DOI:10.16208/j.issn1000-7024.2009.16.016

• 体系结构与外围设备 •

内存管理机制的高效实现研究

魏海涛[1]， 姜昱明[1]， 李建武[1]， 张 娅[2]

(1. 西安电子科技大学 计算机学院，陕西 西安 710071；2. 96425 部队，陕西 宝鸡 721006)

摘 要：为了高效地管理内存，防止内存泄漏、越界访问等问题的出现，在分析传统动态内存管理机制的基础上，提出了一种基于 Windows 虚拟内存管理的高效动态内存管理机制。通过创建内存管理器来维护虚拟地址空间和处理内存申请和释放请求，改进了传统动态内存分配与回收算法，并在 VC++6.0 开发平台上实现了该动态内存管理机制。测试结果分析表明，该方法有效降低了申请与释放内存的时间耗费，减少了内存碎片的产生，提高了动态内存管理效率。

关键词：内存管理； 堆； 分配算法； 回收算法； 内存池

中图法分类号：TP311 **文献标识码**：A **文章编号**：1000-7024 (2009) 16-3708-05

Research of high efficient implementation of memory management mechanism

WEI Hai-tao[1], JIANG Yu-ming[1], LI Jian-wu[1], ZHANG Ya[2]

(1. College of Computer Engineering, Xidian University, Xi ån 710071, China; 2. 96425 Unit of PLA, Baoji 721006, China)

Abstract: Having analyzed the traditional dynamic memory management, an efficient dynamic memory management method based on Windows virtual memory mechanism is presented aiming to both manage memory efficiently, and avoid the problem of memory leak and access violation. The virtual address space and memory allocation and release are managed by a memory manager. The method improves the traditional memory allocation and release algorithm. And memory management mechanism is implemented on the VC++ 6.0 development platform. Analysis of test results shows that the method shortens the time consumption of memory allocation and release, reduces generated memory fragments, and improves dynamic memory management efficiency.

Key words: memory management; heap; allocation algorithm; free algorithm; memory pool

0 引 言

随着计算机硬件的发展和软件需求的增长，软件规模变得越来越庞大，从而对计算机内存的需求也变得越来越高。在软件开发的过程中，如何高效的管理大量的内存、防止内存泄漏、防止越界访问，以及如何提高内存的有效利用率已经成为影响软件质量的重要因素，因此良好的内存管理机制显得非常重要[1]。

本文将探讨 Windows 下的一种高效的动态内存管理机制。与传统的动态内存管理机制相比较，文中提到的动态内存管理机制减少了内存申请与释放的时间耗费，减少了内存碎片，从而提高了动态内存管理的效率。

1 传统动态内存管理机制

32 位 Windows 环境中的进程都至少有一个堆，叫做缺省堆。Win32 子系统和 C 的运行时库都从该缺省堆中动态分配空间。在 C/C++中，用 malloc/free 或 new/delete 来申请或释放动态内存，而它们最终都是通过 Windows 的堆内存管理来实现的。堆管理是 Windows 系统内存管理的核心部分，是 Windows 提供的一种内存管理机制，且主要用来分配小的数据块。

通过调用 C 的 malloc () 函数或者 C++的 new 运算符来动态申请内存，其最终都将调用 NTDLL. DLL 中的 RtlAllocateHeap 从 Windows 内存堆中进行内存的实际分配。通过调用 C 的 free () 函数或者 C++的 delete 运算符来动态释放内存，其最终也都是调用 NTDLL.DLL 中的 RtlFreeHeap 对从系统内存堆中申请的内存进行回收[2]。

分析文献[2]中提到的 Windows 下的堆内存管理机制的时间效率。申请内存时，考虑到需要对空闲块进行分割，则申请内存的时间耗费约等于在空闲块链表中顺序查找到合适大小的空闲块的时间耗费加上将分割后的空闲块插入到空闲块链表中适当位置上的时间耗费；释放内存时，考虑到需要对空闲块进行合并，则释放内存块的时间耗费约等于将合并后的空闲块插入到空闲块链表中适当位置上的时间耗费。

为了使分析更具一般性，这里使用需要遍历链表节点的平均个数来衡量内存申请与释放的时间效率。定义：记 L 为空闲内存块链表的长度，则顺序查找到空闲内存块链表的某一

收稿日期：2008-09-03；修订日期：2008-12-06。

作者简介：魏海涛（1979－），男，陕西西安人，硕士研究生，研究方向为图形图像处理与网络技术； 姜昱明（1949－），男，陕西延安人，副教授，硕士生导师，研究方向为图形图像处理与网络技术； 李建武（1983－），男，陕西西安人，硕士研究生，研究方向为图形图像处理与网络技术； 张娅（1979－），女，河南固始人，助理工程师，研究方向为图形图像处理。E-mail：whtlszy@126.com

个空闲块或者定位到某一个合适的位置需要遍历空闲块的平均个数是$\frac{1}{L}\sum_{i=1}^{L} i=\frac{1}{2}L+\frac{1}{2}$；记$T_1$为传统内存管理机制下内存申请算法中需要遍历链表节点的平均个数，T_2为传统内存管理机制下内存释放算法中需要遍历链表节点的平均个数，通过以上的分析可得

$$T_1=(\frac{1}{2}L+\frac{1}{2})+(\frac{1}{2}L+\frac{1}{2})=L+1 \quad (1)$$

$$T_2=\frac{1}{2}L+\frac{1}{2} \quad (2)$$

由式(1)和式(2)可以看出，随着内存碎片的增多，空闲块链表的长度逐渐增大，申请和释放所用的时间也将增大。因此，如何提高空闲块链表的查找效率以及减少内存碎片成为提高动态内存管理效率的关键。

分析文献[2]中提到的Windows下的堆内存管理机制的空间效率。内存块分为空闲块(FreeBlock)和忙块(BusyBlock)，其对应的内存块管理机构所占用的大小分别为16字节和8字节。这些额外的空间是内存管理所必须的。

2 动态内存管理机制的高效实现

2.1 实现原理

该机制的核心思想是创建一个自定义的内存管理器，用来维护通过调用虚拟存储函数VirtualAlloc()创建的进程虚拟地址空间上的由若干个页组成的区域，并处理客户程序中的内存申请和释放请求，从而实现一个自定义的动态内存管理器。

当客户程序申请内存时，由该内存管理器从其所维护的虚拟地址空间区域为客户程序分配所需大小的空间，而不再通过C/C++中的malloc()/new来实现；当客户程序释放内存时，由该内存管理器进行内存块的回收工作，而不再通过C/C++中的free()/delete来实现。当程序退出时，由该内存管理器调用虚拟存储函数VirtualFree()释放其所占用的进程虚拟地址空间上的页。

2.2 内存管理数据结构

为了实现高效的内存管理，本文参见文献[2]中所提到的Windows下堆内存的管理机构，建立了两种数据结构：内存块管理机构和内存池管理机构。

内存块是客户程序请求分配内存的结果，由内存块管理机构和可供客户程序使用的内存两部分组成。内存块管理机构包含了该内存块的大小、占用标志、指向上下相邻连续内存块的指针、指向上下空闲内存块的指针以及指向其所属的内存池的指针等信息。内存块的结构如图1所示。可以计算出该内存块管理机构(内存块管理头结点)所占用的大小是4×7=28字节，与传统内存管理机制相比，本文提出的动态内存管理机制增加了额外空间开销，即降低了空间效率。

内存池是进程的虚拟地址空间中多个地址连续的占用的页所对应的物理存储空间，其头部是内存池管理机构。内存池管理机构包含了该内存池的起始地址、总大小、最大可用连续空间的大小、指向内存块双向链表的指针、指向空闲内存块双向链表的指针以及指向下一内存池的指针等信息。为了方便地将待回收内存块和与其相邻的空闲内存块合并为一个更

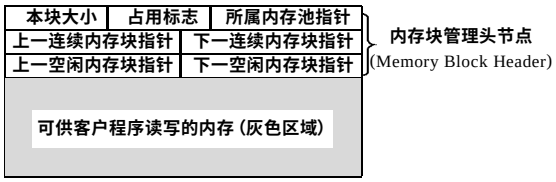

图1 内存块的结构

大的可用空闲内存块，以减少空闲内存块链表的长度，内存块在内存池中按照其物理位置以双向链表进行组织；为了提高内存的利用率，减少内存碎片，分配内存时按照“最小满足”原则查找空闲块链表，为客户程序分配能够满足其大小的最小内存块；空闲内存块在内存池中按照从小到大的顺序以双向链表进行组织。内存池的结构如图2所示。

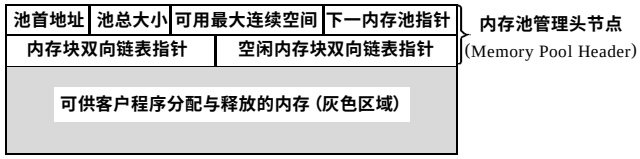

图2 内存池的结构

初始建立的内存池如图3(a)所示，此时整个内存池除了内存池管理头结点外，其余空间都可供客户程序进行内存的申请与释放；经过若干次分配与回收后，典型的内存池如图3(b)所示。

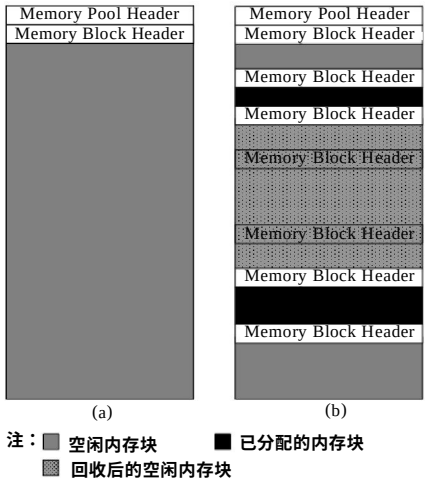

图3 运行时的内存池结构

2.3 内存分配与回收算法分析

对应客户程序的内存申请与释放，动态内存管理机制需要提供相应的内存分配与回收机制。

2.3.1 内存分配算法

当客户程序申请内存时，首先将客户程序申请的字节数按4字节对齐，然后查找能够满足该申请要求的内存池，若找到，则就在该内存池中为客户程序分配内存；否则，根据申请的字节数和一定的规则创建新的内存池，然后在新内存池中进行分配。

创建新内存池的规则是：在内存池的创建过程中，定义可创建最大的内存池的大小为用户申请的字节数按4字节对齐

之后再按Windows系统页面大小对齐的大小的40倍[3],若其40倍超过256MB(可以根据需要进行调整),则按最大256MB创建内存池,这主要是考虑到一般程序对动态内存申请释放的需求,也为了避免影响其它应用程序对内存的需求。

在内存池中分配内存时,顺序查找该内存池的空闲内存块双向链表,查找到的第一个满足分配字节数大小要求的空闲块即为能够满足分配要求的最小空闲块,并将该空闲块从该双向链表上移除。

如果该空闲块的大小除了能够满足客户程序所申请的字节数要求外还有一定大小的可用空闲空间,则将其分割为两个新的内存块,其中第一块是为客户程序分配的内存,将其占用标志置为"占用",第二块是可供客户程序后续申请分配的新空闲块,将其占用标志置为"空闲",并将其作为新的空闲块按照一定的算法(向前查找插入法)插入到空闲块双向链表中;否则,直接将该内存块返回给客户程序,作为为客户程序分配的内存。分配完成后更新内存池管理头结点中的最大可用连续空间的值。其中用到的向前查找插入法的思想是:分割后得到的新空闲块的大小一定小于原空闲块,而空闲块链表是按照块的大小链接的,因此只需要从原空闲块在空闲块链表中的位置向前查找到适当的位置,将新空闲块插入该位置上即可,而不必从头遍历整个空闲块链表,从而提高了空闲块链表的查找效率。使用向前查找插入法需要遍历空闲链表节点的平均个数T_3如式(3)所示,与式(1)相比,内存分配效率得到了提高。内存分配算法的流程如图4所示。

$$T_3=\frac{1}{L}\sum_{i=1}^{L}(i+\frac{1}{i}\sum_{k=1}^{i}k)=\frac{3}{4}L+\frac{5}{4} \tag{3}$$

2.3.2 内存回收算法

在对用户的内存进行回收时,首先对待回收的内存块向前偏移内存块管理头结点大小的字节数,找到内存块管理头结点,将其中的占用标志置为空闲。再通过内存块管理头节点找到其所属内存池的内存池管理头节点,然后检测与待回收内存块相邻的块是否为空闲块,若空闲则进行合并,从而减少了内存碎片,并将合并后的空闲块按照一定的算法(向后查找插入法)插入到所属内存池的空闲内存块链表的适当位置;否则,直接将待回收内存块作为一个新空闲块从头查找并插入到所属内存池的空闲块链表中。回收完成后更新内存池管理头结点中的最大可用连续空间的值。

其中用到的向后查找插入法的思想是:合并后得到的新空闲块的大小一定大于原相邻空闲块,而空闲块链表是按照块大小链接的,因此只需要从原相邻空闲块在空闲块链表中的位置向后查找到适当的位置,将新空闲块插入该位置上即可,而不必从头查找整个空闲块链表,从而提高了空闲块链表的查找效率。使用向后查找插入法需要遍历空闲链表节点的平均个数T_4如式(4)所示,与式(2)相比,内存回收效率得到了提高。

$$T_4=\frac{1}{L}\sum_{i=1}^{L}(\frac{1}{L-i+1}\sum_{k=1}^{L-i+1}k)=\frac{1}{4}L+\frac{3}{4} \tag{4}$$

内存回收算法的流程如图5所示。

2.4 C++的具体实现

在Windows XP下使用VC 6.0开发平台实现了本文提出

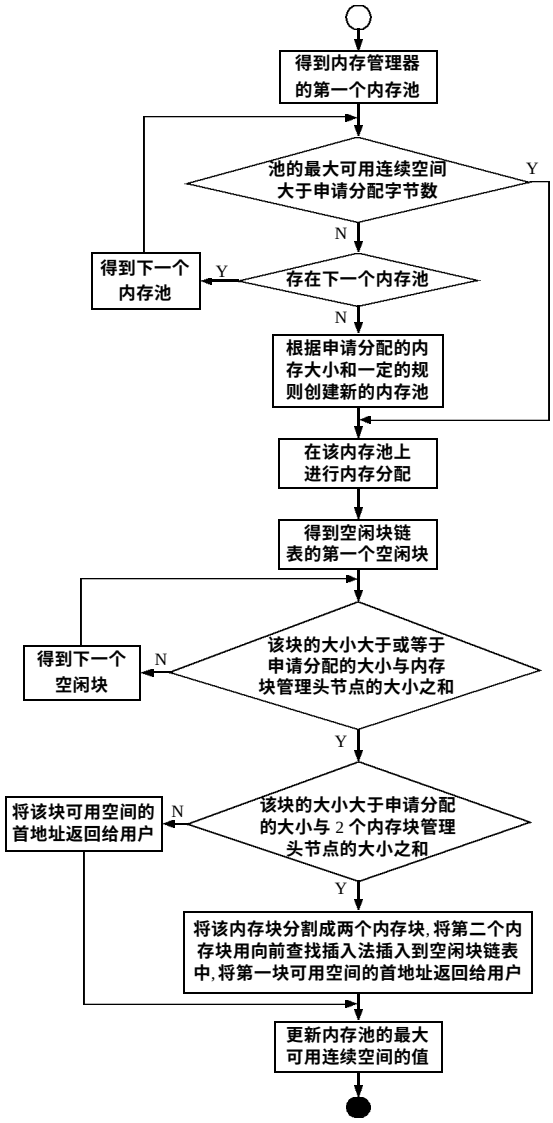

图4 内存分配算法流程

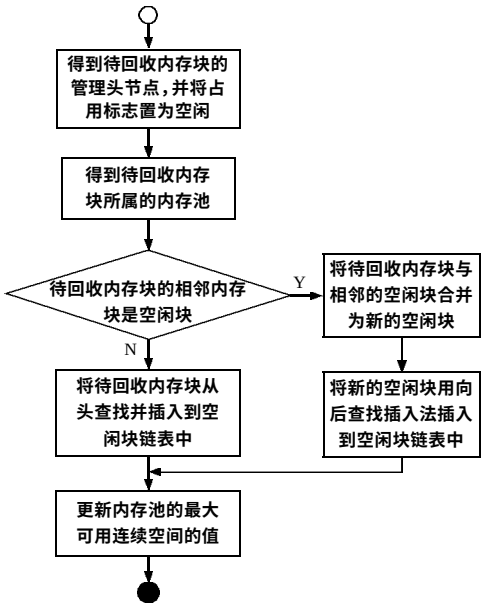

图5 内存回收算法流程

的高效动态内存管理机制。建立了一个称为内存管理器(CMemoryManager)的类，用来处理客户程序对动态内存的申请与释放请求；定义内存块管理头结点(MemoryBlockHeader)为一个结构体，用来维护一个内存块的信息以及其与前后内存块的关系；定义一个内存池(MemoryPool)类，用来维护内存池的管理信息，并且定义了该内存池的内存块的操作方法。各个类和数据结构的UML定义如图6所示。

2.5 测试结果与分析

在Windows XP平台下，对本文提出的动态内存管理机制的实现进行了测试。测试机器的基本配置为：512MB的物理内存，Intel Celeron M 1.50GHz的CPU。测试分为两种，第1种在测试机器相对重载(CPU使用率大于5%，内存使用率大于60%)的情况下进行，测试结果如图7(a)所示；第2种在测试机器相对轻载(CPU使用率小于3%，内存使用率小于40%)的情况下进行，测试结果如图7(b)所示。每种测试中客户程序对传统堆管理器和本文提出的动态内存管理器都进行了400次测试(图7中的一个离散点表示一次测试)，每次测试使用这两种内存管理器分别进行完全相同的n次随机内存申请与释放操作(n是一个小于2100的随机正整数)，其中包含了n次申请操作和n次释放操作，操作顺序是随机的，但是对一个内存块的释放操作必须是在申请操作成功之后发生的，每次申请内存大小是一个小于4MB的随机正整数。

从测试结果可以看出，由于改进了内存的分配与回收算法，缩短了对空闲块链表的查找长度，本文所提出的动态内存管理机制的时间效率高于传统的堆内存管理机制，在机器重载的情况下这种效果更加明显。

3 结束语

高效的内存管理机制有利于提高内存的使用效率，减少内存碎片。为了满足大规模软件对动态内存分配和释放效率的要求，可以为软件量身定做能够满足其要求的动态内存管理机制。

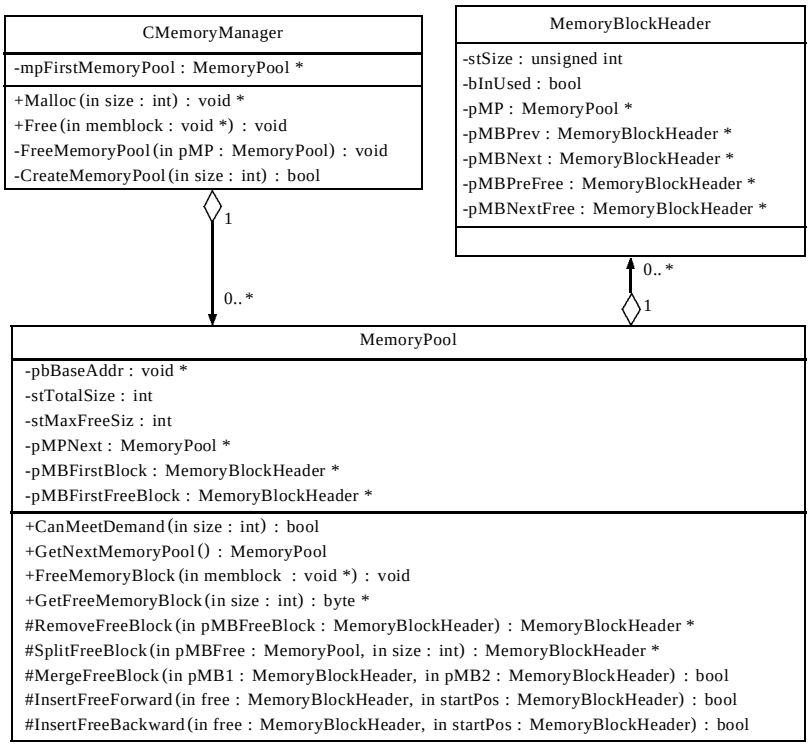

图6 UML类图

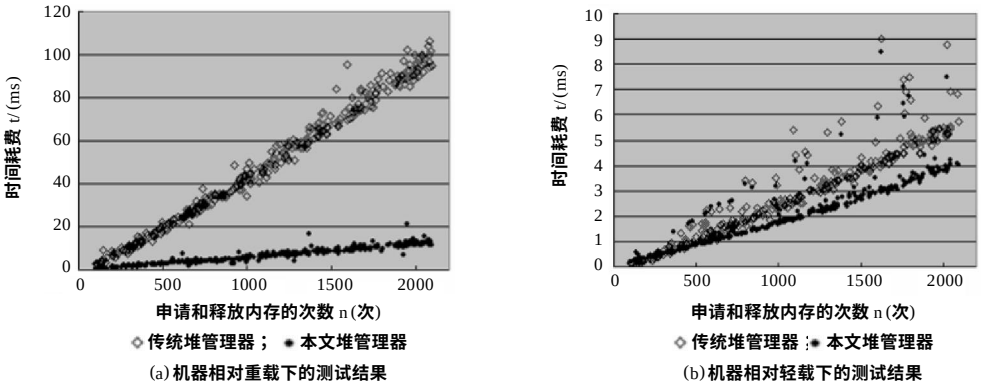

图7 测试结果

测试结果表明，本文所提出并实现的高效的动态内存管理机制缩短了申请与释放内存的时间耗费，减少了内存碎片的产生，提高了动态内存管理的时间效率。同时需要看到，该内存管理机制中的内存管理机构（内存块管理头结点和内存池管理头节点）占用了比较大的空间，增加了内存管理的额外空间开销，所以该内存管理机制在时间上的高效是以占用更多的空间为代价的。

参考文献：

[1] 周政春,吴楷,万旺根.内存管理算法优化及在游戏引擎中的实现[J].微计算机信息,2006,22:212-214.

[2] 胡兆阳,谢余强,舒辉.Windows 下堆内存管理机制研究[J].计算机工程与应用,2005,41(17):59-61.

[3] 侯捷.池內春秋——Memory Pool 的设计哲学和无痛运用[J].程序员,2002(9):94-97.

[4] 王珊,肖艳芹,刘大为,等.内存数据库关键技术研究[J].计算机应用,2007,27(10):2353-2357.

[5] 耿国华.数据结构——C 语言描述[M].西安:西安电子科技大学出版社,2002.

[6] 阎梦天,丁志刚,王挺,等.实时操作系统内存分配性能检测[J].计算机应用,2007,27(11):2838-2840.

[7] 戚海燕.静态内存管理系统的研究与应用[J].计算机时代,2006(12):19-21.

[8] 王明路,王希敏,王哲.嵌入式系统中池式内存分配方法的分析[J].计算机与数字工程,2008(2):57-61.

(上接第 3704 页)

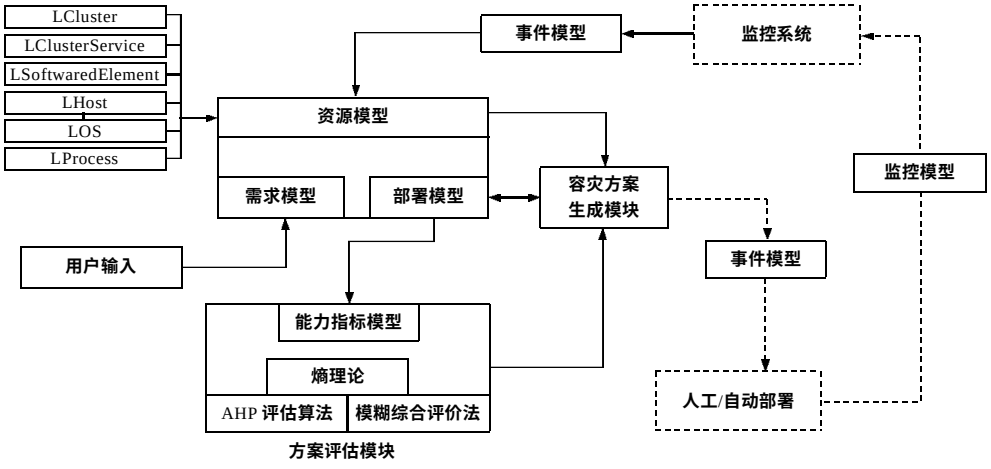

图 5 容原型系统结构

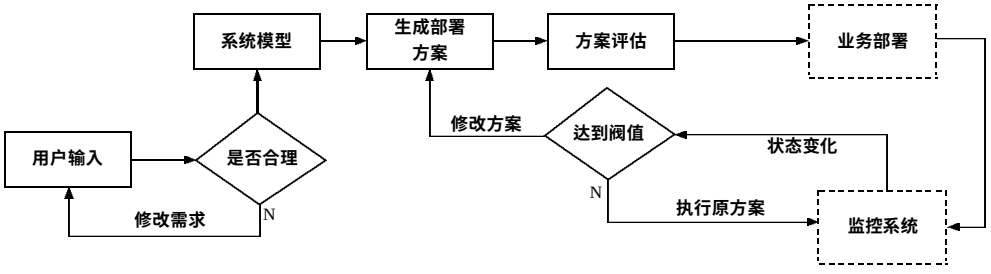

图 6 运行流程

参考文献：

[1] 李涛,刘晓洁.信息系统容灾抗毁原理与应用[M].北京:人民邮电出版社,2007.

[2] ITU-T Recomm.X.710.Common management information service definitions[S].

[3] RTC 2578 structure of management information version 2 (SMIv2)[S].

[4] OMG.The common object request broker: architecture and specification [EB/OL].http://www.omg.org/technology/documents/corba/,2004.

[5] CIM schema v2.8[EB/OL].http://www.dmtf.org/standards/cim/,DMTF Inc,2004.

[6] 陈靖,李增智,王云岚,等.基于公共信息模型的分布式系统业务管理[J].西安交通大学学报,2005,39(2):126-129.

[7] 李航,潘成胜,刘勇.对军事电子信息系统控制中通用信息模型的研究[J].兵工学报,2004,25(5):604-608.

[8] 杨秀梅,吕卫锋.基于 CIM 的网络性能管理[J].计算机工程与设计,2006,27(8):1425-1428.

[9] 肖政,韦卫,侯紫峰.基于 WBEM 的统一管理系统[J].计算机工程,2005,31(1):47-49.