第34卷 第16期 计 算 机 工 程 2008年8月
Vol.34 No.16 Computer Engineering August 2008
·安全技术· 文章编号：1000—3428(2008)16—0159—03 文献标识码：A 中图分类号：TP311.5

基于红黑树的堆内存泄漏动态检测技术

葛 瑶[1]，李晓风[1]，孔德光[2]
(1. 中国科学院合肥物质科学研究院信息中心，合肥 230031；2. 中国科学技术大学自动化系，合肥 230027)

摘 要：设计与实现一个轻量级的堆内存泄漏检测工具，针对使用C++编码的开源代码，通过重载new, delete运算符，动态跟踪程序在执行过程中堆内存块的分配释放情况，在程序运行结束时给出内存泄露的检测结果。实现时采用红黑树管理所分配的堆内存，理论推导和实验表明其具有较高的效率。
关键词：堆内存泄漏；动态检测；红黑树

Dynamic Check Technology of Heap Memory Leak Based on Red-black Tree

GE Yao[1], LI Xiao-feng[1], KONG De-guang[2]
(1. Information Center, Hefei Institutes of Physical Science, Chinese Academy of Sciences, Hefei 230031;
2. Department of Automation, University of Science & Technology of China, Hefei 230027)

【Abstract】 To cope with the problem of heap memory leak, this paper presents a dynamic memory leak check technology based on red-black tree. With the method of dynamically catch the allocated situation of heap memory caused by the operator of new and delete during the execution of the program, decide whether the heap memory leak occurs. The system is implemented based on red-black tree, to manage the allocated heap memory. It is proved by theory induction and experiment that it has higher efficiency and is well platform-independent and scalable.
【Key words】 heap memory leak; dynamic check; red-black tree

随着信息社会对软件依赖的不断增加，软件安全成为一个焦点问题。缓冲区溢出、数组访问越界、悬空指针访问、内存泄露是几种常见的低级safety错误。堆内存泄露由于用户频繁地创建、释放对象和分配、释放内存块而在C++语言中发生。目前Windows平台上流行的内存泄露检测工具有Rational Purify[1], BoundsChecker[2]，但这些工具大多是商业软件，需要花钱购买使用权；构架复杂，需要一定时间才能灵活应用；在检查一些跨平台的程序时并不好用；Linux平台下mpatrol[3]之类的现有工具，易用性、附加开销和性能也都不太理想。本文设计与实现了一个轻量级的堆内存泄漏检测的工具，旨在解决由于new/delete算符的使用疏忽而产生的堆内存泄漏问题。

1 堆内存分配动态检测的基本原理

对于一般的应用程序和系统程序，程序员会频繁使用new, delete算符进行堆内存的分配和释放等相关操作，而这些操作存在着极大的危险性，经常由于程序员的疏忽出现内存块的管理错误，从而导致内存块的分配错误或者泄漏。

C++编译器处理new和delete运算符时调用的是C++语言内置的new operator和delete operator[4-5]。new operator先分配足够的内存，然后调用相应类型的构造函数初始化该内存，其原型如下：

```
void * operator new(size_t size);
//返回值类型是 void*，返回一个未经处理的指针，未初始化的
//内存，参数 size 确定分配多少内存
```

delete operator先调用该类型的析构函数，而后释放内存，其原型如下：

```
void operator delete(void *pointer);
//释放传入的 pointer 参数所指向的一片内存区
```

使用operator new为自定义类型的对象分配内存时，实际得到的内存比对象的实际内存大，其中，“额外的”内存用来记录该内存的相关信息(也称cookie技术)，实现策略随编译器的不同而不同。例如，MFC选择在所分配内存的头部存储对象实际数据，后面的部分存储边界标志和内存大小信息；g++则在所分配内存的前4 B存储相关信息，而后面的内存存储对象实际数据。使用delete operator进行内存释放操作时，delete operator根据这些相关信息正确地释放指针所指向的内存块。这也是编译程序实现堆内存块分配跟踪的基础。为数组分配/释放内存时，虽然仍然使用new operator和delete operator，内部行为却有所变化：new operator调用operator new[]，而后针对每一个组成员调用构造函数；delete operator先对每一个数组成员调用析构函数，而后调用operator delete[]释放内存。当创建或释放自定义数据类型所构成的数组时，编译器为了能够标识出在operator delete[]中所需释放的内存块大小，同样使用与编译器相关的cookie技术。因此，若要动态检测有关堆内存块的泄漏问题，就要重载operator new, operator new[], operator delete, operator delete[] 4个全局函数，截获所需检测的内存操作信息。重载后的new函数原型如下：

```
void* operator new(size_t size, const char* file, int line);
//返回值类型是 void*，返回一个分配的内存块的指针，参数
```

基金项目：国家“863”计划基金资助项目(2006AA01Z449)
作者简介：葛 瑶(1984－)，女，硕士研究生，主研方向：信息安全，软件体系结构；李晓风，研究员；孔德光，博士研究生
收稿日期：2007-09-30 **E-mail：**geyao@aiofm.ac.cn

//size 确定分配多少内存，file 是 new 所在的文件名，line 是 new 算
//符所在的行号

重载后的 delete 函数原型如下，具体实现发生改变：

```
void operator delete(void *pointer);
```

2　基于红黑树的内存检测的设计与实现

2.1　红黑树的简介

红黑树(red-black tree)由 Guibas 和 Sedgewick 提出，是一种只需要部分达到平衡要求的二叉树。由于降低了对旋转的要求，提高了性能，因此应用于很多优化算法中。

2.1.1　红黑树的定义

(1)红黑树是一棵二叉搜索树，树上的每一个节点都有一种颜色。平衡条件通过约束节点的排列方式(基于其颜色)得以维持。(2)满足如下条件：1)每个节点的颜色为红色或者黑色；2)每一个叶子节点(空节点)的颜色为黑色；3)如果一个节点颜色为红，则它的 2 个子节点颜色全为黑；4)从一个节点到叶子节点的各条路径上包括的黑色节点数相同。

2.1.2　红黑树的性质

(1)定义节点的黑高度(black-height)：从一个节点向下到其一个叶子节点的路径上的黑色节点数目，不包括这个节点本身。(2) 带有 n 个节点的红黑树的高度在 $\mathrm{lb}(n+1)$ 和 $2\mathrm{lb}(n+1)$ 之间。(3)在一棵红黑树中，从根节点到叶子节点的任意一条路径上至少有一半节点是黑色的。

2.2　数据结构的实现

使用红黑树来存储所分配的每个内存块的相关信息，可用于跟踪记录每个内存块的分配与释放，其中，关键字 key 对应于可用内存块的地址。使用 new 新分配一块堆内存时，新内存块的组织结构如图 1 所示。

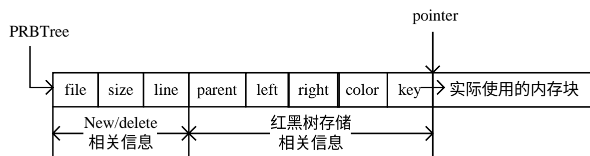

图 1　内存块组织结构

New/delete 操作和红黑树的相关信息作为附加信息附着于每个实际使用的内存块上(即 cookie)。PRBTree 指针指向每个用于保存附加信息的内存块，pointer 指针指向实际使用的内存块。

用于存储 cookie 的红黑树节点的数据结构如下：

```
typedef struct RBTree
{   struct   RBTree *parent;   //父亲节点
    struct   RBTree *left, *right;   //左右子树节点
  KEY key;   //关键字，对应于可用内存块的地址
NODECOLOR     color;   //红黑节点标识
char file[LEN];   // new/delete 所在的文件名
int       line;   // new/delete 出现的行号
size_t size;   //new/delete 分配的内存块地址
}RBTree, *PRBTree;
```

在红黑树中进行查找的几个关键函数的原型定义如下：

```
PRBTree RB_InsertNode(PRBTree root, unsigned pointer, char
*file,int line,size_t size);
//在以 root 为根的红黑树中插入内存块(pointer 为可用内存块地
址) //节点的实现原型
PRBTree RB_InsertNode_Fixup(PRBTree root, PRBTree z);
//在以root为根的红黑树中插入内存块节点后调整以z为根的子
//树的实现原型
PRBTree RB_DeleteNode(PRBTree root, KEY key);
//在以root为根的红黑树中删除关键字key对应的内存块地址的
//实现原型
PRBTree RB_DeleteNode_Fixup(PRBTree root, PRBTree z);
//在以 root 为根的红黑树中删除内存块节点后，调整以 z 为根的
//子树的实现原型
```

2.3　检测部分的实现算法

实现过程：以头文件 memory_check.h 的形式嵌入到待检测代码的源码包中，执行时动态检测。

输入　待检测的源代码程序

输出　是否内存泄漏(若是，给出内存泄漏地址、大小和行号)

(1)在程序执行时：

1)检测是否遇到 new 算符。若遇到，则调用重载过的 void* operator new(size_t size, const char* file, int line)函数，记录分配的内存块的大小、new 算符出现的位置和行号，以便定位；以实际可用内存块地址作为关键字插入到红黑树中。

2)检测是否遇到 delete 算符。若遇到，则调用重载过的 void operator delete(void* pointer)函数，根据释放的内存块的具体位置，在红黑树中查找并删除该节点。如果查找失败，抛出异常，源代码中存在多次 delete 的错误。

(2)在程序退出之前调用 ensure_MemoryLeak()函数，中序遍历内存块节点形成的红黑树，输出内存块泄漏的地址、大小和行号。如果输出为空，则说明没有内存块泄漏产生。

3　工具性能分析与讨论

3.1　理论分析

由于对分配的内存块采取了自管理的方式，因此组织该内存块的存储结构成为影响算法性能的一个重要因素。设 N 为所有分配的堆内存块的数目，M 为泄漏的内存块数目。显然在正常情况下，$M<<N$。但程序员的疏忽可能导致 M 值增大到一个较为可观的数目(不妨假定 $M\leq \mathrm{lb}N$)。其中，内存块的组织与管理可以采用以下几种形式：

(1)链表

插入的时间复杂度为 $O(1)$，删除一个内存块，最好情况下的时间复杂度为 $O(1)$，最坏情况下为 $O(N)$，平均时间为 $O(N)$，最后搜索检查泄漏的内存块，时间复杂度为 $O(M)$。总的时间消耗为

$$NO(1)+(N-M)O(N)+O(M) \tag{1}$$

(2)哈希表

插入与删除的时间复杂度最好为 $O(1)$，但实际上会发生冲突，并且有较大的不确定性。关键是找到较为理想的哈希函数和合适的哈希桶大小(设为 T)，这与被检测代码的规模密切相关。当哈希桶选择得过大时(若 $T>>N$ 时)，导致大多数项为空项，尽管没有同义词，但是搜索检查内存泄漏的时间为 $O(T)>>O(N)$；若哈希桶选择得过小，冲突发生较多，插入与删除的时间增加，但是搜索检查内存泄漏还是效率较高。总时间平均消耗略大于

$$NO(1)+(N-M)O(1)+O(T) \tag{2}$$

(3)红黑树

由红黑树的性质易知，插入与删除的时间复杂度均为 $O(\mathrm{lb}N)$，与红黑树的高度是同一数量级的，不使用普通的二叉树，就避免了最坏时出现删除节点的时间复杂度为 $O(N)$ 的情况；在搜索检查内存泄漏时，中序遍历这棵红黑树上剩余的所有节点，时间复杂度为 $O(M)$。总共时间消耗为

$$NO(\mathrm{lb}N)+(N-M)O(\mathrm{lb}N)+O(M) \tag{3}$$

由式(1)~式(3)得

$$O(N^2-2N\,\mathrm{lb}N-MN+M\,\mathrm{lb}N+N)=$$
$$O(N^2+N+M\,\mathrm{lb}N-MN-2N\,\mathrm{lb}N)>$$
$$O(N^2+N+M\,\mathrm{lb}N-3N\,\mathrm{lb}N)>$$
$$O(N^2+N-3N\,\mathrm{lb}N))=O(N(N+1-3\,\mathrm{lb}N))>0$$

(由于 $M\le \mathrm{lb}N$，因此当 N=8 时，关于 N 的单调增函数 $N+1-3\mathrm{lb}N=0$)

因此，式(1)>式(3)。

令 $T=K\mathrm{lb}N$，根据式(2)、式(3)可得

$$O(2N-2M+(K+M-2N)\mathrm{lb}N) \tag{4}$$

当 $K=N$ 时，式(4)<0；当 $K=2N$ 时，式(4)>0。由零点定理可知，$\exists K\in(1N,2N)$，使得 $T=K\mathrm{lb}N$ 时，式(4)=0。因为式(2)是 T 的单调函数，所以当 $T\ge K\mathrm{lb}N$ 且 $K\in(1N,2N)$ 时，由式(4)=0 可以确定，必有式(2)>式(3)，那么问题转化为估计 $T\ge K\mathrm{lb}N$ 成立的概率大小的问题。实际情况下 T, N 的取值随程序规模的大小而改变，具有很大的不确定性，难以给出定量的分析，而兼顾哈希函数的性质，T 的取值一般较大，因此，这时红黑树的时间效率优于哈希表。

3.2 实验验证

为使程序有较好的通用性，采用了基于红黑树内存结构的存储算法。为了验证不同数据结构对检测堆内存块泄漏性能的影响，实现了内存块在不同数据结构上进行存储的检测算法。实验运行平台为 Pentium4 CPU 1.7 GHz, 256 MB 内存，Win XP 操作系统，运行环境为 VC++6.0。其中，T=10 000, N=25, 50, 75, 100, 125。图 2、图 3 为实验结果。

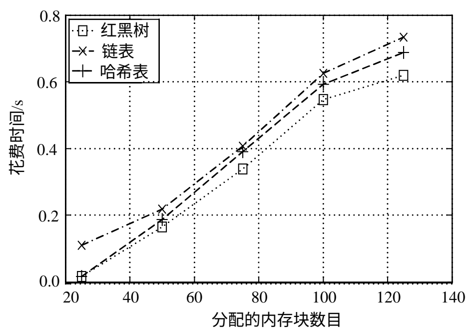

图 2　不同存储结构内存泄漏检测算法效率比较(M=8)

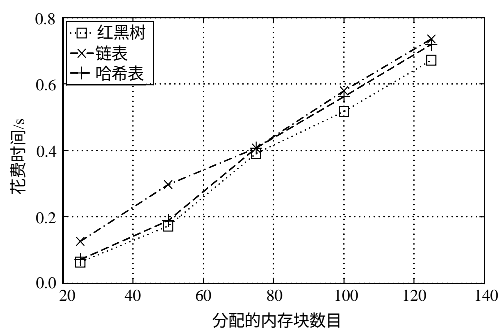

图 3　不同存储结构内存泄漏检测算法效率比较(M=16)

4　结束语

本文设计了基于红黑树的检测算法用以检测由 new/delete 引起的堆内存泄漏，并且具有较高的效率。实现时数据结构不依赖 STL 库，具有良好的平台无关性和可扩展性，可用于 Windows 和 Linux 平台代码的自动检测。存在的不足和改进的方向如下：

(1)能够检测 new/delete/malloc 引起的内存泄漏问题，但是对于一些内存分配的 Windows API 函数(如 RtlHeapAlloc)并没有拦截检测，尽管这些函数出现概率不高，但是在特定的程序中可能频繁出现。下一步将增加相应的拦截函数，跟踪此类函数产生的堆内存的变化情况，以检查此类内存泄漏问题。

(2)鉴于多线程程序在实际中的广泛应用，下一步将扩展实现支持多线程的堆内存泄漏检测。

参考文献

[1] Purify[Z]. (2005-04-34). http://www-900.ibm.com/cn/software/rational/products/purifyplus/index.shtml.

[2] Robbins J. How I Use BoundsChecker[Z]. (2005-12-07). http://www.codeguru.com/cpp/v-s/debug/debuggers/article.php/c4417/.

[3] Mpatrol[Z]. (2005-12-12). http://sourceforge.net/projects/mpatrol/.

[4] ISO/IEC 14882. Programming Languages-C++[S]. 1998.

[5] Cormen T, Leiserson C, Rivest R. Introduction to Algorithms[M]. [S. l.]: MIT Press, 1990.

~~~~~~

(上接第 96 页)

5　结束语

数组终写关系分析的研究尚处于起步阶段，本文算法只适用于完美循环嵌套，对于非完美循环嵌套，还需要进一步的分析和研究。目前，数组终写关系分析的结果仅仅用来实现了精确数据收集，只回收在计算过程中被修改的数据，消除了并行化程序中数据收集过程中的一部分冗余通信。经过进一步的讨论，发现数组终写关系分析的结果同样适用于产生精确数据分布代码，只分布上一次计算中被修改的数据，消除计算前数据分布过程中的通信冗余。下一步工作将展开这两方面内容的研究。

参考文献

[1] 龚雪容, 生拥宏, 沈亚楠. 程序并行化中数据收集代码自动生成算法研究[J]. 计算机应用, 2006, 26(10): 2473-2475.

[2] Ferner C S. Paraguin Compiler Version 1.0 User Manual [EB/OL]. (2002-11-22). http://people.uncw.edu/cferner/Paraguin/userman.pdf.

[3] 张　平. 并行化编译器中并行程序自动生成和性能优化技术研究[D]. 郑州: 解放军信息工程大学, 2006.

[4] 杜　澎. 数据分布和收集代码自动生成及优化技术研究[D]. 郑州: 解放军信息工程大学, 2007.

[5] Maydan D E, Amarasinghe S P. Array Data-flow Analysis and Its Use in Array Privatization[C]//Proceedings of ACM Conference on Principles of Programming Languages. [S. l.]: ACM Press, 1993: 2-15.

[6] Schrijiver A. Theory of Linear and Integer Programming[M]. [S. l.]: Wiley Chichester Press, 1986: 36-43.
~~~~~~