Windows 用户态程序高效排错 (非博文图书版)

该 PDF 收集了作者 2006 年 6 月底完成的一系列排错文章。并不是 2007 年底博文视点出版发行的同名书籍。

2006 年 4 月，作者整理出一些跟排错相关的文章，制作成 PDF 放到网上。下载地址分别在
http://www.cnblogs.com/lixiong/archive/2006/08/16/475520.html
http://blogs.msdn.com/lixiong/archive/2006/08/03/687357.aspx

您现在看到的 PDF，就是作者当时完成的。该 PDF 并不是 2007 年底发行的图书版。

作者制作完这个 PDF 后，有用了大约一年时间对该 PDF 继续编加整理，在 2007 年底，博文视点出版了同名图书，Windows 用户态程序高效排错

该 PDF 和同名图书的相同点：

PDF 中的内容，基本等同图书前两章的内容。

该 PDF 和同名图书的区别在于:

PDF 大约有 93 页，同名图书大约有 400 页

同名图书包含了.NET Debugging 的内容，和对崩溃，性能，资源泄露排错的总结，分别作为第三章和第四章

同名图书修正了 PDF 中的一些疏漏，调整了部分章节的顺序。比如更清楚地描述了 ShellExecute 案例，添加了 Windbg 的入门章节等

根据作者和出版商达成的协议，作者在该书出版两年后，可以免费提供该书电子版的下载。目前并没有图书版书籍的 PDF。

本文介绍什么?

这是一篇介绍 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。同时，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。
下面这些问题截取于本文后面要讨论的一些例子:

- 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，偶尔发生内存访问错误，然后崩溃。
- 程序消耗的 handle 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。
- VC 程序中，使用 ShellExecute 打开一个本地的 TXT 文件。TXT 格式默认打开方式关联到 UltraEdit。发现在 UltraEdit 中除了打开这个 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。

本文的组织结构:

后面分三部分来解释

- 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，可以获取关于问题的关键信息。结合对应的知识就可以分析出问题的根源。
- 第三部分结合前两部分的内容，针对常见的几大类问题进行了总结。包括资源泄漏 (resource leak), 性能问题 (performance), 崩溃 (crash), CLR 调试技巧和 COM+调试技巧。

前两部分已经完成，正在整理。第三部分还没开始写。先把写好的给出来，听听大家的意见。

本文的使用方法和说明:

本文着重介绍思路，经验和工具。对于具体的知识点和工具，给出重点和方法，但是不会做详细的介绍，原因是 MSDN 的解释更加丰富和权威。具体信息可以参考文章中给出的链接。

在第一章阅读过程中，可能会发现某些术语，或者知识点并不熟悉。但是，这些并不会阻碍理解整体的思路。鼓励自主学习。通过 Internet，都可以找到这些知识点的具体说明。同时，很多知识点会在第二章来说明。

内容组织上尽量先给现实的情况和问题，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。在拿到问题后，建议先自主分析如何去获取线索；拿到线索后，建议自主分析如何利用线索。

后面的案例都是这两年来亲手处理过的案例，但是处理的过程是在整个团队(微软大中华区开发语言和工具支持部)的帮助下完成的。没有整个团队的帮助，这些案例和这篇文章都无法完成。

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。在完成这篇文章的时候并不是严肃的，所以大家也别用严肃的眼光来看。希望大家当成一种趣味来阅读。

您的反馈和最新动态：

我保证努力提供正确的内容，但不保证这个事实。所以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。

文章的补充，修正，答疑和其它相关信息都会及时发布到下面的 blog 链接，欢迎您通过这个 blog 提出反馈:

http://blogs.msdn.com/lixiong

第一部分，思考问题

1.0 热身运动：

在进入后面的章节以前，首先跟大家分享导师给我的一个问题：

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排布的。
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一个，远比 Windows 的内存管理简单。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！

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，在相同的机器上，有的时候会同时打开除了这个 TXT 以外的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。

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。但是，如果真的是这样，行为也应该是唯一的。

但是，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。在如果我胸有成竹地告诉客户，这种问题肯定跟代码不相关，唯一可能导致问题的是一些防病毒程序之流 (防病毒程序的确是很多问题的根源，但是有时也是挡箭牌)，那么我们就失去了找到真相的机会。

继续看当时是如何深入这个问题的。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

文件包含显示一个 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。当用户选择菜单命令后，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 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;}
```

有了这样的问题背景，就把问题跟用户的代码联系起来了。额外打开的文件的确是跟这个程序相关的。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，持续直接调用菜单处理函数，而不是通过点选菜单项触发。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。

户是通过在 PreTranslateMessage 函数中截获鼠标的 WM_RBUTTONDOWN 消息，然后显示菜单。显示完毕后，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 0，表示消息没有被处理，需要继续发送给后继的 Message Chain。然而事实上，这个消息已经导致了客户自定义的菜单显示，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。

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，未免太早了。不仅仅是因为把 False 修改成 True 给客户带来了新问题，更重要的是，上面的分析，还是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。如果修改成 True，上面的功能就无法正常工作。

在缩小问题区间后，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 WindowsMessage 的详细信息在 windbg 中打印出来。

通过比较分析正常情况 (return TRUE) 和异常情况(return FALSE)下 wt 命令和 WindowsMessage 函数的调试输入，问题基本上就很容易解决了。只需要分析 callstack 的差异就能够搞清楚原因。

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。

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 是如何导致问题发生的吗？

```
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
```

虽然从 CDropTarget::Drop 这个函数猜到 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. 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 的消息传播到后面导致的结果就是在 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(它不能依赖现有的消息循环因为 ShellExecute 本身不拥有窗口)，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。

还剩下一个疑问是，没有最后的鼠标松开消息，拖动的操作到底是怎么完成的？这个问题 debug 起来不是那么容易，所以也不得不适可而止了。

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。

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 1]

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 2]
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。我不清楚那位工程师是怎么调试出来的，总之根据调试器的输出，他给客户说：“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？说不定是因为:

> 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

看来要区分事实和谎言，不是那么容易……

[案例 3]
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，但是发现操作长度大于 4GB 的文件的时候，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，解决这个问题的方法，应该用 GetFileSizeEx 调用代替 GetFileSize。其实，不单单是客户会犯这个错误，.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 4]
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 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！

客户抱怨，刚刚开发完成的 ASP.NET 工程测试阶段一切正常，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 文件得到的。Log 文件记录的是每个时刻 Session 中的内。

拿到这个问题后，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，没有多少详细观察这个问题的机会。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

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观察这个现象。
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。如果不是，那看看测试用的 session value 是不是丢。如果这个也丢，9 成是用户调了 Session.Clear.
3. 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。这 9 成还是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

lost 导致的。如果是 cookie 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。

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. 在代码中对 session 操作的地方，写 log 到以 sessionid 为文件名的文件里面去
3. 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，重定向是通过返回客户端 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:

根据以往的经验， session lost 还应该特别检查:

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, 其实是两个 session 的。也就说，session 的维护是依靠 session id 这个 InProc cookie,而这个 cookie 完全是由 IE 控制的。用户在一个 IE 进程里面去写 session, 在另外一个 IE 进程里面去读，当然不成功

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！

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。

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，这个问题是跟表格的 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。所以系统组件，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！MSDN 的文章是在.NET Framework 2.0 beta2 的时候写的，并不准确！”

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. 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，但是也可以测试出，存在某些情况下，ADO.NET 2.0 可以把性能提高了 2 个数量级。

同时，当跟其他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，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：

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，以及占整个运行时间的比例。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 client driver。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 Database provider 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！)

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，要用思考来驱动工具的使用
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:

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

关于平衡，取舍，双赢的一个更有趣，更权威的文档是 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，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用户 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，如果要从根源上解决这个问题，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 之间的平衡点：

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！

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. 管理员设定的是中文界面
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成中文界面
6. 变成英文界面后，如果重新启动 IIS, 不一定解决问题

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, 跟客户的设定一直
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，其中某一次该问题会发生
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 SDK 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

同时发现欧洲很多客户最近也有类似问题发生，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，跳到第四步。
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)来填充这个 HashTable
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 上限制只有某一个固定 IP 的客户端才能访问。用这个客户端来发起请求
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. 的确是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，这个 codepath 是很难执行到的。

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目录。但是 SP1 中添加了某一个额外的配置选项，使得 ResourceManager 要额外访问一次 web.config 配置文件。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。因此在访问 web.config 过程中，发生了 Access Denied，继而导致了 first chance CLR exception，所以 ResourceManager 认为某些地方出了问题，于是 ResourceManager 决定加载中立资源，英文就代替了中文。换句话说，ResourceManager 意识到了新添加的功能运行的时候可能失败，但是没有考虑到这个功能需要访问 web.config。而在 web 程序中，没有权限去访问 web.config 是很常见的。由于没有 web 程序中常见的情况特殊处理，导致这个新添加的功能影响了大面积的 web 程序。

所以，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。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

如果仔细分析这个案例的解决过程，会发现当时距离胜利之差一点点。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，而是想说明问题解决问题的核心武器是思考，知识跟工具是辅助。如果这一章的内容能够开阔一点思维，在遇上问题的时候能够多想一分钟再动手，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。这些知识对于开发过程也是非常重要的。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，进程间通信。

本文不会对知识点本身做介绍，而是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 的详细介绍防到第四节。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，后面的例子都会是用这个目录

2.1 汇编，CPU 执行指令的最小单元

汇编是 CPU 执行指令的最小单元。比如在下面一些情况下，必须使用汇编来分析问题:

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，只有看汇编。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。建议先学习简单的 8086/80286 的汇编原理，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

下面的案例介绍如何通过汇编来检查编译器的行为，函数参数的传递和性能相关的问题。

[案例 1]
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汇编然后分析。有兴趣的话先自己调一下，看看跟我下面的分析是否一样

我的分析是基于 VC6, 默认的 release mode 的设定:

```
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)
```

如果还不太明白，可以结合上面的注释参考下面的解释:
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 是整性数组，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。

如果这段汇编让你来写，你能写得编译器更好一点吗？

[案例 2]
不要迷信 compiler 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 环境下工作正常。

```
// 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

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 compiler 的 bug:

1. 程序中除了 cout 和 printf 外，没有牵涉到系统相关的 API. (关于 cout, printf 跟系统 API 的关系，后面有介绍), 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，唯一的可能性就是语法有问题或者编译器有问题。
2. 检查语法后没有发现异常。同时，如果调整一下 std::transform 的位置，在 for loop 后面调用的话，问题也不会发生。语法问题不会导致这么奇怪的结果。

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 compiler 的 bug:

在 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。编译器对参数的传入，读取处理不统一，导致了这个问题。

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

[案例 3]
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 caller 传到 callee 的，当 callee 使用这个指针的时候问题发生。接下来就应该检查 caller 为什么没有把正确的指针传入 callee。

奇怪的时候，caller 中这个指针已经初始化正常了，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]。是不是跟案例 2 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，接受 5 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。由于只能观察到事后的崩溃，只有根据经验猜测版本不一致的起因。最有可能的是：

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从没有更新的服务器同步到了更新后的服务器：

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

[案例 4]
分别在 debug/release 模式下运行下面的代码比较效率：debug 比 release 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？

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。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。用户态程序如果处理完了异常，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。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。异常能够改变代码执行次序，所以异常发生时刻的信息是非常重要的。异常发生的时间，地址，导致异常的指令，异常导致的结果可以精确描述出问题是怎么发生的。

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，操作系统在程序终止

前，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。不管 C++的异常是否有最终的 catch, unwind stack 都要发生的。但是如果有对应的 catch 处理了这个异常，2nd chance exception 就不会在调试器里面观察到了。

上面的解释主要是讨论异常的作用和影响。详细的信息，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，调试器是怎么参与的，下面几篇文章有非常详细的介绍。

所有介绍异常的文章都应该引用的一篇文章。虽然比较长，但是却字字珠玑。

A Crash Course on the Depths of Win32™ Structured Exception Handling
http://www.microsoft.com/msj/0197/Exception/Exception.aspx

用来触发异常的 API, throw 关键字就调用的这个. 里面的 remark section 详细介绍了异常处理函数是如何被分发的

RaiseException
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/debug/base/raiseexception.asp

[案例 1]
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？

我的解法是使用 SHE，加上局部变量析构函数在异常发生时候也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 2]
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 文件进行分析。但是问题再次发生后，Dr. Watson 并没有记录下来 dump 文件。

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。

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;
```

运行后 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 ExitProcess 正常退出，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。换句话说，需要在调试器下运行程序观察。

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

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。当 C++异常发生的时候，程序就由于 unhandled C++ exception 崩溃。修改代码添加对应的 catch 后，问题解决。

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。深入研究后，却先报告在:

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

这里例子并不是要解释如何分析异常的。因为异常发生的原因是跟程序的设计密切相关，分析的时候因情况而异。使用异常来解决问题的关键在于如何在恰当的时机获取恰当的信息。这项技巧需要对异常，操作系统和调试器融会贯通后才能很好地掌握。

最后讨论一下通知(notification). 在MSDN中没有找到多少关于notification的信息。所以没办法对这个东西做一个详细的定义。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。比如:

[案例 3]
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。换一个调试环境也一切正常。

这个问题的思路非常简单，既然只有一台机器有问题，说明肯定是环境的原因。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。那好，下一步就需要知道为什么加载的是 dao350.dll？ 首先，我们根本不熟悉什么情况下会触发加载 DAO360.dll 的操作，也不知道到底是什么函数来加载的。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，这一个简单的 wt 命令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，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. 跟程序相关的就是 `VB6!_DBErrCreateDao36DBEngine` 这个函数。或许应该仔细检查这个函数。

幸运的是，在检查这个函数前，观察到VB6.EXE这个module的版本好像有问题。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。

异常往往是导致问题的关键。由于操作系统提供了异常和通知跟调试器的交流机制. 深入理解这些机制，并且灵活使用，对解决问题有非常大的帮助。

题外话和相关讨论:

如果要想在 windbg 中获取关于异常的详细信息，最好的方法是用 windbg 截取 1st chance exception 进行分析。如果崩溃已经发生，显示错误的对话框已经弹出后，还是有机会找到对应 exception 的信息。系统会把异常信息和异常上下文保存下来，通过检查 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。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
```

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。(对于如何解决崩溃的更详细整理，打算放到第三章)。可以参考：

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

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. 可以提供更友好的界面
2. 可以主动获取异常的一些信息保存下来以便后继分析

这样做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。在 COM+上，这种技术叫做 failfast。如果要获取 COM+程序上 crash 的信息，需要破费一番周则，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

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。

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。

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。不过 **Kwan Hyun Kim** 提供了另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，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，所以它可以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。

幸运的是, HeapManager 的确提供了上述的功能。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，它们还可以修改其它的调试选项，后面都有说明。pageheap 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

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。

从上面的例子，可以很清楚地看到 pageheap 工具对于检查这类问题的帮助。同时也可以看到，pageheap 还是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，后面有一个小节专门解释这方面的问题。

总结一下，前面讨论的是 heap 操作失误。除了这一类问题，还有一类就是内存泄漏。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。要解决这类问题，pageheap 不是很好的工具。往往需要用到 API Hook 技术来检查这样的问题。API Hook 的原理是通过工具使用 DLL Injection 的技术挂接目标程序中所有内存分配，释放用到的 API，同时作薄记工作。当内存涨到一个比较可观的数量的时候，通过检查工具的薄记来找到目标程序那些地方分配了内存却没有释放。由于检查内存泄露是一个比较大的题目，所以放到第三章作详细讨论。

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大块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。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

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。在第三部分，有关于如何检查内存访问错误，stack overflow 和内存泄漏的详细说明

题外话和相关讨论:

关于下面这段代码在激活 pageheap 后不会崩溃的解决方法是使用/unaligned 参数。详细情况请参考 KB816542 中关于/unaligned 的介绍

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

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。

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 是连续调用了 delete 和 delete []。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 pointer 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 pointer 都是泄露的内存地址。由于每一个 heap pointer 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]
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。

前面提到了 debug/release 模式下，相同的代码可能出现不同的结果。原因在于这两种模式下，C 语言运行库的实现有区别。

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。

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。

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。下面这些是常见的一些运行在 API 和 CRT 上的层次。

OLE/COM/DCOM/COM+
Web Application, ASP
Windows Form
.NET Framework
.NET Framework + Web Application (ASP.NET)
ASP with COM+ (3 tire web application)
.NET Framework + Remoting
Office Automation and Script Development
…

每一个层次都对应某一类型的应用，如果展开都可以独立作为一个章节。对于 COM+, .NET

Framework 的调试，在第三章有专门的话题。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

这一节专门介绍调试器，特别是 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。

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，效果跟调试一个正在运行的进程是一样的。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。

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。)

大致归纳一下调试器一般可以直观地看到哪些方面的信息:

- 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
- 反汇编某一个地址上的指令
- 察看某一个内存地址是否可以读/些,内容是什么
- 列举线程的 call stack (需要 symbol)
- 切换到 call stack 上的任意函数的上下文，察看这个函数当前的局部变量
- 对某一个地址上的信息，指定 symbol 中的某一个类型格式化地显示出来
- 查看和修改内存地址上的数据或者寄存器上的数据
- 通过简单命令查看格式化的系统信息，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，打印出传入的参数
- 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

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。这个时候可以用 ctrl+S 打开 symbol 设定窗口，然后可以根据下面的文章来设定如何自动从网上下载 Microsoft 的 symbol，以及设定本地的 symbol cache
http://www.microsoft.com/whdc/devtools/debugging/debugstart.mspx

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
```

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。
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.
```

可以看到，msvcrt 并没有对应的 symbol 加载，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 log:

```
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.
```

这下子 symbol 就加载起来了。

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。

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。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。通过上面的命令，往往就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 命令可以搜索内存。一般来说，用在下面的地方:

1. 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 module 区域搜索这些字符串出现的地址，然后再搜索这些地质用到什么代码中。于是就可以找出这些内存是在什么地方开始分配的。
2. 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情况。

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.
```

用这个命令，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 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.

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。

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
```

如果要找某一个符号，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 命令跟 x 命令相反，dds 把某一个地址对应到符号。当看到某一些数据怀疑可能对应到符号的时候，可以用这个命令。 比如要看看当前函数的 caller 是谁，就可以检查 ebp 下面有哪些函数地址:

```
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，所以这个命令在检查上面两种类型的时候特别有用:

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
```

下面再用一个小例子来演示

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。
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
```

是不是看到跟前面的结果不一样？有的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。

接着，根据前面的序号来切换到对应的函数上下文。用.frame 命令，结合上面的输出找到 frame number，切换到 `exceptioninject!foo1` 这个函数:

```
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
```

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。下面这个命令对地址 0x0039ba80 指定用 MyCls 类型来显示:

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

另外，查看 heap 可以参考!heap 命令。查看 Exception 可以参考.exr 和.cxr 命令。(这两个命令在前面有相关的例子和 KB 文档)

接下来看看 live debug 中的一些有用的命令。在 windbg 中，g 命令让程序继续执行，ctrl+break 让程序停止

Live debug 中最有价值的命令是:

1. 设定断点
2. wt 命令

windbg 里面可以设定条件断点，比如: 当 gInfo 这个全局变量在被修改 100 次以后，如果 stack 上的第二个参数是 100，那么就停下来，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

wt 命令的作用是 watch and trace data。 它可以跟踪一个方法的所有执行过程，并且给出统计信息。其实，他的原理也是条件断点。每次在 call 的后面一条指令，设定条件断点，同时比较 esp，ebp 的变化，就可以确定函数之间的调用关系。

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 命令让程序继续执行

```
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 命令执行到 foo3 函数执行完为止。随着函数的执行，windbg 打印出调用过的子函数。如果想得到详细信息，可以不用-l2 来设定 2 的深度。该命令最后给出统计信息。wt 命令，无论是观察函数执行过程和分支，或者是性能评估上，都是很有帮助的。

条件断点 (condition breakpoint):

1. bp+地址/函数名字可以在某个地址上设定断点。
2. ba (break on access)用来设定访问断点，在某个地址被读/写的时候停下来。
3. Exception 断点。当发生某个 Exception/Notification 的时候停下来。详情请参考 windbg 帮助中的 sx(Set Exception)小结。

条件断点(condition breakpoint)的原理是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。

通过 bp address “”的格式，可以让调试器在 address 断点触发停下来后，执行“”里面的自定义调试器命令。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'
```

接下来用下面的代码演示难一点的:

```
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)

是不是很熟悉?整个第一个引号中执行结果就是打印出当前 exceptioninject!i 的值。接下来的.echo 命令换行，然后 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:

```
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)
```

调试器同时还提供了伪寄存器的功能。考虑这样的情况，要记录程序执行过程中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，在对应的函数上做记录。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。

根据上面的知识，分析一下 Step Out 怎么实现。Step Out 的定义是” Target executes until the current function is complete.”。根据这个定义，只需要在当前函数的 return address 上设定断点就可以了。当前函数的 return address 可以通过 EBP+4 来获取。但如果简单地在 bp EBP+4 上面设定断点有两个问题:

1. 无法判断是否是递归调用还是函数返回，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。

Windbg 的 shift+f11 实现了 Step Out。如果观察 shift+F11 的实现，可以看到:

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

这里的/1 参数避免了第二个问题，/c @$csp 参数避免了第一个问题，@$ra 伪寄存器直接表

示函数返回地址。多方便 :-)

如果有精力，还可以再复杂一点，甚至写一个 script 来自动完成一系列工作。比如可以在 VirtualAlloc/VirtualFree 上设定断点，保存返回/传入的参数和 callstack，然后用 script 来自动分析内存泄漏。

最后来看看 windbg 一些很实用的功能。

Remote debug:

在 windbg 中，可以用.server 命令在本地创建一个 TCP 端口，或者 named pipe, 这样远程的 windbg 可以连接调试。双方都可以输入命令，执行结果在双方的 windbg 上都显示出来。Remote debug 方便调试不易部署的应用程序。

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，然后退出。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，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。第三章会做详细讲解。

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。

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 都停止了。在 CAsyncSocket 对应的网络处理函数上设断点，发现断点都没有触发。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一个 object 释放，这个 object 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。同时 CPU 使用率保持为 0。要解决这个问题，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？

Windbg 提供了下面几个很好的命令来调试 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，对于上面说的第三种情况，就算知道在等待某一个不可能释放的 CriticalSection，但是若找不出线程退出的原因，也无法解决问题。所以，还需要这个工具:

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, 用 windbg attach (!htrace 在当前 windows 系统只能用于 live debug，dump 中不保存 handle trace 信息):

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, 用!handle handlenumber f 的格式察看详细信息:

```
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
```

可以看到这注册表 handle 对应的注册表键是什么。如果某一个 handle 创建的时候带名字的话，也能看到名字是什么。往往!handle 命令就可以解决好多问题，这一个命令解决过:

问题 1: 某一个 handle leak 的问题，用!handle 命令发现 leak 的 handle 类型是 key, 该 key 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

问题 2: 某一个 deadlock 问题，发现死锁的线程都是在等待客户自定义的 event. 由于客户使用了很多 event, 无法找出每一个 event 是在什么时候创建的。但是这个聪明的客户在每次创建 event 的时候都使用了一些很有描述性的名字，所以!handle 打印出死锁线程等待的 event 的名字后，客户一下子就知道应该如何入手检查了。

接下来看看!htrace 这个命令。在用 Application Verifier 激活后，这个命令可以打印出所有 handle 的最近几次操作的 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
```

所以，结合这两个命令，基本上可以看出某一个 handle 的详细信息。结合问题发生时候的其他信息，能方便地找出线索。

上面两个命令是针对 handle 的，对于 CriticalSection,有专门的命令!cs:

比如下面一个例子，线程 13 block 在下面的 call stack:

```
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
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
```

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

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，再运行这个程序，看到效果了吗？在 windbg 中运行一下，看到：

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。

同时提一下，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:

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

这里，gle 等于 6，表示 The handle is invalid. 程序不会被崩溃。但是如果在 Application Verifier 中激活 handle 后，就会产生 break point exception。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。Application Verifier 的这个功能于让潜在问题早点被捕获。

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……

如果满足下面两个条件，问题就很容易发生:

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。这种问题往往是设计上的缺陷。一般会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

同步没有做好，不仅仅会导致死锁，性能问题，还会看到一些非常莫名其妙的现象。下面这个问题就是一个真实的例子。

[案例 1]
问题描述在:

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)
```

是不是应该建议客户去检查 SomeClass 的构造函数？问问客户是怎么访问 ArrayList 的？

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 来简化上面的工作。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。

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？

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 在即将发布的 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。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。

调试的本质：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。

所以，对于前面两个问题的答案是:

1. 取决于开发人员有没有为产品设计调试功能
2. 有 bug 不可怕，凡人都会犯错。但设计的时候没考虑过潜在的 bug，是最严重的错误。不考虑后期排错的设计，是严重的问题。

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，本小结想强调开发阶段考虑调试的重要性。应该把调试功能作为一项重要的项目需求。至少包括:

1. 多使用 assert, trace
2. 适当合理地添加 log
3. 总是编译一个 release 版本，一个 debug 版本

有了上面的准备，当神秘问题发生，一切常规调试手段都没用的时候，就再常规一点:

1. 部署 debug 版本
2. 收集程序的 log

回顾所有的案例，讽刺地发现没有任何一个问题是完全通过客户程序自身的调试功能来解决的。这不是说程序的调试功能机制对调试没用，反而是说，善于“三省吾身”的程序，根本不需要“高超”的调试技巧，自然就大隐隐于市。

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:

有一位客户开发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。希望有问题能通过 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。记录 log 的代码是:

```
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();
    }
    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从开始的 2 秒钟，上涨到 2 小时后的 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。但事实让客户很失望，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。

另一种常见问题在于写 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，又会导致性能问题因为多个线程都依赖同一个锁。

其实，设计多线程环境下高性能的 log 操作，跟设计多线程环境下的数据库操作是同一个课题。简单的 log 功能，往往比完成程序业务功能本身更有挑战性。

暂告一个段落：

前面一章主要介绍排错过程中常用的知识点和工具。其它常用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?%5Fen

coding=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。根据大家的反馈，我会近期整理一个 FAQ 出来。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，请访问：

http://blogs.msdn.com/lixiong