**如何在linux下检测内存泄漏**

本文针对 linux 下的 C++ 程序的内存泄漏的检测方法及其实现进行探讨。其中包括 C++ 中的 new 和 delete 的基本原理，内存检测子系统的实现原理和具体方法，以及内存泄漏检测的高级话题。作为内存检测子系统实现的一部分，提供了一个具有更好的使用特性的互斥体（Mutex）类。

6https://dw1.s81c.com/developerworks/i/v17/dw-cmts-arrow.png [评论](https://www.ibm.com/developerworks/cn/linux/l-mleak/#icomments)

[洪琨](https://www.ibm.com/developerworks/cn/linux/l-mleak/#authorN10017) ([hcode@21cn.com](mailto:hcode@21cn.com?subject=%E5%A6%82%E4%BD%95%E5%9C%A8linux%E4%B8%8B%E6%A3%80%E6%B5%8B%E5%86%85%E5%AD%98%E6%B3%84%E6%BC%8F&cc=)), C++程序员

2003 年 7 月 26 日

* [expand](https://www.ibm.com/developerworks/cn/linux/l-mleak/#toggle)**内容**



在 IBM Bluemix 云平台上开发并部署您的下一个应用。

[**开始您的试用**](https://developer.ibm.com/sso/bmregistration?lang=zh_CN&ca=dwchina-_-bluemix-_-l-mleak-_-sidebar)

**1．开发背景**

在 windows 下使用 VC 编程时，我们通常需要 DEBUG 模式下运行程序，而后调试器将在退出程序时，打印出程序运行过程中在堆上分配而没有释放的内存信息，其中包括代码文件名、行号以及内存大小。该功能是 MFC Framework 提供的内置机制，封装在其类结构体系内部。

在 linux 或者 unix 下，我们的 C++ 程序缺乏相应的手段来检测内存信息，而只能使用 top 指令观察进程的动态内存总额。而且程序退出时，我们无法获知任何内存泄漏信息。为了更好的辅助在 linux 下程序开发，我们在我们的类库项目中设计并实现了一个内存检测子系统。下文将简述 C++ 中的 new 和 delete 的基本原理，并讲述了内存检测子系统的实现原理、实现中的技巧，并对内存泄漏检测的高级话题进行了讨论。

[**回页首**](https://www.ibm.com/developerworks/cn/linux/l-mleak/#ibm-pcon)

**2．New和delete的原理**

当我们在程序中写下 new 和 delete 时，我们实际上调用的是 C++ 语言内置的 new operator 和 delete operator。所谓语言内置就是说我们不能更改其含义，它的功能总是一致的。以 new operator 为例，它总是先分配足够的内存，而后再调用相应的类型的构造函数初始化该内存。而 delete operator 总是先调用该类型的析构函数，而后释放内存（图1）。我们能够施加影响力的事实上就是 new operator 和 delete operator 执行过程中分配和释放内存的方法。

new operator 为分配内存所调用的函数名字是 operator new，其通常的形式是 void \* operator new(size\_t size); 其返回值类型是 void\*，因为这个函数返回一个未经处理（raw）的指针，未初始化的内存。参数 size 确定分配多少内存，你能增加额外的参数重载函数 operator new，但是第一个参数类型必须是 size\_t。

delete operator 为释放内存所调用的函数名字是 operator delete，其通常的形式是 void operator delete(void \*memoryToBeDeallocated)；它释放传入的参数所指向的一片内存区。

这里有一个问题，就是当我们调用 new operator 分配内存时，有一个 size 参数表明需要分配多大的内存。但是当调用 delete operator 时，却没有类似的参数，那么 delete operator 如何能够知道需要释放该指针指向的内存块的大小呢？答案是：对于系统自有的数据类型，语言本身就能区分内存块的大小，而对于自定义数据类型（如我们自定义的类），则 operator new 和 operator delete 之间需要互相传递信息。

当我们使用 operator new 为一个自定义类型对象分配内存时，实际上我们得到的内存要比实际对象的内存大一些，这些内存除了要存储对象数据外，还需要记录这片内存的大小，此方法称为 cookie。这一点上的实现依据不同的编译器不同。（例如 MFC 选择在所分配内存的头部存储对象实际数据，而后面的部分存储边界标志和内存大小信息。g++ 则采用在所分配内存的头 4 个自己存储相关信息，而后面的内存存储对象实际数据。）当我们使用 delete operator 进行内存释放操作时，delete operator 就可以根据这些信息正确的释放指针所指向的内存块。

以上论述的是对于单个对象的内存分配/释放，当我们为数组分配/释放内存时，虽然我们仍然使用 new operator 和 delete operator，但是其内部行为却有不同：new operator 调用了operator new 的数组版的兄弟－ operator new[]，而后针对每一个数组成员调用构造函数。而 delete operator 先对每一个数组成员调用析构函数，而后调用 operator delete[] 来释放内存。需要注意的是，当我们创建或释放由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的大小，也使用了编译器相关的 cookie 技术。

综上所述，如果我们想检测内存泄漏，就必须对程序中的内存分配和释放情况进行记录和分析，也就是说我们需要重载 operator new/operator new[];operator delete/operator delete[] 四个全局函数，以截获我们所需检验的内存操作信息。

[**回页首**](https://www.ibm.com/developerworks/cn/linux/l-mleak/#ibm-pcon)

**3．内存检测的基本实现原理**

上文提到要想检测内存泄漏，就必须对程序中的内存分配和释放情况进行记录，所能够采取的办法就是重载所有形式的operator new 和 operator delete，截获 new operator 和 delete operator 执行过程中的内存操作信息。下面列出的就是重载形式

void\* operator new( size\_t nSize, char\* pszFileName, int nLineNum )

void\* operator new[]( size\_t nSize, char\* pszFileName, int nLineNum )

void operator delete( void \*ptr )

void operator delete[]( void \*ptr )

我们为 operator new 定义了一个新的版本，除了必须的 size\_t nSize 参数外，还增加了文件名和行号，这里的文件名和行号就是这次 new operator 操作符被调用时所在的文件名和行号，这个信息将在发现内存泄漏时输出，以帮助用户定位泄漏具体位置。对于 operator delete，因为无法为之定义新的版本，我们直接覆盖了全局的 operator delete 的两个版本。

在重载的 operator new 函数版本中，我们将调用全局的 operator new 的相应的版本并将相应的 size\_t 参数传入，而后，我们将全局 operator new 返回的指针值以及该次分配所在的文件名和行号信息记录下来，这里所采用的数据结构是一个 STL 的 map，以指针值为 key 值。当 operator delete 被调用时，如果调用方式正确的话（调用方式不正确的情况将在后面详细描述），我们就能以传入的指针值在 map 中找到相应的数据项并将之删除，而后调用 free 将指针所指向的内存块释放。当程序退出的时候，map 中的剩余的数据项就是我们企图检测的内存泄漏信息－－已经在堆上分配但是尚未释放的分配信息。

以上就是内存检测实现的基本原理，现在还有两个基本问题没有解决：

1) 如何取得内存分配代码所在的文件名和行号，并让 new operator 将之传递给我们重载的 operator new。

2) 我们何时创建用于存储内存数据的 map 数据结构，如何管理，何时打印内存泄漏信息。

先解决问题1。首先我们可以利用 C 的预编译宏 \_\_FILE\_\_ 和 \_\_LINE\_\_，这两个宏将在编译时在指定位置展开为该文件的文件名和该行的行号。而后我们需要将缺省的全局 new operator 替换为我们自定义的能够传入文件名和行号的版本，我们在子系统头文件 MemRecord.h 中定义：

#define DEBUG\_NEW new(\_\_FILE\_\_, \_\_LINE\_\_ )

而后在所有需要使用内存检测的客户程序的所有的 cpp 文件的开头加入

#include "MemRecord.h"

#define new DEBUG\_NEW

就可以将客户源文件中的对于全局缺省的 new operator 的调用替换为 new (\_\_FILE\_\_,\_\_LINE\_\_) 调用，而该形式的new operator将调用我们的operator new (size\_t nSize, char\* pszFileName, int nLineNum)，其中 nSize 是由 new operator 计算并传入的，而 new 调用点的文件名和行号是由我们自定义版本的 new operator 传入的。我们建议在所有用户自己的源代码文件中都加入上述宏，如果有的文件中使用内存检测子系统而有的没有，则子系统将可能因无法监控整个系统而输出一些泄漏警告。

再说第二个问题。我们用于管理客户信息的这个 map 必须在客户程序第一次调用 new operator 或者 delete operator 之前被创建，而且在最后一个 new operator 和 delete operator 调用之后进行泄漏信息的打印，也就是说它需要先于客户程序而出生，而在客户程序退出之后进行分析。能够包容客户程序生命周期的确有一人--全局对象（appMemory）。我们可以设计一个类来封装这个 map 以及这对它的插入删除操作，然后构造这个类的一个全局对象（appMemory），在全局对象（appMemory）的构造函数中创建并初始化这个数据结构，而在其析构函数中对数据结构中剩余数据进行分析和输出。Operator new 中将调用这个全局对象（appMemory）的 insert 接口将指针、文件名、行号、内存块大小等信息以指针值为 key 记录到 map 中，在 operator delete 中调用 erase 接口将对应指针值的 map 中的数据项删除，注意不要忘了对 map 的访问需要进行互斥同步，因为同一时间可能会有多个线程进行堆上的内存操作。

好啦，内存检测的基本功能已经具备了。但是不要忘了，我们为了检测内存泄漏，在全局的 operator new 增加了一层间接性，同时为了保证对数据结构的安全访问增加了互斥，这些都会降低程序运行的效率。因此我们需要让用户能够方便的 enable 和 disable 这个内存检测功能，毕竟内存泄漏的检测应该在程序的调试和测试阶段完成。我们可以使用条件编译的特性，在用户被检测文件中使用如下宏定义：

#include "MemRecord.h"

#if defined( MEM\_DEBUG )

#define new DEBUG\_NEW

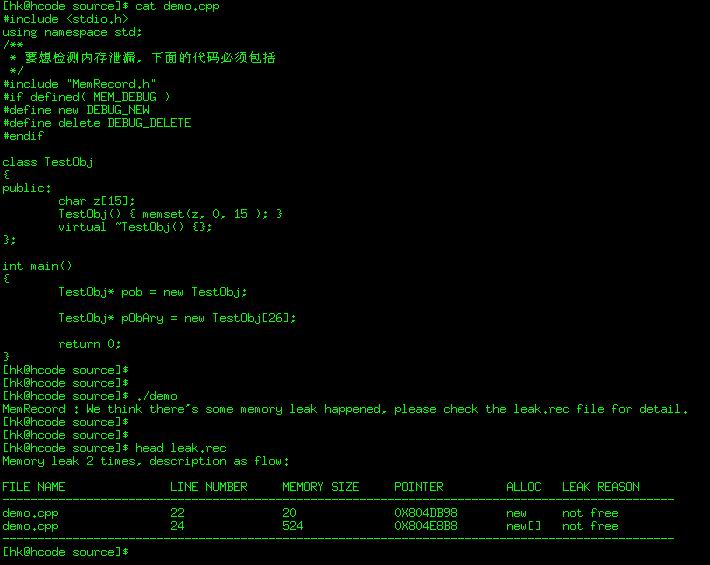
#endif

当用户需要使用内存检测时，可以使用如下命令对被检测文件进行编译

g++ -c -DMEM\_DEBUG xxxxxx.cpp

就可以 enable 内存检测功能，而用户程序正式发布时，可以去掉 -DMEM\_DEBUG 编译开关来 disable 内存检测功能，消除内存检测带来的效率影响。

图2所示为使用内存检测功能后，内存泄漏代码的执行以及检测结果

  
图2

[**回页首**](https://www.ibm.com/developerworks/cn/linux/l-mleak/#ibm-pcon)

**4．错误方式删除带来的问题**

以上我们已经构建了一个具备基本内存泄漏检测功能的子系统，下面让我们来看一下关于内存泄漏方面的一些稍微高级一点的话题。

首先，在我们编制 c++ 应用时，有时需要在堆上创建单个对象，有时则需要创建对象的数组。关于 new 和 delete 原理的叙述我们可以知道，对于单个对象和对象数组来说，内存分配和删除的动作是大不相同的，我们应该总是正确的使用彼此搭配的 new 和 delete 形式。但是在某些情况下，我们很容易犯错误，比如如下代码：

class Test {};

……

Test\* pAry = new Test[10];//创建了一个拥有 10 个 Test 对象的数组

Test\* pObj = new Test;//创建了一个单对象

……

delete []pObj;//本应使用单对象形式 delete pObj 进行内存释放，却错误的使用了数

//组形式

delete pAry;//本应使用数组形式 delete []pAry 进行内存释放，却错误的使用了单对

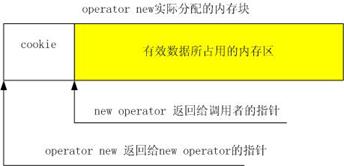
//象的形式

不匹配的 new 和 delete 会导致什么问题呢？C++ 标准对此的解答是"未定义"，就是说没有人向你保证会发生什么，但是有一点可以肯定：大多不是好事情--在某些编译器形成的代码中，程序可能会崩溃，而另外一些编译器形成的代码中，程序运行可能毫无问题，但是可能导致内存泄漏。

既然知道形式不匹配的 new 和 delete 会带来的问题，我们就需要对这种现象进行毫不留情的揭露，毕竟我们重载了所有形式的内存操作 operator new，operator new[]，operator delete，operator delete[]。

我们首先想到的是，当用户调用特定方式（单对象或者数组方式）的 operator new 来分配内存时，我们可以在指向该内存的指针相关的数据结构中，增加一项用于描述其分配方式。当用户调用不同形式的 operator delete 的时候，我们在 map 中找到与该指针相对应的数据结构，然后比较分配方式和释放方式是否匹配，匹配则在 map 中正常删除该数据结构，不匹配则将该数据结构转移到一个所谓 "ErrorDelete" 的 list 中，在程序最终退出的时候和内存泄漏信息一起打印。

上面这种方法是最顺理成章的，但是在实际应用中效果却不好。原因有两个，第一个原因我们上面已经提到了：当 new 和 delete 形式不匹配时，其结果"未定义"。如果我们运气实在太差--程序在执行不匹配的 delete 时崩溃了，我们的全局对象（appMemory）中存储的数据也将不复存在，不会打印出任何信息。第二个原因与编译器相关，前面提到过，当编译器处理自定义数据类型或者自定义数据类型数组的 new 和 delete 操作符的时候，通常使用编译器相关的 cookie 技术。这种 cookie 技术在编译器中可能的实现方式是：new operator 先计算容纳所有对象所需的内存大小，而后再加上它为记录 cookie 所需要的内存量，再将总容量传给operator new 进行内存分配。当 operator new 返回所需的内存块后，new operator 将在调用相应次数的构造函数初始化有效数据的同时，记录 cookie 信息。而后将指向有效数据的指针返回给用户。也就是说我们重载的 operator new 所申请到并记录下来的指针与 new operator 返回给调用者的指针不一定一致（图3）。当调用者将 new operator 返回的指针传给 delete operator 进行内存释放时，如果其调用形式相匹配，则相应形式的 delete operator 会作出相反的处理，即调用相应次数的析构函数，再通过指向有效数据的指针位置找出包含 cookie 的整块内存地址，并将其传给 operator delete 释放内存。如果调用形式不匹配，delete operator 就不会做上述运算，而直接将指向有效数据的指针（而不是真正指向整块内存的指针）传入 operator delete。因为我们在 operator new 中记录的是我们所分配的整块内存的指针，而现在传入 operator delete 的却不是，所以就无法在全局对象（appMemory）所记录的数据中找到相应的内存分配信息。

  
图3

综上所述，当 new 和 delete 的调用形式不匹配时，由于程序有可能崩溃或者内存子系统找不到相应的内存分配信息，在程序最终打印出 "ErrorDelete" 的方式只能检测到某些"幸运"的不匹配现象。但我们总得做点儿什么，不能让这种危害极大的错误从我们眼前溜走，既然不能秋后算帐，我们就实时输出一个 warning 信息来提醒用户。什么时候抛出一个 warning 呢？很简单，当我们发现在 operator delete 或 operator delete[] 被调用的时候，我们无法在全局对象（appMemory）的 map 中找到与传入的指针值相对应的内存分配信息，我们就认为应该提醒用户。

既然决定要输出warning信息，那么现在的问题就是：我们如何描述我们的warning信息才能更便于用户定位到不匹配删除错误呢？答案：在 warning 信息中打印本次 delete 调用的文件名和行号信息。这可有点困难了，因为对于 operator delete 我们不能向对象 operator new 一样做出一个带附加信息的重载版本，我们只能在保持其接口原貌的情况下，重新定义其实现，所以我们的 operator delete 中能够得到的输入只有指针值。在 new/delete 调用形式不匹配的情况下，我们很有可能无法在全局对象（appMemory）的 map 中找到原来的 new 调用的分配信息。怎么办呢？万不得已，只好使用全局变量了。我们在检测子系统的实现文件中定义了两个全局变量（DELETE\_FILE,DELETE\_LINE）记录 operator delete 被调用时的文件名和行号，同时为了保证并发的 delete 操作对这两个变量访问同步，还使用了一个 mutex（至于为什么是 CCommonMutex 而不是一个 pthread\_mutex\_t，在"实现上的问题"一节会详细论述，在这里它的作用就是一个 mutex）。

char DELETE\_FILE[ FILENAME\_LENGTH ] = {0};

int DELETE\_LINE = 0;

CCommonMutex globalLock;

而后，在我们的检测子系统的头文件中定义了如下形式的 DEBUG\_DELETE

extern char DELETE\_FILE[ FILENAME\_LENGTH ];

extern int DELETE\_LINE;

extern CCommonMutex globalLock;//在后面解释

#define DEBUG\_DELETE globalLock.Lock(); \

if (DELETE\_LINE != 0) BuildStack();\ （//见第六节解释）

strncpy( DELETE\_FILE, \_\_FILE\_\_,FILENAME\_LENGTH - 1 );\

DELETE\_FILE[ FILENAME\_LENGTH - 1 ]= '\0'; \

DELETE\_LINE = \_\_LINE\_\_; \

delete

在用户被检测文件中原来的宏定义中添加一条：

#include "MemRecord.h"

#if defined( MEM\_DEBUG )

#define new DEBUG\_NEW

#define delete DEBUG\_DELETE

#endif

这样，在用户被检测文件调用 delete operator 之前，将先获得互斥锁，然后使用调用点文件名和行号对相应的全局变量（DELETE\_FILE,DELETE\_LINE）进行赋值，而后调用 delete operator。当 delete operator 最终调用我们定义的 operator delete 的时候，在获得此次调用的文件名和行号信息后，对文件名和行号全局变量（DELETE\_FILE,DELETE\_LINE）重新初始化并打开互斥锁，让下一个挂在互斥锁上的 delete operator 得以执行。

在对 delete operator 作出如上修改以后，当我们发现无法经由 delete operator 传入的指针找到对应的内存分配信息的时候，就打印包括该次调用的文件名和行号的 warning。

天下没有十全十美的事情，既然我们提供了一种针对错误方式删除的提醒方法，我们就需要考虑以下几种异常情况：

1． 用户使用的第三方库函数中有内存分配和释放操作。或者用户的被检测进程中进行内存分配和释放的实现文件没有使用我们的宏定义。 由于我们替换了全局的 operator delete，这种情况下的用户调用的 delete 也会被我们截获。用户并没有使用我们定义的DEBUG\_NEW 宏，所以我们无法在我们的全局对象（appMemory）数据结构中找到对应的内存分配信息，但是由于它也没有使用DEBUG\_DELETE，我们为 delete 定义的两个全局 DELETE\_FILE 和 DELETE\_LINE 都不会有值，因此可以不打印 warning。

2． 用户的一个实现文件调用了 new 进行内存分配工作，但是该文件并没有使用我们定义的 DEBUG\_NEW 宏。同时用户的另一个实现文件中的代码负责调用 delete 来删除前者分配的内存，但不巧的是，这个文件使用了 DEBUG\_DELETE 宏。这种情况下内存检测子系统会报告 warning，并打印出 delete 调用的文件名和行号。

3． 与第二种情况相反，用户的一个实现文件调用了 new 进行内存分配工作，并使用我们定义的 DEBUG\_NEW 宏。同时用户的另一个实现文件中的代码负责调用 delete 来删除前者分配的内存，但该文件没有使用 DEBUG\_DELETE 宏。这种情况下，因为我们能够找到这个内存分配的原始信息，所以不会打印 warning。

4． 当出现嵌套 delete（定义可见"实现上的问题"）的情况下，以上第一和第三种情况都有可能打印出不正确的 warning 信息，详细分析可见"实现上的问题"一节。

你可能觉得这样的 warning 太随意了，有误导之嫌。怎么说呢？作为一个检测子系统，对待有可能的错误我们所采取的原则是：宁可误报，不可漏报。请大家"有则改之，无则加勉"。

[**回页首**](https://www.ibm.com/developerworks/cn/linux/l-mleak/#ibm-pcon)

**5．动态内存泄漏信息的检测**

上面我们所讲述的内存泄漏的检测能够在程序整个生命周期结束时，打印出在程序运行过程中已经在堆上分配但是没有释放的内存分配信息，程序员可以由此找到程序中"显式"的内存泄漏点并加以改正。但是如果程序在结束之前能够将自己所分配的所有内存都释放掉，是不是就可以说这个程序不存在内存泄漏呢？答案：否！在编程实践中，我们发现了另外两种危害性更大的"隐式"内存泄漏，其表现就是在程序退出时，没有任何内存泄漏的现象，但是在程序运行过程中，内存占用量却不断增加，直到使整个系统崩溃。

1． 程序的一个线程不断分配内存，并将指向内存的指针保存在一个数据存储中（如 list），但是在程序运行过程中，一直没有任何线程进行内存释放。当程序退出的时候，该数据存储中的指针值所指向的内存块被依次释放。

2． 程序的N个线程进行内存分配，并将指针传递给一个数据存储，由M个线程从数据存储进行数据处理和内存释放。由于 N 远大于M，或者M个线程数据处理的时间过长，导致内存分配的速度远大于内存被释放的速度。但是在程序退出的时候，数据存储中的指针值所指向的内存块被依次释放。

之所以说他危害性更大，是因为很不容易这种问题找出来，程序可能连续运行几个十几个小时没有问题，从而通过了不严密的系统测试。但是如果在实际环境中 7×24 小时运行，系统将不定时的崩溃，而且崩溃的原因从 log 和程序表象上都查不出原因。

为了将这种问题也挑落马下，我们增加了一个动态检测模块 MemSnapShot，用于在程序运行过程中，每隔一定的时间间隔就对程序当前的内存总使用情况和内存分配情况进行统计，以使用户能够对程序的动态内存分配状况进行监视。

当客户使用 MemSnapShot 进程监视一个运行中的进程时，被监视进程的内存子系统将把内存分配和释放的信息实时传送给MemSnapShot。MemSnapShot 则每隔一定的时间间隔就对所接收到的信息进行统计，计算该进程总的内存使用量，同时以调用new进行内存分配的文件名和行号为索引值，计算每个内存分配动作所分配而未释放的内存总量。这样一来，如果在连续多个时间间隔的统计结果中，如果某文件的某行所分配的内存总量不断增长而始终没有到达一个平衡点甚至回落，那它一定是我们上面所说到的两种问题之一。

在实现上，内存检测子系统的全局对象（appMemory）的构造函数中以自己的当前 PID 为基础 key 值创建一个消息队列，并在operator new 和 operator delete 被调用的时候将相应的信息写入消息队列。MemSnapShot 进程启动时需要输入被检测进程的 PID，而后通过该 PID 组装 key 值并找到被检测进程创建的消息队列，并开始读入消息队列中的数据进行分析统计。当得到operator new 的信息时，记录内存分配信息，当收到 operator delete 消息时，删除相应的内存分配信息。同时启动一个分析线程，每隔一定的时间间隔就计算一下当前的以分配而尚未释放的内存信息，并以内存的分配位置为关键字进行统计，查看在同一位置（相同文件名和行号）所分配的内存总量和其占进程总内存量的百分比。

图4 是一个正在运行的 MemSnapShot 程序，它所监视的进程的动态内存分配情况如图所示：

  
图四

在支持 MemSnapShot 过程中的实现上的唯一技巧是--对于被检测进程异常退出状况的处理。因为被检测进程中的内存检测子系统创建了用于进程间传输数据的消息队列，它是一个核心资源，其生命周期与内核相同，一旦创建，除非显式的进行删除或系统重启，否则将不被释放。

不错，我们可以在内存检测子系统中的全局对象（appMemory）的析构函数中完成对消息队列的删除，但是如果被检测进程非正常退出（CTRL+C，段错误崩溃等），消息队列可就没人管了。那么我们可以不可以在全局对象（appMemory）的构造函数中使用 signal 系统调用注册 SIGINT，SIGSEGV 等系统信号处理函数，并在处理函数中删除消息队列呢？还是不行，因为被检测进程完全有可能注册自己的对应的信号处理函数，这样就会替换我们的信号处理函数。最终我们采取的方法是利用 fork 产生一个孤儿进程，并利用这个进程监视被检测进程的生存状况，如果被检测进程已经退出（无论正常退出还是异常退出），则试图删除被检测进程所创建的消息队列。下面简述其实现原理：

在全局对象（appMemory）构造函数中，创建消息队列成功以后，我们调用 fork 创建一个子进程，而后该子进程再次调用 fork 创建孙子进程，并退出，从而使孙子进程变为一个"孤儿"进程（之所以使用孤儿进程是因为我们需要切断被检测进程与我们创建的进程之间的信号联系）。孙子进程利用父进程（被检测进程）的全局对象（appMemory）得到其 PID 和刚刚创建的消息队列的标识，并传递给调用 exec 函数产生的一个新的程序映象--MemCleaner。

MemCleaner 程序仅仅调用 kill(pid, 0);函数来查看被检测进程的生存状态，如果被检测进程不存在了（正常或者异常退出），则 kill 函数返回非 0 值，此时我们就动手清除可能存在的消息队列。

[**回页首**](https://www.ibm.com/developerworks/cn/linux/l-mleak/#ibm-pcon)

**6．实现上的问题：嵌套delete**

在"错误方式删除带来的问题"一节中，我们对 delete operator 动了个小手术--增加了两个全局变量（DELETE\_FILE,DELETE\_LINE）用于记录本次 delete 操作所在的文件名和行号，并且为了同步对全局变量（DELETE\_FILE,DELETE\_LINE）的访问，增加了一个全局的互斥锁。在一开始，我们使用的是 pthread\_mutex\_t，但是在测试中，我们发现 pthread\_mutex\_t 在本应用环境中的局限性。

例如如下代码：

class B {…};

class A {

public:

A() {m\_pB = NULL};

A(B\* pb) {m\_pB = pb;};

~A()

{

if (m\_pB != NULL)

行号1 delete m\_pB; //这句最要命

};

private:

class B\* m\_pB;

……

}

int main()

{

A\* pA = new A(new B);

……

行号2 delete pA;

}

在上述代码中，main 函数中的一句 delete pA 我们称之为"嵌套删除"，即我们 delete A 对象的时候，在A对象的析构执行了另一个 delete B 的动作。当用户使用我们的内存检测子系统时，delete pA 的动作应转化为以下动作：

上全局锁

全局变量（DELETE\_FILE,DELETE\_LINE）赋值为文件名和行号2

delete operator A

调用~A()

上全局锁

全局变量（DELETE\_FILE,DELETE\_LINE）赋值为文件名和行号1

delete operator B

调用~B()

返回~B()

调用operator delete B

记录全局变量（DELETE\_FILE,DELETE\_LINE）值1并清除全局变量（DELETE\_FILE,DELETE\_LINE）值

打开全局锁

返回operator delete B

返回delete operator B

返回~A()

调用 operator delete A

记录全局变量（DELETE\_FILE,DELETE\_LINE）值1并清除全局变量（DELETE\_FILE,DELETE\_LINE）值

打开全局锁

返回operator delete A

返回 delete operator A

在这一过程中，有两个技术问题，一个是 **mutex 的可重入问题**，一个是嵌套删除时 **对全局变量（DELETE\_FILE,DELETE\_LINE）**现场保护的问题。

所谓 **mutex 的可重入问题**，是指在同一个线程上下文中，连续对同一个 mutex 调用了多次 lock，然后连续调用了多次 unlock。这就是说我们的应用方式要求互斥锁有如下特性：

1． 要求在同一个线程上下文中，能够多次持有同一个互斥体。并且只有在同一线程上下文中调用相同次数的 unlock 才能放弃对互斥体的占有。

2． 对于不同线程上下文持有互斥体的企图，同一时间只有一个线程能够持有互斥体，并且只有在其释放互斥体之后，其他线程才能持有该互斥体。

Pthread\_mutex\_t 互斥体不具有以上特性，即使在同一上下文中，第二次调用 pthread\_mutex\_lock 将会挂起。因此，我们必须实现出自己的互斥体。在这里我们使用 semaphore 的特性实现了一个符合上述特性描述的互斥体 CCommonMutex（源代码见附件）。

为了支持特性 2，在这个 CCommonMutex 类中，封装了一个 semaphore，并在构造函数中令其资源值为 1，初始值为1。当调用 CCommonMutex::lock 接口时，调用 sem\_wait 得到 semaphore，使信号量的资源为 0 从而让其他调用 lock 接口的线程挂起。当调用接口 CCommonMutex::unlock 时，调用 sem\_post 使信号量资源恢复为 1，让其他挂起的线程中的一个持有信号量。

同时为了支持特性 1，在这个 CCommonMutex 增加了对于当前线程 pid 的判断和当前线程访问计数。当线程第一次调用 lock 接口时，我们调用 sem\_wait 的同时，记录当前的 Pid 到成员变量 m\_pid，并置访问计数为 1，同一线程（m\_pid == getpid()）其后的多次调用将只进行计数而不挂起。当调用 unlock 接口时，如果计数不为 1，则只需递减访问计数，直到递减访问计数为 1 才进行清除 pid、调用 sem\_post。（具体代码可见附件）

**嵌套删除时对全局变量（DELETE\_FILE,DELETE\_LINE）**现场保护的问题是指，上述步骤中在 A 的析构函数中调用 delete m\_pB 时，对全局变量（DELETE\_FILE,DELETE\_LINE）文件名和行号的赋值将覆盖主程序中调用 delete pA 时对全局变量（DELETE\_FILE,DELETE\_LINE）的赋值，造成了在执行 operator delete A 时，delete pA 的信息全部丢失。

要想对这些全局信息进行现场保护，最好用的就是堆栈了，在这里我们使用了 STL 提供的 stack 容器。在 DEBUG\_DELETE 宏定义中，对全局变量（DELETE\_FILE,DELETE\_LINE）赋值之前，我们先判断是否前面已经有人对他们赋过值了--观察行号变量是否等于 0，如果不为 0，则应该将已有的信息压栈（调用一个全局函数 BuildStack() 将当前的全局文件名和行号数据压入一个全局堆栈globalStack），而后再对全局变量（DELETE\_FILE,DELETE\_LINE）赋值，再调用 delete operator。而在内存子系统的全局对象（appMemory）提供的 erase 接口里面，如果判断传入的文件名和行号为 0，则说明我们所需要的数据有可能被嵌套删除覆盖了，所以需要从堆栈中弹出相应的数据进行处理。

现在嵌套删除中的问题基本解决了，但是当嵌套删除与 "错误方式删除带来的问题"一节的最后所描述的第一和第三种异常情况同时出现的时候，由于用户的 delete 调用没有通过我们定义的 DEBUG\_DELETE 宏，上述机制可能出现问题。其根本原因是我们利用stack 保留了经由我们的 DEBUG\_DELETE 宏记录的 delete 信息的现场，以便在 operator delete 和全局对象（appMemory）的 erase 接口中使用，但是用户的没经过 DEBUG\_DELETE 宏的 delete 操作却未曾进行压栈操作而直接调用了 operator delete，有可能将不属于这次操作的 delete 信息弹出，破坏了堆栈信息的顺序和有效性。那么，当我们因为无法找到这次及其后续的 delete 操作所对应的内存分配信息的时候，可能会打印出错误的 warning 信息。

[**回页首**](https://www.ibm.com/developerworks/cn/linux/l-mleak/#ibm-pcon)

**展望**

以上就是我们所实现的内存泄漏检测子系统的原理和技术方案，第一版的源代码在附件中，已经经过了较严格的系统测试。但是限于我们的 C++ 知识水平和编程功底，在实现过程中肯定还有没有注意到的地方甚至是缺陷，希望能够得到大家的指正，我的 email 是[hcode@21cn.com](mailto:hcode@21cn.com)。

在我们所实现的内存检测子系统基础上，可以继续搭建内存分配优化子系统，从而形成一个完整的内存子系统。一种内存分配优化子系统的实现方案是一次性分配大块的内存，并使用特定的数据结构管理之，当内存分配请求到来时，使用特定算法从这块大内存中划定所需的一块给用户使用，而用户使用完毕，在将其划为空闲内存。这种内存优化方式将内存分配释放转换为简单的数据处理，极大的减少了内存申请和释放所耗费的时间。