在链表头，所以此时需要将此内存块整个回收给进程堆，回收后内存池的结构如图 6-8 所示。

图 6-8 回收后内存池的结构




一个内存块在申请后会初始化，主要是为了建立最初的自由分配单元链表，下面是其详细代码：

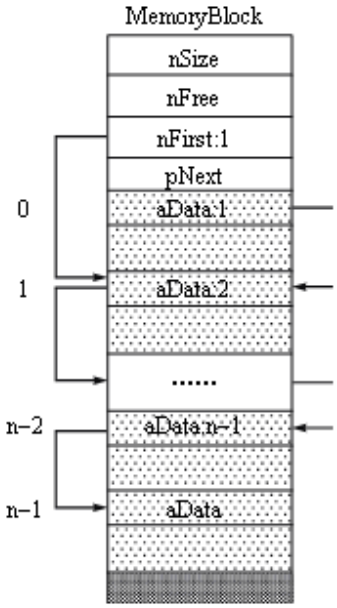
```
MemoryBlock::MemoryBlock (USHORT nTypes, USHORT nUnitSize)
        : nSize  (nTypes * nUnitSize),
          nFree  (nTypes - 1),                          ④
          nFirst (1),                                   ⑤
          pNext  (0)
{
                char * pData = aData;                       ①
                for (USHORT i = 1; i < nTypes; i++) ②
                {
                        *reinterpret_cast<USHORT*>(pData) = i; ③
                        pData += nUnitSize;
                }
}
```

这里可以看到，①处 pData 的初值是 aData，即 0 编号单元。但是②处的循环中 i 却是从 1 开始，然后在循环内部的③处将 pData 的头两个字节值置为 i。即 0 号单元的头两个字节值为 1，1 号单元的头两个字节值为 2，一直到（nTypes-2）号单元的头两个字节值为（nTypes-1）。这意味着内存块初始时，其自由分配单元链表是从 0 号开始。依次串联，一直到倒数第 2 个单元指向最后一个单元。

还需要注意的是，在其初始化列表中，nFree 初始化为 nTypes-1（而不是 nTypes），nFirst 初始化为 1（而不是 0）。这是因为第 1 个单元，即 0 编号单元构造完毕后，立刻会被分配。另外注意到最后一个单元初始并没有设置头两个字节的值，因为该单元初始在本内存块中并没有下一个自由分配单元。但是从上面例子中可以看到，当最后一个单元被分配并回收后，其头两个字节会被设置。

图 6-9 所示为一个内存块初始化后的状态。

**图 6-9 一个内存块初始化后的状态**




当内存池析构时，需要将内存池的所有内存块返回给进程堆：

```
MemoryPool::~MemoryPool()
{
    MemoryBlock* pMyBlock = pBlock;
    while ( pMyBlock )
    {
```

```
        ……
    }
}
```

### 6.2.4 使用方法

分析内存池的内部原理后，本节说明如何使用它。从上面的分析可以看到，该内存池主要有两个对外接口函数，即 Alloc 和 Free。Alloc 返回所申请的分配单元（固定大小内存），Free 则回收传入的指针代表的分配单元的内存给内存池。分配的信息则通过 MemoryPool 的构造函数指定，包括分配单元大小、内存池第 1 次申请的内存块中所含分配单元的个数，以及内存池后续申请的内存块所含分配单元的个数等。

综上所述，当需要提高某些关键类对象的申请 / 回收效率时，可以考虑将该类所有生成对象所需的空间都从某个这样的内存池中开辟。在销毁对象时，只需要返回给该内存池。"一个类的所有对象都分配在同一个内存池对象中"这一需求很自然的设计方法就是为这样的类声明一个静态内存池对象，同时为了让其所有对象都从这个内存池中开辟内存，而不是缺省的从进程堆中获得，需要为该类重载一个 new 运算符。因为相应地，回收也是面向内存池，而不是进程的缺省堆，还需要重载一个 delete 运算符。在 new 运算符中用内存池的 Alloc 函数满足所有该类对象的内存请求，而销毁某对象则可以通过在 delete 运算符中调用内存池的 Free 完成。

### 6.2.5 性能比较

为了测试利用内存池后的效果，通过一个很小的测试程序可以发现采用内存池机制后耗时为 297 ms。而没有采用内存池机制则耗时 625 ms，速度提高了 52.48%。速度提高的原因可以归结为几点，其一，除了偶尔的内存申请和销毁会导致从进程堆中分配和销毁内存块外，绝大多数的内存申请和销毁都由内存池在已经申请到的内存块中进行，而没有直接与进程堆打交道，而直接与进程堆打交道是很耗时的操作；其二，这是单线程环境的内存池，可以看到内存池的 Alloc 和 Free 操作中并没有加线程保护措施。因此如果类 A 用到该内存池，则所有类 A 对象的创建和销毁都必须发生在同一个线程中。但如果类 A 用到内存池，类 B 也用到内存池，那么类 A 的使用线程可以不必与类 B 的使用线程是同一个线程。

另外，在第 1 章中已经讨论过，因为内存池技术使得同类型的对象分布在相邻的内存区域，而程序会经常对同一类型的对象进行遍历操作。因此在程序运行过程中发生的缺页应该会相应少一些，但这个一般只能在真实的复杂应用环境中进行验证。

---

回页首

## 6.3 本章小结

内存的申请和释放对一个应用程序的整体性能影响极大，甚至在很多时候成为某个应用程序的瓶颈。消除内存申请和释放引起的瓶颈的方法往往是针对内存使用的实际情况提供一个合适的内存池。内存池之所以能够提高性能，主要是因为它能够利用应用程序的实际内存使用场景中的某些"特性"。比如某些内存申请与释放肯定发生在一个线程中，某种类型的对象生成和销毁与应用程序中的其他类型对象要频繁得多，等等。针对这些特性，可以为这些特殊的内存使用场景提供量身定做的内存池。这样能够消除系统提供的缺省内存机制中，对于该实际应用场景中的不必要的操作，从而提升应用程序的整体性能。