文章编号:1004 -5422(2017)03 -0257 -05

基于 C ++ 的高效内存池的设计与实现

鄢 涛[1,2], 于 曦[1,2], 刘永红[1,2], 赵卫东[1,2], 余 悦[1], 曾 谊[1]

(1. 成都大学 信息科学与工程学院, 四川 成都 610106;
2. 成都大学 模式识别与智能信息处理四川省高校重点实验室, 四川 成都 610106)

摘 要: 为了高效、安全地利用计算机内存资源,在大型的软件设计中,往往要进行大量的内存分配与回收操作,为此,C ++ 专门提供了 malloc 等相关函数进行操作,这些函数能够满足一般的使用,但由于它们调用了操作系统 API,所以实际使用时会在操作系统中产生大量的内存碎片,让内存分配成为效率瓶颈,从而降低系统性能. 基于此,通过对循环首次适应算法进行改进,设计并实现了基于 C ++ 的高效内存池,大幅提升了内存分配与回收的效率. 同时,还为内存池编写了相关的分配子,使其能与 C ++ 标准库无缝对接,提供了若干具有垃圾回收功能的智能指针,提高了内存管理与程序运行的效率.

关键词: 内存池; 内存分配; 循环首次适应算法; 高效策略

中图分类号: TP311.11 **文献标志码**: A

0 引 言

大型的计算机软件应用中,往往需要进行大量的内存分配与回收工作,这些工作可以通过各种内存分配函数来完成,比如 C 语言的 malloc. 但由于操作系统在分配内存时,需要进行大量的查找工作,所以使用如 malloc 来分配内存实际上是比较低效的. 不仅如此,由于分配算法本身的问题,多次的内存分配/回收还会产生大量系统级别的内存碎片,进而降低内存的利用率. 针对这些问题,内存池将会是一个很好的解决方案,内存池实际上是一种缓冲的机制,它一次性申请大量内存,以后每当有内存分配请求时,不再调用 malloc 这样的函数,而是在内存池中寻找一片合适的区域来使用.

目前,基于 C ++ 的内存池设计与实现有不少方案,其中最为优秀的当属准标准库 Boost 提供的 boost::pool 和 boost::object_pool,前者用于储存各种基本数据,后者用来储存各种类的实例对象. Boost 是一个重量级的库,安装和配置非常繁琐,虽然 boost::pool 在分配指定大小的内存时很高效,但是在分配任意大小的内存时,却显得比较笨拙. 在某些应用场景下,boost 中的内存池不能完美满足开发的需求. 基于此,本研究设计并实现了一款高性能且轻量级的内存池. 比起已有的研究成果,本研究的最终方案不仅拥有让使用者满意的分配效率,而且还能提供很多方便开发者利用的内存池辅助工具.

1 数据结构设计

1.1 内存池对象设计

内存池对象的设计要解决的首要问题就是内存池类的维护和管理,具体为: 首先需要提供至少 2 个接口: allocate 和 deallocate,分别负责内存分配与回收工作; 然后是内存资源的维护,大多数内存池的实现都会采用链表,有所不同的是本研究所维护的链表的基本结点并不是内存,而是“内存池结点对象",这些结点又各自维护着自己的内存块. 采用这种模式,能最大限度地减小内存池各个组件之间的耦合性.

由于内存池的创建和销毁都需要消耗大量的资源,所以不妨在这个设计中利用单例模式来防止用户创建过多的内存池对象. 单例模式是一种最简单的设计模式,只需要把构造/析构函数设置为私有,然后提供一个静态函数作为接口即可. 此外,还需要若干个成员变量,分别表示链表头、链表尾、当前结点等. 内存池类的签名如下,

```
class memPool final
{
public:
    void * allocate(size_t);
```

收稿日期: 2017 -08 -12.
基金项目: 四川省科技厅软件科学研究计划(2017ZR0198)、四川省科技厅应用基础计划(2016JY0255)资助项目.
作者简介: 鄢 涛(1973 —),男,硕士,副教授,从事计算机软件工程研究.

```
    bool deallocate( void * ) ;
    static memPool& memPool( ) ;
private:
    memPool( ) ;
    ~ memPool( ) ;
    memNode * currentNode;
    memNode * lastNode;
    memNode * firstNode;
} ;
```

这个类禁止被继承,所以需使用 final 修饰,同时,为了保证最多只有 1 个实例,还需要禁用拷贝构造函数和赋值运算符.

这样的设计非常简洁,在不影响内存池功能的情况下,最大程度地简化了对外接口,用户能很轻松使用这个内存池.

1.2 内存池结点设计

内存池维护一个“内存池结点”的链表,每到分配内存时,便通过某种算法,找到 1 个合适的结点,然后让结点来执行分配操作,所以结点类依然要有 allocate 和 deallocate 2 个接口. 内存池结点的签名为,

```
struct memPool: : memNode
{
explicit memNode( size _ t) ;
    ~ memNode( ) ;
    void*  allocate( size _ t s) ;
    void deallocate( void * ) ;
    memNode * next;
    void * block;
} ;
```

内存池结点类是内存池类的内部结构体,不能被外部所实例化.

这样设计内存池结点的好处在于,把实际的内存分配工作下发给各个结点,而不由内存池亲自完成,降低了内存池和结点之间的耦合度,同时提高了代码的可读性.

2 分配算法的实现

2.1 结点的分配与回收算法

2.1.1 结点的分配与回收.

内存池结点并不直接与外界打交道,只负责处理来自内存池对象的分配/回收请求,即: 当 1 个结点被请求分配内存时,它被内存池认为是当前最合适的选择,接着它会尝试从自己维护的内存区段中寻找一片合适的内存分区并将其分割,然后返回结果,如果不能找到合适的,则返回 NULL.

结点所维护的内存区段被分为若干个不同的块,每个块由块头、块尾及中间区域 3 部分组成. 块头和块尾标记这块内存是否可以被分配,以及这块内存的大小,而中间的区域则是可以被外界所使用的.

2.1.2 结点的分配算法.

内存分配算法中,目前比较通用的是循环首次适应算法,该算法在分配内存空间时,从上次找到空闲区的下 1 个空闲分区开始查找,直到找到第 1 个能满足要求的空闲区为止,并从中划出一块与请求大小相等的内存空间分配给作业. 该算法能使内存中的空闲区分布得较均匀,但这样会缺少大的空闲分区.

本研究内存分配算法是对循环首次适应算法的改进,主要改进的地方为把内存分区的连接方式从链表改为了线性表. 比起传统的算法,虽然改进后的算法失去了一些优点,但以下的优点却让它更适合内存池: 回收内存时只需修改标记即可,不需要查找任何表,只消耗常量时间; 回收内存后可以很容易地合并空闲分区,这在一定程度上克服了原算法缺少较大的空闲分区的问题; 比起链表,线性表需要更少的布局成本,能让更多的内存被用户使用,而不是用作系统保留.

本算法的思路为: 从第 1 个结点开始,依次检索结点中的空闲分区,直到找到 1 个合适大小的分区为止; 若在某个结点中,无法找到合适的结点,则尝试下 1 个结点,若所有结点都已经试过,则增加 1 个新的结点,或者直接返回错误信息; 若能找到合适的区域,则检查该区域是否“足够大”. 本分配算法的代码为,

```
if ( * blockSize - need - 2 * infoSize < leastBlockSize)
{
    * i = false; /* i 指向“块头"* /
    * (i + * blockSize - infoSize) = false; /* 块尾* /
    this - > available - = * blockSize - 2 * infoSize; /* 可
                                                用部分减少* /
    return i + infoSize; /* infoSize 是“块头”相对可用区域的偏
                                                移* /
}
else
{
    size _ t oldSize = * temp;
    * i = false;
    * (i + 1) = need + 2 * infoSize; /* 由于有分割,所以
                                                要重设大小* /
j + = need + infoSize;
* j = false;
    * (j + 1) = need + 2 * infoSize; /* 设置块尾* /
```

```
    j + = infoSize;
    * j = true; /* 分割出的新块的块头* /
    * (j + 1) = oldSize - (need + 2 * infoSize);
    * (i + oldSize - infoSize) = true; /* 分割出的新块的块
                                          尾* /
    * (i + oldSize - infoSize + 1) = oldSize - (need + 2 *
                                          infoSize);
    available - = need + 2 * infoSize;
    return i + infoSize;
}
```

2.1.3　结点的回收算法.

比起分配算法,结点的回收算法则要简单得多,只需要把块头块尾的标记从 false 改为 true 即可.

需要注意的是,在分配算法中,可能会导致一个内存区段中有越来越多的内存块,太多内存块显然是不利于分配的,所以在内存回收时应该尝试合并这些内存块. 在回收的时候,首先检查块头的标记是否为 false,如果不是,说明这块内存已经遭到破坏,属于严重错误,此时应该调用 abort 函数立即终止程序.

合并内存块分为 2 步: 向前检索和向后检索,即分别检查前面与后面紧邻的内存块是否空闲,如果是则将它们合并,其代码为,

```
size_t prevSize = 0, nextSize = 0;
if (flag ! = start&&* (flag - infoSize)) /* 如果前方有内存块
                                          并且可用* /
    prevSize = * (flag - sizeof(size_t));
if ((flag + tempSize) ! = end&&* (flag + tempSize))
    nextSize = * (flag + tempSize + sizeof(bool));
    * (flag - prevSize) = true;
* (flag - prevSize + sizeof(bool)) = blockSize + prevSize +
                                    nextSize; /* 合并后的大小* /
* (flag + tempSize + nextSize - infoSize) = true;
* (flag + nextSize + tempSize - sizeof(size_t)) = tempSize +
                                    prevSize + nextSize;
```

2.2　内存池的分配与回收算法

2.2.1　内存池的分配算法.

内存池需要完成的工作是寻找一个合适的结点,并调用结点的分配算法. 如果找不到合适的结点还需要额外申请空间以满足需求; 如果请求的分配大小大于单个结点所能提供的最大值,则直接返回 NULL 以表示分配失败.

和结点类似,内存池对象也采用循环首次适应算法,它保存着下一次分配内存的最佳起始位置,然后依次搜索,直到分配成功或者决定新增一片内存为止,代码为,

```
memNode * temp = currentNode;
for (;;)
{
    temp = getNext(temp);
    if (temp - > available > = need + 2 * (currentNode - >
                                          infoSize))
    {
        currentNode = getNext(currentNode);
        result = temp - > allocate(need);
        if (result ! = nullptr)
            break;
    }
    if (temp = = currentNode)
    {
        lastNode - > next = new memNode(nodeSize);
        lastNode = lastNode - > next;
        result = lastNode - > allocate(need);
        currentNode = lastNode;
        currentNodeNum + +;
        break;
    }
}
```

其中,getNext 函数的作用是寻找下一个结点.

2.2.2　内存池的回收算法.

而内存池的回收算法更加简单,只需要遍历链表,判断需要回收的指针属于哪个结点即可,由于链表通常不会有太多结点,因此回收算法甚至可以认为是一个 O(1) 的操作,代码为,

```
memNode * temp = firstNode;
if (ptr = = NULL) /* 处理 NULL 的情况* /
    return;
for (; temp! = NULL; temp = temp - > next)
    if(ptr > = temp - > block&&ptr < temp - > block + temp - >
                                          size)
        temp - > deallocate(ptr);
```

3　分配子的实现

C ++ 标准库中,最重要的一部分是 STL,STL 提供了大量的高性能的容器. 这些容器通常拥有自动管理内存的功能. 默认情况下,它们使用 new 和 delete 来分配内存. 然而,STL 是一款设计精良的产品,组件之间耦合度极低,甚至低到可以由用户单独定制内存分配的方式,而用户指定内存分配规则的关键,就在于分配子.

实际上,C ++ 标准明确规定了一个合格的分配子应该有哪些成员,编写一个分配子也并不复杂,而需要关心的只有负责内存分配/释放的 allocate 和 deallocate. allocate 的全部代码为,

```
pointer allocate(size_type n, const_pointer = 0)
{
    void* p = pool.allocate(n* sizeof(value_type)); /* 从内
```

```
                                        存池获取内存* /
    if (! p)
        throw std::bad_alloc();
    return (pointer) p;
}
```

其中,pool 是一个 memPool 的引用,在分配子的构造函数中被初始化.

deallocate 的编写十分简单,调用 pool 的相关函数即可,

```
void deallocate(pointer p, size_type)
{
    pool.deallocate(p);
}
```

当然,一个分配子不止这么简单,只是剩余部分都可以参考默认的分配子的实现,直接拷贝,而不需要额外编写.

4 智能指针的实现

事实上,从内存池中获取内存都需要遵循“有借有还”的原则,即分配内存后必须显式释放它,这需要用户手动编写释放内存的代码,这显然是很麻烦的.

C ++ 标准库中有一种名叫智能指针的类,它拥有和普通指针类似的行为,但是却能在离开作用域时自动释放内存. 受此启发,本研究专门为内存池打造了相关的智能指针.

首先是指向缓冲区的智能指针,这类智能指针可以当成动态数组来使用,其类的签名如下,

```
class poolPtr
{
public:
    explicit poolPtr(size_t);
    virtual ~poolPtr();
    void reset(size_t);
    void * get();
protected:
    void * ptr;
    AMemPool &pool;
};
```

通过 get 函数可以获得原始指针,通过 reset 可以重设缓冲区的大小. 相关函数的实现比较容易,所以这里不再赘述.

为了能让智能指针指向各种类的对象,还需要设计一款 objectPtr,它派生自 poolPtr:

```
template < class T >
class objectPtr : protected poolPtr
{
public:
    objectPtr()
virtual ~objectPtr();
    T * get();
    T * operator -> ();
    T &operator* ();
};
```

该实现是在 poolPtr 的基础上增加了一些操作符而已,这些操作符底层调用父类的相关操作. 需要注意的是,继承方式是 protected 而不是 public,这么做是为了禁用 reset 函数.

5 性能测试与改进

为了测试本内存池算法的性能,本研究编写了相关代码用于测试,其代码为,

```
const int loopCount = 10000000;
const int blockSize = 10;
memPool::iniPool(1, 1000 * 1024 * 1024); /* 结点数量应尽
                                        量少* /
memPool &pool = memPool::getMemPool();
std::vector < void * , poolAllocator < void * > > s(loopCount);
                                    /* 使用内存池的分配子* /
for (int i = 0; i < loopCount; i + + )
    s[i] = pool.allocate(blockSize);
for (int i = 0; i < loopCount; i + + )
    pool.deallocate(s[i]);
for (int i = 0; i < loopCount; i + + )
    s[i] = malloc(blockSize);
for (int i = 0; i < loopCount; i + + )
    free(s[i]);
```

上述程序用于对自定义函数库和系统函数库进行1 000万次内存分配和回收运算进行比较测试. 利用 Visual Studio 提供的性能分析工具,可以看出效率差距,测试结果如图 1 所示.

函数名	非独占样本	独占样本	非独占样本 %	独占样...
[ntdll.dll]	924	687	100.00	74.35
memPool::memNode::deallocate	87	87	9.42	9.42
main	914	59	98.92	6.39
[ucrtbase.dll]	747	51	80.84	5.52
[ntdll.dll]	16	16	1.73	1.73
<lambda_9e6384b639662f4d55cd3cb24b2a6d63>::operator()	10	10	1.08	1.08
memPool::allocate	24	9	2.60	0.97
memPool::memNode::allocate	15	5	1.62	0.54
__scrt_common_main_seh	924	0	100.00	0.00
[wow64cpu.dll]	16	0	1.73	0.00
[wow64.dll]	16	0	1.73	0.00
[kernel32.dll]	924	0	100.00	0.00

图 1 1 000 万次内存分配与回收性能报告

在整个程序的运行时间中,内存池的所有相关操作仅仅占了 10.97%,剩下的时间用于 malloc/free 以及其他的一些占用时间不长的函数.

事实上,Visual Studio 所提供的性能分析工具非常专业和精准,根据这个测试结果可见,本研究设

计的内存池拥有相当好的性能表现.

6　结　论

本研究对循环首次适应算法进行了改进,设计并实现了基于 C ++ 的高效内存池及一系列的辅助工具,封装成了可分发重用的类库,能够进行分发和复用. 实际测试表明,通过该内存池使用策略来做内存分配/回收的工作,其效率远高于直接调用内存分配/回收的函数. 该内存池最大的优点在于它可以和标准库相配合,保证了它的实用性. 不仅如此,与内存池配套的智能指针也大大减少了手动管理内存的工作量,大幅度提升了使用者的工作效率和程序的运行效率.

参考文献:

[1]刘磊. Linux 内核内存池实现研究[J]. 科学技术与工程, 2007,7(12):2849 -2851.

[2]吴捷,陶志荣. 一种自适应变长块内存池 SVBSMP[J]. 计算机应用,2008,28(S1):280 -283.

[3]余俊良,杨正益. 基于虚拟单元可智能增长的内存池研究[J]. 计算机工程与应用,2014,50(16):127 -130.

[4]许健,于鸿洋. 一种 Linux 多线程应用下内存池的设计与实现[J]. 电子技术应用,2012,38(11):146 -149.

[5]马红斌,张峰,付华楷,等. 嵌入式系统高效内存池的实现方法: CN 101968772 B[P]. 2011 -02 -09.

[6]Wang N, Liu X, He J, et al. *Collaborative memory pool in cluster system* [C]//*International Conference on Parallel Processing*, 2007. Xi'an, China: IEEE Press, 2007.

[7]Wang X M, Wang Z. *Design and implementation of memory pools for embedded DSP* [C]//*International Conference on Computer Science and Software Engineering*, *CSSE* 2008. Wuhan, China: IEEE Press, 2008: 160 -164.

[8]侯捷. STL 源码剖析[M]. 武汉: 华中科技大学出版社, 2002.

[9]Bryant R E. 深入理解计算机系统[M]. 龚奕利,贺莲,译. 北京: 机械工业出版社,2016.

Design and Implementation of Efficient Memory Pool Based on C ++

YAN Tao[1,2], *YU Xi*[1,2], *LIU Yonghong*[1,2], *ZHAO Weidong*[1,2], *YU Yue*[1], *ZENG Yi*[1]

(1. School of Information Science and Engineering, Chengdu University, Chengdu 610106, China;
2. Key Laboratory of Pattern Recognition and Intelligent Information Processing, Chengdu University, Chengdu 610106, China)

Abstract: In the development of large scale software, memory allocation and reclaiming usually occur for efficiency and safety. Therefore, C ++ provides special functions for this, such as malloc, etc. These functions can meet the demands of common users. However, these functions call API, so the actual use of these functions produce massive memory fragments in OS and result in the efficiency bottleneck of memory allocation which lowers the system performance. This paper has improved next-fit algorithm and designed a memory pool with efficient strategies based on C ++. The strategies could extremely upgrade the efficiency of memory allocation and reclaiming. Meanwhile, for the perfect connection of memory pool with C ++ standard library, the strategies provide relevant allocators for memory pool. In addition, there are a number of smart pointers which have the function of garbage collection. They can free users from complex and fallible memory management, and improve the efficiency of the program.

Key words: memory pool; memory allocation; next-fit algorithm; efficient strategy