Windows 用户态程序高效排错

思路 技巧 案例 方法

Contents

前言 8
0.1 本文介绍什么? 8
0.2 本文的组织结构 8
0.3 本文的适合人群 9
0.4 本文的使用方法和说明 9
0.5 您的反馈和最新动态 9
第一部分，思考问题 10
1.0 热身运动 10
1.1 第一个例子，非常非常非常奇怪，但的确合理地发生了！ 10
问题描述 10
第一映像 10
深入分析 11
革命尚未成功 12
结论 14
题外话和相关讨论 15
[案例研究,MSDN也不可靠] 15
[案例研究,你敢说CPU坏了?] 15
[案例研究,DWORD有多长?] 16
[案例研究,0xcdcdcdcd] 16
1.2 第二个例子，非常稀疏平常的Session Lost问题，但是却很棘手！ 18
问题描述 18
下一步准备怎么做 18
制定策略 18
具体操作和结论 19
题外话和相关讨论 20
排查session lost的经验: 20
1.3 第三个例子，刚开始很绝望！ 21
问题描述 21
悲观 21
换位思考 22
排错 23
结论和收获 24
题外话和相关讨论 25
Safehandle 25
平衡，取舍，双赢 25
1.4 第四个例子，本可以做得更好！ 27
问题描述 27
排错步骤 27

错过的线索.....29
写在本章结束前.....29
第二部分，重要的知识和工具.....30
2.1 汇编，CPU执行指令的最小单元.....30
简介和一些资源.....30
[案例研究,VC的优化].....31
问题描述.....31
我的分析.....31
[案例研究,编译器的bug].....32
例子程序.....32
跟踪汇编指令来分析.....34
[案例研究,DLL Hell].....35
[案例研究,Release比 Debug快吗?].....36
题外话和相关讨论.....37
2.2 异常和通知.....38
异常小结.....38
[案例研究,让C++打印出callstack].....39
[案例研究,华生医生和Dump文件].....40
问题描述.....40
背景知识.....40
问题分析.....40
新的做法.....41
问题解决了，可是为什么Dr. Watson抓不到dump呢.....42
通知(notification).....45
[案例研究,VB6 的版本问题].....45
题外话和相关讨论.....47
错过第一现场还能分析dump吗.....47
Adplus.....48
快死吧(FailFast).....48
如何调试SetUnhandledExceptionFilter.....49
2.3 内存，Heap，Stack的一些补充讨论.....51
内存/Heap小结.....51
PageHeap & Gflag.....52
简单例子的多种情况.....53
Heap 泄露和碎片.....54
Stack corruption.....55
题外话和相关讨论:.....58
/unaligned参数.....58
Heap trace.....58
[案例研究,内存碎片].....62
2.4 区分层次.....64
CRT和层次.....64
_CRTDBG_MAP_ALLOC.....65
BSTR Cache.....66

题外话和相关讨论 66
CRT Debug Heap一定对Debug有帮助吗？ 67
2.5 调试器和Windbg 68
调试器 68
符号文件 (Symbol file) 68
调试器能看到啥 69
用IE来操练 69
vertarget 70
!peb 70
lmvm 70
.relad / !sym 71
lmf 72
r,d,e 72
S 74
!runaway 74
~ 75
k,kb,kp,kv,kn 76
u 76
x 76
dds 77
小例子 78
.frame 81
dt 81
Live Debug 82
wt 82
条件断点 (condition breakpoint) 84
伪寄存器 87
Step Out的实现 87
Remote debug 88
Debugger auto attach 88
Windbg command line 88
Adplus 89
Debugger Extension 90
2.6 同步和锁 91
三个问题 91
Handle Leak 91
Deadlock 91
线程争用 (颠簸) 93
Windbg中的相关操作 94
!handle 94
!htrace 96
!cs 96
CriticalSection leak(Orphan CriticalSection) 98
Invalid handle exception 100

[案例研究, IndexOutOfRangeException] 101
问题描述....101
这个异常不简单....102
具体操作....103
结论....104
2.7 调试和设计....106
热心朋友的问题....106
题外话和相关讨论....107
[案例分析,反被聪明误]....107
Log的同步....108
第三部分 CLR, 新的乐土....109
3.1 CLR如何工作....109
动态编译....110
内存分配....111
异常处理....112
3.2 详细的来龙去脉....112
开源的CLR:Rotor....112
庖丁解牛....113
第一批断点....113
mscoree!_CorExeMain....114
EEStartupHelper....114
mscorwks!SystemDomain::ExecuteMainMethod....117
第二批断点....117
CallDescr /MakeJitWorker....117
NtUserWaitMessage....120
第三批断点....122
gc_heap::allocate_more_space/ GCHeap::GarbageCollect....122
小结....125
3.3 CLR调试....126
概览....126
SOS....127
命令介绍....127
3.4 小白鼠....131
.load sos....132
!dumpheap....132
!do....133
!gcroot....133
[案例分析, ASP.NET High CPU]....136
!threads....136
!tp....137
!SyncBlk....140
!ip2md....141
!savemodule....141
If broken it is, fix it you should....142

题外话和相关讨论.....142
ReleaseCOMObject.....142
内存移动.....143
内存不移动.....144
臭名昭著的mixed DLL loading deadlock.....144
有用的练习.....145
写在本章结束前.....145
第四部分 ， 一些经验.....146
4.1 排错开始前.....146
别害怕难题，让难题害怕你.....146
一些有用的提问.....147
记得抓取MPS REPORT.....147
Dump初探.....148
4.2 崩溃.....149
不同的死相.....149
Dump的获取.....153
Adplus.....154
dr Watson.....155
Image File Execution Options.....156
COM+/ASP.NET.....156
分析crash dump.....157
[案例分析,VC程序的崩溃].....158
问题背景.....158
Callstack.....159
源代码.....162
This指针.....163
结论.....167
[更多的案例].....167
本章小结.....168
题外话和相关讨论.....168
StackCorruption.....168
4.3 性能.....170
小心无尽的黑夜.....170
性能调优的步骤.....171
性能监视器.....172
如何使用性能监视器.....172
一些重要的计数器.....174
[案例分析, 博客园的性能问题].....177
[案例分析,是sql呢还是gc].....185
[案例分析,deadlock].....192
[案例分析，一团乱麻].....203
Profiler.....220
[案例分析，foreach和for loop性能的区别].....221
本章小结.....226

题外话和相关讨论.....226
Task manager跟performance monitor的差别.....226
性能监视器的牛逼用法.....227
C++跟C#到底谁快.....228
没有profiler怎么办.....228
!finddebugtrue.....231
4.4 资源泄露.....232
泄露分轻重缓急.....232
内存泄露的排错.....233
关键点.....233
定位类型和趋势.....234
区分managed heap leak和native leak.....234
[案例分析，IE的leak].....235
问题背景.....235
万里长征的第一步.....235
Pageheap+UMDH.....238
异常强大的IIS Diagnostics工具.....246
结论.....254
分析IIS Diag.....254
托管内存泄露.....256
[案例分析，object chain].....256
[案例分析，一个bt的案例].....263
碎片的其它原因.....266
[Handle Leak].....267
题外话和相关讨论.....268
GDI Leak.....268
Desktop heap.....268
写在最后.....270

前言

0.1 本文介绍什么?

这是一篇分享 Windows 系统上 User Mode 程序的排错 (troubleshooting) 方法和技巧的文章。

无论是开发，测试还是支持，都会遇上程序运行结果跟预期效果不一致的情况。找到问题的根源和解决的过程，就是排错。如果问题发生的情况很特殊，比如特别难于重现，或者没有源代码可以参考，在这样的情况下解决问题，非常有挑战性!

后面的章节会通过例子来跟大家分享排错过程中的经验和技巧。

一些例子:

- ASP.NET 的程序在测试环境中一切正常，部署到生产环境中后，在压力比较大的时候，发生 Session 丢失现象。 (ASP.NET Session lost)
- VC 开发的程序运行一段时间后，不定时发生内存访问错误，然后崩溃。
- 程序消耗的句柄数量持续增长，内存使用也持续增长，最后性能下降非常厉害。(Handle leak, Memory leak)
- VC 程序中，使用 ShellExecute 打开一个本地的 TXT 文本文件。TXT 格式打开方式关联到 UltraEdit。ShellExecute 执行后,发现 UltraEdit 除了打开这个 TXT 外，另外还打开了一个 GIF 文件。

问题可以表现得非常简单，或者非常复杂。可能涉及不同的开发工具和技术。如何分析解决，正是后面要讨论的。

0.2 本文的组织结构

后面分四部分来解释

- 第一部分介绍最重要的，通用的思考方法。正确的思维方法能找出问题的核心，制定排错步骤和决定采用何种技术和工具进行研究。
- 第二部分介绍对排错非常有帮助的知识点和工具。包括调试器 (debugger)，异常 (exception)，内存工具，同步等等。选择恰当的工具，在恰当的时间，可以获取关键信息。结合对应的知识就可以分析出问题的根源。
- 第三部分介绍.NET Framework (CLR)的相关知识和调试技巧。随着 CLR 应用的普及，Windows 平台上企业级的应用越来越多地用 CLR 开发，CLR 调试的地位举足轻重。

- 第四部分结合前面的内容，针对常见的几大类问题进行了总结。包括崩溃 (crash)，性能问题 (performance),资源泄漏 (resource leak)。

0.3 本文的适合人群

本文适合于所有乐于思考，参与 Windows 用户态程序开发的人

如果读者熟悉 Windows 用户态程序的机理，您将会得到更多的乐趣！

0.4 本文的使用方法和说明

本文的目的在于分享经验，开阔思路。

文章中会介绍很多现实的案例和分析方法。但是这些分析方法并不是死板的，也不见得是最优的，希望读者能开动脑精，参与到思考中来，体会排错的乐趣。

由于案例多种多样，有个人用户，有企业级用户，有桌面程序，也有网络服务，所以文章中会涉及很多的知识点。对所有这些知识点的介绍并不是本文的目的，文章只会分析部分重要的知识点。对于文章中出现的陌生名词，概念，希望读者能够利用文章中给出的链接，网络和 MSDN 来开阔知识面。

在内容组织上，原则是先给出问题描述，然后提供线索，然后分析，最后是结论和思考。目的是鼓励边阅读边思考，体会思考的乐趣。

后面的案例都是这两年来亲手处理过的案例。跟案例相关的具体信息，比如客户信息，源代码，函数名子等等都已删除或混淆。同时也不会附带提供文章中用过的 DUMP 文件。

我在处理这些问题的时候感觉很有趣，所以希望跟大家分享这种乐趣。这并不是一篇严肃的介绍，而更像是 blog 或者随笔，所以请大家别用严肃的眼光来看。希望大家当成一种趣味来阅读。

0.5 您的反馈和最新动态

文章中肯定会有错误，肯定会有更好的解决方法等你去发现。所以您的反馈非常重要。联系方式 lixiong@microsoft.com

第一部分，思考问题

1.0 热身运动

首先跟大家分享导师 Parker 给我的一个问题：

镜子里面的像，为什么左右是反的而上下不是？

我问过很多朋友这个问题，很少有人能够在 3 分钟内给出准确答案。这里列举出一些比较奇特的想法:

1. 因为人的眼睛是左右对称的。(也是某”面试宝典”中的答案)
2. 如果把镜子横过来，左右就不反了，上下就反了。
3. 因为我们在北半球。

从技术层面上说，这里涉及到的知识点只有镜面反射，远比 Windows 内存管理简单。但是要回答清楚，却不是那么信手拈来。这个例子只是想说明，除了知识以外，解决问题需要清晰的思路。

1.1 第一个例子，非常非常非常奇怪，但的确合理地发生了！

问题描述

有一天，一个电话打进来，客户非常气愤地抱怨，调用 ShellExecute 这个 API，传入本地的一个文本文件的路径，有时候会在打开这个 TXT 的同时，还打开另一个不相干的文件！客户非常明确地告诉我，所有的参数肯定没有传错，而且 ShellExecute 的返回值也正确。

第一映像

我仔细思考了两分钟后，告诉客户，不可能。如果参数正确，API 的行为肯定是唯一的。(在往下面阅读以前，请您花两分钟思考，这有可能吗？)

我的第一感觉是用户的参数传递错了。比如打开的是一个 BAT 文件，或者打开方式被用户修改过。这样才有可能同时打开两个文件。但如果真的是这样，程序的行为也应该是唯一的，而不会有时候才出现。

事实是我错了。在察看了用户的截图，并且用客户的代码在本地重现了一次问题后，我不得不相信一次 ShellExecute 调用会偶尔打开两个文件。

在下面的分析中，你会看到该问题是如何发生的。但是，请先记住第一点，相信事实，不要相信经验。如果我胸有成竹地告诉客户，这种问题肯定跟代码不相关，唯一的可能性是一些防病毒程序之类 (防病毒程序的确是很多问题的根源，但是有时也是挡箭牌)，那么就失去了找到真相的机会。

深入分析

背景是这样。客户用 MFC 开发了一个 Dialog Application, 上面放了一个 HTMLView Control, 这个 Control 会显示本地的一个 HTML 文件。这个 HTML 文件会显示一个 GIF 图片。当用户在这个 GIF 图片上点鼠标右键，默认的 IE 右键菜单被用户自定义的菜单代替。当用户点选其中一个菜单命令后，在客户的消息响应函数中调用 ShellExecute 打开本地的一个 TXT 文件。TXT 文件类型跟 UltraEdit 绑定，所以文件会被 UltraEdit 打开。当问题发生的时候，UltraEdit 会在打开 TXT 的同时，打开另外一个二进制的文件。经过分析，这个二进制的文件就是那个 HTML 中显示的 GIF 文件。

客户在 PreTranslateMessage 函数中用下面的代码弹出自定义菜单代替 IE 默认菜单:

```
if(pMsg->msg==WM_RBUTTONDOWN)
{CMenu menu;
menu.LoadMenu(IDR_MENU_MSG_OPEN);
CMenu *pMenu = menu.GetSubMenu(0);
if (pMenu)
{
     CPoint pt;
     GetCursorPos(&pt);
     pMenu->TrackPopupMenu(TPM_LEFTALIGN, pt.x, pt.y, this);
}
return TRUE;}
```

有了这样的背景，问题跟用户的代码就联系起来了。额外打开的文件(GIF)的确是跟这个程序(HTMLView Control)相关的。那下一步我们该做什么呢？

当时我的思路是这样。ShellExecue 能打开这个 GIF 文件，肯定在某种程度上，这个 GIF 跟 ShellExecue 的调用有联系，至少 ShellExecute 要知道这个 GIF 的路径才可以打开它。为了进一步分析，可以选择的步骤是：

1. 打开 WinDbg (一种调试工具，后面有详细介绍), 在 ShellExecute 上设断点，然后开始调试，一步一步走下去看这个 GIF 文件是如何打开的。
2. 通过阅读客户的代码和做进一步的测试来进一步缩小问题的范围。

在仔细思考后，我放弃了第一种。原因是，首先，打开文件的操作不是在 ShellExecute 中完成的，而是在 UltraEdit 中完成的。目标和范围都太大，操作起来很难。其次，问题是偶尔发生的，所以无法保证每次调试都能够重现问题。

整理后的思路如下：

首先，ShellExecute 是在菜单处理函数中调用的，菜单处理函数是用户点选菜单项触发的。可以尝试隔离检查 ShellExecute 跟两者的关联。所以写了一个 Timer，定时地直接调用菜单处理函数。结果发现，通过这样调用菜单处理函数，问题从来不发生。从这个测试中，基本上可以确定问题跟弹出的菜单相关。所以，下一步就是检查替换 IE 默认菜单相关的代码，也就是 PreTranslateMessage。

用户是通过在 PreTranslateMessage 函数中截获鼠标的 WM_RBUTTONDOWN 消息，然后显示菜单。注意，返回值是 False！MSDN 中对这个函数是这样描述的：

CWnd::PreTranslateMessage
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vclib/html/_mfc_cwnd.3a3a.pretranslatemessage.asp

这里客户返回 False，表示消息没有被处理，需要继续发送给后继的消息处理链。然而事实上，客户的代码已经截获这个消息，并且显示了菜单，所以，我建议客户这里修改成 True 测试。这个建议避免了问题的继续发生。

这里揭示了分析问题一定要有理有据，不要放过问题中的任何一个线索，应该尽可能分析出可能的因果关系，然后决定下一步的操作。一旦确认线索后，要仔细彻底地分析，不要错过细节。

革命尚未成功

通过上面的分析和测试，没有花费太多的精力就找到了解决问题的方法。但是，这个时候就开始高兴，未免太早。这里把 False 修改成 True 给客户带来了新问题，上面的分析也不足以揭示为什么简单的 True/False 就给 ShellExecute 带来这么大的影响。让我们继续走下去。

首先解释一下带来的新问题是什么。在菜单的消息响应函数中，客户除了用 ShellExecute 打开文本文件外，还需要用 HTMLDocument 的一些属性和操作来获取当前用户是在 HTML 页面中的哪一个 tag 上点的鼠标，比如是在<img>标签呢还是<dig>标签。如果修改成 True，上面的功能就无法正常工作。

通过修改 ShellExecute 获取线索后，可以有针对地使用调试器进一步分析。反汇编 ShellExecute 的代码后，发现里面的实现非常复杂，所以不可能首先读懂代码后再进行调试。所以，采取的办法是分析 ShellExecute 在正常情况和异常情况下表现的差异。首先，通过 windbg 的 wt 命令 (这里不详细介绍这些命令的用法，留到下一章)，观察 ShellExecute 内部的调用栈(callstack)。发现 ShellExecute 是通过 DDE 的方法来打开目标文件。还好我对 DDE 技术不是很陌生，在研究了 MSDN 的说明以后，了解到 DDE 其实内部是依靠

WindowsMessage 来完成进程之间通信的。于是，又在 PostMessageW/SendMessageW 上设定条件断点，每次调用这两个函数的时候，就把 Windows Message 的详细信息和 callstack 在 windbg 中打印出来。

问题发生的时候的 callstack 是这样的。仔细看看，你能解释这样的 callstack 是如何引发问题的吗？

```
USER32!PostMessageW
ole32!CDropTarget::Drop
ole32!CDragOperation::CompleteDrop
ole32!DoDragDrop
mshtml!CLayout::DoDrag
mshtml!CElement::DragElement
mshtml!CImgElement::HandleCaptureMessageForImage
mshtml!CElementCapture::CallCaptureFunction
mshtml!CDoc::PumpMessage
mshtml!CDoc::OnMouseMessage
mshtml!CDoc::OnWindowMessage
mshtml!CServer::WndProc
USER32!InternalCallWinProc
USER32!UserCallWinProcCheckWow
USER32!DispatchMessageWorker
USER32!DispatchMessageW
SHELL32!SHProcessMessagesUntilEventEx
SHELL32!CShellExecute::_PostDDEExecute
SHELL32!CShellExecute::_DDEExecute
SHELL32!CShellExecute::_TryExecDDE
SHELL32!CShellExecute::_TryInvokeApplication
SHELL32!CShellExecute::ExecuteNormal
SHELL32!ShellExecuteNormal
SHELL32!ShellExecuteExW
SHELL32!ShellExecuteExA
SHELL32!ShellExecuteA
```

从 CDropTarget::Drop 这个函数可以猜到 gif 是怎么被打开的，但奇怪的地方是 ShellExecute 居然把 mshtml 牵涉进来了。如果仔细回想一下问题重现的步骤，会发现遗漏了一个重要的地方，那就是:

既然在 PreTranslateMessage 中返回了 False, 那么这个消息就相当于没有处理过，那么，IE 的默认菜单为什么没有弹出来呢！

根据这个 callstack, 问题发生的经过是：

1. 用户在 HTMLView 上面的 GIF 文件上点下鼠标右键(注意，这个时候点下去，还没有放开)，系统发送 WM_RBUTTONDOWN 消息
2. 在 PreTranslateMessage 里面，判断 WM_RBUTTONDOWN 消息，用 TrackPopMenu API 显示菜单，并且代码堵塞在 TrackPopMenu API 上
3. TrackPopMenu 显示出菜单
4. 用户放开鼠标右键，并且移动鼠标，点击菜单项。由于当前有菜单弹出，所以松开鼠标，移动鼠标的消息都不会被应用程序处理。
5. 点选菜单后，TrackPopMenu API 返回，同时系统向应用程序发送对应菜单项点击后的 WM_COMMAND 消息
6. PreTranslateMessage 函数返回 False，于是 WM_RBUTTONDOWN 的消息继续发送到后面的消息处理函数
7. WM_RBUTTONDOWN 的消息传播到后面消息处理函数导致的结果是在 HTML 的 GIF 上点下了鼠标。由于 WM_RBUTTONUP 的消息在显示菜单的时候被菜单吃掉了，所以，对于 HTMLView 来说，点下去的鼠标就一直没有放开。因为 IE 是在 WM_RBUTTONUP 的时候显示自身的弹出菜单，所以 IE 自身的菜单也因为缺少 WM_RBUTTONUP 消息而没有显示。
8. WM_COMMAND 的消息导致消息处理函数执行，于是 ShellExecute 得到了调用
9. ShellExecute 通过 DDE,给 UltraEdit 发送打开文件的消息，于是 UltraEdit 就打开了 TXT 文件，同时，UltraEdit 的窗口切换到了前台。
10. 由于 ShellExecute 需要检查 UltraEdit 返回的 DDE 消息来判断打开是否成功，所以 ShellExecute 需要维持自己的消息循环，这也就是 callstack 中 SHProcessMessagesUntilEventEx 函数做的事情。
11. 这个时候，消息处理函数继续分发消息队列中的消息，这个消息会被 ShellExecute 的消息循环分发，从 callstack 可以看到，这些消息到达了 HTMLView
12. 根据第 7 点，HTMLView 得到的消息是鼠标在 GIF 落下，HTMLView 会把 GIF 设定成 Captured 状态，然而接下来的鼠标抬起消息丢失，导致 HTMLView 走了跟普通点击(鼠标一下一上)不一样的执行顺序。由于 GIF 是 Captured 的状态，加上前台窗口是 UltraEdit, 这些非常规的状态和后继消息导致了 HTMLView 认为当前发生了落下鼠标紧接移动鼠标的行为，类似用鼠标右键把这个 GIF 文件从 HTMLView 上拖动到了 UltraEdit 上。GIF 就这样被打开了。

由于消息队列分发消息是在当前用户进程中执行，而打开 TXT 文件是在 UltraEdit 中执行的，同时消息队列中的消息也没有固定的规律，这些不确定性导致了问题的偶然性。

通过比较分析正常情况 (return TRUE) 和异常情况(return FALSE)下 wt 命令和 PostMessageW/SendMessageW 函数的调试输出，问题基本上找到原因了。整体的思路是先隔离问题，然后比较不同情况的差异。当然，获取这些差异的时候要聪明一点，选用正确的方法和工具。

结论

不知道你有什么感想，总之，当时我的感觉就像一个状态机，是一个 Windows+MFC Framework，根据用户的操作，严格地响应消息，执行对应的代码，哪怕这些状态是非常规的。不要迷信，尊重 CPU 和代码运行法则，用 CPU 的节奏和方法来考虑问题，才可以看到问题的来龙去脉。

从程序设计的角度来看，细节是非常重要的。到底应该返回 True 还是 False,到底是在

WM_BUTTONDOWN 还是 WM_BUTTONUP 的时候处理消息，都是需要仔细推敲的问题。

这一个 ShellExecute 的案例就到此为止了，找到问题根源后，正确做法就很明了了:

> How to disable the default pop-up menu for CHtmlView in Visual C++
> http://support.microsoft.com/?id=236312

我的收获是：

1. 尊重事实，而不是经验
2. 详细观察问题发生的过程，对任何线索保持敏感
3. 用对比的方法来寻求问题的根源
4. 用 CPU 的节奏和方法来来理解整个系统

题外话和相关讨论

[案例研究,MSDN 也不可靠]

不仅仅是经验不可信，就算 MSDN，也没有事实确凿可信。下面这篇知识库文章，是详细测试了用户的代码后，确认 MSDN 有一个 bug，然后申请了一篇知识库文章(Knowledge Base)来作专门的解释:

> Description of a documentation error in the "Assembly.Load Method (Byte[])" topic in the .NET Framework Class Library online documentation
> http://support.microsoft.com/kb/915589/en-us

里面的第一句话就是:
The "Assembly.Load Method (Byte[])" topic in the Microsoft .NET Framework Class Library online documentation contains an error

[案例研究,你敢说 CPU 坏了?]

这个案例是美国工程师处理的。客户有两台负载均衡的小型机，每一台机器有 64 颗 CPU。从硬件到软件环境都完全一样。其中一台机器一直正常，另一台机器的 MSDTC 偶尔会崩溃。那位工程师根据调试器的输出，给客户说：“我怀疑问题是 CPU 导致的，你换一块 CPU 试试看”。将信将疑的客户立刻把 Intel 的工程师叫过来，热插拔换了一块 CPU，问题果然就搞定了。我在想，如果某一天，当我从调试器上看到 xor eax,eax 执行后，eax 不等于 0，我敢给客户说这是 CPU 的问题吗？

> There's an awful lot of overclocking out there

http://blogs.msdn.com/oldnewthing/archive/2005/04/12/407562.aspx

同时，现在有的 rootkit 和加壳程序已经能够欺骗调试器了。但有的时候，一些新加入的系统功能也会导致一些有趣的现象:

VS2003 在 push edi 的时候 AV
http://eparg.spaces.msn.com/Blog/cns!1pnPgEC6RF6WtiSBWIHdc5qQ!379.entry

SEH,DEP, Compiler,FS:[0], LOAD_CONFIG and PE format
http://eparg.spaces.msn.com/blog/cns!59BFC22C0E7E1A76!712.entry

要区分事实和谎言，不是那么容易……

[案例研究,DWORD 有多长?]

对某些数字保持敏感有助于找到线索。客户写了一个文件类，但是发现操作产度大于 4GB 文件的时候，就会有一些莫名其妙的问题发生。你能一下子猜出原因吗？

4GB 表示 DWORD 的上限。如果用一个 DWORD 指针来获取文件大小，当文件大小超过 4GB，问题就会发生。所以，客户很可能使用了错误的 API。

解决这个问题，应该用 GetFileSizeEx API 而不是 GetFileSize。其实，不单单是客户会犯这样的错误，.NET Runtime 也有类似的 bug, 由于没有使用正确的方法来获取物理内存数量，导致物理内存超过 4GB 后性能反而下降：

FIX: Generation 1 garbage collections and generation 2 garbage collections occur much more frequently on computers that have 4 GB or more of physical memory in the .NET Framework 1.1
http://support.microsoft.com/kb/893360/en-us

所以，常常会有人用下面的智力题来面试开发人员：*让工人为你工作 7 天，工人得到的回报是一根金条。每天你都必须支付给工人一天的费用，每天的费用为七分之一根金条，但这根金条不能被平分成 7 段，只允许你把金条弄断两次，你如何给工人付费*?

在网上找一下就可以看到答案。如果对二进制比较敏感，你会发现答案跟 2 的 n 次方是相同的，当然，这绝对不是偶然

[案例研究,0xcdcdcdcd]

关于数字敏感的另一个例子：*一段加密解密程序，首先由用户输入 16 个字节的原文，然后程序用固定的密钥加密生成密文，接着再对密文解密得到原来的原文，并且打印。问题的现象是，无论用户输入怎么样的原文，经过加密解密后，打印出来的原文的二进制都是一连串的 0xcdcdcdcd*

仔细回忆一下， VC 在 debug 模式下，对 CRT(C Runtime)分配的堆(Heap)内存，都会初始化填充成 0xcdcdcdcd。目的就是为了方便程序员 debug. 一旦看到 0xcdcdcdcd, 就表示访问了没有初始化的内存。所以，这个问题显然是开发人员忘记把用户的输入拷贝到对应的缓冲区，导致缓冲区里面的值都是分配内存时有 CRT 初始化填入的 0xcdcdcd.

1.2 第二个例子，非常稀疏平常的 Session Lost 问题，但是却很棘手！

问题描述

客户抱怨，刚刚开发完成的 ASP.NET 站点测试阶段一切正常，但是放到生产环境上，压力一大，就会发生 Session lost 现象。问题一共就发生过三次，是通过分析 log 文件分析看到的。Log 文件记录的是每个时刻 Session 中的内容。

下一步准备怎么做

这个问题困难的地方在于重现的几率很小，没有多少详细观察的机会。所以，必须制定非常周全的计划，以便问题再发生的时候，抓到足够多的信息。如何制定周密的计划呢？

思路非常直观，了解 Session 实现的细节，总结出导致问题的所有可能性，抓取信息的时候排查所有的可能性。

关于 ASP.NET Session 的细节，可以参考：
Underpinnings of the Session State Implementation in ASP.NET
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnaspp/html/ASPNetSessionState.asp

制定策略

有了对 Session 的理解后，把这个问题分成了下面这几种情况

1, 最简单的情况就是所有用户，所有 session 都丢了。这种情况一般发生在 In-Proc 的 session mode 上。原因就是 appdomain 重启或者 IIS 进程崩溃。可以通过性能日志或系统日志来排查。
2, 稍微麻烦一点的就是某一个用户的 session 丢了，而没有影响到所有用户。观察方法是，首先在 session start 里面做 log，把每一个 session id 的创建时间都记录到 log 里面，同时往 session 里面添加一个测试用的 session value。问题发生的时候，把受到影响的 session id 记录下来，比较 log 看看这个 session 是不是刚刚建立的。如果是，很有可能是客户端的原因导致 session id 丢了，比如 IE crash 导致 cookie 丢失。
3. 如果不是，那看看测试用的 session value 是不是丢。如果这个也丢，应该是代码中调了 Session.Clear.
4. 如果测试用 session value 没有丢，情况就变成一个用户的 session 里面的一部分 value 丢了。很可能由于用户的代码逻辑导致的。解决方法就是通过更详细的 log 来定位问题，然后阅读代码来检查。

可以看到，问题的特征跟潜在的根源是对应的。目的在于区分出这三种情况：

1. 所有用户的所有 Session 全没有了

2. 一个用户的 Session 全没有了
3. 一个用户的部分 Session 没有了

针对每种情况，采取的 log 策略是：

对于第一类情况，可以在 Application_Start/End 函数中记录下时间来检查 Appdomain 是不是重新启动。

对于第二类情况，log 文件应该记录下 session id 和 session 创建的时间。以便区分是否是 cookie id lost 导致的。如果是 cookie id lost, 那问题就是在客户端，或者是网络原因。

对于第三类情况，可以在工程中搜索所有 Session Clear 的调用，每次调用前写 log 文件来记录。如果工程很大，无法逐一添加，可以加载调试器，在 Session Clear 函数中设定条件断点来记录。

具体操作和结论

总接下来，具体的实现是：

1. 在 session start 里面把这个 session 创建时间记录到 session 里面。这个创建时间也同时充当测试用的 session value.
2. 代码中对 session 操作的地方，写 log 到以 sessionid 为文件名的文件里面去
3. 用 log 文件记录每次 session 的操作，发生在什么函数，发生的时间，session 内容的变化
4. 当 Exception 发生的时候，在 Exception handler 中记录发生问题的 session id 和残留下的 Session value

这样，问题发生的时候，根据 Exception handler 记录的 session id 找到 log 文件，就可以很清楚地得到所需要的信息。

在做了上面的部署后，等了大约一个星期问题重现了。在 log 文件中，发现这样的信息：

1. 某一个用户的部分 Session 丢失。
2. 从 Session 创建时间看，该 Session 已经维持很长时间了。
3. 通过检查 Session Clear 的调用纪录，发现丢失的 Session 的确是由用户自己的代码清除的。同时发现这些代码的运行次序跟设计不吻合。根据设计初衷，在清除 Session 后，页面会重定向到一个专门的页面重新添加 Session，然后继续操作。但是 log 表明了这个专门的页面并没有得到执行。

检查用户的重定向代码后发现，重定向是通过客户端 javascript 来实现的。用 javascript 来维护事务逻辑，犯了 web 开发的大忌。因为 javascript 的行为对客户端浏览器的依赖非常大。重定向最好用 http 302 来实现 (Response.redirect 就是这样实现的)，同时需要在服务器端添加检测代码来确保业务逻辑的正确顺序：

HTTP Status code definition
http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html

在做了上面的改动后，问题解决。Session lost 不是一个很有趣的例子，但是它却非常有代表性：

问题现象简单而且明确，但是很难重现。问题的原因是在早期积累下来，如果只观察问题暴露出来后的现象，为时已晚。对付这类问题，关键在于根据问题的特点，在适当的地方主动抓取信息。

题外话和相关讨论

排查 session lost 的经验：

除了上面的步骤外，遇上 session lost 还应该注意

1. 如果用户开两个 IE 进程 (注意，不是用 ctrl+N 开两个 IE 窗口) 访问同一个 url, 这两个进程分别对应两个独立的 session 的
2. 最容易忽视的情况是 session timeout。可以观察 Session id 是否变化，或者在 session end 里面加 log 进行排查

1.3 第三个例子，刚开始很绝望！

问题描述

根据下面一篇文章的介绍，客户决定升级到.NET Framework 2.0 来借助 ADO.NET 2.0 提高性能。

DataSet and DataTable in ADO.NET 2.0
http://msdn.microsoft.com/msdnmag/issues/05/11/DataPoints/default.aspx

但是根据用户的测试，使用 ADO.NET2.0 后，性能反而下降。

拿到用户的代码一看，非常简单:

```
OracleConnection conn = new OracleConnection();
conn.ConnectionString = "...";
conn.Open();
OracleCommand cmd = new OracleCommand();
cmd.Connection = conn;
OracleDataAdapter dap = new SOracleDataAdapter("select * from mytesttable", conn);
DataTable dt = new DataTable();
DateTime start = System.DateTime.Now;
dap.Fill(dt);
TimeSpan span = DateTime.Now - start;
conn.Close();
Console.WriteLine(span.ToString());
Console.WriteLine("The Columns.Count is" +dt.Columns.Count.ToString());
Console.WriteLine("The Rows.Count is"+dt.Rows.Count.ToString());
```

测试用的数据库表也很简单，25 万行数据，4 个字段。通过检查 span.ToString 的结果，发现同样的代码，ADO.NET 2.0 比 1.1 慢了接近 100%。dap.Fill 方法的执行时间从原来的 3 秒增大到 6 秒。

悲观

应该如何着手这个问题？

当时我测试完成，看到这个结果后，我的感觉：悲观。看看我们能够做什么:

1. 后台数据库表格的定义非常简单。4 个字段都是 int, 都是在没有 index, primary key, foreign key 等约束条件下测试的。也就是说，这个问题是跟数据表的 schema 定义无关的，完全是客户端的问题。
2. 代码已经非常简单. Console 工程里面 Main 函数里面就做这么一件事情。客户端没有任何可以修改和变动的地方。
3. .NET Framework 1.1 和.NET Framework 2.0 共存在同一个客户端，测试也是在同一个客户端测试和比较。所以软硬件环境，比如 Oracle Client 都相同。唯一的区别就是.NET Framework 的版本。这也就是客户最关心的地方。

找不到任何努力的方向。看来接下来就因该向客户坦白，“我没办法，都怪 ADO.NET 2.0 的开发人员把 Fill 方法的性能弄得这么差！”然后推托说 MSDN 的文章是在.NET Framework 2.0 beta2 的时候写的，并不准确。

换位思考

还好没有立刻放弃尝试其它的努力。仔细阅读并且测试了 MSDN 上关于 ADO.NET 2.0 性能的文章后，发现:

1. 事实证明，MSDN 上给出的例子的确证明了 ADO.NET 2.0 的性能更好。不过 MSDN 例子中的表格 Schema 比较复杂，而客户使用的 Schema 非常简单。
2. 虽然在客户的情况中，ADO.NET 2.0 的性能下降，但是也可以找出一些客户没有测试的情况，在那样的情况下，ADO.NET 2.0 可以把性能提高了 2 个数量级。

同时，当跟同事讨论这个问题的时候，他们最感兴趣的不是性能下降了多少百分比，而是关心客户为什么要一次操作 25 万行数据。仔细讨论后，收获了下面一些疑点:

1. 客户的测试表明 25 万行数据总共慢了三秒钟，平均下来每一行慢 8 个微秒。从开发的角度来思考，读取数据库的操作首先把请求从应用程序发送到数据库引擎，然后通过网络到数据库服务器取回数据，然后把数据填充到 DataTable 中。这么一连串复杂的操作中，任何一点小的改动都可能通过蝴蝶效应夸大。换句话说，如果看百分比，性能的确下降了一倍。但是如果看绝对值三秒钟，跟数据量一比较，对性能下降的看法就会有更多的思考空间。
2. 用户的代码是把 25 万行的表格一次读到程序中来。这样的代码会运行在性能敏感的情况下吗？比如 web 服务器。计算一下, 250000 行*4 字段*每个字段开销 20 字节=19MB 数据。如果每一个请求都要带来 3 秒(ADO.NET 1.1 的性能数据)的延迟，20MB 的内存开销，数据库服务器和 Web 服务器大量的 CPU 操作，以及频繁的网络传输，这样的设计肯定是有问题的。所以，正常情况下，这样的代码应该是初始化某一个全局的数据，程序启动后可能一共就运行这样的代码一次。也就是说，虽然这里性能损耗的百分比很明显，但是给最终用户带来的影响，有那么明显吗？
3. 我们并没有考查数据的所有操作。这里客户考察的性能是获取数据的开销，并没有考虑后面使用数据的性能。如果 ADO.NET 2.0 牺牲获取时间来改进了使用数据的性能，使得从 DataSet 中每使用一行收获 1 微妙的收益，如果每行数据都被使用了 8 次以上，从整体来看，性能是有回报的。

有了上面的分析后，我决定首先问问客户真实环境是怎么样的，到底是在怎么样的情况下使用的这段代码。同时，用下面的方法来寻求性能下降的根源：

排错

首先，用 Reflector 反编译分析 DataAdapter.Fill 方法的实现。

Reflector for .NET
http://www.aisto.com/roeder/dotnet/

发现 DataAdapter.Fill 方法可以分解成下面两个部分:

1. 用 DataReader 读取数据。
2. 构造 DataTable,填入数据。

也就是说，DataAdapter.Fill 的实现，其实是把 ADO.NET 中的 DataReader.Read 和 DataTable.Insert 合并在一起。于是，可以用这两个函数来代替 DataAdapter.Fill 方法测试性能：

```
static void TestReader()
{
    OracleConnection conn = new OracleConnection();
    conn.ConnectionString = "...";
    conn.Open();
    OracleCommand cmd = new OracleCommand();
    cmd.Connection = conn;
    cmd.CommandText = " select * from mytesttable ";
    OracleDataReader reader = cmd.ExecuteReader();
    object[] objs=new object[4];
    DateTime start = System.DateTime.Now;
    while (reader.Read())
    {
        reader.GetValues(objs);

    }
    System.TimeSpan span = System.DateTime.Now - start;
    reader.Close();
    conn.Close();
    Console.WriteLine(span.ToString());
}

static void TestDT()
{
    DataTable dt = new DataTable();
    dt.Columns.Add("col1");
    dt.Columns.Add("col2");
    dt.Columns.Add("col3");
```

```
            dt.Columns.Add("col4");
            DateTime start = System.DateTime.Now;
            for (int i = 1; i <= 250000; i++)
            {
                dt.Rows.Add(new object[] { "abc123", "abc123", "abc123", "abc123" });
            }
            System.TimeSpan span = System.DateTime.Now - start;
            Console.WriteLine(span.ToString());
        }
```

根据测试，性能的损失主要是由于 DataReader.Read 方法带来的。如果要再进一步分析 Read 方法的性能损失，使用 Reflector 就太困难了。这里采用的工具是企业版本 VS2005 里面自带的 Profiler。通过 Tools -> Performance Tools -> Performance Wizard 菜单激活。这个工具可以显示出每一个方法(包括子方法)调用所花费的时间，以及占整体运行时间的比例。为了让问题更加明显，这里把数据库的行数增加了 10 万来方便观察。沿着花费时间比例最多的函数一路走下去，发现 DataReader.Read 的方法实现分成两部分，自身的托管代码调用和非托管代码调用。分别占用了 35%左右的时间和 65%左右的时间。有了这个信息后，再通过 Reflector 分析 ADO.NET1.1 中的对应函数的实现(可惜没有找到.NET Framework 1.1 上很好的 Profiler, 不然直接分析两者时间比例就可以方便的看出问题)，发现非托管代码部分的调用几乎没什么差别，都是调入数据库的客户端非托管 DLL。主要差别在托管代码部分。其中引起注意的是 ADO.NET 2.0 增加了 SafeHandle.DangerousAddRef/DangerousRelease 调用。每一对这样的调用就要花费 7%的时间。而每读取一行数据，需要用大约 3 对这样的调用。经过比较分析，认为问题就是在这里。

由于对数据库的操作和数据填充最终通过调用非托管的数据库引擎来完成，所以需要向非托管的 DLL 传入托管代码管理的缓存空间，而 SafeHandle 就是管理这种资源的。在.NET Framework 1.1 中，由于缺少 SafeHandle 类，高负载环境下程序存在表现不稳定的危险，没有完美的解决方法。.NET Framework 2.0 增加了 SafeHandle 来保证程序的可靠性。然而，代价就是 25 万行数据发生 3 秒钟的时间损失。(后来经开发人员确认，这 3 秒钟的损失，在下一版本的 Framework 也可以想办法优化掉！)

结论和收获

在跟客户做进一步的交流后，这个问题的结论如下：

1. MSDN 文章的介绍是正确的。如果用文章里面的例子，的确可以看到性能在数量级上的提升
2. 在客户的真实环境中，损失的 3 秒时间其实不会对最终用户和整个程序造成影响
3. 3 秒钟不是白白损失的，换来的是程序的可靠性

到最后，我们还是没有能够找回这 3 秒钟，但是找到了 3 秒钟背后更多的东西。收获是

1. 至关重要的是思考，不要迷失在问题中，而是要全访问考虑客户和开发人员真正关心的是什么
2. 把问题放到真实环境中去考虑，通过不同的角度分析理解这个问题，看看问题背后是否有更多可以发掘的地方。这一点帮助我们发现了 100%的性能损失在客户的实际环境下是微不足道的。

3. 采取合适的工具。这里使用了 reflector 和 profiler
4. 必要的时候把问题扩大，方便分析。这里通过增加数据量来让结果更明显。这个思路在解决内存泄漏的时候也是非常重要的。因为泄漏 1G 内存的程序检查起来肯定比只泄漏 1k 内存的程序容易。
5. 任何问题都必有因果。某一方面的损失并不代表整体就有问题。取舍(Tradeoff)是必要的。

题外话和相关讨论

Safehandle

关于 Safehandle 更多的一些讨论:

SafeHandle: A Reliability Case Study [Brian Grunkemeyer]
http://blogs.msdn.com/bclteam/archive/2005/03/16/396900.aspx

CLR SafeHandle Consideration [grapef]
http://eparg.spaces.msn.com/blog/cns!59BFC22C0E7E1A76!576.entry

平衡，取舍，双赢

关于平衡，取舍，双赢的一个有趣，权威的文档是 RFC1925:

RFC 1925 (RFC1925)
http://www.faqs.org/rfcs/rfc1925.html

关于 tradeoff 的另一个案例是要不要使用/3GB. 关于/3GB 的信息可以参考：

Large memory support is available in Windows Server 2003 and in Windows 2000
http://support.microsoft.com/kb/283037/en-us

问题是这样的，用户有一个大型的 ASP.NET 站点，压力大的时候常常在跟数据库通信时发生 Invalid Operation 的异常。经过网络工程师和数据库工程师的排查，该问题是由于使用了/3GB 导致的。/3GB 增加了用户态内存地址空间，代价是内核可用内存减少。由于网络 IO 操作在内核完成，内核内存减少导致了这个问题。

但是，我们却不能够建议用户去掉/3GB 开关，因为这个开关是半年前为了解决 ASP.NET 用户态内存空间不够的时候加上去的。(由于客户程序页面很多，默认的 2GB 用户态空间无法满足需求，继而导致 OutOfMemory 的异常) 所以，如果要从根源上同时解决 ASP.NET 内存空间和数据库访问的问题，使用 64 位系统是最好的方案。不过最后还是在系统工程是的帮助下通过下面这篇文章，找到了 2GB 和 3GB 之间的一个平衡点，平衡了内核和用户态空间的分配：

How to use the /userva switch with the /3GB switch to tune the User-mode space to a value between 2 GB and 3 GB
http://support.microsoft.com/kb/316739/en-us

1.4 第四个例子，本可以做得更好！

问题描述

Windows SharePoint Portal 是运行在.NET Framework 上的一个 web 应用程序。管理员可以设定使用英文界面或者中文界面。某一天一个客户抱怨 SharePoint 无法显示出中文界面。所有的页面都用英文显示。经过仔细观察，发现下面的现象:

1. 管理员的设定值是中文界面
2. 该程序在过去的一年中都运行正常
3. 大多数情况下，中文界面工作正常
4. 偶然的情况下，中文界面会变成英文界面
5. 变成英文界面后，过一段时间后会自动变回中文界面
6. 变成英文界面后，如果重新启动 IIS, 有可能解决问题

排错步骤

经过观察，发现客户的安装和配置是没有问题的。大致调试后发现:

1. 客户程序工作线程的 UI Culture 的确是 zh-cn, 跟客户的设定一致
2. 界面显示资源是通过ResourceManager.GetString系统方法来加载的。
   ResourceManager.GetString
   http://msdn2.microsoft.com/en-us/system.resources.resourcemanager.getstring.aspx
3. 该客户的中文资源卫星文件安装正确

根据这三个信息，GetString 应该返回对应的中文资源，但是在问题发生的时候，返回的是英文资源。所以，需要解释的是这个.NET 系统方法为何表现异常。面临的难点是:

1. 无法简单重现问题，需要反复重新启动 IIS 多次后，问题才会偶然发生
2. 问题只有在客户机器上才能重现，需要远程调试，操作起来不是很方便
3. 由于出问题的代码是.NET Framework 的函数，不是客户的代码，没办法用第二个例子的方法来添加 log 分析

当时欧洲很多客户也有类似问题发生，比如西班牙语的站点变成了英语。日本更是严重，有十几个站点都有这个问题，但是几个星期来一直都没有找到问题的根源。

由于问题的线索是 GetString 方法的异常表现，所以排错计划也非常直接: 跟踪 GetString 的执行过程。在做远程调试以前，首先在本地跟踪 GetString 的执行过程来理解逻辑。大致的逻辑是:

1. 资源符号从卫星加载起来后会放到内存的 HashTable 中
2. GetString 检查 HashTable 中是否有请求的资源，如果有，跳到第 5 步。
3. 如果没有, GetString 会让 ResourceManager 去根据当前线程的 UI Culture 寻找对应卫星目录来加载
4. ResourceManager 加载卫星 Assembly 到内存，同时把所有的资源符号都入 HashTable
5. GetString 返回这个 HashTable 的内容。
6. 任何问题发生，比如 ResourceManager 没有找到对应的卫星目录，或者资源文件不存在，就用中立语言(英语)代替
7. 一旦 HashTable 填充完成后，GetString 的后继执行就不需要在牵涉 ResourceManager 了，直接返回 HashTable 的内容。

根据上面的逻辑，客户那里的问题很可能是 ResourceManager 没有找到资源文件，于是用中立语言英语初始化了 HashTable。根据这个信息，决定用 Filemon 来检视资源文件的访问。可惜的是，Filemon 的结果表明每次对资源文件的访问都是成功的。同时，又再次检查了资源文件的安装，并没有发现异常。

走投无路的时候，只有通过远程调试来检查了。跟客户交流后，客户同意在把晚上 10 点到早上 8 点的时间段留给我们来排错。为了排除干扰，用下面的步骤来检查:

1. 在 IIS 上限制只有测试用客户端 IP 才能访问。
2. 把 IIS 和 ASP.NET 的超时时间都设定到无限长。
3. 重新启动 IIS
4. 在客户端刷新一次页面，然后再调试器中检查 GetString 和 ResourceManager 的执行过程。

限制 IP 的目的是为了控制重现问题的时机，防止有其他用户来访问页面。重新启动 IIS 是必须的。因为要观察 HashTable 的初始化过程。设定超时时间是为了防止超时异常干扰调试。在连续几天的努力后，发现下面的线索:

1. 加载卫星文件的时候失败
2. 加载过程中，有两次 first chance CLR exception
3. 问题发生的时候执行了一些异常的 codepath。这个 codepath 上很多函数的名字比较新颖，在本地测试的时候，没看到类似的函数。

根据上面的结果，重新在本地检查对应的代码，然后结合 wt 命令和异常发生的时机，抓取某些 codepath 的执行过程。分析后发现一个比较重要的线索，客户那里部署了.NET Framework 1.1 SP1，一些 SP1 中新加入的 codepath 被执行到了。

经过最后的整理和分析，发现了问题是这样。 在.NET Framework 1.1 上，ResourceManager 判断当前线程的 UI Culture 后就开始寻找对应的卫星文件。但是 SP1 中添加了某一个额外的配置选项，使得寻找的过程可以通过 web.configlai 来配置。由于 web.config 配置文件很重要，普通的 web 用户是没有权限访问的，于是在访问 web.config 过程中，发生了 Access Denied，继而导致了 first chance CLR exception。异常发生后，ResourceManager 认为某些地方出了问题，于是 ResourceManager 停止加载卫星文件，用中立资源代替，于是英文就代替了中文。

换句话说，程序员用 web 用户的当前身份去访问 web.config，没有考虑到 web 程序中，没有权限去访问 web.config 是很常见的，所以导致了问题。

在客户重新启动 IIS 后，如果程序第一个使用者权限比较高，能够访问 web.config 的话，中文资源就可以加载起来。如果第一个使用者权限比较低，问题就发生了，而且会一直保留到下一次 IIS 重新启动再次加载资源的时候。暂时的解决方法是把 web.config 设定为 Everyone Access Allow. 这个 workaround 在一个星期内迅速帮助了全球范围内很多 SPS 站点的显示正常起来。

当然，修改 web.config 的权限是非常危险的，所以申请一个 hotfix 才是最终的解决方案:

> FIX: Your application cannot load resources from a satellite assembly if the impersonated user account does not have permissions to access the application .config file in the .NET Framework 1.1 Service Pack 1
> http://support.microsoft.com/?id=894092

错过的线索

如果仔细分析这个案例的解决过程，会发现排错初期距离胜利就只有一步之遥。已经想到了用 Filemon 去观察卫星文件的访问情况，但是就没想到用 Filemon 观察里面是否有 Access Denied。事后拿出当时的 Filemon 结果，查找 Access Denied, 一共发现了 30 几处。同时，当观察到该问题以前一直没有，突然像洪水一样全球泛滥，就应该意识到最近一段时间发布的一些重要补丁很可疑。

所以，如果想做得更好一点，不在于使用了多少工具，用了多么深奥的调试技术，而在于能够比经验主义多想到一点什么。开拓一下思路，可以很简单地四两拔千斤。

写在本章结束前

这一章不在于演示出一套分析问题的模版，而是想说明解决问题的核心武器是思考，知识跟工具只是辅助。在遇上问题的时候能够多想一分钟再动手，本章的目的就达到了。

第二部分，重要的知识和工具

这一部分主要介绍用户态调试相关的知识和工具。包括: 汇编，异常 (exception chain)，内存布局，堆 (heap)，栈 (stack)， CRT (C Runtime)，handle/Criticalsection/thread context/windbg/dump/live debug，Dr Watson 等。

这里偏向于说明每个知识点在调试中过程中应该如何使用。知识点本身在下面两本书中有非常详细的介绍。

Advanced Windows NT
Debugging Applications for Windows

本章会用穿插使用 windbg 演示调试例子。对于 windbg 的详细介绍放到第四节。如果希望用 windbg 在本地做测试，可以先参考第四节关于 windbg 的基本配置(主要是设定 symbol 路径)

Windbg 的下载地址是:
Install Debugging Tools for Windows 32-bit Version
http://www.microsoft.com/whdc/devtools/debugging/installx86.mspx

建议安装到 C:\Debuggers 目录，后面的例子默认用这个目录

2.1 汇编，CPU 执行指令的最小单元

简介和一些资源

汇编是 CPU 执行指令的最小单元。下面一些情况下，汇编级别的分析通常是必要的:

1. 怎么看都觉得 C/C#代码没问题，但是跑出来的结果就是不对，开始怀疑编译器甚至 CPU 有毛病。
2. 没有源代码的时候.比如，调用某一个 API 的时候出问题，又没有 Windows 的源代码，那就看汇编。
3. 当程序崩溃，访问违例的时候，调试器里看到的直接信息就是汇编。

简单来说，汇编的知识分为两部分:

1. 寄存器的运算，对内存地址的寻址和读写。这部分是跟 CPU 本身相关的。
2. 函数调用时候堆栈的变化，局部变量全局变量的定位，虚函数的调用。这部分是跟编译器相关的。

汇编的知识可以在大学计算机教程里面找到。建议先熟悉简单的 8086/80286 的汇编，再结合 IA32 芯片结构和 32 位 Windows 汇编知识深入。建议的资源:

AoGo 汇编小站
http://www.aogosoft.com/

Intel Architecture Manual volume 1,2,3
http://www.intel.com/design/pentium4/manuals/index_new.htm

[案例研究, VC 的优化]

问题描述

客户开发一个性能很敏感的程序，想知道 VC 编译器对下面这段代码的优化做得怎么样:

```
int hgt=4;
int wid=7;
for (i=0; i<hgt; i++)
     for (j=0; j<wid; j++)
          A[i*wid+j] = exp(-(i*i+j*j));
```

最直接的方法就是察看编译器生成的汇编代码分析。有兴趣的话先自己调一下，看看跟我的分析是否一样

我的分析

我的分析是基于 VC6, release mode:

```
int hgt=4;
int wid=7;
24:       for (i=0; i<hgt; i++)
0040107A   xor         ebp,ebp
0040107C   lea         edi,[esp+10h]
25:        for (j=0; j<wid; j++)
26:           A[i*wid+j] = exp(-(i*i+j*j));
00401080   mov         ebx,ebp
00401082   xor         esi,esi
// The result of i*i is saved in ebx
00401084   imul        ebx,ebp
00401087   mov         eax,esi
// Only one imul occurs in every inner loop (j*j)
00401089   imul        eax,esi
// Use the saved i*i in ebx directly. !!Optimized!!
```

```
0040108C   add         eax,ebx
0040108E   neg         eax
00401090   push        eax
00401091   call        @ILT+0(exp) (00401005)
00401096   add         esp,4
// Save the result back to A[]. The addr of current offset in A[] is saved in edi
00401099   mov         dword ptr [edi],eax
0040109B   inc         esi
// Simply add edi by 4. Does not calculate with i*wid. Imul is never used. !!Optimized!!
0040109C   add         edi,4
0040109F   cmp         esi,7
004010A2   jl          main+17h (00401087)
004010A4   inc         ebp
004010A5   cmp         ebp,4
004010A8   jl          main+10h (00401080)
```

这段代码涉及到的优化有:

1. i*i 在每次内循环中是不变化的，所以只需要在外循环里面重新计算。编译器把外循环计算好的 i*i 放到 ebx 寄存器中，内循环直接使用
2. 对 A[i*wid+j]寻址的时候，在内循环里面，变化的只有 j，而且每次 j 都是增加 1，由于 A 是整型数组，所以每次寻址的变化就是增加 1*sizeof(int)，就是 4. 编译器把 i*wid+j 的结果放到了 EDI 中，在内循环中每次 add edi,4 来实现了这个优化。
3. 对于中间变量，编译器都是保存在寄存器中，并没有读写内存。

如果这段汇编让你手动来写，你能做得编译器更好一点吗？

[案例研究, 编译器的 bug]

不要迷信编译器没有 bug。 如果你在 VS2003 中测试下面的代码，会发现在 release mode 下面，程序会崩溃或者异常，但是 debug mode 下工作正常。

例子程序

```
// The following code crashes/abnormal in release build when "whole program optimizations /GL"
// is set. The bug is fixed in VS2005

#include <string>
#pragma warning( push )
#pragma warning( disable : 4702 )     // unreachable code in <vector>
#include <vector>
#pragma warning( pop )
```

```
#include <algorithm>
#include <iostream>

//vcsig
// T = float, U = std::cstring
template <typename T, typename U>    T func_template( const U & u )
{
    std::cout<<u<<std::endl;
    const char* str=u.c_str();
    printf(str);
    return static_cast<T>(0);
}

void crash_in_release()
{
    std::vector<std::string>    vStr;

    vStr.push_back("1.0");
    vStr.push_back("0.0");
    vStr.push_back("4.4");

    std::vector<float> vDest( vStr.size(), 0.0 );

    std::vector<std::string>::iterator _First=vStr.begin();
    std::vector<std::string>::iterator _Last=vStr.end();
    std::vector<float>::iterator _Dest=vDest.begin();

    std::transform( _First,_Last,_Dest, func_template<float,std::string> );

     _First=vStr.begin();
     _Last=vStr.end();
     _Dest=vDest.begin();


    for (; _First != _Last; ++_First, ++_Dest)
        *_Dest =  func_template<float,std::string>(*_First);
}


int main(int, char*)
{
    getchar();
    crash_in_release();
    return 0;
```

```
}
```

编译设定如下:

1. 取消 precompiled header
2. 编译选项是: /O2 /GL /D "WIN32" /D "NDEBUG" /D "_CONSOLE" /D "_MBCS" /FD /EHsc /ML /GS /Fo"Release/" /Fd"Release/vc70.pdb" /W4 /nologo /c /Wp64 /Zi /TP

跟踪汇编指令来分析

拿到这个问题后，首先在本地重现。根据下面的一些测试和分析，很有可能是编译器的 bug:

1. 程序中除了 cout 和 printf 外，没有牵涉到系统相关的 API., 所有的操作都是寄存器和内存上的操作。所以不会是环境或者系统因素导致的，可能性是代码错误(比如边界问题)或者编译器有问题。
2. 检查代码后没有发现异常。同时，如果调整一下 std::transform 的位置，在 for loop 后面调用的话，问题就不会发生。
3. 问题发生的情况跟编译模式相关。

代码中的 std::transform 和 for loop 的作用都是对整个 vector 调用 func_template 做转换。可以比较 transform 和 for loop 的执行情况进行比较分析，看看 func_template 的执行过程有什么区别。在 VS2003 里面在 main 函数设定断点，停下来后用 ctrl_alt_D 进入汇编模式单步跟踪。通过下面的分析，证明了这是编译器的 bug:

在 VisualStudio 附带的 STL 源代码中，发现 std::transform 的实现中用这样的代码来调用传入的转换函数:

```
*_Dest = _Func(*_First);
```

编译器对于该代码的处理是:

```
EAX = 0012FEA8 EBX = 0037138C ECX = 003712BC EDX = 00371338 ESI = 00371338 EDI = 003712B0
EIP = 00402228 ESP = 0012FE70 EBP = 0012FEA8 EFL = 00000297
   388:             *_Dest = _Func(*_First);
00402228 push        esi
00402229 call        dword ptr [esp+28h]
0040222D fstp        dword ptr [edi]
```

ESI 中保存的是需要传入 func_template 的参数。可以看到，使用 transform 的时候，这个参数是通过 push 指令传入 stack 给 func_template 调用的。

对于 for loop 中的*_Dest = func_template<float,std::string>(*_First);编译器是这样处理的:

```
EAX = 003712B0 EBX = 00371338 ECX = 003712BC EDX = 00000000 ESI = 00371338 EDI = 0037138C EIP
= 00401242 ESP = 0012FE98 EBP = 003712B0 EFL = 00000297
    37:              *_Dest =  func_template<float,std::string>(*_First);
00401240 mov         ebx,esi
00401242 call        func_template
<float,std::basic_string<char,std::char_traits<char>,std::allocator<char> > > (4021A0h)
00401247 fstp        dword ptr [ebp]
```

可以看到，使用 for loop 的时候，参数通过 mov 指令保存到 ebx 寄存器中传入给 func_template 调用。

最后，看一下 func_template 函数是如何来获取传入的参数：

```
004021A0 push        esi
004021A1 push        edi
    16:      std::cout<<u<<std::endl;
004021A2 push        ebx
004021A3 push        offset std::cout (414170h)
004021A8 call        std::operator<<<char,std::char_traits<char>,std::allocator<char> >
(402280h)
```

这里直接把 ebx 推入 stack，然后调用 std::cout，没有访问 stack，func_template (callee)认为参数应该是从寄存器中传入的。然而 transform 函数(caller)却把参数通过 stack 传递。于是使用 transform 调用 func_template 的时候，func_template 无法拿到正确的参数，继而导致崩溃。通过 for loop 调用的时候，func_template 就可以工作正常。

结论是编译器对参数的传入，读取处理不统一，导致了这个问题。

为何问题在 debug 模式下不发生，或者调换函数次序后也不发生，留为你的练习吧 :-P

[案例研究,DLL Hell]

客户的 ASP.NET 程序，访问任何页面都报告 Server Unavailable。观察发现，ASP.NET 的宿主 w3wp.exe 进程，每次刚启动就崩溃。通过调试器观察，崩溃的原因是访问了一个空指针。但是从 call stack 看，这里所有的代码都是 w3wp.exe 和.net framework 的代码，还没有开始执行客户的页面，所以跟客户的代码无关。通过代码检查，发现该空指针是作为函数参数从调用者(caller)传到被调用者(callee)的，当 callee 使用这个指针的时候问题发生。接下来应该检查 caller 为什么没有把正确的指针传入 callee。

奇怪的时候，caller 中这个指针已经正常初始化了，是一个合法的指针，调用 call 语句执行 callee 的以前，这个指针已经被正确地 push 到 stack 上了。为什么 caller 从 stack 上拿的时候，却拿到一个空指针呢？再次单步跟踪，发现问题在于 caller 把参数放到了 callee 的[ebp+8]，但是 callee 在使用这个参数的时候，却是访问[ebp+c]。是不是跟前一个案例很象？但是这次的凶手不是编译器，而是文件版本。Caller 和 callee 的代码位于两个不同的 DLL，其中 caller

是.NET Framework 1.1 带的，callee 是.NET Framework 1.1 SP1 带的。在.NET Framework 1.1 中，callee 函数接受 4 个参数，但是新版本 SP1 对 callee 这个函数做了修改，增加了 1 个参数。由于 caller 还使用 SP1 以前的版本，所以 caller 还是按照 4 个参数在传递，而 callee 按照 5 个参数在访问，所以拿到了错误的参数，典型的 DLL Hell 问题。在重新安装.NET Framework 1.1 SP1 让两个 DLL 保持版本一致，重新启动后，问题解决。

导致 DLL Hell 的原因有很多。根据经验猜测版本不一致的原因可能是：

1. 安装了.NET Framework 1.1 SP1 后没有重新启动，导致某些正在使用的 DLL 必须要等到重新启动后才能够完成更新
2. 由于使用了Application Center做Load Balance, 集群中的服务器没有做好正确的设置，导致系统自动把老版本的文件还原回去了：

   PRB: Application Center Cluster Members Are Automatically Synchronized After Rebooting
   http://support.microsoft.com/kb/282278/en-us

[案例研究,Release 比 Debug 快吗?]

分别在 debug/release 模式下运行下面的代码比较效率：会发现 debug 比 release 更快.你能找到原因吗？

```
long nSize = 200;
char* pSource = (char *)malloc(nSize+1);
char* pDest = (char *)malloc(nSize+1);
memset(pSource, 'a', nSize);
pSource[nSize] = '\0';
DWORD dwStart = GetTickCount();
for(int i=0; i<5000000; i++)
{
     strcpy(pDest, pSource);
}
DWORD dwEnd = GetTickCount();
printf("%d", dwEnd-dwStart);
```

如果让你自己实现一个 strcpy 函数，应该考虑什么？你能做到比系统的 strcpy 函数快吗？

http://eparg.spaces.live.com/blog/cns!59BFC22C0E7E1A76!1498.entry

从效率上说，其到决定性作用的至少有下面两点:

1. 在 32 位芯片上，应该尽量每次 mov 一个 DWORD，而不是 4 个 byte 来提高效率。注意到 mov DWORD 的时候要 4 字节对齐
2. 这里对 strcpy 的调用高达 5000000 次。由于 call 指令的开销，使用内联 (inline) 版本的 strcpy 函数可以极大提高效率。

所以汇编，CPU 习性，操作系统和编译器，是分析细节的最直接武器。

题外话和相关讨论

系统提供的 strcpy 是否内联，取决于编译设定。由于 strcpy 是 CRT (CRuntime C 运行库)函数，函数的实现位于 MSVCRT.DLL 或者 MSVCRTD.DLL。当跨越 DLL 调用函数的时候，这个函数是无法内联的。

关于性能的另外一些讨论:
http://eparg.spaces.msn.com/blog/cns!59BFC22C0E7E1A76!875.entry

2.2 异常和通知

本小结首先介绍异常的原理和相关资料，再举例说明异常跟崩溃和调试是如何紧密联系在一起的。最后说明如何利用工具来监视异常，获取准确的信息。

异常小结

异常是 CPU，操作系统和应用程序控制代码流程的一种机制。正常情况下，代码是顺序执行的，比如下面两行:

```
*p=11;
printf("%d",*p);
```

这里应该会打印出 11。 如果 p 指向的地址是无效地址呢，那么这里对*p 赋值的时候，也就是 CPU 向对应地址做写操作的时候，CPU 就会触发无效地址访问的异常，同时把异常信息保存下来。

异常发生的时候,由于操作系统在内核挂接了对应的 CPU 异常处理函数，CPU 就会跳转去执行操作系统提供的处理函数，所以 printf 就不一定会被执行了。在操作系统的处理函数里面，如果检测到异常发生在用户态的程序，操作系统会再把异常信息发送给用户态进程对应的处理函数，让用户态程序有处理异常的机会。

用户态程序处理完了异常，代码会继续执行，不过执行的次序可以是紧接着的下一个指令，比如 printf，也可是跳到另外的地址开始执行，比如 catch block，这些都是用户态的异常处理函数可以控制的。

如果用户态程序没有处理这个异常，那操作系统的默认行为就是中止程序的执行，然后用户可以看到给 Microsoft 发送错误报告，或者干脆就是一个红色的框框说某某地址上的指令在访问某某地址的时候遭遇了访问违例的错误.

除了上面的非预期异常，也可以手动触发异常来控制执行顺序，C++/C# 中的 throw 关键字就可以触发异常。往往会用 throw 来快速结束很深的函数调用，把控制权很快返还给最外层的函数，而且局部变量的析构函数还会自动被调用。这比一层一层的 return ERROR 方便得多。手动触发异常需要依赖于编译器和操作系统 API 来实现。

异常的类型，是通过异常代码来标示的。比如访问无效地址的号码是 0xc0000005,而 C++异常的号码是：0xe06d7363。其它很多看似跟异常无关的东西，其实都是跟异常联系在一起的，比如调试的时候设置断点，或者单步执行，都有通过 break point exception 来实现的。越权指令，堆栈溢出的处理也依靠异常。在 windbg 帮助文件的 Controlling Exceptions and Events 主题里面，有一张常用异常代码表。

程序的行为跟预期的不一样，直接原因是代码执行次序跟预期的不一样。异常改变了代码执行次序,比如代码中从来都没有什么函数去跳一个红框框出来说某某地址上的指令在访问某某地址的时候遭遇了访问违例。弄清楚异常发生的时间，地址，导致异常的指令和异常导致的结果对排错是至关重要的。

既然异常如此重要，操作系统提供了对应的调试功能。可以使用调试器来检视异常。异常发生后，操作系统在调用用户态程序的异常处理函数前，会检查当前用户态程序是否有调试器加载。如果有，那么操作系统会首先把异常信息发送给调试器，让调试器有观察异常的第一次机会，所以也叫做 first chance exception，调试器处理完毕后，操作系统才让用户态程序来处理。

如果用户态程序处理了这个异常，就没调试器什么事了.否则，在程序因为 unhandled exception 崩溃前，操作系统会再给调试器第二次观察异常的机会，所以也叫做 second chance exception。

请注意，这里的 1st chance, 2nd chance 是针对调试器来说的。虽然 C++异常处理的时候也会有 first phrase find exception handler, second phrase unwind stack 这样的概念，但是两者是不一样的。

操作系统提供的异常处理功能叫做 Structrued Exception Handle(SEH), C++和其它高级语言的异常处理机制都是建立在 SEH 上的.如果要直接使用 SEH,可以在 C/C++中使用__try, __except 关键字.

关于异常处理得详细信息，所有的来龙去脉，操作系统做了些什么事情，C++编译器作了些什么事情，SEH 和 C++异常处理的关系,以及调试器是如何参与的，下面几篇文章有非常详细的介绍。

A Crash Course on the Depths of Win32™ Structured Exception Handling
http://www.microsoft.com/msj/0197/Exception/Exception.aspx
这篇文章出来后,再没见人写第二篇了.深入浅出,字字珠玑

RaiseException
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/debug/base/raiseexception.asp
注意,上面链接中 remark section 详细介绍了异常处理函数是如何被分发的

[案例研究,让 C++打印出 callstack]

如果用 C#或者 Java，在异常发生后，可以获取异常发生时刻的 call stack。但是对于 C++, 除非使用调试器，否则是看不到的。现在用户想尽可能少地修改代码，让 C++程序在发生异常崩溃后，能够打印出 call stack，有什么方法呢？

我的解法是直接使用 SHE，加上局部变量析构函数在异常发生时候会被执行的特点来完成。这个例子当时使用 VC6 在 Windows 2003 上调试通过。当重新整理这个例子的时候，发现在段代码在 VC2005+Windows 2003 SP1 上有奇怪的现象发生。如果用 debug 模式编译，运行

正常。如果用 release 模式编译，程序会在没有任何异常报告的情况下悄然退出。关于整个源代码和对应的分析，请参考:

SEH,DEP, Compiler,FS:[0] and PE format
http://eparg.spaces.msn.com/blog/cns!59BFC22C0E7E1A76!712.entry

[案例研究, 华生医生和 Dump 文件]

问题描述

客户声称用 VC 开发的程序偶尔会崩溃。为了获取详细信息，客户激活了 Dr. Watson，以便程序崩溃的时候可以自动获取 dump 文件。但是问题再次发生后，Dr. Watson 并没有记录下来 dump 文件。

背景知识

dump 文件包含的是内存镜像信息。在 Windows 系统上，dump 文件分为内核 dump 和用户态 dump 两种。前者一般用来分析内核相关的问题，比如驱动程序；后者一般用来分析用户态程序的问题。如果不作说明，本文后面所指的 dump 都表示用户态 dump。用户态的 dump 又分成 mini dump 和 full dump。前者尺寸小，只记录一些常用信息；后者则是把目标进程用户态所有内容都记录下来。Windows 提供了 MiniDumpWriteDump API 可供程序调用来生成 mini dump。通过调试器和相关工具，可以抓取目标程序的 full dump。拿到 dump 后，可以通过调试器检查 dump 中的内容，比如 call stack，memory，exception 等等。关于 dump 和调试器的更详细信息，后面会有更多介绍。跟 Dr. Watson 相关的文档是:

Description of the Dr. Watson for Windows (Drwtsn32.exe) Tool
http://support.microsoft.com/?id=308538

Specifying the Debugger for Unhandled User Mode Exceptions
http://support.microsoft.com/?id=121434

INFO: Choosing the Debugger That the System Will Spawn
http://support.microsoft.com/?id=103861

也就是说，通过设定注册表中的 AeDebug 项，可以在程序崩溃后，选择调试器进行调试。选择 Dr. Watson 就可以直接生成 dump 文件。

问题分析

回到这个问题，客户并没有获取到 dump 文件，可能性有两个:

1. Dr. Watson 工作不正常
2. 客户的程序根本没有崩溃，不过是正常退出而已

为了测试第一点，提供了如下的代码给客户测试:

```
int *p=0;
*p=0;
```

测试上面的代码,Dr. Watson 成功地获取了 dump 文件。也就是说，Dr. Watson 工作是正常的。那看来客户声称的崩溃可能并不是 unhandled exception 导致的。说不定非预料情况下的调用 ExitProcess 退出，被客户误认为是崩溃。所以，当抓取信息的时候不应该局限于 unhandled exception，而应该检查进程退出的原因。

当程序在 windbg 调试器中退出的时候，系统会触发调试器的进程退出消息，可以在这个时候抓取 dump 来分析进程退出的原因。

如果让客户每次都先启动 windbg，然后用 windbg 启动程序，操作起来很复杂。最好有一个自动的方法。Windows 提供了让指定程序随带指定调试器启动的选项。设定注册表后，当设定的进程启动的时候，系统先启动设定的调试器，然后把目标进程的地址和命令行作为参数传递给调试器，调试器再启动目标进程调试。这个选项在无法手动从调试器中启动程序的时候特别有用，比如调试先于用户登录而启动的 Windows Service 程序，就必须使用这个方法：

How to debug Windows services
http://support.microsoft.com/?kbid=824344

在 Windbg 目录下，有一个叫做 adplus.vbs 的脚本可以方便地调用 windbg 来获取 dump 文件。所以这里可以借用这个脚本:

How to use ADPlus to troubleshoot "hangs" and "crashes"
http://support.microsoft.com/kb/286350/EN-US/

详细内容可以参考 adplus /?的帮助

新的做法

结合上面的信息，具体做法是:

1. 在客户机器的 Image File Execution Options 注册表下面创建跟问题程序同名的键
2. 在这个键的下面创建 Debugger 字符串类型子键
3. 设定 Debugger= C:\Debuggers\autodump.bat
4. 编辑 C:\Debuggers\autodump.bat 文件的内容为如下:

cscript.exe C:\Debuggers\adplus.vbs -crash -o C:\dumps -quiet -sc %1

通过上面的设置，当程序启动的时候，系统自动运行 cscript.exe 来执行 adplus.vbs 脚本。Adplus.vbs 脚本的-sc 参数指定需要启动的目标进程路径 (路径作为参数又系统传入，bat 文件中的 1%代表这个参数)，-crash 参数表示监视进程退出，-o 参数指定 dump 文件路径，-quiet 参数取消额外的提示。可以用 notepad.exe 作为小白鼠做一个实验，看看关闭 notepad.exe 的时候，是否有 dump 产生。

根据上面的设定，在问题再次发生后，C:\dumps 目录生成了两个 dump 文件。文件名分别是:

PID-0__Spawned0__1st_chance_Process_Shut_Down__full_178C_DateTime_0928.dmp
PID-0__Spawned0__2nd_chance_CPlusPlusEH__full_178C_2006-06-21_DateTime_0928.dmp

注意看第二个的名字，这个名字表示发生了 2nd chance 的 C++ exception！打开这个 dump 后找到了对应的 call stack，发现的确是客户忘记了 catch 潜在的 C++异常。修改代码添加对应的 catch 后，问题解决。

问题解决了，可是为什么 Dr. Watson 抓不到 dump 呢

当然疑问并没有随着问题的解决而结束。既然是 unhandled exception 导致的 crash，为什么 Dr. Watson 抓不到呢？首先创建两个不同的程序来测试 Dr. Watson 的行为:

```
int _tmain(int argc, _TCHAR* argv[])
{
    throw 1;
    return 0;
}
int _tmain(int argc, _TCHAR* argv[])
{
    int *p=0;
    *p=0;
    return 0;
}
```

果然，对于第一个程序，Dr. Watson 并没有保存 dump 文件。对于第二个，Dr. Watson 工作正常。看来的确跟异常类型相关。

仔细回忆一下。当 AeDebug 下的 Auto 设定为 0 的时候，系统会弹出前面提到的红色框框。对于上面这两个程序，框框的内容是不一样的：

在我这里，看到的对话框分别是 (对话框出现的时候用 ctrl+c ctrl+v 保存的信息):

Microsoft Visual C++ Debug Library

Debug Error!

Program: d:\xiongli\today\exceptioninject\debug\exceptioninject.exe

This application has requested the Runtime to terminate it in an unusual way.
Please contact the application's support team for more information.

(Press Retry to debug the application)

Abort Retry Ignore

---
exceptioninject.exe - Application Error

The instruction at "0x00411908" referenced memory at "0x00000000". The memory could not be "written".

Click on OK to terminate the program
Click on CANCEL to debug the program

OK Cancel

而且这个对话框的细节还跟编译模式 release/debug 相关。

研究后发现，程序可以通过 SetUnhandledExceptionFilter 函数来修改 unhanded exception 的默认处理函数。这里，C++运行库在初始化 CRT (C Runtime)的时候，传入了 CRT 的处理函数(msvcrt!CxxUnhandledExceptionFilter)。如果发生 unhandled exception，该函数会判断异常的号码，如果是 C++异常，就会弹出第一个对话框，否则就交给系统默认的处理函数(kernel32!UnhandledExceptionFilter)处理。第一种情况的 call stack 如下:

```
USER32!MessageBoxA
MSVCR80D!__crtMessageBoxA
MSVCR80D!__crtMessageWindowA
MSVCR80D!_VCrtDbgReportA
MSVCR80D!_CrtDbgReportV
MSVCR80D!_CrtDbgReport
MSVCR80D!_NMSG_WRITE
MSVCR80D!abort
```

```
MSVCR80D!terminate
MSVCR80D!__CxxUnhandledExceptionFilter
kernel32!UnhandledExceptionFilter
MSVCR80D!_XcptFilter
```

第二种情况 CRT 交给系统处理。Callstack 如下:

```
ntdll!KiFastSystemCallRet
ntdll!ZwRaiseHardError+0xc
kernel32!UnhandledExceptionFilter+0x4b4
release_crash!_XcptFilter+0x2e
release_crash!mainCRTStartup+0x1aa
release_crash!_except_handler3+0x61
ntdll!ExecuteHandler2+0x26
ntdll!ExecuteHandler+0x24
ntdll!KiUserExceptionDispatcher+0xe
release_crash!main+0x28
release_crash!mainCRTStartup+0x170
kernel32!BaseProcessStart+0x23
```

详细的信息可以参考:

SetUnhandledExceptionFilter
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/debug/base/setunhandledexceptionfilter.asp
UnhandledExceptionFilter
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/debug/base/unhandledexceptionfilter.asp

上面观察到的信息对于解释 Dr. Watson 的行为有帮助吗？说实话，我看不出来。我认为不能获取 C++ unhandled exception dump 的行为是 Dr. Watson 特有的。因为通过下面的测试，使用 windbg 代替 Dr. Watson，是可以获取 dump 的:

1. 运行 drwtsn32.exe –i 注册 Dr. Watson
2. 打开 AeDebug 注册表，找到 Debugger 项，里面应该是 drwtsn32 -p %ld -e %ld -g
3. 修改 Debugger 为: C:\debuggers\windbg.exe -p %ld -e %ld -c ".dump /mfh C:\myfile.dmp ;q"

当 unhanded exception 发生后，系统会启动 windbg.exe 作为调试器加载到目标进程。但是 windbg.exe 不会自动获取 dump，所以需要用-c 参数来指定初始命令。命令之间可以用分开分割。这里的.dump /mfh C:\myfile.dmp 命令就是用来生成 dump 文件的。接下来的 q 命令是让 windbg.exe 在 dump 生成完毕后自动退出。用这个方法，对于 unhandled C++ exception，windbg.exe 是可以获取 dump 文件的。所以，我认为 Dr. Watson 这个工具在获取 dump 的时候是有缺陷的。研究的发现在:

http://eparg.spaces.msn.com/blog/cns!59BFC22C0E7E1A76!1213.entry

通知(notification)

MSDN 中没有多少关于 notification 的信息。根据我的理解，通知是操作系统在某些事情发生的时候，通知调试器的一个手段。跟异常处理相似，操作系统在某些事件发生的时候，会检查当前进程是否有调试器加载。如果有，就会给调试器发送对应的消息，以便使用调试器进行观察。跟异常不一样的地方就是，只有调试器才会得到通知，应用程序本身是得不到的。同时调试器得到通知后不需要做什么处理，没有 1st/2nd chance 的差别。在 windbg 帮助文件的 Controlling Exceptions and Events 主题里面，可以看到关于通知的所有代号。常见的通知有：DLL 的加载，卸载。线程的创建，退出等。

使用异常和通知能够非常准确地抓到问题的关键。

[案例研究,VB6 的版本问题]

客户用 VB6 开发程序，VB6 IDE 调试的时候无法访问 Access 2003 创建的数据库，访问 Access 97 的数据库却是好的。如果换一台机器测试就一切正常。

这个问题的思路非常简单，既然只有一台机器有问题，说明是环境的原因。既然访问 Access 97 没问题，或许跟 Access 客户端文件，也就是 DAO 的版本有关。通过工具 tlist 工具检查进程中加载的 DLL， 发现有问题的机器加载的是 dao350.dll，没有问题的机器加载的是 dao360.dll。下一步就需要知道为什么加载的是 dao350.dll?

DAO 是一个 COM 对象，很有可能是通过 COM 对象加载的方法完成得。那么，可以采取 ShellExecute 那个问题的处理方法，从创建 COM 的 API：CoCreateInstanceEx 开始，用 wt 命令跟踪整个函数的执行，保存下来后比较两种不同情况的异同。通过这个方法肯定是可以找出原因的，不过要想用 wt 命令一直跟踪到 LoadLibrary 函数加载这个 DLL，可能需要执行一整天。所以，应该找一个可操作性更强一点的方法来检查。既然最后要追踪到 LoadLibrary 为止，那何不在这个函数上设置断点，观察检查 DAO350.DLL 加载起来的情况？

在 LoadLibrary 上设定断点并不是一个很好的方法。因为：

1. 加载 DLL 不一定要调用 LoadLibrary 的。可以直接调用 Native API，比如 ntdll!LdrLoadDll
2. 假设有几十个 DLL 要加载，如果每次 LoadLibrary 都断下来，操作起来也是很麻烦的事情。虽然可以通过条件断点来判断 LoadLibrary 的参数来决定是否断下来，但是设定条件断点也是很麻烦的。

最好的方法，就是使用通知，在 moudle load 的时候，系统给调试器发送通知。由于 windbg 在收到 moudle load 通知的时候，可以使用通配符来判断 DLL 的名字，操作起来就简单多了。首先，在 windbg 中用 sxe ld:dao*.dll 设置截获 Moudle Load 的通知，当文件名是 dao*.dll 的时候，windbg 就会停下来。(对于 windbg 的详细信息，以及这里使

用到的命令，后面都有章节详细介绍）。看到的结果就是：

```
0:008> sxe ld:dao*.dll

ModLoad: 1b740000 1b7c8000 C:\Program Files\Common Files\Microsoft
Shared\DAO\DAO360.DLL
eax=00000001 ebx=00000000 ecx=0013e301 edx=00000000 esi=7ffdf000 edi=20000000
eip=7c82ed54 esp=0013e300 ebp=0013e344 iopl=0 nv up ei pl zr na po nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000246
ntdll!KiFastSystemCallRet:
7c82ed54 c3 ret

ntdll!KiFastSystemCallRet
ntdll!NtMapViewOfSection
ntdll!LdrpMapViewOfDllSection
ntdll!LdrpMapDll
ntdll!LdrpLoadDll
ntdll!LdrLoadDll
0013e9c4 776ab4d0 0013ea40 00000000 00000008 kernel32!LoadLibraryExW
ole32!CClassCache::CDllPathEntry::LoadDll
ole32!CClassCache::CDllPathEntry::Create_rl
ole32!CClassCache::CClassEntry::CreateDllClassEntry_rl
ole32!CClassCache::GetClassObjectActivator
ole32!CClassCache::GetClassObject
ole32!CServerContextActivator::GetClassObject
ole32!ActivationPropertiesIn::DelegateGetClassObject
ole32!CApartmentActivator::GetClassObject
ole32!CProcessActivator::GCOCallback
ole32!CProcessActivator::AttemptActivation
ole32!CProcessActivator::ActivateByContext
ole32!CProcessActivator::GetClassObject
ole32!ActivationPropertiesIn::DelegateGetClassObject
ole32!CClientContextActivator::GetClassObject
ole32!ActivationPropertiesIn::DelegateGetClassObject
ole32!ICoGetClassObject
ole32!CComActivator::DoGetClassObject
ole32!CoGetClassObject
VB6!VBCoGetClassObject
VB6!_DBErrCreateDao36DBEngine
```

通过检查 LoadLibraryExW 的参数，可以看到:

```
0:000> du 0013ea40
0013ea40  "C:\Program Files\Common Files\Mi"
```

```
0013ea80  "crosoft Shared\DAO\DAO360.DLL"
```

从上面的信息可以看到:

1. DAO360 不是通过 CoCreateInstanceEx 加载进来的，而是另外一个 COM API: CoGetClassObject。所以如果对 CoCreateInstanceEx 做想当然的跟踪，就浪费时间了
2. COM 调用的发起者是 VB6!_DBErrCreateDao36DBEngine 这个函数。应该仔细检查这个函数。

有了前面DLL HELL 案例的教训，在检查这个函数前，首先检查VB6.EXE的版本。发现正常情况下的版本是6.00.9782,有问题的机器上的版本是: 6.00.8176。 在有问题的机器上安装Visual Studio 6 SP6升级VB6版本后，问题解决。

题外话和相关讨论

错过第一现场还能分析 dump 吗

前面介绍了用 windbg 截取 1st chance exception 进行分析的方法。

但是好多情况下，程序并没有运行在调试器下。崩溃发生后留在桌面上的是红色的框框，这时候已经错过了第一现场，但是还是有机会找到对应 exception 的信息。

前面介绍过，红色的框框是通过 UnhandledExceptionFilter 函数显示出来的，而 UnhandledExceptionFilter 的参数就包含了异常信息。这个时候检查 UnhandledExceptionFilter 的参数就可以异常信息和异常上下文的地址，然后通过.exr 和.cxr 就可以在 windbg 中把对应信息打印出来:

拿案例 2 中的第二个例子做一个实验。直接运行，崩溃后在弹框的时候加载 windbg, 然后用 kb 命令打印出 call stack，找到 UnhandledExceptionFilter 的参数:

```
0:000> kb
ChildEBP RetAddr  Args to Child
0012f74c 7c821b74 77e999ea d0000144 00000004 ntdll!KiFastSystemCallRet
0012f750 77e999ea d0000144 00000004 00000000 ntdll!ZwRaiseHardError+0xc
0012f9bc 004339be 0012fa08 7ffdd000 0044c4d8
kernel32!UnhandledExceptionFilter+0x4b4
```

第一个参数 0012fa08 保存的就是异常信息和异常上下文的个地址:

```
0:000> dd 0x0012fa08
0012fa08  0012faf4 0012fb10 0012fa34 7c82eeb2
```

接下来用.exr 打印出异常的信息:

```
0:000> .exr 0012faf4
ExceptionAddress: 0041a5a8 (release_crash!main+0x00000028)
   ExceptionCode: c0000005 (Access violation)
  ExceptionFlags: 00000000
NumberParameters: 2
   Parameter[0]: 00000001
   Parameter[1]: 00000000
Attempt to write to address 00000000
```

然后可以用.cxr 来切换上下文:

```
0:000> .cxr 0012fb10
eax=00000000 ebx=7ffde000 ecx=00000000 edx=00000001 esi=00000000 edi=0012fedc
eip=0041a5a8 esp=0012fddc ebp=0012fedc iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010206
release_crash!main+0x28:
0041a5a8 c60000          mov     byte ptr [eax],0x0      ds:0023:00000000=??
```

上下文切换完成后，可以用 kb 命令重新打印出该上下文上的 call stack，看到异常发生时候的状态:

```
0:000> kb
  *** Stack trace for last set context - .thread/.cxr resets it
ChildEBP RetAddr  Args to Child
0012fedc 00427c90 00000001 00361748 003617d0 release_crash!main+0x28
[c:\documents and settings\lixiong\desktop\amobrowser\release_crash.cpp @ 51]
0012ffc0 77e523cd 00000000 00000000 7ffde000
release_crash!mainCRTStartup+0x170
0012fff0 00000000 00418b18 00000000 78746341 kernel32!BaseProcessStart+0x23
```

Adplus

如果要捕获崩溃时候的详细信息，通常可以在调试器下运行程序，或者使用更方便的adplus来自动获取异常产生时候的dump文件。可以参考：

How to use ADPlus to troubleshoot "hangs" and "crashes"
http://support.microsoft.com/kb/286350/

快死吧(FailFast)

在某些特殊情况下，程序员为了需要，会在截获异常后主动退出，而不是等到崩溃被动发生。使用这种技术的有 COM+, ASP.NET，还有淘宝旺旺客户端。

这样做的好处是:

1. 可以自定义界面
2. 可以主动获取异常的一些信息保存下来以便后继分析

实现方法非常简单。一种方法是在程序的 main 函数，或者关键函数中，使用 SEH 的__try 和__except 语句捕获所有的异常。在__except 语句中做相应的操作后(比如显示 UI，保存信息)直接退出程序。

另外一种方法是使用 SetUnhandledExceptionFilter。有很多程序有崩溃后发送异常报告的功能。淘宝旺旺客户端就是这样的一个例子，可以参考:

http://eparg.spaces.msn.com/blog/cns!59BFC22C0E7E1A76!817.entry

根据我的分析， taobao 这里用了 SetUnhandledExceptionFilter 这个函数来定义自己的异常处理函数，在异常处理函数中通过 MiniDumpWriteDump API 实现 dump 的捕获。

使用 failfast 的缺点就是调试器无法接收到 2nd chance exception 了，给调试增加了难度。比如要获取 COM+程序上 crash 的信息，颇费一番周则，还需要使用上面提高的.exr/.cxr 命令:

How To Obtain a Userdump When COM+ Failfasts
http://support.microsoft.com/?id=287643

How to find the faulting stack in a process dump file that COM+ obtains
http://support.microsoft.com/?id=317317

如何调试 SetUnhandledExceptionFilter

根据 MSDN 的描述，UnhandledExceptionFilter 在没有 debugger attach 的时候才会被调用。所以，SetUnhandledExceptionFilter 函数还有一个妙用，就是让某些敏感代码避开 debugger 的追踪。比如你想把一些代码保护起来，避免调试器的追踪，可以采用的方法:

1. 在代码执行前调用 IsDebuggerPresent 来检查当前是否有调试器加载上来。如果有，就退出。
2. 把代码放到 SetUnhandledExceptionFilter 设定的函数里面。通过人为触发一个 unhandled exception 来执行。由于 设定的 UnhandledExceptionFilter 函数只有在调试器没有加载的时候才会被系统调用，这里巧妙地使用了系统的这个功能来保护代码。

不建议第一种方法。看看 IsDebuggerPresent 的实现:

```
0:000> uf kernel32!IsDebuggerPresent
kernel32!IsDebuggerPresent:
  281 77e64860 64a118000000     mov     eax,fs:[00000018]
  282 77e64866 8b4030           mov     eax,[eax+0x30]
  282 77e64869 0fb64002         movzx   eax,byte ptr [eax+0x2]
  283 77e6486d c3               ret
```

IsDebuggerPresent 是通过返回 FS 寄存器上记录的地址的一些偏移量来实现的。([FS:[18]]:30 保存的其实是当前进程的 PEB 地址)。在 debugger 中可以任意操作当前进程内存地址上的值，所以只需要用调试器把[[FS:[18]]:30]:2 的值修改成 0，IsDebuggerPresent 就会返回 false，导致方法 1 失效

对于第二种方法，使用[[FS:[18]]:30]:2 的欺骗方法就没用了。因为 UnhandledExceptionFilter 是否调用取决于系统内核的判断。用户态的调试器要想改变这个行为，要破费一番脑精了。**Kwan Hyun Kim** 提供了一种欺骗系统的方法:

How to debug UnhandleExceptionHandler

http://eparg.spaces.msn.com/blog/cns!59BFC22C0E7E1A76!1208.entry

2.3 内存，Heap，Stack 的一些补充讨论

本小结主要介绍 Heap 可能导致的崩溃和内存泄露，以及如何使用 pageheap 来排错。首先介绍 Heap 的原理，不同层面的内存分配，接下来通过例子代码演示 heap 问题的严重性和欺骗性。最后介绍 pageheap 工具如何高效地对 heap 问题排错。

内存/Heap 小结

内存是容纳代码和数据的空间。无论是 stack, heap 还是 DLL,都是生长在内存上的。代码是对内存上的数据做变化，同时根据内存上的数据，决定下一条代码的地址。内存是导致问题最多的地方，比如内存不足，内存访问违例，内存泄漏等，都是常见的问题。

关于内存的详细信息， Advanced Windows NT 中有详细介绍介绍。这里针对排错做一些补充。

Windows API 中有两类内存分配函数。分别是：VirtualAlloc 和 HeapAlloc。前一种是向操作系统申请 4k 为边界的整块内存，后者是分配任意大小的内存块。区别在于，后者依赖于前者实现的。换句话说，操作系统管理内存的最小单位是 4k, 这个粒度是固定的(其实根据芯片是可以做调整的。这里只讨论最普遍的情况)。但用户的数据不可能恰好都是 4k 大小，用 4k 做单位，难免会产生很多浪费。解决办法是依靠用户态代码的薄计工作，实现比 4k 单位更小的分配粒度。换句话说，用户态的程序需要实现一个 Memory Manager，通过自身的管理，在 4k 为粒度的基础上，提供以字节为粒度的内存分配，释放功能，并且能够平衡好时间利用率和空间利用率。

Windows 提供了 Heap Manager 完成上述功能。HeapAlloc 函数是 Heap Manager 的分配函数。Heap Manager 的工作方式大概是这样。首先分配足够大的，4k 倍数的连续内存空间。然后在这块内存上开辟一小块区域用来做薄计。接下来继续把这一大块内存分割成很多小块，每一小块尺寸不等，把每一小块的信息记录到薄计里面。薄计记录了每一小块的起始地址和长度，以及是否已经分配。

当接收到一个用户的内存请求，HeapManager 根据请求的长度，在薄计信息里面找到大小最合适的一块空间，把这块空间标记成已经分配，然后把这块空间的起始地址返回，这样就完成了一次内存分配。如果找不到合适的小块，Heap Manager 继续以 4k 为粒度向系统申请更多的内存。

当用户需要释放内存的时候，调用 HeapFree，同时传入起始地址。HeapManager 在薄计信息中找到这块地址，把这块地址的信息由已经分配改回没有分配。当 Heap Manager 发现有大量的连续空闲空间的时候，也会调用 VirtualFree 来把这些内存归还给操作系统。在实现上面这些基本功能的情况下，HeapManager 还需要考虑到;

1．分配的内存最好是在 4 字节边界上，这样可以提高内存访问效率。

2．做好线程同步，保证多个线程同时分配内存的时候不会出现错误。
3．尽可能节省维护薄计的开销，提高性能，避免作多于得检查。所以 HeapManager 假设用户的代码没有 bug，比如用户代码永远不会越界对内存块进行存取。

有了上面的理解后，看下面一些情况：

1. 如果首先用 HeapAlloc 分配了一块空间，然后用 HeapFree 释放了这块空间。但是在释放后，继续对这块空间做操作，程序会发生访问违例错误吗？答案是不会，除非 HeapManager 恰好把那块地址用 VirtualFree 返还给操作系统了。但是带来的结果是什么？是“非预期结果”。也就是说，谁都无法保证最后会产生什么情况。程序可能不会有什么问题，也可能会格式化整个硬盘。出现得最多的情况是，这块内存后来被 Heap Manager 重新分配出去。导致两个本应指向不同地址的指针，指向同一个地址。伴随而来的是数据损坏或者访问违例等等
2. 如果用 HeapAlloc 分配了 100k 的空间，但是访问的长度超过了 100k，会怎么样？如果 100k 恰好在 4k 内存边界上，而且恰好后面的内存地址并没有被映射上来，程序不会崩溃的。这时，越界的写操作，要么写到别的内存块上（如果幸运，可能内存块还没有被分配出去），要么就写入薄计信息中，破坏了薄计。导致的结果是 HeapManager 维护的数据损坏，导致“非预期结果”。
3. 其它错误的代码，比如对同一个地址 HeapFree 了两次，多线程访问的时候忘记在调用 HeapAllocate 的第二个参数中传入 SERIALIZE bit 等等，都会导致“非预期结果”。

总的来说，上面这些情况导致的结果非预期的。如果问题发生后程序立刻崩溃，或者抛出异常，则可以在第一时间截获这个错误。但是，现实的情况是这些错误不会有及时的效果，错误带来的后果会暂时隐藏起来，在程序继续执行几个小时后，突然在一些看起来绝对不可能出现错误的地方崩溃。比如在调用 HeapAllocate/HeapFree 的时候崩溃。比如访问一个刚刚分配好的地址的时候崩溃。这个时候哪怕抓到了崩溃的详细信息也无济于事，因为问题潜伏在很久以前。这种根源在前，现象在后的情况会给调试带来极大的困难。

仔细考虑这种难于调试的情况，错误之所以没有在第一时间暴露，在于下面两点：

1. Heap 每一块内存的界限是 Heap Manager 定义的，而内存访问无效的界限，是操作系统定义的。哪怕访问越界，如果越界的地方已经有映射上来的 4k 为粒度的内存页，程序就不会立刻崩溃。
2. 为了提高效率，Heap Manager 不会主动检查自身的数据结构是否被破坏。

所以，为了方便检查 heap 上的错误，让现象尽早表现出来，Heap Manager 应该这样管理内存：

1. 把所有的 heap 内存都分配到 4k 页的结尾，然后把下一个 4k 页面标记为不可访问。越界访问发生时候，就会访问到无效地址，程序就立刻崩溃
2. 每次调用 Heap 相关函数的时候，HeapManager 主动去检查自身的数据结构是否被破坏。如果检查到这样的情况，就主动报告出来。

PageHeap & Gflag

幸运的是, HeapManager 的确提供了上述的主动错误检查功能。只需要在注册表里面做对应的修改，操作系统就会根据设置来改变 HeapManager 的行为。Pageheap 是用来配置该注册

表的工具。关于 heap 的详细信息和原理请参考：

How to use Pageheap.exe in Windows XP and Windows 2000
http://support.microsoft.com/kb/286470/en-us

Pageheap，Gflag 和后面介绍的 Application Verifier 工具一样，都是方便修改对应注册表的工具。如果不使用这两个工具直接修改注册表也可以达到一样的效果。三个工具里面 Application Verifier 是目前主流，Gflag 是老牌。除了 heap 问题外，这两个工具还可以修改其它的调试选项，后面都有说明。Pageheap.exe 工具主要针对 heap 问题，使用起来简单方便。目前 gflag.exe 包含在调试器的安装包中，Application Verifier 可以单独下载安装。Pageheap 的下载在 MSDN 上找不到了。我把 pageheap.exe 放到这里:
http://www.heijoy.com/debugdoc/pageheap.zip

简单例子的多种情况

看几个简单的，但是却很有说明意义的例子：

用 release 模式编译运行下面的代码：

```
char *p=(char*)malloc(1024);
p[1024]=1;
```

这里往分配的空间多写一个字节。代码本身是有问题的，在 release 模式下运行，程序不会崩溃。

假设上面的代码编译成 mytest.exe，然后用下面的方法来对 mytest.exe 激活 pageheap:
C:\Debuggers\pageheap>pageheap /enable mytest.exe /full

C:\Debuggers\pageheap>pageheap
mytest.exe: page heap enabled with flags (full traces)

再运行一次，程序就崩溃了。

上面的例子说明了 pageheap 能够让错误尽快暴露出来。稍微修改一下代码：

```
char *p=(char*)malloc(1023);
p[1023]=1;
```

重新运行，程序会崩溃吗？

根据我的测试，分配 1023 字节的情况下，哪怕激活 pageheap，也不会崩溃。你能说明原因吗？如果看不出来，可以检查一下每次 malloc 返回的地址的数值，注意对这个数值在二进制上敏感一点，然后结合 HeapManager 和 pageheap 的原理思考一下。

对于上面两种代码，如果用 debug 模式编译，激活 pageheap，程序会崩溃吗？根据我的测试，两种情况下， debug 模式都不会崩溃的。你能想到为什么吗？

再来看下面一段代码:

```
char *p=(char*)malloc(1023);
free(p);
free(p);
```

显然有 double free 的问题。
首先取消 pageheap, 然后在 debug 和 release 模式下运行。根据我的测试，debug 模式下会崩溃，release 模式下运行正常

然后再激活 pageheap, 同样在 debug/release 模式下运行。根据我的测试，两种模式都会崩溃。如果细心观察，会发现两种模式下，崩溃后弹出的提示各自不同。你能想到为什么吗？

如果有兴趣，你还可以测试一下 heap 误用的其他几种情况，看看 pageheap 是不是都有帮助。

Heap 泄露和碎片

从上面的例子，可以很清楚地看到 pageheap 对于检查这类问题的帮助。同时也可以看到，pageheap 无法保证检查出所有潜在问题，比如分配 1023 个字节，但是写 1024 个字节这种情况。这就要求理解 pageheap 的工作原理，同时对问题作认真的思考和测试。对于为何 debug/release 模式之间还有这么多差别。

除了 heap 使用不当导致崩溃外，还有一类问题是内存泄漏。内存泄漏是指随着程序的运行，内存消耗越来越多，最后发生内存不足，或者整体性能低下。从代码上看，这类问题是由于对内存使用完毕后，没有及时释放导致的。这里的内存，可以是 VirtualAlloc 分配的，也有可能是 HeapAllocate 分配的。

举个例子，客户开发一个 cd 刻录程序。每次把碟片中所有内容写入内存，然后开始刻录。如果每次刻录完成后都忘记去释放分配的空间，那么最多能够刻三张 CD。因为三张 CD, 每一张 600MB，加在一起就是 1.8GB，濒临 2GB 的上限。检查内存泄露是一个比较大的题目，放到第四章作详细讨论。

另外还有一种跟内存泄漏相关的问题，是内存碎片(Fragmentation). 内存碎片是指内存被分割成很多的小块，以至于很难找到连续的内存来满足比较大的内存申请。导致内存碎片常见原因有两种，一种是加载了过多 DLL。还有一种是小块 Heap 的频繁使用。

DLL 分割内存空间最常见的情况是 ASP.NET 中的 batch compilation 没有打开，导致每一个 ASP.NET 页面都会被编译成一个单独的 DLL 文件。运行一段时间后，就可以看到几千个 DLL 文件加载到进程中。极端的例子是 5000 个 DLL 把 2GB 内存平均分成 5000 份，导致每一份的大小在 400K 左右(假设 DLL 本身只占用 1 个字节)，于是无法申请大于 400k 的内存，哪怕总的内存还有接近 2GB。对于这种情况的检查很简单，列一下当前进程中所有加载起来的 DLL 就可以看出问题来。

对于小块 Heap 的频繁使用导致的内存分片，可以参考下面的解释：

Heap fragmentation is often caused by one of the following two reasons

1. Small heap memory blocks that are leaked (allocated but never freed) over time
2. Mixing long lived small allocations with short lived long allocations

Both of these reasons can prevent the NT heap manager from using free memory efficiently since they are spread as small fragments that cannot be used as a single large allocation

为了更好理解上面的解释，考虑这样的情况。假设设计了一个数据结构来描述一首歌曲，分成两部分。第一部分是歌曲的名字，作者和其他相关的描述性信息，第二部分是歌曲的二进制内容。显然第一部分比第二部分小得多。假设第一部分长度 1k，第二部分 399k。 用这样的数据结构来开发一个播放器。如果每次读入 10 首歌后进行处理，对于每首歌需要调用两次内存分配函数，分别分配数据结构第一部分和第二部分需要的空间。

如果处理完成后，只释放了第二部分，忘记释放第一部分，这样每处理一次，就会留下 10 个 1k 的数据块没有释放。长时间运行后，留下的 1k 数据块就会很多，虽然 HeapManager 的薄记信息中可能记录了有很多 399k 的数据块可以分配，但是如果要申请 500k 的内存，就会因为找不到连续的内存块而失败。对于内存碎片的调试，可以参考最后的案例讨论。在 Windows 2000 上，可以用下面的方法来缓解问题：

The Windows XP Low Fragmentation Heap Algorithm Feature Is Available for Windows 2000
http://support.microsoft.com/?id=816542

关于 CLR 上内存碎片的讨论和图文详解，请参考:

.NET Memory usage - A restaurant analogy
http://blogs.msdn.com/tess/archive/2006/09/06/742568.aspx

Stack corruption

最后一种内存问题是 Stack corruption 和 Stack overrun。stack overrun 很简单，一般是由于递归函数缺少结束条件导致，使得函数调用过深把 stack 地址用光，比如下面的代码：

```
Void foo()
{
Foo();
}
```

只要在调试器里重现问题，调试器立刻就会收到 Stack overflow Exception。检查 callstack 就可以立刻看出问题所在：

```
0:001> g
(cd0.4b0): Stack overflow - code c00000fd (first chance)
```

```
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=cccccccc ebx=7ffdd000 ecx=00000000 edx=10312d18 esi=0012fe9c edi=00033130
eip=004116f9 esp=00032f9c ebp=0003305c iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010206
*** WARNING: Unable to verify checksum for c:\Documents and Settings\Li Xiong\My
Documents\My code\MyTest\debug\MyTest.exe
MyTest!foo+0x9:
004116f9 53              push    ebx
0:000> k
ChildEBP RetAddr
0003305c 00411713 MyTest!foo+0x9
00033130 00411713 MyTest!foo+0x23
00033204 00411713 MyTest!foo+0x23
000332d8 00411713 MyTest!foo+0x23
000333ac 00411713 MyTest!foo+0x23
00033480 00411713 MyTest!foo+0x23
00033554 00411713 MyTest!foo+0x23
00033628 00411713 MyTest!foo+0x23
000336fc 00411713 MyTest!foo+0x23
000337d0 00411713 MyTest!foo+0x23
000338a4 00411713 MyTest!foo+0x23
00033978 00411713 MyTest!foo+0x23
00033a4c 00411713 MyTest!foo+0x23
```

第二种情况 stack corruption 往往是臭名昭著的 stack buffer overflow 导致的。这样的 bug 不单单会造成程序崩溃，还会严重威胁到系统安全性:

http://search.msn.com/results.aspx?q=stack+buffer+overflow+attack&FORM=MSNH&srch_type=0

在当前的计算机架构上, stack 是保存运行信息的地方。当 stack 损坏后，关于当前执行情况的所有信息都丢失了。所以调试器在这种情况下没有用武之地。比如下面的代码:

```
void killstack()
{
    char c;
    char *p=&c;
    for(int i=10;i<=100;i++)
        *(p+i)=0;
}

int main(int, char*)
{
```

```
        killstack();
        return 0;
    }
```

在 VS2005 中用下面的参数，在 debug 模式下编译:

/Od /D "WIN32" /D "_DEBUG" /D "_CONSOLE" /D "_UNICODE" /D "UNICODE" /Gm /EHsc /RTC1 /MDd /Gy /Fo"Debug\\" /Fd"Debug\vc80.pdb" /W3 /nologo /c /Wp64 /Zi /TP /errorReport:prompt

在调试器中运行，看到的结果是:

```
0:000> g
ModLoad: 76290000 762ad000   C:\WINDOWS\system32\IMM32.DLL
ModLoad: 62d80000 62d89000   C:\WINDOWS\system32\LPK.DLL
ModLoad: 75490000 754f1000   C:\WINDOWS\system32\USP10.dll
(1d0.1504): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0012ff5b ebx=7ffda000 ecx=fffffffb edx=0012ffbf esi=00000000 edi=00000000
eip=000000f8 esp=0012ff68 ebp=0012ff68 iopl=0         nv up ei pl zr na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
000000f8 ??              ???
0:000> kb
ChildEBP RetAddr  Args to Child
WARNING: Frame IP not in any known module. Following frames may be wrong.
0012ff64 00000000 00000000 00000000 00000000 0xf8
0:000> dds esp
0012ff68  00000000
0012ff6c  00000000
0012ff70  00000000
0012ff74  00000000
0012ff78  00000000
0012ff7c  00000000
0012ff80  00000000
0012ff84  00000000
0012ff88  00000000
0012ff8c  00000000
0012ff90  00000000
0012ff94  00000000
0012ff98  00000000
0012ff9c  00000000
0012ffa0  00000000
0012ffa4  00000000
```

```
0012ffa8  00000000
0012ffac  00000000
0012ffb0  00000000
0012ffb4  00000000
0012ffb8  00000000
0012ffbc  00000000
0012ffc0  0012fff0
0012ffc4  77e523cd kernel32!BaseProcessStart+0x23
```

Windbg 里面看到 EIP 非法，EBP 指向的地址全是 0，callstack 的信息已经被冲毁。找不到任何线索。

对于 stack corruption，普遍的做法是首先对问题做大致定位，然后检查相关函数，在可疑函数中添加代码写 log 文件。当问题发生后从 log 文件中分析线索。

题外话和相关讨论:

/unaligned 参数

```
char *p=(char*)malloc(1023);
p[1023]=1;
```

前面提到了分配 1023 个字节的问题。在激活 pageheap 后，同时使用/unaligned 参数，才可以检测到这类问题。详细情况请参考 KB816542 中关于/unaligned 的介绍

同理，下面这段代码默认情况下使用 pageheap 也不会崩溃:

```
char *p=new char[1023];
p[-1]='c';
```

解决方法是使用 pageheap 的/backwards 参数。

上面两个例子说明由于 4kb 的粒度限制，哪怕使用 pageheap，也需要根据 pageheap 的原理来调整参数，以便覆盖多种情况。

Heap trace

Pageheap 的另外一个作用是 trace。激活 pageheap 的 trace 功能后，Heap Manager 会在内存中开辟一块专门的空间来记录每次 heap 的操作，把操作 heap 的 callstack 记录下来。当问题

发生后，以便通过导致问题的 heap 地址找到对应的 callstack。参考下面一个例子:

```
char * getmem()
{
    return new char[100];
}

void free1(char *p)
{
    delete p;
}

void free2(char *p)
{
    delete [] p;
}

int main(int, char*)
{
    char *c=getmem();
    free1(c);
    free2(c);
    return 0;
}
```

该程序在 release 模式，不激活 pageheap 的情况下是不会崩溃的。当激活 pageheap 后，pageheap 默认设定会记录 trace。激活 pageheap 后，在 debugger 中运行会看到:

```
0:000> g

=============================================================
VERIFIER STOP 00000007: pid 0x1324: block already freed

  015B1000 : Heap handle
  003F5858 : Heap block
  00000064 : Block size
  00000000 :
=============================================================

(1324.538): Break instruction exception - code 80000003 (first chance)
eax=00000000 ebx=015b1001 ecx=7c81b863 edx=0012fa7f esi=00000064 edi=00000000
eip=7c822583 esp=0012fbe8 ebp=0012fbf4 iopl=0         nv up ei pl nz na pe nc
```

```
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000           efl=00000202
ntdll!DbgBreakPoint:
7c822583 cc              int     3
```

pageheap 激发一个 break point exception，使 debugger 停下来，同时 pageheap 会在 debugger 中打印出 block already freed 信息，表示这是一个 double free 问题

使用 kb 命令打印出 callstack:

```
0:000> kb
ChildEBP RetAddr  Args to Child
0012fbe4 7c85079b 015b1000 0012fc94 0012fc70 ntdll!DbgBreakPoint
0012fbf4 7c87204b 00000007 7c8722f8 015b1000 ntdll!RtlpPageHeapStop+0x72
0012fc70 7c873305 015b1000 00000004 003f5858
ntdll!RtlpDphReportCorruptedBlock+0x11e
0012fca0 7c8734c3 015b1000 003f0000 01001002 ntdll!RtlpDphNormalHeapFree+0x32
0012fcf8 7c8766b9 015b0000 01001002 003f5858
ntdll!RtlpDebugPageHeapFree+0x146
0012fd60 7c860386 015b0000 01001002 003f5858 ntdll!RtlDebugFreeHeap+0x1ed
0012fe38 7c81d77d 015b0000 01001002 003f5858 ntdll!RtlFreeHeapSlowly+0x37
0012ff1c 78134c3b 015b0000 01001002 003f5858 ntdll!RtlFreeHeap+0x11a
0012ff68 00401016 003f5858 003f5858 00000064 MSVCR80!free+0xcd
0012ff7c 00401198 00000001 003f57e8 003f3628 win32!main+0x16
[d:\xiongli\today\win32\win32\win32.cpp @ 77]
0012ffc0 77e523cd 00000000 00000000 7ffde000 win32!__tmainCRTStartup+0x10f
0012fff0 00000000 004012e1 00000000 78746341 kernel32!BaseProcessStart+0x23
```

崩溃发生在 free 函数。Free 函数的返回地址是 00401016，所以 Free 是在 00401016 的前一行被调用的。发生问题的 heap 地址是 0x3f5858，使用!heap 命令加上-p –a 参数打印出保存下来的 callstack:

```
0:000> !heap -p -a 0x3f5858
    address 003f5858 found in
    _HEAP @ 3f0000
  in HEAP_ENTRY: Size : Prev Flags - UserPtr UserSize - state
      3f5830: 0014 : N/A  [N/A] - 3f5858 (70) - (free DelayedFree)
      Trace: 004f
      7c860386 ntdll!RtlFreeHeapSlowly+0x00000037
      7c81d77d ntdll!RtlFreeHeap+0x0000011a
      78134c3b MSVCR80!free+0x000000cd
      401010 win32!main+0x00000010
      77e523cd kernel32!BaseProcessStart+0x00000023
```

从保存的 callstack 看到，在 0x401010 前一行上，已经调用过一次 free 了，00401016，00401010

距离很近，看看分别是什么:

```
0:000> uf 00401010
win32!main [d:\xiongli\today\win32\win32\win32.cpp @ 74]:
   74 00401000 56              push    esi
   75 00401001 6a64            push    0x64
   75 00401003 e824000000      call    win32!operator new[] (0040102c)
   75 00401008 8bf0            mov     esi,eax
   76 0040100a 56              push    esi
   76 0040100b e828000000      call    win32!operator delete (00401038)
   77 00401010 56              push    esi
   77 00401011 e81c000000      call    win32!operator delete[] (00401032)
   77 00401016 83c40c          add     esp,0xc
   78 00401019 33c0            xor     eax,eax
   78 0040101b 5e              pop     esi
   79 0040101c c3              ret
```

这里可以看到，对应的 double free 是前后调用 delete 和 delete []导致的。对应的源代码地址大约在 win32.cpp 的 74 行。(源代码中，delete 和 delete[]是在两个自定义函数中被调用的。这里看不到 free1 和 free2 两个函数的原因在于 release 模式下编译器做了 inline 优化。)

如果有兴趣，可以检查一下 heap 指针 0x3f5858 前后的内容:

```
0:000> dd 0x3f5848
003f5848  7c88c580 0025a5f0 00412920 dcbaaaa9
003f5858  f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
003f5868  f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
003f5878  f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
003f5888  f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
003f5898  f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
003f58a8  f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
003f58b8  f0f0f0f0 a0a0a0a0 a0a0a0a0 00000000
```

这里的红色 dcba 其实是一个标志位，标致位前面的地址上保存的就是对应的 callstack:

```
0:000> dds 00412920
00412920  00000000
00412924  00000001
00412928  0005004f
0041292c  7c860386 ntdll!RtlFreeHeapSlowly+0x37
00412930  7c81d77d ntdll!RtlFreeHeap+0x11a
00412934  78134c3b MSVCR80!free+0xcd
00412938  00401010 win32!main+0x10
0041293c  77e523cd kernel32!BaseProcessStart+0x23
```

了解这个标致位的好处是，可以利用这个特点来解决 memory leak 和 fragmentation。发生泄漏的内存往往是相同 callstack 分配的。当泄露到一个比较严重的程度的时候，程序中残留的大多数的 heap 指针都是泄露的内存地址。由于每一个 heap 指针都带有标志位，通过在程序中搜索这个标志位，就可以找到分别的 callstack。如果某些 callstack 出现得非常频繁，往往这些 callstack 就跟 memory leak 相关。下面就是一个使用这个方法解决 memory leak 的案例。

[案例研究, 内存碎片]

客户程序在内存占用只有 300MB 左右的时候，调用 malloc 分配大于 500K 内存会失败。通过检查问题发生时候的 dump 文件，发现问题是由于 heap fragmentation 导致的。客户的程序有大量的小块内存没有及时释放，导致分片严重。

激活 pageheap 后，再次抓取问题发生时候的 dump，然后使用下面命令在内存空间搜索 dcba 标致位:

```
0:044> s -w 0 L?60030000     0xdcba
00115e9e  dcba 0000 0000 ef98 0012 893d 0047 efc8  ..........=.G...
…
19b90fe6  dcba cfe8 02d8 afe8 2ca3 cfe8 02d8 b22a  ........,....*.
19b92fe6  dcba cfe8 1a52 8fe8 1dff cfe8 1af6 f44f  ....R.........O.
19b9cfce  dcba efd0 23d8 cfd0 1c58 8fd0 15ac c0c0  .....#..X.......
…
2b06efe6  dcba cfe8 02d8 8fe8 258b cfe8 02d8 a6d2  .........%......
2b074fce  dcba 2fd0 1c0f afd0 1c4d dfd0 0e69 c0c0  .../....M...i...
…
2e860fe6  dcba afe8 02d8 2fe8 2ef3 afe8 02d8 0a0b  ......./........
2e868fce  dcba afd0 0881 2fd0 2e92 afd0 0881 c0c0  ......./........
```

根据搜索结果，使用下面的命令来随机打印 callstack，看到:

```
0:044> dds poi(19b92fe6  -6)
005bba0c  005cbe90
005bba10  00031c49
005bba14  00122ddb
005bba18  77fa8468 ntdll!RtlpDebugPageHeapAllocate+0x2f7
005bba1c  77faa27a ntdll!RtlDebugAllocateHeap+0x2d
005bba20  77f60e22 ntdll!RtlAllocateHeapSlowly+0x41
005bba24  77f46f5c ntdll!RtlAllocateHeap+0xe3a
005bba28  0046b404 Customer_App+0x6b404
005bba2c  0046b426 Customer_App+0x6b426
005bba30  00427612 Customer_App+0x27612
```

```
0:044> dds poi(19b9cfce  -6)
005bba0c  005cbe90
005bba10  00031c49
005bba14  00122ddb
005bba18  77fa8468 ntdll!RtlpDebugPageHeapAllocate+0x2f7
005bba1c  77faa27a ntdll!RtlDebugAllocateHeap+0x2d
005bba20  77f60e22 ntdll!RtlAllocateHeapSlowly+0x41
005bba24  77f46f5c ntdll!RtlAllocateHeap+0xe3a
005b8024  0046b404 Customer_App+0x6b404
005b8028  0046b426 Customer_App+0x6b426
005b802c  00427a82 Customer_App+0x27a82

0:044> dds poi(2b06efe6  -6)
005bba0c  005cbe90
005bba10  00031c49
005bba14  00122ddb
005bba18  77fa8468 ntdll!RtlpDebugPageHeapAllocate+0x2f7
005bba1c  77faa27a ntdll!RtlDebugAllocateHeap+0x2d
005bba20  77f60e22 ntdll!RtlAllocateHeapSlowly+0x41
005bba24  77f46f5c ntdll!RtlAllocateHeap+0xe3a
005bd5d4  0046b404 Customer_App+0x6b404
005bd5d8  0046b426 Customer_App+0x6b426
005bd5dc  00427612 Customer_App+0x27612
```

正常情况下，内存指针分配的 callstack 是随机的。但是上面的情况显示大多数内存指针都固定的 callstack 分配，该 callstack 很有可能就是泄漏的根源。拿到客户的 PDB 文件后，把偏移跟源代码对应起来，很快就找到了申请这些内存的源代码。添加对应的内存释放代码后，问题解决。

2.4 区分层次

该小结从 C Runtime 的介绍出发，介绍了排错过程中找准层次关系的重要性。

CRT 和层次

考虑 C 语言中的这些函数和关键字是如何实现的：

fopen
printf
malloc
beginthread
throw/catch

这些函数的实现，最终都需要调用操作系统的 API。从下面的 callstack 中，就可以看到 malloc 函数调用了 RtlAllocateHeap 这个 Native API:

```
0:000> k
ChildEBP RetAddr
0013ff4c 78134d85 ntdll!RtlAllocateHeap
WARNING: Stack unwind information not available. Following frames may be wrong.
0013ff5c 78134d0b MSVCR80!malloc+0x7a
0013ff6c 00401016 MSVCR80!malloc
0013ff7c 00401184 MyTest!wmain+0x16 [c:\documents and settings\li xiong\my
documents\my code\mytest\mytest\mytest.cpp @ 121]
0013ffc0              7c816d4f              MyTest!__tmainCRTStartup+0x10f
[f:\rtm\vctools\crt_bld\self_x86\crt\src\crtexe.c @ 583]
0013fff0 00000000 kernel32!BaseProcessStart+0x23
```

相关的一些信息，可以参考：

What are the C and C++ libraries my program would link with?
http://support.microsoft.com/kb/154753/en-us
Frequently asked questions about the Standard C++ library
http://support.microsoft.com/kb/154419/en-us

前面讨论的知识点，比如异常和内存，都是从操作系统的角度来讨论的。但是程序往往不会直接调用操作系统的 API 来完成异常的处理和内存的分配，对于 C/C++开发的应用程序来说，开发人员使用 C 库函数，C 运行库(CRT)对操作系统进行封装，最后由 C 运行库调用 API。

这样做的好处在于 CRT 作为新增加的抽象层，提供了更多的灵活性，比如不同操作系统之间的移植和调试。

_CRTDBG_MAP_ALLOC

看下面一个例子，开发人员发现了一个几千行的 C 程序中有少量的内存泄漏，应该如何定位和解决？有什么快速的方法吗？

其实，不需要用 pageheap，也不需要用 API Hook，直接而且高效的办法是采用 CRT 的 Debug Heap。在 debug 模式下用 F5 运行下面一段程序：

```
#define _CRTDBG_MAP_ALLOC
#include <stdlib.h>
#include <crtdbg.h>
int _tmain(int argc, _TCHAR* argv[])
{
    char *p=(char*)malloc(100);
    _CrtDumpMemoryLeaks();
    return 0;
}
```

运行结束后，注意观察 VS2005 的 Output 窗口，会看到：

```
Detected memory leaks!
Dumping objects ->
c:\documents and settings\li xiong\my documents\my code\mytest\mytest\mytest.cpp(118) : {84}
normal block at 0x003A7DC0, 100 bytes long.
 Data: <                > CD CD CD CD CD CD CD CD CD CD CD CD CD CD CD CD
Object dump complete.
```

上面的 Output 窗口明确地提示了：

1. 有内存泄漏
2. 泄漏的内存是在 118 行分配的
3. 泄漏的内存的起始地址以及长度
4. 泄漏的内存中的数据是什么

上面的功能就是 debug 模式的 CRT 提供的。操作系统的 Heap API 本身没有提供检查内存泄漏的功能。但是在 CRT 的实现中，既然内存分配使用 malloc 来完成，而不是直接调用 API，那么 CRT 就可以通过对 Heap API 的包装来完成这样的功能。详细的信息，可以参考：

Memory Leak Detection Enabling
http://msdn2.microsoft.com/en-us/library/e5ewb1h3(VS.80).aspx
The CRT Debug Heap

http://msdn2.microsoft.com/en-us/library/974tc9t1(VS.80).aspx

不单单是 memory leak, 对于其他 Heap 问题，比如 double free，debug heap 都能够监测到。

BSTR Cache

另外一个 heap 的问题。客户的代码崩溃在下面的 call stack:

```
ntdll!RtlAllocateHeap+0x1dd
ole32!CRetailMalloc_Alloc+0x16
oleaut32!APP_DATA::AllocCachedMem+0x4f
oleaut32!SysAllocStringByteLen+0x2e
oleaut32!ErrStringCopyNoNull+0x16
oleaut32!VariantChangeTypeEx+0x9ca
oleaut32!CoerceArg+0x3ff
```

显然，崩溃发生在 RtlAllocateHeap 这个 Native API 里面，直接的原因就是 Heap Manager 的内部数据结构在问题发生以前，就已经被损坏了。根据前面的讨论，直接的做法就是 enable pageheap, 使得问题可以在第一时间暴露出来。在使用 pageheap /full 激活后，发现问题还是没有在第一时间发生，似乎 pageheap 没有作用，为什么呢？

如果足够敏感，下面两个函数可能会引起注意: AllocCachedMem, SysAllocStringByteLen. 第一个 API 的名字似乎从某一个 Cache 中分配内存，第二个名字其实是一个 Ole 的 API，用来分配 BSTR 字符串的。这里一个 API 的实现依赖于另外一个 API, 可以看出 Ole 的内存管理跟操作系统的 Heap 是在两个层次上的。

Pageheap 能够保证操作系统上的 Heap 问题能够及时暴露出来，但是这里操作系统的 Heap API 被 Ole 内存管理器包装了一次。如果 Ole 缓存了操作系统分配出来的 Heap, 然后进行第二次分配和管理，那么用户的每次内存操作就不是跟操作系统的 Heap API 直接对应了。当问题第一现场在 Ole 内存管理器，然后才导致 Heap Corrutpion，就无法保证 Pageheap 能够在第一时间把错误暴露出来:

FIX: OLE Automation BSTR caching will cause memory leak sources in Windows 2000
http://support.microsoft.com/?id=139071

在应用了 OANOCACHE=1 的设定后，Ole BSTR Cache 被禁止了，于是每次 BSTR 的分配释放都跟操作系统的 Heap 分配直接对应起来。pageheap 很快就找到了问题的根源

几乎没有程序是运行在一个单一层次上的。当发生问题的时候，弄清楚各层次的关系，然后才能选择正确的入口和方法来解决。

题外话和相关讨论

CRT Debug Heap 一定对 Debug 有帮助吗？

记得前面一节提到两个现象:

1. Debug 模式下，无论是否激活 pageheap，访问越界 1 字节的时候都不会崩溃。
2. Double free 的时候，debug 模式下弹出的信息和 release 模式下使用 pageheap 弹出的信息不一样。

这两个现象都跟 CRT 的包装相关。由于 CRT 在 malloc/new 的实现中对 Windows Heap Manager API 作了再次包装，每次通过 CRT Debug Heap 分配内存的时候，CRT 会申请额外多一点的空间来记录 CRT 自己的薄记信息。由于申请了额外的内存，所以越界的时候可能写到 CRT 的薄记空间中，崩溃就不会发生，pageheap 失效。

对于 double free，由于 free 的时候 CRT 函数会通过检查自己的薄记信息来判断 double free 是否发生，所以 CRT 会通过 CRT 自身的 Assert 来弹出对话框报告问题。

2.5 调试器和 Windbg

这一节介绍调试器，windbg 的相关知识。首先对调试器和符号文件作大致的介绍，然后针对常用的调试命令作演示。接下来介绍强大而灵活的条件断点，最后介绍调试器目录下的相关工具。

关于调试器和 windbg 特别好的英文教程在:

DebugInfo:
http://www.debuginfo.com/

Windows Debuggers: Part 1: A WinDbg Tutorial
http://www.codeproject.com/debug/windbg_part1.asp

建议首先通过这些文章全方面熟悉 Windbg 的使用和功能,然后再结合下面的例子进行实践.

调试器

调试器是用来观察和控制目标进程的工具。对于用户态的进程，调试器可以查看用户态内存空间和寄存器上的数据。对于不同类型的数据和代码，调试器提供了各种命令，方便把这些信息用特定的格式区分和显示出来。调试器还可以把一个目标进程某一时刻的所有信息写入一个文件 (dump)，直接打开这个文件分析。调试器还可以通过设置断点的机制来控制目标程序什么时候挂起，什么时候运行。

关于调试器的工作原理，请参考 Debugging Applications for Windows 这本书。

Windows 上专用调试器 windbg 及其相关工具的下载地址:
http://www.microsoft.com/whdc/devtools/debugging/installx86.mspx

在安装好 windbg 后，可以启动 windbg.exe, 然后按 F1 弹出帮助。这是了解和使用 windbg 的最好文档。每个命令的详细说明，都可以在里面找到。

符号文件 (Symbol file)

当用 VC/VB 编译生成 EXE/DLL 后，往往还会生成 PDB 文件。里面包含的是 EXE/DLL 的符号信息。比如 EXE 中某一个汇编代码对应到源代码中的函数名是什么，这个函数在源代码中位于什么文件的多少行。有了符号文件，当在调试器中试图读取某一个内存地址的时候，调试器会尝试在对应的 PDB 文件中配对，看这个内存地址是否有符号对应。如果能够找到，调试器就可以把对应的符号显示出来。这样极大程度上方便了开发人员的观察。 对于操作系统 EXE/DLL，微软也提供了对应的符号文件下载地址。

(默认情况下，符号文件中包含了所有的结构，函数，以及对应的源代码信息。微软提供的 Windows 符号文件去掉了源代码信息，函数参数定义，和一些内部数据结构的定义。)

调试器能看到啥

调试器一般可以直观地看到下面一些信息:

- 进程运行的状态和系统状态。比如进程运行了多少时间，环境变量是什么
- 当前进程加载的所有 EXE/DLL 的详细信息
- 某一个地址上的汇编指令
- 察看内存地址的内容和属性,比如是否可写
- 每个的 call stack (需要 symbol)
- Call stack 上每个函数的局部变量
- 格式化地显示程序中的数据结构(需要 symbol)
- 查看和修改内存地址上的数据或者寄存器上的数据
- 部分操作系统管理的数据结构,比如 Heap, Handle, CriticalSection 等等

调试器的另外一个作用是设定条件断点。可以设定在某一个指令地址上停下来，也可以设定当某一个内存地址等于多少的时候停下来，或者当某一个 exception/notification 发生的时候停下来 。进入一个函数调用的时候停下来，跳出当前函数调用的时候停下来。停下来后可以让调试器自动运行某些命令，记录某些信息，然后让调试器自动判断某些条件来决定是否要继续运行。通过简单的条件断点功能，可以很方便地实现下面一些功能:

- 当某一个函数被调用的时候，在调试器输出窗口中打印出函数参数
- 计算某一个变量被修改了多少次
- 监视一个函数调用了那些子函数，分别被调用了多少次
- 每次抛 C++异常的时候自动产生 dump 文件

用 IE 来操练

用 windbg 来调试目标进程，有两种方法,分别是通过调试器启动，和用调试器直接监视(attach)正在运行的进程。

通过 File -> Open Executable 菜单，可以选择对应的 EXE 在调试器中启动。通过 File -> Attach to a process 可以选择一个正在运行的进程进行调试。

下面用 IE 作为演示的例子来介绍常用的命令。

打开 IE,访问 www.msdn.com, 然后启动 windbg, 按 F6，选择刚刚启动的(最下面)iexplorer.exe 进程。

加载调试器后会看到 IE 停止运行了，在 windbg 的主窗口里面可以看到一大堆关于这个进程

的信息。用 g 命令可以让目标程序继续执行，用 ctrl+break 可以挂起正在运行目标程序回到调试模式.

目标程序挂起后,可以用 ctrl+S 打开 symbol 设定窗口，然后可以根据下面的文章来设定符号文件路径.
http://www.microsoft.com/whdc/devtools/debugging/debugstart.mspx

在我本地,我的 symbol 路径设定如下:
SRV*D:\websymbols*http://msdl.microsoft.com/download/symbols;D:\MyAppSymbol

这里的 D:\websymbols 目录是用来保存从 msdl.microsoft.com 上自动下载的操作系统符号文件. 而我自己编译生成的符号文件,我都手动拷贝到 D:\MyAppSymbol 路径下.

接下来,在 windbg 的命令窗口中(如果看不到可以用 alt+1 打开),运行下面命令:

vertarget

vertarget 命令显示当前进程的大致信息:

```
0:026> vertarget
Windows Server 2003 Version 3790 (Service Pack 1) MP (2 procs) Free x86 compatible
Product: Server, suite: Enterprise TerminalServer SingleUserTS
kernel32.dll version: 5.2.3790.1830 (srv03_sp1_rtm.050324-1447)
Debug session time: Thu Apr 27 13:53:50.414 2006 (GMT+8)
System Uptime: 15 days 1:59:13.255
Process Uptime: 0 days 0:07:34.508
  Kernel time: 0 days 0:00:01.109
  User time: 0 days 0:00:00.609
```

上面的 0:026> 是命令提示符,026 表示当前的线程 ID.后面会介绍切换线程的命令,到时候就可以看到提示符的变化.

跟大多数的命令输出一样, Vertarget 的输出非常明白直观,显示当前系统的版本和运行时间.

!peb

接着可以用!peb 命令来显示更详细的一些东西. 由于输出太长，这里就省略了。

lmvm

用 lmvm 命令可以看任意一个 DLL/EXE 的详细信息，以及 symbol 的情况:

```
0:026> lmvm msvcrt
start    end        module name
```

```
77ba0000 77bfa000   msvcrt     (deferred)
    Image path: C:\WINDOWS\system32\msvcrt.dll
    Image name: msvcrt.dll
    Timestamp:        Fri Mar 25 10:33:02 2005 (4243785E)
    CheckSum:         0006288A
    ImageSize:        0005A000
    File version:     7.0.3790.1830
    Product version:  6.1.8638.1830
    File flags:       0 (Mask 3F)
    File OS:          40004 NT Win32
    File type:        1.0 App
    File date:        00000000.00000000
    Translations:     0409.04b0
    CompanyName:      Microsoft Corporation
    ProductName:      Microsoft® Windows® Operating System
    InternalName:     msvcrt.dll
    OriginalFilename: msvcrt.dll
    ProductVersion:   7.0.3790.1830
    FileVersion:      7.0.3790.1830 (srv03_sp1_rtm.050324-1447)
    FileDescription:  Windows NT CRT DLL
    LegalCopyright:   © Microsoft Corporation. All rights reserved.
```

命令的第二行显示 deferred，表示目前并没有加载 msvcrt 的 symbol，可以用.reload 命令来加载。在加载前，可以用!sym 命令来打开 symbol 加载过程的详细输出:

.reload / !sym

```
0:026> !sym noisy
noisy mode - symbol prompts on
0:026> .reload /f msvcrt.dll
SYMSRV:  msvcrt.pd_ from http://msdl.microsoft.com/download/symbols: 80847
bytes copied
DBGHELP: msvcrt - public symbols

c:\websymbols\msvcrt.pdb\62B8BDC3CC194D2992DCFAED78B621FC1\msvcrt.pdb
0:026> lmvm msvcrt
start    end       module name
77ba0000 77bfa000   msvcrt     (pdb symbols)
c:\websymbols\msvcrt.pdb\62B8BDC3CC194D2992DCFAED78B621FC1\msvcrt.pdb
    Loaded symbol image file: C:\WINDOWS\system32\msvcrt.dll
    Image path: C:\WINDOWS\system32\msvcrt.dll
    Image name: msvcrt.dll
    Timestamp:        Fri Mar 25 10:33:02 2005 (4243785E)
    CheckSum:         0006288A
    ImageSize:        0005A000
```

```
File version:     7.0.3790.1830
Product version:  6.1.8638.1830
File flags:       0 (Mask 3F)
File OS:          40004 NT Win32
File type:        1.0 App
File date:        00000000.00000000
Translations:     0409.04b0
CompanyName:      Microsoft Corporation
ProductName:      Microsoft® Windows® Operating System
InternalName:     msvcrt.dll
OriginalFilename: msvcrt.dll
ProductVersion:   7.0.3790.1830
FileVersion:      7.0.3790.1830 (srv03_sp1_rtm.050324-1447)
FileDescription:  Windows NT CRT DLL
LegalCopyright:   © Microsoft Corporation. All rights reserved.
```

可以看到,symbol 从 msdl.microsoft.com 自动下载后加载。

lmf

直接输入 lmf 命令可以列出当前进程中加载的所有 DLL 文件和对应的路径:

```
0:018> lmf
start    end       module name
00d40000 00dda000   iexplore C:\Program Files\Internet Explorer\iexplore.exe
04320000 043c9000   atiumdva C:\Windows\system32\atiumdva.dll
10000000 1033d000   googletoolbar2 c:\program files\google\googletoolbar2.dll
37f00000 37f0f000   Cjktl32  E:\Program Files\Powerword 2003\Cjktl32.dll
```

r, d, e

接下来，可以用 r 命令来显示寄存器的信息，用 d 命令来显示内存地址上的值，e 命令修改内存地址上的值。

显示寄存器:

```
0:018> r
eax=7ffdc000 ebx=00000000 ecx=00000000 edx=7707f06d esi=00000000 edi=00000000
eip=77032ea8 esp=054efc14 ebp=054efc40 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
ntdll!DbgBreakPoint:
77032ea8 cc              int     3
```

用d命令显示esp 寄存器指向的内存，默认为byte格式

```
0:018> d esp
```

```
054efc14  a9 f0 07 77 e9 ef 4e 05-00 00 00 00 00 00 00 00  ...w..N.........
054efc24  00 00 00 00 18 fc 4e 05-00 00 00 00 7c fc 4e 05  ......N.....|.N.
054efc34  f2 8b ff 76 a1 f5 03 77-00 00 00 00 4c fc 4e 05  ...v...w....L.N.
054efc44  33 38 b4 75 00 00 00 00-8c fc 4e 05 bd a9 02 77  38.u......N....w
054efc54  00 00 00 00 25 ef 4e 05-00 00 00 00 00 00 00 00  ....%.N.........
054efc64  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
054efc74  58 fc 4e 05 00 00 00 00-ff ff ff ff f2 8b ff 76  X.N............v
054efc84  a1 e2 03 77 00 00 00 00-00 00 00 00 00 00 00 00  ...w............
```

用 dd 命令直接指定 054efc14 地址 ， 第二个 d 表示用 DWORD 格式

```
0:018> dd 054efc14
054efc14  7707f0a9 054eefe9 00000000 00000000
054efc24  00000000 054efc18 00000000 054efc7c
054efc34  76ff8bf2 7703f5a1 00000000 054efc4c
054efc44  75b43833 00000000 054efc8c 7702a9bd
054efc54  00000000 054eef25 00000000 00000000
054efc64  00000000 00000000 00000000 00000000
054efc74  054efc58 00000000 ffffffff 76ff8bf2
054efc84  7703e2a1 00000000 00000000 00000000
```

用 ed 命令把 054efc14 地址 上的值修改为 11112222

```
0:018> ed 054efc14  11112222
```

再次用 dd 命令显示 054efc14 地址上的值，后面的 L4 参数表示长度为 4 个 DWORD

```
0:018> dd 054efc14  L4
054efc14  11112222 40a15c00 00000000 40a15c00
```

有了上面几个命令，就可以访问和修改当前进程中的所有内存。这些命令的参数和格式非常灵活，详细内容参考帮助文档。

用!address 命令可以显示某一个地址上的页信息:

```
0:001> !address 7ffde000
    7ffde000 : 7ffde000 - 00001000
                Type     00020000 MEM_PRIVATE
                Protect  00000004 PAGE_READWRITE
                State    00001000 MEM_COMMIT
                Usage    RegionUsagePeb
```

如果不带参数，可以显示更详细的统计信息。

通过上面这些命令，能够知道一个进程的基本内容，比如有没有加载一些第三方的 DLL，详细的版本信息等等。

S

S 命令可以搜索内存。一般来说，用在下面的地方:

1. 寻找内存泄露的线索。比如知道当前内存泄漏的内容是一些固定的字符串，就可以在 DLL 区域搜索这些字符串出现的地址，然后再搜索这些地址用到什么代码中，找出这些内存是在什么地方开始分配的。
2. 寻找错误代码的根源。比如知道当前程序返回了 0x80074015 这样的一个代码，但是不知道这个代码是由哪一个内层函数返回的。就可以在代码区搜索 0x80074015，找到可能返回这个代码的函数。

下面就是访问sina.com的时候，用windbg搜索ie里面 www.sina.com.cn的结果

```
0:022> s -u 0012ff40 L?80000000 "www.sina.com.cn"
001342a0  0077 0077 0077 002e 0073 0069 006e 0061  w.w.w...s.i.n.a.
00134b82  0077 0077 0077 002e 0073 0069 006e 0061  w.w.w...s.i.n.a.
00134f2e  0077 0077 0077 002e 0073 0069 006e 0061  w.w.w...s.i.n.a.
0013570c  0077 0077 0077 002e 0073 0069 006e 0061  w.w.w...s.i.n.a.
```

结合 S 命令和前面介绍的修改内存命令，根本不需要用什么金山游侠就可以查找/修改游戏中主角的生命了 :-)

接下来，看看跟线程相关的命令:

!runaway

!runaway 可以显示每一个线程所耗费 usermode CPU 时间的统计信息:

```
0:001> !runaway
 User Mode Time
  Thread       Time
   0:83c      0 days 0:00:00.406
  13:bd4      0 days 0:00:00.046
  10:ac8      0 days 0:00:00.046
  24:4f4      0 days 0:00:00.031
  11:d8c      0 days 0:00:00.015
  26:109c     0 days 0:00:00.000
  25:1284     0 days 0:00:00.000
  23:12cc     0 days 0:00:00.000
  22:16c0     0 days 0:00:00.000
  21:57c      0 days 0:00:00.000
  20:c00      0 days 0:00:00.000
  19:14e8     0 days 0:00:00.000
  18:1520     0 days 0:00:00.000
  16:9dc      0 days 0:00:00.000
  15:1654     0 days 0:00:00.000
```

```
  14:13f4     0 days 0:00:00.000
   9:104c     0 days 0:00:00.000
   8:1760     0 days 0:00:00.000
   7:cc8      0 days 0:00:00.000
   6:530      0 days 0:00:00.000
   5:324      0 days 0:00:00.000
   4:178c     0 days 0:00:00.000
   3:1428     0 days 0:00:00.000
   2:1530     0 days 0:00:00.000
   1:448      0 days 0:00:00.000
```

~

用~命令，可以显示线程信息和在不同线程之间切换

```
0:001> ~
   0  Id: c0.83c Suspend: 1 Teb: 7ffdd000 Unfrozen
.  1  Id: c0.448 Suspend: 1 Teb: 7ffdb000 Unfrozen
   2  Id: c0.1530 Suspend: 1 Teb: 7ffda000 Unfrozen
   3  Id: c0.1428 Suspend: 1 Teb: 7ffd9000 Unfrozen
   4  Id: c0.178c Suspend: 1 Teb: 7ffd8000 Unfrozen
   5  Id: c0.324 Suspend: 1 Teb: 7ffdc000 Unfrozen
   6  Id: c0.530 Suspend: 1 Teb: 7ffd7000 Unfrozen
   7  Id: c0.cc8 Suspend: 1 Teb: 7ffd6000 Unfrozen
   8  Id: c0.1760 Suspend: 1 Teb: 7ffd5000 Unfrozen
   9  Id: c0.104c Suspend: 1 Teb: 7ffd4000 Unfrozen
  10  Id: c0.ac8 Suspend: 1 Teb: 7ffd3000 Unfrozen
  11  Id: c0.d8c Suspend: 1 Teb: 7ff9f000 Unfrozen
  13  Id: c0.bd4 Suspend: 1 Teb: 7ff9d000 Unfrozen
  14  Id: c0.13f4 Suspend: 1 Teb: 7ff9c000 Unfrozen
  15  Id: c0.1654 Suspend: 1 Teb: 7ff9b000 Unfrozen
  16  Id: c0.9dc Suspend: 1 Teb: 7ff9a000 Unfrozen
  18  Id: c0.1520 Suspend: 1 Teb: 7ff96000 Unfrozen
  19  Id: c0.14e8 Suspend: 1 Teb: 7ff99000 Unfrozen
  20  Id: c0.c00 Suspend: 1 Teb: 7ff97000 Unfrozen
  21  Id: c0.57c Suspend: 1 Teb: 7ff95000 Unfrozen
  22  Id: c0.16c0 Suspend: 1 Teb: 7ff94000 Unfrozen
  23  Id: c0.12cc Suspend: 1 Teb: 7ff93000 Unfrozen
  24  Id: c0.4f4 Suspend: 1 Teb: 7ff92000 Unfrozen
  25  Id: c0.1284 Suspend: 1 Teb: 7ff91000 Unfrozen
  26  Id: c0.109c Suspend: 1 Teb: 7ff90000 Unfrozen
0:001> ~0s
eax=0013e7c4 ebx=00000000 ecx=0013e7c4 edx=0000000b esi=001642e8 edi=00000000
eip=7c82ed54 esp=0013eb3c ebp=0013ed98 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!KiFastSystemCallRet:
```

```
7c82ed54 c3              ret
```

上面的~0s 命令，切换到了线程 0.

k, kb, kp, kv, kn

用 k 命令，可以显示当前线程的 call stack. 当然还有 kb,kp,kn,kv 等等相关命令:

```
0:000> k
ChildEBP RetAddr
0013eb38 7739d02f ntdll!KiFastSystemCallRet
0013ed98 75ecb30f USER32!NtUserWaitMessage+0xc
0013ee24 75ed7ce5 BROWSEUI!BrowserProtectedThreadProc+0x44
0013fea8 779ac61e BROWSEUI!SHOpenFolderWindow+0x22c
0013fec8 0040243d SHDOCVW!IEWinMain+0x129
0013ff1c 00402748 iexplore!WinMain+0x316
0013ffc0 77e523cd iexplore!WinMainCRTStartup+0x186
0013fff0 00000000 kernel32!BaseProcessStart+0x23
```

可以结合~和 k 命令，来显示所有线程的 call stack. 输入~*k 试一下。

u

如果要反汇编某一个地址，直接用 u 命令加地址:

```
0:000> u 7739d023
USER32!NtUserWaitMessage:
7739d023 b84a120000      mov     eax,0x124a
7739d028 ba0003fe7f      mov     edx,0x7ffe0300
7739d02d ff12           call    dword ptr [edx]
7739d02f c3             ret
```

如果符号文件加载正确，可以用 uf 命令反汇编整个函数，比如:

```
0:000> uf USER32!NtUserWaitMessage
```

x

如果要找某一个符号对应的二进制地址，可以用 x 命令:

```
0:000> x msvcrt!printf
77bd27c2 msvcrt!printf = <no type information>
0:000> u 77bd27c2
msvcrt!printf:
77bd27c2 6a10           push    0x10
77bd27c4 687047ba77      push    0x77ba4770
77bd27c9 e8f65cffff      call    msvcrt!_SEH_prolog (77bc84c4)
77bd27ce bec81cbf77      mov     esi,0x77bf1cc8
77bd27d3 56             push    esi
```

```
77bd27d4 6a01           push    0x1
77bd27d6 e86ea2ffff      call    msvcrt!_lock_file2 (77bcca49)
77bd27db 59             pop     ecx
```

x 命令还支持通配符。用 x ntdll!*试试

dds

dds 命令跟 x 命令相反，dds 把某一个地址对应到符号。比如要看看当前 stack 中保存了哪些函数地址，就可以检查 ebp 检查 ebp 指向的内存:

```
0:000> dds ebp
0013ed98  0013ee24
0013ed9c  75ecb30f BROWSEUI!BrowserProtectedThreadProc+0x44
0013eda0  00163820
0013eda4  0013ee50
0013eda8  00163820
0013edac  00000000
0013edb0  0013ee10
0013edb4  75ece83a BROWSEUI!__delayLoadHelper2+0x23a
0013edb8  00000005
0013edbc  0013edcc
0013edc0  0013ee50
0013edc4  00163820
0013edc8  00000000
0013edcc  00000024
0013edd0  75f36d2c BROWSEUI!_DELAY_IMPORT_DESCRIPTOR_SHELL32
0013edd4  75f3a184 BROWSEUI!_imp__SHGetInstanceExplorer
0013edd8  75f36e80 BROWSEUI!_sz_SHELL32
0013eddc  00000001
0013ede0  75f3726a BROWSEUI! urlmon_NULL_THUNK_DATA_DLN+0x116
0013ede4  7c8d0000 SHELL32!_imp__RegCloseKey <PERF> (SHELL32+0x0)
0013ede8  7c925b34 SHELL32!SHGetInstanceExplorer
```

这里 DDS 命令自动把 `75ecb30f` 地址映射到了符号文件中对应的符号信息。

由于 COM Interface 和 C++ Vtable 里面的成员函数都是顺序排列的，所以这个命令可以方便地找到虚函数表中具体的函数地址。比如用下面的命令可以找到 OpaqueDataInfo 类型中虚函数对应的实际函数地址:

首先用 x 命令找到 OpaqueDataInfo 虚函数表地址:

```
0:000> x ole32!OpaqueDataInfo::`vftable'
7768265c ole32!OpaqueDataInfo::`vftable' = <no type information>
77682680 ole32!OpaqueDataInfo::`vftable' = <no type information>
```

然后用 dds 命令检查虚函数表中的函数名字:

```
0:000> dds 7768265c
7768265c  77778245 ole32!ServerLocationInfo::QueryInterface
77682660  77778254 ole32!ScmRequestInfo::AddRef
77682664  77778263 ole32!ScmRequestInfo::Release
77682668  77779d26 ole32!OpaqueDataInfo::Serialize
7768266c  77779d3d ole32!OpaqueDataInfo::UnSerialize
77682670  77779d7a ole32!OpaqueDataInfo::GetSize
77682674  77779dcb ole32!OpaqueDataInfo::GetCLSID
77682678  77779deb ole32!OpaqueDataInfo::SetParent
7768267c  77779e18 ole32!OpaqueDataInfo::SerializableQueryInterface
77682680  777799b5 ole32!InstantiationInfo::QueryInterface
77682684  77689529 ole32!ServerLocationInfo::AddRef
77682688  776899cc ole32!ScmReplyInfo::Release
7768268c  77779bcd ole32!OpaqueDataInfo::AddOpaqueData
77682690  77779c43 ole32!OpaqueDataInfo::GetOpaqueData
77682694  77779c99 ole32!OpaqueDataInfo::DeleteOpaqueData
77682698  776a8cf6 ole32!ServerLocationInfo::GetRemoteServerName
7768269c  776aad96 ole32!OpaqueDataInfo::GetAllOpaqueData
776826a0  77777a3b ole32!CDdeObject::COleObjectImpl::GetClipboardData
776826a4  00000021
776826a8  77703159 ole32!CClassMoniker::QueryInterface
776826ac  77709b01 ole32!CErrorObject::AddRef
776826b0  776edaff ole32!CClassMoniker::Release
776826b4  776ec529 ole32!CClassMoniker::GetUnmarshalClass
776826b8  776ec546 ole32!CClassMoniker::GetMarshalSizeMax
776826bc  776ec589 ole32!CClassMoniker::MarshalInterface
776826c0  77702ca9 ole32!CClassMoniker::UnmarshalInterface
776826c4  776edbe1 ole32!CClassMoniker::ReleaseMarshalData
776826c8  776e5690 ole32!CDdeObject::COleItemContainerImpl::LockContainer
776826cc  7770313b ole32!CClassMoniker::QueryInterface
776826d0  7770314a ole32!CClassMoniker::AddRef
776826d4  776ec5a8 ole32!CClassMoniker::Release
776826d8  776ec4c6 ole32!CClassMoniker::GetComparisonData
```

小例子

下面再用一个小例子来演示如何具体地观察程序中的数据结构:

首先在 debug 模式下编译并且 ctrl+f5 运行下面的代码:

```
struct innner
{
    char arr[10];
};
class MyCls
```

```
{
private:
     char* str;
     innner inobj;
public:
     void set(char* input)
     {
          str=input;
          strcpy(inobj.arr,str);
     }
     int output()
     {
          printf(str);
          return 1;
     }
     void hold()
     {
          getchar();
     }
};

void foo1()
{
     MyCls *pcls=new MyCls();
     void *rawptr=pcls;
     pcls->set("abcd");
     pcls->output();
     pcls->hold();
};
void foo2()
{
     printf("in foo2\n");
     foo1();
};
void foo3()
{
     printf("in foo3\n");
     foo2();
};

int _tmain(int argc, _TCHAR* argv[])
{
     foo3();
     return 0;
```

```
}
```

当 console 等待输入的时候，启动 windbg，然后用 F6 加载目标进程。
用~0s 命令切换到主线程，察看 callstack:

```
0:000> kn
 # ChildEBP RetAddr
00 0012f7a0 7c821c94 ntdll!KiFastSystemCallRet
01 0012f7a4 7c836066 ntdll!NtRequestWaitReplyPort+0xc
02 0012f7c4 77eaaba3 ntdll!CsrClientCallServer+0x8c
03 0012f8bc 77eaacb8 kernel32!ReadConsoleInternal+0x1b8
04 0012f944 77e41990 kernel32!ReadConsoleA+0x3b
05 0012f99c 10271754 kernel32!ReadFile+0x64
06 0012fa28 10271158 MSVCR80D!_read_nolock+0x584
[f:\rtm\vctools\crt_bld\self_x86\crt\src\read.c @ 247]
07 0012fa74 10297791 MSVCR80D!_read+0x1a8
[f:\rtm\vctools\crt_bld\self_x86\crt\src\read.c @ 109]
08 0012fa9c 102a029b MSVCR80D!_filbuf+0x111
[f:\rtm\vctools\crt_bld\self_x86\crt\src\_filbuf.c @ 136]
09 0012faf0 102971ce MSVCR80D!getc+0x24b
[f:\rtm\vctools\crt_bld\self_x86\crt\src\fgetc.c @ 76]
0a 0012fafc 102971e8 MSVCR80D!_fgetchar+0xe
[f:\rtm\vctools\crt_bld\self_x86\crt\src\fgetchar.c @ 37]
0b 0012fb04 0041163b MSVCR80D!getchar+0x8
[f:\rtm\vctools\crt_bld\self_x86\crt\src\fgetchar.c @ 47]
0c 0012fbe4 00413f82 exceptioninject!MyCls::hold+0x2b
[d:\xiongli\today\exceptioninject\exceptioninject\exceptioninject.cpp @ 61]
0d 0012fcec 0041169a exceptioninject!foo1+0xa2
[d:\xiongli\today\exceptioninject\exceptioninject\exceptioninject.cpp @ 72]
0e 0012fdc0 004114fa exceptioninject!foo2+0x3a
[d:\xiongli\today\exceptioninject\exceptioninject\exceptioninject.cpp @ 77]
0f 0012fe94 004116d3 exceptioninject!foo3+0x3a
[d:\xiongli\today\exceptioninject\exceptioninject\exceptioninject.cpp @ 82]
10 0012ff68 00412016 exceptioninject!wmain+0x23
[d:\xiongli\today\exceptioninject\exceptioninject\exceptioninject.cpp @ 87]
11 0012ffb8 00411e5d exceptioninject!__tmainCRTStartup+0x1a6
[f:\rtm\vctools\crt_bld\self_x86\crt\src\crtexe.c @ 583]
12 0012ffc0 77e523cd exceptioninject!wmainCRTStartup+0xd
[f:\rtm\vctools\crt_bld\self_x86\crt\src\crtexe.c @ 403]
13 0012fff0 00000000 kernel32!BaseProcessStart+0x23
```

是不是跟前面 IE 例子的输出不太一样？这里好多地方有源代码的信息。这是因为 msvcr80D 的 symbol 是包含了源代码信息的。可以运行 kp 命令看看更多的差别。

上面 callstack 中每一行前面的序号叫做 frame number. 通过.frame 命令,可以切换到对应的函数中检查局部变量.比如 `exceptioninject!foo1` 这个函数前面的 frame number 是 d,于是:

.frame

```
0:000> .frame d
0d 0012fcec 0041169a exceptioninject!foo1+0xa2
[d:\xiongli\today\exceptioninject\exceptioninject\exceptioninject.cpp @ 72]
```

用 x 命令显示当前这个函数里面的局部变量

```
0:000> x
0012fce4 pcls = 0x0039ba80
0012fcd8 rawptr = 0x0039ba80
```

dt

在符号文件加载的情况下，dt 命令格式化显示 pcls 成员信息

```
0:000> dt pcls
Local var @ 0x12fce4 Type MyCls*
0x0039ba80
   +0x000 str            : 0x00416648  "abcd"
   +0x004 inobj          : inner
```

用-b -r 参数可以显示 inner class 和数组的信息

```
0:000> dt pcls -b -r
Local var @ 0x12fce4 Type MyCls*
0x0039ba80
   +0x000 str            : 0x00416648  "abcd"
   +0x004 inobj          : innner
      +0x000 arr            :  "abcd"
       [00] 97 'a'
       [01] 98 'b'
       [02] 99 'c'
       [03] 100 'd'
       [04] 0 ''
       [05] 0 ''
       [06] 0 ''
       [07] 0 ''
       [08] 0 ''
       [09] 0 ''
```

对于任意的地址，也可以手动指定符号类型来格式化显示。比如把 `0x0039ba80` 地址上的数据

用 MyCls 类型来显示:

```
0:000> dt 0x0039ba80 MyCls
   +0x000 str            : 0x00416648  "abcd"
   +0x004 inobj          : innner
```

除了上诉命令外,windbg 中还可以通过!heap 命令来检查操作系统中 heap 的详细信息具体内容可以参考

Debug Tutorial Part 3: The Heap
http://www.codeproject.com/debug/cdbntsd3.asp

Live Debug

接下来看看 live debug 中的一些有用的命令。Live debug 中最有价值的命令是:

1. wt 命令

wt 命令的作用是 watch and trace data。 它可以跟踪一个函数的所有执行过程，并且给出统计信息。。

2. 设定断点

windbg 里面可以设定灵活而强大的条件断点.比如可以通过条件断点实现这样的功能:

当某个全局变量在被修改 100 次以后，同时 stack 上的第二个参数是 100，那么就停下来进入调试模式;如果第二个参数是 200，那么就生成一个 dump 文件，否则就只打印出当前的 callstack,然后继续运行

wt

下面首先看 wt 的使用，然后再讨论条件断点:
还是对于上面那个程序:

首先用 bp (break point) 命令在 foo3 上面设断点

```
0:001> bp exceptioninject!foo3
breakpoint 0 redefined
```

然后用 g 命令让程序执行

```
0:001> g
```

执行到 foo3 上的时候，调试器停下来了

```
Breakpoint 0 hit
eax=0000000a ebx=7ffd7000 ecx=0043780e edx=10310bd0 esi=0012fe9c edi=0012ff68
```

```
eip=004114c0 esp=0012fe98 ebp=0012ff68 iopl=0         nv up ei pl zr na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
exceptioninject!foo3:
004114c0 55              push    ebp
```

用 bd (breakpoint disable)命令取消设定好的断点，以免打扰 wt 的执行

```
0:000> bd 0
```

用 wt 命令监视 foo3 的执行，深度设定成 2 (-l2 参数)

```
0:000> wt -l2
Tracing exceptioninject!foo3 to return address 0041186a
   60    0 [  0] exceptioninject!foo3
   28    0 [  1]   MSVCR80D!printf
    5    0 [  2]     MSVCR80D!__iob_func
   32    5 [  1]   MSVCR80D!printf
   12    0 [  2]     MSVCR80D!_lock_file2
   35   17 [  1]   MSVCR80D!printf
    5    0 [  2]     MSVCR80D!__iob_func
   38   22 [  1]   MSVCR80D!printf
   50    0 [  2]     MSVCR80D!_stbuf
   46   72 [  1]   MSVCR80D!printf
    5    0 [  2]     MSVCR80D!__iob_func
   49   77 [  1]   MSVCR80D!printf
  575    0 [  2]     MSVCR80D!_output_l
   52  652 [  1]   MSVCR80D!printf
    5    0 [  2]     MSVCR80D!__iob_func
   57  657 [  1]   MSVCR80D!printf
   33    0 [  2]     MSVCR80D!_ftbuf
   60  690 [  1]   MSVCR80D!printf
    7    0 [  2]     MSVCR80D!printf
   71  697 [  1]   MSVCR80D!printf
   63  768 [  0] exceptioninject!foo3
    1    0 [  1]   exceptioninject!ILT+380(__RTC_CheckEsp)
    2    0 [  1]   exceptioninject!_RTC_CheckEsp
   64  771 [  0] exceptioninject!foo3
    1    0 [  1]   exceptioninject!ILT+340(?foo2YAXXZ)
   60    0 [  1]   exceptioninject!foo2
   71    0 [  2]     MSVCR80D!printf
   63   71 [  1]   exceptioninject!foo2
    1    0 [  2]     exceptioninject!ILT+380(__RTC_CheckEsp)
    2    0 [  2]     exceptioninject!_RTC_CheckEsp
   64   74 [  1]   exceptioninject!foo2
    1    0 [  2]     exceptioninject!ILT+215(?foo1YAXXZ)
  108    0 [  2]     exceptioninject!foo1
   70  183 [  1]   exceptioninject!foo2
    1    0 [  2]     exceptioninject!ILT+380(__RTC_CheckEsp)
```

```
 2    0 [  2]    exceptioninject!_RTC_CheckEsp
73   186 [  1]   exceptioninject!foo2
70  1031 [  0] exceptioninject!foo3
 1    0 [  1]   exceptioninject!ILT+380(__RTC_CheckEsp)
 2    0 [  1]   exceptioninject!_RTC_CheckEsp
73  1034 [  0] exceptioninject!foo3

1107 instructions were executed in 1106 events (0 from other threads)

Function Name                               Invocations MinInst MaxInst AvgInst
MSVCR80D!__iob_func                               4      5      5      5
MSVCR80D!_ftbuf                                   1     33     33     33
MSVCR80D!_lock_file2                              1     12     12     12
MSVCR80D!_output_l                                1    575    575    575
MSVCR80D!_stbuf                                   1     50     50     50
MSVCR80D!printf                                   3      7     71     49
exceptioninject!ILT+215(?foo1YAXXZ)               1      1      1      1
exceptioninject!ILT+340(?foo2YAXXZ)               1      1      1      1
exceptioninject!ILT+380(__RTC_CheckEsp)           4      1      1      1
exceptioninject!_RTC_CheckEsp                     4      2      2      2
exceptioninject!foo1                              1    108    108    108
exceptioninject!foo2                              1     73     73     73
exceptioninject!foo3                              1     73     73     73

0 system calls were executed

eax=00000073 ebx=7ffd7000 ecx=00437c7e edx=10310bd0 esi=0012fe9c edi=0012ff68
eip=0041186a esp=0012fe9c ebp=0012ff68 iopl=0         nv up ei pl zr na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
exceptioninject!wmain+0x4a:
0041186a ebe1           jmp     exceptioninject!wmain+0x2d (0041184d)
```

上面 wt 命令一直监视到 foo3 函数执行完为止。随着函数的执行，windbg 打印出 foo3 调用过的子函数。如果需要更详细的信息，可以不用-l2 来设定 2 的深度。

wt 命令最后给出统计信息。无论是观察函数执行过程和分支，或者是性能评估上，wt 命令都是很有帮助的。

条件断点 (condition breakpoint)

Windbg 中的断点分为三种,命令和功能如下:

1. bp+地址/函数名字可以在某个地址上设定断点。当程序运行到这个地址的时候断点触发.

2. ba (break on access)用来设定访问断点，在某个地址被读/写的时候断点触发。
3. Exception 断点。当发生某个 Exception/Notification 的时候断点触发。详情请参考 windbg 帮助中的 sx(Set Exception)小结。

条件断点(condition breakpoint)的是指在上面三种基本断点停下来后，执行一些自定义的判断。详细说明参考 windbg 帮助中的 Setting a Conditional Breakpoint 小结。

在基本断点命令后加上自定义调试命令,可以让调试器在 address 断点触发停下来后，执行调试器命令。每个命令之间用分号分割。

下面这个命令，在 exceptioninject!foo3 上设断点，每次断下来后，先用 k 显示 callstack, 然后用.echo 命令输出简单的字符串'breaks'，最后 g 命令继续执行

```
0:001> bp exceptioninject!foo3 "k;.echo 'breaks';g"
breakpoint 0 redefined
0:001> g
ChildEBP RetAddr
0012fe94 0041186a exceptioninject!foo3
[d:\xiongli\today\exceptioninject\exceptioninject\exceptioninject.cpp @ 79]
0012ff68 00412016 exceptioninject!wmain+0x4a
[d:\xiongli\today\exceptioninject\exceptioninject\exceptioninject.cpp @ 93]
0012ffb8 00411e5d exceptioninject!__tmainCRTStartup+0x1a6
[f:\rtm\vctools\crt_bld\self_x86\crt\src\crtexe.c @ 583]
0012ffc0 77e523cd exceptioninject!wmainCRTStartup+0xd
[f:\rtm\vctools\crt_bld\self_x86\crt\src\crtexe.c @ 403]
0012fff0 00000000 kernel32!BaseProcessStart+0x23
'breaks'
ChildEBP RetAddr
0012fe94 0041186a exceptioninject!foo3
[d:\xiongli\today\exceptioninject\exceptioninject\exceptioninject.cpp @ 79]
0012ff68 00412016 exceptioninject!wmain+0x4a
[d:\xiongli\today\exceptioninject\exceptioninject\exceptioninject.cpp @ 93]
0012ffb8 00411e5d exceptioninject!__tmainCRTStartup+0x1a6
[f:\rtm\vctools\crt_bld\self_x86\crt\src\crtexe.c @ 583]
0012ffc0 77e523cd exceptioninject!wmainCRTStartup+0xd
[f:\rtm\vctools\crt_bld\self_x86\crt\src\crtexe.c @ 403]
0012fff0 00000000 kernel32!BaseProcessStart+0x23
'breaks'
```

更复杂一点的例子是:

```
int i=0;
int _tmain(int argc, _TCHAR* argv[])
{
```

```
    while(1)
    {
        getchar();
        i++;
        foo3();
    }
    return 0;
}
```

条件断点的命令是:

ba w4 exceptioninject!i "j (poi(exceptioninject!i)<0n40) '**printf** \"exceptioninject!i value is:%d\",**poi**(exceptioninject!i);.echo;g';'.echo stop!'"

分开来看。首先 ba w4 exceptioninject!i 表示在修改 exceptioninject!i 这个全局变量的时候停下来

J(judge)命令的作用是对后面的表达式做条件判断，如果为 true，执行第一个单引号里面的命令，否则执行第二个单引号里面的命令

条件表达式是(poi(exceptioninject!i)<0n40)。 在 windbg 中，exceptioninject!i 符号表示符号所在的内存地址，而不是改符号的数值，相当于 C 语言中的 &操作符的作用。Windbg 命令 poi 的作用是取这个地址上值，相当于 C 语言中的*操作符。所以这个条件的意思就是判断 exceptioninject!i 的值，是否小于十进制(windbg 中十进制用 0n 当前缀)的 40。

如果为真，那么就执行第一个单引号:
printf \"exceptioninject!i value is:%d\",poi(exceptioninject!i);.echo;g

这一个单引号里面有三个命令: .printf, .echo 和 g. 这里的 printf 语法跟 C 中 printf 函数语法一样。不过由于这个 printf 命令本身是在 ba 命令的双引号里面，所以需要用\来转义 printf 中的引号。转义的结果是:

printf “exceptioninject!i valus is %d”, poi(exceptioninject!i)

第一个单引号命令的作用是
1)打印出当前 exceptioninject!i 的值,2).echo 命令换行 3)g 命令继续执行

如果为假，那么就执行第二个单引号: .echo stop! 这个命令就是显示 stop, 由于后没有没有 g 命令，所以 windbg 会停下.运行输出如下:

```
0:001> ba w4 exceptioninject!i "j (poi(exceptioninject!i)<0n40) '.printf
\"exceptioninject!i value is:%d\",poi(exceptioninject!i);.echo;g';'.echo
stop!'"
```

```
breakpoint 0 redefined
0:001> g
exceptioninject!i value is:35
exceptioninject!i value is:36
exceptioninject!i value is:37
exceptioninject!i value is:38
exceptioninject!i value is:39
stop!
eax=00000028 ebx=7ffd5000 ecx=5e186b9c edx=10310bd0 esi=0012fe9c edi=0012ff68
eip=00411872 esp=0012fe9c ebp=0012ff68 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
exceptioninject!wmain+0x52:
00411872 e856f8ffff      call exceptioninject!ILT+200(?foo3YAXXZ) (004110cd)
```

伪寄存器

调试器还提供了伪寄存器的功能。

考虑这样的情况，如果要记录某一个函数被执行了多少次，应该怎么做？简单的做法就是修改代码，在对应的函数入口做记录。可是，如果要记录的函数是系统 API 呢？

下面的例子介绍了如何统计 VirtualAllocEx 被执行了多少次:

bp kernel32!VirtualAllocEx "r $t0=@$t0+1;.printf \"function executes: %d times \",@$t0;.echo;g"

这里用到的$t0 就是 windbg 提供的伪寄存器。可以用来存储中间信息。这里用它来存储函数执行的次数。中间 r 命令可以用来查看，修改寄存器(CPU 寄存器和 windbg 的伪寄存器都有效)的值。随便挑一个繁忙的进程，用这个命令设定断点后观察:

```
0:009> bp kernel32!VirtualAllocEx "r $t0=@$t0+1;.printf \"function executes:
%d times \",@$t0;.echo;g"
0:009> g
function executes: 1 times
function executes: 2 times
function executes: 3 times
function executes: 4 times
…..
```

关于为寄存器信息，可以参考帮助文档中的 Pseudo-Register Syntax 小结。

Step Out 的实现

Step Out 的定义是” Target executes until the current function is complete.”。windbg 中是如何实现这个功能的呢？根据这个定义，可以简单地在当前函数的返回地址上设定 bp 断点就可以了。当前函数的返回地址保存在函数入口时候的 EBP+4 上。但如果简单地在 EBP+4 上面设定断点有两个问题:

1. 无法区分递归调用和函数返回，甚至其它线程对该地址的调用
2. 第一次触发后不会自动清除端点，可能会多次触发。

如果观察 windbg 中 step out 的实现，可以看到:

bp /1 /c @$csp @$ra;g

这里的/1 参数使得断点在触发后自动清除，避免了第二个问题，/c @$csp 参数通过指定 callstack 的最小深度避免了第一个问题。而$ra 伪寄存器直接表示当前函数的返回地址。多方便 :-)

Remote debug

在 windbg 中，可以用.server 命令在本地创建一个 TCP 端口，或者通过 named pipe, 使得远程的 windbg 可以连接到本地调试。双方都可以输入命令，执行结果在双方的 windbg 上都显示出来。

在调试 WinForm 程序的时候，如果要保持目标程序一直全屏运行，就没办法在同一台机器上使用调试器输入命令，检查结果。这种情况下，就可以使用 remote debug，避免调试器对目标程序的干扰。

Debugger auto attach

该功能在前面就已经示范过:
How to debug Windows services
http://support.microsoft.com/?kbid=824344

Windbg command line

该功能也已经在前面示范过。这里再举一个非常现实的例子:

买了中文版的魔兽争霸，但家里的 Windows 却是英文版。中文的魔兽争霸必须要运行到中文的操作系统上，否则就报告操作系统语言不匹配。怎么办呢？重装系统？去网上找破解？其实 windbg 就可以解决问题。

首先，获取系统语言版本的 API 是 GetSystemDefaultUILanguage。用 windbg 启动 war3.exe, 然

后在 GetSystemDefaultUILanguage 上设定断点，果然触发了。在调用完这个 API 后，war3.exe 的下一条语句就是一个 cmp eax, ChineselanID, 判断当前是否是中文系统。那好，在 cmp 这条语句上设定断点，然后用下面的命令把 eax 修改成中文语言符的 ID，接下来再让 cmp 语句执行:

r eax = 0n2052

既然 eax 被修改成了中文的语言符，接下来的 cmp 执行结果就跟中文系统上一样的了，war3 就可以正确运行了。每次都要做这样的修改很麻烦得，为了简化这个过程，创建内容为如下的 script 文件，在 GetSystemDefaultUILanguage API 返回前把 ax 设定为 0n2052:

bp kernel32!GetSystemDefaultUILanguage+0x2c "r ax=0n2052;g"

然后采用 How to debug Windows Services 里面的方法，创建 war3.exe 的键，这样每次 war3.exe 启动的时候自动启动 windbg, 通过-cf 参数自动执行我们的 script 来达到欺骗 war3 的目的。

Adplus

该工具是跟 windbg 在同一个目录的 VBS 脚本。Adplus 主要是用来抓取 dump 文件。 详细的信息，可以参考 windbg 帮助文件中关于 adplus 的帮助。有下面一些常见用法：

假设我们的目标程序是 test.exe:
架设 test.exe 运行一段时间崩溃，在 test.exe 启动后崩溃前的时候，运行下面的命令监视:

Adplus –crash –pn test.exe –o C:\dumps

当 test.exe 发生 2nd chance exception 崩溃的时候，adplus 在 C:\dumps 生成 full dump 文件。当发生 1st chance AV exception 的时候，adplus 在 C:\dumps 生成 mini dump 文件。

也可以用:

Adplus –crash –pn test.exe –fullonfirst –o C:\dumps

差别在于，加上-fullonfirst 参数后，无论是 1st chance 还是 2nd chance, 都会生成 full dump 文件。

假如 test.exe 发生 deadlock, 或者 memory leak, 并不是 crash，需要获取任意时刻的一个 dump, 可以用下面的命令:

Adplus –hang –pn test.exe –o C:\dumps

该命令立刻把 test.exe 的 full dump 抓到 C:\dumps 下

Adplus 更灵活的方法就是用-c 参数带配置文件。在配置文件里面，可以选择何种 exception

发生的时候生成 dump，是 mini dump 还是 full dump，还可以设定断点等等。

Debugger Extension

Debugger Extension 相当于是用户自定义，可编程的 windbg 插件。一个最有用的 extension 就是.NET Framework 提供的 sos.dll. 它可以用来检查.NET 程序中的内存，线程，callstack, appdomain, assembly 等等信息。后面会做详细讲解。

2.6 同步和锁

该小节介绍线程，进程间协同工作，处理数据中可能会发生的一些问题。演示了在 windbg 中如何查看 handle 和 CriticalSection 的信息，如何使用 AppVerifier 来侦测 CriticalSection Leak，最后通过案例说明线程同步可能导致的崩溃。

三个问题

同步是为了多线程/进程下避免资源竞争，提高系统的整体性能和可伸缩性，往往通过 Kernel Object , Critical Section 和 Windows Message 来实现。但是，不恰当的使用，往往是下面这些问题的罪魁祸首:

1. Handle Leak
2. 死锁
3. 线程争用

Handle Leak

随着程序的运行，任务管理器里观察到程序创建的 handle 数量越来越多。原因很简单: 没有及时调用 CloseHandle API。 一个常见的情况是，在调用 CreateThread 以后，很多开发人员不会调用 CloseHandle 来关闭创建出来的 thread 的 handle。有人的理由(错误)是：当 thread 自动退出，对应 handle 会自动关闭。

解决 handle leak 的思路是，首先观察 handle 增长的情况，然后找到增长出来的 handle 是什么类型，是怎么创建出来的。

Deadlock

MSDN 上的定义是: “In multithreaded applications, a threading problem that occurs when each member of a set of threads is waiting for another member of the set.” 现实一点来说，死锁就是该跑的线程不跑了，老在等某一个东西，往往是在 WaitForSingleObject 或者是 EnterCriticalSection 上不返回。从定义上来说，死锁应该是多个 thread 互相依赖，互相等待。但是现实中的情况却广泛得多，这里把程序挂起(hang)也归为死锁(deadlock)。最近遇上的一些死锁情况是:

情况 1:

ASP.NET 页面都打不开。发现所有的工作线程，都等待在下面的 managed call stack (.NET

程序 CLR 的相互调用)上:

```
System.Data.OracleClient.TracedNativeMethods.OCIServerAttach
System.Data.OracleClient.OracleInternalConnection.OpenOnLocalTransaction
System.Data.OracleClient.OracleInternalConnection.Open
System.Data.OracleClient.OracleInternalConnection..ctor
System.Data.OracleClient.OracleConnection.OpenInternal
System.Data.OracleClient.OracleConnectionString.Demand
MethodDesc 0x26ec9f20 +0x21 System.Data.OracleClient.OracleConnection.Open
```

对应的 unmanaged call stack 是:

```
NTDLL!NtWaitForSingleObject+0xb
msafd!SockWaitForSingleObject+0x1a8
msafd!WSPRecv+0x1e9
ws2_32!WSARecv+0x8a
orantcp8!nttini+0x40ef
orantcp8!nttini+0x1f01
oran8!ztch+0x16204
oran8!nsrdr+0x993
oran8!nsdo+0x9a0
oran8!nsnactl+0x430
```

这里可以看到，所有的线程都 block 在 Oracle 数据库的 open 操作上。这个 open 操作一路掉上来，调到了 WSARecv 这个 API, 等待网络上的数据包。也就是说，数据库不返回数据，这边的 ASP.NET 线程就没办法继续跑。细心一点，你会看到, WSARecv 这个 API，也是调用了 WaitForSingleObject 在等待的。

情况 2:

客户的一个 MFC 程序需要异步进行网络操作。于是客户使用了 MFC 中的 CAsyncSocket 类。程序对于某些特定网络请求无法返回，整个 UI 都僵死了。

在 CAsyncSocket 对应的网络处理函数上设断点，发现断点没有触发。检查后发现，CAsyncSocket 的实现是依靠 Windows Message 来派发消息，调用客户自定义的网络处理函数的。在处理某些请求的时候，客户的主线程会在一个 Event 上面等待，这个 Event 会在客户的网络处理函数中激活。

问题在于客户调用 WaitForSingleObject 在主线程等待的时候，主线程的 Windows Message 队列就停止派发了。消息队列停止后，CAsyncSocket 接受到了网络包后没办法通过 Windows Message 通知自定义处理函数执行。自定义处理函数不执行，Event 就不会触发，主线程就不会结束等待。

所以主线程等处理函数执行，处理函数却依赖于主线程的消息队列，死锁就发生了。解决方

法是不要使用 CAsyncSocket；或者等待 Event 的时候，用 MsgWaitForMultipleObjects 来等待 Event, 保持消息队列。

情况 3:

客户的多线程程序，需要在多个线程中共享某一个全局资源。所以每一个线程在使用这个资源以前，先调用 EnterCriticalSection 进入一个公用的 CriticalSection, 然后使用资源，最后调用 LeaveCriticalSection 释放这次争用。这是很经典的同步。客户发现程序运行一段时间后，所以的线程都停住了。经过分析，发现某一个线程，在进入这个 CriticalSection 后，由于发生了某些错误，在调用 LeaveCriticalSection 以前就退出了。于是这个 CriticalSection 永远都无法释放，导致其他的 thread 永远停在了 EnterCriticalSection 上。这叫 CriticalSection Leak。

死锁的共同点都是应该运行的线程在等待某种资源，这个资源可能跟 CriticalSection 相关，可能跟 Windows Message 相关，可能跟网络相关，或者跟自己的程序相关。问题发生时候的 CPU 使用率保持为 0。

对死锁的排错，其实就是回答下面三个问题:

1. 在等什么
2. 等待的时机是否恰当
3. 等待的东西什么时候释放？

线程争用（颠簸）

考虑这样的情况，100 个线程，在等同一个 CriticalSection. 当这个 CriticalSection 释放的时候，100 个线程就去抢这个 CriticalSection, 但是最终只有一个线程抢到，然后那个线程开始执行，剩下的 99 个线程继续等待……

这样的线程争用现象往往发生在:

1. 启动了太多的工作线程
2. 这些工作线程往往要等待同一个 object

线程争用带来的后果是性能下降。如果打开性能监视器，可以看到 CPU 的使用率上下跳动非常频繁。每当争用的 object 释放的时候，CPU 的使用率会立刻上升，并且持续一段时间。其实这些消耗的 CPU 时间，主要用在操作系统调度线程，切换线程上下文的开销上。线程争用往往是设计上的缺陷导致的，可以采取下面的步骤来定位:

1. 观察整体性能情况和 CPU 使用情况
2. 使用性能监视器，抓取 System\Context Switches/sec 情况。如果这个 counter 比较高, 说明线程切换很多，很可能发生 thread contention.
3. 在 CPU 波动的时候抓取 dump 文件，看看是否有很多 thread 在等待相同的 object

对于线程争用，后面的章节有多个案例进行剖析。

Windbg 中的相关操作

windbg 中有几个可以用来检查 handle 和 critical section 的命令，可以很方便地调试 deadlock 和 handle leak:

!handle 可以获取整个进程，或者某一个 handle 的详细信息
!htrace 可以获取某一个 handle 创建时候的 call stack
!cs 可以获取一个 CriticalSection 的详细信息

单单这些命令还不够，因为!htrace 的功能类似 pageheap，需要修改注册表来激活的。同时，对于上面说的 CriticalSection Leak，就算知道在等待某一个不可能释放的 CriticalSection，但是若找不出 CriticalSection 泄漏的根源，也无法解决问题。

Application Verifier 是类似 pageheap 的工具，使用范围更广。可以通过 Application Verifier 方便设定注册表，让操作系统主动地提供关于 handle, critical section, heap 等系统层面的调试信息.

Application Verifier
http://www.microsoft.com/technet/prodtechnol/windows/appcompatibility/appverifier.mspx

安装后，运行 Application Verifier，添加 notepad.exe，然后激活 handles 选项，启动 notepad.exe，加载 windbg (!htrace 在当前 windows 系统只能用于 live debug，dump 中不保存 handle trace 信息):

!handle

首先运行一下!handle，可以看到当前进程的每一个 handle 类型，以及统计信息:

```
0:001> !handle
Handle 4
  Type         Key
Handle c
  Type         KeyedEvent
Handle 10
  Type         Event
…
45 Handles
Type          Count
Event         7
Section           3
File          5
```

```
Port          1
Directory         2
Mutant        7
WindowStation     2
Semaphore         3
Key           7
Thread        1
Desktop           1
IoCompletion      5
KeyedEvent        1
```

然后找一个 Key,察看详细信息:

```
0:001> !handle 4 f
Handle 4
  Type        Key
  Attributes      0
  GrantedAccess   0x8:
        None
        EnumSubKey
  HandleCount     2
  PointerCount    3
  Name        \REGISTRY\MACHINE\SOFTWARE\Microsoft\Windows
NT\CurrentVersion\Image File Execution Options
  Object Specific Information

0:001> !handle 384 f
Handle 384
  Type        Mutant
  Attributes      0
  GrantedAccess   0x1f0001:
        Delete,ReadControl,WriteDac,WriteOwner,Synch
        QueryState
  HandleCount     34
  PointerCount    36
  Name        \BaseNamedObjects\MSCTF.Shared.MUTEX.EKBB
  Object Specific Information
    Mutex is Free
```

对于注册表键,!handle 可以看到这对应的注册表位置是什么。如果 handle 不是匿名 handle，能看到对应的名称。

简单的!handle 命令也发挥过大用途的:

案例 1: 在排查某一个 handle leak 问题的时候，用!handle 命令发现泄漏的 handle 类型是注册表键，而且这些键都指向客户一个自定义的注册表。通过检查源代码中对这个注册表的访问，找到问题所在

案例 2: 在排查某一个 deadlock 问题的时候，发现死锁的线程都是在等待客户自定义的 event handle.客户创建 event 的时候都使用了一些很有描述性的名字，所以用!handle 打印出相关的的 event 名字后，客户一下子就知道应该如何入手检查了。

!htrace

在用 Application Verifier 激活 handle 检查后，!htrace 命令可以打印出跟指定 handle 的最近几次 callstack:

```
0:001> !htrace 384
-------------------------------------
Handle = 0x00000384 - OPEN
Thread ID = 0x000016bc, Process ID = 0x00000810

0x77e56a6d: kernel32!CreateMutexA+0x00000066
0x4b8d4178: MSCTF!CCicMutex::Init+0x00000016
0x4b8d4092: MSCTF!CSharedBlockNT::Init+0x000000ee
0x4b8dcb5a: MSCTF!EnsureSharedBlockForThread+0x000000e1
0x4b8d9afb: MSCTF!HandleSendReceiveMsg+0x00000095
0x4b8d9a5c: MSCTF!CicMarshalWndProc+0x00000161
0x7739c3b7: USER32!InternalCallWinProc+0x00000028
0x7739c484: USER32!UserCallWinProcCheckWow+0x00000151
0x7739c73c: USER32!DispatchMessageWorker+0x00000327
0x7739c778: USER32!DispatchMessageW+0x0000000f
```

在后面的章节中，有专门介绍如何使用!handle 和!htrace 来检查 handle leak 的模板。

!cs

对于 CriticalSection,有专门的命令!cs:

下面的 callstack 中，线程 13 堵塞在 CriticalSection 上，CriticalSection 的 ID 是 RtlEnterCriticalSection 的第一个参数 00a95da4

```
0:013> kb
ChildEBP RetAddr  Args to Child
0189fed0 7c822124 7c83970f 0000071c 00000000 ntdll!KiFastSystemCallRet
0189fed4 7c83970f 0000071c 00000000 00000000 ntdll!NtWaitForSingleObject+0xc
0189ff10 7c839620 00000000 00000004 00a95da0 ntdll!RtlpWaitOnCriticalSection+0x19c
```

```
0189ff30 0040109a 00a95da4 00401fdd 00000000 ntdll!RtlEnterCriticalSection+0xa8
0189ff38 00401fdd 00000000 00000000 00000000 critsec_demo!CCritSec::Enter+0xa
0189ffb8 77e66063 00000000 00000000 00000000 critsec_demo!ThreadProc+0xcd
0189ffec 00000000 00401f10 00000000 00000000 kernel32!BaseThreadStart+0x34
```

!cs 命令可以方便地检查该 CriticalSection 的信息:

```
0:013> !cs 00a95da4
-----------------------------------------
Critical section   = 0x00a95da4 (+0xA95DA4)
DebugInfo          = 0x0015a9d0
LOCKED
LockCount          = 0x7
WaiterWoken        = No
OwningThread       = 0x00001a4c
RecursionCount     = 0x2
LockSemaphore      = 0x71C
SpinCount          = 0x00000000
```

这里 OwnningThread 显示的是 CirticalSection owner thread 的 id, 用~~[]命令可以把 id 转换成线程号:

```
0:013> ~~[0x00001a4c]
  10  Id: d3c.1a4c Suspend: 1 Teb: 7ffd4000 Unfrozen
      Start: critsec_demo!ThreadProc (00401f10)
      Priority: 0  Priority class: 32  Affinity: 3
```

看到这个 CriticalSection 的 owner 是 thread 10, 于是切换到 thread 10:

```
0:013> ~10s
eax=00000001 ebx=00000000 ecx=0159fa94 edx=00000019 esi=00a95e24 edi=00000720
eip=7c82ed54 esp=0159fed4 ebp=0159ff10 iopl=0         nv up ei pl zr na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
ntdll!KiFastSystemCallRet:
7c82ed54 c3              ret
0:010> kb
ChildEBP RetAddr  Args to Child
0159fed0 7c822124 7c83970f 00000720 00000000 ntdll!KiFastSystemCallRet
0159fed4 7c83970f 00000720 00000000 00000000 ntdll!NtWaitForSingleObject+0xc
0159ff10 7c839620 00000000 00000004 00a95e20 ntdll!RtlpWaitOnCriticalSection+0x19c
0159ff30 0040109a 00a95e24 00401fdd 00000000 ntdll!RtlEnterCriticalSection+0xa8
0159ff38 00401fdd 00000000 00000000 00000000 critsec_demo!CCritSec::Enter+0xa
0159ffb8 77e66063 00000000 00000000 00000000 critsec_demo!ThreadProc+0xcd
0159ffec 00000000 00401f10 00000000 00000000 kernel32!BaseThreadStart+0x34
```

看到 thread10 也在等待一个 CriticalSection

```
0:010> !cs 00a95e24
-----------------------------------------
Critical section   = 0x00a95e24 (+0xA95E24)
DebugInfo          = 0x0015aa20
LOCKED
LockCount          = 0x12
WaiterWoken        = No
OwningThread       = 0x00001fc8
RecursionCount     = 0x2
LockSemaphore      = 0x720
SpinCount          = 0x00000000
0:010> ~~[0x00001fc8]
  13  Id: d3c.1fc8 Suspend: 1 Teb: 7ff9e000 Unfrozen
      Start: critsec_demo!ThreadProc (00401f10)
      Priority: 0  Priority class: 32  Affinity: 3
```

哈哈，thread 10 在等 thread 13 拥有的 CriticalSection, thread 13 在等 thread 10 拥有的 CriticalSection. 经典的 deadlock。

CriticalSection leak(Orphan CriticalSection)

再看另外一种情况,测试代码如下:

```
#include "stdafx.h"
#include "windows.h"

CRITICAL_SECTION g_cs;

DWORD WINAPI ThreadProc1(LPVOID lpParameter)
{
    EnterCriticalSection(&g_cs);
    printf("thread1\n");
    ExitThread(0);
    return 0;
}

DWORD WINAPI ThreadProc2(LPVOID lpParameter)
{
    EnterCriticalSection(&g_cs);
    printf("thread2\n");
    return 0;
}
int _tmain(int argc, _TCHAR* argv[])
```

```
{
    InitializeCriticalSection(&g_cs);
    DWORD id;
    CreateThread(NULL,NULL,ThreadProc1,NULL,NULL,&id);
    Sleep(1000);
    CreateThread(NULL,NULL,ThreadProc2,NULL,NULL,&id);

    getchar();
    return 0;
}
```

运行起来后，启动 windbg 观察:

```
0:001> kb
ChildEBP RetAddr  Args to Child
00bbfe80 7c822124 7c83970f 0000078c 00000000 ntdll!KiFastSystemCallRet
00bbfe84 7c83970f 0000078c 00000000 00000000 ntdll!NtWaitForSingleObject+0xc
00bbfec0 7c839620 00000000 00000004 00000000 ntdll!RtlpWaitOnCriticalSection+0x19c
00bbfee0 00411b0b 004294e0 00000000 00000000 ntdll!RtlEnterCriticalSection+0xa8
00bbffb8 77e66063 00000000 00000000 00000000 CSDrop!ThreadProc2+0x2b
00bbffec 00000000 004112e4 00000000 00000000 kernel32!BaseThreadStart+0x34
0:001> !cs 004294e0
-----------------------------------------
Critical section   = 0x004294e0 (CSDrop!g_cs+0x0)
DebugInfo          = 0x0015a628
LOCKED
LockCount          = 0x1
WaiterWoken        = No
OwningThread       = 0x000014c4
RecursionCount     = 0x1
LockSemaphore      = 0x78C
SpinCount          = 0x00000000
0:001> ~~[0x000014c4]
                  ^ Illegal thread error in '~~[0x000014c4]'
```

上面的信息说明，thread 14c4 已经消失了。这种情况，往往是由于某一个线程在调用 LeaveCriticalSection 前就退出了。要解决这个问题，只观察问题发生后的情况是没有多大帮助的，问题的根源在前面就发生了，关键是要找到什么线程退出却忘记调用 LeaveCriticalSection。

解决的方法，就是前面提到的 Application Verifier。启动 Application Verifier 后，添加程序名字，然后勾选 Locks。在 windbg 中用 ctrl+E 打开这个程序重新运行，看到效果了吗？

首先是 windbg 被 break point exception 停下来了，然后看到下面的消息:

```
0:000> g


=======================================
VERIFIER STOP 00000200 : pid 0x15F4: Thread cannot own a critical section.

  00001C40 : Thread ID.
  004294E0 : Critical section address.
  7C889560 : Critical section debug information address.
  004318E8 : Critical section initialization stack trace.


=======================================
This verifier stop is continuable.
After debugging it use `go' to continue.

=======================================
```

然后用 k 命令看看:

```
0:001> k
ChildEBP RetAddr
0194fa6c 003e3368 ntdll!DbgBreakPoint
0194fc3c 01441ffe vrfcore!VerifierStopMessageEx+0x3d3
0194fc60 0143726c vfbasics!VfBasicsStopMessage+0x8e
0194fd20 014372cb vfbasics!AVrfpCheckCriticalSection+0x2ac
0194fd40 01436f1b vfbasics!AVrfpCheckCriticalSectionSplayNode+0x2b
0194fe64 01436e3f vfbasics!AVrfpCheckCriticalSectionTree+0xbb
0194fe74 0143502e vfbasics!AVrfCheckForOrphanedCriticalSections+0x4f
0194fe80 01435063 vfbasics!AVrfpCheckThreadTermination+0xe
0194fe90 01434aff vfbasics!AVrfpCheckCurrentThreadTermination+0x23
0194fea0 00412049 vfbasics!AVrfpExitThread+0x1f
0194ff78 01434e8f CSDrop!ThreadProc1+0x49
0194ffb8 77e66063 vfbasics!AVrfpStandardThreadFunction+0x6f
0194ffec 00000000 kernel32!BaseThreadStart+0x34
```

这里可以看到，ThreadProc1 试图退出当前线程，但是由于还拥有没有释放的 CriticalSection, 所以在激活 Application Verifier 后，就会自动产生一个 break point exception.

通过上面的例子，可以看到对于同步问题，只要采取正确的工具，其实是很容易找到问题所在的。

Invalid handle exception

在 Application Verifier 中激活 handle 后，除了会记录 handle 的 callstack 外，当误用一个 handle 的时候，系统也会自动产生一个 break point exception， 比如下面这段代码:

```
int _tmain(int argc, _TCHAR* argv[])
{
    DWORD d=WaitForSingleObject((HANDLE)-1,0);
    DWORD gle=GetLastError();
```

这里的 gle 等于 6，表示 The handle is invalid。如果不在 Application Verifier 中激活 handle，其间不会有异常发生，程序也不会被崩溃。程序员如果要捕捉到这个错误，就必须检查 gle 的值。

问题在于好多程序员在代码中不习惯检查函数返回值，也不习惯调用 GetLastError 找出具体的错误原因。由于 handle 的使用错误，很可能问题会在程序继续运行一段时间后才表现出来，给排错过程带来很多困难。

跟 heap 相关的问题一样，如果在 Application Verifier 中激活 handle 后，在 debugger 中重新运行，会看到 break point exception。Application Verifier 的这项功能能够让程序员通过 break point 的机制找到误用 handle 的地方，使得问题尽早暴露出来。

[案例研究， IndexOutOfRangeException]

问题描述

http://community.sgdotnet.org/forums/1/23223/ShowThread.aspx

这是一个用 VS2005 开发的 ASP.NET 程序。程序偶尔会报 Exception。 奇怪的地方是，报 Exception 每次都是这个函数:

System.Collections.ArrayList.Add

Exception 的详细信息是:

[IndexOutOfRangeException: Index was outside the bounds of the array.]

第一印象是访问 ArrayList 的时候越界了，下标计算错误。或者是 ArrayList 里面的元素太多了。客户调用 ArrayList.Add 的 callstack 是:

```
System.Collections.ArrayList.Add(System.Object)
SomeNameSpace.SomeClass..ctor(SomeNameSpace.SomeClass.SQLConnectionBlock)
```

```
SomeNameSpace.SomeClass.ReadFromDB2Identity(SomeNameSpace.SomeClass,
System.Data.IDataReader)
SomeNameSpace.SomeClass.ReAuthenticate(Int32)
```

你的第一印象是不是建议客户去检查 SomeClass 的构造函数？问问客户是怎么访问 ArrayList 的？

这个异常不简单

其实仔细思考一下，就会发现问题不是那么简单:

这里是调用 Add 方法添加一个元素到 ArrayList, 而不是直接访问。根据 MSDN 的定义，这里唯一可能出现的 Exception 是 NotSupportedException, 或者是 OutOfMemory。其次，客户强调说，在构造函数中，根本没有定义 constructor。那到底怎么回事？

无论是什么问题，都是需要证据来说话的。所以决定把问题发生时候的 dump 抓过来检查。根据前面的知识，这里可以使用 adplus 在 exception 发生的时候去抓 dump 文件。由于这个 Exception 最后会被 ASP.NET Runtime 捕获，不会导致进程崩溃，所以需要抓 1st chance exception 时候的 dump. 那么是否应该使用-FullOnFirst 参数呢？

如果使用-FullOnFirst, 的确能够在问题发生的时候抓下 dump 来，但是带来的负面效果是:

1. 所有的 1st chance exception 都会被保存下来。不管是 IndexOutOfRangeException, NullReferenceException, TimeOutException, Access Violation 等等，都会被记录下来。由于很多 Exception 是正常情况下也会产生的，是程序预料中的，所以使用-FullOnFirst 会拿到上百个 dump 文件，分析起来不现实
2. 如此频繁的生成 dump 文件，需要很大的磁盘空间。每产生一个 dump 会消耗 30 秒左右，对服务器性能也是极大的影响。

所以，用上面的方法来抓 dump 是不现实的。真正需要的，是发生.NET 的 IndexOutOfRangeException Exception 的时候的 dump。由于.NET 所有 Exception 的异常号(e0434f4d)都一样，所以单纯依靠 windbg 或者 adplus 是无法达到这个目的的。可选的做法是当.NET Exception 发生的时候，执行自定义的 script 去判断 Exception 里面的具体信息或者类型。好在.NET Framework 2.0 已经给提供了这样的 Debugger Extension SOS 来简化上面的工作。(关于 SOS 的内容，后面章节有详细介绍)所以，给客户建立了如下的 adplus 配置文件, 叫做 dump_clr_indexoutofrange.cfg:

```
<PreCommands>
      <Cmd> .load C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\sos.dll   </Cmd>
 </PreCommands>

 <PostCommands>
      <Cmd> sxd -c "!soe System.IndexOutOfRangeException 3;.if(@$t3==1) { .dump /mfh
/u c:\\dumps\\dump} .else {g}" clr </Cmd>
```

</PostCommands>

PreCommands 里面把 sos.dll 这个 extension 调用起来。PostCommands 里面通过 soe 这个命令设置条件，当 IndexOutOfRangeException 发生的时候，用.dump 命令生成 dump 文件。
上面的蓝色部分 sxd –c clr 表示当发生 clr exception 的时候，执行引号中的命令。引号中的红色部分，通过!soe extention 来判断当前 exception 是否是 OutOfRangeException。如果是，就通过.dump 命令抓取 dump 文件。

具体操作

下面的命令通过这个配置文件调用 adplus 抓 dump:

adplus -pn w3wp.exe -c dump_clr_indexoutofrange.cfg -o c:\dumps

拿到 dump 后，发现 call stack 和 exception 果然跟用户描述的一样:

```
System.Collections.ArrayList.Add(System.Object)
SomeNameSpace.SomeClass..ctor(SomeNameSpace.SomeClass.SQLConnectionBlock)
SomeNameSpace.SomeClass.ReadFromDB2Identity(SomeNameSpace.SomeClass,
System.Data.IDataReader)
SomeNameSpace.SomeClass.ReAuthenticate(Int32)
```

回到两个疑点:

1. Add 方法的确触发这个异常，跟 MSDN 描述违背，原因不明
2. 客户说没有定义 constructor，但是 callstack 上却看到了，客户在撒谎？

根据上面的疑点，采取下面的步骤进行分析:

使用 reflector 来分析 ArrayList.Add 方法的实现:

```
public virtual int Add(object value)
{
    if (this._size == this._items.Length)
    {
        this.EnsureCapacity(this._size + 1);
    }
    this._items[this._size] = value;
    this._version++;
    return this._size++;
}
```

发现 ArrayList 内部是通过_items 这个 Array 来操作的。如果访问_items 的时候发生 IndexOutOfRangeException，那么这个异常会直接抛出来。这里对_items 的下标访问只有一

句话，就是 this.items[this.size]=value。但是代码中的 if 语句已经保证了这里的下标不会越界。

同时，检查了一下 dump 中这个 ArrayList 的长度，也非常正常。也检查了一下其它的工作线程，发现工作线程一共有 70 来个，有的比较繁忙。

仔细想想，两行代码，第一行用来保证某一个 buffer 长度，第二行立刻操作新分配出来的 buffer，是不是应该注意什么呢？

结论

答案是同步。检查 ArrayList 的 MSDN 描述:

“Thread Safety
Public static (Shared in Visual Basic) members of this type are thread safe. Any instance members are not guaranteed to be thread safe.

An ArrayList can support multiple readers concurrently, as long as the collection is not modified. To guarantee the thread safety of the ArrayList, all operations must be done through the wrapper returned by the Synchronized method.

Enumerating through a collection is intrinsically not a thread safe procedure. Even when a collection is synchronized, other threads can still modify the collection, which causes the enumerator to throw an exception. To guarantee thread safety during enumeration, you can either lock the collection during the entire enumeration or catch the exceptions resulting from changes made by other threads.”

为了证明客户代码中是否正确地同步，需要看 ArrayList 在调用前，是否使用了 Synchronized method。同时，也可以检查用户是否定义了 Constructor。于是，再次使用 reflector，把 dump 中的 SomeNameSpace.SomeClass 所在的 DLL 用!sos.savemodule 保存到磁盘上，检查 SomeNameSpace.SomeClass..ctor 这个方法，看到:

```
Public Sub New(ByVal ConnBlock As SQLConnectionBlock)
    MyBase.New(ConnBlock)
    SomeClass.__ENCList.Add(New WeakReference(Me))
    Some other code...
End Sub
```

Reflector 明确地说明了这里使用了__ENCList 这个 ArrayList 变量，而且使用的时候并没有通过 Synchronized 保护起来。那用户为什么会说没有用到呢？

百思不得其解，先不管这一点，再次检查其他繁忙的 worker thread，发现有一个的 callstack 也是停留在 SomeNameSpace.SomeClass.ReadFromDB2Identity 附近的方法上。有了上面的证据，可以下结论，问题是由于缺乏同步导致的。当两个线程同时操作同一个 ArrayList 的 Add 方法而不同步，Add 方法中对于 Array 的长度判断的一致性的不到保证，当内部 Array 长度恰好等于 ArrayList 长度的时候，问题就发生了。

在做进一步努力以前，给客户通报了当前的分析情况。幸运的是，客户非常配合地也在使用 reflector 检查这个问题，他发现这个__ENCList 变量在 release 模式下编译的时候就没有。

有了这个线索，后面的事情就容易了。既然这个变量是编译器生成的，在本地也用 VS2005 写了一个简单的 assembly, 发现在本地就可以观察到 release/debug 模式下的区别。在做进一步研究后，发现这个__ENCList 是为了支持 VS2005 里面的新功能: Edit and Continue. 但是开发人员犯了一个错误，忘记同步 ArrayList 的使用。(这个 bug 在 VS2005 SP1 中已经修复)

从上面的例子可以看到，同步做得过多，做得不好，或者做得不够，都会导致问题。对于问题的分析，要以事实和证据为基础，结合线索进行思考，同时使用合适的工具来解决的。

2.7 调试和设计

作为本章的最后一个小结，简单地探讨一下调试的本质。

热心朋友的问题

本小结起源于一位热心朋友的反馈，建议讨论一下难于重现，环境复杂，无从琢磨，飘忽不定的问题，应该如何调试。这个问题也可以引申为:

1. 是否存在无从下手的问题
2. 什么样的问题，才算严重的问题

纵观前面的章节，排错其实是抓取信息，分析信息的过程。入手的关键在于抓取恰当的信息。前面介绍的工具主要是利用系统提供的功能来获取恰当的信息。

假设，系统没有提供这些功能的话，如果没有 pageheap，没有 dump，没有 Debug CRT，没有 AppVerifier，会怎么样？

换句话说，前面介绍的内容不过是熟悉一遍 Windows 提供的现有的功能，一种自检自查的功能。Windows 在设计的时候，已经聪明地把调试功能作为系统重要的一部分无缝融入。如果没有这样的设计，出问题后就无从下手。

调试功能应该是程序自身提供的一种功能，是程序内建的免疫系统，合理利用这项功能找到程序问题的过程就是调试。调试和开发相辅相成，调试的便捷，源于优秀的软件设计。

用 CLR Runtime 的调试作为一个参考。如果某个 CLR 的问题，在采用所有手段都无法对问题定位的时候，会建议:

1. 提供给客户一个 debug 版本的 CLR Runtime
2. 激活一个全局环境变量，该全局变量控制 CLR Runtime 运行时产生 log 的详细程度。问题发生后收集 log 文件分析。

这样做的原因在于:

1. 跟 Debug CRT 类似，Debug 版本的 CLR Runtime 会尽可能频繁地主动检查潜在的错误，使得潜在问题尽可能早地暴露出来。增加重现问题的可能性
2. Debug 版本的 CLR Runtime 会在运行时记录 log 文件。详细程度取决于全局环境变量的设定。问题发生后可以分析 log 文件来了解执行的详细信息。而全局环境变量的设定有助于控制 log 文件的尺寸，尽可能高效地获取信息

还可以开发适合自己使用的 CLR Debugger 来调试:

How can I use ICorDebug?
http://blogs.msdn.com/jmstall/archive/2004/10/05/237954.aspx

除了 CLR 以外，如果 Windows 系统有难于调试的问题，可以向客户提供 debug 版本来获取详细的信息。Windows 2000 Debug 版本的下载地址:

Windows 2000 SP4 Checked Build
http://www.microsoft.com/windows2000/downloads/servicepacks/sp4/sp4build/default.mspx

所以，对于前面两个问题的答案是:

1. 优秀的开发人员总会留下一条排错的捷径。比如 log 或者是 debug 版本。
2. 遇上愚蠢的开发人员，从不给程序的排错留后路，是一个严重的问题。

为了方便调试，优秀的开发人员至少会做到：

1. 多使用 assert, trace
2. 适当合理地添加 log
3. 每次编译发布后，都把 PDB 文件分不同的版本当宝贝一样保存在安全的地方。
4. 总是编译一个 release 版本，一个 debug 版本

有了上面的准备，无论问题多么神秘，下面两招总是有用的:

1. 部署 debug 版本
2. 收集程序的 log

最高的调试技巧是开发人员通盘的考虑跟合理的设计，让任何潜在的问题都可以水道渠成地解决。

题外话和相关讨论

[案例分析, 反被聪明误]

有位客户开发了一个性能敏感的程序。在设计阶段，客户已经考虑到了性能调优，所以在程序执行的每一个重要步骤前后，都获取当前系统时间，然后写入 log 文件。希望能通过 log 文件找到性能的瓶颈。客户还专门写了一个工具来用图形化界面分析生成的 log 文件。用来记录 log 的函数是:

```
public void SaveExecutionLog(string fun_stage,)
{
    XmlDocument logDoc=new XmlDocument();
    logDoc.LoadXml(LogFilePath);
```

```
        XPathNavigator navigator = logDoc.CreateNavigator();
        navigator.MoveToChild(PathPattern, ...);
        navigator.Insert(fun_stag,DateTime.Now);
        logDoc.Save();
    }
```

在调用重要方法的前后，都会调用SaveExecutionLog方法来记录log文件:

```
    public void CoreFoo()
    {
        SaveExecutionLog("CoreFoo enters");
        //do something
        SaveExecutionLog("CoreFoo quits");
    }
```

测试发现程序运行时间越来越长，处理一个请求的时间最开始只需要 2 秒钟，2 小时后上涨到惊人的 5 分钟。既然有完美的 log 机制，检查 log 一定能找到问题所在对吧。

事实让客户很失望，log 文件分析下来，得到一条非常平滑的曲线，显示执行时间均匀上升，而且均匀地分摊到客户所有的方法中，根本没有什么地方是瓶颈。你能看到问题发生在什么地方吗？

问题在于客户使用了 XML 格式作为 log 的存储介质。每次写一个 log 需要 parse 整个 XML，随着 log 文件长度的增加，parse XML 的时间自然就增加了。程序刚开始运行的时候，log 文件为 0 个字节，两个小时后，log 文件有 400 多 M。时间不是花费在程序的逻辑上，而是被程序的 log 功能消耗了。去掉 log 功能后，程序稳定运行。

Log 的同步

类似的问题还有 log 的同步。假设 web 程序使用单一文件作为 log 记录。如果不同步工作线程，很可能导致 log 中内容错位，甚至崩溃。如果加上 writer lock 来同步对 log 的写操作，又会导致性能问题。

其实，设计多线程环境下安全而且高性能的 log 操作是不太现实的。又要安全，又要高效，还要支持多线程，这不是让人设计一个数据库吗？所以，如果真的需要这样 log 功能，直接使用数据库来记录 log 才是明智的选择。

第三部分 CLR, 新的乐土

这部分介绍.NET Framework 上调试相关的知识。.NET Framework 的调试可以分成两部分，一是调试托管代码，比如开发人员用 C#写的方法。另一部分是调试.NET Framework 本身☺. .NET Framework 本身是微软使用非托管代码开发的。

调试.NET Framework 本身有趣的地方在于能够理解.NET Framework 的工作原理，以便更深入地了解托管代码是如何工作的。

.NET Framework 本身是托管代码运行的基础，分析.NET Framework 运行机制有助于找到 CLR 程序运行的关键，以便进行有针对性的排错。本章首先总结 CLR 技术上的关键部分，然后通过 windbg 分析.NET Framework 的关键函数，最后介绍 CLR 上的调试技巧。

3.1 CLR 如何工作

进入正题以前，首先小结一下 CLR 如何工作的。这里假设读者已经对 CLR 有大致的了解，比如 Assembly, Application Domain, Jit Engine 等等。最好的参考读物是 Essential .NET Volume 1 - The Common Language Runtime

托管 Assembly 本身只包含的 CLR 可识别的原数据，不包含机器指令。使用 depends 工具可以观察到所有的托管 Assembly 都跟 mscoree.dll 文件绑定。Mscoree.dll 文件在 system32 目录下，虽然 MSCOREE 可以翻译为 Microsoft Core Execution Engine，但千万不要被牛逼的名字误导。Mscoree 的功能是选择合适的 CLR Execution Engine 加载。

多个版本的 CLR 可以共存。CLR 的目录在 C:\WINDOWS\Microsoft.NET\Framework。当前系统中最新版本的 CLR 对应的 mscoree.dll 文件被拷贝到 system32 目录下。当 mscoree.dll 加载后，它根据托管代码的元数据,配置信息(app.config)文件，选择恰当版本的引擎加载。同时 mscoree 还负责判断应该使用何种 GC Flavor。GC Flavor 包括 Workstation GC 和 Server GC。在 CLR1 中，Workstation GC 对应到 mscorwks.dll 文件，Server GC 对应到 mscorsvr.dll 文件。在 CLR2 中虽然保留了 mscorsvr.dll 文件，但是 mscorwks.dll 已经包含了两种 GC Flavor 的实现，只需要加载 mscorwks 就可以了。关于 GC Flavor 的进一步讨论可以参考：

Using GC Efficiently – Part 2
http://blogs.msdn.com/maoni/archive/2004/09/25/234273.aspx

当 CLR 引擎加载后，CLR 首先初始化引擎需要的各种功能，比如必要的全局变量，引擎需要的模块(比如 ClassLoader，Assembly Loader，JitEngine, Context 等)，启动 Finalizer thread，GC Thread (ServerGC Favor)，创建 System AppDomain, Shared AppDomain，创建 RCDebugger Thread，加载 CLR 基础类，比如 mscorlib.dll 和 system.dll。

当 CLR 引擎初始化完成后，CLR 会找到当前 EXE 的元数据，然后找到 Main 函数，编译 Main 函数，执行 Main 函数。

动态编译

假设 C#函数 foo1 要调用 foo2。当 CLR 编译 foo1 的时候，无论 foo2 是否已经编译成机器代码，call 指令都是把执行指向到跟 foo2 相关的一个内存地址(stub)。当执行到这个 call 指令的时候，如果 foo2 没有被 CLR 编译，stub 中的代码会把执行定向到 CLR JitEngine，这样对 foo2 的调用便导致了 CLR JitEngine 的启动来编译 foo2 函数。JitEngine 编译完成后，CLR 把编译好的机器代码拷贝到进程中的由 CLR 管理的某一块内存上(loader heap)上，然后 JitEngine 把编译好的 foo2 函数入口地址回填到 stub 中。通过这样的技术，当第二次对 foo2 的调用发生的时候, foo2 stub 指向的已经是编译好的地址了，于是不需要再次编译。当然，第一次编译完成后，JitEngine 同时要负责执行刚刚编译好的函数。

虽然对于每个方法，只需要编译一次,但是对于大型程序，编译带来的开销还是客观的。所以 CLR 支持 NGen。也就是预先编译。CLR 的库函数默认已经 NGen 好了。详细信息可以参考：
The Performance Benefits of NGen.
http://msdn.microsoft.com/msdnmag/issues/06/05/CLRInsideOut/

在 CLR 的执行过程中，如果使用到的代码都是托管代码，编译和执行就按照上面的逻辑进行。但是不可避免地，托管代码需要调用非托管代码。这里分两种情况。第一种是调用系统 API。比如 CLR 中使用 FileStream 打开一个文件，最终还是需要调用到 CreateFileW 的。通过 DLLImport 调用自定义的非托管函数，和 COM Interop 也属于这种情况。第二种是调用 CLR 的功能，比如内存分配，异常派发。

这两种情况也都使用 stub 技术。对于第一种情况，不管是 PInvoke 还是 COM Interop 发生的时候，托管代码调用的都是 CLR 创建的 stub。在这个 stub 中 CLR 会做一些必要的工作，然后把控制权交给对应的非托管代码。必要的工作包括把必要的函数参数拷贝到非托管的内存上，marshal 必要的类型，锁住需要跟非托管代码交互的托管内存区域防止 GC 移动这块内存，如果是 COM Interop 还包括对非托管接口指针进行必要的 QueryInterface 等等。当非托管调用完成后，执行权返回到 stub，再次进行必要的工作后，回到托管代码。

第二种情况对 CLR 功能的调用往往是隐式发生的。

一类是编译器直接生成对 CLR stub 的调用。比如 new/throw 关键字。动态编译引擎对这些关键字的处理是生成函数调用到 stub，stub 把执行定位到 CLR 引擎中的关键函数。拿分配内存来说，比如 new 一个 StringBuilder object。动态编译生成的指令把执行权定向到特殊的 stub,该 stub 包含了指令来调用 CLR 中的内存分配函数，同时传入类型信息。

另一类是通过把托管代码标示为 internalcall 来编译。Internal call 表示该托管函数其实是某 unmanaged 函数的映射，编译引擎在编译 internal call 的时候，也是直接生成函数调用到 stub。比如 Assembly.Load 方法最后会通过标示为 internal call 的 nLoad 方法调入 CLR：
[MethodImpl(MethodImplOptions.InternalCall)]

private static extern Assembly nLoad(AssemblyName fileName, string codeBase, Evidence assemblySecurity, Assembly locationHint, ref StackCrawlMark stackMark, bool throwOnFileNotFound, bool forIntrospection);

内存分配

CLR 引擎初始化的时候会向操作系统申请连续内存作为 managed heap。所有的 managed object 都是分配在 managed heap 中。对于任何一种托管类型，由于类型信息保存在元数据中，所以 CLR 清楚如何生成正确的内存格式。当托管类型分配请求定向到 CLR 中后, CLR 首先检查 managed heap 是否足够。如果足够，CLR 直接开辟出内存，然后根据类型信息填入必要的格式数据，接下来执行权返回到托管代码中。如果 managed heap 不够，CLR 执行 GC 试图清扫出一部分内存来使用。如果 GC 无法清扫出内存，CLR 向 OS 请求更多的内存作为 managed heap。如果 OS 拒绝内存请求，OutOfMemory 就发生了。CLR 内存分配的特点是：

1. 大多数情况下比非托管代码内存分配速度快。CLR 保持内部指针指向当前 managed heap 中的 free point。只要 free memory 足够，CLR 直接把当前指针所在地址作为内存分配出去，然后把指针加/减分配出去的内存的长度。对于非托管代码的内存分配，不管是 Heap Manager 还是 Virtual Memory allocation，都需要做相对复杂的计算才能找到合适的内存块进行分配
2. 由于托管 object 受到 CLR 的管理，GC 发生的时候 CLR 可以对托管 object 进行随意移动，然后修正保存 object 的 stub 信息保证托管代码不会为此受到影响。移动 object 可以防止内存碎片产生，提高内存使用效率。对于非托管代码来说，由于程序中可以直接使用指针，所以无法进行内存碎片整理
3. GC 可以在任何时候被触发，但是 GC 不能在任何时候发生。比如某一个线程正在做 regex 的匹配，访问到大量的托管 object，很多 object 的地址保存到了 CPU 寄存器上进行优化。如果 GC 发生导致 object 地址变化，恢复运行后 CPU 寄存器上的指针可能就会无效。所以 GC 必须在所有线程的执行状态都不会受到 GC 影响的时候发生。当线程的执行状态不会受到 GC 影响的时候，该线程的 PreEmptive GC 属性是 1，否则是 0。这个开关受到 CLR 的控制，很多 stub 中的代码会操作这个开关。比如托管代码调用了 MessageBox.Show，该方法最后会调用到 MessageBox API。在 stub 调用 API 从托管代码变化到非托管代码前，stub 会通过 CLR 内部方法把 PreEmptive 设定为 1，表示 GC 可以安全发生了。大致情况是，当线程 idle 的时候（线程 idle 的时候肯定实在等某一个系统 API，比如 Sleep 或者 WaitForSingleObject），PreEmptive 为 1.当线程在托管代码中干活的时候，PreEmptive 为 0. 当 GC 触发的时候，GC 必须等到所有的线程都进入 PreEmptive 模式后，才可以发生(察觉到了吗，一个死锁的危险地！)。

此外,关于 GC 和 Finalizer 的详细讨论，可以从参考：
Garbage Collection: Automatic Memory Management in the Microsoft .NET Framework
http://msdn.microsoft.com/msdnmag/issues/1100/gci/

ServerGC 的一个特殊情况：

You may receive an error message, or the computer may stop responding, when you host Web applications that use ASP.NET on a computer that is running Windows Server 2003
http://support.microsoft.com/?id=911716

异常处理

异常处理在 CLR 中也非常有特色。比如，NullReferenceException 跟 Access Violation 其实是密切相关的。当编译的托管代码执行的时候，对于 NULL object 的访问，首先触发的是 Access Violation。但是聪明的 CLR 已经设定好了对应的 FS:[0]寄存器来截获可能的异常。CLR 截获异常后，首先检查异常类型，对于 Access Violation，CLR 的检查当前的代码是否是托管代码，对应的类型信息是什么。发现是 NULL object 访问后，CLR 把这个 Access Violation 异常标示为已经处理，然后 CLR 生成对应的 NullReferenceException 抛出来。当 NullReferenceException 被 CLR 设定的 FS:[0]截获后，CLR 发现异常类型是 CLR Exception，于是就找对应的 catch 语句执行。

CLR 异常发生后可以打印出 callstack,原因在于 CLR 可以通过元数据获取所有的类型信息，同时 CLR 在 thread 中通过多种机制记录运行状态。保存在 Stack 中的 Frame 就是其中一种重要的数据结构。Frame 是 CLR 保存在 stack 中的小块数据结构。当 thread 的执行状态发生改变的时候，比如从在托管代码和非托管代码切换，异常产生，remoting 调用等等的时候，CLR 会恰当地插入 Frame 来标示状态的改变。thread 中的所有 frame 是通过指针链接在一起的，所以 CLR 可以方便地获取一个 thread 的各种状态情况。

关于 CLR 类型加载，动态编译，异常处理等等细节，以及后面将会提及的一些 CLR 内部类型(MethodTable, MethodDesc, EEClass,SharedDomain 等)，下面一篇文章都有详细的介绍:

Drill Into .NET Framework Internals to See How the CLR Creates Runtime Objects
http://msdn.microsoft.com/msdnmag/issues/05/05/JITCompiler/default.aspx
这篇文章字字珠玑，熟悉整个 CLR 的运行才可以很好地理解后面介绍的 CLR 调试

如果对 CLR 更多的内部细节感兴趣，建议仔细阅读下面的 blog:
http://blogs.msdn.com/cbrumme/

3.2 详细的来龙去脉

有了上面的直观感觉后，一起用 Windbg+rotor 从源代码层次上找到 CLR 程序的来龙去脉

开源的 CLR:Rotor

Rotor 是微软提供的，开源的，稍有简化的.NET Framework 的实现。包含了.NET Framework Runtime 的全部核心功能，简化掉了一些外围的，比如 WinForm Libaray 等。Rotor 中 80%以上的源代码跟发布版的.NET Framework 一致。可以在 Windows 平台和 FreeBSD 平台编译 Rotor，编译后可以直接运行 VS 2005 生成的 EXE。

Rotor 的下载地址是:

Shared Source Common Language Infrastructure 2.0 Release
http://www.microsoft.com/downloads/details.aspx?FamilyId=8C09FD61-3F26-4555-AE17-3121B4F51D4D&displaylang=en

CLR 的核心代码，都在 rotor 的 sscli20\clr\src\vm 目录下。有条件的可以用 SourceInsight 编译 vm 目录，以便快速定位到 CLR 源代码分析。有本书叫 Shared Source CLI Essentials，分析 Rotor 1.0 的实现。国内没有引进，但是下面的分析足够可以找到分析 CLR 的窍门

庖丁解牛

凶器：Windbg, Microsoft symbol
平台：VS 2005, .NET Framework 2.0 Runtime RTM
参考资源: Rotor 源代码
小白鼠: 一个包含了下面的代码的简单 WinForm：

```
private void button2_Click(object sender, EventArgs e)
{
    System.GC.Collect();
    System.GC.Collect();
}
System.Collections.Stack stc = new System.Collections.Stack();

private void button1_Click(object sender, EventArgs e)
{

    byte[] buffer = new byte[1024 * 1024 * 50];
    stc.Push(buffer);
}
```

第一批断点

编译好程序后，启动 windbg，设定好 symbol path，选择 File -> Open Executable 打开 EXE。Windbg 停下来后，设定下面这些断点：

```
0 e 79011b2b    0001 (0001)  0:**** mscoree!_CorExeMain
 1 eu           0001 (0001) (mscorwks!_CorExeMain)
 2 eu           0001 (0001) (mscorwks!ClassLoader::RunMain)
 3 eu           0001 (0001) (mscorwks!SetupThread)
 4 eu           0001 (0001) (mscorwks!EEStartupHelper)
```

```
 5 eu           0001 (0001) (mscorwks!CoInitializeEE)
 6 eu           0001 (0001) (mscorwks!WKS::GCHeap::FinalizerThreadStart)
```

然后 g 执行。

mscoree!_CorExeMain

第一个停下来的地方是 mscoree!_CorExeMain。这是 mscoree 加载 Engine 的地方。

```
Breakpoint 0 hit
eax=00000000 ebx=7ffd9000 ecx=0012ffb0 edx=7c92eb94 esi=00000003 edi=00000018
eip=79011b2b esp=0012ffc4 ebp=0012fff0 iopl=0         nv up ei pl zr na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
mscoree!_CorExeMain:
79011b2b 55              push    ebp
0:000> kb
ChildEBP RetAddr  Args to Child
0012ffc0 7c816fd7 00000018 00000003 7ffd9000 mscoree!_CorExeMain
0012fff0 00000000 79011b2b 00000000 78746341 KERNEL32!BaseProcessStart+0x23
```

继续执行，windbg 中会看到更多的 module 加载进来。这是因为 mscoree 加载了 mscorwks, 而 mscorwks 跟其他很多系统 module 绑定在一起。最后 windbg 停留在：

```
Breakpoint 1 hit
eax=79f05eca ebx=7ffd9000 ecx=7c939aeb edx=7c99c0d8 esi=00000000 edi=00000018
eip=79f05eca esp=0012ffb4 ebp=0012ffc0 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
mscorwks!_CorExeMain:
79f05eca 6a20            push    0x20
0:000> kb
ChildEBP RetAddr  Args to Child
0012ffb0 79011b5f 00000003 79e70000 0012fff0 mscorwks!_CorExeMain
0012ffc0 7c816fd7 00000018 00000003 7ffd9000 mscoree!_CorExeMain+0x2c
0012fff0 00000000 79011b2b 00000000 78746341 KERNEL32!BaseProcessStart+0x23
```

这次是停留在 mscorwks 的_CorExeMain。可以找到 Rotor 中对应函数的实现了解更多。继续 g:

EEStartupHelper

```
0:000> g
```

```
Breakpoint 4 hit
eax=0012ff14 ebx=00000000 ecx=7a38a9dc edx=00000001 esi=7a38a9dc edi=00000000
eip=79eb7b85 esp=0012fef0 ebp=0012ff24 iopl=0         nv up ei pl zr na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
mscorwks!EEStartupHelper:
79eb7b85 55              push    ebp
0:000> kb
ChildEBP RetAddr  Args to Child
0012feec 79ebc7d8 00000002 164efd31 00000000 mscorwks!EEStartupHelper
0012ff24 79ebc78c 00000002 164efd7d 00000000 mscorwks!EEStartup+0x50
0012ff68 79f05f16 00000002 164efda5 00000018 mscorwks!EnsureEEStarted+0xfa
0012ffb0 79011b5f 00000003 79e70000 0012fff0 mscorwks!_CorExeMain+0xb8
0012ffc0 7c816fd7 00000018 00000003 7ffd9000 mscoree!_CorExeMain+0x2c
0012fff0 00000000 79011b2b 00000000 78746341 KERNEL32!BaseProcessStart+0x23
```

这次来到了初始化引擎的核心函数，EEStartupHelper。查看源代码会发现该函数特别的长，中间做了很多 init 的事情。继续 g:

```
0:000> g
Breakpoint 3 hit
eax=7815e475 ebx=00000000 ecx=0000a827 edx=7c92eb94 esi=01000000 edi=79e70000
eip=79ecabc7 esp=0012fe10 ebp=0012fe78 iopl=0         nv up ei pl zr na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
mscorwks!SetupThread:
79ecabc7 6a34            push    0x34
0:000> kb
ChildEBP RetAddr  Args to Child
0012fe0c 79eb7f44 00000000 0012fe6c 79f93fe6 mscorwks!SetupThread
0012feec 79ebc7d8 00000002 164efd31 00000000 mscorwks!EEStartupHelper+0x59a
0012ff24 79ebc78c 00000002 164efd7d 00000000 mscorwks!EEStartup+0x50
0012ff68 79f05f16 00000002 164efda5 00000018 mscorwks!EnsureEEStarted+0xfa
0012ffb0 79011b5f 00000003 79e70000 0012fff0 mscorwks!_CorExeMain+0xb8
0012ffc0 7c816fd7 00000018 00000003 7ffd9000 mscoree!_CorExeMain+0x2c
0012fff0 00000000 79011b2b 00000000 78746341 KERNEL32!BaseProcessStart+0x23
```

来到了 SetupThread 的地方。该函数是 EEStartupHelper 掉用的，可以猜到属于初始化引擎的一部分。该函数的功能是把一个 OS thread 跟一个 CLR thread 绑定在一起。除了初始化引擎的时候要绑定外，创建新的 CLR Thread 的时候该函数也会执行。

继续加载很多 module 后，最后停到了 FinalizerThreadStart 这个函数上。仔细一点，这里函数的 namespace 是 WKS::，说明当前的 GC Flavor 是 workstation 模式。如果是 Server GC,这里的 namespace 是 SVR::

注意看 windbg 的提示符，不再是 0:000>,而是 0:002>。这说明这个断点在第三个 thread 中触发的。看看另外两个 thread 的信息：

```
0:002> ~0 kb100
ChildEBP RetAddr  Args to Child
0012eff8 7c92e591 7c939079 000006fc 00000000 ntdll!KiFastSystemCallRet
0012effc 7c939079 000006fc 00000000 0012f048 ntdll!NtSetEventBoostPriority+0xc
0012f00c 7c92110a 7c99c0d8 7c933281 7c99c0d8 ntdll!RtlpUnWaitCriticalSection+0x30
0012f014 7c933281 7c99c0d8 00000000 00000000 ntdll!RtlLeaveCriticalSection+0x1d
0012f048 7c936433 00000001 05240087 7c9362f3 ntdll!LdrUnlockLoaderLock+0x88
0012f054 7c9362f3 00000000 0012f388 0012f3b0 ntdll!LdrLoadDll+0x2f4
0012f2e4 7c801bb9 001951d8 0012f330 0012f310 ntdll!LdrLoadDll+0x2ac
0012f34c 77d2e2f5 0012f3b0 00000000 00000008 KERNEL32!LoadLibraryExW+0x18e
0012f378 7c92eae3 0012f388 0000006c 0000006c USER32!__ClientLoadLibrary+0x32
0012f3f0 77d2013e 77d20104 00000000 0000c038 ntdll!KiUserCallbackDispatcher+0x13
0012f894 77d201f7 00000000 0000c038 0012f92c USER32!NtUserCreateWindowEx+0xc
0012f940 77d1ff83 00000000 0000c038 0012f92c USER32!_CreateWindowEx+0x1ed
0012f97c 769dee8c 00000000 0000c038 7699d294 USER32!CreateWindowExW+0x33
0012f9b4 769dee3b 00000000 76ab67e8 76ab683c ole32!InitMainThreadWnd+0x3c
0012f9c8 769af031 00000000 00000002 00000000 ole32!wCoInitializeEx+0x94
0012f9e8 79ebfba8 001902f0 00000002 47efd425 ole32!CoInitializeEx+0x112
0012fa30 79f0702d 00000000 00000001 00000000 mscorwks!Thread::SetApartment+0x160
0012fa44 79f0d1ae 00402588 00000000 47efd4e1 mscorwks!SystemDomain::SetThreadAptState+0x84
0012ff18 79f07dc7 00400000 00000000 47efd17d mscorwks!SystemDomain::ExecuteMainMethod+0x12c
0012ff68 79f05f61 00400000 47efd1a5 00000018 mscorwks!ExecuteEXE+0x59
0012ffb0 79011b5f 00000003 79e70000 0012fff0 mscorwks!_CorExeMain+0x11b
0012ffc0 7c816fd7 00000018 00000003 7ffd7000 mscoree!_CorExeMain+0x2c
0012fff0 00000000 79011b2b 00000000 78746341 KERNEL32!BaseProcessStart+0x23
0:002> ~1 kb100
ChildEBP RetAddr  Args to Child
00bafe38 7c92e9ab 7c8094e2 00000003 00bafe64 ntdll!KiFastSystemCallRet
00bafe3c 7c8094e2 00000003 00bafe64 00000001 ntdll!ZwWaitForMultipleObjects+0xc
00bafed8 7c80a075 00000003 00baff1c 00000000 KERNEL32!WaitForMultipleObjectsEx+0x12c
00bafef4 79ed4b06 00000003 00baff1c 00000000 KERNEL32!WaitForMultipleObjects+0x18
00baff54 79ed4a63 4747d191 7a38a9b0 00000000 mscorwks!DebuggerRCThread::MainLoop+0xcf
00baff84 79ed49a6 4747d1a1 7a38a9b0 79eb723d mscorwks!DebuggerRCThread::ThreadProc+0xca
00baffb4 7c80b683 00000000 7a38a9b0 79eb723d mscorwks!DebuggerRCThread::ThreadProcStatic+0x82
00baffec 00000000 79ed4960 00000000 00000000 KERNEL32!BaseThreadStart+0x37
```

第一个 thread 已经退出了引擎初始化，引擎开始通过 ExecuteEXE 执行程序了。目前正在通过 `CoInitializeEx` 初始化 COM。这里的 thread1 跟当前 thread2 都是 CLR 初始化时候创建的，协助 CLR 运行的线程。Thread1 跟调试功能相关，thread2,从名字上看也知道是 FinalizerThread。由于创建线程的调用不会阻塞，所以 main thread 上看不到启动 thread 的调用。(你能自己调试，找到主线程创建这两个线程的具体函数吗？)

mscorwks!SystemDomain::ExecuteMainMethod

回到 main thread，`mscorwks!ExecuteEXE` 和 `mscorwks!SystemDomain::ExecuteMainMethod` 两个函数从名字上看比较显眼。这里也能看到 SystemDomain 上已经有活动了。继续运行，会看到 callstack 停留在了 `ClassLoader::RunMain` 上

```
0:000> kb
ChildEBP RetAddr  Args to Child
0012f7e0 79f082a9 00a72ff8 00000001 0012f81c mscorwks!ClassLoader::RunMain
0012fa48 79f0817e 00000000 640ae507 00000000 mscorwks!Assembly::ExecuteMainMethod+0xa6
0012ff18 79f07dc7 00400000 00000000 640ae09b mscorwks!SystemDomain::ExecuteMainMethod+0x398
0012ff68 79f05f61 00400000 640ae043 00000018 mscorwks!ExecuteEXE+0x59
0012ffb0 79011b5f 00000003 79e70000 0012fff0 mscorwks!_CorExeMain+0x11b
0012ffc0 7c816fd7 00000018 00000003 7ffd7000 mscoree!_CorExeMain+0x2c
0012fff0 00000000 79011b2b 00000000 78746341 KERNEL32!BaseProcessStart+0x23
```

注意看 callstack，已经从 Assembly 细致到了 ClassLoader,这正是 CLR 分析类型的过程。

到这里 CLR 初始化的过程完成。已经定位到具体的托管 main 函数，下一步就是编译 main 函数然后执行。首先取消当前所有的 breakpoint，然后添加下面新的 breakpoint：

第二批断点

```
 0 e 79f08308     0001 (0001)  0:**** mscorwks!ClassLoader::RunMain
 1 e 79e7d64a     0001 (0001)  0:**** mscorwks!Module::LookupMethodDef
 2 e 79e88d1c     0001 (0001)  0:**** mscorwks!MethodDesc::CallDescr
 3 e 79e88f44     0001 (0001)  0:**** mscorwks!CallDescrWorker
 4 e 79e976b4     0001 (0001)  0:**** mscorwks!invokeCompileMethod
```

CallDescr /MakeJitWorker

然后 g：

```
0:000> g
Breakpoint 2 hit
eax=00000001 ebx=0012f6c8 ecx=00a72ff8 edx=0012f5b8 esi=00a72ff8 edi=00000000
eip=79e88d1c esp=0012f654 ebp=0012f724 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
mscorwks!MethodDesc::CallDescr:
79e88d1c 55              push    ebp
0:000> kb
ChildEBP RetAddr  Args to Child
```

```
0012f650 79e88d19 00a73058 0012f72c 0012f6e0 mscorwks!MethodDesc::CallDescr
0012f668 79e88cf6 00a73058 0012f72c 0012f6e0 mscorwks!MethodDesc::CallTargetWorker+0x20
0012f67c 79f084b0 0012f6e0 d1817c18 00000000 mscorwks!MethodDescCallSite::Call+0x18
0012f7e0 79f082a9 00a72ff8 00000001 0012f81c mscorwks!ClassLoader::RunMain+0x220
0012fa48 79f0817e 00000000 d18171c8 00000000 mscorwks!Assembly::ExecuteMainMethod+0xa6
0012ff18 79f07dc7 00400000 00000000 d1817454 mscorwks!SystemDomain::ExecuteMainMethod+0x398
0012ff68 79f05f61 00400000 d181748c 00000018 mscorwks!ExecuteEXE+0x59
0012ffb0 79011b5f 00000003 79e70000 0012fff0 mscorwks!_CorExeMain+0x11b
0012ffc0 7c816fd7 00000018 00000003 7ffde000 mscoree!_CorExeMain+0x2c
0012fff0 00000000 79011b2b 00000000 78746341 KERNEL32!BaseProcessStart+0x23
0:000> g
Breakpoint 3 hit
eax=0012f500 ebx=0012f4ac ecx=00000000 edx=0012f4ac esi=00180f70 edi=00000000
eip=79e88f44 esp=0012f494 ebp=0012f510 iopl=0         nv up ei pl zr na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
mscorwks!CallDescrWorker:
79e88f44 55              push    ebp
0:000> kb
ChildEBP RetAddr  Args to Child
0012f490 79e88ee4 0012f560 00000000 0012f530 mscorwks!CallDescrWorker
0012f510 79e88e31 0012f560 00000000 0012f530 mscorwks!CallDescrWorkerWithHandler+0xa3
0012f650 79e88d19 00a73058 0012f72c 0012f6e0 mscorwks!MethodDesc::CallDescr+0x19c
0012f668 79e88cf6 00a73058 0012f72c 0012f6e0 mscorwks!MethodDesc::CallTargetWorker+0x20
0012f67c 79f084b0 0012f6e0 d1817c18 00000000 mscorwks!MethodDescCallSite::Call+0x18
0012f7e0 79f082a9 00a72ff8 00000001 0012f81c mscorwks!ClassLoader::RunMain+0x220
0012fa48 79f0817e 00000000 d18171c8 00000000 mscorwks!Assembly::ExecuteMainMethod+0xa6
0012ff18 79f07dc7 00400000 00000000 d1817454 mscorwks!SystemDomain::ExecuteMainMethod+0x398
0012ff68 79f05f61 00400000 d181748c 00000018 mscorwks!ExecuteEXE+0x59
0012ffb0 79011b5f 00000003 79e70000 0012fff0 mscorwks!_CorExeMain+0x11b
0012ffc0 7c816fd7 00000018 00000003 7ffde000 mscoree!_CorExeMain+0x2c
0012fff0 00000000 79011b2b 00000000 78746341 KERNEL32!BaseProcessStart+0x23
```

这里看到 `mscorwks!MethodDesc::CallDescr` 和 `mscorwks!CallDescrWorker`。托管方法在 CLR 中对应的非托管结构就是 MethodDesc，这里负责托管代码的编译。继续运行：

```
0:000> kb
ChildEBP RetAddr  Args to Child
0012eeec 79e9767a 0019aa50 0012ef84 0012f010 mscorwks!invokeCompileMethod
0012ef44 79e97516 0019aa50 0012ef84 00000000 mscorwks!CallCompileMethodWithSEHWrapper+0x84
0012f2fc 79e9731c 00a72ff8 00198d80 00107214 mscorwks!UnsafeJitFunction+0x212
0012f3a0 79e7d58c 00198d80 00000000 d18178c4 mscorwks!MethodDesc::MakeJitWorker+0x1c2
0012f3f8 79e7bb73 00000000 d1817f74 0012f69c mscorwks!MethodDesc::DoPrestub+0x44d
0012f448 00391f3e 0012f478 d0104946 00000000 mscorwks!PreStubWorker+0xed
WARNING: Frame IP not in any known module. Following frames may be wrong.
```

```
0012f460 79e88f63 00000000 00180f70 0012f4ac 0x391f3e
0012f490 79e88ee4 0012f560 00000000 0012f530 mscorwks!CallDescrWorker+0x33
0012f510 79e88e31 0012f560 00000000 0012f530 mscorwks!CallDescrWorkerWithHandler+0xa3
0012f650 79e88d19 00a73058 0012f72c 0012f6e0 mscorwks!MethodDesc::CallDescr+0x19c
0012f668 79e88cf6 00a73058 0012f72c 0012f6e0 mscorwks!MethodDesc::CallTargetWorker+0x20
0012f67c 79f084b0 0012f6e0 d1817c18 00000000 mscorwks!MethodDescCallSite::Call+0x18
0012f7e0 79f082a9 00a72ff8 00000001 0012f81c mscorwks!ClassLoader::RunMain+0x220
0012fa48 79f0817e 00000000 d18171c8 00000000 mscorwks!Assembly::ExecuteMainMethod+0xa6
0012ff18 79f07dc7 00400000 00000000 d1817454 mscorwks!SystemDomain::ExecuteMainMethod+0x398
0012ff68 79f05f61 00400000 d181748c 00000018 mscorwks!ExecuteEXE+0x59
0012ffb0 79011b5f 00000003 79e70000 0012fff0 mscorwks!_CorExeMain+0x11b
0012ffc0 7c816fd7 00000018 00000003 7ffde000 mscoree!_CorExeMain+0x2c
0012fff0 00000000 79011b2b 00000000 78746341 KERNEL32!BaseProcessStart+0x23
```

上面的 callstack 有很多可以研究的地方。首先是 invokeCompileMethod。从名字上看，该函数跟动态编译相关。接下来是 MakeJitWorker. 查看源代码中对 MakeJitWorker 的注释，可以看到这是启动 compiler 的方法。接下来非常重要的信息是，”`Frame IP not in any known module`”. 这句话的上面和下面都是 CLR 的非托管代码。中间有一段函数不在任何 module 里面。可能性有两种，一是该 module 已经被 unload 了(这种情况下返回的时候 `PreStubWorker` 肯定会 crash)，二是这部分的代码是动态生成的。这里当然是第二种情况了。如果使用后面会介绍的 sos extention，会看到正在被调用的托管函数：

```
0:000> .load C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\sos
0:000> !clrstack
OS Thread Id: 0xb20 (0)
ESP      EIP
0012f478 79e976b4 [PrestubMethodFrame: 0012f478] SampleCLR.Program.Main()
0012f69c 79e976b4 [GCFrame: 0012f69c]
```

继续运行：

```
0:000> kb 100
ChildEBP RetAddr  Args to Child
0012e19c 79e7e0c7 06000007 d1816930 00000000 mscorwks!Module::LookupMethodDef
0012e20c 79e7dc47 00a72c14 06000007 0012e49c
mscorwks!MemberLoader::GetDescFromMemberDefOrRefThrowing+0x4da
0012e494 79e95ee8 00000001 06000007 0012e5cc
mscorwks!MemberLoader::GetMethodDescFromMemberDefOrRefOrSpecThrowing+0x1fb
0012e520 79e968c4 00a72c14 06000007 0012e5fc
mscorwks!MemberLoader::GetMethodDescFromMemberDefOrRefOrSpecNT+0x76
0012e60c 79e96791 00a72c14 06000007 00a72ff8 mscorwks!CEEInfo::findMethodInternal+0x12b
0012e68c 79061526 0012ef90 00a72c14 06000000 mscorwks!CEEInfo::findMethod+0x107
0012e6a0 7906f78f 06000007 00a72c14 00a72ff8 mscorjit!Compiler::eeFindMethod+0x22
```

```
0012ec68 7906f5e4 012629bc fc90b446 01262a7c mscorjit!Compiler::impImportBlockCode+0x243b
0012ecec 7906f57d 012629bc 00000000 0012f010 mscorjit!Compiler::impImportBlock+0x20c
0012ed00 7906f4b8 012629bc 01260010 7906de9b mscorjit!Compiler::impImport+0xe3
0012ed0c 7906de9b 01260010 0012ed70 7906ebee mscorjit!Compiler::fgImport+0x20
0012ed18 7906ebee 0012ee38 0012f09c 0012ee2c mscorjit!Compiler::compCompile+0xb
0012ed70 7906e8db 00a72c14 0012ef84 0012f010 mscorjit!Compiler::compCompile+0x2d8
0012ee04 7906e831 0012ef84 0012f010 0012ee38 mscorjit!jitNativeCode+0xb8
0012ee3c 79e9776f 790af170 0012ef84 0012f010 mscorjit!CILJit::compileMethod+0x3d
0012eea8 79e976e5 0019aa50 0012ef84 0012f010 mscorwks!invokeCompileMethodHelper+0x72
0012eeec 79e9767a 0019aa50 0012ef84 0012f010 mscorwks!invokeCompileMethod+0x31
0012ef44 79e97516 0019aa50 0012ef84 00000000 mscorwks!CallCompileMethodWithSEHWrapper+0x84
0012f2fc 79e9731c 00a72ff8 00198d80 00107214 mscorwks!UnsafeJitFunction+0x212
0012f3a0 79e7d58c 00198d80 00000000 d18178c4 mscorwks!MethodDesc::MakeJitWorker+0x1c2
0012f3f8 79e7bb73 00000000 d1817f74 0012f69c mscorwks!MethodDesc::DoPrestub+0x44d
0012f448 00391f3e 0012f478 d0104946 00000000 mscorwks!PreStubWorker+0xed
WARNING: Frame IP not in any known module. Following frames may be wrong.
0012f460 79e88f63 00000000 00180f70 0012f4ac 0x391f3e
0012f490 79e88ee4 0012f560 00000000 0012f530 mscorwks!CallDescrWorker+0x33
0012f510 79e88e31 0012f560 00000000 0012f530 mscorwks!CallDescrWorkerWithHandler+0xa3
0012f650 79e88d19 00a73058 0012f72c 0012f6e0 mscorwks!MethodDesc::CallDescr+0x19c
0012f668 79e88cf6 00a73058 0012f72c 0012f6e0 mscorwks!MethodDesc::CallTargetWorker+0x20
0012f67c 79f084b0 0012f6e0 d1817c18 00000000 mscorwks!MethodDescCallSite::Call+0x18
0012f7e0 79f082a9 00a72ff8 00000001 0012f81c mscorwks!ClassLoader::RunMain+0x220
0012fa48 79f0817e 00000000 d18171c8 00000000 mscorwks!Assembly::ExecuteMainMethod+0xa6
0012ff18 79f07dc7 00400000 00000000 d1817454 mscorwks!SystemDomain::ExecuteMainMethod+0x398
0012ff68 79f05f61 00400000 d181748c 00000018 mscorwks!ExecuteEXE+0x59
0012ffb0 79011b5f 00000003 79e70000 0012fff0 mscorwks!_CorExeMain+0x11b
0012ffc0 7c816fd7 00000018 00000003 7ffde000 mscoree!_CorExeMain+0x2c
0012fff0 00000000 79011b2b 00000000 78746341 KERNEL32!BaseProcessStart+0x23
```

这次的 callstack 好长，而且还是停留在 `invokeCompileMethod` 上。从这一连串的名字上看，基本上可以了解到动态 compile 是如何发生的了。有时间可以根据源代码在不同的函数上设定断点做进一步观察。到此为止，动态编译的过程也一目了然了。

NtUserWaitMessage

接下来禁止所有的 break point,让程序运行起来。然后用 ctrl+break 切入，看看程序运行起来后在干什么：

```
0:004> ~0s
eax=001c7000 ebx=013b362c ecx=0012e644 edx=00001000 esi=00000000 edi=013dd1f4
eip=7c92eb94 esp=0012f32c ebp=0012f3c4 iopl=0         nv up ei pl zr na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
```

```
ntdll!KiFastSystemCallRet:
7c92eb94 c3            ret
0:000> kb 100
ChildEBP RetAddr  Args to Child
0012f328 77d19418 7b08491c d0104946 79e71998 ntdll!KiFastSystemCallRet
*** WARNING: Unable to verify checksum for
C:\WINDOWS\assembly\NativeImages_v2.0.50727_32\System.Windows.Forms\87d0ae549013eb419066c02981332f
c4\System.Windows.Forms.ni.dll
0012f3c4 7b08432d 00000000 ffffffff 7c80a661 USER32!NtUserWaitMessage+0xc
0012f434 7b08416b 00000000 00000000 00000000
System_Windows_Forms_ni!System.Windows.Forms.Application+ThreadContext.RunMessageLoopInner(Int32,
System.Windows.Forms.ApplicationContext)+0x17d
0012f464 7b0c69fe 00000000 00000000 00000000
System_Windows_Forms_ni!System.Windows.Forms.Application+ThreadContext.RunMessageLoop(Int32,
System.Windows.Forms.ApplicationContext)+0x53
0012f490 012700a8 00000000 00000000 00000000
System_Windows_Forms_ni!System.Windows.Forms.Application.Run(System.Windows.Forms.Form)+0x2e
*** WARNING: Unable to verify checksum for SampleCLR.exe
0012f490 79e88f63 00000000 00000000 00000000 SampleCLR!SampleCLR.Program.Main()+0x38 [C:\Documents and
Settings\LiXiong\My Documents\MyCode\SampleCLR\SampleCLR\Program.cs @ 17]
0012f490 79e88ee4 0012f560 00000000 0012f530 mscorwks!CallDescrWorker+0x33
0012f510 79e88e31 0012f560 00000000 0012f530 mscorwks!CallDescrWorkerWithHandler+0xa3
0012f650 79e88d19 00a73058 0012f72c 0012f6e0 mscorwks!MethodDesc::CallDescr+0x19c
0012f668 79e88cf6 00a73058 0012f72c 0012f6e0 mscorwks!MethodDesc::CallTargetWorker+0x20
0012f67c 79f084b0 0012f6e0 d1817c18 00000000 mscorwks!MethodDescCallSite::Call+0x18
0012f7e0 79f082a9 00a72ff8 00000001 0012f81c mscorwks!ClassLoader::RunMain+0x220
0012fa48 79f0817e 00000000 d18171c8 00000000 mscorwks!Assembly::ExecuteMainMethod+0xa6
0012ff18 79f07dc7 00400000 00000000 d1817454 mscorwks!SystemDomain::ExecuteMainMethod+0x398
0012ff68 79f05f61 00400000 d181748c 00000018 mscorwks!ExecuteEXE+0x59
0012ffb0 79011b5f 00000003 79e70000 0012fff0 mscorwks!_CorExeMain+0x11b
0012ffc0 7c816fd7 00000018 00000003 7ffde000 mscoree!_CorExeMain+0x2c
0012fff0 00000000 79011b2b 00000000 78746341 KERNEL32!BaseProcessStart+0x23
```

当前 thread 正如预料中的一样，idle 在 `NtUserWaitMessage` 方法上。还看到 main 函数调用 applicatoin.run，然后调了 System.Windows.Form 里面的库函数。这里看到一个特别的名字 `System_Windows_Forms_ni` 。System.Windows.Form 的库函数在 System.Windows.Forms.DLL 中，按道理说我们应该看到 `System_Windows_Forms` 。这里最后的 ni 表示这里加载的是 NGen 后的函数，这是 CLR 2.0 中的改进，1.1 中 NGne 后的 assembly 不会带 ni。

注意，这里 k 命令成功地显示出了 `SampleCLR!SampleCLR.Program.Main()` ,而不再是 `Frame IP not in any known module` 。那现在 Main 的代码已经有 module 对应了吗？让我们看看：

由于 `System_Windows_Forms_ni!System.Windows.Forms.Application.Run` 的返回地址是 `012700a8` ，看看这个地址上有什么：

```
0:000> !u 012700a8
Normal JIT generated code
SampleCLR.Program.Main()
Begin 01270070, size 3c
```

Sos 的 u 命令显示这个地址的确是 main 的地址。那对应的 module 是 `SampleCLR.exe` 吗？

```
0:000> lmvm SampleCLR
start    end        module name
00400000 00408000   SampleCLR C (private pdb symbols)  C:\Documents and Settings\LiXiong\My
Documents\MyCode\SampleCLR\SampleCLR\bin\Debug\SampleCLR.pdb
```

不是的，SampleCLR 加载到 0x400000 地址上，距离 012700a8 远呢。其实，这里也是 CLR 2.0 的一个新功能，哪怕代码不在 module 上，也会显示出正确的托管函数名字。在 CLR1 上，这种情况下只能显示为 `Frame IP not in any known module`。

第三批断点

接下来，设定两个跟内存相关的断点:

```
0 e 79f8dde8     0001 (0001)  0:**** mscorwks!WKS::GCHeap::GarbageCollect
 1 d 79e75035     0001 (0001)  0:**** mscorwks!WKS::gc_heap::allocate_more_space
 2 e 79f134b7     0001 (0001)  0:**** mscorwks!WKS::GCHeap::GarbageCollectGeneration
```

G 后随意操作一下窗口，windbg 就会断下来。用 k 命令可以看到一个很长的 callstack:

gc_heap::allocate_more_space/ GCHeap::GarbageCollect

```
0:000> k 100
ChildEBP RetAddr
0012eb6c 7a0c709f mscorwks!WKS::gc_heap::allocate_more_space
0012eb8c 7a0900b7 mscorwks!WKS::GCHeap::Alloc+0x6c
0012eba0 79e752ad mscorwks!Alloc+0x72
0012ebe0 79e7536f mscorwks!FastAllocateObject+0x38
0012ec84 7b084f01 mscorwks!JIT_NewFast+0x9e
0012ecd4 7b08335c
System_Windows_Forms_ni!System.Windows.Forms.Internal.WindowsRegion.FromHregion(IntPtr,
Boolean)+0x11
0012ecd4 7b06e2da
System_Windows_Forms_ni!System.Windows.Forms.Internal.WindowsGraphics.FromGraphics(System.Drawing.
Graphics, System.Windows.Forms.Internal.ApplyGraphicsProperties)+0x110
0012ed1c 7b06e4a7
System_Windows_Forms_ni!System.Windows.Forms.Control.PaintBackColor(System.Windows.Forms.PaintEven
```

```
tArgs, System.Drawing.Rectangle, System.Drawing.Color)+0x66
013de938 7b06e51f
System_Windows_Forms_ni!System.Windows.Forms.Control.PaintBackground(System.Windows.Forms.PaintEve
ntArgs, System.Drawing.Rectangle, System.Drawing.Color, System.Drawing.Point)+0x97
0012ee50 7b087bc1
System_Windows_Forms_ni!System.Windows.Forms.Control.PaintBackground(System.Windows.Forms.PaintEve
ntArgs, System.Drawing.Rectangle)+0x63
0012ee50 7b085e1e
System_Windows_Forms_ni!System.Windows.Forms.ButtonInternal.ButtonBaseAdapter.PaintButtonBackgroun
d(System.Windows.Forms.PaintEventArgs, System.Drawing.Rectangle, System.Drawing.Brush)+0x29
0012ee50 7b085cbc
System_Windows_Forms_ni!System.Windows.Forms.ButtonInternal.ButtonStandardAdapter.PaintWorker(Syst
em.Windows.Forms.PaintEventArgs, Boolean, System.Windows.Forms.CheckState)+0x14e
0012eea8 7b085c69
System_Windows_Forms_ni!System.Windows.Forms.ButtonInternal.ButtonStandardAdapter.PaintUp(System.W
indows.Forms.PaintEventArgs, System.Windows.Forms.CheckState)+0xc
0012eea8 7b085b85
System_Windows_Forms_ni!System.Windows.Forms.ButtonInternal.ButtonBaseAdapter.Paint(System.Windows
.Forms.PaintEventArgs)+0x35
0012eea8 7b06e19f
System_Windows_Forms_ni!System.Windows.Forms.ButtonBase.OnPaint(System.Windows.Forms.PaintEventArg
s)+0x65
0012eedc 7b06c24f
System_Windows_Forms_ni!System.Windows.Forms.Control.PaintWithErrorHandling(System.Windows.Forms.P
aintEventArgs, Int16, Boolean)+0x5b
0012f040 7b072a17
System_Windows_Forms_ni!System.Windows.Forms.Control.WmPaint(System.Windows.Forms.Message
ByRef)+0x20b
0012f0a4 7b0815d7
System_Windows_Forms_ni!System.Windows.Forms.Control.WndProc(System.Windows.Forms.Message
ByRef)+0x2e7
0012f0e0 7b0814c3
System_Windows_Forms_ni!System.Windows.Forms.ButtonBase.WndProc(System.Windows.Forms.Message
ByRef)+0xff
0012f140 7b07a72d
System_Windows_Forms_ni!System.Windows.Forms.Button.WndProc(System.Windows.Forms.Message
ByRef)+0x2b
0012f140 7b07a706
System_Windows_Forms_ni!System.Windows.Forms.Control+ControlNativeWindow.OnMessage(System.Windows.
Forms.Message ByRef)+0xd
0012f140 7b07a515
System_Windows_Forms_ni!System.Windows.Forms.Control+ControlNativeWindow.WndProc(System.Windows.Fo
rms.Message ByRef)+0xd6
0012f140 003921bb System_Windows_Forms_ni!System.Windows.Forms.NativeWindow.Callback(IntPtr, Int32,
```

```
IntPtr, IntPtr)+0x75
WARNING: Frame IP not in any known module. Following frames may be wrong.
0012f164 77d18734  0x3921bb
0012f190 77d18816 USER32!InternalCallWinProc+0x28
0012f1f8 77d1b4c0 USER32!UserCallWinProcCheckWow+0x150
0012f24c 77d1b50c USER32!DispatchClientMessage+0xa3
0012f274 7c92eae3 USER32!__fnDWORD+0x24
0012f298 77d194d2 ntdll!KiUserCallbackDispatcher+0x13
0012f2e0 77d18a10 USER32!NtUserDispatchMessage+0xc
0012f2f0 0133187e USER32!DispatchMessageW+0xf
0012f30c 7b084766 CLRStub[StubLinkStub]@133187e
0012f3c4 7b08432d
System_Windows_Forms_ni!System.Windows.Forms.Application+ComponentManager.System.Windows.Forms.Uns
afeNativeMethods.IMsoComponentManager.FPushMessageLoop(Int32, Int32, Int32)+0x2ea
0012f434 7b08416b
System_Windows_Forms_ni!System.Windows.Forms.Application+ThreadContext.RunMessageLoopInner(Int32,
System.Windows.Forms.ApplicationContext)+0x17d
0012f464 7b0c69fe
System_Windows_Forms_ni!System.Windows.Forms.Application+ThreadContext.RunMessageLoop(Int32,
System.Windows.Forms.ApplicationContext)+0x53
0012f490 012700a8
System_Windows_Forms_ni!System.Windows.Forms.Application.Run(System.Windows.Forms.Form)+0x2e
0012f490 79e88f63 SampleCLR!SampleCLR.Program.Main()+0x38 [C:\Documents and Settings\LiXiong\My
Documents\MyCode\SampleCLR\SampleCLR\Program.cs @ 17]
0012f490 79e88ee4 mscorwks!CallDescrWorker+0x33
0012f510 79e88e31 mscorwks!CallDescrWorkerWithHandler+0xa3
0012f650 79e88d19 mscorwks!MethodDesc::CallDescr+0x19c
0012f668 79e88cf6 mscorwks!MethodDesc::CallTargetWorker+0x20
0012f67c 79f084b0 mscorwks!MethodDescCallSite::Call+0x18
0012f7e0 79f082a9 mscorwks!ClassLoader::RunMain+0x220
0012fa48 79f0817e mscorwks!Assembly::ExecuteMainMethod+0xa6
0012ff18 79f07dc7 mscorwks!SystemDomain::ExecuteMainMethod+0x398
0012ff68 79f05f61 mscorwks!ExecuteEXE+0x59
0012ffb0 79011b5f mscorwks!_CorExeMain+0x11b
0012ffc0 7c816fd7 mscoree!_CorExeMain+0x2c
0012fff0 00000000 KERNEL32!BaseProcessStart+0x23
```

最上方的是 `allocate_more_space` 和 `Alloc` 这是 CLR 进行内存分配的入口。同时还可以看到，程序的执行从最下面的 CLR 进入了托管代码，然后进入非托管系统 API 进行消息派发，最后再次进入托管代码运行消息处理函数。

去掉 allocte_more_space 断点后继续 g,这时点 button3 使劲分配内存，或者直接点 button2 激活 GC，都可以看到 windbg 停下来：

```
0:000> kb 100
```

```
ChildEBP RetAddr  Args to Child
0012ee04 79f134b2 00000000 00000000 03200010 mscorwks!WKS::GCHeap::GarbageCollectGeneration
0012ee8c 79e7504d 0012eec8 03200010 00000003 mscorwks!WKS::gc_heap::try_allocate_more_space+0x1a1
0012eea8 79ea1f03 0012eec8 03200010 00000003 mscorwks!WKS::gc_heap::allocate_more_space+0x18
0012eee4 7a0c7111 0320000c 00000000 00000000 mscorwks!WKS::gc_heap::allocate_large_object+0x88
0012ef00 7a0900b7 0320000c 00000000 0320000c mscorwks!WKS::GCHeap::Alloc+0xc6
0012ef14 79e7e940 0320000c 00000000 00000000 mscorwks!Alloc+0x72
0012ef68 79e7e9a0 79124418 03200000 00000000 mscorwks!FastAllocatePrimitiveArray+0xbd
0012f028 01270543 013b22d4 013c0f94 013c0f94 mscorwks!JIT_NewArr1+0x148
0012f0c0 7b060a6b 00000000 00000000 00000000 SampleCLR!SampleCLR.Form1.button3_Click(System.Object,
System.EventArgs)+0x2b [C:\Documents and Settings\LiXiong\My
Documents\MyCode\SampleCLR\SampleCLR\Form1.cs @ 34]
0012f0c0              7b105379              00000000              00000000              00000000
System_Windows_Forms_ni!System.Windows.Forms.Control.OnClick(System.EventArgs)+0x57
```

这里的 `allocate_more_space` 检查到当前的 managed heap 剩余空间已经无法满足内存分配需求，于是触发 GC 进行垃圾收集。GarbageCollectionGeneration 就是进行垃圾收集的入口函数。如果我们通过代码触发 GC，GarbageCollectionGeneration 会通过下面的 callstack 被调到：

```
0:000> kb
ChildEBP RetAddr  Args to Child
0012ef84 79f8dd9a ffffffff 00000000 d1817b14 mscorwks!WKS::GCHeap::GarbageCollect
*** WARNING: Unable to verify checksum for
C:\WINDOWS\assembly\NativeImages_v2.0.50727_32\mscorlib\1f7c2c9950edea489eecf6289967bb50\mscorlib.
ni.dll
0012f028 793c4714 0127057e 013b22d4 013c0e48 mscorwks!GCInterface::CollectGeneration+0xa4
0012f0c0 0127057e 00000000 00000000 00000000 mscorlib_ni!System.GC.Collect()+0x8
0012f0c0 7b060a6b 00000000 00000000 00000000 SampleCLR!SampleCLR.Form1.button2_Click(System.Object,
System.EventArgs)+0x1e [C:\Documents and Settings\LiXiong\My
Documents\MyCode\SampleCLR\SampleCLR\Form1.cs @ 20]
```

小结

通过上面的分析，找到了 CLR 运行的关键函数，结合 rotor 分析源代码便可以深入了解 CLR 的细节。CLR 其他重要部分包括：

1. AppDomain 的创建和跨 AppDomain 的方法调用
2. Reflection 方法的实现
3. Security 的实现
4. Assembly 的寻找和加载
5. ThreadPool 的实现和管理
6. 托管 object 的类型信息是如何绑定到 object 上的

7. CLR 中 lock 的实现 (C#中 lock 关键字)
8. 异常的派发和处理
9. stackwalk 和 frame 的作用
10. PInvoke 和 COM Interop

除了 Server Flavor 的 GC Thread, Finalizer thread 和 DebuggerRCThread 外，还会有其他一些辅助 CLR 运行的 thread.详细信息请参考：

Special threads in CLR
http://blogs.msdn.com/yunjin/archive/2005/07/05/435726.aspx

Things to ignore when debugging an ASP.NET hang
http://blogs.msdn.com/tess/archive/2005/12/20/505862.aspx

在阅读源代码的时候，可以考虑下面一些窍门：

1. 重要的文件往往长度比较长
2. 善于在调试器中设断点进行分析。当断下来的时候查看函数之间调用顺序
3. 注意代码中的注释
4. ecall.cpp 包含了 internal call 到对应 CLR 函数的映射关系。比如前面提到的 Assembly.nLoad，通过下面的语句定向到 CLR 的 AssemblyNative::Load 方法

   FCFuncElement("nLoad", AssemblyNative::Load)
5. 编译 rotor，使用 rotor 来运行托管代码，直接进行 rotor 的源代码调试
6. 使用 rotor 运行托管代码的时候可以打开 rotor 的 log，记录更多信息
7. 在 rotor 中加入自己的代码进行测试

3.3 CLR 调试

概览

CLR 程序包含的是元数据，通过动态编译才生成机器代码。传统 C 程序编译直接生成机器代码。所以托管程序一次编译适用于多种平台，而传统 C 程序只能用于单一平台。随着 CLR 编译引擎的优化，所有的 CLR 程序都会受益。而传统的 C 编译器优化后，必须重新编译 C 代码才可以获利。由于原数据包含描述性的类型信息，所以 CLR 程序可以获取类型信息，函数名，支持 reflection。而传统 C 程序是无法在运行时获取类型信息的。CLR 的托管内存处理，可以方便地获取内存的统计信息，还能细致到占用内存的各个 object，而传统程序中内存的操作是不带类型信息的。AppDomain 是 CLR 的执行和访问边界，传统程序的边界是 Process。创建 Process 的开销远大于 Appdomain。同一个 Process 可以容纳多个 AppDomain，所以使用 CLR 可以很好地节省系统资源。ASP.NET 就是一个例子，不同的 Virtual Directory 对应一个 AppDomain，可以一起运行在一个进程中而互不干扰。

从调试上说，下面这些都是 CLR 中可以做，而传统 C++程序无法做到的

1. 可以用 reflector 反编译 CLR Assembly 获取源代码，得到清晰的程序逻辑
2. 发生异常后 CLR 程序可以打印出 callstack
3. 托管程序的 Symbol 不再重要
4. 获取一个 class 的详细定义和所有方法
5. 列出内存中所有的 CLR object
6. 对于任何 CLR object，可以找到该 object 的类型信息

总结下来，CLR 的内存管理机制提供了 CLR 内存使用的一切信息，原数据和动态编译使得所有的 CLR 数据都是带类型信息的。进行 CLR 调试的时候，除了像调试非托管代码一样分析 callstack 和其它相关数据外，还可以：

1. 查看所有 AppDomain 的信息
2. 查看内存使用的统计信息
3. 反编译可疑的 Assembly 成 C#或者 VB.NET 代码检查
4. 打印出值得怀疑的 object 的详细信息，包括 object 成员的信息和类型信息

CLR 的调试分为两种。一种是 VS IDE 使用的，借助目标进程的 DebuggerRCThread 进行调试。通过调试器跟 DebuggerRCThread 的通信，可以方便地设定断点，查看程序变量。关于这一点的详细讨论，可以参考：

Implications of using a helper thread for debugging
http://blogs.msdn.com/jmstall/archive/2004/10/13/241828.aspx

通过 DebuggerRCThread 调试的缺陷是无法利用 CLR 内存管理的优势来获取内存方面的信息。所以这里介绍通过 sos 调试器扩展命令在 windbg 中调试。

SOS

在 CLR 的安装目录下可以找到 Sos.dll。这是随 CLR 一起发布的 windbg debugger extention。Sos.dll 中的调试命令通过读取目标进程中 CLR 内部数据结构进行调试和分析。由于不同版本的 CLR 内部实现可能有区别，所以不同版本的 CLR 需要使用对应版本的 sos。如果分析 CLR1 的程序，可以使用 windbg 安装目录下一个 clr10 子目录下的 sos.dll，功能比较齐全。对于 2.0, 可以使用.NET Framework 2.0 安装目录下的 sos 文件。

命令介绍

SOS 的命令介绍在 ：

SOS Debugging Extension (SOS.dll)
http://msdn2.microsoft.com/en-us/library/bb190764(vs.80).aspx

使用 SOS 调试 CLR 程序的 MSDN 系列教程在:
Production Debugging for .NET Framework Applications
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/DBGrm.asp

下面是一些命令简单的说明。详细的信息，请参考上面的文档和!help 命令给出的帮助。

!help
打印 sos 中的命令列表。对于任何一个命令，可以使用!help 加上命令来获取详细帮助和例子，比如”!help dumpheap”.下面是一些常用命令的介绍：

!dumpdomain
给出当前进程中所有的 appDomain 信息。SystemDomain 和 ShareDomain 是系统创建的。

!threads
列出当前进程中跟 CLR 绑定运行托管代码的 thread。该命令的详细说明可以参考：

Thread, System.Threading.Thread, and !Threads (III)
http://blogs.msdn.com/yunjin/archive/2005/08/30/457756.aspx

!clrstack
列出当前 thread 的托管 callstack。如果要列举所有 thread 的 callstack，可以用
~* e !clrstack

在某些时候，特别是发生运行状态变化，比如从托管执行状态跳入非托管执行状态的时候，!clrstack 无法显示出正确的的 callstack。原因是!clrstack 依赖于 stack 中的 frame 来进行 stackwalk。这个时候由于状态改变，对应的 frame 可能还没有建立完善，所以看不到正确的结果。这种情况下可以通过!dumpstack –ee 命令来列出当前 stack 中所有的托管方法地址，也能看到 callstack。如果不加-ee 参数，无论是托管的和非托管的函数地址，只要在 stack 中出现过，都会被 dumpstack 命令列出来

!dumpstackobjects
打印出当前 thread 的 stack 中保存的所有的托管 object。包含了 object 的地址和类型。

!dumpobj (缩写为!do)
打印出指定 object 的详细信息。比如:

```
0:000> !do 013b22d4
Name: SampleCLR.Form1
MethodTable: 00a7598c
EEClass: 00a714d8
Size: 340(0x154) bytes
 (C:\Documents and Settings\LiXiong\My Documents\MyCode\SampleCLR\SampleCLR\bin\Debug\SampleCLR.exe)
```

```
Fields:…
```

Name 表示该 object 类型的名字
MethodTable 表示该类型的函数表地址
EEClass 表示该类型的类型信息地址

!dumpclass + 类型信息地址
是打印出指定类型信息地址上的类型定义

!dumpmt
跟函数表地址作参数，打印出函数表信息。加上-md 参数可以打印出函数表中所有函数的函数信息：

```
0:000> !dumpmt -md 00a7598c
EEClass: 00a714d8
Module: 00a72c14
Name: SampleCLR.Form1
mdToken: 02000004  (C:\Documents and Settings\LiXiong\My
Documents\MyCode\SampleCLR\SampleCLR\bin\Debug\SampleCLR.exe)
BaseSize: 0x154
ComponentSize: 0x0
Number of IFaces in IFaceMap: 15
Slots in VTable: 377
--------------------------------------
MethodDesc Table
   Entry MethodDesc     JIT Name
7b05c298   7b4a5518   PreJIT System.Windows.Forms.Form.ToString()
793539c0   7913bd50   PreJIT System.Object.Equals(System.Object)
。。。
00a761a0   00a75928     JIT SampleCLR.Form1.InitializeComponent()
00a7607c   00a75930     JIT SampleCLR.Form1..ctor()
00a76a5c   00a75938    NONE SampleCLR.Form1.button2_Click(System.Object, System.EventArgs)
00a76a48   00a75940     JIT SampleCLR.Form1.button1_Click(System.Object, System.EventArgs)
00a76a70   00a75948    NONE SampleCLR.Form1.button3_Click(System.Object, System.EventArgs)
```

在上面的输出中，第一列是函数入口地址，第二列显示该函数 CLR 内部数据结构所在地址，第三列显示该函数状态，最后显示函数名字。

这里第三列有三种不同状态，PreJIT, JIT 和 NONE。PreJit 表示该函数是 NGen 过的，JIT 表示该函数已经被动态编译过了。对于这两种类型，第一列的地址上的代码就是编译后的代码。如果类型是 NONE，表示该函数还没有被编译，指向的其实是一个 stub。看看 stub 是什么：

```
0:000> u 00a76a70
SampleCLR.Form1.button3_Click(System.Object, System.EventArgs):
00a76a70 b84859a700      mov    eax,0xa75948
00a76a75 90              nop
00a76a76 e875d43f79      call   mscorwks!PrecodeRemotingThunk (79e73ef0)
```

```
00a76a7b e98cb491ff      jmp    CLRStub[ThePreStub]@391f0c (00391f0c)
```

这里能够显示出正确的函数名，但是汇编代码却指向了 mscorwks 中的函数。这就是前面介绍的 CLR 动态编译所依赖的 stub。

如果要对托管代码设定断点，如果代码已经 jit 好了，可以直接用 bp 命令。对于还没有 jit 的代码，设定断点需要额外的操作。在 CLR2 上，可以直接使用 sos 中的 BPMD 命令。对于 CLR1，请参考 SDK 的帮助，默认位置在：

C:\Program Files\Microsoft Visual Studio .NET 2003\SDK\v1.1\Tool Developers Guide\Samples\sos

!eeheap
列举出当前进程中 managed heap 的统计信息。最后一行的 `GC Heap Size` 信息表示当前所有的 managed object 加载一起占用了多少内存。这是衡量 managed heap 使用的重要标志。如果这个值持续增加，往往说明有内存泄露方面的问题。

!dumpheap
搜索并且打印出 managed heap 中的所有 object 的地址，最后再给出统计信息。如果只想要统计信息，可以使用!dumpheap –stat 命令。统计信息按照每一类 object 占用内存从小到大排序，同时给出每一类 object 的数量。检查统计信息的时候应该注意：

1. 内存是否是某一中类型的 object 占用的
2. 某一种类型的 object 是不是特别多
3. 几种不同类型的 object 数目上是否有倍数关系。如果有，说明这些类型之间往往有引用关系

!dumpheap 还可以跟-mt 参数，以便只显示特定类型 object 的信息。

!gcroot
对于任何一个托管 object，是否能够被 GC 收集，取决于该 object 是否有 root。!gcroot 命令可以搜索指定 object 的 root。如果没有 root，说明这个 object 可以被回收了。如果!gcroot 命令找到了 root object 后，会把整个引用链打印出来。一个 object 要成为 root，可能是下面情况之一:

1. Class 的 static 成员。Static 成员的生命周期跟整个 appDomain 一样长，CLR 会把所有的，用到过的 static 成员保存在一个 object array 中，然后用 pinned handle 把这个 array 保护起来。
2. 维持程序运行的必要成员。比如 AppDomain 本身。这类 object 会被 CLR 直接用 strong handle 保护起来
3. stack 上的成员。说明该 object 还在被某一个 thread 使用。
4. 被托管代码直接 pin 起来的 object.

!gchandles
打印出当前所有的 GC handle 统计信息。

!objsize
打印出指定 object，以及该 object 循环引用的所有下级 object 的总共大小。

! FinalizeQueue
打印出 FinalizeQueue 里面等待被 Finalize 的 object。

对于大多数实现 Finalizer 的类型来说，如果 Dispose/Close 方法得到调用，这类 object 就不会进入 FinalizerQueue,SqlConnection 就是这样的例子。所以如果发现 SqlConnection 在 FinalizerQueue 中，说明该 SqlConnection 使用完后没有及时 close。

如果 FinalizerQueue 中的 object 特别多，注意切换到 Finalizer thread 检查 callstack,看看是否有堵塞发生。

! SyncBlk
打印出托管代码中的 lock(C#种的 lock 关键字)。详情见帮助。对于 RWLock 或者是信号量等等其它 lock，该命令不起作用。需要通过检查 stack 上的 RWLock 或者信号量 object 来获取信息。

!soe
方法可以在指定类型的 CLR 异常发生时候断下来执行自定义命令。前面调试 IndexOutOfRangeException 的时候就用了这个命令。

!dumplog
这个命令的帮助中可以看到如何启动 CLR StressLog。设定好对应注册表后，CLR 执行的时候会尽可能多地进行内部数据结构的检查，以便让潜在的错误尽早暴露出来，同时把对应 log 保存在特殊的内存区域。!dumplog 就是分析 log 的命令。

!savemodule
最后一个，也是特别有用的命令是!savemodule。该命令能够把 dump 中的指定 module，无论是托管的还是非托管的保存到本地文件！通过这个命令，可以剥离出 dump 的某些 DLL 到文件，然后通过 Reflecto 反编译分析源代码。也就是说，如果抓到了 dump，相当于就是抓到了托管模块的源代码。

3.4 小白鼠

熟悉上面的内容后,一起用前面的 WinForm 程序来演示基本命令的使用。

CLR1 的 sos 文件适用于 CLR1 上的各个版本，包括 CLR 1.0, CLR 1.1 和 CLR 1.1 SP1，不需要其它依赖。

CLR2 的 sos 依赖于 mscordacwks.dll 文件来分析不同的 CLR2 的版本。每次打了 CLR Framework 2.0 的补丁，mscordacwks.dll 文件都会随着更新。所以在抓取 dump 文件的时候，

最好在收集 dump 的时候同时收集目标机器上的 mscordacwks.dll 文件。在调试器中加载了 CLR2 的 sos 后，可以运行.cordll 命令看 mscordacwks.dll 是否匹配。

.load sos

启动程序，加载 windbg,使用 ctrl+break 切入 windbg 后，运行下面的命令加载 sos：

```
.load C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\sos.dll
```

使用.cordll 看看版本匹配信息(我这里故意把 mscordacwks.dll 改名以便介绍,大多数情况下 load 了 sos 后就可以直接使用)：

```
0:004> .cordll
CLR DLL status: ERROR: Unable to load DLL mscordacwks_x86_x86_2.0.50727.42.dll, Win32 error 2
```

看来 mscordacwks 不对。这个时候如果试图运行 sos 中的命令就会看到错误提示，需要修正 mscordacwks 文件后,用`.cordll -ve -u -l` 命令重新加载。于是把跟当前 mscorwks 版本匹配的 mscordacwks.dll 找出来，改名字为 `mscordacwks_x86_x86_2.0.50727.42.dll` ，然后放到 windbg 目录下。再次检查：

```
0:000> .cordll -ve -u -l
CLRDLL: Unable to get version info for 'C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\mscordacwks.dll',
Win32 error 87
CLRDLL: Loaded DLL mscordacwks_x86_x86_2.0.50727.42.dll
CLR DLL status: Loaded DLL mscordacwks_x86_x86_2.0.50727.42.dll
```

恩，这次对了！

!dumpheap

首先用!dumpheap –stat 命令检查当前所有托管类型的统计信息:

```
0:000> !dumpheap -stat
total 8861 objects
Statistics:
     MT    Count    TotalSize Class Name
7b4779b8       1         12 System.Windows.Forms.OSFeature
7b475ca8       1         12 System.Windows.Forms.FormCollection
。。。
791242ec      24       4824 System.Collections.Hashtable+bucket[]
79124228      81      27292 System.Object[]
00156210     250     126872     Free
790fa3e0    6808     397088 System.String
79124418       6  157296876 System.Byte[]
Total 8861 objects
```

恩，byte 数组怎么这么大(157MB)，有问题，继续用!dumpheap 打印出所有 byte 数组类型的

object：

```
0:000> !dumpheap -mt 79124418
 Address       MT     Size
013b59bc 79124418      12
013b73c4 79124418     268
013ce6dc 79124418   10148
037c1000 79124418 52428816
07161000 79124418 52428816
0aca1000 79124418 52428816
total 6 objects
Statistics:
      MT    Count    TotalSize Class Name
79124418        6    157296876 System.Byte[]
Total 6 objects
```

!do

这里看到三个特别大的一个比较小的。用!do 命令看看其中一个是什么东东：

```
0:000> !do 0aca1000
Name: System.Byte[]
MethodTable: 79124418
EEClass: 791244d0
Size: 52428812(0x320000c) bytes
Array: Rank 1, Number of elements 52428800, Type Byte
Element Type: System.Byte
Fields:
None
```

既然是数组，接下来用用!da (dump array)命令看看具体内容。但是这个数组很长，可能会让 windbg 忙上好一会。

!gcroot

这么大的 object,为什么没有被 GC 回收呢？用!gcroot 命令来看看谁 root 了这个数组。

```
0:000> !gcroot 0aca1000
Note: Roots found on stacks may be false positives. Run "!help gcroot" for
more info.
ebx:Root:013b362c(System.Windows.Forms.Application+ThreadContext)->
013b2d10(SampleCLR.Form1)->
013b2e64(System.Collections.Stack)->
```

```
013b2e7c(System.Object[])->
0aca1000(System.Byte[])
Scan Thread 0 OSTHread ee8
Scan Thread 2 OSTHread c00
```

原来是有 root 的啊。Thread 上的 ebx 寄存器上的 Application+ThreadContext object 引用了 Form1, Form1 包含了 Stack，Stack 内部用一个 object 数组来实现。我们的 byte array 是这个 object 数组的成员。这里的 Application+ThreadContext 表示该 object 的类型是 ThreadContext, 同时 ThreadContext 是 Application 类型的一个 inner class

来看看 Form1 object 有些什么内容：

```
0:004> !do 013b2d10
Name: SampleCLR.Form1
MethodTable: 00a75994
EEClass: 00a7150c
Size: 340(0x154) bytes
 (C:\Documents and Settings\LiXiong\My Documents\MyCode\SampleCLR\SampleCLR\bin\Debug\SampleCLR.exe)
Fields:
      MT    Field   Offset                 Type VT     Attr    Value Name
790f9c18  4000184       4       System.Object  0 instance 00000000 __identity
7a745c0c  40008bc       8 ...ponentModel.ISite  0 instance 00000000 site
7a742e54  40008bd       c ....EventHandlerList  0 instance 013cd4e8 events
。。。
790f9c18  4001e29     ba4       System.Object  0   static 013b33f4 EVENT_MAXIMIZEDBOUNDSCHANGED
7a745214  4000002     13c ...tModel.IContainer  0 instance 00000000 components
7b46ff40  4000003     140 ...dows.Forms.Button  0 instance 013cc0e8 button1
7b46ff40  4000004     144 ...dows.Forms.Button  0 instance 013cc254 button2
7b46ff40  4000005     148 ...dows.Forms.Button  0 instance 013cc3c0 button3
7910f690  4000006     14c ...Collections.Stack  0 instance 013b2e64 stc
```

有兴趣可以继续用!do 命令对 object 成员追进去看。哈哈，这里看到 button1, button2, buttont3 三个我们放上去的按钮，追进去：

```
0:004> !do 013cc0e8
Name: System.Windows.Forms.Button
MethodTable: 7b46ff40
EEClass: 7b46fea0
Size: 168(0xa8) bytes

(C:\WINDOWS\assembly\GAC_MSIL\System.Windows.Forms\2.0.0.0__b77a5c561934e089\System.Windows.Forms.
dll)
Fields:
      MT    Field   Offset                 Type VT     Attr    Value Name
790f9c18  4000184       4       System.Object  0 instance 00000000 __identity
7a745c0c  40008bc       8 ...ponentModel.ISite  0 instance 00000000 site
```

```
7a742e54  40008bd       c ....EventHandlerList  0 instance 013cc52c events
790f9c18  40008bb     108       System.Object  0   static 00000000 EventDisposed
7b46d738  4001107      10 ...ntrolNativeWindow  0 instance 013cc1b8 window
7b46b3fc  4001108      14 ...ows.Forms.Control  0 instance 013b2d10 parent
7b46b3fc  4001109      18 ...ows.Forms.Control  0 instance 013b2d10 reflectParent
7b46e894  400110a      1c ...orms.CreateParams  0 instance 013cc220 createParams
790fed1c  400110b      34        System.Int32  0 instance       74 x
790fed1c  400110c      38        System.Int32  0 instance       72 y
790fed1c  400110d      3c        System.Int32  0 instance      174 width
790fed1c  400110e      40        System.Int32  0 instance       47 height
790fed1c  400110f      44        System.Int32  0 instance      174 clientWidth
790fed1c  4001110      48        System.Int32  0 instance       47 clientHeight
790fed1c  4001111      4c        System.Int32  0 instance 16908815 state
790fed1c  4001112      50        System.Int32  0 instance     2120 state2
7b479824  4001113      54        System.Int32  0 instance   421398 controlStyle
790fed1c  4001114      58        System.Int32  0 instance        0 tabIndex
790fa3e0  4001115      20       System.String  0 instance 013cc06c text
…
```

内容够详细了吧。看看 text 变量是不是 button 上的文字：

```
0:004> !do 013cc06c
Name: System.String
MethodTable: 790fa3e0
EEClass: 790fa340
Size: 32(0x20) bytes
 (C:\WINDOWS\assembly\GAC_32\mscorlib\2.0.0.0__b77a5c561934e089\mscorlib.dll)
String: button1
Fields:
     MT    Field   Offset                 Type VT     Attr    Value Name
790fed1c  4000096       4        System.Int32  0 instance       8 m_arrayLength
790fed1c  4000097       8        System.Int32  0 instance       7 m_stringLength
790fbefc  4000098       c         System.Char  0 instance      62 m_firstChar
790fa3e0  4000099      10       System.String  0   shared   static Empty
   >> Domain:Value  0014c9d8:790d6584 <<
79124670  400009a      14       System.Char[]  0   shared   static WhitespaceChars
   >> Domain:Value  0014c9d8:013b17f8 <<
```

恩，果然是。有兴趣的话，可以继续拿小白鼠修修改改，或者随意地找当前系统中运行着的托管程序检查作为练习。更多的命令示例穿插在后面的章节中。

从前面的研究可以看到，对托管代码的调试比非托管代码容易得多。有了类型信息，调试人员可以获取非常直观的统计信息和类型信息。CLR 建立了一个优秀的平台，通过几个关键的命令就可以找出关键的数据结构，定位引发问题的 object，并且获取该 object 方方面面的信息。

[案例分析, ASP.NET High CPU]

问题描述:客户的 aspnet_wp.exe 进程常常发生高 CPU 的情况。需要等很长时间或者重新启动 IIS 服务才能缓解问题

客户在问题发生的时候只抓取了 dump 文件，直接打开 dump 进行分析：

既然是分析托管程序，所以首先加载 sos。由于用户使用的.NET Framework 1.1，所以这里加载 Windbg 安装目录下的 sos 文件，而不是.NET Framework 安装目录下的 sos 文件：

```
0:000> .load clr10\sos
```

为了确保.NET Framework 目录下的 sos 没有加载起来产生混淆，使用!chain 命令看看当前的调试扩展 DLL 都加载了些什么：

```
0:000> !chain
Extension DLL search Path:

D:\Debuggers\winext;D:\Debuggers\winext\arcade;D:\Debuggers\W2KFre;D:\Debuggers\pri;D:\Debuggers;D
:\Debuggers\winext\arcade;
 Extension DLL chain:
   clr10\sos: image 6.6.0007.5, API 1.0.0, built Sun Jul 09 04:11:04 2006
       [path: D:\Debuggers\clr10\sos.dll]
   dbghelp: image 6.6.0007.5, API 6.0.6, built Sun Jul 09 04:11:32 2006
       [path: D:\Debuggers\dbghelp.dll]
   ext: image 6.6.0007.5, API 1.0.0, built Sun Jul 09 04:10:52 2006
       [path: D:\Debuggers\winext\ext.dll]
   uext: image 6.6.0007.5, API 1.0.0, built Sun Jul 09 04:11:02 2006
       [path: D:\Debuggers\winext\uext.dll]
   ntsdexts: image 5.00.2195.6618, built Tue Nov 19 08:21:06 2002
       [path: D:\Debuggers\W2KFre\ntsdexts.dll]
```

恩，一切正常。

!threads

接下来用!threads 命令看看托管线程的统计信息：

```
0:000> !threads
Loaded Son of Strike data table version 5 from "C:\WINNT\Microsoft.NET\Framework\v1.1.4322\mscorwks.dll"
ThreadCount: 22
UnstartedThread: 0
BackgroundThread: 22
PendingThread: 0
```

```
DeadThread: 0
                         PreEmptive   GC Alloc                 Lock
       ID  ThreadOBJ      State    GC      Context         Domain    Count APT Exception
  1  0x7e0 0x001512e8   0x200a220 Enabled  0x00000000:0x00000000 0x00171600     1 MTA
  7  0x958 0x00151ad8     0xb220 Enabled  0x00000000:0x00000000 0x0014be18     0 MTA (Finalizer)
  4  0x928 0x001984e0      0x220 Enabled  0x00000000:0x00000000 0x00171600     3 MTA
  6  0x954 0x001986e0   0x2000220 Enabled  0x00000000:0x00000000 0x00171600     1 MTA
  9  0x980 0x00198920   0x2000220 Enabled  0x00000000:0x00000000 0x00171600     1 MTA
 12  0x974 0x001a2868   0x3800220 Enabled  0x00000000:0x00000000 0x00171600     1 MTA (Threadpool
Worker)
 10  0x990 0x001aa008   0x2000220 Enabled  0x00000000:0x00000000 0x00171600     1 MTA
 。。。
 27  0x24c 0x09027ac8   0x3800220 Enabled  0x00000000:0x00000000 0x00171600     1 MTA (Threadpool
Worker)
 28  0xc7c 0x0c0105e8   0x1800220 Enabled  0x00000000:0x00000000 0x0014be18     0 MTA (Threadpool
Worker)
```

!tp

恩，一共没有多少个线程运行阿。看看 threadpool 的情况：

```
0:000> !tp
CPU utilization 96%
Worker Thread: Total: 8 Running: 8 Idle: 1 MaxLimit: 20 MinLimit: 1
Work Request in Queue: 0
--------------------------------------
Number of Timers: 0
--------------------------------------
Completion Port Thread: Total: 13 Free: 1 MaxFree: 2 CurrentLimit: 1 MaxLimit: 20 MinLimit: 1
```

哇，CPU 利用率 96%阿。!tp 命令还给出了 threadpool 的设定，这里的 `MaxLimit` 才 20 个 thread，明显没有依照下面的 KB 把 threadpool 的设置优化：

Contention, poor performance, and deadlocks when you make Web service requests from ASP.NET applications
http://support.microsoft.com/?id=821268

`MaxLimit` 不一定是 high CPU 问题的根源。使用!runaway f 命名看看各个线程的 CPU 使用情况。一看是不是有线程的确很忙，二看 CPU 使用有没有均摊到多个线程上：

```
0:004> vertarget
Windows 2000 Version 2195 (Service Pack 4) UP Free x86 compatible
Product: Server, suite: TerminalServer SingleUserTS
kernel32.dll version: 5.00.2195.7006
```

```
Debug session time: Fri Jul  7 14:42:59.000 2006 (GMT+8)
System Uptime: 0 days 0:13:39.687
Process Uptime: 0 days 0:11:44.000
  Kernel time: 0 days 0:00:53.000
  User time: 0 days 0:06:18.000
0:004> !runaway f
 User Mode Time
  Thread       Time
   4:928       0 days 0:00:07.109
   1:7e0       0 days 0:00:06.031
   6:954       0 days 0:00:05.484
   9:980       0 days 0:00:05.281
  12:974       0 days 0:00:05.156
...
Kernel Mode Time
  Thread       Time
   1:7e0       0 days 0:01:33.640
   4:928       0 days 0:01:23.296
   9:980       0 days 0:00:41.093
   6:954       0 days 0:00:40.390
  18:aac       0 days 0:00:27.234
...
Elapsed Time
  Thread       Time
   0:188       0 days 0:11:43.123
   2:7e4       0 days 0:11:43.030
   1:7e0       0 days 0:11:43.030
   3:7e8       0 days 0:11:43.014
   4:928       0 days 0:10:22.905
```

恩，进程启动才 11 分钟，排在 CPU 实用率前 5 的 thread 分别在 kernel mode 就花了 1 分钟左右。还没算有可能已经死去的线程。看来高 CPU 果然是当前进程导致的了，而且耗了很多在 kernel mode 上。继续用!clrstack 看看各个线程都在做什么：

```
0:000> ~* e !clrstack
```

由于输出太长，这里省略。结论是几乎所有的 CLR 线程都等在下面一个相同的托管 callstack 上：

```
ESP        EIP
0x00b6f104  0x77f88f03 [FRAME: GCFrame]
0x00b6f1c0  0x77f88f03 [FRAME: HelperMethodFrame]
0x00b6f214  0x09d30ba0 [DEFAULT] [hasThis] Void
log4net.Repository.Hierarchy.Logger.CallAppenders(Class log4net.spi.LoggingEvent)
0x00b6f250  0x09d30aa5 [DEFAULT] [hasThis] Void
```

```
log4net.Repository.Hierarchy.Logger.ForcedLog(String,Class log4net.spi.Level,Object,Class
System.Exception)
0x00b6f26c  0x09d30a4c [DEFAULT] [hasThis] Void log4net.Repository.Hierarchy.Logger.Log(String,Class
log4net.spi.Level,Object,Class System.Exception)
0x00b6f284  0x09d324ef [DEFAULT] [hasThis] Void log4net.spi.LogImpl.Info(Object)
0x00b6f294  0x09d3242c [DEFAULT] Void CustomerCode.CUNameSpace.Util.Logger.Info(String)
0x00b6f2c0  0x09d32c0d [DEFAULT] Void CustomerCode.CUNameSpace.SomeClass.Dispose(Class
System.Data.SqlClient.SqlCommand)
```

对应的非托管 callstack 是：

```
ChildEBP RetAddr  Args to Child
0c30f208 7c59a1fb 00000001 0c30f230 00000000 NTDLL!ZwWaitForMultipleObjects+0xb
0c30f258 792976dc 0c30f230 00000000 00000001 KERNEL32!WaitForMultipleObjectsEx+0xea
0c30f288 79297c3d 00000001 00172ffc 00000001 mscorwks!Thread::DoAppropriateWaitWorker+0xc1
0c30f2dc 792ba257 00000001 00172ffc 00000001 mscorwks!Thread::DoAppropriateWait+0x46
0c30f360 7924e2da 09027ac8 ffffffff 00000000 mscorwks!AwareLock::EnterEpilog+0x9d
0c30f37c 792ed445 0ac2e708 05339f68 053406b0 mscorwks!AwareLock::Enter+0x69
0c30f410 09d30ba0 053406b0 05339f68 0ac2e708 mscorwks!JITutil_MonContention+0x124
WARNING: Frame IP not in any known module. Following frames may be wrong.
0c30f4a8 79990d83 0c30f574 00000000 0aa9f3a8 0x9d30ba0
0c30f4bc 09d359c6 0ac2520c 0ac25128 09d3645c mscorlib_79980000+0x10d83
0c30f5c0 799ec94c 6c9f8000 08c86f6f 0c30f664 0x9d359c6
0c30f5f8 791f48d5 0ab5ae2c 05339f68 053406b0 mscorlib_79980000+0x6c94c
0c30f664 09d34b6a 0c30f6dc 0ab5b5a4 0ab5b390 mscorwks!JITutil_IsInstanceOfBizarre+0x15d
```

另外有一个 thread 等在下面的 callstack:

```
ESP       EIP
0x00e6f0c8  0x77f88c97 [FRAME: NDirectMethodFrameStandalone] [DEFAULT] Boolean
Microsoft.Win32.Win32Native.MoveFile(String,String)
0x00e6f0d8  0x79a9ab67 [DEFAULT] [hasThis] Void System.IO.FileInfo.MoveTo(String)
0x00e6f0fc  0x09d3748a [DEFAULT] [hasThis] Void
log4net.Appender.RollingFileAppender.RollFile(String,String)
0x00e6f134  0x09d37270 [DEFAULT] [hasThis] Void log4net.Appender.RollingFileAppender.RollOverSize()
0x00e6f180  0x09d314a4 [DEFAULT] [hasThis] Void log4net.Appender.RollingFileAppender.Append(Class
log4net.spi.LoggingEvent)
0x00e6f19c  0x09d311ad [DEFAULT] [hasThis] Void log4net.Appender.AppenderSkeleton.DoAppend(Class
log4net.spi.LoggingEvent)
0x00e6f1d4  0x09d30dfc [DEFAULT] [hasThis] I4
log4net.helpers.AppenderAttachedImpl.AppendLoopOnAppenders(Class log4net.spi.LoggingEvent)
0x00e6f218  0x09d30bba [DEFAULT] [hasThis] Void
log4net.Repository.Hierarchy.Logger.CallAppenders(Class log4net.spi.LoggingEvent)
0x00e6f254  0x09d30aa5 [DEFAULT] [hasThis] Void
log4net.Repository.Hierarchy.Logger.ForcedLog(String,Class log4net.spi.Level,Object,Class
```

```
System.Exception)
0x00e6f270  0x09d30a4c [DEFAULT] [hasThis] Void log4net.Repository.Hierarchy.Logger.Log(String,Class
log4net.spi.Level,Object,Class System.Exception)
0x00e6f288  0x09d324ef [DEFAULT] [hasThis] Void log4net.spi.LogImpl.Info(Object)
0x00e6f298  0x09d3242c [DEFAULT] Void CustomerCode.CUNameSpace.Util.Logger.Info(String)
```

对应的非托管 callstack 是：

```
ChildEBP RetAddr  Args to Child
00e6efac 7c587ead 00000a8c 00e6effc 08fbd098 NTDLL!NtSetInformationFile+0xb
00e6f088 7c587b79 0ae96ff0 0ae97464 00000000 KERNEL32!MoveFileWithProgressW+0x319
00e6f0d0 79a9ab67 051e763c 0ae96fa4 0ae96c9c KERNEL32!MoveFileW+0x13
WARNING: Stack unwind information not available. Following frames may be wrong.
00e6f144 792fd28d 0ae96c2c 051e7cac 051e7cac mscorlib_79980000+0x11ab67
00e6f128 09d37270 0ae96ce4 0af04168 0a9fb1d0
mscorwks!AppDomainStringLiteralMap::GetInternedString+0x50
00e6f1f8 7998bf28 00000001 00e6f22c 00000000 0x9d37270
00e6f2e0 791f3d7e 0aebfe0c 00000000 79989d6d mscorlib_79980000+0xbf28
00e6f2ec 79989d6d 00000000 095b88a1 00e6f334 mscorwks!COMString::EqualsString+0x48
00000000 00000000 00000000 00000000 00000000 mscorlib_79980000+0x9d6d
```

这里可以看到，所有 thread 都跟 log4net 这个东西好像有关。Log4net 是一个开发包，专门用来记录 log 文件的。看来用户使用了这个开发包，而且现在所有的 thread 都在执行这个包上的代码。区别是大多数 thread 是 idle 的，因为非托管代码等在 WaitForMutipleObjects 上，而且还跟 CLR 的 AwareLock 相关。Thread4 正在执行 MoveFile 这个 API，所以是 busy。

!SyncBlk

回忆对 rotor 代码的研究，会想起 CLR 似乎是用 AwareLock 来实现的 lock。于是立刻用!syncblk 检查：

```
0:000> !SyncBlk
Index SyncBlock MonitorHeld Recursion   Thread  ThreadID     Object Waiting
  135 0x00172fe8         37         1 0x1984e0 0x928   4   0x05339f68
log4net.Repository.Hierarchy.DefaultLoggerFactory/LoggerImpl
          Waiting threads: 1*** WARNING: Unable to verify checksum for mscorlib.dll
*** ERROR: Module load completed but symbols could not be loaded for mscorlib.dll
 6 9 10 12 13 14 15 16 17 18 20 22 23 24 25 26 27
```

哇，果然是 thread 4 占住了 lock，而其他 17 个 thread 都在等他。前面介绍过，多个 thread 一起等一个资源，当这个资源释放的时候，就会发生争用(contention)导致高 CPU。这样看来问题已经搞定一半了。为了进一步证明这一点，可以再仔细分析一下 log4net 对应方法的实现，在下面的 callstack 中找到 CallAppenders 的 EIP:

```
ESP        EIP
0x00b6f104  0x77f88f03 [FRAME: GCFrame]
0x00b6f1c0  0x77f88f03 [FRAME: HelperMethodFrame]
0x00b6f214  0x09d30ba0 [DEFAULT] [hasThis] Void
log4net.Repository.Hierarchy.Logger.CallAppenders(Class log4net.spi.LoggingEvent)
```

!ip2md

使用!ip2md 找到对应的 module 信息：

```
0:027> !ip2md 0x09d30ba0
MethodDesc: 0x095cc558
Jitted by normal JIT
Method Name : [DEFAULT] [hasThis] Void log4net.Repository.Hierarchy.Logger.CallAppenders(Class
log4net.spi.LoggingEvent)
MethodTable 0x95cc5bc
Module: 0x21fc78
mdToken: 0x060003ce (c:\winnt\microsoft.net\framework\v1.1.4322\temporary asp.net
files\CUPath\7440cfe4\645362bc\assembly\dl2\f47b851c\00603dcf_e644c501\commonobjects.dll)
Flags : 0x80
Method VA : 0x09d30b58
```

这里找到了这个方法是在 commonobjects.dll 文件中了。看看这个文件的信息：

```
0:027> lmvm commonobjects
start    end       module name
081f0000 0823a000   commonobjects   (deferred)
    Image path: c:\WINNT\microsoft.net\framework\v1.1.4322\temporary asp.net
files\mobilepull\7440cfe4\645362bc\assembly\dl2\f47b851c\00603dcf_e644c501\commonobjects.dll
    Image name: commonobjects.dll
    Has CLR image header, track-debug-data flag not set
    File version:     0.0.1.0
    。。。
```

!savemodule

这个文件加载到了 081f0000 这个地址上。所以可以用下面的命令把这个 dll 保存到本地：

```
0:027> !savemodule 081f0000  D:\xiongli\CommonObjects.dll
Successfully saved file: D:\xiongli\CommonObjects.dll
```

使用 reflector 可以看到 CallAppenders 函数的实现包含了：

```
for (Logger logger1 = this; logger1 != null; logger1 = logger1.m_parent)
```

```
{
    lock (logger1)
    {
```

这里果然使用了 lock！所以，该问题是客户的代码里面使用了 log4net，然而 log4net 某一个地方的同步可能会锁住所有使用 log4net 的线程，导致 contention。给客户的建议是：

1. 减少对 log4net 的使用，或者想办法把 log 写入不同的文件尝试看是否可以避免 lock 发生。
2. 修改 machine.config 来优化 threadpool 的配置
3. 联系 log4net 厂商看看有没有其他改进的方法。

If broken it is, fix it you should

http://blogs.msdn.com/tess/

上面这个 blog 全是关于 CLR Debugging 的案例研究，包括 1.1 和 2.0，不仅仅涵盖了 CLR 开发中通病，还包含了一些稀奇古怪的问题。

题外话和相关讨论

CLR 提供了非常完善的平台。无论是 deadlock 还是 memory issue，都非常容易调试。但是根据 RFC 1925 的第七条：

(7) It is always something
(7a) (corollary). Good, Fast, Cheap: Pick any two (you can't have all three).

you can't have all three ，嗯，所以 CLR 上肯定会有不得不权衡让步的地方。考虑下面这些问题：

ReleaseCOMObject

在 CLR 中通过 COM Interop 创建了一个 COM 对象，使用完这个对象后，是否应该立刻使用 ReleaseCOMObject CLR API 来通过接口指针的 Release 方法减少这个这个 COM 对象的引用计数？
http://msdn2.microsoft.com/en-us/library/system.runtime.interopservices.marshal.releasecomobject.aspx

如果不及时调用这个 API，那么这个 COM 组件的释放就会滞留到 GC 发生的时候，通过 Finalizer thread 中 Release 的调用来减少引用计数。COM 组件是带套件属性的，也就是说 COM 接口指针的 Release 方法最终会被派发到对应的线程中执行。

由于 Finalizer thread 是一个 CLR 的辅助线程，标记为 MTA Appartment，STA COM 组件不会在这个 thread 中被创建，所以所有 STA COM 组件的 Release 都会被派发到对应的 STA 线程。

由于 GC 的发生没有固定的规律，所以没有办法保证释放 COM 的 STA 线程是否可以及时完成这项任务。比如 STA 线程被数据库请求阻塞，就无法维持消息循环，那么 COM 就无法释放，整个 Finalizer thread 也会随之被堵塞，导致内存持续增长。如果使用了多个 COM 组件，组件之间如果有依赖关系，需要按特定顺序释放的话，Finalizer thread 也无法保证正确的顺序，所以很可能导致死锁。

如果使用完 COM 就在正确的时机在正确的线程，手动调用 ReleaseCOMObject 来释放，就可以解决上面难题。但是，开发人员一定能找到这一个正确的时机吗？考虑这样一种情况。当函数 A 获取一个 COM object 后，需要把这个 COM object 传递给另一个线程，同时当前函数还会继续使用这个 object。那最后应该是哪一个线程来调用 ReleaseCOMObject 呢？还是两者都调用？如果才能知道两个线程都结束了对这个 COM 的使用呢？

问题的关键在于 CLR 管理内存的机制。CLR 通过检查某一个 object 是否还有 root 来判断是否可以释放，而 COM 通过引用计数来管理。当传递 COM 给其它线程的时候，传递的只是 CLR COM Interop Wrapper object，COM 实际的引用计数并没有增加，所以最后应该有一个地方来调用 ReleaseCOMObject。但是除了 CLR 的 GC 引擎外，谁也不知道这个 CLR COM Interop Wrapper object 什么时候可以被回收，所以没办法找到手动调用 ReleaseCOMObject 的恰当时机。

在这种情况下，唯一能够做的就是都不去手动释放，把责任推给 CLR，让 CLR 通过 root 机制来管理，最后在 Finalizer thread 中去释放。

如果情况更复杂一点，CLR COM Interop Wrapper object 还在托管代码和非托管代码中间传递多次，那应该怎么办呢？

Cbrumme 有一篇 blog 专门讨论了这个问题:
http://blogs.msdn.com/cbrumme/archive/2003/04/16/51355.aspx

其中最后一句话是: “3) If you have a case where you are creating COM objects at a high rate, *passing them around freely*, choking the Finalizer thread, and consuming a lot of unmanaged resources... you are out of luck.”

内存移动

在 CLR 中需要传递一块托管的 buffer 到非托管 API，以便非托管 API 可以异步地向这个 buffer 填充数据。如果开发人员忘记用 C#中的 fixed 关键字和或者 MC++中的__pin 关键字 pin 住

这个 buffer object,那么 GC 发生的时候 buffer 就可能被 GC 移动导致地址发生改变。当非托管 API 试图写入 buffer 的时候，如果 buffer 被 GC 移动，往往会引发 Access Violation 和 crash，更严重的情况是损坏了其它模块的数据，继续运行一段时间后莫名其妙地崩溃。

这种问题调试起来非常头疼。因为问题不会立刻暴露出来，从 callstack 上也看不出任何线索，而且问题不容易重现，每次出错的地方也不尽相同。由于 CLR 增加 OS 上的另一层内存管理机制，本意是减少开发人员的负担，但结果却使得开发人员要考虑更多的因素，比如 buffer 是否要 pin 住。

解决这类问题单通过检查 dump 很难找出根源，一定要有一个重现环境，配合 CLR 的 stresslog 来调试。

所以,CLR 并不能让所有的事情都变得简单。

看了上面的解释，千万不要急着去找你的代码，看看有没有正确地 pin 住 object，或者庆幸自己的代码没有 pin 也没有啥毛病。好多时候，CLR 会帮助你自动 pin 住这些 buffer 的，比如同步调用。多余的 pin 往往还会导致性能问题。详细说明请参考：

Asynchronous operations, pinning
http://blogs.msdn.com/cbrumme/archive/2003/05/06/51385.aspx
“Applications that explicitly pin buffers around PInvoke calls are *often* doing so unnecessarily.”

内存不移动

既然内存移动会导致问题，那就注意该 pin 的时候 pin 住好了。检查 CLR 的 Network Libaray，在进行 TCP 异步调用的时候，CLR 的确 pin 住了必要的 buffer。

这下子程序不会崩溃了，但是 GC 无法移动 pin 住的 buffer，又会带来内存碎片的问题，导致内存使用率低下和 OutOfMemory Exception。下面的文章有详细的介绍：

OutOfMemoryException and Pinning
http://blogs.msdn.com/yunjin/archive/2004/01/27/63642.aspx

看来 pin 或不 pin 都不好啊。

臭名昭著的 mixed DLL loading deadlock

在 CLR1 上，不建议大量使用 MC++开发的 module。原因是存在不可避免的死锁可能：

详细说明可以参考：

Mixed DLL Loading Problem
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dv_vstechart/html/vcconMixedD

LLLoadingProblem.asp

提到 MC++,假设你要在 C#中调用 LoadUserProfile 这个 API，需要接受一大串 C++结构阿，联合类型作为参数，在 C#定义类型是很头疼的。我同事想到一个用 MC++自动生成类型的简单方法：

用 MC++直接写一个例子程序调用对应的 API，使用 windows.h 中定义好的结构。编译后用 reflector 把 assembly 反编译成 C#,就可以轻松获取类型定义的代码了。轻松又准确。

有用的练习

有了 rotor 和 CLR 调试工具后，就可以研究一些复杂程序的实现啦，比如:

1. Windows 2003 的 w3wp.exe 进程用来提供 Web 服务。该进程本身是一个 unmanaged 进程，但是 ASPX 页面也可以运行在 w3wp.exe 进程中。W3wp.exe 是如何加载 CLR 的呢？
2. DataTable 的数据结构是怎样的？在 dump 中找到一个 DataTable 的地址后，如何定位到任意单元格呢？
3. 能否写出 debug script 自动判断是否有 SQL Connection Leak，自动打印出发生泄漏的连接字符串?

写在本章结束前

本章介绍了 CLR 大致的设计，结合 windbg 和 rotor 找到了相关的函数名，以便深入了解 CLR 的细节。接下来介绍了 sos 提供的便利的调试方法，最后介绍了让 CLR 设计和调试尴尬的情况。

由于 CLR 在代码和 OS 之间添加了额外的抽象层，如果要开发高质量的，跟非托管代码互动的程序，需要对 CLR 有详细的了解。Rotor，windbg 和下面这五个 blog 都是非常好的资源。如果能够读完这五个 blog，相信会对 CLR 有非常深刻的理解：

http://blogs.msdn.com/cbrumme
http://www.cnblogs.com/flier
http://blogs.msdn.com/maoni
http://blogs.msdn.com/tess
http://blogs.msdn.com/yunjin

第四部分 ，一些经验

在第一章中，分析了如何思考问题。最后一章，让我们回到问题本身，结合前面介绍的工具和知识，总结一些常见问题的思路和经验。

本章首先介绍分析问题前的准备。然后依次介绍排查崩溃，性能和资源泄露的一些经验。

4.1 排错开始前

别害怕难题，让难题害怕你

回想一下第一章中讨论过的问题，是不是看起来挺有趣的？但是，试想这些问题发生在繁忙的生产环境中，上级要求在两天内解决问题，你还会觉得有趣吗？会不会觉得束手无策？首先想到的是不是找借口来推脱责任？

如果在排错开始前，调试者已经怀畏惧心理，是不可能找到问题真相的。下面几点，希望可以帮助克服这样的畏惧情绪。

1. 屡试不爽的方法是存在的

无论多么复杂的程序，总可以被简化。所以，我们可以先把程序的功能砍掉一半，看看问题是否发生，借此缩小问题的范围。重复使用这样的二分法，大不了把程序简化到只剩一行代码。所以，无论什么问题总可以用这样的方法来解决。

2. 如果问题很难重现，log 是最后的办法

就想第一章中讨论的 session lost 一样，只要细心地在程序中添加 log，当 log 足够详细，分析 log 足够耐心，总可以在问题发生后找到痕迹的。

3. 调试者不过是帮忙找线索而已,不要对自己要求太高

前面的例子可以看到，大多数问题都是开发人员的责任。一个庞大的项目牵涉了很多开发人员，不应该让调试者来承担问题的责任。你可能是整个项目的负责人，可能问题发生后所有人都指望你来解决，但是从如何解决问题的角度上说，不要对自己有太高的期望值。调试者正确的态度是尽可能地找到线索，以便找到更合适的人和资源来解决问题。看看前面的例子，大多数是情况下都是找到问题的线索，最后解决问题还得依靠开发人员修改代码。

当有了正确的态度和必然可行的计划，不应该再对问题有畏惧的心理。加上下面的一系列提问,就可以上路排错了:

一些有用的提问

1. 是否打上了最新的补丁？
2. 是否查询了 support.microsoft.com ，有没有遇上已知问题，有没有现成的 hotfix 可以解决？
3. 问题是偶尔发生还是总可以重现？
4. 只发生在固定的几台机器上还是所有环境下都有问题？
5. 有没有简化的可以重现问题的程序可以测试？
6. 问题发生时候的 screen-shot 抓了吗？
7. 重现问题的具体步骤是什么？有什么特别的吗?
8. 问题发生后，有哪些方法可以暂时解决和缓解？
9. 装了防毒软件吗？有没有用防火墙？
10. 软件架构大致是怎么样的？数据在整个软件环境中是如何流动的？
11. 排错是在生产环境上做呢还是在测试环境上做？
12. 用到数据库了吗？用的什么数据库
13. 问题跟负载相关吗？
14. 用什么开发的？全是托管代码？有没有用 VB 或者 C++？有没有用到 COM+?
15. 网络环境如何？用到NAT了吗？
    http://support.microsoft.com/?id=248809
16. 是 cluster 的环境吗？有没有用负载均衡？
17. 是 domain 的环境吗？
18. 如果不是 domain 环境，有没有尝试在服务器和客户端创建相同密码的相同帐户测试？
19. 最近装了什么系统补丁了吗？有没有尝试卸载掉补丁后测试？
20. 除了问题本身以外，系统有什么异常吗？比如 CPU 和内存使用，还有端口的使用情况
21. 程序的负载在正常范围内吗?
22. 系统日志中有什么异常吗？
23. 有没有尝试改变程序的帐号测试。比创建另外一个管理员帐号登陆来测试
24. 程序有 UI 吗？是双击启动的还是通过任务管理定时启动的？

记得抓取 MPS REPORT

MPS REPORT 是获取系统信息的一个工具。

具体介绍和下载地址分别是:

Overview of the Microsoft Configuration Capture Utility (MPS_REPORTS)
http://support.microsoft.com/kb/818742/
http://www.microsoft.com/downloads/details.aspx?FamilyId=CEBF3C7C-7CA5-408F-88B7-F9C

79B7306C0&displaylang=en

MPS REPORT 收集了下面这些信息,能够帮助排错人员清楚了了解系统的方方面面:

1. 系统日志文件
2. 通过 winmsd 或者 msinfo32 运行 System Information 工具，然后打开 MPS REPORT 收集下来的 NFO 文件，可以看到硬件详细配置，正在运行的进程，正在使用的 DLL，服务的配置信息，驱动程序的信息等等
3. Hotfix.txt 文件包含了打补丁的情况和日期
4. System32.txt 文件包含了 system32 目录下 module 的详细信息

Dump 初探

如果有出错程序的 dump 文件，哪怕 dump 不是在合适的时机获取的，也可以分析出有用信息:

通过 `vertarget` 察看系统版本和系统运行了多少时间

通过`!peb` 察看环境变量的情况。由于很多第三方软件都习惯把自身路径添加到环境变量中，所以这里很多时候可以看出一些已经安装的软件

同时还能看到当前进程所加载的 DLL 和对应路径。可以重点检查：

1 有没有防毒程序的 DLL 加载
2 有没有类似 MSVCRTD 或者 MFC42D 这样的 debug 版本 DLL 加载。如果有，说明程序使用的某些组件是在 Debug 模式下编译的
3 通过 MFC42 这样的 DLL 大致判断程序是如何开发的，比如是否使用 MFC 或者 ATL。如果加载了 mscoree,说明程序中还是用了托管代码
4 通过 lmvm 命令检查值得怀疑的 DLL 的详细版本和公司名称。必要情况下还可以使用 sos!savemodule 命令把 DLL 保存到本地检查链接情况等等
5 通过 lmf 命令还可以检查是否有 unload 的 module。很多问题是由于正在使用的 module 被 unload 导致的，也有很多问题的最后会导致某些 module 被 unload。看看这里 unload module 是不是正常情况
6 检查 module 的数量，是否有动态生成的 module 加载。如果进程中加载的 module 过多，容易导致内存地址碎片。比如对于 asp.net 的程序，很多 module 是对应于 aspx 页面动态编译生成的，需要通过设置 debug=false 来激活 batch compilation 以便减少 module.。
7 通过 dump 文件的尺寸来判断内存使用情况。一般 dump 的大小跟实际内存使用比较一致。
8 检查某些系统 dll 来判断某些系统组件是否已经升级到最新。比如检查 mscorwks 的版本就可以判断.NET Framework 的版本以及是否打过 SP1，检查 msado15 就可以判断 MDAC 的版本

有了上面的准备工作后,下面详细讨论崩溃,性能和资源泄露的一些经验:

4.2 崩溃

崩溃是指进程非预期退出.

绝大多数的崩溃是由 Access Violation Exception 直接导致的.由于代码上的问题,引起空指针引用, Heap corruption 等等,最后导致无效地址访问引发崩溃.

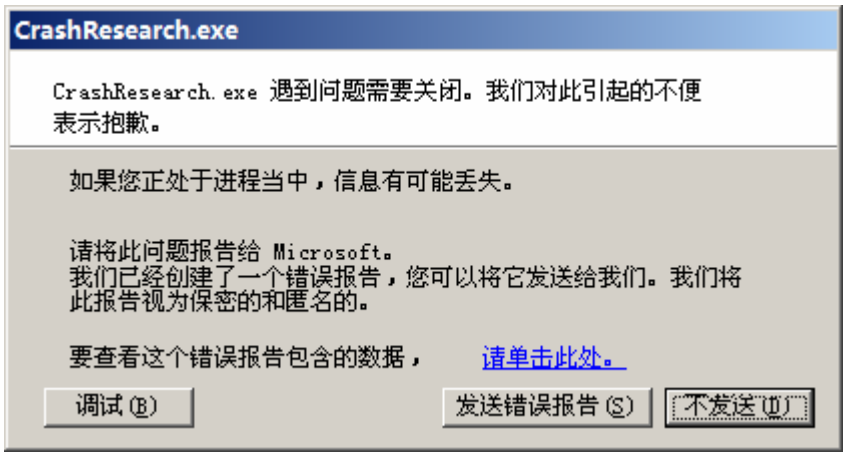
解决崩溃的关键在于对程序细节的了解，抓起准确的信息,分析出问题发生的来龙去脉。

不同的死相

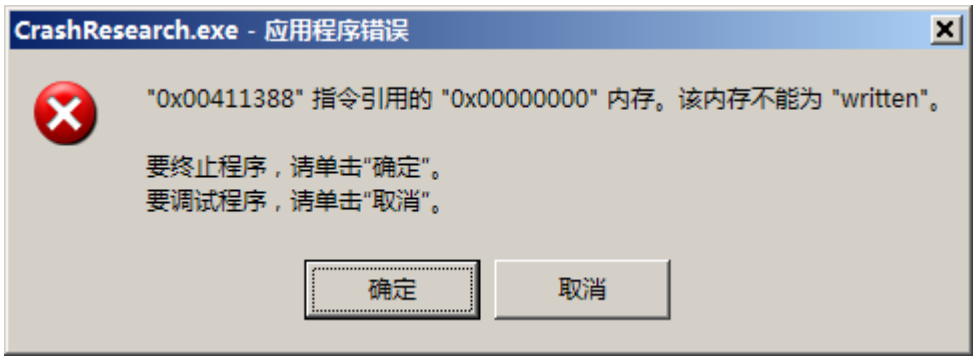
程序崩溃的表现是多姿多彩的。了解程序的死相能够帮助问题的定位。
就下面这段代码，崩溃后可能有两种不同的表现:

```
int _tmain(int argc, _TCHAR* argv[])
{
    char *p=0;
    *p=0;
    return 0;
}
```

激活了操作系统错误报告功能:



禁止错误报告功能:

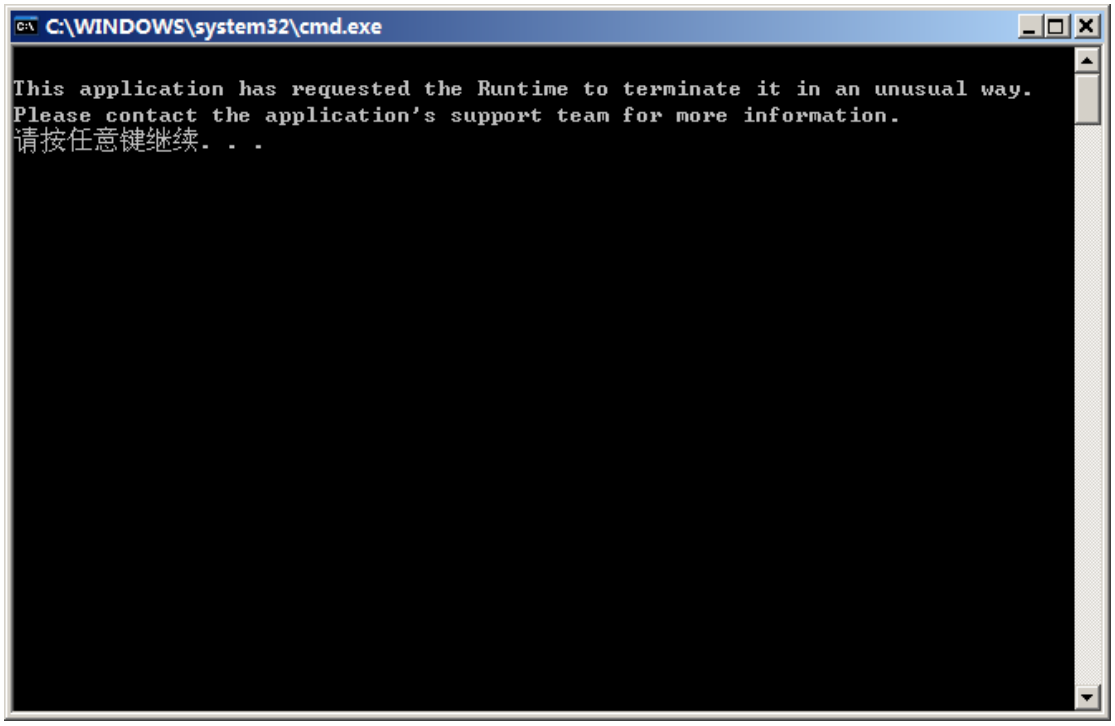


稍微改变一下代码：

```
int _tmain(int argc, _TCHAR* argv[])
{
    throw 0;
    return 0;
}
```

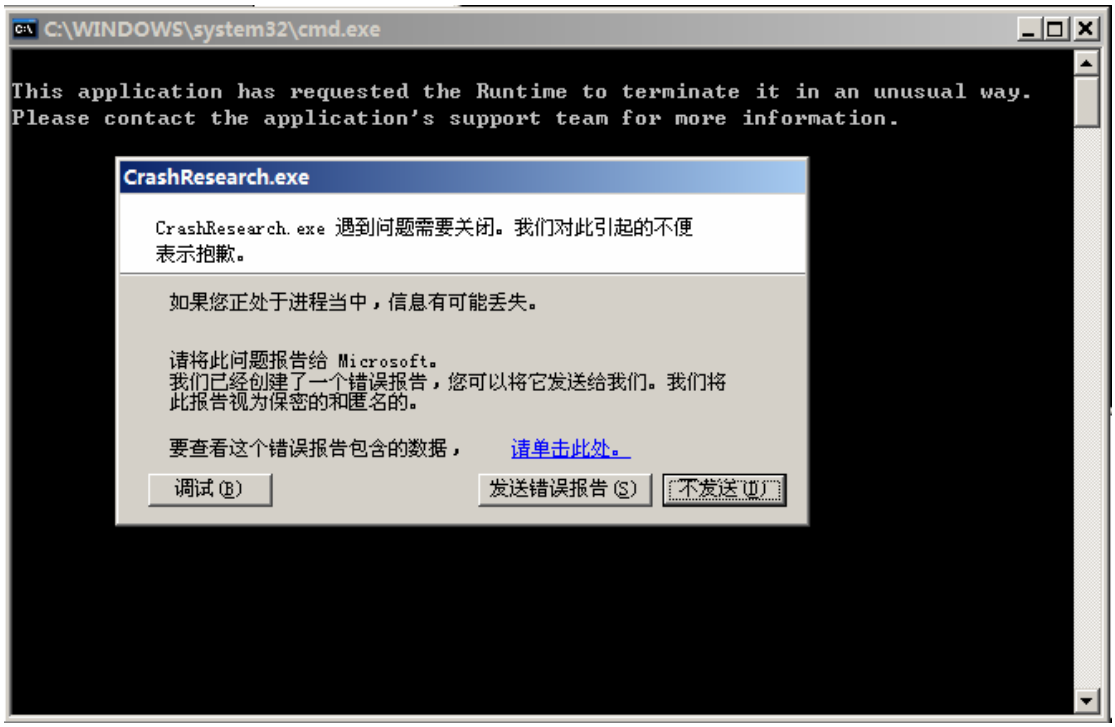
Debug 模式:

Release 模式:

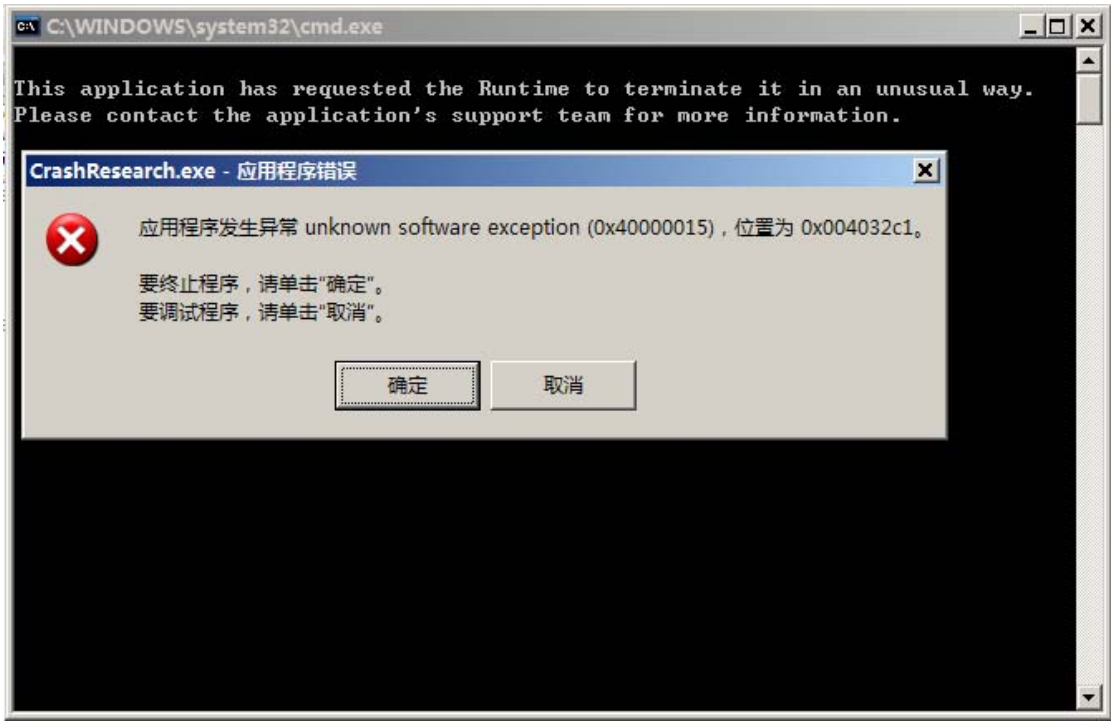


如果在 VS2005 上用 Release 模式编译，在激活了错误报告情况下，情况又会有所不同了:

激活错误报告：



取消错误报告：



VS 2005 和以前版本的实现有差别。以前版本中打印出错信息后 CRT 直接调用 exit 结束进程。从操作系统角度来看，程序就是普通退出，所以不会看到发送错误报告的框框。VS 2005 上除了打印出错信息后，会构造出 0x40000015 的异常，然后把执行定向到 kernel32 的 unhandled exception handler 上。所以从操作系统的角度来看，就像发生了未捕获的异常一样，这样发送错误报告的功能就有作用了。

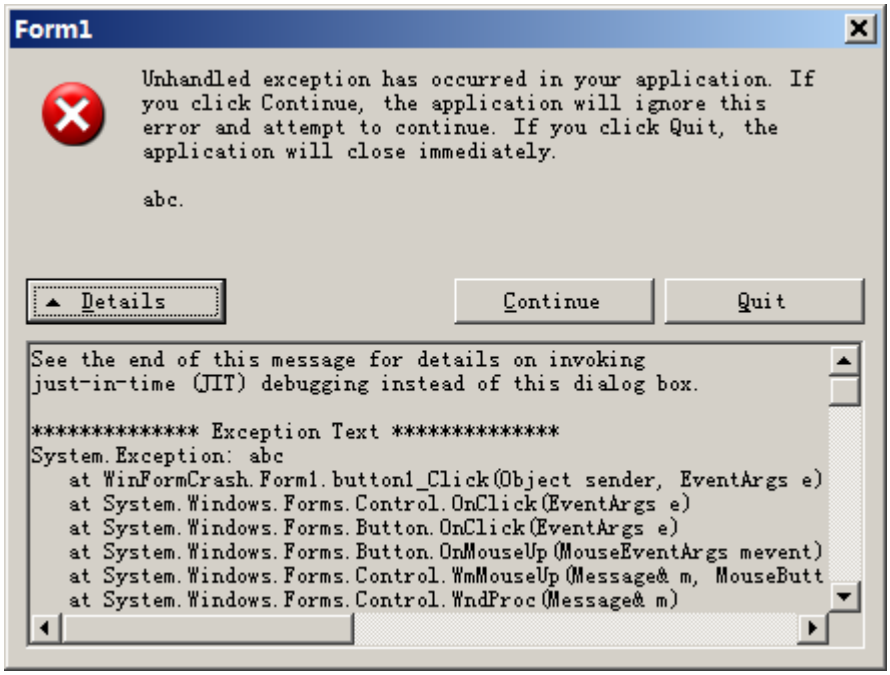
相关信息可以参考:
http://eparg.spaces.live.com/blog/cns!59BFC22C0E7E1A76!1213.entry

出了上面的情况外，eventlog 也会记录程序的死相：

How To Obtain a Userdump When COM+ Failfasts
http://support.microsoft.com/?id=287643
这里介绍了 COM+ Crash 的死相

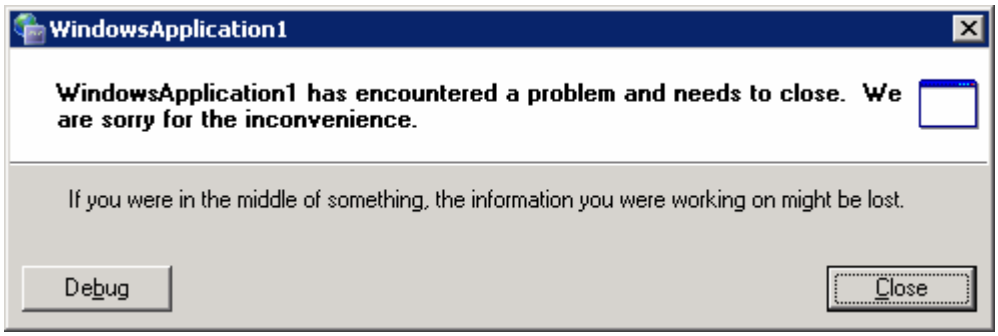
aspnet_wp.exe (PID: %1) stopped unexpectedly.
上面这句话表示 asp.net 工作进程崩溃

对于托管的 WinForm 程序，死相也有两种：



当在程序的配置文件，或者是 macine.config 中使用了如下配置：
<system.windows.forms jitDebugging="true" />

看到的情形又不一样：



所以不同的程序，不同的死因，死相是不一样的。当崩溃发生的时候，抓取 screen-shot 变得必要的。通过死相大致判断程序的类型和死因后，就可以有针对性地来抓取 dump 文件.

Dump 的获取

了解大致的死因后,下一步就是如何抓取崩溃时刻的确切信息。通常 dump 文件是最好的选择。理想的情况是抓取 1st chance exception dump。一般来说可以选择下面一些方式：

Adplus

通过直接执行 adplus 来监视目标进程：

Adplus –crash –p <PID> -quiet –FullOnFirst –o C:\dumps

该命令会监视目标进程中的所有异常，对所有的 first chance exception 保存 dump 文件。该命令在问题能重现的情况下特别有用，因为所以异常都会保存下来。

但如果崩溃无法简单重现，需要用该命令监视比较长的时间，那中间产生的无关信息,比如正常的 C++异常也会导致 dump 的生成，使得 dump 数量巨大，程序性能受到影响。所以,当无法简单重现问题的时候,可以考虑使用：

Adplus –crash –p <PID> -quiet –NoDumpOnFirst –o C:\dumps

这样发生 2nd chance 的时候，dump 才会被保存下来。该命令能够抓到最终导致 crash 的 dump，但是异常发生前的事情就没有记录了。对于某些情况，崩溃前的一些有用线索，比如连续发生 1st chance Access Violation,在使用-NoDumpOnFirst 后就看不到了

一个折中办法是使用：
Adplus –crash –p <PID> -quiet –o C:\dumps

这个命令对 1st chance Access Violation 生成 mini dump，对 2nd chance 生成 full dump。既能够抓到一些 1st chance 的信息，也能够防止无关 dump 数量太多

上面的命令都是使用-p 参数来监视目标进程。如果目标进程刚启动就崩溃，那就没有机会使用-p 参数了。这种情况可以使用-sc 参数来启动目标进程进行监视,比如:

Adplus –crash –quiet –o C:\dumps –sc notepad.exe

如果上面这些命令还不足以满足需求，可以通过 adplus 的配置文件来配置，比如我们可以配置为:

对于 exception 的默认行为是，发生 1st chance 的时候只做 log，然后用 windbg 的 gn 命令继续运行。发生 2nd chance 的时候生成 FullDump 和 log 然后用 q 命令退出。

对于 AccessViolation，发生 1st chance 的时候记录 log 和所有 threads 的 callstack，然后用 gn 继续运行。发生 2nd chance 的时候产生 FullDump 然后用 q 退出。

如果触发 breakpoint exception,无论是 1st chance 还是 2nd chance 都是生成 FullDump。（其实，breakpoint excpetion 不可能有 2nd chance 的对吧）

对应的配置文件内容是:

```
<Exceptions>
  <!--   Configuring all exceptions -->
    <Config>
      <Code> AllExceptions </Code>
      <Actions1> Log</Actions1>
      <Actions2> FullDump;Log </Actions2>
      <ReturnAction1> GN     </ReturnAction1>
      <ReturnAction2> Q      </ReturnAction2>
    </Config>
    <Config>
      <Code> av </Code>
      <Actions1> Log;Stacks</Actions1>
      <Actions2> FullDump;Log </Actions2>
      <ReturnAction1> GN     </ReturnAction1>
      <ReturnAction2> Q      </ReturnAction2>
    </Config>

    <Config>
      <Code> bpe </Code>
      <Actions1> FullDump </Actions1>
      <Actions2> FullDump </Actions2>
      <ReturnAction1> GH     </ReturnAction1>
      <ReturnAction2> Q      </ReturnAction2>

    </Config>
</Exceptions>
```

上面这些，配置文件还可以控制用 minidump 代替 FullDump，某些 exception 发生的时候执行自定义 windbg 命令，还可以设定条件断点来捕获 dump。详细情况参考 windbg 的帮助中关于 ADPlus Configuration Files

无论 adplus 是如何灵活，adplus 固有的缺点是 adplus 命令无法自动执行。如果某一个错误很难发生一次，每次都需要运行 adplus 来监视，这也是一个繁重的工作。为了解决这个麻烦，可以使用 dr Watson:

dr Watson

http://support.microsoft.com/?id=308538

Dr Watson 其实是一个跟 windbg 同样类型的调试器。激活 Dr Watson 后，无论是什么进程，当异常发生导致崩溃的时候，Dr Watson 会自动把 dump 保存下来，所以就不需要用户每次去手动执行命令来进行监视了。

Dr Watson 能够在程序崩溃后抓取 dump 的奥秘在于 AEDebug 这个注册表：

http://support.microsoft.com/?id=103861

前面讨论过，当 unhandled exception 发生，最终会被系统挂接的 unhandled exception handler 捕获。在 handler 里面系统会读取 AEDebug 注册表，根据设定来启动对应的调试器。Dr Watson 其实就是依赖于 AEDebug 注册表启动的。所以当崩溃发生的时候，Dr Watons 就会被系统启动起来，然后根据传入的参数抓取 dump。

根据这个原理，如果想在程序崩溃的时候自动启动 windbg,可以修改注册表用 windbg 来代替 drtwson32。这样程序 crash 后，会自动启动 windbg。可以在注册表中指定 windbg 的命令行来执行自定义的 script 来实现更多功能。

使用这个方法的优点在于用户不需要做额外的操作。但是缺点也很明显，比如对于 unhandled C++ exception，由于在 CRT 的 hanlder 中就直接退出进程，所以 AEDebug 中的设定不起作用。同时这个方法只能抓到 2nd chance dump，而不是 1st chance dump，所以很可能已经错过观察问题的最佳时机。另外,由于 AEDebug 是针对整个系统的，所以无论什么进程发生了崩溃，AEDebug 里面指定的调试器都会启动。无法只针对特定的目标进程调试。

Image File Execution Options

为了解决这个问题，最好的方法是使用前面介绍过的 Image File Execution Options 键。把前面的方法再次摘录在这里：

1. 在客户机器的 Image File Execution Options 注册表下面创建跟问题程序同名的键
2. 在这个键的下面创建 Debugger 字符串类型子键
3. 设定 Debugger= C:\Debuggers\autodump.bat
4. 编辑 C:\Debuggers\autodump.bat 文件的内容为如下:

```
cscript.exe C:\Debuggers\adplus.vbs -crash -o C:\dumps -quiet -sc %1
```

通过上面的设置，当程序启动的时候，系统自动运行 cscript.exe 来执行 adplus.vbs 脚本。Adplus.vbs 脚本的-sc 参数指定需要启动的目标进程路径 (路径作为参数又系统传入，bat 文件中的 1%代表这个参数)，-crash 参数表示监视进程退出，-o 参数指定 dump 文件路径，-quiet 参数取消额外的提示。可以用 notepad.exe 作为小白鼠做一个实验，看看关闭 notepad.exe 的时候，是否有 dump 产生。

这个方法结合了 adplus 的灵活和强大，也不需要用户每次都手动运行 adplus.

COM+/ASP.NET

除了上面的方法外，对于特别的程序，可能需要特别的步骤来获取 dump 文件。比如针对 COM+，需要激活 failfast:

How To Obtain a Userdump When COM+ Failfasts
http://support.microsoft.com/?id=287643

对于 ASP.NET, 往往会通过 recycle 的方法来崩溃。当 IIS 检测到 ASP.NET 的内存使用到某一个阈值，或者长时间没有反应的时候，IIS 会主动让当前 ASP.NET 工作进程退出后重新开始。对于这样的非预期退出，可以参考下面的步骤来设置对应的注册表获取 dump:

http://support.microsoft.com/?id=325947

前面提到的崩溃都是非预期的异常导致的。还有一种情况是程序非预期的正常退出。

换句话说，程序执行了 ExitProcess 退出了，但是这种退出并非程序员的预期的行为。当使用 adplus 抓 dump 的时候，如果程序通过 ExitProcess 退出，在退出前一刻的 dump 也会被 adplus 保存下来的。通过分析 dump 的 callstack，也能够看到 ExitProcess 被触发的原因。当然，也可以通过 adplus 的配置文件在 kernel32!ExitProcess 上设定断点来抓取 dump

分析 crash dump

在获取 dump 文件后，下一步就是分析 dump 来获取尽可能多的信息。抓取 dump 还有模板可循，分析 dump 就不那么简单了。我认为分析 dump 是排错中最有趣，也是最具有挑战性的任务。对于同一套 dump，不同功力的人挖掘出的信息完全不一样。

Crash 都是发生在某一个函数中。95%的 crash 都是下面两种情况的其中一种导致的：

1. 发生 crash 的函数得到了错误的参数
2. 发生 crash 的函数使用了损坏的内部数据

代码的执行过程是对数据进行变化的过程。对于同一段代码，在相同环境下，如果使用到的数据都相同，执行的结果肯定是唯一的。如果函数发生崩溃，肯定是使用到的数据跟理想情况有差别。函数使用到的数据一是来源于传入的函数参数，二是来源于函数体引用到的成员变量或者全局变量。所以分析 crash dump，大多数情况都是寻找错误数据的来源。下面是一些常见的，导致数据错误的例子：

1. 使用了未初始化的变量。比如没有分配内存的指针，没有初始化的 CriticalSection。
2. 错误地计算了函数参数。比如调用函数的时候弄错了传入参数的顺序，字符串操作时算错了字符串长度
3. 对数据错误地使用导致数据发生 corruption。比如 double free 导致 heap corruption，多线程环境下忘记同步导致全局变量计算错误，COM 的 AddRef 和 Release 调用不配对。
4. 违背程序的逻辑使用数据。比如在程序加载必要的资源以前就开始使用以来这些资源的函数。

由此可以见，crash dump 的分析完全取决于程序的情况。能否从 crash dump 中挖掘出有用的信息，取决于：

1. 对目标程序的熟悉程度。包括程序的架构，重要函数的作用，重要的数据结构，函数之间的调用逻辑，关键函数的实现细节。
2. 对基础知识的掌握程度，包括汇编，异常，内存，API，消息，CRT 等等。

一般来说，我对一个 crash dump 是通过下面的步骤入手：

1. 看清楚是何种异常导致的崩溃。
2. 对齐 symbol，找到发生崩溃的函数名字，以及对应的汇编代码和高级语言源代码。
3. 列出 callstack
4. 检查 callstack 是否合理
5. 检查发生崩溃的函数是否得到了正确的参数
6. 检查发生崩溃的函数使用的数据是否正确
7. 结合上面的信息，构想来龙去脉，然后用数据来证明，或者反驳自己的猜想。
8. 通过进一步的操作来获取更有意义的数据。比如激活 pageheap 后重新抓取 dump,或者干脆进行 live debug

下面是一个比较典型的例子：

[案例分析,VC 程序的崩溃]

问题背景

客户用 VC 开发的客户端 WinForm 程序会随机崩溃。该程序使用 MFC 多线程开发，进行数据库操作和网络操作。客户用 Dr. Watson 抓取到了 2nd chance 的 crash dump。同时收集了所有的 PDB 文件和源代码。

打开 dump 后看到:

```
(10e0.10e4): Access violation - code c0000005 (!!! second chance !!!)
eax=00000000 ebx=020d5160 ecx=00000000 edx=00000000 esi=00000000 edi=00000000
eip=00b60cf0 esp=0012e288 ebp=021050c8 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
*** WARNING: Unable to verify checksum for CUSTOMERFactory.dll
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for CUSTOMERFactory.dll -
CUSTOMERFactory!CClientMessage::isError:
00b60cf0 8b510c          mov     edx,dword ptr [ecx+0Ch] ds:0023:0000000c=????????
```

非常明显地看到 crash 是由于 access violation 导致的。问题发生在地址 `00b60cf0` 上的一个

mov 指令。该地址对应到的 module 是 CUSTOMERFactory 。检查一下 CUSTOMERFactory 的信息:

```
0:000> lmvm CUSTOMERFactory
start    end        module name
00b50000 00f56000   CUSTOMERFactory C (export symbols)      CUSTOMERFactory.dll
    Loaded symbol image file: CUSTOMERFactory.dll
    Image path: W:\Program Files\CuExE\Program Files\CUSTOMERFactory.dll
    Image name: CUSTOMERFactory.dll
    Timestamp:        Wed May 17 08:52:46 2006 (446A73DE)
    CheckSum:         00000000
    ImageSize:        00406000
    File version:     1.0.0.2
    Product version:  1.0.0.2
```

该 module 是客户开发的 module,设定好对应的 symbol path 后,用 kb100 显示详细的 callstack:

Callstack

```
0:000> kb 100
ChildEBP RetAddr  Args to Child
0012e284 0046d4ed 021050c8 055232c8 0012e348 CUSTOMERFactory!CClientMessage::isError
[c:\Customer_Code_Folder\CUSTOMERfactory\clientmessage.cpp @ 179]
0012e2b4 7c16fca0 055a0fd0 055232c8 021050c8 CuExE!CMainFrame::OnBrokerOutQMsg+0x9d
[c:\Customer_Code_Folder\CuExE\mainfrm.cpp @ 588]
0012e348 7c16e0b0 0000052c 055a0fd0 007eb350 MFC71!CWnd::OnWndMsg+0x4f2
0012e368 7c16e14f 0000052c 055a0fd0 055232c8 MFC71!CWnd::WindowProc+0x22
0012e3c8 7c16e1b8 00000000 00030024 0000052c MFC71!AfxCallWndProc+0x91
0012e3e8 7c16e1f6 00030024 0000052c 055a0fd0 MFC71!AfxWndProc+0x46
0012e414 77e3a3d0 00030024 0000052c 055a0fd0 MFC71!AfxWndProcBase+0x39
0012e434 77e14605 7c16e1bd 00030024 0000052c USER32!UserCallWinProc+0x18
0012e4c0 77e1a7ba 0012e4e4 00000000 77e334f2 USER32!DispatchMessageWorker+0x2e4
0012e4cc 77e334f2 0012e4e4 00010366 00000000 USER32!DispatchMessageW+0xb
0012e504 77e44184 0001036e 00010366 00000001 USER32!DialogBox2+0x164
0012e528 77e340b0 77e10000 00191400 00000000 USER32!InternalDialogBox+0xd1
0012e7e0 77e3444c 00000030 0012e980 ffffffff USER32!SoftModalMessageBox+0x757
0012e928 77e3393f 00000001 00000000 00000028 USER32!MessageBoxWorker+0x247
0012e980 77e341d7 00010366 00194228 001385e0 USER32!MessageBoxExW+0x77
0012e9b0 77e33370 00010366 053e2c00 01f07440 USER32!MessageBoxExA+0xa0
0012e9d0 7c1dcd91 00010366 053e2c00 01f07440 USER32!MessageBoxA+0x49
0012eb0c 0046ffdb 053e2c00 00000030 00000000 MFC71!CWinApp::DoMessageBox+0xf4
0012eb64 7c16fca0 00000001 053e2c00 021050c8 CuExE!CMainFrame::OnServerConnectionStatusMsg+0x32b
[c:\Customer_Code_Folder\CuExE\mainfrm.cpp @ 861]
0012ebf8 7c16e0b0 0000054b 00000001 007eb428 MFC71!CWnd::OnWndMsg+0x4f2
```

```
0012ec18 7c16e14f 0000054b 00000001 00000001 MFC71!CWnd::WindowProc+0x22
0012ec78 7c16e1b8 00000000 00030024 0000054b MFC71!AfxCallWndProc+0x91
0012ec98 7c16e1f6 00030024 0000054b 00000001 MFC71!AfxWndProc+0x46
0012ecc4 77e3a3d0 00030024 0000054b 00000001 MFC71!AfxWndProcBase+0x39
0012ece4 77e14605 7c16e1bd 00030024 0000054b USER32!UserCallWinProc+0x18
0012ed70 77e1a7ba 0012ed94 00000000 77e334f2 USER32!DispatchMessageWorker+0x2e4
0012ed7c 77e334f2 0012ed94 00070234 00000000 USER32!DispatchMessageW+0xb
0012edb4 77e44184 00010366 00070234 00000001 USER32!DialogBox2+0x164
0012edd8 77e340b0 77e10000 00166470 00000000 USER32!InternalDialogBox+0xd1
0012f090 77e3444c 00000040 0012f230 ffffffff USER32!SoftModalMessageBox+0x757
0012f1d8 77e3393f 00000001 00000000 00000028 USER32!MessageBoxWorker+0x247
0012f230 77e341d7 00070234 0723cca8 001a3570 USER32!MessageBoxExW+0x77
0012f260 77e33370 00070234 04f7ac10 01f07440 USER32!MessageBoxExA+0xa0
0012f280 7c1dcd91 00070234 04f7ac10 01f07440 USER32!MessageBoxA+0x49
0012f3bc 0046ffdb 04f7ac10 00000040 00000000 MFC71!CWinApp::DoMessageBox+0xf4
0012f414 7c16fca0 00000000 04f7ac10 021050c8 CuExE!CMainFrame::OnServerConnectionStatusMsg+0x32b
[c:\Customer_Code_Folder\CuExE\mainfrm.cpp @ 861]
0012f4a8 7c16e0b0 0000054b 00000000 007eb428 MFC71!CWnd::OnWndMsg+0x4f2
0012f4c8 7c16e14f 0000054b 00000000 00000001 MFC71!CWnd::WindowProc+0x22
0012f528 7c16e1b8 00000000 00030024 0000054b MFC71!AfxCallWndProc+0x91
0012f548 7c16e1f6 00030024 0000054b 00000000 MFC71!AfxWndProc+0x46
0012f574 77e3a3d0 00030024 0000054b 00000000 MFC71!AfxWndProcBase+0x39
0012f594 77e14605 7c16e1bd 00030024 0000054b USER32!UserCallWinProc+0x18
0012f620 77e1a7ba 0012f644 00000000 77e334f2 USER32!DispatchMessageWorker+0x2e4
0012f62c 77e334f2 0012f644 00030024 00000000 USER32!DispatchMessageW+0xb
0012f664 77e44184 00070234 00030024 00000001 USER32!DialogBox2+0x164
0012f688 77e340b0 77e10000 001a7768 00000000 USER32!InternalDialogBox+0xd1
0012f940 77e3444c 00000030 0012fae0 ffffffff USER32!SoftModalMessageBox+0x757
0012fa88 77e3393f 00000001 00000000 00000028 USER32!MessageBoxWorker+0x247
0012fae0 77e341d7 00030024 001b7bf0 0723ca18 USER32!MessageBoxExW+0x77
0012fb10 77e33370 00030024 0552b180 01f07440 USER32!MessageBoxExA+0xa0
0012fb30 7c1dcd91 00030024 0552b180 01f07440 USER32!MessageBoxA+0x49
0012fc6c 0046ffdb 0552b180 00000030 00000000 MFC71!CWinApp::DoMessageBox+0xf4
0012fcc4 7c16fca0 00000001 0552b180 021050c8 CuExE!CMainFrame::OnServerConnectionStatusMsg+0x32b
[c:\Customer_Code_Folder\CuExE\mainfrm.cpp @ 861]
0012fd58 7c16e0b0 0000054b 00000001 007eb428 MFC71!CWnd::OnWndMsg+0x4f2
0012fd78 7c16e14f 0000054b 00000001 00000001 MFC71!CWnd::WindowProc+0x22
0012fdd8 7c16e1b8 00000000 00030024 0000054b MFC71!AfxCallWndProc+0x91
0012fdf8 7c16e1f6 00030024 0000054b 00000001 MFC71!AfxWndProc+0x46
0012fe24 77e3a3d0 00030024 0000054b 00000001 MFC71!AfxWndProcBase+0x39
0012fe44 77e14605 7c16e1bd 00030024 0000054b USER32!UserCallWinProc+0x18
0012fed0 77e15b77 0013aff0 00000001 7c169076 USER32!DispatchMessageWorker+0x2e4
0012fedc 7c169076 0013aff0 0013aff0 00890a60 USER32!DispatchMessageA+0xb
0012feec 7c16913e 00890a60 00890a60 0012ffc0 MFC71!AfxInternalPumpMessage+0x3e
0012ff08 7c172fc5 7c590ace 001351c2 00000000 MFC71!CWinThread::Run+0x54
```

```
0012ff18 004f9c0d 00400000 00000000 001351c2 MFC71!AfxWinMain+0x68
0012ffc0 7c59893d 00135cd0 00134b74 7ffdf000 CuExE!WinMainCRTStartup+0x185
[f:\vs70builds\3077\vc\crtbld\crt\src\crtexe.c @ 390]
0012fff0 00000000 004f9a88 00000000 000000c8 KERNEL32!BaseProcessStart+0x3d
```

第一个问题是 callstack 是否合理？

从 `MFC71!AfxWndProcBase` 可以得知该线程是 MFC 窗口的主线程，正在消息循环中。异样的地方是很多函数在通过消息循环嵌套调用。这里出现了三个 MessageBox 和 OnServerConnectionStatusMsg 函数的嵌套调用。最终调用到 IsError 函数上发生崩溃。

虽然说我并不是了解程序的详细设计，但是正常情况下很少会发生 MessageBox 嵌套调用的情况。一般来说，当某些事件发生，需要通过 MessageBox 通知用户的时候，程序逻辑会停止。当用户点击按钮取消了 MessageBox 后，程序再继续执行，否则程序的 callstack 应该一直停留在 MessageBox 的调用上。

从当前的 callstack 上看，情况完全不是这样。当第一个 MessageBox 显示出来后，显然程序的逻辑并没有停止。程序仍然受到了新的消息。由于 MessageBox 内部实现是维护消息循环的，于是新的消息被派发，然后对应的 OnServerConnectionStatusMsg 函数被执行。该函数再次调用了 MessageBox 来显示什么信息。如此循环，使得 OnServerConnectionStatusMsg 和 MessageBox 都被调用了三次。

为了进一步弄清楚这个问题，看看每次 MessageBox 都在显示些什么。由于 MessageBox 第二个参数传入需要显示的字符串，于是用 da 命令检查这几个参数:

```
0:000> da 0552b180
0552b180  "Unable to connect to TOPMC XXX s"
0552b1a0  "erver. I will try connecting eve"
0552b1c0  "ry 60 seconds. If the connection"
0552b1e0  " is restored, you will be notifi"
0552b200  "ed."
0:000> da 04f7ac10
04f7ac10  "The TOPMC server connection is n"
04f7ac30  "ow OK."
0:000> da 053e2c00
053e2c00  "Unable to connect to TOPMC XXX s"
053e2c20  "erver. I will try connecting eve"
053e2c40  "ry 60 seconds. If the connection"
053e2c60  " is restored, you will be notifi"
053e2c80  "ed."
```

很奇怪。先说连接有问题，在客户取消这个 MessageBox 以前，又说连接恢复。然后又说连接有问题。很显然，这样的行为是不合常理的，而且问题似乎跟连接状态相关。

根据上面的信息，很怀疑问题跟代码设计相关。所以下一步就是向开发人员联系，弄清楚 MessageBox 的嵌套是否是预期的行为，在弹出 MessageBox 的时候，是不是应该把整个程序挂起来，等用户取消了 MessageBox 后再执行。发生问题的时候是否有连接丢失的现象等等。总之，通过简单的 kb 命令，我们就有了明确的怀疑对象和进一步的排错步骤。

我相信聪明的开发人员拿到上面的信息后，应该很快可以定位到该问题。但是从调试人员的角度来说，既然已经拿到了客户的代码，看看我们是否可以进一步发挥来获取更多的信息。至少我们还没有看到 MessageBox 的嵌套调用跟 Access Violation 有什么直接联系。

源代码

既然崩溃发生在 isError 函数中，而 isError 是 OnBrokerOutQMsg 调用的，那找到这两个函数,简化变型后的源代码如下：

```
BOOL CClientMessage::isError() const
{
    return (m_pErrors != NULL);
}

LRESULT CMainFrame::OnBrokerOutQMsg(WPARAM wParam, LPARAM lParam)
{
...
while (::PeekMessage(&WinMsg, m_hWnd, WM_XXX_MSGBROKER_OUTQ_MSG,
WM_XXX_MSGBROKER_OUTQ_MSG, PM_REMOVE))
    {
        pMsg     = (CClientMessage*)WinMsg.wParam;
        pMsgHndl = (CMessageHandler*)WinMsg.lParam;

        ASSERT(pMsg && pMsgHndl);

        // same check again
        if (pMsgHndlFact->handlerRemoved(pMsgHndl))
        {
            // the data is of no interest any longer
            delete pMsg;
            pMsg = NULL;
        }
        else
         {
              // errors can also be received here
              if (pMsg->isError() != XXX_OK)
                   pMsgHndl->processMessageFail(pMsg);
```

```
                else
                    pMsgHndl->receiveMessage(pMsg);
            }
        }
```

isError 的逻辑很简单，检查一个成员变量 n_pErrors。从发生问题的汇编上看，正是在把该成员变量读入 CPU 寄存器的过程中发生了崩溃，显然问题是该成员变量发生数据损坏导致的。

对吗？

不对！注意，成员变量发生数据损坏是指该成员变量本来应该是 A,结果现在变成了 B. 但是现在的问题是崩溃发生在读这个成员变量的时候，还没有到使用这个成员变量的时候。所以，并不是这个成员变量发生了损坏，而是保存这个成员变量的变量，发生了损坏。谁保存了成员变量呢？this 指针！

看看 this 指针是什么。首先看导致问题的汇编:

```
0:000> u eip
CUSTOMERFactory!CClientMessage::isError [c:\Customer_Code_Folder\CUSTOMERfactory\clientmessage.cpp @
179]:
00b60cf0 8b510c          mov     edx,dword ptr [ecx+0Ch]
```

这里通过 ecx+offset 的方法来间接寻址,说明 this 指针是保存在 ecx 寄存器上的。0ch 是成员变量的便宜，ecx 是:

```
0:000> r ecx
ecx=00000000
```

原来 this 指针是一个空指针!

重新检查这两个函数，逻辑是这样的。首先通过 PeekMessage 在 MainFrame 窗口上接获自定义的消息。自定义消息的 wParam 参数应该是一个 CClientMessage 类型指针。于是拿到参数后强制转型。接下来进行一些判断后，对这个 object 调用了 isError 成员函数，由于这个 object 是 null,于是哐当，崩了。

注意看代码，这里用了

```
ASSERT(pMsg && pMsgHndl);
```

来确保这两个指针非空。但是由于 ASSERT 只有在 debug 版本中才有作用。所以，这里给我们提示了排错的另外一个思路，使用 debug 版本来重现问题，这里的 Assert 就会触发。

This 指针

继续看下去，发生问题的 this 指针是 CClientMessage 类型，指针的值来源于 PeekMessage 得到的 wParam 参数。那好，为什么 wParam 是 NULL 呢？

wParam 是通过 Message 传递过来的。最有可能的情况是发送者发送了错误的数据。根据前面的分析，程序的连接发生了一些问题，所以跟连接的情况相关？

由于不了解程序的整体设计，所以也清楚该消息会在什么时间发送过来，wParam 的数据又是如何生成的。总之呢，问题很可能在 isError 和 OnBrokerOutQMsg 这两个函数之外。可选的调试方法是在 SendMessage/PostMessage API 上设定条件断点，当发送对应自定义消息，但是 wParam 为 0 的时候停下来检查。

看来我们分析越仔细，思路就越广。继续挑战自己的潜力吧,看看还有什么蛛丝马迹：

切换到 OnBrokerOutQMsg 函数看看从中间变量：

```
0:000> .frame 1
01 0012e2b4 7c16fca0 CuExE!CMainFrame::OnBrokerOutQMsg+0x9d
[c:\Customer_Code_Folder\CuExE\mainfrm.cpp @ 588]
0:000> x
@edx this = 0x00000000
021050cc wParam = 1
021050d0 lParam = 0
021050ac WinMsg = struct tagMSG
0:000> dt WinMsg
Local var @ 0x21050ac Type tagMSG
   +0x000 hwnd           : (null)
   +0x004 message        : 0
   +0x008 wParam         : 0
   +0x00c lParam         : 0
   +0x010 time           : 0x33ff
   +0x014 pt             : tagPOINT
```

这里的 WinMsg 是 PeekMessage 函数中由操作系统回填的，hwnd 变量应该跟传入 PeekMessage 的第二个参数一致。这里 hwnd 为 0，说明 PeekMessage 第二个参数也为 0，那就是说 CMainFrame 的 m_hWnd 为 0 了？

CMainFrame 是 MFC 程序必须的数据结构，CMainFrame 的 m_hWnd 在程序的整个生命周期内都应该不为 0 的。为什么会这样？

为了进一步检查CMainFrame上m_hWnd的值，应该把CMainFrame的this指针找出来。但从上面的log看，`@edx this = 0x00000000` this指针好像也是0,难道CMainFrame的this指针也被破坏了？

理智一点，CMainFrame的this指针为0是不可能的，否则程序在调用PeekMessage以前就已经崩掉了。这里很可能是因为编译器优化导致的。看看编译器对OnBrokerOutQMsg 函数是否有什

么优化:

```
0:000> kvn
 # ChildEBP RetAddr  Args to Child
00 0012e284 0046d4ed 021050c8 055232c8 0012e348 CUSTOMERFactory!CClientMessage::isError (FPO: [0,0,0])
(CONV: thiscall) [c:\Customer_Code_Folder\CUSTOMERfactory\clientmessage.cpp @ 179]
01 0012e2b4 7c16fca0 055a0fd0 055232c8 021050c8 CuExE!CMainFrame::OnBrokerOutQMsg+0x9d (FPO: [Uses EBP]
[2,7,0]) (CONV: thiscall) [c:\Customer_Code_Folder\CuExE\mainfrm.cpp @ 588]
02 0012e348 7c16e0b0 0000052c 055a0fd0 007eb350 MFC71!CWnd::OnWndMsg+0x4f2 (FPO: [Non-Fpo])
```

这里果然用了 FPO 优化。正常情况下，函数调用需要创建新的 stack frame,函数返回的时候恢复 stack frame。EBP 寄存器是用来保存 callee 函数 stack 起始地址的。当 FPO 优化激活后，编译器可以优化掉 stack frame 的创建和恢复，把 callee 的 EBP 寄存器节约下来保存一些中间变量。详细请参考:

/Oy (Frame-Pointer Omission)
http://msdn2.microsoft.com/en-us/library/2kxx5t2c.aspx

How can a program survive a corrupted stack?
http://blogs.msdn.com/oldnewthing/archive/2004/01/16/59415.aspx

所以，这里的 this 指针可能是保存在 EBP 寄存器上的,所以用 dt 直接看 this 看不到准确的信息。看看汇编来研究下:

```
0:000> uf CuExE!CMainFrame::OnBrokerOutQMsg
CuExE!CMainFrame::OnBrokerOutQMsg [c:\Customer_Code_Folder\CuExE\mainfrm.cpp @ 541]:
  541 0046d450 83ec1c         sub     esp,1Ch
  541 0046d453 53             push    ebx
  541 0046d454 55             push    ebp
  541 0046d455 56             push    esi
  541 0046d456 57             push    edi
  541 0046d457 8be9           mov     ebp,ecx
  552 0046d459 ff1524937d00    call    dword ptr [CuExE!_imp_?instanceCMessageHandlerFactorySSAPAV1XZ
(007d9324)]
  。。。。。。

CuExE!CMainFrame::OnBrokerOutQMsg+0x46 [c:\Customer_Code_Folder\CuExE\mainfrm.cpp @ 563]:
  563 0046d496 8b16           mov     edx,dword ptr [esi]
  563 0046d498 ff5220         call    dword ptr [edx+20h]
  564 0046d49b eb05           jmp     CuExE!CMainFrame::OnBrokerOutQMsg+0x52 (0046d4a2)

CuExE!CMainFrame::OnBrokerOutQMsg+0x4d [c:\Customer_Code_Folder\CuExE\mainfrm.cpp @ 565]:
  565 0046d49d 8b06           mov     eax,dword ptr [esi]
  565 0046d49f ff501c         call    dword ptr [eax+1Ch]
```

```
CuExE!CMainFrame::OnBrokerOutQMsg+0x52 [c:\Customer_Code_Folder\CuExE\mainfrm.cpp @ 571]:
  571 0046d4a2 8b4d20          mov     ecx,dword ptr [ebp+20h]
  571 0046d4a5 6a01            push    1
  571 0046d4a7 682c050000      push    52Ch
  571 0046d4ac 682c050000      push    52Ch
  571 0046d4b1 51              push    ecx
  571 0046d4b2 8d542420        lea     edx,[esp+20h]
  571 0046d4b6 52              push    edx
  571 0046d4b7 ff1508b47d00    call    dword ptr [CuExE!_imp__PeekMessageA (007db408)]
  571 0046d4bd 85c0            test    eax,eax
  571 0046d4bf 745e            je      CuExE!CMainFrame::OnBrokerOutQMsg+0xcf (0046d51f)
。。。。。。
```

果然，函数已开始就把 ecx 的值保存到 EBP 中。在调用 PeekMessage 以前，传入的第二个参数 m_hWnd 来自 ecx,而 ecx 来自 ebp+20h。因为 m_hWnd 保存在 this 指针的一个偏移量上，所以这就可以确认 EBP 就是 CMainFrame 的 this 指针。继续检查汇编发现，在调用 PeekMessage 到发生错误中间的汇编代码中，EBP 都没有被修改过，所以直接检查 EBP 就可以看到 CMainFrame 的详细信息:

```
0:000> r ebp
ebp=021050c8
0:000> dt 021050c8 CMainFrame
   +0x000 __VFN_table : 0x7c164734
   =00400000 classCObject     : CRuntimeClass
   =00400000 classCCmdTarget  : CRuntimeClass
   =00400000 _messageEntries  : [0] AFX_MSGMAP_ENTRY
   =00400000 messageMap       : AFX_MSGMAP
   =00400000 _commandEntries  : [0] AFX_OLECMDMAP_ENTRY
   =00400000 commandMap       : AFX_OLECMDMAP
   =00400000 _dispatchEntries : [0] AFX_DISPMAP_ENTRY
   =00400000 _dispatchEntryCount : 0x905a4d
   =00400000 _dwStockPropMask : 0x905a4d
   =00400000 dispatchMap      : AFX_DISPMAP
   =00400000 _connectionEntries : [0] AFX_CONNECTIONMAP_ENTRY
   =00400000 connectionMap    : AFX_CONNECTIONMAP
   =00400000 _interfaceEntries : [0] AFX_INTERFACEMAP_ENTRY
   =00400000 interfaceMap     : AFX_INTERFACEMAP
   =00400000 _eventsinkEntries : [0] AFX_EVENTSINKMAP_ENTRY
   =00400000 _eventsinkEntryCount : 0x905a4d
   =00400000 eventsinkMap     : AFX_EVENTSINKMAP
   +0x004 m_dwRef         : 1
   +0x008 m_pOuterUnknown  : (null)
   +0x00c m_xInnerUnknown  : 0
   +0x010 m_xDispatch      : CCmdTarget::XDispatch
   +0x014 m_bResultExpected : 1
```

```
   +0x018 m_xConnPtContainer : CCmdTarget::XConnPtContainer
   +0x01c m_pModuleState   : (null)
   =00400000 classCWnd       : CRuntimeClass
   +0x020 m_hWnd          : (null)
。。。。。。
```

果然 MainFrame 的 Window handle 为 0。

为了进一步确保 EBP 就是 CMainFrame,看看 VTable 是否指向 MFC 上对应的虚函数:

```
0:000> dds 0x7c164734
7c164734  7c157964 MFC71!CWnd::GetRuntimeClass
7c164738  7c157291 MFC71!CWnd::`vector deleting destructor'
7c16473c  7c148104 MFC71!CHtmlView::OnDraw
7c164740  7c19c621 MFC71!CRecordset::PreBindFields
7c164744  7c148104 MFC71!CHtmlView::OnDraw
7c164748  7c171859 MFC71!CCmdTarget::OnCmdMsg
```

对了，果然没错。

由此看来，问题并不是消息发送方传递了错误的参数，而是接受消息的时候传入了错误的参数。由于 CMainFrame 的 m_hwnd 为 0，导致 PeekMessage 接受当前 thread 的任何消息而不区分目标窗口和消息类型，所以 wParam 参数就不对了。

结论

由此看来，根源在于为何 CMainFrame 的 m_hwnd 为 0。问题很可能是因为程序处于正在退出的过程中。要继续追踪这个问题，可以针对 stack 上的参数和一些成员变量仔细分析，或者在 m_hwnd 上设定条件断点调试。

当时我并没有继续，因为前面的分析已经提供了足够的信息。由于我并不了解程序的设计详情，所以我无法决定下一步如何做最有效率。我把这些信息提供给开发人员后，两天后开发人员直接告诉我找到了对应的 bug,问题解决。

由此可见，dump 分析的效果取决于分析人员对各方面细节的了解。了解得越多，思考得越多，能够看到的东西也就越多。

[更多的案例]

在网上可以找到很多分析异常和崩溃的案例。下面几个例子:

.NET Crash: Managed Heap Corruption calling unmanaged code
http://blogs.msdn.com/tess/archive/2006/02/09/528591.aspx

Are you aware that you have thrown over 40,000 exceptions in the last 3 hours?
http://blogs.msdn.com/tess/archive/2005/11/30/498297.aspx

VS2003 在 push edi 的时候 AV
http://eparg.spaces.live.com/blog/cns!59BFC22C0E7E1A76!379.entry

本章小结

本章首先介绍了程序崩溃的原因，崩溃的现象。然后介绍了抓取 crash dump 的多种方法。最后通过一个典型的例子介绍了 crash dump 的分析步骤。解决崩溃问题的关键在于对程序和系统的理解。深入的理解对信息的获取和分析都是有极大的帮助的。

题外话和相关讨论

下面是一些常见的 crash 和排错的经验：

Unhanded C++/CLR Exception

如果导致 crash 的异常是 C++或者 CLR 异常，可以通过 adplus 的 FullOnFirst 参数，或者通过 config 文件来抓取异常发生时候的 dump。然后：

1. 检查 callstack 找到抛出异常的函数
2. 根据 callstack 和变量信息判断抛出异常是否是预期的行为
3. 检查是否有对应的 catch 函数来处理异常

HeapCorruption

如果 crash 发生在 Heap 相关的 API 中，问题一般是由 HeapCorruption 导致的。根据前面的讨论，应该立刻使用 pageheap，并且使用-full 参数，然后重现问题，使得错误在第一时间暴露出来。

StackCorruption

我最怕调试 StackCorruption 的问题

StackCorruption 一般是由于 stack buffer overrun 导致的。例子已经在第二章演示过了。StackCorruption 损坏了 stack frame，导致 callee 返回的时候把 EIP 指向了错误的地址，从而发生 Crash. 如果在 dump 中看到下面几种情况，往往是发生了 StackCorruption:

1. EIP 指向非法内存，或者指向 unknown module
2. EBP 和 ESP 同时指向非法内存
3. 使用 k 命令无法列出 callstack，比如：

```
0:008> kb
ChildEBP RetAddr Args to Child
WARNING: Frame IP not in any known module. Following frames may be wrong.
016ed96c 00000000 6117a780 00000001 016ed9a8 0x29a33e0
```

由于函数运行的关键信息是保存在 stack 上的， StackCorruption 的毁尸灭迹后行为使得这类问题的调试异常困难。往往 Dump 文件中很难看到确切的信息。如果 ESP/EBP 寄存器的指还幸存下来，可以通过 dds 命令尽量地找一些线索，看看 StackCorruption 发生前大概执行过哪些函数。对于一些可疑的函数可以查看下源代码是否使用了局部数组变量或者是局部字符串。

如果 StackCorruption 是由于 buffer overrun 导致的，那损坏的寄存器上往往是一些相关的数据。可以仔细观察看看有没有什么相关线索，比如检查一下对应的 ASCII 值看看是否有意义。

在 VS 2003 上，为了帮助 StackCorruption 的排错，C++编译器引入了/GS 参数。该参数能够很大程度上在运行时检测 StackCorrupton。所以一旦发生 StackCorruption 后，确保使用/GS 参数编译，然后再次重现问题看看是否有运行时的 Assertion.

Compiler Security Checks In Depth
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dv_vstechart/html/vctchcompilersecuritychecksindepth.asp

如果这些都还不足够解决问题，只有使用返朴归真的 log 机制,让程序在牺牲前留下凶手的记号。

当然，把 stack 搞得乱七八砸的除了 StackCorruption，还有一些其它情况的。比如：

在家用 Windbg 杀小强
http://eparg.spaces.live.com/blog/cns!59BFC22C0E7E1A76!1475.entry

4.3 性能

性能是指程序的处理效率无法达到预期值。

导致性能的原因总的分为两种，外部原因和内部原因。内部因素是指程序代码本身有问题，无法高效地利用资源来完成计算。外部原因是指程序本身不可控因素，比如硬件配置和程序的压力。解决性能问题在于把瓶颈找出来，然后消灭瓶颈。

小心无尽的黑夜

性能排错跟崩溃排错有很大的差别。崩溃的现象很明显，目标也很明确，程序要么崩溃，要么正常。而性能问题则不是可以轻松测量出来的。请看下面一个例子：

客户觉得他的 ASP.NET 网站不时发生性能问题，有时候某些页面需要接近 1 分钟才能够打开。经过一些排查，通过 log 发现有的时候 Request_Begin 和 Page_Load 之间需要大概 20 秒的执行时间。客户问为何这两个函数之间有时候需要这么长时间，如何提高整个 ASP.NET 网站的性能。

这是一个非常典型的客户请求。但如果按照客户的思路走下去，将会进入无尽的黑夜，因为面对的是：

1. 你可能的确认为 Request_Begin 和 Page_Load 之间需要大概 20 秒不太正常，于是你决定先检查这个问题。找到这个问题可能就需要很多功夫
2. 解决了 Request_Begin 和 Page_Load 之间需要大概 20 秒的问题后，你会发现有时候某些页面还是需要接近 1 分钟才能够打开。然后根据 log 又开始怀疑其它的函数，然后重复步骤 1。
3. 经过一切努力后，页面的性能提高了。某些页面已经不需要 1 分钟了，最多 30 秒就可以打开，但是客户还是想知道为什么需要 30 秒
4. 努力啊努力，终于把 30 秒缩短到了 15 秒。客户说，你真牛，不如你再努力给我缩短 10 秒吧
5. 你已经快崩溃了，但是凭借你坚强的毅力和决心，终于让所有请求都在 5 秒内完成了。客户也很满意，于是客户决定把这个应用向更多客户开放，应用的负载立刻提高了一倍。客户发现怎么响应时间又回升到 15 秒了，然后让你再想办法缩短。
6. 你终于崩溃了，同时你发现处理崩溃问题比性能问题来得容易。

为了防止这类事情发生，通常我会问客户：

1. 为什么接近 1 分钟才能打开就是不正常的？根据你的设计，所有请求应该在多少时间内返回才算合理？平均返回时间是多少才合理？
2. 没有文档说明所有情况下 Request_Begin 和 Page_Load 之间应该在 20 秒内完成。比如 GC 发生的时候所有执行都回被挂起，所以某些函数之间停滞 20 秒是完全正常的事情，

我并不认为这个问题跟接近 1 分钟才能打开页面有直接联系。如果疑问是整体的性能问题，我建议先不考虑 Request_Begin 和 Page_Load 之间的细节

3. 问题发生时候的外部环境是什么？程序的负载是不是理想范围内？有没有第三方程序干扰？
4. 有没有具体的数据来说明到底有多少请求超过了 1 分钟？这些请求具体的返回时间多少？
5. 你的期望值是多少？这个期望值是如何计算出来的？
6. 你设计的是一个实时系统吗？如果是，可能 Windows 和 ASP.NET 并不是一个好的平台。如果不是，某些个别请求无法在期望时间内返回是无法避免的。你能接受这样的事实吗？

上面这些问题是着手处理性能问题的前提。正确的思路和原则才能保证高效的排错步骤。如果不区分整体和细节，不顾程序的设计和现实环境去追求性能，反而会陷入迷雾中无法取得进展。

性能调优的步骤

总的来说，性能调优的步骤是：

1. 获取当前的性能指标，比如平均反应时间。
2. 获取测试时候外部因素的量化数据，比如每秒钟请求数。
3. 根据程序的设计，判断在上面的外部环境是否符合预期。如果不符合预期，改善外部环境后重新测试。
4. 如果外部环境符合预期，理想的性能指标应该是多少，结果是否在理想范围内？
5. 如果不在理想返回内，问题是由于内部因素导致的。接下来应该努力找到瓶颈来解决问题。

在讨论如何获取量化数据，如何分析瓶颈以前，先看看下面一些导致性能问题的常见情况：

程序的负荷太重，导致硬件成为性能的瓶颈。
观察到 CPU 利用率一直很高，响应时间随负荷增加而变长。当负荷降下来后性能又恢复正常。如果负载的确超过了程序的承受能力， 解决办法是升级硬件，或者增加服务器节点，用负载均衡把负载分担到多个节点上。

数据库无法及时返回查询结果，导致请求无法及时完成。
应用服务器上的 CPU 利用率都正常,通过检查 dump 发现工作线程都在等待数据库请求。解决方法是把监察重点从应用服务器转移到数据库服务器，检查为何数据库请求无法及时返回。可能是数据库很繁忙，或者数据库发生死锁。

由于代码的原因发生死锁。
现象是 CPU 利用率为 0，但是请求总是不返回。解决办法是通过 dump 文件检查进程各个线程状态，找到阻塞的原因

由于代码的原因发生死循环。

现象是负载正常，但是 CPU 利用率很高。比如死循环，或者程序使用了轮询。解决办法是通过 dump 文件检查高 CPU 的时候执行的是些什么代码，是否总是在一个循环体中无法结束

争用。
特点是进程创建了很多线程，但是这些线程都在争用同一个全局资源，比如 CriticalSection。如果该资源长时间被某一个线程拥有，其他线程都处于等待状态，无法处理请求。如果该资源只被短时间拥有，该资源会在各个线程之间流动，使得系统需要花费大量 CPU 时间进行线程切换和调度。根据资源的状态，从现象上看可能是高 CPU 也可能是低 CPU

细节上的缺陷。
如果测量下来 CPU，负载，外部资源等各方面数据都正常，但是响应时间总比设计中的响应时间慢一点点，很可能是在某一个细节上的优化不够。比如代码中频繁使用正则表达式进行字符串的搜索和替换，但是正则表达式的效率又无法达到预期，使得整体性能下降。解决办法就是优化正则表达式的使用，或者使用有针对性的，特殊优化后的普通的字符串操作来代替。这类问题其实是最难解决的。往往需要通过 log 文件来分析到底是哪一个细节比较慢，然后想办法方法问题来寻找根源。一个有效的 profiler 程序往往能够事半功倍地解决这样的问题。

从上面这些情况可以看到，区分性能问题的一个重要指标是 CPU 利用率。

当性能有问题，但是 CPU 利用率低的时候，往往是发生了死锁或者是在等待一些资源，比如数据或者网络。解决的思路是检查各个线程在等待什么。

当 CPU 比较高的时候，要观察负载的情况。如果负载正常，那么高 CPU 往往是由于程序的设计有问题导致的。解决办法是检查导致 CPU 繁忙的原因。

性能监视器

性能监视器是 Windows 自带的系统资源和性能监视工具。性能监视器能够量化地提供 CPU 使用率，内存分配情况，异常派发情况，线程调度频率等等信息。对于 ASP.NET，还能提供每秒钟请求数目，请求执行时间等等。性能监视器能够监视一段时间内上述资源的利用情况，提供平均值和峰值。

性能监视器有助于获取关于性能的具体指标，监视问题出现时候系统资源的变化情况。通过检查性能监视器中一些重要计数器的变化情况，往往能够找到一些非常有用的线索。比如比较 ASP.NET 每秒请求数量，请求响应时间和 CPU 利用率是否有相同的变化曲线就能看出性能是否跟负载相关。

下面先大致介绍一下性能监视器的用法，然后针对几个重要的计数器进行说明。

如何使用性能监视器

点击开始 -> 运行，输入 perfmon，打开性能监视器。默认模式是 System Monitor, 就是监视当前的系统状态，默认刷新频率是 1 秒。右边的窗口通过曲线实时更新性能数据。下面的窗口列举了当前添加的性能技术器。中间窗口显示了当前选定的性能计数器的历史峰值，平均值和当前值。

按 ctrl+i 可以打开性能计数器添加窗口。窗口上方可以选择目标计算机，在下拉菜单中可以选择 performance object。不同的性能计数器归类到不同的 performance object 中，处理器，磁盘，进程等等都是 performance object。对于每一个 performance object，在左下窗口中可以选择具体的计数器。选定计数器后，往往还可以进一步选择该计数器的对应目标 (instance)。比如选定了 Process -> Virtual Memory 计数器后，还可以在右下窗口中选择需要监视的具体进程。对于每一个计数器，选定后可以通过点击 Explain 按钮来获取关于该计数器的详细技术说明。

按 ctrl+q 打开性能监视器的属性设置。通过调整 Data 栏和 Graph 栏的数据，可以更清晰地获取数据的变化信息。在 Graph 栏中，可以调节曲线窗口的刻度范围。默认刻度范围是 0-100。该范围对 CPU 利用率来说是非常合适的，但是对于内存使用来说，由于计量单位是字节，所以 0-100 的客户范围显然无法容纳内存使用计数器。所以，可以增加 Graph 中的刻度范围方便对不同计数器进行针对性的观察。

调节 Graph 中的刻度虽然能够为不同计数器找到合适的刻度，但是依旧不方便同时观察多个计数器的共同变化趋势。比如无法找到一个刻度来同时清晰地显示 CPU 利用率和内存使用关系。为了解决这个问题，可以在 Data 栏中针对不同计数器选择缩放倍数。比如可以把内存使用的倍数选为 0.0000001, 即便是刻度范围是 0-100，也能容纳下 0-1GB 的内存使用变化。

仅仅观察当前性能状况对于解决问题往往是没有太多帮助的。性能监视器牛逼的地方在于能够把一段时间内的性能状况记录下来。通过分析比较各个技术器的关系和变化情况，能够量化性能状况，帮助找到问题的线索。具体的步骤是在左边选择 Performance logs and alert -> Counter logs. 然后新建立 log, 添加需要监视的计数器，选定 log 文件路径，刷新频率就可以了。具体步骤请参考:

How to create a log using System Monitor in Windows
http://support.microsoft.com/?id=248345

生成好性能日志后，可以通过下面的方法在性能监视器中打开:

1. 运行性能监视器，打开属性设置
2. 切换到 Source 栏，选择 log files，点 add 按钮
3. 选择 log 文件
4. 点 Time Range 按钮选择设置需要观察的时间区间
5. 打开完毕后，按照前面的步骤添加性能计数器进行观察。只是添加的计数器反映的不再是当前系统的实时情况，而是 log 中的情况。

一些重要的计数器

解决性能问题的时候，往往会让客户添加下面一些计数器进行性能收集。

Process object 下的所有计数器。
Processor object 下的所有计数器
System object 下的所有计数器
Memory object 下的所有计数器

如果客户的程序是.NET 程序，还会添加 .NET 开头的 object 下的所有计数器
如果客户使用 ASP.NET，还会添加 ASP.NET 开头的 object 下的所有计数器

分析性能日志的时候，我会重点观察下面这些计数器

Process object
Process object 中的计数器可以针对目标进程分析内存，CPU,线程数目和 handle 数目。选定出问题的目标进程，然后分析目标进程的下面一些计数器：

% Processor Time
该计数器是该进程占用 CPU 资源的指标。即便进程繁忙的时候，CPU 平均占用率应该在 80%以内。如果超过该数值，程序可以认为发生了 high CPU 的问题。另外一种问题是 CPU 波动幅度大。虽然平均占用率不高，但是上下跳动频繁。在某一个短时间段里面，会有连续高 CPU 的情况出现。

Handle Count
该计数器记录了当前进程使用的 kernel object handle 数量。Kernel object 是重要的系统资源。当程序进入稳定运行状态的时候，Handle Count 数量也应该维持在一个稳定的区间。如果发现 Handle Count 在整个程序周期内总体趋势是连续向上，可以考虑程序是否有 Handle Leak

ID Process
该计数器记录了目标进程的进程 ID。你可能觉得奇怪，ID 有什么好观察的。进程 ID 是用来观察程序是否有重启发生。比如 ASP.NET 工作进程可能会自动回收。由于进程名都相同，只有通过进程 ID 来判断是否进程有重新启动现象。如果 ID 有变化，看看程序是否发生崩溃或者 Recycle

Private Bytes
该计数器记录了当前通过 VirtualAlloc API Commit 的 Memory 数量。无论是直接调用 API 申请的内存，被 Heap Manager 申请的内存，或者是 CLR 的 managed heap，都算在里面。跟 Handle Count 一样，如果在整个程序周期内总体趋势是连续向上，说明有 Memory Leak

Virtual Bytes
该计数器记录了当前进程申请成功的用户态总内存地址，包括 DLL/EXE 占用的地址和通过 VirtualAlloc API Reserve 的内存地址数量，所以该计数器应该总大于 Private Bytes。一

般来说，Virtual Bytes 跟 Private Bytes 的变化大致一致。由于内存分片的存在，Virtual Bytes 跟 Private Byes 一般保持一个相对稳定的比例关系。当 Virtual Bytes 跟 Private Bytes 的比例关系大于 2 的时候，程序往往有比较严重的内存地址分片。

Processor object

Processor object 记录系统中芯片的负载情况。由于普通程序并不刻意绑定到某个具体 CPU 上执行，所以在多 CPU 机器上观察 Total Instance 也就足够了

% Processor Time 该计数器跟 Process 下的% Processor Time 的意义一样，不过这里记录的不是针对具体某一个进程，而且整个系统。通过把这个计数器跟 Process 下的同名计数器一起比较，就能看出系统的高 CPU 问题是否是由于单一的某个进程导致的

System

System object 记录系统中一个整体的统计信息。所以不区分 Instance. 通过比较 System object 下的 counter 和其他 counter 的变化趋势，往往能看出一些线索

Context Switch/sec

Context Switch 标示了系统中整体线程的调度，切换频率。线程切换是开销比较大的操作。频繁的线程切换导大量 CPU 周期被浪费。所以看到高 CPU 的时候，一定要跟 Context Switch 一起比较。如果两者有相同的变化趋势，高 CPU 往往是由于 contention 导致的，而不是死循环。

Exception Dispatches/sec

Exception Dispatches 表示了系统中异常派发，处理的频繁程序。跟线程切换一样，异常处理也需要大量的 CPU 开销。分析方法跟 Context Switch 雷同。

File Data Operations/sec

File Data Operations 记录了当前系统中磁盘文件读写的频繁程度。通过观察该计数器跟其他性能指标的变化趋势，通常能够判断磁盘文件操作是否是性能瓶颈。类似的计数器还有 Network Interface\Bytes total/sec

Memory

Memory object 记录了当前系统中整体内存的统计信息。

Available Mbytes 和 Committed Bytes

Available Mbytes 记录了当前剩余的物理内存数量。Committed Bytes 记录了所有进程 commit 的内存数量。结合两个计数器可以观察到：

1) 两者相加可以粗略估计系统总体可用内存多少，便于估计物理配置
2) 当 Available Mbytes 少于 100MB 的时候，说明系统总体内存吃紧，会影响到整个系统所有进程的性能。应该考虑增加物理内存或者监察内存泄露
3) 通过比较 Process\Private Bytes 跟 Virtual Bytes，便于进一步确认是否有内存泄露，判断内存泄露是否是某一单个进程导致

Free System Page Table Entries，Pool Paged Bytes 和 Pool Paged Bytes

这三个计数器可以衡量核心态空闲内存的数量。特别是当使用/3GB 开关后，核心态内存地址被压缩，容易导致核心态内存不足，继而引发一些非常妖怪的问题。可以参考前面提到的文章：

How to use the /userva switch with the /3GB switch to tune the User-mode space to a value between 2 GB and 3 GB

http://support.microsoft.com/kb/316739/en-us

.NET CLR Memory

.NET CLR Memory object 记录了 CLR 进程中跟 CLR 相关的内存信息。该类别下的所有计数器都很有意思，意思也非常直接。建议用一个例子程序进行测试和研究。下面是两个最常用的计数器

Bytes in all heaps

Bytes in all heaps 记录了上次 GC 发生时候所统计到的，进程中不能被回收的所有 CLR object 占用的内存空间。该计数器不是实时的，每次 GC 发生的时候该计数器才更新。跟同一进程的 Process\Private bytes 比较，可以区分出 managed heap 和 native memory 的变化情况。对于 memory leak，便于区分是 managed heap 的 leak 还是 native memory 的 leak

%Time in GC

%Time in GC 记录了 GC 发生的频繁程度。一般来说 15%以内算比较正常。当超过 20%说明 GC 发生过于频繁。由于 GC 不仅仅带来很高的 CPU 开销，还需要挂起目标进程的 CLR 线程，所以高频率 GC 是非常危险的。通过跟 CPU 利用率和其他性能指标比较，往往能够看出 GC 对性能的影响。高频率的 GC 往往因为

1) 负载过高
2) 不合理的架构，对内存使用效率不高
3) 内存泄露，内存分片导致内存压力

如果目标程序是 ASP.NET，在 ASP.NET 开头的 object 中，下面这些计数器对于测量 ASP.NET 的性能非常有用。由于不少计数器存在于多个 object 类别中，下面只列出具体的计数器名字，而不去对应到具体的 object:

Application Restarts

Application Restarts 记录了 ASP.NET Application Domain 重启的次数。导致 ASP.NET appDomain 重启的原因往往是虚拟目录中被修改。比如修改了 web.config 文件，或者防毒程序对虚拟目录进行扫描。通过该计数器可以观察是否有异常的重启现象

Request Execution time

Request Execution time 记录了请求的执行时间，是衡量 ASP.NET 性能的最直接参数。通过该计数器的平均值来衡量性能是否合乎预期值。需要注意的地方是由于 Windows 并非实时系统，所以不能用峰值来衡量整体性能。比如当 GC 发生的时候，请求执行时间肯定都要超过 GC 的时间。所以平均值才是有效的标准

Request Current

Request Current 记录了当前正在处理的和等待处理的请求。最理想的情况是 Request Current 等于 CPU 的数量，这说明请求跟硬件资源能并发处理的能力恰好吻合，硬件投资正运行在最优状态。但是一般说来，当负荷比较大的时候，Request Current 也随着增高。如果 Request Current 在一段时间内有超过 10 的情况，说明性能有问题。注意观察这个时候对应的 CPU 情况和其他的资源。如果 CPU 不高，很可能是程序中有 blocking 发生，比如等待数据库请求，导致请求无法及时完成。

Request/second
Request/second 计数器记录了每秒钟到达 ASP.NET 的请求数。这是衡量 ASP.NET 负载的直接参数。注意观察 Request/second 是否超过程序的预期吞吐量。如果 Request/Second 有突发的波动，注意看是否有拒绝服务攻击。通过把 Request/second,Request Current, Request Execution time 和系统资源一起比较，往往能够看出来 ASP.NET 整体性能的变化和各个因素之间的影响

Request in Application Queue
当 ASP.NET 没有空余的工作线程来处理新进入的请求的时候，新的请求会被放到 Application Queue 中。当 Application Queue 堆积的请求也超过设定数值的时候，ASP.NET 直接返回 503 Server too busy 错误，同时丢弃该请求。所以正常情况下，Request in Application Queue 应该总为 0，否则说明已经有请求堆积，性能问题严重

下面是性能问题的典型案例。

[案例分析，博客园的性能问题]

在准备案例研究的时候，恰好遇上了博客园的 dudu 抱怨网站偶尔崩溃，性能上也有问题。

博客园 Blog 程序遇到的奇怪问题
http://www.cnblogs.com/dudu/archive/2006/08/09/472162.html

从问题描述上看，首先是” 浏览器处于连接状态，却一直得不到服务器的响应”,这说明服务器发生了 hang 的问题。在事件日志中观察到 Recycle 的原因是工作进程没有及时地响应 IIS 的心跳包，进一步说明服务器很可能发生了死锁。所以建议抓取性能日志和 dump。

下面是问题发生时候的 dump 分析:

首先用!threads 命令看看线程的统计情况：

```
0:056> !threads
ThreadCount: 48
UnstartedThread: 0
BackgroundThread: 48
PendingThread: 0
DeadThread: 0
Hosted Runtime: no
```

```
                                PreEmptive   GC Alloc         Lock
      ID OSID ThreadOBJ    State    GC      Context      Domain   Count APT Exception
  12    1  d60 001541e0   1808220 Enabled  281f904c:281faa6c 1e8517b0    1 Ukn (Threadpool Worker)
  17    2  978 00195400     b220 Enabled  238bb8a0:238bd5e8 001dc760    0 MTA (Finalizer)
  18    3  ee4 001dd368    80a220 Enabled  00000000:00000000 001dc760    0 MTA (Threadpool Completion
Port)
  19    4  7e0 001f90f0   880b220 Enabled  00000000:00000000 001dc760    0 MTA (Threadpool Completion
Port)
…
  62   2e  968 1ea139d0   180b220 Enabled  241eaaa8:241ec99c 1e8517b0    1 MTA (Threadpool Worker)
  63   2f  e4c 0ee769e8   180b220 Enabled  23ef5560:23ef63a8 1e8517b0    1 MTA (Threadpool Worker)
  64   30  798 0ee11340   180b220 Enabled  00000000:00000000 1e8517b0    1 MTA (Threadpool Worker)
  66   20  e40 00173578   880b220 Enabled  00000000:00000000 001dc760    0 MTA (Threadpool Completion
Port)
```

统计下来一共有 48 个 CLR 线程。根据以往的经验，超过 30 个 CLR 线程表明程序中往往有 blocking 发生。仔细观察!threads 命令的输出，注意看有没有线程处于 GC 状态。(触发 GC 的线程在最后一栏会有 GC 标志)

如果有线程处于 GC 状态，很有可能 blocking 是 GC 导致的，大多数线程在等待 GC 完成。在这个案例中，并没有看到 GC 发生，所以接下来检查各个 CLR 线程的具体 callstack:

```
0:056> ~* e !clrstack
…
```

48 个线程的输入很长，所以这里就省略了完整的输出。从该命令的结果可以观察到，几乎所有的 CLR 线程都等待在类似的 callstack 上。下面是一些节选:

```
OS Thread Id: 0x798 (64)
ESP      EIP
3437de28 7d61c824 [InlinedCallFrame: 3437de28] <Module>.SNIReadSync(SNI_Conn*, SNI_Packet**, Int32)
3437de24 6518f867 SNINativeMethodWrapper.SNIReadSync(System.Runtime.InteropServices.SafeHandle,
IntPtr ByRef, Int32)
3437de9c 65307213
System.Data.SqlClient.TdsParserStateObject.ReadSni(System.Data.Common.DbAsyncResult,
System.Data.SqlClient.TdsParserStateObject)
3437ded4 65306f5b System.Data.SqlClient.TdsParserStateObject.ReadPacket(Int32)
3437dee0 653066b7 System.Data.SqlClient.TdsParserStateObject.ReadBuffer()
3437dee8 65306912 System.Data.SqlClient.TdsParserStateObject.ReadByte()
3437def0 652fc46c System.Data.SqlClient.TdsParser.Run(System.Data.SqlClient.RunBehavior,
System.Data.SqlClient.SqlCommand, System.Data.SqlClient.SqlDataReader,
System.Data.SqlClient.BulkCopySimpleResultSet, System.Data.SqlClient.TdsParserStateObject)
3437df44 652d7850 System.Data.SqlClient.SqlDataReader.ConsumeMetaData()
3437df58 652d5a8f System.Data.SqlClient.SqlDataReader.get_MetaData()
```

```
3437df84 652c8fcc
System.Data.SqlClient.SqlCommand.FinishExecuteReader(System.Data.SqlClient.SqlDataReader,
System.Data.SqlClient.RunBehavior, System.String)
3437dfbc 652c8a36 System.Data.SqlClient.SqlCommand.RunExecuteReaderTds(System.Data.CommandBehavior,
System.Data.SqlClient.RunBehavior, Boolean, Boolean)
3437e00c 652c8735 System.Data.SqlClient.SqlCommand.RunExecuteReader(System.Data.CommandBehavior,
System.Data.SqlClient.RunBehavior, Boolean, System.String, System.Data.Common.DbAsyncResult)
3437e04c 652c8699 System.Data.SqlClient.SqlCommand.RunExecuteReader(System.Data.CommandBehavior,
System.Data.SqlClient.RunBehavior, Boolean, System.String)
3437e068 652c75ef System.Data.SqlClient.SqlCommand.ExecuteReader(System.Data.CommandBehavior,
System.String)
3437e0a4 652c734d System.Data.SqlClient.SqlCommand.ExecuteDbDataReader(System.Data.CommandBehavior)
3437e0a8 65237618
System.Data.Common.DbCommand.System.Data.IDbCommand.ExecuteReader(System.Data.CommandBehavior)
3437e0ac 303c32b3 DuDuServer.Framework.Data.AdoHelper.ExecuteReader(System.Data.IDbCommand,
AdoConnectionOwnership)
3437e0d8 303c2f43 DuDuServer.Framework.Data.AdoHelper.ExecuteReader(System.Data.IDbConnection,
System.Data.IDbTransaction, System.Data.CommandType, System.String, System.Data.IDataParameter[],
AdoConnectionOwnership)
3437e11c 303c1691 DuDuServer.Framework.Data.AdoHelper.ExecuteReader(System.String,
System.Data.CommandType, System.String, System.Data.IDataParameter[])
3437e154 303c1625 DuDuServer.Framework.Data.SqlHelper.ExecuteReader(System.String,
System.Data.CommandType, System.String, System.Data.SqlClient.SqlParameter[])
3437e168 303c15bc DuDuServer.Framework.Data.SqlDataProvider.GetReader(System.String,
System.Data.SqlClient.SqlParameter[])
3437e19c 306f148c
DuDuServer.Framework.Data.SqlDataProvider.GetPagedEntriesReader(DuDuServer.Framework.Components.Pa
gedEntryQuery)
3437e1a8 307e00ea
DuDuServer.Framework.Data.DataDTOProvider.GetPagedEntries(DuDuServer.Framework.Components.PagedEnt
ryQuery)
3437e1d8 306ffefe
DuDuServer.Framework.Entries.GetPagedEntries(DuDuServer.Framework.Components.PagedEntryQuery)
3437e1dc 306ffec0 DuDuServer.Service.EntryService.GetPagedEntries(System.String,
DuDuServer.Framework.Components.PagedEntryQuery,
Microsoft.Practices.EnterpriseLibrary.Caching.ICacheItemExpiration)
3437e1f0 306ffd61
DuDuServer.Service.EntryService.GetPagedHomeEntries(DuDuServer.Framework.Configuration.BlogConfig,
Int32, Int32)
3437e25c 306ffbcf DuDuServer.Web.AggSite.PagedPosts.PagedDataBind()
3437e26c 303c576b DuDuServer.Web.AggSite.PagedPosts.Page_Load(System.Object, System.EventArgs)


OS Thread Id: 0xe4c (63)
ESP      EIP
```

```
342fdc7c 7d61c824 [InlinedCallFrame: 342fdc7c] <Module>.SNIReadSync(SNI_Conn*, SNI_Packet**, Int32)
342fdc78 6518f867 SNINativeMethodWrapper.SNIReadSync(System.Runtime.InteropServices.SafeHandle,
IntPtr ByRef, Int32)
342fdcf0 65307213
System.Data.SqlClient.TdsParserStateObject.ReadSni(System.Data.Common.DbAsyncResult,
System.Data.SqlClient.TdsParserStateObject)
342fdd28 65306f5b System.Data.SqlClient.TdsParserStateObject.ReadPacket(Int32)
342fdd34 653066b7 System.Data.SqlClient.TdsParserStateObject.ReadBuffer()
342fdd3c 65306912 System.Data.SqlClient.TdsParserStateObject.ReadByte()
342fdd44 652fc46c System.Data.SqlClient.TdsParser.Run(System.Data.SqlClient.RunBehavior,
System.Data.SqlClient.SqlCommand, System.Data.SqlClient.SqlDataReader,
System.Data.SqlClient.BulkCopySimpleResultSet, System.Data.SqlClient.TdsParserStateObject)
342fdd98 652d7850 System.Data.SqlClient.SqlDataReader.ConsumeMetaData()
342fddac 652d5a8f System.Data.SqlClient.SqlDataReader.get_MetaData()
342fddd8 652c8fcc
System.Data.SqlClient.SqlCommand.FinishExecuteReader(System.Data.SqlClient.SqlDataReader,
System.Data.SqlClient.RunBehavior, System.String)
342fde10 652c8a36 System.Data.SqlClient.SqlCommand.RunExecuteReaderTds(System.Data.CommandBehavior,
System.Data.SqlClient.RunBehavior, Boolean, Boolean)
342fde60 652c8735 System.Data.SqlClient.SqlCommand.RunExecuteReader(System.Data.CommandBehavior,
System.Data.SqlClient.RunBehavior, Boolean, System.String, System.Data.Common.DbAsyncResult)
342fdea0 652c8699 System.Data.SqlClient.SqlCommand.RunExecuteReader(System.Data.CommandBehavior,
System.Data.SqlClient.RunBehavior, Boolean, System.String)
342fdebc 652c75ef System.Data.SqlClient.SqlCommand.ExecuteReader(System.Data.CommandBehavior,
System.String)
342fdef8 652c734d System.Data.SqlClient.SqlCommand.ExecuteDbDataReader(System.Data.CommandBehavior)
342fdefc 65237618
System.Data.Common.DbCommand.System.Data.IDbCommand.ExecuteReader(System.Data.CommandBehavior)
342fdf00 6524063e System.Data.Common.DbDataAdapter.FillInternal(System.Data.DataSet,
System.Data.DataTable[], Int32, Int32, System.String, System.Data.IDbCommand,
System.Data.CommandBehavior)
342fdf58 65240146 System.Data.Common.DbDataAdapter.Fill(System.Data.DataSet, Int32, Int32,
System.String, System.Data.IDbCommand, System.Data.CommandBehavior)
342fdf9c 6523ff2b System.Data.Common.DbDataAdapter.Fill(System.Data.DataSet)
342fdfcc 303c9baf DuDuServer.Framework.Data.AdoHelper.ExecuteDataset(System.Data.IDbCommand)
342fe004 303c9ac8 DuDuServer.Framework.Data.AdoHelper.ExecuteDataset(System.Data.IDbConnection,
System.Data.CommandType, System.String, System.Data.IDataParameter[])
342fe028 303c99fb DuDuServer.Framework.Data.AdoHelper.ExecuteDataset(System.String,
System.Data.CommandType, System.String, System.Data.IDataParameter[])
342fe064 305ca76f DuDuServer.Framework.Data.AdoHelper.ExecuteDataset(System.String,
System.Data.CommandType, System.String)
342fe070 305ca746 DuDuServer.Framework.Data.SqlHelper.ExecuteDataset(System.String,
System.Data.CommandType, System.String)
342fe080 305cfb92 DuDuServer.Web.Controls.FocusedBloggerControl.Render(System.Web.UI.HtmlTextWriter)
```

```
OS Thread Id: 0x968 (62)
ESP      EIP
3427df54 7d61c824 [InlinedCallFrame: 3427df54] <Module>.SNIReadSync(SNI_Conn*, SNI_Packet**, Int32)
3427df50 6518f867 SNINativeMethodWrapper.SNIReadSync(System.Runtime.InteropServices.SafeHandle,
IntPtr ByRef, Int32)
3427dfc8 65307213
System.Data.SqlClient.TdsParserStateObject.ReadSni(System.Data.Common.DbAsyncResult,
System.Data.SqlClient.TdsParserStateObject)
3427e000 65306f5b System.Data.SqlClient.TdsParserStateObject.ReadPacket(Int32)
3427e00c 653066b7 System.Data.SqlClient.TdsParserStateObject.ReadBuffer()
3427e014 65306912 System.Data.SqlClient.TdsParserStateObject.ReadByte()
3427e01c 652fc46c System.Data.SqlClient.TdsParser.Run(System.Data.SqlClient.RunBehavior,
System.Data.SqlClient.SqlCommand, System.Data.SqlClient.SqlDataReader,
System.Data.SqlClient.BulkCopySimpleResultSet, System.Data.SqlClient.TdsParserStateObject)
3427e070 652d7850 System.Data.SqlClient.SqlDataReader.ConsumeMetaData()
3427e084 652d5a8f System.Data.SqlClient.SqlDataReader.get_MetaData()
3427e0b0 652c8fcc
System.Data.SqlClient.SqlCommand.FinishExecuteReader(System.Data.SqlClient.SqlDataReader,
System.Data.SqlClient.RunBehavior, System.String)
3427e0e8 652c8a36 System.Data.SqlClient.SqlCommand.RunExecuteReaderTds(System.Data.CommandBehavior,
System.Data.SqlClient.RunBehavior, Boolean, Boolean)
3427e138 652c8735 System.Data.SqlClient.SqlCommand.RunExecuteReader(System.Data.CommandBehavior,
System.Data.SqlClient.RunBehavior, Boolean, System.String, System.Data.Common.DbAsyncResult)
3427e178 652c8699 System.Data.SqlClient.SqlCommand.RunExecuteReader(System.Data.CommandBehavior,
System.Data.SqlClient.RunBehavior, Boolean, System.String)
3427e194 652c75ef System.Data.SqlClient.SqlCommand.ExecuteReader(System.Data.CommandBehavior,
System.String)
3427e1d0 652c734d System.Data.SqlClient.SqlCommand.ExecuteDbDataReader(System.Data.CommandBehavior)
3427e1d4 65237618
System.Data.Common.DbCommand.System.Data.IDbCommand.ExecuteReader(System.Data.CommandBehavior)
3427e1d8 6524063e System.Data.Common.DbDataAdapter.FillInternal(System.Data.DataSet,
System.Data.DataTable[], Int32, Int32, System.String, System.Data.IDbCommand,
System.Data.CommandBehavior)
3427e230 65240146 System.Data.Common.DbDataAdapter.Fill(System.Data.DataSet, Int32, Int32,
System.String, System.Data.IDbCommand, System.Data.CommandBehavior)
3427e274 6523ff2b System.Data.Common.DbDataAdapter.Fill(System.Data.DataSet)
3427e2a4 303c9baf DuDuServer.Framework.Data.AdoHelper.ExecuteDataset(System.Data.IDbCommand)
3427e2dc 303c9ac8 DuDuServer.Framework.Data.AdoHelper.ExecuteDataset(System.Data.IDbConnection,
System.Data.CommandType, System.String, System.Data.IDataParameter[])
3427e300 303c99fb DuDuServer.Framework.Data.AdoHelper.ExecuteDataset(System.String,
System.Data.CommandType, System.String, System.Data.IDataParameter[])
3427e33c 303c999a DuDuServer.Framework.Data.SqlHelper.ExecuteDataset(System.String,
System.Data.CommandType, System.String, System.Data.SqlClient.SqlParameter[])
```

```
3427e350 30519a7d DuDuServer.Web.UI.Controls.BlogRank.Page_Load(System.Object, System.EventArgs)

OS Thread Id: 0x660 (61)
ESP      EIP
33fbde90 7d61c824 [InlinedCallFrame: 33fbde90] <Module>.SNIReadSync(SNI_Conn*, SNI_Packet**, Int32)
33fbde8c 6518f867 SNINativeMethodWrapper.SNIReadSync(System.Runtime.InteropServices.SafeHandle,
IntPtr ByRef, Int32)
33fbdf04 65307213
System.Data.SqlClient.TdsParserStateObject.ReadSni(System.Data.Common.DbAsyncResult,
System.Data.SqlClient.TdsParserStateObject)
33fbdf3c 65306f5b System.Data.SqlClient.TdsParserStateObject.ReadPacket(Int32)
33fbdf48 653066b7 System.Data.SqlClient.TdsParserStateObject.ReadBuffer()
33fbdf50 65306912 System.Data.SqlClient.TdsParserStateObject.ReadByte()
33fbdf58 652fc46c System.Data.SqlClient.TdsParser.Run(System.Data.SqlClient.RunBehavior,
System.Data.SqlClient.SqlCommand, System.Data.SqlClient.SqlDataReader,
System.Data.SqlClient.BulkCopySimpleResultSet, System.Data.SqlClient.TdsParserStateObject)
33fbdfac 652d7850 System.Data.SqlClient.SqlDataReader.ConsumeMetaData()
33fbdfc0 652d5a8f System.Data.SqlClient.SqlDataReader.get_MetaData()
33fbdfec 652c8fcc
System.Data.SqlClient.SqlCommand.FinishExecuteReader(System.Data.SqlClient.SqlDataReader,
System.Data.SqlClient.RunBehavior, System.String)
33fbe024 652c8a36 System.Data.SqlClient.SqlCommand.RunExecuteReaderTds(System.Data.CommandBehavior,
System.Data.SqlClient.RunBehavior, Boolean, Boolean)
33fbe074 652c8735 System.Data.SqlClient.SqlCommand.RunExecuteReader(System.Data.CommandBehavior,
System.Data.SqlClient.RunBehavior, Boolean, System.String, System.Data.Common.DbAsyncResult)
33fbe0b4 652c8699 System.Data.SqlClient.SqlCommand.RunExecuteReader(System.Data.CommandBehavior,
System.Data.SqlClient.RunBehavior, Boolean, System.String)
33fbe0d0 652c75ef System.Data.SqlClient.SqlCommand.ExecuteReader(System.Data.CommandBehavior,
System.String)
33fbe10c 652c734d System.Data.SqlClient.SqlCommand.ExecuteDbDataReader(System.Data.CommandBehavior)
33fbe110 65237618
System.Data.Common.DbCommand.System.Data.IDbCommand.ExecuteReader(System.Data.CommandBehavior)
33fbe114 303c32b3 DuDuServer.Framework.Data.AdoHelper.ExecuteReader(System.Data.IDbCommand,
AdoConnectionOwnership)
33fbe140 303c2f43 DuDuServer.Framework.Data.AdoHelper.ExecuteReader(System.Data.IDbConnection,
System.Data.IDbTransaction, System.Data.CommandType, System.String, System.Data.IDataParameter[],
AdoConnectionOwnership)
33fbe184 303c1691 DuDuServer.Framework.Data.AdoHelper.ExecuteReader(System.String,
System.Data.CommandType, System.String, System.Data.IDataParameter[])
33fbe1bc 303c1625 DuDuServer.Framework.Data.SqlHelper.ExecuteReader(System.String,
System.Data.CommandType, System.String, System.Data.SqlClient.SqlParameter[])
33fbe1d0 303c15bc DuDuServer.Framework.Data.SqlDataProvider.GetReader(System.String,
System.Data.SqlClient.SqlParameter[])
33fbe204 3051f587 DuDuServer.Framework.Data.SqlDataProvider.GetCommentOwnerBlogID(Int32)
33fbe218 3051f45a DuDuServer.Framework.Data.DataDTOProvider.GetCommentOwnerBlogID(Int32)
```

```
33fbe248 3051f426 DuDuServer.Framework.Entries.GetCommentOwnerBlogID(Int32)
33fbe24c 3051f38e DuDuServer.Web.UI.Controls.Comments.IsOwner(Int32)
33fbe284 3047f0b5 DuDuServer.Web.UI.Controls.Comments.CommentsCreated(System.Object,
System.Web.UI.WebControls.RepeaterItemEventArgs)
33fbe36c 688b1f7e
System.Web.UI.WebControls.Repeater.OnItemCreated(System.Web.UI.WebControls.RepeaterItemEventArgs)
33fbe380 688b1b31 System.Web.UI.WebControls.Repeater.CreateItem(Int32,
System.Web.UI.WebControls.ListItemType, Boolean, System.Object)
33fbe3a0 688b19bb System.Web.UI.WebControls.Repeater.CreateControlHierarchy(Boolean)
33fbe3e0 688b1da2 System.Web.UI.WebControls.Repeater.OnDataBinding(System.EventArgs)
33fbe3e8 688b1bf9 System.Web.UI.WebControls.Repeater.DataBind()
33fbe3f0 3047a77d DuDuServer.Web.UI.Controls.Comments.BindComments()
33fbe41c 3047a620 DuDuServer.Web.UI.Controls.Comments.OnLoad(System.EventArgs)
```

……

从上面的 callstack 可以看到，这些 CLR 线程最终都是等待在 SQL 操作上。引发 SQL 操作的原因多姿多彩。既有 UserControl 在做 DataBindg, 也有页面本身在做数据查询，还有的似乎在获取 blog 的 comments 和 rank 信息。从这些信息可以清楚地看到:

1. 导致线程停止运行的原因是 SQL Operation Blocking。问题在于数据库没有及时地返回结果
2. 虽然都是在等 SQL，但是引发 SQL 操作的代码各不一样。说明并非某一个特别的 SQL 语句无法返回，而是整个 SQL 数据库有性能问题。

有了这些信息，接下来就是在 SQL 端检查 SQL Blocking 的具体原因，比如是否是 SQL 繁忙，还是某些 SQL 语句导致了锁表。为了让 SQL 端的检查更加方便，可以进一步从 dump 中挖掘出正在等待数据库返回的 SQL 语句。具体步骤是切换到发生 blocking 的 CLR 线程，用!dumpstackobjects 找到当前 stack 中的 SqlCommand 对象，然后找出 Command 对象的 SQL 命令.下面的步骤是找到 61 线程正在等待的 SQL 语句的步骤：

```
0:056> ~61s
eax=c000007c ebx=00000000 ecx=00000000 edx=00000000 esi=0000091c edi=00000000
eip=7d61c824 esp=33fbdc84 ebp=33fbdcf0 iopl=0         nv up ei pl nz na po nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000202
ntdll!NtWaitForSingleObject+0x15:
7d61c824 c20c00          ret     0Ch
0:061> !dumpstackobjects
OS Thread Id: 0x660 (61)
ESP/REG  Object   Name
33fbdd1c 06c54d70 System.Data.SqlClient.TdsParserStateObject
33fbdd24 06c5cc94 System.Data.SqlClient.SNIPacket
33fbddf0 06c58d3c System.Data.SqlClient.SNIHandle
…
33fbdf7c 241860f4 System.Data.SqlClient.SqlDataReader
```

```
33fbdf90 241860f4 System.Data.SqlClient.SqlDataReader
33fbdf94 24185f5c System.Data.SqlClient.SqlCommand
...
0:061> !do 24185f5c
Name: System.Data.SqlClient.SqlCommand
MethodTable: 653c38e8
EEClass: 653c3868
Size: 132(0x84) bytes
GC Generation: 0
 (C:\WINDOWS\assembly\GAC_32\System.Data\2.0.0.0__b77a5c561934e089\System.Data.dll)
Fields:
     MT    Field   Offset                 Type VT     Attr    Value Name
790f9c18  4000184       4       System.Object  0 instance 00000000 __identity
7a745c0c  40008bc       8 ...ponentModel.ISite  0 instance 00000000 site
7a742e54  40008bd       c ....EventHandlerList  0 instance 00000000 events
790f9c18  40008bb     104       System.Object  0   shared   static EventDisposed
    >> Domain:Value  001dc760:NotInit  1e8517b0:02d2595c <<
790fed1c  4001618      58        System.Int32  0 instance   217265 ObjectID
790fa3e0  4001619      10       System.String  0 instance 06b73cf4 _commandText
653de914  400161a      5c        System.Int32  0 instance       4 _commandType
…
653dd3f0  4001638      54 ...DeferedProcessing  0 instance 00000000 _outParamEventSink
790fed1c  4001617     814        System.Int32  0   shared   static _objectTypeCount
    >> Domain:Value  001dc760:NotInit  1e8517b0:NotInit  <<
0:061> !do 06b73cf4
Name: System.String
MethodTable: 790fa3e0
EEClass: 790fa340
Size: 70(0x46) bytes
GC Generation: 2
 (C:\WINDOWS\assembly\GAC_32\mscorlib\2.0.0.0__b77a5c561934e089\mscorlib.dll)
String: blog_GetCommentOwnerBlogID
Fields:
     MT    Field   Offset                 Type VT     Attr    Value Name
790fed1c  4000096       4        System.Int32  0 instance      27 m_arrayLength
790fed1c  4000097       8        System.Int32  0 instance      26 m_stringLength
790fbefc  4000098       c         System.Char  0 instance      62 m_firstChar
790fa3e0  4000099      10       System.String  0   shared   static Empty
    >> Domain:Value  001dc760:790d6584 1e8517b0:790d6584 <<
79124670  400009a      14       System.Char[]  0   shared   static WhitespaceChars
    >> Domain:Value  001dc760:02b403f0 1e8517b0:06ba5fb4 <<
```

拿到这些数据后，剩下的工作就是跟 DB Admin 一起排查 SQL 数据库。SQL 的排查可以专门写本书，超出了这里讨论的范围。下面有两篇文章描述了 SQL 发生死锁的案例：

分析及解决 SQLServer 死锁问题
http://blog.joycode.com/juqiang/archive/2006/12/18/89218.aspx
http://blog.joycode.com/juqiang/archive/2006/12/18/89223.aspx

当看到 ASP.NET 问题跟数据库相关的时候，除了找到发生问题的 SQL 语句，下面一些额外的检查往往能够发现一些潜在的问题:

1. 通过! FinalizeQueue 检查是否有大量的 SqlConnection 对象等待被 Finalize. 通常 Finalize queue 中的 SqlConnection 对象应该小于 10。当数量超过 30 的时候，通常说明代码中有使用完 SqlConnection 后忘记及时调用 Close 或者 Dispose 的情况。
2. 通过!dumpheap –stat 检查内存中是否有大量的 DataTable 对象。如果有，通过!gcroot 察看这些对象是否被保存到 Session 中。在 Session 中保存大量的数据容易导致很高的内存压力。

前面提到了检查!threads 结果来判断问题是否跟 GC 相关。下面就是一个例子。

[案例分析, 是 sql 呢还是 gc]

问题背景：

客户观察到 ASP.NET 工作进程在系统繁忙的时候发生高 CPU 现象，同时停止对客户端的响应。在发生高 CPU 的时候抓取 dump，对 dump 分析如下：

```
0:000> !t
Loaded Son of Strike data table version 5 from
"C:\WINDOWS\Microsoft.NET\Framework\v1.1.4322\mscorsvr.dll"
ThreadCount: 26
UnstartedThread: 0
BackgroundThread: 26
PendingThread: 0
DeadThread: 0
                            PreEmptive   GC Alloc                 Lock
       ID  ThreadOBJ      State     GC      Context         Domain     Count APT Exception
 10 0x1438 0x000ded58   0x180a222 Enabled  0x00000000:0x00000000 0x0013fd68     1 MTA (Threadpool
Worker)
 16 0x1308 0x000d41d0     0xb220 Enabled  0x00000000:0x00000000 0x000e0a50     0 MTA (Finalizer)
System.Web.HttpException
 18  0x2c8 0x022c10e8   0x1800220 Enabled  0x00000000:0x00000000 0x0013fd68     1 MTA (Threadpool
Worker)
 20  0x154 0x02e70078   0x1800220 Enabled  0x00000000:0x00000000 0x0013fd68     1 MTA (Threadpool
Worker)
 21 0x1330 0x02ebde18   0x1800220 Enabled  0x00000000:0x00000000 0x0013fd68     2 MTA (GC) (Threadpool
Worker)
```

```
。。。
105  0xdf4 0x27930210      0x220 Enabled  0x00000000:0x00000000 0x000e0a50     0 MTA
106 0x17a8 0x2707b1c8    0x1800220 Enabled  0x00000000:0x00000000 0x0013fd68      1 MTA (Threadpool
Worker)
```

从!threads 结果可以看到，线程 21 带 GC 标志，说明该 thread 触发了 GC。切换到线程 21，检查触发 GC 的 callstack:

```
0:021> !clrstack
Thread 21
ESP        EIP
0x03e6eb3c  0x7c82ed54 [FRAME: HelperMethodFrame]
0x03e6eb74  0x07eb618d [DEFAULT] SZArray String
Microsoft.VisualBasic.Strings.SplitHelper(String,String,I4,I4)
0x03e6ebb8  0x07eb5f3b [DEFAULT] SZArray String
Microsoft.VisualBasic.Strings.Split(String,String,I4,ValueClass
Microsoft.VisualBasic.CompareMethod)
0x03e6ebe4  0x08537f4d [DEFAULT] [hasThis] SZArray String
CommonEngines.CommonPage.cCPSQL.GetArrayFromDataset(String)
0x03e6ebe8  0x08537a40 [DEFAULT] [hasThis] Void CommonEngines.CommonPage.cCPSQL.GetNavigationLinks()
0x03e6ec0c  0x085373d5 [DEFAULT] [hasThis] Void XXNNN.CommonPage.PageDetails()
0x03e6ec1c      0x08532b89    [DEFAULT]    [hasThis]    Void    XXNNN.CommonPage.Page_Load(Object,Class
System.EventArgs)
```

对应的 unmanaged callstack 是:

```
0:021> kb
ChildEBP RetAddr  Args to Child
03e6e748 7c822124 77e6baa8 00000384 00000000 ntdll!KiFastSystemCallRet
03e6e74c 77e6baa8 00000384 00000000 00000000 ntdll!NtWaitForSingleObject+0xc
03e6e7bc 77e6ba12 00000384 ffffffff 00000000 kernel32!WaitForSingleObjectEx+0xac
03e6e7d0 791fee7b 00000384 ffffffff 00000000 kernel32!WaitForSingleObject+0x12
03e6e7f0 7920273d 00000000 00000000 00000000 mscorsvr!GCHeap::GarbageCollectGeneration+0x1a9
03e6e820 791d556b 03e6e840 00015ff8 00000003 mscorsvr!gc_heap::allocate_more_space+0x181
03e6e864 792117f6 00015ff4 00000002 01a81044 mscorsvr!gc_heap::allocate_large_object+0x95
03e6ea84 791b6269 02ebde50 00015ff4 00000002 mscorsvr!GCHeap::Alloc+0x15b
03e6ea98 791c0e9a 00015ff4 00000000 00040000 mscorsvr!Alloc+0x3a
03e6eae8 791c0efe 00a81046 03e6eb68 00000001 mscorsvr!AllocateArrayEx+0x161
03e6eb6c 07eb618d 00000000 2331e240 14755800 mscorsvr!JIT_NewArr1+0xbb
```

从上面的信息可以看到，客户代码中调用了 Split 字符串分割函数。在该函数的操作过程中触发了 GC。由于这里使用了 Server GC Flavor, 所以具体的 GC 执行是在 GC thread 上进行的:

```
0:021> ~12s
```

```
eax=00000010 ebx=00000001 ecx=36634fd0 edx=0000001c esi=000da7c8 edi=3668e518
eip=792001e9 esp=01defe54 ebp=01defed0 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
mscorsvr!gc_heap::plan_phase+0x26a:
792001e9 743b          je      mscorsvr!gc_heap::plan_phase+0x2ae (79200226) [br=0]
0:012> kb
ChildEBP RetAddr  Args to Child
01defed0 791ff43d 00000002 000da7c8 00000000 mscorsvr!gc_heap::plan_phase+0x26a
01deff24 791ff065 00000000 000da7c8 77e670b2 mscorsvr!gc_heap::gc1+0x78
01deff88 791fe6c3 00000000 00000000 000da7c8 mscorsvr!gc_heap::garbage_collect+0x22f
01deffac 792356be 00000000 01deffec 77e66063 mscorsvr!gc_heap::gc_thread_function+0x42
01deffb8 77e66063 000da7c8 00000000 00000000 mscorsvr!gc_heap::gc_thread_stub+0x1e
01deffec 00000000 792356a0 000da7c8 00000000 kernel32!BaseThreadStart+0x34
```

线程 12 正在进行 GC，代码正运行在 Plan_phrase 的 je 汇编指令上

接下来看看 GC 对整个系统的影响。看看其他线程在做什么:

```
Thread 106
ESP       EIP
0x01b9c72c  0x7c82ed54 [FRAME: NDirectMethodFrameStandalone] [DEFAULT] UI2
System.Data.Common.UnsafeNativeMethods/Dbnetlib.ConnectionRead(ValueClass
System.Runtime.InteropServices.HandleRef,SZArray UI1,UI2,UI2,UI2,ByRef I)
0x01b9c748  0x04364a1c [DEFAULT] [hasThis] Void System.Data.SqlClient.TdsParser.ReadNetlib(I4)
0x01b9c7a8  0x04364971 [DEFAULT] [hasThis] Void System.Data.SqlClient.TdsParser.ReadBuffer()
0x01b9c7b0  0x043648a3 [DEFAULT] [hasThis] UI1 System.Data.SqlClient.TdsParser.ReadByte()
0x01b9c7b8  0x04364102 [DEFAULT] [hasThis] Boolean System.Data.SqlClient.TdsParser.Run(ValueClass
System.Data.SqlClient.RunBehavior,Class System.Data.SqlClient.SqlCommand,Class
System.Data.SqlClient.SqlDataReader)
0x01b9c800  0x04367637 [DEFAULT] [hasThis] Void System.Data.SqlClient.SqlDataReader.ConsumeMetaData()
0x01b9c814  0x043675b6 [DEFAULT] [hasThis] SZArray Class System.Data.SqlClient._SqlMetaData
System.Data.SqlClient.SqlDataReader.get_MetaData()
0x01b9c81c  0x04366edb [DEFAULT] [hasThis] Class System.Data.SqlClient.SqlDataReader
System.Data.SqlClient.SqlCommand.ExecuteReader(ValueClass System.Data.CommandBehavior,ValueClass
System.Data.SqlClient.RunBehavior,Boolean)
0x01b9c86c  0x04366a06 [DEFAULT] [hasThis] Class System.Data.SqlClient.SqlDataReader
System.Data.SqlClient.SqlCommand.ExecuteReader(ValueClass System.Data.CommandBehavior)
0x01b9c880  0x043669be [DEFAULT] [hasThis] Class System.Data.IDataReader
System.Data.SqlClient.SqlCommand.System.Data.IDbCommand.ExecuteReader(ValueClass
System.Data.CommandBehavior)
0x01b9c884  0x028e13fc [DEFAULT] Class System.Data.IDataReader
CommonFunctions.Data.GetDataReader(String,Boolean,String,Boolean,Boolean)
0x01b9c8c0  0x07ea02d5 [DEFAULT] [hasThis] Void CuPages.Security.cAccessRights.GetAccess(Boolean)

Thread 87
```

```
ESP        EIP
0x29b7f704  0x7c82ed54 [FRAME: HelperMethodFrame]
0x29b7f73c  0x04367737 [DEFAULT] [hasThis] SZArray Class System.Data.SqlClient._SqlMetaData
System.Data.SqlClient.TdsParser.ProcessMetaData(I4)
0x29b7f75c  0x043646e2 [DEFAULT] [hasThis] Boolean System.Data.SqlClient.TdsParser.Run(ValueClass
System.Data.SqlClient.RunBehavior,Class System.Data.SqlClient.SqlCommand,Class
System.Data.SqlClient.SqlDataReader)
0x29b7f7a4  0x04367637 [DEFAULT] [hasThis] Void System.Data.SqlClient.SqlDataReader.ConsumeMetaData()
0x29b7f7b8  0x043675b6 [DEFAULT] [hasThis] SZArray Class System.Data.SqlClient._SqlMetaData
System.Data.SqlClient.SqlDataReader.get_MetaData()
0x29b7f7c0  0x04366edb [DEFAULT] [hasThis] Class System.Data.SqlClient.SqlDataReader
System.Data.SqlClient.SqlCommand.ExecuteReader(ValueClass System.Data.CommandBehavior,ValueClass
System.Data.SqlClient.RunBehavior,Boolean)
0x29b7f810  0x07da5fe4 [DEFAULT] [hasThis] Object System.Data.SqlClient.SqlCommand.ExecuteScalar()
0x29b7f844  0x04bbed28 [DEFAULT] Object
CommonFunctions.Data.GetSQLDataScalar(String,String,Boolean,Boolean)
0x29b7f880  0x04bbebcc [DEFAULT] Object CommonFunctions.Data.GetDataScalar(String,Boolean,String)
0x29b7f8b8  0x07ea0556 [DEFAULT] [hasThis] Void CuPages.Security.cAccessRights.GetAccess(Boolean)
0x29b7f8c8  0x07dcfeb4 [DEFAULT] [hasThis] Void CuPages.Template.AccessRights.GetAccess(Class
CuPages.Template.IGlobal,Boolean)
0x29b7f8dc  0x07dcf9cc [DEFAULT] [hasThis] Void XXNNN.Links.PageInit()

Thread 35
ESP        EIP
0x07cff5c8  0x7c82ed54 [FRAME: NDirectMethodFrameStandalone] [DEFAULT] UI2
System.Data.Common.UnsafeNativeMethods/Dbnetlib.ConnectionRead(ValueClass
System.Runtime.InteropServices.HandleRef,SZArray UI1,UI2,UI2,UI2,ByRef I)
0x07cff5e4  0x04364a1c [DEFAULT] [hasThis] Void System.Data.SqlClient.TdsParser.ReadNetlib(I4)
0x07cff644  0x04364971 [DEFAULT] [hasThis] Void System.Data.SqlClient.TdsParser.ReadBuffer()
0x07cff64c  0x043648a3 [DEFAULT] [hasThis] UI1 System.Data.SqlClient.TdsParser.ReadByte()
0x07cff654  0x04364102 [DEFAULT] [hasThis] Boolean System.Data.SqlClient.TdsParser.Run(ValueClass
System.Data.SqlClient.RunBehavior,Class System.Data.SqlClient.SqlCommand,Class
System.Data.SqlClient.SqlDataReader)
0x07cff69c  0x04367637 [DEFAULT] [hasThis] Void System.Data.SqlClient.SqlDataReader.ConsumeMetaData()
0x07cff6b0  0x043675b6 [DEFAULT] [hasThis] SZArray Class System.Data.SqlClient._SqlMetaData
System.Data.SqlClient.SqlDataReader.get_MetaData()
0x07cff6b8  0x04366edb [DEFAULT] [hasThis] Class System.Data.SqlClient.SqlDataReader
System.Data.SqlClient.SqlCommand.ExecuteReader(ValueClass System.Data.CommandBehavior,ValueClass
System.Data.SqlClient.RunBehavior,Boolean)
0x07cff708  0x07da5fe4 [DEFAULT] [hasThis] Object System.Data.SqlClient.SqlCommand.ExecuteScalar()
0x07cff73c  0x04bbed28 [DEFAULT] Object
CommonFunctions.Data.GetSQLDataScalar(String,String,Boolean,Boolean)
0x07cff778  0x04bbebcc [DEFAULT] Object CommonFunctions.Data.GetDataScalar(String,Boolean,String)
0x07cff7b0  0x085e7b36 [DEFAULT] [hasThis] Void CommonEngines.CommonList.cCLSQL.GetPageDetails()
0x07cff7e8  0x085e6715 [DEFAULT] [hasThis] Void CommonEngines.CommonList.cCLSQL.GetSettings()
```

```
0x07cff7f0  0x085e611f [DEFAULT] [hasThis] Void XXNNN.CommonList.GetCLSQL()
0x07cff81c      0x08516925    [DEFAULT]    [hasThis]    Void    XXNNN.CommonList.Page_Load(Object,Class
System.EventArgs)
```

看起来是不是很眼熟？这些线程都 block 在 SQL 操作上。根据前面的分析，接下来应该检查导致问题的 SQL 语句，然后到 SQL 服务器上分析。错，仔细检查一下这几个线程的 unmanaged callstack:

```
0:035> k
ChildEBP RetAddr
07cff4d0 7c822124 ntdll!KiFastSystemCallRet
07cff4d4 77e6baa8 ntdll!NtWaitForSingleObject+0xc
07cff544 77e6ba12 kernel32!WaitForSingleObjectEx+0xac
07cff558 791fed5c kernel32!WaitForSingleObject+0x12
07cff564 791fe901 mscorsvr!GCHeap::WaitUntilGCComplete+0x4f
07cff574 792d0ce3 mscorsvr!Thread::RareDisablePreemptiveGC+0xb5
07cff57c 792d0d6e mscorsvr!StubRareDisableRETURNWorker+0x36
07cff668 791d6f52 mscorsvr!UMThunkStubRareDisableWorker+0x7f
07cff63c 04364971 mscorsvr!GCHandleInternalAlloc+0x15
WARNING: Frame IP not in any known module. Following frames may be wrong.
07cff668 791d6f52 0x4364971
07cff754 7999f0e8 mscorsvr!GCHandleInternalAlloc+0x15
07cff794 799a9dae mscorlib_79990000+0xf0e8
00000000 00000000 mscorlib_79990000+0x19dae
0:035> ~87s
eax=00000059 ebx=00000000 ecx=1ccc4dd4 edx=7c82ed54 esi=00000390 edi=00000000
eip=7c82ed54 esp=29b7f35c ebp=29b7f3cc iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
ntdll!KiFastSystemCallRet:
7c82ed54 c3              ret
0:087> k
ChildEBP RetAddr
29b7f358 7c822124 ntdll!KiFastSystemCallRet
29b7f35c 77e6baa8 ntdll!NtWaitForSingleObject+0xc
29b7f3cc 77e6ba12 kernel32!WaitForSingleObjectEx+0xac
29b7f3e0 791fed5c kernel32!WaitForSingleObject+0x12
29b7f3ec 791fe901 mscorsvr!GCHeap::WaitUntilGCComplete+0x4f
29b7f3fc 79206661 mscorsvr!Thread::RareDisablePreemptiveGC+0xb5
29b7f424 791c0ccd mscorsvr!gc_heap::allocate_more_space+0x13c
29b7f64c 791b6269 mscorsvr!GCHeap::Alloc+0x7b
29b7f660 791c0e9a mscorsvr!Alloc+0x3a
29b7f6b0 791c0efe mscorsvr!AllocateArrayEx+0x161
29b7f734 04367737 mscorsvr!JIT_NewArr1+0xbb
WARNING: Frame IP not in any known module. Following frames may be wrong.
29b7f770 791d6f52 0x4367737
```

```
29b7f85c 7999f0e8 mscorsvr!GCHandleInternalAlloc+0x15
29b7f89c 799a9dae mscorlib_79990000+0xf0e8
00000000 00000000 mscorlib_79990000+0x19dae
0:087> ~106s
eax=00000119 ebx=00000000 ecx=01b9c5ac edx=7ffe0300 esi=00000390 edi=00000000
eip=7c82ed54 esp=01b9c638 ebp=01b9c6a8 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
ntdll!KiFastSystemCallRet:
7c82ed54 c3              ret
0:106> k
ChildEBP RetAddr
01b9c634 7c822124 ntdll!KiFastSystemCallRet
01b9c638 77e6baa8 ntdll!NtWaitForSingleObject+0xc
01b9c6a8 77e6ba12 kernel32!WaitForSingleObjectEx+0xac
01b9c6bc 791fed5c kernel32!WaitForSingleObject+0x12
01b9c6c8 791fe901 mscorsvr!GCHeap::WaitUntilGCComplete+0x4f
01b9c6d8 792d0ce3 mscorsvr!Thread::RareDisablePreemptiveGC+0xb5
01b9c6e0 792d0d6e mscorsvr!StubRareDisableRETURNWorker+0x36
01b9c7cc 791d6f52 mscorsvr!UMThunkStubRareDisableWorker+0x7f
01b9c7a0 04364971 mscorsvr!GCHandleInternalAlloc+0x15
WARNING: Frame IP not in any known module. Following frames may be wrong.
01b9c7cc 791d6f52 0x4364971
01b9c8a4 799a9dae mscorsvr!GCHandleInternalAlloc+0x15
00000000 00000000 mscorlib_79990000+0x19dae
```

从这里可以看到，unmanaged callstack 并不是在等来自于数据库的网络包，而是在等待 GC 完成。虽然说 managed callstack 是在等在 SQL 操作，但是这次的 SQL 操作停滞是由于 GC 导致的，而不是数据库服务器没有返回。

由此可以看到，分析问题不要被表面现象所迷惑，不能只看 managed callstack，一定要细心地弄清楚来龙去脉

弄清楚问题的关键后，接下来的问题是如何解决。对于 CLR 程序来说，是无法阻止 GC 发生的，关键在于 GC 是否合适而且高效地发生。由于客户没有提供性能日志，所以无法从 %time in GC 计数器中直接察看 GC 的频率。这里只有尝试着从 dump 中来找线索。

回到触发 GC 的线程 12,从前面的 callstack 看到导致 GC 的 function 是 String.Split.这里有什么鬼怪么,用!dso 看看相关的数据:

```
0:021> !dso
Thread 21
ESP/REG   Object    Name
0x3e6e924 0x14dd5cf4 System.Data.SqlClient.TdsParser
0x3e6e9b0 0x14dd5cf4 System.Data.SqlClient.TdsParser
0x3e6e9c0 0x14dd5cf4 System.Data.SqlClient.TdsParser
```

```
0x3e6e9d4 0x32986d7c System.Collections.ArrayList/ArrayListEnumeratorSimple
0x3e6eaac 0x204508f0 System.Object[]
0x3e6eab8 0x102d39b0 System.Globalization.CompareInfo
0x3e6eb04 0x2331e240 System.String    13»14»15»16»18»19»21»22»23»24»25»26»27»2
0x3e6eb0c 0x204508f0 System.Object[]
0x3e6eb1c 0x2331e240 System.String    13»14»15»16»18»19»21»22»23»24»25»26»27»2
0x3e6eb60 0x2331e240 System.String    13»14»15»16»18»19»21»22»23»24»25»26»27»2
0x3e6eb78 0x2331e240 System.String    13»14»15»16»18»19»21»22»23»24»25»26»27»2
…
```

哇，13,14,15…看起来好奇怪的字符串。用!do 命令检查后发现，改字符串是一连串用>>分割的数字，长度是:

```
0:021> !objsize 0x2331e240
sizeof(0x2331e240) =  388,804 ( 0x5eec4) bytes (System.String)
```

《神雕侠侣》一共有 40 回，总共也才 80 万字左右，这里一个字符串就相当于 10 回的神雕侠侣。

程序中使用 String.Split 对长字符串进行分割，结果保存在一个数组中。从上面的结果可以看到，0x204508f0 很有可能就是这个数组的地址，看看:

```
0:021> !do -v 0x204508f0
Name: System.Object[]
MethodTable 0x01b1209c
EEClass 0x01b12018
Size 90020(0x15fa4) bytes
GC Generation: 3
Array: Rank 1, Type CLASS
Element Type: System.Object
Content: 22,501 items
------ Will only dump out valid managed objects ----
  Address       MT  Class Name
0x1ce23e080x79b94638System.String
0x1ce23e200x79b94638System.String
。。。
```

这一个 String.Split 的调用就多了 2 万个 object 出来。虽然总体内存消耗并不一定多，但是频繁的小块内存创建带来不小的内存压力，使得 GC 频繁发生。

征得用户同意后，使用 Reflector 看到在客户的函数中大量使用了字符串的搜索，分割操作，这些操作本身需要消耗 CPU 资源，同时还带来内存压力导致 GC 的频繁发生。通过下面这片文章使用 StringBuilder 进行字符串操作优化后，问题解决:

Improving String Handling Performance in .NET Framework Applications

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndotnet/html/vbnstrcatn.asp

[案例分析, deadlock]

这个问题还是关于 ASP.NET 的. 问题背景:

客户的 ASP.NET 服务器 CPU 使用率为 0,客户端得不到任何反应.问题发生的时候抓取了 hang dump

打开 dump 后,一如既往地用!threads 命令检查发现一共有 20 个活动的 CLR thread,其中没有标记为 GC 的线程. 接下来用~* e !clrstack 命令查看对应的 CLR Callstack, 发现:

```
0:017> ~* e !clrstack
Thread 0
Not a managed thread.
Thread 1
Not a managed thread.
Thread 2
Not a managed thread.
Thread 3
Not a managed thread.
Thread 4
ESP       EIP
Thread 5
ESP       EIP
Thread 6
ESP       EIP
Thread 7
Not a managed thread.
Thread 8
Not a managed thread.
Thread 9
Not a managed thread.
Thread 10
ESP       EIP
0x01a9ec48  0x7c82ed54 [FRAME: GCFrame]
0x01a9f7e4  0x7c82ed54 [FRAME: ECallMethodFrame] [DEFAULT] Class System.Reflection.Assembly
System.Reflection.Assembly.nLoad(Class System.Reflection.AssemblyName,String,Boolean,Class
System.Security.Policy.Evidence,Boolean,Class System.Reflection.Assembly,ByRef ValueClass
System.Threading.StackCrawlMark)
0x01a9f808  0x799afb84 [DEFAULT] Class System.Reflection.Assembly
System.Reflection.Assembly.InternalLoad(Class System.Reflection.AssemblyName,Boolean,Class
System.Security.Policy.Evidence,ByRef ValueClass System.Threading.StackCrawlMark)
```

```
0x01a9f830  0x799ea867 [DEFAULT] Class System.Reflection.Assembly
System.Reflection.Assembly.Load(Class System.Reflection.AssemblyName)
0x01a9f838  0x0cdd2bef [DEFAULT] Class System.Collections.Hashtable
System.Web.UI.Util.GetReferencedAssembliesHashtable(Class System.Reflection.Assembly)
0x01a9f84c  0x0c119870 [DEFAULT] [hasThis] Boolean
System.Web.Compilation.PreservedAssemblyEntry.LoadDataFromFileInternal(Boolean)
0x01a9f89c  0x0c1194a2 [DEFAULT] [hasThis] Boolean
System.Web.Compilation.PreservedAssemblyEntry.LoadDataFromFile(Boolean)
0x01a9f8c4  0x0c118a20 [DEFAULT] Class System.Web.Compilation.PreservedAssemblyEntry
System.Web.Compilation.PreservedAssemblyEntry.GetPreservedAssemblyEntry(Class
System.Web.HttpContext,String,Boolean)
0x01a9f8dc  0x0c118764 [DEFAULT] [hasThis] Class System.Web.UI.ParserCacheItem
System.Web.UI.TemplateParser.GetParserCacheItemFromPreservedCompilation()
0x01a9f8f4  0x0c11740b [DEFAULT] [hasThis] Class System.Web.UI.ParserCacheItem
System.Web.UI.TemplateParser.GetParserCacheItemInternal(Boolean)
0x01a9f91c  0x0c1186ae [DEFAULT] [hasThis] Class System.Web.UI.ParserCacheItem
System.Web.UI.TemplateParser.GetParserCacheItemWithNewConfigPath()
0x01a9f94c  0x0c1172dc [DEFAULT] [hasThis] Class System.Web.UI.ParserCacheItem
System.Web.UI.TemplateParser.GetParserCacheItem()
0x01a9f97c  0x0c115d84 [DEFAULT] Class System.Type
System.Web.UI.ApplicationFileParser.GetCompiledApplicationType(String,Class
System.Web.HttpContext,ByRef Class System.Web.UI.ApplicationFileParser)
0x01a9f994  0x0c115c4c [DEFAULT] [hasThis] Void
System.Web.HttpApplicationFactory.CompileApplication(Class System.Web.HttpContext)
0x01a9f9a4  0x0c115b4f [DEFAULT] [hasThis] Void System.Web.HttpApplicationFactory.Init(Class
System.Web.HttpContext)
0x01a9f9e4  0x0c115953 [DEFAULT] Class System.Web.IHttpHandler
System.Web.HttpApplicationFactory.GetApplicationInstance(Class System.Web.HttpContext)
0x01a9fa10  0x0be76a97 [DEFAULT] [hasThis] Void System.Web.HttpRuntime.ProcessRequestInternal(Class
System.Web.HttpWorkerRequest)
0x01a9fa4c  0x0be76690 [DEFAULT] Void System.Web.HttpRuntime.ProcessRequest(Class
System.Web.HttpWorkerRequest)
0x01a9fa58  0x0be7320d [DEFAULT] [hasThis] I4 System.Web.Hosting.ISAPIRuntime.ProcessRequest(I,I4)
0x01a9fb20  0x79217188 [FRAME: ContextTransitionFrame]
0x01a9fc00  0x79217188 [FRAME: ComMethodFrame]
Thread 11
Not a managed thread.
Thread 12
Not a managed thread.
Thread 13
Not a managed thread.
Thread 14
ESP       EIP
Thread 15
Not a managed thread.
```

```
Thread 16
ESP        EIP
Thread 17
ESP        EIP
0x0d1aeaf4  0x7c82ed54 [FRAME: GCFrame]
0x0d1af690  0x7c82ed54 [FRAME: ECallMethodFrame] [DEFAULT] Class System.Reflection.Assembly
System.Reflection.Assembly.nLoad(Class System.Reflection.AssemblyName,String,Boolean,Class
System.Security.Policy.Evidence,Boolean,Class System.Reflection.Assembly,ByRef ValueClass
System.Threading.StackCrawlMark)
0x0d1af6b4  0x799afb84 [DEFAULT] Class System.Reflection.Assembly
System.Reflection.Assembly.InternalLoad(Class System.Reflection.AssemblyName,Boolean,Class
System.Security.Policy.Evidence,ByRef ValueClass System.Threading.StackCrawlMark)
0x0d1af6dc  0x799afa15 [DEFAULT] Class System.Reflection.Assembly
System.Reflection.Assembly.InternalLoad(String,Class System.Security.Policy.Evidence,ByRef
ValueClass System.Threading.StackCrawlMark)
0x0d1af6ec  0x799c2014 [DEFAULT] Class System.Reflection.Assembly
System.Reflection.Assembly.Load(String)
0x0d1af6f4  0x0c11ab17 [DEFAULT] Class System.Collections.Hashtable
System.Web.UI.CompilationConfiguration.LoadAssemblies(Class System.Collections.Hashtable)
0x0d1af758  0x0c11a8df [DEFAULT] [hasThis] Void System.Web.UI.TemplateParser.AppendConfigAssemblies()
0x0d1af784  0x0c119d01 [DEFAULT] [hasThis] Void System.Web.UI.TemplateParser.PrepareParse()
0x0d1af794  0x0c119b35 [DEFAULT] [hasThis] Void System.Web.UI.TemplateParser.Parse()
0x0d1af7c4  0x0c119a03 [DEFAULT] [hasThis] Class System.Web.UI.ParserCacheItem
System.Web.UI.TemplateParser.GetParserCacheItemThroughCompilation()
0x0d1af7f4  0x0c117434 [DEFAULT] [hasThis] Class System.Web.UI.ParserCacheItem
System.Web.UI.TemplateParser.GetParserCacheItemInternal(Boolean)
0x0d1af81c  0x0c1186ae [DEFAULT] [hasThis] Class System.Web.UI.ParserCacheItem
System.Web.UI.TemplateParser.GetParserCacheItemWithNewConfigPath()
0x0d1af84c  0x0c1172dc [DEFAULT] [hasThis] Class System.Web.UI.ParserCacheItem
System.Web.UI.TemplateParser.GetParserCacheItem()
0x0d1af87c  0x0c115d84 [DEFAULT] Class System.Type
System.Web.UI.ApplicationFileParser.GetCompiledApplicationType(String,Class
System.Web.HttpContext,ByRef Class System.Web.UI.ApplicationFileParser)
0x0d1af894  0x0c115c4c [DEFAULT] [hasThis] Void
System.Web.HttpApplicationFactory.CompileApplication(Class System.Web.HttpContext)
0x0d1af8a4  0x0c115b4f [DEFAULT] [hasThis] Void System.Web.HttpApplicationFactory.Init(Class
System.Web.HttpContext)
0x0d1af8e4  0x0c115953 [DEFAULT] Class System.Web.IHttpHandler
System.Web.HttpApplicationFactory.GetApplicationInstance(Class System.Web.HttpContext)
0x0d1af910  0x0be76a97 [DEFAULT] [hasThis] Void System.Web.HttpRuntime.ProcessRequestInternal(Class
System.Web.HttpWorkerRequest)
0x0d1af94c  0x0be76690 [DEFAULT] Void System.Web.HttpRuntime.ProcessRequest(Class
System.Web.HttpWorkerRequest)
0x0d1af958  0x0be7320d [DEFAULT] [hasThis] I4 System.Web.Hosting.ISAPIRuntime.ProcessRequest(I,I4)
0x0d1afa20  0x79217188 [FRAME: ContextTransitionFrame]
```

```
0x0d1afb00  0x79217188 [FRAME: ComMethodFrame]
Thread 18
ESP        EIP
0x0ec5ce80  0x7c82ed54 [FRAME: GCFrame]
0x0ec5d378  0x7c82ed54 [FRAME: GCFrame]
0x0ec5eac8  0x7c82ed54 [FRAME: GCFrame]
0x0ec5f664  0x7c82ed54 [FRAME: ECallMethodFrame] [DEFAULT] Class System.Reflection.Assembly
System.Reflection.Assembly.nLoad(Class System.Reflection.AssemblyName,String,Boolean,Class
System.Security.Policy.Evidence,Boolean,Class System.Reflection.Assembly,ByRef ValueClass
System.Threading.StackCrawlMark)
0x0ec5f688  0x799afb84 [DEFAULT] Class System.Reflection.Assembly
System.Reflection.Assembly.InternalLoad(Class System.Reflection.AssemblyName,Boolean,Class
System.Security.Policy.Evidence,ByRef ValueClass System.Threading.StackCrawlMark)
0x0ec5f6b0  0x799ea867 [DEFAULT] Class System.Reflection.Assembly
System.Reflection.Assembly.Load(Class System.Reflection.AssemblyName)
0x0ec5f6b8  0x0cdd2bef [DEFAULT] Class System.Collections.Hashtable
System.Web.UI.Util.GetReferencedAssembliesHashtable(Class System.Reflection.Assembly)
0x0ec5f6cc  0x0c119870 [DEFAULT] [hasThis] Boolean
System.Web.Compilation.PreservedAssemblyEntry.LoadDataFromFileInternal(Boolean)
0x0ec5f71c  0x0c1194a2 [DEFAULT] [hasThis] Boolean
System.Web.Compilation.PreservedAssemblyEntry.LoadDataFromFile(Boolean)
0x0ec5f744  0x0c118a20 [DEFAULT] Class System.Web.Compilation.PreservedAssemblyEntry
System.Web.Compilation.PreservedAssemblyEntry.GetPreservedAssemblyEntry(Class
System.Web.HttpContext,String,Boolean)
0x0ec5f75c  0x0c118764 [DEFAULT] [hasThis] Class System.Web.UI.ParserCacheItem
System.Web.UI.TemplateParser.GetParserCacheItemFromPreservedCompilation()
0x0ec5f774  0x0c11740b [DEFAULT] [hasThis] Class System.Web.UI.ParserCacheItem
System.Web.UI.TemplateParser.GetParserCacheItemInternal(Boolean)
0x0ec5f79c  0x0c1186ae [DEFAULT] [hasThis] Class System.Web.UI.ParserCacheItem
System.Web.UI.TemplateParser.GetParserCacheItemWithNewConfigPath()
0x0ec5f7cc  0x0c1172dc [DEFAULT] [hasThis] Class System.Web.UI.ParserCacheItem
System.Web.UI.TemplateParser.GetParserCacheItem()
0x0ec5f7fc  0x0c115d84 [DEFAULT] Class System.Type
System.Web.UI.ApplicationFileParser.GetCompiledApplicationType(String,Class
System.Web.HttpContext,ByRef Class System.Web.UI.ApplicationFileParser)
0x0ec5f814  0x0c115c4c [DEFAULT] [hasThis] Void
System.Web.HttpApplicationFactory.CompileApplication(Class System.Web.HttpContext)
0x0ec5f824  0x0c115b4f [DEFAULT] [hasThis] Void System.Web.HttpApplicationFactory.Init(Class
System.Web.HttpContext)
0x0ec5f864  0x0c115953 [DEFAULT] Class System.Web.IHttpHandler
System.Web.HttpApplicationFactory.GetApplicationInstance(Class System.Web.HttpContext)
0x0ec5f890  0x0be76a97 [DEFAULT] [hasThis] Void System.Web.HttpRuntime.ProcessRequestInternal(Class
System.Web.HttpWorkerRequest)
0x0ec5f8cc  0x0be76690 [DEFAULT] Void System.Web.HttpRuntime.ProcessRequest(Class
System.Web.HttpWorkerRequest)
```

```
0x0ec5f8d8  0x0be7320d [DEFAULT] [hasThis] I4 System.Web.Hosting.ISAPIRuntime.ProcessRequest(I,I4)
0x0ec5f9a0  0x79217188 [FRAME: ContextTransitionFrame]
0x0ec5fa80  0x79217188 [FRAME: ComMethodFrame]
Thread 19
ESP       EIP
0x0ecaee18  0x7c82ed54 [FRAME: ECallMethodFrame] [DEFAULT] String
System.Security.Util.Config._GetMachineDirectory()
0x0ecaee28  0x799a0a9f [DEFAULT] Void System.Security.Util.Config.GetFileLocales()
0x0ecaee34  0x799a07b7 [DEFAULT] [hasThis] Void System.Security.PolicyManager.InitData()
0x0ecaee50  0x799a072b [DEFAULT] Boolean System.Security.SecurityManager.InitPolicy()
0x0ecaee78  0x799a046b [DEFAULT] Class System.Security.PermissionSet
System.Security.SecurityManager.ResolvePolicy(Class System.Security.Policy.Evidence,Class
System.Security.PermissionSet,Class System.Security.PermissionSet,Class
System.Security.PermissionSet,ByRef Class System.Security.PermissionSet,Boolean)
0x0ecaeeb8  0x799a02b1 [DEFAULT] Class System.Security.PermissionSet
System.Security.SecurityManager.ResolvePolicy(Class System.Security.Policy.Evidence,Class
System.Security.PermissionSet,Class System.Security.PermissionSet,Class
System.Security.PermissionSet,ByRef Class System.Security.PermissionSet,ByRef I4,Boolean)
0x0ecaf180  0x791b3208 [FRAME: GCFrame]
0x0ecaf694  0x791b3208 [FRAME: ECallMethodFrame] [DEFAULT] [hasThis] Void
System.AppDomain.SetupDomainSecurity(String,Class System.Security.Policy.Evidence,Class
System.Security.Policy.Evidence,I)
0x0ecaf6b0  0x799cd45b [DEFAULT] Void
System.AppDomain.InternalRemotelySetupRemoteDomainHelper(String,Class
System.AppDomainSetup,I,SZArray Char,SZArray Char,SZArray UI1)
0x0ecaf6d8  0x799cd2c3 [DEFAULT] Void
System.AppDomain.InternalRemotelySetupRemoteDomain(I4,I4,String,Class
System.AppDomainSetup,I,SZArray Char,SZArray Char,SZArray UI1)
0x0ecaf6e4  0x799ccfda [FRAME: ContextTransitionFrame]
0x0ecaf738  0x799ccfda [DEFAULT] Void System.AppDomain.RemotelySetupRemoteDomain(Class
System.AppDomain,String,Class System.AppDomainSetup,Class System.Security.Policy.Evidence,Class
System.Security.Policy.Evidence,I)
0x0ecaf774  0x799c5a9a [DEFAULT] Class System.AppDomain System.AppDomain.CreateDomain(String,Class
System.Security.Policy.Evidence,Class System.AppDomainSetup)
0x0ecaf790  0x0be70507 [DEFAULT] [hasThis] Object
System.Web.Hosting.AppDomainFactory.Create(String,String,String,String,String,I4)
0x0ecaf96c  0x79217188 [FRAME: ComMethodFrame]
Thread 20
Not a managed thread.
Thread 21
Not a managed thread.
Thread 22
…
Thread 30
ESP       EIP
```

```
Thread 31
Not a managed thread.
Thread 32
ESP       EIP
Thread 33
ESP       EIP
Thread 34
ESP       EIP
Thread 35
Not a managed thread.
Thread 36
Not a managed thread.
Thread 37
…
Thread 50
ESP       EIP
Thread 51
Not a managed thread.
…
Thread 64
Not a managed thread.
```

受到篇幅限制,中间省略了部分Not a managed thread和空callstack的结果.从上面的输出来看,虽然 CLR thread 有 20 个,但大多数都是 idle 的.发生 blocking 的 CLR thread 只有四个,其中三个等在 Assembly.Load 上面.似乎在加载 Assembly 的时候出问题了.

由于最终堵塞线程的必然是 API 调用，比如 EnterCriticalSection, WaitForSingleObject，所以接下来检查对应的 unmanaged callstack。切换到第一个发生 blocking 的 CLR 线程检查:

```
0:010> kb100
ChildEBP RetAddr  Args to Child
01a9db1c 7c822124 7c83970f 00000154 00000000 ntdll!KiFastSystemCallRet
01a9db20 7c83970f 00000154 00000000 00000000 ntdll!NtWaitForSingleObject+0xc
01a9db5c 7c839620 00000000 00000004 00000001 ntdll!RtlpWaitOnCriticalSection+0x19c
01a9db7c 7c832ad0 7c889d94 00000000 00000000 ntdll!RtlEnterCriticalSection+0xa8
01a9dbb0 7c833ea9 00000001 00000000 01a9dbec ntdll!LdrLockLoaderLock+0xe4
01a9de20 77e41c90 1c68d7b8 01a9de6c 01a9de4c ntdll!LdrLoadDll+0xc9
01a9de88 791e0610 368ec3b8 00000000 00000009 kernel32!LoadLibraryExW+0x1b2
01a9dea4 791ea056 368ec3b8 00000000 00000009 mscorsvr!WszLoadLibraryEx+0x5f
01a9decc 791e7e58 368ec50c 36922588 00129000 mscorsvr!CorMap::BaseAddress+0x8b
01a9dee8 791e7ec4 36922378 368ec50c 00000000 mscorsvr!PEFile::Setup+0x45
01a9df04 791ea014 368ec50c 01a9e018 00000000 mscorsvr!PEFile::Create+0x38
01a9df1c 791ea117 368ec50c 368ec50c 01a9e018 mscorsvr!PEFile::CreateImageFile+0x3a
01a9df90 791ea151 00000001 00000000 3691f3b8 mscorsvr!PEFile::VerifyModule+0x76
```

```
01a9dfd4 791e88f6 01a9e248 00000000 26000000 mscorsvr!PEFile::Create+0x11a
01a9e01c 791e94d8 01a9e248 00000000 26000000 mscorsvr!SystemDomain::LoadFile+0x184
01a9e46c 791e8084 1d2312d8 3685d0f0 01a9f56c mscorsvr!AssemblySpec::GetAssemblyFromFusion+0x630
01a9e6f4 791e7f4b 00000000 01a9e7ac 00000001 mscorsvr!AssemblySpec::LowLevelLoadManifestFile+0x17d
01a9e714 791e2c95 01a9f53c 01a9e7a4 01a9e7ac mscorsvr!AppDomain::BindAssemblySpec+0x50
01a9ec24 791eb38b 01a9eca0 01a9eca8 01a9f7bc mscorsvr!AssemblySpec::LoadAssembly+0x98
01a9f7a4 01bab9ac 00000000 01a9f830 00000000 mscorsvr!AssemblyNative::Load+0x407
WARNING: Frame IP not in any known module. Following frames may be wrong.
01a9f820 799ea867 01a9f830 00000000 00000001 0x1bab9ac
01a9f824 01a9f830 00000000 00000001 0cdd2bef mscorlib_79990000+0x5a867
01a9f828 00000000 00000001 0cdd2bef 01a9f894 0x1a9f830
```

从 callstack 上看到，CLR 最后调用到 LoadLibrary API 上，在试图获取 DLL Loader Lock 的时候堵塞了。

DLL Loader Lock 是 OS 维护的一个锁，用来确保系统加载 DLL 时候的顺序和同步。该锁是通过 CriticalSection 来实现的。在一个 DLL 的 DLLMain 里面不允许调用 LoadLibrary 加载其它的 DLL，也跟该锁相关。详细的信息可以查看 MSDN 上关于 LoadLibrary 的详细说明。

这里最后堵塞在 CriticalSection 上，根据第二章的介绍，!cs 命令可以派上用途了。看看谁占用了这个 CriticalSection 没有释放:

```
0:010> !cs 7c889d94
-----------------------------------------
Critical section   = 0x7c889d94 (ntdll!LdrpLoaderLock+0x0)
DebugInfo          = 0x7c889dc0
LOCKED
LockCount          = 0x12
WaiterWoken        = No
OwningThread       = 0x00000948
RecursionCount     = 0x2
LockSemaphore      = 0x154
SpinCount          = 0x00000000
0:010> ~~[0x00000948]
  18  Id: cc4.948 Suspend: 1 Teb: 7ffa3000 Unfrozen
      Start: mscorsvr!ThreadpoolMgr::intermediateThreadProc (791fda8b)
      Priority: 0  Priority class: 32  Affinity: 3
```

于是切换到线程 18 检查:

```
0:010> ~18s
eax=449db1b4 ebx=00000000 ecx=791b4ef3 edx=00000001 esi=793ef47c edi=00000588
eip=7c82ed54 esp=0ec5c384 ebp=0ec5c3c0 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
ntdll!KiFastSystemCallRet:
```

```
7c82ed54 c3            ret
0:018> kb100
ChildEBP RetAddr  Args to Child
0ec5c380 7c822124 7c83970f 00000588 00000000 ntdll!KiFastSystemCallRet
0ec5c384 7c83970f 00000588 00000000 00000000 ntdll!NtWaitForSingleObject+0xc
0ec5c3c0 7c839620 00000000 00000004 0ec5c65c ntdll!RtlpWaitOnCriticalSection+0x19c
0ec5c3e0 791b2f0c 793ef47c 0ec5c618 791e78ab ntdll!RtlEnterCriticalSection+0xa8
0ec5c3ec 791e78ab 793ef47c 791e7c89 0c067cb8 mscorsvr!EE_EnterCriticalSection+0xc
0ec5c3f4 791e7c89 0c067cb8 00000000 00000000 mscorsvr!CorMap::Enter+0x13
0ec5c618 791e7d32 0000009c 00000003 0ec5c65c mscorsvr!CorMap::OpenFile+0x73
0ec5c648 791e88f6 0ec5c8bc 00000000 26000000 mscorsvr!PEFile::Create+0xc3
0ec5c690 791e94d8 0ec5c8bc 00000000 26000000 mscorsvr!SystemDomain::LoadFile+0x184
0ec5cae0 791e8084 1d2130b8 368f3390 0ec5cf2c mscorsvr!AssemblySpec::GetAssemblyFromFusion+0x630
0ec5cd68 791e7f4b 00000000 0ec5cee0 00000001 mscorsvr!AssemblySpec::LowLevelLoadManifestFile+0x17d
0ec5cd88 79208071 0ec5cefc 0ec5cef8 0ec5cee0 mscorsvr!AppDomain::BindAssemblySpec+0x50
0ec5d19c 79207d79 00000003 00000024 00000000
mscorsvr!Assembly::ComputeBindingDependenciesClosure+0x25a
0ec5d1c4 791e31ca 369291b8 369210e0 0ec5d324 mscorsvr!BaseDomain::CreateShareableAssemblyNoLock+0x7e
0ec5d340 79258d85 36928d80 369210e0 0ec5d3cc mscorsvr!BaseDomain::LoadAssembly+0xacd
0ec5d5e0 79019ad7 00000000 00000001 00000000 mscorsvr!ExecuteDLL+0x281
0ec5d5fc 7901a789 44940000 00000001 00000000 mscoree!CorDllMainWorker+0x6c
0ec5d638 7c82257a 44940000 00000000 00000000 mscoree!_CorDllMain+0x106
0ec5d658 7c8358fb 449594e2 44940000 00000001 ntdll!LdrpCallInitRoutine+0x14
0ec5d760 7c835bcb 00000000 00000000 00000000 ntdll!LdrpRunInitializeRoutines+0x367
0ec5d9f4 7c833ee5 00000000 1c626830 0ec5dcbc ntdll!LdrpLoadDll+0x3cd
0ec5dc70 77e41c90 1c626830 0ec5dcbc 0ec5dc9c ntdll!LdrLoadDll+0x198
0ec5dcd8 791e0610 36821d50 00000000 00000008 kernel32!LoadLibraryExW+0x1b2
0ec5dcf4 791ea056 36821d50 00000000 00000008 mscorsvr!WszLoadLibraryEx+0x5f
0ec5dd1c 791e7e58 36821ed4 368f7740 368f7530 mscorsvr!CorMap::BaseAddress+0x8b
0ec5dd38 7922c8ba 368f7530 36821ed4 00000000 mscorsvr!PEFile::Setup+0x45
0ec5ddac 791ea151 00000000 36922378 369210e0 mscorsvr!PEFile::VerifyModule+0x195
0ec5ddf0 791e88f6 0ec5e0c8 00000000 26000000 mscorsvr!PEFile::Create+0x11a
0ec5de38 791e94d8 0ec5e0c8 00000000 26000000 mscorsvr!SystemDomain::LoadFile+0x184
0ec5e2ec 791e8084 1d2130b8 368cc008 0ec5f3ec mscorsvr!AssemblySpec::GetAssemblyFromFusion+0x630
0ec5e574 791e7f4b 00000000 0ec5e62c 00000001 mscorsvr!AssemblySpec::LowLevelLoadManifestFile+0x17d
0ec5e594 791e2c95 0ec5f3bc 0ec5e624 0ec5e62c mscorsvr!AppDomain::BindAssemblySpec+0x50
0ec5eaa4 791eb38b 0ec5eb20 0ec5eb28 0ec5f63c mscorsvr!AssemblySpec::LoadAssembly+0x98
0ec5f624 01bab9ac 00000000 0ec5f6b0 00000000 mscorsvr!AssemblyNative::Load+0x407
WARNING: Frame IP not in any known module. Following frames may be wrong.
0ec5f6a0 799ea867 0ec5f6b0 00000000 00000001 0x1bab9ac
0ec5f6a4 0ec5f6b0 00000000 00000001 0cdd2bef mscorlib_79990000+0x5a867
0ec5f6a8 00000000 00000001 0cdd2bef 0ec5f714 0xec5f6b0
```

这个 callstack 好长。理清楚逻辑，Assembly.Load 调用被分发到 CLR Runtime,CLR Runtime

调用 API LoadLibaray 加载 DLL，在加载结束以前，系统会用 DLL_PROCESS_ATTACH 作为 dwReason 调用当前进程里面所有的 DLL 的 Entry Point (DLLMain). 所以 mscoree!_CorDllMain 被调用。Mscoree!_CorDllMain 再次调入 CLR Runtime 做一些 DLL/Assembly 相关的操作，操作的过程中最后等待到某一个 CriticalSectoin 上

那再次检查对应的 CriticalSection 被谁占用:

```
0:018> !cs 793ef47c
-----------------------------------------
Critical section   = 0x793ef47c (mscorsvr!CorMap::m_pCorMapCrst+0x0)
DebugInfo          = 0x000ddfd0
LOCKED
LockCount          = 0x1
WaiterWoken        = No
OwningThread       = 0x00000c6c
RecursionCount     = 0x1
LockSemaphore      = 0x588
SpinCount          = 0x00000000
0:018> ~~[0x00000c6c]
  17  Id: cc4.c6c Suspend: 1 Teb: 7ffa4000 Unfrozen
      Start: mscorsvr!ThreadpoolMgr::intermediateThreadProc (791fda8b)
      Priority: 0  Priority class: 32  Affinity: 3
```

Thread 17?那继续看:

```
0:018> ~17s
eax=00000000 ebx=00000000 ecx=00000002 edx=00000000 esi=7c889d94 edi=00000154
eip=7c82ed54 esp=0d1ae278 ebp=0d1ae2b4 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
ntdll!KiFastSystemCallRet:
7c82ed54 c3              ret
0:017> kb
ChildEBP RetAddr  Args to Child
0d1ae274 7c822124 7c83970f 00000154 00000000 ntdll!KiFastSystemCallRet
0d1ae278 7c83970f 00000154 00000000 00000000 ntdll!NtWaitForSingleObject+0xc
0d1ae2b4 7c839620 00000000 00000004 00000000 ntdll!RtlpWaitOnCriticalSection+0x19c
0d1ae2d4 7c83a023 7c889d94 00000004 368f44f8 ntdll!RtlEnterCriticalSection+0xa8
0d1ae3dc 77e67b95 44a10000 368f44f8 00000000 ntdll!LdrUnloadDll+0x35
0d1ae3f0 7924f974 44a10000 368f44f8 000d6540 kernel32!FreeLibrary+0x41
0d1ae408 79208491 00000000 00000001 368594b8 mscorsvr!CorMap::ReleaseHandleResources+0x70
0d1ae420 791e7937 0d1ae440 791e83ae 368f4510 mscorsvr!CorMapInfo::Release+0x63
0d1ae428 791e83ae 368f4510 368ff258 791e83bb mscorsvr!CorMap::ReleaseHandle+0xe
0d1ae434 791e83bb 000d6540 0d1ae5b4 791eb711 mscorsvr!PEFile::~PEFile+0x37
0d1ae440 791eb711 00000001 00000000 0d1af3e8 mscorsvr!PEFile::`scalar deleting destructor'+0xb
0d1ae5b4 791e2d29 368ff258 3691d008 0d1ae608 mscorsvr!BaseDomain::LoadAssembly+0x28b
```

```
0d1aead0 791eb38b 00000000 0d1aeb54 0d1af668 mscorsvr!AssemblySpec::LoadAssembly+0x4da
0d1af650 01bab9ac 00000000 0d1af6ec 00000000 mscorsvr!AssemblyNative::Load+0x407
*** WARNING: Unable to verify checksum for mscorlib.dll
*** ERROR: Module load completed but symbols could not be loaded for mscorlib.dll
WARNING: Frame IP not in any known module. Following frames may be wrong.
0d1af6cc 799afa15 0d1af6ec 00000000 28d395a0 0x1bab9ac
0d1af7ac 799b71a5 00000000 00000000 0d1af7ec mscorlib_79990000+0x1fa15
0d1af7b0 00000000 00000000 0d1af7ec 0c119a03 mscorlib_79990000+0x271a5
0:017> !cs 7c889d94
-----------------------------------------
Critical section   = 0x7c889d94 (ntdll!LdrpLoaderLock+0x0)
DebugInfo          = 0x7c889dc0
LOCKED
LockCount          = 0x12
WaiterWoken        = No
OwningThread       = 0x00000948
RecursionCount     = 0x2
LockSemaphore      = 0x154
SpinCount          = 0x00000000
0:017> ~~[0x00000948]
  18  Id: cc4.948 Suspend: 1 Teb: 7ffa3000 Unfrozen
      Start: mscorsvr!ThreadpoolMgr::intermediateThreadProc (791fda8b)
      Priority: 0  Priority class: 32  Affinity: 3
```

结论出来了． Thread10 在等 thread 18 占用的 CriticalSection,而 thread18 跟 thread17 发生了 deadlock.

根据 callstack，前因后果是:
thread18 首先调用 LoadLibrary 占用 DLL loader lock，然后在调用 CorMap 相关功能的时候等待 CorMap 相关的 CriticalSection 释放。

Thread17 在执行 CorMap 相关功能的时候，需要进入 DLL loader lock。这里可以推断出，Thread17 进入 CorMap 相关功能前占用了 CorMap 相关的 CriticalSection。

原因看明白了，解决方法呢？整理一下思路，这里无论是 managed callstack 还是 unmanaged callstack,所有的参与函数都是 CLR Runtime 和 OS API，中间没有涉及到任何客户自己的函数。说明问题在很大程度上跟客户的具体应用没有关系，很有可能是 CLR 或者 OS 的 bug

关于 CLR 跟 loader lock 的冲突，最臭名昭著的就是前面介绍过的 mixed DLL loading:

Mixed DLL Loading Problem
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dv_vstechart/html/vcconMixedDLLLoadingProblem.asp

为了看看是不是足够幸运地遇上这个问题，接下来尝试去找到引发问题的 DLL 的名字，看看是不是用 managed C++写的。检查 LoadLibaray 的第一个参数可以看到:

```
0:018> du 36821d50
36821d50  "c:\windows\microsoft.net\framewo"
36821d90  "rk\v1.1.4322\temporary asp.net f"
36821dd0  "iles\root\c01b9164\6d04ba4\assem"
36821e10  "bly\dl2\ce86c938\007079bd_9d34c6"
36821e50  "01\Company.product.usage"
36821e90  "name.Class1.dll"
0:018> du 368ec3b8
368ec3b8  "c:\windows\microsoft.net\framewo"
368ec3f8  "rk\v1.1.4322\temporary asp.net f"
368ec438  "iles\root\6882f6e1\6ac01cc1\asse"
368ec478  "mbly\dl2\a3ad3432\5b25229b_ffa0c"
368ec4b8  "601\Company.Class2.dll"
```

上面的输出提供了两个信息:

1. DLL 的具体名字
2. 从 DLL 的路径可以看到，该 DLL 是从 ASP.NET 临时文件夹加载的。所以该 DLL 应该是部署到 ASP.NET 的 bin 目录下

从 dump 中保存下这两个 DLL，用 reflector 检查后发现下面的信息:

1. 这两个 Assembly 是 StrongName 的
2. 这两个 Assembly 是用 managed C++开发的，因为里面包含了下面的类型信息，这些类型信息是 MC++编译器生成的

```
[StructLayout(LayoutKind.Explicit, Size=520, Pack=1), DebugInfoInPDB]
internal struct $ArrayType$0x47914e9e
{
}
```

这里遭遇著名的 mixed dll loading 问题。

在 CLR 1 上面，该问题是没有完美解决方案的。最显而易见的解决方法是升级到 CLR 2.0.

对企业环境的生产机升级 CLR 版本，需要严格的测试，是一个高风险的动作。所以再挑战一下自己，看看能不能在现有的基础上尽量做点什么，尽可能用最小的风险解决问题。

首先，从 callstack 上看到，问题根源在于两个 thread 互相等待。其实，只要 DLL Loader 早一点释放，或者 CorMap lock 早一点释放，问题都不会发生。也就是说，问题并非想重现就可以找到恰当的时机重现的。客户的程序肯定是执行了相同的 code path 很多次，终于有一

次遇上了死锁的条件。换句话说，要避免这个问题，只需要尽可能地减少加载 Assembly 这一部分代码的执行频率就可以了。

那 Assembly 的加载卸载跟什么相关呢？appDomain!Assembly 不能单独卸载，所以 Assembly 的频繁加载卸载肯定是 appDomain 对应的操作导致的。接下来可以用!dumpdomain 看看 appDomain 跟 Assenbly 的情况:

```
0:018> !dumpdomain -stat
    Domain      Num Assemblies Size Assemblies     Name
0x793f16a0                 1        2,138,112 System Domain
0x793f2b70               328      288,662,016      Shared Domain
0x000cf478                 2        2,490,368 DefaultDomain
0x001334b0                77       20,086,784 /LM/W3SVC/262227220/Root-1-127973823709142110
…
0x1d2312d8                27       16,484,864 /LM/W3SVC/1064537246/Root-36-127973891453478142
0x368594b8                11        7,123,456 /LM/W3SVC/182767242/Root-37-127973891487384175
0x36915188                 0                0   Domain39
Total 41 Domains
```

哇，这么多 appDomain。再去掉-stat 参数看看每个 appDomain 记载的信息，发现上面导致问题的两个 Assembly 在超过 30 个 appDomain 中都有加载。由此看来，这两个 MC++ Assembly 中包含的应该是一些公用的组件，由于在多个应用中都会用到，所以分别部署到了各个 Virtual Directory 的 bin 目录下。

这里有超过 30 个 Virtual Directory 被分配到了同一个 app pool，每一个 Virtual Directory 对应一个 appDomain，同一个 w3wp.exe 进程中频繁的 appDomian 加载和卸让发了引发问题的 code path 频繁执行。所以可以采取下面的方法来尽可能地避免问题的发生:

1. 检查是否有防毒程度频繁扫描 web.config 文件导致 appDomain 的重启。尽量避免对 Virtual Directory 的修改来避免 appDomain 的重启
2. 把多个 Virtual Directory 合并，以便共同引用同一套 MC++ Assembly，减少 MC++ Assembly 的加载
3. 拆分 Virtual Directory 到多个不同的 app pool.由于 Loader lock 根 CorMap lock 使用 CriticalSection,是进程级别的锁，所以分隔到多个进程可以减少触发问题的频率
4. 既然这两个Assembly是Strong Named的，应该放到GAC中，而不是bin目录下，详细参考: ASP.NET: Strong named assemblies should not be stored in the bin directory http://blogs.msdn.com/tess/archive/2006/04/13/575361.aspx

一个更麻烦点的案例

[案例分析，一团乱麻]

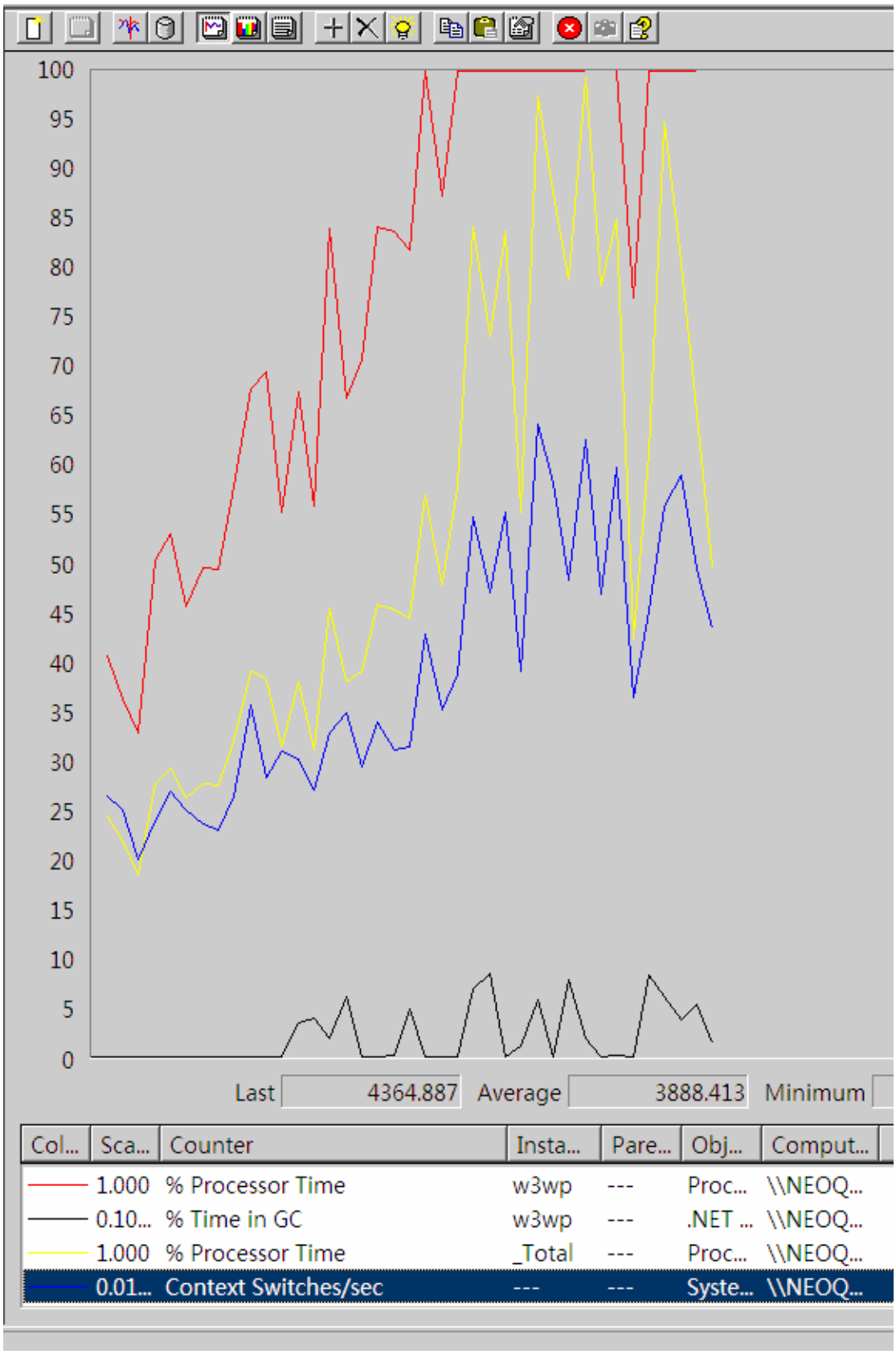
问题背景:

ASP.NET程序。客户观察到总体性能不满意。好多时候请求平均执行时间超过30秒，同时伴随CPU利用率频繁波动。建议客户在问题发生前开始抓取性能日志。当性能下降，CPU波动

厉害的时候抓取dump文件。

拿到性能日志后，很明显地看到:

1. 检查CPU Object下的CPU总体利用率, 发现有连续的频繁波动
2. 检查Process object下的w3wp.exe进程，该进程的CPU利用率有连续波动
3. 检查System object下的Context Switch/sec，该性能指标也有连续波动
4. 检查.NET Memory下w3wp.exe进程的%Time in GC，平均值超过20%



更直观的是，从图形上可以清晰地看到，总体CPU利用率，w3wp.exe进程的CPU利用率和Context Switch指标的图形非常相似，说明CPU问题是w3wp.exe单个进程导致的，而且很可

能跟频繁的线程切换相关。GC的发生频率比较高，有可能是GC导致的频繁切换，但是曲线跟上面三个指标不完全吻合，所以除了GC外，还有可能是其它原因。从性能日志中虽然无法看出问题根源，但是找到了分析的方向。接下来进行dump分析:

首先检查活动的 CLR 线程：

```
0:000> !t
Loaded Son of Strike data table version 5 from
"C:\WINDOWS\Microsoft.NET\Framework\v1.1.4322\mscorsvr.dll"
ThreadCount: 168
UnstartedThread: 0
BackgroundThread: 167
PendingThread: 0
DeadThread: 1
                            PreEmptive   GC Alloc                 Lock
       ID  ThreadOBJ      State     GC      Context          Domain     Count APT Exception
 12 0x2a40 0x000ce078   0x180a220 Enabled  0x00000000:0x00000000 0x000f41c8     1 MTA (Threadpool
Worker)
 16 0x156c 0x000cf248     0xb220 Enabled  0x00000000:0x00000000 0x000f41c8     0 MTA (Finalizer)
 18  0xd70 0x0c98cbe8   0x2001220 Enabled  0x00000000:0x00000000 0x000f41c8     0 Ukn
  5 0x29bc 0x0f0759d8      0x220 Enabled  0x00000000:0x00000000 0x000c4460     0 Ukn
 22 0x29a0 0x0f07fe00   0x3800220 Enabled  0x00000000:0x00000000 0x000f41c8     1 MTA (Threadpool)
…
93 0x1d44 0x1a478438     0x4220 Enabled  0x00000000:0x00000000 0x000c4460     0 STA
 95 0x2ad8 0x1a45e770   0x1800220 Enabled  0x29f6a2f4:0x29f6c11c 0x000f41c8     2 MTA (GC) (Threadpool
Worker)
 96 0x2a3c 0x1a57b0c0   0x1800220 Enabled  0x00000000:0x00000000 0x000f41c8     1 MTA (Threadpool
Worker)
191 0x2d18 0x1aac9e18   0x1800220 Enabled  0x00000000:0x00000000 0x000f41c8     1 MTA (Threadpool
Worker)
195 0x29b0 0x1a754d38   0x1800220 Enabled  0x00000000:0x00000000 0x000f41c8     0 MTA (Threadpool
Worker)
196 0x2fb8 0x2116fd90      0x220 Enabled  0x00000000:0x00000000 0x000c4460     0 Ukn
```

首先，活动的 CLR 线程非常多，线程 95 触发了 GC。接下来用~* e !clrstack 检查各个线程的 CLR Callstack。观察统计后，把 callstack 分为下面几种类型:

类型 1:

```
Thread 12
ESP        EIP
0x01bcdfc4  0x7c96ed54 [FRAME: GCFrame]
0x01bce07c  0x7c96ed54 [FRAME: ECallMethodFrame] [DEFAULT] I
System.Runtime.InteropServices.Marshal.GetComInterfaceForObject(Object,Class System.Type)
0x01bce08c  0x11891df0 [DEFAULT] [hasThis] I
System.EnterpriseServices.ServicedComponentProxy.SupportsInterface(ByRef ValueClass System.Guid)
```

```
0x01bce0b4  0x11891bdc [DEFAULT] [hasThis] Void
System.EnterpriseServices.ServicedComponentProxy.SendCreationEvents()
0x01bce0dc  0x11890ffc [DEFAULT] [hasThis] Void
System.EnterpriseServices.ServicedComponentProxy..ctor(Class
System.Type,Boolean,Boolean,Boolean,Boolean)
0x01bce114  0x11890cb2 [DEFAULT] [hasThis] Class System.MarshalByRefObject
System.EnterpriseServices.ServicedComponentProxyAttribute.System.Runtime.InteropServices.ICustomFa
ctory.CreateInstance(Class System.Type)
0x01bce12c  0x79ac6ed1 [DEFAULT] Class System.MarshalByRefObject
System.Runtime.Remoting.Activation.ActivationServices.CreateObjectForCom(Class System.Type,SZArray
Object,Boolean)
0x01bcf3b4  0x791f39fa [FRAME: InlinedCallFrame]
0x01bcf39c  0x118903b7 [DEFAULT] I System.EnterpriseServices.Thunk.Proxy.CoCreateObject(Class
System.Type,Boolean,ByRef Boolean,ByRef String)
0x01bcf498  0x1185e633 [DEFAULT] [hasThis] Class System.MarshalByRefObject
System.EnterpriseServices.ServicedComponentProxyAttribute.CreateInstance(Class System.Type)
0x01bcf4d4  0x799e79bb [DEFAULT] Class System.MarshalByRefObject
System.Runtime.Remoting.Activation.ActivationServices.IsCurrentContextOK(Class System.Type,SZArray
Object,Boolean)
0x01bcf584  0x791f39fa [FRAME: HelperMethodFrame]
0x01bcf5b8  0x15c48709 [DEFAULT] [hasThis] Void
CUNameSpcs.CUNameSpcs.CUPageName.Page_Load(Object,Class System.EventArgs)
  at [+0x601] [+0x323]
0x01bcf8ac  0x1185e2ac [DEFAULT] [hasThis] Void System.Web.UI.Control.OnLoad(Class System.EventArgs)
0x01bcf8bc  0x1185e1ec [DEFAULT] [hasThis] Void System.Web.UI.Control.LoadRecursive()
0x01bcf8d0  0x1185d1af [DEFAULT] [hasThis] Void System.Web.UI.Page.ProcessRequestMain()
0x01bcf914  0x1185a3be [DEFAULT] [hasThis] Void System.Web.UI.Page.ProcessRequest()
0x01bcf950  0x11859e2b [DEFAULT] [hasThis] Void System.Web.UI.Page.ProcessRequest(Class
System.Web.HttpContext)
0x01bcf958  0x11859e04 [DEFAULT] [hasThis] Void
System.Web.HttpApplication/CallHandlerExecutionStep.System.Web.HttpApplication+IExecutionStep.Exec
ute()
0x01bcf968  0x117dcb18 [DEFAULT] [hasThis] Class System.Exception
System.Web.HttpApplication.ExecuteStep(Class IExecutionStep,ByRef Boolean)
0x01bcf9b0  0x117dc582 [DEFAULT] [hasThis] Void System.Web.HttpApplication.ResumeSteps(Class
System.Exception)
0x01bcf9f8  0x1218e4de [DEFAULT] [hasThis] Void
System.Web.HttpApplication.ResumeStepsFromThreadPoolThread(Class System.Exception)
0x01bcfa04  0x11859c4d [DEFAULT] [hasThis] Void
System.Web.HttpApplication/AsyncEventExecutionStep.OnAsyncEventCompletion(Class
System.IAsyncResult)
0x01bcfa34  0x11859bb5 [DEFAULT] [hasThis] Void
System.Web.HttpAsyncResult.Complete(Boolean,Object,Class System.Exception)
0x01bcfa44  0x1218e321 [DEFAULT] [hasThis] Void
System.Web.SessionState.SessionStateModule.PollLockedSessionCallback(Object)
```

```
0x01bcfd50  0x791b3208 [FRAME: ContextTransitionFrame]
```

大约 50%的 CLR thread 都堵塞在该类型的 callstack 上
从 callstack 中看到，页面文件的 Page_Load 函数中有代码创建 ServicedComponent,创建的过程中堵塞在 InteropSerivce 和 ServicedComponent 的函数调用上。
当在 CLR 中使用 COM+组件的时候，CLR 中对应的 Class 往往从 ServicedComponent 继承。所以这里看出客户很可能使用了 COM+,在创建 COM+组件的时候发生了堵塞

这一类 managed callstack 对应的 unmanaged callstack 是：

```
 12  Id: 24e8.2a40 Suspend: 0 Teb: 7ff98000 Unfrozen
ChildEBP RetAddr  Args to Child
01bcdf8c 7c962194 7c80ad7b 791e0dda 01bcdff0 ntdll!KiFastSystemCallRet
01bcdf90 7c80ad7b 791e0dda 01bcdff0 7931b9b3 ntdll!NtYieldExecution+0xc
01bcdf94 791e0dda 01bcdff0 7931b9b3 00000000 kernel32!SwitchToThread+0x6
01bcdf9c 7931b9b3 00000000 00121d10 000ce078 mscorsvr!__SwitchToThread+0x26
01bcdfb0 792177bb 01bce060 00000000 11869208 mscorsvr!SpinLock::SpinToAcquire+0x4c
01bcdff0 79217888 1186ab3c 00000000 01bce060 mscorsvr!ComCallWrapper::CreateWrapper+0xa3
01bce004 792410ca 01bce060 11869208 01bce05c mscorsvr!ComCallWrapper::InlineGetWrapper+0x23
01bce034 792b0f52 00000000 80004002 00000001 mscorsvr!GetComIPFromObjectRef+0xd6
01bce050 020d4275 01bce060 023275d0 25dddbec mscorsvr!Interop::GetComInterfaceForObject+0x99
WARNING: Frame IP not in any known module. Following frames may be wrong.
01bce124 79ac6ed1 01bce154 000ce078 791f39fa 0x20d4275
00000000 00000000 00000000 00000000 00000000 mscorlib_79990000+0x136ed1
```

上面的 unmanaged callstack 说明最后 block 发生在一个 SpinLock 上，该 SpinLock 跟 CLR 的内部类 ComCallWrapper 相关。从 ComCallWrapper 名字可以猜到该类是用在 COM 组件操作上的。SpinLock 是一个种用户态的轻量级 lock，详细信息可以参考 rotor 源代码。这里说明这一类 thread 都在争用 COM 调用相关的锁。

类型 2：

```
Thread 22
ESP       EIP
0x11f7eda0  0x7c96ed54 [FRAME: GCFrame]
0x11f7ee5c  0x7c96ed54 [FRAME: HelperMethodFrame]
0x11f7eeb0  0x11be0aa9 [DEFAULT] Void System.EnterpriseServices.IdentityTable.RemoveObject(I,Object)
0x11f7eee0  0x11bcf212 [DEFAULT] [hasThis] Void
System.EnterpriseServices.ServicedComponentProxy.ReleaseContext()
0x11f7eeec  0x11bceb6b [DEFAULT] [hasThis] Void
System.EnterpriseServices.ServicedComponentProxy.Dispose(Boolean)
0x11f7ef18  0x11bce9d6 [DEFAULT] Void
System.EnterpriseServices.ServicedComponent.DisposeObject(Class
System.EnterpriseServices.ServicedComponent)
0x11f7ef24  0x11892d8c [DEFAULT] [hasThis] Class System.Runtime.Remoting.Messaging.IMessage
System.EnterpriseServices.ServicedComponentProxy.HandleDispose(Class
```

```
System.Runtime.Remoting.Messaging.IMessage)
0x11f7ef34  0x11892b33 [DEFAULT] [hasThis] Class System.Runtime.Remoting.Messaging.IMessage
System.EnterpriseServices.ServicedComponentProxy.CrossCtxInvoke(Class
System.Runtime.Remoting.Messaging.IMessage)
0x11f7ef58  0x11892afd [DEFAULT] [hasThis] Class System.Runtime.Remoting.Messaging.IMessage
System.EnterpriseServices.ServicedComponentProxy.Invoke(Class
System.Runtime.Remoting.Messaging.IMessage)
0x11f7ef68  0x799cdea0 [DEFAULT] [hasThis] Void
System.Runtime.Remoting.Proxies.RealProxy.PrivateInvoke(ByRef ValueClass
System.Runtime.Remoting.Proxies.MessageData,I4)
0x11f7f100  0x791f39fa [FRAME: TPMethodFrame] [DEFAULT] [hasThis] Void
System.EnterpriseServices.ServicedComponent.Dispose()
0x11f7f110  0x01f21e4b [DEFAULT] String
CUNAME.mdCompare.set_cmd(String,String,String,String,String,String,String,String,String)
0x11f7f18c  0x15b05b56 [DEFAULT] [hasThis] Void CUNAME.pip_SPFoooo_show_1_3.PlanSave(String,String)
0x11f7f228  0x15b0502a [DEFAULT] [hasThis] Void CUNAME.pip_SPFoooo_show_1_3.Enter_Save()
0x11f7f234  0x1565ca83 [DEFAULT] [hasThis] Void CUNAME.pip_SPFoooo_show_1_3.Page_Load(Object,Class
System.EventArgs)
0x11f7f32c  0x1185e2ac [DEFAULT] [hasThis] Void System.Web.UI.Control.OnLoad(Class System.EventArgs)
0x11f7f33c  0x1185e1ec [DEFAULT] [hasThis] Void System.Web.UI.Control.LoadRecursive()
0x11f7f350  0x1185d1af [DEFAULT] [hasThis] Void System.Web.UI.Page.ProcessRequestMain()
0x11f7f394  0x1185a3be [DEFAULT] [hasThis] Void System.Web.UI.Page.ProcessRequest()
0x11f7f3d0  0x11859e2b [DEFAULT] [hasThis] Void System.Web.UI.Page.ProcessRequest(Class
System.Web.HttpContext)
0x11f7f3d8  0x11859e04 [DEFAULT] [hasThis] Void
System.Web.HttpApplication/CallHandlerExecutionStep.System.Web.HttpApplication+IExecutionStep.Exec
ute()
0x11f7f3e8  0x117dcb18 [DEFAULT] [hasThis] Class System.Exception
System.Web.HttpApplication.ExecuteStep(Class IExecutionStep,ByRef Boolean)
0x11f7f430  0x117dc582 [DEFAULT] [hasThis] Void System.Web.HttpApplication.ResumeSteps(Class
System.Exception)
0x11f7f478  0x1218e4de [DEFAULT] [hasThis] Void
System.Web.HttpApplication.ResumeStepsFromThreadPoolThread(Class System.Exception)
0x11f7f484  0x11859c4d [DEFAULT] [hasThis] Void
System.Web.HttpApplication/AsyncEventExecutionStep.OnAsyncEventCompletion(Class
System.IAsyncResult)
0x11f7f4b4  0x11859bb5 [DEFAULT] [hasThis] Void
System.Web.HttpAsyncResult.Complete(Boolean,Object,Class System.Exception)
0x11f7f4c4  0x1218e321 [DEFAULT] [hasThis] Void
System.Web.SessionState.SessionStateModule.PollLockedSessionCallback(Object)
0x11f7f7d0  0x791b3208 [FRAME: ContextTransitionFrame]
```

大约有 3 个 thread 堵塞在上面类似的 callstack 中。跟 thread12 类似的是，堵塞都发生在 ServicedComponent 的操作上。不同的是这里最后堵塞在 ServicedComponent 的 Dispose 函数，

而不是创建 ServicedComponent。对应的 unmanged callstack:

```
 22  Id: 24e8.29a0 Suspend: 0 Teb: 7ff4b000 Unfrozen
ChildEBP RetAddr  Args to Child
11f7ec44 7c962114 7c82711b 00000001 11f7ec94 ntdll!KiFastSystemCallRet
11f7ec48 7c82711b 00000001 11f7ec94 00000000 ntdll!NtWaitForMultipleObjects+0xc
11f7ecf0 791e0b3b 00000001 0f098128 00000001 kernel32!WaitForMultipleObjectsEx+0x11a
11f7ed20 791e0bdd 00000001 0f098128 00000001 mscorsvr!Thread::DoAppropriateWaitWorker+0xc1
11f7ed74 791fccfe 00000001 0f098128 00000001 mscorsvr!Thread::DoAppropriateWait+0x46
11f7edf8 791fcc17 0f07fe00 ffffffff 00000000 mscorsvr!AwareLock::EnterEpilog+0x9d
11f7ee14 791fd43e 228ec690 0a1f4f6c 0a1f4fb0 mscorsvr!AwareLock::Enter+0x78
11f7eea8 11be0aa9 2a051704 0a1f4f6c 228ec690 mscorsvr!JITutil_MonContention+0x124
WARNING: Frame IP not in any known module. Following frames may be wrong.
11f7ef60 799cdea0 11f7efa8 118657e0 11f7efc0 0x11be0aa9
11f7ef94 791bc816 11f7f520 791f39fa 00000001 mscorlib_79990000+0x3dea0
11f7ef74 00000000 118657e0 11f7f014 00000000 mscorsvr!MDInternalRO::GetSigOfMethodDef+0x2b
```

跟 thread12 不一样，上面的 unmanaged callstack 并不是 block 在 CLR 的 COM 操作上，而是 CLR 引擎本身的某一个 AwareLock 中。但是这里看不出来该 AwareLock 是否跟某一个具体的 CLR 功能相关

类型 3：

```
Thread 49
ESP       EIP
0x18a8e990  0x7c96ed54 [FRAME: HelperMethodFrame]
0x18a8e9bc  0x154b2d81 [DEFAULT] String CCWCChungYak.CCWChunCom.MidH(String,I4,I4)
0x18a8e9ec  0x1711b296 [DEFAULT] [hasThis] Void CUNAME. setOutput(String)
0x18a8ea08  0x15bbe6cf [DEFAULT] [hasThis] Void CUNAME.FooName2.GetIpLogDetail(String,String,String)
0x18a8eaa0  0x15bbda2b [DEFAULT] [hasThis] Void CUNAME.FooName2.Page_Load(Object,Class
System.EventArgs)
0x18a8ec2c  0x1185e2ac [DEFAULT] [hasThis] Void System.Web.UI.Control.OnLoad(Class System.EventArgs)
0x18a8ec3c  0x1185e1ec [DEFAULT] [hasThis] Void System.Web.UI.Control.LoadRecursive()
0x18a8ec50  0x1185d1af [DEFAULT] [hasThis] Void System.Web.UI.Page.ProcessRequestMain()
0x18a8ec94  0x1185a3be [DEFAULT] [hasThis] Void System.Web.UI.Page.ProcessRequest()
0x18a8ecd0  0x11859e2b [DEFAULT] [hasThis] Void System.Web.UI.Page.ProcessRequest(Class
System.Web.HttpContext)
0x18a8ecd8  0x11859e04 [DEFAULT] [hasThis] Void
System.Web.HttpApplication/CallHandlerExecutionStep.System.Web.HttpApplication+IExecutionStep.Exec
ute()
0x18a8ece8  0x117dcb18 [DEFAULT] [hasThis] Class System.Exception
System.Web.HttpApplication.ExecuteStep(Class IExecutionStep,ByRef Boolean)
0x18a8ed30  0x117dc582 [DEFAULT] [hasThis] Void System.Web.HttpApplication.ResumeSteps(Class
System.Exception)
0x18a8ed78  0x1218e4de [DEFAULT] [hasThis] Void
System.Web.HttpApplication.ResumeStepsFromThreadPoolThread(Class System.Exception)
```

```
0x18a8ed84  0x11859c4d [DEFAULT] [hasThis] Void
System.Web.HttpApplication/AsyncEventExecutionStep.OnAsyncEventCompletion(Class
System.IAsyncResult)
0x18a8edb4  0x11859bb5 [DEFAULT] [hasThis] Void
System.Web.HttpAsyncResult.Complete(Boolean,Object,Class System.Exception)
0x18a8edc4  0x1218e321 [DEFAULT] [hasThis] Void
System.Web.SessionState.SessionStateModule.PollLockedSessionCallback(Object)
0x18a8f0d0  0x791b3208 [FRAME: ContextTransitionFrame]
```

Thread49 堵塞在客户自己定义函数上。大约有 20 来个 thread 有类似的 callstack，虽然函数名字不一样，但是对应的 unmanaged callstack 都类似：

```
49  Id: 24e8.1f84 Suspend: 0 Teb: 7ff38000 Unfrozen
ChildEBP RetAddr  Args to Child
18a8e620 7c962124 7c82baa8 000002c0 00000000 ntdll!KiFastSystemCallRet
18a8e624 7c82baa8 000002c0 00000000 00000000 ntdll!NtWaitForSingleObject+0xc
18a8e694 7c82ba12 000002c0 ffffffff 00000000 kernel32!WaitForSingleObjectEx+0xac
18a8e6a8 791fed5c 000002c0 ffffffff 791fe901 kernel32!WaitForSingleObject+0x12
18a8e6b4 791fe901 00000001 13663bb0 00000001 mscorsvr!GCHeap::WaitUntilGCComplete+0x4f
18a8e6c4 7926ec73 0000000c 13663be8 0000000c mscorsvr!Thread::RareDisablePreemptiveGC+0xb5
18a8e6ec 791c0ccd 13663be8 0000000c 00000000 mscorsvr!gc_heap::allocate_more_space+0xe6
18a8e914 791b6269 13663be8 0000000c 00000000 mscorsvr!GCHeap::Alloc+0x7b
18a8e928 791c0f77 0000000c 00000000 00000000 mscorsvr!Alloc+0x3a
18a8e948 791c0fc5 79baaf78 0a2b9134 000003f8 mscorsvr!FastAllocateObject+0x25
18a8e9b4 154b2d81 089711d8 0896ffd8 089703ec mscorsvr!JIT_NewFast+0x2c
WARNING: Frame IP not in any known module. Following frames may be wrong.
18a8e9e0 1711b296 00000002 18a8ea90 25e3ee7c 0x154b2d81
18a8e9e4 00000000 18a8ea90 25e3ee7c 089703ec 0x1711b296
```

Unmanaged Callstack 说明 GC 正在运行，要等待 GC 完成后才能继续。

类型 4：

```
Thread 70
ESP       EIP
0x1c89ee00  0x7c96ed54 [FRAME: HelperMethodFrame]
0x1c89ee2c  0x11eb5cb9 [DEFAULT] [hasThis] Class System.Data.SqlClient.SqlReturnValue
System.Data.SqlClient.TdsParser.ProcessReturnValue(I4)
0x1c89ee44  0x1189f74e [DEFAULT] [hasThis] Boolean System.Data.SqlClient.TdsParser.Run(ValueClass
System.Data.SqlClient.RunBehavior,Class System.Data.SqlClient.SqlCommand,Class
System.Data.SqlClient.SqlDataReader)
0x1c89ee8c  0x11bc5787 [DEFAULT] [hasThis] Void System.Data.SqlClient.SqlDataReader.ConsumeMetaData()
0x1c89eea0  0x11bc570e [DEFAULT] [hasThis] SZArray Class System.Data.SqlClient._SqlMetaData
System.Data.SqlClient.SqlDataReader.get_MetaData()
0x1c89eea8  0x11bc385b [DEFAULT] [hasThis] Class System.Data.SqlClient.SqlDataReader
System.Data.SqlClient.SqlCommand.ExecuteReader(ValueClass System.Data.CommandBehavior,ValueClass
```

```
System.Data.SqlClient.RunBehavior,Boolean)
0x1c89eef8  0x12180212 [DEFAULT] [hasThis] Class System.Data.SqlClient.SqlDataReader
System.Data.SqlClient.SqlCommand.ExecuteReader()
0x1c89ef08  0x1218243d [DEFAULT] [hasThis] Class System.Web.SessionState.SessionStateItem
System.Web.SessionState.SqlStateClientManager.DoGet(String,Class System.Data.SqlClient.SqlCommand)
0x1c89ef74  0x12182265 [DEFAULT] [hasThis] Class System.Web.SessionState.SessionStateItem
System.Web.SessionState.SqlStateClientManager.GetExclusive(String)
0x1c89efa0  0x121821ef [DEFAULT] [hasThis] Class System.IAsyncResult
System.Web.SessionState.SqlStateClientManager.System.Web.SessionState.IStateClientManager.BeginGet
Exclusive(String,Class System.AsyncCallback,Object)
0x1c89efb4  0x12182114 [DEFAULT] [hasThis] Boolean
System.Web.SessionState.SessionStateModule.GetSessionStateItem()
0x1c89efc4  0x1218e2ef [DEFAULT] [hasThis] Void
System.Web.SessionState.SessionStateModule.PollLockedSessionCallback(Object)
0x1c89f2d0  0x791b3208 [FRAME: ContextTransitionFrame]
```

Thread70 看起来是在等 SQL SessionState 服务器，但是检查 unmanaged callstack，可以看到实质跟 Thread49 一样，在等 GC 完成。大约 10 个类似的 thread

从上面的初步分析可以看到，各个线程上引发堵塞的客户函数不尽相同。有的在创建 COM+ object,有的在通过 Dispose 释放 COM+ object,有的在执行自定义函数，还有的在执行 SQL 操作.但是归根到底，堵塞的原因又三种：

1. 堵塞在 CLR 的 COM+相关的 SpinLock 上。在某一个线程中，CLR 独占了 COM+相关的锁，使得其它的 COM+操作堵塞
2. 堵塞在 CLR 的 AwareLock 中。但是看不出来该 AwareLock 是否跟某一个具体的 CLR 功能相关
3. 堵塞在 CLR 的 GC 上，GC 正在发生，在等待 GC 完成

根据上面的信息，已经可以看到 CPU 跳动的原因。由于 SpinLock 是很多 thread 正在抢占的资源，当 CLR 的 SpinLock 释放后，处于等待状态的几十个 thread 只有一个 thread 可以抢占到这个 SpinLock 开始执行。当这个 thread 执行完毕再次释放 SpinLock 的时候，会再次发生抢占的情况。所以这里会发生 contention，导致高 CPU. 另外正在进行的 GC 是需要消耗大量 CPU 时间，也是导致高 CPU 的原因。

需要注意的是，这里也有很多线程在等 GC,当 GC 完成的时候这些线程都会恢复运行。但这个行为并非 contention，也不会带来 CPU 的上下跳动，因为 GC 跟 SpinLock 不一样。

等待 SpinLock 的多个线程中，最终只有一个线程能够抢到这个 SpinLock;但是 GC 完成后，等待 GC 的线程都可以启动，不需要争抢。所以 GC 的释放不会产生 contention，除非系统内存压力特别大，使得 GC 非常频繁地发生。

接下来考虑怎么解决这个问题。既然已经看到 contention 跟 CLR 中 COM+相关的 SpinLock 相关，那么如何提高 SpinLock 的使用效率呢。是否跟客户代码中 COM+的具体用法相关？这里的 SpinLock 完全是 CLR 的内部实现，会不会是 CLR 设计上的缺陷？带着上面的问题

再仔细检查各个 thread，看看有没有异常情况。

首先值得检查的是带 GC 标记的 thread 95:

```
Thread 95
ESP       EIP
0x1e28ea50  0x7c96ed54 [FRAME: HelperMethodFrame]
0x1e28ea88  0x799978d0 [DEFAULT] [hasThis] SZArray Char System.String.ToCharArray(I4,I4)
0x1e28eaa0  0x799e9e57 [DEFAULT] [hasThis] SZArray UI1 System.Text.Encoding.GetBytes(String)
0x1e28eaac  0x154b2db4 [DEFAULT] String CCWCChungYak.CCWChunCom.MidH(String,I4,I4)
0x1e28eadc  0x1711d5c9 [DEFAULT] [hasThis] Void CUNAME.Foo.setOutput(String)
0x1e28eaf8  0x16f0d90a [DEFAULT] [hasThis] String CUNAME.FooName.Concern(String)
0x1e28ebac  0x16f0208f [DEFAULT] [hasThis] Void CUNAME.FooName.Page_Load(Object,Class
System.EventArgs)
0x1e28eda8  0x1185e2ac [DEFAULT] [hasThis] Void System.Web.UI.Control.OnLoad(Class System.EventArgs)
0x1e28edb8  0x1185e1ec [DEFAULT] [hasThis] Void System.Web.UI.Control.LoadRecursive()
0x1e28edcc  0x1185d1af [DEFAULT] [hasThis] Void System.Web.UI.Page.ProcessRequestMain()
0x1e28ee10  0x1185a3be [DEFAULT] [hasThis] Void System.Web.UI.Page.ProcessRequest()
0x1e28ee4c  0x11859e2b [DEFAULT] [hasThis] Void System.Web.UI.Page.ProcessRequest(Class
System.Web.HttpContext)
0x1e28ee54  0x11859e04 [DEFAULT] [hasThis] Void
System.Web.HttpApplication/CallHandlerExecutionStep.System.Web.HttpApplication+IExecutionStep.Exec
ute()
0x1e28ee64  0x117dcb18 [DEFAULT] [hasThis] Class System.Exception
System.Web.HttpApplication.ExecuteStep(Class IExecutionStep,ByRef Boolean)
0x1e28eeac  0x117dc582 [DEFAULT] [hasThis] Void System.Web.HttpApplication.ResumeSteps(Class
System.Exception)
0x1e28eef4  0x117dc453 [DEFAULT] [hasThis] Class System.IAsyncResult
System.Web.HttpApplication.System.Web.IHttpAsyncHandler.BeginProcessRequest(Class
System.Web.HttpContext,Class System.AsyncCallback,Object)
0x1e28ef10  0x020268b7 [DEFAULT] [hasThis] Void System.Web.HttpRuntime.ProcessRequestInternal(Class
System.Web.HttpWorkerRequest)
0x1e28ef4c  0x02026468 [DEFAULT] Void System.Web.HttpRuntime.ProcessRequest(Class
System.Web.HttpWorkerRequest)
0x1e28ef58  0x02022fe5 [DEFAULT] [hasThis] I4 System.Web.Hosting.ISAPIRuntime.ProcessRequest(I,I4)
0x1e28f020  0x79217188 [FRAME: ContextTransitionFrame]
0x1e28f100  0x79217188 [FRAME: ComMethodFrame]
0:095> kb
ChildEBP RetAddr  Args to Child
1e28e6cc 7c962124 7c82baa8 000002b4 00000000 ntdll!KiFastSystemCallRet
1e28e6d0 7c82baa8 000002b4 00000000 00000000 ntdll!NtWaitForSingleObject+0xc
1e28e740 7c82ba12 000002b4 ffffffff 00000000 kernel32!WaitForSingleObjectEx+0xac
1e28e754 791fee7b 000002b4 ffffffff 00000000 kernel32!WaitForSingleObject+0x12
1e28e774 7920273d 00000000 00000000 00006854 mscorsvr!GCHeap::GarbageCollectGeneration+0x1a9
1e28e7a4 791c0ccd 1a45e7a8 00006854 00000000 mscorsvr!gc_heap::allocate_more_space+0x181
```

```
1e28e9cc 791b6269 1a45e7a8 00006852 00000000 mscorsvr!GCHeap::Alloc+0x7b
1e28e9e0 791b8873 00006852 00000000 00000000 mscorsvr!Alloc+0x3a
1e28ea00 791b8814 01cc236c 00003423 00000000 mscorsvr!FastAllocatePrimitiveArray+0x45
1e28ea80 799978d0 1e28ead0 00000009 0631adac mscorsvr!JIT_NewArr1+0xbb
WARNING: Stack unwind information not available. Following frames may be wrong.
1e28ead0 1711d5c9 00000009 1e28eba4 00000000 mscorlib_79990000+0x78d0
1e28ead4 00000000 1e28eba4 00000000 05c0e594 0x1711d5c9
```

这里引发 GC 的是普通的字符操作，没有什么异常。既然 GC 正在发生，看看对应的 Server Mode GC thread 正在做什么：

```
 15  Id: 24e8.2fb0 Suspend: 0 Teb: 7ff95000 Unfrozen
ChildEBP RetAddr  Args to Child
01ebfd84 7c962124 7c82baa8 000002bc 00000000 ntdll!KiFastSystemCallRet
01ebfd88 7c82baa8 000002bc 00000000 00000000 ntdll!NtWaitForSingleObject+0xc
01ebfdf8 7c82ba12 000002bc ffffffff 00000000 kernel32!WaitForSingleObjectEx+0xac
01ebfe0c 791feef0 000002bc ffffffff 000ce780 kernel32!WaitForSingleObject+0x12
01ebfe20 79200e68 000ce780 00000000 000ce780 mscorsvr!t_join::join+0x54
01ebfe88 791fe6c3 00000000 00000000 000ce780 mscorsvr!gc_heap::garbage_collect+0x36
01ebfeac 792356be ffffffff 00000000 8083a8ca mscorsvr!gc_heap::gc_thread_function+0x42
01ebffb8 7c826063 000ce780 00000000 00000000 mscorsvr!gc_heap::gc_thread_stub+0x1e
01ebffec 00000000 792356a0 000ce780 00000000 kernel32!BaseThreadStart+0x34
```

奇怪，进行 GC 的 thread 并没有干活，而是停留在 WaitForSingleObject 上。这里在等什么呢？想想跟 GC 相关的除了 GC thread 外，还有 Finalizer，于是去!thread 的输出中找到 Finalizer:

```
16 0x156c 0x000cf248     0xb220 Enabled  0x00000000:0x00000000 0x000f41c8     0 MTA (Finalizer)
```

于是来看看 thread 16 在做什么：

```
16  Id: 24e8.156c Suspend: 0 Teb: 7ff94000 Unfrozen
ChildEBP RetAddr  Args to Child
01eff3a8 7c962114 7c82711b 00000001 01eff3f8 ntdll!KiFastSystemCallRet
01eff3ac 7c82711b 00000001 01eff3f8 00000001 ntdll!NtWaitForMultipleObjects+0xc
01eff454 7751722f 00000001 210cb490 00000000 kernel32!WaitForMultipleObjectsEx+0x11a
01eff4d0 7684ce02 00000000 ffffffff 00000001 ole32!CoWaitForMultipleHandles+0x100
01eff55c 7684b4e3 210cb458 00000001 01eff638 comsvcs!CActivity::EnterActivity+0x146
01eff570 77514d16 210cb448 01eff638 217feeb0 comsvcs!CActivity::Enter+0x16
01eff5a0 774c4f5f 00000001 00000001 01eff6f4 ole32!CPolicySet::DeliverEvents+0x1cd
01eff618 7750f13e 01eff638 00000002 01eff6f4 ole32!CPolicySet::Notify+0x317
01eff66c 7750fba0 00000000 2e27802c 79622d20 ole32!EnterForCallback+0xaf
01eff7cc 775100aa 01eff6a4 79622d20 01eff9e8 ole32!SwitchForCallback+0x1a3
01eff7f8 7749408c 2e27802c 79622d20 01eff9e8 ole32!PerformCallback+0x54
01eff890 775128f3 217feeb0 79622d20 01eff9e8 ole32!CObjectContext::InternalContextCallback+0x159
01eff8e0 79270c98 217feec0 79622d20 01eff9e8 ole32!CObjectContext::ContextCallback+0x85
```

```
01eff950 01c9a045 000cf248 01eff960 0f1742e8 mscorsvr!PInvokeCalliWorker+0x130
WARNING: Frame IP not in any known module. Following frames may be wrong.
01effaa8 799cdea0 01effaf0 11865810 01effb08 0x1c9a045
01effadc 791bc816 01effcb0 791f39fa 00000001 mscorlib_79990000+0x3dea0
01effabc 00000000 11865810 01effb5c 00000000 mscorsvr!MDInternalRO::GetSigOfMethodDef+0x2b
```

好像很繁忙的样子，看看对应的 managed callstack:

```
Thread 16
ESP       EIP
0x01eff97c  0x7c96ed54 [FRAME: PInvokeCalliFrame]
0x01eff9a0  0x11893898 [DEFAULT] [hasThis] Class System.Runtime.Remoting.Messaging.IMessage
System.EnterpriseServices.Thunk.Callback.DoCallback(Object,Class
System.Runtime.Remoting.Messaging.IMessage,I,Boolean,Class System.Reflection.MemberInfo,Boolean)
0x01effa7c  0x11892c66 [DEFAULT] [hasThis] Class System.Runtime.Remoting.Messaging.IMessage
System.EnterpriseServices.ServicedComponentProxy.CrossCtxInvoke(Class
System.Runtime.Remoting.Messaging.IMessage)
0x01effaa0  0x11892afd [DEFAULT] [hasThis] Class System.Runtime.Remoting.Messaging.IMessage
System.EnterpriseServices.ServicedComponentProxy.Invoke(Class
System.Runtime.Remoting.Messaging.IMessage)
0x01effab0  0x799cdea0 [DEFAULT] [hasThis] Void
System.Runtime.Remoting.Proxies.RealProxy.PrivateInvoke(ByRef ValueClass
System.Runtime.Remoting.Proxies.MessageData,I4)
0x01effc48  0x791f39fa [FRAME: TPMethodFrame] [DEFAULT] [hasThis] Void
System.EnterpriseServices.ServicedComponent._internalDeactivate(Boolean)
0x01effc58  0x11bceaa2 [DEFAULT] [hasThis] Void
System.EnterpriseServices.ServicedComponentProxy.Dispose(Boolean)
0x01effc84  0x11bedb00 [DEFAULT] [hasThis] Void
System.EnterpriseServices.ServicedComponentProxy.Finalize()
0x01effe00  0x79204678 [FRAME: ContextTransitionFrame]
0x01effe5c  0x79204678 [FRAME: GCFrame]
```

原来 Finalizer 线程正在忙着调用可以回收的 object 的 Finalizer，这里是一个 ServicedComponent 的 Dispose 方法正在被调用。由于 ServicedComponent 对应于一个 COM+ object,所以在 Dispose 的过程中 CLR 最后把调用派发到了 ole32 以及 comsvcs 两个 module 中。最后停留在 comsvcs module 的 CActivity 内部 class 上。从名字上看这里似乎在做 COM+ 的一些 Activity 同步。

由此看来，问题似乎比想象的复杂。本来看到 GC 归 GC，COM+ SpinLock 归 COM+ SpinLock。但是这里看到 GC 相关的 Finalizer thread 正在释放 COM+的资源，Finalizer thread 正等在 COM+的操作上。反过来想，COM+会不会在等 GC 呢？非常有可能。因为程序中的确有很多 CLR 中的 COM+操作，中间肯定需要操作 managed 内存。而所有 managed 内存操作，都必须跟 GC 同步，要等到 GC 完成后才能继续。回头仔细检查一下各个线程的 unmanaged callstack，看看是否有可疑的地方。

仔细检查后发现 thread 173 比较特殊：

```
173  Id: 24e8.bd8 Suspend: 0 Teb: 7f50d000 Unfrozen
ChildEBP RetAddr  Args to Child
24ffd328 7c962124 7c82baa8 000002c0 00000000 ntdll!KiFastSystemCallRet
24ffd32c 7c82baa8 000002c0 00000000 00000000 ntdll!NtWaitForSingleObject+0xc
24ffd39c 7c82ba12 000002c0 ffffffff 00000000 kernel32!WaitForSingleObjectEx+0xac
24ffd3b0 791fed5c 000002c0 ffffffff 791fe901 kernel32!WaitForSingleObject+0x12
24ffd3bc 791fe901 0001ab26 1a337278 00000000 mscorsvr!GCHeap::WaitUntilGCComplete+0x4f
24ffd3cc 791fd388 02394854 21d369b8 00000001 mscorsvr!Thread::RareDisablePreemptiveGC+0xb5
24ffd460 11890e21 00000001 00000000 02394854 mscorsvr!JITutil_MonContention+0xd7
WARNING: Frame IP not in any known module. Following frames may be wrong.
24ffd4a4 79ac6ed1 24ffd4d4 1a337278 791f39fa 0x11890e21
24ffd4bc 791f3a4e 79ac6e80 02394854 00000000 mscorlib_79990000+0x136ed1
24ffd4ec 791f7f70 79ac6e80 02394854 00000000 mscorsvr!CTPMethodTable::CallTarget+0x4e
24ffd510 79257396 11c4b44c 00000001 00000000 mscorsvr!CRemotingServices::CreateProxyOrObject+0x5f
24ffd5fc 7922988d 00000000 11c4b44c 00000000 mscorsvr!EEAllocateInstance+0x102
24ffd624 792298b0 00000000 11c4b44c 00000000 mscorsvr!EEInternalAllocateInstance+0x5b
24ffd648 774ab0d8 1a774490 00000000 24ffe440 mscorsvr!EEClassFactory::CreateInstance+0x1e
24ffd6d0 774a9b78 7758fd64 00000000 24ffe0b4 ole32!CServerContextActivator::CreateInstance+0x168
24ffd710 7685e45d 24ffe0b4 00000000 24ffd7e0
ole32!ActivationPropertiesIn::DelegateCreateInstance+0xf7
24ffd784 774a9b78 1a91a628 00000000 24ffe0b4 comsvcs!CObjectActivator::CreateInstance+0x2c0
24ffd7c4 775247a5 24ffe0b4 00000000 24ffd7e0
ole32!ActivationPropertiesIn::DelegateCreateInstance+0xf7
24ffd7e4 7750f153 1a67a3c0 22bbfc98 00000000 ole32!DoServerContextCCI+0x1d
24ffd830 7750fba0 00000000 22bbfc98 77524788 ole32!EnterForCallback+0xc4
```

对应的 managed callstack:

```
Thread 173
ESP       EIP
0x24ffd438  0x7c96ed54 [FRAME: GCFrame]
0x24ffd414  0x7c96ed54 [FRAME: HelperMethodFrame]
0x24ffd468  0x11890e21 [DEFAULT] Object System.EnterpriseServices.IdentityTable.FindObject(I)
0x24ffd494  0x11890c80 [DEFAULT] [hasThis] Class System.MarshalByRefObject
System.EnterpriseServices.ServicedComponentProxyAttribute.System.Runtime.InteropServices.ICustomFa
ctory.CreateInstance(Class System.Type)
0x24ffd4ac  0x79ac6ed1 [DEFAULT] Class System.MarshalByRefObject
System.Runtime.Remoting.Activation.ActivationServices.CreateObjectForCom(Class System.Type,SZArray
Object,Boolean)
0x24ffe734  0x791f39fa [FRAME: InlinedCallFrame]
0x24ffe71c  0x118903b7 [DEFAULT] I System.EnterpriseServices.Thunk.Proxy.CoCreateObject(Class
System.Type,Boolean,ByRef Boolean,ByRef String)
```

```
0x24ffe818  0x1185e633 [DEFAULT] [hasThis] Class System.MarshalByRefObject
System.EnterpriseServices.ServicedComponentProxyAttribute.CreateInstance(Class System.Type)
0x24ffe854  0x799e79bb [DEFAULT] Class System.MarshalByRefObject
System.Runtime.Remoting.Activation.ActivationServices.IsCurrentContextOK(Class System.Type,SZArray
Object,Boolean)
0x24ffe904  0x791f39fa [FRAME: HelperMethodFrame]
0x24ffe938  0x12ed92c7 [DEFAULT] [hasThis] String CUNameSpc.CCGFMEFBrowser.GetUserSSN(String)
  at [+0x47] [+0x10]
0x24ffe970  0x12ed9155 [DEFAULT] [hasThis] Void CUNameSpc.CCGFMEFBrowser.Page_Load(Object,Class
System.EventArgs)
  at [+0x9d] [+0x20]
0x24ffe9a8  0x1185e2ac [DEFAULT] [hasThis] Void System.Web.UI.Control.OnLoad(Class System.EventArgs)
0x24ffe9b8  0x1185e1ec [DEFAULT] [hasThis] Void System.Web.UI.Control.LoadRecursive()
0x24ffe9cc  0x1185d1af [DEFAULT] [hasThis] Void System.Web.UI.Page.ProcessRequestMain()
0x24ffea10  0x1185a3be [DEFAULT] [hasThis] Void System.Web.UI.Page.ProcessRequest()
0x24ffea4c  0x11859e2b [DEFAULT] [hasThis] Void System.Web.UI.Page.ProcessRequest(Class
System.Web.HttpContext)
0x24ffea54  0x11859e04 [DEFAULT] [hasThis] Void
System.Web.HttpApplication/CallHandlerExecutionStep.System.Web.HttpApplication+IExecutionStep.Exec
ute()
0x24ffea64  0x117dcb18 [DEFAULT] [hasThis] Class System.Exception
System.Web.HttpApplication.ExecuteStep(Class IExecutionStep,ByRef Boolean)
0x24ffeaac  0x117dc582 [DEFAULT] [hasThis] Void System.Web.HttpApplication.ResumeSteps(Class
System.Exception)
0x24ffeaf4  0x117dc453 [DEFAULT] [hasThis] Class System.IAsyncResult
System.Web.HttpApplication.System.Web.IHttpAsyncHandler.BeginProcessRequest(Class
System.Web.HttpContext,Class System.AsyncCallback,Object)
0x24ffeb10  0x020268b7 [DEFAULT] [hasThis] Void System.Web.HttpRuntime.ProcessRequestInternal(Class
System.Web.HttpWorkerRequest)
0x24ffeb4c  0x02026468 [DEFAULT] Void System.Web.HttpRuntime.ProcessRequest(Class
System.Web.HttpWorkerRequest)
0x24ffeb58  0x02022fe5 [DEFAULT] [hasThis] I4 System.Web.Hosting.ISAPIRuntime.ProcessRequest(I,I4)
0x24ffec20  0x79217188 [FRAME: ContextTransitionFrame]
0x24ffed00  0x79217188 [FRAME: ComMethodFrame]
```

可以看到,managed callstack 跟前面分析的 thread12 非常一致，都是在 Page_Load 中创建 ServicedComponent. 不过 thread12 最后 block 在 SpinLock 上，而 thread 173 最后 block 在 GC 上。由于 thread 173 需要创建 ServicedComponent，所以肯定也会获取 SpinLock，那当前 SpinLock 是否被 thread 173 占有呢?如果 SpinLock 正被 thread 173 占有，而 thread 173 又在等待 GC,那么 SpinLock 的释放也需要等 GC 完成，问题就更加复杂了。

暂且不去深入检查 thread 173 是否拥有 SpinLock，回头把所有的情况联系在一起考虑，会发现 COM+的操作，Finalizer 的操作和 GC 是错综复杂交织在一起的，彼此都有潜在的依赖关系，似乎很容易发生 deadlock。这样的设计是否有缺陷？

我认为这样的设计的确是有缺陷的，但是这个缺陷是没有办法克服的。由于 COM 本身有线程模型和同步机制，所以进行 COM 操作的时候线程之间是有依赖关系的。但是 CLR GC 也有内在的线程依赖关系。由于 COM 模型跟 CLR Runtime 并不是平行关系，所以当两者线程模型有冲突的时候，是没有完美的解决方案的。

但是我们并不应该为此而感到沮丧。在恶劣的条件下找到解决办法才是需要努力去做的。回到这个问题，我们最后在 Finalizer thread 中发现 CLR 的 GC 和 COM+的 SpinLock 有联系。那我们有没有办法尽可能地消除这样的联系呢，或者尽可能地消除 CLR GC 和 COM+的线程模型的冲突？

COM+的跨线程操作发生在 COM+创建，调用和销毁上。COM+的创建和调用是依赖于程序逻辑，并没有多少可以改变的地方，而 COM+组件的销毁，在当前的 dump 中看到两部分：

1. 在 Finalizer thread 中销毁
2. 同 thread 22 类似，在代码中显示调用 Dispose 销毁

其中第二种方法是程序可控的，而第一种方法是依赖于 GC 发生的时序。所以，这里似乎更应该手动调用 Dispose 方法来及时释放，而不应该留到 GC 发生的时候释放。如果及时释放，会占有下面一些优势：

1. ServicedComponent 不会堆积。如果依赖于 GC,每次 Finalizer 运行的时候会一次性清理可以被释放的 ServicedComponent. 由于 ServicedComponent 的释放可能要调用 COM+的操作，所以整个 Finalize 过程需要比较长的时间才能完成，导致 GC 效率变慢，大量 thread 堆积。如果及时释放，会把释放需要的开销分担，而不是累积。
2. 创建 COM+的 thread 往往会跟该 COM+组件有线程相依性。所以及时地在 COM+组件的创建线程中释放，可以避免线程切换带来的开销，让释放的过程就在当前线程解决。如果留给 Finalizer thread 去释放，释放的调用往往需要派发到该 COM+组件相依的线程中，带来多余的开销
3. 避免死锁。还是由于 COM+线程相依性。在 Finalizer thread 中释放，调用需要派发到该 COM+组件相依的线程。由于 Finalizer thread 的随机性，所以当 Finalizer thread 运行的时候，无法保证派发到的目标线程在一个合理的状态去释放 COM+。如果目标线程正在等 GC 完成，deadlock 很可能就会发生。

其实上面的分析并不是孔穴来风。下面这几篇文章，有直观的说服力：

The system memory usage and the handle counts increase more than you may expect when your application contains components that are derived from the System.EnterpriseServices.ServicedComponent class
http://support.microsoft.com/kb/312118/en-us
“To resolve this problem, follow the common language runtime coding pattern by calling the **Dispose** method and the **Dispose** object when you finish with the managed objects that support **IDisposable**.”

BUG: Multithreaded applications can deadlock because of asynchronous cleanup
http://support.microsoft.com/kb/327443/en-us
“You must call the **Dispose** method when you have finished using **ServicedComponent** objects.”

FIX: Various Problems When You Call Transactional COM+ Components from ASP.NET
http://support.microsoft.com/kb/318000/en-us
“You must explicitly call the **Dispose** method on objects that inherit from the **System.EnterpriseServices.ServicedComponent** class. Under stress, the handle count can increase by up to several thousand (10,000 to 30,000) before being freed if you do not explicitly call **Dispose**.”

有了这些理解和信息后，看看客户的程序中是否堆积了大量的 ServicedComponent 在 Finalizer queue 中：

```
0:016> !finalizequeue
SyncBlock to be cleaned up: 22196
-----------------------------------
MTA interfaces to be released: 0
Total STA interfaces to be released: 0
-----------------------------------
------------------------------
Heap 0
generation 0 has 0 finalizable objects (0x22e12330->0x22e12330)
generation 1 has 0 finalizable objects (0x22e12330->0x22e12330)
generation 2 has 16,658 finalizable objects (0x22e01ee8->0x22e12330)
Ready for finalization 4 objects (0x22e12330->0x22e12340) - Freachable queue
------------------------------
Heap 1
generation 0 has 0 finalizable objects (0x2277e810->0x2277e810)
generation 1 has 0 finalizable objects (0x2277e810->0x2277e810)
generation 2 has 14,848 finalizable objects (0x22770010->0x2277e810)
Ready for finalization 8 objects (0x2277e810->0x2277e830) - Freachable queue
All Finalizable Objects Statistics:
       MT     Count     TotalSize Class Name
0x02249de8         1           12 System.Web.Configuration.ImpersonateTokenRef
0x126285fc         1           24 System.CodeDom.Compiler.TempFileCollection
…
0x11822950       502       30,120 System.Data.SqlClient.SqlConnection
0x79bf4a4c     1,939       46,536 System.Threading.Timer
0x1186727c     1,251      100,080 System.Data.DataSet
0x79bcbc50     8,489      135,824 System.WeakReference
0x11ad1c58     2,525      161,600 System.Data.SqlClient.SqlCommand
0x11868430     2,631      178,908 System.EnterpriseServices.ServicedComponentProxy
```

```
0x1186cf38       773      179,336 System.Data.DataTable
0x11862cd4     4,497      179,880 System.Web.UI.WebControls.Style
0x11bd3238     6,690      856,320 System.Data.DataColumn
Total 31,520 objects, Total size: 1,963,104
```

这里不仅仅看到有 2 千多个 ServicedComponent 没有调用 Close/Dispose 释放外，还看到很多 SqlConnection 在 Finalizer queue 中。看来除了 ServicedComponent 没有及时释放外，连 SqlConnection 用完后都没有及时 Close

由此看来，客户已经足够幸运了，程序不过是有 high CPU, contention 和性能问题而已。程序中现有的问题足够 deadlock 好多次了。在修改代码，及时调用 Dispose 方法后，问题再也没有发生。

小结:
首先从性能日至中看到问题跟 contention 相关，然后通过检查 callstack 找到 contention 具体跟 COM+操作相关。再通过仔细分析和推敲，找到 COM+操作和 GC 的关系，最后结合所有的信息一起思考弄清楚了问题的关键，最后通过 KB 上的文章证实了想法，也找到了问题的解决方案。下面一片文章，总结了 ASP.NET 上的危险区域：

Nine tips for a healthy "in production" ASP.NET application
http://blogs.msdn.com/dougste/archive/2006/04/05/568671.aspx

Profiler

在前面的例子中，性能问题的表征都非常明显。从性能日志中能够宏观地对问题进行测量，通过有效的 dump 分析，往往可以看到问题的根源。但是在另外一些场合，需要对程序的性能进行微观统计，比较和分析。Profiler 则是专门用来解决这一类问题的工具。

考虑下面一些典型的需求

1. 统计一个核心函数在整个程序中被调用到的频率以及对整体性能的影响
2. 测量程序的性能主要是受哪些函数影响，以便有针对性地调优
3. 分析比较一个问题的两种不同算法性能上的优劣

Profiler 是一类性能测量工具,能够微观地测量程序中不同函数的执行时间,被调用到的情况。同时可以用多种方式来显示测量得到的性能信息。跟性能日志不一样，性能日志是宏观的，而 profiler 是微观的。Porfiler 往往以函数，模块为单位进行性能测量。

从程序的设计上说,如果在每一个函数的入口和结束的地方都有相应代码来记录该函数的执行时间，在程序运行完成后应该是可以拿到完整的统计信息。对于非托管代码，记录性能的代码需要手动添加,或者借助编译器来完成。对于托管代码,由于 Framework 提供了 reflection 的机制，同时托管代码本身是带类型信息的，所以可以设计出非常通用的 profiler 程序。

Visual Studio 2005 企业版集成了托管代码的 profiler 工具。下面是一个典型例子。

[案例分析，foreach 和 for loop 性能的区别]

问题背景

使用.NET Framework 2.0 开发的 WinForm 程序。程序过程中动态创建很大的 DataTable 对象，然后对 DataTable 对象中的每一个数据进行分析。在循环读取每一个数据的时候，发现使用 index 访问和使用 foreach 访问，效率上有很大差别。

简化后的重现问题的代码如下:

```
using System;
using System.Data;

namespace DTPerf
{
    class Program
    {
        static DataTable tb = new DataTable();
        static int maxcolumn = 2000;
        static int maxrow = 30000;
        static int count = 0;
        static void buildtb()
        {

            object[] rowsample = new object[maxcolumn];
            for (int i = 1; i <= maxcolumn; i++)
            {
                tb.Columns.Add(i.ToString(), typeof(string));
                rowsample[i - 1] = i.ToString();
            }
            for (int i = 1; i <= maxrow; i++)
            {
                tb.Rows.Add(rowsample);
            }
        }
        static void useobj(object o)
        {
            if (count%(maxcolumn *maxrow/7)==0 )
            {
                Console.WriteLine(o.ToString());
            }
            count++;
        }
        static void AccessWithIndex()
```

```
        {
            DateTime beg = DateTime.Now;
            count = 0;
            for (int i = 1; i <= maxrow; i++)
            {
                for (int j = 1; j <= maxcolumn; j++)
                {
                    useobj(tb.Rows[i-1][j-1]);
                }
            }
            DateTime end = DateTime.Now;
            TimeSpan dif = end - beg;
            Console.WriteLine("AccessWithIndex uses "+dif.Seconds+ " seconds");
           // Console.WriteLine(count);
        }
        static void AccessWithForEach()
        {
            DateTime beg = DateTime.Now;
            count = 0;
            foreach(DataRow rw in tb.Rows )
            {
                for (int j = 1; j <= maxcolumn; j++)
                {
                    useobj(rw[j - 1]);
                }
            }
            DateTime end = DateTime.Now;
            TimeSpan dif = end - beg;
            Console.WriteLine("AccessWithForEach uses " + dif.Seconds + " seconds");
          //  Console.WriteLine(count);

        }
        static void Main(string[] args)
        {
            buildtb();
            AccessWithIndex();
            AccessWithForEach();

        }
    }
}
```

运行程序后看到的结果是:

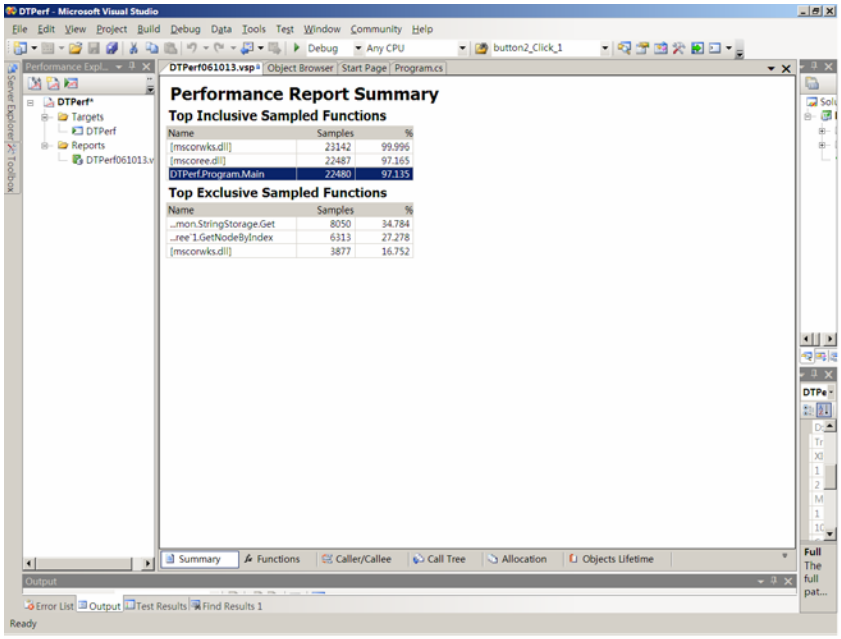
```
1
1429
857
285
1713
1141
569
1997
AccessWithIndex uses 48 seconds
1
1429
857
285
1713
1141
569
1997
AccessWithForEach uses 22 seconds
Press any key to continue . . .
```

两者相差 50%

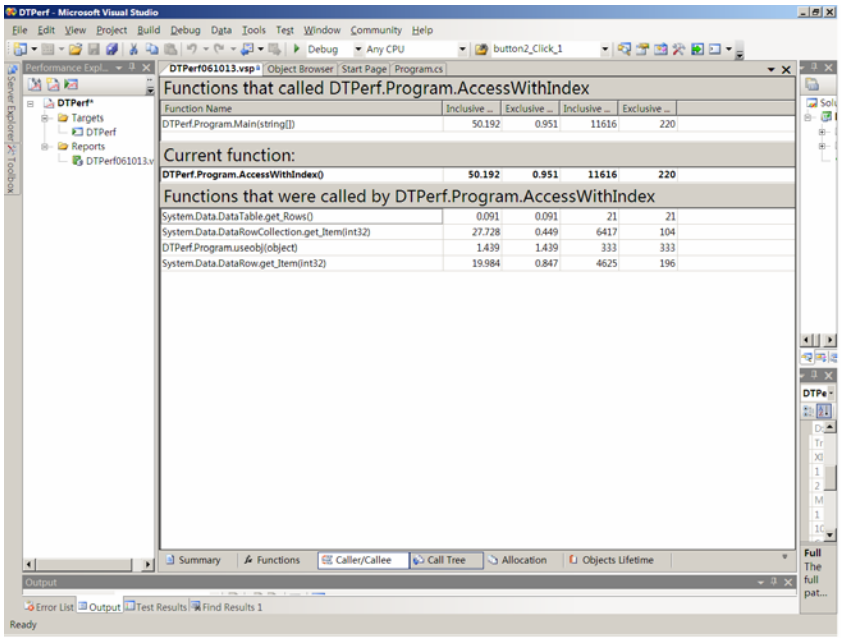
上面的代码已经经过简化。接下来，一种方法是通过 reflector 直接检查两个函数的差异。另外一种方法是使用 profiler。两种方法排错的体验完全不一样。

使用 profiler：

在 VS 2005 企业版中打开该工程。然后点击菜单 Tools -> Performance tools -> Performance Wizard. 一路 next 下去，就可以看到 Performance Explorer 窗口。右键点击 Performance Exporer 窗口中的根节点，选择 Launch 开始运行。等待运行完成后，VS IDE 中直接可以看到性能的 Summary, 如图 1

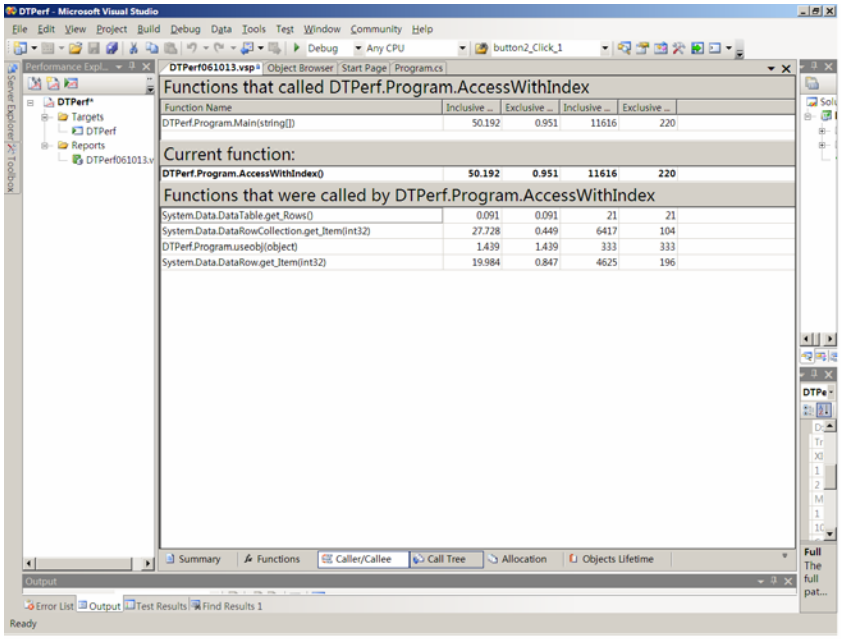


切换到 functions 的统计信息



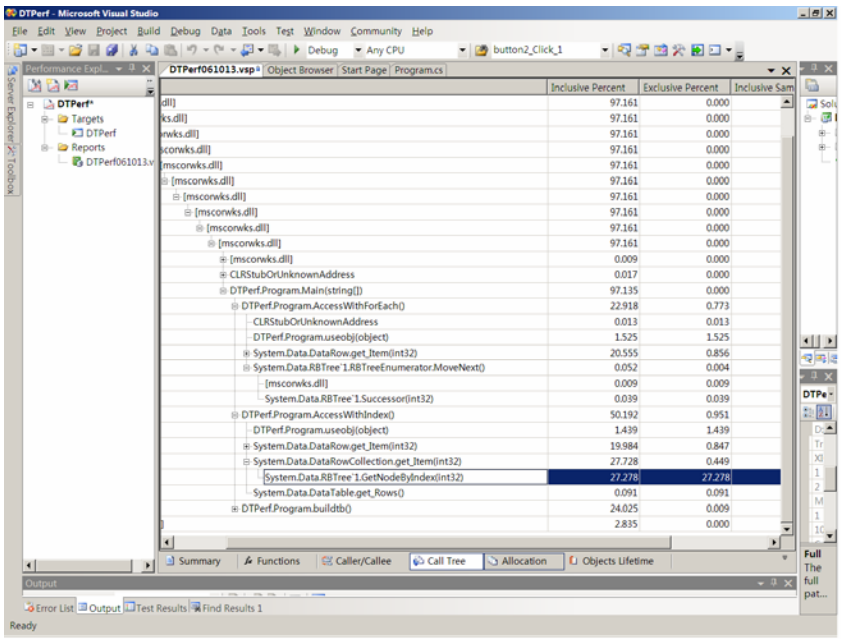
这里可以清楚地看到 AccessWithForEach 跟 AccessWithIndex 函数对程序整体性能的影响。系统 23%的时间花在了 AccessWithForEach 函数上，而 AccessWithIndex 花费了超过 50%的程序运行时间。

为了进一步分析 AccessWithForEach 的开销，双击 AccessWithForEach 函数，来到 Caller/Callee 项。



这里例举了 AccessWithForEach 使用到的函数，可以看到 AccessWithForEach 函数的开销主要是由另外两个 get_item 函数带来的。

上面三项的统计信息，在第四项 call tree 中全部都可以看到。切换到 call tree，



一路展开下来，把 AccessWithForEach 和 AccessWithIndex 都展开后比较。可以看到，这两个函数都调用了 DataRow.get_Item，而且 DataRow.get_Item 在两个函数中的开销都是 20%，非常一致。性能的区别在于 AccessWithForEach 调用了 MoveNext 函数，而 AccessWithIndex 调用了 DataRowCollection.get_Item 函数。前者的开销不到 0.1%，而后者的开销高达 27%. 所

以，两者函数的性能差异在于 AccessWithForEach 使用的遍历函数 MoveNext 比 AccessWithIndex 的便利函数快

更细致地比较 MoveNext 和 get_Item 的差别。前者使用了 RBTree`1.Successor，后者使用了 RBTree`1.GetNodeByIndex. 这里 RBTree`1 表示 RBTree 是一个 generic 类型，该类型的 Successor 函数比 GetNodeByIndex 函数快。RBTree 其实是指红黑树。看来 CLR 2.0 使用了红黑树作为数据结构来保存 DataRow 的 index,以减小删除和添加 DataRow 时候的开销。

有了上面的信息后，再使用 reflector 做有针对性地检查就方便多了。

面对细节上的性能问题，首先是尽可能地简化程序。然后通过反复调用的方式让问题明显化。如果能够使用 profiler 工具，用 profiler 工具帮助分析信息。

本章小结

本章分析了如何排查性能问题。高 CPU，hang 和 contention 都属于性能问题。解决性能问题的总体步骤是:

1. 观察性能相关的指标是否满足预期值。比如 CPU 利用率和程序相应时间
2. 分析性能日志获得宏观认识
3. 制定步骤，在问题发生的关键点抓取 dump 文件。比如 CPU 100%的时候，系统没有响应的时候
4. 分析 dump 来获取问题线索
5. 有条件的时候结合 profiler 工具完成细节统计和比较，简化分析过程

题外话和相关讨论

Task manager 跟 performance monitor 的差别

前面提到在 performance monitor 中可以通过 private bytes 和 virtual bytes 来衡量程序的内存使用。在 task manager 中，也有 Memory Usage 和 VM Size 两项。但是仔细比较会发现 Memory Usage 并不是对应 private bytes, 而 VM Size 也不是对应 virtual bytes。其实，task manager 中的 Memory Usage 对应的是 working set, VM Size 对应的是 private bytes. 所以如果使用 task manager 观察内存使用，应该注意这个差别

有趣的一个问题是，working set 指目前程序所消耗的物理内存，private bytes 指 commit 的内存，为何有的进程的 working set 比 private bytes 还大。要回答这个问题，需要仔细看看两者的定义：

“Working Set refers to the number of pages of virtual memory committed to a given process, both shared and private.”

“Private Bytes is the current size, in bytes, of memory that this process has allocated that cannot be shared with other processes.”

所以 working set 包含了可能被其它程序共用的内存，而 private bytes 只包括只能被当前进程使用的内存。DLL 是一个典型的可能被其它程序共用的资源。DLL 的加载使用文件映射，包含 DLL 的物理内存可以被同时映射到多个进程上。所以进程中加载 DLL 的内存可以算到了 working set 上，而不能被算到 private bytes 上。

性能监视器的牛逼用法

解决 hang 的问题需要在问题发生的时候抓到 dump。如果一个 hang 的问题很久才发生一次，你会不会很郁闷？你会不会独自在公司输入好 adplus 的命令，然后一直等啊等，可是无论你等多久，问题就是不发生。好不容易问题发生一次，不过当时你有去了 WC，错过了抓 dump 的机会

其实，性能监视器除了可以监视性能状态外，还可以指定当某个计数器到一个指定值的时候，执行一项自定义功能。同事 Leo 发明了用性能监视器来自动获取 dump 的方法，具体步骤如下。

问题背景是 IIS 偶尔发生 ASP.NET hang 的情况，导致堆积的请求过多，服务器最后发生 503 Server too busy 错误。在性能监视器中，ASP.NET 堆积的请求在 Request in application queue 计数器中有记录,通过下面的步骤配置性能监视器，当 Request in application queue 的值超过一定数目的时候，自动抓 dump:

1. 创建 CaptureDump.vbs, 复制到 c:\
2. 修改 CaptureDump.vbs 中相应的目录名.
3. 打开性能监视器.
4. 展开 Performance Logs and Alerts -> Alerts.
5. 新建一个 alert, 并设置以下信息:

    将相应的 Request in application queue 计数器加入. 条件可以设为> 60.
    取样间隔为 60 秒.
    在 Action 标签栏上, 选择 Log an entry in the application event log.和 Run this program.
    在 run this program 框中, 输入 C:\WINDOWS\system32\cscript.exe
    点击 Command Line Arguments 按钮.
    只选择 Text message 框.
    在 Text message 框中, 输入 c:\capturedump.vbs.
    在 General 标签栏上, 输入 Run As 的管理员的用户名和密码.
    点击 OK.
    启动这个 Alert.

CaptureDump.vbs 的详细内容:

On Error Resume Next

Dim o

Err.Clear

Set o = WScript.CreateObject("WScript.Shell")

If Err Then

Log "CreateObject - WScript.Shell. Error # " & CStr(Err.Number) & " " & Err.Description

End If

'3rd parameter is 0 means: Hides the window and activates another window.

'4th parameter is true means: script execution halts until the program finishes

Err.Clear

o.Run "c:\debuggers\adplus.vbs -quiet -hang -pn w3wp.exe -o c:\dumps", 0, true

---

其实自动获取 dump 的终极法宝还有另外一个，将在最后一章介绍。

C++跟 C#到底谁快

比较两种语言的优劣永远是敏感的话题。博客园讨论过一个典型的性能比较. 大家先猜测一下，分别用 C++和 C# 调用 pow(2) 函数做平方运算，C# 比 C++快还是慢？快多少？慢多少？为什么呢？

然后再参考下面两篇讨论，看看跟自己的想法是否一样:

托管代码和非托管代码效率的对比
http://www.cnblogs.com/wuchang/archive/2006/12/07/584997.html

真相，看问题的层次
http://eparg.spaces.live.com/blog/cns!59BFC22C0E7E1A76!2274.entry

没有 profiler 怎么办

前面体验了 VS 2005 企业版中 profiler 工具的利害。但是对于 unmanaged 代码，没有合适通用的 profiler 程序。如果要对这类程序分析某一个函数的性能问题就不是那么直接了当。

排查问题的首先步骤还是简化程序，然后通过循环调用来让问题更加明显。比如需要分析的目标函数叫做 foo(), 那么可以创建一个死循环来调用 foo():

```
while(1){foo();}
```

在死循环执行的过程中，随机地抓 hang dump (或者 live debug)。在 windbg 中打开检查 callstack，根据执行时间多的 callstack 出现频率肯定高的道理，统计这 10 套 dump 中类似的 callstack 出现的频率。下面是用这个方法检查 AccessWithIndex 例子的结果:

修改代码的 main 函数为:

```
static void Main(string[] args)
{
    buildtb();
    Console.WriteLine("AccessWithIndex begins!");
    while (true)
    {
        AccessWithIndex();
    }
    //AccessWithForEach();

}
```

当 AccessWithIndex 开始执行的时候，加载 windbg。随机地用 ctrl+c 停下来检查 callstack:

```
(13a0.554): Break instruction exception - code 80000003 (first chance)
eax=7ffd6000 ebx=00000001 ecx=00000002 edx=00000003 esi=00000004 edi=00000005
eip=7c822583 esp=004fffcc ebp=004ffff4 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000246
ntdll!DbgBreakPoint:
7c822583 cc              int     3
0:004> .load C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\sos
0:004> ~0s
*** WARNING: Unable to verify checksum for
C:\WINDOWS\assembly\NativeImages_v2.0.50727_32\System.Data\5abff844071bdb47a4fde3534bc6dd74\System
.Data.ni.dll
eax=00000264 ebx=0013f4ac ecx=06fa4760 edx=06fa4760 esi=00200268 edi=00000020
eip=651c033d esp=0013f40c ebp=013f2c8c iopl=0         nv up ei ng nz na po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000283
System_Data_ni!__dyn_tls_init_callback+0x31e3d:
651c033d 3b4204          cmp     eax,dword ptr [edx+4] ds:0023:06fa4764=00000400
0:000> !clrstack
OS Thread Id: 0x11ac (0)
ESP      EIP
0013f40c 651c033d System.Data.RBTree`1[[System.__Canon, mscorlib]].GetNodeByIndex(Int32)
0013f428 651bc31f System.Data.DataRowCollection.get_Item(Int32)
0013f430 011902ad DTPerf.Program.AccessWithIndex()
0013f47c 01190103 DTPerf.Program.Main(System.String[])
0013f69c 79e88f63 [GCFrame: 0013f69c]
```

```
0:000> g
(13a0.126c): Break instruction exception - code 80000003 (first chance)
eax=7ffd6000 ebx=00000001 ecx=00000002 edx=00000003 esi=00000004 edi=00000005
eip=7c822583 esp=004fffcc ebp=004ffff4 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000246
ntdll!DbgBreakPoint:
7c822583 cc              int     3
0:003> ~0s
eax=0f5b3770 ebx=0013f4ac ecx=0f5abba8 edx=0f5abba8 esi=002203de edi=00000002
eip=651c0375 esp=0013f40c ebp=013f2c8c iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
System_Data_ni!__dyn_tls_init_callback+0x31e75:
651c0375 85f6            test    esi,esi
0:000> !clrstack
OS Thread Id: 0x11ac (0)
ESP      EIP
0013f40c 651c0375 System.Data.RBTree`1[[System.__Canon, mscorlib]].GetNodeByIndex(Int32)
0013f428 651bc31f System.Data.DataRowCollection.get_Item(Int32)
0013f430 011902ad DTPerf.Program.AccessWithIndex()
0013f47c 01190103 DTPerf.Program.Main(System.String[])
0013f69c 79e88f63 [GCFrame: 0013f69c]
0:000> g
(13a0.778): Break instruction exception - code 80000003 (first chance)
eax=7ffd6000 ebx=00000001 ecx=00000002 edx=00000003 esi=00000004 edi=00000005
eip=7c822583 esp=004fffcc ebp=004ffff4 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000246
ntdll!DbgBreakPoint:
7c822583 cc              int     3
0:003> ~0s
eax=000003e2 ebx=0013f4ac ecx=0f5dbd1c edx=0f5dbd1c esi=002403e4 edi=00000024
eip=651c0340 esp=0013f40c ebp=013f2c8c iopl=0         nv up ei ng nz na pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000287
System_Data_ni!__dyn_tls_init_callback+0x31e40:
651c0340 0f8387ae1c00    jae     System_Data_ni!__dyn_tls_init_callback+0x1fcccd (6538b1cd) [br=0]
0:000> !clrstack
OS Thread Id: 0x11ac (0)
ESP      EIP
0013f40c 651c0340 System.Data.RBTree`1[[System.__Canon, mscorlib]].GetNodeByIndex(Int32)
0013f428 651bc31f System.Data.DataRowCollection.get_Item(Int32)
0013f430 011902ad DTPerf.Program.AccessWithIndex()
0013f47c 01190103 DTPerf.Program.Main(System.String[])
0013f69c 79e88f63 [GCFrame: 0013f69c]
0:000> g
(13a0.12d0): Break instruction exception - code 80000003 (first chance)
eax=7ffd6000 ebx=00000001 ecx=00000002 edx=00000003 esi=00000004 edi=00000005
```

```
eip=7c822583 esp=004fffcc ebp=004ffff4 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000246
ntdll!DbgBreakPoint:
7c822583 cc              int     3
0:003> ~0s
eax=00000380 ebx=0013f4ac ecx=06fa4620 edx=00000020 esi=00200380 edi=0013f430
eip=651c02fd esp=0013f40c ebp=013f2c8c iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
System_Data_ni!__dyn_tls_init_callback+0x31dfd:
651c02fd 3b5104          cmp     edx,dword ptr [ecx+4] ds:0023:06fa4624=00000040
0:000> !clrstack
OS Thread Id: 0x11ac (0)
ESP      EIP
0013f40c 651c02fd System.Data.RBTree`1[[System.__Canon, mscorlib]].GetNodeByIndex(Int32)
0013f428 651bc31f System.Data.DataRowCollection.get_Item(Int32)
0013f430 011902ad DTPerf.Program.AccessWithIndex()
0013f47c 01190103 DTPerf.Program.Main(System.String[])
0013f69c 79e88f63 [GCFrame: 0013f69c]
```

可以看到，一共停下来 4 次，每次停下来的时候正在执行的 CPU 代码不尽相同，但是对应的 callstack 都是一样的。根据这个信息可以归纳出程序大多数时间都在执行 RBTree 的 GetNodeByIndex 函数。所以 RBTree 的使用是理解这个问题的关键

RBTree:

在.NET Framework 1.1 中并不支持 generic, 那 CLR1 是如何存储 DataTable 的行的呢?CLR1 根 CLR2 两种存储方式各自有什么优缺点呢?从前面的分析可以看到 IndexAccess 跟 ForEachAccess 使用不同的方法来获取下一行数据.那么 ForEachAccess 能够保证访问的顺序跟 IndexAccess 一致吗?如果把相同的代码拿到 CLR1 下运行,性能表现又如何呢?这些有趣的问题就留给读者作为消遣吧.

!finddebugtrue

在 ASP.NET 1.1 中,如果 web.config 中设定为 debug 模式,每一个 ASPX 页面都会被编译成一个独立的 assembly. 如果站点包含了很多 ASPX 页面,程序中就会有很多 assembly 生成,会带来严重的内存分片和性能问题.在 sos extention 中,可以用!finddebugtrue 来快速寻找这样的 assembly. 详细讨论请参考:

Who is this OutOfMemory guy and why does he make my process crash when I have plenty of memory left?
http://blogs.msdn.com/tess/archive/2005/11/25/496898.aspx

4.4 资源泄露

资源泄露是指程序运行中消耗的资源越来越多，比如内存使用持续增长，handle 数量持续增长.跟性能问题和崩溃相比，资源泄露更为常见。使用性能监视器观察长时间繁忙运行的 IE/MSN Messenger/QQ 等进程，无一例外地都会发现内存使用和 handle 使用总体呈现向上的趋势。

在排错的过程中，跟崩溃和性能比较，资源泄露的优先级往往比较低。因为资源泄露往往不会导致程序当前的运行有太大问题。一般来说，资源泄露的影响在繁忙的服务器上才有明显表现。

由于内存泄露是资源泄露的绝对典型，所以后面主要用内存泄露来说明。

泄露分轻重缓急

首先，用户态进程一旦退出，该进程所独占的资源都回被系统强制回收。即便有内存泄漏,“泄露”掉的内存在进程结束后立刻得到归还。对于桌面程序来说，客户往往不会运行一个进程很长时间。所以即便有内存泄露，当程序退出的时候，问题也就随之消失了。其次，对于服务器进程来说，很多程序都有 recycle 机制，比如 IIS。这些进程也会定时重新启动。同时，很多管理员也有定期重启系统的习惯。所以总体来说，资源泄露给不同程序带来的影响是不同的。

但是，如果服务器进程一直比较繁忙，资源泄露带来的后果往往是致命的。这种情况下，进程消耗的资源持续上升，导致系统剩余资源持续减少，带来的后果有两个：

1. 对系统中的其它进程造成影响。比如内存泄露会导致系统不得不使用大量的磁盘页面充当虚拟内存。频繁的页面交换会导致整个系统的性能受到影响
2. 对于发生内存泄露的进程来说，由于可管理的内存地址空间只有 2GB，当内存使用上升到某一个阈值的时候，会发生 OutOfMemory 错误，无法再继续分配内存。接下来进程往往会异常退出，导致服务受到影响和数据丢失。

所以资源泄露在不同情况下带来的影响也是不一致的。总的来说，资源泄露是指：程序消耗的资源，即便是在程序稳定运行的时候，也无法保持在一个相对稳定的区域。资源使用总体上呈现持续上升的情形，最终给整个系统带来不利影响，不得不通过非正常手段(比如杀死进程)来强制资源回收

要点是：

1. 要区分泄露跟缓存。缓存是指程序预先分配内存以便提高整体性能。由于缓存也是不会及时释放的资源，所以很容易误会为泄露。比如可以配置 SQL Server 的缓存设置，尽可能地让 SQL Server 管理更多内存来提高整体性能。区分缓存跟泄露的关键在于，缓存在程序稳定运行后会保持相对稳定，不会继续上升。

2. 要考虑程序的负荷。如果程序的内存使用随着程序的负荷变化，也是正常的。不能指望既要马儿跑，又要马儿不吃草。负荷大，消耗大也是合理的。所以观察泄露因该是在程序负荷稳定的时候观察
3. 要考量内存泄露带来的后果。对于复杂的桌面程序，比如 IE,由于极大地依赖于用户的具体操作，运行的环境(比如各种复杂的 HTML 页面以及可以加载到 IE 上的插件)不可能完全在程序设计者考虑范围以内，所以某些情况下的内存泄露是难免的。比如在 IE6 上，如果 IFrame 里面包含 javascript，就会有少量的内存泄露。对于这样的程序来说，首先泄露的量不大，繁忙运行几小时可能才泄露 50MB 左右，并不会带来非常明显的影响；其次对于桌面程序来说，重新启动程序是很平常的用户操作。所以内存泄露的影响可以很简单地扼杀掉。最后，对于复杂程度特别高的桌面程序来说，丝毫没有内存泄露是 Mission Impossible。在已经很复杂的程序上用更多地代码来管理内容，会导致” 多做之错”。所以，在检查内存泄露的时候，应该对不同程序，泄漏的多少区分对待
4. 目前很多程序都提供了 recycle 功能，比如 IIS。Recycle 是指程序合理的自动重启。这是某种程度上解决资源泄露的合理方法。既能够保证服务不受影响，干净的新进程又能够避免潜在的资源泄露问题。

内存泄露的排错

关键点

排察资源泄露的关键点在于，泄漏了什么，谁分配的，为什么无法释放。

泄露了什么是指泄露掉的资源的具体类型。就内存泄露来说，因该弄清楚泄露掉的是 Memory Space, Memory Page 还是 Heap。 具体的内存地址是什么范围。

谁分配的指申请这块资源时候的 callstack。这是弄清楚资源来历的最直接方式

为什么无法释放是指资源驻留在程序中的原因。比如忘记在合适的地方调用内存释放函数，内存指针在运行过程中受到破坏，或者还有 object reference 导致 GC 无法回收内存。

“谁分配的”这个问题是排错的关键。特别是对于 unmanaged 程序，不使用后面介绍到的一些奇巧淫技，很难找到分配时候的 callstack。对于 managed 程序来说，由于所有的资源都受到带类型描述的 CLR 管理，所以问题解决起来要轻松不少。

从 Win32 的角度来说，内存分配最终会通过 VirtualAllocEx 来完成。VirtualAllocEx 可以首先预留内存地址空间，然后再把内存映射到分配好的空间上去。通过性能监视器中的 Virtual Bytes 和 Virtual Bytes，可以观察这两种内存的使用情况

除了 VirtualAllocEx 以外，更多情况下 unmanaged 程序会在 Heap 上分配内存。Heap 分配往往通过 HeapAlloc API 完成。不同开发语言有不同的调用方式。比如 C/C++通过 malloc/new 分配。在性能监视器中，并没有计数器表示 heap 的大小。

对于 CLR 程序，内存是 CLR Runtime 使用 VirtualAllocEx 分配整块内存作为 CLR Heap 管理，提供给 CLR 代码分配。在性能监视器中，可以检查# Bytes in all heaps，%Time in GC 和 Loader Heap size 来检查 CLR Heap 的使用情况

定位类型和趋势

借助性能监视器，首先定位泄露内存的类型和整体趋势。

在程序刚运行的时候，添加 Private Bytes 和 Virtual Bytes，如果是 CLR 程序，还可以添加# Bytes in all heaps. 如果是内存泄露，会看到至少其中一个计数器是一条斜向上的曲线。

常见的情况是 Virtual Bytes 跟 Private Bytes 以类似斜率一起上升。这种情况是使用 new/malloc 或者 VirtaulAllocEx 分配内存页面后没有释放，所以内存空间使用和私有内存使用一起上升。大多数程序的内存泄露都属于这一类

如果看到 Virtual Bytes 跟 Private Bytes 一起上升，但是 Virtual Bytes 比 Private Bytes 上升得快，或者 Virtual Bytes 跟 Private Bytes 的比例超过 3:1，说明不仅仅有泄漏，而且泄露还导致了内存碎片。，关于内存碎片的详细解释，请参考第二部分的说明以及

.NET Memory usage - A restaurant analogy
http://blogs.msdn.com/tess/archive/2006/09/06/742568.aspx

导致内存碎片常见的一些原因是:

1. 小块 Heap 内存泄露，导致 Heap Fragmentation
2. CLR 程序加载了大量的 Assembly。比如 ASP.NET 动态编译的页面
3. CLR 程序 pin 住了大量小块内存。比如 socket 同时做 Receive 操作。

对于 CLR 程序，如果#Bytes in all heaps 跟 privates bytes 一致增长，说明泄漏掉的内存属于 managed heap。如果#Bytes in all heaps 保持稳定，说明泄漏掉的内存属于 unmanaged memory。如果#Bytes in all heaps 增长，但是不如 privates bytes 增长得快，说明同时有 managed leak 和 unmanaged leak，或者是 managed heap 发生 fragmentation

一种少见的情况是 Virtual Bytes 上升很快，Private Bytes 相对保持平稳。有可能是程序的调用了 VirtualAllocEx 预留了内存空间，但是一直都没有映射内存上去，发生了 Virtual Space Leak;也有可能是程序加载了大量的 DLL。Virtual Space leak 不会对系统和其他程序造成明显影响。

区分 managed heap leak 和 native leak

接下来分为 managed heap leak 和 native leak 两种情况检查。在 CLR 程序中，由于所有的内存分配都跟具体的类型绑定，从 dump 中可以直接获取整个 managed heap 的统计，所以 managed 程序的内存问题排错起来相对容易。

对于 native leak，由于内存操作直接通过 API 完成，dump 文件给出的是一片片荒芜的二进制地址，最多能够分辨出某个地址是否可读，可写，是否属于 DLL 映射或者是 Heap 的一部分。所以 native leak 的排错是极具挑战性的

下面会通过例子首先解释如何对 native leak 进行排查。然后再讨论 managed heap 上的 leak 和 fragmentation

[案例分析，IE 的 leak]

问题背景

客户新开发的 B/S 办公自动化系统，所有的操作都可以在浏览器上完成。为了支持灵活的 UI，页面中使用了很多 javascript/DHTML。比如 javascript 可以根据客户在某一个页面中前半部分的用户输入，动态生成后半部分的 HTML 表格。

新系统上线后受到广泛的好评。随着使用熟练程度提高，用户每天提交的业务也直线上升。但是问题也在这个时候出现了。某些部门发现使用这个系统几个小时后，客户端机器的性能有明显的下降，比如切换窗口缓慢。客户的 IT 人员检查用户的系统后发现，问题发生的时候 iexplore 进程占用了接近 1GB 的内存。

万里长征的第一步

拿到这个问题后，可以肯定的一点就是 iexplore 的内存泄漏。经过一系列的初步定位后，发现内存泄漏的问题跟某些特定的 DHTML 相关。比如每次动态操作<table>元素生成报表后，哪怕做了对应的释放调用，内存使用还是会增加。根据这个信息，对一部分代码进行简化后，得到可以重现问题的 sample:

Frame.htm:

```
<INPUT Type="Button" OnClick="clkadd()" Value="Click me to add row">
<INPUT Type="Button" OnClick="clkdel()" Value="Click me to del row">
<script>
function clkadd(){
  var i;
```

```
    for(i=0;i<100;i++){
    oRow = oTable.insertRow();
    var j;
        for(j=0;j<4;j++){
        oCell = oRow.insertCell();
        oCell.innerHTML = "<IFRAME FRAMEBORDER=0 width='100%' height='100%'
SRC='hostframe.htm'></IFRAME>"
        }
    }
}

function clkdel(){
    var i;
    for(i=0;i<100;i++){
    oTable.deleteRow(0);
    }
}
</script>
<TABLE name="oTable" id="oTable">
</TABLE>
```

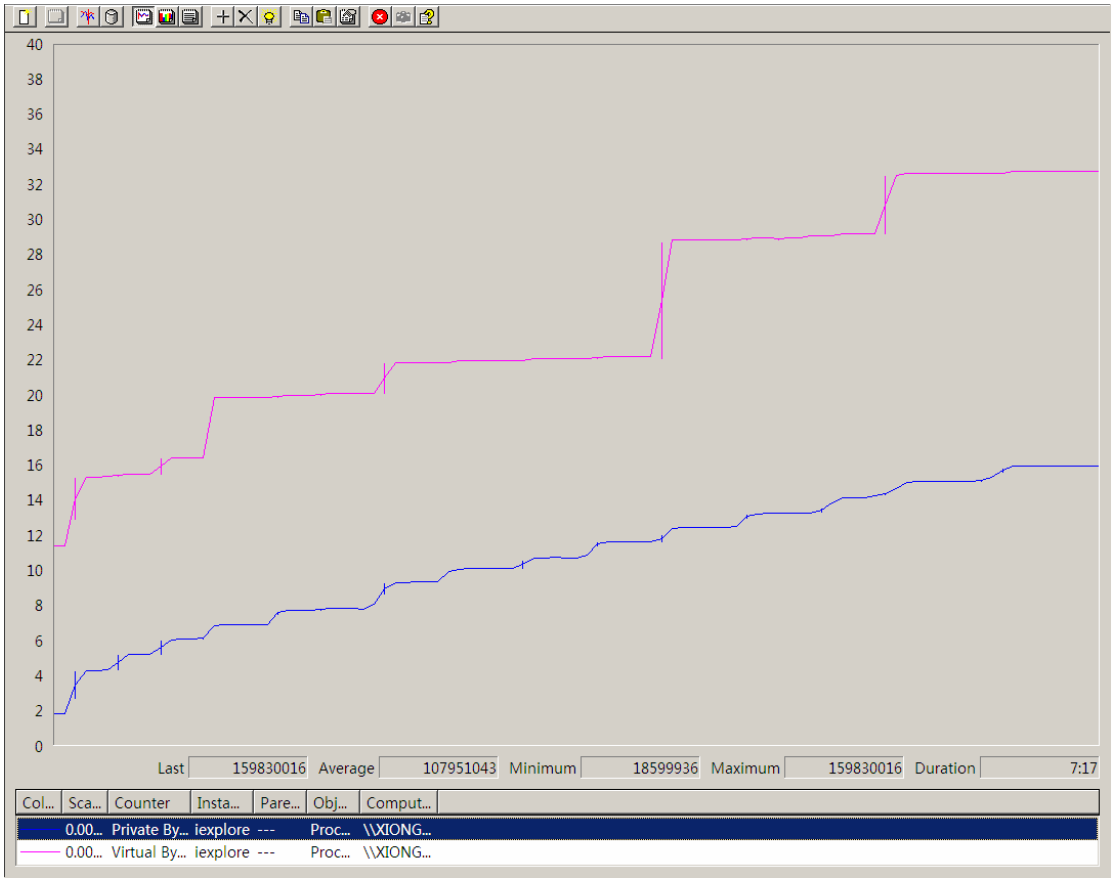
Hostframe.htm:

```
<BODY onload="t.focus()">
  <INPUT TYPE="text" id="testid" value="some value">
</BODY>
```

用 IE 打开 frame.htm，按第一个按钮，动态生成一个 100*4 的 table。按第二个按钮可以删除整个 table。无论使用 IE6 还是 IE7，连续进行一系列的创建/删除操作后，就可以看到 iexplore.exe 进程的内存占用明显上升。

作为内存泄漏的排错例子，接下来会用一系列不同的工具和方法来尽可能多地探寻这个问题的根源。首先会抓取性能日志，然后分析内存分配的 callstack，接下来还会针对 dump 做一些分析，最后讨论解决办法。目的在于演示典型的 unmanaged 内存泄漏问题的完整排错步骤。

首先使用性能监视器观察内存的具体变化。添加 virtual bytes 和 private bytes，连续点击页面中的按钮测试七分钟左右，看到如下的曲线：



从图形中可以看到，private bytes 跟 virtual bytes 一起飙升。Private bytes 从最初的 18.5MB 上升到最后的 159MB。Private bytes 根 virtual bytes 增长的斜率近似，所以是很典型的泄漏，暂时排除 fragmentation 的可能。

对于 unmanaged leak,手中的武器非常有限。在没有 CLR 的世界里，既没有办法检查某内存地址上数据的类型，也无法获取数据之间的引用关系。跟 CLR 程序不同，直接从 dump 能找到的线索非常有限。

前面提到过，排错的终极武器是 log。如果静态的 dump 分析没有帮助，那么可以动态地记录内存的操作。考虑下面这种办法:

每次内存分配的时候记录下分配内存的 callstack，当释放这块内存的时候，也同时释放记录下的 callstack。当问题发生后，残留 callstack 对应的内存都是泄漏掉的。分析 callstack 就可以找到这些泄漏掉的内存是如何分配的

如何在每次内存分配的时候记录下分配内存的 callstack 呢？记录 callstack 很容易，只需要保存发生内存分配时候 ESP 所指向的地址附近的数据，最后通过 symbol 文件就可以找到 callstack。截获内存分配的操作稍微困难，可以在不同的层次来完成:

1. 如果程序使用 C/C++开发，根据第二章的介绍，可以利用 CRT Debug Heap 进行自动分析。甚至可以重载 C/C++中的 operation new, operator new []和 malloc 来实现自己的内存分配函数，以便记录。
2. Windows Heap Manager 也提供了 callstack trace 功能。在激活 pageheap 的 trace 功能后，

通过 HeapAlloc API 发生分配时候的 callstack 被 Heap Manager 自动保存在进程中。通过检查 dump，或者其它的工具能够读取保存下来的 callstack

3. 所有的用户台函数分配都要通过 API 完成。可以通过 API Hook 技术截获目标进程的 Virtual 系列和 Heap 系列内存操作，保存成 log 分析

Pageheap+UMDH

三种方法中第三种实用性最广，但是实现起来难度最大。回头看当前这个 IE Leak 问题。首先没有 IE 的源代码，所以第一种方法就没用。第二种方法只能排查 Heap leak 的情况，所以要用第二种方法的话，可以先抓去 dump，使用!heap 命令检查进程中所有 Heap 的大小，然后跟总体内存使用比较 Heap 所占的百分比:

当 virtual bytes 到 160MB 的时候抓 dump,使用!heap –s 统计 Heap 使用：

```
0:000> !heap -s
LFH Key: 0xd4a587a8
Affinity manager status:
   - Virtual affinity limit 4
   - Current entries in use 3
   - Statistics:  Swaps=10, Resets=0, Allocs=10

  Heap     Flags   Reserv  Commit  Virt   Free  List   UCR  Virt  Lock  Fast
                    (k)     (k)    (k)     (k) length     blocks cont. heap
-------------------------------------------------------------------------------
00140000 00000002  131072  87368  88456   4612   336    27    0     2   LFH
00240000 00008000      64     12     12     10     1     1    0     0
00360000 00001002    1088    536    536     17     6     1    0     0   L
00370000 00001002    3136   1380   1412     53     4     4    0     0   L
01050000 00000002    1024     20     20      2     1     1    0     0   L
013e0000 00001002     256    228    228     64     1     1    0     0   L
01550000 00001003     256    256    256    250     1     0    0   bad
01590000 00001003     256     56     56      4     1     1    0   bad
015d0000 00001003      64      8      8      3     2     1    0   bad
015e0000 00001002      64     32     32     18     1     1    0     0   L
02090000 00001002      64     12     12      1     1     1    0     0   L
03210000 00001002     256     32     32      1     1     1    0     0   L
037f0000 00001002    1024   1024   1024   1016     2     0    0     0   L
03260000 00001002      64     44     44      8     1     1    0     0   L
01b30000 00001002      64     16     16      7     1     1    0     0   L
04430000 00001002      64     48     48     40     2     1    0     0   L
041c0000 00001002      64     16     16      3     1     1    0     0   L
04320000 00001002   64768  34984  34984   1271     0     1    1     0   L
-------------------------------------------------------------------------------
```

从上面的表格可以看到，Heap 140000 和 Heap 4320000 分别 reverse 了 131MB 和 64MB 内存，commit 了 87MB 和 35MB 内存。Commit 的内存总合是 120MB。由此可以看出泄漏的内存几乎都是 Heap

在第二章的中已经介绍过如何利用 pageheap 的 trace 功能检查内存泄漏和内存碎片。当时是随机挑选出 heap pointer，根据统计信息总结出泄漏的地址都是从一个固定的 callstack 分配。这样的做法毕竟不够严谨。完善的做法是采用 pageheap 跟 umdh 结合来自动分析 pageheap 记录下的 callstack。

Umdh 在 windbg 安装目录下就可以找到。对于激活了 heap trace 功能的进程来说，umdh 可以随时提取出进程中记录的所有 heap pointer 的 callstack 保存到 log 文件。Umdh 还可以自动分析 log 文件的差异。所以，在激活 heap trace 后，在增长前用 umdh 抓一次 log, 内存增长后再抓一次 log。然后用 umdh 自动比较就可以得到严谨的统计结果。详细步骤请参考:

Umdhtools.exe： 如何使用 umdh.exe 来查找内存泄漏
http://support.microsoft.com/kb/268343/zh-cn

下面是使用 umdh 排查这个问题的详细步骤:

首先要对 IE 进程激活 heap trace.如果使用 pageheap.exe 工具，可以用:

```
Pageheap /enable iexplore.exe
```

Pageheap 非常方便，但是不够灵活。上面的命令除了激活 heap trace 外，同时还激活了 heap 校验。排查内存泄漏只需要 heap trace 就可以了，heap 校验可能会带来性能上的额外开销。所以这里换一种方法,使用文章中描述的 gflag 工具来激活 heap trace:

```
C:\Debuggers>gflags -i iexplore.exe +ust
Current Registry Settings for iexplore.exe executable are: 00001000
    ust - Create user mode stack trace database
```

当 umdh 读取 callstack 的时候需要 symbol 文件。所以要用下面的命令设定 symbol path

```
C:\Debuggers>set _NT_SYMBOL_PATH=SRV*D:\websymbols*http://msdl.microsoft.com/download/symbols
```

完成上面两步后，就可以开始测试了。首先启动 IE 进程，打开发生问题的页面，然后立刻抓取一次 heap trace log:

```
C:\Debuggers>umdh -p:5728 -f:ie1.log
```

这里的 5728 是 IE 的进程号，log 文件保存到 ie1.log 中

接下来点几次按钮，让内存上升 30MB 左右。这样保证 IE 完成了必要的 cache，然后抓一次 log:

C:\Debuggers>umdh -p:5728 -f:ie2.log
然后开始狂点按钮，让问题明显化。当内存上涨到 160MB 左右的时候，抓取第三次 log:

C:\Debuggers>umdh -p:5728 -f:ie3.log

Ie3.log 有 10MB 左右。打开 ie3.log 文件可以看到，文件头列出了当前所有的 Heap Handle:

- - - - - - - - - - Heap summary - - - - - - - - - -

00140000
00240000
00360000
00370000
02050000
023E0000
02550000
02590000
025D0000
025E0000
03090000
03050000
04730000
04830000
049A0000
05720000

接下来是所有的保存下来的 callstack。

*- - - - - - - - - - Start of data for heap @ 00140000 - - - - - - - - - -

Flags: 58000062
Entry Overhead: 8
*- - - - - - - - - - Heap 00140000 Hogs - - - - - - - - - -

000000AC bytes in 0x1 allocations (@ 0x0000003C + 0x0000001C) by: BackTrace00002
ntdll!RtlAllocateHeapSlowly+00000041
ntdll!RtlAllocateHeap+00000E9F
ntdll!LdrpAllocateUnicodeString+00000035
ntdll!LdrpCheckForKnownDll+0000017D
ntdll!LdrpMapDll+000000F2
ntdll!LdrpLoadImportModule+0000017C

ntdll!LdrpHandleOneOldFormatImportDescriptor+0000005B
ntdll!LdrpHandleOldFormatImportDescriptors+0000001C
ntdll!LdrpWalkImportDescriptor+000001A5
ntdll!LdrpInitializeProcess+00000E3E
ntdll!_LdrpInitialize+000000D0
ntdll!KiUserApcDispatcher+00000025

000000B0 bytes in 0x1 allocations (@ 0x00000050 + 0x00000018) by: BackTrace00003
ntdll!RtlAllocateHeapSlowly+00000041
ntdll!RtlAllocateHeap+00000E9F
ntdll!LdrpAllocateDataTableEntry+00000031
ntdll!LdrpMapDll+00000462
ntdll!LdrpLoadImportModule+0000017C
ntdll!LdrpHandleOneOldFormatImportDescriptor+0000005B
ntdll!LdrpHandleOldFormatImportDescriptors+0000001C
ntdll!LdrpWalkImportDescriptor+000001A5
ntdll!LdrpInitializeProcess+00000E3E
ntdll!_LdrpInitialize+000000D0
ntdll!KiUserApcDispatcher+00000025
。。。。。。

分析 log 文件的一种方法是根据出现次数对 callstack 进行排序。出现得最多的 callstack 最值得怀疑。另外一种方法是使用 umdh 自动分析两次抓到的 log 文件:

C:\Debuggers>umdh **ie2.log** **ie3.log** > cmp23.txt

打开 cmp23.txt 可以看到:

```
+     36a40 ( 3ebc0 -    8180)       1f allocs  BackTrace20461
+     10cd0 ( 119a0 -     cd0)      276 allocs BackTrace19986
+      d260 (    d680 -     420)       34 allocs       BackTrace19985
+      c69c ( 108d0 -    4234)        4 allocs BackTrace19816
+      5b30 (    6260 -     730)       44 allocs       BackTrace19221
+      4654 (    4aa4 -     450)       45 allocs       BackTrace19215
+      3ba0 (    3da0 -     200)       7a allocsBackTrace19592
+      3580 (    3580 -       0)        8 allocs       BackTrace20575
…
-       140 (       0 -     140)        0 allocsBackTrace13963
-       140 (       0 -     140)        0 allocsBackTrace04660
-       148 (       0 -     148)        0 allocsBackTrace13972
-       148 (       0 -     148)        0 allocsBackTrace04178
-       1a0 (       0 -     1a0)        0 allocs BackTrace19445
-       268 (    1810 -    1a78)        a allocs BackTrace00531
```

Total increase == b8cba

总的来说，分析的结果给出了 callstack 的统计信息，并且按照导致的内存增量排序。数据用 1 6 进制表示。

拿第一行作说明。BackTrace20461 是 callstack 的 ID。在 log 文件中查找 BackTrace20461 就可以找到具体的 callstack 信息：

00002060 bytes in 0x1 allocations (@ 0x00002000 + 0x00000018) by: BackTrace20461
    ntdll!RtlAllocateHeapSlowly+00000041
    ntdll!RtlAllocateHeap+00000E9F
    …

1f allocs 表示 log3 中该 callstack 一共出现了 1f 次。3ebc0 其实是 1f*2060 的近似结果，2060 是该 callstack 每次导致的内存增长量。3ebc0 是 log3 中该 callstack 分配的内存总量。8180 是 log2 中该 callstack 分配的内存总量。差值 36a40 就是该 callstack 导致的内存增长。

需要注意的是，Heap Manager 用来保存 callstack 的 buffer 是有限的。所以无法保证记录下所有的 Heap 分配的 callstack。对 log 的分析应该是统计意义上的，而不能用作具体的量化。所以这里的数据只表示 BackTrace20461 对应的 callstack 相对其它 callstack 来说导致了更多的内存增量，36a40 是一个参考数据，单是并不保证 36a40 就是实际的内存增长。

根据上面的分析，BackTrace20461 和 BackTrace19986 的增量相对其它增量来说要高一个数量级。找到对应的 callstack:

00002060 bytes in 0x1 allocations (@ 0x00002000 + 0x00000018) by: BackTrace20461
    ntdll!RtlAllocateHeapSlowly+00000041
    ntdll!RtlAllocateHeap+00000E9F
    mshtml!_MemAllocClear+00000023
    mshtml!CTxtBlk::InitBlock+00000021
    mshtml!CTxtArray::AddBlock+0000002A
    mshtml!CTxtPtr::InsertRange+00000048
    mshtml!CTxtPtr::InsertRepeatingChar+0000005F
    mshtml!CMarkup::CreateInitialMarkup+00000093
    mshtml!CMarkup::DestroySplayTree+00000370
    mshtml!CMarkup::UnloadContents+000002AB
    mshtml!CMarkup::TearDownMarkupHelper+00000090
    mshtml!CMarkup::TearDownMarkup+00000044
    mshtml!CFrameSite::TearDownFrameContent+0000004F
    mshtml!CElement::PrivateRelease+00000029
    mshtml!CMimeTypes::Release+0000000E
    mshtml!CSpliceTreeEngine::RemoveSplice+0000097E
    mshtml!CMarkup::SpliceTreeInternal+00000092
    mshtml!CDoc::CutCopyMove+000000D8

mshtml!CDoc::Remove+00000017
mshtml!CElement::RemoveOuter+00000063
mshtml!CTableLayout::deleteElement+0000005D
mshtml!CTable::deleteRow+000000AA
mshtml!Method_void_oDolong+0000005C
mshtml!CBase::ContextInvokeEx+000004EF
mshtml!CElement::ContextInvokeEx+00000070
mshtml!CTable::ContextThunk_InvokeEx+000000B5
jscript!IDispatchExInvokeEx2+000000AC
jscript!IDispatchExInvokeEx+00000056
jscript!InvokeDispatchEx+00000078
jscript!VAR::InvokeByName+000000BA
jscript!VAR::InvokeDispName+00000043
jscript!VAR::InvokeByDispID+000000B9

00000070 bytes in 0x1 allocations (@ 0x00000010 + 0x00000018) by: BackTrace19986
ntdll!RtlAllocateHeapSlowly+00000041
ntdll!RtlAllocateHeap+00000E9F
mshtml!_MemAllocClear+00000023
mshtml!CCollectionCache::InitReservedCacheItems+0000006A
mshtml!CMarkup::InitCollections+0000005A
mshtml!CWindow::GetDispID+00000146
mshtml!COmWindowProxy::GetDispID+000000DF
mshtml!COmWindowProxy::subGetDispID+00000017
jscript!IDispatchExGetDispID+0000004B
jscript!GetDex2DispID+00000038
jscript!VAR::GetDispID+000000F2
jscript!CScriptRuntime::GetVarVal+00000031
jscript!CScriptRuntime::Run+000008FC
jscript!ScrFncObj::Call+0000008D
jscript!CSession::Execute+000000A1
jscript!NameTbl::InvokeDef+00000179
jscript!NameTbl::InvokeEx+000000CB
mshtml!CBase::InvokeDispatchWithThis+000001D7
mshtml!CBase::InvokeEvent+000001AD
mshtml!COmWindowProxy::FireEvent+00000146
mshtml!COmWindowProxy::Fire_onload+000000CF
mshtml!CMarkup::OnLoadStatusDone+0000040C
mshtml!CMarkup::OnLoadStatus+0000004C
mshtml!CProgSink::DoUpdate+00000533
mshtml!CProgSink::OnMethodCall+0000000F
mshtml!GlobalWndOnMethodCall+00000101
mshtml!GlobalWndProc+00000181

USER32!InternalCallWinProc+00000028
USER32!UserCallWinProcCheckWow+00000151
USER32!DispatchMessageWorker+00000327
USER32!DispatchMessageW+0000000F
IEFRAME!CTabWindow::_TabWindowThreadProc+00000189

根据上面的分析，成功地找到了可疑的 callstack。对该 callstack 的分析稍后再说，先回顾一下拿到 callstack 的关键。关键点不在于 umdh 工具的使用，而是 Heap Manager 自带的 heap trace。在确定了泄露的内存是 Heap 以后，激活 heap trace 就可以捕捉到 callstack。有了 callstack 后具体如何分析可以根据实际情况而变化。

如果不使用 umdh 工具，直接察看激活了 heap trace 的 dump 文件也是可以的。在抓取 log3 的时候同时用 adplus 抓取 iexplore 的 dump，分析如下:

首先用!heap –p 命令检查 pageheap 的设置:

```
0:000> !heap -p

    Active GlobalFlag bits:
        hpc - Enable heap parameter checking
        ust - Create user mode stack trace database

    StackTraceDataBase @ 004a0000 of size 01000000 with 000050eb traces
    ..
```

Ust 表示 user mode stack trace 已经激活。根据前面!heap -s 命令的输出，发现 heap 00140000 占用内存最多。用!heap 00140000 查看 Heap 140000 所管理的内存块:

```
0:000> !heap 00140000
Index   Address  Name     Debugging options enabled
  1:   00140000
    Segment at 00140000 to 00240000 (00100000 bytes committed)
    Segment at 05500000 to 05600000 (000f6000 bytes committed)
    Segment at 05a50000 to 05c50000 (001d3000 bytes committed)
    Segment at 05c50000 to 06050000 (0035f000 bytes committed)
    Segment at 06050000 to 06850000 (006a7000 bytes committed)
    Segment at 06850000 to 07850000 (00d5e000 bytes committed)
    Segment at 07a50000 to 09a50000 (005ba000 bytes committed)
```

由于 callstack 是跟每一个 heap 指针绑定的，所以关检查内存块没用，需要检查详细的 heap pointer。可以用 `!heap 00140000` 加上-a 参数或者-h 参数来得到。把所有的 heap pointer 打印出来需要一定时间，而且整个输出比床单还长，下面是一个节选:

08675c70: 00760 . 00028 [07] - busy (10), tail fill
08675c98: 00028 . 00028 [07] - busy (10), tail fill
08675cc0: 00028 . 00030 [07] - busy (10), tail fill
08675cf0: 00030 . 00028 [07] - busy (10), tail fill
08675d18: 00028 . 00018 [04] free fill
08675d30: 00018 . 000a0 [07] - busy (88), tail fill
08675dd0: 000a0 . 00040 [07] - busy (20), tail fill
08675e10: 00040 . 00058 [04] free fill
08675e68: 00058 . 00028 [07] - busy (10), tail fill
08675e90: 00028 . 00028 [07] - busy (10), tail fill
08675eb8: 00028 . 00030 [07] - busy (10), tail fill
08675ee8: 00030 . 000a8 [04] free fill

第一行带 busy 标记，表示对应的地址 08675c70 已经分配出去. 由于激活了 heap trace, 用-p -a 命令可以直接获取该地址对应的 callstack:

```
0:000> !heap -p -a 08675c70
    address 08675c70 found in
    _HEAP @ 140000
  in HEAP_ENTRY: Size : Prev Flags - UserPtr UserSize - state
      8675c70: 0005 : N/A  [N/A] - 8675c78 (10) - (busy)
        mshtml!CElementAryCacheItem::`vftable'
      Trace: 4e12
      7c85fc22 ntdll!RtlAllocateHeapSlowly+0x00000041
      7c81d4df ntdll!RtlAllocateHeap+0x00000e9f
      6362b4e0 mshtml!_MemAllocClear+0x00000023
      635e8f3f mshtml!CCollectionCache::InitReservedCacheItems+0x0000006a
      635ea520 mshtml!CMarkup::InitCollections+0x0000005a
      635ad0d4 mshtml!CWindow::GetDispID+0x00000146
      635ad36b mshtml!COmWindowProxy::GetDispID+0x000000df
      635ad2ae mshtml!COmWindowProxy::subGetDispID+0x00000017
      6339d10f jscript!IDispatchExGetDispID+0x0000004b
      6339d15a jscript!GetDex2DispID+0x00000038
      6339ceba jscript!VAR::GetDispID+0x000000f2
      6339b3ec jscript!CScriptRuntime::GetVarVal+0x00000031
      633a1ed2 jscript!CScriptRuntime::Run+0x000008fc
      633a2fcb jscript!ScrFncObj::Call+0x0000008d
      633a31c9 jscript!CSession::Execute+0x000000a1
      633a3f69 jscript!NameTbl::InvokeDef+0x00000179
      633a2eed jscript!NameTbl::InvokeEx+0x000000cb
      635bf023 mshtml!CBase::InvokeDispatchWithThis+0x000001d7
      635bef3c mshtml!CBase::InvokeEvent+0x000001ad
      63619e1d mshtml!COmWindowProxy::FireEvent+0x00000146
      6360ba6f mshtml!COmWindowProxy::Fire_onload+0x000000cf
      6360b7f3 mshtml!CMarkup::OnLoadStatusDone+0x0000040c
      6360b1b6 mshtml!CMarkup::OnLoadStatus+0x0000004c
```

```
6360f2a5 mshtml!CProgSink::DoUpdate+0x00000533
6360abb6 mshtml!CProgSink::OnMethodCall+0x0000000f
6363a3c9 mshtml!GlobalWndOnMethodCall+0x00000101
636511b2 mshtml!GlobalWndProc+0x00000181
7739c3b7 user32!InternalCallWinProc+0x00000028
7739c484 user32!UserCallWinProcCheckWow+0x00000151
7739c73c user32!DispatchMessageWorker+0x00000327
7739c778 user32!DispatchMessageW+0x0000000f
1b23c6e ieframe!CTabWindow::_TabWindowThreadProc+0x00000189
```

上面的命令也打印出了具体的 callstack。除此之外可以看到，8675c70 是一个 Heap pointer 地址，但是该 pointer 并非 HeapAlloc 返回给程序的内存地址。该 pointer 是 Heap Manager 维护的一个地址，真正返回给程序的是 UserPtr 字段对应的 8675c78。

在第二章提到了搜索 Heap Trace 保存的一个快捷方法是寻找 0xdcba。但是在当前 dump 中，这一招没用。因为 Heap Manager 如何维护 Heap，如何保存 stack trace 是 Windows 的内部实现。在不同版本不同情况下情况不尽相同。所以正确的方法还是使用 umdh 工具，或者用最新的 windbg 的!heap 命令检查。

异常强大的 IIS Diagnostics 工具

回到这个案例。前面用 pageheap 的 trace 功能演示了如何找到根泄露相关的 callstack。如果泄露的内存不属于 heap，而是代码用 VirtualAllocEx 直接分配的呢？

前面介绍的第三种方法就使用 API Hook 监视内存操作。IIS Diagnostics 就是使用这种技术的现成工具:

IIS Diagnostics Toolkit (x86)
http://www.microsoft.com/downloads/details.aspx?familyid=9BFA49BC-376B-4A54-95AA-73C9156706E7&displaylang=en

工具的名字带 IIS， 但其实该工具几乎可以用各种场合(crash, performance 或者 leak)，调试各种程序(IIS, Service 或者桌面程序)，这里先介绍如何使用它分析内存泄露，更多功能请参考帮助。

安装完成后，在开始菜单中可以找到 Debug Diagnostics Tool 1.0 工具。该工具的帮助文档包含了该工具个方面的说明。另外还可以参考该工具开发团队的 blog:

http://blogs.msdn.com/debugdiag/

用 IIS Diagnostics Tool 排查内存泄漏的步骤是:

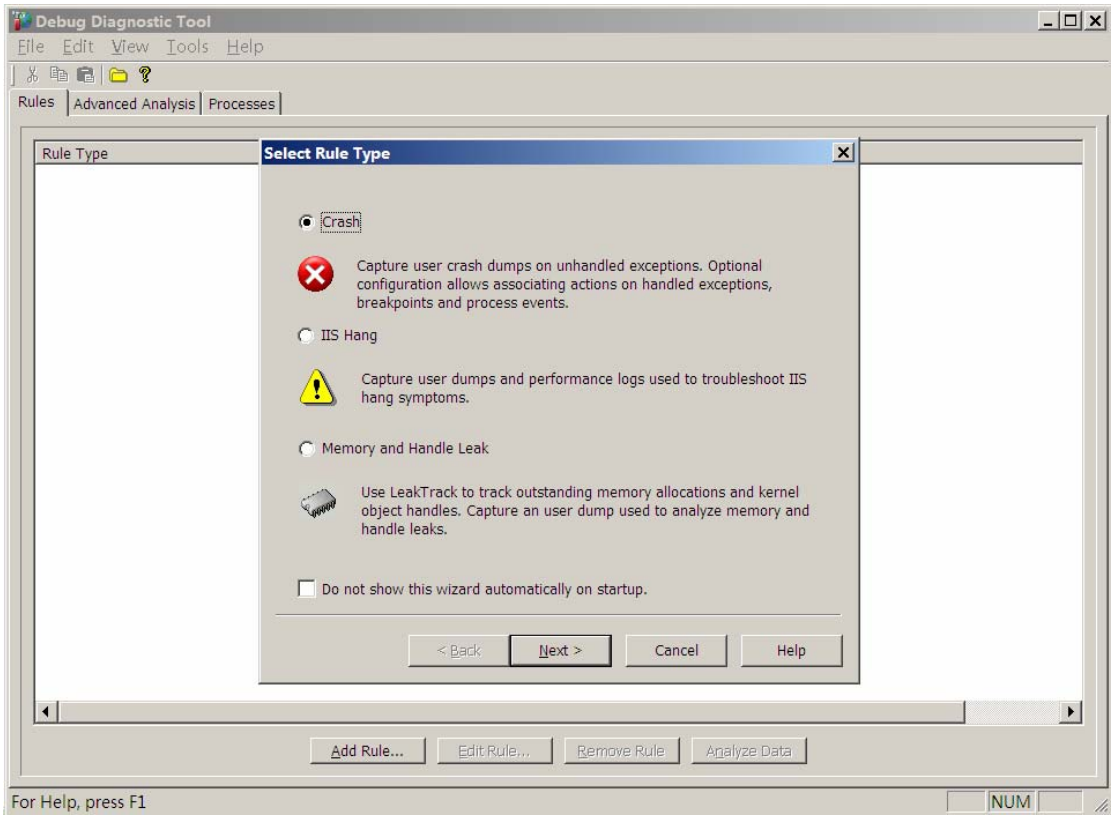
1. 泄露发生前启动这个工具,并且指定目标进程开始监视. IIS Diag 会通过 DLL Injection 的

方法注入 API Hook 所需的 DLL,监视内存分配/释放操作.

2. 重现问题. 在这个过程中, IIS Diag 所注入的 DLL 会把内存分配/释放操作的详细信息记录到目标进程的内存空间中
3. 当内存泄漏比较明显后,用 IIS Diag 工具直接抓取目标进程的 dump 文件.由于内存操作的 log 保存在目标进程内存空间中,所以 Dump 文件中直接包含了 log 信息
4. 通过 IIS Diag 工具自动分析 dump 文件,生成 HTML 格式的详细报表,检查报表获取结论

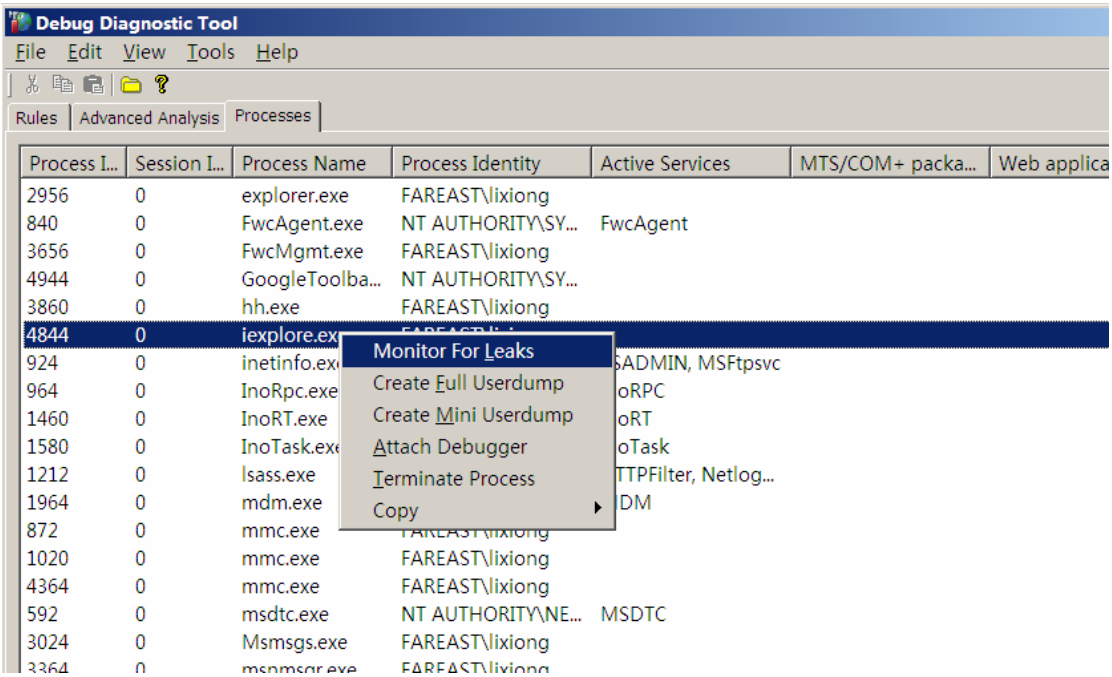
下面是分析 IE leak 的具体步骤:

1. 启动 IIS Diag,看到如下界面:



通过该界面可以配置自动监视的规则.这里不需要使用这些规则,所以选择取消

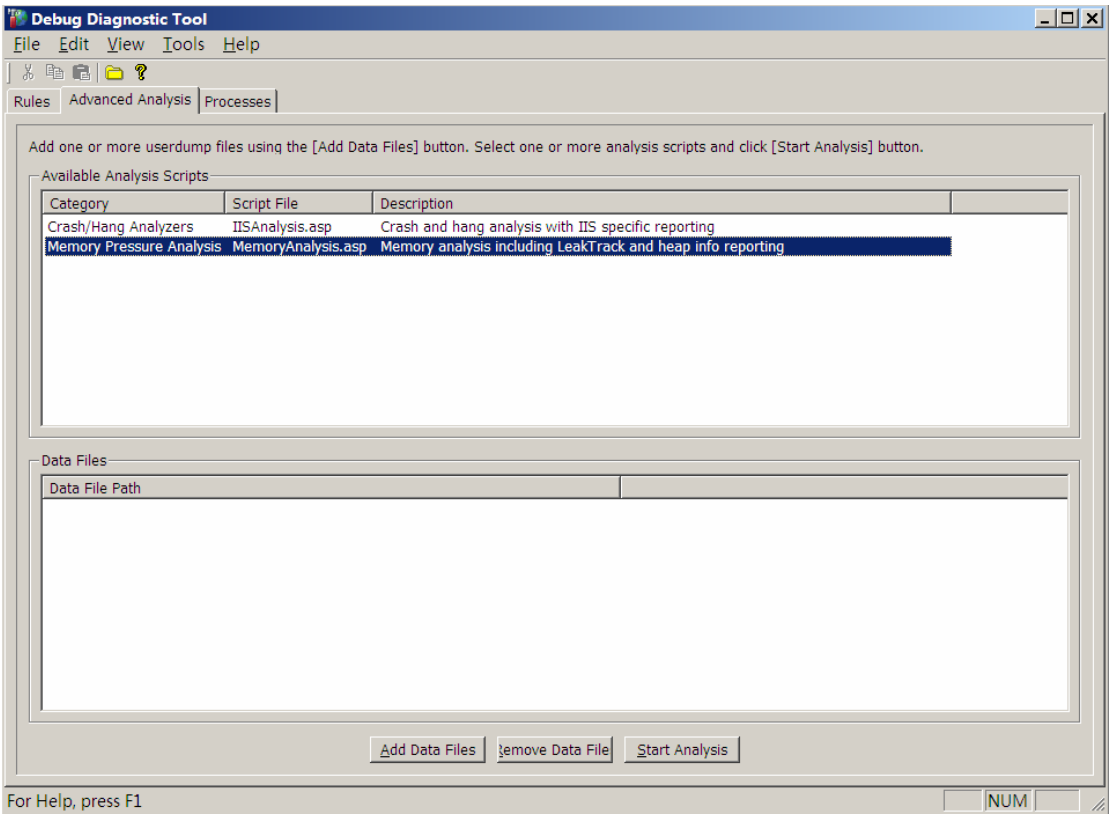
2. 点击 Tools -> Options and settings 设定 symbol 路径,dump 生成路径等等
3. 进行手动内存泄漏监视的时候,需要激活 Options and settings -> Preferences -> User service mode to overcome terminal server limitations
4. 完成上面的设置后,启动 IE 进程,然后切换到主窗口的 Process tab
5. 找到 IE 进程,右点后选择 monitor for leaks



6. 切换到 IE,狂点按钮重现问题
7. 等到内存上涨到 160MB 左右,切换回 IIS Diag，找到被监视的 IE 进程,右点后选择 Create Full Userdump. Dump 文件会生成到前面设定的路径中

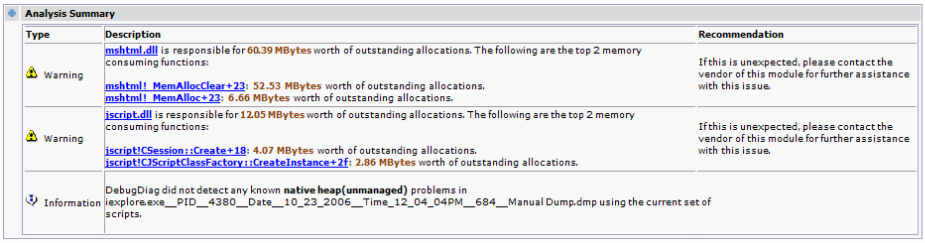
到此为止,信息收集工作就搞定了,后面开始自动 dump 分析．无论是否用 IIS Diag 抓取的 dump，无论有没有用 IIS Diag 监视内存泄漏，都可以用 IIS Diag 进行自动分析．如果目标进程激活了 IIS Diag 的内存泄漏监视，IIS Diag 的 dump 分析结果中会包含更多的信息．下面是进行自动分析的详细步骤:

1. 切换到 Advanced Analysis tab，选择 memory pressure analysis



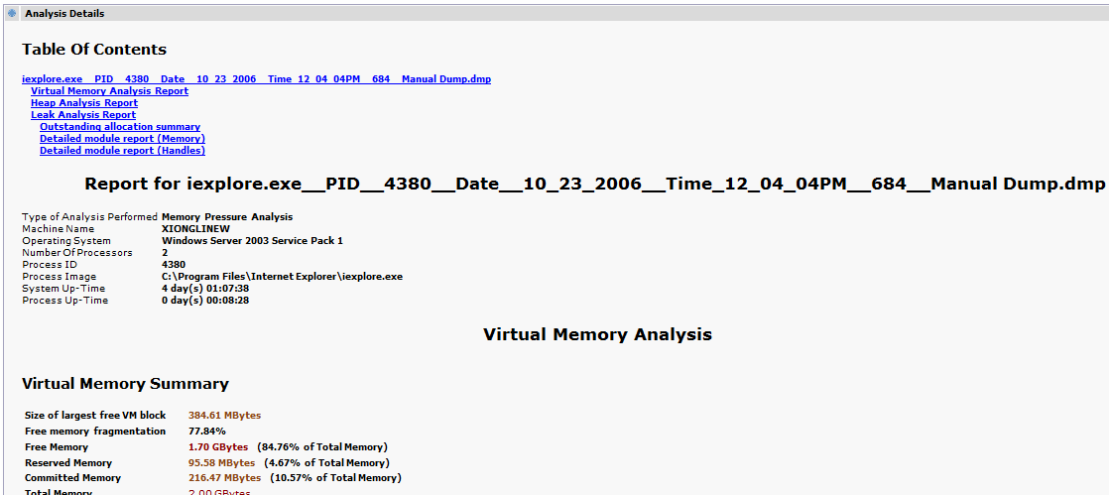
2. 点 Add data files,选定刚刚生成的 dump 文件,然后点 start analysis 开始分析
3. 过一段时间后,分析完成,结果文件会自动在浏览器中打开

生成的结果文件图文并茂,生动有趣. 请坐下来，泡一杯茶，然后体会一下看报表的乐趣吧.



Analysis Summary

Type	Description	Recommendation
Warning	mshtml.dll is responsible for 60.39 MBytes worth of outstanding allocations. The following are the top 2 memory consuming functions: mshtml!_MemAllocClear+23: 52.53 MBytes worth of outstanding allocations. mshtml!_MemAlloc+23: 6.66 MBytes worth of outstanding allocations.	If this is unexpected, please contact the vendor of this module for further assistance with this issue.
Warning	jscript.dll is responsible for 12.05 MBytes worth of outstanding allocations. The following are the top 2 memory consuming functions: jscript!CSession::Create+18: 4.07 MBytes worth of outstanding allocations. jscript!CJScriptClassFactory::CreateInstance+2f: 2.86 MBytes worth of outstanding allocations.	If this is unexpected, please contact the vendor of this module for further assistance with this issue.
Information	DebugDiag did not detect any known **native heap(unmanaged)** problems in iexplore.exe__PID__4380__Date__10_23_2006__Time_12_04_04PM__684__Manual Dump.dmp using the current set of scripts.	



Analysis Details

Table Of Contents

iexplore.exe__PID__4380__Date__10_23_2006__Time_12_04_04PM__684__Manual Dump.dmp
- Virtual Memory Analysis Report
- Heap Analysis Report
- Leak Analysis Report
  - Outstanding allocation summary
  - Detailed module report (Memory)
  - Detailed module report (Handles)

Report for iexplore.exe__PID__4380__Date__10_23_2006__Time_12_04_04PM__684__Manual Dump.dmp

Type of Analysis Performed	**Memory Pressure Analysis**
Machine Name	**XIONGLINEW**
Operating System	**Windows Server 2003 Service Pack 1**
Number Of Processors	**2**
Process ID	**4380**
Process Image	**C:\Program Files\Internet Explorer\iexplore.exe**
System Up-Time	**4 day(s) 01:07:38**
Process Up-Time	**0 day(s) 00:08:28**

Virtual Memory Analysis

Virtual Memory Summary

Size of largest free VM block	384.61 MBytes
Free memory fragmentation	**77.84%**
Free Memory	1.70 GBytes **(84.76% of Total Memory)**
Reserved Memory	95.58 MBytes **(4.67% of Total Memory)**
Committed Memory	216.47 MBytes **(10.57% of Total Memory)**
Total Memory	2.00 GBytes

总的来说结果文件包含下面一些信息:

最上方的 **Analysis Summary** 包含了内存泄漏和内存碎片方面的统计信息.如果分析到有可能有内存泄漏, 这里会列出可疑的地方. 比如 IIS Diag 分析到:

mshtml.dll is responsible for **60.39 MBytes** worth of outstanding allocations. The following are the top 2 memory consuming functions:

mshtml!_MemAllocClear+23: **52.53 MBytes** worth of outstanding allocations.
mshtml!_MemAlloc+23: **6.66 MBytes** worth of outstanding allocations.

这里说明 mshtml 模块导致了 60.39MB 内存增量,其中 52.53MB 是通过 mshtml!_MemAllocClear 直接调用 API 分配的

如果 IIS Diag 分析到内存碎片严重，也会给出详细信息.
在 **Analysis Details** 中包含了下面一些详细信息

Table Of Contents

iexplore.exe PID 4380 Date 10_23_2006 Time 12_04_04PM 684 Manual Dump.dmp
- **Virtual Memory Analysis Report**
- **Heap Analysis Report**
- **Leak Analysis Report**
  - **Outstanding allocation summary**
  - **Detailed module report (Memory)**
  - **Detailed module report (Handles)**

Virtual Memory Analysis Report 中包含了虚拟内存的统计信息. 这里可以看到已分配的地址空间的大小，剩余空间大小，私有内存大小，还会估算出剩余内存空间的分片比率. 下面是节选:

Virtual Memory Summary

Size of largest free VM block	**384.61 MBytes**
Free memory fragmentation	**77.84%**
Free Memory	**1.70 GBytes (84.76% of Total Memory)**
Reserved Memory	**95.58 MBytes (4.67% of Total Memory)**
Committed Memory	**216.47 MBytes (10.57% of Total Memory)**
Total Memory	2.00 GBytes
Largest free block at	0x00000000`1a524000

Heap Analysis Report 更为详细了．会列出进程中所有的 Heap handle， 然后依次列举出每个 Heap 的统计信息，比如：

Heap 1 - 0x00140000

Heap Name	**Default process heap**
Heap Description	**This heap is created by default and shared by all modules in the process**
Reserved memory	**128.06 MBytes**
Committed memory	**85.32 MBytes (66.63% of reserved)**
Uncommitted memory	**42.74 MBytes (33.37% of reserved)**
Number of heap segments	**8 segments**
Number of uncommitted ranges	27 range(s)
Size of largest uncommitted range	41.46 MBytes
Calculated heap fragmentation	3.00%

接下来就是最激动人心的 Leak Analysis Report 了

首先给出内存分配的整体统计：

Outstanding allocation summary

Number of allocations	**319,562 allocations**
Total outstanding handle count	**17 handles**
Total size of allocations	**74.56 MBytes**
Tracking duration	**0 day(s) 00:07:09**

接下来根据每个 module 每次分配的大小排一次序，再根据分配的次数排一次序．由于 IIS Diag 不仅仅监视 Virtual 系列的内存分配，还监视 Heap, CRT 和 COM 相关的内存分配，所以 IIS Diag 还能自动根据内存的用途分类，给出每一类的内存分别占用多少:

Heap memory manager 62.49 MBytes

C/C++ runtime memory manager 12.06 MBytes

OLE/COM memory manager 6.08 KBytes

OLE automation BSTR memory manager 930 Bytes

Virtual memory manager 0 Bytes

然后再依次列出所记录到的每一个模块潜在的泄漏情况，根据泄漏的多少排序．第一个就是 mshtml:

Module details for mshtml

Module Name	**mshtml**
Allocation Count	**235014 allocation(s)**
Allocation Size	**60.39 MBytes**

再给出最值得怀疑的内存分配 callstack:

Function details

Function	**mshtml!_MemAllocClear+23**
Allocation type	Heap allocation(s)
Heap handle	0x00140000
Allocation Count	**156044 allocation(s)**
Allocation Size	**52.53 MBytes**
Leak Probability	**87%**

最后再列出导致这个函数调用的典型 callstack:

Call stack sample 1

Address	0x05f6d420
Allocation Time	0 day(s) 00:05:00 since tracking started
Allocation Size	16 Bytes

Function	Source	Destination
mshtml!_MemAllocClear+23		ntdll!RtlAllocateHeap
mshtml!CCollectionCache::GetDisp+1d9		mshtml!_MemAllocClear
mshtml!CDocument::getElementsByTagName+7d		mshtml!CCollectionCache::GetDisp
ieframe!CWebPageMetaInfo::_InitMeta+40		
ieframe!CWebPageMetaInfo::Init+95		ieframe!CWebPageMetaInfo::_InitMeta
ieframe!CDocObjectHost::_HandleMetaTags+4f		ieframe!CWebPageMetaInfo::Init
ieframe!CDocObjectHost::_HandleShdocvwCmds+128		ieframe!CDocObjectHost::_HandleMetaTags
ieframe!CDocObjectHost::Exec+13e		ieframe!CDocObjectHost::_HandleShdocvwCmds
mshtml!CTExec+40		
mshtml!CMarkup::OnLoadStatusParseDone+8a		mshtml!CTExec
mshtml!CMarkup::OnLoadStatus+8b		mshtml!CMarkup::OnLoadStatusParseDone
mshtml!CProgSink::DoUpdate+533		mshtml!CMarkup::OnLoadStatus

mshtml!CProgSink::OnMethodCall+f	mshtml!CProgSink::DoUpdate
mshtml!GlobalWndOnMethodCall+101	
mshtml!GlobalWndProc+181	mshtml!GlobalWndOnMethodCall
user32!DispatchHookW+33	
ieframe!ATL::CComPtr<IStream>::CopyTo<IStrea m>+23	
kernel32!BaseThreadStart+34	

Call stack sample 4

Address	0x08e640c0
Allocation Time	0 day(s) 00:05:04 since tracking started
Allocation Size	8.00 KBytes

Function	Source	Destination
mshtml!_MemAllocClear+23		ntdll!RtlAllocateHeap
mshtml!CTxtBlk::InitBlock+21		mshtml!_MemAllocClear
mshtml!CTxtArray::AddBlock+2a		mshtml!CTxtBlk::InitBlock
mshtml!CTxtPtr::InsertRange+48		mshtml!CTxtArray::AddBlock
mshtml!CTxtPtr::InsertRepeatingChar+5f		mshtml!CTxtPtr::InsertRange
mshtml!CMarkup::CreateInitialMarkup+93		mshtml!CTxtPtr::InsertRepeatingChar
mshtml!CMarkup::DestroySplayTree+370		mshtml!CMarkup::CreateInitialMarkup
mshtml!CMarkup::UnloadContents+2ab		mshtml!CMarkup::DestroySplayTree
mshtml!CMarkup::TearDownMarkupHelper+90		mshtml!CMarkup::UnloadContents
mshtml!CMarkup::TearDownMarkup+44		mshtml!CMarkup::TearDownMarkupHelper
mshtml!CFrameSite::TearDownFrameContent+4f		mshtml!CMarkup::TearDownMarkup
mshtml!CFrameSite::Passivate+27		mshtml!CFrameSite::TearDownFrameCont ent
mshtml!CBase::PrivateRelease+2d		
mshtml!CWindow::PrivateRelease+1b		mshtml!CBase::PrivateRelease
jscript!IDispatchExInvokeEx2+ac		
jscript!IDispatchExInvokeEx+56		jscript!IDispatchExInvokeEx2
jscript!InvokeDispatchEx+78		jscript!IDispatchExInvokeEx
jscript!VAR::InvokeByName+ba		jscript!InvokeDispatchEx
jscript!VAR::InvokeDispName+43		jscript!VAR::InvokeByName
jscript!VAR::InvokeByDispID+b9		jscript!VAR::InvokeDispName
jscript!CScriptRuntime::Run+16c9		jscript!VAR::InvokeByDispID
jscript!ScrFncObj::Call+8d		jscript!CScriptRuntime::Run
jscript!CSession::Execute+a1		
jscript!NameTbl::InvokeDef+179		jscript!CSession::Execute
jscript!NameTbl::InvokeEx+cb		jscript!NameTbl::InvokeDef
jscript!IDispatchExInvokeEx2+ac		

jscript!IDispatchExInvokeEx+56	jscript!IDispatchExInvokeEx2
jscript!NameTbl::InvokeEx+2c5	jscript!IDispatchExInvokeEx
mshtml!CScriptCollection::InvokeEx+8c	
mshtml!CWindow::InvokeEx+6dc	
shlwapi!_StrCmpLocaleW+24	
shlwapi!StrCmpW+16	shlwapi!_StrCmpLocaleW
shlwapi!SHPathCreateFromUrl+139	
mshtml!CMarkup::GetSecurityID+188	
mshtml!CMimeTypes::Release+e	
mshtml!CBase::FireEvent+105	
mshtml!CElement::BubbleEventHelper+27c	mshtml!CBase::FireEvent
mshtml!CElement::FireEvent+286	mshtml!CElement::BubbleEventHelper
mshtml!CElement::Fire_onclick+1c	mshtml!CElement::FireEvent
mshtml!CElement::DoClick+80	mshtml!CElement::Fire_onclick
mshtml!CInput::DoClick+3c	mshtml!CElement::DoClick
mshtml!CDoc::PumpMessage+cb6	
mshtml!CDoc::OnMouseMessage+3d7	mshtml!CDoc::PumpMessage
mshtml!CDoc::OnWindowMessage+8e9	mshtml!CDoc::OnMouseMessage
user32!DispatchHookA+108	
user32!CallHookWithSEH+21	
user32!__fnHkINLPMOUSEHOOKSTRUCTEX+25	user32!CallHookWithSEH
user32!GetWindowLongW+49	
user32!InternalCallWinProc+28	
user32!UserCallWinProcCheckWow+151	user32!InternalCallWinProc
user32!DispatchMessageWorker+327	user32!UserCallWinProcCheckWow
user32!DispatchMessageW+f	user32!DispatchMessageWorker
ieframe!CTabWindow::_TabWindowThreadProc+189	user32!DispatchMessageW
kernel32!BaseThreadStart+34	

上面的 callstack 跟用 heap trace 抓到的结果一模一样．但是排错体验要爽得多.

结论

找到了导致内存泄漏的 callstack。跟 IE 开发人员联系后，得到的结果是 IE 的内存管理就是这样设计的。这部分内存会在页面跳转或者刷新的时候释放。

换句话说，IE 没有实现如此细粒度的内存管理。要解决这个问题，可以修改客户端脚本定时刷页面，或者避免 IFrame+Script 这样的方式。

分析 IIS Diag

接下来可以用 windbg 分析一下 IIS Diag 如何抓到这些详细信息的．打开 dump 后看到:

```
0:000> u kernel32!VirtualAllocEx
kernel32!VirtualAllocEx:
77e6be89 e96a10348c      jmp     LeakTrack!CVirtualMemoryLT::VirtualAllocExDetour (041acef8)
0:000> u ntdll!RtlAllocateHeap
ntdll!RtlAllocateHeap:
7c82f9fd e9d3ad9787      jmp     LeakTrack!CHeapMemoryLT::HeapAllocDetour (041aa7d5)
0:000> u oleaut32!SysAllocString
oleaut32!SysAllocString:
77d04646 e99bfa498c      jmp     LeakTrack!CBSTRLT::SysAllocStringDetour (041a40e6)
```

上面三个函数分别负责页内存分配，Heap 分配和 COM BSTR 内存分配．从汇编代码看到，这三个函数入口被一个 jmp 命令强行跳转到 LeakTrace 中的对应函数．LeakTrace 就是 IIS Diag 进行 API Hook 的 module．从 LeakTrace 中对应的函数名字可以大致猜到，LeakTrace 使用了微软研究院的 Detour API Hook 库进行的 API Hook:

http://research.microsoft.com/sn/detours/

前面提到了，如果 dump 不是用 DebugDiag 工具抓到的，也可以让 DebugDiag 帮忙分析。下面就是一个用 adplus 抓的 dump,分析结果是:

DebugDiag did not detect LeakTrack.dll loaded in **gsmemoryexception.dmp**, so no leak analysis was performed on this file. If you are troubleshooting a memory leak, please ensure LeakTrack.dll is injected into the target process using the DebugDiag tool before or generating new dumps.

For information regarding installation and usage of the IISDiag tool, please see the included help file.

Detected symptoms of high fragmentation in the following heaps in gsmemoryexception.dmp

0x00d40000 (**GLMS_Server+725340** - 99.97% Fragmented)

Heap 16 - 0x00d40000

Heap Name	**GLMS_Server+725340**
Heap Description	**This heap is used by GLMS_Server**
Reserved memory	**1.89 GBytes**
Committed memory	**251.25 MBytes (12.96% of reserved)**
Uncommitted memory	**1.65 GBytes (87.04% of reserved)**
Number of heap segments	**64 segments**
Number of uncommitted ranges	4379 range(s)
Size of largest uncommitted range	500.00 KBytes

Calculated heap fragmentation 99.97%

直接分析出了 heap fragmentation 的问题!

托管内存泄露

CLR 程序的内存释放依赖 GC。GC 何时发生依赖于 CLR Runtime 的判断，而不需要程序干预。但这并不等于说 CLR 程序就不会有内存泄露。GC 只会收 un-rooted 的 object，而 object 是否有 root，是受程序员控制的。比如程序中有一个静态的 Array 变量。运行过程中只往该 Array 中添加 object 而不删除，那 Array 中所有的 object 都不会被 GC 回收，内存使用就会持续增长

跟调试 unmanaged leak 相比，检查托管程序的内存泄露容易得多。

对于 unmanaged 程序，需要激活 heap trace，才能够在问题发生后找到内存分配的相关信息。对于托管程序，由于 CLR Runtime 维护了 managed heap 上的类型信息，所以往往直接从 dump 中就可以分析出高内存的原因

[案例分析，object chain]

问题背景：

客户的 asp.net 程序运行一段时间后会抛出 OutOfMemory 异常。从 task manager 上看，问题发生前 VM Size 到大约 1.2GB

Asp.net 可以配置内存限制。当内存到达某个级别的时候，程序会自己动 recycle。但这不是解决问题的根本办法。

正常情况下用户态进程内存使用到 500MB 以上就算比较高了。所以当内存使用到 700MB 以上，就可以建议客户抓取 dump 文件，并不需要等到 OutOfMemory 异常最终发生。拿到 dump 后，分析如下：

```
0:015> .load clr10\sos
0:015> !eeheap -gc
Loaded Son of Strike data table version 5 from
"C:\WINDOWS\Microsoft.NET\Framework\v1.1.4322\mscorsvr.dll"
Number of GC Heaps: 4
------------------------------
Heap 0 (0x000db6c8)
generation 0 starts at 0x2a576d94
generation 1 starts at 0x296dc3a0
generation 2 starts at 0x102d0030
```

```
ephemeral segment allocation context: (0x2c2c3984, 0x2c2c3990)
 segment        begin             allocated         size
0x102d0000 0x102d0030    0x1426b32c 0x3f9b2fc(66,695,932)
0x282d0000 0x282d0030    0x2c2c3984 0x3ff3954(67,058,004)
Large object heap starts at 0x202d0030
 segment        begin             allocated         size
0x202d0000 0x202d0030    0x205de2b8 0x0030e288(3,203,720)
Heap Size    0x829ced8(136,957,656)
------------------------------
Heap 1 (0x000dbfe8)
generation 0 starts at 0x2cecdca8
generation 1 starts at 0x2c2d0030
generation 2 starts at 0x142d0030
ephemeral segment allocation context: (0x2ee5a000, 0x2ee5a00c)
 segment        begin             allocated         size
0x142d0000 0x142d0030    0x17d9cd38 0x3accd08(61,656,328)
0x2c2d0000 0x2c2d0030    0x2ee5a000 0x2b89fd0(45,653,968)
Large object heap starts at 0x212d0030
 segment        begin             allocated         size
0x212d0000 0x212d0030    0x21531c18 0x00261be8(2,497,512)
Heap Size    0x68b88c0(109,807,808)
------------------------------
Heap 2 (0x000dc8b0)
generation 0 starts at 0x25e2ae60
generation 1 starts at 0x25118104
generation 2 starts at 0x182d0030
ephemeral segment allocation context: (0x27d73000, 0x27d7300c)
 segment        begin             allocated         size
0x182d0000 0x182d0030    0x1c10c920 0x3e3c8f0(65,259,760)
0x242d0000 0x242d0030    0x27d73000 0x3aa2fd0(61,485,008)
Large object heap starts at 0x222d0030
 segment        begin             allocated         size
0x222d0000 0x222d0030    0x2249d7b0 0x001cd780(1,890,176)
Heap Size    0x7aad040(128,634,944)
------------------------------
Heap 3 (0x000dd178)
generation 0 starts at 0x0d0644f4
generation 1 starts at 0x0c239e88
generation 2 starts at 0x1c2d0030
ephemeral segment allocation context: (0x0ed9b000, 0x0ed9b00c)
 segment        begin             allocated         size
0x1c2d0000 0x1c2d0030    0x20214da0 0x3f44d70(66,342,256)
0xb010000 0xb010030    0xed9b000 0x3d8afd0(64,532,432)
Large object heap starts at 0x232d0030
```

segment begin allocated size
0x232d0000 0x232d0030 0x23641f60 0x00371f30(3,612,464)
Heap Size 0x8041c70(134,487,152)

Reserved segments:

GC Heap Size 0x1e644448(509,887,560)

上面的结果看到受到GC管理的managed heap占用了510MB左右的内存，占整体内存的80%了，所以可以判定高内存是managed heap导致的。

(这里除了受到GC管理的managed heap外，还有其它类型的managed heap，比如加载assembly,生成class信息需要的loader heap。可以使用!eeheap –loader检查)

接下来通过!dumpheap –stat命令可以显示出managed heap中object的统计信息：

0:015> !dumpheap -stat

Heap 0
total 1,178,596 objects

Heap 1
total 1,355,363 objects

Heap 2
total 1,317,576 objects

Heap 3
total 1,406,665 objects
---
total 5,258,200 objects
Statistics:
MT Count TotalSize Class Name
0x79bff2e4 1 12
System.Runtime.Remoting.Activation.ActivationListener
0x79bff11c 1 12 System.Runtime.Remoting.Activation.LocalActivator
…
0x04ec8b98 83,002 10,624,256 System.Data.DataColumn
0x01b6292c 124,328 11,624,628 System.Byte[]
0x04eccf8c 185,213 11,853,632 System.Data.DataColumnPropertyDescriptor
0x79baaf78 1,589,467 19,073,604 System.Int32
0x79b94638 141,215 26,056,372 System.String
0x79ba3adc 1,584,116 38,018,784
System.Collections.ArrayList/ArrayListEnumeratorSimple

0x01b66d9c 6,275 60,342,820 System.DateTime[]
0x01b631e8 64,901 84,694,248 System.Int32[]
0x01b6209c 210,924 205,428,488 System.Object[]
Total 5,258,200 objects, Total size: 509,887,548

上面的输出看到占用内存最多的是 System.Object[]数组类型，一共占用了 205MB。其它占用比较高的类型有 string, ArrayList, DateTime 数组，Int32 数组。另外值得注意的是，DataColumn 类型占用也有 10MB 左右，一共有 8 万多个 DataColumn

注意，这里并不是指 System.Object[]数组中包含的所有 object 占用了 205MB，而是指 System.Object[]数组本身占用了 205MB。对于一个 System.Object[]数组，本身占用的内存是用来维护数组元素的空间。每一个 object 元素需要一个 DWORD 来保存该 object 的引用。对于一个长度固定的 System.Object[]数组来说，无论其中的元素是何种 objct 类型，或者整个数组中所有的元素都是 null,数组本身占用的内存都是固定的。

所以，上面的统计信息给出了下面一些疑点：

1. System.Object[]数组一共有 21 万个，程序为何需要这么多数组。
2. System.Object[]数组本身占用的内存就达到了 managed 内存的一半，说明其中包含的元素大多是 null。否则整体内存不会只有 509MB
3. 对内存占用贡献最大的三甲都是数组类型。
4. System.Data.Column 数量也很多。由于 System.Data.Column 和 DataTable 内部都通过数组维护数据，所以大量的数组跟 DataColumn 很可能有关系。

接下来继续用 dumpheap 检查 System.Object[]类型。通过-min 参数，打印出大于 20K 的 object, 以便继续观察这些 System.Object[]的内容

```
0:015> !dumpheap -mt 0x01b6209c       -min 20000
------------------------------
Heap 0
    Address               MT       Size    Gen
0x1066d608 0x01b6209c     32,784       2 System.Object[]
0x107d6f18 0x01b6209c     32,784       2 System.Object[]
0x107df334 0x01b6209c     32,784       2 System.Object[]
。。。
0x288c3bc8 0x01b6209c     32,784       2 System.Object[]
0x288d131c 0x01b6209c     32,784       2 System.Object[]
total 837 objects
------------------------------
Heap 1
    Address               MT       Size    Gen
total 0 objects
------------------------------
Heap 2
```

```
     Address               MT        Size    Gen
total 0 objects
------------------------------
Heap 3
     Address               MT        Size    Gen
total 0 objects
------------------------------
total 837 objects
```

从结果可以看到，大于 20K 的 object[]数组大小都一致，绝大多数都是在 generation2。这说明这些 object[]已经生存了很长时间。随便找一个 object[]分析：

```
0:015> !do -v 0x2886aaf8
Name: System.Object[]
MethodTable 0x01b6209c
EEClass 0x01b62018
Size 32784(0x8010) bytes
GC Generation: 2
Array: Rank 1, Type CLASS
Element Type: System.Object
Content: 8,192 items
------ Will only dump out valid managed objects ----
     Address               MT  Class Name
0x14392710    0x79b94638    System.String
0x14392710    0x79b94638    System.String
0x14392710    0x79b94638    System.String
0x14392710    0x79b94638    System.String
0x14392710    0x79b94638    System.String
..
```

前面提到，object[]中应该有大量的 null，但实际情况有点诧异。数组中包含的数据并不是 null,而是相同的 string object，string 的地址在 0x14392710。

```
0:015> !do 0x14392710
Name: System.String
MethodTable 0x79b94638
EEClass 0x79b94984
Size 40(0x28) bytes
GC Generation: 2
mdToken: 0x0200000f  (c:\windows\microsoft.net\framework\v1.1.4322\mscorlib.dll)
String: DBNullONsN
FieldDesc*: 0x79b949e8
           MT        Field        Offset                 Type        Attr        Value Name
```

0x79b94638 0x4000013 0x4 System.Int32 instance 11 m_arrayLength
0x79b94638 0x4000014 0x8 System.Int32 instance 10 m_stringLength
0x79b94638 0x4000015 0xc System.Char instance 0x44 m_firstChar
0x79b94638 0x4000016 0 CLASS shared static Empty
>> Domain:Value 0x000d1050:0x1c2d0224 0x000ff588:0x1c2d0224 0x037427b0:0x1c2d0224 0x0465db58:0x1c2d0224 <<
0x79b94638 0x4000017 0x4 CLASS shared static WhitespaceChars
>> Domain:Value 0x000d1050:0x1c2d0238 0x000ff588:0x142d687c 0x037427b0:0x1c3da18c 0x0465db58:0x1860369c <<

察看 string 的详细信息，发现内容是 DBNullONsN。其中的 DBNull 让人联想到问题跟 Database 类型相关。

有了大致的信息后，下一步就是检查这些类型为何没有被 GC 收集：

0:015> !gcroot 0x2886aaf8
Scan Thread 10 (0x1b4)
Scan Thread 16 (0x934)
Scan Thread 18 (0x13ec)
Scan Thread 4 (0x1510)
Scan Thread 19 (0x1470)
Scan Thread 5 (0xda8)
Scan Thread 2 (0x8cc)
Scan Thread 3 (0xd94)
Scan Thread 24 (0x119c)
Scan Thread 27 (0x143c)
ESP:a8e8cc:Root:0x187ec434(MMNN.NewCase.BF.CustoNameSpace.CustomerController)->0x187ed174(System.Collections.Hashtable)->0x1f743220(System.Collections.Hashtable/bucket[])->0x2465a32c(MMNN.NewCase.BF.CustoNameSpace.ReplicationData)->0x28708e7c(System.Data.DataSet)->0x28708ef8(System.Data.DataTableCollection)->0x28708f18(System.Collections.ArrayList)->0x28708f30(System.Object[])->0x28717314(MMNN.NewCase.DS.FacadeDataSet.TransmissionFDS/CaseActionDataTable)->0x2871ae00(System.ComponentModel.PropertyDescriptorCollection)->0x2871a350(System.Object[])->0x2871ab40(System.Data.DataColumnPropertyDescriptor)->0x287198c0(System.Data.DataColumn)->0x2875ce88(System.Data.Common.StringStorage)->0x2886aaf8(System.Object[])
Scan HandleTable 0xd5e90
Scan HandleTable 0xd84f8
Scan HandleTable 0x13f250
Scan HandleTable 0x5d34bb0
Scan HandleTable 0x63889c0

从上面的结果可以看到，对 object[]的引用关系是从一个客户定义的类型 **CustomerController** 开始。该类型包含一个 HashTable 集合，其中包含了客户的另外一种类型 **ReplicationData** ，其中包含了 DataSet 的引用。DataSet 包含 DataTable，DataTable 的 DataColumn 最终引用了 object[]数组。由此可以了解到，object[]数组其实是包含 DataTable 数据的最终存储空间。整个引用链是从客户的自定义类型开始。由于该类型还存活在线程 27 的 stack 上，所以 GC 不会回收这些内存。得到最终的 root 后，用 objsize 计算一下该 object 及其该 object 所 root 的所有 object 的内存占用数量：

0:015> !objsize 0x187ec434
sizeof(0x187ec434) = 379,545,960 (0x169f6968) bytes (MMNN.NewCase.BF.CustoNameSpace.CustomerController)

该 object 导致了 380MB 左右的占用，占整个 GC 内存的 75%左右，所以这里就是问题的根源。

解决这个问题，接下来需要跟开发人员联系以便弄清楚：

1. CustomerController 是否为何存活在堆栈上，是否可以释放
2. 整个引用链是否符合开发人员最初的设计，为何这样设计
3. 引用链中保存在 HashTable 或者 ArrayList 等容器中的 object 是否可以通过代码在适当的时候删除
4. DataTable 为何有那么大，而且包含的都是 DNBull 数据

对于第一点，通过检查线程 27 的 callstack，可以提供下面的信息给开发人员协助检查:

```
0:027> kL
ChildEBP RetAddr
00a8e43c 7c822124 ntdll!KiFastSystemCallRet
00a8e440 77e6baa8 ntdll!NtWaitForSingleObject+0xc
00a8e4b0 77e6ba12 kernel32!WaitForSingleObjectEx+0xac
00a8e4c4 791fee7b kernel32!WaitForSingleObject+0x12
00a8e4e4 7920273d mscorsvr!GCHeap::GarbageCollectGeneration+0x1a9
00a8e514 791c0ccd mscorsvr!gc_heap::allocate_more_space+0x181
00a8e73c 791b6269 mscorsvr!GCHeap::Alloc+0x7b
00a8e750 791c0e9a mscorsvr!Alloc+0x3a
00a8e7a0 791c0efe mscorsvr!AllocateArrayEx+0x161
00a8e824 04f36ccb mscorsvr!JIT_NewArr1+0xbb
WARNING: Frame IP not in any known module. Following frames may be wrong.
00a8e828 00000000 0x4f36ccb
0:027> !clrstack
Thread 27
ESP              EIP
0x00a8e7f4    0x7c82ed54 [FRAME: HelperMethodFrame]
0x00a8e82c    0x04f36ccb [DEFAULT] [hasThis] Void System.Data.Common.DateTimeStorage.SetCapacity(I4)
```

```
0x00a8e83c    0x04f366d5 [DEFAULT] [hasThis] Void System.Data.DataColumn.SetCapacity(I4)
0x00a8e848    0x04f3664d [DEFAULT] [hasThis] Void System.Data.RecordManager.set_RecordCapacity(I4)
0x00a8e860    0x04f3652d [DEFAULT] [hasThis] Void System.Data.RecordManager.GrowRecordCapacity()
0x00a8e870    0x04f364a7 [DEFAULT] [hasThis] I4 System.Data.RecordManager.NewRecordBase()
0x00a8e880    0x072a0938 [DEFAULT] [hasThis] I4 System.Data.RecordManager.CopyRecord(Class System.Data.DataTable,I4,I4)
0x00a8e8ac    0x072a07df [DEFAULT] [hasThis] Void System.Data.DataTable.CopyRow(Class System.Data.DataTable,Class System.Data.DataRow)
0x00a8e8c8    0x072a9539 [DEFAULT] [hasThis] Class System.Data.DataSet System.Data.DataSet.Copy()
0x00a8e904    0x0748b93b [DEFAULT] [hasThis] Void MMNN.NewCase.BF.CustoNameSpace.CustomerController.AddDataSet(Class
System.Data.DataSet,ValueClass System.Guid)
   at [+0xbb] [+0x0]
0x00a8ebe0    0x072f70bb [DEFAULT] Class System.Data.DataTable MMNN.NewCase.BF.CustoNameSpace.IntegrationBF.SendReplicationData(Class
System.Data.DataSet,String)
   at [+0x7ab] [+0x2bb]
0x00a8ef7c    0x072f6784 [DEFAULT] [hasThis] Void MMNN.NewCase.Framework.Aspects.ReplicationAspect.replicateData(Class
System.Data.DataSet,String)
0x00a8f088    0x791b3208 [FRAME: GCFrame]
0x00a8f180    0x791b3208 [FRAME: ECallMethodFrame] [DEFAULT] [hasThis] Object
System.Runtime.Remoting.Messaging.StackBuilderSink.PrivateProcessMessage(Class System.Reflection.MethodBase,SZArray
Object,Object,I4,Boolean,ByRef SZArray Object)
0x00a8f1a4    0x79af513e [DEFAULT] [hasThis] Class System.Runtime.Remoting.Messaging.IMessageCtrl
System.Runtime.Remoting.Messaging.StackBuilderSink.AsyncProcessMessage(Class System.Runtime.Remoting.Messaging.IMessage,Class
System.Runtime.Remoting.Messaging.IMessageSink)
0x00a8f20c    0x79aea95f [DEFAULT] Void System.Runtime.Remoting.Proxies.AgileAsyncWorkerItem.ThreadPoolCallBack(Object)
0x00a8f440    0x791b3208 [FRAME: ContextTransitionFrame]
```

经过客户开发人员的确认，代码中忘记把处理后的数据及时从 CustomerController 类型中删除。修改代码后，问题解决。

从这个案例可以看到对 managed leak 的调试非常简单。直接从 dump 中就可以找出占用内存的 object 类型以及整个引用链。

对于大多数的托管程序来说，通过上面的步骤就可以定位到问题。但是也有一些例外：

[案例分析，一个 bt 的案例]

客户的.NET Service 使用 Remoting 技术处理客户端请求。服务器和客户端都在局域网内。程序总体压力比较大，白天晚上都有业务要处理。奇怪的地方是，白天业务繁忙的时候也没见出什么问题，反而是晚上业务量减少的时候，客户端会不时地报告 OutOfMemory 异常。

拿到客户端的截图和性能日志，分析后发现 OutOfMemory 其实不是客户端有问题。而是 Remote Server 在处理客户请求的时候发生了 OutOfMemory，通过序列化把 OutOfMemory 异常传递到了客户端，在客户端观察到了这个异常。问题还是发生在服务器上。

在服务器上抓取白天和晚上的性能日志分析后发现，的确晚上内存消耗比白天要多。但是客户分析 SQL 日志后发现晚上处理的业务明显只有白天的一半不到。是什么原因呢？

抓取 dump 分析是最直接的途径了。由于问题在晚上发生，所以通过前面介绍过的方法配置性能监视器，当目标进程 private memory 到 800MB 的时候，抓取了 hang dump。拿过来一看，发现：

0x00f24820 204 82,872 System.Collections.Hashtable/bucket[]
0x00a43068 929 220,368 System.Object[]
0x03af6d6c 13,040 417,280 System.Threading.Overlapped
0x00a44300 9,480 642,028 System.String
0x03af5f1c 31,707 887,796 System.AsyncCallback
0x04778a94 18,667 1,045,352 System.Net.Sockets.AcceptAsyncResult
0x0477bb30 13,040 1,564,800
System.Net.Sockets.OverlappedAsyncResult
0x00ea5100 13,112 6,984,088 System.Byte[]
0x00171d78 12,926 **765,727,520** Free
Total 120,951 objects, Total size: 777,867,152

Fragmented blocks larger than 0.5MB:
Addr Size Followed by
0x020592dc 0.7MB large free object followed by 0x0210d8f0 System.Byte[] 0x087ff03c 0.6MB large free object followed by 0x0889cdcc System.Byte[] 0x0ae29e40 0.5MB large free object followed by 0x0aeb104c System.Byte[] 0x0de1cf50 0.6MB large free object followed by 0x0deb792c System.Byte[]
0x0ee32f68 0.7MB large free object followed by 0x0eeec850 System.Byte[]
0x12f27104 0.6MB large free object followed by 0x12fbb3cc System.Byte[] 0x16f2538c 0.7MB large free object followed by 0x16fd2dcc System.Byte[] 0x17f48a8c 0.6MB large free object followed by 0x17fed534 System.Byte[]
0x19f14c84 0.6MB large free object followed by 0x19fa42d8 System.Byte[] 0x1af4f3b0 0.7MB large free object followed by 0x1b004958 System.Byte[]
0x1cf28498 0.6MB large free object followed by 0x1cfbb4f4 System.Byte[]
0x2601c728 0.6MB large free object followed by 0x260bf1d0 System.Byte[]
0x27020e88 0.7MB large free object followed by 0x270cd88c System.Byte[]
0x30000f34 0.7MB large free object followed by 0x300b2a44 System.Byte[]
0x3100cd34 0.6MB large free object followed by 0x310a0ffc System.Byte[]

居然 managed heap 中空闲空间有 765MB!你看着内存碎片严重的。看看线程统计：

ThreadCount: 63
UnstartedThread: 0
BackgroundThread: 62
PendingThread: 0
DeadThread: 0

```
                                      PreEmptive   GC Alloc                    Lock
       ID  ThreadOBJ        State      GC          Context              Domain       Count APT
Exception
   0   0x2dc 0x00140280       0xa020 Enabled   0x00000000:0x00000000 0x0013b290       0 MTA
   2   0x2e8 0x00148ba0       0xb220 Enabled   0x00000000:0x00000000 0x0013b290       0 MTA (Finalizer)
   6   0x630 0x00203d38       0x1220 Enabled   0x6e1bfc98:0x6e1bfff4 0x0013b290      0 Ukn
   7   0x570 0x00216008       0x1220 Enabled   0x00000000:0x00000000 0x0013b290       0 Ukn
   9   0x548 0x0021e418     0x800220 Enabled   0x00000000:0x00000000 0x0013b290       0 MTA
(Threadpool Completion Port)
  10   0xb68 0x03ee1810     0x800220 Enabled   0x00000000:0x00000000 0x0013b290       0 MTA
(Threadpool Completion Port)
…
```

大量的线程堵塞在下面的 callstack 上:

```
0x07aef3a8    0x77f88f13 [FRAME: NDirectMethodFrameStandalone] [DEFAULT] I4 System.Net.UnsafeNclNativeMethods/OSSOCK.recv(I,I,I4,ValueClass
System.Net.Sockets.SocketFlags)
0x07aef3bc    0x7b28a536 [DEFAULT] [hasThis] I4 System.Net.Sockets.Socket.Receive(SZArray UI1,I4,I4,ValueClass System.Net.Sockets.SocketFlags)
0x07aef3fc    0x03e9a540 [DEFAULT] [hasThis] I4 System.Runtime.Remoting.Channels.SocketStream.Read(SZArray UI1,I4,I4)
0x07aef410    0x03e9a4d5 [DEFAULT] [hasThis] I4 System.Runtime.Remoting.Channels.SocketHandler.ReadFromSocket(SZArray UI1,I4,I4)
0x07aef424    0x03e9a4a3 [DEFAULT] [hasThis] I4 System.Runtime.Remoting.Channels.SocketHandler.BufferMoreData()
0x07aef42c    0x03e96d61 [DEFAULT] [hasThis] I4 System.Runtime.Remoting.Channels.SocketHandler.Read(SZArray UI1,I4,I4)
0x07aef450    0x03e9856a [DEFAULT] [hasThis] I4 System.Runtime.Remoting.Channels.Tcp.TcpFixedLengthReadingStream.ReadByte()
0x07aef458    0x799ae516 [DEFAULT] [hasThis] Void System.IO.BinaryReader.FillBuffer(I4)
0x07aef46c    0x799ae77a [DEFAULT] [hasThis] UI1 System.IO.BinaryReader.ReadByte()
0x07aef474    0x799c0824 [DEFAULT] [hasThis] Void System.Runtime.Serialization.Formatters.Binary.SerializationHeaderRecord.Read(Class
System.Runtime.Serialization.Formatters.Binary.__BinaryParser)
0x07aef49c    0x799c079c [DEFAULT] [hasThis] Void System.Runtime.Serialization.Formatters.Binary.__BinaryParser.ReadSerializationHeaderRecord()
0x07aef4ac    0x799c0309 [DEFAULT] [hasThis] Void System.Runtime.Serialization.Formatters.Binary.__BinaryParser.Run()
0x07aef4e0    0x799c00da [DEFAULT] [hasThis] Object System.Runtime.Serialization.Formatters.Binary.ObjectReader.Deserialize(Class
System.Runtime.Remoting.Messaging.HeaderHandler,Class System.Runtime.Serialization.Formatters.Binary.__BinaryParser,Boolean,Class
System.Runtime.Remoting.Messaging.IMethodCallMessage)
0x07aef520    0x799bfd08 [DEFAULT] [hasThis] Object System.Runtime.Serialization.Formatters.Binary.BinaryFormatter.Deserialize(Class
System.IO.Stream,Class System.Runtime.Remoting.Messaging.HeaderHandler,Boolean,Class System.Runtime.Remoting.Messaging.IMethodCallMessage)
0x07aef540    0x03e98445 [DEFAULT] Class System.Runtime.Remoting.Messaging.IMessage
System.Runtime.Remoting.Channels.CoreChannel.DeserializeBinaryRequestMessage(String,Class System.IO.Stream,Boolean,ValueClass
System.Runtime.Serialization.Formatters.TypeFilterLevel)
0x07aef55c    0x03e97eeb [DEFAULT] [hasThis] ValueClass System.Runtime.Remoting.Channels.ServerProcessing
System.Runtime.Remoting.Channels.BinaryServerFormatterSink.ProcessMessage(Class System.Runtime.Remoting.Channels.IServerChannelSinkStack,Class
System.Runtime.Remoting.Messaging.IMessage,Class System.Runtime.Remoting.Channels.ITransportHeaders,Class System.IO.Stream,ByRef Class
System.Runtime.Remoting.Messaging.IMessage,ByRef Class System.Runtime.Remoting.Channels.ITransportHeaders,ByRef Class System.IO.Stream)
0x07aef5c0    0x03e97a37 [DEFAULT] [hasThis] ValueClass System.Runtime.Remoting.Channels.ServerProcessing
Microsoft.ApplicationBlocks.MCSChina.Remoting.CompressServerChannelSink.ProcessMessage(Class
System.Runtime.Remoting.Channels.IServerChannelSinkStack,Class System.Runtime.Remoting.Messaging.IMessage,Class
```

System.Runtime.Remoting.Channels.ITransportHeaders,Class System.IO.Stream,ByRef Class System.Runtime.Remoting.Messaging.IMessage,ByRef Class System.Runtime.Remoting.Channels.ITransportHeaders,ByRef Class System.IO.Stream)

0x07aef5ec 0x03e96919 [DEFAULT] [hasThis] Void System.Runtime.Remoting.Channels.Tcp.TcpServerTransportSink.ServiceRequest(Object)

0x07aef614 0x03e9680f [DEFAULT] [hasThis] Void System.Runtime.Remoting.Channels.SocketHandler.ProcessRequestNow()

0x07aef63c 0x03e966c0 [DEFAULT] [hasThis] Void System.Runtime.Remoting.Channels.RequestQueue.ProcessNextRequest(Class System.Runtime.Remoting.Channels.SocketHandler)

0x07aef640 0x03e96661 [DEFAULT] [hasThis] Void System.Runtime.Remoting.Channels.SocketHandler.BeginReadMessageCallback(Class System.IAsyncResult)

0x07aef66c 0x7b29153c [DEFAULT] Void System.Net.Sockets.OverlappedAsyncResult.CompletionPortCallback(UI4,UI4,Ptr ValueClass System.Threading.NativeOverlapped)

看明白了么?问题是由于大量的 socket 操作无法完成，导致大量的小块 managed 内存被 pin 住无法被 GC 释放和移动。由此引发的内存碎片很快耗光了连续的内存块。内存碎片导致找不到连续内存来完成新的内存分配请求，发生 OutOfMemory 异常。

接下来要回答的是为何晚上 socket 操作无法快速完成。答案是 bt。

客户那里的员工习惯白天干活，在下班后开着电脑用 bt 通宵下载。由于 bt 暂用了大量的网络资源，使得程序的 socket 操作无法及时完成，所以问题只在晚上发生。

在这个案例中，最后通过改善网络环境解决了问题。但是从中可以看出 CLR 内存管理还是有可以改善的地方。在 CLR 2.0 中，内存碎片问题有了极大的改善，关于该问题更多的讨论可以参考：

http://blogs.msdn.com/maoni/archive/2004/12/19/327149.aspx
http://blogs.msdn.com/maoni/archive/2005/10/03/476750.aspx
http://blogs.msdn.com/tess/archive/2005/11/25/496898.aspx

碎片的其它原因

除了 socket 操作外，引发内存碎片问题的另外一个凶手往往是 ASP.NET 的编译模式。在 ASP.NET 站点的 web.config 文件中，可以通过<compilation>设定来改变动态编译模式。在 CLR 1.1 中，如果编译模式设定 debug=true，默认会关闭 ASP.NET 的 batch compilation 功能。这样每一个 ASPX 文件都会在第一次被访问到的时候编译成一个动态的 assembly 加载到内存中。如果客户有很多页面，随着程序运行，动态 assembly 越来越多，也会导致严重的内存碎片。

在下面这篇文章中，发现 high memory 后，首先就建议检察 compilation 模式

Quick things to check when you experience high memory levels in ASP.NET
http://support.microsoft.com/kb/893660/en-us

除了 ASP.NET 的页面编译外，如果使用 XSLT，也容易遭遇类似问题：

PRB: Cannot unload assemblies that you create and load by using script in XSLT
http://support.microsoft.com/kb/316775/en-us

[Handle Leak]

除了内存泄露外，常见的资源泄露还有 Handle Leak。解决 handle leak 的思路跟解决 unmanaged leak 的思路一样，关键在于找到泄露掉的 handle 的类型和分配时候的 callstack。

第二章中介绍过使用 Application Verifier 可以让操作系统在分配 handle 的时候纪录下 callstack。跟 unmanaged heap leak 稍有不同的是，在当前 Windows 实现中(Windows 2003/XP/2000)，这些 callstack 信息不会保存在 Dump 中，所以不能使用类似 umdh 这样的工具来方便获取 callstack，而是要用 windbg 做实时调试。Windbg 的!handle 命令和!htrace 命令提供了这样的功能。

大致的步骤是，首先激活 Application Verifier 中目标程序的 handle trace。启动目标程序，加载 windbg。在问题发生以前纪录下当前进程所创建的所有 handle object 信息，以及每个 handle 分配时候的 callstack。这些信息是在 windbg 输出窗口看到的，所以需要保存 windbg 的输出。重现问题，让 handle 增长，用 windbg 纪录下问题发生后的所有 handle object 信息以及每个 handle 分配时候的 callstack。最后比较前后两次的纪录，分析什么类型的 handle 持续增长，分配时候的 callstack 是什么。具体整理下来的步骤是：

1. 安装 Windbg 到 C:\Debuggers 目录
2. 安装 Application Verifier。对需要调试的程序激活“Handles - Detect invalid handle usage”
3. 确保编译的时候生成了目标程序的 symbol 文件，并且统一部署到自定义的 symbol 目录。
4. 启动目标程序
5. 启动性能监视器开始监视目标程序，添加
Process
Processor
Memory
System
6. 启动 windbg，设定好 symbol 路径，开始监视目标程序
7. 在 windbg 命令窗口输入:
.logopen c:\log.txt
记录 windbg 输出到文本文件
8. 运行!handle 和!htrace 命令，等待命令执行完成
9. 输入下列命令避免 1st chance exception 干扰问题的重现
SXN *
SXN av
SXN clr
SXN eh
SXN cc

10. 输入 g 命令，让目标程序开始运行
11. 重现问题，问题发生后用 windbg 挂起程序
12. 再次输入!handle 和!htrace，保存 handle 信息
13. 运行.dump 命令抓取问题发生后的 dump 文件
14. 输入.logclose 关闭 log 文件
15. 收集性能日志文件

通过分析 C:\log.txt 文件中问题发生前后的差异，找到发生泄漏的 handle 是在什么 callstack 中分配的，以及 handle 的类型。在结合源代码定位。

题外话和相关讨论

除了 memory leak 和 handle 外，其它类型的资源泄露还有 GDI Leak 和 desktop heap high usage。

GDI Leak

关于 GDI Leak，可以参考：

Detect and Plug GDI Leaks in Your Code with Two Powerful Tools for Windows XP
http://msdn.microsoft.com/msdnmag/issues/03/01/GDILeaks/default.aspx

Resource Leaks: Detecting, Locating, and Repairing Your Leaky GDI Code
http://msdn.microsoft.com/msdnmag/issues/01/03/leaks/default.aspx

Desktop heap 是一类特殊的内存。下面这些 kb 对此有一些描述：

INFO: Services, Desktops, and Window Stations
http://support.microsoft.com/kb/171890/en-us

PRB: User32.dll or Kernel32.dll fails to initialize
http://support.microsoft.com/kb/184802/en-us

"Out of Memory" error message appears when you have a large number of programs running
http://support.microsoft.com/kb/126962/en-us

Desktop heap

Windows 除了使用进程来管理资源外，还是用 Session 和 desktop 来管理资源。比如只有在

同一个 Session 里面的进程才可以共享剪贴板数据，Windows Message 只能在属于同一个 desktop 的进程之间传递。

desktop heap 是操作系统管理的，为不同 session 创建的，由同一 session 内所有 desktop 共享的内存。当创建进程，创建 GUI 的时候，都会消耗 desktop heap。当 Desktop Heap 用光后，系统中各种莫名其妙的问题就会发生。比如无法创建新进程，无法弹出菜单，API 调用会莫名其妙地出错。

Desktop heap 用光的原因往往是太多进程同时运行，或者创建了太多 GUI object。怀疑是 Desktop heap 相关问题的时候，首先可以用下面这篇文章的方法来检查是否 Desktop heap 用光：

A new System log entry is not generated if the desktop heap is exhausted in Microsoft Windows 2000
http://support.microsoft.com/kb/810807/en-us

如果确认是 desktop heap 问题后，检察系统是否有太多进程在同时运行，(我曾经见过一百多个 dllhost 进程同时运行的情况)，或者某一程序有特殊的 GUI 操作。在优化系统，优化代码后，如果的确需要使用很多 desktop heap，可以参考前面的 kb 改变注册表来做调整。监视 Desktop heap 可以使用：

Desktop Heap Monitor Version 8.1
http://www.microsoft.com/downloads/details.aspx?FamilyID=5CFC9B74-97AA-4510-B4B9-B2DC98C8ED8B&displaylang=en

除了上面这些系统资源外，还有其它很多资源都可能出现问题，比如 TCP Port 用光。

When you try to connect from TCP ports greater than 5000 you receive the error 'WSAENOBUFS (10055)'
http://support.microsoft.com/kb/196271/en-us

前面还提到过 CRT Debug heap 也可以用来检查内存泄露。因为 CRT 通过 malloc 函数代理内存分配，所以可以记录泄露内存是什么时候分配的。在 C++中，往往建议用 new 来分配内存。这一定是一个好建议吗？

C++ operator new[]和 Debug Heap
http://eparg.spaces.live.com/blog/cns!59BFC22C0E7E1A76!1490.entry

写在最后

排错的工具还有下面一些。关于详细信息在网上都可以找到

Filemon/Regmon
监视文件/注册表访问信息，发生 access denied 的时候，还会显示出进程所用的用户名

Netmon
网络抓包

Process Viewer
检查每一个进程的详细信息，包括打开了哪些 kernel object，分别有什么权限

Netstat/TCPView
检查端口的开放情况

Spyxx
检查窗口相关信息，包括属性，Windows Message

Performance Monitor
监视系统范围或者进程范围内的信息，对于内存使用，handle 使用，CPU 使用，线程争用，IO 访问等问题的定位提供详细统计信息

Internet Information Services Diagnostic Tools
这是微软最近发布的最新一款调试工具。集成了自定义条件抓取 dump，自动分析内存泄露功能。

我收藏夹中的一些跟调试相关的站点和书籍:

DebugInfo 整体介绍了 windbg 的使用，有很多例子，适合初学者
http://www.debuginfo.com/

Debug Tutorial Part 1: Beginning Debugging Using CDB and NTSD 介绍 CDB (也就是 Windbg 的引擎)的使用，里面的 part3 介绍 heap debugging，非常详细
http://www.codeproject.com/debug/cdbntsd.asp

Production Debugging for .NET Framework Applications .NET 调试的详细介绍。很全面的 bible
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/DBGrm.asp

Debugging Applications for Microsoft .NET and Microsoft Windows (Hardcover) 非常有名的调

试书籍
http://www.amazon.com/gp/product/0735615365/ref=pd_sim_b_1/102-3584666-9558510?%5Fencoding=UTF8&v=glance&n=283155

一些 blog:

Mike Stall's .NET Debugging Blog
http://blogs.msdn.com/jmstall/default.aspx

Yun Jin's WebLog
https://blogs.msdn.com/yunjin/archive/category/3452.aspx

Flier's Sky
http://flier.cnblogs.com/

如果时间允许，计划中的第三章准备针对下面一些典型问题。这些问题的分析在前面两章都出现过，上面的链接中也有分别的介绍。精力允许的话，准备综合起来介绍整体思路，技巧和经验，有可能的话提供 debug log。

1. .NET Application Debug, memory leak, crash and deadlock
2. Crash with heap corruption
3. Memory leak or OutOfMemory
4. Handle leak
5. High CPU

本文目前就到此为止了，CCF和身边的很多朋友对本文提供了有力的帮助和支持。如果这些内容能够带来一些乐趣，目的就达到了☺。如果有什么想法，请联系 lixiong@microsoft.com
Messenger: eparg@msn.com