内存泄露解决方案调研

目录

[内存泄露解决方案调研 0](#_Toc478626280)

[1. DLL库 2](#_Toc478626281)

[1.1 DLL程序集 2](#_Toc478626282)

[1.1.1 C++ DLL 2](#_Toc478626283)

[1.1.2 C#程序集 3](#_Toc478626284)

[1.2 DLL加载方式 4](#_Toc478626285)

[1.2.1 DllImport 4](#_Toc478626286)

[1.2.2 Dynamic 5](#_Toc478626287)

[2. 内存泄露 6](#_Toc478626288)

[2.1 检查工具 6](#_Toc478626289)

[2.1.1 Visual Leak Debug 6](#_Toc478626290)

[2.1.2 DevPartner 7](#_Toc478626291)

[2.2 解决方案 9](#_Toc478626292)

[2.2.1 测试案例 9](#_Toc478626293)

[2.2.2 测试代码 12](#_Toc478626294)

# 1. DLL库

## 1.1 DLL程序集

### 1.1.1 C++ DLL

1、DLL\_PROCESS\_ATTACH

DLL文件映射到进程的地址空间：静态链接DllImport和动态链接的LoadLibrary或者LoadLibraryEx。DLL文件被映射到进程的地址空间时，系统调用该DLL的DllMain函数，传递的fdwReason参数为DLL\_PROCESS\_ATTACH。再次调用LoadLibrary或者LoadLibraryEx，操作系统只增加DLL的使用次数，不会再用DLL\_PROCESS\_ATTACH调用DLL的DllMain函数。

2、DLL\_PROCESS\_DETACH

DLL被从进程的地址空间解除映射时，系统调用了它的DllMain，传递的fdwReason值是DLL\_PROCESS\_DETACH，应该执行进程相关的清理工作。

**DLL被从进程的地址空间解除映射方式：**

**A，FreeLibrary解除DLL映射**

**B，进程结束而解除DLL映射，在进程结束前还没有解除DLL的映射，进程结束后会解除DLL映射**

当用DLL\_PROCESS\_ATTACH调用DLL的DllMain函数时，如果返回FALSE，说明没有初始化成功，系统仍会用DLL\_PROCESS\_DETACH调用DLL的DllMain函数。

3、DLL\_THREAD\_ATTACH

当进程创建一线程时，系统查看当前映射到进程地址空间中的所有DLL文件映像，并用值DLL\_THREAD\_ATTACH调用DLL的DllMain函数。

新创建的线程负责执行这次的DLL的DllMain函数，只有当所有的DLL都处理完这一通知后，系统才允许进程开始执行它的线程函数。

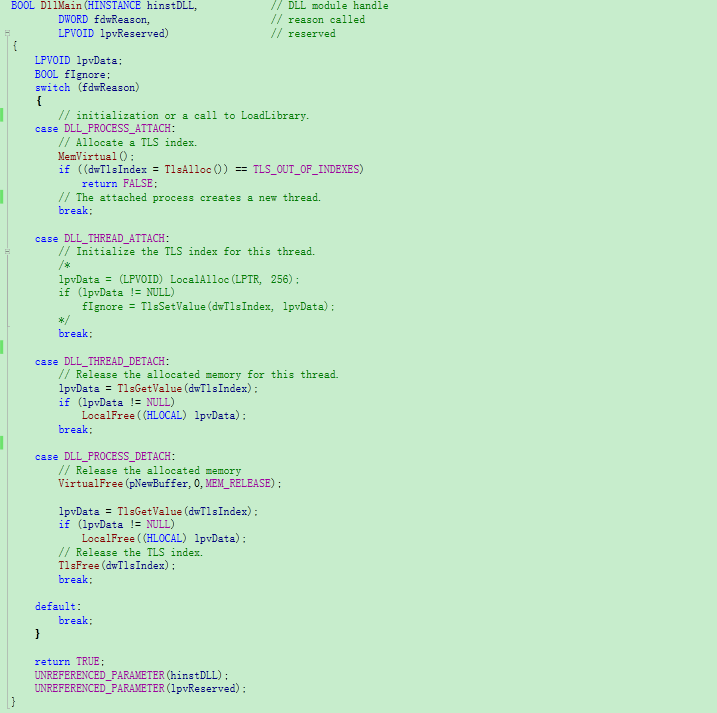
进程中的每次建立线程，都会用值DLL\_THREAD\_ATTACH调用DllMain函数。

4、DLL\_THREAD\_DETACH

如果线程调用了ExitThread来结束线程（线程函数返回时，系统也会自动调用ExitThread），系统查看当前映射到进程空间中的所有DLL文件映像，并用DLL\_THREAD\_DETACH来调用DllMain函数，通知所有的DLL去执行线程级的清理工作。

5、DisableThreadLibraryCalls函数

用于使指定DLL的DLL\_THREAD\_ATTACH和DLL\_THREAD\_DETACH通知无效，对于一个拥有很多DLL的多线程应用程序而已，如果这些DLL频繁地创建和销毁线程，而且这些DLL不需要线程创建和销毁通知，则在DLL中使用DisableThreadLibraryCalls函数将能够起到优化应用程序的作用。



### 1.1.2 C#程序集

1、进程

一个进程无法直接访问另一个进程的数据，一个进程运行的失败也不会影响其他进程的运行，Windows系统就是利用进程把工作划分为多个独立的区域的。进程可以理解为一个程序的基本边界。

2、程序集

A, 标示（名称、版本、区域、公钥）

B, 导出的类型

C, 该程序集所依赖的其他程序集

D, 运行所需的安全权限

程序集是自我描述的安装单元，由一个或多个文件组成，一个程序集可以使一个包含元数据的DLL或EXE，也可以由多个文件组成，例如资源文件、元数据、DLL和EXE。由.net编译生成的exe文件或dll文件就是程序集。程序集是.net框架应用程序的生成块，包含编译好的代码的逻辑单元，创建每一个项目文件都会产生一个程序集dll。

.Net中加载程序集方法:

A,隐式加载#pragma comment (lib, "\*.lib")

B,AppDomain.Load

C,Assembly.Load

D,Assembly.LoadFrom

E,Assembly.LoadFile

G,统VC动态库的调用:Dllimport,Loadlibrary



3、应用程序域

AppDomain类似与系统的进程，AppDomain是由CLR进行创建。一个给定的AppDomain必须驻留在一个操作系统的进程中，而一个给定的进程可以寄宿多个AppDomain。

对于应用AppDomain的资源被加载，一直在内存中，卸载AppDomain资源是唯一卸载模块或者程序集的途径，卸载AppDomain资源也是回收类型静态字段所占内存的唯一方式。

透明代理：

在上下文的接口当中存在着一个消息接收器负责检测拦截和处理信息，当对象是MarshalByRefObject的子类的时候，CLR将会建立透明代理，实现对象与消息之间的转换。

应用程序域是CLR中资源的边界，应用程序域中的对象不能被外界的对象所访问。而MarshalByRefObject 的功能就是允许在支持远程处理的应用程序中跨应用程序域边界访问对象。



## 1.2 DLL加载方式

### 1.2.1 DllImport

调用DLL中的非托管函数一般方法

DllImportAttribute 属性：

EntryPoint入口点用于标识函数在 DLL 中的位置

CharSet 指示用在入口点中的字符集，如：CharSet=CharSet.Ansi；

SetLastError 指示方法是否保留 Win32“上一错误”，如：SetLastError=true；

ExactSpelling 指示 EntryPoint 是否必须与指示的入口点的拼写完全匹配，如：ExactSpelling=false；

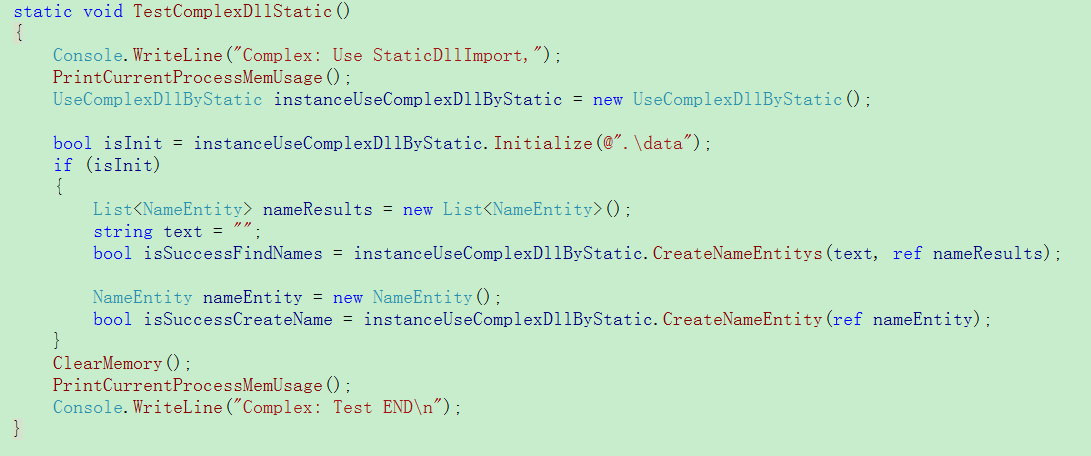
PreserveSig指示方法的签名应当被保留还是被转换， 如：PreserveSig=true；

CallingConvention指示入口点的调用约定， 如：CallingConvention=CallingConvention.Winapi；

A，DllImport导入函数：



B，DllImport方式调用Dll

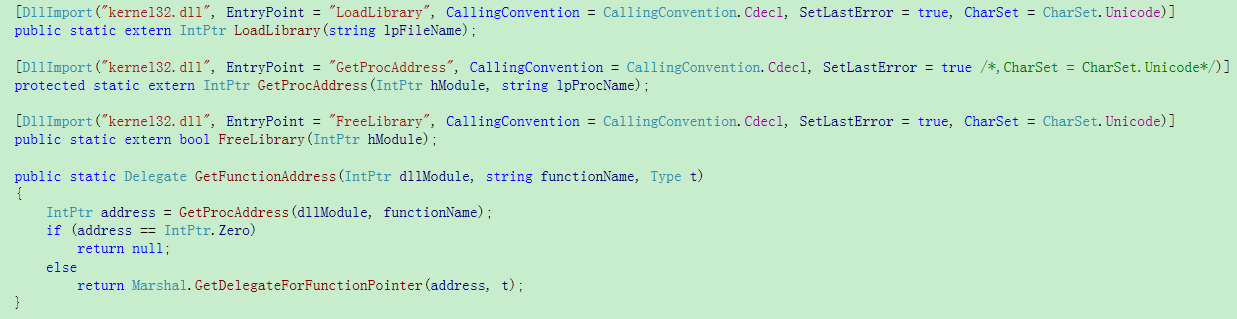


### 1.2.2 Dynamic

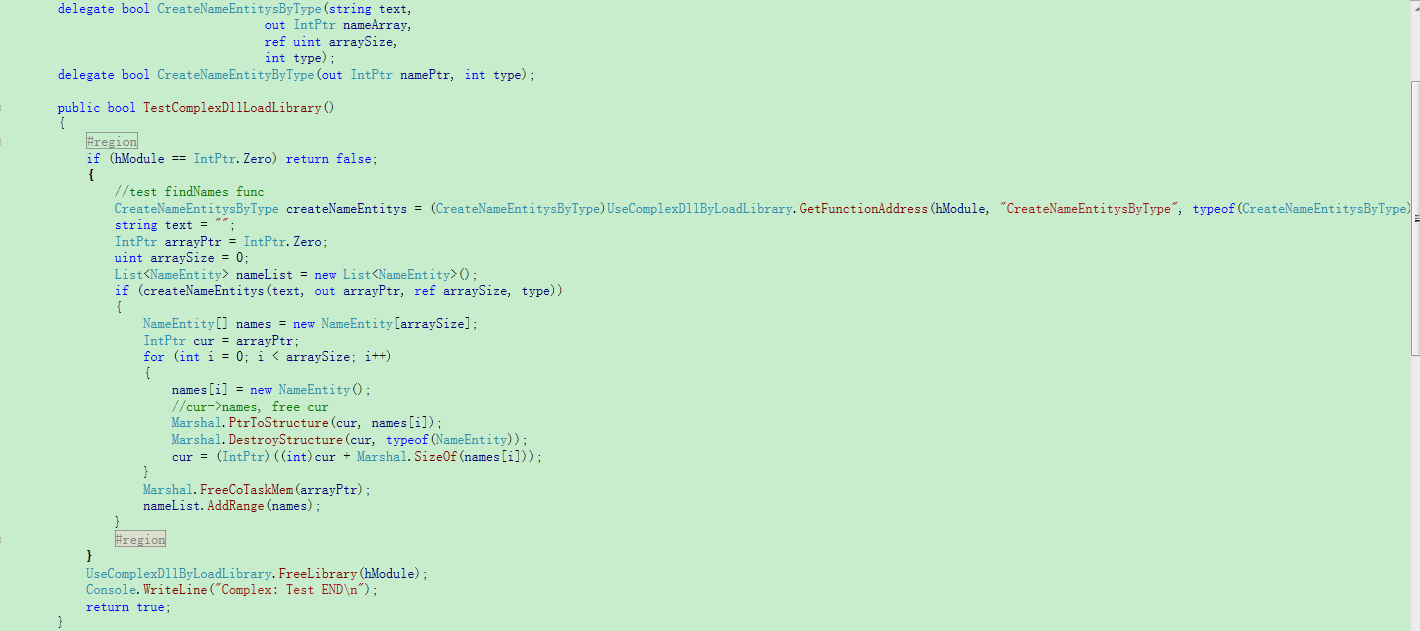
动态装载、调用DLL中的非托管函数(LoadLibrary,FreeLibrary)

DllImport调用DLL中的非托管函数，但是这个是全局的函数，假若DLL中的非托管函数有一个静态变量S，每次调用这个函数的时候，静态变量S就自动加1。结果，当需要重新计数时，就不能得出想要的结果。

A，导入LoadLibrary，FreeLibrary，GetProcAddress函数



B，测试使用Dll函数



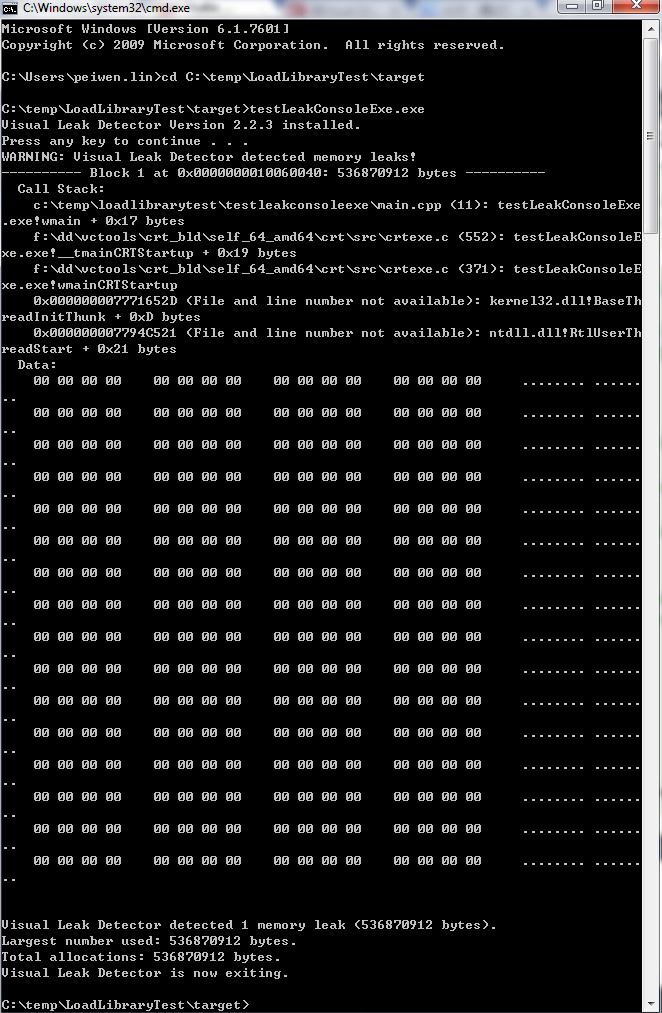
# 2. 内存泄露

## 2.1 检查工具

### 2.1.1 Visual Leak Debug

Visual Leak Detector (VLD)

只需要在需要进行内存泄漏检测的的主程序代码中加入#include “vld.h”， 编译好的Lib文件放入默认的库目录，然后在Visual C++用Debug模式编译运行程序，程序运行结束后便可在Visual C++的输出窗口看到内存泄露检测结果。检测报告直观，显示泄漏内存的内容。



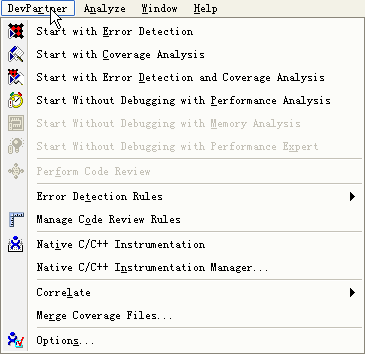
### 2.1.2 DevPartner

DevPartner Studio Professional Edition含有静态代码分析、错误检测、性能、覆盖率和内存分析以及性能**专家功能，用于 Visu**al Studio 2012、Visual Studio 2010、Visual Studio 2008 和 Visual Studio 2005。

主页：<http://www.borland.com/products/devpartner/read/>

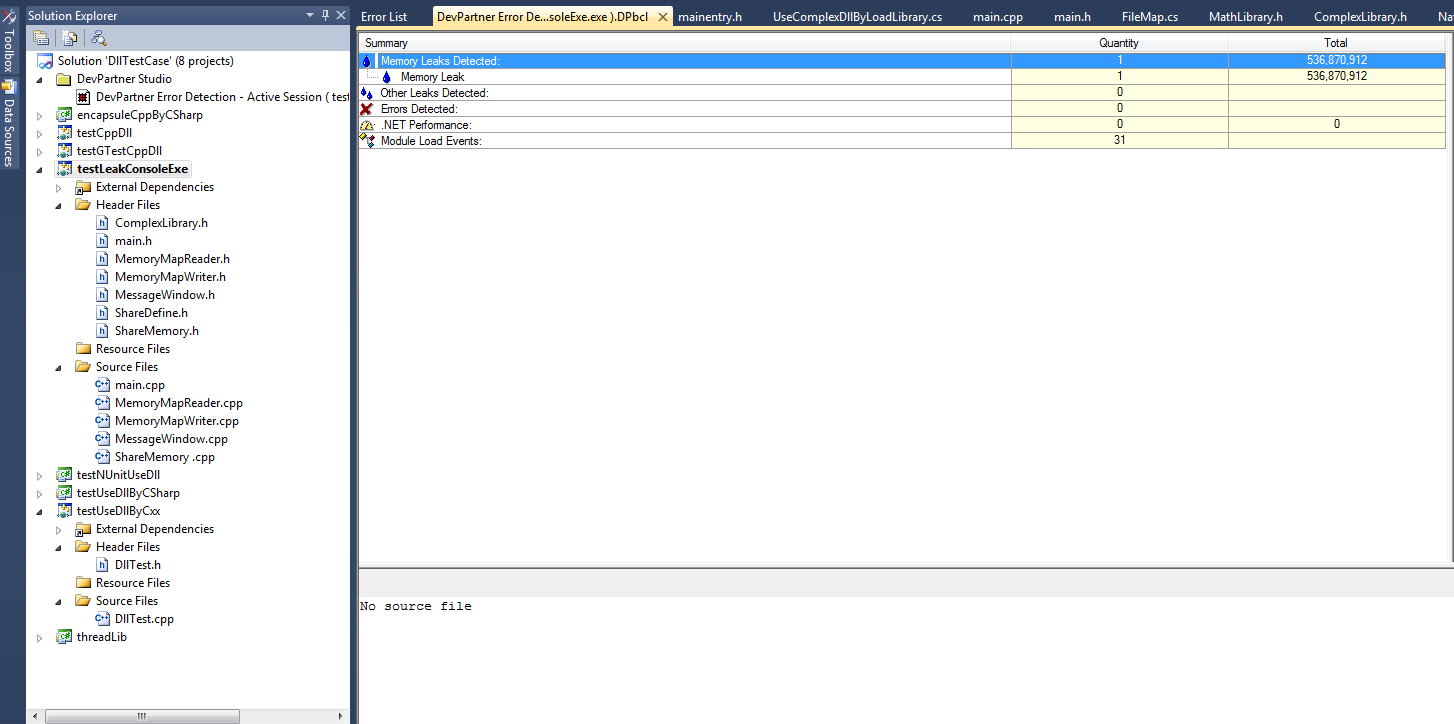


下载安装DevPartner Studio Professional Edition 11试用版，记得需要安装Micro Focus License Manager。在Visual Studio 2008菜单栏上多出了一个"**DevPartner**"项，如下图所示选：



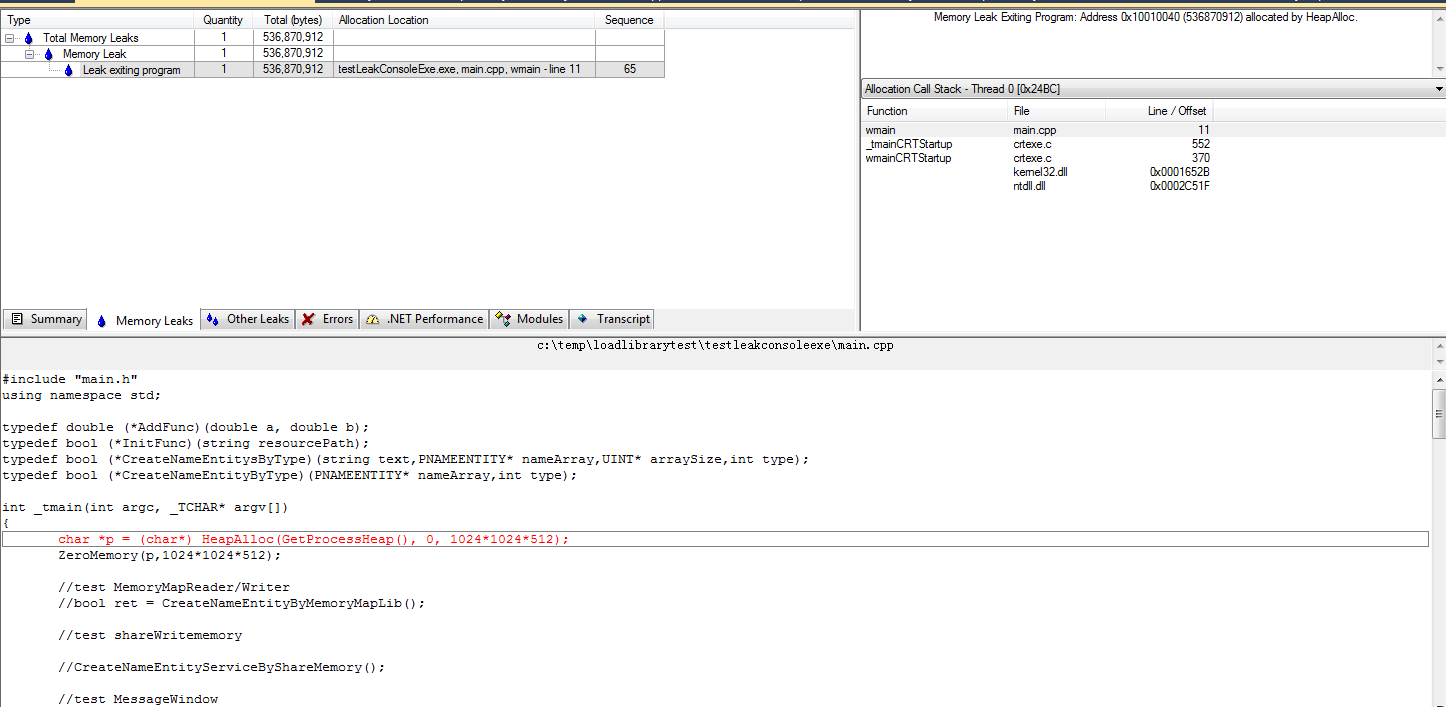
[测试](http://lib.csdn.net/base/softwaretest)内存泄露检测：

1.准备一个测试工程，这里为控制台程序；  
2.菜单"**DevPartner**"→"**Start with Error Detection**"，可以看到解决方案下，自动添加了一个"**DevPartner Studio**"项，右侧包含内存泄露、检测到的错误数等等，如下图：

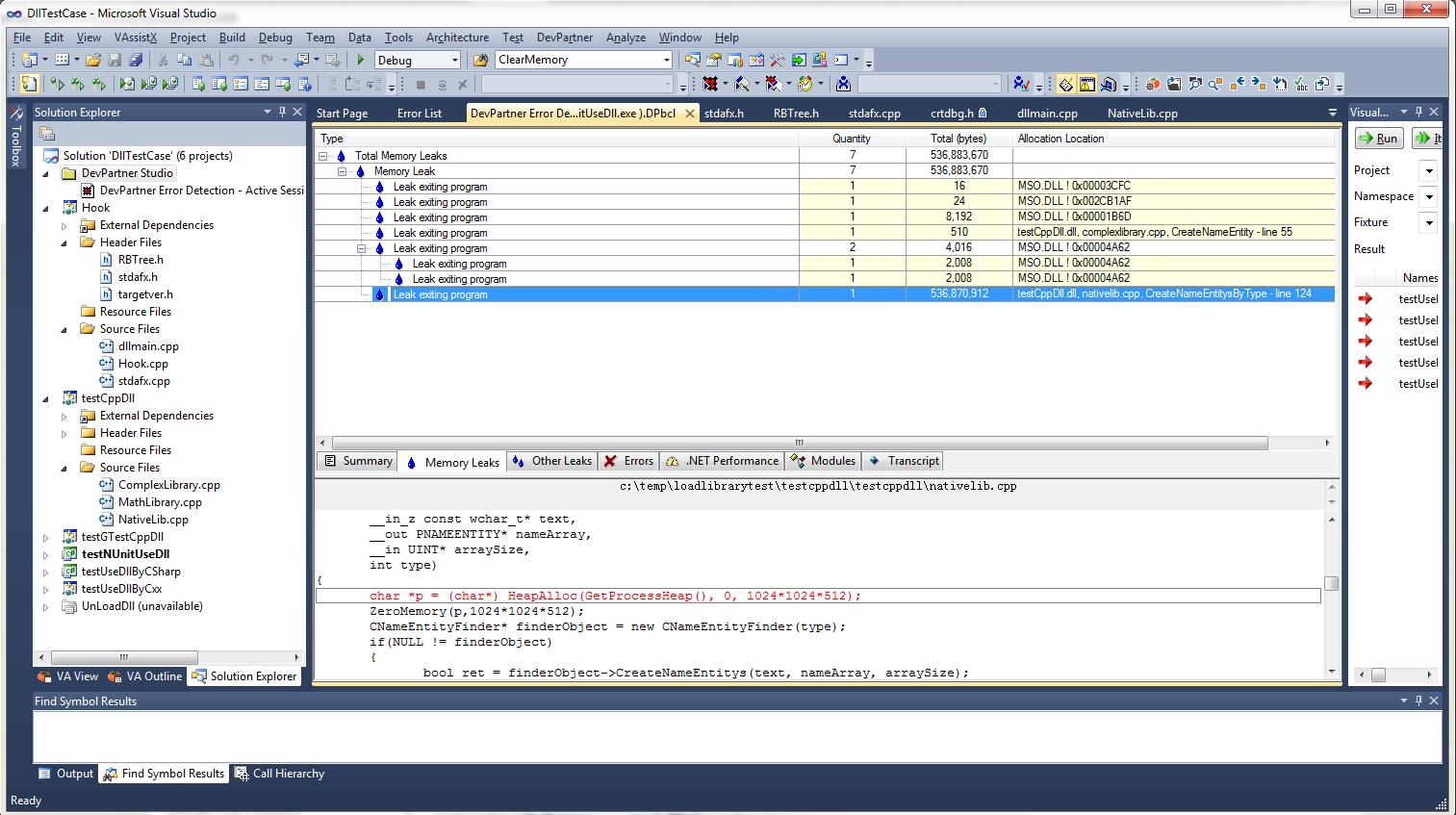


3.现在在测试代码中，加入测试语句;

4.再次"**Start with Error Detection**"，程序结束后，可以看到报告了内存泄露，如下图：



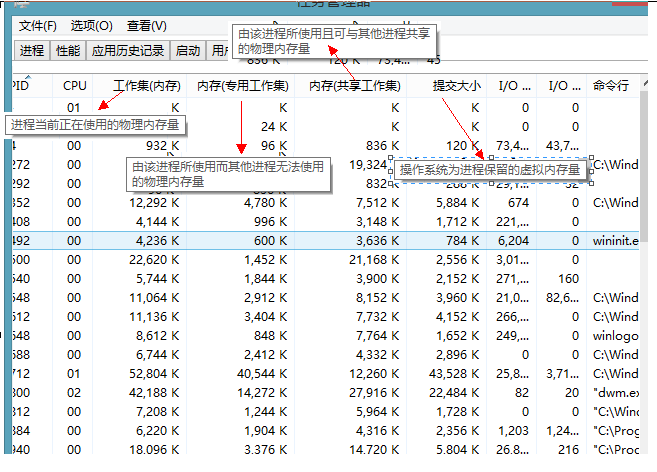
在"**内存泄露**"面板，展开详细，可以看到如下内容：

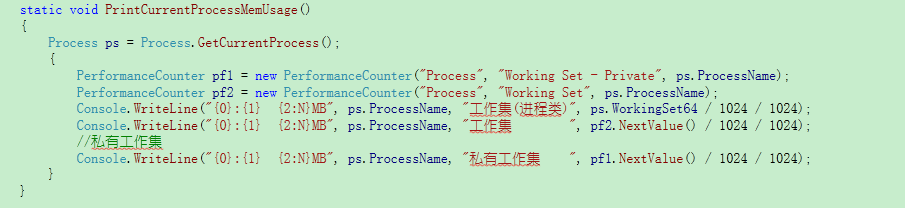


## 2.2 解决方案

### 2.2.1 测试案例

A，查看进程内存使用





B，如下四种测试情况以及测试结果和原因分析：

1、DllImport方式加载C++ DLL库

C#(B)调用C++ dll(A)库: B将通过新建进程调用A，将A进程attach到C#进程B，那么新建的A进程的内存只有等到B退出才释放。



因此如果DLL库存在内存泄露，只有等到主进程退出才能得到释放

2、C#通过LoadLibrary加载C++ dll库

C#(B)调用C++ dll(A)库: B将通过新建进程调用A，将A进程attach到C#进程B，那么新建的A进程的内存只有等到B进程调用FreeLibrary函数时退出才释放。



但是dll库中的函数如果存在内存泄露，仍然无法得到释放，可以考虑将内存申请提至dll的全局变量，当FreeLibrary时主动释放全局变量的内存申请。

3、C#通过New Process调用C# DLL库

C++ DLL（A，C）🡪 C# DLL（B，封装A，C）🡪 C# main（D模块）

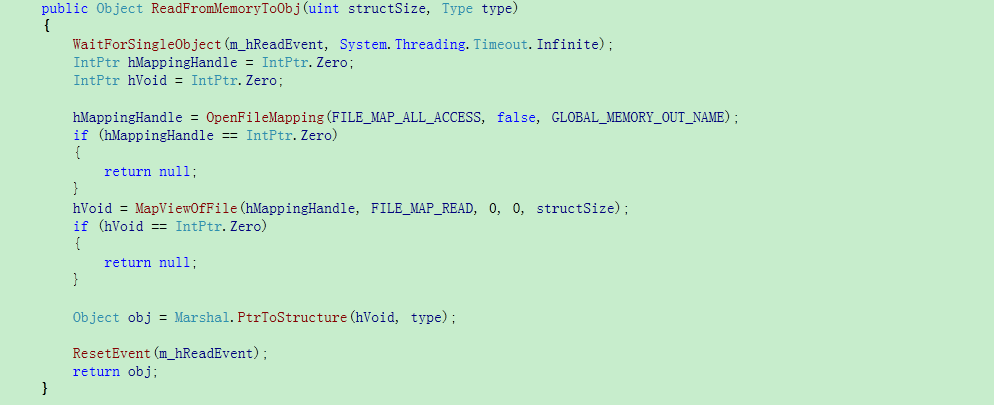
B是C#工程，用于封装C++实现的A和C库的 中间库

D是C#工程，去调用B，从而调用A和C库。

如果A和C存在内存泄露，C#(D)通过新建进程的方式调用 (B)， 调用结束后，由于新建的进程执行完毕后，内存泄露得到释放。

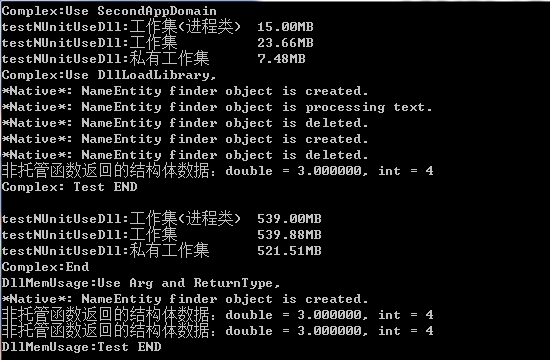
如下测试用例：待完善，未跑通





4、通过AppDomain A(C#)调用另一个AppDomain B(C#,这个Domain去动态调用C++的dll)，然后A去unload appdomainB。

由于Appdoain A和B同属于同一个主进程，因此只有主进程退出时才能释放泄露的内存。



AppDomain B代码



AppDomain A代码调用AppDomain B



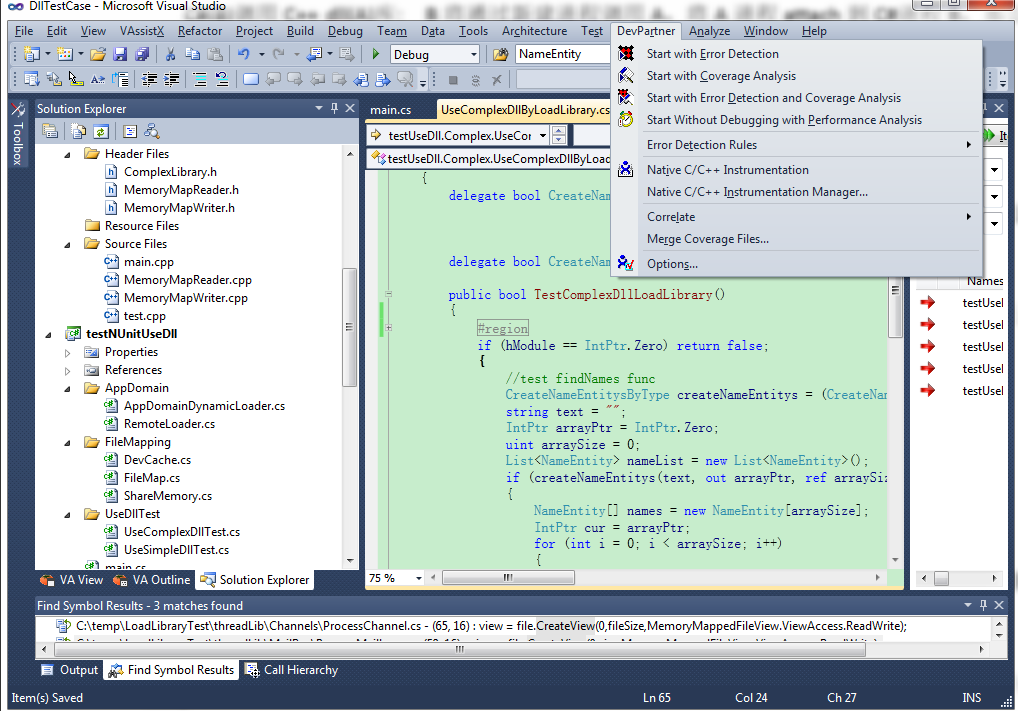
4、使用Visual Leak Detector（导入lib和头文件，项目中代码调用即可)，去完整跑一遍项目程序，然后正常退出，将输出内存泄露报告，根据报告找到并修改内存泄露的代码。

或者

使用visual studio的DevPartner 插件去完整跑一遍项目程序，然后正常退出，将输出内存泄露结果，根据结果找到并修改内存泄露的代码。

上述优化完成之后，对函数内可疑的使用内存管理的变量放置全局，调用dll的方式改为动态方式，可解决内存泄露问题。

上述优化完成之后，仍然存在不稳定的C++dll库，再将这些dll按照2封装成C# DLL库接口，C#主程序通过创建进程的方式调用C# DLL接口去调用C++库。



### 2.2.2 测试代码

A，入口函数

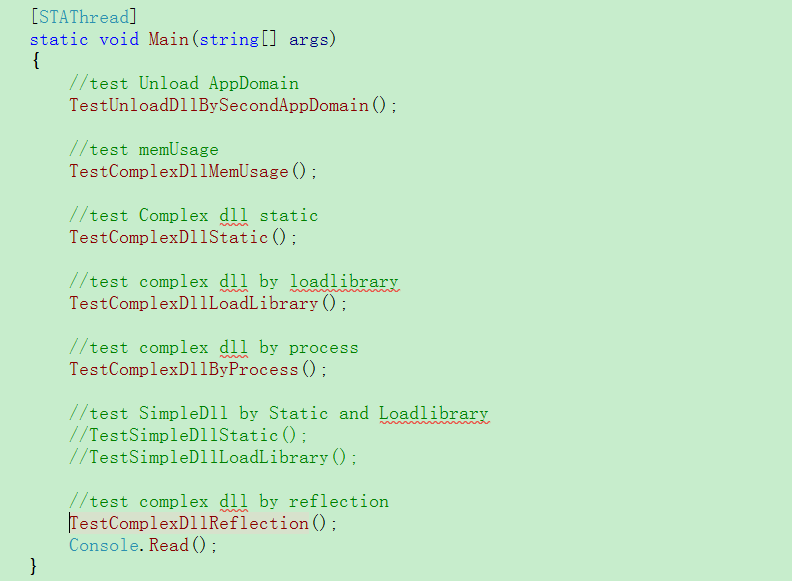
TestUnloadDllBySecondAppDomain()用于测试“测试案例”中的第三种情况

TestComplexDllStatic()用于测试DllImport方式调用Dll库

TestComplexDllLoadLibrary用于测试dynamic加载Dll库，用于测试“测试案例”中的第二种情况

TestComplexByProcess()用户测试用于测试“测试案例”中的第三种情况

TestComplexDllReflection()反射方式，利用dynamic加载并调用dll库



B，代码路径

<https://10.10.20.204:8080/svn/pet/trunk/nonTOF-PET/Software/LoadLibraryTest>

testCppDll工程，C++实现的DLL库

testGTestCppDll工程，用于测试Dll库的GTest框架

testLeakConsoleExe工程，用于C++实现的内存泄露的可执行测试，以及内存映射的实现

testUseDllByCSharp工程，用于C#调用C++库

testUseDllByCxx工程，用于C++调用C++库的测试

testNUnitUseDll工程，用于测试案例的入口main函数和UnitTest框架

encapsuleCppByCSharp工程，用于封装各C++库的C#库，用于其他C#工程启动进程去调用该工程。

C, 工具路径

\\10.10.20.79\Molecular Imaging\Software\linpeiwen