基于 C++自定义内存分配器的实现

肖钦定，余小喜

（92961 部队，海南 三亚 572021）

摘　要：一些需要长时间可靠运行的特殊系统，在进行频繁的内存分配和释放操作的过程中，容易产生内存碎片，影响内存分配的速度，降低内存利用率，导致系统运行越来越慢。虽然，静态分配内存的方案可以解决部分问题，但容易造成内存空间的浪费。一个简单的自定义内存分配器，实现了在提高内存使用率的同时，还能减少内存碎片的产生。

关键词：自定义内存分配器；内存使用率；内存碎片

DOI:10.16184/j.cnki.comprg.2017.11.012

1　概述

对于 C++程序员来说，内存的分配和回收是影响程序运行的关键因素。C++语言没有自动的内存管理机制，内存区域的管理由程序员自主控制。这种方式虽然提高了程序设计的灵活性，充分利用了内存的性能，但也给系统的内存管理带来了一系列问题。例如，在最近开发的一个多型异构设备互联项目中，传输模块通过对接不同型号的采集设备，实现数据的快速接收、处理和转发，这一过程需要进行频繁的内存分配和回收操作。一方面，频繁的内存分配操作容易因遗忘回收而造成内存泄漏；另一方面，模块中小对象的频繁分配操作容易产生大量的内存碎片，给系统的内存分配造成压力，从而影响整个系统的运行效率。

典型的解决方案是采用静态分配方案，事先声明所有对象，从而摆脱动态内存分配的局限。但是，静态声明会导致一些对象在不活动时也占用内存空间，造成内存空间的极大浪费。而且，相对于动态分配，静态分配存在不符合实际、灵活性不够和不贴近自然等问题，需要设计一种既兼具动态分配和静态分配优点，又能灵活回避两者不足的新方案。

采取利用静态池实现小对象动态分配的方案，构建了一种基于 C++的自定义内存分配器，有效综合了静态分配和动态分配的优点，提高了内存分配效率，大大减少了内存碎片的产生。

2　实现方法

2.1　类的实现

（1）设计一个类 Allocator，在构造函数中通过传入参数，一次性从堆中动态分配固定大小的区域，构建静态池，然后以堆栈的方式进行块管理，内存池初始状态如图 1 所示。

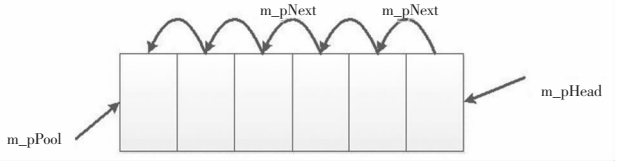

图 1　固定内存池初始状态

（2）当需要进行内存分配时，取出 m_pHead 所指向的块区，分配给申请对象使用。当静态池不能满足需求时，向系统申请空间。动态分配后静态池变化情况如图 2 所示。

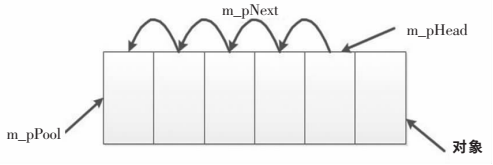

图 2　动态分配后静态池变化情况

（3）当需要进行内存释放时，并不是将真正的内存释放给系统，而是简单地将其入栈，交由 m_pHead 堆栈管理。当 Allocator 对象的生命周期结束时，由析构函数释放回收 m_pPool 指向的内存空间。

2.2　类的应用

为了方便类的使用，以及符合程序员的使用习惯，定义宏 IMPORT_ALLOCATOR，宏自动完成对 new 和 delete 运算符的重载以及静态 Allocator 对象的定义。在完成定

作者简介：肖钦定（1983–），男，工程师，本科，研究方向：指挥自动化；余小喜（1983–），男，工程师，本科，研究方向：指挥自动化。

收稿日期：2017–03–13

义后，需要完成静态池空间的初始分配。代码如下：

```
#define IMPORT_ALLOCATOR \
    public: \
        void* operator new(size_t size) { \
            return _alloc.Alloc(size); \
        } \
        void operator delete(void* pB) { \
            _alloc.DeAlloc(pB); \
        } \
    private: \
        static Allocator _alloc;
#define INIT_ALLOCATOR (T, counts)  Allocator T::_alloc(sizeof(T), counts);
```

例如：

```
#include "Allocator.h"          //引入头文件
class TestClass
{
    IMPORT_ALLOCATOR   //引入 Allocator 类
    // 定义的其他成员
};
 INIT_ALLOCATOR(TestClass, 500)     //静态池空
//间的初始分配
```

然后可以通过 new/delete 操作申请/释放内存空间，这样符合 C++程序员的编码习惯，十分自然。在由 TestClass 派生的类中，同样可以使用，示例如下：

```
TestClass* pT =new TestClass;
delete  pT;
```

2.3 主要代码

```
 Allocator::Allocator (size_t size, UINT counts):
m_pHead(NULL),m_pPool(NULL)
{
 m_pPool = new CHAR[size * counts]; //静态池的
//初始化
 for(UINT i=0; i<counts;i++)
 {
      CHAR *pT = m_pPool + i * size;
 this->Push((Block *) pT );    //构建空闲块栈链表
 }
 }
 void Allocator::Push(Block * pNode)     //入栈函数
 {
 pNode->pNext = m_pHead;
 m_pHead = pNode;
 }
 void* Allocator::Pop()                  //出栈函数
 {
 Block* pBlock = NULL;
    if (m_pHead)
    {
        pBlock = m_pHead;
        m_pHead = m_pHead->pNext;
    }
    return (void*)pBlock;
 }
 void* Allocator::Alloc(size_t size)      //分配函数
 {
 if(m_pHead)
 {
      return Pop();
 }
 else
 {
      CHAR *pT = new char[size];  //如果静态池分
//配完毕,需要从系统中申请
      Push((Block *)pT);
      return pT;
 }
 }
 void Allocator::DeAlloc(void *pB)    //回收函数
 {
 Push((Block *)pB);   //回收内存,内存块简单入栈
 }
```

3 实验结果

本次实验在虚拟机环境下进行，通过定义一个简单的类 T，记录在使用自定义内存分配器和不使用的情况下，动态构造、释放 5000 个对象消耗的时间，代码如下：

```
class T
{
IMPORT_ALLOCATOR;
public:
      int        x;
      int        y;
};
INIT_ALLOCATOR(T, 5000);
T*  pT[5000];
for(int i=0; i<5000; i++)
pT[i] = new T;
```

4 次实验结果如表 1 所示，从实验数据来看，在使用自定义内存分配器进行分配时，无论是分配速度还是

(下转第 50 页)

```
TABLE PDcj, TABLE BZrq}, null, null, null, null, null,
null, null, null, null);
}else string alert = "姓名和身份证号码不匹配！ ";
public void onClick(View v) {
    // TODO Auto-generated method stub
      textView.setText  ("xm");  textView.setText ("xb");
textView.setText("zsbh"); textView.setText("sfzh");
 }
 }
```

3.4 系统实现

当查询者输入和数据库匹配的姓名和身份证号码时，系统出现图 2 界面。

姓名：	陈沁林
性别：	男
出生日期：	1999 年 10 月 17 日
证书编号：	1622030000424014
身份证号：	510304199910174118
职业（工种）及等级：	维修电工四级
理论知识考试成绩：	60.0
操作技能考核成绩：	60.0
评定成绩：	合格
颁证日期：	2016-06-24

图 2

4 结语

分析了移动互联网及 Android 智能平台的应用需求，设计了基于 Android 的校园鉴定成绩查询系统，该系统的运行，方便了学生对鉴定成绩及证书的查询，家长对学生学习状况的掌握，企业对学生技能水平的了解，对技能证书的推广及学生的鉴定成绩管理有着重要的意义。

参考文献

[1] 杨丰盛. Android 应用开发揭秘. 机械工业出版社.

[2] E2Ecloud. 深入浅出 GoogleAndroid. 人民邮电出版社.

[3] [美] Ed Burnette. Android 基础教程. 人民邮电出版社.

[4] 利用 JSON 构建 Android 终端的 WebAPI. 中国信息技术教育，2015，(5).

(上接第 34 页)

写入缓存中，待缓存写满时系统先通知 Journal，再将文件写入硬盘，完成后再通知 Journal，资料已完成写入工作；在 Ext3 中，也就是有 Journal 机制里，系统开机时检查 Journal 的内容，来查看是否有错误产生，这样就加快了开机速度。

5 结语

研究了 Ext 文件系统的产生和发展，对文件系统基本原理进行阐述。同时对不同阶段 Ext 文件系统的优缺点进行了比较，为在 Ext 文件系统进行文件存储和数据恢复提供了一定的理论指导。

参考文献

[1] 郭东强. 现代管理信息系统 [M] . 北京：清华大学出版社，2010：3-8.

[2] 毛德操，胡希明. Linux 内核源代码情景分析 [M] . 杭州：浙江大学出版社，2012：32-39.

[3] 李善平，陈文智. 边学边干 Linux 内核知道 [M] . 浙江大学出版社，2002.

[4] 王俊伟，吴俊海. Linux 标准教程 [M] . 北京：清华大学出版社，2006.

(上接第 39 页)

释放速度都有较大的提高，并且大大减少了内存碎片的生成。

表 1 两种方式动态构造、释放 5000 个对象的时间消耗表

实验次数	不使用		使用	
	分配（mS）	释放（mS）	分配（mS）	释放（mS）
1	3451	1295	236	210
2	1951	1588	345	305
3	2493	1601	376	305
4	2556	2337	345	305

4 结语

通过使用静态池以及 new/delete 运算符重载的方法，实现了一种基于 C++的自定义内存分配器，成功解决了由于频繁的对象创建与回收所带来的一系列问题，提升了内存使用率，减少了内存碎片，提高了系统整体性能。但这种方法在实际使用中仍会存在一定的局限性，比如存在使用不够灵活，无法完成对象数组的定义操作等问题，需要进一步进行深入的探索研究。