



Software Debugging

■ 张银奎 著

调软 试件

本书直面软件工程中的最困难任务——**侦错**
 围绕软件世界中的最强大工具——**调试器**
 全方位展示了软件调试技术的**无比威力**和**无穷魅力**

您将学习到：

- CPU的调试支持，包括异常、断点、单步执行、分支监视、JTAG、MCE等。
- Windows操作系统中的调试设施，包括内核调试引擎、用户态调试子系统、验证器、Dr. Watson、WER、ETW、故障转储、WHEA等。
- Visual C/C++编译器的调试支持，包括编译期检查、运行期检查，以及调试符号。
- WinDbg调试器的发展历史、模块结构、工作模型、使用方法、主要调试功能的实现细节，以及遍布全书的应用实例。
- 内核调试、用户态调试、JIT调试、远程调试的原理、实现和用法。
- 异常的概念、分发方法、处理方法（SEH、VEH、CppEH），未处理异常，以及编译器编译异常处理代码的方法。
- 调试符号的作用、产生过程、存储格式和使用方法。
- 栈和堆的结构布局、工作原理和有关的软件问题，包括栈的自动增长和溢出，缓冲区溢出，溢出攻击，内存泄漏，堆崩溃等。
- 软件的可调试性和提高可调试性的方法。

此外，书中还诠释了很多较难理解的概念，思考了一系列耐人深思和具有普遍意义的问题。本书是对软件调试技术在过去50年中所取得成就的全面展示，也是笔者本人在软件设计和系统开发第一线奋战10多年的经验总结。本书适合每一位希望深刻理解软件和自由驾驭软件的人阅读，不论您是否直接参与软件开发和测试；不论您是热爱软件，还是憎恨软件；不论您是想发现软件中的瑕疵，还是想领略其中蕴含的智慧！

可从高端调试网站 <http://advdbg.org/books/swdbg/>
 下载到本书的附带工具和源代码，包括：

- 80个示例程序的源程序文件和项目文件
- 浏览符号文件的SymView工具
- 与内核调试引擎对话的KdTalker工具
- 直接浏览用户态转储文件的UdmpView工具
- 显示CPU执行轨迹（分支）的CpuWhere工具
- 观察IDT、GDT和系统对象的SoZoomer工具

网上订购：www.dearbook.com.cn
 第二书店·第一服务



策划编辑：周筠
 责任编辑：陈元玉
 责任美编：胡文佳

本书贴有激光防伪标志，凡没有防伪标志者，属盗版图书。



Software Debugging

调
软
件

上架建议 软件工程 ▶ 软件调试

ISBN 978-7-121-06407-4



9 787121 064074 >

定价：128.00元



Software Debugging

■ 张银奎 著

調軟
件試

電子工業出版社

Publishing House of Electronics Industry

北京 · BEIJING



内 容 简 介

围绕如何实现高效调试这一主题，本书深入系统地介绍了以调试器为核心的各种软件调试技术。本书共 30 章，分为 6 篇。第 1 篇介绍了软件调试的概况和简要历史。第 2 篇以英特尔架构（IA）的 CPU 为例，介绍了计算机系统的硬件核心所提供的调试支持，包括异常、断点指令、单步执行标志、分支监视、JTAG 和 MCE 等。第 3 篇以 Windows 操作系统为例，介绍了计算机系统的软件核心中的调试设施，包括内核调试引擎、用户态调试子系统、异常处理、验证器、错误报告、事件追踪、故障转储、硬件错误处理等。第 4 篇以 Visual C/C++ 编译器为例，介绍了生产软件的主要工具的调试支持，重点讨论了编译期检查、运行期检查及调试符号。第 5 篇讨论了软件的可调试性，探讨了如何在软件架构设计和软件开发过程中加入调试支持，使软件更容易被调试。在前 5 篇内容的基础上，第 6 篇首先介绍了调试器的发展历史、典型功能和实现方法，然后全面介绍了 WinDBG 调试器，包括它的模块结构、工作模型、使用方法和主要调试功能的实现细节。

本书是对软件调试技术在过去 50 年中所取得成就的全面展示，也是对作者本人在软件设计和系统开发第一线奋战 10 多年的经验总结。本书理论与实践紧密结合，选取了大量具有代表性和普遍意义的技术细节进行讨论，是学习软件调试技术的宝贵资料，适合每一位希望深刻理解软件和自由驾驭软件的人阅读，特别是从事软件开发、测试、支持的技术人员和有关的研究人员。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。

版权所有，侵权必究。

图书在版编目（CIP）数据

软件调试 / 张银奎著. —北京：电子工业出版社，2008.6

ISBN 978-7-121-06407-4

I. 软… II. 张… III. 软件—调试 IV. TP311.5

中国版本图书馆 CIP 数据核字（2008）第 052852 号

策划编辑：周 笛

责任编辑：陈元玉

印 刷：北京智力达印刷有限公司

装 订：三河市金马印装有限公司

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱 邮编 100036

开 本：787×1092 1/16 印张：65 字数：1200 千字

印 次：2008 年 6 月第 1 次印刷

印 数：5000 册 定价：128.00 元

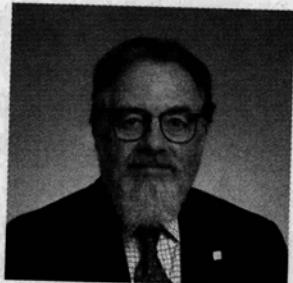
凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：(010) 88254888。

质量投诉请发邮件至 zts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：(010) 88258888。

历史回眸

我是 1949 年进入麻省理工学院 (MIT) 的。就在那一年，第一台存储程序计算机在英国的剑桥和曼彻斯特开始运行。我的一个本科同学 Kenneth Ralston 是学数学的，他偶尔会和我如痴如醉地谈起一台神秘的机器，这台机器正在 MIT 附近的 Smart 街上的 Barta 楼内组装。我的好奇心后来在 1954 年的秋天得到了满足，那时我开始学习我的第一门计算机课程《数字计算机编码与逻辑》。那门课程是 Charles Adams 教的，他是自动编程（现在被称为编译）领域的先锋。当时使用的机器叫做旋风，被放置在一间充满了真空管电路的房间内。它由海军投资建立，用来研究飞机模拟。



因为我的知识背景及我所完成的电子工程专业的硕士课程，一个助研基金约请我在旋风计算机上用“最速下降法”解决一个最优化问题。这让我彻底熟悉了那一套繁琐的程序准备工作。我们以旋风机器的汇编语言编写程序，然后使用 Friden 电传打字机将以字符和数字表示的代码以打孔的方式输出到纸带上。纸带是用一个 Ferrante 光电读出器读入计算机的，然后交给“综合系统 2”的“系统软件”进行处理。处理结果是一个二进制纸带，以大约每秒钟 10 行的速度打孔出来，每行代表一个 6 位字符。而后，用户可以调用一个简单的装载程序将二进制的纸带装入到 2048 字的内存中，装载程序是保存在几个可以换来换去的内存单元中的。之后就是期待程序的正常运行。用户也可以在控制台的电传打字机上调用“综合系统”的输出例程来把结果打印出来，或者把它们写到一个原始的磁带单元中，留待以后离线打印。

那时最漂亮的输出设备是 CRT 显示屏，用户可以在上面一个点一个点地画出图表和图片。上面装备了一个照相机，可以把显示的图片录制在胶片上。系统程序员们已经开发好了“崩溃照相”功能，可以把程序出错时内存中的内容显示在 CRT 上。用户可以在第二天早上取到显影后的胶片，然后用一个缩微胶卷阅读器来研究上面的八进制数字。在那时，这是调试旋风程序的最主要方法，除此之外就是把中间结果打印出来。

大多数我们这样的普通用户不知道的是，在 Barta 楼里有一个后屋，在那里第一个基于计算机的飞机跟踪和威胁检测系统上的分类工作正在进行。那儿设置了一些更先进的设备，有很多台 PPI（计划和位置显示器）显示器；并且已经开发出了第一个定点设备——光笔，用来跟计算机实时交互。

旋风计算机最初的主内存是威廉斯管型的，这还不足以满足实时操作的可靠性标准。这一需求带动了相关研究工作并导致了磁芯内存的产生。旋风工程师建造了一个非常简单的计算机，称作内存测试机（MTC），用来测试新的内存。新内存表现良好，所以被立刻安装在旋风计算机上，而后 MTC 也就功成身退了。

旋风计算机上的工作导致了 MIT 林肯实验室的形成，实验室的主要责任是基于旋风计算机上的实时系统技术开发一个国家空中防御系统。同时，林肯实验室也进行计算机技术的研究，并建立了两台使用新的晶体管技术的机器，TX-0 和 TX-2。之所以编号都是偶数，是因为奇数在英文中同时有古怪的意思，主管设计者之一 Wesley Clark 曾经说：“林肯不做奇数的（古怪的）计算机”。TX-0 和 TX-2 的关系类似于 MTC 和旋风的关系：TX-0 被用作测试非常大的（按当时标准）内存，然后这些内存再被用于功能更强大的 TX-2。这些新机器继承了旋风系统中使用 CRT 显示屏和发光笔这些与用户实时交互的能力，同时也保留了使用纸带作为程序的主要介质。

在开发 TX-0 的同时，一台 IBM 704 机器在 MIT 安装起来。它用来补充、并最终接替了旋风作为 MIT 一般用户的主计算机。当 Lincoln 实验室不再需要 TX-0 后，MIT 电子工程系长期租用了它。MIT 的人们，特别是电子研究实验室，都为拥有了一台计算机而大喜过望，因为从此研究人员们便可以自由使用并亲手操作这台计算机，这要比 704 采用的批处理方式方便得多。

我于 1958 年 8 月完成了我的博士论文，成为一个四处寻找机遇的学校教员。我的新办公室在康普顿实验室楼（26 号楼）的第二层。有一天那里发生的事情引起了我的注意，人们正在一块宽广的区域安装一台 TX-0，它的位置就在 IBM 704 设施的正上方。

与 TX-0 一起到来的软件工具只有两个，一个是简单的汇编器程序，另一个是“UT-3”（3 号工具纸带）。两个程序都是二进制打孔纸带的形式，没有源代码。因为他们是以八进制代码手工输入的。UT-3 通过一个控制台打字机与用户交互（这里仍然是一个电传打字机，它包含了普通打字机的功能，可以被用户或被 TX-0 所驱动，将输入的字符传递到计算机或打印在纸上；这台打字机还带有一个机械纸带打孔器和阅读器，可以将字符打在纸带上或从纸带读字符到计算机上）。用户可以以八进制的方式输入数据到指定的内存位置，也可以要求打印指定内存位置或区域的内容。在 MIT，我们马上着手给这两个程序增加功能。汇编器最后演化为一个叫做 MACRO 的程序，除了有其他熟悉的汇编语言功能外，它还支持宏指令（宏功能是从 Doug McIlroy 在贝尔实验室的研究工作中得到启发的）。

有了汇编器后，就使得大范围重写和扩展 UT-3 成为可行。Tom Stockham 和我使新的程序支持符号，它可以使使用汇编器生成的符号表。我们把这个程序称作 FLIT（电

传打字机询问纸带), 这个名字仿用了当时一个很常用的杀虫喷雾剂的名字。(当 Grace Hopper 在哈佛的继电器计算机上工作时, 跟踪到一次故障是继电器触点上的一只飞蛾造成的, 从此人们开始把计算机的问题称作 Bug, 即“臭虫”) FLIT 的最重要功能是为调试程序(“除虫”)提供了断点设施。用户可以要求 FLIT 在被测试程序中向指定的指令位置插入最多四个断点。当被测试的程序遇到一个断点时, FLIT 会通知用户, 并且允许用户分析或修改内存的内容。分析结束后, 用户可以要求 FLIT 恢复程序继续执行, 就像没有中断过一样。FLIT 程序是后来的 DDT (另一种杀虫剂) 调试程序的典范, DDT 是 MIT 的学生为 DEC 公司生产的 PDP-1 计算机开发的。

FLIT (以及 TX-0) 的弱点之一是, 没有办法防止被测试程序向调试程序占用的内存里存储数据, 这会使调试程序停止工作。当我们给 DEC PDP-1 建立分时系统时, 我们做了特别的设计, 使得 DDT 与待测试的程序在各自的地址空间中执行, 但 DDT 仍可以观察和改变被测试程序中的信息, 我们把它称为“隐身调试器”。为了提供这种保护, 需要对 PDP-1 增加一些逻辑, 它们是随着为支持分时系统而作的更改和增加一起安装的。这个系统在 1963 年前后开始运行。

PDP-1 上的分时系统为 UC Berkely 在 SDS 940 上建立的分时系统提供了典范 (L. Perter Deutsch 兜里装着那个小操作系统从 MIT 到了 Berkely)。我相信 (虽然不很有把握), 隐身调试器的机制对于 DEC PDP-11/45 的设计产生了重要影响, 贝尔实验室就是为这个系统开发了 UNIX。

Jack B. Dennis

2008 年 4 月, Belmont, Massachusetts

联系博文视点

您可以通过如下方式与本书的出版方取得联系。

读者信箱: reader@broadview.com.cn

投稿信箱: bvtougao@gmail.com

北京博文视点资讯有限公司(武汉分部)

湖北省 武汉市 洪山区 吴家湾 邮科院路特1号 湖北信息产业科技大厦1402室

邮政编码: 430074

电话: (027) 87690813 传真: (027) 87690813 转 817

若您希望参加博文视点的有奖读者调查, 或对写作和翻译感兴趣, 欢迎您访问:

<http://bv.csdn.net>

关于本书的勘误、资源下载及博文视点的最新书讯, 欢迎您访问博文视点官方博客:

<http://blog.csdn.net/bvbook>

序言

2007 年，我和 Raymond（张银奎）第一次在上海见面。他对 Windows 操作系统的浓厚兴趣给我留下了深刻的印象，他的兴趣遍及有关 Windows 的所有细节，包括这个产品背后的人们及其演变过程。

Raymond 已经在软件开发岗位工作了十几年。现在他把多年的经验和对 Windows 操作系统的深刻理解结合起来，创作了这本关于调试的惊世之作。调试是计算机领域中最耗费时间和充满挑战的任务之一，也是许多软件工程师都需要提高的一个领域。

这本书所覆盖主题的广度是惊人的。从最低层硬件对调试的支持落笔，Raymond 带你遍历了系统中支持调试的所有层面——从用户态到内核态。此外，他还全面深入地介绍了编译器的调试支持、调试工具和各种基础设施。

据我多年在 VMS 操作系统开发团队工作的经验，我发现有些工程师掌握调试技术，而有些工程师并不具备这样的能力。利用可用工具插入合适的断点和分析追踪信息都需要很特殊的技巧。

细细品味这本书，会帮助你获得这些重要的软件开发技巧，增强你控制软件和编写代码的能力。

我多么希望这本书是用英文写的！

David Solomon
co-author, Windows Internals (Microsoft Press)
President, David Solomon Expert Seminars, Inc.

www.solsem.com

Preface

Raymond's intense interest in the details of the Windows operating system—including the people behind the product and its evolution—impressed me from our first meeting in Shanghai in 2007.

Raymond has been working as a software developer for more than 10 years. Now he's taken his years of software development experience and intimate knowledge of the Windows operating system and created this incredible book on debugging. Debugging is one of the most time-consuming and challenging tasks in computer world. It's an area to improve for a lot of software engineers.

The breadth of topics in this book is impressive. Starting from the lowest machine level hardware support for debuggers, Raymond takes you through the entire stack of debugging support - from user to kernel mode - as well as a comprehensive look at compiler's debug support, the debugging tools and infrastructure.

In my years working in the VMS operating system development group, there were those who could debug and those that couldn't. It takes a special skill to be able to leverage the available tools to insert the appropriate breakpoints and debug trace messages.

Digesting this book will help you gain these important software development skills, strengthen your capability to control software and write code.

I only wish it was in English!

David Solomon
co-author, Windows Internals (Microsoft Press)
President, David Solomon Expert Seminars, Inc.
www.solsem.com

专家寄语

Writing software is one thing. Being sure it works as intended is another. Raymond Zhang's new book on software debugging enables us to make that next step with confidence. It will prove invaluable to software engineers.

— Professor David J. Hand, Imperial College London

Raymond Zhang has written a thorough and comprehensive guide to software debugging, perhaps the most critical step in any successful software project. He demystifies the subject, moving from essential basics to advanced techniques. This book should be part of every working programmer's library.

— G. Pascal Zachary, author of "*Showstopper: Windows NT and the Next Generation at Microsoft*"

In my experience, Raymond Zhang is an extremely accomplished individual who has most graciously provided feedback to me on several of my books, occasionally pointing out instances where I was in error (also very graciously). I'm quite sure that his monumental new book on debugging techniques will prove of great value to the engineering community.

— Tom Shanley, President of Mindshare

Indeed, a debugger is an essential tool to master if you're going to do any sort of system programming.

— Matt Pietrek, *Under the Hood* columnist for *MSDN Magazine*

调试程序比编写程序更像一门艺术。程序员在调试程序时，想象力的基础是各种调试技术，张银奎先生的这本书系统地介绍了各个层次上的程序调试技术，我相信每一位阅读这本书的程序员都可以丰富自己的调试知识库，从而在实践中碰到程序问题时有更丰富的想象力，快速地“逮”到程序代码中的“臭虫（Bug）”。

— 潘爱民，研究员，微软亚洲研究院

感谢张银奎给 Syser Debugger 开发提供了指导性的意见。张先生这本调试巨著详细介绍了关于软件调试的方方面面，是目前为止软件调试方面的最权威著作之一。相信这本书一定能让各位读者在软件调试和开发方面受益匪浅。这本书应该成为每个软件开发人员的必备宝典。

— 吴岩峰、陈俊豪，Syser 调试器设计者

调试技术是成为高素质软件开发人员必备的一项关键技术，可惜在中国技术界却没有得到应有的重视。本书秉承了 Raymond 一贯的技术传播特点与风格：循循善诱，深入底层，切中肯綮，酣畅淋漓。相信本书会成为国内调试技术领域的扛鼎之作，每一位严肃程序员之案头必备。

— 李建忠，IT 技术作译者，祝成科技培训讲师

要在速成的年代里，转向对艰辛的思考和品位的召唤，并不容易。

——文怀沙

前言

现代计算机是从 20 世纪 40 年代开始出现的。当时的计算机比今天的要庞大很多，很多部件也不一样，但是有一点是完全相同的，那就是靠执行指令而工作。

一台计算机认识的所有指令被称为它的指令集（Instruction Set）。按照一定格式编写的指令序列被称为程序（Program）。在同一台计算机上，执行不同的程序，便可以完成不同的任务，因此，现代计算机在诞生之初常被冠以“通用”字样，以突出其通用性。在获得通用性带来好处的同时，人们也很快意识到了两个严峻的问题：首先是编写程序需要很多时间；其次是程序在执行时很可能出现意料之外的怪异行为。

程序对计算机的重要性和编写程序的复杂性让一些人看到了商机。大约在 20 世纪 50 年代中期，专门编写程序的公司出现了。几年后，模仿硬件（Hardware）一词，人们开始使用软件（Software）这个词来称呼计算机程序和它的文档，并把将用户需求转化为软件产品的整个过程称为软件开发（Software Development），将大规模生产软件产品的社会活动称为软件工程（Software Engineering）。

如今，几十年过去了，我们看到的是一个繁荣而庞大的软件产业。但是前面描述的两个问题依然存在：一是编写程序仍然需要很多时间；二是编写出的程序在运行时仍然会出现意料外的行为。而且后一个问题的表现形式越来越多，可能突然报告一个错误，可能给出一个看似正确却并非需要的结果，可能自作聪明地自动执行一大堆无法取消的操作，可能忽略用户的命令，可能长时间没有反应，可能直接崩溃或者永远僵死在那里……而且总是可能有无法预料的其他意外情况出现。这些“可能”大多是因为隐藏在软件中的设计失误而导致的，即所谓的软件臭虫（bug），或者称软件缺陷（defect）。

计算机是在软件指令的控制下工作的，让存在缺陷的软件控制硬件是件危险的事，可能导致惊人的损失和灾难。2003 年 8 月 14 日发生的北美大停电（Northeast Blackout of 2003）使 50 万人受到影响，直接经济损失 60 亿美元，其主要原因是软件缺陷导致报警系统没有报警。1999 年 9 月 23 日，美国的火星气象探测船因为没有进入预定轨道而受到大气压力和摩擦被摧毁，其原因是不同模块使用的计算单位不同，使计算出的轨道数据出现严重错误。1990 年 1 月 15 日，AT&T 公司的 100 多台交换机崩溃并反复重新启动，导致 6 万用户在 9 个小时中无法使用长途电话，其原因是新使用的软件在接收到某一种消息后会导致系统崩溃，并把这种症状传染给与它相邻的系统。1962 年 7 月 22 日，水手一号太空船发射 293 秒后因为偏离轨道而被销毁，其原因也与软件错误有直接关系。类似的故事还有很多，尽管我们不希望它们发生。

一方面，软件缺陷难以避免；另一方面其危害又很大，这使得消除软件缺陷成为软件工程中的一项重要任务。消除软件缺陷的前提是要找到导致缺陷的根本原因。我们把探索软件缺陷的根源并寻求其解决方案的过程称为软件调试（Software Debugging）。

本书的写作目的

在复杂的计算机系统中寻找软件缺陷的根源不是一个简单的任务，需要对软件和计算机系统有深刻的理解，选用科学的方法，并使用强有力的工具。这些正是作者写作本书的初衷。具体来说，写作本书有三个主要目的。

第一，论述软件调试的一般原理，包括CPU、操作系统和编译器是如何支持软件调试的，内核态调试和用户态调试的工作模型，以及调试器的工作原理。软件调试是计算机系统中多个部件之间的一个复杂交互过程，要理解这个过程，必须要了解每个部件在其中的角色和职责，以及它们的协作方式。学习软件调试原理不仅对提高软件工程师的调试技能至关重要，而且有利于提高它们对计算机系统的理解，将计算机原理、编译原理、操作系统等多个学科的知识融会贯通在一起。

第二，探讨可调试性（Debuggability）的内涵和实现软件可调试性的原则和方法。所谓软件的可调试性就是在软件内部加入支持调试的代码，使其具有自动记录、报告和诊断的能力，从而更容易被调试。软件自身的可调试性对于提高调试效率、增强软件的可维护性，以及保证软件的如期交付都有着重要意义。

第三，交流软件调试的方法和技巧。尽管论述一般原理是本书的重点，本书仍穿插了许多实践性很强的内容。包括调试用户态程序和系统内核模块的基本方法，如何诊断系统崩溃（BSOD）和应用程序崩溃，如何调试缓冲区溢出等与栈有关的问题，如何调试内存泄漏等与堆有关的问题。特别是，本书非常全面地介绍了WinDBG调试器的使用方法，给出了大量使用这个调试器的实例。

总之，笔者希望通过本书让读者懂得软件调试的原理，意识到软件可调试性的重大意义，学会使用基本的软件调试方法和调试工具，并能应用这些方法和工具解决问题和掌握更多软硬件知识。

本书的读者

首先，本书是写给所有程序员的。程序员是软件开发的核心力量。他们花大量的时间来调试他们所编写的代码，有时为此工作到深夜。笔者希望程序员朋友们读过本书后能提高调试能力，并自觉地在代码中加入调试支持，使调试效率大大提高，减少因为调试程序而加班的次数。本书中关于CPU、中断、异常和操作系统的介绍，是很多程序员需要补充的知识，因为对硬件和系统底层的深刻理解有利于写出更好的应用

目录一览

第1篇 绪论	1
第1章 软件调试基础	3
第2篇 CPU的调试支持	27
第2章 CPU基础	29
第3章 中断和异常	65
第4章 断点和单步执行	75
第5章 分支记录和性能监视	107
第6章 机器检查架构（MCA）	133
第7章 JTAG 调试	147
第3篇 操作系统的调试支持	163
第8章 Windows概要	165
第9章 用户态调试模型	193
第10章 用户态调试过程	227
第11章 中断和异常管理	273
第12章 未处理异常和JIT调试	309
第13章 硬错误和蓝屏	359
第14章 错误报告	391
第15章 日志	405
第16章 事件追踪	421
第17章 WHEA	445

第 18 章 内核调试引擎	461
第 19 章 Windows 的验证机制	513
第 4 篇 编译期的调试支持	539
第 20 章 编译和编译期检查	541
第 21 章 运行库和运行期检查	559
第 22 章 栈和函数调用	581
第 23 章 堆和堆检查	643
第 24 章 异常处理代码的编译	711
第 25 章 调试符号	739
第 5 篇 可调试性	781
第 26 章 可调试性概览	783
第 27 章 可调试性的实现	801
第 6 篇 调试器	833
第 28 章 调试器概览	835
第 29 章 WinDBG 及其实现	867
第 30 章 WinDBG 用法详解	905
附录 A 示例程序列表	999
附录 B WinDBG 标准命令列表	1001
索引	1003

目 录

第1篇 绪论	1
第1章 软件调试基础	3
1.1 简介	3
1.2 基本特征	6
1.3 简要历史	8
1.4 分类	12
1.5 调试技术概览	15
1.6 错误与缺欠	20
1.7 与软件工程的关系	24
1.8 本章总结	26
第2篇 CPU的调试支持	27
第2章 CPU基础	29
2.1 指令和指令集	29
2.2 IA-32处理器	32
2.3 CPU的操作模式	38
2.4 寄存器	40
2.5 理解保护模式	46
2.6 段机制	50
2.7 分页机制（Paging）	55
2.8 系统概貌	62
2.9 本章总结	64
第3章 中断和异常	65
3.1 概念和差异	65
3.2 异常的分类	67
3.3 异常例析	69
3.4 中断/异常优先级	72
3.5 中断/异常处理	73
3.6 本章总结	74
第4章 断点和单步执行	75
4.1 软件断点	75

4.2 硬件断点	83
4.3 陷阱标志	95
4.4 实模式调试器例析	100
4.5 本章总结	105
第5章 分支记录和性能监视	107
5.1 分支监视概览	107
5.2 使用寄存器的分支记录	108
5.3 使用内存的分支记录	113
5.4 DS示例：CpuWhere	117
5.5 性能监视	123
5.6 本章总结	132
第6章 机器检查架构（MCA）	133
6.1 奔腾处理器的机器检查机制	134
6.2 MCA	135
6.3 编写 MCA 软件	141
6.4 本章总结	145
第7章 JTAG 调试	147
7.1 简介	147
7.2 JTAG 原理	149
7.3 JTAG 应用	154
7.4 IA-32 处理器的 JTAG 支持	156
7.5 本章总结	161
第3篇 操作系统的调试支持	163
第8章 Windows 概要	165
8.1 简介	165
8.2 进程和进程空间	167
8.3 内核模式和用户模式	176
8.4 架构和系统部件	184
8.5 本章总结	192
第9章 用户态调试模型	193
9.1 概览	193
9.2 采集调试消息	196
9.3 发送调试消息	200
9.4 调试子系统服务器（XP之后）	203
9.5 调试子系统服务器（XP之前）	210
9.6 比较两种模型	219

9.7 NTDLL 中的调试支持例程	221
9.8 调试 API	224
9.9 本章总结	226
第 10 章 用户态调试过程	227
10.1 调试器进程	227
10.2 被调试进程	231
10.3 从调试器中启动被调试程序	234
10.4 附加到已经启动的进程	240
10.5 处理调试事件	243
10.6 中断到调试器	251
10.7 输出调试字符串	259
10.8 终止调试会话	266
10.9 本章总结	271
第 11 章 中断和异常管理	273
11.1 中断描述符表	273
11.2 异常的描述和登记	280
11.3 异常分发过程	284
11.4 结构化异常处理 (SEH)	290
11.5 向量化异常处理 (VEH)	302
11.6 本章总结	308
第 12 章 未处理异常和 JIT 调试	309
12.1 简介	309
12.2 默认的异常处理器	311
12.3 未处理异常过滤函数	318
12.4 应用程序错误对话框	328
12.5 JIT 调试和 Dr. Watson	334
12.6 顶层异常过滤函数	340
12.7 Dr. Watson	343
12.8 DRWTSN32 的日志文件	347
12.9 用户态转储文件	351
12.10 本章总结	357
第 13 章 硬错误和蓝屏	359
13.1 硬错误提示	359
13.2 蓝屏终止 (BSOD)	366
13.3 系统转储文件	371
13.4 分析系统转储文件	374
13.5 辅助的错误提示方法	380

13.6 配置错误提示机制	384
13.7 防止滥用错误提示机制	389
13.8 本章总结	390
第 14 章 错误报告.....	391
14.1 WER 1.0	392
14.2 系统错误报告	395
14.3 WER 服务器端	397
14.4 WER 2.0	399
14.5 CER	403
14.6 本章总结	404
第 15 章 日志.....	405
15.1 日志简介	405
15.2 ELF 的架构	406
15.3 ELF 的数据组织	409
15.4 察看和使用 ELF 日志	413
15.5 CLFS 的组成和原理.....	414
15.6 CLFS 的使用方法.....	416
15.7 本章总结	420
第 16 章 事件追踪.....	421
16.1 简介	421
16.2 ETW 的架构.....	422
16.3 提供 ETW 消息.....	424
16.4 控制 ETW 会话.....	425
16.5 消耗 ETW 消息.....	427
16.6 格式描述	428
16.7 NT Kernel Logger.....	432
16.8 Global Logger Session.....	436
16.9 Crimson API	440
16.10 本章总结	443
第 17 章 WHEA	445
17.1 目标和架构	445
17.2 错误源	450
17.3 错误处理过程	452
17.4 错误持久化	457
17.5 注入错误	459
17.6 本章总结	459

第 18 章 内核调试引擎	461
18.1 概览	462
18.2 连接	465
18.3 启用	475
18.4 初始化	478
18.5 内核调试协议	483
18.6 与内核交互	492
18.7 建立和维持连接	502
18.8 本地内核调试	509
18.9 本章总结	511
第 19 章 Windows 的验证机制	513
19.1 简介	514
19.2 驱动验证器的工作原理	515
19.3 使用驱动验证器	521
19.4 应用程序验证器的工作原理	526
19.5 使用应用程序验证器	533
19.6 本章总结	537
第 4 篇 编译器的调试支撑	539
第 20 章 编译和编译期检查	541
20.1 程序的构建过程	541
20.2 编译	543
20.3 Visual C++ 编译器	544
20.4 编译错误和警告	549
20.5 编译期检查	551
20.6 标准标注语言	555
20.7 本章总结	558
第 21 章 运行库和运行期检查	559
21.1 C/C++ 运行库	559
21.2 链接运行库	562
21.3 运行库的初始化和清理	565
21.4 运行期检查	569
21.5 报告运行期检查错误	574
21.6 本章总结	580
第 22 章 栈和函数调用	581
22.1 简介	581
22.2 栈的创建过程	585

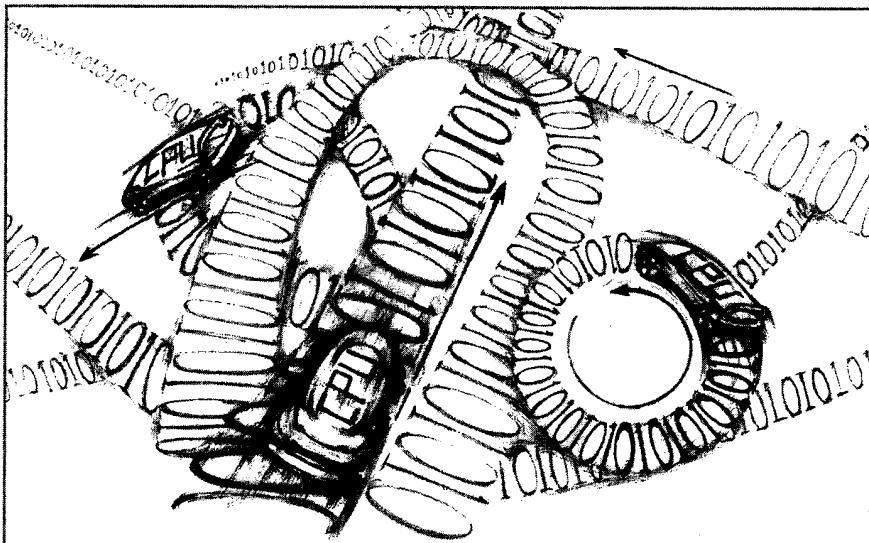
22.3 CALL 和 RET 指令	590
22.4 局部变量和栈帧	595
22.5 帧指针省略 (FPO)	604
22.6 栈指针检查	606
22.7 调用协定	609
22.8 栈空间的增长和溢出	616
22.9 栈下溢	623
22.10 缓冲区溢出	624
22.11 变量检查	628
22.12 基于 Cookie 的安全检查	636
22.13 本章总结	642
第 23 章 堆和堆检查	643
23.1 理解堆	644
23.2 堆的创建和销毁	646
23.3 分配和释放堆块	649
23.4 堆的内部结构	654
23.5 低碎片堆 (LFH)	661
23.6 堆的调试支持	662
23.7 栈回溯数据库	666
23.8 堆溢出和检测	670
23.9 页堆	677
23.10 准页堆	683
23.11 CRT 堆	688
23.12 CRT 堆的调试堆块	692
23.13 CRT 堆的调试功能	698
23.14 堆块转储	700
23.15 泄漏转储	704
23.16 本章总结	709
第 24 章 异常处理代码的编译	711
24.1 概览	711
24.2 FS:[0]链条	713
24.3 遍历 FS:[0]链条	716
24.4 执行异常处理函数	721
24.5 __try{}__except()结构	724
24.6 安全问题	732
24.7 本章总结	737

第 25 章 调试符号	739
25.1 名称修饰	739
25.2 调试信息的存储格式	742
25.3 目标文件中的调试信息	745
25.4 PE 文件中的调试信息	753
25.5 DBG 文件	762
25.6 PDB 文件	764
25.7 有关的编译和链接选项	771
25.8 PDB 文件中的数据表	775
25.9 本章总结	780
第 5 篇 可调试性	781
第 26 章 可调试性概览	783
26.1 简介	783
26.2 Showstopper 和未雨绸缪	784
26.3 基本原则	787
26.4 不可调试代码	792
26.5 可调试性例析	794
26.6 与安全、性能和商业秘密的关系	798
26.7 本章总结	799
第 27 章 可调试性的实现	801
27.1 角色和职责	801
27.2 可调试架构	804
27.3 通过栈回溯实现可追溯性	808
27.4 数据的可追溯性	815
27.5 可观察性的实现	821
27.6 自检和自动报告	830
27.7 本章总结	832
第 6 篇 调试器	833
第 28 章 调试器概览	835
28.1 TX-0 计算机和 FLIT 调试器	835
28.2 小型机和 DDT 调试器	837
28.3 个人计算机和它的调试器	841
28.4 调试器的功能	845
28.5 分类标准	852
28.6 实现模型	853
28.7 经典架构	859

28.8 HPD 标准	862
28.9 本章总结	866
第 29 章 WinDBG 及其实现	867
29.1 WinDBG 溯源	867
29.2 C 阶段的架构	872
29.3 重构	875
29.4 调试器引擎的架构	881
29.5 调试目标	887
29.6 调试会话	892
29.7 接收和处理命令	899
29.8 本章总结	904
第 30 章 WinDBG 用法详解	905
30.1 工作空间	905
30.2 命令概览	908
30.3 用户界面	911
30.4 输入和执行命令	916
30.5 建立调试会话	923
30.6 终止调试会话	927
30.7 理解上下文	930
30.8 调试符号	933
30.9 事件处理	944
30.10 控制调试目标	951
30.11 单步执行	955
30.12 使用断点	962
30.13 控制进程和线程	969
30.14 观察栈	973
30.15 分析内存	978
30.16 遍历链表	987
30.17 调用目标程序的函数	992
30.18 命令程序	994
30.19 本章总结	997
附录 A 示例程序列表	999
附录 B WinDBG 标准命令列表	1001
索引	1003

第1篇

绪论



1955年，一个名叫 Computer Usage Corporation (CUC) 的公司诞生了，它是世界上第一个专门从事软件开发和服务的公司。CUC 公司的创始人是 Elmer Kubie 和 John W. Sheldon，他们都在 IBM 工作过。从当时计算机硬件的迅速发展中，他们看到了软件方面所潜在的机遇。CUC 的诞生标志着一个新兴的产业正式起步了。

与其他产业相比，软件产业的发展速度是惊人的。短短 50 几年后，我们已经难以统计世界上共有多少个软件公司，只知道一定是一个很庞大的数字，而且这个数量还在不断增大。同时，软件产品的数量也达到了难以统计的程度，各种各样的软件已经渗透到人类生产和生活的各个领域，越来越多的人开始依赖软件工作和生活。

与传统的产品相比，软件产品具有根本的不同，其生产过程也有着根本的差异。在开发软件的整个过程中，存在非常多的不确定性因素。在一个软件真正完成之前，

很难预计它的完成日期。很多软件项目都经历了多次的延期，还有很多中途夭折。直到今天，人们还没有找到一种有效的方法来控制软件的生产过程。导致软件生产难以控制的根本原因是来源于软件本身的复杂性。一个软件的规模越大，它的复杂度也越高。

简单来说，软件是程序（program）和文档（document）的集合，程序的核心内容便是按一定顺序排列的一系列指令（instruction）。如果把每个指令看作一块积木，那么软件开发就是使用这些积木修建一个让 CPU（中央处理器）在其中运行的交通系统。这个系统中有很多条不同特征的道路（函数）。有些道路只允许一辆车在上面行驶，一辆车驶出后另一辆才能进入，有些道路可以让无数辆车同时在上面飞奔。这些道路都是单行道，只可以沿一个方向行驶。在这些道路之间，除了明确的入口（entry）和出口（exit）之外，还可以通过中断和异常等机制从一条路飞越到另一条，另一条又可以飞转到第三条或直接飞回到第一条。在这个系统中行驶的车辆也很特殊，它们速度很快，而且“无人驾驶”，完全不知道会跑到哪里，唯一的原则就是上了一条路便沿着它向前跑……

如果说软件的执行过程就好像是 CPU 在无数条道路（指令流）间飞奔，那么开发软件的过程就是设计和构建这个交通网络的过程。其基本目标是要让 CPU 在这个网络中奔跑时可以完成需求（requirement）中所定义的功能。对这个网络的其他要求通常还有可靠（reliable）、灵活（flexible）、健壮（robust）、易于维护（maintainable），可以简单地改造就能让其他类型的车辆（CPU）在上面行驶（portable）……

开发一个满足以上要求的软件系统不是一件简单的事，通常需要经历分析（analysis）、设计（design）、编码（code）和测试（test）等多个环节。通过测试并发布（release）后，还需要维护（maintain）和支持（support）工作。在以上环节中，每一步都可能遇到这样那样的技术难题。

在软件世界中，螺丝刀、万用表等传统的探测和修理工具都不再适用了，取而代之的是以调试器为核心的各种软件调试（Software Debugging）工具。

软件调试的基本手段有断点、单步执行、栈回溯等，其初衷就是跟踪和记录 CPU 执行软件的过程，把动态的瞬间凝固下来供检查和分析。

软件调试的基本目标是定位软件中存在的设计错误（bug）。但除此之外，软件调试技术和工具还有很多其他用途。比如，分析软件的工作原理，分析系统崩溃，辅助解决系统和硬件问题等。

概而言之，软件是通过指令的组合来指挥硬件，既简单，又复杂，充满神秘与挑战。而软件调试是帮助人们探索和征服这个神秘世界的有力工具。

软件调试基础

著名的计算机科学家 Brian Kernighan 曾经说过，软件调试要比编写代码困难一倍，如果你发挥出了最佳才智编写代码，那么你的智商便不足以调试这个代码。

另一方面，软件调试是软件开发和维护中非常频繁的一项任务，几乎在软件生命周期的每个阶段，都有很多这样那样的问题需要进行调试。

一方面是难度很高，另一方面是任务很多。因此，在一个典型的软件团队中，花费在软件调试上的人力和时间通常是很可观的。据不完全统计，一半以上的软件工程师把一半以上的时间用在软件调试上。很多时候，调试一个软件问题可能就需要几天乃至几周的时间。从这个角度来看，提高软件工程师的调试效率对于提高软件团队的工作效率有着重要意义。

本书旨在从多个角度和多个层次解析软件调试的原理、方法和技巧。在分别深入介绍这些内容之前，本章将做一个概括性的介绍，使读者了解一个简单的全貌，为阅读后面的章节做准备。

1.1 简介

这一节我们首先给出软件调试的解释性定义，而后介绍软件调试的基本过程。

1.1.1 定义

首先，什么是软件调试？我们不妨从英文的原词 `software debug` 说起，`debug` 是在 `bug` 一词前面加上词头 `de`，意思是分离和去除 `bug`。

`Bug` 的本意就是昆虫，但至少早在 19 世纪时，人们就开始使用这个词来描述电子设备中的设计缺欠，著名发明家托马斯·阿尔瓦·爱迪生（1847-2-11—1931-10-18）就使用这个词来描述电路方面的设计错误。

关于 `Bug` 一词在计算机方面的应用流传着一个有趣的故事。时间是在 20 世纪 40

年代，当时的电子计算机都还非常庞大，数量也非常少，主要用在军事方面。1944年制造完成的 Mark I, 1946 年 2 月开始运行的 ENIAC(Electronic Numerical Integrator And Computer) 和 1947 年完成的 Mark II 是其中赫赫有名的几台。Mark I 是由哈佛大学的 Howard Aiken 教授设计，IBM 公司制造的，Mark II 是由美国海军出资制造的。与使用电子管制造的 ENIAC 不同，Mark I 和 Mark II 主要是使用开关和继电器制造的。另外，Mark I 和 Mark II 都是从纸带或磁带上读取指令并执行的，因此，它们不属于从内存读取和执行指令的存储程序计算机 (stored-program computer)。

1947 年 9 月 9 日，当人们测试 Mark II 计算机时，它突然发生了故障。经过几个小时的检查后，工作人员发现一只飞蛾被打死在面板 F 的第 70 号继电器中。当把这个飞蛾取出后，机器便恢复了正常。当时为 Mark II 计算机工作的著名女科学家 Grace Hopper 将这只飞蛾粘贴到当天的工作手册中(见图 1-1)，并在上面加了一行注释，“First actual case of bug being found”，当时的时间是 15:45。随着这个故事的广为流传，越来越多的人开始使用 Bug 一词来指代计算机中的设计错误，并把 Grace Hopper 上登记的那只飞蛾看作是计算机历史上第一个被记录在文档 (documented) 中的 Bug。

92

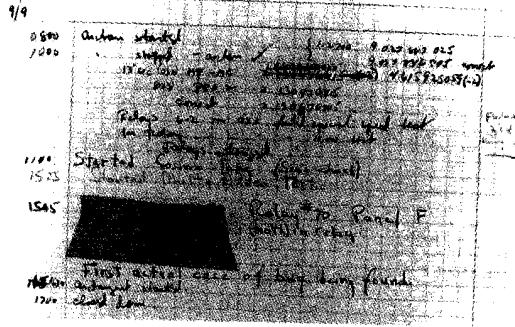


图 1-1 计算机历史上第一个被记录在文档中的 Bug

在 Bug 一词广泛使用后，人们自然地开始使用 debug 这个词来泛指排除错误的过程。关于谁最先创造和使用了这个词，目前还没有公认的说法，但是可以肯定的是，Grace Hopper 在 20 世纪 50 年代发表的很多论文中就已频繁使用这个词了。因此可以肯定地说，在 20 世纪 50 年代人们已经开始使用这个词来表达软件调试这一含义，而且一直延续到今天。

尽管从字面上看，debug 的直接意思就是去除 Bug，但它实际上包含了寻找和定位 Bug。因为去除 Bug 的前提就是要找到 Bug，如何找到 Bug 大都比发现后去除它要难得多。而且，随着计算机系统的发展，软件调试已经变得越来越不像在继电器间“捉虫”那样轻而易举了。因此，在台湾，人们把 software debug 翻译为软件侦错。这个翻译没有按照英文原词的字面含义直译，超越了原本的单指“去除”的境界，融入了侦查的含义，是个很不错的意译。

在大陆，通常将 software debug 翻译为软件调试，泛指重现软件故障（failure）、定位故障根源，并最终解决软件问题的过程。这种理解与英语文献中对 software debug 的深层解释也是一致的。如微软的计算机词典（Microsoft Computer Dictionary, Fifth Edition）对 debug 一词的解释是：

debug vb. To detect, locate, and correct logical or syntactical errors in a program or malfunctions in hardware.

对软件调试的另一种更通俗的解释是指使用调试工具求解各种软件问题的过程，例如跟踪软件的执行过程，探索软件本身或与其配套的其他软件，或者硬件系统的工作原理等，这些过程有可能是为了去除软件缺欠，也可能不是。

1.1.2 基本过程

尽管取出那只飞虫非常轻松，但是找到它还是耗费了几个小时的时间。因此，软件调试从一开始实际上就包含了定位错误和去除错误这两个基本步骤。进一步讲，一个完整的软件调试过程是图 1-2 所示的循环过程，它由以下几个步骤组成。

第一，重现故障，通常是在用于调试的系统上重复导致故障的步骤，使要解决的问题出现在被调试的系统中。

第二，定位根源，即综合利用各种调试工具，使用各种调试手段寻找导致软件故障的根源（root cause）。通常测试人员报告和描述的是软件故障所表现出的外在症状，比如界面或执行结果中所表现出的异常；或者是与软件需求（requirement）和功能规约（function specification）不符的地方，即所谓的软件缺欠（defect）。而这些表面的缺欠总是由于一个或多个内在因素所导致的，这些内因要么是代码的行为错误，要么是不行为错误（该做而未做）。定位根源就是要找到导致外在缺欠的内因。

第三，探索和实现解决方案，即根据寻找到的故障根源、资源情况、紧迫程度等设计和实现解决方案。

第四，验证方案，在目标环境中测试方案的有效性，又称为回归（regress）测试。如果问题已经解决，那么就可以关闭问题。如果没有解决，则回到第 3 步调整和修改解决方案。

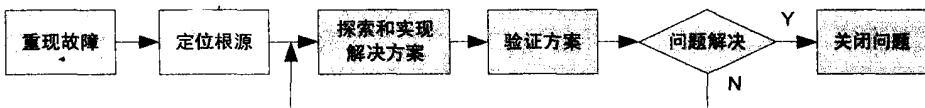


图 1-2 软件调试过程

在以上各步骤中，定位根源常常是最困难也是最关键的步骤，它是软件调试过程的核心和灵魂。如果没有找到故障根源，那么解决方案便很可能是隔靴搔痒，或者头痛医脚，有时似乎缓解了问题，但事实上没有彻底解决问题，甚至是白白浪费时间。

1.2 基本特征

上一节介绍了软件调试的定义和基本过程。本节将进一步介绍它的基本特征，我们将分3个方面来讨论。

1.2.1 难度大

诚如 Brian Kernighan 先生所说的，软件调试是一项复杂度高、难度大的任务。以下是导致这种复杂性的几个主要因素。

第一，如果把定位软件错误看作是一种特别的搜索问题，那么它通常是个很复杂的搜索问题。首先，被搜索的目标空间是软件问题所发生的系统，从所包含的信息量来看，这个空间通常是很庞大的，因为一个典型的计算机系统中包含着几十个硬件部件、数千个软件模块，每个模块又包含着以 KB 或 MB 为单位的大量指令（代码）。另一方面，这个搜索问题并没有明确的目标和关键字，通常只知道不是非常明确的外在症状，必须通过大量的分析，才能逐步接近真正的内在原因。

第二，为了探寻问题的根源，很多时候必须深入到被调试模块或系统的底层，研究内部的数据和代码。与顶层不同，底层的数据大多是以原始形态存在的，理解和分析的难度比顶层要大。举例来说，对于顶层看到的文字信息，在底层看到的可能只是这些文字的某种编码（ANSI 或 UNICODE 等）。对于代码而言，底层意味着低级语言或汇编语言，甚至机器码，因为当无法进行源代码级的调试时，我们不得不进行汇编一级的跟踪和分析。对于通信有关的问题，底层意味着需要观察原始的通信数据包和检查包的各个部分。另外，很多底层的数据和行为是没有文档的，不得不做大量的跟踪和分析才能摸索出一些线索和规律。从 API 的角度来看，底层意味着不仅要理解 API 的原型和使用方法，有时还必须知道它内部是如何实现的，执行了哪些操作，这一点也论证了 Brian Kernighan 所说的调试要比编写代码困难。

第三，因为要在较大的问题域内定位错误，所以要求调试者必须有丰富的知识，熟悉问题域内的各个软硬件模块，以及它们之间的协作方式。从纵向来看，要理解系统从最上层到最下层的各个层次。从横向来看，要理解每个层次内的各个模块。对于每个模块，不仅要知道其概况，有时还必须深刻理解其细节。举例来说，对于那些包含驱动程序的软件，有时必须同时进行用户态的调试和内核态调试，要求调试者对应用程序、操作系统和硬件都要有比较深刻的理解。

第四，每个软件调试任务都有很多特殊性，或者说很难找到两个调试任务是一样的。这意味着，当执行一个软件调试任务时，很难找到可以模仿或借鉴的先例，几乎每一步都必须靠自己的探索来完成。而编写代码和其他软件活动经常有示例代码或模板可以参考或套用。

第五，软件的大型化、层次的增多、多核和多处理器系统的普及都在增加软件调试的难度。

以上介绍的第一、第二个因素是软件调试所固有的，第三、第四个是可以随着软件技术的发展和人们对软件调试的重视程度不断提高而改善的。

1.2.2 难以估计完成时间

就像侦破一个案件所需的日期很难确定一样，对于一个软件错误，到底需要多久才能定位到它的根源并解决这个问题是一个很难回答的问题。这是因为软件调试问题的问题域比较大，调试过程中包含的随机性和不确定性很多，调试人员对问题及相关模块和系统的熟悉程度、对调试技术的熟练程度也会加入很多不确定性。

调试任务的难以预测性经常给软件工程带来重大的麻烦，其中最常见的便是导致项目延期。事实上，很多软件项目的延期是与无法定位和解决存留的 Bug 有关的。Showstopper（参考文献 2）一书生动地讲述了 Windows NT（3.1）内核开发中因为严重 Bug 而多次延期的故事（详见 26.2.1 节）。比 NT 3.1 还不幸的项目有很多，在它们被多次延期后，仍然有大量的问题无法解决，最后因为资金等问题不得不被取消和放弃。

在现实中，很多软件难题经常成为整个项目的瓶颈，是项目团队中所有人关注的焦点，包括市场部门和一些高级管理者。这时，对于接受调试任务的工程师来说，除了要面对技术上的难题外，还要承受很多其他方面的压力。这种压力有时会加快问题的解决，有时会使他们手忙脚乱而变得效率更低。

对于如何才能更好地预测软件调试任务的完成时间，目前还没有很有效的方法，为了降低风险，应该尽可能地让经验丰富的工程师来做预测，并综合考虑多个人的估计结果。

1.2.3 广泛的关联性

很多调试机制是操作系统、中央处理器和调试器相互协作的复杂过程，比如 Windows 本地调试中的软件断点功能通常是依赖于 CPU 的断点指令（对于 x86，即 INT 3）的，CPU 执行到断点指令时中断下来，并以异常的方式报告给操作系统，操作系统再将这个事件分发给调试器。

另外，软件调试与软件编译有着密切的关系。软件的调试版本包含了很多用来辅助软件调试的信息，具有更好的可调试性。调试信息中很重要的一个部分便是调试符号，它是进行源代码级调试所必需的。

综上所述，软件调试与计算机系统的硬件核心（CPU）和软件核心（操作系统）都有着很紧密的耦合关系，与软件生产的最主要机器——编译器也息息相关。因此，可以说软件调试具有广泛的关联性，有时也被称为系统性。

软件调试的广泛关联性增加了理解软件调试过程的难度，同时也导致了软件调试技术难以在短时间内迅速发展和升级。因为要开发一种新的调试手段，通常需要硬件、操作系统和工具软件三个环节的支持，要涉及很多个厂商或组织。这也是软件调试技术滞后于其他技术的一个原因。一般来说，对于一种新出现的软硬件技术，对应的有效软件调试技术要滞后一段时间才出现。

从学习的角度来看，软件调试的广泛关联性使其成为让学习者达到融会贯通境界的一种绝好途径。在一个人对 CPU、操作系统、编译器、编程语言等都基本掌握后，可以通过学习软件调试技术和实践来加深对这些知识的理解，并把它们联系起来。

1.3 简要历史

当计算机领域的拓荒者们设计最初的计算机系统时，他们就考虑到了调试问题，包括如何调试系统中的硬件，也包括如何调试系统中的软件。现代计算机（Modern Computers）是从 20 世纪 40 年代开始出现并迅速发展起来的，经历了从大型机到小型机再到微型计算机的几个主要阶段。

关于早期大型机和小型机的原始文档已经成为珍贵的历史资料了，大多被收藏在博物馆中。但幸运的是，在笔者收集到的关于早期计算机的有限资料中，几乎每一本都包含了关于调试的内容。这不仅是因为运气，更是因为当时人们就非常重视调试。

本节将以大型机、小型机和微型机三个阶段中有代表性的计算机系统为例，介绍它们实现调试功能的方式，目的是了解典型软件调试功能的演进过程，勾勒出软件调试的简要发展历史。

1.3.1 单步执行

UNIVAC I (UNIVersal Automatic Computer I) 是世界上最早大规模生产的商用现代计算机，之前的计算机都是只生产一台而用于军事和学术领域。从 1951 年开始，共有 46 台 UNIVAC I 销售给不同的公司和组织，每台的售价都高于 100 万美元，其中一些一直工作到 1970 年。1952 年哥伦比亚广播公司租用 UNIVAC I 准确预测出了当年美国总统的大选结果，这不仅使 UNIVAC 声名大振，也使人们对计算机的功能有了新的认识。

与需要一个楼面来安放的 ENIAC 相比，UNIVAC I 已经小了很多，但整个系统仍然需要一个 30 多平方米的房间才能放得下。典型的 UNIVAC I 系统由主机 (Central Computer)、磁带驱动器 (名为 UNISERVO，最多可配置 10 台)、打印机 (Uniprinter)、打字机 (Typewriter)、监视控制台 (Supervisory Control) 和用于维护的示波器所组成。

在写字台大小的 UNIVAC I 监视控制台上有很多指示灯和开关。其中有一个名为 Interrupted Operation Switch (IOS) 的开关 (见图 1-3) 与软件调试有着密切的关系。

IOS 开关共有中间和上下左右五个位置，分别代表五种运行模式。中间位置代表正常模式，在此模式下计算机机会连续执行内存中的程序指令，因此这个模式又称为连续（Continuous）模式。其他四个位置代表不同作用的“单步”模式，分别为 ONE OPERATION（上）、ONE INSTRUCTION（下）、ONE STEP（左）和 ONE ADDITION（右），即一次执行一个操作、一条指令、一步和一次加法运算。



图 1-3 UNIVAC I 监视控制台上的 IOS 开关

当 IOS 开关位于四种单步模式之一时，CPU 执行完一条指令或一个操作后便会停下来，让用户检查当前的寄存器和内存状态。在检查后，只要按键盘（监视控制台的一部分）上的开始键（SATRT BAR）便可以让系统继续执行。

UNIVAC I 的操作手册详细介绍了 IOS 开关的使用方法、如何使用不同的模式来启动和调试程序，以及诊断软硬件问题。

笔者不能确认在 UNIVAC I 之前的计算机是否已经使用了类似 IOS 这样的硬件开关来控制程序单步执行。但是可以说这是比较早的单步执行方式。而且这种方式一直延续到小型机时代。在图 1-4 所示的著名小型机 PDP-1 的控制面板照片上，可以看到其右上角的三个开关中，中间一个便是 SINGLE STEP（单步），下面的是 SINGLE INST（Single Instruction）（单指令）。

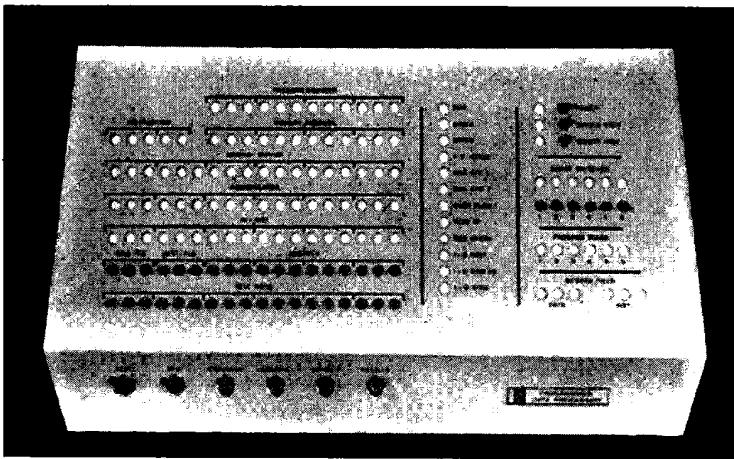


图 1-4 PDP-1 的控制面板

1971 年，Intel 成功推出了世界上第一款微处理器 4004，标志着计算机开始向微型化

方向发展。1978年,x86 CPU 的第一代 8086 CPU 问世,在其标志寄存器(FLAGS)中(见图 1-5),专门设计了一个用于软件调试的标志位,叫做 TF(Trace Flag)。

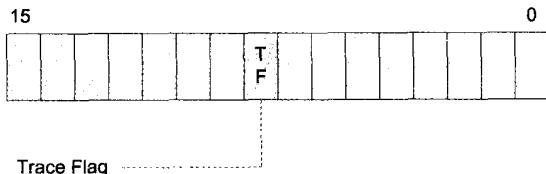


图 1-5 8086 CPU 的标志寄存器(FLAGS)

TF 位主要是供调试器软件来使用的,当用户需要单步跟踪时,调试器会设置 TF 位,当 CPU 执行完一条指令后会检查 TF 位,如果这个位为 1,那么便会产生一个调试异常(INT 1),目的是停止执行当前的程序,中断到调试器中。

从上面的介绍中,我们看到了单步执行功能从专门的硬件开关向寄存器中的一个标志位演进的过程。这种变化趋势是与计算机软硬件的总体发展相适应的。因为,在 UNIVAC I 时代,还没有完善的软件环境和调试器软件,因此,使用一个专门的硬件开关是一种很合理有效的方案。在微处理器出现的时代,软件已经大大发展起来,操作系统和调试器都已经比较成熟,因此,使用寄存器的一个标志位来代替专门的硬件也变得水到渠成,因为这样不仅简化了硬件设计、降低了成本,而且适合让调试器软件以程序方式控制。

1.3.2 断点指令

在 UNIVAC I 的 43 条指令中,有一条使用逗号(,)表示的指令,是专门用来支持断点功能的,称为逗号断点(Comma Breakpoint)指令。同时,在 UNIVAC I 的监视控制台上有一个名为逗号断点的两态开关(Comma Breakpoint Switch),如果按下这个开关,那么当计算机执行到逗号断点指令时就会停下来,让用户检查程序状态,进行调试。如果没有按下开关,那么计算机会将其视作跳过(Skip)指令,不做任何操作,执行后面的指令。

除了逗号断点指令,UNIVAC 的打印指令 50 m(m 为内存地址)也可以产生断点效果。它是与监视控制台上的输出断点开关(Output Breakpoint Switch)配合工作的。这个开关有三个状态(位置):正常(Normal)、跳过(Skip)和断点(Breakpoint)。如果这个开关在正常位置,那么执行 50 m 指令输出内存地址 m 的内容;如果开关在跳过位置,那么这条指令会被忽略;如果 m 在断点位置,那么执行到这里时计算机会中断下来。可见这条指令不仅实现了一种可随时开启关闭的监视点功能,而且还可以根据需要停在监视点位置,这时又相当于一种外部可控的断点。

总之,UNIVAC I 提供了两种断点指令和与指令协同工作的硬件开关,实现了一种主要靠硬件来工作的非常简朴的断点功能。这种实现方式不需要软件调试器来参与,

也没有为实现软件调试器提供足够支持。

下面我们再来看小型机 PDP-1 上是如何提供断点支持的。概而言之，PDP-1 提供了一条名为 jda 的指令，供调试器开发者来实现断点功能。这条指令的语法是：

```
jda Y
```

它执行的操作是将 AC (Accumulator) 寄存器的内容存入地址 Y，然后把程序计数器 (Program Counter，相当于 IP) 的值放入 AC 寄存器，并跳转到 Y+1。利用这条指令，调试器可以这样实现断点功能。

当向某一地址设置断点时，将这一地址及它的值都保存起来，并将这一地址处的内容替换成一条 jda 指令，指令的操作符 Y 是仔细设计好的，指向调试器的数据和代码。

当程序执行到断点位置时，系统会执行位于那里的 jda 指令，跳转到调试器的代码。调试器根据 AC 寄存器的内容知道这个断点的发生位置，找到它所对应的断点记录，然后保存寄存器的内容（上下文），并打印出存储在位置 Y 的 AC 寄存器内容给调试者，调试者可以输入内存观察命令或执行其他调试功能，待调试结束后，输入某一个命令恢复执行。这时调试器需要恢复寄存器的值，将保存的指令恢复回去，然后跳转回去继续执行。

在 x86 系列 CPU 中，提供了一条使用异常机制的断点指令，即 INT 3，供调试器来设置断点，当 CPU 执行到这里时，会产生一个异常，跳转到异常处理例程，然后中断到调试器中。我们将在以后的章节中详细介绍其细节。

1.3.3 分支监视

程序中的分支和跳转指令对于软件的执行流程和执行结果起着关键作用，不恰当的跳转往往是很多软件问题的错误根源。有时跟踪一个程序，是为了检查它的跳转时机和跳转方向。因此，监视和报告程序的分支位置和当时的状态对软件调试是很有意义的。

UNIVAC I 的条件转移断点 (Conditional Transfer Breakpoint) 功能正是针对这一需求而设计的。同样，这一机制由两个部分组成：一个部分是条件转移指令 Qn m 和 Tn m；另一部分是监视控制台上的按钮和指示灯。指令中的 m 是跳转的目标地址，n 是 0 到 9 的十个值之一，与控制台上的 0~9 十组按键（称为条件转移断点选择按钮）和指示灯相对应。图 1-6 是控制面板的相关部分，下面一排共有 12 个按钮，上面一排为指示灯，当某个按钮按下时，它上面的指示灯会变亮。最左侧按钮的作用是将所有按钮复位。当程序执行到 Qn 和 Tn 指令时，系统会检查对应的条件转移断点选择按钮是否被按下。如果不是，那么系统会正常执行；如果是或按钮 ALL 被按下，那么系统会中断执行，相当于遇到一个断点。

当 UNIVAC I 因为条件转移断点而停止后，按钮左侧的条件转移 (CONDITIONAL

TRANSFER) 指示灯会根据指令的比较结果，显示即将跳转与否。如果调试人员希望执行与比较结果相反的动作，那么可以通过右侧的开关强制跳转或不跳转。



图 1-6 UNIVAC I 的条件转移断点控制按钮和指示灯

英特尔 P6 系列 CPU 引入了记录分支、中断和异常的功能，以及针对分支设置断点和单步执行，我们将在第 2 篇详细介绍这些功能。

以上简要介绍了 3 种调试功能的发展历史，从中我们可以看出从单纯的硬件机制到软硬件相互配合来调试软件的基本规律。使用软件来调试软件的最重要工具就是调试器（Debugger）。关于调试器的详细发展历史我们将在第 28 章介绍。

1.4 分类

根据被调试软件的特征、所使用的调试工具，以及软件的运行环境等要素，可以把软件调试分成很多个子类。本节将介绍几种常用的分类方法，并介绍每一种分类方法中的典型调试任务。

1.4.1 按调试目标的系统环境分

软件调试所使用的工具和方法与操作系统有着密切的关系。例如，很多调试器是针对操作系统所设计的，只能在某一种或几种操作系统上运行。对软件调试的一种基本分类标准就是被调试程序（调试目标）所运行的系统环境（操作系统）。按照这个标准，可以把调试分为 Windows 下的软件调试、Linux 下的软件调试、DOS 下的软件调试，等等。

这种分类方法主要是针对编译为机器码的本地（native）程序而言的，对于使用 Java 和.NET 等动态语言所编写的运行在虚拟机中的程序，它们具有较好的跨平台特性，与操作系统的关联度较低，因此不适用于这种分类方法（见下文）。

1.4.2 按目标代码的执行方式分

脚本语言具有简单易学、不需要编译等优点，比如网页开发中广泛使用的 JavaScript 和 VBScript。脚本程序是由专门的解释程序解释执行的，不需要产生目标代码，与编译执行的程序有很多不同。调试使用脚本语言编写的脚本程序的过程被称为脚本调试。使用的调试器被称为脚本调试器。

对于编译执行的程序，又主要分成两类。一类是先编译为中间代码，在运行时再动态编译为当前 CPU 能够执行的目标代码。典型的代表便是使用 C# 开发的.NET 程序。另一类是直接编译和链接成目标代码的程序，比如传统的 C/C++ 程序。为了区分，针对前一类代码的调试一般被称为托管调试，针对后一类程序的调试被称为本地调试（Native Debugging）。如果希望在同一个调试会话中既调试托管代码，又调试本地代码，那么这种调试方式被称为混合调试（Inter-op Debugging）。

图 1-7 归纳出了按照执行和编译方式来对软件调试进行分类的判断方法和步骤。

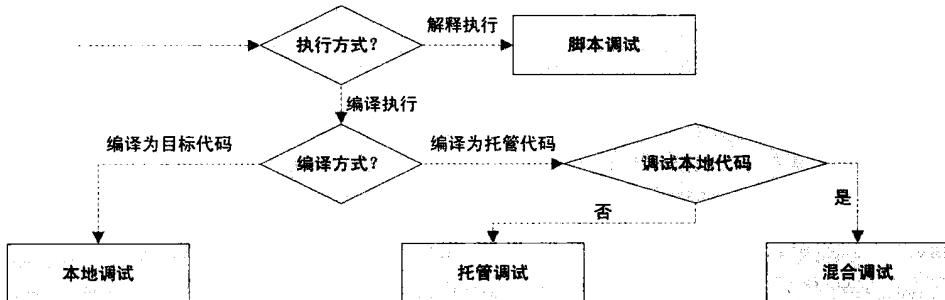


图 1-7 按照执行和编译方式对软件调试进行分类

本书重点讨论的是本地调试。

1.4.3 按目标代码的执行模式分

在 Windows 这样的多任务操作系统中，作为保证安全和秩序的一个根本措施，系统定义了两种执行模式，即低特权级的用户模式（User Mode）和高特权级的内核模式（Kernel Mode）。应用程序代码是运行在用户模式下的，操作系统的内核、执行体和大多数设备驱动程序是运行在内核模式的。因此，根据被调试程序的执行模式，可以把软件调试分为用户态调试（User Mode Debugging）和内核态调试（Kernel Mode Debugging）。

因为运行在内核态的代码主要是本地代码，以及很少量的脚本，例如 ASL 语言编写的 ACPI 脚本，所以内核态调试主要是调试本地代码。而用户态调试包括调试本地应用程序和调试托管应用程序等。

本书后面的章节将详细介绍 Windows 下的用户态调试和内核态调试。

1.4.4 按软件所处的阶段分

根据被调试软件所处的开发阶段，可以把软件调试分为开发期调试和产品期调试。二者的分界线是产品的正式发布。

产品期调试旨在解决产品已经发布后才发现的问题，问题的来源主要是客户通过

电子邮件、电话等方式报告的，或者通过软件的自动错误报告机制（将在第14章详细讨论）而得到的。与开发期调试相比，产品期调试具有如下特征。

- 因为产品期的问题没有被产品发布之前的测试过程所发现，所以它们很可能与特定的使用环境和使用方式有关。有时可能无法在调试者的环境中再现问题，这时可能要使用远程调试方法，或者到用户的环境中去，或者使用在用户环境中产生的故障转储文件。
- 产品期调试通常是在一个更大的范围内分析问题，因此，一个基本的思路就是逐渐缩小范围，逐步靠近问题根源。有时问题的根源不属于产品本身，调试的过程只是要证明这一点。
- 产品期调试时，被调试的模块大多是发布版本的，有些模块可能是其他公司的，没有源代码和符号文件。因此，产品期调试往往需要汇编级的分析和跟踪，或者分析堆栈中的原始数据。
- 如果是在客户的环境中进行调试，那么客户通常不愿意向他们的系统安装大量的工具或其他文件。如果不得不这样做，就需要先征得他们的同意。
- 产品期调试的时间要求往往更紧急，因为客户可能等待使用这个产品，或者无法理解为什么需要较长的时间。

总之，产品期调试的难度一般更大，对调试者的要求更高。

1.4.5 按调试器与调试目标的相对位置分

如果被调试程序（调试目标）和调试器在同一个计算机系统中，那么这种调试被称为本机调试（Local Debugging）。这里的同一个计算机系统是指在同一台计算机上的同一个操作系统中，不包括运行在同一个物理计算机上的多个虚拟机。

如果调试器和被调试程序分别位于不同的计算机系统中，它们通过以太网络或某种电缆进行通信，那么这种调试方式被称为远程调试（Remote Debugging）。远程调试通常需要在被调试程序所在的系统中运行一个调试服务器程序，这个服务器程序和远程的调试器相互联系，向调试器报告调试事件，并执行调试器下达的命令。当在第6篇讨论调试器时，我们将进一步讨论远程调试的工作方式。

利用Windows内核调试引擎所作的活动内核调试需要使用两台机器，两者之间通过串行口、1394或USB2.0进行连接。尽管这种调试的调试器和调试目标也在两台机器中，但是通常不将其归入远程调试的范畴。

1.4.6 按调试目标的活动性分

软件调试的目标通常是当时在实际运行的程序，但也可以是转储文件（Dump File）。因此，根据调试目标的活动性，可以把软件调试分为活动目标调试（Live Target

Debugging) 和转储文件调试 (Dump File Debugging)。转储文件以文件的形式将调试目标的内存状态凝固下来，包含了某一时刻的程序运行状态。转储文件调试是定位产品期问题和调试系统崩溃及应用程序崩溃的一种简便而有效的方法。

1.4.7 按调试工具分

也可以根据软件调试所使用的工具来对软件调试进行分类。最简单的就是按照调试时是否使用调试器分为使用调试器的软件调试和不使用调试器的软件调试。使用调试器的调试可以使用断点、单步执行、跟踪执行等强大的调试功能。不使用调试器的调试主要依靠调试信息输出、日志文件、观察内存和文件等。后者具有简单的优点，适用于调试简单的问题或无法使用调试器的情况。

本书主要讨论的是使用调试器的情况。

以上介绍了软件调试的几种常见分类方法，目的是让大家对典型的软件调试任务有概括性的了解。有些分类方法是有交叉性的，比如调试浏览器中的 JavaScript 属于脚本调试，也属于用户态调试。

1.5 调试技术概览

本节我们将浏览各种常用的软件调试技术，并简要讨论学习调试技术的意义。

1.5.1 断点

断点 (breakpoint) 是使用调试器进行调试时的最常用调试技术之一。其基本思想是在某一个位置设置一个“陷阱”，当 CPU 执行到这个位置时便停止执行被调试的程序，中断到调试器 (break into debugger) 中，让调试者进行分析和调试。调试者分析结束后，可以让被调试程序恢复执行。

根据断点的设置空间可以把断点分为如下几种。

- 代码断点：设置在内存空间中的断点，其地址通常为某一段代码的起始处。当 CPU 执行指定内存地址的代码（指令）时断点命中（hit），中断到调试器。使用调试器的图形界面或快捷键在某一行源代码或汇编代码处设置的断点便是代码断点。
- 数据断点：设置在内存空间中的断点，其地址一般为要监视变量（数据）的起始地址。当被调试程序访问指定内存地址的数据时断点命中。根据需要，可以定义触发断点的访问方式（读/写）和宽度（字节、字、双字等）。
- I/O 断点：设置在 I/O 空间中的断点，其地址为某一 I/O 地址。当程序访问指定 I/O 地址的端口时中断到调试器。与数据断点类似，也可以根据需要设置断点被触发的访问宽度。

根据断点的设置方法，可以把断点分为软件断点和硬件断点。软件断点通常是通过向指定的代码位置插入专用的断点指令来实现的，比如 IA32 CPU 的 INT 3 指令（机器码为 0xCC）就是断点指令。硬件断点通常是通过设置 CPU 的调试寄存器来设置的。IA32 CPU 定义了 8 个调试寄存器：DR0~DR7，可以同时设置最多 4 个硬件断点（对于一个调试会话）。通过调试寄存器可以设置以上 3 种断点中的任一种，但是通过断点指令只可以设置代码断点。

当中断到调试器时，系统或调试器会将被调试程序的状态保存到一个数据结构中，通常称为执行上下文（CONTEXT）。中断到调试器后，被调试程序是处于静止状态的，直到用户输入恢复执行命令。

追踪点（Tracepoint）是断点的一种衍生形式。其基本思路是：当设置一个追踪点时，调试器内部会当作特殊的断点来处理。当执行到追踪点时，系统会向调试器报告断点事件，在调试器收到后，会检查内部维护的断点列表，发现目前发生的是追踪点后，便执行这个追踪点所定义的行为，通常是打印提示信息和变量值，然后便直接恢复被调试程序执行。因为调试器是在执行追踪动作后立刻恢复被调试程序执行的，所以调试者没有感觉到被调试程序中断到调试器的过程，尽管实际上是发生的。

条件断点（Conditional Breakpoint）的工作方式也与此类似。当用户设置一个条件断点时，调试器实际插入的还是一个无条件断点，在断点命中、调试器收到调试事件后，它会检查这个断点的附加条件。如果条件满足，便中断给用户，让用户开始交互式调试；如果不满足，那么便立刻恢复被调试程序执行。

1.5.2 单步执行

单步执行（step by step）是最早的调试方式之一。简单来说，就是让应用程序按照某一步骤单位一步一步执行。根据每次要执行的步骤单位，又分为如下几种：

- 每次执行一条汇编指令，称为汇编语言一级的单步跟踪。其实现方法一般是设置 CPU 的单步执行标志，以 IA32 CPU 为例，设置 CPU 标志寄存器的 TF（Trap Flag，即陷阱标志位）位，可以让 CPU 每执行完一条指令便产生一个调试异常（INT 1），中断到调试器。
- 每次执行源代码（比汇编语言更高级的程序语言，如 C/C++）的一条语句，又称为源代码级的单步跟踪。高级语言的单步执行一般也是通过多次汇编一级的单步执行而实现的。当调试器每次收到调试事件时，它会判断程序指针（IP）是否还属于当前的高级语言语句，如果是便再次设置单步执行标志并立刻恢复执行，让 CPU 再执行一条汇编指令，直到程序指针指向的汇编指令已经属于其他语句。调试器通常是通过符号文件中的源代码行信息来判断程序指针所属于的源代码行的。
- 每次执行一个程序分支，又称为分支到分支单步跟踪。设置 IA32 CPU 的 DbgCtl MSR 寄存器的 BTF（Branch Trap Flag）标志后，再设置 TF 标志，便可以让 CPU 执行到

下一个分支指令时触发调试异常。WinDBG 的 tb 命令用来执行到下一个分支。

- 每次执行一个任务（线程），即当指定任务被调度执行时中断到调试器。当 IA32 CPU 切换到一个新的任务时，它会检查任务状态段（TSS）的 T 标志。如果该标志为 1，那么便产生调试异常。但目前的调试器大多还没有提供对应的功能。

单步执行可以跟踪程序执行的每一个步骤，观察代码的执行路线和数据的变化过程，是深入诊断软件动态特征的一种有效方法。但是随着软件向大型化方向的发展，从头到尾跟踪执行一个软件乃至一个模块，一般都不再可行了。一般的做法是先使用断点功能将程序中断到感兴趣的位置，然后再单步执行关键的代码。我们将在第 4 章详细介绍 CPU 的单步执行调试。

1.5.3 输出调试信息

打印和输出调试信息（debug output/print）是一种简单而“古老”的软件调试方式。其基本思想就是在程序中编写专门用于输出调试信息的语句，将程序运行的位置、状态和变量取值等信息以文本的形式输出到某一个可以观察到的地方，可以是控制台、窗口、文件，或者调试器。这种方法的优点是简单方便、不依赖于调试器和复杂的工具。因此至今仍在很多场合广泛使用。

这种简单的方式也有一些明显的缺点，比如需要在被调试程序中加入代码，如果被调试程序的某个位置没有打印语句，那么便无法观察到那里的信息，如果要增加打印语句，那么需要重新编译和更新程序。另外，这种方法容易影响程序的执行效率，打印出的文字所包含的信息有限，容易泄漏程序的技术细节，通常不可以动态开启、信息不是结构化的、难以分析和整理等。在第 27 章（27.5.2 节）我们将介绍使用这种方法应该注意的一些细节。

在 Windows 操作系统中，驱动程序可以使用 DbgPrint/DbgPrintEx 来输出调试信息，应用程序可以调用 OutputDebugString，控制台程序可以直接使用 print 函数打印信息。

利用这种方法输出的调试信息具有实时性的优点，但是同时也有容易丢失和被覆盖的缺点，不适用于长期保存和事后分析。

1.5.4 日志

与输出调试信息类似，写日志（log）是另一种被调试程序自发的辅助调试手段。其基本思想是在编写程序时加入特定的代码将程序运行的状态信息写到日志文件或数据库中的。

日志文件通常自动按时间取文件名，每一条记录也有详细的时间信息，因此适合长期保存和事后检查与分析。因此很多需要连续长时间在后台运行的服务器程序都具有日志机制。

Windows 操作系统提供了基本的日志记录、观察和管理(删除和备份)功能(Event Log)。Windows Vista 新引入了名为 Common Log File System (CLFS.SYS) 的内核模块，用于进一步加强日志功能。我们将在第 15 章详细介绍这些调试支持的内容。

1.5.5 事件追踪

打印调试信息和日志都是以文本形式来输出和记录信息的，因此不适合处理数据量庞大且速度要求高的情况。事件追踪机制 (Event Trace) 正是针对这一需求而设计的，它使用结构化的二进制形式来记录数据，观察时再根据格式文件将信息格式转化为文本形式，因此适用于监视频繁且复杂的软件过程，比如监视文件访问和网络通信等。

ETW (Event Trace for Windows) 是 Windows 操作系统内建的一种事件追踪机制，Windows 内核本身和很多 Windows 下的软件工具 (如 Bootvis、TCP/IP View) 都使用了该机制。我们将在第 16 章详细介绍 ETW 机制及其应用。

1.5.6 转储文件

某些情况下，我们希望将发生问题时的系统状态像拍照片一样永久保存下来，发送或带走后再进一步分析和调试，这就是转储文件 (Dump File) 的基本用途。理想情况下，转储文件是转储时目标程序运行系统的一个快照，包含了当时内存中的所有信息，包括代码和各种数据。但在实际情况下，考虑到转储文件过大时不仅要占用大量的磁盘空间，而且不便于发送和传递，因此转储文件通常分为小、中、大几种规格，最小的通常称为 Mini Dump。

Windows 操作系统提供了为应用程序和整个系统产生转储文件的机制，可以在不停止程序或系统运行的情况下产生转储文件。

1.5.7 栈回溯

目前的主流 CPU 架构都是使用栈来进行函数调用的，栈上记录了函数的返回地址，因此，通过递归式寻找放在栈上的函数返回地址，便可以追溯出当前线程的函数调用序列，这便是栈回溯 (Stack Backtrace) 的基本原理。通过栈回溯产生的函数调用信息被称为 Calling Stack。

栈回溯是记录和探索程序执行踪迹的极佳方法，使用这种方法，准确且基本不需要额外的开销。很多追踪和记录机制都是基于这种方法设计的。

因为从栈上得到的只是函数返回地址 (数值)，不是函数名称，所以为了便于理解，可以利用调试符号 (debug symbol) 文件将返回地址翻译成函数名。大多数编译器都支持在编译时生成调试符号。微软的调试符号服务器 (<http://msdl.microsoft.com/download/symbols>) 包含了多个 Windows 版本的系统文件的调试符号。我们将在第 25 章深入讨论调试符号。

大多数调试器都提供了栈回溯的功能，某些非调试器工具也可以记录和产生栈回溯信息。

1.5.8 反汇编

所谓反汇编 (disassemble)，就是将目标代码（指令）翻译为汇编代码。因为汇编代码与机器码有着简单的对应关系，所以反汇编是了解程序目标代码的一种非常直接而且有效的方式。有时当我们对高级语言的某一条语句的执行结果百思不得其解时，可以看一下它所对应的汇编代码，这时往往可以更快发现问题的症结。以下一节（1.6.1 节）将介绍的 `bad_div` 函数为例，看一下汇编指令，我们就可知编译器是将 C++ 中的除法操作编译为无符号整除指令 (DIV)，而不是有符号整除 (IDIV)。这正是错误所在。

另外，反汇编的依赖性非常小，根据二进制的可执行文件就可以得到汇编语言表示的程序。这也是反汇编的一大优点。

调试符号对于反汇编有着积极的意义，反汇编工具可以根据调试符号得到函数名和变量名等信息，这样产生的汇编代码具有更好的可读性。

大多数调试器提供了反汇编和跟踪汇编代码的能力。一些工具也提供了反汇编功能，IDA (Interactive Disassembler) 是其中非常著名的一个。

1.5.9 观察和修改数据

观察被调试程序的数据是了解程序内部状态的一种直接的方法。很多调试器提供了观察和修改数据的功能，包括变量和程序的栈及堆等重要数据结构。在调试符号的支持下，可以按照数据类型来显示结构化的数据。

寄存器值代表了程序运行的瞬时状态。观察和修改寄存器的值也是一种常见的调试技术。

1.5.10 控制被调试进程和线程

像 WinDBG 这样的调试器支持同时调试多个进程，每个进程又可以包含多个线程。调试器提供了单独挂起和恢复某一个或多个线程的功能，这对于调试多线程和分布式软件是很有帮助的。我们将在第 30 章详细介绍控制进程和线程的方法。

1.5.11 学习调试技术的意义

为什么要学习软件调试技术呢？

首先，软件调试技术是解决复杂软件问题的最强大工具。如果把解决复杂软件问

题看作一场战斗，那么软件调试技术便是一种可以直击要害而且锐不可当的武器。说直击要害是因为利用调试技术可以从问题的正面迎头而上，从问题症结着手，直接深入到内部。而不像很多其他技术那样需要从侧面探索，间接地推测，然后做大量的排查。说锐不可当是因为核心的调试技术大多来源于CPU和操作系统的直接支持，因此具有非常好的健壮性和稳定性，有较高的优先级。

第二，提高调试技术水平有利于提高软件工程师特别是程序员的工作效率，降低工作强度。很多软件工程师都认为调试软件花去了他们大半的时间。因此提高调试软件的技术水平和效率对于提高总的工作效率是非常有意义的。

第三，调试技术是学习其他软硬件技术的一个极好工具。通过软件调试技术的强大观察能力和断点、找回溯、跟踪等功能可以快速地了解一个软件和系统的模块、架构和工作流程，因此是学习其他软硬件技术的一个快速而有效的方法。笔者经常使用这种方法来学习新的开发工具、应用软件和操作系统。

另外，相对其他软件技术，软件调试技术具有更好的稳定性，不会在短时间内被淘汰。事实上，我们前面介绍的大多数调试技术都已经有几十年的历史了。因此，可以说软件调试技术是一门一旦掌握，便可以长期受用的技术。

1.6 错误与缺欠

软件缺欠是软件调试和测试过程的主要工作对象。现实中，人们经常交替使用几个名词来称呼软件问题，比如：Error、Bug、Fault、Failure、Defect。本节将介绍对这几个名词的一种常见的区分方法，说明本书的用法，并讨论有关的几个问题。

1.6.1 内因与表象

区分以上几个术语的一种方法是从内因和表面现象的角度来分析。一般认为，Failure（失败）是用来描述软件问题的可见部分，即外在的表现和症状（symptom）。而Error是导致这种表象的内因（root cause）。Fault是指因为内因而导致表象出现的那个错误状态。而Bug和Defect是对软件错误和失败的通用说法，二者之间没有显著的差异，或许Bug一词更通俗和口语化，而Defect（缺欠）一词正式一些。

以第一个登记到文档中的Bug为例，那只飞蛾是Error，计算机停止工作是Failure，70#继电器断路是Fault。当不区分内因和表象时，便可以模糊地说是Mark II中的一个缺欠，或者Mark II中的一个Bug。

进一步来说，一个错误（Error）可能导致很多个失败（Failure），也就是所谓的多个问题是同样根源（Same root cause）。另一方面，如果没有满足特定的条件，那么“错

误”是不会导致“失败”的，或者说错误是在一定条件下才表现出来的，表现的形式可能有多种。

以下面的函数为例：

```
int bad_div(int n,unsigned int m)
{
    return n/m;
}
```

当这样调用它时：

```
printf("%d/%d=%d!\n",6,3,bad_div(6,3));
```

打印出的结果是正确的：

```
6/3=2!
```

但是当这样调用它时：

```
printf("%d/%d=%d!\n",-6,3,bad_div(-6,3));
```

打印出的结果却是错误的：

```
-6/3=1431655763!
```

当然，如果参数 n 为 -10，m 为 2，那么结果也是错误的。其中的原因因为参数 m 是无符号整数，所以编译器在编译 n/m 时采用了无符号除法指令（DIV），这相当于把参数 n 也假设为无符号整数。因此，当 n 为负数时，实际上被当作了一个较大的正数来做除法，除后的商被返回。

对于这个例子，函数 `bad_div` 的代码存在错误，不应该将有符号整数和无符号整数直接做除法。这个错误当两个参数都为正数 6 和 3 时不会体现出来，但是当参数 n 为负数，m 不等于 1 时，可以体现出来，会导致“失败”症状，特别的，-6 除以 +3 会得到结果 1431655763。

在本书中，除非特别指出，我们通常使用 Bug 或软件缺欠（Defect）来描述软件调试所面对的软件问题。

1.6.2 谁的 Bug

在软件工程中，一个值得注意的问题是不要把 Bug 轻易归咎于某一个程序员。讨论 Bug 时，不要使用“你的 Bug”这样的说法，因为这样可能是不公平的，容易伤害程序员的自尊心，不利于调动他们的积极性。

简单来说，测试过程中发现的任何与软件需求规约不一致的现象都可以当作 Bug/Defect 报告出来。其中有些可能是因为代码中确实存在过失而导致的，而有些可能是与需求定义和前期设计有关的。因此，把和某个模块有关的 Bug 都归咎于负责这个模块的程序员可能是不恰当的。

所以，一种较好的方式是称呼某某模块的 Bug，而不要说成是某某人的 Bug。这

样，与这个模块有关的人员可以相互协作，共同努力，迅速将其解决，这对于个人和整个团队都是有好处的。

1.6.3 Bug 的生命周期

图 1-8 描述了一个典型的软件 Bug 从被发现到被消除所经历的主要过程。图中不带格线的矩形框代表的是测试人员的活动，而带格线的矩形框代表的是开发和调试人员的活动。

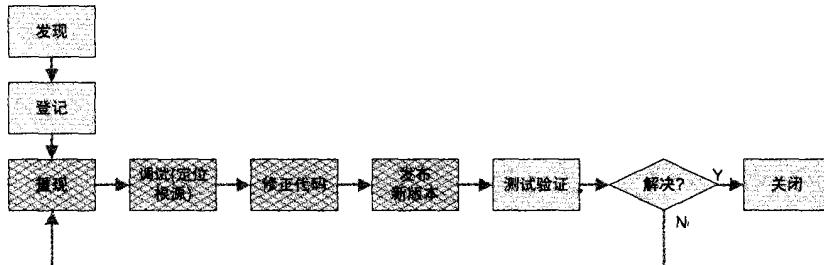


图 1-8 Bug 的生命周期

当登记一个 Bug 时，通常要为其指定如下属性：

- 严重程度。一般分为低、中、高、Critical、Showstopper 等。
- 状态。一般的做法是：新登记一般记录为 New，指定了负责人后修改为 Assigned，开发人员实现了解决方案等待测试验证时可以设置为 Resolved，测试人员验证问题已经解决后改为 Closed，由于某种原因此问题又重新出现后，那么可以设置为 Reopened。
- 环境。包括硬件平台（x86、x64、安腾等）、操作系统，等等。

Bug 被登录到系统（如 Bugzilla）中后，它会被指派一个负责人，这个负责人会先在自己的系统中重现问题，然后调试和定位问题的根源，找到根源后，修正代码，并作初步的测试，没有问题后将修正载入即将发布给测试人员的下一个版本中，并将系统中的 Bug 状态修改为 Resolved。而后测试人员进行测试和验证。如果经过一段时间的测试证明问题确实解决了，那么就可以关闭这个问题。对于严重程度很高的问题，可能需要通过团队会议讨论后才能关闭。

1.6.4 软件错误的开支曲线

我们把与一个软件错误直接相关的人力和物力投入称为此软件错误的开支（Cost）。

如果一个错误在设计或编码阶段就被发现和解决了，那么它所导致的开支主要是设计者或开发者所用的时间。

如果一个错误是在发布给测试团队后被测试人员所发现的，那么其开支便要包括

测试过程的各种投入，测试团队和开发团队相互沟通所需的人力和时间开销，重现问题和定位问题根源所需的投入，设计和实现解决方案及重新验证解决方案的投入。

如果一个错误是在软件正式发布后才发现的，那么其导致的危害通常会更大，可能的开支项目有处理客户投诉，远程支持，开发及发布补丁程序，客户退货，产品召回，赔偿导致的其他损失等（参见下文）。

容易看出，软件错误被发现和纠正得越早，其开支就越小。如果在开发阶段发现和得到纠正，那么就不需要测试阶段的开支了。如果等产品都已经发布给最终用户才发现问题，那么其导致的开支会是以前的数十倍乃至更多。Barry W. Boehm 在 *Software Engineering Economics* 一书中给出了在软件生命周期的不同阶段修正软件错误的相对成本（表 1-1）。

表 1-1 软件错误的相对开支

错误被检测和纠正的阶段	相对开支的中值
需求	2
设计	5
编码	10
开发测试（Development test）	20
接受测试（Acceptance test）	50
运行	150

图 1-9 是根据表 1-1 中的数据所画出的曲线，横轴代表软件生命周期的各个阶段（时间），纵轴代表发现和纠正软件错误的相对成本（中值）。

根据图 1-9 中的曲线，软件错误的开支是随着发现的时间呈指数形式上升的，所以应该尽可能早地发现和纠正问题，要做到这一点，需要软件团队中所有成员的共同努力，从一开始就注重程序的可测试性和可调试性。我们将在第 5 篇详细讨论可调试性和更多有关的问题。

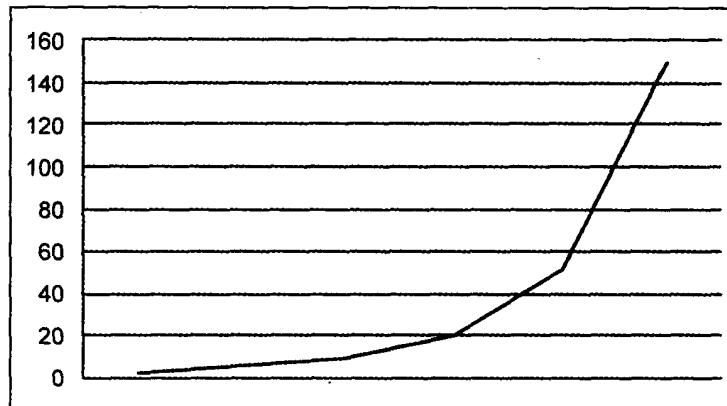


图 1-9 软件错误开支相对于软件生命周期各阶段的曲线

1.7 与软件工程的关系

从软件工程的角度来讲，软件调试是软件工程的一个重要部分，软件调试出现在软件工程的各个阶段。从最初的可行性分析、原型验证、到开发和测试阶段、再到发布后的维护与支持，都需要软件调试技术。

定位和修正 Bug 是几乎所有软件项目的重要问题，越临近发布，这个问题的重要性越高！很多软件项目的延期是由于无法在原来的期限内修正 Bug 所造成的。为了解决这个问题，整个软件团队都应该重视软件的可调试性，重视对软件调试风险的评估和预测，并预留时间。本节将简要介绍软件调试与软件工程中其他活动的关系。

1.7.1 调试与编码的关系

调试与编写代码（Coding）是软件开发中不同但密切联系的两个过程。在软件的开发阶段，对于一个模块（一段代码）来说，它的编写者通常也是它的调试者。或者说，一个程序员要负责调试他所编写的代码。这样做有两个非常大的好处。

- 调试过程可以让程序员了解程序的实际执行过程，检查与自己设计时的预想是否一致，如果不一致，那么很可能预示着有问题存在，应该引起重视。
- 调试过程可以让程序员更好地认识到提高代码可调试性和代码质量的重要性。从此，自觉改进编码方式，合理添加可用来支持调试的代码。

编码和调试是程序员日常工作中两项最主要的任务，这两项任务是相辅相成的，编写具有可调试性的高质量代码，可以明显提高调试效率，节约调试时间。另一方面，调试可以让程序员真切感受程序的实际执行过程，反思编码和设计中的问题，加深对软件和系统的理解，提高对代码的感知力和控制力。

在软件发布后，有些调试任务是由技术支持人员来完成的，但是当他们将错误定位到某个模块并且无法解决时，有时还要找到它的本来设计者。

很多经验丰富的程序员都把调试放在头等重要的位置，他们会利用各种调试手段观察、跟踪和理解代码的执行过程。通过调试，他们可以发现编码和设计中的问题，并把这些问题在发布给测试人员之前便纠正了。于是，人们认为他们编写代码的水平非常高，没有或者很少有 Bug，在团队中有非常好的口碑。对于测试人员发现的问题，他们也仿佛先知先觉，看了测试人员的描述后，一般很快就意识到了问题所在，因为他们已经通过调试把代码的静态和动态特征都放在大脑中了，对其了然于胸。

但也有些程序员很少跟踪和调试他们编写的代码，对这些代码何时被执行和执行多少次也心中无数。对于测试人员报告的一大堆问题，他们也经常是一头雾水，不知所措。

毋庸置疑，忽视调试对于提高程序员的编程水平和综合能力都是很不利的。因此，

Debugging by Thinking 一书的作者 Robert Charles Metzger 在其第 1 章中便说，导致今天的软件有如此多缺欠的原因有很多，其中之一就是很多程序员不擅长调试，一些程序员对待软件调试就像对待个人所得税申报表那样消极。

1.7.2 调试与测试的关系

简单地说，测试的目的是在不知道有问题存在的情况下寻找和发现问题，而调试是在已经知道问题存在的情况下定位问题根源的。从因果关系的角度来看，测试是旨在发现软件“表面”的不当行为和属性，而调试是寻找这个表象下面的内因。因此二者是有明显区别的，尽管有些人时常将它们混淆在一起。

如果说代码是联系调试与编码的桥梁，那么软件缺欠（defect）便是联系调试与测试的桥梁。缺欠是测试过程的成果（输出），是调试过程的输入。测试的目标首先是要发现缺欠，其次是如何协助关闭这些缺欠。

测试与调试的宗旨是一致的，那就是软件的按期交付。为了实现这一共同目标，测试人员与调试人员应该相互尊重，密切配合。例如，测试人员应该尽可能准确详细地描述缺欠，说明错误的症状，实际的结果和期待的结果，发现问题的软硬件环境，重现问题的方法，以及需要注意的细节。测试人员应该鼓励在软件中加入检查错误和辅助调试的手段，以便更快地定位问题。

软件的调试版本包含更多的错误检查环节，以便更容易测试出错误，因此除了测试软件的发布版本外，测试调试版本是提高测试效率、加快整个项目进度的有效措施。著名的调试专家 John Robbins 建议根据软件的开发阶段来安排测试调试版本的时间，在项目的初始阶段，对两个版本的测试时间应该是基本一样的，随着软件的成熟，逐渐过渡到只测试发布版本。

为了使以上方法更有效，编码时应该加入恰当的断言并建立合适的错误报告和记录机制，我们将在第 21 章介绍运行期检查时更详细讨论这个问题。

1.7.3 调试与逆向工程的关系

典型的软件开发过程是设计、编码，再编译为可执行文件（目标程序）的过程。因此，所谓逆向工程（Reverse Engineering）就是根据可执行文件反向推导出编码方式和设计方法的过程。

调试器是逆向工程中的一种主要工具。符号文件、跟踪执行、变量监视和观察、断点这些软件调试技术都是实施逆向工程时经常使用的技术手段。

逆向工程的合法性依赖于很多因素，需要视软件的授权协议、所在国家的法律、逆向工程的目的等具体情况而定，其细节超出了本书的讨论范围。

1.7.4 调试尚未得到应有的重视

尽管软件调试始终是软件开发中必不可少的一步，但迄今为止它仍然没有得到应该得到的重视。

在教育和研究领域，软件调试技术尚未像软件测试和编译原理那样成为一个独立的学科，有关的理论和知识尚未系统化，专门讨论软件调试的书籍和资料非常有限。根据笔者的了解，还没有一所大学或软件学院开设专门关于软件调试的课程。这导致很多软件工程师没有接受过系统的软件调试培训，对软件调试的基本原理知之甚少。

在软件工程中，很多时候，软件调试还处于被忽略的位置，当定义日程表时，很少专门评估软件调试方面的风险，为其预留专门的时间；当架构设计时，很少考虑软件的可调试性；在开发阶段，针对调试方面的管理和约束也很薄弱，一个项目中经常存在着多种调试机制，相互重叠，而且都很简陋。在员工培训方面，针对软件调试的培训也比较少。

在第5篇（第26~27章）我们将进一步讨论软件调试与软件工程的更多话题，特别是软件的可调试性。

1.8 本章总结

本章的前两节介绍了软件调试的解释性定义、基本过程（1.1节）和特征（1.2节）。第1.3节讨论了断点、单步执行和分支监视3种基本的软件调试技术的简要发展历史。第1.4节从多个角度介绍了常见的软件调试任务。第1.5节介绍了软件调试所使用的基本技术。第1.6节探讨了关于软件错误的一些术语和概念。最后一节介绍了软件调试与软件工程中其他活动的关系。

作为全书的开篇，本章旨在为读者勾勒出一个关于软件调试的总体轮廓，建立一些初步的概念和印象。所以，本章的内容大多是概括性的介绍，没有深入展开，如果读者不能理解其中的某些术语和概念，也不要紧，因为后面的章节会有更详细的介绍。

参考文献

1. Robert Charles Metzger. *Debugging by Thinking*. Elsevier Digital Press, 2003
2. G. Pascal Zachary. *Showstopper: The Breakneck Race to Create Windows NT and the Next Generation at Microsoft*. The Free Press, 1994
3. *Manual of Operations for UNIVAC System*. Remington Rand Inc., 1954

CPU 的调试支持



如果把程序（Program）中的每一条指令看作是电影胶片的一帧，那么执行程序的 CPU 就像是一台飞速运转的放映机。以英特尔 P6 系列 CPU 为例，其处理能力大约在 300（第一代产品 Pentium Pro）~3000（奔腾 III）MIPS 之间，MIPS 的含义是 CPU 每秒钟能执行的指令数（以百万指令为单位）。如果按 3000 MIPS 计算，那么意味着每秒钟大约有 30 亿条指令“流”过这台高速的“放映机”。这大约是电影胶片放映速度（24 帧每秒）的 1.25 亿倍。如此高的执行速度，如果在程序中存在错误或 CPU 内部发生了错误该如何调试呢？

CPU 的设计者们一开始就考虑到了这个问题——如何在 CPU 中包含对调试的支持。就像在制作电影过程中人们可以慢速放映或停下来分析每一帧一样，CPU 也提供了一系列机制允许一条一条地执行指令，或者停在指定的位置。

以英特尔的 IA 结构 CPU 为例，其提供的调试支持有。

- INT 3 指令：又叫断点指令，当 CPU 执行到该指令时便会产生断点异常，以便中断到调试器程序。INT 3 指令是软件断点的实现基础。
- 标志寄存器 (EFLAGS) 中的 TF 标志：陷阱标志位，当该标志为 1 时，CPU 每执行完一条指令就产生调试异常，陷阱标志位是单步执行的实现基础。
- 调试寄存器 DR0~DR7：用于设置硬件断点和报告调试异常的细节。
- 断点异常 (#BP)：当 INT 3 指令执行时，会导致此异常，CPU 转到该异常的处理例程。异常处理例程会进一步将异常分发给调试器软件。
- 调试异常 (#DB)：当除 INT 3 指令以外的调试事件发生时，会导致此异常。
- 任务状态段 (TSS) 的 T 标志：任务陷阱标志，当切换到设置了 T 标志的任务时，CPU 会产生调试异常，中断到调试器。
- 分支记录机制：用来记录上一个分支、中断和异常的地址等信息。
- 性能监视：用于监视和优化 CPU 及软件的执行效率。
- JTAG 支持：可以与 JTAG 调试器一起工作来调试单独靠软件调试器无法调试的问题。

除了对调试功能的直接支持，CPU 的很多核心机制也为实现调试功能提供了硬件基础，比如异常机制、保护模式和性能监视功能等。

本篇首先将概括性地介绍 CPU 的基本概念和常识（第 2 章），包括指令集、寄存器及保护模式等重要概念。然后介绍与调试功能密切相关的中断和异常机制（第 3 章），包括异常的分类，优先级等。这两章的内容旨在帮助读者了解现代 CPU 的概貌和重要特征，为理解本书后面的章节打下基础。对于了解 CPU 较少的读者，应该认真阅读这两章的内容。对于其他读者，这些内容可以作为复习资料和温故知新。第 4 章将详细讨论软件断点、硬件断点和陷阱标志的工作原理，从 CPU 层次详细解析支持常用调试功能的硬件基础。第 5 章将介绍 CPU 的分支监视、调试存储和性能监视机制。第 6 章将介绍 CPU 的机器检查机制 (Machine Check Architecture)，包括机器检查异常和处理方法等。第 7 章将介绍 JTAG 原理和 IA-32 CPU 的 JTAG 支持。

CPU 基础

CPU 是 Central Processing Unit 的缩写，即中央处理单元，或者叫中央处理器，有时也简称为处理器(Processor)。第一款集成在单一芯片上的 CPU 是英特尔公司于 1969 年开始设计并于 1971 年推出的 4004，与当时的其他 CPU 相比，它的体积可算是微乎其微，因此，人们把这种实现单一芯片上的 CPU (Single-chip CPU) 称为微处理器(Microprocessor)。今天，绝大多数(即使不是全部)CPU 都是集成在单一芯片上的，甚至多核技术还把多个 CPU 内核(Core) 集成在一块芯片上，因此微处理器和处理器这两个术语也几乎被等同起来了。

尽管现代 CPU 的集成度不断提高，结构也变得越来越复杂，但是它在计算机系统中的角色仍然非常简单，那就是从内存中读取指令(fetch instructions)，然后解码(decode)和执行(execute)。指令是 CPU 可以理解并执行的操作(operation)，它是 CPU 能够“看懂”的唯一语言。本章我们将以这一核心任务为线索，介绍关于 CPU 的基本知识和概念。

2.1 指令和指令集

某一类 CPU 所支持的指令集合被简称为指令集(Instruction Set)。根据指令集的特征，可以把 CPU 划分为两大阵营，即 RISC 和 CISC。

RISC (Reduced Instruction Set Computer，即精简指令集计算机) 是 IBM 研究中心的 John Cocke 博士于 1974 年最先提出的。其基本思想是通过减少指令的数量和简化指令的格式来优化和提高 CPU 执行指令的效率。RISC 出现后，人们很自然地把与 RISC 相对的另一类指令集称为 CISC (Complex Instruction Set Computer，即复杂指令集计算机)。

RISC 处理器的典型代表有 SPARC 处理器、PowerPC 处理器、惠普公司的 PA-RISC 处理器、MIPS 处理器、Alpha 处理器和 ARM 处理器等。

CISC 处理器的典型代表有 x86 处理器和 DEC VAX-11 处理器等。第一款 x86 处理器是英特尔公司于 1978 年推出的 8086，其后的 8088、80286、80386、80486、奔腾处理器及 AMD 等公司的兼容处理器都是兼容 8086 的，因此人们把基于该架构的处理器统称为 x86 处理器。

2.1.1 基本特征

下面将以比较的方式来介绍 RISC 处理器和 CISC 处理器的基本特征和主要差别。除非特别说明，我们使用 ARM 处理器代表 RISC 处理器，使用 x86 处理器代表 CISC 处理器。

首先，大多数 RISC 处理器的指令都是等长度的（通常为 4 个字节，即 32 比特），而 CISC 处理器的指令长度是不确定的，最短的指令是 1 个字节，有些长的指令有十几个字节（x86）甚至几十个字节（VAX-11）。定长的指令有利于解码和优化，缺点是目标代码占用的空间比较大（因为有些指令是没必要用 4 字节的）。对于软件调试而言，定长的指令有利于实现反汇编和代码断点（4.1 节将详细讨论）。

第二，RISC 处理器的寻址方式（addressing mode）比 CISC 少了很多种，我们稍后将单独介绍。

第三，与 RISC 相比，CISC 处理器的通用寄存器（general register）数量较少。例如 16 位和 32 位的 x86 处理器有 8 个通用寄存器：EAX、EBX、ECX、EDX、ESI、EDI、EBP、ESP（EBP 和 ESP 通常用于维护栈——stack）。64 位的 x86 处理器增加了 8 个通用寄存器（R8~R15），但是总量仍然远远小于 RISC 处理器（通常多达 32 个）。寄存器位于 CPU 内部，CPU 可以直接使用，与访问内存相比，其效率更高。

第四，RISC 的指令数量也相对较少。就以跳转指令为例，8086 有 32 条跳转指令（JA、JAE、JB、JPO、JS、JZ 等），而 ARM 处理器只有两条跳转指令：BLNV 和 BLEQ。跳转指令对流水线执行很不利，因为一旦遇到跳转指令，CPU 就需要做分支预测（branch prediction），而一旦预测失败，就要把已经执行的错误分支结果清理掉，这会降低 CPU 的执行效率。但是丰富的跳转指令为编程提供了很多方便，这是 CISC 处理器的优势。

第五，从函数（或子程序）调用（function/procedure call）来看，二者也有所不同。RISC 处理器因具有较多的寄存器，通常就有足够多的寄存器来传递函数的参数。而在 CISC 中，即使使用所谓的快速调用（fast call）协定，也只能将两个参数用寄存器来传递，其他参数仍然需要用栈来传递。从执行速度看，使用寄存器的速度更快。我们将在第 22 章的调用协定一节（22.5 节）中进一步讨论函数调用的细节。

因为以上特征，RISC 处理器的实现相对来说简单一些，这也是很多低成本的嵌入式系统使用的处理器大多采用 RISC 架构的一个原因。关于 RISC 和 CISC 的优劣，

一直存在着很多争论，采用两种技术的处理器也在相互借鉴对方的优点。比如从 P6 系列处理器的第一代产品 Pentium Pro 开始，英特尔的 x86 处理器就开始将 CISC 指令先翻译成等长的微操作（micro-ops 或 μops），然后再执行。微操作与 RISC 指令很类似，因此很多时候又被称为微指令。因此可以说今天的主流 x86 处理器（不包括那些用于嵌入式系统的 x86 处理器）的内部已经具有了 RISC 的特征。

2.1.2 寻址方式

指令由操作码（opcode）和 0 到多个操作数组成。寻址方式定义了如何得到操作数，是指令系统乃至 CPU 架构的关键特征。下面以 x86 汇编语言为例简要介绍常见的寻址方式。

立即寻址（immediate addressing）：操作数直接跟随在操作码之后，作为指令的一部分存放在代码段里，比如在 MOV AL, 5（机器码 B0 05）这条指令中，源操作数采用的就是立即寻址方式。

寄存器寻址（register addressing）：操作数被预先放在寄存器中，指令中指定寄存器号，例如在 MOV AX, BX（机器码 8A C3）中源操作数使用的是寄存器寻址方式。

直接寻址（direct addressing）：操作数的有效地址（Effective Address，简称 EA）直接作为指令的一部分跟随在操作码之后，比如 MOV AX, [402128H]（机器码 B8 28 21 40 00）指令中的，源操作数采用的就是直接寻址方式。

寄存器间接寻址（register indirect addressing）：操作数的地址被预先放在一个或多个寄存器中，比如 ADD AX, [BX] 这条指令是把 BX 寄存器所代表地址中的值累加到 AX 寄存器中的，其源操作数采用的就是寄存器间接寻址方式。

间接寻址方式还有几种，这里不再一一列举。间接寻址方式为处理表格和字符串等数据结构带来了很大的方便。例如，可以把表格或字符串的起始地址放入一个基址寄存器中，然后用一个变址寄存器指向各个元素。但间接寻址带来的问题就是使指令格式变得复杂，导致解码和优化的难度增大。为回避这样的问题，RISC 处理器通常只支持简单的寻址方式，不支持间接寻址，不支持在一条指令中既访问内存又进行数学运算（如从内存中读出并累加到目标操作数，ADD AX, [BX]）。也就是说，RISC 处理器中的运算指令在执行过程中通常是不访问内存的，运算指令的所有操作数要么是立即数，要么就是被预先读到寄存器中，而且执行结果也是被保存到寄存器中的。这样做的优点是涉及内存访问的指令变得很少，有利于提高 CPU 执行流水线（pipeline）的效率。

2.1.3 指令的执行过程

图 2-1 以英特尔 P6 系列 CPU 为例简单地画出了指令在 CPU 中的执行过程。

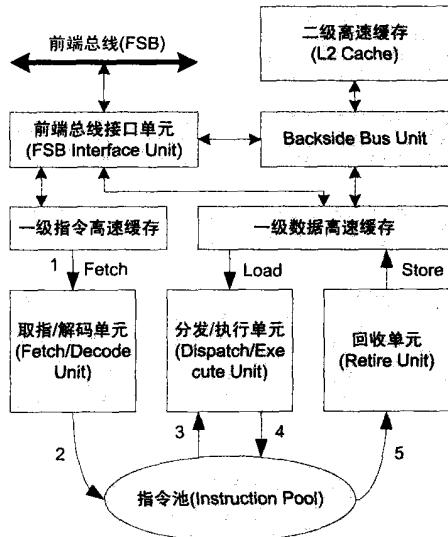


图 2-1 指令在 CPU 中的执行过程 (P6 处理器)

箭头 1 代表取指/解码单元 (Fetch/Decode Unit) 从高速缓存中读取指令并解码成等长的微操作后放入指令池中 (箭头 2)。箭头 3 代表分发/执行单元 (Dispatch/Execute Unit) 从指令池中挑选出等待执行的指令分发到合适的执行单元中进行执行，执行单元执行后把结果再放回到指令池中 (箭头 4)。箭头 5 代表回收单元 (Retire Unit) 检查指令池中的指令状态，根据程序逻辑将临时执行结果按顺序永久化。《P6 系列处理器硬件开发者手册》(参考文献 11) 更详细地描述了以上过程，感兴趣的读者可以从英特尔的网站上下载和阅读。

本节介绍了 RISC 和 CISC 指令集的概况，目的主要是让读者对整个 CPU 世界有个概括性的了解。本节之后及本书的其他章节讨论的主要 x86 阵营中的 IA-32 处理器，下一节将对 IA-32 处理器做概括性的介绍。

2.2 IA-32 处理器

IA-32 处理器是对英特尔设计和生产的 x86 系列 32 位微处理器的总称。IA-32 (或 IA32) 是 Intel Architecture 32-bit 的缩写，用来泛指 IA-32 处理器所使用的架构和共同特征。

IA-32 处理器的典型代表有 80386、80486、Pentium (奔腾)、Pentium Pro、Pentium II、Pentium III、Pentium 4 (简称 P4)、Pentium M、Core Duo、Core 2 Duo 及 Celeron (赛扬) 和 Xeon (至强) 处理器。其中 Pentium Pro、Pentium II 和 Pentium III 因为都是基于共同的 P6 处理器内核的，所以又被统称为 P6 系列处理器。赛扬和至强处理器分别是 P6 (奔腾 II 和奔腾 III) 和 P4 等处理器针对低端和服务器市场的改进版本，从

指令和架构上看并没有根本的变化。2006 年推出的 Core 2 Duo 处理器采用了针对多核特征而设计的 Intel Core Architecture，代表了 CPU 向多核方向发展的趋势。

概而言之，可以把迄今为止的 IA-32 处理器概括为：386 系列、486 系列、奔腾系列、P6 系列、P4 系列和 Core 系列。因为本章及后面的章节还经常要提到这些处理器，下面便按时间顺序简要概括它们的特征。需要特别强调的是，以下介绍的只是与本书内容密切相关的特征，并不是全部特征。

2.2.1 80386 处理器

1985 年 10 月，英特尔在推出 80286 三年半之后推出了 80386（简称 386），它是 x86 系列处理器中的第一款 32 位处理器，被认为是 IA-32 架构的开始。尽管 286 最先引入了保护模式的概念，为运行多任务操作系统打下了基础，但是 386 的推出，才真正使基于 x86 处理器的多任务操作系统（Windows）流行起来。386 处理器引入的新特性主要有。

- 32 位地址总线，可以最多支持 4GB 的物理内存。
- 平坦内存模型（flat memory model），即每个应用程序可以使用地址 0~ 2^{32} -1 来索引 4GB 的平坦内存空间。因为 4GB 足以容纳大多数应用程序的所有代码和数据，所以采用此模型后，可以把程序的所有代码都放在一个段内，这样在跳转时便都是段内跳转，而不必考虑段的界限。此前的实模式下一个段最大只有 64KB，程序要维护多个代码段和多个数据段，不得不考虑段间跳转和段边界等棘手问题。
- 分页机制（Paging），即以页为单位来组织内存，通过页表进行管理，暂时不用的页可以交换到硬盘等外部存储器上，需要时再交换回来。分页机制是实现虚拟内存的硬件基础。在 386 中，每个内存页的大小为 4KB，Pentium 处理器引入了大页面支持，可以建立 4MB 的大内存页。分页机制是今天的所有主流操作系统必须依赖的硬件功能。值得说明的是，在 Windows 系统中禁用专门的分页文件（Paging File），并不代表 Windows 就不再依赖分页机制，文件映射、栈增长（22.6 节）等机制还是要依赖 CPU 的分页机制的。
- 调试寄存器，支持更强大的调试功能，我们将在第 4 章（4.2 节）讨论其细节。
- 虚拟 8086 模式，使 16 位的 8086/8088 程序可以更高效地在 32 位处理器中执行。

2.2.2 80486 处理器

1989 年推出的 80486 处理器引入了以下特性。

- 在 CPU 内部集成高速缓存（cache）。486 是第一个在芯片内部集成一级高速缓存（L1 Cache）的 IA-32 处理器，此后的所有 IA-32 处理器内部都有集成的一级高速缓存。
- 将数学协处理器（FPU）集成到 CPU 内。

- 内存对齐检查异常 (alignment check exception, exception 17)，我们将在第3章中详细讨论CPU的异常机制。
- 系统管理模式 (System Management Mode)，用于执行系统管理功能，如电源管理、硬件控制或执行OEM设计的与平台相关的固件程序 (Firmware)。需要说明的是，最先引入SMM支持的英特尔CPU是386 SL，但是从486 DX开始SMM才被加入到所有IA-32处理器中，成为IA-32架构的特征。

2.2.3 奔腾处理器

从1993年开始，Pentium（奔腾）处理器陆续推出，奔腾处理器可以说是IA-32处理器新一轮变革的开始，不仅从名字上不再沿用80386、80486这一惯例，而且从结构和性能上也有很大的突破。最重要的一点就是引入第二条执行流水线，支持指令一级的并行执行 (instruction level parallelism)。指令级的并行机制有助于更充分地利用处理器资源，在主频保持不变的情况下大幅提升处理器的执行能力。奔腾以后的IA-32处理器都延续了这一思想。或许是因为新一轮革新的开始，最早推出的奔腾处理器和后来推出的奔腾处理器(不是指Pentium Pro和奔腾II、奔腾III)除了主频上的差异外，其内部特征也有较大差异。人们通常根据内核代号将这些不同特征的奔腾处理器分为P5、P54C和P55C这三类。最早推出的P5内核奔腾处理器(主要有P60和P66)引入的新特性主要有。

- 数据总线的宽度从32位增加到64位，一次便可以从内存读取8个字节的数据。
- 加入第二条执行流水线 (execution pipeline)，允许同时有两条指令在解码和执行，因此每个时钟周期最多可以执行的指令数由以前的一条提升为两条，该特征又被称为超标量 (superscalar) 架构。因为指令间的相互依赖性，所以很多时候必须等待前面的指令执行完毕，才能执行后面的指令，也就是两条流水线并不是总能同时在使用。为了提高每条流水线的利用率，奔腾处理器第一次引入了分支预测功能。即预测最可能的分支并提早解码和执行可能的分支。
- 内部(一级)高速缓存增加为16KB，其中8KB用于数据，8KB用于代码。这也是第一次将数据高速缓存和代码高速缓存分开。
- 支持4MB的大页面内存。即除了4KB的内存页，还可以创建4MB的大页面。
- 引入性能监视机制，可以监视内部事件(计数)和流水线的执行情况。包括性能监视计数器、TSC (Time Stamp Counter) 寄存器和读取TSC寄存器的RDTSC指令等，我们将在第5章(5.3节)详细讨论。
- 引入内部错误探测功能，即机器检查机制 (Machine Check Architecture, MCA)，包括BIST (Built In Self Test) 和机器检查异常等，我们将在第6章详细讨论。
- 引入JTAG调试 (IEEE 1149.1) 支持，我们将在第7章详细讨论。
- 双(多)处理器支持。

1994 年开始推出的 P54C 内核奔腾处理器（如 P75、P90、P100、P120、P133、P150、P166、P200）的一个最大的变化就是在处理器内部加入了 APIC，称为本地 APIC（Local APIC）。APIC 是 Advanced Programmable Interrupt Controller 的缩写，即高级可编程中断控制器。本地 APIC 除了负责接收来自处理器的中断管脚、内部中断源（比如内部的温度传感器、性能计数器和 APIC 时钟）和外部 IO APIC 的中断外，在多处理器系统中还负责通过 APIC 串行总线（3 根线组成）与其他处理器进行通信。IO APIC 是芯片组的一部分（通常集成在南桥内），负责接收所有外部硬件中断，并翻译成消息发给处理中断的处理器。

1996 年 10 月推出的带有 MMX（MultiMedia eXtensions）功能的奔腾处理器（Intel Pentium with MMX Technology，简称奔腾 MMX）使用的是 P55C 内核，奔腾 MMX 在以下几方面做了增强。

- 支持 MMX（MultiMedia eXtensions）技术，即通过单一指令处理多个数据（Single Instruction Multiple Data，简称 SIMD）的方式提高并行运算能力，尤其是多媒体处理方面的性能。
- 一级高速缓存加倍（数据和代码各 16KB）。
- 优化了分支预测单元和指令解码器。
- 引入了 MSR（Model Specific Register）寄存器和访问 MSR 寄存器的两条指令 RDMSR 及 WRMSR，详见下一节关于寄存器的介绍。

2.2.4 P6 系列处理器

从 1995 年开始，英特尔相继推出了 P6 系列中的各款处理器。1995 年 11 月推出的 Pentium Pro 处理器作为 P6 系列的第一款处理器引入了很多此前 x86 处理器没有的重要属性。最值得一提的就是引入了类似 RISC 指令的微操作（micro-ops）：将原本的 x86 指令先翻译成等长的微操作后再执行。这一革命性的变化带来很多好处，包括更有利于提高执行效率和更适合多流水线执行等，其后的 IA-32 处理器都保持了这一特征。此外，Pentium Pro 处理器还引入如下新特性。

- 首次在内部集成二级高速缓存（256KB 或 1MB）。
- 地址总线的宽度从 32 位增加到 36 位，从而可以最多寻址 64GB 物理内存。这个特征又被称为 PAE-36 模式，即 Physical Address Extension 36-bit。
- 3 路超标量微架构（3-way Superscalar Micro-Architecture）。将指令执行任务划分为三个相对独立的单元：取指/解码单元、分发/执行单元和回收单元。解码单元的三个并行解码器首先将 x86 指令翻译（分解）为等长的微操作，然后将微操作送入 RAT（Register Alias Table）单元对寄存器重命名（register renaming），目的是将原本使用同一个寄存器的多条指令通过寄存器重命名使它们使用不同的别名寄存

器，以消除依赖性。接下来，在为每个微操作加上状态信息后将它们放入指令池（instruction pool）。指令池又被称为 ROB（Re-Order Buffer）。分发单元的 Reservation Station 负责监视 ROB，将 ROB 中已经就绪（操作数齐备）的微操作通过 5 个端口之一分配给执行单元去执行。因为 ROB 中的微操作只要状态就绪就可以执行，因此指令的执行顺序和程序中的先后顺序可能是不一致的，所以这种执行方式被称为乱序执行（out of order execution）。回收单元（Retire Unit）负责监视 ROB 中的微操作并将执行完毕的微操作按照程序本来的顺序从 ROB 中移除。每次回收的结果会被写入 Retirement Register File（RRF）。每个时钟周期，回收单元最多可以回收两条微操作。

- 投机取指/投机执行（Speculative Prefetch / Speculative Execution），即根据分支预测（Branch Prediction）得出的预测结果在没有执行到分支语句处就预先对最可能的分支进行取指和执行。这种投机式的做法可以减少执行流水线的空闲，还可以进一步提高乱序执行的效率。
- 去除了 MMX 支持（Pentium II 又恢复）。
- 引入了内存类型范围寄存器（Memory Type and Range Register），简称 MTRR。MTRR 用以标示某一段内存区（物理内存空间）的内存特征，比如只读、可写、是否应该高速缓存等。共定义了 5 种内存类型：分别是 Uncacheable，简称 UC，例如映射到内存空间中的外部设备上的存储器；Write-Combining，简称 WC，例如显存；Write-Through，简称 WT；Write-Protect，简称 WP，例如复制到内存中的原本存储在 ROM 中的 BIOS 代码；Write-Back，简称 WB，例如系统中的正常内存（RAM）。

1997 年 5 月推出的 Pentium II 处理器（代号 Klamath）的一大变化是将 Pentium Pro 集成到芯片内的二级缓存移到芯片外的一块特制的电路板（称为 Single Edge Contact Cartridge ——SECC）上。从架构上看，Pentium II 与 Pentium Pro 的变化不大。

- 重新加入由 Pentium 引入，但被 Pentium Pro 去掉了的 MMX 支持。
- 数据和指令高速缓存都从 16KB 提高到 32KB。
- 增加了快速系统调用和返回（Fast System Call/Return）指令，我们将在第 8 章详细介绍这一特征（8.3 节）。

在奔腾 II 处理器推出后不久，英特尔便将 P6 处理器划分为 3 个产品线：Pentium II 处理器针对中高端台式机市场；至强（Xeon）处理器面向工作站和服务器市场；赛扬（Celeron）处理器面向低端台式机市场。这种模式一直延续至今。

1999 年 2 月推出的奔腾 III 处理器引入了以下新特征。

- 单指令多数据扩展（Streaming SIMD Extensions），简称 SSE。除了加入 70 条新的指令，还引入 8 个 128 位的数据寄存器（XMM[7:0]），可以对单精度浮点数进行 SIMD 运算。
- 增加了 FXSAVE 和 FXRSTOR 指令，可以把 FPU（Floating Point Unit 浮点运算单

元) 和 SSE 寄存器的数据保存到内存或从内存中恢复回来。

2003 年 3 月推出的 Pentium M 处理器是专门为笔记本电脑等移动平台 (mobile platform) 设计的, 具有非常好的低功耗特征。2006 年推出的 Intel Core Duo 处理器将两个 CPU 内核集成在一个物理处理器中。从 CPU 架构和指令集的角度来看, 这两种处理器仍是基于 P6 架构的。

2.2.5 奔腾 4 处理器

2000 年开始陆续推出的奔腾 4 处理器采用了与 P6 差异很大的被称为 NetBurst 的超流水线微架构 (Hyper-pipelines Micro-architecture), 对处理器内核进行了完全的重新设计。其主要特征有。

- 流水线的级 (stage) 数由 P6 处理器的 10 级增加到 20 级 (奔腾 4 Prescott 增为 31 级)。
- 超线程 (Hyper-Threading, 简称 HT), 即在一块 CPU 芯片内实现两个处理器 (被称为逻辑处理器) 的功能, 使两个线程可以同时执行。
- 加入了分支踪迹存储 (Branch Trace Store, 简称 BTS) 功能, 使分支、中断和异常记录功能更强大。
- 加入了 SSE2 指令, 奔腾 4 Prescott 加入了 SSE3 指令。
- 性能计数器从 P6 的两个增加到 18 个。
- 温度监控功能, 当集成的温度传感器检测到内部温度超过跳变点 (trip point) 时, 会设置 PROCHOT#信号通知温控电路 (thermal monitor circuit)。
- P4 的 6xx 系列和支持超线程的 Extreme 版本引入了 EM64T (Extended Memory 64 Technology) 技术, 通过 IA-32e 模式支持 64 位计算。

2.2.6 Core 2 系列处理器

从 2006 年 7 月 27 日开始推出的 Core 2 系列处理器是基于被称为第 8 代 x86 架构的英特尔多核微架构 (Intel Core Microarchitecture) 的。典型的产品有双核的 Core 2 Duo CPU 和四核的 Core 2 Quad CPU。该系列 CPU 的主要特征有。

- 对 P6 引入的动态执行能力做了进一步增强, 每个 CPU 内核在一个时钟周期最多可以执行 4 条指令, 称为 Wide Dynamic Execution。
- 可以把某些常见的 x86 指令合并成一条微操作来执行, 称为 Macro-Fusion。
- 继承和增强了 Pentium M 的低功耗设计。
- 增加了用于提高乱序执行效率的 Memory Disambiguation 机制, 基于 IP 指针的缓存预取机制, 统称为 Intel Smart Memory Access 技术。

本书中关于处理器的内容主要是针对以上 IA-32 处理器的, 但是系统介绍这些处

理器的功能和结构远远超出了本书的范围。了解 IA-32 处理器的一个极佳途径就是阅读英特尔公司的《IA-32 架构软件开发者手册》(IA-32 Intel® Architecture Software Developer's Manual)。该手册的目前版本分为 3 卷 5 册。第 1 卷《基本架构》介绍了基本执行环境、数据类型、指令集概要、过程调用、中断和异常、一般编程和使用 FPU、MMX、SSE 编程等内容。第 2 卷《指令参考》分两册（卷 2A 和卷 2B）按字母顺序详细地介绍了每一条指令。最后一卷《系统编程指南》（卷 3A 和卷 3B）从更深的层次阐述了 IA-32 架构的关键特征，包括保护模式下的内存管理、保护机制、中断和异常处理、任务管理、多处理器管理、高级可编程中断控制器（APIC）、处理器管理和初始化、高速缓存管理、MMX/SSE/SSE2 系统编程、系统管理、MCA、调试和性能监控、8086 模拟、混合 16 位和 32 位代码，以及 IA32 兼容性等内容。可以从英特尔网站免费下载 IA-32 手册的电子版本，英特尔公司也不定期地免费提供这些手册的印刷版本，详细情况请访问英特尔公司的网站。

2.3 CPU 的操作模式

英特尔公司 1978 年推出的 8086 处理器是 x86 处理器的第一代产品，其后的 IA-32 处理器都是从 8086 的基础上发展演变而来的。尽管今天的 IA-32 处理器与二十多年前的 8086 相比，功能上已经有天壤之别，但是它们仍保持着对包括 8086 在内的低版本处理器的向下兼容性。因此，今天的 IA-32 处理器仍然可以很好地执行多年前为 8086 处理器编写的软件。那么 32 位的 IA-32 处理器是如何执行 16 位的 8086/80286 程序的呢？要回答这个问题，就要了解 CPU 的操作模式（Operation Mode）。可以把操作模式理解为 CPU 的工作方式，在不同的操作模式下 CPU 按照不同的方式来工作，目的是可以执行不同种类的程序、完成不同的任务。迄今为止，IA-32 处理器定义了图 2-2 所示的 5 种操作模式，分别介绍如下。

保护模式（Protected Mode）：所有 IA-32 处理器的本位（native）模式，具有强大的虚拟内存支持和完善的任务保护机制，为现代操作系统提供了良好的多任务（multitasking）运行环境。2.4 节和 2.5 节将进一步介绍与保护模式有关的重要概念。

实地址模式（Real-address Mode）：简称实模式（Real Mode），即模拟 8086 处理器的工作模式。工作在此模式下的 IA-32 处理器相当于高速的 8086 处理器。实模式提供了一种简单的单任务环境，可以直接访问物理内存和 I/O 空间，操作系统和应用软件运行在同一个内存空间中和同一个优先级上，因此操作系统的数据很容易被应用软件所破坏。DOS 操作系统运行在实模式下。CPU 在上电或复位后总是处于实模式状态。

虚拟 8086 模式（Virtual-8086 Mode）：保护模式下用来执行 8086 任务（程序）的准模式（quasi-operating mode）。通过该模式，可以把 8086 程序当作保护模式的一

项任务来执行。实地址模式无疑为运行 8086 程序提供了良好的硬件环境，但由于实地址模式无法运行现代的主流操作系统，从保护模式切换到实模式来运行 8086 程序需要较大的开销，难以实现。虚拟 8086 模式允许在不退出保护模式的情况下执行 8086 程序，当 CPU 切换到一个 8086 任务时，它便以类似实模式的方式工作，当 CPU 被切换到其他普通 32 位任务时，仍然以正常的方式工作，这样就可以在一个操作系统下“同时”运行 8086 任务和普通的 32 位任务了。需要注意的是，运行在虚拟 8086 模式下的 8086 任务在 I/O 访问方面会受到一些限制，与运行在实模式下是有所不同的，但这是为了保证操作系统和其他任务的安全所必需的。

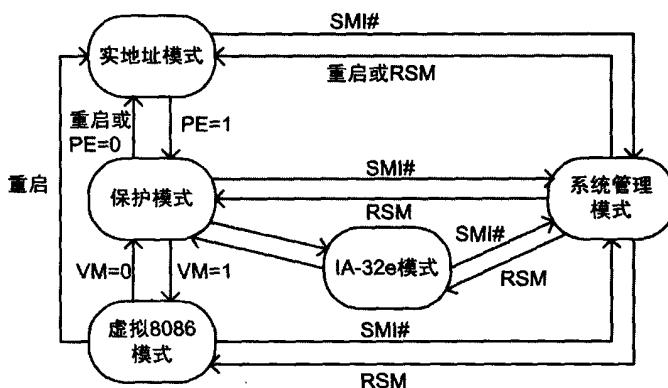


图 2-2 CPU 的操作模式 (摘自 IA-32 手册卷 3A)

系统管理模式 (System Management Mode, SMM): 供系统固件 (firmware) 执行电源管理、安全检查或与平台相关的特殊任务。当 CPU 的系统管理中断管脚 (SMI#) 被激活时，处理器会将当前正在执行的任务的上下文保存起来，然后切换到另一个单独的 (separate) 地址空间中执行专门的 SMM 例程。SMM 例程通过 RSM 指令使处理器退出 SMM 模式并恢复到响应系统管理中断前的状态。386 SL 处理器最先引入系统管理模式，其后的所有 IA-32 处理器都支持该模式。

IA-32e 模式: 支持 Intel64 的 64 位工作模式，曾经被称为 64 位内存扩展技术 (Extended Memory 64 Technology, 简称 EM64T)，是 IA-32 CPU 支持 64 位的一种扩展技术，具有对现有 32 位程序的良好兼容性。由两个子模式组成：64 位模式和兼容模式。64 位模式提供了 64 位的线性寻址能力，并能够访问超过 64GB 的物理内存 (32 位下启用 PAE 功能后最多访问 64GB 物理内存)。兼容模式用于执行现有的 32 位应用程序，使它们可以不做任何改动就可以运行在 64 位操作系统上。运行在 IA-32e 模式下的 64 位操作系统，系统内核和内核态的驱动程序一定是 64 位的代码，工作在 64 位模式下，应用程序可以是 32 位的 (在兼容模式下执行)，也可以是 64 位的 (在 64 位模式下执行)。本书讨论的情况除了特别说明外不包括 IA-32e 模式。

处理器在上电开始运行时或复位后是处于实地址模式的，CR0 控制寄存器的 PE

(Protection Enable) 标志用来控制处理器处于实地址模式还是保护模式。标志寄存器(EFlags)的 VM 标志用来控制处理器是在虚拟 8086 模式还是普通保护模式下, EFER 寄存器(Extended-Feature-Enable Register)的 LME(Long Mode Enable)用来启用 IA-32e 模式。关于模式切换的细节,感兴趣的读者可以参阅 IA-32 手册 卷 3A 第 9 章中的模式切换 (Mode Switching) 一节。

2.4 寄存器

寄存器 (registers) 是位于 CPU 内部的高速存储单元, 用来临时存放计算过程中用到的操作数、结果、程序指针或其他信息。CPU 可以直接操作寄存器中的值, 因此访问寄存器的速度比访问内存要快得多。

通用数据寄存器的宽度 (size) 决定了 CPU 可以直接表示的数据范围。比如 32 位的数据寄存器可以直接表示的最大整数值为 $2^{32}-1$, 这也意味着采用 32 位寄存器的 CPU 单次计算支持的最大整数位数是 32 位。因此, 寄存器的宽度和寄存器的个数的多少是 CPU 的最基本指标。我们通常所说的 CPU 位数, 比如 16 位 CPU, 32 位 CPU 或 64 位 CPU, 指的就是 CPU 中通用寄存器的位数 (宽度)。

与 RISC CPU 相比, CISC CPU 的通用寄存器数量是比较少的。x86 CPU 定义的用于程序执行的基本寄存器共有 16 个: 包括 8 个通用寄存器、6 个段寄存器、1 个标志寄存器和 1 个程序指针寄存器 (EIP)。随着 CPU 功能的增加, IA-32 CPU 逐渐加入了控制寄存器、调试寄存器、用于浮点和向量计算的向量寄存器、性能监视的寄存器, 以及与 CPU 型号有关的 MSR 寄存器, 下面我们分别进行介绍。

2.4.1 通用数据寄存器

通用数据寄存器又称 GPR (General Purpose Register), 共有 8 个, 分别为 EAX、EBX、ECX、EDX、ESP、EBP、ESI 和 EDI, 每个的最大宽度是 32 位。E 代表 Extended (扩展), 因为这些寄存器的名字来源于 16 位的 x86 处理器 (8086、80286 等), 当时称为 AX、BX 等。其中 EAX、EBX、ECX 和 EDX 可以按字节 (比如 AL、AH、BL、BH 等) 或字 (比如 AX、BX 等) 来访问。

尽管 GPR 寄存器大多数时候是通用的, 可以用作任何用途, 但是在某些情况下, 它们也有特定的隐含用法。比如 ECX、ESI 和 EDI 在串循环操作中分别用作计数器、源和目标。EBP 和 ESP 主要用来维护堆栈, ESP (Extended Stack Pointer) 通常指向栈的顶部, EBP (Extended Base Pointer) 指向当前栈帧 (frame) 的起始地址 (base pointer 的名字由来)。值得注意的是, 在 x86 系统中, 栈是向下生长的, 因此当向栈中压入数据时, ESP 的值会减小, 而不是增大, 我们将在第 22 章中详细地介绍栈。

2.4.2 标志寄存器 EFLAGS

IA-32 CPU 有一个 32 位的标志寄存器，名为 EFLAGS（如图 2-3 所示）。标志寄存器的作用相当于大型机时代的监视控制面板。每个标志位相当于面板上的一个按钮或指示灯，分别用来切换 CPU 的工作参数或显示 CPU 的状态。

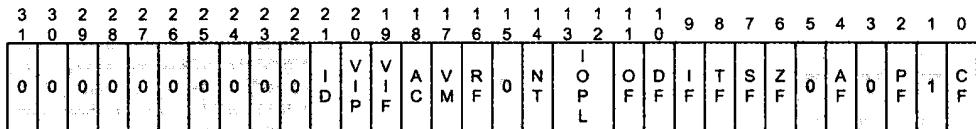


图 2-3 EFLAGS 寄存器

EFLAGS 寄存器包含了 3 类标志：用于报告算术指令（如 ADD、SUB、MUL、DIV 等）结果的状态标志（CF、PF、AF、ZF、SF、OF）；控制字符串指令操作方向的控制标志（DF）；供系统软件执行管理操作的系统标志。表 2-1 列出了每个标志位的含义。

表 2-1 EFLAGS 寄存器中各个标志位的含义

标志	位	意义
CF (Carry Flag)	0	进位或借位
PF (Parity Flag)	2	当计算结果的最低字节中包含偶数个 1 时，该标志为 1
AF (Adjust Flag)	4	辅助进位标志，当位 3（半个字节）处有进位或借位时该标志为 1
ZF (Zero Flag)	6	计算结果为 0 时，该标志为 1，否则为 0
SF (Sign Flag)	7	符号标志，结果为负时为 1，否则为 0
OF (Overflow Flag)	11	溢出标志，结果超出机器的表达范围时为 1，否则为 0
DF (Direction Flag)	10	方向标志，为 1 时使字符串指令每次操作后递减变址寄存器（ESI 和 EDI），为 0 时递增
TF (Trap flag)	8	陷阱标志，详见 4.3 节
IF (Interrupt enable Flag)	9	中断标志，为 0 时禁止响应可屏蔽中断，为 1 时打开
IOPL (I/O Privilege Level)	12 和 13	用于表示当前任务（程序）的 I/O 权限级别
NT (Nested Task flag)	14	任务嵌套标志，为 1 时表示当前任务是链接到前面执行的任务的，通常是由于中断或异常触发了 IDT 表中的任务门
RF (Resume Flag)	16	控制处理器对调试异常（#DB）的响应，为 1 时暂时禁止由于指令断点（是指通过调试寄存器设置的指令断点）导致的调试异常，详见 4.2.5 节

续表

标志	位	意义
VM (Virtual-8086 Mode flag)	17	为1时启用虚拟8086模式，清除该位返回到普通的保护模式
AC (Alignment Check flag)	18	设置此标志和CR0的AM标志可以启用内存对齐检查
VIF (Virtual Interrupt Flag)	19	与VIP标志一起用于实现奔腾处理器引入的虚拟中断机制
VIP (Virtual Interrupt Pending flag)	20	与VIF标志一起用于实现奔腾处理器引入的虚拟中断机制
ID (Identification flag)	21	用于检测是否支持CPUID指令，如果能够成功设置和清除该标志，则支持CPUID指令

其中，CF位可以由STC和CLC指令来设置和清除，DF位可以由STD和CLD指令来设置和清除，IF位可以通过STI和CLI指令来设置和清除（有权限要求），而其他大多数标志都是不可以直接设置和清除的。

2.4.3 MSR寄存器

MSR (Model Specific Register) 的本义是指这些寄存器与CPU型号有关，还没有正式纳入IA-32架构中，也有可能不会被以后的CPU所兼容。但尽管本义如此，某些MSR寄存器因为已经被多款CPU所广泛支持也已经逐渐成为IA-32架构的一部分，比如第6章将介绍的用于机器检查机制(MCA)的MSR寄存器。MSR寄存器的默认大小是64位，但是有些MSR的某些位保留不用。每个MSR寄存器除了具有一个简短的帮助记忆的代号外，还具有一个整数ID用作标识，有时也把MSR寄存器的ID称为该寄存器的地址。例如，用于控制IA-32e模式的EFER寄存器的地址是0xC0000080。

RDMSR指令用于读取MSR寄存器，首先应该将要读的MSR的ID放入ECX寄存器，然后执行RDMSR指令，如果操作成功，返回值会被放入EDX:EAX中（高32位在EDX中，低32位在EAX中）。WRMSR指令用来写MSR寄存器，也是先把要写的MSR的ID放入ECX寄存器，并把要写入的数据放入EDX:EAX寄存器中，然后执行WRMSR指令。

2.4.4 控制寄存器

IA-32 CPU设计了5个控制寄存器CR0~CR4(如图2-4所示)，用来决定CPU的操作模式以及当前任务的关键特征。其中CR0和CR4包含了很多与CPU工作模

式关系密切的重要标志位，详见表 2-2。CR1 自从 386 开始就一直保留未用。CR2 用来记录导致页错误异常的线性地址。CR3 又被称为页目录基地址寄存器（Page-Directory Base Register, PDBR），包含了页目录的基地址（物理地址）和两个用来控制页目录缓存（caching）的标志 PCD 和 PWT。页目录基地址的低 12 位被假定为 0，因此页目录所在的内存一定是按照页（4KB）边界对齐的。

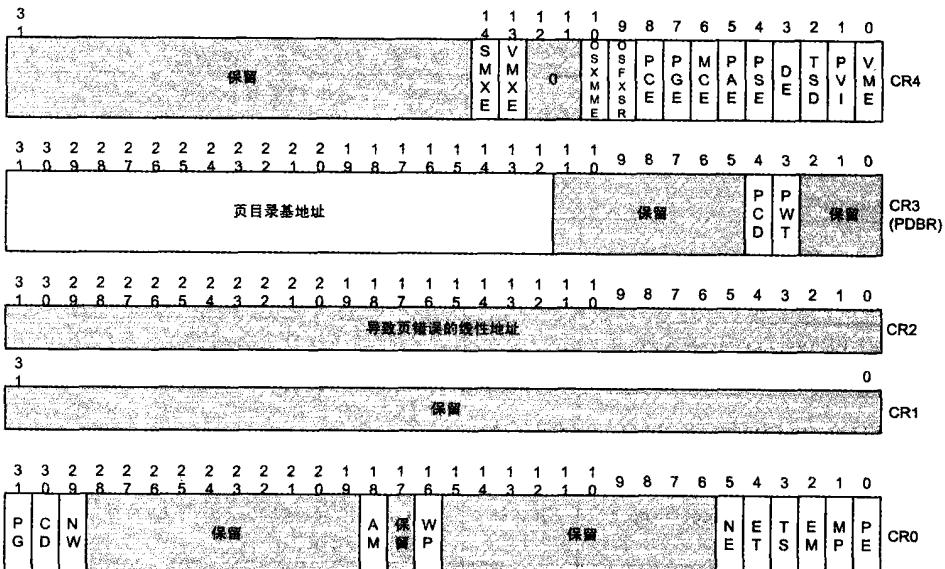


图 2-4 控制寄存器

表 2-2 控制寄存器中的标志位

标志	位	意义
PE (Protection Enable)	CR0[0]	为 1 时启用保护模式，为 0 时启用实地址模式
MP(Monitor Coprocessor)	CR0[1]	用来控制 WAIT/FWAIT 指令对 TS 标志的检查，详见 2.11 节有关设备不可用异常 (#NM) 的介绍
EM (Emulation)	CR0[2]	为 1 时表示使用软件来模拟浮点单元 (FPU) 进行浮点运算；为 0 时表示处理器具有内部的或外部的 FPU
TS (Task Switched)	CR0[3]	当 CPU 在每次切换任务时设置该位，在执行 x87 FPU 和 MMX/SSE/SSE2/SS3 指令时检查该位，主要用于支持在任务切换时延迟保存 x87 FPU 和 MMX/SSE/SSE2/SS3 上下文
ET (Extension Type)	CR0[4]	对于 386 和 486 的 CPU，为 1 时表示支持 387 数学协处理器指令，对于 486 以后的 IA-32 CPU，该位保留（固定为 1）
NE (Numeric Error)	CR0[5]	用来控制 x87 FPU 错误的报告方式，为 1 时启用内部的本位 (native) 机制，为 0 时启用与 DOS 兼容的 PC 方式

续表

标志	位	含义
WP (Write Protect)	CR0[16]	为1时，禁止内核级代码写用户级的只读内存页；为0时允许
AM (Alignment Mask)	CR0[18]	为1时启用自动内存对齐检查，为0时禁止
NW (Not Write-through)	CR0[29]	与CD标志共同控制高速缓存有关的选项
CD (Cache Disable)	CR0[30]	与NW标志共同控制高速缓存有关的选项
PG (Paging)	CR0[31]	为1时启用页机制(paging)，为0时禁止
PCD (Page-level Cache Disable)	CR3[3]	控制是否对当前页目录进行高速缓存(caching)，为1禁止，为0时允许
PWT (Page-level Writes Transparent)	CR3[4]	控制页目录的缓存方式，为1时启用write-through方式缓存；为0时启用write-back方式缓存
VME (Virtual-8086 Mode Extensions)	CR4[0]	为1时启用虚拟8086模式下的中断和异常处理扩展：将中断和异常重定向到8086程序的处理例程以减少调用虚拟8086监视程序(monitors)的开销
PVI (Protected-Mode Virtual Interrupts)	CR4[1]	为1时启用硬件支持的虚拟中断标志(VIF)，为0时禁止VIF标志
TSD (Time Stamp Disable)	CR4[2]	为1时只有在0特权级才能使用RDTSC指令，为0时所有特权级都可以使用该指令读取时间戳
DE (Debugging Extensions)	CR4[3]	为1时引用DR4和DR5寄存器将导致无效指令(#UD)异常，为0时引用DR4和DR5等价于引用DR6和DR7
PSE (Page Size Extensions)	CR4[4]	为1时启用4MB内存页，为0时限制内存页为4KB
PAE (Physical Address Extension)	CR4[5]	为1时支持36位或36位以上的物理内存地址，为0时限定物理地址为32位
MCE (Machine-Check Enable)	CR4[6]	为1时启用机器检查异常，为0时禁止
PGE (Page Global Enable)	CR4[7]	为1时启用P6处理器引入的全局页功能，为0时禁止
PCE (Performance-Monitoring Counter Enable)	CR4[8]	为1时允许所有特权级的代码都可以使用RDPMC指令读取性能计数器，为0时只有在0特权级才能使用RDPMC指令
OSFXSR (Operating System Support for FXSAVE and FXRSTOR instructions)	CR4[9]	操作系统使用，表示操作系统对FXSAVE、FXRSTOR及SSE/SSE2/SSE3指令的支持，以保证较老的操作系统仍然可以运行在较新的CPU上
OSXMMEXCPT (Operating System Support for Unmasked SIMD Floating-Point Exceptions)	CR4[10]	操作系统使用，表示操作系统对奔腾III处理器引入的SIMD浮点异常(#XF)的支持。如果该位为0表示操作系统不支持#XF异常，那么CPU会通过无效指令异常(#UD)来报告#XF异常，以防止针对奔腾III以前处理器设计的操作系统在奔腾III或更新的CPU上运行时出错

MOV CRn 命令用来读写控制寄存器的内容，只有在 0 特权级才能执行这个指令。

2.4.5 其他寄存器

除了上面介绍的寄存器，IA-32 CPU 还有如下寄存器。

CS、DS、SS、ES、FS 和 GS 是 6 个 16 位的段寄存器，当 CPU 工作在实模式下时，其内容代表的是段地址的高 16 位，也就是将其乘以 16（或左移 4 位，或者说将十六进制表示的值末位加 0）便可以得到该段的基址。例如，如果 ES=2000H，那么指令 MOV AL, ES:[100H] 就是把地址 $2000H * 10H + 100H = 20100H$ 处的一个字节放入 AL 寄存器中。在保护模式下，段寄存器内存放的是段选择子，详见 2.6 节对保护模式的介绍。

1 个 32 位的程序指针寄存器 EIP (Extended Instruction Pointer)，指向的是 CPU 要执行的下一条指令，其值为该指令在当前代码段中的偏移地址。如果一条指令有多个字节，那么 EIP 指向的是该指令的第一个字节。

8 个 128 位的向量运算寄存器 XMM0~XMM7，供 SSE/SSE2/SSE3 指令使用以支持对单精度浮点数进行 SIMD 计算。

8 个 80 位的 FPU 和 MMX 两用寄存器 ST0~ST7，当执行 MMX 指令时，其中的低 64 位用作 MMX 数据寄存器 MM0~MM7；当执行 x87 浮点指令时，它们被用作浮点数据寄存器 R0~R7。

1 个 32 位的中断描述符表寄存器 IDTR，用于记录中断描述符表 (IDT) 的基地址和边界 (limit)，详见 3.5 节。

1 个 32 位的全局描述符表寄存器 GDTR，用于描述全局描述符表 (GDT) 的基地址和边界，详见 2.6 节。

1 个 16 位的局部描述符表 (LDT) 寄存器 LDTR，存放的是局部描述符表的选择子。

1 个 16 位的任务寄存器 TR，用于存放选取任务状态段 (Task State Segment，简称 TSS) 描述符的选择子。TSS 用来存放一个任务的状态信息，在多任务环境下，CPU 在从一个任务切换到另一个任务时，前一个任务的寄存器等状态被保存到 TSS 中。

1 个 64 位的时间戳计数器 (Time Stamp Counter，简称 TSC)，每个时钟周期其数值加 1，重启动时清零。RDTSC 指令用来读取 TSC 寄存器，但是只有当 CR4 寄存器的 TSD 位为 0 时，才可以在任何优先级执行该指令，否则，只有在最高优先级下 (级别 0) 才可以执行该指令。

内存类型范围寄存器 (Memory Type and Range Register，简称 MTRR)，定义了内存空间中各个区域的内存类型，CPU 据此知道相应内存区域的特征，比如是否可以对其做高速缓存等。

性能监视寄存器，我们将在第5章中讨论。

调试寄存器DR0~DR7，用于支持调试，我们将在第4章中讨论。

2.4.6 64位模式时的寄存器

当支持64位的IA-32 CPU工作在64位模式(IA-32e)时，所有通用寄存器和大多数其他寄存器都延展为64位(段寄存器始终为16位)，并可以使用RXX来引用它们，例如RAX、RBX、RCX、RFLAGS、RIP等。此外，64位模式增加了如下寄存器。

- 8个新的通用寄存器R8~R15，可以分别使用RnD、RnW、RnB(n=8~15)来引用这些寄存器的低32位、低16位或低8位。
- 8个新的SIMD寄存器XMM8~XMM15。
- 控制寄存器CR8，又称为任务优先级寄存器(Task Priority Register)。
- Extended-Feature-Enable Register(EFER)寄存器，用来启用扩展的CPU功能，作用与标志寄存器类似。

关于每个寄存器的细节，大家需要时可以参考IA-32手册的第1卷和第3卷。大多数调试器都提供了读取和修改寄存器的功能，比如在WinDBG中可以使用r命令来显示或修改普通寄存器，使用rdmsr和wrmsr命令来读取和编辑MSR寄存器。二者的工作原理是有所不同的，r命令操作的是在中断到调试器时被调试程序保存在内存中的寄存器上下文，而rdmsr和wrmsr操作的是CPU内部的物理寄存器。

2.5 理解保护模式

大多数现代操作系统(包括Windows9X/NT/XP和Linux等)都是多任务的，CPU的保护模式是操作系统实现多任务的基础。了解保护模式的底层原理对学习操作系统和本书后面的章节有着事半功倍的作用。

保护模式是为实现多任务而设计的，其名称中的“保护”就是保护多任务环境中的各个任务的安全。多任务环境的一个基本问题就是当多个任务同时运行时，如何保证一个任务不会受到其他任务的破坏，同时也不会破坏其他任务。也就是要实现多个任务在同一个系统中“和平共处、互不侵犯”。所谓任务，从CPU层来看就是CPU可以独立调度和执行的程序单位。从Windows操作系统的角度来看，一个任务就是一个线程(Thread)。

进一步来说，可以把保护模式对任务的保护机制划分为任务内的保护和任务间的保护。任务内的保护是指同一任务空间内不同级别的代码不会相互破坏。任务间的保护就是指一个任务不会破坏另外一个任务。简单来说，任务间的保护是靠内存映射机制(包括段映射和页映射)实现的，任务内的保护是靠特权级别检查实现的。下面分

别加以介绍。

2.5.1 任务间的保护机制

任务间保护主要是靠虚拟内存映射机制来实现的，即在保护模式下，每个任务都被置于一个虚拟内存空间之中，操作系统决定何时以及如何把这些虚拟内存映射到物理内存。例如在 Win32(泛指 Windows 的 32 位版本，例如 Windows 95/98, Windows XP, Windows NT 和 Windows Server 2003 等)下，每个任务都被赋予 4GB 的虚拟内存空间，可以用地址 0~0xFFFFFFFF 来访问这个空间中的任意地址。尽管不同任务可以访问相同的地址（比如 0x00401010），但因为这个地址仅仅是本任务空间中的虚拟地址，不同任务处于不同的虚拟空间中，不同任务的虚拟地址可以被映射到不同的物理地址，这样就可以很容易防止一个任务内的代码直接访问另一个任务的数据。IA-32 CPU 提供了两种机制来实现内存映射：段机制（Segmentation）和页机制（Paging），我们将在下一节做进一步介绍。

2.5.2 任务内的保护

任务内保护的主要目的是保护操作系统。

操作系统的代码和数据通常被映射到系统中每个任务的内存空间中，并且对于所有任务其地址是一样的。例如，在 Windows 系统中，操作系统的代码和数据通常被映射到每个进程的高 2GB 空间中。这意味着操作系统的空间对于应用程序是“可触及的”，应用程序中的指针可以指向操作系统所使用的内存。

任务内保护的核心思想是权限控制，即为代码和数据根据其重要性指定特权级别，高特权级的代码可以执行和访问低特权级的代码和数据，低特权级的代码不可以直接执行和访问高特权级的代码和数据。高特权级通常被赋给重要的数据和可信任的代码，比如操作系统的数据和代码。低特权级通常被赋给不重要的数据和不信任的代码，比如应用程序。这样，操作系统可以直接访问应用程序的代码和数据，而应用程序虽然可以指向系统的空间，但是不能访问，一旦访问就会被系统所发现并禁止。在 Windows 系统中，我们有时会看到图 2-5 所示的应用程序错误对话框，导致这种情况的一个典型原因就是应用程序有意或无意地访问了禁止访问的系统空间（Access Violation），而被系统发现。

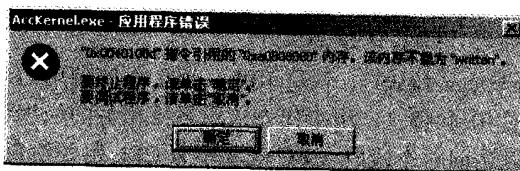


图 2-5 应用程序试图访问系统使用的内存时遭到系统禁止

清单 2-1 列出了导致图 2-5 所示错误的应用程序（AccKernel）的源代码。

清单 2-1 AccKernel 程序的源代码

```
int main(int argc, char* argv[])
{
    printf("Hi, I want to access kernel space!\n");
    *(int *)0xA0808080=0x22;

    printf("I would never reach so far!\n");
    return 0;
}
```

以上分析说明了应用程序尽管可以指向系统的内存，但是访问时会被系统发现并禁止。我们将在第 12 章中进一步讨论应用程序错误。

事实上，应用程序只能通过操作系统公开的接口（API）来使用操作系统的服务，即所谓的系统调用。系统调用相当于在系统代码和用户代码之间开了一扇有人看守的小门。我们将在第 8 章对此作进一步的介绍。

2.5.3 特权级

IA-32 处理器定义了 4 个特权级，又称为环（Ring），分别用 0、1、2、3 表示。0 代表的特权级最高，3 代表的特权级最低。最高的特权级通常是分配给操作系统的内核代码和数据的。比如 Windows 操作系统的内核模块是在特权级 0（Ring 0）运行的，Windows 下的各种应用程序（例如 MS Word、Excel 等）是在特权级 3 运行的。因为特权级 0 下运行的通常都是内核模块，所以人们便把在特权级 0 运行说成在内核模式（Kernel Mode）运行。把在特权级 3 运行说成在用户模式（User Mode）运行。并因此把编写内核模式下执行的程序称为内核模式编程，把为内核模式编写的驱动程序称为内核模式驱动程序等。

进一步说，处理器通过以下 3 种方式来记录和监控特权级别以实现特权控制。

- 描述符特权级别（Descriptor Privilege Level），简称 DPL，位于段描述符或门描述符中，用于表示一个段或门（Gate）的特权级别。
- 当前特权级别（Current Privilege Level），简称 CPL，位于 CS 和 SS 寄存器的位 0 和位 1 中，用于表示当前正在执行的程序或任务的特权级别。通常 CPL 等于当前正被执行的代码段的 DPL。当处理器切换到一个不同 DPL 的段时，CPL 会随之变化。但有一个例外，因为一致代码段（conforming code segment）可以被特权级别与其相等或更低（数值上大于或等于）的代码所访问，所以当 CPU 访问 DPL 大于 CPL（数值上）的一致代码段时，CPL 保持不变。
- 请求者特权级别（Requestor Privilege Level），简称 RPL，用于系统调用的情况，位于保存在栈中的段选择子的位 0 和位 1，用来代表请求系统服务的应用程序的特权级别。当 CPU 判断是否可以访问一个段时，CPU 既要检查 CPL 也要检查 RPL。这样做的目的是防止高特权级的代码代替应用程序访问应用程序本来没有权力访

问的段。举例来说，当应用程序调用操作系统的服务时，操作系统会检查保存在栈中的来自应用程序的段选择子的 RPL，确保它与应用程序代码段的特权级别一致，IA-32 CPU 专门设计了一条指令 ARPL（Adjust Requested Procedure Level）用来辅助这一检查。而后，当操作系统访问某个段时，系统会检查 RPL。此时如果只检查 CPL，那么因为正在执行的是操作系统的代码，所以 CPL 反映的不是真正发起访问者的特权级。

以访问数据段为例，当 CPU 要访问位于数据段中的操作数时，CPU 必须先把指向该数据段的段选择子加载到数据段寄存器（DS、ES、FS、GS）或栈段寄存器（SS）中。在 CPU 把一个段选择子加载到段寄存器之前，CPU 会进行特权检查。具体来说就是比较当前代码的 CPL（也就是当前正在执行的程序或任务的特权级）、RPL 和目标段的 DPL。仅当 CPL 和 RPL 数值上小于或等于 DPL 时，也就是说 CPL 和 RPL 对应的权限级别等于或大于 DPL 时，加载才会成功，否则便会抛出保护性异常。这样便保证了一段代码仅能访问与它同特权级或特权级比它低的数据。

访问不同特权级的代码段时的权限检查更为复杂，我们将在第 8 章讨论系统调用时略加介绍，有兴趣的读者请参考 IA-32 手册中有关门描述符、调用门和一致代码段等内容。

2.5.4 特权指令

为了防止低特权级的应用程序擅自修改权限等级和重要的系统数据结构，某些重要的指令只可以在最高特权级（Ring 0）下执行，这些指令被列为特权指令（Privileged Instructions）。表 2-3 列出了 IA-32 处理器目前定义的大多数特权指令。

表 2-3 特权指令列表

指令	功能
CLTS	清除CR0寄存器中的Task Switched标志
HLT	使CPU停止执行指令进入HALT状态，中断、调试异常、或BINIT#、RESET#等硬件信号可以使CPU脱离HALT状态
INVD	使高速缓存无效，不回写（不必把数据写回到主内存）
WBINVD	使高速缓存无效，回写（需要把数据写回到主内存）
INVLPG	使TLB表项无效
LGDT	加载GDTR寄存器
LIDT	加载IDTR寄存器
LLDT	加载LDTR寄存器
LMSW	加载机器状态字（Machine Status Word），也就是CR0寄存器的0到15位
LTR	加载任务寄存器（TR）
MOV to/from CRn	读取或为控制寄存器赋值
MOV to/from DRn	读取或为调试寄存器赋值
MOV to/from TRn	读取或为测试寄存器赋值，386手册介绍了测试寄存器TR6和TR7，用来测试TLB。最新的IA-32手册不再包含该内容
RDMSR	读MSR（Model-Specific Registers）寄存器
WRMSR	写MSR（Model-Specific Registers）寄存器

续表

RDPMC	读性能监控计数器，CR4寄存器的PCE标志为1可以允许所有特权级的代码都可以使用RDPMC指令
RDTSC	读时间戳计数器，CR4寄存器的TSD标志为0可以允许所有特权级的代码都可以使用RDTSC指令

本节围绕任务保护这一主题，介绍了保护模式的基本概念和实现保护的基本机制，包括特权级别和特权指令，这些机制对系统的安全运行起着重要作用。下一节我们将介绍保护模式下的内存管理。

2.6 段机制

内存是计算机系统的关键资源，程序必须被加载到内存中才可以被CPU所执行。程序运行的过程中，也要使用内存来记录数据和动态的信息。在一个多任务系统中，每个任务都需要使用内存资源，因此系统需要有一套机制来隔离不同任务所使用的内存。要使这种隔离既安全又高效，那么硬件一级的支持是必须的。IA-32 CPU提供了多种内存管理机制，这些机制为操作系统实现内存管理功能提供了硬件基础。

很多软件问题都是与内存有关的，深刻理解内存的使用规则对于软件调试是很重要的。本节和下一节将分别介绍IA-32 CPU的两种内存管理机制：段机制和页机制。

CPU的段机制（Segmentation）提供了一种手段可以将系统的内存空间划分为一个个较小的受保护区域，每个区域称为一个段（segment）。每个段都有自己的起始地址（基地址）、边界（Limit）和访问权限等属性。实现段机制的一个重要数据结构便是段描述符（Segment Descriptor）。

2.6.1 段描述符

在保护模式下每个内存段都有一个段描述符，这是其他代码访问该段的基本条件。每个段描述符是一个8字节长的数据结构，用来描述一个段的位置、大小、访问控制和状态等信息。图2-6画出了段描述符的通用格式。

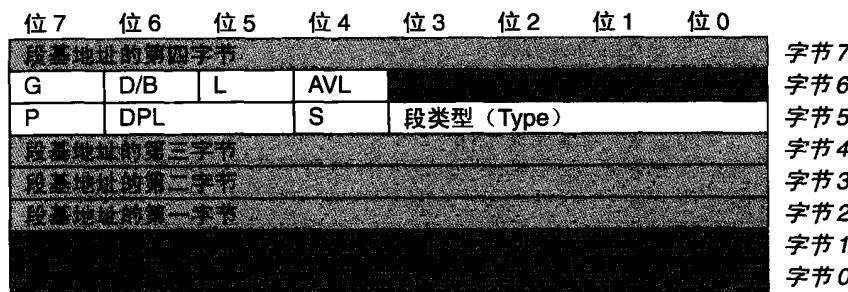


图2-6 段描述符的通用格式

段描述符的最基本内容是这个段的基地址和边界，基地址是以4个字节表示的（字节2、3、4和7），它可以是4GB线性地址空间中的任意地址（00000000~0xFFFFFFFF）。段边界是用20个比特位表示的（字节0、1和字节6的低四位），其单位由粒度（Granularity）位（字节6的最高位）决定，当G=0时，段边界的单位是1字节，当G=1时是4KB。因此一个段的最大边界值是 $(2^{20}-1)$ ，最大长度是 $2^{20}*4KB=4GB$ 。表2-4介绍了段描述符的其他各个域的含义。

表2-4 段描述符的各个域

S	系统（System）	S=0代表该描述符描述的是一个系统段，S=1代表该描述符描述的是代码段、数据段或堆栈段
P	存在（Present）	该位代表被描述的段是否在内存中，P=1表示该段已经在内存中，P=0表示该段不在内存中。内存管理软件可以使用该位来控制将哪些段实际加载到物理内存中。这为虚拟内存管理提供了页机制之外的另一种方法。事实上，最早支持保护模式的286CPU就没有分页机制
DPL	描述符特权级（Descriptor Privilege Level）	这两位定义了该段的特权级别（0~3），简单来说，仅当要访问该段的程序的特权级别（称为CPL）等于或高于这个段的级别时CPU才允许其访问，否则便会产生保护性异常（GPF）
D/B	Default/Big	对于代码段，该位表示的是这个代码段的默认位数（Default Bit）。D=0表示16位代码段，D=1表示32位代码段* 对于栈数据段，该位被称为B（Big）标志，B=1表示使用32位的堆栈指针（保存在ESP中），B=0表示使用16位堆栈指针（保存在SP中）
Type	段类型	位0简称A位，表示该段是否被访问过（Accessed），A=1表示被访问过。 ■ 对于数据/堆栈段，位2是扩展方向位，简称E（Expand）位，E=0表示向高端扩展，反之为1；位1是读写控制位简称W（Write）位，W=0表示该段只可以读，W=1表示可以读写 ■ 对于代码段，位2表示该段是否是一致（Conforming）代码段，简称C位。位1表示该段是否可读（Read），简称R位，R=1表示该段既可以执行，又可以读；R=0表示该段只可以执行，不可以读 位3是D/C位，决定了该段是数据/堆栈段（Data/stack）（C/D=0）还是代码段（Code）（C/D=1）。
L	64-bit代码段	用于描述IA-32e模式下的代码段，L=1表示代码段包含的是64位代码，L=0表示该段包含的兼容模式的代码
AVL	Available and reserved bits	供系统软件（操作系统）使用

*默认代码长度属性定义的是默认的地址和操作数长度，可以用地址大小前缀和操作数大小前缀改变默认长度。

值得说明的是，对于向下扩展（Expand-Down）的栈数据段，段边界（Limit）指定的是该段的最小偏移，B标志用来指定偏移的最大有效值（即上边界），当B=1时，最大偏移是0xFFFFFFFF，这样，如果Limit=0，那么段的总长度便是4GB（G=0），如果B=0，那么上边界便是0xFFFF。

2.6.2 描述符表

在一个多任务系统中通常会同时存在着很多个任务，每个任务会涉及多个段，每个段都需要一个段描述符，因此系统中会有很多段描述符。为了便于管理，系统用线性表来存放段描述符。根据用途不同，IA-32 处理器有 3 种描述符表：全局描述符表 GDT（Global Descriptor Table）、局部描述符表 LDT（Local Descriptor Table）和中断描述符表 IDT（Interrupt Descriptor Table）。

GDT 表是全局的，一个系统中通常只有一个 GDT 表，供系统中的所有程序和任务使用。LDT 与任务相关，每个任务可以有一个 LDT，也可以让多个任务共享一个 LDT。IDT 表的数量是和处理器的数量相关的，系统通常会为每个 CPU 建立一个 IDT 表。

GDTR 和 IDTR 寄存器分别用来标识 GDT 表和 IDT 表的位置和边界。这两个寄存器的格式是相同的，在 32 位模式下，长度是 48 位，高 32 位是基址，低 16 位是边界；在 IA-32e 模式下，长度是 80 位，高 64 位是基址，低 16 位是边界。

LGDT 和 SGDT 指令分别用来读取和设置 GDTR 寄存器。LIDT 和 SIDT 指令分别用来设置 IDTR 寄存器。操作系统在启动初期会建立 GDT 和 IDT 并初始化 GDTR 和 IDTR 寄存器。

位于 GDT 表中第一个表项（0 号）的描述符保留不用，被称为空描述符（null descriptor）。当把指向空描述符的段选择子加载到段寄存器时不会产生异常。

当创建 LDT 表时，GDT 已经准备好，因此，LDT 被创建为一种特殊的系统段，其段描述符被放在 GDT 表中。GDT 表本身只是一个数据结构，没有对应的段描述符。

使用 WinDBG 的 r 命令可以观察 GDTR 和 IDTR 寄存器的值，因为它们是 48 位的，所以应该分两次，分别读取它们的基地址和边界：

```
kd> r gdtr
gdtr=8003f000
kd> r idtr
idtr=8003f400
kd> r gdtl
gdtl=000003ff
kd> r idtl
idtl=000007ff
```

从上面的 gdtl 值可以看出这个 GDT 的边界是 1023，总长度是 1024 字节（1KB），共有 128 个表项。IDT 表的长度是 2KB，共有 256 个表项。

2.6.3 段选择子

局部描述符表寄存器 LDTR 表示当前任务的 LDT 表在 GDT 中的索引，其格式是典型的段选择子格式（见图 2-7）。

15 在描述表中的索引	3 2 1 0
	TI RPL

图 2-7 段选择子

段选择子的 TI 位代表要索引的段描述符表 (Table Indicator), TI=0 表示全局描述符表, TI=1 表示局部描述符表。

段选择子的高 13 位是描述符索引, 即要选择的段描述符在 TI 所表示的段描述符表中的索引号。因为这里使用的是 13 位, 意味着最多可索引 $2^{13} = 8\text{KB}$ 8192 个描述符, 所以 GDT 和 LDT 表的最大表项数都是 8192。因为 x86 CPU 最多支持 256 个中断向量, 所以 IDT 表的最多表项数是 256。

段选择子的低两位表示的是请求特权级 RPL (Requestor Privilege Level), 用于特权检查, 详细介绍见下文。

任务状态段寄存器 TR 中存放的也是一个段选择子, 指向的是全局段描述表 (GDT) 中描述当前任务状态段 (Task State Segmentation, 简称 TSS) 的段描述符。任务状态段是保存任务上下文信息的特殊段, 其基本长度是 104 字节, 操作系统可以附加更多内容。TSS 是实现任务切换的重要数据结构。当进行任务切换时, 处理器先把当前任务的执行现场——包括 CS:EIP 在内的寄存器保存到 TR 所指定的 TSS 中, 然后再把指向下一任务的 TSS 的选择子装入 TR (使用 LTR 指令), 接下来再从 TSS 中把下一任务的寄存器信息加载到各个寄存器中, 然后开始执行下一任务。

除了 LDTR 和 TR, 在保护模式下所有段寄存器 (CS、DS、ES、FS 和 GS) 中存放的也是段选择子, 不再是实模式时的高 16 位地址。

2.6.4 观察段寄存器

可以使用调试工具 (WinDBG 或 Visual Studio) 来观察段寄存器的值, 图 2-8 显示的是将记事本进程中断到调试器后所看到的情况。

15 在描述表中的索引	3 2 1 0
CS=0x001B	3 (0000000000011b)
DS=0x0023	4 (00000000000100b)
ES=0x0023	4 (00000000000100b)
FS=0x0038	7 (00000000000111b)
GS=0x0000	0 (00000000000000b)
SS=0x0023	4 (00000000000100b)

图 2-8 保护模式下的段寄存器内容示例

首先, 很容易看出这些段寄存器指向的都是全局段描述表 (GDT) 中的段描述符 (TI=0), GS 和 FS 的 RPL 是 0, 其他都是 3。

可以使用 WinDBG 的 dg 命令来显示一个段选择子所指向的段描述符的详细信息。例如，以下是将 CS 寄存器内的段选择子传递给 dg 命令而显示出的结果：

```
0:002> dg 1b
          P   Si   Gr   Pr   Lo
Sel     Base     Limit    Type   l   ze   an   es   ng   Flags
-----
001B 00000000 ffffffff Code RE Ac 3   Bg   Pg   P   Nl  00000cfb
```

其中 Sel 代表选择子（Selector），Base 和 Limit 分别是基地址和边界，Type 是段的类型，RE 代表只读（Read Only）和可以执行（Executable），Ac 代表被访问过。Type 后面的 Pl 代表特权级别（Privilege Level），数值 3 代表用户特权级，Size 代表代码的长度，Bg（Big）意味着 32 位代码，Gran 代表粒度，Pg 意味着粒度的单位是内存页（4KB），Pres 代表 Present，即这个段是否在内存中，因为 Windows 系统使用分页机制来实现虚拟内存，所以段的存在标志不再起重要作用，Long 下的 Nl 表示 Not Long，意味着这不是 64 位代码。

以下是 DS 和 ES 所代表描述符的信息：

```
0:000> dg 23
          P   Si   Gr   Pr   Lo
Sel     Base     Limit    Type   l   ze   an   es   ng   Flags
-----
0023 00000000 ffffffff Data RW Ac 3   Bg   Pg   P   Nl  00000cf3
```

因为 DS 和 ES 是用来选择数据段的，所以可以看到类型属性中有数据（Data）和读写（RW）。另外，值得注意的是以上两个段的基地址都是 0，边界都是 4GB-1，像这样将段的基地址设为 0，长度设为整个内存空间大小（4GB）的段使用方式被称为平坦模型（Flat Model）。

下面再观察一下 FS 寄存器所指向的段描述符：

```
0:001> dg 38
          P   Si   Gr   Pr   Lo
Sel     Base     Limit    Type   l   ze   an   es   ng   Flags
-----
0038 7ffdde000 00000fff Data RW Ac 3   Bg   By   P   Nl  000004f3
```

易见这个段的基地址不再为 0，边界也不是 4GB-1，而是 4KB-1（4095）。事实上，在 Windows 系统中，FS 段是用来存放当前线程的线程环境块，即 TEB 结构的。TEB 结构是在内核中创建，然后映射到用户空间的。

使用~命令列出线程的基本信息，可以看到线程 1 的 TEB 结构地址正是 FS 所代表段的基地址。

```
0:001> ~
0 Id: fd4.1e10 Suspend: 1 Teb: 7ffdf000 Unfrozen
1 Id: fd4.1294 Suspend: 1 Teb: 7ffdde000 Unfrozen
```

观察同一个 Windows 系统中的其他进程，我们会发现，其他进程的 CS、DS、ES、GS 寄存器的值和上面的一模一样的，这说明多个进程是共享 GDT 表中的段描述符的。这是因为使用了平坦模型后，大家的基地址、边界都一样，属性也一样，因此没有必要

建立多个。

除了上面介绍的代码段和数据段描述符 (S 位为 1)，另外一类重要的描述符是系统描述符 (S 位为 0)，包括描述 LDT 所在段的段描述符、描述 TSS 段的段描述符、调用门描述符、中断门描述符、陷阱门描述符和任务门描述符，后 4 种通称门描述符，11.1 节将作进一步介绍。

归纳一下，段机制使保护模式下的所有任务都在系统分配给它的段空间中执行。每个任务的代码（函数）和数据（变量）地址都是相对于它所在段的一个段内偏移。处理器根据段选择子在段描述符表 (LDT、GDT 或 IDT) 中找到该段的段描述符，然后再根据段描述符定位这个段。每个段具有自己的特权级别以实现对代码和数据的保护。

2.7 分页机制 (Paging)

从 386 开始，IA-32 处理器开始支持分页机制 (Paging)。分页机制的主要目的是高效地利用内存，按页来组织和管理内存空间，把暂时不用的数据放到外部存储器（通常是硬盘）上。在启用分页机制后，操作系统将线性地址空间划分为固定大小的页面 (4KB、2MB 或 4MB)。每个页面可以被映射到物理内存或外部存储器上的虚拟内存文件中。

当程序中的指令访问某一地址（虚拟地址）时，CPU 首先会根据段寄存器的内容将虚拟地址转化为线性地址，具体过程是根据段寄存器中的段选择子找到段描述符，然后将段描述符中的基址加上程序中的偏移地址便得到线性地址。接下来，如果 CPU 发现包含该线性地址的内存页不在物理内存中便会产生一个页错误异常 (#PF)。该异常的处理程序通常是操作系统的内存管理器例程。内存管理器得到异常报告后会根据异常的状态信息，特别是 CR2 寄存器中包含的线性地址，将所需的内存页加载到物理内存中。然后异常处理例程返回使处理器重新执行导致页错误异常的指令，这时所需的内存页已经在物理内存中，所以便不会再导致页错误异常了。

下面我们来看一下控制寄存器中 3 个与分页机制关系密切的标志。

- CR0 的 PG (Paging) 标志：用于启用分页机制，从 386 开始的所有 IA-32 处理器都支持该标志。
- CR4 的 PAE (Physical Address Extension) 标志：启用物理地址扩展，可以最多寻址 64GB 物理内存，否则最多寻址 4GB 物理内存。Pentium Pro 处理器引入此标志。
- CR4 的 PSE (Page Size Extension) 标志：用于启用大页面支持，当 PAE=1 时，大页面为 2MB，当 PAE=0 时，大页面为 4MB。奔腾处理器引入此标志。

CPU 和内存管理器使用页目录、页表和页目录指针表来管理从线性地址到物理地址的映射。

2.7.1 页目录 (Page Directory)

页目录是用来存放页目录表项 (Page-Directory Entry, 简称 PDE) 的线性表。每个页目录占一个 4KB 的内存页，每个 PDE 的长度为 32 个比特位 (4 字节)，因此每个页目录中最多包含 1024 个 PDE。当没有启用 PAE 时，一共有两种 PDE 格式，分别用于指向 4KB 的页表和 4MB 的内存页。图 2-9 画出了指向 4KB 页表的 PDE 的格式。图中高 20 位代表该 PDE 所指向页表的起始物理地址的高 20 位，该起始物理地址的低 12 位固定为 0，所以页表一定是按照 4KB 边界对齐的。

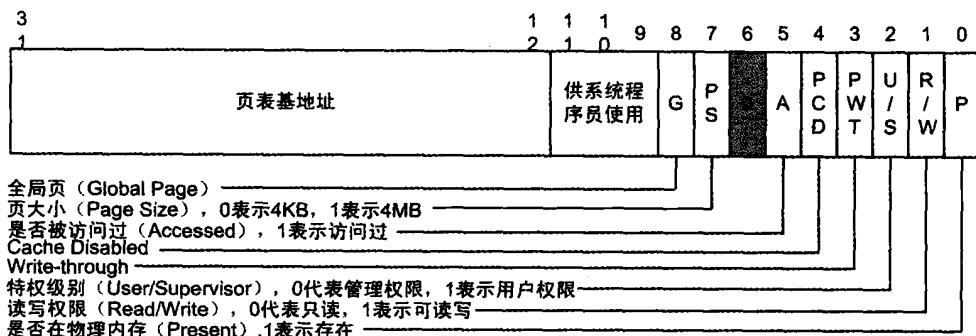


图 2-9 指向页表的页目录表项 (PDE) 的格式 (未启用 PAE)

图 2-10 画出了用于指向 4MB 内存页的 PDE 格式，高 10 位代表的是 4MB 内存页的起始物理地址的高 10 位，该起始物理地址的低 22 位固定为 0，因此 4MB 的内存页一定是按 4MB 进行边界对齐的。

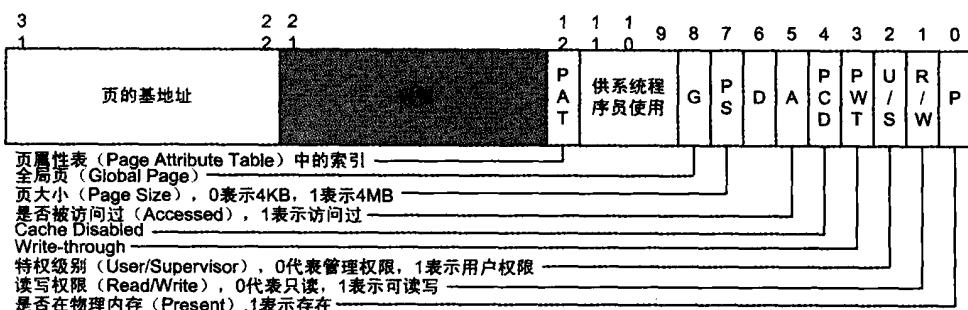


图 2-10 指向 4MB 内存页的页目录表项 (PDE) 的格式 (未启用 PAE)

2.7.2 页表 (Page Table)

页表是用来存放页表表项 (Page-Table Entry, 简称 PTE) 的线性表。每个页表占一个 4KB 的内存页，每个 PTE 的长度为 32 个比特位，因此每个页表中最多包含 1024 个 PTE。2MB 和 4MB 的大内存页是直接映射到页目录表项，不需要使用页表的。图 2-11 画出了 PTE 的具体格式，其中高 20 位代表的是 4KB 内存页的起始物理地址的高 20 位，该起始物理地址的低 12 位假定为 0，所以 4KB 内存页都是按 4KB 进行边界对齐的。

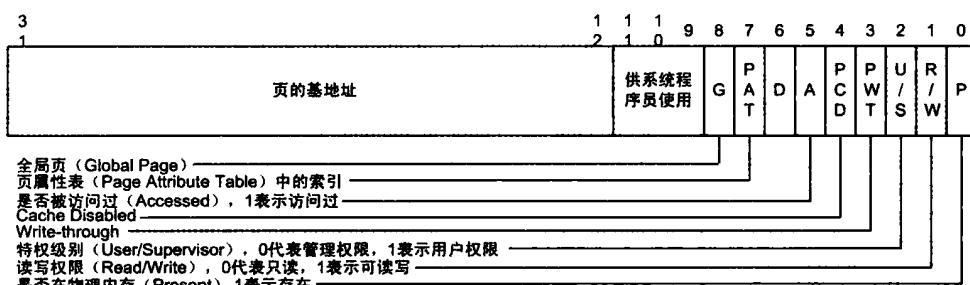


图 2-11 页表表项 (PTE) 的格式 (未启用 PAE)

2.7.3 页目录指针表 (Page-Directory-Pointer Table)

页目录指针表包含 4 个 64 位的表项，每个表项指向一个页目录，仅用于启用 PAE 时。启用 PAE 后，每个 PDE 和 PTE 的长度都增大为 64 位，但包含的标志变化不大，其细节从略。

2.7.4 地址翻译

有了前面的基础后，下面来看一下 CPU 是如何利用页目录和页表等数据结构将一个 32 位的虚拟地址翻译为 32 位的物理地址的（没有启用 PAE 的情况）。其过程可以概括为如下步骤。

1. 通过 CR3 寄存器定位到页目录的起始地址，正因为如此，CR3 寄存器又被称为页目录基址寄存器 (PDBR)。
2. 取线性地址的高 10 位作为索引选取页目录的一个表项，也就是 PDE。
3. 根据 PDE 中的页表基址 (取 PDE 的高 20 位，低 12 位设为 0) 定位到页表。
4. 取线性地址的 12 位到 21 位 (共 10 位) 作为索引选取页表的一个表项，也就是 PTE。

5. 取出 PTE 中的内存页地址（取 PTE 的高 20 位，低 12 位设为 0）。
6. 取线性地址的低 12 位作为页中偏移与上一步的内存页地址相加便得到物理地址。

图 2-12 画出了将线性地址映射到物理内存的过程。

对于 4MB 的大内存页，PDE 的高 10 位便是 4MB 内存页的基地址的高 10 位，线性地址的低 22 位是页中偏移。

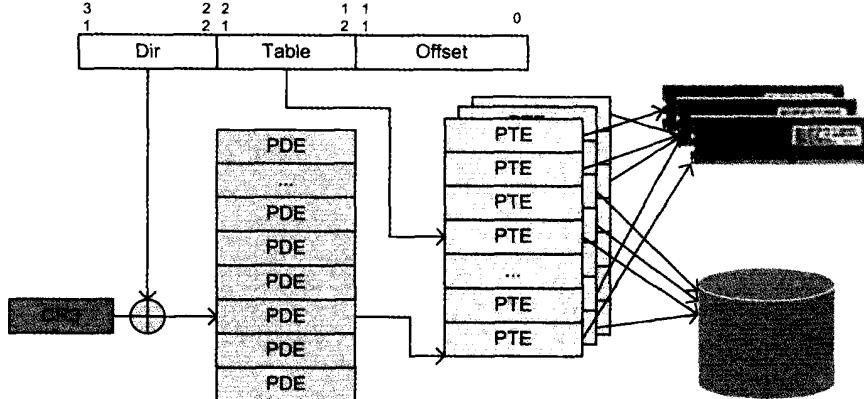


图 2-12 将线性地址映射到 4KB 的内存页

2.7.5 使用 WinDBG 观察分页机制

下面通过一个试验来加深大家对以上内容的理解，建议您在安装好 WinDBG 工具的 Windows XP SP1 或者 Windows Vista 系统上按照以下提示亲自做这个实验，如果在 Vista 上操作，那么应以调试方式启动，参见 18.8 节关于本地内核调试的介绍。

1. 启动计算器程序 (calc.exe)，键入一串数字（如 123456789）以便后面观察。
2. 启动 WinDBG，并附加到计算器程序上开始调试（选择 File>Attach to a process...）。
3. 在 WinDBG 的命令区输入 `x calc!gpszNum*` 命令列出计算器程序中以 g 开头的所有符号。注意其中包含的 `gpszNum` 行。

```
0:0014db0 calc!gpszNum = <no type information>
```

4. 在 WinDBG 的命令区输入 `dd calc!gpszNum 11` 命令，查看该符号地址的内容。

```
0:0014db0 000a6c88
```

5. 继续查看地址 000a6c88（应该换作您试验时看到的值）处的内容：`dd 000a6c88`

```
0:002> dd 000a6c88
000a6c88 00320031 00340033 00360035 00380037
000a6c98 00000039 00000000 00040004 000c0100
```

6. 看来 000a6c88 指向的很像是我们前面输入的数字。使用显示字符串命令进一步验证：

```
0:002> du 000a6c88
000a6c88 "123456789"
```

看来真是如此，尝试键入其他的数字，再重复第 4 到第 6 步，可以进一步验证。注意 `gpszNum11` 的值是会变化的，也就是它是指向一个动态分配的缓冲区的，该缓冲区包含了用户输入的字符串。

下面我们看一看如何把这个字符串地址（**000a6c88**）翻译为物理地址。

7. 先将这个虚拟地址转换为二进制格式，以便了解它的各个位域的值。这可以使用`.formats` 命令。

```
0:002> .formats 000a6c88
Evaluate expression:
Hex: 000a6c88
Decimal: 683144
Octal: 00002466210
Binary: 00000000 00001010 01101100 10001000
Chars: ..l.
Time: Fri Jan 09 05:45:44 1970
Float: low 9.57289e-040 high 0
Double: 3.37518e-318
```

根据图 2-12，虚拟地址 000a6c88 的页目录索引（高 10 位）为 0；页表索引（中间 10 位）为 001010 0110b，即 0xA6；页内偏移为（低 12 位）1100 10001000b，即 c88。

8. 再启动一个 WinDBG 实例，并开始本地内核调试以便观察计算器进程的页目录基址。以下所有操作都是在这个 WinDBG 中进行的。
9. 先通过`!process 0 0` 命令列出所有进程，并在其中找到关于 calc.exe 的内容。

```
1kd> !process 0 0
**** NT ACTIVE PROCESS DUMP ****
...
PROCESS 86474030 SessionId: 0 Cid: 0bf8 Peb: 7ffdf000 ParentCid: 0d98
DirBase: 3a744000 ObjectTable: e25acb08 HandleCount: 39.
Image: calc.exe
...
```

10. 上面的 DirBase 项描述的就是 calc 进程的页目录基地址的物理地址（CR3 寄存器内容）。也就是说，该进程的内存页目录表的基地址是 0x3a744000。

11. 使用`!dd` 命令（显示物理内存，注意有！号）显示页目录表的内容。

```
1kd> !dd 3a744000
#3a744000 15f93067 1b4c8067 0c8ec067 1b1e7067
#3a744010 2c792067 00000000 00000000 00000000
#3a744020 00000000 00000000 00000000 00000000
...
```

12. 页目录的每一项是 4 个字节，因为我们要转换的地址的页目录索引为 0，也就是第一个表项。其内容为 15f93067。根据图 2-11，其高 20 位为页表起始地址的高 20 位，即我们需要的地址的页表基地址是 0x15f93000；低 12 位（067）代表的是页表属性，将 0x67 转换为二进制 0000 0110 0111b，便得到其各位由低到高的含义如下。

- 位 0, Present 位，1 表示该内存页在物理内存中；
- 位 1, R/W 位，即读写权限，1 表示可读可写；

- 位2, U/S位, 即用户还是系统权限, 1表示用户权限;
 - 位3, Page level Write Through位, 用于控制高速缓存(write-back还是write-through)策略, 1表示write-through;
 - 位4, Page level cache disable(禁止页级缓存)位, 0表示没有禁止缓存该页;
 - 位5, A(Accessed)位, 内存管理器在把内存页加载到物理内存后, 通常会清除此位, 当有访问发生时再设置此位, 0表示加载到物理内存后, 还没有被访问过;
 - 位6, D(Dirty)位, 1表示该页被写过;
 - 位7, PS(Page size)位, 即页大小, 0表示4KB, 1表示4MB或2MB(如果启用了扩展物理寻址(PAE)功能);
 - 位8, G位, 即是否为全局页, 0表示不是全局页, 全局页是Pentium Pro引入的功能, 如果某个内存页被标记为全局页, 而且CR4的PGE标志为1时, 那么当CR3寄存器内容变化或任务切换时, TLB中用于全局页的页表和页目录表项不会失效;
 - 位9~11, 供内存管理软件(操作系统的内存管理器)使用;
13. 继续使用!dd命令观察页表, 因为页表索引是0xA6, 每个页表表项的长度是4字节, 所以应该观察物理地址15f93000+0xA6*4。

```
1kd> !dd 15f93000+A6*4
#15f93298 3a6ae067 00000000 00000000 00000000
```

也就是说, 页目录项的内容为3a6ae067, 根据图2-9, 其含义如下。

高20位为所在内存页的起始地址的高20位, 即目标地址所在内存页的基地址是0x3a6ae000; 低12位(067)代表的是内存页的属性。

14. 得到了页的基址后, 加上页内偏移(0xc8)便是最终的物理地址了。综合以上结果, 线性地址000a6c88的物理地址是3a6aec88。

```
1kd> !dd 3a6aec88
#3a6aec88 00320031 00340033 00360035 00380037
#3a6aec89 00000039 00000000 00040004 000c0100
#3a6eca8 00000001 00000001 00000000 00000001
```

易见物理地址3a6aec88处的内容与我们在第5步看到的虚拟地址000a6c88处的内容是完全一致的。

2.7.6 4MB内存页的情况

下面再通过一个实例看一下4MB内存页的情况。我们以系统进程中的线性地址0x94a0d678为例。

首先依然使用.process命令找到进程结构的地址和页目录的地址:

```
0: kd> .process
Implicit process is now 82a34d90
0: kd> !process 82a34d90 0
PROCESS 82a34d90 SessionId: none Cid: 0004 Peb: 00000000 ParentCid: 0000
```

```
DirBase: 00122000 ObjectTable: 85c00298 HandleCount: 404.
Image: System
```

然后取线性地址的高 10 位得到 PDE 在页目录表中的索引，即 $94a/4$ ，然后再乘以 4，得到偏移值，再加上页目录的基址便得到 PDE 的物理地址。

```
0: kd> !dd 00122000+94a/4*4
# 122948 d6c009eb d70009eb d74009eb d78009eb
```

也就是说，线性地址 0x94a0d678 对应的 PDE 等于 d6c009eb，使用 .formats 命令显示其细节：

```
0: kd> .formats d6c009eb
Evaluate expression:
  Hex:      d6c009eb
  Decimal: -692057621
  Octal:   32660004753
  Binary:  11010110 11000000 00001001 11101011
```

参考图 2-10，位 7 的 1 代表的是 4MB 内存页。PDE 的高 12 位便是物理地址的高 12 位，因此将 PDE 的高 12 位加上线性地址的低 24 位便得到了我们要找的物理地址，即 $d6c00000+20d678$ ，使用 !dd 和 dd 命令可以进行验证：

```
0: kd> !dd d6c00000+20d678
#d6e0d678 04000000 02000001 0040000e 0c000000
0: kd> dd 94a0d678
94a0d678 04000000 02000001 0040000e 0c000000
```

可见这两个地址的内容是一样的。

2.7.7 WinDBG 的有关命令

以上我们介绍了如何使用手工方法来直接观察页目录表和页表并翻译内存地址。除了手工方法，也可以使用调试器的命令来完成这些任务，比如 WinDBG 提供了以下几个命令。

- dg：显示段选择子所指向的段描述符的信息。
- !pte：显示出页目录和页表的地址。
- !vtop：将虚拟地址翻译为物理地址。
- !vpdd：显示物理地址、虚拟地址和内存的内容。
- !ptov：显示指定进程中所有物理内存到虚拟内存之间的映射。
- !sysptes：显示系统的页目录表项。

当程序执行时，是 CPU 内部的内存管理单元（Memory Management Unit，MMU）负责将线性地址翻译为物理地址。页表和页目录位于内存中。为了减少当翻译地址时访问页表和页目录所造成的开销，CPU 会把最近使用的页表和页目录表项存储在 CPU 内的专用高速缓存中，该缓存被称为译址旁视缓存（Translation Lookaside Buffer，简称 TLB）。有了 TLB，大多数对页目录和页表的访问请求都可以从 TLB 中读取，这大

大提高了地址翻译的速度。

启用分页机制不是进入保护模式的必要条件（前面讲的分段机制是保护模式所必需的）。而且是否启用分页机制也不会影响段管理模式，只是增加了一级映射，系统要根据页表将分段机制形成的线性地址转换为物理地址。如果不启用分页机制（对于 286 根本没有分页机制），那么段机制形成的线性地址就是物理地址。

从图 2-10、图 2-11、图 2-12 中我们可以看到 PDE 和 PTE 中包含的很多标志与段描述符中的标志很类似，如访问标志、读写标志、存在标志和 AVL 标志，表 2-5 归纳了这些标志的含义。

表 2-5 段描述符和 PTE/PDE 中的相似标志

段描述符	PTE 或 PDE	描述
P 标志	P 标志	存在标志
Type 中的 A 标志	A 标志	是否被访问过
Type 中的 R 和 W 标志	R/W	读写控制
AVL	Avail	留给系统软件使用
DPL	U/S	特权级别

这里再讨论一下 PDE 和 PTE 中的 U/S 标志与段描述符中的 DPL 标志的相似性。U/S 标志代表一个（PTE）或一组（PDE）内存页的特权级别。U/S 标志为 0 时代表的是管理特权（supervisor privilege level），U/S 标志为 1 代表的是用户特权级（user privilege level）。DPL 代表的是该描述符所描述的段的特权级别，0 最高，3 最低。尽管 DPL 有 4 个值，但是实际上被使用的主要是 0（系统特权级）和 3（用户特权级）。因此可以说 U/S 标志与 DPL 标志具有同样的作用。

看到这里读者可能会问，为什么要在分段和分页两种机制中重复定义这些标志呢？对于大多数 IA-32 系统，段机制和页机制都是同时启用的，会不会因为两个地方都要检查这些标志而影响性能呢？简单的回答是为了兼容。段机制是 x86 处理器与生俱来的重要特征之一，从第一代 8086 CPU 开始所有 x86 处理器都保持着该特征。尽管今天看来，有了分页机制后，分段机制已经变得越来越不重要了，但是为了兼容性和权限管理，还必须保留分段机制。可以通过若干措施来淡化分段机制的影响和作用，比如将段的基址设为 0，大小设为 4GB，这样一个任务的整个地址空间便都在一个段中了，从效果上相当于取消了分段。在 IA-32e（64 位）模式下，段描述符的基地址和边界值有时会被忽略，但是其中的某些标志和特权级别仍然是被使用的。

2.8 系统概貌

前面几节对 CPU 的内部特征、功能和发展做了初步介绍。本节从计算机系统的角度简要描述 CPU 是如何与其他部件交互的。尽管本节的概念也适用于某些服务器和工

工作站系统，但是我们以配备 IA-32 处理器的常见 PC（个人计算机）系统为例。

图 2-13 粗略勾勒出了一个典型 PC 系统的主要部件和连接方式。从上而下，CPU 通过前端总线（Front Side Bus）与内存控制器（Memory Controller Hub，简称 MCH）相连接。在多核技术出现之前，大多数 PC 系统通常只配备一个 CPU，多核技术使一个 CPU 外壳内包含多个 CPU 内核。比如，英特尔的奔腾 D 950（D 代表 Dual，即双内核）CPU 内部就包含两个完整的处理器内核，每个内核有自己的寄存器和高速缓存。因此随着多核技术的普及，越来越多的 PC 系统将是多 CPU 的。目前的前端总线设计是每个总线上最多有 4 个 CPU，如果要支持更多的 CPU，那么可以通过 Cluster Bridge 将多个前端总线连接在一起。

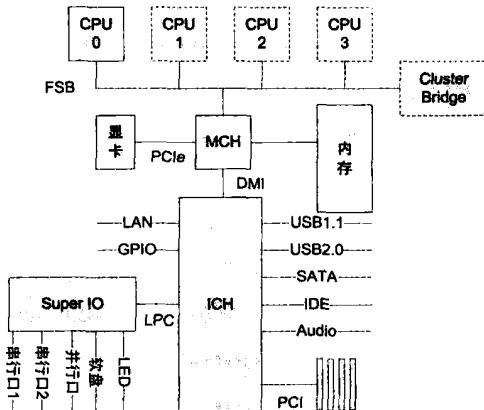


图 2-13 前端总线和 PC 系统示意图

MCH 上除了有内存接口外，通常还有显示卡接口，比如 AGP(Accelerated Graphics Port，即图形加速端口)或 PCI Express 16 x 接口。MCH 的下面是输入输出控制器(I/O Controller Hub，简称 ICH)。ICH 集成了用以和外部设备进行通信的各种接口，如连接 USB 设备的 USB 接口(USB1.1 和 USB2.0)；连接普通硬盘的 IDE 接口(即 PATA 接口)；连接 SATA 硬盘的 SATA(Serial ATA)接口；连接 BIOS 芯片的 SPI(Serial Peripheral Interface) 接口，等等。此外，ICH 还提供了对一些通用总线的支持，比如 I2C(Inter-Integrated Circuit)总线、LPC(Low Pin Count)总线和 PCI(Peripheral Component Interconnect)总线等。例如，通过 LPC 总线与 ICH 相连接的 Super IO 芯片(比如 LPC47m172)上集结了很多小数据量的外部设备，包括串口、并口、PS/2 键盘鼠标和各种 LED 指示灯等。ICH 内部通常还包含集成的网卡和声卡(AC'97 或 HD Audio，HD 是 High Definition 的缩写)。

MCH 和 ICH 就好像两个桥梁将整个系统联系起来。因此人们通常又把它们分别称为北桥和南桥。北桥和南桥是计算机主板上的最重要的芯片，经常被统称为芯片组(chipset)。MCH 和 ICH 之间是通过专用的被称为 DMI(Direct Media Interface)的高速接口相连接的。

2.9 本章总结

很多软件工程师的一个弱点是对硬件了解得太少，甚至不愿意去学习硬件知识，事实上，了解必要的硬件知识对理解软件经常会有事半功倍的效果，扎实的硬件基础对于软件工程师来说也是非常重要的。

本章首先介绍了指令集和指令的执行过程（2.1节），而后介绍了IA-32处理器的发展历程和主要功能（2.2节）。2.3节介绍了CPU的操作模式和每种操作模式的基本特征和用途。2.4节介绍了IA-32 CPU的寄存器。2.5节介绍了保护模式的内涵和主要保护机制。2.6节和2.7节详细介绍了保护模式下的内存管理机制。2.8节介绍了个人计算机系统的基本架构。

虽然本章的部分内容与软件调试没有直接的关系，但是这些内容对于理解计算机系统的底层原理和进行系统级调试有着重要意义，是成为软件调试高手必须掌握的基础内容。下一章我们将介绍CPU的中断和异常机制。

参考文献

1. Jack Doweck. Inside Intel® Core™ Microarchitecture and Smart Memory Access: An In-Depth Look at Intel Innovations for Accelerating Execution of Memory-Related Instructions. Intel Corporation, 2006
<http://download.intel.com/technology/architecture/sma.pdf>
2. 毛德操, 胡希明著. 嵌入式系统——采用公开源码和 StrongARM/Xscale 处理器. 杭州: 浙江大学出版社, 2003
3. IA-32 Intel® Architecture Software Developer's Manual Volume 1. Intel Corporation
4. IA-32 Intel® Architecture Software Developer's Manual Volume 2A. Intel Corporation
5. IA-32 Intel® Architecture Software Developer's Manual Volume 2B. Intel Corporation
6. IA-32 Intel® Architecture Software Developer's Manual Volume 3. Intel Corporation
7. INTEL 80386 PROGRAMMER'S REFERENCE MANUAL. Intel Corporation
8. Tom Shanley. The Unabridged Pentium 4: IA-32 Processor Genealogy. Addison Wesley, 2004
9. P6 Family of Processors: Hardware Developer's Manual. Intel Corporation
<http://www.intel.com/design/pentiumII/manuals/244001.htm>

中断和异常

当形容一个人固执不知变通时，人们会说他“死心眼，顺着一条路跑到天黑”，用这句话来描述 CPU 也非常恰当。因为无论把 CPU 的指令指针（IP）指向哪个内存地址，它都会试图执行那里的指令，执行完一条，再取下一条执行，如此往复。为了让 CPU 能够暂时停下当前的任务，转去处理突发事件或其他需要处理的任务，人们设计了中断（interrupt）和异常（exception）机制。

在计算机系统中，包括任务切换、时间更新、软件调试在内的很多功能都是依靠中断和异常机制实现的。毫不夸张地说，中断和异常是计算机原理中最重要的概念之一，充分理解中断和异常是理解 CPU 和系统软件的关键。

3.1 概念和差异

本节将先介绍中断和异常的概念，了解其基本特征，然后比较它们之间有什么不同。

3.1.1 中断

中断通常是由 CPU 外部的输入输出设备（硬件）所触发的，供外部设备通知 CPU “有事情需要处理”，因此又叫中断请求（Interrupt Request）。中断请求的目的是希望 CPU 暂时停止执行目前正在执行的程序，转去执行中断请求所对应的中断处理例程（Interrupt Service Routine，简称 ISR）。

考虑到有些任务是不可打断的，为了防止 CPU 这时也被打扰，可以通过执行 CLI（clear interrupt）指令清除标志寄存器的 IF 位，以使 CPU 暂时“不受打扰”。但有个例外，这样做只能使 CPU 不受可屏蔽中断（maskable interrupt）的打扰，一旦有不可屏蔽中断（Non-Maskable Interrupt，简称为 NMI）发生时，CPU 仍要立即转去处理。不过因为 NMI 中断通常很少发生，而且不可打断代码通常也比较短，所以大多数情况下，还是不存在问题的。可屏蔽中断请求信号通常是通过 CPU 的 INTR 引脚发给 CPU 的，不可屏蔽中断信号通常是通过 NMI 引脚发给 CPU 的。

中断机制为 CPU 和外部设备间的通信提供了一种高效的方法，有了中断机制，

CPU 就可以不用去频繁地查询外部设备的状态了，因为当外部设备有“事”需要 CPU 处理时，它可以发出中断请求通知 CPU。但是如果有太多的设备都向 CPU 发请求，那么也会导致 CPU 频繁地在各个中断处理例程间“奔波”，从而影响正常程序的执行。这好比我们通常仅仅把手机号码公开给熟悉的人，不然就可能会被频繁的来电“中断”正常的工作。从这个意义上讲，中断是计算机系统中非常宝贵的资源。如果为某个设备分配了中断资源，那么便赋予了它随时打断 CPU 的权力。

在硬件级，中断是由一块专门芯片来管理的，通常称为中断控制器（Interrupt Controller）。它负责分配中断资源和管理各个中断源发出的中断请求。为了便于标识各个中断请求，中断管理器通常用 IRQ（Interrupt ReQuest）后面加上数字来表示不同路（line）的中断请求信号，比如 IRQ0、IRQ1 等。根据从最初的个人计算机（IBM PC）系统传承下来的约定，IRQ0 通常是分配给系统时钟的；IRQ1 通常是分配给键盘的；IRQ3 和 IRQ4 通常是分配给串口 1 和串口 2 的；IRQ6 通常是分配给软盘驱动器的。

图 3-1 显示了笔者使用的系统中各个中断请求的分配情况。从图中可以看出，IRQ0、1、4 和 6 的用途仍然与最初 PC 系统的用途是一致的。IRQ9 则是由很多个设备所共享的，这是因为 PCI 总线标准支持多个 PCI 设备共用一个中断请求信号。

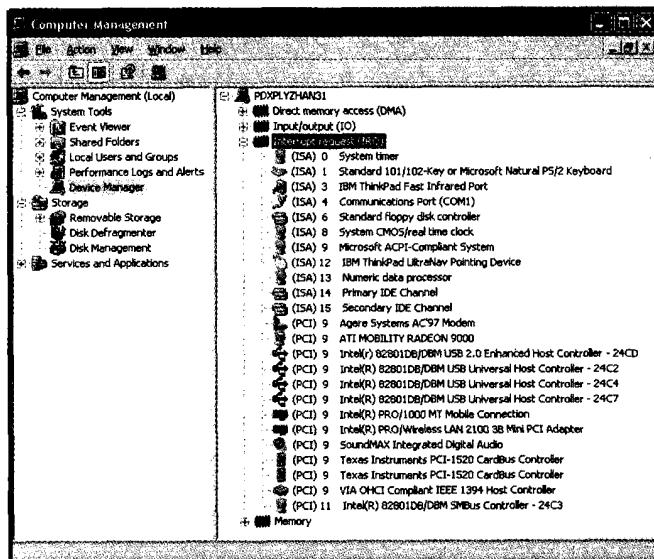


图 3-1 中断请求（IRQ）的使用情况

3.1.2 异常

与中断不同，异常通常是 CPU 在执行指令时因为检测到预先定义的某个（或多个）条件而产生的同步事件。

异常的来源有 3 种，第一种是程序错误，即当 CPU 在执行程序指令时遇到操作数有错误或检测到指令规范中定义的非法情况。前者的一个典型例子是执行除法指令时

遇到除数为零，后者的典型例子包括在用户模式下执行特权指令等。

异常的第二种来源是某些特殊指令，这些指令的预期行为就是产生相应的异常，比如 INT 3 指令，该指令的目的就是产生一个断点异常，让 CPU 中断（break）进调试器。换句话说，这个异常是“故意”产生的，是预料内的。这样的指令还有 INTO、INT n 和 BOUND。

异常的第三种来源是奔腾 CPU 引入的机器检查异常（Machine Check Exception），即当 CPU 执行指令期间检测到 CPU 内部或外部的硬件错误，详细情况将在第 6 章介绍。

3.1.3 比较

至此，我们可以归纳出中断和异常的根本差异是：异常来自于 CPU 本身，是 CPU 主动产生的；而中断来自于外部设备，是中断源发起的，CPU 是被动的。

对于机器检查异常，虽然有时是因为外部设备通过设置 CPU 的特殊管脚（BUSCHK#或 BINIT#和 MCERR#）触发的，但是从产生角度来看，仍然是 CPU 检测到管脚信号然后产生异常的，所以机器检查异常仍然是在 CPU 内部产生的。

在很多文献和书籍中，尤其是在早期的资料中，把由 INT n 指令产生的异常称为软件中断（software interrupts），把来自外部硬件的中断（包括可屏蔽中断和不可屏蔽中断）叫外部中断。尽管今天仍然有很多地方使用这种说法，但是大家应该意识到，严格来说，INT n 导致的软件中断不是中断而是异常。因为 INT n 指令的行为更符合异常的特征——产生于 CPU 内部，来源是正在执行的指令本身。在本书中，如不加特殊说明，中断就是指来自 CPU 外部的硬件中断。

3.2 异常的分类

根据 CPU 报告异常的方式和导致异常的指令是否可以安全地重新执行，IA-32 CPU 把异常分为 3 类：错误（Fault）、陷阱（Trap）和中止（Abort）。

3.2.1 错误类异常

导致错误类异常的情况通常可以被纠正，而且一旦纠正后，程序可以无损失地恢复执行。此类异常的一个最常见例子就是内存页错误（page fault）。在 Windows XP 这样的操作系统中，页错误异常几乎每秒钟都在发生，比如在笔者写作本内容时所用的 Windows XP 系统中，其发生频率大约是每秒钟 60~80 次。页错误异常之所以如此频繁，是因为它是虚拟内存机制的基础。因为物理内存的空间有限，所以操作系统会把某些暂时不用的内存以页为单位交换到外部存储器（通常是硬盘）上。当有程序访问到这些不在物理内存中的页所对应的内存地址时，CPU 便会产生一个页错误异常（有时简称为缺页错误或缺页异常），并转去执行该异常的处理程序，后者会调用内存管理器的函数把对应的内存页交换回物理内存，然后再让 CPU 返回到导致该异常的那条指令处

恢复执行。当第二次执行刚才导致异常的指令时，对应的内存页已经在物理内存中（错误情况被纠正），因此就不会再产生页错误异常了。以上介绍省去了一些细节，我们会在以后的内容中逐步补充。

当 CPU 报告错误类异常时，CPU 将其状态恢复成导致该异常的指令被执行之前的状态。而且在 CPU 转去执行异常处理程序前，在栈中保存的 CS 和 EIP 指针是指向导致异常的这条指令的（而不是下一条指令）。因此，当异常处理程序返回继续执行时，CPU 接下来执行的第一条指令仍然是刚才导致异常的那条指令。所以，如果导致异常的情况还没有被消除，那么 CPU 会再次产生异常。

3.2.2 陷阱类异常

下面再来看陷阱类异常，与错误类异常不同，当 CPU 报告陷阱类异常时，导致该异常的指令已经执行完毕，压入栈的 CS 和 EIP 值（也就是异常处理程序的返回地址）是导致该异常的指令执行后紧接着要执行的下一条指令。值得说明的是，下一条指令并不一定是与导致异常的指令相邻的下一条。如果导致异常的指令是跳转指令或函数调用指令，那么下一条指令可能是内存地址不相邻的另一条指令。

导致陷阱类异常的情况通常也是可以无损失地恢复执行的。比如 INT 3 指令导致的断点异常就属于陷阱类异常，该异常会使 CPU 中断到调试器（详见 4.1 节），从调试器返回后，被调试器程序可以继续执行。

3.2.3 中止类异常

中止类异常主要用来报告严重的错误，比如硬件错误和系统表中包含非法值或不一致的状态等。这类异常不允许恢复继续执行。原因有二，首先，当这类异常发生时，CPU 并不能保证报告的导致异常的指令地址是精确的。另外，出于安全性的考虑，这类异常可能是由于导致该异常的程序执行非法操作导致的，因此就应该强迫其中止退出。表 3-1 列出了 3 类异常的关键特征。

表 3-1 异常分类

分类	报告时间	保存的 CS 和 EIP 地址	可恢复性
错误 (Fault)	开始执行导致异常的指令时	导致异常的那条指令	可以恢复执行(参见下文)
陷阱 (Trap)	执行完导致异常的指令时	导致异常的那条指令的下一条指令	可以恢复执行
中止 (Abort)	不确定	不确定	不可以

需要说明的是，某些情况的错误类异常是不可恢复的。比如，如果执行 POPAD (Pop All General-Purpose Registers Doublewords) 指令时栈指针 (ESP) 超出了栈所在段的边界，那么 CPU 会报告栈错误异常。对于这种情况，尽管异常处理例程所看到的 CS 和 EIP 指针仍然被恢复成好像 POPAD 指令没有被执行过那样，但是处理器内部的状态已经变化了，某些通用寄存器的值可能已经被改变了。这种情况大多是由于程序使用堆

栈不当所造成的，比如压栈和弹出操作不匹配，所以操作系统应该将此类异常当作程序错误来处理，终止导致这类异常的程序。

3.3 异常例析

本节我们将介绍 IA-32 CPU 已经定义的所有异常及异常的错误码，并通过一个小程序来演示除零异常。

3.3.1 列表

在系统中，每个中断或异常都被赋予一个整数 ID，称为向量号（Vector No.），系统（CPU 和操作系统等软件）通过向量号来识别该中断或异常。IA-32 架构规定 0 到 31 号向量供 CPU 设计者（英特尔等设计 x86 处理器的公司）使用，32~255 号向量（224 个）供操作系统和计算机系统生产厂商（OEM 等）或其他软硬件开发商使用（详见第 11 章）。表 3-2 简要列出了迄今为止 IA-32 架构中定义的所有中断和异常。

表 3-2 中断和异常列表

向量号	助记符	类型	描述	来源
0	#DE	错误	除零错误	DIV 和 IDIV 指令
1	#DB	错误/陷阱	调试异常，用于软件调试	任何代码或数据引用
2		中断	NMI 中断	不可屏蔽的外部中断
3	#BP	陷阱	断点	INT 3 指令
4	#OF	陷阱	溢出	INTO 指令
5	#BR	错误	数组越界	BOUND 指令
6	#UD	错误	无效指令（没有定义的指令）	UD2 指令（奔腾 Pro CPU 引入此指令）或任何保留的指令
7	#NM	错误	数学协处理器不存在或不可用	浮点或 WAIT/FWAIT 指令
8	#DF	中止	双重错误（Double Fault）	任何可能产生异常的指令、不可屏蔽中断或可屏蔽中断
9	#MF	错误	向协处理器传送操作数时检测到页错误（Page Fault）或段不存在，自从 486 把数学协处理器集成到 CPU 内部后，本异常便保留不用	浮点指令
10	#TS	错误	无效 TSS	任务切换或访问 TSS
11	#NP	错误	段不存在	加载段寄存器或访问系统段
12	#SS	错误	栈段错误	栈操作或加载 SS 寄存器
13	#GP	错误	通用保护（GP）异常，如果一个操作违反了保护模式下的规定，而且该情况不属于其他异常，则 CPU 便产生通用保护异常，很多时候也被翻译为一般保护异常	任何内存引用和保护性检查

续表

向量号	助记符	类型	描述	来源
14	#PF	错误	页错误	任何内存引用
15	保留			
16	#MF	错误	浮点错误	浮点或 WAIT/FWAIT 指令
17	#AC	错误	对齐检查	对内存中数据的引用 (486CPU 引入)
18	#MC	中止	机器检查 (Machine Check)	错误代码和来源与型号有关 (奔腾 CPU 引入)
19	#XF	错误	SIMD 浮点异常	SIMD 浮点指令 (奔腾 III CPU 引入)
20~31	保留			
32~255	用户定义中断	中断	可屏蔽中断	来自 INTR 的外部中断或 INT n 指令

3.3.2 错误代码

CPU 在产生某些异常时，会向栈中压入一个 32 位的错误代码。其格式如图 3-2。

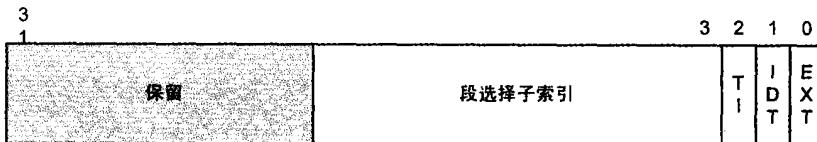


图 3-2 异常的错误代码

其各个位域的含义如下。

- EXT (External Event) (位 0): 如果为 1 表示外部事件导致该异常。
- IDT (Descriptor Location) (位 1): 描述符位置, 如果为 1, 表示错误码的段选择子索引部分指向的是 IDT 表中的门描述符, 如果为 0, 表示索引部分指向的是 LDT 或 GDT 中的描述符。
- TI (GDT/LDT) (位 2): 仅当 IDT 位为 0 时有效, 当该位为 1 时, 表示索引部分指向的 LDT 中的段或门描述符。如果为 0, 表示索引部分指向的 GDT 中的描述符。
- 段选择子索引域表示与该错误有关的描述符在 IDT、LDT 或 GDT 表中的索引。

页错误异常的错误码采用的格式与此不同。

3.3.3 示例

下面通过一个小程序来进一步理解错误类异常, 如清单 3-1。该例子使用了 Windows 操作系统的结构化异常机制, 对其很陌生的读者可以先读一下本书的第 11 章(11.4 节)。

清单 3-1 演示错误类异常的 Fault 小程序

```

1 // fault.cpp : demonstrate fault exception with divide by zero.
2 // Raymond Zhang 2005 Dec.
3 #include <stdio.h>
4 #include <windows.h>
```

```

5
6 #define VAR_WATCH() printf("nDividend=%d, nDivisor=%d, nResult=%d.\n", \
7     nDividend,nDivisor,nResult)
8
9 int main(int argc, char* argv[])
10 {
11     int nDividend=22,nDivisor=0,nResult=100;
12
13     __try
14     {
15         printf("Before div in __try block:");
16         VAR_WATCH();
17
18         nResult=nDividend / nDivisor;
19
20         printf("After div in __try block: ");
21         VAR_WATCH();
22     }
23     __except(printf("In __except block: "),VAR_WATCH(),
24             GetExceptionCode()==EXCEPTION_INT_DIVIDE_BY_ZERO?
25             (nDivisor=1,
26              printf("Divide Zero exception detected: "), VAR_WATCH(),
27              EXCEPTION_CONTINUE_EXECUTION):
28              EXCEPTION_CONTINUE_SEARCH)
29     {
30         printf("In handler block.\n");
31     }
32     return getchar();
33 }

```

在以上小程序中，我们故意设计了一个除零操作，即第 18 行，该行对应的汇编指令如下：

```

18:           nResult=nDividend / nDivisor;
00401087 8B 45 E4      mov     eax,dword ptr [ebp-1Ch]
0040108A 99            cdq
0040108B F7 7D E0      idiv    eax,dword ptr [ebp-20h]
0040108E 89 45 DC      mov     dword ptr [ebp-24h],eax

```

IA-32 手册中对 IDIV 指令内部操作的定义开始几行是：

```

IF SRC = 0
  THEN #DE; (* Divide error *)
FI;
.....

```

也就是当 CPU 在执行 IDIV 指令时，首先会检查源操作数（除数）是否等于零，如果等于零，那么就产生除零异常。#DE 是除零异常的简短记号（参见表 3-2）。

对于我们的例子，当 CPU 执行到 0040108B 地址处的 IDIV 指令时，因为源操作数的值是零，所以 CPU 会检测到此情况，并报告除零异常。接下来 CPU 会把 EFLAGS 寄存器、CS 寄存器和 EIP 寄存器的内容压入栈保存起来，然后转去执行除零异常对应的异常处理程序（如何寻找到处理程序的细节将在 3.5 节中讨论）。异常处理程序在执行完一系列检查和预处理后（细节请参见 11.2 节和 11.3 节），会调用 __except 块的过滤表达式，并期望得到以下 3 个值之一。

- **EXCEPTION_CONTINUE_SEARCH (0)**: 本保护块不处理该异常，请继续寻找其他的异常保护块。

- **EXCEPTION_CONTINUE_EXECUTION (-1)**: 异常情况被消除, 请回去继续执行。
- **EXCEPTION_EXECUTE_HANDLER (1)**: 请执行本块中的处理代码。

过滤表达式可以包含函数调用或其他表达式, 只要其最终结果是以上 3 个值中的一个。我们的例子利用逗号运算符, 在其中包含了一系列操作: 第 23 行打印出位置信息和当时的各变量值; 第 24 行到第 28 行通过条件运算符来判断发生的是何种异常, 如果不是除零异常 (异常代码不等于 EXCEPTION_INT_DIVIDE_BY_ZERO), 那么就返回 EXCEPTION_CONTINUE_SEARCH, 让异常处理程序继续搜索其他的保护块, 如果是除零异常, 就执行第 25、26 和 27 行。第 25 行, 将除数改为 1 (纠正错误情况), 第 26 行打印出当前信息, 然后第 27 行返回 EXCEPTION_CONTINUE_EXECUTION, 让 CPU 回到导致该异常的指令位置继续执行。

执行这个小程序, 得到的结果如下:

```
Before div in __try block: nDividend=22, nDivisor=0, nResult=100.
In __except block: nDividend=22, nDivisor=0, nResult=100.
Divide Zero exception detected: nDividend=22, nDivisor=1, nResult=100.
After div in __try block: nDividend=22, nDivisor=1, nResult=22.
```

容易看出, 以上实际执行结果和我们的分析是一致的, 异常情况被纠正后, 程序又继续正常运行了。

3.4 中断/异常优先级

CPU 在同一时间只可以执行一个程序, 如果多个中断请求或异常情况同时发生, CPU 应该以什么样的顺序来处理呢? 是按照优先级高低依次处理, 先处理优先级最高的。截至本书写作之时, IA-32 架构定义了 10 个中断/异常优先级别, 具体情况如表 3-3。

表 3-3 中断/异常的优先级别

优先级	描述
1 (最高)	硬件重启动和机器检查异常 (Machine Check Exception)
2	任务切换陷阱 (参见 4.3 节)
3	外部硬件 (例如芯片组) 通过 CPU 引脚发给 CPU 的特别干预 (interventions): <ul style="list-style-type: none"> ■ #FLUSH: 强制 CPU 刷新高速缓存 ■ #STPCLK (Stop Clock): 使 CPU 进入低功耗的 Stop-Grant 状态 ■ #SMI (System Management Interrupt): 切换到系统管理模式 (SMM) ■ #INIT: 热重启动 (soft reset)
4	上一指令导致的陷阱: <ul style="list-style-type: none"> ■ 执行 INT 3 (断点指令) 导致的断点 ■ 调试陷阱, 包括单步执行异常 (EFlags[TF]=1) 和利用调试寄存器设置的数据或输入输出断点 (详见 4.2 节)
5	不可屏蔽 (外部硬件) 中断 (NMI)
6	可屏蔽的 (外部硬件) 中断
7	代码断点错误异常, 即从内存取指令时检测到与调试寄存器中的断点地址相匹配, 也就是利用调试寄存器设置的代码断点

续表

优先级	描述
8	取下一条指令时检测到的错误: ■ 违反代码段长度限制 ■ 代码内存页错误(即代码属性的内存页导致页错误)
9	解码下一指令时检测到的错误: ■ 指令长度大于15字节(包括前缀) ■ 非法操作码 ■ 协处理器不可用
10(最低)	执行指令时检测到的错误: ■ 溢出,当EFlags[OF]=1时执行INTO指令 ■ 执行BOUND指令时检测到边界错误 ■ 无效的TSS(任务状态段) ■ 段不存在 ■ 栈异常 ■ 一般保护异常 ■ 数据页错误 ■ 对齐检查异常 ■ x87 FPU异常 ■ SIMD浮点异常

IA-32 架构保证上表中各优先级别的定义对于所有 IA-32 处理器都是一致的,但是,同一年级中的各种情况的优先级可能与 CPU 型号有关。

3.5 中断/异常处理

尽管中断和异常从产生的根源来看有着本质的区别,但是系统(CPU 和操作系统)是用统一的方式来响应和管理它们的。中断和异常处理的核心数据结构是中断描述符表(Interrupt Descriptor Table,简称IDT)。当中断和异常发生时,CPU 通过查找 IDT 表来定位处理例程的地址,然后转去执行该处理例程。这个查找的过程是在 CPU 内部进行的。通常,系统软件(操作系统或 BIOS 固件)在系统初始化阶段就准备好中断处理例程和 IDT 表,然后把 IDT 表的位置通过 IDTR(IDT Register)寄存器告诉 CPU。IDTR 寄存器的长度是 48 位,高 32 位是 IDT 表的基址址(线性地址),低 16 位是 IDT 表的边界(以字节为单位的最大偏移,数值为 IDT,表总长度-1)。LIDT 和 SIDT 指令分别用来从内存变量加载 IDTR 和把 IDTR 的值存储到内存变量。只有在 0 特权级下才可以执行 LIDT 和 SIDT 指令,这可以保证 IDT 表不会被恶意的用户态程序所破坏。CPU 上电或重新启动后 IDTR 寄存器会被初始化为如下默认值:基地址=0x00000000,边界=0xFFFF(IA-32 手册卷 III,2.4.3 节)。这恰好与 8086 CPU 和 DOS 操作系统下中断向量表(Interrupt Vector Table,简称IVT)的位置相一致。实模式下 IVT 表位于物理地址 0 开始的 1KB 内存区中,每个 IVT 表项的长度是 4 个字节,共有 256 个表项,与 x86 CPU 的 256 个中断向量一一对应。前面我们介绍过,CPU 上电或重新启动后的初始状态总是

实模式状态。所以 IDTR 寄存器的初始值也正好符合实模式下中断向量表的要求。

但这里要说明的是，最早的 x86 处理器 8086 并没有 IDTR 寄存器（286 引入），因此当时 IVT 表的位置一定是在地址 0 开始的 1KB 内存区域中，是不可配置的。对于后来的 IA-32 处理器，可以使用 IDTR 寄存器来改变 IVT 表的位置。但是如果要与基于 8086 设计的系统软件（比如 DOS）兼容，那么 IVT 表的位置应该保持不变。

实模式下，每个 IVT 表项的 4 个字节分为两个部分，高两个字节为中断例程的段地址；低两个字节为中断例程的偏移地址。因为是在实模式下，所以段地址左移 4 位再加上偏移地址便可以得到 20 位的中断例程地址。

下面归纳一下实模式下 IA-32 CPU 响应中断和异常的全过程。

1. 将代码段寄存器 CS 和指令指针寄存器（EIP）的低 16 位压入堆栈。
2. 将标志寄存器 EFLAGS 的低 16 位压入堆栈。
3. 清除标志寄存器的 IF 标志，以禁止其他中断。
4. 清除标志寄存器的 TF（Trap Flag）、RF（Resume Flag）、AC（Alignment Check）标志。
5. 使用向量号 n 作为索引，在 IVT 中找到对应的表项（n*4+IVT 表基地址）。
6. 将表项中的段地址和偏移地址分别装入 CS 和 EIP 寄存器中，并开始执行对应的代码。
7. 中断例程总是以 IRET 指令结束。IRET 指令会从堆栈中弹出前面保存的 CS、IP 和标志寄存器值，然后返回执行被中断的程序。

保护模式下的中断和异常处理要考虑权限控制、任务切换等问题，所以情况会复杂得多，我们将在第 11 章中介绍。

3.6 本章总结

本章首先介绍了中断和异常这两个重要概念（3.1 节和 3.2 节），然后介绍了 IA-32 CPU 定义的各个异常（3.3 节）。3.4 节讨论了中断和异常的优先级。3.5 节介绍了实模式下的中断/异常响应和处理。

异常与调试有着更为密切的关系，本章从 CPU 的角度首次介绍了异常的基本概念，第 11 章和第 24 章将分别从操作系统和编译器（程序语言）的角度做进一步阐述。

参考文献

1. IA-32 Intel® Architecture Software Developer’s Manual Volume 3. Intel Corporation
2. Tom Shanley. The Unabridged Pentium 4: IA-32 Processor Genealogy. Addison Wesley, 2004

断点和单步执行

断点和单步执行是两个经常使用的调试功能，也是调试器的核心功能。本章我们将介绍 IA-32 CPU 是如何支持断点和单步执行功能的。前两节将分别介绍软件断点和硬件断点，第 4.3 节介绍用于实现单步执行功能的陷阱标志。在前三节的基础上，第 4.4 节将分析一个真实的调试器程序，看它是如何实现断点和单步执行功能的。

4.1 软件断点

x86 系列处理器从其第一代产品英特尔 8086 开始就提供了一条专门用来支持调试的指令，即 INT 3。简单地说，这条指令的目的就是使 CPU 中断（break）到调试器，以供调试者对执行现场进行各种分析。当我们调试程序时，可以在可能有问题的地方插入一条 INT 3 指令，使 CPU 执行到这一点时停下来。这便是软件调试中经常用到的断点（breakpoint）功能，因此 INT 3 指令又被称为断点指令。

4.1.1 感受 INT 3

下面通过一个小实验来感受一下 INT 3 指令的工作原理。在 Visual C++ Studio 6.0（以下简称为 VC6）中创建一个简单的 HelloWorld 控制台程序 HiInt3，然后在 main() 函数的开头通过嵌入式汇编插入一条 INT 3 指令：

```
int main(INT argc, char* argv[])
{
    // manual breakpoint
    _asm INT 3;
    printf("Hello INT 3!\n");
    return 0;
}
```

当在 VC 环境中执行以上程序时，会得到图 4-1 所示的对话框。点 OK 按钮后，程序便会停在 INT 3 指令所在的位置。由此看来，我们刚刚插入的一行（_asm INT 3）相当于在那里设置了一个断点。实际上，这也正是通过注入代码手工设置断点的方法，这种方法在调试某些特殊的程序时还非常有用。

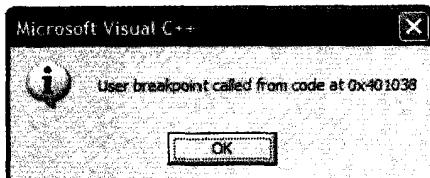


图 4-1 CPU 遇到 INT 3 指令时会把执行权移交给调试器

此时打开反汇编窗口，可以看到内存地址 00401028 处确实是 INT 3 指令：

```
10: _asm INT 3;
00401028 int 3
```

打开寄存器窗口，可以看到程序指针寄存器的值也是 00401028。

```
EAX = CCCCCCCC EBX = 7FFDE000 ECX = 00000000 EDX = 00371588
ESI = 00000000 EDI = 0012FF80
EIP = 00401028 ESP = 0012FF34 EBP = 0012FF80 .....
```

根据我们在第 3 章中的介绍，断点异常（INT 3）属于陷阱类异常，当 CPU 产生异常时，其程序指针是指向导致异常的下一条指令的。但是，现在我们观察到的结果却是指向导致异常的这条指令的。这是为什么呢？简单地说，是操作系统为了支持调试对程序指针做了调整。我们将在后面揭晓答案。

4.1.2 在调试器中设置断点

下面考虑一下调试器是如何设置断点的。当我们在调试器（例如 VC6 或 Turbo Debugger 等）中对代码的某一行设置断点时，调试器会先把这里的本来指令的第一个字节保存起来，然后写入一条 INT 3 指令。因为 INT 3 指令的机器码为 11001100b（0xCC），仅有一个字节，所以设置和取消断点时也只需要保存和恢复一个字节，这是设计这条指令时须考虑好的。

顺便说一下，虽然 VC6 是把断点的设置信息（断点所在的文件和行位置）保存在和项目文件相同位置且相同主名称的一个.opt 文件中，但是请注意，该文件并不保存每个断点处应该被 INT 3 指令替换掉的那个字节，因为这种替换是在启动调试时和调试过程中动态进行的。这可以解释，有时在 VC6 中，在非调试状态下，我们甚至可以在注释行设置断点，当开始调试时，会得到一个图 4-2 所示的警告对话框。这是因为当用户在非调试状态下设置断点时，VC6 只是简单地记录下该断点的位置信息。当开始调试（让被调试程序开始运行）时，VC6 会一个一个地取出 OPT 文件中的断点记录，并真正将这些断点设置到目标代码的内存映像中，也就是要将断点位置对应的指令的第一个字节先保存起来，再替换为 0xCC，即 INT 3 指令，这个过程被称为落实断点（resolve breakpoint）。

在落实断点的过程中，如果 VC6 发现某个断点的位置根本对应不到目标映像的代码段，那么便会发出图 4-2 所示的警告。

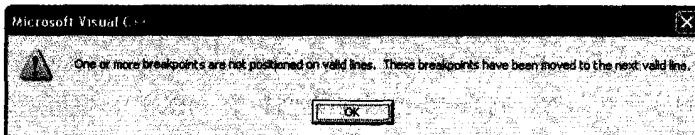


图 4-2 VC6 在开始调试时才真正设置断点，会对无法落实的断点发出提示

4.1.3 断点命中

当 CPU 执行到 INT 3 指令时，由于 INT 3 指令的设计目的就是中断到调试器，因此，CPU 执行这条指令的过程也就是产生断点异常（breakpoint exception，简称#BP）并转去执行异常处理例程的过程。在跳转到处理例程之前，CPU 会保存当前的执行上下文，包括段寄存器、程序指针寄存器等内容。清单 4-1 列出了 CPU 工作在实模式时执行 INT 3 的过程（摘自《英特尔 IA-32 架构软件开发手册（卷 2A）》）。

清单 4-1 实模式下 INT 3 指令的执行过程

```

1  REAL-ADDRESS-MODE:
2  IF ((vector_number * 4) + 3) is not within IDT limit
3  THEN #GP;
4  FI;
5  IF stack not large enough for a 6-byte return information
6  THEN #SS;
7  FI;
8  Push (EFLAGS[15:0]);
9  IF ← 0; (* Clear interrupt flag *)
10 TF ← 0; (* Clear trap flag *)
11 AC ← 0; (* Clear AC flag *)
12 Push(CS);
13 Push(IP);
14 (* No error codes are pushed *)
15 CS ← IDT(Descriptor (vector_number * 4), selector));
16 EIP ← IDT(Descriptor (vector_number * 4), offset)); (* 16 bit offset AND
17 000FFFFH *)
18 END

```

其中第 2 行是检查根据中断向量号计算出的向量地址是否超出了中断向量表的边界（limit）。实模式下，中断向量表的每个表项是 4 个字节，分别处理例程的段和偏移地址（各两字节）。如果超出了，那么便产生保护性错误异常。#GP 即 General Protection Exception，通用保护性异常。第 7 行的 FI 是 IF 语句的结束语句。

第 5 行是检查栈上是否有足够的空间来保存寄存器，当堆栈不足以容纳接下来要压入的 6 字节的（CS、IP 和 EFLAGS 的低 16 位）内容时，便产生堆栈异常#SS。第 9 行到第 11 行是清除标志寄存器的 IF、TF 和 AC 位。第 12 行和第 13 行是将当前的段寄存器和程序指针寄存器的内容保存在当前程序的栈中。

第 15 行和第 16 行是将注册在中断向量表（IDT）中的异常处理例程的入口地址加载到 CS 和 IP（程序指针）寄存器中。这样，CPU 执行好这条指令后，接下来便会执行异常处理例程的函数了。

对于 DOS 这样的在实模式下的单任务操作系统，断点异常的处理例程通常就是调试器程序注册的函数，因此，CPU 便开始执行调试器的代码了。当调试器执行好调试功能需要恢复被调试程序执行时，它只要执行中断返回指令（IRET），便可以让 CPU 从断点的位置继续执行了（见下文）。

在保护模式下，INT 3 指令的执行过程虽然有所不同，比如是在由 IDTR 寄存器标识的 IDT 表中寻找异常处理函数，找到后会检查函数的有效性，但是其原理是一样的，也是保存好寄存器后，便跳转去执行异常处理例程。

对于 Windows 这样工作在保护模式下的多任务操作系统，INT 3 异常的处理函数是操作系统的内核函数（KiTrap03）。因此执行 INT 3 会导致 CPU 执行 nt!KiTrap03 例程。因为我们现在讨论的是应用程序调试，断点指令位于用户模式下的应用程序代码中，因此 CPU 会从用户模式转入内核模式。接下来，经过几个内核函数的分发和处理（将在第 11 章详细讨论），因为这个异常是来自用户模式的，而且该异常的拥有进程正在被调试（进程的 DebugPort 非 0），所以，内核例程会把这个异常通过调试子系统以调试事件的形式分发给用户模式的调试器，对于我们的例子也就是 VC6。在通知 VC6 后，内核的调试子系统函数会等待调试器的回复。收到调试器的回复后，调试子系统的函数会层层返回，最后返回到异常处理例程，异常处理例程执行中断返回指令，使被调试的程序继续执行。

在调试器（VC6）收到调试事件后，它会根据调试事件数据结构中的程序指针得到断点异常的发生位置，然后在自己内部的断点列表中寻找与其匹配的断点记录。如果能找到，则说明这是“自己”设置的断点，执行一系列准备动作后，便允许用户进行交互式调试。如果找不到，就说明导致这个异常的 INT 3 指令不是 VC6 动态替换进去的，因此会显示一个图 4-1 所示的对话框，意思是说一个“用户”插入的断点被触发了。

值得说明的是，在调试器下，我们是看不到动态替换到程序中的 INT 3 指令的。大多数调试器的做法是在被调试程序中断到调试器时，会先将所有断点位置被替换为 INT 3 的指令恢复成原来的指令，然后再把控制权交给用户。对于不做这种断点恢复的调试器（如 VC6），它的反汇编功能和内存观察功能也都有专门的处理，让用户看到的始终是断点所在位置本来的内容。本节后面我们会给出两种观察方法。

在 Windows 系统中，操作系统的断点异常处理函数（KiTrap03）对于 x86 CPU 的断点异常会有一个特殊的处理，会将程序指针寄存器的值减 1。

```
nt!KiTrap03+0x9a:
8053dd0e 8b5d68      mov    ebx,dword ptr [ebp+68h]
8053dd11 4b          dec    ebx
8053dd12 b903000000  mov    ecx,3
8053dd17 b803000080  mov    eax,80000003h
8053dd1c e8a3f8ffff  call   nt!CommonDispatchException (8053d5c4)
```

出于这个原因，我们在调试器看到的程序指针指向的仍然是 INT 3 指令的位置，

而不是它的下一条指令。这样做的目的有如下两个。

1. 调试器在落实断点时，不管所在位置的指令是几个字节，它都只替换一个字节。因此，如果程序指针指向下一个指令位置，那么指向的可能是原来的多字节指令的第二个字节，不是一条完整的指令。
2. 因为有断点在，所以被调试程序在断点位置的那条指令还没有执行。按照程序指针总是指向即将执行的那条指令的原则，应该把程序指针指向这条要执行的指令，也就是倒退回一个字节，指向本来指令的起始地址。

这也就是我们前面问题的答案。

归纳一下，当 CPU 执行 INT 3 指令时，它会跳转到异常处理例程，让当前的程序接受调试，调试结束后，异常处理例程使用中断返回机制让 CPU 再继续执行原来的程序。下面我们将详细介绍恢复执行的过程。

4.1.4 恢复执行

当用户结束分析希望恢复被调试程序执行时，调试器通过调试 API 通知调试子系统，这会导致系统内核的异常分发函数返回到异常处理例程，然后异常处理例程通过 IRET/IRETD 指令触发一个异常返回动作，使 CPU 恢复执行上下文，从发生异常的位置继续执行。注意，这时的程序指针是指向断点所在那条指令的，此时刚才的断点指令已经被替换成本来的指令，于是程序会从断点位置的原来指令继续执行。

这里有一个问题，前面我们说当断点命中中断到调试器时，调试器会把所有断点处的 INT 3 指令恢复成本来的内容。因此，在用户发出了恢复执行命令后，调试器在通知系统真正恢复程序执行前，调试器需要将断点列表中的所有断点再落实一遍。但是对于刚才命中的这个断点需要特别对待，试想如果把这个断点处的指令也替换为 INT 3，那么程序一执行便又触发断点了。但是如果不变换，那么这个断点便没有被落实，程序下次执行到这里时就不会触发断点，而用户并不知道这一点。对于这个问题，大多数调试器的做法都是先单步执行一次。也就是说，先设置单步执行标志（下一节将详细讨论），然后恢复执行，将断点所在位置的指令执行完。因为设置了单步标志，所以，CPU 执行完断点位置的这条指令后会立刻再中断到调试器中，这一次调试器不会通知用户，会做一些内部操作后便立刻恢复程序执行，而且将所有的断点都落实（使用 INT 3 替换）。如果用户在恢复程序执行前，已经取消了当前的断点，那么就不需要先单步执行一次了。

4.1.5 特别用途

因为 INT 3 指令的特殊性，所以它有一些特别的用途。让我们从一个有趣的现象说起。当我们用 VC6 进行调试时，常常会观察到一块刚分配的内存或字符串数组里面被填充满了“CC”。如果是在中文环境下，因为 0xCCCC 恰好是汉字“烫”字的简码，所以

会观察到很多“烫烫烫……”（见图 4-3），而 0xCC 又正好是 INT 3 指令的机器码，这是偶然的么？答案是否定的。因为这是编译器故意这样做的。为了辅助调试，编译器在编译调试版本时会用 0xCC 来填充刚刚分配的缓冲区。这样，如果因为缓冲区或堆栈溢出时程序指针意外指向了这些区域，那么便会因为遇到 INT 3 指令而马上中断到调试器。

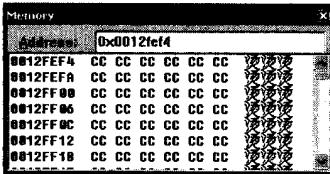


图 4-3 填充了 INT 3 指令的缓冲区

事实上，除了以上用法，编译器还使用了 INT 3 指令来填充函数或代码段末尾的空闲区域，也就是用它来做内存对齐。这也解释为什么有时我们没有手工插入任何对 INT 3 的调用，但也还会遇到图 4-1 所示的对话框。

4.1.6 断点 API

Windows 操作系统提供了 API 供应用程序向自己的代码中插入断点。在用户模式下，可以使用 `DebugBreak()` API，在内核模式下可以使用 `DbgBreakPoint()` 或者 `DbgBreakPointWithStatus()`。

把前面 HiInt3 程序中的对 INT 3 的直接调用改为调用 Windows API `DebugBreak()`（需要在开头 `include <windows.h>`），然后执行，可以看到产生的效果是一样的。通过反汇编很容易看出这些 API 在 x86 平台上其实都只是对 INT 3 指令的简单包装。

```
1 lkd> u nt!DbgBreakPoint
2 nt!DbgBreakPoint:
3 804df8c4 cc           int     3
4 804df8c5 c3           ret
```

以上反汇编是用 WinDBG 的本地内核调试环境而做的。提示符 `lkd>` 的含义是“local kernel debug”。本地内核调试需要 Windows XP 或以上的操作系统才能支持。

`DbgBreakPointWithStatus()` 允许向调试器传递一个整型参数。

```
1 lkd> u nt!DbgBreakPointWithStatus
2 804df8d1 8b442404      mov     eax, [esp+0x4]
3 804df8d5 cc           int     3
```

其中 `[esp+0x4]` 代表 `DbgBreakPointWithStatus` 函数的第一个参数。

4.1.7 系统对 INT 3 的优待

关于 INT 3 指令还有一点要说明的是，INT 3 指令与当 $n=3$ 时的 INT n 指令（通常所说的软件中断）并不同。INT n 指令对应的机器码是 0xCD 后跟 1 字节 n 值，比如 INT 23H

会被编译为 0xCD23。与此不同的是，INT 3 指令具有独特的单字节机器码 0xCC。而且系统会对 INT 3 指令给予一些特殊的待遇，比如在虚拟 8086 模式下免受 IOPL 检查等。

因此，当编译器看见 INT 3 时会特别将其编译为 0xCC，而不是 0xCD03。尽管没有哪个编译器会将 INT 3 编译成 0xCD03，但是可以通过某些方法直接在程序中插入 0xCD03，比如可以使用如下嵌入式汇编，利用_EMIT 伪指令直接嵌入机器码。

```
__asm __emit 0xcd __asm __emit 0x03
```

将前面的 HiInt3 小程序略作修改，使用_EMIT 伪指令插入机器码 0xCD03，并在其前后再加入一两行其他指令用做“参照物”（如清单 4-2 所示）。

清单 4-2 HiInt3 程序的源代码

```
7 int main(int argc, char* argv[])
8 {
9     // manual breakpoint
10    __asm INT 3;
11    printf("Hello INT 3!\n");
12
13    __asm
14    {
15        mov eax,eax
16        __asm __emit 0xcd __asm __emit 0x03
17        nop
18        nop
19    }
20    //or use Windows API
21    DebugBreak();
22    //
23    return 0;
24 }
```

在 VC6 下编译以上代码，然后执行，先会得到两次如图 4-1 所示的对话框，第二次是我们用 EMIT 方法插入的 0xCD03 所导致的，但是再执行会反复得到访问违例异常，无法继续。

为了一探究竟，我们使用比 VC6 集成调试器更强大的 WinDBG 调试器。启动 WinDBG 后通过 File>Open Executable 打开可执行程序（\bin\debug\HiInt3.exe）。然后使用反汇编命令 u `hiint3!HiInt3.cpp:11` 观察源代码从第 11 行起的汇编代码（见清单 4-3）。

清单 4-3 HiInt3 程序的汇编代码（第 11 行起）

```
0:000> u `hiint3!HiInt3.cpp:11`
HiInt3!main+0x19 [C:\dig\dbg\author\code\chap04\HiInt3\HiInt3.cpp @ 11]:
00401029 681c204200 push    offset HiInt3!`string' (0042201c)
0040102e e82d000000 call    HiInt3!printf (00401060)
00401033 83c404 add     esp,4
00401036 8bc0      mov     eax,eax
00401038 cd03      int     3
0040103a 90      nop
0040103b 90      nop
0040103c 8bf4      mov     esi,esp
```

可以看到，我们使用 EMIT 伪指令向可执行文件中成功地插入了机器码 0xCD03，

而且反汇编程序也将其反汇编成 INT 3 指令。0xCD03 的地址是 00401038。它后面是两个 NOP 指令，机器码为 0x90。

按 F5 让程序执行，先会遇到 main 函数开头的 INT 3。按 F5 再执行，WinDBG 会接收到断点异常事件，并显示如下信息：

```
(cf8.f28): Break instruction exception - code 80000003 (first chance)
eax=0000000d ebx=7ffdc000 ecx=00424a60 edx=00424a60 esi=0151f764 edi=0012ff80
eip=00401039 esp=0012ff34 ebp=0012ff80 iopl=0 nv up ei pl nz na po nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000202
HiInt3!main+0x29:
00401039 0390908bf4ff          add     edx,dword ptr [eax-0B7470h]
ds:0023:fff48b9d=????????
```

其中 80000003 是 Windows 操作系统定义的断点异常代码。注意，此时的程序指针寄存器的值等于 00401039，这指向的是 0xCD03 的第二个字节。观察最后一行的汇编指令，看来已经出了问题，EIP 指针已经指向了一条指令的中间字节而不是起始处，接下来的指令都“错位”了，本来不属于同一指令的两个 NOP 指令的机器码（0x90），以及它后面的 MOV 指令被强行组合成一条虚假的 ADD 指令，新的指令已经和以前的大相径庭了。根据规定，EIP 指针应该总是指向即将要执行的下一条指令的第一个字节。现在由于 EIP 指向错位了，所以当前的指令变成了一个 ADD 指令，它引用的地址是 fff48b9d，这是指向内核空间的一个地址，是不允许用户代码直接访问的，这正是继续执行会产生访问违例的原因。

```
0:000> g
(1374.d28): Access violation - code c0000005 (first chance)
...
```

那么是什么原因导致 EIP 指针错位的呢？正如前面我们介绍的，Windows 的断点异常处理函数 KiTrap03 在分发这个异常前总是会将程序指针减 1，对于单字节的 INT 3 指令，这样做减法后，刚好指向 INT 3 指令（或者本来指令的起始处）。但对于双字节的 0xCD03 指令，执行这条指令后的 EIP 指针的值是这个指令的下一指令的地址，即 00401040，因此减 1 后等于 00401039，也就是指向 0xCD03 的第二个字节了。

此时，可以通过 WinDBG 的寄存器修改命令将 EIP 寄存器的值手工调整到下一条指令（nop）位置：

```
r eip=0040103a
```

这样调整后，程序便可以继续顺利执行了。

4.1.8 观察调试器写入的 INT 3 指令

可以通过两种方法来观察调试器所插入的断点指令（0xCC，INT 3）。一是使用 WinDBG 调试器的 Noninvasive 调试功能。举例来说，在 VC6 中启动调试，并在第 11 行（printf("Hello INT 3 !\n")）设置一个断点。然后启动 WinDBG，选择 File>Attach to a Process，在对话框的进程列表中选择 HiInt3 进程，并选中下面的 Noninvasive 复选

框，而后按 OK 等待 WinDBG 附加到 HiInt3 进程显示命令提示符后，输入以下命令，对第 11 行源代码所对应的位置进行反汇编：

```
0:000> u `hiint3!HiInt3.cpp:11`  
HiInt3!main+0x19 [C:\dig\dbg\author\code\chap04\HiInt3\HiInt3.cpp @ 11]:  
00401029 cc          int     3  
0040102a 1c20        sbb    al,20h  
0040102c 42          inc    edx  
0040102d 00e8        add    al,ch  
...
```

其中，地址 00401029 处的 0xCC 就是 VC6 调试器插入的断点指令。由于插入了这条指令，所以导致 WinDBG 的反汇编程序以为 0040102a 是下一条指令的开始而继续反汇编，得到了完全错误的结果。与清单 4-3 所示的正确汇编结果相比较，我们知道，事实上从 00401029 开始的 5 个字节 651c204200 都是一条指令，即 push offset HiInt3!`string'(0042201c)，目的是将 printf 的字符串参数压入栈。当插入断点时，push 指令的第一个字节 0x65 被替换为了 0xCC (INT 3)，反汇编程序把 push 指令的其余字节当作新的指令了。

4.1.9 归纳

因为使用 INT 3 指令产生的断点是依靠插入指令和软件中断机制工作的，因此人们习惯把这类断点称为软件断点，软件断点具有如下局限性。

- 属于代码类断点，即可以让 CPU 执行到代码段内的某个地址时停下来，不适用于数据段和 I/O 空间。
- 对于在 ROM (只读存储器) 中执行的程序 (比如 BIOS 或其他固件程序)，无法动态增加软件断点。因为目标内存是只读的，无法动态写入断点指令。这时就要使用我们后面要介绍的硬件断点。
- 在中断向量表或中断描述表 (IDT) 没有准备好或遭到破坏的情况下，这类断点是无法或不能正常工作的，比如系统刚刚启动时或 IDT 被病毒篡改后，这时只能使用硬件级的调试工具。

虽然软件断点存在以上不足，但因为它使用方便，而且没有数量限制 (硬件断点需要寄存器记录断点地址，有数量限制)，所以目前仍被广泛应用。

第 9 章、第 22 章和第 30 章将分别从操作系统、编译器和调试器的角度进一步介绍软件断点。

4.2 硬件断点

1985 年 10 月，英特尔在推出 286 三年半之后推出了 386。这是 PC 历史上又一个具有划时代意义的产品，作为 IA-32 架构的鼻祖，它真正将个人计算机带入了 32 位时代。在调试方面，386 也引入了很多新的功能，其中最重要的就是调试寄存器和硬件断点。

4.2.1 调试寄存器概览

IA-32 处理器定义了 8 个调试寄存器，分别称为 DR0~DR7。在 32 位模式下，它们都是 32 位的；在 64 位模式下，都是 64 位的。本节我们将以 32 位的情况为例来讨论（如图 4-4 所示）。

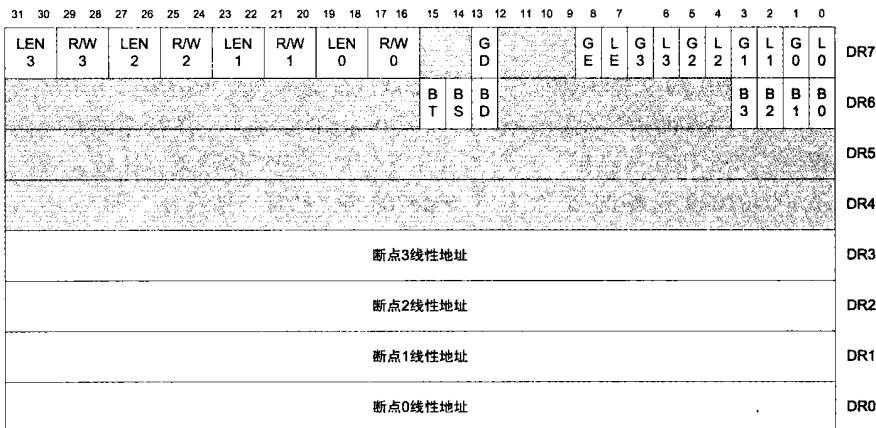


图 4-4 调试寄存器 DR0~DR7

首先，DR4 和 DR5 是保留的，当调试扩展（debug extension）功能被启用（CR4 寄存器的 DE 位设为 1）时，任何对 DR4 和 DR5 的引用都会导致一个非法指令异常（#UD），当此功能被禁止时，DR4 和 DR5 分别是 DR6 和 DR7 的别名寄存器，即等价于访问后者。

其他 6 个寄存器分别是。

- 4 个 32 位的调试地址寄存器（DR0~DR3），64 位下是 64 位的。
- 1 个 32 位的调试控制寄存器（DR7），64 位时，高 32 位保留未用。
- 1 个 32 位的调试状态寄存器（DR6），64 位时，高 32 位保留未用。

通过以上寄存器可以最多设置 4 个断点，其基本分工是 DR0~DR3 用来指定断点的内存（线性地址）或 I/O 地址。DR7 用来进一步定义断点的中断条件。DR6 的作用是当调试事件发生时，向调试器（debugger）报告事件的详细信息，以供调试器判断发生的是何种事件。下面我们分别介绍。

4.2.2 调试地址寄存器

调试地址寄存器（DR0~DR3）用来指定断点的地址。对于设置在内存空间中的断点，这个地址应该是断点的线性地址而不是物理地址，因为 CPU 是在线性地址被翻译为物理地址之前来做断点匹配工作的。这意味着，在保护模式下，我们不能使用调试寄存器来针对一个物理内存地址设置断点。

4.2.3 调试控制寄存器

在 DR7 寄存器中，有 24 位是被划分成四组分别与四个调试地址寄存器相对应的，比如 L0、G0、R/W0 和 LEN0 这 6 位是与 DR0 相对应的，L1、G1、R/W1 和 LEN1 这 6 位是与 DR1 相对应的，其余的依此类推。表 4-1 列出了 DR7 中各个位域的具体含义。

表 4-1 调试控制寄存器 (DR7)

位域	名称	比特位	描述
R/W0 ~R/W3	读写域	R/W0: 16, 17 R/W1: 20, 21 R/W2: 24, 25 R/W3: 28, 29	分别与 DR0 ~ DR3 四个调试地址寄存器相对应，用来指定被监控地址的访问类型，其含义如下。 <ul style="list-style-type: none"> ■ 00: 仅当执行对应地址的指令时中断 ■ 01: 仅当向对应地址写数据时中断 ■ 10: 386 和 486 不支持此组合。对于以后的 CPU，可以通过把 CR4 寄存器的 DE(调试扩展)位设为 1 时启用该组合，其含义为“当向相应地址进行输入输出(即 I/O 读写)时中断” ■ 11: 当向相应地址读写数据时都中断，但是从该地址读取指令除外
LEN0 ~LEN3	长度域	LEN0: 18, 19 LEN1: 22, 23 LEN2: 26, 27 LEN3: 30, 31	分别与 DR0 ~ DR3 四个调试地址寄存器相对应，用来指定要监控的区域长度，其含义如下。 <ul style="list-style-type: none"> ■ 00: 1 字节长 ■ 01: 2 字节长 ■ 10: 8 字节长(奔腾 4 或至强 CPU)或未定义(其他处理器) ■ 11: 4 字节长 <p>注意：如果对应的 R/Wn 为 0(即执行指令中断)，那么这里的设置应该为 0，参见下文</p>
L0~L3	局部断点启用	L0: 0 L1: 2 L2: 4 L3: 6	分别与 DR0 ~ DR3 四个调试地址寄存器相对应，用来启用或禁止对应断点的局部匹配。如果该位设为 1，当 CPU 在当前任务中检测到满足所定义的断点条件时便中断，并且自动清除此位。如果该位设为 0，便禁止此断点。
G0~G3	全部断点启用	G0: 1 G1: 3 G2: 5 G3: 7	分别对应 DR0 ~ DR3 四个调试地址寄存器，用来全局启用和禁止对应的断点。如果该位设为 1，当 CPU 在任何任务中检测到满足所定义的断点条件时都会中断；如果该位设为 0，便禁止此断点。与 L0~L3 不同，断点条件发生时，CPU 不会自动清除此位。
LE 和 GE	Local and Global (exact) breakpoint Enable	LE: 8 GE: 9	从 486 开始的 IA-32 处理器都忽略这两位的设置。此前这两位是用来启用或禁止数据断点匹配的。对于早期的处理器，当设置有数据断点时，需要启用本设置，这时 CPU 会降低执行速度，以监视和保证当有指令要访问符合断点条件的数据时产生调试异常
GD	General Detect Enable	13	启用或禁止对调试寄存器的保护。当设为 1 时，如果 CPU 检测到将修改调试寄存器 (DR0~DR7) 的指令时，CPU 会在执行这条指令前产生一个调试异常

通过表 4-1 的定义可以看出，调试控制寄存器的各个位域提供了很多选项允许我们通过不同的位组合定义出各种断点条件。

下面我们先进一步介绍读写域 R/Wn，通过对它的设置，我们可以指定断点的访问类型（又称访问条件），也就是以何种方式（读写数据、执行代码还是 I/O）访问地址寄存器中指定的地址时中断。读写域占两个二进制位，可以指定 4 种访问方式，满足不同调试情况的需要。以下是 3 类典型的使用方式，其中第一类又分两种情况。

读/写内存中的数据时中断：这种断点又被称为数据访问断点（data access breakpoint）。利用数据访问断点，可以监控对全局变量或局部变量的读写操作。例如，当进行某些复杂的系统级调试，或者调试多线程程序时，我们不知道是哪个函数或线程在何时修改了某一变量，这时我们就可以设置一个数据访问断点。以 WinDBG 调试器为例，可以通过 ba 命令来设置这样的断点，如 ba w4 00401200。其中 ba 代表 break on access，w4 00401200 的含义是对地址 00401200 开始的 4 字节内存区进行写操作时中断。如果我们希望有线程或代码读这个变量时也中断，那么只要把 w4 换成 r4 便可以了。现代调试器大多还都支持复杂的条件断点，比如当某个变量等于某个确定的值时中断，这其实也可以用数据访问断点来实现，其基本思路是设置一个数据访问断点来监视这个变量，当每次这个变量的值发生改变时，CPU 都会通知调试器，调试器就会检查这个变量的值，如果不满足规定的条件，就立刻返回让 CPU 继续执行；如果满足，就中断到调试环境。

执行内存中的代码时中断：这种断点又被称为代码访问断点（code access breakpoint）或指令断点（instruction breakpoint）。代码访问断点从实现的功能上看与软件断点类似，都是当 CPU 执行指定地址开始的指令时中断。但是通过寄存器实现的代码访问断点与软件断点相比有个优点，就是不需要像软件断点那样向目标代码中插入断点指令。这个优点在某些情况下非常重要。例如，当我们调试位于 ROM（只读存储器）上的代码（比如 BIOS 中的 POST 程序）时，根本没有办法向那里插入软件断点（INT 3）指令，因为目标内存是只读的。另外，软件断点的另一个局限是，只有当目标代码被加载进内存后才可以向该区域设置软件断点。而调试寄存器断点没有这些限制，因为只要把需要中断的内存地址放入调试地址寄存器（DR0~DR3），并设置好调试控制寄存器（DR7）的相应位就可以了。

读写 I/O（输入输出）端口时中断：这种断点又被称为 I/O 访问断点（Input/Output access breakpoint）。I/O 访问断点对于调试使用输入输出端口的设备驱动程序非常有用。也可以利用 I/O 访问断点来监视对 I/O 空间的非法读写操作，提高系统的安全性。因为某些恶意程序在实现破坏动作时，需要对特定的 I/O 端口进行读写操作。

读写域定义了要监视的访问类型，地址寄存器（DR0~DR3）定义了要监视的起始地址。那么要监视的区域长度呢？这便是长度域 LENn（n=0, 1, 2, 3，位于 DR7 中）

的任务。LENn 位段可以指定 1、2、4 或 8 字节长的范围。

对于代码访问断点，长度域应该为 00，代表 1 字节长度。另外，地址寄存器应该指向指令的起始字节。也就是说，CPU 只会用指令的起始字节来检查代码断点匹配。

对于数据和 I/O 访问断点，有两点需要注意：第一，只要断点区域中的任一字节在被访问的范围内，都会触发该断点。第二，边界对齐要求，2 字节区域必须按字（word）边界对齐；4 字节区域必须按双字（doubleword）边界对齐；8 字节区域必须按 4 字（quadword）边界对齐。也就是说，CPU 在检查断点匹配时会自动去除相应数量的低位。因此，如果地址没有按要求对齐，可能无法实现预期的结果。例如，假设希望通过将 DR0 设为 0xA003，LEN0 设为 11（代表 4 字节长）实现任何对 0xA003~0xA006 内存区的写操作都会触发断点；那么只有当 0xA003 被访问时会触发断点，对 0xA004、0xA005 和 0xA006 处的内存访问都不会触发断点。因为长度域指定的是 4 字节，所以 CPU 在检查地址匹配时，会自动将起始地址 0xA003 的低 4 位屏蔽掉，只是匹配 0xA000。而 0xA004、0xA005 和 0xA006 屏蔽低 2 位后都是 0xA004，所以无法触发断点。

4.2.4 指令断点

关于指令断点还有一点要说明。对于如下所示的代码片段，如果指令断点被设置在紧邻 MOV SS EAX 的下一行，那么该断点永远不会被触发。原因是为了保证栈段寄存器（SS）和栈顶指针（ESP）的一致性，CPU 执行 MOV SS 指令时会禁止所有中断和异常，直到执行完下一条指令。

```
MOV SS, EAX
MOV ESP, EBP
```

类似的，紧邻 POP SS 指令的下一条指令处的指令断点也不会被触发。例如，如果指令断点指向的是下面的第二行指令，那么该断点永远不会被触发。

```
POP SS
POP ESP
```

但是，当有多个相邻的 MOV SS 或 POP SS 指令时，CPU 仅仅只会保证对第一条指令采取如上“照顾”。例如，对于下面的代码片段，指向 MOV ESP, EBP 的指令断点是会被触发的。

```
MOV SS, EDX
MOV SS, EAX
MOV ESP, EBP
```

IA-32 手册推荐使用 LSS 指令来加载 SS 和 ESP 寄存器，通过 LSS 指令，一条指令便可以改变 SS 和 ESP 两个寄存器。

4.2.5 调试异常

IA-32 架构专门分配了两个中断向量来支持软件调试，向量 1 和向量 3。向量 3 用于 INT 3 指令产生的断点异常（breakpoint exception，简称#BP）。向量 1 用于其他情况的调试异常，简称为调试异常（debug exception，简称#DB）。硬件断点产生的是调试异常，所以当硬件断点发生时 CPU 会执行 1 号向量所对应的处理例程。

表 4-2 列出了各种导致调试异常的情况及该情况所产生异常的类型。

表 4-2 导致调试异常的各种情况

异常情况	DR6 标志	DR7 标志	异常类型
因为 EFlags[TF]=1 而导致的单步异常	BS=1		陷阱
调试寄存器 DRn 和 LENn 定义的指令断点	Bn=1 and (Gn=1 or Ln=1)	R/Wn=0	错误
调试寄存器 DRn 和 LENn 定义的写数据断点	Bn=1 and (Gn=1 or Ln=1)	R/Wn=1	陷阱
调试寄存器 DRn 和 LENn 定义的 I/O 读写断点	Bn=1 and (Gn=1 or Ln=1)	R/Wn=2	陷阱
调试寄存器 DRn 和 LENn 定义的数据读（不包括取指）写断点	Bn=1 and (Gn=1 or Ln=1)	R/Wn=3	陷阱
当 DR7 的 GD 位为 1 时，企图修改调试寄存器	BD=1		错误
任务状态段（TSS）的 T 标志为 1 时进行任务切换	BT=1		陷阱

对于错误类调试异常，因为恢复执行后断点条件仍然存在，所以，为了避免反复发生异常，调试软件必须在使用 IRETD 指令返回重新执行触发异常的指令前将标志寄存器（EFlags）的 RF（Resume Flag）位设为 1，告诉 CPU，不要在执行返回后的第一条指令时产生调试异常，CPU 执行完该条指令会自动清除 RF 标志。

4.2.6 调试状态寄存器

调试状态寄存器（DR6）的作用是当 CPU 检测到匹配断点条件的断点或有其他调试事件发生时，用来向调试器的断点异常处理程序传递断点异常的详细信息，以便使调试器可以很容易地识别出发生的是什么调试事件。例如，如果 B0 被置为 1，那么就说明满足 DR0、LEN0 和 R/W0 所定义条件的断点发生了。表 4-3 列出了 DR6 中各个标志位的具体含义。

表 4-3 调试状态寄存器 (DR6)

简称	全称	位位置	描述
B0	Breakpoint 0	0	如果处理器检测到满足断点条件 0 的情况，那么处理器会在调用异常处理程序前将此位置为 1
B1	Breakpoint 1	1	如果处理器检测到满足断点条件 1 的情况，那么处理器会在调用异常处理程序前将此位置为 1
B2	Breakpoint 2	2	如果处理器检测到满足断点条件 2 的情况，那么处理器会在调用异常处理程序前将此位置为 1
B3	Breakpoint 3	3	如果处理器检测到满足断点条件 3 的情况，那么处理器会在调用异常处理程序前将此位置为 1
BD	Debug register access detected	13	这一位与 DR7 的 GD 位相联系，当 GD 位被置为 1，而且 CPU 发现了要修改调试寄存器 (DR0~DR7) 的指令时，那么 CPU 会停止继续执行这条指令，把 BD 位设为 1，然后把执行权交给调试异常 (#DB) 处理程序
BS	Single step	14	这一位与标志寄存器 (EFLAGS) 的 TF (trap flag) 位相联系，如果该位为 1，则表示异常是由单步执行 (single step) 模式触发的。与导致调试异常的其他情况相比，单步情况的优先级最高，因此当此标志为 1 时，也可能有其他标志也为 1
BT	Task switch	15	这一位与任务状态段 TSS 的 T 标志 (调试陷阱标志，debug trap bit) 相联系。当 CPU 在进行任务切换时，当发现下一个任务的 TSS 的 T 标志为 1 时，会设置 BT 位，并中断到调试中断处理程序

因为单步执行、硬件断点等多种情况触发的异常，使用的都是一个向量号即 1 号。所以，调试器需要使用调试状态寄存器来判断到底是什么原因触发的异常。

4.2.7 示例

下面通过一些例子来加深理解。表 4-4 列出了对调试寄存器的设置，通过这些设置我们定义了 4 个硬件断点，表格的最后一列是我们预期的断点触发条件。

表 4-4 断点示例

编号	断点地址	R/W	LEN	预期条件
0	DR0=A0001H	R/W0=11 (读/写)	LEN0=00 (1B)	读写 A0001H 开始的 1 字节
1	DR1=A0002H	R/W1=01 (写)	LEN1=00 (1B)	写 A0002H 开始的 1 字节
2	DR2=B0002H	R/W2=11 (读/写)	LEN2=01 (2B)	读写 B0002H 开始的 2 字节
3	DR3=C0000H	R/W3=01 (写)	LEN3=11 (4B)	写 C0000H 开始的 4 字节

对于上面的调试器设置，表 4-5 列出了一些读写操作 (数据访问)，并说明它们是否会命中断点。

表 4-5 内存访问示例

访问类型	访问地址	访问长度	触发断点是否
读或写	A0001H	1	触发 (与断点 0 匹配)
读或写	A0001H	2	触发 (读与断点 0 匹配，写与断点 0 和 1 都匹配)

续表

访问类型	访问地址	访问长度	触发断点与否
写	A0002H	1	触发(与断点1匹配)
写	A0002H	2	触发(与断点1匹配)
读或写	B0001H	4	触发(与断点2匹配,对B0002和B0003的访问落入断点2定义的区域)
读或写	B0002H	1	触发(与断点2匹配)
读或写	B0002H	2	触发(与断点2匹配)
写	C0000H	4	触发(与断点3匹配)
写	C0001H	2	触发(与断点3匹配)
写	C0003H	1	触发(与断点3匹配)
读或写	A0000H	1	否
读	A0002H	1	否(断点1的访问类型是写)
读或写	A0003H	4	否
读或写	B0000H	2	否
读	C0000H	2	否(断点3定义的访问类型是写)
读或写	C0004H	4	否

表格最后一列说明了断点的命中情况及原因。可以看到一个数据访问可能与多个断点定义的条件相匹配,这时,CPU会设置状态寄存器的多个位,显示出所有匹配的断点。

再举个实际的例子。在WinDBG中打开上一节的HiInt3程序。根据清单4-2可以知道,printf函数所使用的字符串的内存地址是0042201c。考虑到printf函数执行时会访问这个地址,所以我们尝试对其设置断点。但当初始断点命中时执行ba命令,WinDBG会提示设置失败:

```
0:000> ba w1 0042201c
^ Unable to set breakpoint error
The system resets thread contexts after the process
breakpoint so hardware breakpoints cannot be set.
Go to the executable's entry point and set it then.
'ba w1 0042201c'
```

上面的信息表示此时尚不能设置硬件断点,原因是此时进程的初始化尚未完全结束,系统还会刷新线程的上下文,调试寄存器的内容也会被刷新。WinDBG提示先执行到程序的入口,然后再设置硬件断点。可以通过对main函数设置软件断点让程序运行到入口函数。

因为HiInt3程序的main函数开始处已经有一条INT3指令,这等同于一个软件断点。所以我们直接按F5让程序继续执行。待断点如期命中后,在0042201c附近设置如下3个断点:

```
0:000> ba w1 0042201c
0:000> ba r2 0042201e
0:000> ba r1 0042201f
```

设置以上断点后立刻观察调试寄存器:

```
0:000> r dr0,dr1,dr2,dr3,dr6,dr7
dr0=00000000 dr1=00000000 dr2=00000000 dr3=00000000 dr6=00000000 dr7=00000000
```

可见，这时这些断点尚未设置到调试寄存器中，因为调试器是在恢复被调试程序执行时，才把这些寄存器通过线程的上下文设置到CPU的寄存器中的。但使用WinDBG的列断点命令，可以看到已经设置的断点：

```
0:000> bl
0 e 0042201c w 1 0001 (0001) 0:**** HiInt3!`string'
1 e 0042201e r 2 0001 (0001) 0:**** HiInt3!`string'+0x2
2 e 0042201f r 1 0001 (0001) 0:**** HiInt3!`string'+0x3
```

按F5让程序执行，断点1会先命中：

```
Breakpoint 1 hit
eax=0042201e ebx=7ffdd000 ecx=0012fc6c edx=0012fc00 esi=0159f764 edi=0012ff80
eip=004014f9 esp=0012fc48 ebp=0012fefc iopl=0 nv up ei pl nz ac pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000
HiInt3!_output+0x29:
004014f9 884dd8      mov     byte ptr [ebp-28h],cl    ss:0023:0012fed4=65
```

使用kp命令显示栈回溯信息，可以看到当前在执行_output函数，它是被printf函数所调用的。

```
0:000> kp
ChildEBP RetAddr
0012fefc 004010bb HiInt3!_output+0x29 [output.c @ 371]
0012ff28 00401033 HiInt3!printf+0x5b [printf.c @ 60]
0012ff80 00401209 HiInt3!main+0x23 [C:\...\code\chap04\HiInt3\HiInt3.cpp @ 11]
0012ffc0 7c816fff7 HiInt3!mainCRTStartup+0xe9 [crt0.c @ 206]
0012ffff0 00000000 kernel32!BaseProcessStart+0x23
```

观察程序指针的值eip=004014f9，需要注意的是，这并非触发断点的指令。因为数据访问断点是陷阱类断点，所以当断点命中时，触发断点的指令已经执行完毕，程序指针指向的是下一条指令的地址。可以使用ub 004014f9 12命令来观察前面的两条指令：

```
0:000> ub 004014f9 12
HiInt3!_output+0x24 [output.c @ 371]:
004014f4 8b450c      mov     eax,dword ptr [ebp+0Ch]
004014f7 8a08         mov     cl,byte ptr [eax]
```

可见，当前程序指针的前一条指令是mov cl,byte ptr [eax]，其含义是将EAX寄存器的值所指向的一个字节赋给CL寄存器（ECX寄存器的最低字节）。EAX的值是0042201e，因此这条指令是从内存地址0042201e读取一个字节赋给CL的，这正好符合断点1的条件。

此时观察调试寄存器的内容：

```
0:000> r dr0,dr1,dr2,dr3,dr6,dr7
dr0=0042201c dr1=0042201e dr2=0042201f dr3=00000000 dr6=ffff0ff2 dr7=03710515
```

可以看到DR0~DR2存放的是3个断点的地址。DR3还没有使用。为了便于观察，我们把DR6和DR7寄存器的各个位域和取值画在图4-5中。

DR6的位1为1，表明断点1命中。DR7的R/W0为01表明断点0的访问类型为写。R/W1和R/W2为11，表明断点1和断点2的访问类型为读写都命中。LEN1等于01表明2字节访问，LEN0和LEN2等于00，表明是1字节访问。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LEN 3	R/W 3	LEN 2	R/W 2	LEN 1	R/W 1	LEN 0	R/W 0			G D			G E	L E	G 3	L 3	G 2	L 2	G 1	L 1	G 0	L 0								DR7	
00	00	00	11	01	11	00	01	00	00	0	001	0	1	0	0	0	1	0	1	0	1	0	1	0	1	0	1	0	DR7		
										B T	B S	B D										B 3	B 2	B 1	B 0				DR6		
11111111	11111111									0	0	0	011111	11111	0	0	1	0	0	0	1	0	0	1	0	0	1	0	DR6		

00000011 01110001 00000101 00010101

图 4-5 观察 DR6 和 DR7 寄存器的值

按 F5 继续执行，WinDBG 会显示断点 1 和断点 2 都命中：

```
0:000> g
Breakpoint 1 hit
Breakpoint 2 hit
eax=0042201f ebx=7ffdd000 ecx=0012fc6c edx=0012fc00 esi=7c9118f1 edi=0012ff80
eip=004014f9 esp=0012fc48 ebp=0012fefc iopl=0 nv up ei pl nz ac pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000216
HiInt3!_output+0x29:
004014f9 884dd8      mov     byte ptr [ebp-28h],cl      ss:0023:0012fed4=6c
```

EIP 的指针与刚才的相同，因此触发断点的指令与刚才的一样，表明程序在循环执行。从 EAX 的值可以看到，这次访问的内存地址是 0042201f，这刚好落入断点 1 定义范围的第二个字节，断点 2 定义范围的第一个字节。再观察调试寄存器：

```
0:000> r dr0,dr1,dr2,dr3,dr4,dr5,dr6,dr7
dr0=0042201c dr1=0042201e dr2=0042201f dr3=00000000 dr4=ffff0ff6 dr5=03710515 dr6=ffff0ff6 dr7=03710515
```

这次 DR6 的位 2 和位 1 都为 1，表明断点 2 和断点 1 都命中了。

4.2.8 硬件断点的设置方法

只有在实模式或保护模式的内核优先级（ring 0）下才能访问调试寄存器，否则便会导致保护性异常。这是出于安全性的考虑。那么，像 Visual Studio 2005（VS2005）这样的用户态调试器是如何设置硬件断点的呢？答案是通过访问线程的上下文（CONTEXT）数据来间接访问调试寄存器。CONTEXT 结构用来保存线程的执行状态，在多任务系统中，操作系统通过让多个任务轮换运行来使多个程序同时运行。当一个线程被挂起时，包括通用寄存器值在内的线程上下文信息会被保存起来，当该线程恢复执行时，保存的内容又会被恢复到寄存器中。清单 4-4 显示了当使用 VS2005 调试本地的 C++ 程序时，VS2005 调用 SetThreadContext API 来设置调试寄存器的函数调用过程（栈回溯）。

清单 4-4 VS2005 的本地调试器设置调试寄存器的过程

```
0:026> kn 100
# ChildEBP RetAddr
00 07bee11c 5be24076 kernel32!SetThreadContext          // 调用系统 API 设置线程上下文
01 07bee128 5be96b9c NatDbgDE!Win32Debugger::RawSetThreadContext+0x2e
02 07bee410 5be96166 NatDbgDE!SetupDebugRegister+0x14f
03 07bee42c 5be5e5a7 NatDbgDE!DrSetupDebugRegister+0x2a // 设置调试寄存器
```

```

04 07bee44c 5be1d63d NatDbgDE!_HPRCX::SetupDataBps+0x5b // 设置数据断点
05 07bee45c 5be1e7f6 NatDbgDE!AddQueue+0x82
06 07bee474 5be27635 NatDbgDE!_HPRCX::ContinueThread+0x3d
07 07bee4a8 5be2b694 NatDbgDE!SetupSingleStep+0x94 // 设置单步标志
08 07bee4ec 5be2b701 NatDbgDE!StepOverBpAtPC+0xfb // 单步越过断点, 见下文
09 07bee570 5be35eee NatDbgDE!ReturnStepEx+0x196
0a 07bee5cc 5be25f3b NatDbgDE!PushRunningThread+0x93
0b 07beea10 5be25f7f NatDbgDE!ProcessContinueCmd+0x103 // 处理继续运行命令
0c 07beea34 5be12fa2 NatDbgDE!DMLib::DMFunc+0x149 // DM 层的分发函数
0d 07beea44 5be124e9 NatDbgDE!TLClientLib::Local_TLFunc+0x8c
                                         // 转给本地的传输层函数
0e 07beea68 5be12510 NatDbgDE!CallTL+0x33 // 调用传输层
0f 07beea88 5be126c2 NatDbgDE!EMCallBackTL+0x18
10 07beeb0 5be25e83 NatDbgDE!SendRequestX+0x7d
11 07beeaе0 5be25e34 NatDbgDE!Go+0x4a // 执行 Go 命令
12 07bef7dc 5be12496 NatDbgDE!EMFunc+0x53b // EM 层的分发函数
13 07bef804 5be25e19 NatDbgDE!CallEM+0x20 // 调用 EM (执行模型) 层
14 07bef840 5be2603f NatDbgDE!CNativeThread::Go+0x57 // 执行 Go 命令
15 07bef85c 5be26081 NatDbgDE!CDebugProgram::ExecuteEx+0x66 // 命令解析和分发
16 07bef864 77e7a1ac NatDbgDE!CDebugProgram::Execute+0xd
... // 省略多行

```

从上面的栈回溯可以清楚地看到, VS2005 的本地调试引擎 (NatDbgDE) 执行命令 (Go) 的过程, 从 EM (Execution Model) 层到传输层 (Transport Layer), 再到 DM (Debugge Module) 层。最后由 DM 层调用 SetThreadContext API 将调试寄存器设置到线程上下文结构中。我们将在第 28 章介绍 Visual Studio 调试器的分层模型和各个层的细节。

下面给出一个 C++ 例子, 演示如何手工设置数据访问断点, 清单 4-5 列出了这个小程序 (DataBp) 的源代码。

清单 4-5 DataBp 程序的源代码

```

1 // DataBP.cpp : Demonstrate setting data access breakpoint manually.
2 // Raymond Zhang Jan. 2006
3 //
4
5 #include "stdafx.h"
6 #include <windows.h>
7 #include <stdlib.h>
8
9 int main(int argc, char* argv[])
10 {
11     CONTEXT ctxt;
12     HANDLE hThread=GetCurrentThread();
13     DWORD dwTestVar=0;
14
15     if(!IsDebuggerPresent())
16     {
17         printf("This sample can only run within a debugger.\n");
18         return E_FAIL;
19     }
20     ctxt.ContextFlags=CONTEXT_DEBUG_REGISTERS|CONTEXT_FULL;
21     if(!GetThreadContext(hThread,&ctxt))
22     {
23         printf("Failed to get thread context.\n");
24         return E_FAIL;

```

```

25      }
26      ctxt.Dr0=(DWORD) &dwTestVar;
27      ctxt.Dr7=0xF0001;//4 bytes length read& write breakponits
28      if(!SetThreadContext(hThread,&ctxt))
29      {
30          printf("Failed to set thread context.\n");
31          return E_FAIL;
32      }
33
34      dwTestVar=1;
35      GetThreadContext(hThread,&ctxt);
36      printf("Break into debugger with DR6=%X.\n",ctxt.Dr6);
37
38      return S_OK;
39  }

```

第 11 行和第 12 行读取当前线程的 CONTEXT 结构，其中包含了线程的通用寄存器和调试寄存器信息。第 15 行检测当前程序是否正在被调试，如果不是正在被调试，那么当断点被触发时便会导致异常错误。第 26 行是将内存地址放入 DR0 寄存器。第 27 行是设置 DR7 寄存器，F 代表 4 字节读写访问；01 代表启用 DR0 断点。第 28 行通过 SetThreadContext() API 使寄存器设置生效。第 34 行尝试修改内存数据以触发断点。

在 VC6 下运行该程序（不设置任何软件断点），会发现 VC6 停在 dwTestVar=1 的下一行。为什么会停在下一行而不是访问数据这一行呢？正如我们前面所介绍的，这是因为数据访问断点导致的调试异常是一种陷阱类（TRAP）异常，当该类异常发生时，触发该异常的指令已经执行完毕。与此类似，INT 3 指令导致的断点异常也属于陷阱类异常。但是通过调试寄存器设置的代码断点触发的调试异常属于错误类（FAULT）异常，当错误类异常发生时，CPU 会将机器状态恢复到执行导致异常的指令被执行前的状态，这样，对于某些错误类异常，比如页错误（page fault）和除零异常，异常处理例程可以纠正“错误”情况后重新执行导致异常的指令。

4.2.9 归纳

因为以上介绍的断点不需要像软件断点那样向代码中插入软件指令，依靠处理器本身的功能便可以实现，所以人们习惯上把这些使用调试寄存器（DR0~DR7）设置的断点叫硬件断点（hardware breakpoint），以与软件断点区别开来。

硬件断点具有很多优点，但是也有不足，最明显的就是数量限制，因为只有 4 个断点地址寄存器，所以每个 IA-32 CPU 允许最多设置 4 个硬件断点。这基本可以满足大多数情况下的调试需要。

在多处理器系统中，硬件断点是与 CPU 相关的，也就是说，针对一个 CPU 设置的硬件断点并不适用于其他的 CPU。

另一点需要说明的是，可以使用调试寄存器来实现变量监视和数据断点。但并不是所有调试器的数据断点功能都是使用调试寄存器来实现的。举例来说，WinDBG 的

ba 命令及 VS2005 的 C/C++ 调试器都是使用调试寄存器来设置数据断点的，但是 VC6 调试器不是这样做的。一旦设置并启用了数据断点，VC6 调试器便会记录下每个变量的当前值，然后以单步的方式恢复程序执行（下一节将详细讨论单步标志）。这样，被调试程序执行一条汇编指令后便会因为调试异常而中断到 VC6 调试器，VC6 调试器收到调试事件后会读取断点列表中的每个数据变量的当前值，并与它们的保存值相比较，如果发生变化，那么就说明该断点命中，VC6 会显示图 4-6 所示的对话框。如果没有变化，那么便再设置单步标志，让被调试程序再执行一条指令。

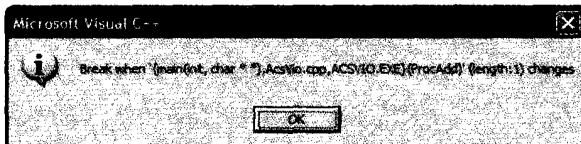


图 4-6 VC6 显示的数据断点命中对话框

当显示以上对话框时，修改变量的那条指令已经执行完毕，所以按 OK 后，调试器显示的执行位置箭头指向的是导致变量变化的代码的下一行。

由于 VC6 的数据断点功能不是使用调试寄存器设置的，所以没有数量限制，但这种实现方法的明显缺点是效率低，会导致被调试程序的运行速度变慢。

4.3 陷阱标志

在 4.1 节和 4.2 节中，我们分别介绍了通过 INT 3 指令设置的软件断点和通过调试寄存器设置的硬件断点。无论是软件断点还是硬件断点，其目的都是使 CPU 执行到指定位置或访问指定位置时中断到调试器。除了断点，还有一类常用的方法使 CPU 中断到调试器，这便是调试陷阱标志（debug trap flag）。可把陷阱标志想象成一面“令旗”，当有陷阱标志置起时，CPU 一旦检测到符合陷阱条件的事件发生，就会报告调试异常通知调试器。IA-32 处理器所支持的调试陷阱标志共有以下 3 种。

- 8086 支持的单步执行标志（标志寄存器 EFLAGS 的 TF 位）。
- 386 引入的任务状态陷阱标志（任务状态段 TSS 的 T 标志）。
- 奔腾 Pro 引入的分支到分支单步执行标志（DebugCtl 寄存器中的 BTF 标志）。

下面分别详细介绍。

4.3.1 单步执行标志（TF）

在 x86 系列处理器的第一代产品 8086 CPU 的程序状态字 PSW（Program Status Word）寄存器中有一个陷阱标志位（bit 8），名为 Trap Flag，简称 TF。当 TF 位为 1 时，CPU 每执行完一条指令便会产生一个调试异常（#DB），中断到调试异常处理程序，

调试器的单步执行功能大多是依靠这一机制来实现的。从 80286 开始，程序状态字寄存器被改称为标志寄存器（FLAGS），80386 又将其从 16 位扩展到 32 位，简称为 EFLAGS，但都始终保持着 TF 位。

调试异常的向量号是 1，因此，设置 TF 标志会导致 CPU 每执行一条指令后，都转去执行 1 号异常的处理例程。在 8086 和 286 时代，这个处理例程是专门用来处理单步事件的。从 386 开始，当硬件断点发生时也会产生调试异常，调用 1 号服务例程，但可利用调试状态寄存器（DR6）来识别发生的是何种事件。为了表达方便，我们把因 TF 标志触发的调试异常称为单步异常（single-step exception）。

单步异常属于陷阱类异常（3.2 节介绍了异常的 3 种类别），也就是 CPU 总是在执行完导致此类异常的指令后才产生该异常。这意味着当因单步异常中断到调试器中时，导致该异常的指令已经执行完毕了。软件断点异常（#BP）和硬件断点中的数据及 I/O 断点异常也是陷阱类异常，但是硬件断点中的指令访问异常是错误类异常，也就是当由于此异常而中断到调试器时，相应调试地址寄存器（DRn）中所指地址处的指令还没有执行。这是因为 CPU 是在尝试（attempts）执行下一条指令时进行此类断点匹配的。

CPU 是何时检查 TF 标志的呢？IA-32 手册的原文是“while an instruction is being executed”（IA-32 手册卷 III 的 15.3.1.4 Single-Step Exception Condition），也就是在执行一个指令的过程中。尽管没有说过程中的哪个阶段（开始、中间还是末尾），可以推测应该是一条指令即将执行完毕的时候。也就是说，当 CPU 在即将执行完一条指令的时候检测 TF 位，如果该位为 1，那么 CPU 就会先清除此位，然后准备产生异常。但是这里有个例外，对于那些可以设置 TF 位的指令（例如 POPF），CPU 不会在执行这些指令期间做以上检查。也就是说，这些指令不会立刻产生单步异常，而是其后的下一条指令将产生单步异常。

因为 CPU 在进入异常处理例程前会自动清除 TF 标志，因此，当 CPU 中断到调试器中后再观察 TF 标志，它的值总是 0。

既然调试异常的向量号是 1，可不可以像 INT 3 那样通过在代码中插入 INT 1 这样的指令来实现手工断点呢？对于应用程序，答案是否定的。INT 3 尽管具有 INT n 的形式，但是它具有独特的单字节机器码，而且其作用就是产生一个断点异常（breakpoint exception，#BP）。因此系统对其有特别的对待，允许其在用户模式下执行。而 INT 1 则不然，它属于普通的 INT n 指令，机器码为 0xCD01。在保护模式下如果执行 INT n 指令时当前的 CPL 大于引用的门描述符的 DPL，那么便会导致通用保护异常（#GP）。在 Windows 2000 和 XP 这样的操作系统下，INT 1 对应的中断门描述符的 DPL 为 0，这就要求只有内核模式的代码才能执行 INT 1 指令，访问该中断门。也就是说，用户模式下的应用程序没有权利使用 INT 1 指令。一旦使用，就会导致一个一般保护性异常（#GP），Windows 会将其封装为一个访问违例错误（见图 4-7）。在内核模式下，可以在代码（驱动程序）中写入 INT 1 指令，CPU 执行到该指令时会转去执行 1 号向量

对应的处理例程，如果在使用 WinDBG 进行内核级调试，那么会中断到 WinDBG 中，WinDBG 会以为是发生了一个单步异常。

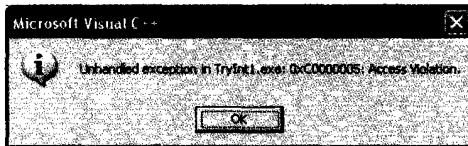


图 4-7 应用程序中执行 INT 1 指令会导致一般保护性异常

4.3.2 高级语言的单步执行

下面谈谈调试高级语言时的单步机制。由于高级语言的一条语句通常都对应多条汇编指令，例如，在清单 4-6 中，C++的一条语句 `i=a+b*c+d/e+f/g+h;` 对应于 15 条汇编语句。因此可容易想到单步执行这条 C++语句的几种可能方法。第一种是用 TF 标志一步步地走过每条汇编指令，这种方法意味着会产生 15 次调试异常，CPU 中断到调试器 15 次，不过中间的 14 次都是简单地重新设置起 TF 标志，便恢复被调试程序执行，不中断给用户。第二种方法是在 C++语句对应的最后一条汇编指令处动态地插入一条 INT 3 指令，让 CPU 一下子跑到那里，然后再单步一次将被替换的那条指令执行完，这种方法需要 CPU 中断到调试器两次。第三种方法是在这条 C++语句的下一条语句的第一条汇编指令处（即第 18 行）替换入一个 INT 3，这样 CPU 中断到调试器一次就可以了。

清单 4-6 高级语言的单步跟踪

1	10:	<code>i=a+b*c+d/e+f/g+h;</code>	
2	00401028	8B 45 F8	mov eax,dword ptr [ebp-8]
3	0040102B	0F AF 45 F4	imul eax,dword ptr [ebp-0Ch]
4	0040102F	8B 4D FC	mov ecx,dword ptr [ebp-4]
5	00401032	03 C8	add ecx,eax
6	00401034	8B 45 F0	mov eax,dword ptr [ebp-10h]
7	00401037	99	cdq
8	00401038	F7 7D EC	idiv eax,dword ptr [ebp-14h]
9	0040103B	8B F0	mov esi,eax
10	0040103D	03 75 E0	add esi,dword ptr [ebp-20h]
11	00401040	8B 45 E8	mov eax,dword ptr [ebp-18h]
12	00401043	99	cdq
13	00401044	F7 7D E4	idiv eax,dword ptr [ebp-1Ch]
14	00401047	03 F1	add esi,ecx
15	00401049	03 C6	add eax,esi
16	0040104B	89 45 DC	mov dword ptr [ebp-24h],eax
17	11:	<code>j=i;</code>	
18	0040104E	8B 55 DC	mov edx,dword ptr [ebp-24h]
19	00401051	89 55 D8	mov dword ptr [ebp-28h],edx
20	12:		
21	13:	<code>if(a)</code>	
22	0040D6C4	83 7D FC 00	cmp dword ptr [ebp-4],0
23	0040D6C8	74 0B	je main+55h (0040d6d5)
24	14:	<code>i+=a;</code>	
25	0040D6CA	8B 45 DC	mov eax,dword ptr [ebp-24h]
26	0040D6CD	03 45 FC	add eax,dword ptr [ebp-4]

```

27 0040D6D0 89 45 DC      mov    dword ptr [ebp-24h],eax
28 15:      else if(b)      jmp    main+75h (0040d6f5)
29 0040D6D3 EB 20          cmp    dword ptr [ebp-8],0
30 0040D6D5 83 7D F8 00    je     main+66h (0040d6e6)
31 0040D6D9 74 0B          i+=b;
32 16:      i+=b;          mov    ecx,dword ptr [ebp-24h]
33 0040D6DB 8B 4D DC      add    ecx,dword ptr [ebp-8]
34 0040D6DE 03 4D F8      mov    dword ptr [ebp-24h],ecx
35 0040D6E1 89 4D DC      mov    17:      else if(c)
36

```

后两种方法较第一种方法速度会快很多，但是不幸的是，并不能正确地预测出高级语言对应的最后一条指令和下一条语句的开始指令（要替换为 INT 3 的那一条指令）。比如对于第 28 行的 else if (b) 语句，就很难判断出它对应到最后一条汇编语句和下一条高级语言语句的起始指令。因此，今天的大多数调试器当进行高级语言调试时都是使用第一种方法来实现单步跟踪的。

关于 TF 标志，还有一点值得注意，INT n 和 INTO 指令会清除 TF 标志，因此调试器在单步跟踪这些指令时，必须做特别处理。

4.3.3 任务状态段陷阱标志

除了标志寄存器中的陷阱标志（TF）位以外，386 还引入了一种新的调试陷阱标志，任务状态段（TSS）中的 T 标志。任务状态段（Task-State Segment）用来记录一个任务（CPU 可以独立调度和执行的程序单位）的状态，包括通用寄存器的值、段寄存器的值和其他重要信息。当任务切换时，当前任务的状态会被保存到这个内存段里。当要恢复执行这个任务时，系统会根据 TSS 中的保存记录把寄存器的值恢复回来。

在 TSS 中，字节偏移为 100 的 16 位字（word）的最低位是调试陷阱标志位，简称 T 标志。如果 T 标志被置为 1，那么当 CPU 切换到这个任务时，便会产生调试异常。准确地说，CPU 是在将程序控制权转移到新的任务，但还没有开始执行新任务的第一条指令时产生异常的。调试中断处理程序可以通过检查调试状态寄存器（DR6）的 BT 标志来识别出发生的是不是任务切换异常。值得注意的是，如果调试器接管了调试异常处理，而且该处理例程属于一个独立的任务，那么一定不要设置该任务的 TSS 段中的 T 位；否则便会导致死循环。

4.3.4 分支到分支单步执行标志（BTF）

在 IA-32 处理器家族中，所有的 Pentium Pro、Pentium II 和 Pentium III 处理器，包括相应的 Celeron（赛扬）和 Xeon（至强）版本，因为都是基于相同的 P6 内核（Core）而被统称为 P6 处理器。P6 处理器引入了一项新的对调试非常有用的功能：监视和记录分支、中断和异常，以及针对分支单步执行（Single-step on branch）。奔腾 4 处理器对这一功能又做了很大的增强。下面具体介绍按分支单步执行的功能和使用方法。第 5 章将介绍分支、中断和异常记录功能。

首先解释分支到分支单步执行的含义。前面我们介绍过，当 EFlags 寄存器的 TF 位为 1 时，CPU 每执行完一条指令便会中断到调试器，也就是以指令为单位单步执行。顾名思义，针对分支单步执行就是以分支为单位单步执行，换句话说，每步进（step）一次，CPU 会一直执行到有分支、中断或异常发生。

那么，如何启用按分支单步执行呢？简单来说，就是要同时置起 TF 和 BTF 标志。TF 标志位于 EFlags 寄存器中，大家已经很熟悉。BTF 标志位于 MSR（Model Specific Register）寄存器中，在 P6 处理器中，这个 MSR 的名字叫 DebugCtlMSR，在奔腾 4 处理器中被称为 DebugCtlA，在奔腾 M 处理器中被称为 DebugCtlB。BTF 位是在这些 MSR 寄存器的位 1 中。

下面结合清单 4-7 中的代码进行说明。

清单 4-7 按分支单步执行

```

1 #define DEBUGCTRL_MSR 0x1D9
2 #define BTF 2
3 int main(int argc, char* argv[])
4 {
5     int m,n;
6     MSR_STRUCT msr;
7
8     if(!EnablePrivilege(SE_DEBUG_NAME)
9         || !EnsureVersion(5,1)
10        || !GetSysDbgAPI())
11    {
12        printf("Failed in initialization.\n");
13        return E_FAIL;
14    }
15    memset (&msr,0,sizeof(MSR_STRUCT));
16
17    msr.MsrNum=DEBUGCTRL_MSR;
18    msr.MsrLo|=BTF;
19    WriteMSR(msr);
20
21    //以下代码将全速运行
22    m=10,n=2;
23    m=n*2-1;
24    if (m==m*m/m)
25        m=1;
26    else
27    {
28        m=2;
29    }
30    //一次可以单步到这里
31    m*=m;
32
33    if (ReadMSR(msr))
34    {
35        PrintMsr(msr);
36    }
37    else
38        printf("Failed to ReadMSR().\n");
39
40    return S_OK;
41 }
```

在 VC6 的 IDE 环境下（系统的 CPU 应该是 P6 或更高），先在第 22 行设置一个断

点，然后按 F5 运行到这个断点位置。第 19 行是用来启用按分支单步执行功能的，也就是设置起 BTF 标志，细节我们等一下再讲。接下来，我们按 F10 单步执行，会发现一下子会执行到第 31 行。也就是从第 22 行单步一次就执行到了第 31 行，这便是按分支单步执行的效果。那么为什么会执行到第 31 行呢？按照分支到分支单步执行的定义，CPU 会在执行到下一次分支、中断或异常发生时停止。对于我们的例子，CPU 在执行第 22 行对应的第一条汇编指令时，CPU 会检测到 TF 标志（因为我们是按 F10 单步执行的，所以 VC6 会帮助我们设置 TF 标志），此外，P6 及以后的 IA-32 CPU 还会检查 BTF 标志，当发现 BTF 标志也被置起时，CPU 会认为当前是在按分支单步执行，所以会判断当前指令是否是分支指令，如果不是，CPU 便会继续执行。依此类推，CPU 会连续执行到第 24 行的 if 语句对应的第一条汇编指令 jne（参见清单 4-8）。因为这条语句是分支语句，所以，当 CPU 在执行这条指令的后期检查 TF 和 BTF 标志时，会认为已经满足停止执行的条件，在清除 TF 和 BTF 标志后，就产生单步异常中断到调试器。因为 EIP 总是指向即将要执行的指令，所以 VC6 会将当前位置设到行 31，而不是行 24。也就是说中断到调试器时，分支语句已经执行完毕，但是跳转到的那条语句（即清单 4-7 中的行 31）还没有执行。

清单 4-8 第 24 行的汇编代码

1	128:	if (m==m*m/m)		
2	0040DBBB	8B 45 FC	mov	eax,dword ptr [ebp-4]
3	0040DBBE	0F AF 45 FC	imul	eax,dword ptr [ebp-4]
4	0040DBC2	99	cdq	
5	0040DBC3	F7 7D FC	idiv	eax,dword ptr [ebp-4]
6	0040DBC6	39 45 FC	cmp	dword ptr [ebp-4],eax
7	0040DBC9	75 09	jne	main+0B4h (0040dbd4)

对以上过程还有几点需要说明。

如果在从第 22 行执行到第 24 行的过程中，有中断或异常发生，那么 CPU 也会认为停止单步执行的条件已经满足。因此，按分支单步执行的全称是按分支、异常和中断单步（single-step on branches, exceptions and interrupts）。

CPU 认为有分支发生的条件是执行以下分支指令：JMP（无条件跳转）、Jcc（各种条件跳转指令，如 JA、JAE、JNE 等）、LOOP（循环）和 CALL（函数/过程调用）。

由于只有内核代码才能访问 MSR 寄存器（通过 RDMSR 和 WRMSR 指令），所以在上面的例子中，在 WriteMSR() 函数中使用了一个未公开的 API ZwSystemDebugControl() 来设置 BTF 标志。

4.4 实模式调试器例析

前面几节我们介绍了 IA-32 CPU 的调试支持，本节我们将介绍两个实模式下的调

试器，看它们是如何利用 CPU 的调试支持实现各种调试功能的。

4.4.1 Debug.exe

20世纪80年代和90年代初的个人电脑大多安装的是DOS操作系统。很多在DOS操作系统下做过软件开发的人都使用过DOS系统自带的调试器Debug.exe。它体积小巧（DOS 6.22附带的版本为15718字节），只要有基本的DOS环境便可以运行；但它功能却非常强大，具有汇编、反汇编、断点、单步跟踪、观察/搜索/修改内存、读写IO端口、读写寄存器、读写磁盘（按扇区）等功能。

在今天的Windows系统中，仍保留着这个程序，位于system32目录下。在运行对话框或命令行中都可以通过输入debug而启动这个经典的实模式调试器。Debug程序启动后，会显示一个横杠，这是它的命令提示符。此时就可以输入各种调试命令了，Debug的命令都是一个英文字母（除了用于扩展内存的X系列命令），附带0或多个参数。比如可以使用L命令把磁盘上的数据读到内存中，使用G命令让CPU从指定的内存地址开始执行，等等。输入问号（?）可以显示出命令清单和每个命令的语法（见清单4-9）。

清单4-9 Debug调试器的命令清单

-?		
assemble	A [address]	; 汇编*
compare	C range address	; 比较两个内存块的内容
dump	D [range]	; 显示指定内存区域的内容
enter	E address [list]	; 修改内存中的内容
fill	F range list	; 填充一个内存区域
go	G [=address] [addresses]	; 设置断点并从=号的地址执行**
hex	H value1 value2	; 显示两个参数的和及差
input	I port	; 读指定端口
load	L [address] [drive] [firstsector] [number]	; 读磁盘数据到内存
move	M range address	; 复制内存块
name	N [pathname] [arglist]	; 指定文件名，供L和W命令使用
output	O port byte	; 写IO端口
proceed	P [=address] [number]	; 单步执行，类似于Step Over
quit	Q	; 退出调试器
register	R [register]	; 读写寄存器
search	S range list	; 搜索内存
trace	T [=address] [value]	; 单步执行，类似于Step Into
unassemble	U [range]	; 反汇编
write	W [address] [drive] [firstsector] [number]	; 写内存数据到磁盘
allocate expanded memory	XA [#pages]	; 分配扩展内存
deallocate expanded memory	XD [handle]	; 释放扩展内存
map expanded memory pages	XM [Lpage] [Ppage] [handle]	; 映射扩展内存页
display expanded memory status	XS	; 显示扩展内存状态

* 也就是将用户输入的汇编语句翻译为机器码，并写到内存中，地址参数用来指定存放机器码的起始内存地址。

** 如果不指定“=”参数，那么便从当前的CS:IP寄存器的值开始执行。第二个参数可以是多个地址值，调试器会在这些地址的内存单元替换为INT 3指令的机器码0xCC。

上面的第一列是命令的用途(主要功能),第二列是命令的关键字,不区分大小写,后面是命令的参数。双分号后的部分是笔者加的中文说明。

纵观这个命令清单,虽然命令的总数不多,不算后面的4个用于扩展内存的命令,只有19个,但是这些命令囊括了所有的关键调试功能。

其中L和W命令既可以读写指定的扇区,也可以读写N命令所指定的文件名。以下是debug程序的几种典型用法。

- 当启动debug时,在命令行参数中指定要调试的程序,如debug debuggee.com。这样,Debug程序启动后会自动把被调试的程序也加载到内存中。因为是实模式,所以它们都在一个内存空间中。我们稍后再详细讨论这一点。
- 不带任何参数启动debug,然后使用N命令指定要调试的程序,再执行L命令将其加载到内存中,并开始调试。
- 不带任何参数启动debug,然后使用它的L命令直接加载磁盘的某些扇区,比如当调试启动扇区中的代码和主引导扇区中的代码(MBR)时,通常使用这种方法。
- 不带任何参数启动debug,然后使用它的汇编功能,输入汇编指令,然后执行,这适用于学习和试验。

Debug程序是我们接下来要介绍的8086 Monitor程序的DOS系统版本,将在介绍8086 Monitor之后一起介绍它们的关键实现。

4.4.2 8086 Monitor

DOS操作系统的最初版本是由被称为DOS之父的Tim Paterson先生设计的。开始时间是1980年的4月,第一个版本QDOS 0.10于1980年8月开始发售。Tim Paterson当时所工作的公司是Seattle Computer Products,简称SCP。

在此如此快的时间内完成一个操作系统,速度可以说是惊人的。其原因当然离不开设计者的技术积累。而其中非常关键的应该是Tim Paterson从1979年开始设计的Debug程序的前身,即8086 Monitor。

8086 Monitor是与SCP公司的8086 CPU板一起使用的一个调试工具,表4-6列出了1.4A版本的8086 Monitor的所有命令。

表4-6 8086 Monitor的命令(1.4A版本)

命令	功能
B <ADDRESS>...<ADDRESS>	启动,读取磁盘的0道0扇区到内存并执行
D <ADDRESS> <RANGE>	显示指定内存地址或区域的内容
E <ADDRESS> <LIST>	编辑内存
F <ADDRESS> <LIST>	填充内存区域
G <ADDRESS>...<ADDRESS>	设置断点并执行
I <HEX4>	从I/O端口读取数据

续表

命令	功能
M <RANGE> <ADDRESS>	复制内存块
O <HEX4> <BYTE>	向 I/O 端口输出数据
R [REGISTER NAME]	读取或修改寄存器
S <RANGE> <LIST>	搜索内存
T [HEX4]	单步执行

从以上命令可以看出，8086 Monitor 已经具有了非常完备的调试功能。把这些命令与清单 4-9 所示的 Debug 程序的命令相比，大多数关键命令都已经存在了。

8086 Monitor 是在 1979 年初开始开发的，1.4 版本的时间是 1980 年 2 月。Tim Paterson 先生在给作者的邮件中讲述了他最初开发 8086 Monitor 时的艰辛。因为没有其他调试器和逻辑分析仪可以使用，他只好使用示波器来观察 8086 CPU 的信号，以此来了解 CPU 的启动时序和工作情况。因此，开发 8086 Monitor 不仅为后来开发 DOS 准备了一个强有力的工具，而且让 Tim Paterson 对 8086 CPU 和当时的个人计算机系统了如指掌。这些基础对于后来 Tim Paterson 能在两个多月里完成 DOS 的第一个版本起到了重要作用。

事实上，Windows NT 的开发团队也是在开发的初期就开发了 KD 调试器，并一直使用这个调试器来辅助整个开发过程。我们将在第 6 篇详细介绍 KD 调试器。

4.4.3 关键实现

在对 Debug 和 8086 Monitor 的基本情况有所了解后，下面我们看看它们的主要功能是如何实现的。

8086 Monitor 是完全使用汇编语言编写的。整个程序的机器码大约有 2000 多个字节，可谓是非常精炼。在 Tim Paterson 先生目前公司的网站上有 8086 Monitor 程序的汇编代码清单。其链接为：http://www.patersontech.com/dos/Docs/Mon_86_1.4a.pdf。以下讨论将结合这份清单中的代码，为了节约篇幅，我们只引用关键的代码段，并在括号中标出页码，建议读者将上面的文件打印出来对照阅读。

因为在实模式下的系统中只有一个任务在运行，所以调试器和被调试程序的代码和数据都是在一个内存空间中的，而且这个空间中的地址就是物理地址。为了避免冲突，调试器和被调试程序各自使用不同的内存区域。以 8086 Monitor 为例，它使用从 0xFF80 开始的较高端内存空间，把低端留给被调试程序。

调试器与被调试程序在一个内存空间中为实现很多调试功能提供了很大的便利。例如，对于所有与内存有关的命令，内存地址不需要做任何转换就可以直接访问。当设置断点时，也可以直接把断点指令写到被调试程序的代码中。这与多任务操作系统下的情况完全不同，在多任务系统中，调试器与被调试程序各自属于不同的内存空间，调试器需要借助操作系统的支持来访问被调试程序的空间。

为了响应调试异常，8086 Monitor 会改写中断向量表的表项 1（地址 4~7）、3（地址 0xC~0xF）和 19H（地址 0x64~0x67），分别对应于 INT 1、INT 3 和 INT 19H。INT 1 用于处理单步执行时的异常，INT 3 用于处理断点异常，INT 19H 用于串行口通信接收命令。

下面以断点为例来介绍调试器的工作过程。当 CPU 执行到断点指令（INT 3）时，会转去执行中断向量表中 3 号表项所指向的代码。当 8086 Monitor 初始化时，已经将其指向标号 BREAKFIX 所开始的代码（Mon_86_1.4a.pdf 的 27 页），即：

```
BREAKFIX:  
EXHG SP, BP  
DEC [BP]  
XCHG SP, BP
```

在跳转到异常处理的代码前，CPU 把当时的程序指针寄存器的值保存在栈中，因此以上 3 条指令的作用是将放在栈顶的程序指针寄存器的值减 1 后再放回去，减 1 的目的是使其恢复为 INT 3 指令指向的值，也就是执行 INT 3 指令，同时也就是设置断点的位置。

以上 3 行的下面便是标号 REENTER 开始的代码（也是 27 页），这也是 INT 1 和 INT 19H 的处理器入口。这样便很自然地实现了 3 个异常处理代码的共享。

REENTER 代码块首先将当前寄存器的值保存到变量中。调试时 R 命令显示的寄存器值都是从这些变量中读取的。也就是说，这些变量的作用与 Windows 系统中的 CONTEXT 结构的作用是一样的。

接下来，调用 CRLF 开始一个新的行，调用 DISPREG 显示寄存器的值，然后对变量 TCOUNT 递减 1，TCOUNT 用于记录 T 命令的参数，即单步执行的指令条数，如果 TCOUNT 不等于 0，那么就跳到 STEP1（27 页）去再单步执行一次。否则，判断 BRKCNT 变量，检查当前的断点个数，如果大于 0，那么就自然向下执行清除断点的代码（标号 CLEANBP），也就是将设置断点用的断点指令恢复成本来的指令内容。当恢复断点时，或者当 BRKCNT 等于 0 时，便跳转到标号 COMMAND（13 页），等待用户输入命令，开始交互式调试。

当用户输入命令后，调试器（8086 Monitor）会根据一个命令表来跳转到处理该命令的代码。执行完一个命令并显示结果后，调试器会等待下一个命令，直到接收到恢复程序执行的命令 T 或 G。以 G 命令为例，它可以跟最多 10 个地址参数，用来定义最多 10 个断点。调试器会依次解析每个地址，然后将其保存到内部的断点表中，而后将断点地址处的一个字节保存起来，并替换成 0xCC（即 INT 3 指令）。设置断点后（或 G 命令没有带参数，不需要设置断点），调试器会将 TCOUNT 命令设置为 1，然后跳转到 EXIT 标号（26 页）所代表的用于异常返回的代码。

EXIT 代码会先设置异常向量，然后把保存在变量中的寄存器内容恢复到物理寄存

器中，最后把变量 FSAVE、CSSAVE 和 IPSAVE 的值压入到栈中，然后执行中断返回指令 IRET。

```
MOV SP, [SPSAVE]
PUSH [FSAVE]
PUSH [CSSAVE]
PUSH [IPSAVE]
MOV DS,[DSSAVE]
IRET
```

FSAVE 变量用于保存标志寄存器的值，CSSAVE 和 IPSAVE 分别用于保存段寄存器和程序指针寄存器的值。当产生异常时，CPU 便会把这 3 个寄存器的值压入到栈中，当异常返回时，CPU 是从栈中读取这 3 个寄存器的值，赋给 CPU 中的对应寄存器，然后从 CS 和 IP（程序指针）寄存器指定的地址开始执行。因为标志寄存器和 IP 寄存器的特殊作用，8086 架构没有设计直接对标志寄存器和程序指针寄存器赋值的指令，修改它们的最主要方式就是当中断返回时通过栈来间接修改。因为在调试器中可以修改 FSAVE、CSSAVE 和 IPSAVE 变量，因此可在调试器中通过修改这 3 个变量来影响恢复执行时它们的值。单步执行命令就是通过设置 FSAVE 的 TF 标志而实现的。通过修改 IPSAVE 变量可以达到改变执行位置的目的，让程序“飞跃”到任意的地址恢复执行。

本节我们简要介绍了实模式下的调试器的实现方法。因为是单任务环境，所以实现比较简单。在保护模式和 Windows 这样的多任务操作系统下，因为涉及到任务之间的界限和用户态及内核态的界限，所以要实现调试变得复杂很多，调试器必须与操作系统相互配合。我们将在后面的章节中逐步介绍。

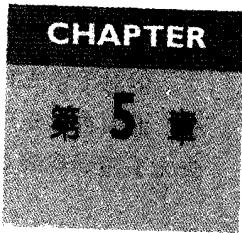
4.5 本章总结

本章使用了较大的篇幅，详细介绍了 CPU 对断点（4.1 节和 4.2 节）和单步执行（4.3 节）这两大关键调试功能的支持。4.4 节以实模式调试器为例，介绍了调试器是如何使用这些支持来实现有关功能的。

下一章我们将介绍 IA-32 CPU 的分支记录和性能监视机制。

参考文献

1. IA-32 Intel® Architecture Software Developer's Manual Volume 3. Intel Corporation
2. 8086 Monitor Instruction Manual. Seattle Computer Products Inc.



分支记录和性能监视

沿着正确的轨迹执行是软件正常工作的必要条件。很多软件错误都是因为程序运行到了错误的分支。尽管这通常不是错误的根本原因，但却是可以顺藤摸瓜的直接线索。因此，了解软件的运行轨迹对于寻找软件问题的根源有着重要意义。因为很多性能问题是因为执行了不必要的代码或循环而导致的，所以运行轨迹对于分析软件的运行效率和软件调优也有着重要意义。

如何记录软件的运行轨迹呢？因为 CPU 是软件的执行者，每一条指令不论是顺序执行还是分支和跳转，都是由它来执行的，所以让 CPU 来记录软件的运行轨迹是最适合的。

CPU 的设计者们早已经意识到了这一点。上一章我们介绍过 P6 处理器引入的按分支单步执行的功能，其实该功能是基于一个更基本的功能，那就是监视和记录分支（Branch）、中断（Interrupt）和异常（Exception）事件，简称为分支监视和记录。奔腾 4 处理器对这一功能又做了很大的增强，允许将分支信息记录到内存中一块被称为 BTS（Branch Trace Buffer）的缓存区中，称为调试存储机制（Debug Store）。

本节将首先介绍分支监控功能的意义和一般方式（5.1 节），而后详细介绍 P6 处理器引入的基于寄存器的分支记录功能（5.2 节），以及奔腾 4 处理器引入的基于内存的调试存储机制（5.3 节）。在第 5.4 节中我们将编写一个名为 CpuWhere 的小程序来演示调试存储机制的用法。最后我们将简要地介绍性能监视机制（5.5 节）。

5.1 分支监视概览

分支监视是跟踪和记录 CPU 执行路线（history）的基本措施，对软件优化和软件调试都有着至关重要的地位。在 CPU 没有集成内部高速缓存（Cache）之前，所有的内存读写操作都要通过前端总线进行。因此，可以使用逻辑分析仪等工具监视到 CPU 的所有内存读写动作，特别是取指动作（从某一内存地址读取指令）。特别的，在 CPU 执行分支指令后，可以通过分析它接下来取指的地址知道 CPU 执行了哪个分支。也就是说，

对于内部没有高速缓存的传统处理器，可以通过观察前端总线观察 CPU 的执行路线。

但对于集成有高速缓存的处理器来说（图 5-1），CPU 是成批地将代码读入高速缓存，而后再从高速缓存中读取指令并解码和执行。这使得位于前端总线上的调试工具失去了精确观察 CPU 所有取指操作的能力，不能像以前那样观察到 CPU 的执行轨迹。

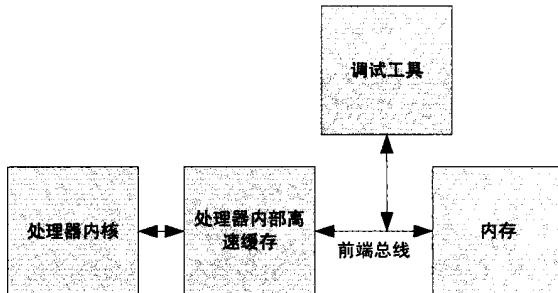


图 5-1 位于前端总线上的调试工具

为了解决以上问题，奔腾处理器引入了一种专门的总线事务（bus transaction），称为 Branch Trace Message（分支踪迹消息），简称为 BTM。在 BTM 功能被启用后（后文讨论），当 CPU 每次执行分支和改变执行路线时，CPU 都会发起一个 BTM 事务，将分支信息发送到前端总线上。这样，调试工具便可以通过监听和读取 BTM 消息跟踪 CPU 的执行路线了。

因为硬件调试工具的价格通常都比较昂贵，而且设置和使用也都比较麻烦，所以 P6 处理器引入了通过内部寄存器来记录分支信息的功能。这样，只要进行必要的设置，CPU 便会把分支信息记录到特定的寄存器中。寄存器可以记录的信息毕竟有限，于是奔腾 4 处理器引入了通过特定的内存区域来记录分支信息的功能。后面两节我们将分别讨论这两种分支监控机制。

5.2 使用寄存器的分支记录

这一节我们先介绍使用 MSR 寄存器来记录分支的方法。P6 处理器最先引入了这种方法，可以记录最近一次分支的源和目标地址，称为 Last Branch Recording，简称为 LBR。奔腾 4 处理器对其做了增强，增加了寄存器个数，以栈的方式可以保存多个分支记录，称为 LBR 栈（LBR Stack）。下面我们依次介绍。

5.2.1 LBR

P6 处理器设计了 5 个 MSR 寄存器（Machine/Model Specific Registers）用来实现 LBR 机制。这 5 个 MSR 寄存器是。

- 用来记录分支的 LastBranchToIP 和 LastBranchFromIP 寄存器对。

- 用来记录异常的 LastExceptionToIP 和 LastExceptionFromIP 寄存器对。
- 一个 MSR 寄存器来控制新加入的调试功能，称为 DebugCtl，其格式如图 5-2 所示。

当发生分支时，LastBranchFromIP 用来记录分支指令的地址，LastBranchToIP 用来记录这个分支要转移到到达的目标地址。

当异常（调试异常除外）或中断发生时，CPU 会先把 LastBranchToIP 和 LastBranchFromIP 中的内容分别复制到 LastExceptionToIP 和 LastExceptionFromIP 寄存器中，然后，再把发生异常或中断时被打断的地址更新到 LastBranchFromIP，把异常或中断处理程序的地址更新到 LastBranchToIP 寄存器中。

31	7	6	5	4	3	2	1	0
保留	T R	P B	P B	P B	P B	B T	L B	R

图 5-2 DebugCtl MSR 寄存器（P6 处理器）

虽然 DebugCtl MSR 是一个 32 位的寄存器，但是只使用了低 7 位，其中各个位的含义如下。

- LBR 位用来启用 LBR 机制，如果此位被置为 1，那么处理器会使用上面介绍的 4 个寄存器来记录分支和异常或中断位置。对于 P6 处理器，当 CPU 产生调试异常发生时，CPU 会自动清除此位，以防止调试异常处理函数内部的分支覆盖掉原来的结果。
- BTF（Branch Trace Flag）的作用是启用按分支单步执行。如果此位被置为 1，那么 CPU 会将标志寄存器 EFLAGS 的 TF（陷阱标志）位看作是“single-step on branches（针对分支单步执行）”。换句话说，当 BTF 位和 TF 位都为 1 时，在 CPU 执行完下一个分支指令后便会产生调试异常。我们在上一章（4.3 节）详细介绍了这一功能。
- PB0 与 CPU 上的 BP0#（Breakpoint and Performance Monitoring output pin 0）引脚相对应。如果此位被置为 1，那么当 CPU 检测到 DR0（调试地址寄存器 0）所定义的断点条件时会设置 BP0#引脚，以通知硬件调试工具。如果此位被置为 0，那么当性能计数器 0 的值增加或溢出（由 PerfEvtSel0 寄存器控制）时，CPU 会反转（toggle）BP0#引脚上的电平。

PB1~PB3 与 PB0 类似，只是与 CPU 上的 BP1#~BP3#引脚和 DR1~DR3 寄存器相对应。

TR（Trace message enable）位用来启用（设为 1）或禁止向前端总线（FSB）上发送分支踪迹消息（Branch Trace Messages，简称 BTM）。当为 1 时，每当 CPU 检测到分支、中断或异常时，都会向 FSB 总线上发送 BTM 消息，以通知总线分析仪（bus analyzer）之类的调试工具。启用这项功能会影响 CPU 的性能。

5.2.2 LBR 栈

P6 的 LBR 机制只可以记录最近一次的分支和异常，奔腾 4 处理器对其做了增强，引入了所谓的“最近分支记录堆栈”(Last Branch Record Stack)，简称 LBR 栈，可以记录 4 次或更多次的分支和异常。奔腾 M 处理器和 Core 系列处理器也支持 LBR 栈。

LBR 栈是一个环形堆栈，由数个用来记录分支地址的 MSR 寄存器(称为 LBR MSR)和一个表示栈顶(Top Of Stack)指针的 MSR 寄存器(称为 MSR_LASTBRANCH_TOS)构成。CPU 在把新的分支记录放入这个堆栈前会先把 TOS 加 1，当 TOS 达到最大值时，会自动归 0。

LBR 栈的容量因 CPU 型号的不同而不同，目前产品的可能值为 4、8 或 16。可以通过 CPUID 指令取得 CPU 的 Family 和 Model 号码，再根据 Model 号码确定 LBR MSR 的数量。

以奔腾 4 CPU 为例，Model 号为 0~2 的处理器有 4 个 LBR MSR 寄存器，名称为 MSR_LASTBRANCH_0 到 MSR_LASTBRANCH_3，每个的长度是 64 位，高 32 位是分支的目标地址(TO)，低 32 位是分支指令的地址(FROM)。这样，这个堆栈可以最多记录最近 4 次的分支、中断或异常。

Model 号大于等于 3 的奔腾 4 处理器有 32 个 LBR MSR 寄存器，被分为 16 对，分别是 MSR_LASTBRANCH_0_FROM_LIP~MSR_LASTBRANCH_15_FROM_LIP 和 MSR_LASTBRANCH_0_TO_LIP~MSR_LASTBRANCH_15_TO_LIP，每个 MSR 的长度是 64 位，但高 32 位保留未用，因此最多可以记录 16 次最近发生的分支、中断或异常。

奔腾 M 处理器定义了 8 个 LBR 寄存器，MSR_LASTBRANCH_0~MSR_LASTBRANCH_7，地址为 0x40~0x47。这 8 个寄存器都是 64 位的，低 32 位用来记录 From 地址，高 32 位用来记录 To 地址。

Core 微架构的 CPU 通常有 8 个 LBR 寄存器，MSR_LASTBRANCH_0_FROM_IP~MSR_LASTBRANCH_3_FROM_IP 用来记录分支的源地址，MSR_LASTBRANCH_0_TO_IP~MSR_LASTBRANCH_3_TO_IP 用来记录分支的目标地址。这 8 个寄存器都是 64 位的，可以记录最近 4 次分支、中断或异常。

LBR 寄存器中内容的含义可能因为 CPU 型号的不同而不同。在 P6 处理器中，4 个分支记录寄存器所保存的地址都是相对于当前代码段的偏移。在 Pentium 4 处理器中，LBR 栈中记录的是线性地址。在 Core 微架构的 CPU 中，可以通过 IA32_PERF_CAPABILITIES 寄存器的 0 到 5 位([5:0])的值来进行判断，具体信息请参见 IA-32 手册卷 3B 的第 18 章。

5.2.3 应用示例

为了演示如何使用 LBR 寄存器了解 CPU 的执行轨迹，我们编写了一个 WinDBG

扩展模块 (DLL)，名为 LBR.DLL。执行这个模块的 lbr 命令，便可以列出 LBR 寄存器的内容。清单 5-1 列出了这个模块的主要函数的源代码。完整的代码和项目文件在 chap05\lbr 目录下。

清单 5-1 显示 LBR 栈的 WinDbg 扩展命令源代码

```

1  // 
2  // Extension to read LBR registers for Pentium M processors
3  //
4  #define LBR_COUNT 8
5  #define LBR_MSR_START_ADDR 0x40
6  #define MSR_LASTBRANCH_TOS 0x1c9
7  #define MSR_DEBUGCTLB      0x1d9
8  DECLARE_API( lbr )
9  {
10     ULONG64 llDbgCtrl,l1LBR;
11     ULONG    ulFrom,ulTo,ulTos;
12     CHAR     szSymbol[MAX_PATH];
13     ULONG    ulDisplacement;
14     int      nToDelete;
15
16     Version();
17     ReadMsr(MSR_DEBUGCTLB,&llDbgCtrl);
18     dprintf("MSR_DEBUGCTLB=%x\n", (ULONG)llDbgCtrl);
19     llDbgCtrl&=0xFFFFFFF0; // clear the LBR bit, bit 0
20     WriteMsr(MSR_DEBUGCTLB,llDbgCtrl);
21     dprintf("LBR bit is cleared now.\n");
22
23     ReadMsr(MSR_LASTBRANCH_TOS,&l1LBR);
24     ulTos=l1LBR&0xF;
25     dprintf("MSR_LASTBRANCH_TOS=%x\n", ulTos);
26
27     nToDelete=ulTos;
28     for (int i=0; i< LBR_COUNT;i++)
29     {
30         ReadMsr(LBR_MSR_START_ADDR+nToDelete,&l1LBR);
31         ulFrom = l1LBR;
32         ulTo = (l1LBR>>32);
33
34         szSymbol[0] = '!';
35         GetSymbol((PVOID)ulTo, (PUCHAR)szSymbol, &ulDisplacement);
36         dprintf("MSR_LASTBRANCH_%x: [%08lx] %s+%x\n", nToDelete,
37                 ulTo,szSymbol,ulDisplacement);
38
39         szSymbol[0] = '!';
40         GetSymbol((PVOID)ulFrom, (PUCHAR)szSymbol, &ulDisplacement);
41         dprintf("MSR_LASTBRANCH_%x: [%08lx] %s+%x\n", nToDelete,
42                 ulFrom, szSymbol,ulDisplacement);
43
44         nToDelete--;
45         if(nToDelete<0)
46             nToDelete=LBR_COUNT-1;
47     }
48
49     llDbgCtrl+=1; // set bit 0
50     WriteMsr(MSR_DEBUGCTLB,llDbgCtrl);
51     dprintf("LBR bit is set now.\n");
52 }
```

清单中的代码是以 Pentium M 处理器为例的。Pentium M 处理器的 LBR 栈有 8 个 LBR 寄存器，地址为 0x40~0x47，可以记录 8 次程序分支。它的 LBR 栈栈顶寄存器

(MSR_LASTBRANCH_TOS)的地址为 0x1C9, 调试控制寄存器(MSR_DEBUGCTLB)的地址为 0x1D9。第 4 到 7 行的 4 个常量是用来标志这些信息的。对于其他类型的寄存器, 这些常量的值可能有所不同。

第 17 行和第 18 行是读出并显示调试控制寄存器的值。第 19 行和第 20 行是将调试控制寄存器的 LBR 标志(位 0)清除。这样做的目的是暂时禁止 CPU 的 LBR 机制, 使 LBR 寄存器的内容保持稳定。不然, 我们读写这些寄存器的代码可能会使 LBR 寄存器的值不断变化。

第 23 行是从 MSR_LASTBRANCH_TOS 寄存器读出 LBR 栈的栈顶寄存器号(TOS)。这个寄存器的低 4 位有效, 因此第 24 行做了一个与操作, 去除其他位。

第 28 到 47 行的循环体依次读取 8 个 LBR 寄存器中的每一个。因为编号为 TOS 的寄存器记录的是最近的分支, 所以我们从这个寄存器开始读取, 并使用 nToRead 代表要读取的 LBR 寄存器号。

对于每个 LBR 寄存器, 其低 32 位代表分支的 From 地址, 我们将其赋给 ulFrom 变量, 高 32 位代表分支的 To 地址, 我们将其赋给 ulTo 变量。然后我们调用 WinDBG 的 GetSymbol 函数查询这两个地址的符号。第 41 行将得到的结果打印到调试器中。

第 49 行和第 50 行将调试控制寄存器的 LBR 位重新设置起来。

可以在安装有 Pentium M 处理器的系统(或者目标系统)上运行以上模块。方法是将 lbr.dll 复制到 WinDBG 的 winext 目录中, 然后启动一个本地内核调试对话(如果本机是 Pentium M 处理器)或双机内核调试对话(目标系统是 Pentium M 处理器), 并执行!lbr.lbr。清单 5-2 显示了在本机内核调试会话中的执行结果。

清单 5-2 在本机内核调试对话中执行 lbr 命令

```
1kd> !lbr.lbr
Access LBR (Last Branch Recording) registers of IA-32 CPU.
Version 1.0.0.2 by Raymond
MSR_DEBUGCTLB=1
LBR bit is cleared now.
MSR_LASTBRANCH_TOS=5
MSR_LASTBRANCH_5: [804ff190] nt!WRMSR+0
MSR_LASTBRANCH_5: [8065ef6e] nt!KdpSysWriteMsr+1c
MSR_LASTBRANCH_4: [8065ef5e] nt!KdpSysWriteMsr+c
MSR_LASTBRANCH_4: [805374da] nt!_SEH_prolog+3a
MSR_LASTBRANCH_3: [805374a0] nt!_SEH_prolog+0
MSR_LASTBRANCH_3: [8065ef59] nt!KdpSysWriteMsr+7
MSR_LASTBRANCH_2: [8065ef52] nt!KdpSysWriteMsr+0
MSR_LASTBRANCH_2: [8060d364] nt!NtSystemDebugControl+356
MSR_LASTBRANCH_1: [8060d356] nt!NtSystemDebugControl+348
MSR_LASTBRANCH_1: [8060d0c3] nt!NtSystemDebugControl+b5
MSR_LASTBRANCH_0: [8060d0b6] nt!NtSystemDebugControl+a8
MSR_LASTBRANCH_0: [8060d0a1] nt!NtSystemDebugControl+93
MSR_LASTBRANCH_7: [8060d09c] nt!NtSystemDebugControl+8e
MSR_LASTBRANCH_7: [8060d08d] nt!NtSystemDebugControl+7f
MSR_LASTBRANCH_6: [8060d089] nt!NtSystemDebugControl+7b
MSR_LASTBRANCH_6: [8060d082] nt!NtSystemDebugControl+74
LBR bit is set now.
```

在以上结果中, TOS 的值为 5, 也就是 5 号 LBR 寄存器 (MSR_LASTBRANCH_5) 记录的是最近一次分支的 From 和 To 信息, 因此我们从这个寄存器开始显示。然后依次显示 4、3、2、1、0、7、6。这样的结果与栈回溯类似, 上面的是后执行的。或者说, CPU 的执行路线是从下至上的。

对于显示 LBR 寄存器的各行, 第 1 列是 LBR 寄存器的名称, 每个寄存器占 2 行, 靠上的是高 32 位, 即 To 地址, 靠下的是低 32 位, 即 From 地址。以从 MSR_LASTBRANCH_3 向上的 6 行为例, 8065ef59 是 MSR_LASTBRANCH_3 的低 32 位内容, nt!KdpSysWriteMsr+7 是地址 8065ef59 所对应的符号。上面一行 nt!_SEH_prolog+0 是 MSR_LASTBRANCH_3 的 To 地址所对应的符号。因此 MSR_LASTBRANCH_3 记录的分支就是从 nt!KdpSysWriteMsr+7 向 nt!_SEH_prolog+0 转移的。类似的, MSR_LASTBRANCH_4 记录的是从 nt!_SEH_prolog+3a 向 nt!KdpSysWriteMsr+c 转移的。

观察 KdpSysWriteMsr 的反汇编 (清单 5-3), 可以看到, MSR_LASTBRANCH_3 记录的是第 3 行汇编的 CALL 调用所导致的跳转, 它的低 32 位记录的是当前指令的地址 (8065ef59), 高 32 位记录的是被调用函数的地址 (805374a0)。类似的, MSR_LASTBRANCH_4 记录的是从 nt!_SEH_prolog 函数返回到 KdpSysWriteMsr 函数的跳转。MSR_LASTBRANCH_5 记录的是调用 WRMSR 函数的 CALL 指令所导致的分支。

清单 5-3 KdpSysWriteMsr 函数的反汇编 (局部)

```
1kd> u nt!KdpSysWriteMsr la
nt!KdpSysWriteMsr:
8065ef52 6a08      push    8
8065ef54 68d88c4d80  push    offset nt!RamdiskBootDiskGuid+0x74 (804d8cd8)
8065ef59 e84285edff call    nt!_SEH_prolog (805374a0)
8065ef5e 33f6      xor     esi,esi
8065ef60 8975fc      mov    dword ptr [ebp-4],esi
8065ef63 8b450c      mov    eax,dword ptr [ebp+0Ch]
8065ef66 ff7004      push   dword ptr [eax+4]
8065ef69 ff30      push   dword ptr [eax]
8065ef6b ff7508      push   dword ptr [ebp+8]
8065ef6e e81d02eaff call   nt!WRMSR (804ff190)
```

5.3 使用内存的分支记录

上一节介绍的使用 MSR 寄存器的分支记录机制有一个明显的局限, 那就是可以记录的分支次数太少, 其应用价值比较有限。因为寄存器是位于 CPU 内部的, 所以靠增加 LBR 寄存器的数量来提高记录分支的次数是不经济的。于是, 人们很自然地想到设置一个特定的内存区供 CPU 来保存分支信息。这便是分支踪迹存储 (Branch Trace Store) 机制, 简称为 BTS。

BTS 允许把分支记录保存在一个特定的被称为 BTS 缓冲区的内存区内。BTS 缓冲区与用于记录性能监控信息的 PEBS 缓冲区是使用类似的机制来管理的, 这种机制被称为调试存储区 (Debug Store), 简称为 DS 存储区。

PEBS 的全称是 Precise Event-Based Sampling, 即精确的基于事件采样, 是奔腾 4

处理器引入的一种性能监控机制。当某个性能计数器被设置为触发 PEBS 功能且这个计数器溢出时，CPU 便会把当时的寄存器状态以 PEBS 记录的形式保存到 DS 存储区中的 PEBS 缓冲区内。每个 PEBS 记录的长度是固定的，32 位模式时为 40 个字节，包含了 10 个重要寄存器（EFLAGS、EIP、EAX、EBX、ECX、EDX、ESI、EDI、EBP 和 ESP）的值，IA-32e 模式时为 144 字节，除了以上 10 个寄存器外，还有 R8~R15 这 8 个新增的通用寄存器。下一节我们将详细讨论性能监视功能。本节将集中讨论如何建立 DS 存储区和使用它来记录分支信息。

5.3.1 DS 存储区

下面我们仔细看看 DS 存储区的格式。因为当 CPU 工作在 64 位的 IA-32e 模式时，所有寄存器和地址字段都是 64 位的，需要比工作在 32 位模式时更大的存储空间，所以 DS 存储区的格式也有所不同。本节将以 32 位为例进行介绍。

首先，DS 存储区由以下 3 个部分组成。

- 管理信息区。用来定义 BTS 和 PEBS 缓冲区的位置和容量。管理信息区的功能与文件头的功能很类似，CPU 通过查询管理信息区来管理 BTS 和 PEBS 缓冲区。
- BTS 缓冲区。用来以线性表的形式存储 BTS 记录。每个 BTS 记录的长度固定为 12 个字节，分成 3 个双字（DWORD），第一个 DWORD 是分支的源地址，第二个 DWORD 是分支的目标地址，第三个 DWORD 只使用了第 4 位（bit 4），用来表示该记录是否是预测出的。
- PEBS 缓冲区。用来以线性表的形式存储 PEBS 记录。每个 PEBS 记录的长度固定为 40 个字节。

图 5-3 画出了 DS 存储区的管理信息区的数据布局。

				偏移
3	2	1	0	
				00h
				04h
				08h
				0Ch
				10h
				14h
				18h
				1Ch
				20h
				24h
				28h
				2Ch
				30h

图 5-3 DS 存储区的管理信息区

从图 5-3 中可以看到，DS 管理信息区又分成了两部分，分别用来指定和管理 BTS 记录和 PEBS 记录。

IA-32 手册 (18.6.8.2 Setting Up the DS Save Area) 定义 DS 存储区应该符合的条件如下。首先，DS 存储区（3 个部分）应该在非分页（non-paged）内存中。也就是说，这段内存是不可以交换到硬盘上的，以保证 CPU 随时可以向其写入分支信息。其次，DS 存储区必须位于内核空间中。对于所有进程，包含 DS 缓冲区的内存页必须被映射到相同的物理地址。也就是说，CR3 寄存器的变化不会影响 DS 缓冲区的地址。第三，DS 存储区不要与代码位于同一内存页中，以防止 CPU 写分支记录时会触发防止保护代码页的动作。第四，当 DS 存储区处于活动状态时，要么应该防止进入 A20M 模式，要么应该保证缓冲区边界内地址的第 20 位（bit 20）都为 0。第五，DS 存储区应该仅用在启用了 APIC 的系统中，APIC 中用于性能计数器的 LVT 表项必须初始化为使用中断门，而不是陷阱门。

DS 存储区的大小可以超过一个内存页，但是必须映射到相邻的线性地址。BTS 缓冲区和 PEBS 缓冲区可以共用一个内存页，其基地址不需要按 4KB 边界对齐，只需要按 4 字节对齐。IA-32 手册建议 BTS 和 PEBS 缓冲区的大小应该是 BTS 记录（12 字节）和 PEBS 记录（40 字节）大小的整数倍。

5.3.2 启用 DS 机制

了解了 DS 存储区的格式和内存要求后，下面我们看看如何启用 DS 机制。

首先应该判断当前处理器对 DS 机制的支持情况。判断方法如下。

- 先将 1 放入 EAX 寄存器，然后执行 CPUID 指令，EDX[21]（DS 标志）应该为 1。
- 检查 IA32_MISC_ENABLE MSR 寄存器的位 11（BTS_UNAVAILABLE），如果该位为 0，表示该处理器支持 BTS 功能，如果该位为 1，则不支持。
- 检查 IA32_MISC_ENABLE MSR 寄存器的位 12（PEBS_UNAVAILABLE），如果该位为 0，则表示该处理器支持 PEBS 功能，如果该位为 1，则不支持。

第二步，根据前面的要求分配和建立 DS 内存区。

第三步，将 DS 存储区的基地址写到 IA32_DS_AREA MSR 寄存器。这个寄存器的地址可以在 IA-32 手册卷 3B 的附录中查到，目前 CPU 对其分配的地址都是 0x600。

第四步，如果计划使用硬件中断来定期处理 BTS 记录，那么设置 APIC 局部向量表（LVT）的性能计数器表项，使其按固定时间间隔产生中断（fixed delivery and edge sensitive），并在 IDT 表中建立表项并注册用于处理中断的中断处理例程。在中断处理例程中，应该读取已经记录的分支信息和 PEBS 信息，将这些信息转存到文件或其他位置，然后将缓冲区索引字段复位。

第五步，设置调试控制寄存器，启用 BTS，我们将在下一段中讨论。

5.3.3 调试控制寄存器

在支持分支监视和记录机制的处理器中，都有一个用来控制增强调试功能的 MSR 寄存器，称为调试控制寄存器（Debug Control Register）。对于不同的处理器，这个寄存器的名称和格式有所不同。主要有以下 4 种。

- P6 系列处理器中的 DebugCtl 寄存器，我们在 5.2 节对其格式做过详细的介绍。
- 奔腾 4 系列处理器中的 DebugCtlA 寄存器，其格式如图 5-4 所示。
- 奔腾 M 系列处理器中的 DebugCtlB 寄存器，其格式如图 5-5 所示。
- Core 系列和 Core 2 系列处理器中的 IA32_DEBUGCTL 寄存器，其格式如图 5-6 所示。从名称上来看，这个名称已经带有 IA-32 字样，被称为架构中的标准寄存器，以后的 IA-32 系列处理器应该会保持这个名称。

图 5-4 显示了奔腾 4 的 DebugCtlA MSR 寄存器。

31	7	6	5	4	3	2	1	0
保留	B O U	B O O	B T I	B T S	T T I	T R S	B T F	L B R

图 5-4 DebugCtlA MSR 寄存器（奔腾 4 处理器）

31	8	7	6	5	4	3	2	1	0
保留	B I N T	B T S	T R	P B 3	P B 2	P B 1	P B 0	B T F	L B R

图 5-5 DebugCtlB MSR 寄存器（奔腾 M 处理器）

31	8	7	6	5	4	3	2	1	0
保留	B I N T	B T S	T R	保留	保留	保留	保留	B T F	L B R

图 5-6 IA32_DEBUGCTL MSR 寄存器（Core、Core 2 及更新的 IA-32 处理器）

其中 LBR、BTF 的含义与 P6 中的一样。概括说来，TR 位用来启用分支机制，BTS 用来控制分支信息的输出方式，如果为 1，则将分支信息写到 DS 内存区的 BTS 缓冲区中，如果为 0，则向前端总线发送分支跟踪消息（BTM），供总线分析仪等设备接收。

BTI（Branch Trace INTerrupt）如果被置为 1，那么当 BTS 缓冲区已满时，会产生中断。如果为 0，CPU 会把 BTS 缓冲区当作一个环形缓冲区，写到缓冲区的末尾后，CPU 会自动回转到缓冲器的头部。

BOO（BTS_OFF_OS）和 BOU（BTS_OFF_USER）用来启用 BTS 的过滤机制，

如果 BOO 为 1，则不再将 CPL 为 0 的 BTM 记录到 BTS 缓冲区中，也就是不再记录内核态的分支信息。如果 BOU 为 1，则不再将 CPL 不为 0 的 BTM 记录到 BTS 缓冲区中，也就是不再记录用户态的分支信息。

尽管名称和格式有所不同，对于目前的 CPU，以上 4 种 MSR 寄存器的地址都是 0x1D9。

启用 DS 机制，需要编写专门的驱动程序来建立和维护 DS 存储区，我们将在下一节给出一个例子。

5.4 DS 示例：CpuWhere

上一节我们介绍了 IA-32 处理器的调试存储（DS）功能。为了演示其用法，加深大家的理解，这一节我们将编写一个示例性的应用，这个应用的名字叫 **CpuWhere**，意思是使用这个应用，用户可以看到 CPU 曾经运行过哪些地方（Where has CPU run）。

5.4.1 驱动程序

因为访问 MSR 寄存器和分配 BTS 缓冲区都需要在内核态进行，所以要使用 DS 机制，需要编写一个驱动程序，我们将其命名为 **CpuWhere.sys**。

首先，我们需要定义两个数据结构：**DebugStore** 和 **BtsRecord**。

DebugStore 结构用来描述 DS 存储区的管理信息区，代码如下：

```
typedef struct tagDebugStore
{
    DWORD    dwBtsBase;           // BTS 缓冲区的基址
    DWORD    dwBtsIndex;          // BTS 缓冲区的索引，指向可用的 BTS 缓冲区
    DWORD    dwBtsAbsolute;       // BTS 缓冲区的极限值
    DWORD    dwBtsIntThreshold;   // 报告 BTS 缓冲区已满的中断阈值
    DWORD    dwPebsBase;          // PEBS 缓冲区的基址
    DWORD    dwPebsIndex;         // PEBS 缓冲区的索引，指向可用的 BTS 缓冲区
    DWORD    dwPebsAbsolute;      // PEBS 缓冲区的极限值
    DWORD    dwPebsIntThreshold;  // 报告 PEBS 缓冲区已满的中断阈值
    DWORD    dwPebsCounterReset;  // 计数器的复位值
    DWORD    dwReserved;          // 保留
} DebugStore, *PDebugStore;
```

BtsRecord 结构用来描述 BTS 缓冲区的每一条数据记录，代码如下：

```
typedef struct tagBtsRecord
{
    DWORD    dwFrom;             // 分支的发起地址
    DWORD    dwTo;               // 分支的目标地址
    DWORD    dwFlags;             // 标志
} BtsRecord, *PBtsRecord;
```

以上两个结构都是用于 32 位模式的，如果系统工作在 64 位的 IA-32e 模式下，那么需要将大多数组段从 **DWORD** 改为 8 字节的 **DWORD64**。

定义了以上结构后，便可以使用 Windows 操作系统的 ExAllocatePoolWithTag 函数来在非分页内存区中建立 DS 存储区了。清单 5-4 给出了主要的源代码。

清单 5-4 建立 DS 内存区的源代码

```

1 #define BTS_RECORD_LENGTH sizeof(BtsRecord)
2
3 PDebugStore g_pDebugStore=NULL;
4 PVOID g_pBtsBuffer=NULL;
5 DWORD g_dwMaxBtsRecords=0;
6 BOOLEAN g_bIsPentium4=0xFF;
7 BOOLEAN g_bIsTracing=0;
8 DWORD g_dwOptions=0;
9
10 NTSTATUS SetupDSArea(DWORD dwMaxBtsRecords)
11 {
12     if(g_pDebugStore==NULL)
13         g_pDebugStore=ExAllocatePoolWithTag(
14             NonPagedPool,sizeof(DebugStore),
15             CPUWHERE_TAG);
16
17     memset(g_pDebugStore,0,sizeof(DebugStore));
18
19     if(g_pBtsBuffer && g_dwMaxBtsRecords!=dwMaxBtsRecords)
20     {
21         ExFreePoolWithTag(g_pBtsBuffer,CPUWHERE_TAG);
22         g_pBtsBuffer=NULL;
23     }
24     g_pBtsBuffer=ExAllocatePoolWithTag(
25         NonPagedPool,dwMaxBtsRecords*BTS_RECORD_LENGTH,
26         CPUWHERE_TAG);
27     if(g_pBtsBuffer==NULL)
28     {
29         DBGOUT(("No resource for BTS buffer %d*%d",
30                 dwMaxBtsRecords, BTS_RECORD_LENGTH));
31         return STATUS_NO_MEMORY;
32     }
33
34     g_dwMaxBtsRecords=dwMaxBtsRecords;
35     // zerolize the whole buffer
36     memset(g_pBtsBuffer,0, dwMaxBtsRecords*BTS_RECORD_LENGTH);
37
38     g_pDebugStore->dwBtsBase=(ULONG)g_pBtsBuffer;
39     g_pDebugStore->dwBtsIndex=(ULONG)g_pBtsBuffer;
40     g_pDebugStore->dwBtsAbsolute=(ULONG)g_pBtsBuffer
41         +dwMaxBtsRecords*BTS_RECORD_LENGTH;
42     // In order to prevent generating an interrupt, when working with
43     // circular BTS buffer, SW need to set BTS interrupt threshold to a value
44     // greater than BTS absolute maximum (fields of the DS buffer
45     // management area). It's not enough to clear the BTINT flag itself only.
46     g_pDebugStore->dwBtsIntThreshold=(ULONG)g_pBtsBuffer
47         +(dwMaxBtsRecords+1)*BTS_RECORD_LENGTH;
48
49     DBGOUT(("DS is setup at %x: base %x, index %x, max %x, int %x",
50             g_pDebugStore,g_pDebugStore->dwBtsBase,
51             g_pDebugStore->dwBtsIndex,
52             g_pDebugStore->dwBtsAbsolute,
53             g_pDebugStore->dwBtsIntThreshold))
54
55     return STATUS_SUCCESS;
56 }
```

第 12~17 行分配一个 DebugStore 结构，将其线性地址赋给全局变量 g_pDebugStore，并将整个结构用 0 填充。第 19~34 行分配用于保存分支记录的 BTS 缓冲区。其大小是由参数 dwMaxBtsRecords 所决定的。第 36 行将这个缓冲区初始化为 0。第 38~47 行用来初始化 DebugStore 结构。因为我们不打算使用中断方式来报告 BTS 缓冲区已满，所以将产生中断的阈值（dwBtsIntThreshold 字段）设得很大，比缓冲区的最大值还大一些。

准备好了 DS 存储区后，就可以通过设置 MSR 寄存器来启用 DS 机制了。清单 5-5 给出了用于启用和禁止 BTS 机制的 EnableBTS 函数的源代码。

清单 5-5 启用和禁止 BTS 的源代码

```

1  NTSTATUS EnableBTS(BOOLEAN bEnable, BOOLEAN bTempOnOff)
2  {
3      DWORD dwEDX, dwEAX;
4
5      if(!bTempOnOff)
6      {
7          ReadMSR(IA32_MISC_ENABLE, &dwEDX, &dwEAX);
8          if(bEnable && ( (dwEAX & (1<<BIT_BTS_UNAVAILABLE))!=0 ) )
9          {
10             DBGOUT(("BTS is not supported %08x:%08x", dwEDX, dwEAX));
11             return -1;
12         }
13         if(bEnable) // for disable, we left the register as is
14         {
15             // set the DS memroy to MSR
16             dwEDX=0;
17             dwEAX=bEnable?(DWORD)g_pDebugStore:0;
18
19             WriteMSR(IA32_DS_AREA, dwEDX, dwEAX);
20         }
21     }
22
23     // enable MSR flags
24     ReadMSR(IA32_DEBUGCTL, &dwEDX, &dwEAX);
25     DBGOUT(("Old IA32_DEBUGCTL=%08x:%08x", dwEDX, dwEAX));
26
27     // Set the TR and BTS flags in the MSR_DEBUGCTLA
28     if(bEnable)
29     {
30         dwEAX|=(1 << (g_bIsPentium4?BIT_P4_BTS:BIT_BTS) );
31         dwEAX|=(1 << (g_bIsPentium4?BIT_P4_TR:BIT_TR) );
32         // Clear the BTINT flag in the MSR_DEBUGCTLA
33         dwEAX&=~(1<< (g_bIsPentium4?BIT_P4_BTINT:BIT_BTINT) );
34     }
35     else
36     {
37         dwEAX&=~(1<< (g_bIsPentium4?BIT_P4_BTS:BIT_BTS) );
38         dwEAX&=~(1<< (g_bIsPentium4?BIT_P4_TR:BIT_TR) );
39     }
40     WriteMSR(IA32_DEBUGCTL, dwEDX, dwEAX);
41
42     // show new value after write
43     ReadMSR(IA32_DEBUGCTL, &dwEDX, &dwEAX);
44     DBGOUT(("Current IA32_DEBUGCTL=%08x:%08x", dwEDX, dwEAX));
45
46     return STATUS_SUCCESS;
47 }
```

参数 bEnable 用来指定是启用还是禁止 BTS，参数 bTempOnOff 用来指定本次操作是否是暂时性的，当我们读取 BTS 缓冲区时，需要暂时禁用 BTS 机制，读好后再启用它。暂时性的禁用只操作调试控制寄存器，不操作 IA32_DS_AREA 寄存器。第 7 行到第 12 行用于检查 CPU 是否支持 BTS，即读取 IA32_MISC_ENABLE 寄存器并检查它的 BTS_UNAVAILABLE 标志。第 13 行到第 20 行用于设置 IA32_DS_AREA 寄存器。剩下的代码用于操作调试控制寄存器。因为对于我们关心的 TR、BTINT 和 BTS 位，奔腾 4 之外的两种调试控制寄存器（IA32_DEBUGCTL 和 DebugCtlB）的这些位是一样的，所以我们只是判断当前的 CPU 是否是奔腾 4，全局变量 g_bIsPentium4 记录了这一特征。

除了以上代码，驱动程序中还实现了以下一些函数和代码：用于启动和停止追踪的 StartTracing 函数，它内部会调用 SetupDSArea 和 EnableBTS；用于读取 BTS 记录的 GetBtsRecords 函数，它会根据 DebugStore 结构中的信息来读取 CPU 已经产生的 BTS 记录，读好后，再把索引值恢复原位。此外，还有负责与应用程序通信的 IRP 响应函数，以及其他 WDM 模型定义的驱动程序函数。

5.4.2 应用界面

有了驱动程序后，还需要编写一个简单的应用程序来管理驱动程序，以及读取和显示 BTS 记录，我们将其命名为 CpuWhere.exe。图 5-7 是 CpuWhere.exe 的执行界面。窗口左侧是一系列控制按钮，编辑框用来指定 BTS 缓冲区可以容纳的 BTS 记录数，也就是 SetupDSArea 函数的参数。窗口右侧的列表框用来显示从驱动程序读取到的 BTS 记录。显示的顺序与栈回溯类似，最近发生的在上方。或者说，CPU 的运行轨迹是从下到上的。



图 5-7 CpuWhere 程序的界面

在列表框中，每条 BTS 记录显示为两行，上面一行用来显示分支的目标地址（方括号中），地址前以>符号表示，地址后为这个地址所对应的符号；下面一行为分支的发起地址，地址前以<表示，大括号中是本条 BTS 记录的标志字段（dwFlags）。每一行的开头是以#开始的流水号。

以图 5-7 中的第 2 行为例，#00004365 - [<0x80526bed]: nt!PsGetCurrentProcessId + d {flag 0x0}。其中，地址前的小于号代表这是一个 BTS 记录的发起行，0x80526bed 是 BtsRecord 中的 dwFrom 字段的值，nt!PsGetCurrentProcessId + d 是这个地址所对应的符号和位移（displacement）。

观察 nt!PsGetCurrentProcessId 函数的反汇编，可以看到地址 0x80526bed 是 ret 指令的下一条指令的地址，因此 CPU 是在执行 ret 指令时产生这条 BTS 记录的。

```
1kd> u nt!PsGetCurrentProcessId
nt!PsGetCurrentProcessId:
80526be0 64a124010000    mov      eax,dword ptr fs:[00000124h]
80526be6 8b80ec010000    mov      eax,dword ptr [eax+1ECh]
80526bec c3              ret
80526bed cc              int     3
```

观察第 1 行 (#00004365 - [>0xbff801a73]: win32k!HmgLock + 2e)，它是这个 BTS 记录的目标地址，于是，可以推测出这个 BTS 记录记载的是从 PsGetCurrentProcessId 函数返回 HmgLock 这一事件。第 3 行和第 4 行 (#00004366) 记载的是 HmgLock 函数调用 PsGetCurrentProcessId 时的分支。

在图 5-7 所示列表框的倒数第 2 行和第 3 行，记录了调用系统服务时从用户态向内核态的转移过程。它们记载了从用户态地址 [<0x7c90eb8f] 跳转到内核态地址 [>0x8053cad0] 的过程。

CpuWhere.exe 的大多数实现都是非常简单的。比较复杂的地方就是如何查找 BTS 记录所对应的符号。因为 BTS 记录中的地址有内核态的地址，也有用户态的地址，简单地使用 DbgHelp 库中的符号函数（SymFromAddr 等）是不能满足我们的需要的。

为了用比较少的代码解决以上问题，我们使用了 WinDBG 的调试引擎。通过调试引擎所输出的接口，我们启动了一个本地内核调试会话，然后利用调试引擎来为分支记录中的地址寻找合适的符号。其核心代码如清单 5-6 所示。

清单 5-6 启动本地内核调试的 StartLocalSession 方法

```
1 // start a local kernel debug session
2 HRESULT CEngMgr::StartLocalSession(void)
3 {
4     HRESULT hr;
5
6     if(m_Client==NULL)
7         return E_FAIL;
8     if ((hr = m_Client->SetOutputCallbacks(&m_OutputCallback)) != S_OK)
9     {
10        Log("StartLocalSession", "SetOutputCallbacks failed, 0x%X\n", hr);
11        return hr;
```

```

12      }
13      // Register our event callbacks.
14      if ((hr = m_Client->SetEventCallbacks(&m_EventCb)) != S_OK)
15      {
16          Log("StartLocalSession", "SetEventCallbacks failed, 0x%X\n", hr);
17          return hr;
18      }
19      hr = m_Client->AttachKernel(DEBUG_ATTACH_LOCAL_KERNEL, NULL);
20      if (hr != S_OK)
21      {
22          Log("StartLocalSession",
23              "AttachKernel(DEBUG_ATTACH_LOCAL_KERNEL, NULL) failed with %x", hr);
24          return hr;
25      }
26
27      if ((hr = m_Control->WaitForEvent(DEBUG_WAIT_DEFAULT,
28                                         INFINITE)) != S_OK)
29      {
30          Log("StartLocalSession", "WaitForEvent failed, 0x%X\n", hr);
31      }
32      return hr;
33 }

```

第 8 行到第 12 行设置一个输出回调类，用来接收调试引擎的信息输出。我们将这些输出定向到列表框中。第 19 行到第 25 行是启动本地内核调试，第 27 行到第 31 行是等待初始的调试事件。等待这一事件后，调试引擎的内部类会针对本地内核的实际情况进行初始化，此后就可以使用调试引擎的各种服务了。我们将在第 29 章进一步介绍调试引擎的细节。

因为依赖于调试引擎，所以运行 `CpuWhere` 程序需要先安装 WinDBG，而后将 `CpuWhere.exe` 复制到 WinDBG 程序的目录中再运行它。

5.4.3 局限性和扩展建议

`CpuWhere` 程序可以观察到 CPU 的运行轨迹，并将其翻译为程序符号。通过这些信息，我们可以精确地了解 CPU 的运行历史，为软件调试和研究软件的工作过程提供第一手资料。

但是 `CpuWhere` 毕竟是一个简单的示例性的小程序，它只是 IA-32 CPU 的 BTS 功能的一个初级应用，还有如下局限。

- 没有考虑多处理器的情况，如果系统中有多个 CPU，那么应该为每个 CPU 建立独立的 DS 存储区，并分别进行显示和管理。
- 没有考虑进程上下文。目前只是简单地将 BTS 记录中的地址传递给内核调试引擎寻找匹配的符号，因为我们没有仔细地设置和维护进程上下文，所以查找到的用户态地址的符号可能是不准确的，甚至是错误的。

- 我们只是以单一的线性列表来显示 BTS 记录，一种更好的显示方式是以调用图（Calling Graph）的方式来显示函数的调用和返回。

英特尔公司的 VTune 及下一节介绍的性能监视机制，是一个强大的辅助调试和性能分析的工具。感兴趣的读者可以到英特尔公司的网站查阅更多的信息。

5.5 性能监视

很多程序员都有过这样的经历：为了评估一段代码的执行效率，分别在这段代码的前面和后面取系统时间，然后通过计算时间差得到这段代码的执行时间。这可以说是最简单的性能监视（Performance Monitoring）方法。因为这种方法忽略了很多因素，所以得到的结果只是一个非常粗略的估计。比如在一个多任务的操作系统中，CPU 在执行这段代码的过程中，很可能被多次切换去执行其他的程序或处理各种中断请求，而这些时间是不固定的。

性能监视对软件调优（tuning）和软件调试都有着重要的意义，为了更好地满足性能监视任务的需要，IA-32 处理器从奔腾开始就提供了性能监视机制，包括专门的计数器、寄存器、CPU 管脚和中断支持等。

需要指出的是，虽然从奔腾 CPU 开始的所有 IA-32 处理器都包含了性能监视支持，但是直到 Core Solo 和 Core Duo 处理器产生，才将一部分性能监视机制纳入到 IA-32 架构中，其他部分仍是与处理器型号相关的。换句话说，IA-32 CPU 的性能监视支持是与处理器型号相关的，当使用时，应该先检查 CPU 的型号。下面按照由简单到复杂的顺序分别介绍不同 IA-32 CPU 的性能监视机制。

5.5.1 奔腾处理器的性能监视机制

奔腾处理器是第一个引入性能监视机制的 IA-32 处理器，该机制包括两个 40 位的性能计数器（PerfCtl0 和 PerfCtl1），一个 32 位的名为 CESR（Counter Event Select Register）的控制寄存器，以及处理器上的 PM0/BP0 和 PM1/BP1 管脚。CESR 寄存器用于选择要监视的事件和配置监视选项，PM0/BP0 和 PM1/BP1 管脚用于向外部硬件通知计数器状态（各对应一个计数器）。PerfCtl0、PerfCtl1 和 CESR 寄存器都是 MSR 寄存器，可以通过 RDMSR 和 WRMSR 指令来访问。下面我们以 CESR 寄存器的格式为线索，介绍奔腾处理器的性能监视机制。

图 5-8 画出了 CESR 寄存器的各个位域。

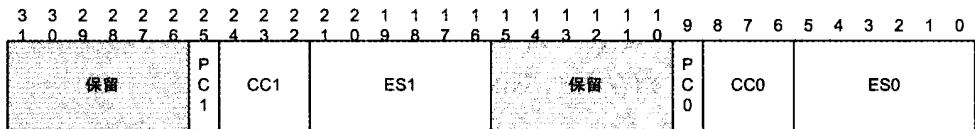


图 5-8 奔腾处理器的 CESR 寄存器

显而易见，CESR 寄存器的高 16 位和低 16 位的布局是相同的。低 16 位对应计数器 0，高 16 位对应计数器 1。每个部分都包含下面 3 个域。

6 位的事件选择域 ES (Event Select): 用来选择要测量（监视）的事件类型，例如 DATA_READ、DATA_WRITE、CODE_READ 等。IA-32 手册卷 3B 的附录 A 中列出了每种 IA-32 处理器所支持的全部事件类型。

3 位的计数器控制域 CC (Counter Control): 用来设置计数器选项，其可能值和含义如下。

- 000：停止计数；
- 001：当 CPL=0、1 或 2 时对选定的事件计数；
- 010：当 CPL=3 时对选定的事件计数；
- 011：不论 CPL 值为何值，都对选定的事件计数；
- 100：停止计数；
- 101：当 CPL=0、1 或 2 时对时钟（clocks）计数，相当于记录 CPU 在内核态的持续时间；
- 110：当 CPL=3 时对时钟（clocks）计数，相当于记录 CPU 在用户态的持续时间；
- 111：对时钟计数，不论 CPL 值为何值。

显而易见，最高位是用来控制对事件计数还是对时钟计数（即持续时间）；中间位用来使能（enable）当 CPL 为 3（用户模式下）时是否计数；最低位用来使能（enable）当 CPL 为 0、1 或 2（内核模式下）时是否计数。

1 位的管脚控制域 PC (Pin Control): 用来设置对应的 PM/BP 管脚行为，如果该位为 1，那么当对应计数器溢出时，PM/BP 管脚信号被置起（asserted）。如果该位为 0，那么当对应计数器递增时，PM/BP 管脚信号被置起。

5.5.2 P6 处理器的性能监视机制

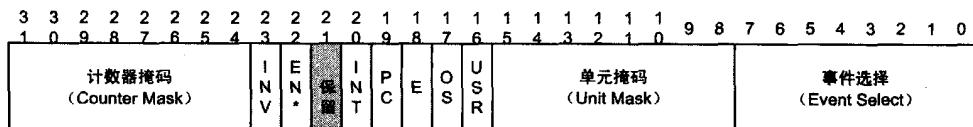
与奔腾处理器的性能监视机制相比，P6 处理器的性能监视机制可以大体概括如下。

第一，仍然实现了两个 40 位的性能计数器 PerfCtl0 和 PerfCtl1。

第二，增加了 RDPMC 指令用于读取性能计数器的值。因为性能计数器是 MSR 寄存器，所以可以通过 RDMSR 指令来读取，但是 RDMSR 指令只能在内核模式（或实模式）下执行。而 RDPMC 指令可以在任何特权级别下执行，因此有了 RDPMC 指令后就可以在用户模式下读取性能计数器了。其使用方法是将计数器号（0 和 1）放入 ECX 寄存器中，然后执行 RDPMC 指令，其结果会被放入 EDX:EAX 对（EAX 中包含低 32 位，EDX 中包含高 8 位）中。

第三，使用两个 32 位的 MSR 寄存器 PerfEvtSel0 和 PerfEvtSel1，分别控制计数器 PerfCtl0 和 PerfCtl1。不再像奔腾处理器那样使用一个 CESR 寄存器的高低 16 位来分别控制两个计数器。可以用 RDMSR 和 WRMSR 指令来访问 PerfEvtSel0 和 PerfEvtSel1 寄存器（在内核模式或实模式下）。它们的地址分别是 186H 和 187H。

图 5-9 画出了 PerfEvtSel 寄存器的位布局。



*启用计数器，仅适用于PerfEvtSel0

图 5-9 P6 处理器的 PerfEvtSel 寄存器

其各个位域的含义如下。

8 位的事件选择域 ES (Event Select): 用来选择要监视的事件类型。具体类型定义被列在 IA-32 手册卷 3B 的附录 A 中。

8 位的单元掩码域 UMASK (Unit Mask): 进一步定义 ES 域中指定的要监视事件。可以理解为是 ES 域中指定的要监视事件的更详细的条件和参数。

1 位的用户模式域 USR (User Mode): 指定当处理器处于特权级 1、2 或 3（即用户模式下）时是否计数。

1 位的系统模式域 OS (Operating System Mode): 指定当处理器处于特权级 0（即内核模式下）时是否计数。

1 位的边缘检测域 E(Edge Detect): 用来数被监视事件（其他域指定）从 deasserted

到 asserted 的状态过渡次数。

1位的管脚控制域 PC (Pin Control): 其含义与奔腾处理器相同, 参见上文。

1位的中断使能域 INT (APIC Interrupt Enable): 用于控制当相应计数器溢出时是否让本地的 APIC (Advanced Programmable Interrupt Controller, 即集成在 CPU 内部的可编程中断控制器)产生一个中断。事先应该设置好 APIC 的局部向量表 LVT (Local Vector Table)、中断服务例程及 IDT 表。

1位的计数器使能域 EN (Enable Counters): 当设为 1 时, 启动两个计数器, 当为 0 时, 禁止两个计数器。该位仅在 PerfEvtSel0 中实现。

8位的计数器掩码域 CMASK (Counter Mask): 用作计数器的计数条件阈值, 当事件数与这个阈值比较满足条件时才改变计数器的值。下面的 INV 位用来指定比较方法。

1位的取反域 INV (Invert): 该位为 1 时, 当事件数量少于 CMASK 中的值时才将事件计入到计数器中。如果该位为 0, 那么当事件数量大于等于 CMASK 中的值时, 才将事件计入到计数器中。

5.5.3 P4 处理器的性能监视

与 P6 系列和奔腾处理器相比, P4 处理器对性能监视支持做了非常大的改进和增强。尽管选择和过滤事件类型, 以及通过 WRMSR、RDMSR 或 RDPMC 指令来访问有关寄存器的一般方法没有变, 但是 MSR 寄存器的布局和设置机制都有了很大的变化。概括如下。

第一, 性能计数器的数量由 2 个增加至 18 个 (每个仍然是 40 位的), RDPMC 指令也做了增强, 可以以更快的速度读取这些寄存器; CR4 寄存器增加了 PCE 位允许操作系统限制在用户模式下执行 RDPMC 指令。

18 个计数器被分为 9 对, 又进一步划分为以下 4 组。

BPU (Branch Prediction Unit) 组包含 2 个计数器对:

- MSR_BPU_COUNTER0 (编号 0) 和 MSR_BPU_COUNTER1 (编号 1)
- MSR_BPU_COUNTER2 (编号 2) 和 MSR_BPU_COUNTER3 (编号 3)

MS (Microcode Store) 组包含 2 个计数器对:

- MSR_MS_COUNTER0 (编号 4) 和 MSR_MS_COUNTER1 (编号 5)
- MSR_MS_COUNTER2 (编号 6) 和 MSR_MS_COUNTER3 (编号 7)

FLAME 组包含 2 个计数器对:

- MSR_FLAME_COUNTER0 (编号 8) 和 MSR_FLAME_COUNTER1 (编号 91)
- MSR_FLAME_COUNTER2 (编号 10) 和 MSR_FLAME_COUNTER3 (编号 11)
- IQ (Instruction Queue) 组包含 3 个计数器对:
 - MSR_IQ_COUNTER0 (编号 12) 和 MSR_IQ_COUNTER1 (编号 13)
 - MSR_IQ_COUNTER2 (编号 14) 和 MSR_IQ_COUNTER3 (编号 15)
 - MSR_IQ_COUNTER4 (编号 16) 和 MSR_IQ_COUNTER5 (编号 17)

如果希望记录更大的范围, 那么可以将一个计数器与本组内不属于同一对的其他计数器进行级联 (称为 counter cascading)。

第二, 事件选择控制寄存器 (ESCR) 的数量也大幅增加, 多达 43 到 45 个 (与处理器的详细型号有关), 用于选择和过滤要监视的事件, 以及控制特定的计数器。一个计数器可以最多与 8 个 ESCR 之一相关联。一个 ESCR 也可能被用于多个计数器。ESCR 寄存器的布局如图 5-10 所示。



图 5-10 P4 处理器的 ESCR 寄存器

25 到 30 位用来选择要监视的事件大类 (event class); 9 到 24 位用来选择事件大类中的具体事件; 5 到 8 位可以指定一个与微指令相关联的标记 (tag) 值, 用来辅助对回收期事件计数; 位 4 用于启用或禁止微指令标记 (tagging) 功能; 位 3 (OS) 和位 2 (USR) 与以前的含义相同。

第三, 新增 18 个计数器配置控制寄存器 CCCR (Counter Configuration Control Register), 与 18 个计数器一一对应, 用于设置计数的方式和参数。CCCR 寄存器的布局如图 5-11 所示。

位 12 (enable) 用来启动对应的计数器; 位 13 到 15 (ESCR select) 用来指定与对应计数器相关联的 ESCR 寄存器, 即间接选择要监视的事件; 位 18 (compare) 用来启动位 19 到位 24 所定义的事件过滤; 位 19 (complement) 为 1 时, 当事件数小于等于阈值时递增计数器, 为 0、大于阈值时递增计数器; 位 20 到 23 (threshold) 指定用于比较的阈值, 具体值与被监视的事件类型有关; 位 24 (edge) 用于启用或禁止上升沿 (false-to-true) 检测; 位 25 (FORCE_OVF) 为 1 时, 计数器每次递增都会强制计

数器溢出，为 0 时且仅当计数器真正溢出时才发生溢出；位 26 (OVF_PMI) 为 1 时，每当计数器溢出都会产生 PMI 中断；位 30 (级联标志) 用于启用和禁止计数器级联；位 31 (OVF 标志) 为 1 时表示对应计数器已经溢出，此标志位不会自动清除（必须由软件显式清除）。

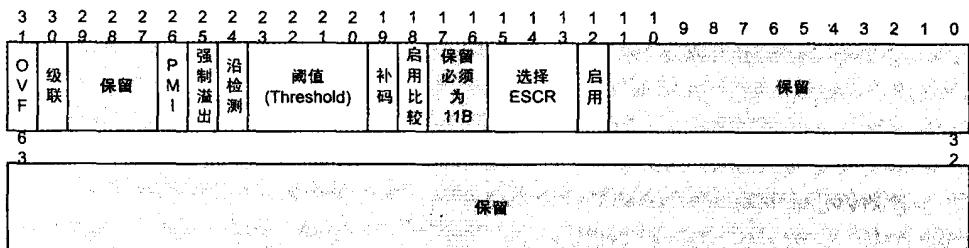


图 5-11 CCCR 寄存器

第四，将事件分为如下两种类型：回收阶段事件 (at-retirement event) 和非回收阶段事件 (non-retirement event)，前者是指发生在指令执行的回收阶段 (retirement stage) 的事件；后者是指发生在指令执行过程中任何时间的事件，如总线事务等。针对回收期事件的计数仅记录与分支预测正确的路径上的微操作有关的事件。针对非回收期事件的计数会记录指定类型的所有事件，即使该事件属于预测错误的分支（不会进入回收期）。

第五，将采样 (sampling) 计数器的方式（也就是使用计数器的模式）归纳为如下 3 种。

1. 定期读取 (Event Counting)：在计数器计数期间，软件以一定的间隔读取计数器的值。
2. 计数器溢出时产生中断：当计数器溢出时，产生性能监视中断 (Performance Monitoring Interrupt, 简称 PMI)。中断处理程序记录下返回指令指针 (return instruction pointer, 简称 RIP, 也就是被中断程序的指令地址)，复位计数器，然后重新开始计数。IA-32 手册将此方式称为基于事件的非精确采样 (non-precise event-based sampling)。通过分析 RIP 的分布，可以分析代码的执行情况以供性能优化使用。英特尔的 VTune 工具可以将 RIP 分布等信息以图形的方式显示出来。
3. 计数器溢出时自动保存状态：当计数器溢出时，自动将通用寄存器、EFlags 寄存器和 EIP 寄存器的值保存到调试存储区 (Debug Store)。IA-32 手册将此方式称为基于事件的精确采样 (Precise Event-Based Sampling, 简称 PEBS)。该方式不仅对

目标程序代码影响最小，而且保存的状态信息最多，因此对软件性能优化非常有用。但是该方式仅适用于一部分回收期事件（Execution_event、Front_end_event 和 Replay_event），不能用于非回收期事件。

第六，IA32_MISC_ENABLE 寄存器中增加了两个位域用于检测处理器对性能监视的支持能力（位 7 和位 12）。

IA-32 手册卷 3B 附录 A 中列出了奔腾 4 处理器支持的所有非回收期事件，以及每个事件的参数设置信息。下面简要介绍如何开始对非回收期事件进行计数。

1. 选择要监视的事件。
2. 根据 IA-32 手册卷 3B 附录 A 中指导信息为每个要监视的事件选取一个支持该事件的 ESCR 寄存器。
3. 选取一个与选择的 ESCR 相关联的 CCCR 寄存器和计数器（CCCR 寄存器和计数期是一一对应的）。并从 IA-32 手册查找到选取的计数器、ESCR 寄存器、CCCR 寄存器的地址。
4. 使用 WRMSR 指令设置 ESCR 寄存器，指定要监视的事件和要计数的特权级别。
5. 使用 WRMSR 指令设置 CCCR 寄存器，指定 ESCR 寄存器和事件过滤选项。
6. [可选]设置计数器级联选项。
7. [可选]设置 CCCR 寄存器以便当计数器溢出时产生性能监控中断（PMI）。如果启用 PMI，那么必须设置本地 APIC、IDT 表及相应的中断处理例程。
8. 使用 WRMSR 指令置起 CCCR 寄存器的启用标志（Enable），开始事件计数。如果要停止计数，则将该标志清零。

下面再来看看启动 PEBS 的过程。

1. 建立 DS 内存区，详见上一节。
2. 通过设置 IA32_PEBS_ENABLE MSR 寄存器的 Enable PEBS（位 24）启用 PEBS。
3. 设置中断和中断处理例程。PEBS 可以与分支监控和 NPEBS（non-precise event-based sampling）共享一个中断和中断处理例程。
4. 设置 MSR_IQ_COUNTER4 计数器和相关联的 CCCR 寄存器，以及一个或多个 ESCR 寄存器（指定要监视的事件）。只能使用 MSR_IQ_COUNTER4 计数器进行 PEBS 采样。

完成以上设置后，CPU 便会使用 MSR_IQ_COUNTER4 计数器对 ESCR 指定的事件进行计数，当计数器溢出时，CPU 便会自动将当时的寄存器内容以 PEBS 记录的形式写到 DS 内存区中的 PEBS 缓冲区中。当 PEBS 缓冲区已满（或满足 DS 管理区中定义的中断条件）时，CPU 便会产生性能监视中断（PMI），并转去执行对应的中断处理例程（称为 DS ISR）。DS ISR 应该将 DS 内存区中的信息转存到文件中，并清空已满

的缓冲区、复位计数器值，然后返回。

5.5.4 架构性的性能监视机制

2006 年推出的 Core Duo 和 Core Solo 处理器将 CPU 内的性能监视设施分为两类，一类是架构性的（Architectural），另一类是非架构性的（Non-architectural）。所谓架构性就是说这部分机制会成为 IA-32 架构中的一个标准部分，会被以后的 IA-32 处理器所兼容。非架构性的仍然与处理器相关。

Core Duo 和 Core Solo 中引入的架构性性能监视设施主要包括。

- 有限数量的事件选择寄存器，名称为 IA32_PERFEVTSELx，第一个的地址为 0x186，其他寄存器的地址是连续的。
 - 有限数量的事件记数寄存器，名称为 IA32_PMCx，第一个的地址为 0xC1，其他寄存器的地址是连续的。
 - 用于检查性能监视机制支持情况的检测机制，即 CPUID 指令的 0xA 号分支（leaf），简称为 CPUID.0AH。详细描述请参见 IA-32 手册卷 2A 中关于 CPUID 指令的介绍。

图 5-12 显示了 IA32_PERFEVTSELx 寄存器的位布局。

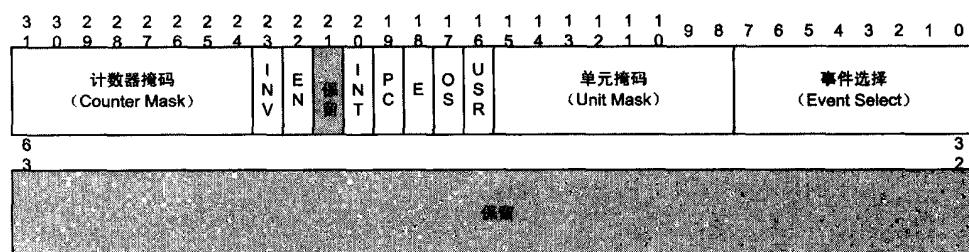


圖 5-12 IA32 PERFEVTSELx 寄存器

显而易见，其布局和位定义与 P6 处理器的 PerfEvtSel 寄存器（图 5-9）是完全一样的，其含义也基本相同，在此不再赘述。

5.5.5 Core 微架构处理器的性能监视机制

Core 微架构的 IA 处理器(例如 Core 2 Duo 和 Core 2 Quad 等)除了支持 Core Solo 和 Core Duo 处理器所引入的架构性能监视机制外,还提供了以下性能监视设施。

- 3个固定功能计数器,名为MSR_PERF_FIXED_CTR0~MSR_PERF_FIXED_CTR2,地址为0x309~0x30A。这3个计数器分别用来专门监视以下3个事件:INSTR RETIRED,ANY、CPU CLK UNHALTED,CORE和CPU CLK UNHALTED,CORE

REF。启用这 3 个计数器不需要设置任何事件选择寄存器(IA32_PERFEVTSELx)，只需要设置一个新引入的 MSR_PERF_FIXED_CTR_CTRL 寄存器。

- 3 个全局的计数器控制寄存器，用于简化频繁使用的操作，分别是。

MSR_PERF_GLOBAL_CTRL: 用于启用或禁止计数器，每个二进制位（保留位除外）对应一个专用的（MSR_PERF_FIXED_CTRx）或通用的性能计数器。将某一位设置为 1 便启用对应的计数器，清 0 便停止计数。因此通过这个寄存器，只要使用一条 WRMSR 指令便可以控制多个计数器。

MSR_PERF_GLOBAL_STATUS: 用于读取计数器的溢出状态，每个二进制位对应一个计数器或一种状态。通过这个寄存器，只要使用一条 RDMSR 指令便可以读到多个计数器及 PEBS 缓冲区的当前状态。

MSR_PERF_GLOBAL_OVF_CTRL: 用于清除计数器或缓冲区的溢出标志，位定义与 MSR_PERF_GLOBAL_STATUS 相对应。

5.5.6 资源

详细介绍如何利用硬件设施编写性能监控软件，超出了本书讨论的范围，感兴趣的读者可以参考以下开源项目或工具。

- Brinkley Sprunt 的 Brink and Abyss 项目，用于 Linux，链接如下：[\(http://www.eg.bucknell.edu/~bsprung/emon/brink_abbyss/brink_abbyss.shtml\)](http://www.eg.bucknell.edu/~bsprung/emon/brink_abbyss/brink_abbyss.shtml)。
- Don Heller 的 Rabbit (A Performance Counters Library) 项目 (Linux)：[\(http://www.scl.ameslab.gov/Projects/Rabbit/\)](http://www.scl.ameslab.gov/Projects/Rabbit/)。
- Mikael Pettersson 的 perfctr (Linux 下的 x86 性能监视计数器驱动)：[\(http://user.it.uu.se/~mikpe/linux/perfctr/\)](http://user.it.uu.se/~mikpe/linux/perfctr/)。
- 美国田纳西大学 ICL 实验室的 PAPI (Performance Application Programming Interface) 项目 (支持 Windows 和 Linux)：[\(http://icl.cs.utk.edu/papi/\)](http://icl.cs.utk.edu/papi/)。
- PCL (Performance Counter Library) 项目 (支持 Linux 和 Solaris 等操作系统)：[\(http://www.fz-juelich.de/zam/PCL/\)](http://www.fz-juelich.de/zam/PCL/)。
- 英特尔公司的 VTune 工具 (VTune Performance Analyzer)。

在性能优化方面，另一个宝贵的资源就是 IA-32 手册中的优化手册，全称为 *Intel® 64 and IA-32 Architectures Optimization Reference Manual*，可以从英特尔公司的网站自由下载它的电子版本 (<http://www.intel.com/products/processor/manuals/index.htm>)。

5.6 本章总结

本章介绍了 IA-32 CPU 的分支监视、记录和性能监视机制。这些机制为软件调试、优化和性能分析提供了硬件支持。在 5.2 节和 5.4 节中我们给出了两个示例性的应用，演示了如何利用 CPU 的分支记录机制来观察 CPU 的运行轨迹。5.5 节介绍了性能监视机制，并列出了一些资源。

参考文献

1. Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3B. Intel Corporation
2. Intel® 64 Architecture x2APIC Specification. Intel Corporation

机器检查架构 (MCA)

在软件开发中，我们经常使用写日志或 print 类语句将程序运行时遇到的错误情况（比如函数调用失败）记录到文件中或打印到屏幕上，以辅助调试。当 CPU 执行程序指令时，它也可能遇到各种错误情况，包括内部或外部（比如前端总线、内存或 MCH）的故障，这时它该如何处理呢？

早期针对个人电脑设计的 CPU 检测到硬件错误时，通常的办法是立即重新启动，以防止继续运行而造成更严重的后果。但随着 CPU 的高速化和复杂化，以及个人电脑上运行的重要任务越来越多，人们意识到应该有一种机制来报告 CPU 检测到的硬件错误供调试分析时使用。

那么如何让 CPU 来报告硬件错误呢？因为错误发生在 CPU 内部或前端总线一级，所以 CPU 比其上运行的软件更清楚错误的原因，按照“谁拥有最多知识谁就承担职责”的原则，CPU 应该负责记录下发生的错误及发生错误时的相关信息。但是 CPU 只能直接读写寄存器或内存这些临时性存储器，记录在这些地方的信息，计算机一旦重新启动，信息就会丢失。所以要解决这个问题还必须有软件的配合。于是，一种很自然的方法是，CPU 先收集好要记录的信息，并把它们存储到特定的寄存器或内存区域中，然后通过产生异常的方式把控制权交给软件，接下来，软件将这些信息写到外部存储器（如硬盘）上永久记录下来。这便是 IA-32 处理器的机器检查（Machine Check）机制的基本原理。

最早引入机器检查机制的 IA-32 处理器是奔腾处理器，其后推出的 P6 和奔腾 4 系列处理器进一步强化了该功能，并将其纳入 IA-32 架构规范，称为机器检查架构（Machine Check Architecture，简称 MCA）。

通过 CPUID 指令可以检查处理器对机器检查机制的支持情况，EDX 寄存器的 MCA（位 14）和 MCE（位 7）分别表示处理器是否实现了机器检查架构和机器检查异常。

下面便从奔腾处理器的机器检查异常（MCE）入手，按照由简到繁的顺序介绍 MCA 的工作原理和使用方法。

6.1 奔腾处理器的机器检查机制

奔腾处理器的机器检查 (MC) 机制又被称为内部错误检测 (internal error detection)，其主要设施包括。

- 用以记录错误的机器检查地址寄存器 (Machine Check Address Register，简称 MCAR)，以及机器检查类型寄存器 (Machine Check Type Register，简称 MCTR)。二者在 IA-32 手册中的名字分别为 P5_MC_ADDR 和 P5_MC_TYPE。
- 用以向系统软件汇报机器检查错误的机器检查异常 (Machine Check Exception，简称#MC)，以及 CR4 寄存器中的 MCE 标志 (CR4[MCE]，位 6)，通过 MCE 标志可以启用或禁止机器检查异常。机器检查异常的向量号是 18，通常操作系统会设置异常处理例程来处理该异常。
- 用以向系统芯片组报告错误的管脚信号，包括报告地址校验错误的 APCHK#信号、数据校验错误的 PCHK#信号和内部奇偶或冗余 (Functional Redundancy Check) 校验的 IERR#信号。
- 用以接收芯片组报告的总线错误的 BUSCHK#信号。

下面我们来介绍以上设施的用法。

为了校验地址和数据，奔腾处理器配备了 8 个数据校验信号 (pin) DP[7:0] (每一位对应于 64 位数据总线的一个字节) 和 1 个地址校验信号 AP (Address Parity)。如果 CPU 检测到地址信号奇偶校验错误，那么它就会置起 (assert) APCHK#信号 (因为是低电平有效，所以是置低)，通知系统有错误发生。对于该类错误，奔腾处理器将错误处理的任务交给了主板上的系统芯片组。

如果 CPU 当从内存中读数据时检测到数据信号奇偶校验错误，它就会置起 PCHK# 信号，通知系统有错误发生，同时如果 PEN# (Parity Enable) 信号有效，那么 CPU 会将这一次总线周期 (bus cycle) 的地址写入机器检查地址寄存器 MCAR 中，并将这次总线周期的类型参数记录在寄存器检查类型寄存器 MCTR 中。此外，如果 CR4 寄存器中的 MCE 标志 (bit 6) 为 1，那么 CPU 会产生机器检查异常，目的是向系统软件报告有错误发生。然后 CPU 会转去执行机器检查异常处理例程 (通常是操作系统的一部分)。异常处理例程可以通过读取 MCAR 和 MCTR 寄存器的内容了解发生错误的地址和错误类型，并采取进一步的措施。

P5_MC_ADDR 和 P5_MC_TPYE 都是 64 位的 MSR 寄存器，可以通过 RDMSR 来访问。P5_MC_ADDR 用于存放失败总线周期的物理地址，P5_MC_TPYE 用于存放失败总线周期的类型。P5_MC_TPYE 只使用了低 5 位，位布局如图 6-1 所示。

CHK 位为 1，表示 P5_MC_ADD 和 P5_MC_TPYE 中包含有效的数据。使用 RDMSR 读取寄存器后，该位会自动复位为 0。W/R、D/C 和 M/IO 位用来表示当错误发生时 W/R#、

D/C# 和 M/IO# 信号（均为奔腾处理器的管脚信号）的值，这些信号值代表了总线周期的类型：读或写、数据或代码、访问内存或 IO。LOCK 位表示当时 LOCK# 信号是否有效。



图 6-1 P5_MC_TPYE 寄存器

除了奇偶校验错误，当 CPU 检测到 BUSCHK# 信号（输入信号）被置起时（通常是芯片组在一个总线周期结束时设置此信号，表示该总线周期没有成功完成），CPU 也会将当时的总线周期地址和类型详细记录到 P5_MC_ADDR 和 P5_MC_TPYE 寄存器中。在记录后，如果 CR4[MCE] 为 1，即允许机器检查异常，那么 CPU 便会产生一个机器检查异常。

归纳一下，奔腾处理器的 MCA 机制能够记录的机器检查错误有以下两种。

1. 读周期（read cycle）中的数据奇偶校验错误。
2. 没有成功完成的总线周期，也就是系统内存控制器（MCH）通过 BUSCHK# 信号向 CPU 报告的失败总线周期。

6.2 MCA

奔腾之后的 P6 和奔腾 4 系列处理器对奔腾引入的机器检查机制做了改进和增强，并将其纳入 IA-32 架构规范，称为机器检查架构（Machine Check Architecture，简称 MCA）。

6.2.1 概览

首先，与奔腾处理器的架构相比，扩展后的 MCA 可以报告更多类型的错误，包括。

- 前端总线（FSB）上的总线事务错误（transaction error）。
- 内部高速缓存或 FSB 上的 ECC 错误，不论是可纠正的 ECC 还是不可纠正的 ECC。
- FSB 上或内部的奇偶校验错误。
- 内部高速缓存或 TLB 中的存储错误。TLB 是 Translation Lookaside Buffer 的缩写，TLB 位于 CPU 内部，用于缓存页面表，以减少在将虚拟内存地址转化为物理内存地址时对内存的访问次数。

从实现方面看，仍然可以将 MCA 的设施概括为机器检查异常和机器检查寄存器两个方面。机器检查异常的工作方式与奔腾相比没什么变化。变化很大的是机器检查寄存器，其数量明显增加了，除了一套全局控制寄存器（IA32_MCG_xxx）外，还有

多组与不同硬件单元相对应的错误报告寄存器(参见图6-2)。

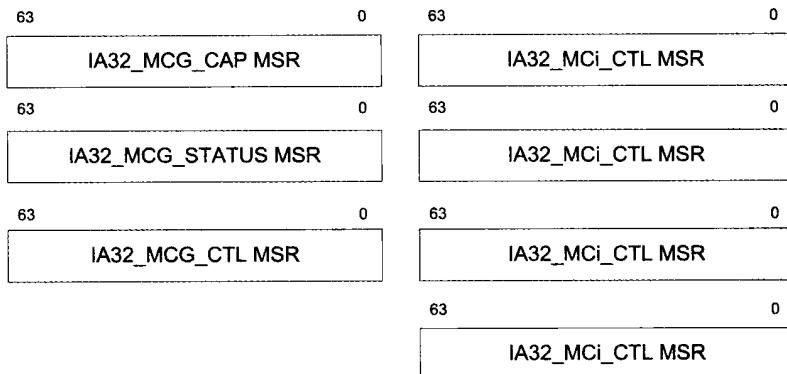


图6-2 机器检查架构的寄存器

每组错误报告寄存器被称为一个 bank，包含一个控制寄存器(IA32_MCI_CTL MSR)、一个状态寄存器(IA32_MCI_STATUS MSR)、一个地址寄存器(IA32_MCI_ADDR MSR)和一个附加信息寄存器(IA32_MCI_MISC MSR)。因为这些寄存器都是MSR寄存器(64位)，因此可以通过RDMSR指令来访问。IA-32架构规定第一个错误报告寄存器(IA32_MCG_CTL_MSR)的MSR地址总是400H，因此系统软件可以很方便地遍历所有的错误报告寄存器。错误报告寄存器的具体组数因CPU型号不同而有所不同，P6有5组，P4有4组。具体的组数是记录在机器检查全局寄存器IA32_MCG_CAP中的。

6.2.2 MCA全局寄存器

下面我们来详细认识一下MCA的每个全局寄存器。

IA32_MCG_CAP寄存器：只读寄存器，用以描述MCA的具体实现情况。包含以下位域(如图6-3所示)。

- Count(位0~7)：错误报告寄存器组数。
- MCG_CTL_P(Control MSR Present)(位8)：用以表示是否实现了IA32_MCG_CTL寄存器，如果实现，则为1，否则为0。
- MCG_EXT_P(Extended MSRs Present)(位9)：用以表示是否实现了扩展的机器检查状态寄存器(从地址180H开始的MSR寄存器)。如果实现，则为1，否则为0。
- MCG_EXT_CNT(Extended MSRs Count)(位16~23)：用以表示扩展的机器检查状态寄存器的数量(从地址180H开始的MSR寄存器)。仅当MCG_EXT_P位为1时才有意义。

6	2	2	2	2	1	1	1	1	1	1	1	1	1	0
3	4	3	2	1	0	9	8	7	6	5	4	3	2	1
	保留		MCG_EXT_CNT		保留		E	C	T	L	P			Count
							X	T	T	L	P			

图 6-3 IA32_MCG_CAP MSR 寄存器

P6 处理器的 MCG_CAP 寄存器和以上介绍的 IA32_MCG_CAP 寄存器在 0 到 8 位的含义相同，9 到 63 位保留。

IA32_MCG_STATUS 寄存器：当机器检查异常发生时，用以描述处理器的状态。包含以下位域（如图 6-4 所示）。

- RIPV (Restart IP Valid) (位 0): 表示是否可以安全地从异常发生时压入栈中的指令指针处重新开始执行。1 表示可以，0 表示不可以。
- EIPV (Error IP Valid) (位 1): 当为 1 时，表示异常发生时压入栈中的指令指针与导致异常的错误直接关联。0 表示二者可能无关。
- MCIP (Machine Check In Progress) (位 2): 当为 1 时表示已经产生了机器检查异常。软件可以设置或清除这个标记。当该位为 1 时，如果再有机器检查异常发生，那么 CPU 会进入关机 (shutdown) 状态（停止执行指令，直到收到 NMI 中断、SMI 中断、硬件复位或 INIT#信号）。

6	3	2	1	0
3	保留			
2		M	E	R
1		C	I	I
0		I	P	P
		V	V	V

图 6-4 IA32_MCG_STATUS 寄存器

IA32_MCG_CTL 寄存器：用来启用或禁止机器检查异常报告功能。IA-32 手册没有明确定义每个位对应的具体机器异常内容，只是说全部写为 1，会启用所有异常报告，全部写为 0，会禁止所有异常报告。本寄存器只有当 IA32_MCG_CAP 的 MCG_CTL_P 位为 1 时才存在。

6.2.3 MCA 错误报告寄存器

下面让我们介绍 MCA 的各组错误报告寄存器的工作方式。每组错误报告寄存器都包含一个控制寄存器 (IA32_MCi_CTL MSR)、一个状态寄存器 (IA32_MCi_STATUS MSR)、一个地址寄存器 (IA32_MCi_ADDR MSR) 和一个附加信息寄存器 (IA32_MCi_MISC MSR)。

IA32_MCi_CTL (控制寄存器，P6 称为 MCi_CTL) 寄存器的各个位分别用来启用或禁止报告与其对应的错误。因为 MSR 寄存器有 64 位，因此该寄存器最多可以控制

64种错误的报告与否。该寄存器的实际使用位数与处理器及其所对应的硬件单元有关。当修改这个寄存器时，处理器仅修改已经被使用的各个位。另外P6处理器仅允许向控制寄存器写全1或全0，并且建议只有BIOS代码才能使用MC0_CTL寄存器。

IA32_MCi_STATUS（状态寄存器，P6称为MCi_STATUS）用来表示错误的具体信息。其各个位的布局如图6-5所示。

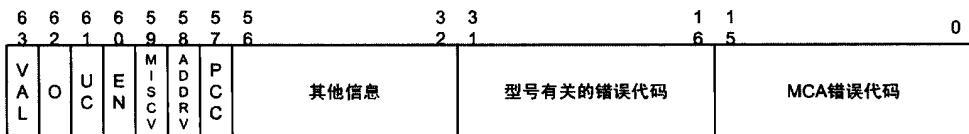


图6-5 IA32_MCi_STATUS寄存器

MCA Error Code（位0~15）：MCA错误代码，定义方式我们将在下文介绍。该错误代码的含义对于所有IA-32处理器都是一致的。

Model-Specific Error Code（位16~31）：型号相关的错误代码。与处理器型号有关的错误代码。该错误代码的含义会因处理器的型号不同而可能不同，IA-32手册卷3B的附录E中列出了各种型号处理器的错误码定义。

Other Information（位32~56）：错误相关信息。该信息与处理器型号有关。

PCC（Processor Context Corrupt）（位57）：如果为1，则表示处理器的状态可能已经被发生的错误所破坏，不能安全地恢复执行。如果为0，则表示发生的错误没有影响处理器的状态。

ADDRV（MCi_ADDR register valid）（位58）：如果为1，则表示IA32_MCi_ADDR寄存器中包含有效的错误发生地址。如果为0，则表示IA32_MCi_ADDR寄存器不存在或不包含有效的地址信息。

MISCV（MCi_MISC register valid）（位59）：如果为1，则表示IA32_MCi_MISC寄存器中包含有效的错误相关信息。如果为0，则表示IA32_MCi_MISC寄存器不存在或不包含有效的附加信息。

EN（Error enabled）（位60）：表示IA32_MCi_CTL寄存器中的相应位是否启用该错误。

UC（Uncorrected error）（位61）：如果为1，则表示处理器没有或不能纠正错误情况，如果为0，表示处理器能够纠正错误情况。

OVER（Error overflow）（位62）：如果为1，则表示当上一个错误还没有清理时又发生了一个错误。当CPU在报告错误时，如果发现目标IA32_MCi_STATUS寄存器

的 VAL 位为 1，也就是已经有错误信息在寄存器中，那么 CPU 会根据以下规则决定是否覆盖寄存器中的错误信息：启用的错误可以覆盖没有启用的错误；不可纠正的错误可以覆盖可以纠正的错误；不可纠正的错误不可以覆盖还有效的上一个不可纠正的错误。如果可以覆盖，那么处理器会用新的错误信息覆盖前一个错误信息，并将此位置为 1，软件负责清除此位。

VAL (MCi_STATUS register valid) (位 63)：表示本寄存器中的信息是否有效。处理器在写入错误信息后设置此位，软件在读取该寄存器后应该清除此位。

IA32_MCi_ADDR (地址寄存器，P6 称为 MCi_ADDR) 用来记录产生错误的代码地址或数据地址。只有当 IA32_MCi_STATUS 的 ADDRV 位为 1 时，本寄存器中的内容才有效。

该寄存器中的地址因错误情况不同，可能是段内的偏移地址、线性地址或 36 位的物理地址。

IA32_MCi_MISC (附加信息寄存器，P6 称为 MCi_MISC) 用来记录与错误相关的附加信息。只有当 IA32_MCi_STATUS 的 MISCV 位为 1 时，本寄存器中的内容才有效。

6.2.4 扩展的机器检查状态寄存器

从奔腾 4 和至强处理器开始，IA-32 处理器还包含了数量不等的 MCA 扩展状态寄存器，用来进一步记录机器检查异常发生时的处理器状态。具体实现情况和数量可以通过读取 IA32_MCG_CAP 寄存器的 MCG_EXT_P 位和 MCG_EXT_CNT 位域获得。如果支持扩展的机器检查状态寄存器，那么这些寄存器的起始地址是 180H，如表 6-1 所示。

表 6-1 MCA 扩展状态寄存器

MSR	地址	描述
IA32_MCG_EAX	180H	机器检查错误发生时 EAX 寄存器的值
IA32_MCG_EBX	181H	机器检查错误发生时 EBX 寄存器的值
IA32_MCG_ECX	182H	机器检查错误发生时 ECX 寄存器的值
IA32_MCG_EDX	183H	机器检查错误发生时 EDX 寄存器的值
IA32_MCG_ESI	184H	机器检查错误发生时 ESI 寄存器的值
IA32_MCG_EDI	185H	机器检查错误发生时 EDI 寄存器的值
IA32_MCG_EBP	186H	机器检查错误发生时 EBP 寄存器的值
IA32_MCG_ESP	187H	机器检查错误发生时 ESP 寄存器的值
IA32_MCG_EFLAGS	188H	机器检查错误发生时 EFLAGS 寄存器的值
IA32_MCG_EIP	189H	机器检查错误发生时 EIP 寄存器的值。
IA32_MCG_MISC	18AH	只使用 1 位 (位 0)，如果为 1，则表示在操作调试存储区 (Debug Store) 时发生了内存页错误 (或 Page Assist)
IA32_MCG_RESERVED1 ~ IA32_MCG_RESERVEDn	18BH~18AH+n	保留供将来使用

需要说明的是，以上寄存器属于读或写零(read/write zero)寄存器，意思是软件可以读这些寄存器，或者向这些寄存器写零。如果软件企图向这些寄存器写入非零值，那么将导致一般性保护异常(#GP)。当硬件重启(开机或RESET)时这些寄存器会被清零，但是当软件重启(INIT)时，这些寄存器的值会被保持不变。

6.2.5 MCA 错误编码

当处理器检测到机器检查错误时，它会向对应的IA32_MCI_STATUS寄存器的低16位写入一个错误代码(MCA Error Code)，并将该寄存器的VAL位置为1。视错误情况的不同，处理器还可能向16到31位写入一个与处理器型号有关的错误码。这里我们介绍一下MCA错误码的编码和解析方法。MCA错误代码的含义对于所有IA-32处理器都是一致的。

首先，所有错误码被分为简单错误码和复合错误码两种。简单错误码的位组合及含义如表6-2所示。

表6-2 IA32_MCI_STATUS寄存器的MCA简单错误码

错误	位编码	含义
无错误	0000 0000 0000 0000	没有错误
未分类的错误	0000 0000 0000 0001	还没有分类的错误
微指令(Microcode)ROM奇偶校验错误	0000 0000 0000 0010	处理器内部的微指令(Microcode)ROM中存在奇偶校验错误
外部错误	0000 0000 0000 0011	来自其他处理器的BINIT#信号导致该处理器进入机器检查
FRC错误	0000 0000 0000 0100	FRC(Functional Redundancy Check)错误
内部未分类错误	0000 01xx xxxx xxxx	处理器内部的未分类错误

复合错误码用来描述某一类型的错误，同一类型中又使用不同的位域来进一步分类。例如1xxxx是TLB错误，xxxx这4位又分为TTLL两个位域，TT用来代表事务类型(Transaction Type)，LL用来代表缓存级别(memory hierarchy level)。详情见表6-3到表6-7。

表6-3 MCA复合错误码的编码规则

类型	模式	译码
TLB错误	0000 0000 0001 TTLL	{TT}TLB{LL}_ERR
Memory Hierarchy Errors	0000 0001 RRRR TTLL	{TT}CACHE{LL}_{RRRR}_ERR
内部时钟	0000 0100 0000 0000	
总线或互连错误	0000 1PPT RRRR IILL	BUS{LL}_{PP}_{RRRR}_{II}_{T}_ERR

表6-4 TT(Transaction Type)位域的编码

事务类型	助记符	二进制编码
指令	I	00
数据	D	01
通用(generic)	G	10

表 6-5 LL (Memory Hierarchy Level) 位域的编码

Hierarchy Level	助记符	二进制编码
0 级	L0	00
1 级	L1	01
2 级	L2	10
通用 (generic)	LG	11

表 6-6 RRRR (Request) 位域的编码

请求类型	助记符	二进制编码
一般错误 (generic error)	ERR	0000
一般读 (generic read)	RD	0001
一般写 (generic write)	WR	0010
读数据 (data read)	DRD	0011
写数据 (data write)	DWR	0100
取指 (instruction fetch)	IRD	0101
预取 (prefetch)	PREFETCH	0110
Eviction	EVICT	1111
Snoop	SNOOP	1000

表 6-7 PP (Participation)、T (Time-out) 和 II (Memory or I/O) 位域的编码

位域	事务	助记符	二进制编码
PP	本处理器发起请求	SRC	00
PP	本处理器响应请求	RES	01
PP	本处理器作为第三方观察到错误	OBS	10
PP	通用 (Generic)		11
T	请求超时	TIMEOUT	1
T	请求没有超时	NOTTIMEUT	0
II	内存访问	M	00
II	保留		01
II	I/O	IO	10
II	其他事务		11

6.3 编写 MCA 软件

本章篇首我们提到过, MCA 的整体设计思路是通过硬件与软件的配合来实现对硬件错误的记录 (logging) 和处理 (handling) 的。

6.3.1 基本算法

简单来说, 当 CPU 自身在运行过程中发生了错误, 或者系统的其他部件 (如其他 CPU、内存或 MCH) 发生了错误, 并通过某种方式 (比如设置 CPU 的某些管脚信号) 告知了 CPU, 那么, CPU 会先将事故现场的相关信息记录到寄存器中, 然后通过以下方式通知软件。

如果是不可纠正的错误，而且机器检查异常(MC#)被允许(CR4寄存器)，那么CPU便会产生一个机器检查异常，然后转去执行软件事先设置好的异常处理例程。

如果是可纠正的错误，或者机器检查异常(MC#)被禁止，那么CPU不会产生异常(只是将错误情况记录在那儿，期望软件来读取)。对于这类情况，软件应该定期查询MCA寄存器来检测是否曾经有机器检查错误发生。

清单6-1给出了机器检查异常处理例程的伪代码，该代码的最初版本来源于IA-32手册(卷3A 14.8.2)，但笔者对其做了部分修改。

清单6-1 机器检查异常处理例程

```

1 IF CPU supports MCE
2   THEN
3     IF CPU supports MCA
4       THEN
5         call errorlogging routine; (* returns restartability *)
6       ELSE (* Pentium(R) processor compatible *)
7         READ P5_MC_ADDR
8         READ P5_MC_TYPE;
9         report RESTARTABILITY to console;
10        FI;
11      FI;
12      IF error is not restartable
13        THEN
14          report RESTARTABILITY to console;
15          abort system;
16        FI;
17      IF CPU supports MCA
18        THEN CLEAR MCIP flag in IA32_MCG_STATUS;
19      FI;

```

第1行到第3行是通过CPUID指令来检查处理器对MCE和MCA的支持情况(EDX的相应位)的。第7~9行是针对奔腾处理器的特殊处理的。

清单6-2给出了查询和记录机器检查错误的伪代码。

清单6-2 记录机器检查错误的伪代码

```

1 Assume that execution is restartable;
2 IF the processor supports MCA
3   THEN
4     FOR each bank of machine-check registers
5       DO
6         READ IA32_MCI_STATUS;
7         IF VAL flag in IA32_MCI_STATUS = 1
8           THEN
9             IF ADDR flag in IA32_MCI_STATUS = 1
10               THEN READ IA32_MCI_ADDR;
11             FI;
12             IF MISCV flag in IA32_MCI_STATUS = 1
13               THEN READ IA32_MCI_MISC;
14             FI;
15             IF MCIP flag in IA32_MCG_STATUS = 1
16               (* Machine-check exception is in progress *)
17               AND PCC flag in IA32_MCI_STATUS = 1
18               OR RIPV flag in IA32_MCG_STATUS = 0
19               (* execution is not restartable *)

```

```

20      THEN
21          RESTARTABILITY = FALSE;
22          return RESTARTABILITY to calling procedure;
23      FI;
24      Save time-stamp counter and processor ID;
25      Set IA32_MCI_STATUS to all 0s;
26      Execute serializing instruction (i.e., CPUID);
27      FI;
28      OD;
29  FI;

```

从第 4 行开始一个 FOR 循环，每次处理一组（Bank）错误报告寄存器。可以通过查询 IA32_MCG_CAP（或 P6 的 MCG_CAP）寄存器得到错误报告寄存器的组数。

第 15~23 行用来检查是否可以返回发生错误的地方并重新执行，第 17 行判断 PCC（Process Context Corrupt）标志位，该位为 1 表示处理器的上下文已经损坏，第 18 行判断 RIPV（Restart IP Valid）标志，该位为 0 表示不可以恢复执行。IA-32 手册中的原始代码（Example 14-3）在第 18 行处的 OR 位置是 AND，笔者认为不当，因为这两个条件之一满足，就不可以恢复执行了。

6.3.2 简单示例

下面通过一个使用 MFC 编写的小程序 MCAViewer 来演示如何检测 CPU 对 MCA 的支持情况，以及如何通过查询方式读取错误报告寄存器的内容。图 6-6 是 MCAViewer 在笔者使用的电脑上运行的一个执行画面。

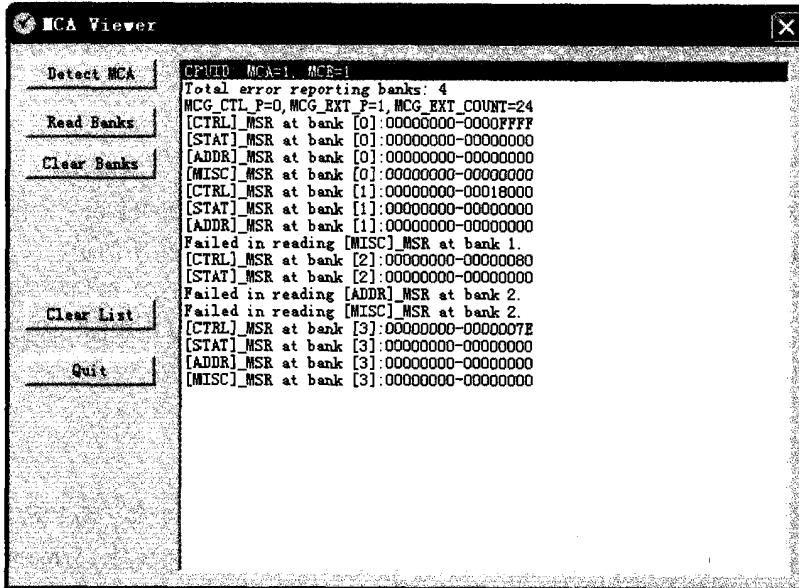


图 6-6 MCAViewer 的执行界面

从图 6-6 中可以看到，运行 MCAViewer 的 CPU 既支持 MCE，又支持 MCA，说

明一定是 P6 或 P6 后的 CPU(实际上是 Pentium 4)。列表的第二行显示该 CPU 共配备了 4 组错误报告寄存器。清单 6-3 给出了如何读取各个错误报告寄存器的源代码，整个程序的完整代码位于 chap06\McaViewer 目录中。

清单 6-3 读取 MCA 的错误报告寄存器

```

1 #define MCA_MCIBANK_BASE 0x400
2 void CMcaPoller::PollBanks(CListBox &lb)
3 {
4     MSR_STRUCT msr;
5     TCHAR szMsg[MAX_PATH];
6     //
7     LPCTSTR szBankMSRs[] = {"CTRL", "STAT", "ADDR", "MISC"};
8     //There are four MSRs per bank, we make the name in
9     // same length to improve display
10    //
11
12    if(m_nTotalBanks<=0)
13        DetectMCA(lb);
14    if(m_nTotalBanks<=0)
15        return;
16
17    msr.MsrNum=MCA_MCIBANK_BASE;
18    for(int i=0;i<m_nTotalBanks;i++)
19    {
20        //loop each
21        for(int j=0;j<4;j++)
22        {
23            if(m_DvrAgent.RDMSR(msr)<0)
24                sprintf(szMsg,"Failed in reading [%s]_MSR at bank %d.",
25                        szBankMSRs[j],i);
26            else
27                sprintf(szMsg,"%[s]_MSR at bank [%d]:%08X-%08X",
28                        szBankMSRs[j],i,msr.MsrHi,msr.MsrLo);
29            lb.AddString(szMsg);
30            msr.MsrNum++;
31        }
32    }
33 }

```

因为 IA-32 架构将错误报告寄存器的起始地址确定为 0x400，每组包含 4 个寄存器，所以只要使用两层循环就可以很简单地遍历所有寄存器了。值得说明的是，并不是每一组都全部实现 4 个寄存器，应该根据每组的状态寄存器(STATUS)来判断该组是否包含地址(ADDR)和附加信息(MISC)寄存器。以上代码省略这个判断，因此图 6-6 中包含了几条读取 MISC 和 ADDR 寄存器失败的记录。

6.3.3 在 Windows 系统中的应用

在现实的计算机系统中，系统软件(操作系统)会负责处理机器检查异常，并提供接口给其他驱动程序或上层的应用程序。以 Windows 操作系统为例，Windows XP 操作系统的硬件抽象层(HAL)中包含了对 MCA 的基本支持，并可通过 HalSetSystemInformation()注册更复杂的 MCA 处理例程。

```

McaDriverInfo.ExceptionCallback = MCADriverExceptionCallback;
McaDriverInfo.DpcCallback = MCADriverDpcCallback;
McaDriverInfo.DeviceContext = McaDeviceObject;

Status = HalSetSystemInformation(
    HalMcaRegisterDriver,
    sizeof(MCA_DRIVER_INFO),
    (PVOID)&McaDriverInfo
);

```

Windows DDK 中包含了一个完整的例子，名为 IMCA (DDKROOT\src\kernel\mca\imca)。感兴趣的读者可以进一步阅读其代码或对其进行编译安装。

当有不可纠正的机器检查错误发生时，Windows 会出现蓝屏 (BSOD)，并终止系统运行，对应的 Bug Check 编号是 0x9C (MACHINE_CHECK_EXCEPTION)，错误参数中会包含我们前面介绍的错误报告寄存器中的内容。

Windows Vista 操作系统设计了更完善的机制来管理硬件一级的错误，称为 WHEA (Windows Hardware Error Architecture)，我们将在第 17 章中详细介绍操作系统对 MCA 的支持机制及 WHEA。

6.4 本章总结

本章介绍了 IA-32 CPU 的机器检查架构 (MCA)。MCA 既代表了 CPU 自身的可调试性，同时也对调试系统级错误及硬件错误提供了支持。

参考文献

1. Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3A. Intel Corporation
2. Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3B. Intel Corporation
3. Tom Shanley. The Unabridged Pentium 4: IA-32 Processor Genealogy. Addison Wesley, 2004





JTAG 调试

大多数软件调试任务是在可以启动的系统上进行的。这些系统上已经具有了基本的运行环境，可以启动到图形化的操作界面或某种形式的命令行，可以运行调试器或基本的调试工具。那么，如果在基本的启动过程中出现故障，比如系统开机后还没有启动到任何可以操作的界面就停滞不前了，应该如何调试呢？当开发一个新的计算机系统（比如主板）或基本的启动软件及系统软件时也有类似的问题。针对这些问题的一种基本解决方案就是使用基于 JTAG 技术的硬件调试工具。硬件调试工具的最大优点就是不需要在目标系统上运行任何软件，可以在目标系统还不具备基本的软件环境时进行调试，因此，JTAG 调试非常适合调试 BIOS、操作系统的加载程序，以及使用软件调试器难以调试的特殊软件。

本章我们将首先介绍硬件调试工具的简单发展历程（7.1 节），然后介绍 JTAG 的工作原理（7.2 节）和典型应用（7.3 节）。第 7.4 节将介绍 IA-32 CPU 的 JTAG 支持、ITP/XDP 调试端口，以及典型应用。

7.1 简介

随着印刷电路板（Print Circuit Board，简称 PCB）和集成电路（Integrated Circuit，简称 IC）的不断发展和普及，验证和调试 PCB 及 IC 的难度也在不断加大。早期的芯片大多管脚较少且封装工艺简单，例如 8086 有 40 个管脚，使用的是 DIP（Dual In-Line Package）封装方式。当这样的芯片安插在电路板上后，可以很容易地测试到每个管脚的信号。另外，当时电路板的面积也相对较大，线路比较稀疏，可以比较容易测量到各个管脚或元器件的电压、波形等信息。

不过，单纯观察某几个管脚的信号通常难以解决比较复杂的问题，比如某个芯片与其他芯片间的通信问题。于是在 20 世纪 70 年代出现了在线仿真（In-Circuit Emulation）技术，简称 ICE。

7.1.1 ICE

简单来说，ICE 调试就是用一个专门的仪器（调试工具）暂时替代要调试的芯片（通常是微处理器），让其与被调试系统的其他硬件一起工作，并运行被调试的软件，这个仪器会模拟原来芯片的功能，因此人们通常将该仪器称为仿真器（emulator）。由于仿真器是针对调试目的而设计的，所以它集成了各种调试功能，比如单步执行、观察寄存器等。

典型的 ICE 调试通常由 3 大部分组成：被调试的目标系统（又称下位机）、用于调试的主机（又称上位机）和仿真器。仿真器通过专门设计的接口接入到目标系统中，并且通过电缆和上位机相连接。

ICE 调试技术一出现，很快就被广泛地使用，直到今天，ICE 调试仍然是调试嵌入式系统的一种常用方法。可以进行单机内核调试的著名软件调试器 SoftICE 的名称就来源于硬件仿真器，暗指具有类似于 ICE（In-Circuit Emulation）的强大功能。

ICE 调试的主要问题是调试不同类型或型号的芯片（微处理器）通常需要使用不同的仿真器。因为仿真器与目标系统的连接方式是与目标芯片的封装结构紧密相关的，要使仿真器连入目标系统，通常必须使用针对目标芯片开发的仿真器。为了缓解这一问题，很多调试工具厂商将与目标系统连接的部分独立出来，称为仿真头（header），以便增加仿真器的适用面，但仍没有从根本上解决问题。另外，芯片的升级速度通常超过仿真器的发展速度，一个新的芯片或新的版本出现后，能够仿真它的仿真器要过一段时间才能出现，难以及时满足市场的需要。

7.1.2 JTAG

当传统方式的 ICE 调试遇到以上问题的同时，最原始的手工测试方法（直接测量管脚或元器件）也变得越来越困难。一方面，随着集成电路技术的发展，在芯片功能不断增强的同时，芯片的管脚数量不断增加，封装方式也不断革新。如采用 LGA775 封装的奔腾 4 CPU 有 775 个管脚，需要通过专门的插槽固定在主板上，这样不仅从正面看不到管脚，而且背面也很难找到，因为能够支持奔腾 4 CPU 的主板，其印刷电路板大多是多层的（通常为 4 层）。另一方面，随着芯片速度和信号频率的不断提高，PCB 版的布线要求也越来越严格，线路的长短和走向都必须遵循严格的规定。因此，当我们观察主板电路时，会发现密如蛛网的线路穿上传下。对于这样密集的电路，直接测量线路信号不仅非常困难，而且容易损坏板上的芯片或元器件。

如果不解决以上问题，就会影响和制约集成电路的进一步发展，人们很快就意识到了问题的严重性。于是在 1985 年，一些大的电子和半导体厂商联合成立了一个工作小组，目的是寻找一种统一的方案来解决电路板级（board level）的测试问题，然后使

其成为一个工业标准。这个小组的名称叫 Joint Test Action Group，简称为 JTAG。JTAG 小组提出的方案在 1990 年得到了 IEEE（电气和电子工程师协会）的批准，并被定名为 Test Access Port and Boundary Scan Architecture（测试访问端口和边界扫描架构），简称为 IEEE 1149.1—1990。由于该标准是由 JTAG 组织制定的，因此更多时候人们还是叫它 JTAG 标准。并将基于该标准的调试方法称为 JTAG 调试。

1993 年和 1995 年 IEEE 对 JTAG 标准做了两次补充，分别称为 IEEE 1149.1a—1993 和 IEEE 1149.1b—1995。

JTAG 标准推出后，得到了广泛的认可和支持，有着非常广泛的应用，本章后面的章节将集中介绍它的原理和应用。

7.2 JTAG 原理

JTAG 的核心思想就是将测试点和测试设施集成在芯片内部(build test facilities/test points into chip)，并通过一组标准的信号（接口）向外输出测试结果，这些标准信号被称为 TAP (Test Access Port) 信号。有了标准的 TAP 信号，那么基于 JTAG 技术的调试工具就可以与这个芯片进行通信，而不必关心它内部的实现细节。这样，一个 JTAG 调试工具就可以比较容易地调试很多种芯片。这与软件中的对象抽象思想非常类似。

为了支持 JTAG，芯片内部通常需要实现一个边界扫描链路和一个 TAP 控制器。下面我们将分别讨论。

7.2.1 边界扫描链路

在支持 JTAG 技术的芯片内会为每个需要测试的输入输出管脚配备一个移位寄存器单元（移位寄存器的一个位），称为边界扫描单元（boundary-scan cells，简称 BSC）。简单理解，BSC 就是一个可控制的信号采集器，可以让信号直接通过，也可以将信号（电平值）记录下来。

一个芯片内的多个边界扫描单元通常被串联起来，形成一个链路，被称为边界扫描链（Boundary-Scan Chain），如图 7-1 所示。

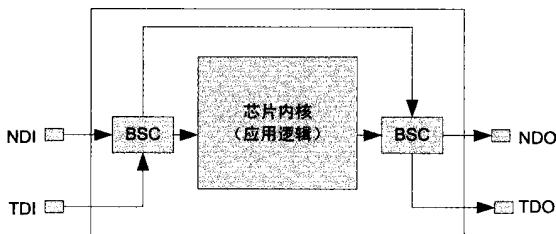


图 7-1 边界扫描示意图

图 7-1 画出了包含边界扫描单元的一个简单芯片的示意图，图中画出了针对两个正常信号（NDI 和 NDO）的 JTAG 链路，每个信号配备一个边界扫描单元（BSC），两个 BSC 串联起来形成一个简单的链路。中央的矩形表示芯片的内部逻辑，NDI（Normal Data Input）代表正常的数据输入信号，NDO 代表正常的数据输出信号，TDI 代表测试数据输入信号，TDO 代表测试数据输出信号。当芯片正常工作时，NDI 和 NDO 信号自由穿过 BSC，对芯片的本身逻辑不产生任何影响。在边界测试模式（boundary-test mode）下，根据需要，BSC 可以有两种工作模式。

对于外部测试（External Testing），也就是当测试该芯片与外围电路或其他芯片的互联情况时，输出管脚处的 BSC 可以将由 TDI 输入的测试触发（test stimulus）输出供外部器件；输入管脚处的 BSC 可以捕捉到来自外部的输入并将结果通过 TDO 移出给外界的调试器观察。在这种模式下，芯片本身的内部逻辑好像被替换掉，它的输入被发送到外部的调试器，它的输出也是外部调试器所指定的。因此通过这种方法，可以实现我们前面介绍的仿真调试（ICE）。

对于内部测试（Internal Testing），也就是当测试芯片内部的应用逻辑时，输入管脚处的 BSC 会将通过 TDI 输入的测试触发发给芯片，输出管脚处的 BSC 会将芯片的输出捕捉下来并通过 TDO 发给外部的调试器。这样可以很容易地监视到芯片的输出信号，解决了前面说的管脚信号难以测量的问题。

在对 BSC 有了基本的了解后，理解 JTAG 工作原理的另一个重要问题就是如何控制和访问 BSC。要回答这个问题，我们需要了解 JTAG 的控制机制，即测试访问端口（Test Access Port），简称为 TAP。

图 7-2 画出了一个支持 JTAG 调试/测试的芯片的更多细节。

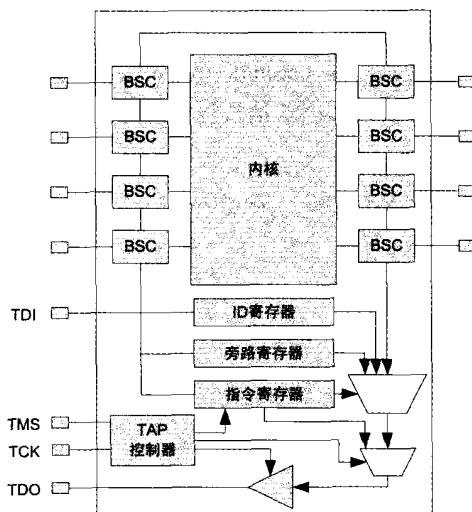


图 7-2 Test Access Port 工作原理示意图

下面我们从 3 个角度来理解这幅图：TAP 信号、TAP 控制器（TAP Controller）和 TAP 寄存器。

7.2.2 TAP 信号

JTAG 标准定义了以下 5 个标准信号供外部调试器和被调试芯片通信。

TCK (Test Clock): 测试时钟输入信号，TAP 的所有操作都是通过这个信号来驱动的。TCK 与芯片本身的时钟信号是分开的。这样做的好处是，只要它们符合 JTAG 标准，都可以通过一个 TCK 信号来驱动同一电路板上的多个不同频率的芯片。

TMS (Test Mode Selection): 测试模式选择输入信号，TMS 信号用来控制 TAP 状态机（将在下文详述）的转换。通过 TMS 信号，可以控制 TAP 在不同的状态间相互转换。TMS 信号在 TCK 的上升沿有效。

TDI (Test Data Input): 测试数据输入信号，TDI 是数据输入的通道。所有要输入到特定寄存器的数据都是通过 TDI 一位一位串行输入的（由 TCK 驱动）。TDI 信号在 TCK 的上升沿有效。

TDO (Test Data Output): 测试数据输出信号，TDO 是输出数据的通道。所有要从 JTAG 寄存器中输出的数据都是通过 TDO 一位一位串行输出的（由 TCK 驱动）。TDO 信号在 TCK 的下降沿有效。

TRST (Test Reset): 测试复位信号，TRST 可以用来对 TAP Controller 进行复位（初始化）。TRST 在 IEEE 1149.1 标准里是可选的。因为通过 TMS 也可以对 TAP Controller 进行复位（初始化）。

IEEE 1149.1 标准规定前 4 个信号是必须实现的。而且不论芯片的种类和其他管脚有多大差异，这几个信号的工作方式是不变的，这为调试工具的标准化提供了很好的基础。换句话说，芯片可以通过这 4 个或 5 个标准化的信号来隐藏其内部结构和外部形状（封装工艺）的差异，只要芯片按照标准实现这几个信号，那么它就可以与 JTAG 调试器通信，实现 JTAG 所支持的强大调试功能。这是 JTAG 标准被广泛支持的重要原因。

7.2.3 TAP 寄存器

JTAG 标准定义了如下几种寄存器，其中有些是必须实现的，有些根据芯片的设计需要，是可选实现的。

指令寄存器：用来选择需访问的数据寄存器，或者选择需要执行的测试。每个支持 JTAG 调试的芯片必须包含一个指令寄存器。

旁路 (Bypass) 寄存器：一位的移位寄存器，用以当不需要进行任何测试的时候，在 TDI 和 TDO 之间提供一条长度最短的串行路径。这样允许测试数据可以快速地通

过当前芯片到板上的其他芯片上去。

设备 ID 寄存器：用以标示芯片生产厂商及版本信息，使调试器可以自动识别当前调试的是什么芯片。

边界扫描（Boundary-Scan）寄存器：边界扫描寄存器的每一位就是我们前面介绍的一个边界扫描单元（BSC），因此，一个边界扫描链就是一个边界扫描寄存器，访问边界扫描寄存器就相当于访问边界扫描链中的边界扫描单元。换言之，可以通过操纵边界扫描寄存器，来实现对测试器件的输入输出信号进行观测和控制，以达到测试或调试的目的。

所有 TAP 寄存器可以分为指令寄存器和数据寄存器两类。旁路寄存器、设备 ID 寄存器和边界扫描寄存器都属于数据寄存器。除了设备 ID 寄存器外，其他几个都是必须实现的。JTAG 标准允许芯片设计厂商实现更多的数据寄存器和指令寄存器（称为私有指令寄存器）。

7.2.4 TAP 控制器

TAP 控制器（TAP Controller）既是驱动 TAP 各部件工作的发动机，又是指挥指令和数据寄存器协同工作的控制中枢。从实现来看，TAP 控制器是一个包含 16 个状态的有限状态机。在 TMS 信号的驱动下，TAP 在各个状态间切换，实现各种操作。图 7-3 画出了这个状态机的所有状态和转换关系。图中的箭头表示所有可能的状态转换流程。箭头旁边的数字表示选择该转换流程的条件，也就是 TMS 的值（电平）。举例来说，如果当前状态为 Capture-DR（捕捉数据寄存器），而且在 TCK 的下一个上升沿时 $TMS=0$ ，那么 TAP 控制器就进入 Shift-DR 状态（对当前数据寄存器移位）；如果在 TCK 的下一个上升沿时 $TMS=1$ ，那么 TAP 控制器进入 Exit1-DR 状态（退出移位状态）。

在 16 个状态中，有 6 个稳定状态：Test-Logic-Reset、Run-Test/Idle、Shift-DR、Pause-DR、Shift-IR 和 Pause-IR。即状态机可以停留在这些状态上。例如在 Shift-DR 状态，如果下一个 TMS 值是 0，那么状态机仍然处于该状态上，这样便实现了循环移位操作。

从图 7-3 中，还可以看到该状态机有两个主要的状态序列：针对数据寄存器的 XX-DR 序列和针对指令寄存器的 XX-IR 序列。从 Run-Test/Idle 状态开始，如果下一个 TMS 值是 0，那么状态机就停留在这个状态上，即处于空闲状态（idle）；如果是 1，那么状态机就进入 Select-DR-Scan 状态；如果再下一个 TMS 是 0，那么便进入 Capture-DR 状态，接下来可以选择对数据寄存器进行各种操作。如果当前状态是 Idle，下一个 TMS 是 1，那么便进入 Select-IR-Scan 状态，用以选择进入对指令寄存器进行各种操作的状态，依此类推，不再赘述。

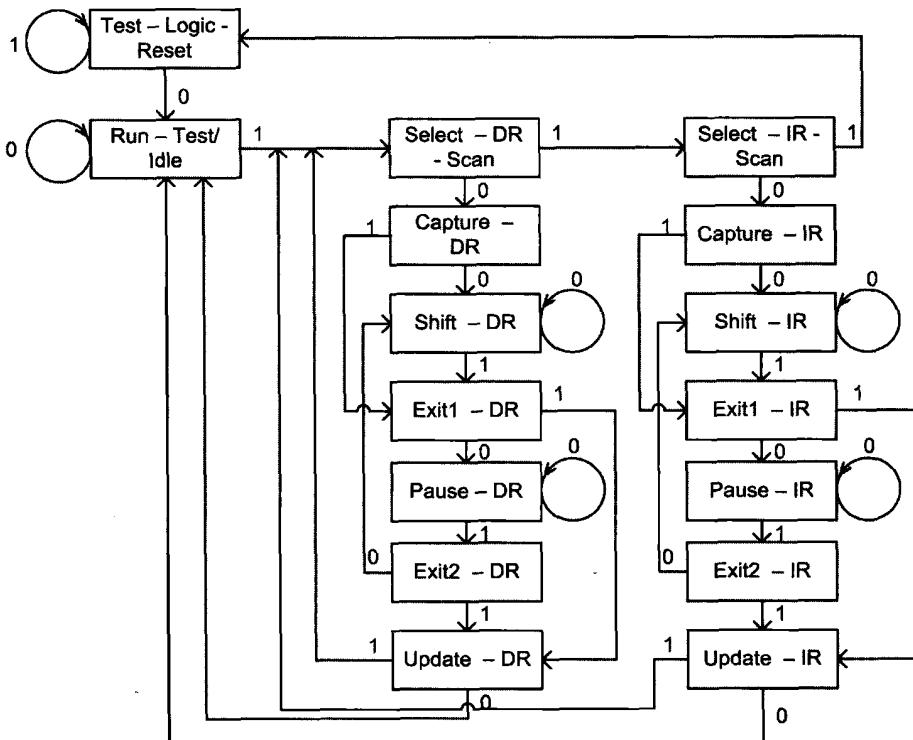


图 7-3 TAP 控制器的有限状态机模型

7.2.5 TAP 指令

TAP 指令用来通知 TAP 控制器应该执行何种操作。JTAG 标准定义了 9 条指令，其中有 3 条是必须要实现的，其他 6 条是可选的。芯片设计者也可以定义其他指令。JTAG 标准没有指定指令的长度，芯片设计者可以根据具体情况制定。

BYPASS 指令（必须）：选中旁路寄存器，使其将 TDI 和 TDO 连接起来形成通路，从而可以让测试数据没有影响地流过该芯片。JTAG 标准规定该指令代码的所有位必须为 1。

SAMPLE/PRELOAD 指令（必须）：选中边界扫描寄存器，使其与 TDI 和 TDO 形成通路，然后利用数据扫描操作访问边界扫描寄存器：将各个边界扫描单元中的内容送出去，实现对芯片的输入输出信号进行采样的目的，或者向边界扫描单元预先装载（preload）数据，供 EXTEST 指令使用。

EXTEST 指令（必须）：使芯片进入外部测试模式（external boundary-test mode），同时选中边界扫描寄存器使其与 TDI 和 TDO 形成通路。在该模式下，输入信号处的

边界扫描单元会记录下输入信号的值；输出信号处的边界扫描单元会将测试数据向外输出给外部的其他器件（可以是不支持 JTAG 的）。JTAG 标准规定该指令的代码所有位必须为 0。

INTEST 指令（可选）：使芯片进入内部测试模式（internal boundary-test mode），同时选中边界扫描寄存器使其与 TDI 和 TDO 形成通路。在该模式下，输入信号处的边界扫描单元会将自己的值送给芯片；输出信号处的边界扫描单元会记录下芯片的输出信号。

RUNBIST 指令（可选）：使芯片进入自检模式（built-in self-test mode），同时选中用于自检的用户定义的（user-specified）数据寄存器使其与 TDI 和 TDO 形成通路。在该模式下，边界扫描单元使芯片处于孤立的状态：不受输入的影响；其输出也不影响其他部件。

CLAMP 指令（可选）：将输出信号处的边界扫描单元的值输出给外部电路。在加载这个指令之前，可以使用 SAMPLE/PRELOAD 事先将边界扫描单元的值准备好。该指令会选中旁路寄存器，使其与 TDI 和 TDO 形成通路。

HIGHZ 指令（可选）：使芯片的所有输出信号（包括 two-state 和 three-state 信号）进入禁止（高危，high-impedance）状态。该指令会选中旁路寄存器，使其与 TDI 和 TDO 形成通路。

IDCODE 指令（可选）：使芯片保持正常工作的同时，选中设备 ID 寄存器使其与 TDI 和 TDO 连接形成通路，以便通过数据扫描操作将设备 ID 寄存器的内容输出给调试器。

USERCODE 指令（可选）：与 IDCODE 类似用来读取附加的设备信息。

本节简要介绍了 JTAG 技术的基本原理，其细节超出了本书的范围，感兴趣的读者可以阅读参考文献中列出的资料。

7.3 JTAG 应用

JTAG 标准一推出便得到了广泛的认可和支持，并被应用到众多领域中。以下列出的只是一小部分的典型应用。

设计验证和调试。例如对芯片进行在线仿真（ICE）调试；访问芯片的自检功能（使芯片开始自检，然后读取自检结果）；通过边界扫描采集信号，进行芯片、电路板和系统测试；下载和上传程序或数据（比如刷新 EEPROM 或 FLASH，读写内存）等。关于调试方面的应用，本节后文会详细论述。

生产测试。利用 JTAG 的串行化能力，进行大规模的自动化测试。比如可以把要

测试的部件通过一个简单的总线串联起来，在系统控制台发布测试指令，测试结果通过 JTAG 信号传回控制台以供分析。

系统配置和维护。对于数据中心、电站等单位，可以利用 JTAG 标准采集或发送数据，以配置各种不同的设备。

7.3.1 JTAG 调试

典型的 JTAG 调试环境由 3 大部分组成：被调试系统、调试机和将二者联系起来的 JTAG 工具（如图 7-4 所示）。被调试系统，又叫下位机，通常是正处于开发过程中的一套组件，包括主板、CPU 等部件。调试机，又叫主机（host）或上位机，可以是普通的台式机、笔记本电脑等。

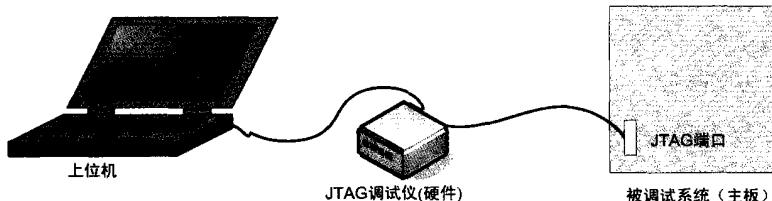


图 7-4 JTAG 调试示意图

JTAG 工具通常由以下几部分组成。

下位机接口：即与被调试系统上的调试端口（debug port）进行连接的物理接口和电缆。对于不同厂商的 CPU，其调试端口的信号数量和物理形式有所不同，因此也就要求 JTAG 工具需要有不同的接口与之连接。但因为调试端口包含的信号中被真正使用的主要有 5 路 JTAG 信号，因此很多 JTAG 工具厂商都设计了各种各样的转接口，以便支持更多的目标系统。本节下文会详细介绍与包含 IA-32 处理器的被调试系统相连接的各种方式。

缓存和控制仪：JTAG 工具的核心部件，通常是一个小盒子，用于通过 JTAG 协议与目标系统的被调试芯片通信。它的面板上有一些基本的显示和控制界面（开关等），内部主要包含有接口电路、控制电路和用于缓存（buffer）数据的存储器。这个小盒子的名称有几种叫法，如 JTAG 调试器、JTAG 仿真器等。我们将其称为 JTAG 调试仪。

上位机接口：与调试机（debugger）连接的物理接口和电缆。通常有几种形式，一种是通过专用的 PCI 插卡插入调试机的 PCI 插槽中。另一种是使用 USB 这样的通用接口，只要插入调试机的空闲 USB 口就可以了。

调试软件：供调试人员调试目标系统的软件环境，通常是一个增强的软件调试器，除了具备普通软件调试器的各种调试功能外，还具有增强的设置 JTAG 端口，设置硬件断点，察看目标系统的底层信息等各种功能。

基于 JTAG 标准的调试工具有很多，其名称也各不相同，如 XX 仿真器、XXICE、XX 调试器等。尽管有些文档将 ICE 调试和 JTAG 调试混在一起使用，但是我们应该清楚这两个术语本质上是有差异的。ICE 的内涵就是指用调试工具“冒充”被调试系统中的某一部件（常常是微处理器）以实现观察和调试目标系统的目的。可以将 ICE 理解为是一种调试思想，有很多种具体手段来实现这种思想。而 JTAG 调试是指基于 JTAG 标准（边界扫描技术和 TAP）的一种实现途径，它支持包括 ICE 在内的各种调试功能。总之，ICE 调试和 JTAG 调试是两种不同角度的命名方法，它们之间既有联系，又有区别。

7.3.2 调试端口 (Debug Port)

调试端口是指被调试系统上用来与调试工具连接的物理接口或接头 (header)。通常位于电路板上的一个方形接口 (接头)。以针对 IA-32 处理器设计的主板为例，调试端口主要有 ITP (30 针)、ITP700 (25 针) 和 XDP (60 pin) 3 种 (下一节将详细介绍)。

尽管大多数主板上都留有调试端口所需的电路，在开发阶段的样板中也配有调试端口，但是在大批量生产时往往不再焊接调试接头。那么，如果要再调试这样的系统怎么办呢？一种常见的方法就是使用如图 7-5 所示的转接头 (interposer)。将该接头插到目标主板的插槽上，然后再将 CPU 插在该接头上，该接口带有调试端口。对于不同的 CPU 插槽，应该使用不同的转接头。

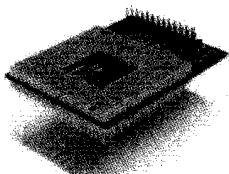


图 7-5 带有调试端口的 CPU 转接头 (interposer)

7.4 IA-32 处理器的 JTAG 支持

奔腾处理器是 IA-32 处理器中最早实现 JTAG 支持的处理器。其后的 IA-32 处理器也都包含了这一支持。下面先以 P6 系列处理器为例介绍 IA-32 处理器的 JTAG 支持，然后介绍探测模式和 ITP/XDP 端口。

7.4.1 P6 处理器的 JTAG 实现

首先，P6 处理器的管脚中包含了 JTAG 标准定义的所有 5 种信号，即测试时钟信号 TCK、测试模式选择信号 TMS、测试数据输入信号 TDI、测试复位信号 TRST# 和测试数据输出信号 TDO。

P6 处理器实现了 7 条 TAP 指令，包括 3 条 JTAG 标准规定必须实现的指令和 4 条可选指令。P6 处理器的 TAP 指令的长度为 6 个比特位，其操作码定义如表 7-1 所示。

表 7-1 P6 处理器的 TAP 指令（选自《P6 系列处理器硬件开发手册》）

TAP 指令	操作码	处理器管脚输入来源	选中的 TAP 数据寄存器	状态动作			
				Run-Test/Idle	Capture-DR	Shift-DR	Update-DR
EXTEST	000000	边界扫描单元	边界扫描寄存器	—	对所有处理器的管脚采样	移位数据寄存器	更新数据寄存器
SAMPLE/PRELOAD	000001	—	边界扫描寄存器	—	对所有处理器的管脚采样	移位数据寄存器	更新数据寄存器
IDCODE	000010	—	设备 ID 寄存器	—	加载处理器的唯一标识代码	移位数据寄存器	—
CLAMP	000100	边界扫描单元	旁路寄存器	—	复位旁路寄存器	移位数据寄存器	—
RUNBIST	000111	边界扫描单元	BIST 结果寄存器	开始自检测试(BIST)	捕捉自检结果	移位数据寄存器	—
HIGHZ	001000	悬置(floated)	旁路寄存器	—	复位旁路寄存器	移位数据寄存器	—
BYPASS	111111	—	旁路寄存器	—	复位旁路寄存器	移位数据寄存器	—

表 7-1 的第 3 列是相应指令执行期间处理器核心的输入来源，第 4 列是指令要选中的数据寄存器，第 5~8 列是指在状态机的 Run-Test/Idle、Capture-DR、Shift-DR 和 Update-DR 状态时 TAP 控制器所采取的动作。以 RUNBIST 指令为例，其指令编码（操作码）为 000111，在此指令执行期间，处理器的输入是由边界扫描单元驱动的，RUNBIST 指令会选中 BIST 结果寄存器，使其与 TDI 和 TDO 形成通路，在 Run-Test/Idle 状态，TAP 控制器执行的操作是开始内建的自检测试(Built-In Self Test)，在 Capture-DR 状态，会将自检结果捕捉到 BIST 结果寄存器，在 Shift-DR 状态，会对 BIST 结果寄存器进行移位操作。

表 7-2 列出了 P6 处理器的所有 TAP 寄存器，每个寄存器的长度和选中该寄存器的 TAP 指令。

边界扫描寄存器的长度是 159 个比特位，这与 P6 处理器的输入输出信号的数量相吻合，也就是每个输入输出信号都配备有一个边界扫描单元，用以控制和记录该信号。

设备 ID 寄存器的长度是 32 位，被划分成几个部分用以表示版本型号等信息，详细定义列在表 7-3 中。

表 7-2 P6 处理器的 TAP 数据寄存器

TAP 数据寄存器	长度	哪些指令会选中该寄存器
旁路寄存器	1	BYPASS、HIGHZ、CLAMP
设备 ID 寄存器	32	INCODE
BIST 结果寄存器	1	RUNBIST
边界扫描寄存器	159	EXTEST、SAMPLE/PRELOAD

表 7-3 P6 处理器的设备 ID 寄存器

版本	产品编号					厂商 ID	1	整个代码
	Vec	产品类型	Generation (第几代)	型号				
长度	4	1	6	4	5	11	1	32
二进制	Xxxx	0	00000 1	0110	0001 1	0001 1	1	xxxx300000101100 0011000000010011
十六进制	X	0	01	6	03	09	1	x02c3013

7.4.2 探测模式

下面介绍 IA-32 处理器的探测模式 (Probe Mode)。探测模式是专门用于调试的一种处理器内部模式。与实模式和保护模式这些软件可以控制的操作模式不同，只有通过专门的管脚信号和边界扫描寄存器才可以访问探测模式。在探测模式下，CPU 中断正常的操作，从外部看就好像处于休眠的状态，但通过调试工具可以与其通信，分析并修改系统的状态，包括内存、I/O 空间和各种寄存器。以奔腾处理器为例，调试工具可以通过将 R/S#管脚信号置低要求处理器进入探测模式。当处理器检测到该信号后，在完成正在执行的指令后，停止执行下一条指令 (freeze on next instruction boundary)，并设置 PRDY (Probe Ready) 信号。调试工具发现 PRDY 信号后，便可以通过 JTAG 协议向目标处理器发送各种调试命令和数据了。比如观察内存、设置断点等。在调试工具完成设置后，可以通过解除 R/S#信号 (恢复其高电平) 使处理器退出探测模式恢复执行原来的程序。

从实现角度来看，探测模式是通过扩展 JTAG 标准实现的。包括增加 TAP 寄存器、TAP 指令和管脚信号。

可能是为了避免混淆，英特尔的各种文档没有系统的介绍探测模式。通常是用进入“空闲状态”等词语一带而过。但从调试的角度分析，CPU 此时是在另一种模式下工作着的。

7.4.3 ITP 和 XDP

为了更好地调试基于 IA-32 处理器设计的主板和系统，英特尔定义了 3 种调试端

口用以和集成在处理器中的调试功能通信。这 3 种端口是：ITP（30 针）、ITP700（25 针）和 XDP（60 pin）。

ITP 是 In-Target Probe 的缩写，意思是目标内探测器。下面我们详细介绍应用最广泛的 ITP700 接口。

ITP700 是一个 25 针（pin）的接口（如图 7-6 所示），主要包含 3 类信号：JTAG 信号、系统信号和执行信号。JTAG 信号即 IEEE 1149.1 中定义的标准信号。系统信号用来指示整个系统的状态。执行信号用来控制目标处理器的执行和复位。

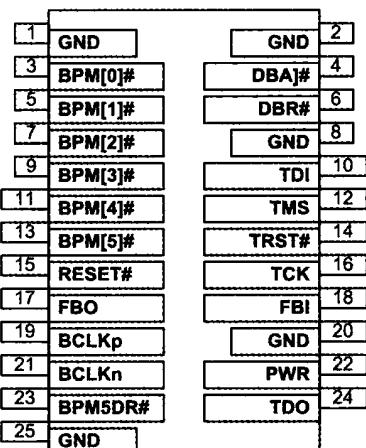


图 7-6 ITP700 端口示意图

表 7-4 列出并解释了 ITP700 端口的各个信号，其中第 2 列给出的输入和输出方向是相对于 ITP 工具的。

表 7-4 ITP 信号说明

信号	编号	输入/输出	描述
PWR	22	输入	目标系统的电源。ITP 工具可以用来：1) 判断目标系统的电源是否稳定；2) 用作产生 BPM[5:0]# 和 RESET# 信号时的参考电压；3) 与 DBA# 信号一起用于仲裁（arbitrate）边界扫描链的使用权
BCLK (p/n)	19/21	输入	目标系统的差分驱动总线信号。用于采样执行信号等
DBA# (Debug Port Active)	4	输出	指示调试端口是否活动（active）。ITP 使用该信号表示它正在使用目标系统的 TAP 接口
DBR# (Debug Port Reset)	6	输出	复位调试端口。ITP 可以使用该信号来复位目标系统
FBI	18	输出	TAP 主时钟（master clock）的另一个来源。对于大多数 ITP 实现，该信号都是可选的。因为通常都使用 TCK 作为 TAP 的主时钟

续表

信号	编号	输入/输出	描述
FBO	17	输入	是 TCK 的回馈。
TCK	16	输出	TAP 主时钟的标准来源。如果 FBI 被用为主时钟信号，那么 ITP 工具可以不提供 TCK。
TDI	10	输出	TAP 的数据输入信号。ITP 工具通过该信号向目标系统发送数据。
TDO	24	输入	TAP 的数据输出信号。ITP 工具通过该信号从目标系统接收数据。
TMS	12	输出	TAP 状态管理信号。
TRST# (Test Reset)	14	输出	测试逻辑复位信号。
BPM[5:0]#	13、11、 9、7、5、 3	输入	来自目标系统的断点信号。BPM[3:0]#用来表示对应的调试寄存器断点被触发（或对应的性能计数器溢出，依赖于 DebugCtl 寄存器）。 BPM4#相当于奔腾处理器的 PRDY 信号，目标 CPU 用以回应 ITP 工具的探测请求（Probe Request），通知 ITP 工具已经进入探测模式，可以接受各种调试命令了。 BPM5#应该与 BPM5DR 短接，因此没有独立功能。
RESET#	15	输入	来自目标系统的复位信号。
BPM5DR#	23	输出	相当于 PREQ（Probe Request）信号，ITP 工具用之请求对目标处理器的控制权。

参考文献 5 更详细地介绍了 ITP700 端口的细节，以及设计中应该注意的问题。

7.4.4 XDP

XDP 是 eXtended Debug Port 的缩写，意思是扩展的调试端口。ITP 所包含的信号是 XDP 的一个子集，因此可以通过一个转接头将 XDP 端口转为 ITP。

简单来说，XDP 是一个如图 7-7 所示的接头，共有 60 针，除了 JTAG 标准中定义的信号外，还包括很多实现扩展功能的信号。与 ITP 相比，尽管 XDP 的信号数量更多，但是由于设计不同，它占用的主板空间更小。



图 7-7 XDP 端口

市场上销售的大多数主板并没有安装 XDP 端口。但是在其背面（不安装零件的一面）通常留有一组焊点，可以焊接上称为 XDP-SSA（Second Side Attach）的调试端口。

XDP-SSA 共有 31 个信号，是 XDP 的一个子集，因此不再具有 XDP 的某些增强功能。通常处于开发阶段的主板才带有完全的 XDP 调试端口。

参考文献 6 中详细介绍了 XDP 接口的所有信号及功能。

7.4.5 典型应用

正如本章开头所说的，硬件调试工具通常用于软件调试器无法解决的问题。以下是使用 ITP 调试器的一些典型场景。

- 调试系统固件代码，包括 BIOS 代码、EFI 代码，以及支持 AMT 技术的 ME（Management Engine）代码。
- 调试操作系统加载程序，以及系统临界状态的问题，比如进入睡眠和从睡眠状态恢复过程中发生的问题。
- 软件调试器无法调试的其他情况，比如开发软件调试器时调试实现调试功能的代码（例如 Windows 的内核调试引擎），以及调试操作系统内核的中断处理函数、任务调度函数等。

观察 CPU 的微观状态，比如 CPU 的 ACPI 状态（C State）。

7.5 本章总结

本章介绍了硬件调试工具广泛使用的 JTAG 标准，以及它在 IA 架构中的应用。使用硬件调试工具的优点是依赖少，可控性强，因此非常适合调试系统软件、敏感的中断处理代码，以及与硬件有关的问题。另外，因为不需要在目标系统中运行任何调试引擎或其他软件，所以使用硬件调试工具比使用软件工具对被调试软件的影响更小，也就是具有更小的海森伯效应（将在 28.6 节详细介绍）。

硬件调试工具的局限性是价格比较昂贵，要求目标系统具有调试接口（如 ITP 或 XDP），连接和设置也相对复杂些。

本章是这一篇的最后一章。总体而言，本篇主要以 IA-32 处理器为例介绍了 CPU 对软件调试的支持，下面从“需求”的角度再对这些功能概括如表 7-5 所示：

表 7-5 调试需求和 CPU 的支持

调试需求	CPU 的支持	章节
执行到指定地址处的指令时中断到调试器	指令访问断点	4.2
执行完每一条指令后都中断到调试器	单步执行标志（陷阱标志）	4.3
执行完当前分支后中断到调试器	按分支单步执行（陷阱标志）	4.3, 5.3
访问指定内存地址的内存数据（读写内存）时中断到调试器	数据访问断点（硬件断点）	4.2

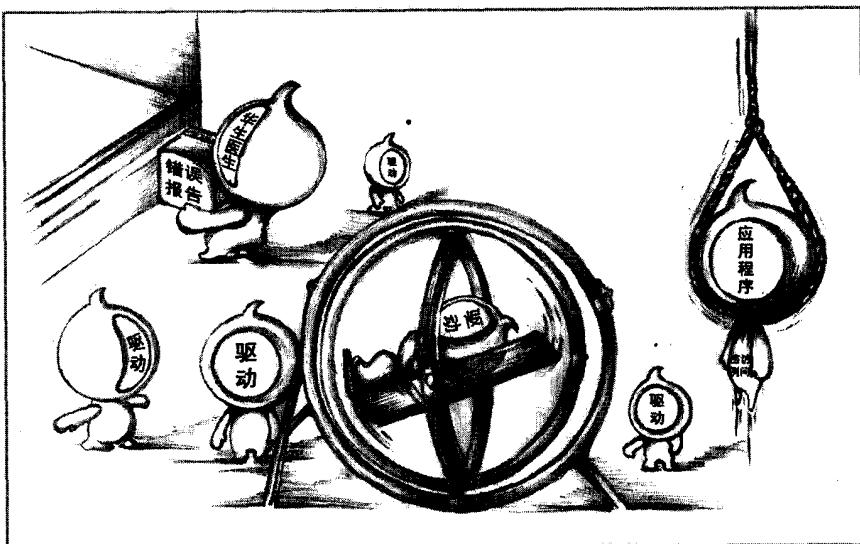
续表

调试需求	CPU 的支持	章节
访问指定 I/O 地址的 I/O 数据（输入输出）时中断到调试器	I/O 访问断点（硬件断点）	4.2
遇到该指令就中断到调试器	断点指令（软件断点）	4.1
切换到指定的任务就中断到调试器	TSS 中的 T 标志	4.3
记录软件/CPU 的执行轨迹	分支记录机制	5.2, 5.3
监视 CPU 和软件的执行效率	性能监视	5.5
记录下 CPU 遇到的硬件错误	MCA	6.2
调试 CPU 本身的问题，或以上手段都难以解决的其他调试任务	JTAG 支持	7.4

参考文献

1. Dual-Core Intel® Xeon® Processor LV and Intel® 3100 Chipset Development Kit User's Manual. Intel Corporation
2. IEEE Std 1149.1(JTAG) Testability Primer. Texas Instrument
3. P6 Family Processors Hardware Developer's Manual. Intel Corporation
4. Robert R. Collins. Overview of Pentium Probe Mode.
<http://www.x86.org/articles/probemd/probemode.htm>
5. ITP700 Debug Port Design Guide. Intel Corporation
<http://www.intel.com/design/xeon/guides/24967914.pdf>
6. Debug Port Design Guide for UP/DP Systems. Intel Corporation
<http://download.intel.com/design/Pentium4/guides/31337301.pdf>

操作系统的调试支持



操作系统（Operating System，简称 OS）是计算机系统中的基本软件，它负责统一管理系统中的软硬件资源，为系统中运行的应用软件（applications）提供服务，是应用软件运行的基础。操作系统所提供的服务因操作系统的 design 目的和使用环境不同而有所差异，但通常都包括文件管理、内存管理、进程管理、打印管理、网络管理等基本功能。除了这些功能外，如何支持调试也是操作系统设计的一项根本任务，从被调试对象角度来看，可以把操作系统的调试支持分为以下 3 个方面。

第一，对应用程序调试（Application Debugging）的支持，即如何简单高效地调试运行在系统中的各种应用程序。应用程序通常在操作系统分配的较低优先级下运行，其代码属于操作系统不信赖的代码。

第二，对设备驱动程序调试（Device Driver Debugging）的支持，设备驱动程序或其他运行在内核模式的模块是操作系统的可信赖代码，通常与操作系统运行在同一个优先级下和同一个地址空间中，因此调试这些模块通常与调试应用程序有很大不同。

第三，对操作系统自身调试的支持，即如何调试操作系统的各个组成部分。例如调试正在开发过程中的系统模块，以及定位产品发布后出现的系统故障。

调试器（Debugger）是软件调试的最重要工具，使用调试器调试是解决复杂软件问题的首选途径。但是在某些情况下（比如产品发布后发生在用户环境中的问题），并不具备使用调试器调试的条件，这时就必须考虑如何在没有调试器的情况下进行调试。从这个意义上讲，对于以上每类调试对象（任务），还必须考虑两种情况。

第一，使用调试器的调试，即通过有效的模型和系统机制来支持调试器软件操纵和访问被调试对象。

第二，不使用调试器的调试，即通过操作系统的基础服务，支持软件实现各种不依赖于调试器的调试途径，比如错误提示、事件追踪、日志和错误报告等。

综合以上分析，可以把操作系统的调试支持归纳为如下表所示的 6 个问题。

	使用调试器的调试	不使用调试器的调试
调试应用程序	如何使系统中的应用程序可以被调试器调试？（问题 A）	如何使系统中的应用程序在没有调试器时也具有很好的可调试性？（问题 B）
调试驱动程序	如何使系统中的驱动程序可以被调试器调试？（问题 C）	如何使系统中的驱动程序在没有调试器时也具有很好的可调试性？（问题 D）
调试操作系统自身	如何使操作系统自身的代码可以被调试器调试？（问题 E）	如何使系统本身在没有调试器时也具有很好的可调试性？（问题 F）

本篇将以 Windows 操作系统为例详细探讨操作系统对软件调试的支持。第 8 章介绍 Windows 操作系统的概况和关键概念，为理解后面章节的内容打基础。此后的内容可以分为 5 块。

第 9 章和第 10 章将着重讨论问题 A，包括支持应用程序调试的用户态调试子系统，调试会话的建立过程及调试事件的产生和分发机制等。

与第 2 章的 CPU 异常相呼应，第 11 章和第 12 章将从操作系统的层面分析异常的分发和处理机制及系统处置未处理异常的方法。这一块内容的主题是异常处理，它与所有 6 个问题都相关。

第 13~17 章将把视线转向调试器之外的辅助调试机制（问题 B、D 和 F）。第 13 章将分析 Windows 系统提供的错误提示机制。第 14 章会介绍 Windows XP 系统引入的错误报告机制（WER）。第 15 章会分析 Windows 的事件日志（Event Log）机制。第 16 章将分析 Windows 事件追踪机制（ETW）。

内核调试对于解决系统级的问题和学习操作系统有着非常重要的意义。因此，第 18 章我们将深入介绍 Windows 的内核调试引擎。

第 19 章将介绍用于提高测试和调试效率的验证机制（Verifier），包括应用程序验证和驱动程序验证。

Windows 概要

本章我们将介绍 Windows 操作系统的基本知识和概念，包括简要历史（8.1 节）、访问模式（8.2 节）、进程和线程（8.3 节），以及 Windows 的架构和核心部件（8.4 节）。介绍这些内容的主要目的是加深大家对 Windows 操作系统的理解，以便更好地阅读本书的其他内容。

8.1 简介

Windows 是微软公司开发的图形界面（GUI）操作系统，第一个版本（Windows 1.0）于 1985 年 11 月正式发布，但其功能非常有限，销量并不好。1987 年的 2.0 版做了一些改进，但仍有待完善。直到 1990 年 Windows 3.0 版的推出，图形界面有了重大改进，内存管理能力也大大增强，更多的应用程序可以在其上运行，从此 Windows 才走上了成功之路。在此后的十几年中，Windows 不断发展，推出了很多个版本，比如 Windows 98、Windows Me、Windows 2000、Windows XP、Windows Server 2003、Windows Vista 等。

根据各个 Windows 版本所基于的不同内核，可以把它们分为 3 大类。

第一类，16 位的 Windows，包括 Windows 1.0（1985 年发布）、2.0（1987 年发布，能够访问 1MB 内存）、3.0（1990 年 5 月 22 日发布，支持 16 位的保护模式，可以访问 16MB 内存）、3.1（1992 年 4 月发布，只可以运行在保护模式下）和 3.2（中文版本）等。16 位的 Windows 源于 DOS 操作系统，目前该版本已经很少有人使用。

第二类，Windows 9x，包括 Windows 95（1995 年 8 月发布）、Windows 98（1998 年 6 月发布），和 Windows Me（Millennium Edition）（2000 年 9 月发布）。Windows 9x 是从 16 位的 Windows 发展而来的，这个产品线在推出 Windows Me 后便不再开发了。

第三类，Windows NT（New Technology）系列，包括 Windows NT 3.1（1993 年 7 月发布）、Windows NT 4.0（1996 年 7 月）、Windows 2000（2000 年 2 月发布）、Windows XP（2001 年 10 月发布）、Windows Server 2003、Windows Vista 等，这些版本都源于 Windows NT 内核。

与针对个人计算机（PC 机）市场的 16 位 Windows 和 Windows 9x 不同，Windows

NT在设计之初就是针对硬件配置较高的办公环境和服务器市场的。因此，Windows 9x 和 Windows NT 这两个产品线虽然相互借鉴，在 API 一级也保持着很好的兼容性，但是由于其内核的根本差异，二者间在很多方面还是不同的，比如设备驱动程序就是不兼容的，也就是说，对于同一个硬件，Windows 9x 使用的驱动程序与 Windows NT 使用的驱动程序是不能相互替代的。

从针对的使用模式和用户角度看，可以把 Windows 分为终端版本和服务器版本，前者用于笔记本或台式电脑上，满足办公和家庭用户的需要，后者用于服务器系统。图 8-1 画出了 Windows 的终端和服务器版本的发展例程。

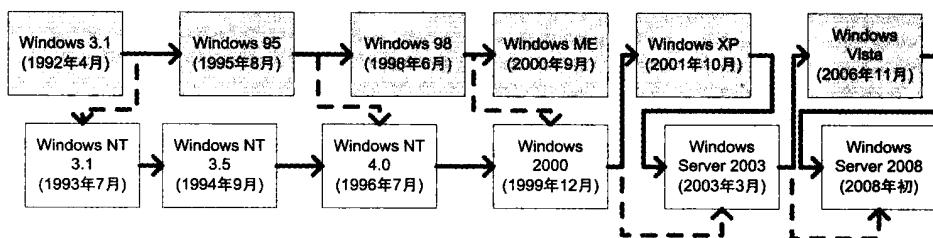


图 8-1 Windows 产品线示意图

图 8-1 中的实线箭头代表直接的版本演进，箭头指向的版本是基于前一版本的代码（Code Base）而开发的。虚线箭头代表借鉴和影响关系。Windows 2000 第一次将桌面版本和服务器版本统一起来，尽管 Windows 2000 也有服务器版本和个人用户版本之分，但是二者操作系统内核是相同的。不过，像 Windows 2000 那样用一个产品线来覆盖台式机和服务器这两大市场的模式很快就遇到了困难，因为服务器版本强调安全性，台式机版本强调易用性，而安全和易用很多时候是难以两全的。因此在 Windows 2000 之后，又分成了 Windows XP 和 Windows Server 两个产品线，Windows XP 定位于台式机用户，而 Windows Server（如 Windows Server 2003）是针对服务器市场的。Windows XP 除了有家庭版、专业版这些针对普通台式机和笔记本电脑的版本外，还有用于 Tablet PC 的版本和针对嵌入式系统的版本（称为 Windows XP Embedded）。

表 8-1 列出了源于 NT 内核的各个 Windows 版本的推出时间和内部版本号。

表 8-1 源于 NT 内核的 Windows 操作系统

产品名称	内部版本号	发布日期	备注
Windows NT 3.1	3.1	1993 年 7 月	
Windows NT 3.5	3.5	1994 年 9 月	
Windows NT 3.51	3.51	1995 年 5 月	
Windows NT 4.0	4.0	1996 年 7 月	
Windows 2000	5.0	1999 年 12 月	分为台式机版本（Windows 2000 Professional）和服务器版本
Windows XP	5.1	2001 年 8 月	
Windows Server 2003	5.2	2003 年 3 月	
Windows Vista	6.0	2006 年 11 月	
Windows Server 2008		2008 年 2 月 4 日	

考虑到 16 位的 Windows 和 Windows 9x 都已经过时，所以本书主要讨论的是表 8-1 列出的源于 NT 内核的各个 Windows 版本。如无特别说明，我们介绍的内容都是针对这些版本的。为了行文方便，我们大多时候就使用 Windows 来指代这些版本。

8.2 进程和进程空间

Windows 是个典型的多任务操作系统，它允许有多个程序同时在系统中运行，只要打开任务管理器（Task Manager），就可以看到当前正在运行着的程序，这些处于运行状态的程序（programs）通常又被称为进程（process）。

8.2.1 进程和线程

进程和程序的关系好比是类和实例的关系，当我们运行一个程序时，操作系统便会为这个程序创建一个实例。一个类可以有多个实例，一个程序也可以有多个实例在运行。比如我们可以启动记事本程序多次，使用不同的实例操作不同的文件。有些程序在启动时会检测是否已经有自己的实例在运行，如果有，新的实例就立刻退出，以保证只有一个实例在工作。

从操作系统的角度来讲，每个进程又被称为一个任务（task），Windows 任务管理器的名称即由此而来。但值得说明的是，因为每个 Windows 进程可以包含一个到多个线程，每个线程又都是可以被调度执行的，所以 Windows 系统中的任务与 CPU 一级的任务是有所不同的。CPU 一级的任务相当于 Windows 系统下每个进程中的一条线程。操作系统中的任务是指系统中运行着的各个进程，从这个意义上讲，Windows 操作系统的每个任务对应于 CPU 的一个或多个任务。

8.2.2 进程空间

为了保证系统中每个任务或进程的安全，Windows 为不同的进程分配了独立的进程空间（process space）。进程空间是操作系统分配给每个进程的虚拟内存空间（virtual address space），每个进程运行在这个受操作系统保护的空间之中，它的程序指针所代表的是本进程空间中的一个虚拟地址，根本无法指到另一个进程空间中的数据，这样便保证了一个进程的数据和代码不会轻易受到其他进程的侵害，一个进程内的错误也不会波及同一系统内运行着的其他进程。或者说每个进程都在操作系统分配给它的虚拟空间中运行，它无法直接访问其他进程的空间，也不必担心自己的空间会被其他进程所侵占。

对于不同的硬件平台，进程空间的大小也有所不同。对于 32 位的 x86 系统，每个进程的进程空间是 4GB，即地址 0x00000000 到地址 0xFFFFFFFF。为了高效地调用和执行操作系统的各种服务，Windows 会把操作系统的内核数据和代码映射到系统中所有进程的进程空间中。因此，4GB 的进程空间总是被划分为两个区域：用户空间和系

统空间。用户空间和系统空间的默认大小各为 2GB，低 2 GB 为用户空间，高 2GB 为系统空间。Windows 2000 Advanced Server、Windows 2000 Datacenter Server、Windows XP 和 Windows Server 2003 支持/3GB 启动选项使用户空间为 3GB，以便满足数据库系统等某些特殊应用程序的需要。要使用该功能，除了要在启动配置文件（boot.ini）中设置/3GB 选项外，还需要在可执行映像的头信息中设置大用户空间标志（IMAGE_FILE_LARGE_ADDRESS_AWARE flag）。Windows XP 和 Windows Server 2003 还支持 /USERVA 选项，该选项可以设定一个 2GB~3GB 之间（以 MB 为单位）的值用于定义用户空间的大小。由于对于大多数 32 位系统和大多数进程，用户空间都是 2GB 大小，所以如不特别指出，本书讨论的都是用户空间为 2GB 的情况，也就是地址 0x00000000 到地址 0x7FFFFFFF 为用户空间，地址 0x80000000 到地址 0xFFFFFFFF 为系统空间，如图 8-2 所示。

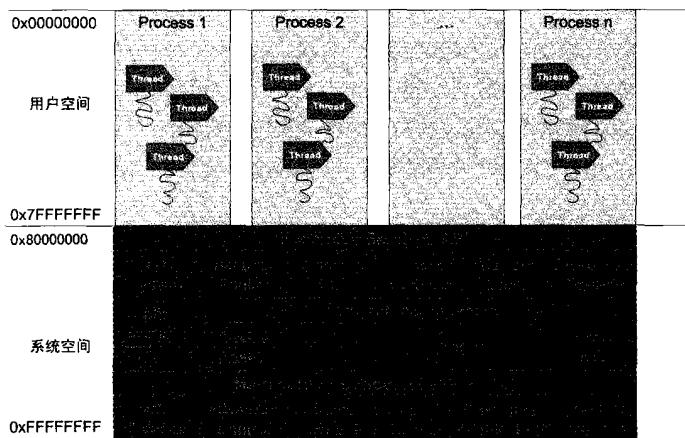


图 8-2 虚拟地址空间布局（用户和系统空间各为 2GB）

64 位系统下，用户空间和系统空间都增大了很多。比如在 x64 系统中，用户空间的范围是 0x0~0x07FFF FFFFFFFF，大小为 8192GB（8TB），系统空间的范围是 0xFFFF8000`00000000~0xFFFFFFFF FFFFFFFF，大小为 8TB。在 IA-64（安腾）平台上，用户空间为 7152GB（7TB），系统空间为 6144GB。考虑到 64 位系统下的大多数数据都是 64 位的，不方便表达和排版，所以除非特别说明，本书讨论的都是 32 位的情况。

8.2.3 进程资源

除了虚拟地址空间，每个 Windows 进程还拥有。

- 一个全局唯一的进程 ID（又称用户 ID，Client ID），简称为 PID。
- 一个可执行映像（image），也就是该进程的程序文件（可执行文件）在内存中的表示。
- 一个或多个线程。

- 一个位于内核空间中的名为 EPROCESS (executive process block, 即进程执行块) 的数据结构, 用以记录该进程的关键信息, 包括进程的创建时间、映像文件名称等, 详见下文。
- 一个位于内核空间中的对象句柄表, 用以记录和索引该进程所创建/打开的内核对象。操作系统根据该表格将用户模式下的句柄翻译为指向内核对象的指针。
- 一个用于描述内存目录表起始位置的基地址, 简称页目录地址 (DirBase), 当 CPU 切换到该进程/任务时, 会将该地址加载到 CR3 寄存器, 这样当前进程的虚拟地址才会被翻译为正确的物理地址 (参见 2.7 节)。
- 一个位于用户空间中的进程环境块 (Process Environment Block, 简称 PEB), 详见下文。
- 一个访问权限令牌 (access token), 用于表示该进程的用户、安全组, 以及优先级别。

为了更好地理解以上列出的每个项目, 下面我们将结合 WinDBG 的进程观察命令 (!process) 来介绍。这样既易于理解, 又可以帮助大家熟悉调试器的用法。

首先启动记事本程序 (notepad.exe), 我们将把这个典型的 Windows 程序作为被观察对象。然后启动 WinDBG, 并开始本地内核调试 (File > Kernel Debug, 然后选择 Local)。在内核调试会话建立后, 执行 !process 0 0 命令, 列出系统内的所有进程。

```
1kd> !process 0 0
***** NT ACTIVE PROCESS DUMP *****
...
PROCESS 86a7d030 SessionId: 0 Cid: 0f20 Peb: 7fffd000 ParentCid: 0d98
DirBase: 1f350000 ObjectTable: e1771668 HandleCount: 33.
Image: notepad.exe
...
```

!process 0 0 命令的第一个参数用来指定要显示的进程 ID, 0 代表所有进程。第二个参数用来指定要显示的进程属性, 0 代表基本的进程属性。

从 WinDBG 显示出的命令结果中可以找到关于 notepad.exe 的内容 (见上), 下面我们将以其为例来解说各个字段的含义。

8.2.4 EPROCESS 结构

PROCESS 后面的地址 (86a7d030) 指向的便是进程的 EPROCESS 结构, 很多时候 Windows 内核就是用这个指针来代表一个进程的。使用 dt 命令 (显示类型结构命令) 可以观察该结构的各个字段和取值 (//后是注释) (见清单 8-1)。

清单 8-1 EPROCESS 结构

1kd> dt _EPROCESS 86a7d030	
+0x000 Pcb	: _KPROCESS // 内核进程块, 用来记录任务调度有关的信息
+0x06c ProcessLock	: _EX_PUSH_LOCK
+0x070 CreateTime	: _LARGE_INTEGER 0x1c6aec5`7d9a68a0 // 创建时间

```

+0x078 ExitTime : _LARGE_INTEGER 0x0 // 退出时间
+0x080 RundownProtect : _EX_RUNDOWN_REF
+0x084 UniqueProcessId : 0x00000f20 // 进程 ID
+0x088 ActiveProcessLinks : _LIST_ENTRY [ 0x87a5f0b8 - 0x878b80b8 ]
+0x090 QuotaUsage : [3] 0xa50
+0x09c QuotaPeak : [3] 0xa50
+0x0a8 CommitCharge : 0x15a
+0x0ac PeakVirtualSize : 0x1e61000
+0x0b0 VirtualSize : 0x1cc5000 // 
+0x0b4 SessionProcessLinks : _LIST_ENTRY [ 0x87a5f0e4 - 0x861080e4 ]
+0x0bc DebugPort : (null) // 用户态调试端口, 参见下文
+0x0c0 ExceptionPort : 0xe27767a0 // 异常端口
+0x0c4 ObjectTable : 0xe1771668 _HANDLE_TABLE // 对象句柄表
+0x0c8 Token : _EX_FAST_REF // 访问令牌
+0x0cc WorkingSetLock : _FAST_MUTEX
+0x0ec WorkingSetPage : 0x653
+0x0f0 AddressCreationLock : _FAST_MUTEX
+0x110 HyperSpaceLock : 0
+0x114 ForkInProgress : (null)
+0x118 HardwareTrigger : 0
+0x11c VadRoot : 0x871126d0 // 虚拟地址描述符二叉树的根节点
+0x120 VadHint : 0x86dc1f78
+0x124 CloneRoot : (null)
+0x128 NumberOfPrivatePages : 0x110
+0x12c NumberOfLockedPages : 0
+0x130 Win32Process : 0xe13b8de8
+0x134 Job : (null)
+0x138 SectionObject : 0xe1d16900
+0x13c SectionBaseAddress : 0x01000000
+0x140 QuotaBlock : 0x8768c480 _EPROCESS_QUOTA_BLOCK
+0x144 WorkingSetWatch : (null)
+0x148 Win32WindowStation : 0x00000030
+0x14c InheritedFromUniqueProcessId: 0x00000d98
+0x150 LdtInformation : (null)
+0x154 VadFreeHint : (null)
+0x158 VdmObjects : (null)
+0x15c DeviceMap : 0xe27058d0
+0x160 PhysicalVadList : _LIST_ENTRY [ 0x86a7d190 - 0x86a7d190 ]
+0x168 PageDirectoryPte : _HARDWARE_PTE_X86
+0x168 Filler : 0
+0x170 Session : 0xf7c9b000 // 所属会话对象
+0x174 ImageFileName : [16] "notepad.exe"
+0x184 JobLinks : _LIST_ENTRY [ 0x0 - 0x0 ]
+0x18c LockedPagesList : (null)
+0x190 ThreadListHead : _LIST_ENTRY [ 0x861fe25c - 0x861fe25c ] // 线程列表
+0x198 SecurityPort : (null)
+0x19c PaeTop : (null)
+0x1a0 ActiveThreads : 1
+0x1a4 GrantedAccess : 0x1f0fff
+0x1a8 DefaultHardErrorProcessing: 1 // 参见 13.1 节
+0x1ac LastThreadExitStatus : 0
+0x1b0 Peb : 0x7ffd000 _PEB // 进程环境块
+0x1b4 PrefetchTrace : _EX_FAST_REF
+0x1b8 ReadOperationCount : _LARGE_INTEGER 0x1
+0x1c0 WriteOperationCount : _LARGE_INTEGER 0x0
+0x1c8 OtherOperationCount : _LARGE_INTEGER 0x9a
+0x1d0 ReadTransferCount : _LARGE_INTEGER 0x46da
+0x1d8 WriteTransferCount : _LARGE_INTEGER 0x0
+0x1e0 OtherTransferCount : _LARGE_INTEGER 0x1e6
+0x1e8 CommitChargeLimit : 0
+0x1ec CommitChargePeak : 0x23a

```

```

+0x1f0 AweInfo : (null)
+0x1f4 SeAuditProcessCreateInfo: _SE_AUDIT_PROCESS_CREATION_INFO
+0x1f8 Vm : _MMSUPPORT
+0x238 LastFaultCount : 0
+0x23c ModifiedPageCount : 0
+0x240 NumberOfVads : 0x42
+0x244 JobStatus : 0
+0x248 Flags : 0xd0840
+0x248 CreateReported : 0y0
+0x248 NoDebugInherit : 0y0
+0x248 ProcessExiting : 0y0 // 正在退出标志
+0x248 ProcessDelete : 0y0 // 删除标志
+0x248 Wow64SplitPages : 0y0
+0x248 VmDeleted : 0y0
+0x248 OutswapEnabled : 0y1
+0x248 Outswapped : 0y0
+0x248 ForkFailed : 0y0
+0x248 HasPhysicalVad : 0y0
+0x248 AddressSpaceInitialized : 0y10
+0x248 SetTimerResolution : 0y0
+0x248 BreakOnTermination : 0y0
+0x248 SessionCreationUnderway : 0y0
+0x248 WriteWatch : 0y0
+0x248 ProcessInSession : 0y1
+0x248 OverrideAddressSpace : 0y0
+0x248 HasAddressSpace : 0y1
+0x248 LaunchPrefetched : 0y1
+0x248 InjectInpageErrors : 0y0
+0x248 Unused : 0y000000000000 (0)
+0x24c ExitStatus : 0x103
+0x250 NextPageColor : 0x5504
+0x252 SubSystemMinorVersion: 0 ''
+0x253 SubSystemMajorVersion: 0x4 ''
+0x252 SubSystemVersion : 0x400 // 环境子系统版本号
+0x254 PriorityClass : 0x2 ''
+0x255 WorkingSetAcquiredUnsafe : 0 ''

```

从清单 8-1 中可以看到，EPROCESS 结构几乎包括了进程的所有关键信息和重要“资产”，比如与调试密切相关的 DebugPort 和 ExceptionPort（将在第 9 章详述），指向进程的虚拟地址描述符（VAD）二叉树的根节点的 VadRoot（使用!vad 命令可以列出这些描述符），以及指向进程内所有线程列表表头的 ThreadListHead，进程环境块地址等。我们将在下文和以后的章节中逐步介绍其中的重要字段。

也可以用!process 命令加上 EPROCESS 结构的地址来显示该进程的关键信息：

```

lkd> !process 86a7d030
PROCESS 86a7d030 SessionId: 0 Cid: 0f20 Peb: 7ffdf000 ParentCid: 0d98
DirBase: 1f350000 ObjectTable: e1771668 HandleCount: 33.
Image: notepad.exe
VadRoot 871126d0 Vads 66 Clone 0 Private 272. Modified 0. Locked 0.
DeviceMap e27058d0
Token e24d8510
Elapsed Time 00:10:35.754
User Time 00:00:00.010
Kernel Time 00:00:00.060
QuotaPoolUsage[PagedPool] 30276
QuotaPoolUsage[NonPagedPool] 2640
Working Set Sizes (now,min,max) (966, 50, 345) (3864KB, 200KB, 1380KB)
PeakWorkingSetSize 966
VirtualSize 28 Mb

```

```

PeakVirtualSize           30 Mb
PageFaultCount           996
MemoryPriority            BACKGROUND
BasePriority               8
CommitCharge              346

THREAD 861fe030 Cid 0f20.0f90 Teb: 7ffde000 Win32Thread: e1d1d650 WAIT:
(WrUserRequest) UserMode Non-Alertable
    86f45030 SynchronizationEvent
        Not impersonating
        DeviceMap             e27058d0
        Owning Process        86a7d030      Image: notepad.exe
        Wait Start TickCount 26300875      Ticks: 62664 (0:00:10:27.542)
        Context Switch Count 424          LargeStack
        UserTime              00:00:00.0000
        KernelTime             00:00:00.0060
        Start Address kernel32!BaseProcessStartThunk (0x77e813f2)
        Win32 Start Address WinDBG!'string' (0x01006ae0)
        Stack Init f7921000 Current f7920c20 Base f7921000 Limit f791b000 Call 0
        Priority 10 BasePriority 8 PriorityDecrement 0 DecrementCount 16
        Kernel stack not resident.

```

8.2.5 访问令牌

EPROCESS 结构中的 Token 字段记录着这个进程的 TOKEN 结构的地址，进程的很多与安全有关的信息是记录在这个结构中的。在前面的显示结果中找到 Token 字段的值，然后使用 !Token 命令便可以观察其详细信息（见清单 8-2）。

清单 8-2 访问令牌 (token) 的属性

```

lkd> !Token e24d8510
_TOKEN e24d8510
TS Session ID: 0
User: S-1-5-21-1757981266-725345543-1404487317-19316
Groups:
 00 S-1-5-21-1757981266-725345543-1404487317-513
    Attributes - Mandatory Default Enabled
 01 S-1-1-0
    Attributes - Mandatory Default Enabled
...
Primary Group: S-1-5-21-1757981266-725345543-1404487317-513
Privils:
 00 0x000000017 SeChangeNotifyPrivilege      Attributes - Enabled Default
 01 0x000000008 SeSecurityPrivilege         Attributes -
 02 0x000000011 SeBackupPrivilege           Attributes -
 03 0x000000012 SeRestorePrivilege          Attributes -
 04 0x00000000c SeSystemtimePrivilege       Attributes -
 05 0x000000013 SeShutdownPrivilege         Attributes -
 06 0x000000018 SeRemoteShutdownPrivilege   Attributes -
 07 0x000000009 SeTakeOwnershipPrivilege   Attributes -
 08 0x000000014 SeDebugPrivilege           Attributes -
 09 0x000000016 SeSystemEnvironmentPrivilege Attributes -
 10 0x00000000b SeSystemProfilePrivilege   Attributes -
 11 0x00000000d SeProfileSingleProcessPrivilege Attributes -
 12 0x00000000e SeIncreaseBasePriorityPrivilege Attributes -
 13 0x00000000a SeLoadDriverPrivilege      Attributes - Enabled
 14 0x00000000f SeCreatePagefilePrivilege  Attributes -
 15 0x000000005 SeIncreaseQuotaPrivilege   Attributes -
 16 0x000000019 SeUndockPrivilege          Attributes - Enabled
 17 0x00000001c SeManageVolumePrivilege    Attributes -

```

```

Authentication ID:      (0,1b496)
Impersonation Level:   Anonymous
TokenType:             Primary
Source: User32          TokenFlags: 0x9 ( Token in use )
Token ID: 14cb4cc       ParentToken ID: 0
Modified ID:            (0, 14cb4ce)
RestrictedSidCount:    0     RestrictedSids: 00000000

```

也可以使用 `dt nt!_TOKEN` 加上令牌对象的地址（如 `e24d8510`）来观察令牌对象。详细解释令牌有关的内容超出了本书的范围，感兴趣的读者可以阅读参考文献 1 的第 8 章。

8.2.6 PEB

PEB 的全称是进程环境块，它包含了进程的大多数用户态信息。与 EPROCESS 结构是位于内核空间中的不同，PEB 是在内核态建立后映射到用户空间的。因此，在一个系统中，多个进程的 PEB 地址可能是同一个值。

使用 `dt _PEB` 命令可以显示出 PEB 结构的字段及其当前值。因为 PEB 的地址位于用户空间，所以在内核调试会话中应该先用 `.process` 命令设置当前的隐含进程（见清单 8-3）。

清单 8-3 PEB 结构

```

1kd> .process 86a7d030
Implicit process is now 86a7d030
1kd> dt _PEB 7ffdf000
+0x000 InheritedAddressSpace : 0 ''
+0x001 ReadImageFileExecOptions : 0 ''
+0x002 BeingDebugged        : 0 ''           // 是否在被调试
+0x003 SpareBool            : 0 ''
+0x004 Mutant               : 0xffffffff
+0x008 ImageBaseAddress    : 0x01000000 // 执行映像 (EXE) 的基地址
+0x00c Ldr                  : 0x00191e90 _PEB_LDR_DATA
+0x010 ProcessParameters   : 0x00020000 _RTL_USER_PROCESS_PARAMETERS
+0x014 SubSystemData        : (null)
+0x018 ProcessHeap          : 0x00090000 // 进程堆，参见 23.1 节
+0x01c FastPebLock          : 0x77fc49e0 _RTL_CRITICAL_SECTION
+0x020 FastPebLockRoutine   : 0x77f5b2a0
+0x024 FastPebUnlockRoutine : 0x77f5b380
+0x028 EnvironmentUpdateCount : 1
+0x02c KernelCallbackTable : 0x77d429b8
+0x030 SystemReserved       : [1] 0
+0x034 ExecuteOptions       : 0y00
+0x034 SpareBits            : 0y00000000000000000000000000000000 (0)
+0x038 FreeList              : (null)
+0x03c TlsExpansionCounter : 0
+0x040 TlsBitmap             : 0x77fc4680
+0x044 TlsBitmapBits         : [2] 0xffff
+0x04c ReadOnlySharedMemoryBase : 0x7f6f0000
+0x050 ReadOnlySharedMemoryHeap : 0x7f6f0000
+0x054 ReadOnlyStaticServerData : 0x7f6f0688 -> (null)
+0x058 AnsiCodePageData     : 0x7ffa0000
+0x05c OemCodePageData      : 0x7ffa0000
+0x060 UnicodeCaseTableData : 0x7ffd1000
+0x064 NumberOfProcessors   : 1           // CPU 个数

```

```

+0x068 NtGlobalFlag          : 0x24400      //全局标志
+0x070 CriticalSectionTimeout: _LARGE_INTEGER 0xffffe86d`079b8000
+0x078 HeapSegmentReserve   : 0x100000     //默认进程堆的总保留空间, 1MB
+0x07c HeapSegmentCommit    : 0x2000       //默认进程堆的已提交空间
+0x080 HeapDeCommitTotalFreeThreshold : 0x10000
+0x084 HeapDeCommitFreeBlockThreshold : 0x1000
+0x088 NumberOfHeaps        : 0xc          //堆的个数
+0x08c MaximumNumberOfHeaps : 0x10         //堆的最大个数
+0x090 ProcessHeaps         : 0x77fc5a80 -> 0x00090000 //保存堆句柄的数组地址
+0x094 GdiSharedHandleTable : 0x003b0000   //GDI 共享句柄表
+0x098 ProcessStarterHelper : (null)
+0x09c GdiDCAttributeList   : 0x14
+0x0a0 LoaderLock          : 0x77fc1774
+0x0a4 OSMajorVersion       : 5            //操作系统主版本号
+0x0a8 OSMinorVersion       : 1            //操作系统子版本号
+0x0ac OSBuildNumber        : 0xa28        //操作系统构建号, 即 2600
+0x0ae OSCSDVersion         : 0x100        //Service Pack 版本号
+0x0b0 OSPlatformId         : 2            //系统类别, 2 代表 NT, 1 代表 9x, 3 代表 Windows CE
+0x0b4 ImageSubsystem       : 2            //环境子系统 ID
+0x0b8 ImageSubsystemMajorVersion: 4        //环境子系统主版本号
+0x0bc ImageSubsystemMinorVersion: 0xa      //环境子系统子版本号
+0x0c0 ImageProcessAffinityMask: 0
+0x0c4 GdiHandleBuffer      : [34] 0
+0x14c PostProcessInitRoutine : (null)
+0x150 TlsExpansionBitmap   : 0x77fc4660
+0x154 TlsExpansionBitmapBits: [32] 0
+0x1d4 SessionId           : 0            //所属会话的 ID
+0x1d8 AppCompatFlags        : _ULARGE_INTEGER 0x0
+0x1e0 AppCompatFlagsUser    : _ULARGE_INTEGER 0x0
+0x1e8 pShimData            : (null)
+0x1ec AppCompatInfo         : (null)
+0x1f0 CSDVersion           : _UNICODE_STRING "Service Pack 1"
+0x1f8 ActivationContextData: 0x00080000
+0x1fc ProcessAssemblyStorageMap : 0x000930f8
+0x200 SystemDefaultActivationContextData : 0x00070000
+0x204 SystemAssemblyStorageMap : (null)
+0x208 MinimumStackCommit    : 0

```

也可以使用!peb 扩展命令来观察进程环境块, 比如使用!peb 7fffd000 命令便可以显示位于 0x7fffd000 处的 PEB 结构。

8.2.7 其他字段

下面再观察!process 0 0 命令显示的另外几个重要信息。因为篇幅限制, 我们只对它们做简要的介绍。

8.2.8 SessionId

进程的 SessionId 是指该进程所在的 Windows 会话 (Session) 的 ID 号。当有多个用户同时登录时, Windows 会为每个登录用户建立一个会话, 每个会话有自己的 WorkStation 和桌面 (Desktop)。这样大家便可以工作在不同的“会话”中共同使用同一个 Windows 系统。对于典型的 XP 系统, 当只有一个用户登录时, 用户启动的程序

和系统服务都运行在 Session 0。当切换到另一个用户账号时（Switch User，不是 Log off），系统会建立 Session 1，依此类推。为了提高系统服务的安全性，Vista 只允许系统服务运行在 Session 0，因此一登录到 Vista 中，我们就会看到有两个会话，有两个 Windows 子系统进程（CSRSS）在运行。

8.2.9 进程 ID

Cid 即进程 ID，又叫 Client ID（用户 ID）。进程 ID 是标识进程的一个整数，很多用户态的函数使用它作为标识进程的参数。内核态主要使用 EPROCESS 指针来标识一个进程。

8.2.10 父进程 ID

Parent Cid 是父进程的进程 ID，即创建该进程的那个进程的进程 ID。

8.2.11 页目录基地址

DirBase 代表的是该进程的页目录基地址，即切换到该进程时，CR3 寄存器的内容。页目录地址是将虚拟地址转换为物理地址的必须参数。我们在 2.5 节中介绍如何将虚拟地址转为物理地址时曾经提到过页目录基地址。

因为页目录是按 4KB 边界对齐的，所以 DirBase 的低 12 位总是 0。因为只需要使用高 20 位的内容，所以 DirBase 的高 20 位又有一个专门的名字，叫页帧编号（Page Frame Number），简称为 PFN。例如，前面命令结果中显示记事本进程的 DirBase 值是 0x1f350000，那么它的 PFN 便是 0x1f350。

WinDBG 的一些内存命令是使用 PFN 作为参数的，例如使用!ptov 扩展命令加上 PFN，便可以列出对应进程中所有物理地址到虚拟地址间的映射，如!ptov 1f350（输出结果非常长，从略）。

8.2.12 对象表格

ObjectTable 的含义是该进程的内核对象和句柄表格。Windows 就是使用这个表格将句柄翻译为指向内核对象的指针。使用!handle 命令可以查看句柄和对象信息。

在内核调试对话中，该命令的格式是：

```
!handle [要显示的句柄索引 [显示标志 [进程 ID 或 EPROCESS 指针 [类型]]]]
```

比如 !handle 0 0 86a7d030 会显示出 86a7d030 进程的所有句柄概况。

在用户调试对话中，命令格式为：

```
!handle [要显示的句柄索引 [显示标志 [类型]]]
```

使用!object 命令可以进一步查看内核对象的信息。

8.2.13 句柄数量

HandleCount 即该进程所使用的句柄个数，也就是 ObjectTable 所包含的表项数。

8.2.14 观察进程

观察进程的一个更简单的方法就是使用 Windows 操作系统自带的任务管理器（如图 8-3 所示）。有 3 种方法可以启动任务管理器：按 Ctrl+Shift+Esc 热键；在任务栏上点击右键，然后选择任务管理器；按 Ctrl+Alt+Del 热键，然后选择任务管理器。通过 View 菜单的“Select Columns...”可以选择要观察的字段。

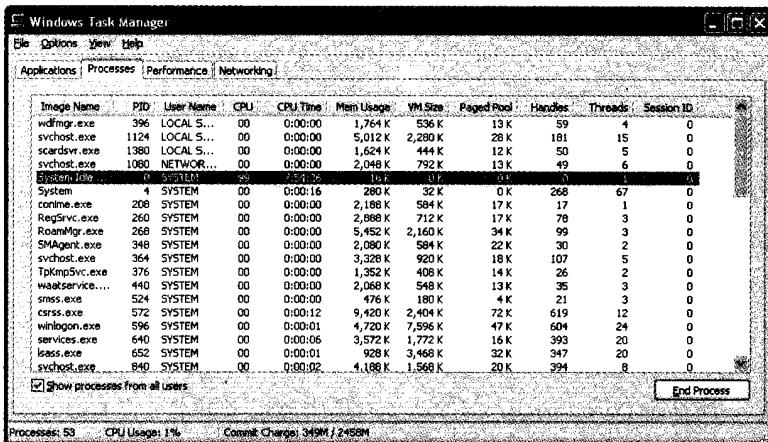


图 8-3 通过任务管理器观察系统中的进程信息

有了前面的基础后，大家就可以很容易地把任务管理器中的各个列（图 8-3）与 EPROCESS 结构对应起来，在此不再赘述。

除了任务管理器，还有一些工具也可以观察进程的内部信息，比如 Process and Thread Status (PStat.exe)、Process Tree (PTree.exe)、Process Explorer (ProcExp.exe)、Process Viewer (PView.exe) 和 Task List (TList.exe) 等。

8.3 内核模式和用户模式

根据上一节的介绍，Windows 会把操作系统的代码和数据映射到系统中所有进程的系统空间中。这样，用户空间中的程序指针便可以指向系统空间中的数据和代码。那么如何防止系统空间中的操作系统数据或代码被破坏呢？答案是利用权限控制来实现对系统空间的保护。

8.3.1 访问模式

Windows 定义了两种访问模式 (access modes): 用户模式 (user mode) 和内核模式 (kernel mode)。应用程序 (代码) 运行在用户模式下, 操作系统代码运行在内核模式下。内核模式对应于处理器的最高权限级别, 在内核模式下执行的代码可以访问所有系统资源并具有使用所有特权指令的权利。相对而言, 用户模式对应于较低的处理器优先级, 在用户模式下执行的代码只可以访问系统允许其访问的内存空间, 并且没有使用特权指令的权利。

在第 2 章我们介绍过, IA-32 处理器定义了 4 种特权级别 (privilege level), 或者称为环 (ring), 分别为 0、1、2、3, 优先级 0 (环 0) 的特权级别最高。处理器在硬件一级保证高优先级别的数据和代码不会被低优先级的代码破坏。Windows 使用了 IA-32 处理器所定义的 4 种优先级别中的两种, 优先级 3 (环 3) 用于用户模式, 优先级 0 用于内核模式。之所以只使用了其中的两种, 主要是因为有些处理器只支持两种优先级别, 比如 Compaq Alpha 处理器。值得说明的是, 对 x86 处理器来说, 并没有任何寄存器表明处理器当前处于何种模式 (或优先级) 下, 优先级别只是代码或数据所在的内存段或页的一个属性, 参见 2.6 节和 2.7 节。

因为内核模式下的数据和代码位于较高的优先级上, 所以用户模式下的代码不可以直接访问系统空间中的数据, 也不可以直接调用系统空间中的任何函数或例程。任何这样的尝试都会导致保护性错误。也就是说, 即使用户空间中的代码指针正确指向了要访问的数据或代码, 但一旦访问发生, 那么处理器便会检测到该访问是违法的, 会停止该访问并产生保护性异常 (#GP)。

虽然不可以直接访问, 但是用户程序可以通过调用系统服务来间接访问系统空间中的数据或间接调用执行系统空间中的代码。当调用系统服务时, 调用线程会从用户模式切换到内核模式, 调用结束后再返回到用户模式, 也就是所谓的模式切换, 有时也被称为上下文切换 (Context Switch)。在每个线程的 KTHREAD 结构中, 有一个名为 ContextSwitches 的字段, 专门用来记录这个线程的模式切换次数。模式切换是通过软中断或专门的快速系统调用 (Fast System Call) 指令来实现的。下面通过一个例子来分别介绍这两种切换机制。

8.3.2 使用 INT 2E 切换到内核模式

图 8-4 画出了在 Windows 2000 中从应用程序调用 ReadFile() API 的过程。因为 ReadFile() API 是从 Kernel32.dll 导出的, 所以我们看到该调用首先转到 Kernel32.dll 中的 ReadFile() 函数, ReadFile() 函数在对参数进行简单检查后便调用 NtDll.dll 中的 NtReadFile() 函数。

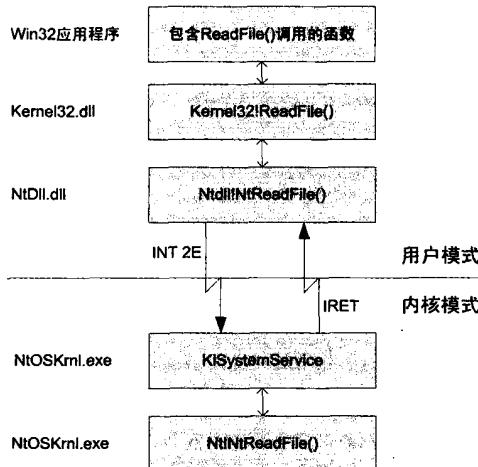


图 8-4 通过 INT 2E 进行的系统调用

通过反汇编可以看到，NtDll.dll 中的 NtReadFile() 函数非常简短，首先将 ReadFile() 对应的系统服务号（0xa1，与版本有关）放入 EAX 寄存器，参数指针放入 EDX 寄存器，然后便通过 INT n 指令发出调用。这里要说明的一点是虽然每个系统服务都具有一个唯一的号码，但微软公司没有公开这些服务号，也不保证这些号码在不同的 Windows 版本中会保持一致。

```

ntdll!NtReadFile: // Windows 2000
77f8fb5d b8a1000000      mov     eax, 0xa1
77f8fb62 8d542404      lea     edx, [esp+0x4]
77f8fb66 cd2e           int    2e
77f8fb68 c22400          ret    0x24
  
```

在 WinDBG 下通过 !idt 2e 命令可以看到 2e 向量对应的服务例程是 KiSystemService()。KiSystemService() 是内核态中专门用来分发系统调用的例程。

```

1kd> !idt 2e
Dumping IDT:
2e: 804db1ed nt!KiSystemService
  
```

Windows 将 2e 号向量专门用作系统调用，在启动早期初始化中断描述表（IDT）时（第 11 章中会进一步讨论）便注册好了合适的服务例程。因此当 NtDll 中的 NtReadFile 发出 INT 2E 指令后，CPU 便会通过 IDT 表找到 KiSystemService 函数。因为 KiSystemService 函数是位于内核空间的，所以 CPU 在把执行权交给 KiSystemService 函数前，会做好从用户态切换到内核态的各种工作，包括：

- 权限检查，即检查源位置和目标位置所在的代码段权限，核实是否可以转移。
- 准备内核态使用的栈，为了保证内核安全，所有线程在内核态执行时都必须使用位于内核空间的内核栈（kernel stack），内核栈的大小一般为 8KB 或 12KB。

KiSystemService 会根据服务 ID 从系统服务分发表（System Service Dispatch Table）中查找到要调用的服务函数地址和参数描述，然后将参数从用户态栈复制到该

线程的内核栈中，最后 KiSystemService 调用内核中真正的 NtReadFile() 函数，执行读文件的操作，操作结束后会返回到 KiSystemService()，KiSystemService() 会将操作结果复制回该线程用户态栈，最后通过 IRET 指令将执行权交回给 NtDll.dll 中的 NtReadFile() 函数（继续执行 INT 2E 后面的那条指令）。

通过 INT 2E 进行系统调用时，CPU 必须从内存中分别加载门描述符和段描述符后才能得到 KiSystemService() 的地址，即使门描述符和段描述符已经在高速缓存中，CPU 也需要通过“内存读（memory read）”操作从高速缓存中读出这些数据，然后进行权限检查。因为系统调用是非常频繁的操作，所以，如果能减少这些开销还是非常有意义的。可以从两个方面来降低开销，一是把系统调用服务例程的地址放到寄存器中以避免读内存操作，因为读寄存器的速度比读内存的速度要快很多；二是避免权限检查，也就是使用特殊的指令让 CPU 省去那些对系统服务调用来说根本不需要的权限检查。奔腾 II 处理器引入的 SYSENTER/SYSEXIT 指令正是按这一思路设计的。AMD K7 引入的 SYSCALL/SYSRETURN 指令也是为这一目的而设计的。相对于 INT 2E，使用这些指令可以加快系统调用的速度，因此利用这些指令进行的系统调用被称为快速系统调用（Fast System Call）。

8.3.3 快速系统调用

下面我们介绍 Windows 是如何利用 IA-32 处理器的 SYSENTER/SYSEXIT 指令（从奔腾 II 开始）实现快速系统调用的。首先，Windows 2000 或之前的 Windows 不支持快速系统调用，它们只能使用前面介绍的 INT 2E 方式进行系统调用。Windows XP 和 Windows Server 2003 或更新的版本在启动过程中会通过 CPUID 指令检测 CPU 是否支持快速系统调用指令（EDX 寄存器的 SEP 标志位）。如果 CPU 不支持这些指令，那么仍使用 INT 2E 方式，如果 CPU 支持这些指令，那么 Windows 便会决定使用新的方式进行系统调用，并做好如下准备工作：

- 在 GDT（全局描述符表）中建立 4 个段描述符，分别用来描述供 SYSENTER 指令进入内核模式时使用的代码段（CS）和栈段（SS），以及 SYSEXIT 指令从内核模式返回用户模式时使用的代码段（CS）和栈段（SS）。这 4 个描述符在 GDT 表中的排列应该严格按照以上顺序，这样只要指定一个段描述符的位置便能计算出其他的。
- 设置表 8-2 中专门用于系统调用的 MSR 寄存器（关于 MSR 寄存器的详细介绍见 2.4.3 节），SYSENTER_EIP_MSR 用于指定新的程序指针，也就是 SYSENTER 指令要跳转到的目标例程地址。Windows 会将其设置为 KiFastCallEntry 的地址，因为 KiFastCallEntry 例程是 Windows 内核中专门用来受理快速系统调用的。SYSENTER_CS_MSR 寄存器用来指定新的代码段，也就是 KiFastCallEntry 所在的代码段。SYSENTER_ESP_MSR 用于指定新的栈指针（ESP）。新的栈段是由 SYSENTER_CS_MSR 的值加 8 得来的。

- 将一小段名为 SystemCallStub 的代码复制到 SharedUserData 内存区，该内存区会被映射到每个 Win32 进程的进程空间中。这样当应用程序每次进行系统调用时，NtDll 中的残根（stub）函数便调用这段 SystemCallStub 代码。SystemCallStub 的内容会因系统硬件的不同而不同，对于 IA-32 处理器，该代码使用 SYSENTER 指令，对于 AMD 处理器使用 SYSCALL 指令。

表 8-2 供 SYSENTER 指令使用的 MSR 寄存器

MSR 寄存器名称	MSR 地址	用途
SYSENTER_CS_MSR	174h	目标代码段的 CS 选择子
SYSENTER_ESP_MSR	175h	目标 ESP
SYSENTER_EIP_MSR	176h	目标 EIP

例如在配有 Pentium M CPU 的 Windows XP 系统上，以上 3 个寄存器的值分别是：

```
1kd> rdmsr 174
msr[174] = 00000000`00000008
1kd> rdmsr 175
msr[175] = 00000000`bacd8000
1kd> rdmsr 176
msr[176] = 00000000`8053cad0
```

其中 SYSENTER_CS_MSR 寄存器的值为 8，这是 Windows 系统的内核代码段的选择子，即常量 KGDT_R0_CODE 的值。WinDBG 帮助文件中关于 dg 命令的说明中列出了这个常量。SYSENTER_ESP_MSR 的值是 8053cad0，检查 nt 内核中 KiFastCall-Entry 函数的地址：

```
1kd> x nt!KiFastCallEntry
8053cad0 nt!KiFastCallEntry = <no type information>
```

可见，Windows 把快速系统调用的目标指向的是内核代码段中的 KiFastCall-Entry 函数。

通过反汇编 Windows XP 下 NtDll 中的 NtReadFile() 函数，可以看到 SystemCallStub 被映射到进程的 0x7ffe0300 位置。与前面 Windows 2000 下的版本相比，容易看到该服务的系统服务号码在这两个版本间是不同的。

```
kd> u ntdll...
ntdll!NtReadFile: // Windows XP
77f5bfa8 b8b7000000    mov     eax, 0xb7
77f5bfad ba0003fe7f    mov     edx, 0x7ffe0300
77f5bfb2 ffd2          call    edx {SharedUserData!SystemCallStub (7ffe0300)}
77f5bfb4 c22400         ret    0x24
77f5bfb7 90             nop
```

反汇编 SystemCallStub 会发现，它只包含 3 条指令：将栈指针（ESP 寄存器）放入 EDX 寄存器，这样做有两个目的，一是传递参数；二是指定从内核模式返回时的栈地址，执行 SYSENTER 指令；返回。因为笔者使用的是英特尔奔腾 M 处理器，所以此处是 SYSENTER 指令，对于 AMD 处理器，此处应该是 SYSCALL 指令。

```
kd> u...
SharedUserData!SystemCallStub:
7ffe0300 8bd4          mov     edx, esp
7ffe0302 0f34          sysenter
7ffe0304 c3             ret
```

下面让我们看一下 KiFastCallEntry 例程，其清单如下所示：

```
kd> u nt!KiFastCallEntry L20
nt!KiFastCallEntry:
804db1bb 368b0d40f0dff mov    ecx,ss:[ffff040]
804db1c2 368b6104      mov    esp,ss:[ecx+0x4]
804db1c6 b90403fe7f    mov    ecx,0x7ffe0304
804db1cb 3b2504f0dff cmp    esp,[ffff004]
804db1d1 0f84cc030000 je     nt!KiServiceExit2+0x13f (804db5a3)
804db1d7 6a23      push   0x23
804db1d9 52       push   edx
804db1da 83c208    add    edx,0x8
804db1dd 6802020000 push   0x202
804db1e2 6a02      push   0x2
804db1e4 9d       popfd 
804db1e5 6a1b      push   0x1b
804db1e7 51       push   ecx // Fall Through, 自然进入 KiSystemService 函数
nt!KiSystemService:
804db1e8 90      nop
804db1e9 90      nop
804db1ea 90      nop
804db1eb 90      nop
804db1ec 90      nop
nt!KiSystemService:
804db1ed 6a00      push   0x0
804db1ef 55      push   ebp
```

显而易见，KiFastCallEntry 在做了些简单操作后，便进入（Fall Through）到 KiSystemService 函数了，也就是说，快速系统调用和使用 INT 2E 进行的系统调用在内核中的处理绝大部分都是一样的。另外，请大家注意 ECX 寄存器，第 3 行（mov ecx,0x7ffe0304）将其值设为 0x7ffe0304，也就是 SharedUserData 内存区里 SystemCallStub 例程中 ret 指令的地址（参见上文的 SystemCallStub 代码）。在进入 nt!KiSystemService 之前，ECX 连同其他一些参数被压入栈。事实上，ECX 是用来指定 SYSEXIT 返回用户模式时的目标地址。当使用 INT 2E 进行系统调用时，由于 INT n 指令会自动将中断发生时的 CS 和 EIP 寄存器压入栈，当中断处理例程通过执行 IRET 返回时，IRETD 指令会使用栈中保存的 CS 和 EIP 值返回到合适的位置。由于 SYSENTER 指令不会向栈中压入要返回的位置，所以 SYSEXIT 指令必须通过其他机制知道要返回的位置。这便是压入 ECX 寄存器的原因。通过反汇编 KiSystemCallExit2 例程，我们可以看到在执行 SYSEXIT 指令之前，ECX 寄存器的值又从栈中恢复出来了。

```
kd> u nt!KiSystemCallExit 120
nt!KiSystemCallExit:
804db3b4 cf      iretd
nt!KiSystemCallExit2:
804db3b5 5a      pop    edx
804db3b6 83c408  add    esp,0x8
804db3b9 59      pop    ecx
804db3ba fb      sti
804db3bb 0f35    sysexit
nt!KiSystemCallExit3:
804db3bd 59      pop    ecx
804db3be 83c408  add    esp,0x8
804db3c1 5c      pop    esp
804db3c2 0f07    sysret
```

以上代码中包含了3个从系统调用返回的例程，KiSystemCallExit、KiSystemCallExit2和KiSystemCallExit3，它们分别对应于使用INT 2E、SYSENTER和SYSCALL发起的系统调用，如表8-3所示。

表8-3 系统调用归纳

发起系统调用的方法	入口内核例程	返回时的指令	返回时调用的内核例程
INT 2E	KiSystemService	IRET	KiSystemCallExit
SYSENTER	KiFastCallEntry	SYSEXIT	KiSystemCallExit2
SYSCALL	KiFastCallEntry	SYSRETURN	KiSystemCallExit3

图8-5画出了使用SYSENTER/SYSEXIT指令对进行系统调用的完整过程（以调用ReadFile服务为例）。

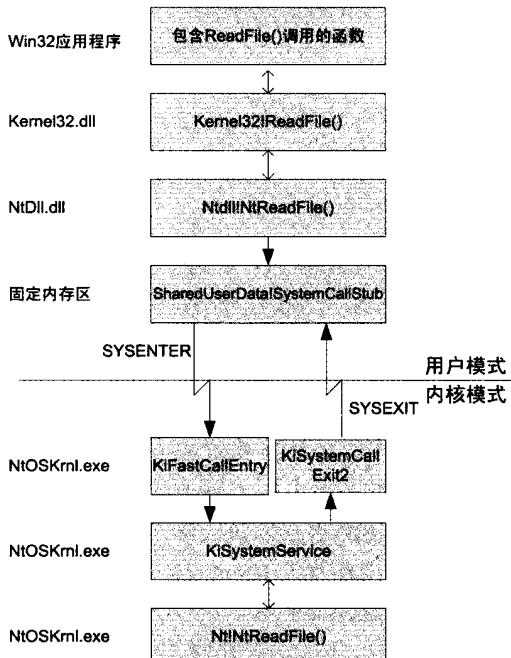


图8-5 快速系统调用（针对IA-32处理器）

8.3.4 逆向调用

前面我们介绍了从用户态进入内核态的两种方法，通过这两种方法，用户态的代码可以“调用”位于内核态的系统服务。那么内核态的代码是否可以主动调用用户态的代码呢？答案是肯定的，这种调用通常被称为逆向调用（Reverse Call）。

简单来说，逆向调用的过程是这样的。首先内核代码使用内核函数KiCallUserMode发起调用。接下来的执行过程与从系统调用返回（KiServiceExit）时类似，不过进入用户态时执行的是NtDll中的KiUserCallbackDispatcher。而后KiUserCallbackDispatcher会调用内核希望调用的用户态函数。当用户态的工作完成后，执行返回动作的函数会执

行 INT 2B 指令，也就是触发一个 0x2B 异常。这个异常的处理函数是内核态的 KiCallbackReturn 函数。于是，通过 INT 2B 异常，CPU 便又跳回到内核态继续执行了。

```
1kd> !idt 2b
Dumping IDT:
2b: 8053d070 nt!KiCallbackReturn
```

以上是使用 WinDBG 的!idt 命令观察到的 0x2B 异常的处理函数。

8.3.5 实例分析

下面通过一个实际例子来进一步理解系统调用和逆向调用的执行过程。清单 8-4 显示了使用 WinDBG 的内核调试会话捕捉到的记事本进程发起系统调用进入到内核、内核函数又执行逆向调用的全过程（栈回溯序列）。

清单 8-4 系统调用和逆向调用的执行过程

```
kd> kn
# ChildEBP RetAddr
00 0006fe94 77fb4da6 USER32!XyCallbackReturn
01 0006fe94 8050f8ae ntdll!KiUserCallbackDispatcher+0x13
02 f4fc19b4 80595d2c nt!KiCallUserMode+0x4
03 f4fc1a10 bf871e98 nt!KeUserModeCallback+0x87
04 f4fc1a90 bf8748d4 win32k!SfnDWORD+0xa0
05 f4fc1ad8 bf87148d win32k!xxxSendMessageToClient+0x174
06 f4fc1b24 bf8714d3 win32k!xxxSendMessageTimeout+0x1a6
07 f4fc1b44 bf8635f6 win32k!xxxSendMessage+0x1a
08 f4fc1b74 bf84a620 win32k!xxxMouseActivate+0x22d
09 f4fc1c98 bf87a0c1 win32k!xxxScanSysQueue+0x828
0a f4fc1cec bf87a8ad win32k!xxxRealInternalGetMessage+0x32c
0b f4fc1d4c 804da140 win32k!NtUserGetMessage+0x27
0c f4fc1d4c 7ffe0304 nt!KiSystemService+0xc4
0d 0006feb8 77d43a21 SharedUserData!SystemCallStub+0x2
0e 0006feb8 77d43c95 USER32!NtUserGetMessage+0xc
0f 0006fed8 010028e4 USER32!GetMessageW+0x31
10 0006ff1c 01006c54 notepad!WinMain+0xe3
11 0006ffcc 77e814c7 notepad!WinMainCRTStartup+0x174
12 0006ffff 00000000 kernel32!BaseProcessStart+0x23
```

根据执行的先后顺序，最下面一行 (#12) 对应的是进程的启动函数 BaseProcessStart，而后是编译器生成的进程启动函数 WinMainCRTStartup，以及记事本程序自己的入口函数 WinMain。帧#0f 是记事本程序在调用 GetMessage API 进入消息循环。接下来 GetMessage API 调用 Windows 子系统服务的残根函数 NtUserGetMessage。从第 2 列的栈帧地址（将在第 22 章详细介绍）都小于 0x8000000000 可以看出，帧#12～#0d 都是在用户态执行的。#0d 是在执行我们前面分析过的 SystemCallStub，而后 (#0c) 便进入了内核态的 KiSystemService。KiSystemService 根据系统服务号码，将调用分发给 Windows 子系统内核模块 win32k 中的 NtUserGetMessage 函数。

帧#0a～#05 是内核态的窗口消息函数在工作。帧#07～#05 显示要把一个窗口消息发送到用户态。#04 的 SfnDWORD 是在将消息组织好后调用 KeUserModeCallback 函数发起逆向调用。#02 表明在执行 KiCallUserMode 函数，#01 表明已经进入到用户态执行，这两行之间的部分过程没有显示出来。同样，#01 和#00 之间的执行用户态函

数的过程没有完全体现出来。`XyCallbackReturn` 函数是用于返回内核态的，它的代码很简单，只有如下几条指令。

```
USER32!XyCallbackReturn:  
001b:77d44168 8b442404    mov    eax,dword ptr [esp+4] ss:0023:0006fe84=00000000  
001b:77d4416c cd2b        int    2Bh  
001b:77d4416e c20400      ret    4
```

第1行是把用户态函数的执行结果赋给EAX寄存器，第2行便是执行INT 2B指令。执行过INT 2B后，CPU便转去执行异常处理程序KiCallbackReturn，回到了内核态。

8.4 架构和系统部件

本节将介绍 Windows 系统的总体架构和重要的系统部件，主要包括关键的进程和系统文件，以及环境子系统。

8.4.1 概览

图 8-6 简要地勾画出了 Windows 操作系统的基本架构。图中画出了内核模式下的关键组件及用户态下的重要进程和动态链接库。该图参考了《深入解析 Windows 操作系统, 第 4 版》(电子工业出版社, 2007 年) 的图 2-3, 本书笔者增加了 Vista 系统新引入的部件并对原图其他地方做了部分修改。

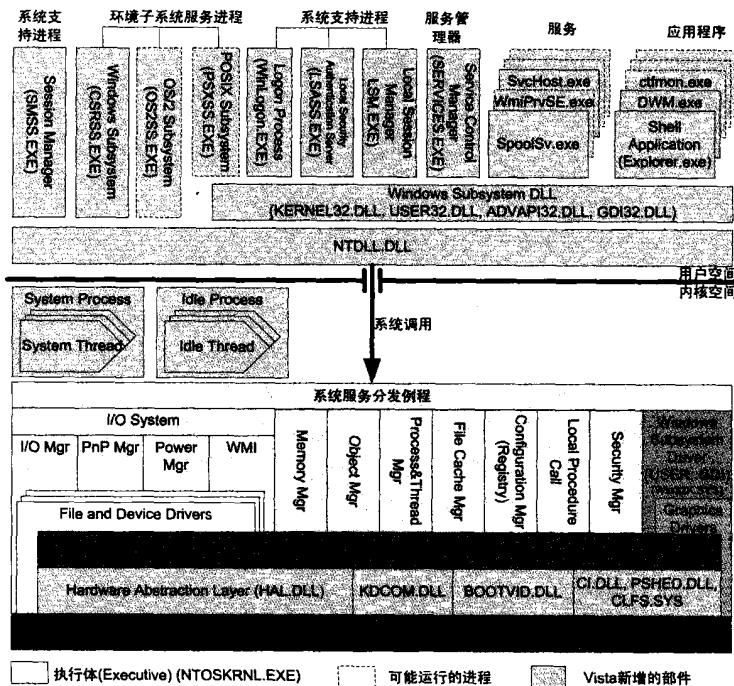


图 8-6 Windows 操作系统的核心部件

以中间的黑线为界，上面描述的是用户空间，下面描述的是内核空间。

8.4.2 内核空间

从职能角度来看，内核空间主要包含以下部件。

- 硬件抽象层（Hardware Abstraction Layer），简称 HAL。HAL 的主要作用是隔离硬件差异性，使内核和顶层模块可以通过统一的方式来访问硬件。
- 操作系统内核，负责线程调度、中断处理、异常分发、多处理器同步等关键任务，是操作系统最核心的部分。
- 执行体（Executive），执行体包含了操作系统的基本服务，包括内存管理、进程管理、输入输出（I/O）、网络和进程间通信等。如果把操作系统内核比喻成系统的最高权力机构，那么执行体便是它的一个个职能部门，负责各方面的事务。
- 内核态驱动程序，包括文件系统和图形显示驱动程序，以及用于其他硬件的驱动程序，有些是 Windows 自带的，有些是其他厂商开发的。驱动程序是对内核功能的补充。
- Windows 子系统驱动程序（Win32K.SYS），包括 USER 和 GDI 两大部分，USER 部分负责窗口管理、用户输入等，GDI（Graphics Device Interface）部分负责显示输出和各种图形操作。
- 内核支持模块，包括用于内核调试的 KDCOM.DLL，用于启动阶段的显示驱动 BOOTVID.DLL，Windows Vista 引入的用于检查模块完好性的 CI.DLL（CI 是 Code Integrity 的缩写），支持日志功能的 CLFS.SYS，支持 WHEA 的 PSHED.DLL。

内核支持模块中的 PSHED.DLL 和 BOOTCOM.DLL，我们将分别在第 17 章和第 18 章中介绍。下面我们先介绍内核文件和 HAL 文件。

8.4.3 内核文件和 HAL 文件

从文件角度来看，内核与执行体都位于一个文件中，即通常所说的 NT 内核文件。NT 内核文件有几种版本，它们是使用同一套源代码通过不同编译选项而编译出来的。

- 针对单处理器系统优化的单处理器版本，在 64 位的 Windows 中，它的原始文件名为 NTOSKRNL.EXE，在 32 位的 Windows 中，根据是否支持 PAE（Physical Address Extension），它的原始文件名称是 NTKRNLPA.EXE（支持 PAE）或 NTOSKRNL.EXE（不支持 PAE）。
- 可用于多处理器系统的多处理器版本，在 64 位的 Windows 中，它的原始文件名为 NTKRNLMP.EXE，在 32 位的 Windows 中，支持 PAE 的版本的原始文件名称是 NTKRPAMP.EXE，不支持 PAE 的版本是 NTKRNLMP.EXE。

那么，系统是如何决定使用以上版本中的哪一个呢？首先安装程序在安装时会根据系统中的处理器个数，选择单处理器版本或多处理器版本中的一个复制到用户系统中（system32目录）。如果复制的是多处理器版本，那么会将其改为与单处理器版本相同的名字。这也是我们上面强调“原始文件名”的原因，它是相对安装在用户系统后的文件名而言的。

因为PAE是可以在启动选项中开启或关闭的，所以安装程序会将支持PAE的版本和不支持PAE的版本都复制到用户系统中（system32目录），但是会根据上面的原则选择多处理器版本和单处理器版本中的一个，并改名为单处理器版本的名字。

综上所述，在安装好的Windows系统的system32目录中，我们通常可以看到两个NT内核文件，NTOSKRNL.EXE和NTKRNLP.AXE。在Windows系统启动过程中，系统的加载程序（NTLDR或WinLoad）会根据启动选项中是否启用了PAE加载其中的一个。

需要说明的是，多处理器版本也是可以运行在单处理器系统中的，只不过在多处理器系统中必须有的同步措施在单处理器系统中是没有必要的，会牺牲一些性能。随着超线程（Hyper Threading）和多核技术的迅速发展，多处理器系统逐渐成为主流，因此Windows Vista不再安装单处理器版本，无论系统是否有多个处理器，安装程序都会为其安装多处理器版本（NTKRNLM.P.EXE或NTKRPAMP.EXE），然后将其改名为与单处理器版本一样的名字。

通过文件属性对话框中版本页（Version Tab）的Original File name，可以观察到内核文件的原始名称。在笔者的单处理器的Windows Vista系统中，磁盘上的NTOSKRNL.EXE的原始文件名是NTKRNLM.P.EXE。

与内核文件类似，硬件抽象层模块也有多种版本，以适用于不同的硬件平台。

- HAL.DLL 标准平台。
- HALACPI.DLL 符合 ACPI 标准的硬件平台。
- HALAPIC.DLL 支持 APIC（高级可编程中断控制器）的硬件平台。
- HALAACPI.DLL 同时支持 ACPI 和 APIC 的硬件平台。
- HALMPS.DLL 系统中有多个处理器。
- HALMACPI.DLL 支持 ACPI 的多处理器系统。

当安装Windows时，安装程序会根据检测到的情况选择一个合适的HAL文件复制到系统目录中，并改名为HAL.DLL。同样，使用文件属性对话框可以观察它的原始文件名。在笔者的Windows XP系统中，HAL.DLL的原始文件名是halaacpi.dll，即同时支持ACPI和APIC的版本。

Dependency Walker（depends.exe，简称Depends）是了解系统文件用途和相互关

系的一个简单而有效的工具。它可以显示 EXE 和 DLL 文件的输入输出(import/export)情况，比如 DLL 的输出函数列表，文件的依赖关系等。SDK、VC6 和 VS2003/2005 中都包含了该工具。在 Windows XP 系统中，你只要启动 Depends 一次，它便会自动注册，以便以后右键点击 DLL 或 EXE 文件时，快捷菜单(shortcut menu)中便会包含“View Dependencies”项。选择此项，系统便会启动 Depends。

图 8-7 中 Depends 显示的是 Windows Vista 的内核文件(ntoskrnl.exe)的依赖情况。

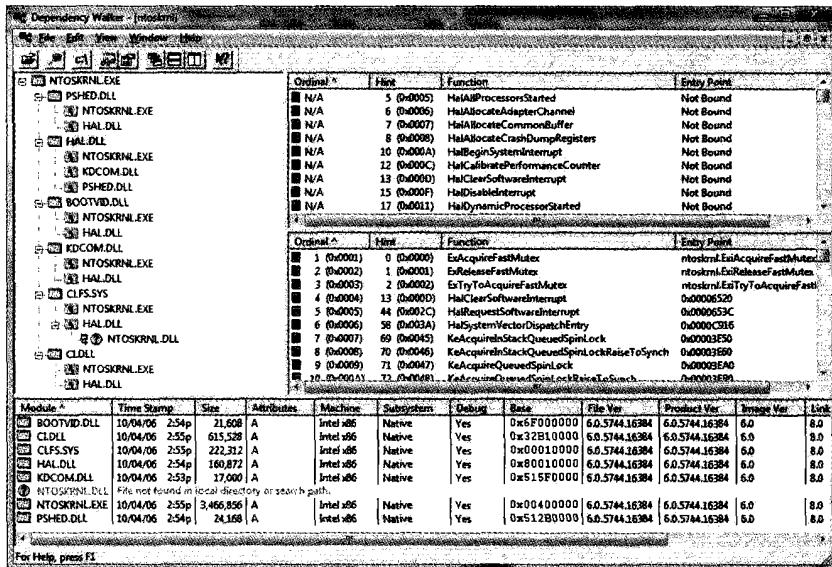


图 8-7 使用 Depends 工具观察 Windows Vista 的 NT 内核文件

左边的树控件显示了 ntoskrnl.exe 直接依赖 6 个模块：PSHED.DLL、BOOTVID.DLL、HAL.DLL、KDCOM.DLL、CLFS.SYS、CI.DLL。

Depends 窗口中部的两个列表分别是，左侧选中模块(HAL.DLL)包含的被 ntoskrnl.exe 使用的函数列表(上)和选中模块输出的所有函数。

窗口最下部的列表显示的是每个文件的大小、属性、版本等信息。

8.4.4 系统和 IDLE 进程

在图 8-6 中靠近中部黑线的左下方，我们画出了两个特殊的进程：系统进程(System Process)和空闲进程(Idle Process)。之所以说它们特殊，是因为这两个进程都具有如下特征。

- 普通的 Windows 进程都是通过使用 CreateProcess 或类似的 API 并指定一个可执行映像文件而创建的。但这两个进程不是，它们没有对应的磁盘映像文件，是在系统启动时“捏造”出来的。

- 普通的Windows进程都有用户态和内核态两个部分。但是这两个进程都只有内核态的部分，没有用户态。或者说，这两个进程只在内核态执行。
- 固定的进程ID，IDLE进程的进程ID是0，系统进程的进程ID是8(Windows 2000)或4(Windows XP及之后)。

简单来说，系统进程是系统内核和大多数系统线程的宿主和载体。在一个典型的Windows系统中，系统进程中几十个乃至上百个系统线程在工作。比如当笔者写作此内容时，系统进程中102个系统线程。

在内核调试会话中，可以使用!process命令来观察系统进程：

```
1kd> !process 4 1
Searching for Process with Cid == 4
PROCESS 8a6f2660 SessionId: none Cid: 0004 Peb: 00000000 ParentCid: 0000
DirBase: 0072c000 ObjectTable: e1001c38 HandleCount: 705.
Image: System
VadRoot 8a6eb870 Vads 4 Clone 0 Private 3. Modified 504318. Locked 0.
DeviceMap e10087c0
Token e1000728
ElapsedTime 2 Days 22:24:34.216 // 约等于系统总运行时间
UserTime 00:00:00.000
KernelTime 00:03:00.203
QuotaPoolUsage[PagedPool] 0
QuotaPoolUsage[NonPagedPool] 0
Working Set Sizes (now,min,max) (75, 0, 345) (300KB, 0KB, 1380KB)
PeakWorkingSetSize 654
VirtualSize 1 Mb
PeakVirtualSize 3 Mb
PageFaultCount 12850
MemoryPriority BACKGROUND
BasePriority 8
CommitCharge 7
```

注意，它的Peb是0，在用户空间的运行时间(UserTime)也是0，这是因为它没有用户态的部分。父进程ID是0，即IDLE进程。映像文件名是System，这个名称是杜撰的，磁盘上并不存在System.exe。

类似的，IDLE进程是IDLE线程的载体，IDLE线程的个数与系统中启用的处理器个数是一致的。当CPU没有其他线程需要执行时(空闲)，CPU会执行IDLE线程。第一个IDLE线程是从系统的初始启动线程蜕变而来的，是系统中“年龄最大”的线程。

!process命令不显示IDLE进程，但可以使用如下方法来观察IDLE进程。在内核调试会话中执行!prcb命令显示处理器控制块(processor control block)：

```
1kd> !prcb
PRCB for Processor 0 at ffdf120:
Threads-- Current 88368470 Next 00000000 Idle 80551d20
Number 0 SetMember 00000001
Interrupt Count -- 04080105
Times -- Dpc 000037a7 Interrupt 00001784
Kernel 00725fff User 0005d1be
```

观察Threads一行，Current字段是当前CPU正在执行的线程的ETHRAD结构(作用相当于进程的EPROCESS结构)的地址，Next是等待执行的线程，Idle字段

的值就是当前 CPU 的 IDLE 线程的 ETHRAD 结构地址。使用!thread 命令可以显示它的详细信息：

```
1kd> !thread 80551d20
THREAD 80551d20 Cid 0000.0000 Teb: 00000000 Win32Thread: 00000000 RUNNING on
processor 0
Not impersonating
Owning Process      80551f80     Image:      Idle
Wait Start TickCount 6399946      Ticks: 9878376 (1:18:52:29.625)
Context Switch Count 23038498
UserTime             00:00:00.000
KernelTime           1 Day 04:18:18.593
Stack Init 80549500 Current 8054924c Base 80549500 Limit 80546500 Call 0
Priority 16 BasePriority 0 PriorityDecrement 0 DecrementCount 0
Unable to get context for thread running on processor 0, HRESULT 0x80004001
```

进程 ID(Cid)字段为 0，即这个线程属于 IDLE 进程，Owning Process 后便是 IDLE 进程的 EPROCESS 结构地址，Image 后的映像文件名 Idle 也是杜撰的。UserTime 为 0，表示 IDLE 线程也只是在内核态执行的。KernelTime 是在内核态的执行时间。最下面一行显示没有能读取到 IDLE 线程的上下文，这是因为我们使用的是本地内核调试对话。如果使用真正的双机内核调试，就可以显示出线程的栈回溯。

有了 IDLE 进程的 EPROCESS 结构地址后，便可以使用!process 命令来观察它了，在此从略。

8.4.5 用户空间

简单来说，用户空间是应用程序代码和各种用户态模块运行的主要场所。从进程的角度来看，用户空间中运行着以下进程。

- 会话管理器进程（SMSS.EXE），它是系统中第一个根据映像文件创建的进程，是在系统启动后期由执行体的初始化函数创建的。它运行后，会加载和初始化 Win32 子系统的内核模块 Win32K.SYS，创建 Win32 子系统服务器进程（CSRSS.EXE），并创建登录进程（WinLogon.EXE）。
- Windows 子系统服务器进程（CSRSS.EXE），负责维护 Windows 子系统的“日常事务”，为子系统中的各个进程提供服务。例如登记进程和线程、管理控制台窗口、管理 DOS 程序虚拟机（VDM）进程等。CSRSS 是 Client/Server Runtime Server Subsystem 的缩写，即客户/服务器（运行时）子系统。
- 登录进程（WinLogon.EXE），负责用户登录和安全有关的事务。它启动后，会创建 LSASS 进程和系统服务管理进程（Services.exe）。Windows XP 的文件保护功能（Windows File Protection，简称 WFP）也是在这个进程中实现的（sfc.dll 和 sfc_os.dll）。
- 本地安全和认证进程（LSASS.EXE），负责用户身份验证，LSASS 是 Local Security Authority Subsystem Service 的缩写。

- 服务管理进程（Services.exe），负责启动和管理系统服务程序。系统服务程序是按照 NT 系统服务（NT Service）规范编写的 EXE 程序，通常没有用户界面，只在后台运行。图 8-5 中列出了几个常见的系统服务，Spoolsv.exe 是打印机脱机服务，WmiPrvSE.exe 是 WMI 提供器管理服务，SvcHost.exe 是一个通用的服务宿主程序。
- OS/2 子系统和 POSIX 子系统服务进程，用于在 Windows 系统中运行 OS/2 和符合 POSIX 标准的程序。这两个只有在需要时才启动，稍后将做进一步讨论。
- 壳程序（Shell），默认为 Explorer.exe，负责显示开始菜单、任务栏和桌面图标等。除了以上列出的常规进程，用户空间中可能还运行着用户安装的各种应用程序和服务。

8.4.6 NTDLL.DLL

NTDLL.DLL 是沟通用户空间和内核空间的桥梁，用户空间的代码通过这个 DLL 来调用内核空间的系统服务。同时，它也是操作系统内核在用户模式下的“代理”，系统会在启动阶段便把它加载到内存中，并把它映射到所有用户进程的进程空间中，而且是映射在相同的位置（虚拟地址）。当内核需要用户空间的配合时，它会使用这个 DLL，因为只有这个 DLL 才存在于每个用户进程的用户空间的固定位置上。例如，上一节我们介绍的内核空间调用用户空间的逆向调用机制的用户态着陆点就是位于 NTDLL.DLL 中的。当我们观察 Windows 系统中的进程时，会发现每个进程中都有这个 DLL。

除了作为内核空间和用户空间的桥梁外，NTDLL.DLL 中还包含了很多支持例程，比如程序映像加载（Ldr 开头的函数）、运行时函数（Rtl 开头的函数）、异常分发例程和调试支持例程等。我们将在下一章中介绍 NTDLL 的调试支持。

8.4.7 环境子系统

为了能够在 Windows 操作系统上运行其他类型的软件，Windows 设计者在定义 Windows 架构时便设计了环境子系统（Environment Subsystem）的概念。不同类型的应用程序运行在不同的环境子系统中。一个 Windows 系统中可以有多个不同的环境子系统同时存在，因此可以同时运行不同类型的应用程序。

Windows 2000 支持如下 3 个环境子系统。

- POSIX 子系统，POSIX 是 Portable Operating System Interface based on UniX 的缩写，用于运行符合 POSIX 标准的程序。系统目录（例如 c:\winnt\system32）下的 PSXSS.EXE（POSIX SubSystem）和 PSXDLL.DLL 是 POSIX 子系统的核心文件。
- OS/2 子系统，支持 16 位的 OS/2(1.2) 程序，系统目录下的 OS2.EXE、OS2SRV.EXE 和 OS2SS.EXE 是 OS/2 子系统的核心文件。

- Windows 子系统，即支持 Windows 程序运行的子系统，包括 Windows 子系统服务器进程（CSRSS.EXE）、子系统驱动程序（Win32K.SYS）、子系统 DLL（KERNEL32.DLL、USER32.DLL、ADVAPI32.DLL、GDI32.DLL）和设备相关的显示和打印驱动程序等。

尽管 Windows 的基本设计思想是支持多个独立的环境子系统，但是为了避免多个子系统都重复实现类似的功能，Windows 子系统有着与其他子系统不同的地位，Windows 子系统可以独立存在而且是系统中不可缺少的部分，其他子系统是以 Windows 子系统为基础的，必须依赖于 Windows 子系统，而且是按需要运行的（当第一次有相应类型的应用程序运行时，才启动所需要的子系统）。

表 8-4 列出了 Windows 子系统的重要文件及其主要功能，大多数 Windows API（Win32 API）是由 Windows 子系统的 DLL 输出的。

表 8-4 Windows 子系统的核心文件

名称	模式	描述
CSRSS.EXE	用户模式	Windows 子系统服务进程的主程序
ADVAPI32.DLL	用户模式	Windows 子系统 DLL 之一，包含如下 API 的入口： 数据加密（Crpt 开头）； 用户和账号管理（Lsa 开头）； 注册表操作（Reg 开头）； WMI（Wmi 开头）； 终端服务（Wts 开头）
GDI32.DLL	用户模式	Windows 子系统 DLL 之一，包含各种图形文字绘制 API（GDI）的入口，如 TextOut()、BitBlt() 等。其中大多数 API 是被转换为系统服务发给内核态的 Windows 子系统驱动程序（Win32K.SYS）
KERNEL32.DLL	用户模式	Windows 子系统 DLL 之一，包含如下 API 的入口。 进程/线程管理，如 CreateThread()； 调试（Debug 开头）； 文件操作，包括创建、打开、读写、搜索等； 内存分配（Local 开头和 Global 开头） 其中大多数 API 是被转换为系统服务发给内核态的执行体
USER32.DLL	用户模式	Windows 子系统 DLL 之一，包含窗口管理、消息处理和用户输入 API，如 EndDialog()、BeginPaint()、SetWindowPos()、MessageBox() 等。其中大多数 API 是被转换为系统服务发给内核态的 Windows 子系统驱动程序（Win32K.SYS）

Windows XP 去除了 OS/2 子系统，安装包中也不再包含 POSIX 子系统，但是可以免费下载。以下注册表项包含了系统中各个子系统的情况：

HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\SessionManager\SubSystems

Windows Vista 企业版（Enterprise）带有用于运行 Unix 类应用程序（UNIX-based Applications）的子系统，并提供了免费的 SDK 供用户开发这样的程序。

8.5 本章总结

Windows 是个庞大的系统，本章的介绍仅仅覆盖了 Windows 操作系统中最基本的一些概念。在后面的章节中，我们会根据需要穿插介绍其他有关的系统机制和数据结构，尤其是与软件调试密切相关的内容。下一章我们会比较深入地挖掘 Windows 调试子系统的架构和工作原理。一旦精通了调试技术，就可以使用它来学习探索其他软件问题了。

参考文献

1. Mark E. Russinovich and David A. Solomon. *Windows Internals 4th Edition*. Microsoft Press, 2005
2. Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3B. Intel Corporation

用户态调试模型

应用程序（application program 或 application）是指能够解决某一个问题或满足某一种应用的特定程序，它是相对系统程序而言的。从用户的角度来看，大多数用户使用计算机是使用应用程序，系统程序的作用是为应用程序提供服务。以操作系统为例，用户购买和安装操作系统的目的是主要是在上面运行应用程序。换言之，操作系统存在的意义在于通过应用程序满足用户的使用需要。因此，是否具有丰富的应用程序是操作系统的成功的一个关键。

如何才能有丰富的应用程序呢？除了要有一套强大而且易用的应用程序编程接口（Application Programming Interface，简称 API）外，还需要有高效的开发和调试环境。提高应用程序开发效率的一个重要课题就是提高应用程序调试（application debugging）的效率，因为应用程序调试很多时候耗费了比设计编码还要多的时间。

支持应用程序调试一直是操作系统设计中的一项重要任务。为了与调试运行在内核模式下的驱动程序或操作系统自身的代码相区分，通常把调试应用程序称为用户态调试（user mode debugging）。

本章将介绍 Windows 操作系统中用于支持用户态调试的模型和各种基础设施，包括内核中的调试支持例程、调试子系统和调试 API 等。

需要说明的一点是，本章和下一章所说的调试都是指使用调试器进行的调试，而不是广义上的软件纠错行为。

9.1 概览

回想我们在调试器（比如 VC6）中调试程序（比如一个 HelloWorld 程序）的情景，可以随时将被调试程序中断到调试器中，然后观察变量信息或跟踪执行，仿佛一切都可在调试器中进行。但事实上，从进程的角度来看，VC6 和 HelloWorld 是两个分别运行在调试器中进行。但事实上，从进程的角度来看，VC6 和 HelloWorld 是两个分别运行在调试器中进行。但事实上，从进程的角度来看，VC6 和 HelloWorld 是两个分别运行在调试器中进行。而且根据我们在上一章对进程和进程空间的介绍，

每个进程的空间都是受到严密的系统机制所保护的。那么，调试器进程是如何“轻而易举”地观察和控制被调试进程的呢？简单的回答是使用调试 API。但要深入理解调试 API 是如何工作的，就必须挖掘调试子系统在调试中所起的作用。

事实上，当我们使用 VC6 调试 HelloWorld 进程时，除了 VC6 进程和 HelloWorld 进程，还有一些重要角色积极地参与到这个过程中，正是它们的“努力工作”，才使得调试过程如此得心应手。它们是谁呢？这一节我们就来概览参与用户态调试的各个角色。

9.1.1 参与者

图 9-1 画出了在 Windows 下进行用户态调试时参与调试过程的各个角色，包括调试器进程（Debugger Process）、被调试进程（Debuggee Process）、调试子系统、调试 API，以及位于 NTDLL 和内核中的支持函数。

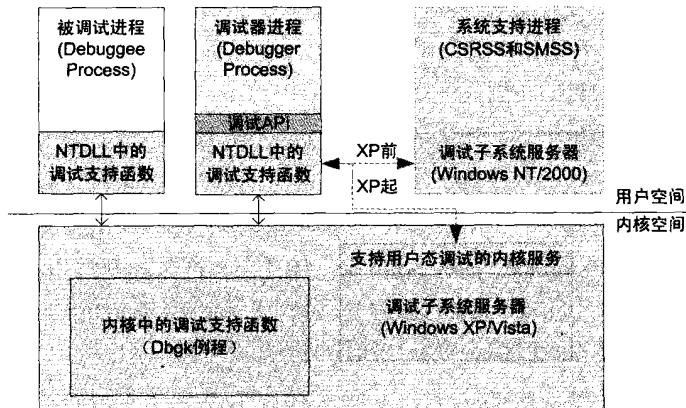


图 9-1 参与用户态调试的各个角色

首先，调试器进程（Debugger Process）是调试过程的主导者，它负责发起调试对话，读取和处理调试事件，并通过用户界面接受调试人员下达的指令，然后执行。调试器进程通过调试 API（9.8 节介绍）与系统的调试支持函数和调试子系统交互，这样不仅简化了调试器的开发工作，而且大大降低了调试器与调试子系统的耦合度。当对调试子系统进行革新的时候，只要保持调试 API 不变就可以保证调试器依然可以工作。

被调试进程（Debuggee Process）是调试的目标。为了降低海森伯效应（将在 28.6 节介绍），应尽可能少地向被调试进程中加入支持调试的设施，以避免影响问题的重现和分析。但某些调试功能需要在被调试进程中做少量标记或执行简单的动作（后文详述）。

调试子系统是沟通被调试进程和调试器进程的桥梁，它的职责是帮助调试器完成各种调试功能，比如控制和访问被调试进程，管理和分发调试事件，接收和处理调试器的服务请求。从内部来看，调试器大多时候是在和调试子系统对话。

9.1.2 调试子系统

调试子系统主要由 3 个部分组成：位于 NTDLL 中的支持函数，位于内核文件中的支持函数，以及调试子系统服务器。

NTDLL.DLL 中的调试支持例程主要分为 3 类。第一类是以 DbgUi 开头的，供调试器使用；第二类是以 DbgSs 开头的，供调试子系统使用，这一部分在 Windows 2000 之后被移除；第三类是以 Dbg 开头（非前两种）的，用于实现调试 API，如 DbgBreakpoint 是 DebugBreak API 的实现。根据上一章对 NTDLL 的介绍，尤其是图 8-5，我们知道，NTDLL.DLL 是所有用户态进程都使用的一个模块，因此放在这个模块中的函数具有非常方便的共享性。这也是要把以上 3 类调试函数放在 NTDLL.DLL 中的一个原因。

内核文件中的调试支持例程负责采集和传递调试事件，以及控制被调试进程。这些内核函数都是以 Dbgk 开头的，我们将在第 9.2 节和第 9.3 节详细介绍它们。

调试子系统服务器的主要职能是管理调试会话和调试事件，是调试消息（事件）的集散地，也是所有调试设施的核心。

调试子系统是操作系统的一个组成部分，其实现因操作系统的不同而不同。对于同一操作系统，不同的版本也可能包含不同的实现。Windows NT(3.x 和 4.0)和 Windows 2000 的调试子系统基本一致。Windows XP 做了较大改进，增加了专门用于应用程序调试的内核对象（Debug Object），并将调试子系统服务器从用户态移入到内核态中，但是在 API 一层仍与以前兼容。Windows Vista 沿用了 Windows XP 改进后的实现。

9.1.3 调试事件驱动

与 Windows 程序的消息驱动机制类似，Windows 用户态调试是通过调试事件来驱动的。调试器程序在与被调试进程建立调试会话（第 10.3 节和 10.4 节详细讨论）后，调试器进程便进入所谓的调试事件循环（Debug Event Loop），等待调试事件的发生，然后处理，再等待，直到调试会话终止，其核心代码如下：

```
while (WaitForDebugEvent(&DbgEvt, INFINITE)) // 等待事件
{
    // 处理等待得到的事件
    // 处理后，恢复调试目标继续执行
    ContinueDebugEvent(DbgEvt.dwProcessId, DbgEvt.dwThreadId,
        dwContinueStatus);
}
```

其中，WaitForDebugEvent 用于等待和接收调试事件，收到调试事件后，调试器便根据事件的类型（事件 ID）来分发和处理，并根据情况决定是否要通知用户并进入交互式调试（命令模式，29.7 节详细讨论）。在处理调试事件的过程中，被调试进程是处于挂起状态的（9.3 节将详细讨论）。处理调试事件后，调试器调用 ContinueDebugEvent 将处理结果回复给调试子系统，让被调试程序继续运行，调试器则再次调用

`WaitForDebugEvent` 等待下一个调试事件。`WaitForDebugEvent` 和 `ContinueDebugEvent` 都是 Windows 提供的调试 API，我们在第 9.8 节详细介绍。

调试子系统的内核部分设计了一系列函数用来采集调试事件，并以一个消息结构发送给调试子系统，使其保存在调试子系统的调试消息队列中。调试子系统和调试器之间是靠一个内核对象来同步的。当有调试消息需要读取时，调试子系统服务器会设置这个同步对象，使等待这个对象的调试器线程被唤醒。

在内核中，调试事件有时也被称为调试消息，并使用一个名为 `DBGKM_APIMSG` 的结构来描述。在发送给调试器时，调试 API 使用的是一个名为 `DEBUG_EVENT` 的结构（参见 10.5.1 节）。因为这两个结构是不同的，所以需要一个转化过程，这个工作是调试子系统服务器和 NTDLL 中的用户态函数来完成的。简单来说，子系统服务器会将自己使用的结构转化为 NTDLL 使用的 `DBGUI_WAIT_STATE_CHANGE`，NTDLL 再将这个结构转化为调试器使用的 `DEBUG_EVENT` 结构。

本节概括性地介绍了用户态调试的所有参与者，以及它们之间是如何以调试事件为纽带而协同工作的。接下来的几节我们将围绕调试事件作进一步介绍。

9.2 采集调试消息

为了能了解到调试有关的系统动作，调试子系统的内核部分对外公开了一系列函数，供内核的其他部分调用，以便得到“知情”和处理机会。这些函数都是以 `Dbgk` 开头的（不是 `Dbgkp`，`p` 代表内部过程），是调试子系统公开给内核其他部件的接口函数，我们将它们简称为 `Dbgk` 采集例程。

9.2.1 消息常量

`Dbgk` 采集例程将所有调试事件（消息）分为 8 种类型，并使用以下常量来代表不同类型的调试消息：

```
typedef enum _DBGKM_APINUMBER
{
    DbgKmExceptionApi = 0,           // 异常
    DbgKmCreateThreadApi = 1,         // 创建线程
    DbgKmCreateProcessApi = 2,        // 创建进程
    DbgKmExitThreadApi = 3,          // 线程退出
    DbgKmExitProcessApi = 4,         // 进程退出
    DbgKmLoadDllApi = 5,             // 映射 DLL
    DbgKmUnloadDllApi = 6,            // 反映射 DLL
    DbgKmErrorReportApi = 7,          // 内部错误
    DbgKmMaxApiNumber = 8,            // 这组常量的最大值
} DBGKM_APINUMBER;
```

其中 `DbgKmErrorReportApi` 用来报告调试子系统内部的错误，目前已经不再使用。下面将分别介绍其他几种调试消息的采集过程。

9.2.2 进程和线程创建消息

操作系统的一大核心任务就是管理系统中运行的各个进程和线程，包括创建新的进程和线程，调度等待运行的线程，进程间通信，终止进程和线程，以及资源分配回收等，这些任务通常被统称为进程管理。操作系统的很多模块都与进程管理有关，但进程管理的最核心部分是由位于 Windows 执行体（executive）（NTOSKRNL 的上半部分，参见图 8-5）中的一系列函数完成的，这些函数大多以 Ps 或 Psp 开头，比如 PsCreateSystemThread()、PspShutdownThread() 等。为了描述简便，通常把这些函数（以及这些函数使用的数据结构）泛称为进程管理器（Process Manager）。

当进程管理器创建新的用户态 Windows 线程时，它首先要为该线程建立必要的内核对象与数据结构，并分配栈（stack）空间，这些工作完成后，该线程就处于挂起状态（CREATE_SUSPEND），而后进程管理器会通知环境子系统，子系统会作必要的设置和登记，最后进程管理器会调用 PspUserThreadStartup 例程，准备启动该线程。为了支持调试，PspUserThreadStartup 总是会调用调试子系统的内核函数 DbgkCreateThread，以便让调试子系统得到处理机会。

DbgkCreateThread 会检查新创建线程所在的进程是否正在被调试（根据 DebugPort 是否为空），如果不是，便立即返回，如果是（DebugPort 不为空），则会继续检查该进程的用户态运行时间（UserTime）是否为 0，目的是判断该线程是否是进程中的第一个线程，如果是，则通过 DbgkpSendApiMessage() 函数向 DebugPort 发送 DbgKmCreateProcessApi 消息，如果不是，则发送 DbgKmCreateThreadApi 消息。调试器收到的进程创建（CREATE_PROCESS_DEBUG_EVENT，值为 3）和线程创建（CREATE_THREAD_DEBUG_EVENT，值为 2）事件就是源于这两个消息的。

9.2.3 进程和线程退出消息

进程管理器的 PspExitThread 函数负责线程的退出和清除。为了支持调试，在销毁该线程的结构和资源之前，PspExitThread 会调用调试子系统的函数以便让调试器（如果有）得到处理机会。如果正在退出的不是进程中的最后一个线程，那么 PspExitThread 会调用 DbgkExitThread 函数通知指定线程要退出，如果是最后一个线程，那么 PspExitThread 便会调用 DbgkExitProcess 函数通知指定进程要退出。

DbgkExitThread 会检查进程的 DebugPort 是否为 0，如果不为 0，则会先将该进程挂起，然后通过 DbgkpSendApiMessage 函数向 DebugPort 发送 DbgKmExitThreadApi 消息，待发送函数返回后再恢复进程运行。

DbgkExitProcess 的执行过程非常类似，只不过发送的是 DbgKmExitProcessApi 消息，而且没必要执行挂起和恢复动作，因为进程管理器已经对该线程做了删除标记。

调试器接收到的线程退出（EXIT_THREAD_DEBUG_EVENT，值为 4）和进程退出

(EXIT_PROCESS_DEBUG_EVENT, 值为 5) 事件是源于这两个消息。

9.2.4 模块映射和反射消息

自 Windows 诞生以来, 动态链接库 (Dynamic-link Library, 简称 DLL) 一直是 Windows 中使用最多的技术之一。Windows 操作系统内核、Windows API 和无以计数的大量 Windows 应用程序都普遍使用了 DLL 技术。比如, Windows 的内核文件 NTOSKRNL.EXE 虽然是以 EXE 为后缀, 其实质就是一个 DLL; NTDLL.DLL 是连接用户态和操作系统内核的桥梁 (参见图 8-5), 用户态的代码通过它访问内核服务; Windows 子系统 DLL (KERNEL32.DLL、ADVAPI32.DLL、USER32.DLL、GDI32.DLL) 是 Windows API 的载体, 当使用 Windows API 的应用程序在执行时都离不开这些 DLL。观察 Windows 的系统目录 (winnt\system32) 还会看到很多其他的 DLL。除了 Windows 自带的 DLL, 应用软件本身也经常有自己的 DLL。DLL 不可以独立拥有进程和运行, 但可以被 EXE 程序加载到其所属的进程空间中, 并被调用和执行。

很多工具可以用来观察进程中的 DLL 使用情况。运行记事本程序, 启动 VC6, 通过 Build>Start Debug>Attach to Process... 菜单弹出 Attach Process 对话框, 然后选择 notepad。而后再通过 Debug>Modules... 菜单弹出如图 9-2 所示的模块列表, 便可以看到 notepad 进程中的 DLL 了。第二列是该模块在进程空间中的地址 (虚拟地址, 均小于 0x80000000, 可见这些模块都是位于用户空间中的)。

Module	Address	Path
ntdll.dll	0x77F50000 - 0x77F6FFF	C:\WINNT\system32\ntdll.dll
notepad.exe	0x01000000 - 0x01012FFF	C:\WINNT\system32\notepad.exe
kernel32.dll	0x77E50000 - 0x77F4FFF	C:\WINNT\system32\kernel32.dll
comdlg32.dll	0x763B0000 - 0x763F4FFF	C:\WINNT\system32\comdlg32.dll
SHLWAPI.dll	0x70A70000 - 0x70AD5FFF	C:\WINNT\system32\SHLWAPI.dll
advapi32.dll	0x77D00000 - 0x77E5CFFF	C:\WINNT\system32\advapi32.dll
rpcrt4.dll	0x78000000 - 0x78086FFF	C:\WINNT\system32\rpcrt4.dll
gdi32.dll	0x7F000000 - 0x7F041FFF	C:\WINNT\system32\gdi32.dll
user32.dll	0x77D40000 - 0x77DCFFF	C:\WINNT\system32\user32.dll
msvcr7.dll	0x77C10000 - 0x77C62FFF	C:\WINNT\system32\msvcr7.dll
comctl32.dll	0x71950000 - 0x71A34FFF	C:\WINNT\WinSxS\x86_Microsoft.Windows.Common-Controls_65e046eef1ce149d_7.0.5072.4000\comctl32.dll
shell32.dll	0x7CD00000 - 0x7D4AFFF	C:\WINNT\system32\shell32.dll
winspool.drv	0x73000000 - 0x73022FFF	C:\WINNT\system32\winspool.drv
imm32.dll	0x76390000 - 0x763ABFFF	C:\WINNT\system32\imm32.dll
lpk.dll	0x629C0000 - 0x629C7FFF	C:\WINNT\system32\lpk.dll
usp10.dll	0x72FA0000 - 0x72FF9FFF	C:\WINNT\system32\usp10.dll
uxtheme.dll	0x5AD70000 - 0x5ADA3FFF	C:\WINNT\system32\uxtheme.dll

图 9-2 使用 VC6 观察进程中的模块

在图 9-2 中可以看到熟悉的 NTDLL.DLL 和 Windows 子系统 DLL, 重复上面的步骤观察其他的 Windows 进程, 通常也会看到这些 DLL。那么, 这些存在于多个进程空间中的 DLL 是不是要重复占用内存呢? 答案是否定的, 当 Windows 的 DLL 加载函数 (LoadLibrary() 和 LoadLibraryEx() API 及未公开的用户态函数和内核 API) 要加载一个 DLL 时, 会首先判断该 DLL 是否已经加载过, 如果是, 则不会重复加载, 只要将该 DLL 对应的内存页面映射 (map) 到目标进程的内存空间, 并把该 DLL 的引用

次数加 1。当一个进程退出或调用 `FreeLibrary()` API 要卸载一个 DLL 时, Windows 会从进程的虚拟内存空间中把该 DLL 的映射删除 (unmap), 并递减该 DLL 的引用次数, 如果引用次数变为 0, 那么该 DLL 会被彻底移出内存。

Windows 内核中的内存管理器 (Memory Manager) 负责 DLL 的映射和反射。在内部, 内存管理器使用 Section 对象 (Windows 子系统称为 file mapping object, 即文件映射对象) 来表示一块可被多个进程共享的内存区域, 并设计了一系列内核服务和函数来实现各种映射和反射任务。使用 WinDBG 的 `x` 命令可以看到这些内核函数。

```
1kd> x nt!NtMapViewOfSection
805a6526 nt!NtMapViewOfSection = <no type information>
805a733c nt!NtUnMapViewOfSection = <no type information>
```

其中 `NtMapViewOfSection` 是用来映射模块的内核服务的, `NtUnMapViewOfSection` 是用来反映射的。

为了支持调试, 当 `NtMapViewOfSection` 在把一个模块映像 (表示为 section 对象) 成功映射到指定进程的空间中时 (使用 `MmMapViewOfSection`), `NtMapViewOfSection` 会调用调试子系统的 `DbgkMapViewOfSection` 函数通知调试子系统。清单 9-1 所示的函数调用序列显示了进程初始化期间加载 DLL 文件的整个过程。

清单 9-1 模块映射过程

```
kd> kn
# ChildEBP RetAddr
00 bcb59abc 80494840 nt!LpcRequestWaitReplyPort // 使用 LPC 发送并等待回复
01 bcb59bdc 8052a3cb nt!DbgkpSendApiMessage+0x43 // 发送调试消息
02 bcb59ca8 804b61a8 nt!DbgkMapViewOfSection+0xe8 // 通知调试子系统
03 bcb59d34 80461691 nt!NtMapViewOfSection+0x333 // 系统服务的内核函数
04 bcb59d34 77f86839 nt!KiSystemService+0xc4 // 系统服务分发例程
05 0006f7dc 77f94020 ntdll!ZwMapViewOfSection+0xb // 调用系统服务
06 0006f870 77f85478 ntdll!LdrpMapDll+0x199 // 加载器的模块映射函数
07 0006f8a4 77f95f18 ntdll!LdrpLoadImportModule+0x62 // 加载依赖的模块
08 0006f8fc 77f8548a ntdll!LdrpWalkImportDescriptor+0x96 // 遍历模块的输入表
09 0006f920 77f95f18 ntdll!LdrpLoadImportModule+0x70 // 加载依赖的模块
0a 0006f978 77f8651e ntdll!LdrpWalkImportDescriptor+0x96 // 遍历模块的输入表
0b 0006fc98 77f96416 ntdll!LdrpInitializeProcess+0x70a
0c 0006fd1c 77f9fb67 ntdll!LdrpInitialize+0x175 // 加载器的初始化函数
0d 00000000 00000000 ntdll!KiUserApcDispatcher+0x7 // 异步过程调用到用户态
```

由下而上, 栈帧 #0c~#06 是 NTDLL 中的映像加载函数, 它们都以 Ldr (Loader) 开头, 是 Windows 系统的加载器函数, #0a 是遍历模块的输入表, #09 准备加载一个模块, #08 是遍历这个要加载模块的输入表, #06 是用户态的 DLL 映射函数, 它内部调用系统服务 `ZwMapViewOfSection`, 后便进入内核态。

`DbgkMapViewOfSection` 被调用后会检查当前进程的 `DebugPort` 是否为空, 如不为空, 则通过 `DbgkpSendApiMessage` 函数发送 `DbgKmLoadDllApi` 消息。

类似的, 当内存管理器反映射一个模块映像时, `MmUnMapViewOfSection` 函数会调用调试子系统的 `DbgkUnMapViewOfSection` 函数。该函数在检测到当前进程的 `DebugPort`

不为空后，会发送 DbgKmUnloadDllApi 消息。

调试器接收到的模块映射 (LOAD_DLL_DEBUG_EVENT, 值为 6) 和反射 (UNLOAD_DLL_DEBUG_EVENT, 值为 7) 事件是源于这两个消息的。

9.2.5 异常消息

异常与调试有着密不可分的关系，很多软件错误是与异常有关的，因此调试器应该能够知道并控制被调试程序中的异常。另外，很多调试机制是以异常机制为基础的，比如断点和单步执行分别是依靠断点异常 (#BP) 和调试异常 (#DB) 来工作的。

为了支持调试，系统会把被调试程序中发生的所有异常发送给调试器。我们将在第 11 章详细介绍异常的分发过程，在此我们只需要知道内核中的 KiDispatchException 函数是分发异常的枢纽，它会给每个异常安排最多两轮被处理的机会，对于每一轮处理机会，它都会调用调试子系统的 DbgkForwardException 函数来通知调试子系统。

DbgkForwardException 函数既可以向进程的异常端口发送消息，也可以向调试端口发送消息，KiDispatchException 函数在调用它时会通过一个布尔类型的参数来指定。如果要向调试端口发送消息，那么 DbgkForwardException 函数会判断进程的 DebugPort 字段是否为空，如果不为空，便通过 DbgkpSendApiMessage 函数发送 DbgKmExceptionApi 消息。

调试器收到的异常事件 (EXCEPTION_DEBUG_EVENT, 值为 1) 和输出调试字符串 (OUTPUT_DEBUG_STRING_EVENT, 值为 8) 事件，都是源于 DbgKmExceptionApi 消息的。我们将在第 10.7 节详细介绍输出调试字符串的细节。

前面分别介绍了各种调试消息的采集过程。简单来说，系统的进程管理器、内存管理器和异常分发函数会调用调试子系统的 Dbgk 采集例程，来向调试子系统通报调试消息，这些例程被调用后会根据当前进程的 DebugPort 字段来判断当前进程是否处于被调试状态。如果不是，便忽略这次调用，直接返回；如果是，便产生一个 DBGKM_APIMSG 结构，然后调用下一节将介绍的 DbgkpSendApiMessage 函数来发送调试消息。

9.3 发送调试消息

上一节介绍了调试子系统的内核函数采集调试事件的方法和过程。本节将继续介绍调试消息发送给调试子系统服务器的过程。

9.3.1 调试消息结构

首先，调试子系统的内核函数使用以下结构来描述和传递调试消息：

```

typedef struct _DBGKM_APIMSG
{
    PORT_MESSAGE h;                                //LPC 端口消息结构, XP 之前使用
    DBGKM_APINUMBER ApiNumber;                     //消息类型
    NTSTATUS ReturnedStatus;                       //调试器的回复状态
    union {
        DBGKM_EXCEPTION Exception;                 //异常
        DBGKM_CREATE_THREAD CreateThread;          //创建线程
        DBGKM_CREATE_PROCESS CreateProcessInfo;     //创建进程
        DBGKM_EXIT_THREAD ExitThread;              //线程退出
        DBGKM_EXIT_PROCESS ExitProcess;            //进程退出
        DBGKM_LOAD_DLL LoadDll;                   //映射 DLL
        DBGKM_UNLOAD_DLL UnloadDll;                //反映射 DLL
    } u;
} DBGKM_APIMSG, *PDBGKM_APIMSG;

```

其中, `ApiNumber` 就是我们上一节介绍过的枚举常量, 用来表示消息的类型。`ReturnedStatus` 用来存放调试器的回复信息。联合结构 `u` 的内容会因为消息类型的不同而不同, 用来描述消息的参数和详细信息。例如, 当 `ApiNumber` 等于 `DbgKmExceptionApi(0)` 时, 联合部分是一个 `DBGKM_EXCEPTION` 结构, 其余的依此类推。

调试消息采集函数在确认需要向调试子系统报告消息后, 它会填写 `DBGKM_APIMSG` 结构, 然后将其作为参数传给 `DbgkpSendApiMessage` 函数。

9.3.2 DbgkpSendApiMessage 函数

`DbgkpSendApiMessage` 函数是用来将一个调试消息发送到调试子系统服务器的。

```

NTSTATUS DbgkpSendApiMessage(
    IN OUT PDBGKM_APIMSG ApiMsg,
    IN PVOID Port,
    IN BOOLEAN SuspendProcess)

```

其中 `ApiMsg` 用来描述消息的详细信息, `Port` 用来指定要发往的端口, 大多数时候就是 `EPROCESS` 结构中的 `DebugPort` 字段的值, 偶尔是进程中的异常端口, 即 `ExceptionPort` 字段。

如果 `SuspendProcess` 为真, 那么这个函数会先调用下面将要介绍的 `DbgkpSuspendProcess` 函数挂起当前进程, 然后发送消息, 等收到消息回复后再调用 `DbgkpResumeProcess` 函数唤醒当前进程。

因为 Windows NT 和 Windows 2000 的调试子系统服务器位于用户态, 所以在这些系统中, `DbgkpSendApiMessage` 会通过 `LPC` 机制来发送调试消息。这时 `Port` 参数指定的是一个 `LPC` 端口, 这个端口的监听者通常是 Windows 环境子系统的服务器进程, 即 `CSRSS`。`CSRSS` 收到消息后再发给位于会话管理器进程中的调试子系统服务器, 后者再通知等候调试事件的调试器。因为 `DbgkpSendApiMessage` 内部是调用 `LpcRequestWaitReplyPort` 函数来完成具体的 `LPC` 收发任务的, 而且 `LpcRequestWaitReplyPort` 函数是阻塞的, 所以只有在收到回复后, `LpcRequestWaitReplyPort` 才会返回。

在 Windows XP 及其后续的 Windows 中, 调试子系统服务器被移到内核空间中,

因此这些版本中的 `DbgkpSendApiMessage` 改为通过调用 `DbgkpQueueMessage` 来发送消息。`DbgkpQueueMessage` 会根据参数决定是否需要等待，如果需要，它会调用等待函数，直到收到调试器的回复后才返回。不需要等待的情况只用于发送杜撰消息（将在下一节介绍）等特殊情况，因此如不特别说明，我们讨论的都是需要等待的情况。

9.3.3 控制被调试进程

调试子系统设计了两个内核函数来控制被调试进程，它们是 `DbgkpSuspendProcess` 和 `DbgkpResumeProcess`。

在调试子系统向调试器发送调试事件之前，通常会先调用 `DbgkpSuspendProcess` 函数。这个函数会内部调用 `KeFreezeAllThreads()` 函数冻结（Freeze）被调试进程中除调用线程之外的所有线程。进一步讲，在调用 `DbgkpResumeProcess` 函数后，被调试进程中便只有当前线程（发送调试消息的这个线程）还活动着，接下来它会执行实际的消息发送函数，在 Windows XP 之前是调用 LPC 函数，从 XP 开始是调用 `DbgkpQueueMessage` 函数。无论哪个函数，对于大多数调试事件，它们都是堵塞的，也就是会调用等待函数（`KeWaitForSingleObject` 等）来使当前线程进入等待状态。因为当前进程的其他线程已被此前的 `DbgkpSuspendProcess` 调用所挂起。所以，一旦当前线程进入等待，那么整个被调试进程的所有线程便都不再执行。这可以解释为什么当被调试进程被中断到调试器时，被调试程序没有响应。

接下来的任务就是调试子系统服务器通知调试器来读取调试消息，作处理后回复给调试子系统，后者再唤醒被调试进程的等待线程（发送调试消息的那个），等待线程醒来后，再执行 `DbgkpResumeProcess` 函数，后者内部会调用 `KeThawAllThreads()` 恢复（Unfreeze）被调试进程中的所有线程。

从数据结构的角度来看，在每个 Windows 线程的 KTHREAD 结构中，有两个与线程执行状态密切相关的字段，一个叫 `FreezeCount`，一个叫 `SuspendCount`。对于可调度执行的线程来说，这两个字段的值都为 0。`KeFreezeAllThreads()` 函数和 `KeThawAllThreads()` 操作的是 `FreezeCount` 字段，而 `SuspendThread()` 和 `ResumeThread()` API（对应于 `NtSuspendThread` 内核服务和 `KeSuspendThread`）操作的是 `SuspendCount` 字段。

当被调试进程中断到调试器中时，它的当前线程的 `FreezeCount` 通常为 0，其他线程的 `FreezeCount` 通常为 1，因为 `KeFreezeAllThreads` 不会冻结当前线程，包括 WinDBG 在内的调试器在收到调试事件后，会对被调试进程中的所有线程依次调用 `SuspendThread` API，这样一来，所有线程的 `SuspendCount` 计数通常都为 1。例如，当使用 WinDBG 调试包含两个线程的 `MulThreads` 程序时，当设置在 `kernel32!SleepEx` 处的断点命中后，使用本地内核调试会话可以观察到 `MulThreads` 进程内的所有线程的详细信息，如清单 9-2 所示（为节约篇幅，格式进行调整过，并增加行号）。

清单 9-2 观察中断到调试器中的被调试线程

```

1 lkd> !PROCESS 88a3a600 2    ** 2 代表只显示线程状态信息
2 PROCESS 88a3a600 SessionId: 0 ...      Image: MulThrds.exe
3 THREAD 881fb020 Cid 1e40.1e44 Peb: 7ffd000 Win32Thread: e4ff3eb0 WAIT:
4 (Suspended) KernelMode Non-Alertable SuspendCount 1 FreezeCount 1
5             881fb1bc Semaphore Limit 0x2
6 THREAD 87716ce0 Cid 1e40.1e48 Peb: 7ffde000 Win32Thread: e11a1af8 WAIT:
7 (Executive) KernelMode Non-Alertable SuspendCount 1
8             a9a1d7d4 SynchronizationEvent

```

从第 4 行可以看到，线程 1e44 的 SuspendCount 和 FreezeCount 都是 1，从第 7 行看，线程 1e48 的 SuspendCount 为 1，FreezeCount 为 0（大于 0 才显示）。这是因为线程 1e48 中发生了断点异常，调试事件的发送过程是发生在这个线程中的，所以当 KeFreezeAllThreads 执行时，没有冻结这个线程。WinDBG 的断点和调试异常处理函数（ProcessBreakpointOrStepException）会对所有线程调用 SuspendThread API，因此，这两个线程的 SuspendCount 都为 1。

WinDBG 的~n 和~m 命令允许用户调整被调试线程的 SuspendCount，这两个命令实际上调用系统的是 SuspendThread 和 ResumeThread API。

我们前面说过，Windows XP 对调试子系统作了重大改变，特别是子系统服务器部分。因此，Windows XP 之前和之后（包括 XP）的调试子系统服务器是不同的。接下来的两节我们将分别介绍这两种子系统服务器。

9.4 调试子系统服务器（XP 之后）

与 Windows 2000 和 NT 相比，Windows XP 对用户态调试子系统作了重大改进，将调试子系统服务器由用户态移入内核态，Windows Vista 沿用了这一改动。新的子系统服务器是以新引入的内核对象 DebugObject 为核心的。本节将围绕这个内核对象介绍 Windows XP 和 Windows Vista 的调试子系统服务器。

9.4.1 DebugObject

DebugObject 内核对象是专门用于用户态调试的，它不仅承担了同步调试器和调试子系统的功能，而且也是调试器和调试子系统之间传递数据的重要纽带，取代了调试子系统各部件间本来使用的 LPC 通信方式。以下是 DebugObject 对象的内部结构：

```

typedef struct _DEBUG_OBJECT
{
    KEVENT EventsPresent;           //+0x00 用于指示有调试事件发生的事件对象
    FAST_MUTEX Mutex;              //+0x10 用于同步的互斥对象
    LIST_ENTRY StateEventListEntry; //+0x30 保存调试事件的链表
    ULONG Flags;                  //+0x38 标志，见下文
} DEBUG_OBJECT, *PDEBUG_OBJECT;

```

其中最值得关注的就是 StateEventListEntry 字段，它是一个用来存储调试事件的

链表，我们将其称为调试消息队列。EventsPresent 用来同步调试器进程和被调试进程，调试子系统服务器通过设置此事件来通知调试器读取消息队列中的调试消息。调试器进程通过 WaitForDebugEvent API 来等待调试事件，而 WaitForDebugEvent API 对应的 NtWaitForDebugEvent 内核服务内部实际上等待的就是这个 EventsPresent 对象。互斥对象（Mutex）用来锁定对这个数据结构的访问，以防止多个线程同时读写造成数据错误。Flags 字段包含多个标志位，比如，位 1 代表结束调试会话时是否终止被调试进程（KillProcessOnExit），DebugSetProcessKillOnExit API 实际上设置的就是这个标志位。

9.4.2 创建调试对象

内核服务 NtCreateDebugObject 用来创建调试对象。当调试器与调试子系统建立连接时（将在第 10.3 和 10.4 节介绍），调试子系统会为其创建一个调试对象，并将其保存在调试器当前线程的线程环境块的 DbgSsReserved[1] 字段中。DbgSsReserved[1] 字段中保存的调试对象是这个调试器线程区别于其他普通线程的重要标志，详见 10.1.3 节。

9.4.3 设置调试对象

调试对象通常是在调试器进程中创建的，为了起到联系被调试进程和调试进程的作用，需要将其设置到被调试进程的 EPROCESS 结构的 DebugPort 字段中。

建立应用程序调试对话有两种典型的情况：一种是在调试器中启动被调试程序；另一种是把调试器附加到一个已经运行的进程中。对于前一种情况，系统在创建进程时，会把调试器线程 TEB 结构的 DbgSsReserved[1] 字段中保存的调试对象句柄传递给创建进程的内核服务，然后内核中的进程创建函数会将这个句柄所对应的对象指针赋给新创建进程的 EPROCESS 结构的 DebugPort 字段。我们前面介绍过，DebugPort 字段是系统判断一个进程是否正在被调试的标志。收集调试消息的 Dbgk 函数就是通过判断 DebugPort 字段决定是否要产生和发送调试消息的。对于后一种情况，系统会调用内核中的 DbgkpSetProcessDebugObject 函数来将一个创建好的调试对象附加到其参数所指定的进程中，也就是要被调试的进程。

DbgkpSetProcessDebugObject 函数内部除了将调试对象赋给 EPROCESS 结构的 DebugPort 字段外，还会调用 DbgkpMarkProcessPeb 函数设置进程环境块（PEB）的 BeingDebugged 字段。

9.4.4 传递调试消息

DbgkpQueueMessage 函数用于向一个调试对象的消息队列中追加调试事件。指定调试对象的方法有两个：一是直接在参数中指定调试对象；二是指定 EPROCESS 结构，DbgkpQueueMessage 函数会使用这个结构中的 DebugPort 字段所代表的调试对象。

调试消息队列的每个节点是一个名为 DEBUG_EVENT 的数据结构，其名称与调试 API 中的 DEBUG_EVENT 结构同名，但内容完全不同，为了避免混淆，本书将内核中的 DEBUG_EVENT 结构称为 DBGKM_DEBUG_EVENT。根据参考文献 1，DBGKM_DEBUG_EVENT 结构的定义如下：

```
typedef struct _DBGKM_DEBUG_EVENT
{
    LIST_ENTRY EventList;           //与兄弟节点相互链接的节点结构
    KEVENT ContinueEvent;          //用于等待调试器回复的事件对象
    CLIENT_ID ClientId;            //调试事件所在线程的线程 ID 和进程 ID
    PEPROCESS Process;              //被调试进程的 EPROCESS 结构地址
    PETHREAD Thread;               //被调试进程中触发调试事件的线程的 ETHREAD 地址
    NTSTATUS Status;                //对调试事件的处理结果
    ULONG Flags;                   //标志
    PETHREAD BackoutThread;        //产生杜撰消息 (faked messages) 的线程
    DBGKM_MSG ApiMsg;              //调试事件的详细信息
} DBGKM_DEBUG_EVENT, *PDBGKM_DEBUG_EVENT;
```

其中 ClientId 字段是一个 CLIENT_ID 结构，包含两个 DWORD，分别代表调试事件所属的进程 ID 和线程 ID。

在把 DBGKM_DEBUG_EVENT 结构赋值后，DbgkpQueueMessage 函数会将其插入到调试对象的消息链表中 (StateEventListEntry)。

而后 DbgkpQueueMessage 函数会根据参数中是否指定了不需等待 (NOWAIT，值为 2) 的标志，决定是否要立刻通知调试器来读取消息。如果指定了，便返回。如果没有指定，便设置调试对象的 EventsPresent 对象，通知调试器有消息需要读取，然后调用 KeWaitForSingleObject 等待 DEBUG_EVENT 结构中的 ContinueEvent 对象，等待调试器的回复。

在调试器处理好调试事件后，它会通过 ContinueDebugEvent API 间接调用间接调用或直接调用 nt!NtDebugContinue 内核服务。NtDebugContinue 会根据参数中指定的 CLIENT_ID 结构找到要恢复的调试事件结构，然后设置它的 ContinueEvent 事件对象，使处于等待的被调试线程被唤醒而继续执行。

清单 9-3 所示的函数调用序列记录了断点异常的分发和发送过程，从触发异常开始 (#07)，到放入调试事件队列 (#01)，再到设置 EventsPresent 对象 (#00)。该线程便是所谓的 RemoteBreakin 线程 (10.6.4 节将详述)，是调试器 (WinDBG) 在被调试进程中创建的，用于产生断点异常，以响应中断到调试器 (break) 的命令。

清单 9-3 发送调试事件（被调试进程）

kd> kn	# ChildEBP RetAddr	
00	f5fb87bc 805bd170 nt!KeSetEvent+0x1	[设置事件]
01	f5fb8894 805bc37a nt!DbgkpQueueMessage+0x13f	[放入调试事件链表]
02	f5fb88b4 805bc505 nt!DbgkpSendApiMessage+0x43	[格式化为消息结构]
03	f5fb8940 804feb8a nt!DbgkForwardException+0x8d	[转发给调试子系统]
04	f5fb8cf4 804dab0e nt!KiDispatchException+0x150	[异常分发]
05	f5fb8d5c 804db119 nt!CommonDispatchException+0x4d	[建立异常结构]
06	f5fb8d5c 77f767ce nt!KiTrap03+0x97	[执行 INT3 异常的处理例程]

```
07 0079ffc8 77f7285c ntdll!DbgBreakPoint+0x1      [执行 INT 3 指令, 产生断点异常]
08 0079ffff4 00000000 ntdll!DbgUiRemoteBreakin+0x36 [调用 DbgUi 的中断函数]
```

在第 9.1 节中我们简要介绍过, 建立调试会话后, 调试器工作线程便进入调试事件循环, 等待调试事件, 这实际上就是调用 NtWaitForDebugEvent 内核服务等待调试对象中的 EventsPresent 对象。因此, 当被调试进程中设置了这个对象时, 调试器的工作线程就会被唤醒, 并开始读取调试对象中的消息队列 (StateEventListEntry)。读到一个调试事件后, NtWaitForDebugEvent 会调用 DbgkpConvertKernelToUserStateChanged 函数将 DBGKM_DEBUG_EVENT 结构转换为用户态使用的 DBGUI_WAIT_STATE_CHANGE 结构。清单 9-4 显示了调试器线程等待和读取调试事件的完整过程, 从线程启动 (BaseThreadStart) 到进入调试消息循环 (EngineLoop), 到等待调试事件 (ZwWaitForDebugEvent), 再到得到通知去读取和转换调试事件 (DbgkpConvertKernelToUserStateChanged)。

清单 9-4 读取调试事件 (调试器线程)

```
kd> k
ChildEBP RetAddr
f9afac80 805bce25 nt!DbgkpConvertKernelToUserStateChanged [读取调试事件]
f9afad4c 804da140 nt!NtWaitForDebugEvent+0x1b8   [NtWaitForDebugEvent 内核服务]
f9afad4c 7ffe0304 nt!KiSystemService+0xc4        [内核服务分发]
00a1fd00 77f766fc SharedUserData!SystemCallStub+0x4  [系统调用]
00a1fd04 02242d3e ntdll!ZwWaitForDebugEvent+0xc  [调用等待调试事件的内核服务]
00a1fd00 02107d23 dbgeng!LiveUserDebugServices::WaitForEvent+0x12e
00a1ff10 020a3c3f dbgeng!LiveUserTargetInfo::WaitForEvent+0x3b3
00a1ff34 020a401e dbgeng!WaitForAnyTarget+0x5f    [依次等待每个调试目标]
00a1ff80 020a4290 dbgeng!RawWaitForEvent+0x2ae   [调试引擎的内部函数]
00a1ff98 0102925f dbgeng!DebugClient::WaitForEvent+0xb0  [调试引擎的等待接口]
00a1ffb4 77e7d33b WinDBG!EngineLoop+0x13f     [调试循环]
00a1ffec 00000000 kernel32!BaseThreadStart+0x37  [调试线程启动]
```

读取一个调试事件后, NtWaitForDebugEvent 会在这个事件 DBGKM_DEBUG_EVENT 结构的 Flags 字段中设置一个已读标志。

如上文曾提到的, 在调试器处理好一个调试事件后, 它会调用 ContinueDebugEvent API 让被调试进程继续运行。这个 API 内部会调用 NtDebugContinue 内核服务。这个内核服务会遍历调试对象的消息队列, 找到匹配的调试事件后调用 DbgkpWakeTarget 函数, 来设置 ContinueEvent 对象唤醒等待着的被调试线程。

9.4.5 杜撰的调试消息

当将调试器附加到一个已经运行的进程时, 为了向调试器报告以前发生的但目前仍有意义的调试事件, 调试子系统会“捏造”一些调试消息来模拟过去的调试事件, 这样的调试消息被称为杜撰的调试消息 (Faked Debug Messages)。

NtDebugActiveProcess 是用来与已经运行的进程建立调试会话的内核服务。它在调用 DbgkpSetProcessDebugObject 将调试对象设置到要调试的进程前, 会调用调试子系统的 DbgkpPostFakeProcessCreateMessages 函数。

`DbgkpPostFakeProcessCreateMessages` 会先调用 `DbgkpPostFakeThreadMessages`, 后者会遍历被调试进程的所有线程, 以向调试对象的消息队列中投放杜撰的进程和线程来创建消息。而后 `DbgkpPostFakeProcessCreateMessages` 会调用 `DbgkpPostFakeModuleMessages` 来投放杜撰的模块加载消息。`DbgkpPostFakeThreadMessages` 和 `DbgkpPostFakeModuleMessages` 都是调用 `DbgkpQueueMessage` 来向消息队列添加调试消息的, 因为在参数中指定了不需等待的标志 (NOWAIT), 所以 `DbgkpQueueMessage` 将事件放入队列后便会返回, 不会设置 `EventsPresent` 对象以避免它通知调试器来读取。

在 `DbgkpPostFakeProcessCreateMessages` 返回后, `NtDebugActiveProcess` 会调用 `DbgkpSetProcessDebugObject` 函数来将调试对象设置到要调试的进程。`DbgkpSetProcessDebugObject` 内部在成功设置调试对象后, 会遍历事件队列中的所有事件, 并会设置调试对象的 `EventsPresent` 字段。这样, 当 `NtDebugActiveProcess` 服务返回后, 调试器再调用 `NtWaitForDebugEvent` 时, 它便可以立刻等待成功并读取到事件队列中的调试事件。清单 9-5 所示的栈回溯序列, 显示了向消息队列中投递杜撰调试消息的执行过程。

清单 9-5 添加杜撰的调试消息

```
kd> kn
# ChildEBP RetAddr
00 f50cbc18 805bd26b nt!DbgkpQueueMessage          [放入消息队列]
01 f50cbcec 805bc143 nt!DbgkpPostFakeThreadMessages+0x155 [杜撰线程创建消息]
02 f50cbd30 805bcf2c nt!DbgkpPostFakeProcessCreateMessages+0x2a
03 f50cbd54 804da140 nt!NtDebugActiveProcess+0x8d      [调试已经运行进程]
04 f50cbd54 7ffe0304 nt!KiSystemService+0xc4        [系统服务分发函数]
05 00a1fc4 00000000 SharedUserData!SystemCallStub+0x4    [系统调用]
```

值得注意的是, 以上过程是在调试器进程中执行的。清单 9-3 所示的对 `DbgkpQueueMessage` 的调用是发生在被调试进程中的。

9.4.6 清除调试对象

当调试结束需要撤销调试会话时, 系统会调用 `DbgkClearProcessDebugObject` 将被调试进程的 `DebugPort` 字段恢复为 NULL。在恢复时, 这个函数会遍历调试对象的消息队列, 将关于这个进程的调试事件清除。这个函数并不破坏调试对象, 这是因为一个调试器可以同时调试多个被调试进程, 这个调试对象可能还在被其他被调试进程所使用。

9.4.7 内核服务

配合调试子系统的改变, Windows XP 引入了 8 个新的内核服务。表 9-1 列出了这些服务的名称和主要功能。

表 9-1 Windows XP 中支持用户态调试的内核服务

服务名称	描述
NtCreateDebugObject	创建调试对象
NtRemoveProcessDebug	分离调试对话
NtDebugActiveProcess	与已经运行的进程建立调试会话
NtSetInformationDebugObject	设置调试对象的属性
NtDebugContinue	回复调试事件，恢复被调试进程
NtWaitForDebugEvent	等待调试事件
NtQueryDebugFilterState	查询调试信息输出（debug output）的过滤级别
NtSetDebugFilterState	设置调试信息输出（debug output）的过滤级别

以上内核服务大多是通过调试 API 提供给运行在用户态的调试器程序的，我们将在第 9.8 节介绍调试 API。

9.4.8 全景

图 9-3 画出了 Windows XP 及 Windows Vista 用户态调试子系统的完整模型。图中虚线矩形框代表调试对象，其中双向链表是用来临时存储调试事件的消息队列，即 Debug Object 的 StateEventListEntry 字段。小旗帜代表调试对象中的 EventsPresent 事件对象。

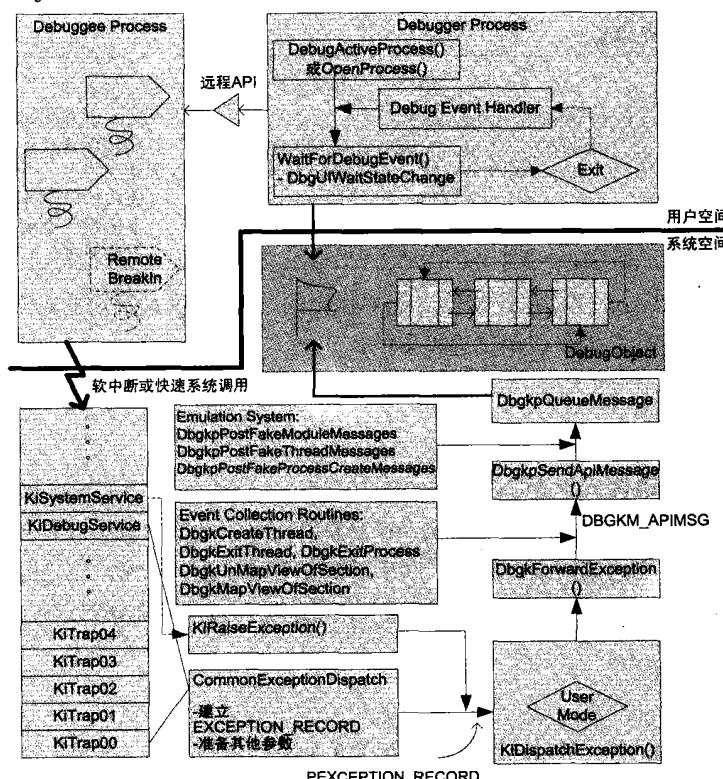


图 9-3 Windows XP 和 Vista 的用户态调试模型

图中左侧的表格代表的是IDT表，靠下方的异常分发过程将在第11章中详细讨论。图中还列出了部分Dbgk例程的名字。表9-2归纳了所有Dbgk例程的名称和用途，第3列表示该函数是否是Windows XP引入的。

表9-2 支持用户态调试的内核函数(Dbgk)

No	名称	XP	描述
1	nt!DbgkCreateThread	否	采集线程创建事件
2	nt!DbgkClearProcessDebugObject	是	将调试对象从指定进程中分离
3	nt!DbgkpConvertKernelToUserStateChange	是	供 NtWaitForDebugEvent 使用，将DBGKM_DEBUG_EVENT 结构转换为DBGUL_WAIT_STATE_CHANGE 结构
N/A	nt!DbgkDebugObjectType	是	调试对象类型 (type) 的全局指针
4	nt!DbgkpMarkProcessPeb	是	当建立和解除调试会话时修改被调试进程 PEB 的 BeingDebugged 字段
5	nt!DbgkpSetProcessDebugObject	是	当建立调试会话时将调试对象写入被调试进程 (EPROCESS) 的 DebugPort 字段
6	nt!DbgkMapViewOfSection	否	采集模块映射事件
7	nt!DbgkExitProcess	否	采集进程退出事件
8	nt!DbgkOpenProcessDebugPort	是	访问指定进程 DebugPort 字段指定的调试对象
9	nt!DbgkpWakeTarget	是	设置 ContinueEvent 对象，唤醒等待调试器回复的线程
10	nt!DbgkpQueueMessage	是	向调试事件队列中加入消息
11	nt!DbgkpResumeProcess	否	恢复执行被调试进程
12	nt!DbgkpOpenHandles	是	打开进程、线程对象，增加应用计数
N/A	nt!DbgkInitialize	是	初始化调试对象，在系统启动早期被调用
13	nt!DbgkpFreeDebugEvent	是	释放调试事件
14	nt!DbgkUnMapViewOfSection	否	采集模块反映射 (unmap) 事件
15	nt!DbgkForwardException	否	向调试子系统通报异常
16	nt!DbgkpPostFakeProcessCreateMessages	是	向调试子系统发送杜撰的进程创建消息
17	nt!DbgkpPostFakeThreadMessages	是	向调试子系统发送杜撰的线程创建消息
18	nt!DbgkpSendApiMessageLpc	是	主要用于向当前进程的异常端口 (ExceptionPort) 发送异常的第二轮处理机会
19	nt!DbgkpCloseObject	是	关闭调试对象，会枚举系统内的所有进程，如果发现某个进程的 DebugPort 字段的值与要关闭的对象相同，则将其置为 0
20	nt!DbgkpPostFakeModuleMessages	是	向调试子系统发送杜撰的模块消息
21	nt!DbgkpDeleteObject	是	目前没有使用
22	nt!DbgkpSendApiMessage	否	发送调试事件，在XP中，调用DbgkpQueueMessage

续表

No	名称	XP	描述
23	nt!DbgkExitThread	否	采集线程退出事件
N/A	nt!DbgkpProcessDebugPortMutex	是	全局的互斥量对象，用于保护对EPROCESS结构中的DebugPort字段的访问
24	nt!DbgkCopyProcessDebugPort	是	当创建新的进程时，根据需要将父进程的DebugPort对象复制给新的进程
25	nt!DbgkpSectionToFileHandle	否	取得Section对象对应的文件句柄
26	nt!DbgkpSuspendProcess	否	挂起被调试进程

表 9-2 中很多新引入的函数所实现的功能在 Windows XP 之前是在用户态实现的。比如，DbgkpPostFakeXXXMessages 函数所实现的投递杜撰消息功能是在 CSRSS 进程中实现的。下一节将详细介绍 XP 之前的调试子系统服务器。

9.5 调试子系统服务器（XP 之前）

本节将介绍 Windows 2000 和 Windows NT 中的调试子系统服务器，我们将其简称为 XP 之前的调试子系统服务器。与上一节介绍的 Windows XP 和 Windows Vista 的调试子系统服务器相比，XP 之前的调试子系统服务器有两个显著特征：在用户态实现，使用 LPC（Local Procedure Call）机制传递调试事件。

为了表达的简洁性，如不特别指出，本节下文中的 Windows 就是指 Windows 2000 或 Windows NT（3.1~4.0）。

9.5.1 概览

图 9-4 画出了 Windows 2000/NT 的用户态调试模型，参与调试的所有成员，也简单地表示了这些角色之间的通信和协作关系，图中的圆柱代表的是 LPC 调用。

在详细介绍每个角色之前，我们先简单介绍如下。

- **调试子系统内核例程：**调试子系统的内核部分，负责采集异常调试事件，以及控制（如挂起和恢复）被调试进程。这一部分与上一节介绍的 Windows XP 开始的情况是一样的。
- **会话管理器（Session Manager, SMSS.EXE）进程：**如果把调试器看作是请求调试服务的客户（client），那么 SMSS 便是提供服务的服务器。调试器通过 LPC 端口与 SMSS 通信，发送请求和接收调试事件。
- **Windows 环境子系统服务器进程（CSRSS.EXE）：**尽管调试器是与 SMSS 直接通信的，但是 SMSS 通常并不真正处理请求，而是把请求转发给相应的子系统服务器进程。CSRSS 便是 Windows 子系统的服务器进程。

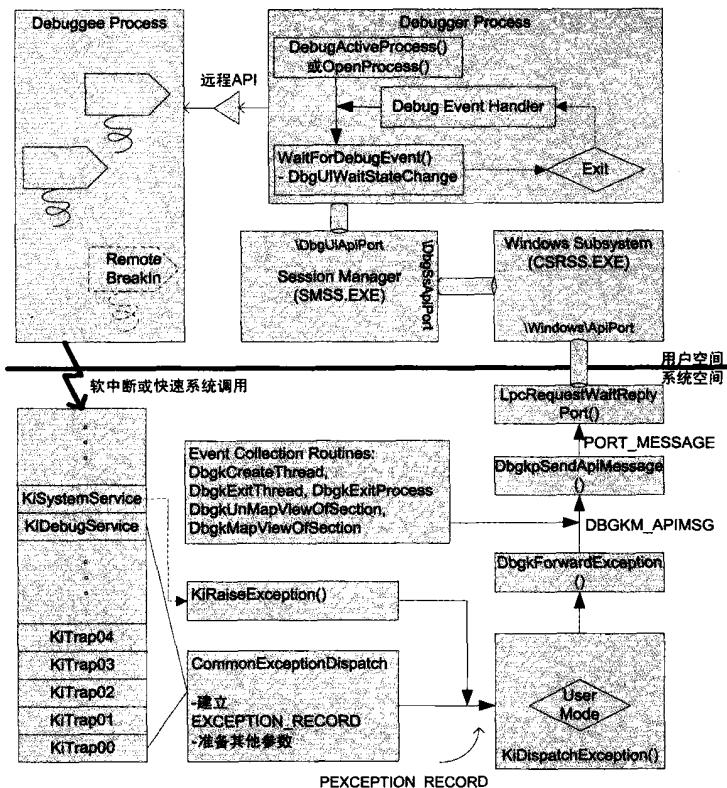


图 9-4 Windows 2000/NT 的用户态调试模型

在对参与调试的各个角色有了基本印象后，下面将对其分别作介绍。

9.5.2 Windows 会话管理器

会话管理器（Session Manager, SMSS.EXE）是 Windows 系统启动后创建的第一个用户态进程。它负责启动和监护 Windows 子系统服务器进程（CSRSS.EXE）和 WinLogon 进程，它对系统的正常运行起着重要的作用。SMSS 在用户态调试中也占据着核心地位（Windows XP 之前）。它负责创建和维护调试子系统与调试器进行通信的 LPC 端口，是调试子系统的服务器（Server）。

图 9-5 所示的注册表表项定义了系统中的各个环境子系统，每次启动时，SMSS 就是根据这里的定义决定加载哪些子系统的。

HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\SessionManager\SubSystems

从图 9-5 中可以看到，除了 Windows、Posix 等定义子系统进程的表项外，里面还有一个 Required 项。

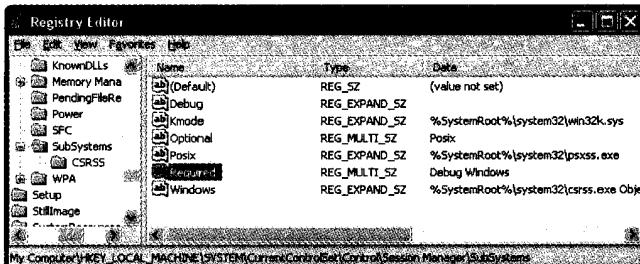


图 9-5 环境子系统定义

Required 项的类型是 REG_MULTI_SZ，即可以包含多个字符串。通常这里的定义都是 Debug Windows。毋庸置疑，Windows 代表的是要 SMSS 启动 Windows 子系统。那么 Debug 项的含义是什么呢？答案是建立调试子系统，更准确地说是建立用户态调试子系统的服务器，包括 LPC 端口、链表结构和服务线程等，分别叙述如下。

第一，\DbgUiApiPort 端口，这是调试子系统与调试器之间的通信通道。当调试器建立调试会话时，不论其使用的是 OpenProcess(..., DEBUG_PROCESS, ...) 还是 DebugActiveProcess() API，其内部都会调用 DbgUiConnectToDbg() 函数（位于 NTDLL.DLL 中）。DbgUiConnectToDbg 函数的操作步骤之一，就是使用 NtConnectPort 函数与 \DbgUiApiPort 端口连接。

第二，\DbgSsApiPort 端口，这是调试子系统与各个环境子系统进行通信的 LPC 端口。SMSS 接收到来自该端口的事件并对其进行过滤后通过\DbgUiApiPort 端口分发给等待在那里的调试器进程。通常环境子系统服务器会连接该端口，以便把本系统的调试事件发给 SMSS（调试子系统服务器进程）。因此可以说，\DbgSsApiPort 端口是 SMSS 与环境子系统服务器进程之间的联系通道。

第三，\SmApiPort 端口，该端口是 SMSS 对外提供服务的 LPC 通道。SMSS 用它来接收系统内的其他进程发给它的各种服务请求（API）。在调试方面，当需要调试环境子系统进程（比如 CSRSS）自身时，需要使用该端口（稍后继续讨论）。使用 Process Explorer 工具可以观察到 SMSS 进程中的各个 LPC 端口（见图 9-6）。

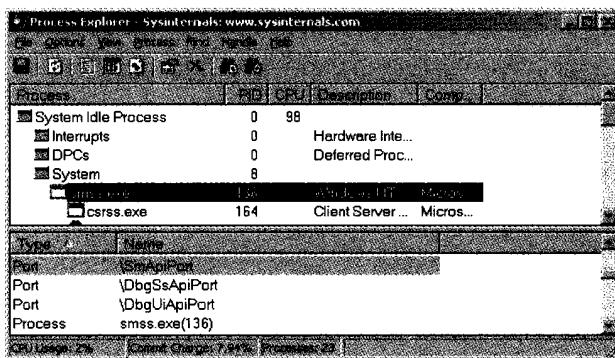


图 9-6 SMSS 进程及它创建的 LPC 端口（Windows 2000）

第四，监听 \DbgUiApiPort 端口的 DbgUiThread 的线程，DbgUiThread 接收来自调试器的消息请求，包括：

- 由于 DbgUiConnectToDbg() 而触发的连接请求。DbgUiConnectToDbg 是调试器用来与调试子系统建立连接的关键函数，其内部主要是通过调用 NtConnectPort 与 \DbgUiApiPort 端口建立连接。DbgUiThread 收到连接请求后会创建一个用于通知调试状态变化的信号量，并分配一个数据结构记录下请求进程（即调试器进程）的进程 ID 和这个信号量。该信号量会被作为连接应答信息（ConnectionInfo）返回给 DbgUiConnectToDbg 函数，DbgUiConnectToDbg 会将此信号量保存在线程环境块结构（TEB）中的 DbgSsReserved[0] 字段中，WaitForDebugEvent 函数就是使用该信号量来等待调试事件的。DbgUiConnectToDbg 将 NtConnectPort 函数返回的 LPC 句柄保存到 TEB 的 DbgSsReserved[1] 字段中。有一点需要说明的是，对于 Windows 2000 和其后的 Windows，LPC 端口就是可以等待的对象。那么为什么还要额外再使用一个信号量呢？这主要是为了与 Windows NT 4.0 兼容，因为在 NT 4.0 中，端口不是可以等待的内核对象。
- 由于调试器进程调用 WaitForDebugEvent() 而触发的提取调试状态变化消息的请求（DbgUiStateChangeMsg）。DbgUiThread 收到该请求后会将最近一次调试事件的详细信息返回给调试进程。WaitForDebugEvent() API 内部会等待通过 DbgUiConnectToDbg() 获得的信号量（保存在线程环境块中的 DbgSsReserved[0]）。在等待到该信号后，便会发送 DbgUiStateChangeMsg 消息提取调试事件。
- 由于调试器调用 ContinueDebugEvent 而触发的继续消息（DbgUiContinueMsg）。通常该消息中包含了调试器对最近一次调试事件的处理结果（response），DbgUiThread 收到该消息后会通过 \DbgSsApiPort 端口将应答转发给环境子系统进程。

第五，监听 \DbgSsApiPort 端口的 DbgSsThread 线程，主要是接收来自环境子系统的连接请求和汇报调试事件的各种消息，包括异常、线程创建和退出、进程创建和退出、DLL 映射（Map）与反映射（Unmap）等。

第六，用以维护已经注册的调试器进程和被调试进程的信息列表。该列表用于查找调试事件（被调试进程）和调试器进程间的对应关系，在分发调试事件和终止调试对话时起着重要作用。

9.5.3 Windows 环境子系统服务器进程

Windows 环境子系统服务器进程的映像文件名是 CSRSS.EXE，因此常被简称为 CSRSS。尽管从 NT4 开始，窗口管理（包括屏幕输出、用户输入和消息传递）和 GDI 的主体实现被移入到内核态的 win32k.sys 中，但 CSRSS 仍然是 Windows 子系统的灵魂，它监管着系统内运行着的所有 Windows 进程和线程，每个进程在创建后都要到它

这里注册登记后方能运行，退出时也要到此报告注销。除了掌管着各个进程的“生死存亡”外，CSRSS 在桌面管理、终端登录、控制台管理、HardError 报告和 DOS 虚拟机等方面也起着重要作用。在调试方面（Windows XP 之前），CSRSS 也承担着很多重要职责，包括：

- 创建和维护\Windows\ApiPort 端口，该端口是 CSRSS 对外服务的“窗口”，系统的其他进程可以通过该端口向 CSRSS 发送服务请求。在调试方面，该端口是传递调试事件的重要通道。被调试进程的 DebugPort 字段所记录的通常就是这个端口（请参见下文）。
- 调用 DbgSsInitialize() 函数（位于 NTDLL.DLL 中）向调试子系统服务器（SMSS）注册。DbgSsInitialize 会通过\DbgSsApiPort 端口与调试子系统服务器建立连接，并将返回的句柄以全局变量的形式记录下来（名为 ntdll!DbgSspApiPort）。接下来，DbgSsInitialize 会启动一个线程专门监听这个端口，用于接收 SMSS 发起的消息，一旦收到消息便将其发送到\Windows\ApiPort 端口（为了使用方便，DbgSsInitialize() 会将该端口的句柄赋给名为 ntdll!DbgSspKmReplyPort 的全局变量）。因为 CSRSS 与 SMSS 之间的通信大多是异步的，所以 SMSS 发起的消息通常是对前面发生的调试事件的应答，是由调试器通过调用 ContinueDebugEvent 发起的。
- 启动线程用来监听\Windows\ApiPort 端口，该线程的名字通常是 CsrApiRequestThread（其实现位于 CSRSRV.DLL 中）。当 CsrApiRequestThread 接收到消息类型为调试事件的 LPC 消息后（LPC_DEBUG_EVENT），它会调用 DbgSsHandleKmApiMsg() 函数（位于 NTDLL.DLL 中）。DbgSsHandleKmApiMsg 函数会根据调试事件的类型分发给具体的事件处理函数，如 DbgSspException、DbgSspCreateThread 等。这些函数会将调试事件格式化为 DBGSS_APIMSG 结构，并填写合适的枚举结构，然后将 DBGSS_APIMSG 结构通过\DbgSsApiPort 端口转发给 SMSS 进程。全局变量 DbgSspApiPort 记录着 CSRSS 与\DbgSsApiPort 端口的连接句柄。
- CSRSS 的另一个调试支持就是它的仿真系统（Emulation System）。也就是模拟并发送过去发生的调试事件，即杜撰的调试事件，这对于把调试器附加到已经运行的进程很有用。

除了以上功能，CSRSS 的子系统服务中还包含了专门用于调试的服务，在介绍这些服务前，让我们先来看一下每个 Windows 进程是如何调用它们的子系统服务的。

9.5.4 调用 CSRSS 的服务

CSRSS 作为 Windows 子系统的服务器进程，系统内的其他进程可以通过\Windows\ApiPort 端口向其发送服务请求。事实上，每个 Windows 进程在进程启动阶段就已经做好了与 CSRSS 通信的准备。下面通过一个小实验来证明这一点。

启动 WinDBG，然后打开一个可执行文件（File>Open Executable）或附加到某个已

经运行的进程。

键入 `x ntdll!csr*`, 观察 NTDLL.DLL 中包含的以 CSR 开头的符号。

```
0:000> x ntdll!csr*
...
77fc4710 ntdll!CsrPortName = <no type information>
77fc46ac ntdll!CsrServerProcess = <no type information>
77fc46b4 ntdll!CsrServerApiRoutine = <no type information>
77fc46a4 ntdll!CsrProcessId = <no type information>
77fc46b0 ntdll!CsrPortHandle = <no type information>
77f5ec1e ntdll!CsrClientConnectToServer = <no type information>
77f5e9ca ntdll!CsrpConnectToServer = <no type information>
77f5ee8a ntdll!CsrClientCallServer = <no type information>
77fc46c0 ntdll!CsrInitOnceDone = <no type information>
...
```

注意上面的显示, 其中 `ntdll!CsrPortName` 是个 UNICODE 的字符串, 使用 `ds` (`S` 要大写) 可以显示其内容:

```
0:000> ds 77fc4710
00141ea0  "\Windows\ApiPort"
```

可见, `CsrPortName` 变量记录的就是我们刚才介绍的 CSRSS 进程所公开的 LPC 端口名。

`ntdll!CsrPortHandle` 中包含了当前进程与 CSRSS 的 `ApiPort` 连接所得到的句柄:

```
0:000> !handle 7ec
Handle 7ec
Type          Port
```

`ntdll!CsrProcessId` 变量包含了 CSRSS 进程的进程 ID:

```
0:000> dd 77fc46a4 11
77fc46a4  000004cc
```

将十六进制的 `4cc` 转为十进制, 然后打开 Task Manager, 可以发现转换结果与 CSRSS 的进程 ID 是一致的。

`ntdll!CsrInitOnceDone` 变量用来保证与 CSRSS 建立连接的服务只运行一次, 其值为 1, 表示初始化已经完毕。

变量 `ntdll!CsrServerProcess` 用来标记当前进程是否就是服务器 (CSRSS) 进程, 因为 CSRSS 进程本身也会加载和使用 NTDLL.DLL。如果是, 就直接将 CSRSS 的服务例程 (位于 CSRSRV.DLL 中) 的函数地址放入 `ntdll!CsrServerApiRoutine` 数组中, 这样只需直接通过服务的索引号找到这个数组中的函数指针, 就可以调用服务了, 也省去了通过 LPC 通信的过程。如果是调试 CSRSS 进程, 我们可以看到 `ntdll!CsrServerProcess` 和 `ntdll!CsrServerApiRoutine` 变量为非零值, 在普通的进程中, 它们都是 0。

```
0:000> dd ntdll!CsrServerProcess 11
77fc46ac  00000000
0:000> dd ntdll!CsrServerApiRoutine 11
77fc46b4  00000000
```

有了以上的准备工作，每个 Windows 进程就可以很方便地请求 CSRSS 的服务。为了进一步简化这一操作，NTDLL.DLL 中包含了一个名为 CsrClientCallServer() 的未公开 API。这个 API 封装了通过 LPC 端口发送请求和接收应答的细节，使客户进程只要通过调用该函数便可以“享受”CSRSS 提供的服务，不用关心通信的细节。事实上，CsrClientCallServer 内部就是向全局变量 ntdll!CsrPortName 记录的端口（\ApiPort 端口）发送 LPC 消息，监听在这个端口的 CSRSS 的工作线程 CsrApiRequest-Thread 会收到这个消息，然后根据请求中的 API 编号，分发给真正的服务处理函数，最后再把应答发回给请求者。

```
NTSTATUS NTAPI CsrClientCallServer(
    struct _CSR_API_MESSAGE *Request,
    struct _CSR_CAPTURE_BUFFER *CaptureBuffer,
    ULONG ApiNumber,
    ULONG RequestLength)
```

真正完成各种 CSRSS 服务的各个函数主要位于 BASESRV.DLL、CSRSRV.DLL 和 WINSRV.DLL 三个 DLL 模块中。通过 depends 工具，可以观察到这些模块中所包含的函数信息（见图 9-7）。

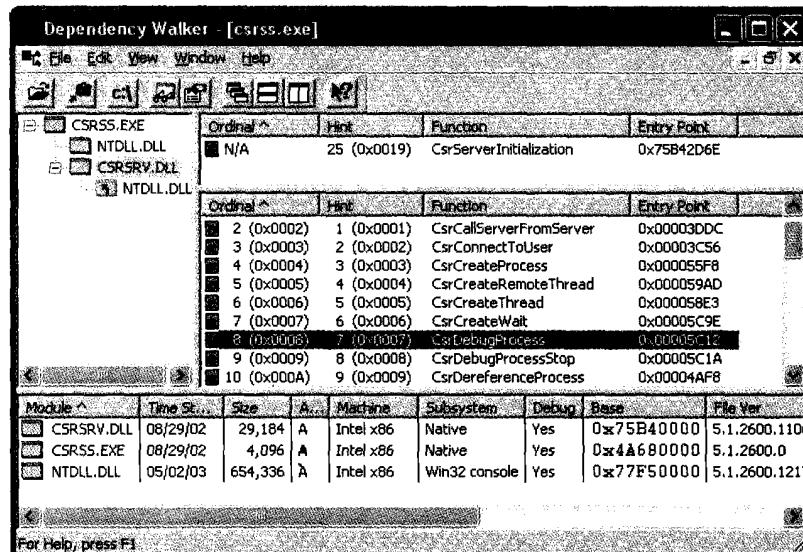


图 9-7 观察 CSRSS 的服务模块和服务例程

9.5.5 CsrCreateProcess 服务

从图 9-7 可以看到，CSRSRV 模块中包含了很多与调试有关的函数，如 CsrCreateProcess、CsrDebugProcess 和 CsrDebugProcessStop 等。我们先来看一下 CsrCreateProcess。

我们知道，大多数 Windows 进程都是通过 CreateProcess 系列 API 创建的。在这个

API 成功调用 NtCreateProcess 或 NtCreateProcessEx 内核服务完成新进程的内核部分创建工作后，它会通过 CsrClientCallServer 向 CSRSS 请求 CreateProcess 服务，该服务的主要目的是向子系统服务器报告新进程的产生，这有点像新生儿的登记。CSRSS 的 CsrApiRequestThread 线程收到 CreateProcess 请求后，会将该请求分发给 CsrCreateProcess 函数来处理。CsrCreateProcess 函数所做的处理主要包括。

1. 为新进程分配一个 CSR_PROCESS 结构，用来登记新进程的各种信息。
2. 调用 NtSetInformationProcess，将新进程的 EPROCESS 结构的 ExceptionPort 字段设置为 \Windows\Apiport 端口对象。
3. 如果进程的创建标志设置了调试标志（DEBUG_PROCESS），那么调用 NtSetInformationProcess 将新进程的 EPROCESS 结构的 DebugPort 字段设置为 \Windows\Apiport 端口对象。这里不必判断正在创建的进程是否为 CSRSS 进程，因为 CsrDebugPorcess 函数一定是在 CSRSS 进程创建后才工作的。
4. 将填写完整的 CSR_PROCESS 结构插入用来记录子系统内所有进程的一个全局链表。

9.5.6 CsrDebugProcess 服务

除了 CsrCreateProcess，另一个与调试密切相关的 CSRSS 服务就是 CsrDebugPorcess。

当调试器使用 DebugActiveProcess 附加到一个已经运行着的程序时，DebugActiveProcess 函数便会通过 CsrClientCallServer 向 CSRSS 请求 DebugProcess 服务。请求的消息结构中包含了当前进程（即调试器进程）的进程 ID 和要调试进程的进程 ID。CSRSS 的 CsrApiRequestThread 线程收到请求后，便会分发给 CsrDebugProcess 函数来处理该请求。CsrDebugProcess 函数所做的处理主要包括。

1. 检查被调试进程是否是 CSRSS 本身。如果是，那么通过调用 RtlGetNtGlobalFlags 得到系统的全局标志，然后检查全局标志中是否设置了 FLG_ENABLE_CSRDEBUG（0x20000）标志位，如果该位为 0，那么不允许调试 CSRSS，CsrDebugProcess 会返回 STATUS_ACCESS_DENIED（访问被拒绝）。可以使用 GFlags 工具（gflags /r +20000）或修改以下注册表项的 GlobalFlag（REG_DWORD）值设置 FLG_ENABLE_CSRDEBUG 标志位。HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager
2. 从包含有子系统的所有进程的链表中找到被调试进程所对应的节点。CSRSS 为每个进程分配并维护一个 CSR_PROCESS 结构，并以链表的形式存储起来。找到被调试进程的 CSR_PROCESS 结构后，CsrDebugProcess 将调试器进程（也就是发起请求者）的进程 ID，调试器进程发起调试的线程 ID，以及该进程正在被调试等属性记录在 CSR_PROCESS 结构中。

3. 挂起被调试进程。
4. 如果被调试进程不是 CSRSS 自己，调用 NtSetInformationProcess 将被调试进程的 EPROCESS 结构的 DebugPort 字段设置为连接到\Windows\ApiPort 端口的 LPC 句柄。
5. 调用 DbgSsHandleKmApiMsg() 函数，依次向调试器发送杜撰的进程创建、线程创建和 DLL 加载事件。因为是附加到已经运行的进程，所以这些事件事实上是以前发生的，现在 CSRSS 根据记录重新“播放”给调试器，这一功能又被称为 CSRSS 的调试事件“仿真”功能。
6. 唤醒被调试进程。
7. 如果被调试进程是 CSRSS 本身，调用 NtSetInformationProcess 将其 EPROCESS 结构的 DebugPort 字段设置为连接到\SmApiPort 端口的句柄。之所以调试 CSRSS 进程时要将 DebugPort 设置为\SmApiPort 端口，是因为\SmApiPort 端口是由 SMSS 进程创建并维护的，而\Windows\ApiPort 端口是由 CSRSS 进程维护的。如果调试 CSRSS 进程时仍使用\Windows\ApiPort 端口，那么 CSRSS 被中断到调试器后，就“无人”监听和维护该端口了，这样会导致死锁。

上面我们简要介绍了 Windows 2000 和 NT 的调试子系统服务器，包括组成部分和每一部分所承担的功能。下面简要介绍这些部件的协作方法。首先，调试器通过 DbgUiConnectToDbg 函数连接到位于 SMSS 中的调试子系统服务器，也就是\DbgUiApiPort 端口。如果是调试新创建的进程，那么 CSRSS 的创建进程服务（CsrCreateProcess）会将新创建进程的 DebugPort 设置为连接到 CSRSS 的 LPC 端口（\Windows\ApiPort）。如果是调试已经运行的程序，那么 CSRSS 的调试服务（CsrDebugProcess）会将\Windows\ApiPort 端口设置到要调试进程的 DebugPort 字段。

当内核中的 DbgkForwardException 收集到异常事件时，它会调用 DbgkpSend-ApiMessage，后者再调用 LpcRequestWaitReplyPort 将消息发到 DebugPort 所标识的 LPC 端口，即\Windows\ApiPort。这样，CSRSS.EXE 便收到了调试消息，它会将其转发给\DbgSsApiPort。守候\DbgSsApiPort 端口的会话管理器进程（SMSS.EXE）收到调试消息后会根据 LPC 消息的 AppClientId 检查是否有调试器进程与其匹配，如果有，便释放与这个被调试进程相关的信号量 NtReleaseSemaphore（StateChangeSemaphore）通知调试器有事件发生。与 XP 中一样，调试器建立调试对话后便使用 WaitForDebugEvent API 等待调试事件，不过这个 API 在 XP 之前的实现是在用户态调用 NtWaitForSingle-Object 服务等待 DbgStateChangeSemaphore 信号。调试器得到信号后，便通过 NtRe-questWaitReplyPort(..., DbgUiWaitStateChangeApi, ...) 函数向\DbgUiApiPort 端口读取消息。会话管理器发现来自\DbgUiApiPort 的 DbgUiWaitStateChangeApi 请求后，便会将属于该调试器的消息发给调试器。

9.6 比较两种模型

前面两节我们分别介绍了 Windows XP 之后(包括 XP)和之前的用户态调试模型，二者的主要差异是调试子系统服务器的位置。本节对这两种模型做一个简单的总结和比较。

9.6.1 Windows 2000 调试子系统的优点

总体来说，Windows 2000 调试子系统的设计是非常优秀的。

第一，架构非常清晰。整个子系统以调试事件的产生、传递、分发和处理为线索，被清楚地划分为职责明确的几大模块：内核例程、NTDLL 中的支持例程和调试子系统服务器。

第二，接口非常简单。尽管整个调试系统包含了很多个进程和模块，每个模块内部的功能都比较复杂，但是部件之间的接口非常简洁。对于调试器而言，它只需要与调试子系统服务器通信；对于内核例程来说，只需要向被调试进程的 DebugPort 发送信息，根本不关心这个端口到底是谁创建的，谁在监听。这些精湛的低耦合设计带来了极强的灵活性和可扩展性，可以方便地扩充和修改子系统内某一部分的设计，同时保证其他部分能照常工作。举例来说，Windows XP 对调试子系统的服务器部分作了较大改动，但是针对 Windows 2000 或 NT 设计的调试器依然可以一如既往地工作，老的应用程序也仍然可以被调试。

第三，代码复用方面也做得非常好，把需要共享的调试支持例程（DbgUi 系列、DbgSs 系列和 Csr 系列）放到 NTDLL 中，因为 NTDLL 是 Windows 内包括环境子系统服务器在内的所有程序都加载和使用的模块，这样便可以让这些代码得到最有效的复用。比如，DbgSsInitialize 是用来完成与调试子系统服务器（SMSS）建立连接等初始化工作的，今天看来这部分代码只有 CSRSS 需要，没有必要放在 NTDLL 中，但是我们要知道，Windows 2000 和 NT 设计时都曾考虑到要支持多个环境子系统的，如果把 DbgSsInitialize 放到 CSRSS 模块中，那么 POSIX 子系统服务器就要重复编写这部分代码了。

9.6.2 Windows 2000 调试子系统的安全问题

金无足赤，Windows 2000 调试子系统模型（事实上 Windows NT 4 就使用这样的模型）在应用了多年之后，2002 年 3 月有人（EliCZ）在互联网上公开了它的一个安全漏洞，取名为 DebPloit（Deb 代表调试）。攻击这个漏洞的基本步骤如下。

第一，使用 DbgUiConnectToDbg 与调试子系统服务器的 UI 端口建立连接，这是调试器通常采取的动作。

第二，通过 `ZwConnectPort` 与调试子系统服务器的 LPC 端口 (`\DbgSsApiPort`) 建立连接，这是环境子系统服务器程序（如 `CSRSS`）通常采取的动作。主要的漏洞也出现在这里——`\DbgSsApiPort` 端口在创建时所设定的安全权限允许所有权限等级的进程都可以连接。设计者之所以这么做，应该是为了照顾 `POSIX` 和 `OS/2` 等子系统服务器进程，使它们在非管理员账号下也能成功连接调试服务器。

第三，成功进行以上两步后，对调试服务器（`SMSS`）而言，攻击程序便既有了调试器身份，又有了环境子系统身份。于是它便可利用环境子系统身份来模拟调试事件，欺骗 `SMSS` 为其服务，然后再利用调试器身份接收服务结果。接下来，攻击程序模拟一个调试器附加到被攻击进程的调试事件，将要攻击的其他系统进程的进程 ID（或线程 ID）和假调试器（攻击程序自身）的进程 ID 放到消息结构中，发给 `SMSS`，目的是让调试子系统为假调试器与被攻击进程建立调试关系。

第四，一旦调试关系建立后，攻击程序便可以控制和操作被攻击进程了，最简单的一种攻击就是直接退出“调试器”，因为调试器退出会导致被调试进程也退出，这样攻击程序的退出便可能使重要的系统进程也随着终止，重要系统进程（如 `WinLogon` 和 `Lsass` 等）的终止会导致系统崩溃。

修正以上漏洞的方法之一，是拒绝没有管理员权限的进程连接 `\DbgSsApiPort` 端口，也就是只要在创建该端口时将安全描述符设为 `NUL`（使用 `SMSS` 默认的管理员权限，使低于该权限的进程无法访问）。

2002 年 5 月 22 日微软公布了关于这个漏洞的安全公告（*Security Bulletin*），代号为 `MS02-024`，同时提供了修正这个问题的 `HotFix`，(<http://support.microsoft.com/kb/320206>)，此后的 Windows 2000 Service Patch (SP3 或更高) 也都包含了这个修正。Windows XP 操作系统将调试子系统服务器移入到内核模式中，不再使用 `\DbgSsApiPort` 端口，因此不再存在这个漏洞。

9.6.3 Windows XP 模型的优点

Windows XP 的调试模型以调试对象（`DebugObject`）为核心，去除了原本相对复杂的利用 LPC 多级通信的通信模型。以前，调试消息要通过 `CSRSS` 的 `ApiPort`、`SMSS` 的 `DbgSsApiPort` 和 `DbgUiApiPort` 三级传递，显得有些累赘。新的模型利用调试对象中的事件链表直接通信，使通信过程大大简化。

从进程角度来看，Windows XP 的模型也更加简洁，在调试事件的驱动下，调试器进程和被调试进程以调试对象为纽带相互通信。

从功能上看，Windows XP 模型的改变为增加下面我们将介绍的新调试功能创造了便利条件。

9.6.4 Windows XP 引入的新调试功能

因为使用了 DebugObject 内核对象和新的调试模型(调试子系统服务器移入内核), Windows XP 的调试子系统支持跨 Windows 登录会话 (Windows Session) 进行调试。举例来说, 用户 A 可以登录系统并调试用户 B 运行的程序。登录的方式可以是通过终端服务 (Terminal Service) 或使用 Windows XP 的快速用户切换 (Fast User Switch) 功能。跨 Windows 登录会话调试对于 Windows 2000 的调试模型来说是做不到的, 因为不同的登录会话 (session) 会启动不同的 CSRSS 进程, 这意味着用户 A 的调试器进程与用户 B 的进程分别属于不同的 Windows 子系统, 二者无法建立调试对话。

分离调试会话并保持被调试进程继续运行是 XP 之后的调试子系统的一个新功能。此前结束调试会话时, 被调试进程也随之终止。不过, 这一功能在以前的模型中也是可以实现的。

因为调试 API (DebugActiveProcess、WaitForDebugEvent) 的功能和函数原型没有任何改变, 因此以上两种模型在 API 一层是兼容的。也就是说, 老的调试器无需任何修改仍能非常好地工作在新的系统中, 感觉不到调试模型的变化。

从数据结构角度来看, 新旧模型也保持了很好的兼容性。比如, 在调试器一侧, 仍然使用调试器线程 TEB 的 DbgSsReserved[2] 数组, 来记录用来与调试子系统通信的重要信息。以前 DbgSsReserved[1] 记录的是 LPC 端口句柄 (DbgUiApiPort), DbgSsReserved[0] 记录的是等待调试消息的事件对象。现在, DbgSsReserved[1] 记录的是调试对象 (DebugObject) 句柄, DbgSsReserved[0] 改为他用 (参见 10.1.3), 因为只要直接等待调试对象就可以了。

9.7 NTDLL 中的调试支持例程

在第 8 章 (8.4.6 节) 中我们简要介绍过 NTDLL.DLL, 它是 Windows 系统中一个很特别的模块, 不仅所有的 Windows 应用程序都与它有依赖关系, 而且系统的进程也要使用它。NTDLL 的重要性首先体现在它所包含的用于访问系统服务的残根函数 (stub), 例如 NTDLL 中的 NtCreateFile 函数是内核中真正的 NtCreateFile 函数的残根函数, 这些残根函数是用户态的应用程序访问内核服务的唯一正式方法。此外, NTDLL 的重要性还体现在它内部包含了很多关键的支持函数, 比如上一节讲到的 CsrClientCallServer 是调用 Windows 子系统服务的函数。

在调试方面, NTDLL 中也包含了很多非常重要的函数。可以把这些函数分为如下 3 类: DbgUi 类、DbgSs 类、Dbg 类, 下面分别进行介绍。

9.7.1 DbgUi 函数

为了让调试器程序可以方便地使用调试子系统所定义的功能, NTDLL 中设计了一系列函数, 它们都是以 DbgUi 开头的, 我们将其称为 DbgUi 函数。在 Windows 2000 的 NTDLL.DLL 中, 包含了 3 个 DbgUi 函数, 分别是用于和调试子系统建立连接的 DbgUiConnectToDbg, 用于等待调试事件的 DbgUiWaitStateChange 和用于继续调试事件的 DbgUiContinue 函数。Windows XP 进一步丰富了 DbgUi 系列函数, 其数量从原来的 3 个增加到 10 个, 表 9-3 列出了所有这些函数, 包括 Windows 2000 中已经存在的, 第二列说明该函数是否是 Windows XP 版本新引入的。

表 9-3 NTDLL 中的 DbgUi 系列函数

函数名	XP	说明
DbgUiDebugActiveProcess	是	DebugActiveProcess API 的实现, KERNEL32 中在将进程 ID 转为句柄后调用 DbgUiDebugActiveProcess, 后者再调用内核服务 NtDebugActiveProcess。Windows 2000 是直接在 Kernel32.dll 中实现这个 API 的
DbgUiConnectToDbg	否	连接调试子系统, XP 中主要是调用 ZwCreateDebugObject
DbgUiConvertStateChangeStructure	是	将 DBGUI_WAIT_STATE_CHANGE 结构转换为调试器所需要的 DEBUG_EVENT 结构
DbgUiGetThreadDebugObject	是	从调试器工作线程 TEB 中 (偏移 0xf24 处) 读取调试对象
DbgUiSetThreadDebugObject	是	将调试对象记录到 TEB 中 (偏移 0xf24 处)。建立调试对话中, 先调用 NtCreateDebugObject 创建调试对象, 然后再记录到 TEB
DbgUiIssueRemoteBreakin	是	在被调试进程中创建远程线程以使其中断到调试器, 是 DebugBreakProcess API (KERNEL32.DLL 输出) 的实现
DbgUiContinue	否	恢复被调试进程, XP 中调用系统服务 NtDebugContinue, Windows 2000 中通过 LPC 回复消息给调试子系统
DbgUiWaitStateChange	否	等待调试事件, XP 中调用系统服务 NtWaitForDebugEvent, 后者再等待保存在 TEB 的 DbgSsReserved[0] 中的 DebugObject 对象, Windows 2000 中就在该函数中使用 NtWaitForSingleObject 等待保存在 DbgSs Reserved[0] 的信号量, 等待成功后, 通过 NtRequestWaitReplyPort() 读取调试事件
DbgUiStopDebugging	是	停止调试, 调用系统服务 NtRemoveProcessDebug

在 Windows 2000 中, DbgUiConnectToDbg 主要是调用 NtConnectPort() 与会话管理器 (SMSS.EXE) 的 \DbgUiApiPort 连接。返回的端口句柄和用于等待端口数据的信号量 (Semaphore) 被保存在调用线程的线程环境块中 (Teb()->DbgSsReserved[1] 和

`Teb()->DbgSsReserved[0]`)。调试器等待调试事件（执行 `DbgUiWaitStateChange`）实际就是等待保存在 Teb 中的信号量。

从软件架构的角度来讲，`DbgUi` 函数是调试子系统向外（调试器）提供的接口。

9.7.2 DbgSs 函数

在 Windows XP 以前，调试子系统服务其实是实现在会话管理器（SMSS）中的，为了让各个环境子系统可以方便地与调试子系统建立联系，并支持调试功能，`NTDLL.DLL` 实现了一系列以 `DbgSs` 开头的函数，我们将其称为 `DbgSs` 函数。例如 `DbgSsInitialize()` 函数是供环境子系统来初始化调试支持的（见清单 9-6），包括连接调试子系统服务器的 `\DbgSsApiPort` 端口和注册回复端口（`KmReplyPort`）。在 `WinDBG` 中使用符号搜索命令很容易列出这些函数。

清单 9-6 NTDLL.DLL 中包含的供环境子系统使用的调试支持函数（Windows 2000）

0:000> x ntdll!DbgSs*	
77f9ae47 ntdll!DbgSsInitialize = <no type information>	[初始化]
77fcf2a4 ntdll!DbgSspKmApiMsgFilter = <no type information>	
77f9b04f ntdll!DbgSspSrvApiLoop = <no type information>	
77f9ae8d ntdll!DbgSsHandleKmApiMsg = <no type information>	[见正文]
77f9adad ntdll!DbgSspLoadDll = <no type information>	[报告模块映射消息]
77f9acba ntdll!DbgSspCreateProcess = <no type information>	[报告进程创建消息]
77f9ac1c ntdll!DbgSspException = <no type information>	[报告异常消息]
77f9adfd ntdll!DbgSspUnloadDll = <no type information>	[报告反映射消息]
77fcf144 ntdll!DbgSspUiLookUpRoutine = <no type information>	[函数指针]
77f9ad63 ntdll!DbgSspExitProcess = <no type information>	[报告进程退出消息]
77f9ac6a ntdll!DbgSspCreateThread = <no type information>	[报告线程创建消息]
77f9abde ntdll!DbgSspConnectToDbg = <no type information>	
77f9ad19 ntdll!DbgSspExitThread = <no type information>	[报告线程退出消息]
77fcf1f0 ntdll!DbgSspApiPort = <no type information>	[保存连接 SMSS 端口的全局变量]
77fcf158 ntdll!DbgSspKmReplyPort = <no type information>	[全局变量]
77fcf2a8 ntdll!DbgSspSubsystemKeyLookupRoutine = <no type information>	

其中，`DbgSsHandleKmApiMsg()` 供环境子系统处理（包装并发送给调试服务器）收到的调试消息（`LPC_DEBUG_EVENT`）。以 `DbgSsp` 开头的符号是调试子系统的内部函数或全局变量。

Windows XP 将调试子系统移到内核中，因此其 `NTDLL.DLL` 中不再存在以上 `DbgSs` 函数。

9.7.3 Dbg 函数

除了以上两类函数，`NTDLL.DLL` 中还有一系列以 `Dbg` 开头（非 `DbgUi` 或 `DbgSs`）的调试支持函数，我们将其称为 `Dbg` 函数。例如用于触发断点事件的 `DbgBreakPoint`，它是调试 API `DebugBreak` 的实现，事实上，在 x86 平台中这个函数的内部就是一条 INT 3 指令。

```
0:001> uf ntdll!DbgBreakPoint
ntdll!DbgBreakPoint:
7c901230 cc          int     3
7c901231 c3          ret
```

此外，还有 DbgUserBreakPoint(断点指令)、DbgPrint(打印调试信息)、DbgPrompt(提示输入)、DbgPrintReturnControlC 等。Windows XP 又引入了 DbgBreakPointWithStatus、DbgSetDebugFilterState(设置调试信息输出的过滤级别，内部调用 NtSetDebugFilterState 内核服务)、DbgQueryDebugFilterState 和 DbgPrintEx。

9.8 调试 API

Windows SDK 中公开了一系列 API 供调试器与调试子系统交互以实现各种调试功能。大多数 SDK 中公开的调试 API 都是从 KERNEL32.DLL 导出的。其中有些就是在 KERNEL32.DLL 中实现的，有些是调用上一节介绍的 NTDLL 中的调试支持函数的。表 9-4 列出了目前 SDK 中已经文档化了的调试 API。

表 9-4 SDK 中公开的调试 API

API	版本	描述	实现
BOOL CheckRemoteDebuggerPresent(HANDLE hProcess, PBOOL pbDebuggerPresent)	XP SP1	判断指定的进程是否处于被调试状态	调用 NtQueryInformationProcess 查询进程环境块 (PEB)
BOOL ContinueDebugEvent(DWORD dwProcessId, DWORD dwThreadId, DWORD dwContinueStatus)	9x, NT	供调试器恢复被调试进程运行，回复调试事件	调用 NTDLL 中的 DbgUiContinue()
BOOL DebugActiveProcess(DWORD dwProcessId)	9x, NT	供调试器附加到已经运行的进程	调用 NTDLL 中的 DbgUiDebugActiveProcess
BOOL DebugActiveProcessStop(DWORD dwProcessId)	XP	分离调试会话	将进程 ID 转换为句柄后调用 NTDLL 中的 DbgUiStopDebugging
void DebugBreak(void)	9x, NT	在当前进程中产生断点异常	调用 NTDLL 中的 DbgBreakpoint
BOOL DebugBreakProcess(HANDLE Process)	XP	在指定进程中产生断点异常	调用 NTDLL 中的 DbgUiIssueRemoteBreakin
BOOL DebugSetProcessKillOnExit(BOOL KillOnExit)	XP	指定调试器线程退出时是否终止被调试进程	使用 DbgUiGetThreadDebugObject 和 NtSetInformationDebugObject 实现
void FatalExit(int ExitCode)	9x, NT	16 位 Windows 遗留下来的 API。最初是供调试阶段的应用程序使用强制中断到调试器	目前 (2000, XP) 实现只是调用 ExitProcess

续表

API	版本	描述	实现
BOOL FlushInstructionCache(HANDLE hProcess, LPCVOID lpBaseAddress, SIZE_T dwSize)	9x, NT	当调试器修改代码段时, 可以使用此 API 冲转 (flush) 缓存	调用 NtFlushInstructionCache
BOOL GetThreadContext(HANDLE hThread, LPCONTEXT lpContext)	9x, NT	取得指定线程的上下文 (CONTEXT) 结构	调用 NtGetContextThread
BOOL GetThreadSelectorEntry(HANDLE hThread, DWORD dwSelector, LPLDT_ENTRY lpSelectorEntry)	9x, NT	从指定线程的局部描述符表 (LDT) 中取得指定选择子所对应的表项 Entry	调用 NtQueryInformationThread
BOOL IsDebuggerPresent(void)	9x, NT	判断调用进程是否在被调试	检查 PEB 的 BeingDebugged 字段
void OutputDebugString(LPCTSTR lpOutputString)	9x, NT	供应用程序输出调试信息。当被调试时, 这些信息会显示到调试器。参见后文	通过产生异常实现: RaiseException(DBG_PRINTEXCEPTION_C,0,2,Exception Arguments), 详见 10.7 节
BOOL ReadProcessMemory(HANDLE hProcess, LPCVOID lpBaseAddress, LPVOID lpBuffer, SIZE_T nSize, SIZE_T* lpNumberOfBytesRead)	9x, NT	读取指定进程空间中的指定内存区域	调用 NtReadVirtualMemory
BOOL SetThreadContext(HANDLE hThread, const CONTEXT* lpContext)	9x, NT	设置指定线程的 CONTEXT 信息	调用 NtSetContextThread
BOOL WaitForDebugEvent(LPDEBUG_EVENT lpDebugEvent, DWORD dwMilliseconds)	9x, NT	供调试器的工作线程等待调试事件	调用 NTDLL 中的 DbgUiWaitStateChange
BOOL WriteProcessMemory(HANDLE hProcess, LPVOID lpBaseAddress, LPCVOID lpBuffer, SIZE_T nSize, SIZE_T* lpNumberOfBytesWritten);	9x, NT	向指定进程空间中的指定内存区域写入数据	调用 NtWriteVirtualMemory

上表中, 第 2 列给出的是支持该 API 的 Windows 最低版本, 最后一列描述该 API 的主要实现方法。大家在使用这些 API 前, 应该进一步查阅 SDK 文档以了解其详细的用法。

9.9 本章总结

本章比较详细地介绍了 Windows 操作系统用户态调试模型及用于实现这一模型的各个模块和函数。9.1 节是概要介绍，9.2 节和 9.3 节分别介绍了调试消息的采集和发送过程，而后介绍了调试模型的核心部分——调试子系统服务器（9.4 节至 9.6 节）。9.7 节介绍了 NTDLL 中的调试支持例程，9.8 节简要介绍了调试 API。

总的来说，本章介绍了 Windows 系统中用于支持用户态调试的基础设施，下一章我们将进一步介绍这些设施是如何协同配合而完成各种调试功能的。如果说本章是为 Windows 的调试设施拍一幅静态的照片，那么下一章我们将介绍这些设施的动态特征。

参考文献

1. Alex Ionescu. Kernel User-Mode Debugging Support (Dbgk).
<http://www.alex-ionescu.com/dbgk-3.pdf>
2. MSDN Library for Visual Studio 2005. Microsoft Corporation

用户态调试过程

上一章我们介绍了 Windows 操作系统中用于支持用户态调试的各种基础设施，描述了这些设施的静态特征和功能。本章将讨论这些设施的动态特征，解析 Windows 系统中用户态调试的关键过程，特别是调试器、被调试程序及调试子系统这三者是如何相互配合完成各种调试功能的。我们先介绍调试器进程和被调试进程的基本特征（10.1 节和 10.2 节），然后介绍建立调试会话的两种情况：在调试器中启动被调试程序（10.3 节）和附加到已运行的进程（10.4 节）。10.5 节将介绍调试器处理调试事件的基本方法。10.6 节介绍被调试进程中断到调试器的典型情况。10.7 节介绍 OutputDebugString API 的工作原理及有关的工具。10.9 节介绍调试过程的最后一个步骤，即调试会话的终止和分离。

因为本章讨论的是用户态调试，为了行文简洁，除非特别说明，本章提到的调试器就是指用户态调试器。

10.1 调试器进程

调试器进程和被调试进程是调试过程的两个主角。从用户的角度来看，调试过程就是使用调试器进程来控制和观察被调试进程的过程。本节和下一节将简要描述调试过程中这两个主角的基本特征，以及它们是如何联系起来的。

调试器进程（Debugger Process）就是指运行着的调试器程序，或者说是调试器程序的运行实例（instance），比如运行着的 MSDEV.EXE（VC6 的 IDE）、DEVENV.EXE（VS2005 的 IDE）或 WinDBGEXE 等。

10.1.1 线程模型

可以把调试器的主要功能分成如下两个方面。

- 人机接口，以某种界面的形式将调试功能呈现给用户，并监听和接收用户的输入（命令），在收到用户输入后，进行解析和执行，然后把执行结果显示给用户。

- 与被调试进程交互，包括与被调试进程建立调试关系，然后监听和处理调试事件，根据需要将被调试进程中断到调试器，读取和修改被调试进程的数据，或者操纵它的其他行为。根据上一节的介绍，调试器主要是通过调试子系统与被调试进程交互的。但是从用户的角度来看，可以认为调试器是直接与被调试程序交互的。本章将使用这种粗略的描述方法。

总而言之，调试器进程一方面是与用户对话的，另一方面是与被调试进程对话的。为了及时地响应来自每一方面的对话请求，调试器通常会使用两个线程，每个线程负责一个方面的对话。负责与人（用户）对话的被称为 UI 线程，负责与被调试进程对话的被称为调试器工作线程(Debugger's Worker Thread)或调试会话线程，简称 DWT。以 WinDBG 调试器为例，在它启动后，通常只有一个 UI 线程，即初始线程，在开始调试另一个程序后，那么它便会创建工作线程。清单 10-1 显示了这两个线程的函数执行过程。

清单 10-1 WinDBG 调试器的 UI 线程和工作线程

```
0:001> ~* k
0  Id: 1774.bd0 Suspend: 1 Teb: 7ffdf000 Unfrozen
ChildEBP RetAddr
0006df24 7e419408 ntdll!KiFastSystemCallRet      //在内核态执行系统服务
0006ff7c 0104f252 USER32!NtUserWaitMessage+0xc    //调用等待窗口消息的子系统服务
0006ffc0 7c816ff7 windbg!_wmainCRTStartup+0xfd   //程序的启动函数
0006fff0 00000000 kernel32!BaseProcessStart+0x23  //系统的进程启动函数
// 上面是 UI 线程，下面是工作线程
# 1  Id: 1774.1358 Suspend: 1 Teb: 7ffde000 Unfrozen
ChildEBP RetAddr
00cef04 02242d3e ntdll!ZwWaitForDebugEvent      //调用等待调试事件的内核服务
00cefda0 02107d23 dbgeng!LiveUserDebugServices::WaitForEvent+0x12e
00ceff10 020a3c3f dbgeng!LiveUserTargetInfo::WaitForEvent+0x3b3
00ceff34 020a401e dbgeng!WaitForAnyTarget+0x5f
00ceff80 020a4290 dbgeng!RawWaitForEvent+0x2ae   //调试引擎的内部函数
00ceff98 0102925f dbgeng!DebugClient::WaitForEvent+0xb0  //等待调试事件
00ceffb4 7c80b6a3 windbg!EngineLoop+0x13f      //调试事件循环
00ceffec 00000000 kernel32!BaseThreadStart+0x37  //线程的初始函数
```

可见，以上两个线程分别是在等待用户输入和调试事件。

当然，并不是所有调试器都采用双线程模式，有些命令行界面的调试器只使用一个线程，同时负责以上两种对话，这一个线程频繁地在二者间“奔走”，哪一方需要对话便响应哪一方，在响应结束后再看另一方是否有对话请求。据笔者观察，命令行接口的 NTSD 调试器就是这样设计的。

包括 WinDBG 在内的大多数调试器使用的都是双线程模式，所以本书如不特别说明，讨论的都是这种情况。需要指出的是，双线程模式并不代表调试器进程中只有这两个线程，因为在执行某些调试功能时调试器可能还会创建其他线程。

10.1.2 调试器的工作线程

下面我们来看调试器工作线程的主要逻辑。清单 10-2 显示了调试器工作线程的核心代码。

清单 10-2 调试器工作线程（DWT）的基本框架

```

1 //*****
2 // Backbone of a Debugger's Worker Thread (DWT)
3 // ****
4     if(bNewProcess)
5         CreateProcess ( ..., DEBUG_PROCESS , ... );
6     else
7         DebugActiveProcess(dwPID)
8
9     while ( 1 == WaitForDebugEvent (&DbgEvt, INFINITE) )
10    {
11        switch (DbgEvt.dwDebugEventCode)
12        {
13            case EXIT_PROCESS_DEBUG_EVENT:
14                break;
15            //other cases
16        }
17        ContinueDebugEvent ( ... ) ;
18    }

```

其中，第 4~7 行用于建立调试对话（后面两节将详细讨论），第 9~18 行是调试事件循环，类似于 Windows 程序的消息循环。第 9 行是调用 `WaitForDebugEvent` 等待调试事件，当被调试程序中有调试事件发生时（线程创建退出、加载 DLL 或产生异常等），调试子系统便会通过 `DEBUG_EVENT` 结构通知调试器。接收到调试事件后，第 11~16 行会根据 `DEBUG_EVENT` 结构中的事件代码（`dwDebugEventCode`）来判断调试事件的类型，采取适当的动作，并根据需要中断给用户，让用户可以进行各种诊断和分析。在调试器处理好一个事件或接收到用户的恢复执行命令（如 WinDBG 的 `g` 命令）后，第 17 行会向调试子系统发送一个回复命令，典型的回复命令是 `DBG_CONTINUE`，即恢复运行被调试程序。接下来，DWT 开始等待下一个调试事件，如此往复，直到收到被调试进程退出的事件或发生其他终止情况（第 10.8 节讨论）为止。

10.1.3 DbgSsReserved 字段

从线程的内部数据结构来看，调试器的 UI 线程与普通线程没什么特别，但是调试器的工作线程（DWT）与普通线程通常是有所不同的，具体来说，它的线程环境块（TEB）结构的 `DbgSsReserved` 字段通常与普通线程的取值不同。

上一章我们已零散地介绍过 `DbgSsReserved` 字段，简单来说，TEB 结构的 `DbgSsReserved[2]` 数组就是专门用来记录调试器工作线程与调试子系统之间通信用的同步对象和通信对象的。在 Windows XP 之前，`DbgSsReserved[1]` 记录的是 LPC 端口句柄（`DbgUiApiPort`），`DbgSsReserved[0]` 记录的是等待调试消息的事件对象。

从 Windows XP 开始，`DbgSsReserved[1]` 用来记录调试对象（`DebugObject`）句柄，`DbgSsReserved[0]` 用来记录被调试线程链表的表头，这个链表的每个节点是一个 `DBGSS_THREAD_DATA` 结构，用来描述被调试进程中的一个线程。

```

typedef struct _DBGSS_THREAD_DATA
{

```

```

struct _DBGSS_THREAD_DATA *Next;      //指向下一个节点
HANDLE ThreadHandle;                //线程句柄(被调试进程中)
HANDLE ProcessHandle;               //被调试进程的进程句柄
DWORD ProcessId;                   //被调试进程的进程 ID
DWORD ThreadId;                    //线程 ID(被调试进程中)
BOOLEAN HandleMarked;              //退出标记
} DBGSS_THREAD_DATA, *PDBGSS_THREAD_DATA;

```

例如，以下是使用 WinDBG 观察 MSDEV 调试器的 DMT 线程所得到的结果：

```

0:011> dt -b ntdll!_Teb 7ffd4000 -y DbgSsReserved
ntdll!_TEB
+0xf20 DbgSsReserved :
[00] 0x001ff878
[01] 0x000003b8

```

其中 0x000003b8 是调试对象的句柄，使用 !handle 命令可以确认这一点：

```

0:011> !handle 3b8
Handle 3b8
Type          DebugObject

```

0x001ff878 是被调试线程链表的头节点，调试 API WaitForDebugEvent 和 ContinueDebugEvent 会维护这个链表。

Windows XP 的 NTDLL.DLL 中新引入的两个 DbgUi 函数 DbgUiGetThreadDebugObject 和 DbgUiSetThreadDebugObject，实际上就是读写 DbgSsReserved[1] 字段的，例如：

```

ntdll!DbgUiGetThreadDebugObject:
7c9506de 64a118000000    mov     eax,dword ptr fs:[00000018h]
7c9506e4 8b80240f0000    mov     eax,dword ptr [eax+0F24h]
7c9506ea c3             ret

```

那么，为什么要使用 DbgSsReserved 数组，而不是把这些信息直接返回给调试器程序来管理呢？这主要是为了简化调试器的设计，使其不用保存和维护这些数据，也不用关心其中的细节。从调试 API 的函数原型也可以看到这一点，比如 WaitForDebugEvent API 只有两个参数，一个是用来接收调试事件的，另一个是等待的毫秒数。这个 API 内部会从当前线程的 TEB 结构中读取要等待对象的句柄。这也是必须在开始调试会话的线程中调用 WaitForDebugEvent API 的原因。

目前版本的 WinDBG 调试器在 XP 系统中运行时不是使用调试 API (Kernel32.DLL 输出)，而是直接调用 NTDLL 中的调试支持函数或系统的内核服务，并且自己保存和维护调试对象句柄和线程数据，因此，如果观察其调试器工作线程的 TEB 结构，那么可能看到 DbgSsReserved 数组的两个元素都为 0。为什么说可能呢？是因为与观察的时机有关，在调用 CreateProcess 创建调试会话时，CreateProcess 内部调用的 DbgUiConnectToDbg 函数会将 DbgSsReserved[1] 设置为非 0，但在 CreateProcess 返回后，WinDBG 会调用 DbgUiSetThreadDebugObject 将其设置为 0。

本节简要介绍了调试器进程的主要特征，特别是调试器工作线程的属性。最后要说明的一点是，操作系统并不区分调试器进程和普通的进程，一个调试器进程同时也

可以被调试，成为被调试进程。笔者在写作此书时便经常启动几个调试器实例，一个调试另一个，以便了解调试器的工作原理。

10.2 被调试进程

被调试进程（Debuggee Process）泛指处于被调试状态的程序运行实例，比如运行在调试器下的控制台程序、窗口程序，或者 Windows 系统服务，等等。

10.2.1 特征

为了不影响问题的重现和分析结果，调试过程本身应该尽可能少地改变被调试进程的属性。也就是说，一个进程在被调试时与没有被调试时越相近越好。但为了实现某些调试功能，系统不得不修改被调试进程的某些属性或在其中执行一些用于调试的代码。概括地说，一个处于被调试状态的 Windows 进程与普通进程相比，会有如下差异。

1. 进程执行块（Executive Process Block，即 EPROCESS 结构）的 DebugPort 字段不为空。这是在内核空间中判断一个进程是否正在被调试的主要特征。
2. 进程环境块（Process Environment Block，简称 PEB）的 BeingDebugged 字段不等于 0。这是用户态判断一个进程是否正在被调试的主要方法。
3. 可能会存在一个由调试器远程启动的线程，这个线程的作用是将被调试进程中断到调试器，我们称之为 RemoteBreakin 线程（10.6.4 节将详细讨论）。
4. 响应调试热键（F12），按调试热键可以将处于被调试状态的进程中断到调试器，没有被调试的进程通常不响应调试热键。

下面，我们将深入介绍 DebugPort 字段和 BeingDebugged 字段。10.6 节将介绍 RemoteBreakin 线程和调试热键。

10.2.2 DebugPort 字段

在第 8 章我们简单介绍了进程执行块，它是所有 Windows 进程都拥有的一个数据结构，名为 EPROCESS。EPROCESS 结构位于内核空间中，是系统用来标识和管理每个 Windows 进程的基本数据结构。EPROCESS 结构的具体定义（字段个数和偏移位置）因 Windows 的版本不同而不同，但是都包含一个有关调试的重要字段，即 DebugPort。如果一个进程不在被调试，那么 DebugPort 字段为 NULL。如果一个进程在被调试，那么 DebugPort 字段是一个指针，指向的内容可能因 Windows 版本不同而不同。在 Windows XP 之前，DebugPort 字段保存着用于接收调试事件的 LPC 端口对象指针。当有调试事件发生时，系统会向这个端口发送调试消息。因为 Windows XP 使用了新的专门用于调试的内核对象 DebugObject 代替 LPC 端口，所以在 Windows XP 下，

DebugPort 字段指向的是 DebugObject 对象。不论是指向 LPC 端口还是指向调试对象，尽管类型不同，但是它们的作用都是用来传递调试事件的。因此我们将 DebugPort 中所指向的对象统称为调试端口。

从调试对话的角度来讲，调试端口是联系调试器进程和被调试进程的纽带。被调试进程中的调试事件就是由这个端口发送到调试器进程的。

10.2.3 BeingDebugged 字段

进程环境块是每个 Windows 进程的另一个重要数据结构，与 EPROCESS 不同，PEB 结构是位于用户空间中的，而且其地址通常是位于用户空间的较高地址区域，例如 0x7FFDF000。

```
typedef struct _PEB
{
/*000*/ BOOLEAN InheritedAddressSpace;
/*001*/ BOOLEAN ReadImageFileExecOptions;
/*002*/ BOOLEAN BeingDebugged;
...
} PEB, *PPEB, **PPPEB;
```

如果一个进程不在被调试状态，那么其 PEB 结构的 BeingDebugged 字段为 0，否则为 1。IsDebuggerPresent() API 就是通过判断 BeingDebugged 字段实现的。其汇编指令如下：

```
0:001> uf kernel32!IsDebuggerPresent
kernel32!IsDebuggerPresent:
77e7276b 64a118000000 mov eax,fs:[00000018] // 取得当前线程的线程环境块结构 TEB
77e72771 8b4030      mov eax,[eax+0x30]    // 从 TEB 中取出 PEB 指针
77e72774 0fb64002   movzx eax,byte ptr [eax+0x2] // 取 PEB 中的 BeingDebugged 字段
77e72778 c3         ret
```

10.2.4 观察 DebugPort 字段和 BeingDebugged 字段

下面通过一个小实验来观察记事本程序在被调试前和被调试时 DebugPort 字段和 BeingDebugged 字段的变化。因为需要内核调试环境，而且 Windows 2000 不支持本地内核调试，所以我们以 Windows XP 为例。

先运行 notepad 程序，启动 WinDBG，并开始本地内核调试（File>Kernel Debug>Local）。

在 WinDBG 的命令提示符（lkd）后键入 !process 0 0，列出所有进程，找到关于 notepad.exe 进程的信息，并记录下它的 EPROCESS 结构的地址。

```
PROCESS 86f55648 SessionId: 0 Cid: 0eb8 Peb: 7ffdf000 ParentCid: 04d0
DirBase: 2db71000 ObjectTable: e2a828a8 HandleCount: 38.
Image: notepad.exe
```

然后使用 dt 命令显示 notepad 的 EPROCESS 结构的各个字段值：

```
1kd> dt nt!_EPROCESS 86f55648 // 省略了无关显示
+0x0bc DebugPort : (null)
+0x0c0 ExceptionPort : 0xe29fa040
```

可见，此时 DebugPort 字段的值为空，说明该进程还没有被调试。ExceptionPort 处已经有值，使用如下命令可以知道它是一个 LPC 端口的指针：

```
1kd> !lpc port 0xe29fa040
```

```
Server connection port e29fa040 Name: ApiPort // 参见 9.5.3 节
Handles: 1 References: 250
Server process : 87122da8 (csrss.exe) // CSRSS 在监听此端口，参加按 11.3.3
Queue semaphore : 8795d710
Semaphore state 0 (0x0)
The message queue is empty
The LpcDataInfoChainHead queue is empty
```

键入 .process 86f55648 命令，将 notepad 进程切换为默认进程。然后输入 !peb 命令观察 BeingDebugged 字段：

```
1kd>.process 86f55648 // 观察用户态的内存前，必须切换默认进程
Implicit process is now 86f55648
1kd> !peb
PEB at 7ffdf000 // 省略了无关显示
BeingDebugged: No
```

再运行一个 WinDBG 实例，附加到刚才的 notepad 进程，选择 File>Attach to a Process，然后键入 notepad 进程的 ID 或从列表中选取 ID，如果系统中有多个 notepad 实例，请注意不要选错。

再次使用 dt 命令显示 notepad 的 EPROCESS 结构：

```
1kd> dt nt!_EPROCESS 86f55648 // 省略了无关显示
+0x0bc DebugPort : 0x87aa58c8
+0x0c0 ExceptionPort : 0xe29fa040
```

可见，DebugPort 字段的值已经不再为 NULL 了，使用 !object 命令可以观察到这是一个 DebugObject 对象。

```
1kd> !object 0x87aa58c8
Object: 87aa58c8 Type: (87baa040) DebugObject
ObjectHeader: 87aa58b0
HandleCount: 1 PointerCount: 2
```

对于 Windows 2000，DebugPort 记录的是 LPC 端口对象，因此可以通过 !lpc port xxxxxxxx 命令来观察。

重复前面观察 PEB 的步骤或在第二个 WinDBG 中输入 !peb 命令，可以观察到被调试进程的 PEB 中的 BeingDebugged 字段已经为真。

```
0:001> !peb // 省略了无关显示
PEB at 7ffdf000
InheritedAddressSpace: No
ReadImageFileExecOptions: No
BeingDebugged: Yes
ImageBaseAddress: 01000000
```

10.2.5 调试会话

我们把调试器进程与被调试进程之间的交互称为调试会话（debugging session）。一次调试会话是从建立调试关系开始，直到这种关系解除为止。更准确地说，建立调试关系的标准是被调试进程与调试器进程之间通过调试端口建立的通信连接。调试关系解除的标准是被调试进程的调试端口被清除。

建立调试会话是调试过程的第一步，其建立方式包含两种情况：一种情况是在调试器中启动被调试程序（清单 10-2 的第 4、5 行），比如我们在 VC6 或 VS2005（Visual Studio 2005）等集成开发环境（IDE）中开始执行程序，或者在 WinDBG 中打开一个 EXE 程序。另一种情况是当开始调试时，被调试程序已经在运行，这时，需要把调试器附加（attach）到被调试进程上。后一种情况对于调试系统服务或 DLL 形式的各种插件非常有用。接下来的两节将分别介绍这两种情况。

10.3 从调试器中启动被调试程序

本节我们介绍创建调试会话的第一种情况，也就是从调试器进程中创建被调试进程并开始调试。简单来说，这种方式就是当调用创建进程 API（如 CreateProcess）时指定 DEBUG_PROCESS 或 DEBUG_ONLY_THIS_PROCESS 标志。

10.3.1 CreateProcess API

Windows 操作系统提供了几个 API 用于创建新的进程，比如 CreateProcess()、CreateProcessAsUser()、CreateProcessWithTokenW() 和 CreateProcessWithLogonW()。因为后面几个 API 可以看作是 CreateProcess API 的超集，所以我们就以 CreateProcess() API 为例来讨论。它的函数原型如下：

```
BOOL CreateProcess( LPCTSTR lpApplicationName, LPTSTR lpCommandLine,
    LPSECURITY_ATTRIBUTES lpProcessAttributes,
    LPSECURITY_ATTRIBUTES lpThreadAttributes,
    BOOL bInheritHandles, DWORD dwCreationFlags,
    LPVOID lpEnvironment, LPCTSTR lpCurrentDirectory,
    LPSTARTUPINFO lpStartupInfo, LPPROCESS_INFORMATION lpProcessInformation);
```

其中 dwCreationFlags 参数用于指定创建新进程的选项，可以是一系列标志位的组合。以下两个标志位是专门用于调试的：

```
#define DEBUG_PROCESS          0x00000001      // 调试正在创建的进程和它的子进程
#define DEBUG_ONLY_THIS_PROCESS 0x00000002      // 声明不要调试调试目标的子进程
```

系统在创建进程时，会检查创建标志中是否包含以上标志，如果包含，那么系统会把调用进程当作调试器（debugger）进程，把新创建的进程当作被调试程序（debuggee），为二者建立起调试关系，主要执行以下 3 个动作。

第一，在进程创建的早期（执行内核服务 NtCreateProcess/NtCreateProcessEx 之前），调用 DbgUiConnectToDbg() 使调用线程与调试子系统建立连接。在 Windows XP 以前，DbgUiConnectToDbg() 内部会调用 NtConnectPort() 与会话管理器（SMSS.EXE）的 \DbgUiApiPort 连接，如果连接成功，便将返回的端口句柄和用于等待端口数据的信号对象（Semaphore）分别放入当前线程环境块的 DbgSsReserved[1] 和 DbgSsReserved[0] 中保存起来。这两个指针有着重要作用，端口对象是与调试子系统进行通信的重要通道，信号对象用来等待调试事件，很多调试 API 都需要这两个指针才能工作，如 WaitForDebugEvent() 和 ContinueDebugEvent()。在 Windows XP 中，DbgUiConnectToDbg() 内部会调用 ZwCreateDebugObject 创建 DEBUG_OBJECT 内核对象，并将其保存在当前线程环境块的 DbgSsReserved[1] 字段中。需要说明的是，为了防止重复操作，DbgUiConnectToDbg() 在调用 NtConnectPort 或 ZwCreateDebugObject 之前会先检查 DbgSsReserved[1] 是否为空。完成这一步后，调用线程便由普通线程晋升为调试子系统“眼”里的调试器工作线程了。清单 10-3 显示了 WinDBG 的调试工作线程执行 CreateProcess API 和 DbgUiConnectToDbg 函数的过程。

清单 10-3 CreateProcess API 调用 DbgUiConnectToDbg 函数的过程

```
0:001> k
ChildEBP RetAddr
0140f198 7c842d54 ntdll!DbgUiConnectToDbg      [连接调试子系统]
0140fbcc 7c80235e kernel32!CreateProcessInternalW+0x12a2 [创建进程的内部函数]
0140fbfc 02241011 kernel32!CreateProcessW+0x2c   [创建进程 API]
0140fcbb 020c8109 dbgeng!LiveUserDebugServices::CreateProcessW+0x241
0140fcf8 0209667f dbgeng!LiveUserTargetInfo::StartCreateProcess+0xf9
0140fd44 01028bf5 dbgeng!DebugClient::CreateProcessAndAttach2Wide+0xcf
0140ffa4 0102913b windbg!StartSession+0x445    [开始调试会话]
0140ffb4 7c80b6a3 windbg!EngineLoop+0x1b       [调试事件循环]
0140ffec 00000000 kernel32!BaseThreadStart+0x37 [调试器工作线程的初始函数]
```

第二，当调用进程创建内核服务 NtCreateProcess 或 NtCreateProcessEx 时，将 DbgSsReserved[1] 字段中记录的对象句柄以参数（第 7 个参数）形式传递给内核中的进程管理器。接下来，内核中的进程创建函数（PspCreateProcess）会检查这个句柄是否为空，如果不为空，会取得它的对象指针，然后设置到进程执行块（EPROCESS 结构）的 DebugPort 字段中。因为系统内部是以 DebugPort 是否为 0 来判断一个进程是否在被调试，并向该端口发送调试消息的。所以完成这一步后，新创建进程便由普通的进程晋升为调试子系统“眼”里的被调试进程了。

清单 10-4 显示了使用内核调试器观察到的 WinDBG 的调试器工作线程在内核态中执行进程创建函数的过程。

清单 10-4 进程创建函数在内核中的执行过程

```
kd> k
ChildEBP RetAddr
f8740ce4 805909b2 nt!PspCreateProcess          [进程管理器中的进程创建函数]
f8740d38 804da140 nt!NtCreateProcessEx+0x7e    [内核服务]
```

```

f8740d38 7ffe0304 nt!KiSystemService+0xc4
00c3f1c4 77f75a0f SharedUserData!SystemCallStub+0x4
00c3f1c8 77e7fe05 ntdll!ZwCreateProcessEx+0xc [调用内核服务]
00c3fb04 77e61bb8 kernel32!CreateProcessInternalW+0x1111
...
[省略用户态的其它栈帧, 参见清单 10-2]
...

```

使用 dd 命令显示 PspCreateProcess 函数的栈帧和参数:

```

kd> dd f8740ce4
f8740ce4 00000286 805909b2 00c3f928 001f0fff
f8740cf4 00000000 ffffffff 00000006 0000020c
f8740d04 000001f4 00000000 00000000 f8740d64
...

```

其中第 7 个参数 (000001f4) 是调试端口句柄, 使用 !handle 观察到它对应的是 WinDBG 进程中的 DebugObject 对象:

```

kd> !handle 000001f4
processor number 0, process 81e4ba58
PROCESS 81e4ba58 SessionId: 0 Cid: 07fc Peb: 7ffdf000 ParentCid: 0710
DirBase: 0be39000 ObjectTable: e1923490 HandleCount: 190.
Image: windbg.exe

Handle table at e1954000 with 190 Entries in use
01f4: Object: 81e40740 GrantedAccess: 001f000f Entry: e19543e8
Object: 81e40740 Type: (81fc9778) DebugObject
ObjectHeader: 81e40728 (old version)
HandleCount: 1 PointerCount: 1

```

第三, 当 PspCreateProcess 调用 MmCreatePeb 函数创建新进程的进程环境块时, MmCreatePeb 函数内部会根据 EPROCESS 结构的 DebugPort 字段设置 BeingDebugged 字段。如果 DebugPort 不为空, 那么 BeingDebugged 会被设为真。

10.3.2 第一批调试事件

如果以上三步都成功结束 (CreateProcess API 成功返回), 那么调试器与被调试程序的调试对话便建立起来了。调试器线程接下来应该进入调试事件循环来接收调试事件。

一个新创建进程的初始线程是从内核中的 KiThreadStartup 开始执行的, KiThreadStartup 很简短, 在将线程的 IRQL 降低到 APC 级别后, 便将执行权交给了 PspUserThreadStartup 函数。

PspUserThreadStartup 函数内部会调用 DbgkCreateThread 向调试子系统通知新线程创建事件。于是如我们在上一章所讲的, 调试子系统会向调试事件队列中放入一个进程创建事件, 并等待调试器来处理和回复。

结合本节的上下文, 当调试器的工作线程等待调试事件时, 它会立刻收到进程创建事件。调试器收到进程创建事件后会为调试这个新的进程做一些准备工作, 而后它调用 ContinueDebugEvent 回复调试事件, 被调试进程开始继续执行。

新进程会在自己的上下文中执行一系列初始化工作, 包括映射和加载映像文件。因此, 调试器接下来会收到一系列加载 DLL (LOAD_DLL_DEBUG_EVENT) 事件。

清单 10-5 显示了在 WinDBG 调试器中启动 TinyDbge 小程序时 WinDBG 所输出的信息。

清单 10-5 WinDBG 调试器在建立调试会话后针对第一批调试事件的输出（节选）

```

1 *** Create process 1470 [创建进程成功, 1470 是进程 ID]
2 Symbol search path is: SRV*d:\symbols*http://msdl.microsoft.com/download/ symbols
3 Executable search path is: [未设置可执行文件搜索路径]
4 Process created: 1470.1f9c [收到进程创建事件]
5 OUTPUT_PROCESS: *** Create process *** [进程创建事件触发的输出信息]
6 id: 1470 Handle: 314 index: 0
7 id: 1f9c hThread: 334 index: 0 addr: 00401120
8 ModLoad: 00400000 0042c000 TinyDbge.exe [收到模块加载事件]
9 OUTPUT_PROCESS: *** Load dll *** [模块加载事件触发的信息输出]
10 hFile: 2f0 base: 00400000
11 ModLoad: 7c900000 7c9b0000 ntdll.dll [映射 NTDLL.DLL 的事件]
12 OUTPUT_PROCESS: *** Load dll ***
13 hFile: 308 base: 7c900000
14 ModLoad: 7c800000 7c8f5000 C:\WINDOWS\system32\kernel32.dll
15 OUTPUT_PROCESS: *** Load dll ***
16 hFile: 2bc base: 7c800000
17 (1470.1f9c): Break instruction exception - code 80000003 (first chance)
18 eax=00241eb4 ebx=7ffd6000 ecx=0 edx=1 esi=00241f48 edi=00241eb4
19 eip=7c901230 esp=0012fb20 ebp=0012fc94 iopl=0 nv up ei pl nz na po nc
20 cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000
21 efl=00000202
22 Loading symbols for 7c900000 ntdll.dll -> ntdll.dll
23 ntdll!DbgBreakPoint: [发生异常的函数]
24 7c901230 cc int 3 [触发异常的断点指令]

```

首先要看到清单 10-5 中的全部信息，需要按 **Ctrl+Alt+V** 或选中 **View** 菜单中的 **Verbose Output** 让 WinDBG “输出详细信息”。第 1 行显示调试器的工作线程成功调用 **CreateProcess API**，新进程的进程 ID 是 0x1470。第 2、3 行显示了当前的符号搜索路径和映像文件搜索路径。第 4 行显示的是接收到进程创建事件，实际上，也就是创建初始线程的事件，0x1f9c 是初始线程的线程 ID。第 5~7 行是打印当前被调试进程的统计信息，下面每收到一个调试事件，会再显示一次（第 9、10、12、13、15、16 行），为了节约篇幅，我们删除了重复的信息行。

第 8 行代表 WinDBG 收到第一个模块加载 (**LOAD_DLL_DEBUG_EVENT**) 事件，即程序的 EXE 模块 **TinyDbge.exe**，冒号后面的两个数字分别是模块的起始地址和结束地址。第 11 行和第 14 行分别是收到映射 **NTDLL.DLL** 和 **Kernel3.DLL** 事件。第 17 行表示收到初始断点异常事件，我们在下一小节讨论。

10.3.3 初始断点

当新进程的初始线程在自己的上下文中初始化时，作为进程初始化的一个步骤，**NTDLL.DLL** 中的 **LdrpInitializeProcess** 函数会检查正在初始化的进程是否处于被调试状态（查询进程环境块的 **BeingDebugged** 字段），如果是，它会调用 **DbgBreakPoint()** 触发一个断点异常，目的是中断到调试器。这实际上相当于系统在

新进程中为我们设置了一个断点，这个断点通常被称为初始断点。

清单 10-5 中的第 17~24 行便是调试器收到初始断点事件后所输出的信息。括号中是进程 ID 和线程 ID。0x80000003 是断点异常的代码，(first chance) 代表是异常的第一轮处理机会（第 11 章详细讨论）。第 18~21 行是发生断点时 CPU 中各个寄存器的值。从 EIP 指针的值 (eip=7c901230) 可以看出断点指令位于 NTDLL 模块中 (EIP 值介于 NTDLL 模块的起止地址 7c900000~7c9b0000 之间)。第 22 行显示调试器在为 NTDLL.DLL 加载符号文件，因为要寻找位于这个模块中的当前指令所对应的函数符号，即第 23 行所显示的信息。第 24 行是触发断点异常的指令地址、机器码和汇编语言表示。

执行栈回溯命令 k 可以看到初始线程调用 DbgBreakPoint 的完整过程。

```
0:000> k
ChildEBP RetAddr
0012fb1c 7c93edc0 ntdll!DbgBreakPoint
0012fc94 7c921639 ntdll!LdrpInitializeProcess+0xffa
0012fd1c 7c90eac7 ntdll!_LdrpInitialize+0x183
00000000 00000000 ntdll!KiUserApcDispatcher+0x7
```

可以看出是 NTDLL 中的 LdrpInitializeProcess 调用了 DbgBreakPoint。Ldr 是 Loader 的缩写，是 NTDLL 中的进程加载器系列函数的前缀。

当初始断点发生时，被调试程序自己的主函数还没开始执行，因此这个调试时间是很早的，对于调试在程序初始化阶段发生的问题是非常有意义的。

包括 MSDEV 调试器在内的一些调试器不会将初始断点报告给用户，WinDBG 默认会向用户报告初始断点，但是可以通过命令行参数-g 来忽略初始断点，即收到初始断点后立刻恢复执行，而不是停下来。

10.3.4 自动启动调试器

有时，要调试的进程可能是被系统或其他进程动态启动的。在调试器中执行它可能无法提供合适的参数和运行条件。而且，当我们发现它启动并将调试器附加到该进程时，需要调试的代码可能已经运行结束了。对于这种情况，可以通过在注册表中设置“映像文件执行选项 (Image File Execution Options)”来让操作系统先启动调试器，然后再从调试器中启动目标进程。

Windows 系统在创建进程时，会在注册表中查询如下注册表表键来读取关于这个程序的执行选项：

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options
```

查询的方法就是看以上表键下是否存在以要执行的映像文件名（不包含路径）命名的子键。如果有这样的子键，那么系统会继续查询该子键是否存在名为“Debugger”的键值。

如果 Debugger 键值存在而且包含内容（参见图 10-1），那么系统便会先将当前的命令行附加到 Debugger 键值所定义的内容之后，再将此作为新的命令行来启动。

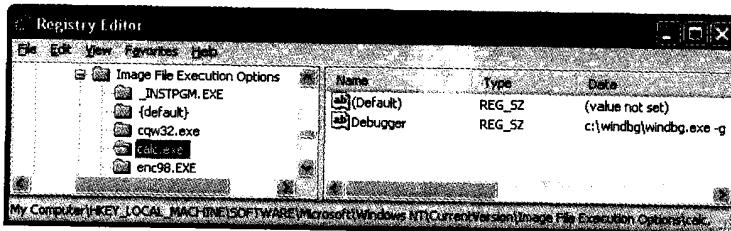


图 10-1 通过注册表设置自动启动调试器

举例来说，我们先在 Image File Execution Options 表键下建立一个名为 calc.exe（计算器程序）的子键，并在其下加入 Debugger 键值，内容如下：

```
C:\WinDBG\WinDBG.exe -g
```

也就是 WinDBG 调试器的主程序的全路径，加上-g 开关，含义是忽略初始断点，让被调试进程启动后就继续运行，而不是直接中断到调试器。如果希望新进程启动后就中断到调试器，那么去掉-g 开关。

完成以上修改后，再以某种方式（开始菜单或运行 calc.exe）启动计算器程序，就会发现系统会自动启动 WinDBG，因为系统将启动 calc.exe 的完整路径以命令行参数的形式传给了 WinDBG，所以 WinDBG 启动后会立即启动计算器程序。

如果系统启动 Debugger 键值指定的调试器失败，比如我们将刚才的 WinDBG 路径改为错误的值，那么系统仍会显示无法发现原来的映像文件（见图 10-2）。

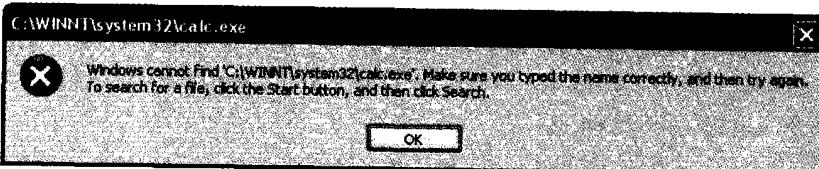


图 10-2 系统启动执行选项指定的调试器失败后显示的错误信息

讲到这里，大家可能会想到下一个有趣的问题，如果为 Debugger 键值中定义的调试器映像再定义 Debugger 执行选项会怎么样呢？比如我们再加入一个名为 WinDBGexe 的子键，并加入 Debugger 键值，其内容为：vsjitdebugger.exe（Visual Studio .NET 2003 的 JIT 调试器）。完成这一修改后，再运行计算器程序，我们得到图 10-3 所示的对话框。尽管这是个错误提示，但仔细观察提示中的命令行，可以发现系统是在按我们估计的递归方式工作的，启动 calc.exe 时根据执行选项先启动 WinDBGexe，启动 WinDBGexe 时执行选项又要先启动 vsjitdebugger.exe。但 vsjitdebugger.exe 出错了，于是就“搁置”在这了。这颇有点“螳螂捕蝉，黄雀在后”的意味（见图 10-3）。

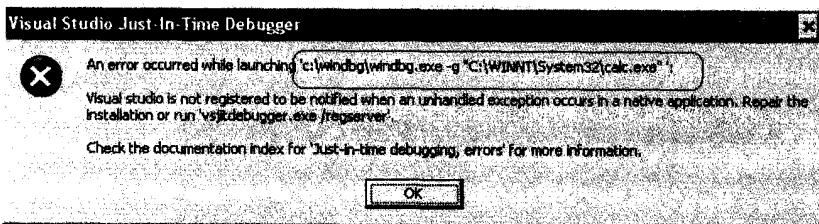


图 10-3 “螳螂捕蝉，黄雀在后”的画面

根据以上逻辑，如果 Debugger 键值指定的仍然是同名的一个程序，那么会出现“死循环”。笔者试验的结果是，系统忙碌了数秒钟后，出现类似图 10-2 的错误对话框。

10.4 附加到已经启动的进程

本节介绍建立调试会话的另一种情况，也就是当要调试的程序已经运行时是如何建立调试对话的，即如何将调试器附加（attach）到已经运行的进程。简单来说，这是通过 `DebugActiveProcess()` API 来完成的。

10.4.1 DebugActiveProcess API

`DebugActiveProcess` API 的原型非常简单，即 `BOOL DebugActiveProcess (DWORD dwProcessId)`。也就是只将被调试进程的进程 ID 传递给 `DebugActiveProcess` API，系统便会将这个 API 的进程与 `dwProcessId` 参数指定的进程建立起调试关系。下面以 Windows XP 系统中的情况为例，介绍 `DebugActiveProcess` API 的内部工作过程。

第一，通过 `DbgUiConnectToDbg()` 使调用进程与调试子系统建立连接，实质上就是获得一个调试通信对象并存放在当前线程 TEB 结构的 `DbgSsReserved` 数组中。这与调试一个新进程的第一步相同。

第二，调用 `ProcessIdToHandle` 函数，获得指定进程 ID 的进程句柄，这个函数内部会调用 `OpenProcess()` API，进而调用 `NtOpenProcess` 内核服务。在执行这一步时需要调用进程，它与目标进程有同样或更高的权限，否则这一步便会失败，调用 `GetLastError()` 返回的错误码通常是 5，意思是“Access is denied.”，即访问被拒绝。如果调试器进程具有 `SE_DEBUG_NAME` 权限，那么它通常有权限调试系统内的任何进程。

第三，调用 NTDLL 中的 `DbgUiDebugActiveProcess` 函数。这个函数内部主要是调用 `NtDebugActiveProcess` 内核服务，并将要调试进程的句柄（参数 1）和调试对象的句柄（参数 2）作为参数传递给这个内核服务。`NtDebugActiveProcess` 内部主要执行以下三个动作：(1) 根据参数中指定的句柄取得被调试进程的 `EPROCESS` 结构和调试对象的对象指针。(2) 向调试对象发送杜撰的调试事件（参见 9.4.5 节）。(3) 调用 `DbgkpSetProcessDebugObject` 函数，这个函数内部会将调试对象设置到被调试进程

的调试端口(DebugPort 字段),并调用 DbgkpMarkProcessPeb 来设置 BeingDebugged 字段。清单 10-6 显示了一个名为 TinyDbgr 的调试器调用 DebugActiveProcess API 和执行 NtDebugActiveProcess 函数的过程。

清单 10-6 设置被调试进程的 DebugPort 字段和 BeingDebugged 字段

```
kd> knL
# ChildEBP RetAddr
00 f4b92cfc 805bb048 nt!DbgkpMarkProcessPeb          // 标记 PEB
01 f4b92d2c 805bcf39 nt!DbgkpSetProcessDebugObject+0x222 // 设置调试端口
02 f4b92d54 804da140 nt!NtDebugActiveProcess+0x9a        // 系统服务
03 f4b92d54 7ffe0304 nt!KiSystemService+0xc4           // 系统服务分发函数
04 0012fdf8 77f75a96 SharedUserData!SystemCallStub+0x4
05 0012fdfc 77f8f4db ntdll!ZwDebugActiveProcess+0xc      // NTDLL 中的系统服务残根
06 0012fe10 77eaeaae ntdll!DbgUiDebugActiveProcess+0x18 // NTDLL 中的 DbgUi 函数
07 0012fe20 0040149a kernel32!DebugActiveProcess+0x2e    // 附加到已运行进程的 API
08 0012ff80 00401f69 TinyDbgr!main+0xfa                 // 主函数
09 0012ffc0 77e814c7 TinyDbgr!mainCRTStartup+0xe9       // 编译器插入的启动函数
0a 0012fff0 00000000 kernel32!BaseProcessStart+0x23     // 线程启动函数
```

在 NtDebugActiveProcess 成功返回后, DbgUiDebugActiveProcess 会调用 DbgUiIssueRemoteBreakin, 目的是在远程进程中创建远程中断线程(10.6.4 节将详细介绍), 使被调试进程中断到调试器中。

以上操作都成功后, DebugActiveProcess() 会返回真, 通知调用进程已经成功建立调试对话。接下来调试器便进入调试事件循环开始接收和处理调试事件了。它首先会接收到一系列杜撰的调试事件, 包括进程创建、模块加载等。最后收到远程中断线程产生的断点事件, 调试器收到这一事件后, 通常会停下来报告给用户。

DebugActiveProcess API 在 Windwos 2000 中的工作过程与上面介绍的类似, 较大的差异是 DbgUiDebugActiveProcess 函数, 它内部的主要操作不再是调用系统服务, 而是调用 CSRSS 的 CsrDebugPorcess 服务(参见 9.5.6 节)。

至此, 我们比较详细地介绍了目标程序已经运行和还没有运行两种情况下调试对话的建立过程。下面通过一个演示程序来进一步理解这些内容。

10.4.2 示例: TinyDbgr 程序

为了演示用户态调试的工作原理, 我们使用 C++语言编写了一个小型的 Windows 调试器程序, 名为 TinyDbgr, 意思是微型调试器。清单 10-7 给出了该程序的主函数和 Help 函数的源代码。

清单 10-7 TinyDbgr 程序的主函数和 Help 函数

```
void Help()
{
    printf ( "TinyDbgr <PID of Program to Debug>|\n"
             "<Full Exe File Name> [Prgram Parameters]|\n" );
}
```

```

int main(int argc, char* argv[])
{
    if(argc<=1)
    {
        Help();  return -1;
    }
    if (strstr(strupr(argv[1]), ".EXE"))
    {
        TCHAR szCmdLine[ MAX_PATH ] ;
        szCmdLine[ 0 ] = '\0' ;

        for ( int i = 1 ; i < argc ; i++ )
        {
            strcat ( szCmdLine , argv[ i ] ) ;
            if ( i < argc )
            {
                strcat ( szCmdLine , " " ) ;
            }
        }
        if(!DbgNewProcess(szCmdLine))
        {
            return -2;
        }
    }
    else
    if(!DebugActiveProcess(atoi(argv[1])))
    {
        printf("Failed in DebugActiveProcess() with %d.\n",GetLastError());
        return -2;
    }
    return DbgMainLoop();
}

```

从以上代码可以看出，TinyDbgr 需要有至少一个命令行参数，如果没有，就会显示帮助信息。

TinyDbgr 支持两种参数，分别适用于调试已经运行的程序和尚未运行的程序。如果 TinyDbgr 发现第一个参数中不包含 “.EXE”，那么便认为要调试一个已经运行的进程，该参数是要调试进程的进程 ID（十进制整数）；否则，TinyDbgr 便会把所有参数当作一个命令行，并试图通过该命令行启动和调试该程序。

让我们先就第一种情况做个实验。启动计算器程序（开始菜单选运行，在运行对话框中键入 calc 后回车）。使用任务管理器找到计算器程序的进程 ID，当笔者做实验时为 4752。启动控制台窗口，并转到本书附带代码的 bin\debug 目录，然后键入 tinydbgr 4752（4752 应该换作具体的进程 ID）。接下来应该看到 TinyDbgr 迅速打印出如下结果：

```

Debug event received from process 4752 thread 3632: CREATE_PROCESS_DEBUG_EVENT.
Debug event received from process 4752 thread 3632: LOAD_DLL_DEBUG_EVENT.
[省略多行与上面一行一样的 LOAD_DLL_DEBUG_EVENT 事件]
Debug event received from process 4752 thread 2672: CREATE_THREAD_DEBUG_EVENT.
Debug event received from process 4752 thread 2672: EXCEPTION_DEBUG_EVENT.
-Debuggee breaks into debugger; press any key to continue.

```

第一行是创建进程的消息，正如我们前面所介绍的，尽管该进程早已经创建，但是调试子系统会模拟出这个以前发生的事件以帮助调试器补充历史信息。接下来是一

系列加载或映射 DLL 的事件，这些也是调试子系统模拟出的历史事件。倒数第三行的 CREATE_THREAD_DEBUG_EVENT 事件来自一个新的线程，即 DbgUiIssueRemoteBreakin 所创建的远程线程。最后一行是远程中断线程触发的断点异常 (EXCEPTION_DEBUG_EVENT)。

此时试图操作计算器程序，发现无法将其激活，因为它已经中断到调试器中了，也就是被调试子系统挂起了。按任意键让调试器恢复，并让被调试程序继续运行，显示的是：

```
Debug event received from process 4752 thread 2672: EXIT_THREAD_DEBUG_EVENT.
```

这是因为远程中断线程退出了。这时，计算器程序可以使用了。

退出计算器程序，显示的是：

```
Debug event received from process 4752 thread 3632: EXIT_PROCESS_DEBUG_EVENT.
```

即被调试进程退出了。

下面再试验另一种情况，输入 tinydbgr calc.exe 让 TinyDbgr 启动一个新的计算器进程并开始调试，显示结果如下：

```
Debug event received from process 1584 thread 5196: CREATE_PROCESS_DEBUG_EVENT.
Debug event received from process 1584 thread 5196: LOAD_DLL_DEBUG_EVENT.
[省略 8 行与上面一行一样的 LOAD_DLL_DEBUG_EVENT 事件]
Debug event received from process 1584 thread 5196: EXCEPTION_DEBUG_EVENT.
-Debuggee breaks into debugger, press any key to continue.
```

这便是第 10.3.3 节介绍的因为 LdrpInitializeProcess 函数调用 DbgBreakPoint 而产生的初始断点。

按任意键恢复程序运行，会看到类似如下的输出。

```
Debug event received from process 1584 thread 5196: LOAD_DLL_DEBUG_EVENT.
[省略 7 行与上面一行一样的 LOAD_DLL_DEBUG_EVENT 事件]
Debug event received from process 1584 thread 5196: UNLOAD_DLL_DEBUG_EVENT.
```

这一行是反映射 (卸载) DLL 的消息，FreeLibrary() 可能会导致此动作。

与已经运行程序建立调试对话的一个典型的应用，就是在一程序发生错误（未处理异常）即将被系统终止时，将调试器附加到这个进程，即所谓的 JIT 调试，我们将在第 12 章对其详细讨论。

10.5 处理调试事件

上一章介绍 Windows 调试模型的基本特征时 (9.1.2 节)，我们说过 Windows 的用户态调试是通过调试事件来驱动的。而后，介绍了调试事件（消息）的采集和传递过程。本节我们将介绍调试器是如何读取和处理调试事件的。

10.5.1 DEBUG_EVENT 结构

在调试 API 一层, Windows 使用名为 DEBUG_EVENT 的结构体来表示调试事件, 该结构的定义如下:

```
typedef struct _DEBUG_EVENT {
    DWORD dwDebugEventCode;           //事件代码
    DWORD dwProcessId;               //发生调试事件的进程 ID
    DWORD dwThreadId;                //发生调试事件的线程 ID
    union {
        EXCEPTION_DEBUG_INFO Exception; //异常事件的详细信息
        CREATE_THREAD_DEBUG_INFO CreateThread; //线程创建事件的详细信息
        CREATE_PROCESS_DEBUG_INFO CreateProcessInfo; //进程创建事件的详细信息
        EXIT_THREAD_DEBUG_INFO ExitThread; //线程退出事件的详细信息
        EXIT_PROCESS_DEBUG_INFO ExitProcess; //进程退出事件的详细信息
        LOAD_DLL_DEBUG_INFO LoadDll; //映射 DLL 事件的详细信息
        UNLOAD_DLL_DEBUG_INFO UnloadDll; //反映射 DLL 事件的详细信息
        OUTPUT_DEBUG_STRING_INFO DebugString; //输出调试字符串事件的详细信息
        RIP_INFO RipInfo; //内部错误事件的详细信息
    } u;
} DEBUG_EVENT, *LPDEBUG_EVENT;
```

其中 dwDebugEventCode 用来标识调试事件的类型, 其值是表 10-1 所示的 9 种调试事件类型常量中的一个。dwProcessId 为发生调试事件进程的进程 ID, 当调试器同时调试多个进程时, 可以使用该 ID 来区分调试事件来自哪个进程。dwThreadId 为发生调试事件线程的线程 ID, 该线程属于 dwProcessId 所指定的进程。接下来是一个联合体 (union), 定义了描述 9 种事件详细信息的 9 个结构体。因为是联合结构, 所以应该根据 dwDebugEventCode 决定实际包含的是哪个结构, 其对应关系如表 10-1 的第 4 列所示。

表 10-1 调试事件的类型

事件类型 (dwDebugEventCode)	值	说明	详细信息所使用的结构
EXCEPTION_DEBUG_EVENT	1	异常	EXCEPTION_DEBUG_INFO
CREATE_THREAD_DEBUG_EVENT	2	创建线程	CREATE_THREAD_DEBUG_INFO
CREATE_PROCESS_DEBUG_EVENT	3	创建进程	CREATE_PROCESS_DEBUG_INFO
EXIT_THREAD_DEBUG_EVENT	4	线程退出	EXIT_THREAD_DEBUG_INFO
EXIT_PROCESS_DEBUG_EVENT	5	进程退出	EXIT_PROCESS_DEBUG_INFO
LOAD_DLL_DEBUG_EVENT	6	映射 DLL	LOAD_DLL_DEBUG_INFO
UNLOAD_DLL_DEBUG_EVENT	7	反映射 DLL	UNLOAD_DLL_DEBUG_INFO
OUTPUT_DEBUG_STRING_EVENT	8	输出调试信息	OUTPUT_DEBUG_STRING_INFO
RIP_EVENT	9	内部错误	RIP_INFO

第 4 列中的结构在 SDK 中有详细的定义和说明, 本书从略。

10.5.2 WaitForDebugEvent API

Windows 设计了 WaitForDebugEvent API 来供调试器等待和接收调试事件。这个 API 是实现在 Kernel32.DLL 中的。

```
BOOL WaitForDebugEvent( LPDEBUG_EVENT lpDebugEvent, DWORD dwMilliSeconds);
```

第一个参数是一个指向 DEBUG_EVENT 结构的指针，用来保存收到的调试事件。第二个参数用来指定要等待的毫秒数，或者使用 INFINITE 常量（0xFFFFFFFF），意思是无限期等待。

调用 WaitForDebugEvent() 会导致所在线程阻塞，直到有调试事件发生，或等待时间已到或发生错误才返回。这也是大多数调试器都使用多线程的原因，可使用其他线程处理 UI 更新和用户对话。

WaitForDebugEvent 内部主要完成两项任务，一是调用 NTDLL.DLL 中的 DbgUiWaitStateChange 函数，二是将这个函数返回的以 DBGUI_WAIT_STATE_CHANGE 结构表示的调试事件转化为 DEBUG_EVENT 结构。从 Windows XP 开始，转化工作是调用 NTDLL.DLL 中新增的 DbgUiConvertStateChangeStructure 函数来完成的。

DBGUI_WAIT_STATE_CHANGE 结构的定义如下：

```
typedef struct _DBGUI_WAIT_STATE_CHANGE {
    DBG_STATE NewState;                                // 枚举常量，代表新的调试状态
    CLIENT_ID AppClientId;                            // 结构体，包含进程和线程句柄
    union {                                            // 描述详细信息的联合结构
        DBGKM_EXCEPTION Exception;                     // 异常
        DBGUI_CREATE_THREAD CreateThread;              // 创建线程
        DBGUI_CREATE_PROCESS CreateProcessInfo;        // 创建进程
        DBGKM_EXIT_THREAD ExitThread;                  // 线程退出
        DBGKM_EXIT_PROCESS ExitProcess;                // 进程退出
        DBGKM_LOAD_DLL LoadDll;                       // 映射模块
        DBGKM_UNLOAD_DLL UnloadDll;                   // 反映射模块
    } StateInfo;
} DBGUI_WAIT_STATE_CHANGE, *PDBGUI_WAIT_STATE_CHANGE;
```

其中 NewState 是一个枚举常量，其定义如下：

```
typedef enum _DBG_STATE {
    DbgIdle,     DbgReplyPending,   DbgCreateThreadStateChange,
    DbgCreateProcessStateChange,  DbgExitThreadStateChange,
    DbgExitProcessStateChange,   DbgExceptionStateChange,
    DbgBreakpointStateChange,    DbgSingleStepStateChange,
    DbgLoadDllStateChange,      DbgUnloadDllStateChange
} DBG_STATE, *PDBG_STATE;
```

DBGUI_WAIT_STATE_CHANGE 结构名中的 DBGUI 代表这是调试子系统向外提供的接口结构。在内核调试中，有一个和它作用类似的结构叫 DBGKD_WAIT_STATE_CHANGE，我们将在第 18 章介绍。相对而言，DEBUG_EVENT 是更高一层的结构，即用户 API 一层。

上一章我们介绍过，内核中是使用 DBGKM_APIMSG 结构来描述调试事件的，调试 API 一层是使用 DEBUG_EVENT 结构来描述的，而 DbgUi 函数是使用 DBGUI_WAIT_

STATE_CHANGE 结构的，图 10-4 显示了这些结构的使用场合和转换函数。

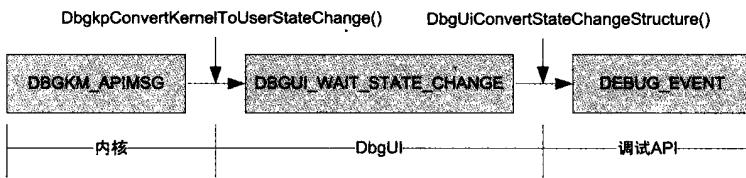


图 10-4 调试事件在不同层次的表示（从 Windows XP 开始）

10.5.3 调试事件循环

在对调试事件有比较深入的了解之后，接下来的问题是调试器应该如何接收和处理调试事件，这是设计调试器的一个主要工作。先让我们看一下 TinyDbgr 程序的简单调试事件循环（见清单 10-8）。

清单 10-8 TinyDbgr 程序的简单调试事件循环

```

1  BOOL DbgMainLoop(DWORD dwWaitMS)
2  {
3      DEBUG_EVENT DbgEvt;           // 用来读取调试事件的数据结构
4      DWORD dwContinueStatus = DBG_CONTINUE; // 恢复继续执行用的状态代码
5      BOOL bExit=FALSE;           // 退出标志
6
7      while(!bExit)
8      {
9          // 等待调试事件发生，第 2 个参数是等待的毫秒数，如果指定为 INFINITE，那么直到
10         // 有调试事件发生时这个函数才返回
11         if(!WaitForDebugEvent(&DbgEvt, dwWaitMS))
12         {
13             printf("WaitForDebugEvent() returned False %d.\n", GetLastError());
14             bExit=TRUE;
15             continue;
16         }
17
18         // 以下是处理调试事件的代码
19         printf("Debug event received from process %d thread %d: %s.\n",
20                 DbgEvt.dwProcessId, DbgEvt.dwThreadId,
21                 DbgEventName[DbgEvt.dwDebugEventCode>
22                         MAX_DBG_EVENT?MAX_DBG_EVENT:
23                         DbgEvt.dwDebugEventCode-1]);
24         switch (DbgEvt.dwDebugEventCode)
25         {
26             case EXCEPTION_DEBUG_EVENT:
27                 // 处理异常事件，需要设置继续参数(dwContinueStatus), ContinueDebugEvent
28                 // 函数需要这个参数
29                 printf("-Debuggee breaks into debugger; press any key to
30                         continue.\n");
31                 getchar();
32                 switch (DbgEvt.u.Exception.ExceptionRecord.ExceptionCode)
33                 {
34                     case EXCEPTION_ACCESS_VIOLATION:
35                         // 访问违例，对第一轮机会不处理，对最后一轮机会显示错误信息
36                         break;
37                     case EXCEPTION_BREAKPOINT:
38                         // 断点异常，对第一轮机会显示当前指令和寄存器值
39                 }
40             }
41         }
42     }
43 }

```

```
38         break;
39     case EXCEPTION_DATATYPE_MISALIGNMENT:
40         // 内存对齐异常，对第一轮机会不处理，对最后一轮机会显示错误信息
41         break;
42     case EXCEPTION_SINGLE_STEP:
43         // 单步和硬件断点异常，对于第一轮机会显示当前指令和寄存器值
44         break;
45     case DBG_CONTROL_C:
46         // Ctrl+C，对第一轮机会不处理，对最后一轮机会显示错误
47         break;
48     default:
49         // 处理错误
50         break;
51     }
52     case CREATE_THREAD_DEBUG_EVENT: //线程创建事件
53     // 根据需要可以调用 GetThreadContext 和 SetThreadContext
54     // API 来分析和设置寄存器，调用 SuspendThread 和 ResumeThread API 来
55     // 挂起和恢复线程
56     break;
57     case CREATE_PROCESS_DEBUG_EVENT: //进程创建事件
58     // 根据需要可以调用 GetThreadContext 和 SetThreadContext
59     // API 来访问进程的初始线程，调用 ReadProcessMemory 和
60     // WriteProcessMemory API 来访问内存
61     break;
62     case EXIT_THREAD_DEBUG_EVENT: //线程退出事件
63     // 显示线程的退出代码
64     break;
65     case EXIT_PROCESS_DEBUG_EVENT: //进程退出事件
66     // 显示进程的退出代码
67     bExit=TRUE;
68     break;
69     case LOAD_DLL_DEBUG_EVENT: // 加载模块事件
70     break;
71     case UNLOAD_DLL_DEBUG_EVENT: // 模块卸载事件
72     // 显示信息
73     break;
74     case OUTPUT_DEBUG_STRING_EVENT:
75     // 从被调试进程读取调试信息字符串，参见 10.7 节
76     break;
77   }
78   // 恢复被调试进程运行
79   ContinueDebugEvent (DbgEvt.dwProcessId,
80                       DbgEvt.dwThreadId, dwContinueStatus);
81 }
82 return TRUE;
83 }
```

清单 10-8 中的 `DbgMainLoop` 函数演示了如何等待和接收调试事件，其代码主要来自 MSDN 中关于编写调试器主循环（Writing the Debugger's Main Loop）的示例。尽管它已经用了 80 多行代码（包括注释），但是大家可以看到它还没有真正处理各个调试事件，它只是简单地打印出每个调试事件的主要信息（供我们试验使用）。对于一个事件，调试器通常有以下几种处理方式。

- 什么也不做或只是更新内部状态，用户察觉不到有调试事件发生。比如对 C++ 异常的第一轮处理机会，调试器的默认行为是什么都不做，让系统继续分发异常。

- 中断给用户，开始交互式调试，直到用户发出继续运行命令（F5 或 g 等）。例如，当接收到断点异常时，调试器总是会中断给用户。
- 输出提示信息。

在处理调试事件后，调试器会调用 `ContinueDebugEvent` API 来回复调试事件，并让被调试程序继续运行（第 84、85 行）。

10.5.4 回复调试事件

调试器在处理好调试事件后，应该调用 `ContinueDebugEvent` API 来向调试子系统回复处理结果：

```
BOOL ContinueDebugEvent( DWORD dwProcessId, DWORD dwThreadId,
    DWORD dwContinueStatus);
```

`dwProcessId` 和 `dwThreadId` 即收到的调试事件（`DEBUG_EVENT`）中所包含的进程 ID 和线程 ID。`dwContinueStatus` 可以为 `DBG_CONTINUE` (`0x00010002L`) 和 `DBG_EXCEPTION_NOT_HANDLED` (`0x00010001L`) 两个常量之一。对于异常事件（`EXCEPTION_DEBUG_EVENT`）之外的其他所有事件，这两个常量没有差异，调试子系统收到后，都会恢复被调试进程（调用 `DbgkpResumeProcess`）运行。对于异常事件，其差异如下。

`DBG_CONTINUE` 表示调试器处理了该异常。`DbgkForwardException` 函数（异常事件便是该函数接收并传给调试子系统的）收到此返回值后会向它的调用者（`KiDispatchException`）返回真，`KiDispatchException` 看到 `DbgkForwardException` 函数返回真后，便知道调试器处理了该异常，于是结束对该异常的分发过程。

`DBG_EXCEPTION_NOT_HANDLED` 表示调试器不处理该异常。该回复会导致 `DbgkForwardException` 返回假给 `KiDispatchException`。我们前面提到过，`KiDispatchException` 会多次调用 `DbgkForwardException`，分别为了不同轮次的处理机会。对于第一轮处理机会，`KiDispatchException` 收到 `DbgkForwardException` 返回假后，会继续分发过程，通常是寻找异常处理块（详见第 11 章）。对于第二轮处理机会，`KiDispatchException` 收到 `DbgkForwardException` 返回假后，会再次调用 `DbgkForwardException` 发给异常端口（Exception Port）。如果这次调用也返回假，则终止该进程，但这通常不会发生。因为对于第二轮处理机会，调试器默认会返回 `DBG_CONTINUE`，也就是“假装”处理了！这会导致异常分发过程结束，产生异常的代码被再次执行，于是又发生异常，如此反复不断。另外，即使调试器对于第二轮机会返回 `DBG_EXCEPTION_NOT_HANDLED`，因为异常端口指定的通常是 Windows 子系统服务器进程（`CSRSS`）监听的 `Windows\Apiport` 端口，`CSRSS` 收到此消息后，就会强行终止该进程（`End Program`），并向 `KiDispatchException` 回复真。

10.5.5 定制调试器的事件处理方式

大多调试器都允许调试人员随时修改对每个调试事件的处理和回复方式。比如，当使用 VC6 调试时，其 Debug 菜单中便有一个 Exceptions 项，选择该项便会得到图 10-5 所示的异常处理方式对话框。

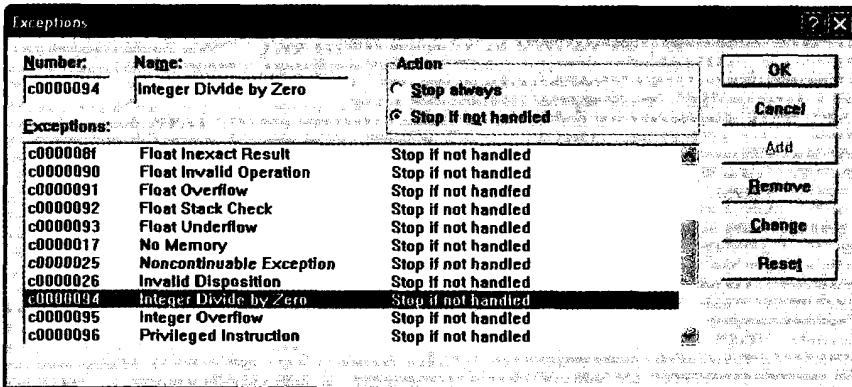


图 10-5 VC6 的异常处理方式对话框

对于列表中的每种异常，VC6 允许有如下两种选项。

Stop always，该项的含义是一旦接收到该异常，不论是第一轮处理机会还是第二轮处理机会，都中断（停止）到调试器。如果是第一轮处理机会，当用户按下 F5（让被调试进程继续运行）时，VC6 会弹出图 10-6 所示的询问对话框。如果选择 No，那么 VC6 会回复 DBG_CONTINUE (ContinueDebugEvent)，声明已经处理该异常。如果选择 Yes，那么 VC6 会回复 DBG_EXCEPTION_NOT_HANDLED，也就是自己没有处理该异常，让系统继续分发异常（传递给应用程序）。如果是第二轮处理机会，当用户按下 F5 继续时，VC6 不会询问，直接返回 DBG_CONTINUE。

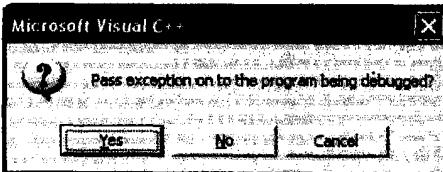


图 10-6 VC6 询问如何回复第一轮处理机会

Stop if not handled，该项的含义是对于第一轮处理机会直接返回 DBG_EXCEPTION_NOT_HANDLED。对于第二次处理机会才中断到调试器。直接返回的含义是 VC6 收到此异常会不做任何处理就直接返回 DBG_EXCEPTION_NOT_HANDLED，让系统继续分发此异常。因为只有当没有异常处理块处理一个异常时，才会有第二轮分发，这也正是这个选项如此命名的原因，即“如果没有被（应用程序的异常处理块）处理才停止到调试器（Stop if not handled）”。

对于大多数异常，VC6 的默认设置都是 Stop if not handled，即当得到第二轮处理机会时才停止到调试器中。但断点事件和单步执行事件除外。

可以使用清单 10-9 所示的小程序来加深对以上内容的理解。使用 VC6 打开 EvtFilter 项目（code\chap10\evtfilter），开始调试（Build>Start Debug>Go），在运行到第 8 行等待输入时，通过 Debug>Exception 菜单打开图 10-5 所示的异常处理方式对话框，从列表框中选择 Integer Divide by Zero，然后将 Action 切换到 Stop Always，再点击 Change 按钮。确认列表框中的 Integer Divide by Zero 行已经是 Stop Always 后，点击 OK 关闭对话框。在运行 EvtFilter 程序的控制台窗口中按任意键，使其继续。

清单 10-9 用来试验调试器异常处理方式的小程序

```

1 #include "stdafx.h"
2 #include <windows.h>
3
4 int main(int argc, char* argv[])
5 {
6     int i,m=1,n;
7     printf("Hello World!\n");
8     getchar();
9
10    __try
11    {
12        n=0;
13        i=m/n;
14    }
15    __except(EXCEPTION_EXECUTE_HANDLER)
16    {
17        OutputDebugString("I got the exception.\n");
18        return -1;
19    }
20    return 0;
21 }
```

运行到第 13 行时，因为除数是 0，所以一定会发生异常。由于是在调试器中运行，所以该异常会被发送给调试器。VC6 收到异常事件后，根据 dwDebugEventCode 和 Exception 结构中的异常代码（DbgEvt.u.Exception.ExceptionRecord.ExceptionCode）可以判断出是整数除 0 异常。然后 VC6 在项目的配置信息中查询该异常的处理方式，在发现是 Stop Always 后，它会准备中断给用户，因为是第一轮处理机会就中断给用户，所以 VC6 会显示一个图 10-7 所示的对话框。

按确定后，VC6 会显示出当前的代码（第 13 行）。这时，如果不做任何改动直接按 F5 继续，那么 VC6 会弹出图 10-6 所示的对话框，询问返回 DBG_CONTINUE（如果选 No）或 DBG_EXCEPTION_NOT_HANDLED（如果选 Yes）。返回 DBG_CONTINUE 会导致异常代码重新执行，于是再次发生异常，VC6 再次显示图 10-7 所示的对话框。返回 DBG_EXCEPTION_NOT_HANDLED 会导致系统继续分发该异常，第 17~18 行的异常处理块被执行。

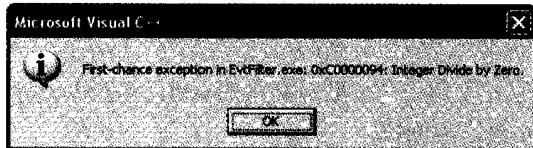


图 10-7 VC6 告诉用户这是第一轮处理机会

如果我们在第一轮处理机会时，在 VC6 中将变量 n 改为 1（或其他非零值）（通过变量观察窗口），然后按 F5，并对图 10-6 所示的对话框选 No（已经处理了异常情况，不必继续分发异常），那么第 13 行也会被重新执行，但是此时导致异常的错误情况已经消除，不会再发生异常，程序便顺利执行而正常退出了。

WinDBG 允许用户定制包括异常事件在内的所有调试事件的处理方式，我们将在第 30 章详细讨论。

10.6 中断到调试器

我们在介绍 `DbgkpSendApiMessage` 函数时（第 9.3.2 节），该函数代表调试子系统把调试事件发给调试器之前会调用 `DbgkpSuspendProcess` 挂起当前进程（被调试进程）。这样做是为了防止被调试进程继续运行会发生状态变化，给分析和观察带来困难。从被调试进程的角度来看，一旦它被调试子系统所挂起，那么它便“戛然而止”了，代码（用户态的应用程序代码）停止执行，一切状态都被冻结起来，在调试领域，我们将这种现象称为“中断到调试器（break into debugger）”。从调试器的角度讲，又叫将被调试进程“拉进调试器（bring debugger in）”。

既然被调试进程内发生调试事件时，它就会中断到调试器，因此，触发调试事件很自然地成为将被调试进程中断到调试器的一种基本途径。由于断点事件很容易被触发，所以在被调试进程中触发断点异常便成为被调试进程中断到调试器的最常用方法。下面我们先介绍被调试进程中断到调试器的典型方法，然后介绍几个有关的问题。

10.6.1 初始断点

如我们在 10.3.3 节所介绍的，一个新创建进程的初始线程在初始化时会检查当前进程是否在被调试，如果是，那么便调用 NTDLL 中的 `DbgBreakPoint` 函数，触发一个断点异常，使新进程中断到调试器中。这通常是当调试器开始调试一个新创建进程时接收到的第一个断点异常，因此称为初始断点。

如果当前进程不再被调试，那么进程的初始化函数不会调用 `DbgBreakPoint`，可以正常运行。

10.6.2 编程时加入断点

Windows 的调试 API 中包含了一个用于产生断点异常的 API，名为 DebugBreak，它的原型非常简单，没有参数，也没有返回值：

```
void DebugBreak(void);
```

当编写程序时，如果希望在某种情况下中断到调试器中，可以加入如下代码：

```
if (IsDebuggerPresent() && <希望中断的附加条件>)
    DebugBreak();
```

这样，当程序执行到这里时，如果有调试器在，并且中断的附加条件成立，那么便会中断到调试器中，这对于调试某些复杂的多线程问题或随机发生的问题是很有用的，因为在应用程序中检测到希望中断到调试器的条件（包括条件断点难以实现的判断条件），然后中断到调试器中。

事实上，在 x86 平台上，DebugBreak API 等价于一条 INT 3 指令，所以直接使用如下嵌入式汇编也可以达到同样的效果：

```
_asm{int 3};
```

使用 API 具有更好的跨平台性，代码看起来也更优雅。

10.6.3 通过调试器设置断点

前面介绍的两种方法都是在被调试程序的代码中静态地埋入断点指令。通过调试器的断点功能可以向被调试进程动态地插入断点指令，当被调试进程遇到这些断点指令时，触发断点异常而中断到调试器中。

10.6.4 通过远程线程触发断点异常

前面的 3 种方法都是在程序的固定位置植入断点指令，只有当被调试进程执行到那里时，才会中断到调试器。如果希望被调试进程立刻中断到调试器中，比如按下一个热键就中断下来，那么前面的方法就不合适了。这种根据用户的即时需要而将被调试进程中断到调试器的功能通常被称为异步阻停（Asynchronous Stop）。

实现异步阻停的一种方法是利用 Windows 操作系统的 CreateRemoteThread API，在被调试进程中创建一个远程线程，让这个线程一运行便执行断点指令，把被调试进程中断到调试器。要做到这一点，我们需要被调试进程中有一个包含断点指令的函数，为了能让这个函数可以作为新线程的启动函数，它的函数原型应该符合 SDK 所定义的线程启动函数原型，即：

```
DWORD WINAPI ThreadProc( LPVOID lpParameter);
```

事实上，NTDLL.DLL 中已经设计好了这样一个函数，即 DbgUiRemoteBreakin，它内部会调用 DbgBreakPoint 执行断点指令，而且是在一个结构化异常保护块（SEH）

中做的调用，其伪代码如下：

```
DWORD WINAPI DbgUiRemoteBreakin( LPVOID lpParameter)
{
    __try
    {
        if(NtCurrentPeb()->BeingDebugged)
            DbgBreakPoint();
    }
    __except(EXCEPTION_EXECUTE_HANDLER)
    {
        return 1;
    }
    RtlExitUserThread(0); //never return
}
```

增加异常保护（`__except`）的目的是捕捉断点异常，万一调试器没有处理，那么这个异常处理器会处理它，以防止因无人处理而导致整个程序被终止。第 5 行的判断用来检测当前进程是否在被调试，如果不被调试，就不要触发断点异常，出于这个原因，向一个不在被调试的进程中创建远程线程并执行这个函数，并不会触发断点异常。倒数第 2 行用来强制退出当前线程。

介绍到这，我们明白了可以创建一个远程线程来执行 `DbgUiRemoteBreakin` 函数，以触发断点异常，为了进一步简化这项任务，Windows XP 引入了一个新的 API，叫 `DebugBreakProcess`，只要调用这个 API 就可以了。

```
BOOL DebugBreakProcess( HANDLE Process);
```

跟踪这个 API 的执行过程，可以发现它内部是调用 NTDLL 中的 `DbgUiIssueRemoteBreakin` 函数，后者使用 `DbgUiRemoteBreakin` 作为线程函数创建远程线程。

我们把以上介绍的用于触发断点异常的远程线程称为远程中断线程（**Remote Breakin Thread**），包括 WinDBG 在内的很多调试器所提供的 Break 功能都使用了远程中断线程。例如，在 WinDBG 中，选择 Debug 菜单中的 Break 项，或者按 Ctrl+Break 热键便会发出 Break 命令。WinDBG 接到此命令后会通过远程中断线程在被调试进程中产生一个断点异常，使其中断到调试器。明白这个原理后，我们就能理解为什么在被调试进程被中断后，WinDBG 总是显示类似如下的内容：

```
(1e74.1dc4): Break instruction exception - code 80000003 (first chance)
eax=7ffde000 ebx=00000001 ecx=00000002 edx=00000003 esi=00000004 edi=00000005
eip=7c901230 esp=00beffcc ebp=00befff4 iopl=0 nv up ei pl zr na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=0038 gs=0000
ntdll!DbgBreakPoint:
7c901230 cc          int     3
```

所有这些内容都是关于远程中断线程的。此时执行栈回溯命令，会看到这个线程的启动函数 `DbgUiRemoteBreakin` 调用 `DbgBreakPoint` 的过程：

```
0:001> k
ChildEBP RetAddr
00beffc8 7c9507a8 ntdll!DbgBreakPoint
00befff4 00000000 ntdll!DbgUiRemoteBreakin+0x2d
```

此时可以使用线程切换命令来切换到应用程序的线程，比如~0 s 切换到 0 号线程（初始线程）。远程中断线程会在被调试进程恢复运行后很快退出，因此它大多时候不会给调试工作带来副作用。

当调试器附加到一个已经运行的进程（第 10.4 节）时，通常也是通过远程中断线程来将被调试进程中断到调试器，这个断点被称为调试已运行进程的初始断点。

WinDBG 调试器附带了一个名为 Breakin.exe 的小工具，是以命令行方式运行的，其功能就是向指定进程（通过命令行参数）创建一个远程中断线程。针对一个不在被调试的进程执行这个动作，不会对其造成大的伤害，正如我们前面所说的，`DbgUiRemoteBreakin` 函数会先判断所处的进程是否在被调试。

10.6.5 在线程当前执行位置设置断点

利用远程线程实现异步阻停的一个明显不足，就是要在被调试进程中启动一个新的线程，这样做对被调试进程的执行环境（对象、句柄、栈和内存等）有较大的影响，而且可能会干扰被调试程序的自身逻辑。实现这一功能的另一种方式是在被调试进程的现有线程中触发断点。其主要步骤如下。

首先，将被调试进程中的所有线程挂起，这可以使用 `SuspendThread` API 来完成。

然后，取得每个线程的执行上下文（可以调用 `GetThreadContext` API），得到线程的程序指针寄存器（PC）的值，并在这个值对应的代码处设置一个断点，也就是把本来的一个字节保存起来，并写入断点指令（INT 3）的机器码（0xCC）。因为 PC 寄存器的值指向的是 CPU 下次执行这个线程时要执行的指令，因此在这个地方设置断点的目的就是当 CPU 下一次执行这个线程时就立刻触发断点异常而中断到调试器。对进程中的每个线程都重复此步骤。

最后，恢复所有线程（`ResumeThread` API），让它们继续执行，以触发断点。一旦有断点被触发，调试器会清除第二步设置的所有断点。

VC6 调试器和重构前的 WinDBG 调试器（第 29 章将详细介绍）是使用以上方法来实现异步阻停的。

因为当用户发出中断命令时，被调试进程通常是在执行某种等待函数（除非它在用户态有特别多的运行任务），而且大多数等待函数（`GetMessage`, `Sleep`, `SleepEx`）都是调用内核服务而进入内核态执行的，所以调试器取得的 PC 值通常指向的是系统服务返回后将执行的 `ret` 指令。这个指令地址在 Windows XP 中对应的调试符号是 `ntdll!KiFastSystemCallRet`。清单 10-10 显示了 VC6 调试器在被调试进程中设置的断点指令。

清单 10-10 VC6 调试器动态设置的断点指令

```
0:000> u 0x7c90eb94
ntdll!KiFastSystemCallRet:
7c90eb94 cc     int     3
```

在以上指令位置本来是 `ret` 指令，机器码是 `0xC3`。在断点命中后，VC6 会将其恢复成本来的指令。

值得注意的是，对于在内核态执行的线程，动态的断点指令是设置在内核服务返回位置的，这意味着只有在内核服务返回后，才能遇到断点指令。

10.6.6 动态调用远程函数

与刚才介绍的在程序指针的位置动态替换断点指令类似的一种方法是动态地调用一个函数，这个函数再执行断点指令。Windows 2000 的 NTDLL.DLL 和 Kernel32.DLL 中已经包含了这种方法的实现。简单来说，就是利用 NTDLL 中的 `RtlRemoteCall` 函数远程调用被调试进程内位于 KERNEL32.DLL 中的 `BaseAttachComplete` 函数。

具体来说，应该先将目标线程挂起，或使其锁定在一个稳定的内核状态，然后调用 `RtlRemoteCall` 函数，并将 KERNEL32.DLL 中的 `BaseAttachCompleteThunk` 的地址作为调用点（`CallSite`）参数传递给 `RtlRemoteCall`。`BaseAttachCompleteThunk` 是一小段汇编代码，它的作用就是调用 `BaseAttachComplete` 函数。

`RtlRemoteCall` 内部先通过 `NtGetContextThread` 内核服务取得目标线程的上下文，得到目标线程的栈地址，然后调整栈指针将取得的上下文结构（`CONTEXT`）和参数用 `NtWriteVirtualMemory` 写到目标线程的栈上，而后，将线程上下文结构中的程序指针字段（`EIP`）设置为参数指定的调用点地址（`CallSite`）。接下来通过 `NtSetContextThread` 将修改后的 `CONTEXT` 结构设置回目标线程。这些准备工作做好后，便可以调用 `NtResumeThread` 了，即恢复目标线程运行，而且它一运行便应该执行调用点所指向的代码。

`BaseAttachComplete` 函数内部查询当前进程是否在被调试，如果是，便执行 `DbgBreakPoint` 触发断点。清单 10-11 显示了被调试的记事本程序中断到调试器（WinDBG）后 `kn` 命令所显示的栈回溯。

清单 10-11 因为动态调用而中断到调试器的栈回溯（Windows 2000）

```
0:000> kn
# ChildEBP RetAddr
00 0006fbe4 77e8be83 ntdll!DbgBreakPoint
01 0006fbf4 77e88929 KERNEL32!DebugActiveProcess+0x1e0
02 0006fee4 01002a01 KERNEL32!BaseAttachCompleteThunk+0x13
03 0006ff24 01006576 notepad!WinMain+0x63
04 0006ffc0 77e67903 notepad!WinMainCRTStartup+0x156
05 0006fff0 00000000 KERNEL32!SetUnhandledExceptionFilter+0x5c
```

首先，栈帧#01 所对应的函数应该是 `BaseAttachComplete`，因为缺少它的符号，

所以调试器就以 DebugActiveProcess 函数为参照物。

观察上面的栈回溯，就好像是记事本程序（notepad）的 WinMain 函数调用 BaseAttachCompleteThunk，事实根本不是这样的，WinMain 实际调用的是 GetMessageW API，这个 API 进而调用内核态中的子系统服务。但在内核态执行（等待）时，RtlRemoteCall 函数执行了前面描述的动作，使这个线程“飞”到 BaseAttachCompleteThunk 处。

当 DbgBreakPoint 返回后，BaseAttachComplete 会调用 NtContinue，并将其保存在栈上的 CONTEXT 结构作为参数，这样，这个线程便又被恢复成原来的样子了，前面执行的动作就好像梦游一样被遗忘了。

10.6.7 挂起中断

以上 3 种异步阻停的方法都是希望被调试程序继续执行，然后遇到断点指令就中断到调试器，也就是假定被调试进程依然可以继续执行用户态代码的。但是，如果被调试进程因为某种原因不能继续执行用户态代码，那么这 3 种方法就都无法工作了。举例来说，如果被调试进程因为某个同步对象被死锁无法创建或启动新的线程，那么远程中断线程方法就不能工作了。对于前面介绍的在内核服务返回处设置的动态断点，如果线程在内核态无限期等待，即所谓的挂在内核态（Hang in Kernel），那么这样的断点也不再会命中了。

针对以上问题，一种替代性的方法是强行将被调试进程的所有线程挂起，然后进入一种准调试状态。我们称这种方法为挂起中断（Breakin by Suspend）。之所以叫准调试状态，是因为通过这种方式中断到调试器后，不可以执行单步执行等跟踪命令。

WinDBG 调试器在使用远程线程中断功能超时后会使用挂起中断方式。具体来说，WinDBG 在创建远程中断线程后，会等待断点事件发生，等待数秒钟后会显示清单 10-12 所示的前两行信息。再等待 30 秒后，WinDBG 就会使用挂起中断方式，并提示第 3 行和第 4 行所示的信息。在将所有线程都挂起后，调试引擎的底层函数会模拟一个唤醒调试器（Wake Debugger）的异常事件，调试器的事件处理函数收到这个事件后便会中断给用户，提示第 5~10 行所示的信息。

清单 10-12 WinDBG 使用挂起中断方式时输出的提示信息

```

Break-in pending
Break-in sent, waiting 30 seconds...
WARNING: Break-in timed out, suspending.
        This is usually caused by another thread holding the loader lock
(abc.1530): Wake debugger - code 80000007 (first chance)
eax=00000000 ebx=00000000 ecx=00000000 edx=00000000 esi=ffffffff edi=00000001
eip=7c90eb94 esp=0013dbc8 ebp=0013e618 iopl=0          nv up ei ng nz na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000
ntdll!KiFastSystemCallRet:
7c90eb94 c3          ret

```

因为线程通常是挂在内核态，所以用户态看到的程序指针通常是指向 KiFastSystemCallRet 的。这一点与前一种方法产生的中断类似。使用挂起方式中断后，在调试器中可以执行各种观察和编辑类命令来分析和操作被调试程序的栈回溯、内存和栈等信息。但是不可以执行跟踪类命令，例如，以下是执行 p 命令时，WinDBG 给出的错误提示：

```
0:000> p
Due to the break-in timeout the debugger cannot step or trace
^ Operation not supported in current debug session 'p'
```

10.6.8 调试热键（F12）

除了在调试器中发出 Break 命令和执行 Breakin 这样的小工具，还有一种方式可以“激发”被调试进程中断到调试器，那就是向被调试进程输入调试热键（默认为 F12）。举例来说，当我们使用 WinDBG 调试计算器程序时，除了可以通过在 WinDBG 中按 Ctrl+Break 将计算器中断到调试器外，还可以向计算器程序按 F12（也就是当计算器程序在前台时按 F12）。

在内部，该功能是因为 Windows 子系统的内核部分接收到此热键，然后通过 LPC 请求 CSRSS 中的 SrvActivateDebugger 服务。

SrvActivateDebugger 首先检查要调试的进程是不是自己，如果是，而且自己处于被调试状态，便调用 DbgBreakPoint 中断到调试器，如果不是，便试图通过远程方式触发断点事件。触发的方式因为 Windows 版本的不同而略有不同。

在 Windows XP 之前，SrvActivateDebugger 使用的是前面介绍动态调用（RtlRemoteCall）远程函数的方法。从 Windows XP 开始，SrvActivateDebugger 是使用前面介绍的向要调试进程创建新的远程中断线程方法。从调试器的角度来看，前一种方法的断点异常发生在被调试进程的 UI 线程中，即现有线程中。后一种方法激发的断点异常发生在新创建的远程中断线程中。

对于 Windows XP 所使用的方法，因为也是依靠远程中断线程机制工作的，所以在使用这种方法中断后，调试器的当前线程是远程中断线程，这与调试器前面介绍的调试器自己创建远程中断线程的情况是一样的。事实上这两种方法只是远程线程的创建者有所不同。

可以通过如下注册表键下的 UserDebuggerHotKey 选项，来指定其他按键作为调试热键：

```
HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\AeDebug
```

10.6.9 窗口更新

在调试进程被中断到调试器后，被调试进程是处于“停顿”状态的，直到调试器

恢复其运行。在这一阶段，如果被调试进程是窗口程序，那么它的窗口是僵死的，不可移动，不可改变大小，会遮挡住它所对应的桌面区域，对“显示桌面命令”也不响应。另外，被中断到调试器的进程自己也不能刷新窗口内容，所以它的窗口区域处于“无人更新”状态。为了行文方便，我们把中断到调试器中的被调试进程的窗口简称被中断窗口（Broke Window）。

因为不同版本的 Windows 系统，使用的窗口管理策略有所不同，所以被中断窗口表现出来的“症状”也有所不同。

在 Windows 2000 中，系统会更新被中断窗口的非用户区（Non-Client），包括标题栏和边框等。对于用户区，如果有其他窗口移过这个窗口，那么被中断窗口的对应区域会被擦成背景色（默认白色）。如图 10-8 所示。

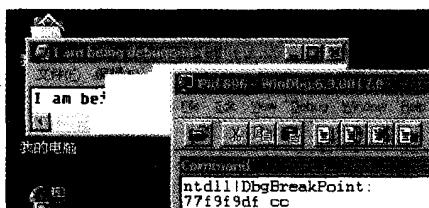


图 10-8 Windows 2000 中的被中断窗口

在图 10-8 中，记事本程序在被 WinDBG 程序调试，并且被中断到调试器中。将 WinDBG 的窗口（或者其他窗口）在记事本窗口上方移动，记事本的用户区域会被涂抹掉，但是标题栏和边框不会，因为系统会更新非用户区。

在 Windows XP 中的情况，如图 10-9 所示，整个被中断窗口都处于被任意涂抹状态，而且会留下移动窗口的痕迹。

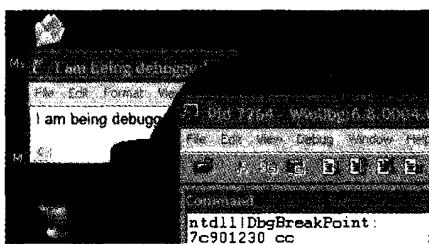


图 10-9 Windows XP 中的被中断窗口

有必要说明的是，在 Windows XP 中，对于未被调试的进程，如果它的主窗口在几秒钟内没有响应，那么系统会将其视为无响应窗口（not responding window），并为其创建一个所谓的精灵窗口（Ghost Window）。精灵窗口与原来窗口具有相同的 Z 顺序（Z-Order）、位置、大小，但其窗口标题会包含无响应字样（Not Responding）。用户可以移动精灵窗口、调整其大小，也可以通过窗口标题条的关闭按钮（默认菜单，或 Alt+F4 热键）触发系统终止这个应用程序。

为了更好地理解和感受精灵窗口的特性，我们特意编制了一个小程序 HungWin（code\chap10\HungWin），直接（非调试）启动它，点击 File 菜单的 Hang，那么它便会调用 SuspendThread API 将所在线程（该程序的唯一线程）挂起。

```
case IDM_HANG:  
    SuspendThread(GetCurrentThread());
```

在几秒钟后，再点击 HungWin 程序的窗口，便会发现其菜单不见了，而且标题栏出现了 Not Responding 字样，这便是所谓的精灵窗口（Ghost Window）。接下来可以感受一下它的特性，比如移动和改变大小等。尝试将精灵窗口移动至中断到调试器的计算器窗口区域（见图 10-10），然后拖动第 3 个窗口（CMD.EXE）在其上面晃动，我们会发现计算器窗口区域留下了很多痕迹，而精灵窗口始终保持着清洁状态。

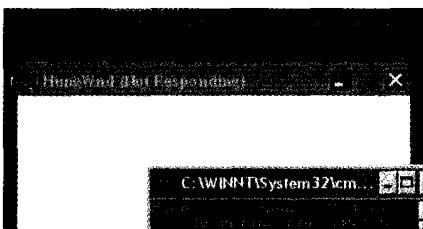


图 10-10 体验精灵窗口（Ghost Window）

如果在窗口僵死前就将调试器附加到这个程序上，那么 Windows XP 就不会对其应用精灵窗口策略，这对于分析问题和软件调试是有利的。

对于 Windows Vista，如果使用新引入的 Aero 风格来显示外观（Appearance），那么被中断窗口会始终保持中断前的显示。如果使用以前的显示外观，那么被中断窗口的刷新策略与 Windows 2000 和 Windows XP 中的情况类似，在此不再赘述。

10.7 输出调试字符串

输出调试字符串（Debug Output）是一种常用的辅助调试手段。在 DOS 程序或 Windows 的控制台程序中，我们可以使用 print（或 printf）函数来打印调试字符串。对于 Windows 下的非控制台程序，print 函数不再适用。事实上，不论是控制台程序还是窗口程序，Windows 系统都可以使用 OutputDebugString() API 来输出调试字符串。

根据 Windows SDK 的说明，OutputDebugString 可以把参数指定的字符串发送给调试器，如果程序不在被调试，那么系统调试器（内核调试器）会显示该字符串，如果系统调试器也没有被激活，那么 OutputDebugString 什么也不做。

10.7.1 发送调试信息

下面我们先来看一看 OutputDebugString 是如何把参数指定的字符串发送给调

试器的，这并不是一个不值一提的问题，因为发送字符串的应用程序和接收字符串的调试器分别属于不同的进程，而且还要保证不论在有无调试器的情况下都能按规定工作，所以就有不少值得探索的细节。

简单来说，OutputDebugString 是利用 RaiseException() API 产生一个特殊的异常，该异常的代码固定为 DBG_PRINTEXCEPTION_C，ntstatus.h 中有其定义：

```
#define DBG_PRINTEXCEPTION_C ((NTSTATUS)0x40010006L)
```

我们把这个异常称为调试打印（Debug Print）异常。清单 10-13 给出了 OutputDebugString 函数调用 RaiseException API 发起调试打印异常的伪代码。

清单 10-13 OutputDebugString API 的伪代码（部分）

```
_try
{
    ExceptionArguments[0] = strlen (lpOutputString) + 1;
    ExceptionArguments[1] = (ULONG_PTR)lpOutputString;
    RaiseException (DBG_PRINTEXCEPTION_C, 0, 2, ExceptionArguments);
}
__except (EXCEPTION_EXECUTE_HANDLER)
{
    // 异常处理代码
}
```

从上面的代码可以看到，调试信息（lpOutputString）的长度和地址是作为异常的参数传递给 RaiseException API 的。以上代码有两点值得说明，第一，OutputDebugString 是在 Kernel32.DLL 中实现的，分为 UNICODE 版本（函数名为 OutputDebugStringW）和 ANSI 版本（函数名为 OutputDebugStringA），与其他 API 将 ANSI 版本的参数转为 UNICODE 然后调用 UNICODE 版本的函数不同，OutputDebugString API 是 UNICODE 版本的函数把 UNICODE 类型的字符串转换为 ANSI，然后调用 ANSI 版本的函数。因此，我们看到上面代码中使用的是适用于 ANSI 字符串的 strlen 函数。第二，注意第 4 行，这里只是把字符串参数（lpOutputString）的地址存入 ExceptionArguments[1]，传递给 RaiseException API，这意味着，异常信息中将只包含字符串的地址，而不是其实际内容。

RaiseException API 被调用后，会产生一个标准的异常结构 EXCEPTION_RECORD，然后调用内核服务将这个模拟的异常发送到内核中进行分发。

10.7.2 使用调试器接收调试信息

内核中的异常分发函数 KiDispatchException 会按照统一的流程来分发调试打印异常。如我们在上一章所介绍的，KiDispatchException 会调用支持用户态调试的内核例程 DbgkForwardException 向调试子系统通报异常。DbgkForwardException 如果检查到当前进程在被调试，它会将这个异常通过调试子系统服务器发给调试器。

我们知道，调试器的工作线程是通过 WaitForDebugEvent API 调用 NTDLL 中的

DbgUiWaitStateChange 函数，来等待调试事件的。在接收到异常事件后，Windows XP 开始的 Windows 是使用 DbgUiConvertStateChangeStructure 函数将 DBGUI_WAIT_STATE_CHANGE 结构组装成 DEBUG_EVENT 结构，在 Windows XP 之前，是 WaitForDebugEvent API 自己组装的。但不论哪个函数来做结构组装工作，对于异常事件，如果异常代码等于 DBG_PRINTEXCEPTION_C，那么它们都会将事件代码字段（dwDebugEventCode）设置为 OUTPUT_DEBUG_STRING_EVENT，而不是 EXCEPTION_DEBUG_EVENT，并将异常参数中的调试信息填写到 DEBUG_EVENT 结构的 DebugString 子结构中，DebugString 是一个 OUTPUT_DEBUG_STRING_INFO 结构，其定义如下：

```
typedef struct _OUTPUT_DEBUG_STRING_INFO
{
    LPSTR lpDebugStringData; // 调试信息字符串的地址
    WORD fUnicode; // 是否为 UNICODE，参见正文
    WORD nDebugStringLength; // 调试信息字符串的长度
} OUTPUT_DEBUG_STRING_INFO, *LPOUTPUT_DEBUG_STRING_INFO;
```

如前面所介绍的，对于 UNICODE 程序，OutputDebugString API 的 UNICODE 版本（OutputDebugStringW）会将 UNICODE 格式的字符串先转换为 ANSI 格式。因此，目前版本的 Windows 系统中，调试信息都是使用非 Unicode 方式传递的，所以上面的 fUnicode 字段会被固定设为 FALSE。

概括一下，OUTPUT_DEBUG_STRING_EVENT 事件是异常事件的一个特例，它是以异常事件的形式而产生和分发的，只有在发送给调试器之前，系统（WaitForDebugEvent API 或 DbgUiConvertStateChangeStructure）才将其翻译为 OUTPUT_DEBUG_STRING_EVENT 事件。

调试器收到 OUTPUT_DEBUG_STRING_EVENT 事件后，会从其参数中得到调试信息字符串的地址和长度，然后使用内存访问函数从被调试进程的空间中读取调试信息字符串，然后显示出来。显示后，调试器默认会立即调用 ContinueDebugEvent 回复此事件，表明自己处理了该异常，于是 KiDispatchException 函数结束异常分发，被调试程序便继续正常运行了。

10.7.3 使用工具接收调试信息

接下来看一下应用程序没有被调试的情况。首先我们注意在上面的伪代码中，产生异常的代码（调用 RaiseException API）是被放在一个异常保护块中的（__try 和 __except，将在第 11 章详细讨论）。

在应用程序没有被调试的情况下，异常不会发给用户态调试器，因此 KiDispatchException 函数会继续分发这个异常，也就是寻找异常处理器来处理这个异常。其细节我们将在第 11 章介绍，现在我们只要知道，系统会找到并执行异常处理块中的代码，即上面伪代码中 __except 块所对应的内容。

__except 块中的异常处理代码的主要逻辑是试图将字符串发给 DBWIN 工具。

DBWIN 是旧的 Windows SDK 中包含的一个 16 位的小工具（由 DBWin.DLL 和 DBWin.EXE 组成），用以捕捉应用程序中使用 OutputDebugString API 所输出的信息。目前的 SDK 不再包含 DBWIN，但其中包含的 DBMon（Debug Monitor）（控制台程序）可以完成类似的任务。可以完成类似任务的工具还有很多，比如 Debug View 等。为了描述简单，我们仍然使用 DBWIN 来泛指这类工具。

那么，异常处理块是如何将字符串发给 DBWIN 程序的呢？简单来说，是通过几个内核对象使用进程间的通信机制来通信的，其主要步骤如下：

首先，OutputDebugString 会检查静态变量判断是否创建过名为“DBWinMutex”的互斥（Mutex）对象，如果没有则调用 CreateDBWinMutex 函数来创建，并将其句柄保存在另一个静态变量中。DBWinMutex 用于同步 OutputDebugString 所在进程与 DBWIN 进程间的通信，保证同一时间只能有一个线程与 DBWIN 通信（传递数据），以避免当有多个线程都包含对 OutputDebugString 的调用时导致数据丢失或其他错误。因为创建 DBWinMutex 时指定了对象名称，因此对象管理器会保证一旦系统中已经存在具有该名称的对象，那么后来的调用只是打开已经存在的对象。

创建或打开 DBWinMutex 对象后，OutputDebugString 会使用 WaitForSingleObject 函数无限期地等待 DBWinMutex 对象的使用权。

等待成功后，OutputDebugString 会通过 OpenFileMapping API 试图打开名为“DBWIN_BUFFER”的内存映射文件（file mapping）对象（section 对象）。DBWIN_BUFFER 对象是由 DBWIN 程序通过调用 CreateFileMapping API 创建的，OutputDebugString 只是调用 OpenFileMapping API 试图打开该对象，如果打开失败，则说明系统内没有 DBWin 程序在运行。这时 OutputDebugString 会调用 DbgPrint 函数，试图将字符串打印到内核调试器。

如果打开 DBWIN_BUFFER 成功，那么，OutputDebugString 会调用 MapViewOfFile API 将映射文件映射到本进程空间中。

接下来，OutputDebugString 会调用 OpenEvent API 打开两个事件（Event）对象，分别名为“DBWIN_BUFFER_READY”和“DBWIN_DATA_READY”。DBWIN_BUFFER_READY 对象用来供 DBWIN 程序指示缓冲区是否准备好，OutputDebugString 通过等待该事件对象判断是否可以向缓冲区（DBWIN_BUFFER 映射文件）写数据。DBWIN_DATA_READY 供 OutputDebugString 在将数据写到内存映射文件后，通知 DBWIN 程序来读取数据。当数据量较大时，OutputDebugString 会分多次发送，每次将数据写到内存映射文件后，便设置 DBWIN_DATA_READY 通知 DBWIN 读取数据，然后开始等待 DBWIN_BUFFER_READY 事件。DBWIN 得到通知后便读取数据，读取完毕后便设置 DBWIN_BUFFER_READY 事件，OutputDebugString 得到 DBWIN_BUFFER_READY 事件后，便继续写剩下的数据，如此反复，直到发完所有数据。

当数据传递结束后，OutputDebugString 会释放 DBWinMutex 互斥对象，以便让

其他线程或程序可以与 DBWIN 通信。

DBWIN_BUFFER_READY 事件的另一个作用，是供各种 DBWIN 程序通过检查它的存在来保证系统中只有一个 DBWIN 程序在运行。DBWIN 程序当初始化期间创建 DBWIN_BUFFER_READY 事件对象时（使用 CreateEvent API），即使成功，也会通过检查 GetLastError() 的返回值是否为 ERROR_ALREADY_EXISTS，来判断系统中是否已经存在同名的对象。如果有，则停止继续创建其他对象。例如，当运行 DBMon 的第二个实例时，它会显示“already running（已经运行）”然后退出。

以上对象除了 DBWinMutex，其他都是由 DBWIN 程序创建的，使用 WinDBG 附加到 DBWIN 程序（dbmon）上，然后使用 !handle 0 5 命令便可以看到这些对象。清单 10-14 显示了在 DBMon 进程中用于与 OutputDebugString 通信的各个对象。

清单 10-14 DBMon 进程中用于与 OutputDebugString 通信的各个对象

```
0:000> !handle 0 5 //参数 0 代表列出所有句柄，参数 5 代表显示对象名称和类型
...
Handle 30 Type Event
  Name      \BaseNamedObjects\DBWIN_BUFFER_READY
Handle 34 Type Event
  Name      \BaseNamedObjects\DBWIN_DATA_READY
Handle 38 Type Section
  Name      \BaseNamedObjects\DBWIN_BUFFER
```

因为当 OutputDebugString API 与 DBWIN 程序通信时才需要 DBWinMutex 对象，所以，没有调用过 OutputDebugString API 或调用了 OutputDebugString API 但处于调试状态的进程中不会有 DBWinMutex 对象。为了证明这一点，我们特意编写了一个名为 DbgString 的小程序，清单 10-15 列出了它的源代码。

清单 10-15 用于观察 DBWinMutex 内核对象的 DbgString 小程序

```
1 #include "stdafx.h"
2 #include <windows.h>
3
4 int main(int argc, char* argv[])
5 {
6     printf("Program to test existence of DBWinMutex object.\n");
7     BOOL bDebuggerPresent=IsDebuggerPresent();
8
9     if(!bDebuggerPresent)
10    {
11        printf("Inspect this process now to verify no DBWinMutex.\n");
12        getchar();
13        OutputDebugString(szMsg);
14        printf("Inspect this process now to verify DBWinMutex exists.\n");
15        getchar();
16    }
17    else
18    {
19        OutputDebugString(szMsg);
20        printf("Inspect this process now to verify no DBWinMutex.\n");
21        getchar();
22    }
23    return 0;
24 }
```

如果在控制台下直接运行该程序（调试版本或非调试版本均可），那么当运行到第 12 行等待用户按键时，使用 WinDBG 附加到该进程，然后使用 !handle 0 命令列出所有句柄，我们会发现进程中还不存在 DBWinMutex 对象。输入 g 命令让 DbgString 程序运行，并按任意键让 getchar 函数返回，而后 DbgString 继续运行调用 OutputDebugString 后停在第 15 行等待输入，这时按 Ctrl+Break 将 DbgString 中断到调试器。再次执行 !handle 命令，我们会发现进程中还是不存在 DBWinMutex 对象。这是因为有调试器存在时，调试子系统会通过 OUTPUT_DEBUG_STRING_EVENT 事件将字符串发给调试器，根本不会执行到异常保护块中与 DBWIN 程序通信的代码。由此我们也很容易理解，为什么当一个程序运行在调试器中时，DBWIN 程序就收不到通过 OutputDebugString 输出的信息了。

关闭 WinDBG 和 DbgString，然后再次运行 DbgString，等到 DbgString 第二次等待输入（第 15 行）时将 WinDBG 附加到调试器，执行 !handle 命令，我们会看到进程中已经有 DBWinMutex 对象了：

```
0:000> !handle 0 6 Mutant           // 只显示互斥类型的对象
Handle 10                           // 句柄号
Attributes      0
GrantedAccess   0x120001:
    ReadControl,Synch
    QueryState
HandleCount     33                  // 打开这个互斥对象的句柄总数
PointerCount    35                  // 引用这个对象的指针数量
Name            \BaseNamedObjects\DBWinMutex // 对象名称
```

从句柄计数等于 0x33 判断，系统中有很多其他进程打开过这个对象。

图 10-11 画出了 OutputDebugString() API（右侧）与 DBWIN 程序（左侧）之间通信的基本流程。

这里有几点值得说明。首先，系统中可能有多个程序都调用 OutputDebugString（图中只画出了一个），但活动的 DBWIN 程序应该只有一个。当多个进程中的 OutputDebugString 都要与 DBWIN 程序进行通信时，它们会通过等待 DBWinMutex 互斥对象进行排队，谁获得 DBWinMutex 对象，谁便与 DBWIN 程序通信。

另外，图中列出的函数调用只是用来示意关键参数，其他参数可能没有列出。想自己编程实现 DBWIN 程序的读者可以参考如下两个例子：(1) MSDN 中的 DBMON 源代码，即接收 OutputDebugString 的小控制台程序的完整代码。(2) Andrew Tucker、Daniel Christian 和 Dwayne Towell 编写的 DBWIN32 程序，DBWIN32 是 32 位的 Windows GUI 程序，它在 DBMON 代码的基础上增加了图形化界面，并继承了 16 位 DBWIN 的部分功能，比如通过一个驱动程序输出信息到另一个单色监视器（Monochromic Monitor）。

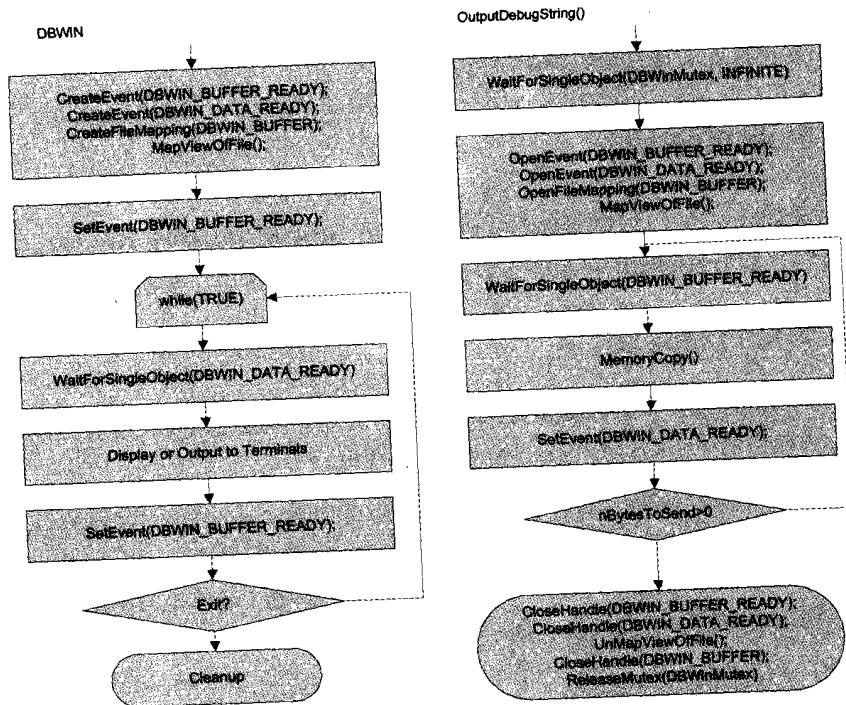


图 10-11 OutputDebugString() API (右) 和 DBWIN 程序 (左) 的基本流程

最后想说明的一点是，从性能角度来看，OutputDebugString 是一种执行效率较低的方法。对于效率要求较高的程序，过多使用 OutputDebugString 会影响程序的运行速度。导致 OutputDebugString 效率较低的原因主要有以下几点。

首先，OutputDebugString 是通过 RaiseException 抛出异常来触发与调试器或 DBWIN 程序的通信的，RaiseException 会导致当前线程从用户态切换到内核态，然后经由内核态的异常处理函数进行分发，最后才将信息送给用户态的调试器或 DBWIN 程序。

其次，当非调试状态执行时，如果系统中有 DBWIN 程序运行，那么与 DBWIN 程序通信需要一定的开销（打开、等待多个内核对象）。如果系统中有大量对 OutputDebugString 的调用，那么 DBWIN 程序可能很忙碌，调用 OutputDebugString 的线程需要排队等待。

使用 OutputDebugString 输出调试信息的另一个缺点是安全性和可控性差。也就是说，通过 OutputDebugString 输出的信息，可以很容易被各种 DNWIN 程序接收。因此不该使用 OutputDebugString 输出可能泄露知识产权的信息。可控性差是指 OutputDebugString 函数本身没有提供动态开启或关闭信息输出的机制，如果希望不重新编译就开启或关闭使用 OutputDebugString 输出的信息，那么需要自己编写代码对其进行封装，在封装函数中包含动态控制机制。

尽管 `OutputDebugString` 有以上不足，但其也有很多优点，比如使用方便，可靠性高，在调试和非调试状态下都可以工作等。

10.8 终止调试会话

本节我们介绍终止调试会话的几种典型情况，探索每种情况的内部过程，并比较它们的异同。除非特别指出，本节主要讨论 Windows XP 之后（包括 XP）的情况。

10.8.1 被调试进程退出

不同类型的程序有不同的退出方式、菜单和热键等，但无论使用哪种方式，通常都会执行内核中的 `PspExitThread` 函数来退出线程。

`PspExitThread` 函数内部会通过 `EPROCESS` 结构的 `DebugPort` 字段检查当前进程是否在被调试，如果是，它会根据当前是否是最后一个线程而调用 `DbgkExitProcess` 或 `DbgkExitThread` 来通知调试子系统。清单 10-16 显示了在使用 Alt+F4 热键，关闭被调试的记事本程序时的执行过程。

清单 10-16 进程退出时的典型过程

```

kd> k
ChildEBP RetAddr
f4a56c60 805bbbc4 nt!DbgkExitProcess           // 通知调试子系统
f4a56d08 8058ac25 nt!PspExitThread+0x2a7        // 进程管理器的线程退出函数
f4a56d28 80591d13 nt!PspTerminateThreadByPointer+0x50 // 终止线程
f4a56d54 804da140 nt!NtTerminateProcess+0x116     // 终止进程的内核服务
f4a56d54 7ffe0304 nt!KiSystemService+0xc4       // 内核服务分发函数
0006fdf4 77f7664a SharedUserData!SystemCallStub+0x4 // 调用内核服务
0006fdf8 77e798ec ntdll!NtTerminateProcess+0xc    // 内核服务的残根函数
0006fef0 77e7990f kernel32!_ExitProcess+0x57      // Kernel32 的进程退出函数
0006ff04 77c379c8 kernel32!TerminateProcess       // 终止进程 API
0006ff0c 77c37ad9 msvcrt!__crtExitProcess+0x2f     // C 运行库的进程退出函数
0006ff18 77c37aea msvcrt!_cinit+0xe4   // 此处的 cinit 只是作为参照点，实际应为 doexit
0006ff28 01006c65 msvcrt!exit+0xe             // C 运行库的退出函数
0006ffc0 77e814c7 notepad!WinMainCRTStartup+0x185 // 编译器插入的启动函数
0006fff0 00000000 kernel32!BaseProcessStart+0x23  // 进程启动函数

```

如果程序在被调试，那么 `DbgkExitProcess` 会通过调试子系统向调试器发送进程退出事件。调试器收到此事件后便知道被调试进程正在退出。接下来的处理与调试器的实现和调试器中对进程退出事件的设置有关。对于 MSDEV 等调试器（VC6 的集成调试器），收到进程退出事件后，它们便清理内部状态，结束本次调试了。在 WinDBG 中，默认情况下，它会向用户报告此事件，显示清单 10-17 所示的内容，并中断下来。

清单 10-17 WinDBG 收到进程退出事件后输出的内容

```

eax=00000000 ebx=00000000 ecx=7c800000 edx=7c97c080 esi=7c90e88e edi=00000000
eip=7c90eb94 esp=0007fde8 ebp=0007fee4 iopl=0          nv up ei pl zr na pe nc

```

```
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000          efl=00000246
ntdll!KiFastSystemCallRet:
7c90eb94 c3          ret
```

值得说明的是，上面显示的 `KiFastSystemCallRet` 是用来调用内核服务的 `KiFastSystemCall` 函数（参见 8.3.3 节）的一部分（返回部分），它是目前程序指针寄存器的值（`7c90eb94`）所对应的符号。程序指针总是指向将要执行的那条指令，所以，现在还没有执行到这条 `ret` 指令。事实上，被调试进程目前是在内核中执行，因为执行线程退出函数而被调试子系统挂起，目前处于睡眠状态。因为 `PspExitThread` 是在释放线程的用户态栈之前通知调试子系统的，所以当收到进程退出事件时，调试器仍然可以观察被调试进程中的信息，包括观察栈、内存和 PEB、TEB 等结构，这对于调试应用程序退出的原因是非常有用的。但是，此时不能再执行让被调试进程运行的 `p`（单步执行）和 `g` 等命令，如果执行，WinDBG 会提示没有可以运行的被调试程序：

```
0:000> p
^ No runnable debuggees error in 'p'
```

此时，进程的句柄和 `EPROCESS` 结构仍在，所以在任务管理器中，仍然可以看到这个进程。要想让被调进程继续完成其退出工作，需要触发 WinDBG 继续调试事件（`ContinueDebugEvent`），恢复被调试进程运行。这可以通过 WinDBG 的停止调试功能（`Debug > Stop Debugging`）或分离调试对话功能（`Debug > Detach Debuggee`）来完成。`PspExitThread` 函数恢复运行后，会继续完成线程的清理工作，释放线程所使用的资源。

进程的最后清理和删除工作是由进程管理器的工作线程执行 `PspProcessDelete` 函数来完成的。`PspProcessDelete` 函数内部会检查进程 `EPROCESS` 结构的 `DebugPort`，如果不为空，会调用 `ObDereferenceObject` 函数取消引用。而后 `PspProcessDelete` 调用内存管理器的 `MmDeleteProcessAddressSpace` 删除进程的地址空间，该进程彻底在系统中消失。

10.8.2 调试器进程退出

退出调试器是结束调试会话的另一种简单方式。调试器进程退出的过程与其他 Windows 进程相比，大多数步骤都是一样的。值得我们注意的便是有关调试对象的操作。

当调试器的工作线程退出时，`PspExitThread` 函数会检查 `TEB` 结构的 `DbgSsReserved[1]` 字段，如果该字段不为空，会调用 `ObCloseHandle` 关闭调试对象的句柄。清单 10-18 显示了调试记事本程序的 WinDBG 进程退出时（执行 `PspExitThread`）通过内核调试器观察到的调试对象句柄的信息。

清单 10-18 观察调试器退出时的调试对象

```
kd> !handle 0748
processor number 0, process 81caf958
PROCESS 81caf958 SessionId: 0 Cid: 0268 Peb: 7ffdf000 ParentCid: 065c
        DirBase: 0c884000 ObjectTable: e18f80e8 HandleCount: 139.
        Image: windbg.exe
```

```

Handle table at e1288000 with 139 Entries in use
0748: Object: 81f6e8e8 GrantedAccess: 001f000f Entry: e1288e90
Object: 81f6e8e8 Type: (81fc9778) DebugObject
    ObjectHeader: 81f6e8d0 (old version)
    HandleCount: 1 PointerCount: 2

```

空行上方的信息是关于所在进程的，映像文件名为 `windbg.exe`，进程中共有 0x139 个句柄。空行下方的信息是关于参数（0x748）所指定句柄的，对象的类型为 `DebugObject`，即调试对象，句柄计数目前为 1，引用这个对象的指针计数为 2。

需要指出的是，因为 WinDBG 在调用 `DbgUiConnectToDbg` 后会将 `DbgSsReserved[1]` 字段设置为 0（参见 10.1 节）。所以在执行 `PspExitThread` 函数时，检测到 `DbgSsReserved[1]` 字段为 0，就不会调用 `ObCloseHandle` 来关闭调试对象句柄。但这并不要紧，因为后面清理句柄表（Handle Table）时，会递减句柄计数，相当于执行 `ObCloseHandle`。

清单 10-19 显示了 `PspExitThread` 函数调用 `ObKillProcess` 函数清理句柄表的执行过程。

清单 10-19 调试器进程退出的典型过程

```

kd> kn
# ChildEBP RetAddr
00 f4fdb620 805812b4 nt!DbgkpCloseObject           // 调用对象的关闭函数
01 f4fdb650 80581108 nt!ObpDecrementHandleCount+0x119 // 递减关闭句柄的引用
02 f4fdb678 80591fb8 nt!ObpCloseHandleTableEntry+0x14b // 关闭句柄表中的一项
03 f4fdb694 80592041 nt!ObpCloseHandleProcedure+0x1d // 关闭表中所有句柄
04 f4fdb6c4 80591f6e nt!ExSweepHandleTable+0x4d     // 清理句柄表
05 f4fdb6f0 80591e01 nt!ObKillProcess+0x5a           // 对象管理器的针对进程退出所用的函数
06 f4fdb798 8058ac25 nt!PspExitThread+0x5cb        // 线程退出
07 f4fdb7b8 80591d13 nt!PspTerminateThreadByPointer+0x50 // 终止线程
08 f4fdb7e4 804da140 nt!NtTerminateProcess+0x116      // 终止进程内核服务
09 f4fdb7e4 7ffe0304 nt!KiSystemService+0xc4         // 系统服务分发函数
0a 0006c310 77f7664a SharedUserData!SystemCallStub+0x4 // 调用系统服务
0b 0006c314 77e798ec ntdll!NtTerminateProcess+0xc      // 系统服务的残根
0c 0006c40c 77e7990f kernel32!_ExitProcess+0x57       // Kernel32 内的进程退出函数
0d 0006c420 0103b76a kernel32!TerminateProcess        // 终止进程 API
0e 0006c430 0103aef7 windbg!TerminateApplication+0xda // 终止应用程序
0f 0006cfcc 77d43a68 windbg!FrameWndProc+0x18df      // 窗过过程
...                                         // 消息分发过程和其他函数省略

```

栈帧 1 是递减被关闭句柄的引用。这个函数会调用句柄所对应的对象类型注册的关闭函数，并将对象的句柄计数作为参数传递给关闭函数。调试对象（`DebugObject`）的关闭函数是 `DbgkpCloseObject`。`DbgkpCloseObject` 函数共有 5 个参数，分别是指向 `EPROCESS` 结构的指针、指向调试对象的指针、赋给句柄的权限、进程的句柄计数、调试对象的句柄计数。

`DbgkpCloseObject` 函数内部会先检测参数中指定的调试对象句柄计数是否大于 1，如果是，便返回。对于我们讨论的情况，从清单 10-18 可以看出，引用计数等于 1，

因为尽管 ObpDecrementHandleCount 会在调用对象关闭函数前递减句柄计数，但是它在递减前会把本来的计数保存起来，并作为参数传递给关闭函数。也就是说，在调用 DbgkpCloseObject 函数时，使用 !object 命令观察到的句柄计数已经是 0 了：

```
kd> !object 81f6e8e8
Object: 81f6e8e8 Type: (81fc9778) DebugObject
    ObjectHeader: 81f6e8d0 (old version)
    HandleCount: 0 PointerCount: 2
```

但是在传递给 DbgkpCloseObject 函数的参数中，句柄计数仍是 1。

在 DbgkpCloseObject 确认需要关闭调试对象后，它接下来做的一个主要动作就是列举系统内的所有进程，对于每个进程检查它的 DebugPort 字段，看是否与要关闭的调试对象相同（与参数 2 比较），如果相同，那么说明这个进程正在被要退出的调试器进程所调试，于是 DbgkpCloseObject 会对这个被调试进程执行 3 个动作：第一，将这个进程的 DebugPort 字段设置为空；第二，调用 DbgkpMarkProcessPeb 函数设置这个进程的 PEB 结构中的 BeingDebugged 字段；第三，检查调试对象的标志字段（Flags），如果包含 KillOnExit 标志（位 1），那么便调用 PspTerminateProcess 终止这个进程，这便是为什么被调试进程随着调试器的退出而退出的原因。

总之，当退出正在调试记事本程序的 WinDBG 程序时，发生的主要动作依次是。

1. [WinDBG] 调用系统服务 NtTerminateProcess，开始终止 WinDBG 进程。
2. [WinDBG] 执行 PspExitThread 函数，WinDBG 的调试工作线程退出。
3. [WinDBG] 执行 PspExitThread 函数，WinDBG 的主线程（UI 线程）退出。
4. [WinDBG] 执行 ObKillProcess 函数，开始清理 WinDBG 进程的句柄表。
5. [WinDBG] 执行 DbgkpCloseObject 函数，关闭调试对象，将被调试进程的 DebugPort 字段设置为空，并调用 DbgkpMarkProcessPeb 设置 PEB 的 BeingDebugged 字段，并引发终止被调试进程（记事本进程）。
6. [记事本] 执行 PspExitThread，记事本进程的主线程开始退出。
7. 执行 PspProcessDelete 和 MmDeleteProcessAddressSpace 函数，删除记事本和 WinDBG 进程的内核对象和进程空间。

每一步骤前的方括号内是函数执行的进程上下文，最后一步可能是在系统服务进程（svchost.exe）环境内执行的，也可能是在系统进程内执行的。

10.8.3 分离被调试进程

Windows XP 允许被调试进程脱离（detach）调试对话并保持运行，也就是在保持调试器进程和被调试进程都不退出的前提下，将二者分离开来。在以前的版本（Windows 2000, NT）中，调试器一旦与被调试进程建立调试关系，那么就只有当二者之一退出时，调试关系才结束，而且调试器退出会强制被调试程序也退出。

调试器可以使用 Windows XP 新引入的调试 API DebugActiveProcessStop 来分离调试对话，其原型为：

```
BOOL DebugActiveProcessStop (DWORD dwProcessId);
```

也就是调试器只要指定要分离的被调试进程的进程 ID，就可以与其终止调试关系了。利用这一特征，调试器可以附加到一个运行的进程，通过生成 DUMP 文件采集其内部的信息，然后再与其安全分离，也就是在基本上不影响目标进程的条件下采集目标进程的信息。这对于调试需要持续运行的进程（如数据库系统或其他系统服务）来说是非常重要的。

值得说明的是，必须在建立调试会话的线程中调用 DebugActiveProcessStop API，否则会得到 0xC0000022 错误（LastError=5，访问被拒绝）。

与其他调试 API 一样，DebugActiveProcessStop 函数也是从 KERNEL32.DLL 中导出的，其内部的工作过程是调用 NTDLL.DLL 中的 DbgUiStopDebugging 函数，清单 10-20 给出了它的伪代码：

清单 10-20 DebugActiveProcessStop API 的伪代码

```
BOOL DebugActiveProcessStop (DWORD dwProcessId)
{
    NTSTATUS nStatus;
    HANDLE hProcess = ProcessIdToHandle(dwProcessId);
    if(hProcess==NULL)
        return FALSE;

    CloseAllProcessHandles(dwProcessId);
    nStatus = DbgUiStopDebugging(hProcess);
    NtClose(hProcess);

    if(NT_SUCCESS(nStatus))
        return TRUE;

    SetLastError(5);
    return FALSE;
}
```

DbgUiStopDebugging 函数的实现也非常简单，只是调用内核服务 NtRemoveProcessDebug()。

```
NTSTATUS DbgUiStopDebugging (HANDLE hProcess)
{
    return NtRemoveProcessDebug(hProcess, NtCurrentPeb()->DbgSsReserved[1]);
```

可见，DbgUiStopDebugging 向 NtRemoveProcessDebug 传递了两个参数，一个是被调试进程的句柄，另一个是调试器工作线程中保存着的调试对象句柄。NtRemoveProcessDebug 内部会调用调试子系统的 DbgkClearProcessDebugObject 函数，简单来说，后者就是去除调试进程的 DebugPort 字段对调试对象的引用，并将其设置为 NULL。将 DebugPort 设为空后，DbgkClearProcessDebugObject 会遍历调试对象的调试事件队列，删除有关这个被调试进程的事件。

WinDBG 调试器提供的分离被调试进程（Detach Debuggee）的功能就是使用以上方法实现的。

10.8.4 退出时分离

Windows XP 引入的与分离调试会话有关的另一个功能就是可以设置当调试器退出时不强制退出被调试进程，也就是当调试器退出时分离被调试进程，而不是将其一起退出，这可以预防调试器意外终止导致被调试进程也被“杀掉”。

在上一段的介绍中，当退出调试器时，被调试的记事本程序也被强制退出了。但是当时我们说，在调用终止进程的函数前，`DbgkpCloseObject` 会检查调试对象的 `Flags` 字段是否包含 `KillOnExit` 标志，如果清除了这个标志，那么便不会退出被调试进程，`DebugSetProcessKillOnExit()` API 就是用来设置这个标志的。

```
BOOL DebugSetProcessKillOnExit(BOOL KillOnExit);
```

如果 `KillOnExit` 参数为真，那么调试器线程的退出会导致系统终止被调试进程。否则调试器线程的退出不会导致它所调试的被调试进程终止。有 3 点值得注意的是：第一，这里说的调试器线程是指调试器进程中启动调试会话的那个线程。第二，这里说的是线程，也就是说即使调试器进程中其他线程仍然运行，只要启动调试会话的线程退出了，那么对应的被调试程序就会退出（假设没有调用过 `DebugSetProcessKillOnExit`）。第三，如果调试器同时调试多个进程，那么这个设置会影响所有的进程，因为这个标志是设置在调试对象中的。

10.9 本章总结

本章按照创建调试会话、使用调试会话和终止调试会话的顺序深入地介绍了用户态调试的整个过程。下一章我们将开始本篇的另一个主题，即 Windows 操作系统中的异常分发和处理。

参考文献

1. Alex Ionescu. Kernel User-Mode Debugging Support (Dbgk).
<http://www.alex-ionescu.com/dbgk-3.pdf>



中断和异常管理

在第 3 章中我们从硬件（CPU）角度介绍了中断和异常机制，本章将从操作系统（Windows）的角度来进一步讨论中断和异常机制。包括管理中断和异常的核心数据结构（11.1 节和 11.2 节），Windows 分发异常的基本过程（11.3 节），以及 Windows 系统的结构化异常处理（SEH）和向量化异常处理（VEH）机制（11.4 节和 11.5 节）。

除非特别说明，本章的内容是针对 Windows XP 的 32 位版本的，但是绝大多数内容也适用于 Windows 的其他 32 位版本（NT 和 Windows 2000 和 Windows Vista），并且可以比较容易地推广到 64 位版本的 Windows。

11.1 中断描述符表

在保护模式下，当有中断或异常发生时，CPU 是通过中断描述符表（Interrupt Descriptor Table，简称 IDT）来寻找处理函数的。因此，可以说 IDT 表是 CPU（硬件）与操作系统（软件）交接中断和异常的关口（Gate）。操作系统在启动早期的一个重要任务就是设置 IDT 表，准备好处理异常和中断的各个函数。

11.1.1 概况

简单来说，IDT 表是一张位于物理内存中的线性表，共有 256 个表项。在 IA-32e（64 位）模式下，每个 IDT 表项的长度是 16 个字节，IDT 表的总长度是 4096 字节（4KB）。在 32 位模式下，每个 IDT 表项的长度是 8 个字节，IDT 表的总长度是 2048 字节（2KB）。32 位与 64 位的主要差异在于地址长度的变化，因此，下文只讨论 32 位的情况。

IDT 表的位置和长度是由 CPU 的 IDTR 寄存器来描述的。IDTR 寄存器共有 48 位，高 32 位是 IDT 表的基地址，低 16 位是 IDT 表的长度（limit）。LIDT（Load IDT）指令用于将操作数指定的基地址和长度加载到 IDTR 寄存器中，也就是改写 IDTR 寄存器的内容。SIDT（Store IDT）指令用于将 IDTR 寄存器的内容写到内存变量中，也就

是读取 IDTR 寄存器的内容。LIDT 和 SIDT 指令只能在实模式或保护模式的高特权级 (Ring 0) 下执行。在内核调试时，可以使用 rigtr 和 rigtl 命令观察 IDTR 寄存器的内容（参见 2.6.2 节）。

在 Windows 操作系统中，IDT 表的初始化过程大致是这样的。IDT 的最初建立和初始化工作是由 Windows 系统的加载程序 (NTLDR 或 WinLoad) 在实模式下完成的。在准备好一个内存块后，加载程序先执行 CLI 指令关闭中断处理，然后执行 LIDT 指令将 IDT 表的位置和长度信息加载到 CPU 中，而后，加载程序将 CPU 从实模式切换到保护模式，并将执行权移交给 NT 内核的入口函数 KiSystemStartup。接下来，内核中的处理器初始化函数会通过 SIDT 指令取得 IDT 表的信息，对其进行必要的调整，然后以参数形式传递给 KiInitializePci 函数，后者将其记录到描述处理器的基本数据区 PCR (Processor Control Region) 和 Prcb (Processor Control Block) 中。

以上介绍的过程都是发生在 0 号处理器中的，也就是所谓的 Bootstrap Processor，简称为 BSP。因为即使是多 CPU 的系统，在把 NTLDR 或 WinLoad 及执行权移交给内核的阶段都只有 BSP 在运行。在 BSP 完成了内核初始化和执行体的阶段 0 初始化后，在阶段 1 初始化时，BSP 才会执行 KeStartAllProcessors 函数来初始化其他 CPU，称为 AP (Application Processor)。对于每个 AP，KeStartAllProcessors 函数会为其建立一个单独的处理器状态区，包括它的 IDT 表，然后调用 KiInitProcessor 函数，后者会根据启动 CPU 的 IDT 表为要初始化的 AP 复制一份，并作必要的修改。

在内核调试会话中，可以使用 !pcr 命令观察 CPU 的 PCR 内容，清单 11-1 显示了 Windows Vista 系统中 0 号 CPU 的 PCR 内容。

清单 11-1 观察处理器的控制区 (PCR)

```

kd> !pcr
KPCR for Processor 0 at 81969a00: // KPCR 结构的线性内存地址
    Major 1 Minor 1 // KPCR 结构的主版本号和子版本号
    NtTib.ExceptionList: 9f1d9644 // 异常处理注册链表
    [...] // 省略数行关于 NTTIB 的信息
        SelfPcr: 81969a00 // 本结构的起始地址
        Prcb: 81969b20 // KPRCB 结构的地址
        Irql: 0000001f // CPU 的中断请求级别 (IRQL)
        IRR: 00000000 //
        IDR: ffff20f0 //
    InterruptMode: 00000000 //
        IDT: 834da400 // IDT 表的基址
        GDT: 834da000 // GDT 表的基址
        TSS: 8013e000 // 任务状态段 (TSS) 的地址

    CurrentThread: 84af6270 // 当前在执行的线程，ETHREAD 地址
    NextThread: 00000000 // 下一个准备执行的线程
    IdleThread: 8196cdc0 // IDLE 线程的 ETHREAD 结构地址

```

内核数据结构 KPCR 描述了 PCR 内存区的布局，因此也可以使用 dt 命令来观察 PCR，例如，kd> dt nt!_KPCR 81969a00。

11.1.2 门描述符

IDT 表的每个表项是一个所谓的门描述符（Gate Descriptor）结构。之所以这样称呼，是因为 IDT 表项的基本用途就是引领 CPU 从一个空间到另一个空间去执行，每个表项好像是一个从一个空间进入到另一个空间的大门（Gate）。在穿越这扇门时 CPU 会做必要的安全检查和准备工作。

IDT 表中可以包含以下 3 种门描述符。

- 任务门（task-gate）描述符：用于任务切换，里面包含用于选择任务状态段（TSS）的段选择子。可以使用 JMP 或 CALL 指令通过任务门来切换到任务门所指向的任务，当 CPU 因为中断或异常转移到任务门时，也会切换到指定的任务。
- 中断门（interrupt-gate）描述符：用于描述中断处理例程的入口。
- 陷阱门（trap-gate）描述符：用于描述异常处理例程的入口。

图 11-1 描述了以上 3 种门描述符的内容布局。

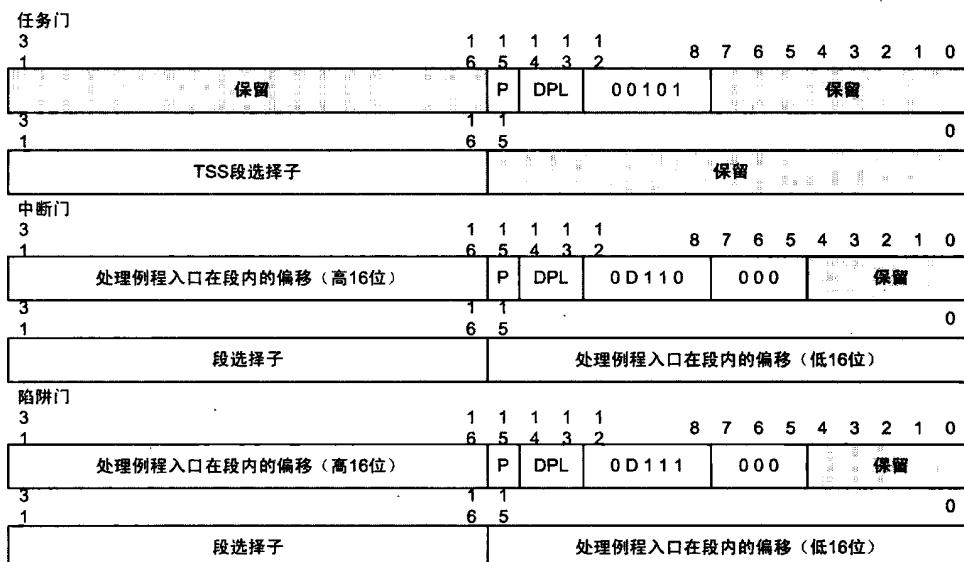


图 11-1 IDT 中的 3 种门描述符

从图 11-1 中可以看出，3 种描述符的格式非常相似，有很多共同的字段。其中 DPL 代表描述符优先级（Descriptor Privilege Level），用于优先级控制，P 是段存在标志。段选择子用于选择一个段描述符（位于 LDT 或 GDT 中，选择子的格式参见 2.6.3 节），偏移部分用来指定段中的偏移，二者共同定义一个准确的内存位置，对于中断门和陷

阱门，它们指定的就是中断或异常处理例程的地址，对于任务门，它们指定的就是任务状态段的内存地址。

系统通过门描述符的类型字段，即高 4 字节的 6~12 位，来区分一个描述符的种类。例如任务门的类型是 0b00101（b 代表二进制数），中断门的类型是 0b0D110，其中 D 位用来表示描述的是 16 位门（0）还是 32 位门（1），陷阱门的类型是 0b0D111。

11.1.3 执行中断和异常处理函数

下面我们看看当有中断或异常发生时，CPU 是如何通过 IDT 表寻找和执行处理函数的。首先，CPU 会根据其向量号码和 IDTR 寄存器中的 IDT 表基址信息找到对应的门描述符。然后判断门描述符的类型，如果是任务描述符，那么 CPU 会执行硬件方式的任务切换，切换到这个描述符所定义的线程，如果是陷阱描述符或中断描述符，那么 CPU 会在当前任务上下文中调用描述符所描述的处理例程。下面分别加以讨论。

我们先来看任务门的情况。简单来说，任务门描述的是一个 TSS 段，CPU 要做的是切换到这个 TSS 段所代表的线程，然后开始执行这个线程。TSS 段是用来保存任务信息的一段内存区，其格式是 CPU 所定义的。图 11-2 给出了 IA-32 CPU 的 TSS 段格式。从中我们看到 TSS 段中包含了一个任务的关键上下文信息，如段寄存器、通用寄存器和控制寄存器，其中特别值得注意的是靠下方的 SS0~SS2 和 ESP0~ESP2 字段，它们记录着一项任务在不同优先级执行时所应使用的栈，SSx 用来选择栈所在的段，ESPx 是栈指针值。

31	15	0
I/O Map Base Address	Reserved	T
Reserved	LDT Segment Selector	100
Reserved	GS	96
Reserved	FS	92
Reserved	DS	88
Reserved	SS	84
Reserved	CS	80
Reserved	ES	76
	EDI	72
	ESI	68
	EBP	64
	ESP	60
	EBX	56
	EDX	52
	ECX	48
	EAX	44
	EFLAGS	40
	EIP	36
	CR3(PDBR)	32
Reserved	SS2	28
Reserved	ESP2	24
Reserved	SS1	20
Reserved	ESP1	16
Reserved	SS0	12
Reserved	ESP0	8
	Previous Task Link	4
	Reserved bits. Set to 0.	0

图 11-2 32 位的任务状态段 (TSS)

CPU 在通过任务门的段选择子找到 TSS 段描述符后，会执行一系列的检查动作，比如确保 TSS 段描述符中的存在标志是 1，边界值应该大于 0x67，B（Busy）标志不为 1 等。所有检查都通过后，CPU 会将当前任务的状态保存到当前任务的 TSS 段中。然后把 TSS 段描述符中的 B 标志设置为 1。接下来，CPU 要把新任务的段选择子（与门描述符中的段选择子等值）加载到 TR 寄存器，然后把新任务的寄存器信息加载到物理寄存器中。最后，CPU 开始执行新的任务。

下面通过一个小实验来加深大家的理解。首先，在一个调试 Windows Vista 的内核调试会话中，通过 `r idtr` 命令得到系统 IDT 表的基地址：

```
kd> r idtr
idtr=834da400
```

因为双重错误异常（Double Fault, #DF）通常是使用任务门来处理的，所以我们观察这个异常对应的 IDT 表项，因为#DF 异常的向量号是 8，每个 IDT 表项的长度是 8，所以我们可以使用如下命令显示出 8 号 IDT 表项的内容：

```
kd> db 834da400+8*8 18
834da440 00 00 50 00 00 85 00 00 ...P.....
```

其中第 2, 3 两个字节（0 数起，下同）组成的 WORD 是段选择子，即 0x0050。第 5 个字节（0x85）是 P 标志（为 1）、DPL（0b00）和类型（0b00101）。

接下来使用 `dg` 命令显示段选择子所指向的段描述符：

```
kd> dg 50
          P  Si Gr Pr Lo
Sel   Base     Limit    Type      l  ze an es ng Flags
----- -----
0050  81967000 00000068 TSS32 Avl  0  Nb By P   N1  00000089
```

也就是说，TSS 段的基地址是 0x81967000，长度是 0x68 个字节（Gran 位指示 By 即 Byte）。Type 字段显示这个段的类型是 32 位的 TSS 段（TSS32），它的状态为可用（Available），并非 Busy。

至此，我们知道了#DF 异常对应的门描述符所指向的 TSS 段，是位于内存地址 0x81967000 开始的 0x68 个字节。使用内存观察命令便可以显示这个 TSS 的内容了（见清单 11-2）。

清单 11-2 观察#DF 门描述符所指向的 TSS 段

```
kd> dd 81967000
81967000 00000000 81964000 00000010 00000000
81967010 00000000 00000000 00000000 00122000
81967020 8193f0ao 00000000 00000000 00000000
81967030 00000000 00000000 81964000 00000000
81967040 00000000 00000000 00000023 00000008
81967050 00000010 00000023 00000030 00000000
81967060 00000000 20ac0000 00000000 81964000
81967070 00000010 00000000 00000000 00000000
```

参考清单 11-2，从上至下，81964000 是在优先级 0 执行时的栈指针，00000010

是优先级 0 执行时的栈选择子，00122000 是这个任务的页目录基地址寄存器（PDBR，即 CR3）的值，8193f0a0 是程序指针寄存器（EIP）的值，当 CPU 切换到这个任务时便是从这里开始执行的。接下来，依次是标志寄存器（EFLAGS）和通用寄存器的值。偏移 0x48 处的 0x23 是 ES 寄存器的值，相邻的 00000008 是 CS 寄存器的值，即这个任务的代码段的选择子。而后是 SS 寄存器的值，即栈段的选择子，再往后是 DS、FS 和 GS 寄存器的值（0x23、0x30 和 0）。偏移 0x64 处的 20ac0000 是 TSS 的最后 4 个字节，它的最低位是 T 标志（0），即我们在第 4 章介绍过的 TSS 段中的陷阱标志。高 16 字节是用来定位 IO 映射区基地址的偏移地址，它是相对于 TSS 的基地址的。

使用 `ln` 命令可以观察 EIP 的值对应的就是内核函数 `KiTrap08`：

```
kd> ln 8193f0a0
(8193f0a0)  nt!KiTrap08  |  (8193f118)  nt!Dr_kit9_a
Exact matches:
nt!KiTrap08 = <no type information>
```

也就是说，当有#DF 异常发生时，CPU 便会切换到以上 TSS 所描述的线程，然后在这个线程环境中执行 `KiTrap08` 函数。之所以要切换到一个新的线程，而不是像其他异常那样在原来的线程中处理，是因为#DF 异常指的是在处理一个异常时又发生了异常，这可能意味着本来的线程环境已经不可靠了，所以有必要切换到一个新的线程来执行。

类似的，代表紧急任务的不可屏蔽中断（NMI）也是使用任务门机制来处理的。最后要说明的是，因为 x64 架构不支持硬件方式的任务切换，所以 IDT 表中也不再有任务门了。

大多数中断和异常都是利用中断门或陷阱门来处理的，下面我们看看这两种情况。

首先，CPU 会根据门描述符中的段选择子定位到段描述符，然后再进行一系列检查，如果检查通过后，CPU 就判断是否需要切换栈。如果目标代码段的特权级别比当前特权级别高（级别的数值小），那么 CPU 需要切换栈，其方法是从当前任务的任务状态段（TSS）中读取新堆栈的段选择子（SS）和堆栈指针（ESP），并将其加载到 SS 和 ESP 寄存器。然后，CPU 会把被中断过程（旧的）的堆栈段选择子（SS）和堆栈指针（ESP）压入新的堆栈。接下来，CPU 会执行如下两项操作。

- 把 EFLAGS、CS 和 EIP 的指针压入堆栈。CS 和 EIP 指针代表了转到处理例程前 CPU 正在执行代码的位置。
- 如果发生的是异常，而且该异常具有错误代码（参见 3.3.2 节），那么把该错误代码也压入堆栈。

如果处理例程所在代码段的特权级别与当前特权级别相同，那么 CPU 便不需要进行堆栈切换，但仍要执行上面的两步操作。

TR 寄存器中存放着指向当前任务 TSS 段的段选择子，使用 WinDBG 可以观察 TSS 段的内容。

```

kd> r tr
tr=00000028
kd> dg 28
          P Si Gr Pr Lo
          l ze an es ng Flags
Sel   Base    Limit     Type
-----
0028 8013e000 000020ab TSS32 Busy  0 Nb By P NL 0000008b

```

经常做内核调试的读者可能会发现，TR 寄存器的值大多时候是固定的，也就是说，并不随着应用程序的线程切换而变化。事实上，Windows 系统中的 TSS 个数并不是与系统中的线程个数相关的，而是与 CPU 个数相关的。在启动期间，Windows 会为每个 CPU 创建 3~4 个 TSS，一个用于处理 NMI，一个用于处理#DF 异常，一个处理机器检查异常（与版本有关，在 XP SP1 中存在），另一个供所有 Windows 线程所共享。当 Windows 切换线程时，它把当前线程的状态复制到共享的 TSS 中。也就是说，普通的线程切换并不会切换 TSS，只有当 NMI 或#DF 异常发生时，才会切换 TSS，这就是所谓的以软件方式切换线程（任务）。

11.1.4 IDT 表一览

使用 WinDBG 的!idt 扩展命令可以列出 IDT 表中的各个表项，不过该命令做了很多翻译，显示出的不是门描述符的原始格式。

```

1kd> !idt -a
Dumping IDT:
00: 804dbe13 nt!KiTrap00 // 0 号异常，即除 0
01: 804dbf6b nt!KiTrap01
02: Task Selector = 0x0058 // NMI 的门描述符，显示的是 TSS 段的段选择子
03: 804dc2bd nt!KiTrap03

```

表 11-1 列出了 Windows Vista (32 位) 的 IDT 表设置，对于其他的 Windows 版本或硬件配置不同的系统，某些表项可能有所不同，但是大多数表项是一致的。

表 11-1 IDT 表设置一览 (Windows Vista 32 位)

向量号	门类型	处理器/TSS 选择子	中断/异常	说明
00	中断	nt!KiTrap00	除零错误	
01	中断	nt!KiTrap01	调试异常	
02	任务	0x0058	不可屏蔽中断 (NMI)	切换到系统线程处理该中断，使用的函数是 KiTrap02
03	中断	nt!KiTrap03	断点	
04	中断	nt!KiTrap04	溢出	
05	中断	nt!KiTrap05	数组越界	
06	中断	nt!KiTrap06	无效指令	
07	中断	nt!KiTrap07	数学协处理器不存在或不可用	

续表

向量号	门类型	处理例程/TSS 选择子	中断/异常	说明
08	任务	0x0050	双重错误 (Double Fault)	切换到系统线程处理该异常，执行 KiTrap08
09	中断	nt!KiTrap09	协处理器段溢出	
0a	中断	nt!KiTrap0A	无效的 TSS	
0b		nt!KiTrap0B	段不存在	
0c		nt!KiTrap0C	栈段错误	
0d		nt!KiTrap0D	一般保护错误	
0e		nt!KiTrap0E	页错误	
0f		nt!KiTrap0F	保留	
10		nt!KiTrap10	浮点错误	
11		nt!KiTrap11	内存对齐	
12		nt!KiTrap0F	机器检查	
13		nt!KiTrap13	SIMD 浮点错误	
14~1f		nt!KiTrap0F	保留	KiTrap0F 会引发 0x7F 号蓝屏
20~29	00	NULL		(未使用)
2a		nt!KiGetTickCount		
2b		nt!KiCallbackReturn		从逆向调用返回 (参见 8.3.4 节)
2c		nt!KiRaiseAssertion		断言
2d		nt!KiDebugService		调试服务
2e		nt!KiSystemService		系统服务
2f		nt!KiTrap0F		
30		hal!Halp8254ClockInterrupt	IRQ0	时钟中断
31~3F		驱动程序通过 KINTERRUPT 结构注册的处理例程	IRQ1~IRQ15	其他硬件设备的中断
40~FD		nt!KiUnexpectedInterruptX	N/A	没有使用

在 Windows XP 系统中，处理机器检查异常 (#MC) 的 18 号表项处是一个任务门描述符，指向一个单独的 TSS 段，对应的处理函数是 hal 模块中的 HalpMcaExceptionHandlerWrapper。

11.2 异常的描述和登记

为了更好地管理异常，Windows 系统定义了专门的数据结构来描述异常，并定义了一系列代码来标识典型的异常。

在操作系统层次，除了 CPU 产生的异常，还有通过软件方式模拟出的异常，比如调用 `RaiseException API` 而产生的异常和使用编程语言的 `throw` 关键字抛出的异常。为了行文方便，我们把前一类称为 CPU 异常（或硬件异常），后一类称为软件异常。`Windows` 是使用统一的方式来描述和分发这两类异常的。本节介绍异常的描述方式，下一节将介绍异常的分发过程。

11.2.1 EXCEPTION_RECORD 结构

`Windows` 使用 `EXCEPTION_RECORD` 结构来描述异常，清单 11-3 给出了这个结构的定义。

清单 11-3 EXCEPTION_RECORD 结构

```
typedef struct _EXCEPTION_RECORD {
    DWORD ExceptionCode;                                // 异常代码
    DWORD ExceptionFlags;                               // 异常标志
    struct _EXCEPTION_RECORD* ExceptionRecord;          // 相关的另一个异常
    PVOID ExceptionAddress;                            // 异常发生地址
    DWORD NumberParameters;                           // 参数数组中的元素个数
    ULONG_PTR ExceptionInformation[EXCEPTION_MAXIMUM_PARAMETERS]; // 参数数组
} EXCEPTION_RECORD, *PEXCEPTION_RECORD;
```

其中 `ExceptionCode` 为异常代码，是一个 32 位的整数，其格式是 `Windows` 系统的状态代码格式，`NtStatus.h` 中包含了已经定义的所有状态代码，在 `WinBase.h` 中可以看到异常代码只是状态代码的别名，例如：

```
#define EXCEPTION_BREAKPOINT           STATUS_BREAKPOINT
#define EXCEPTION_SINGLE_STEP          STATUS_SINGLE_STEP
```

表 11-2 列出了常见的用于异常代码的状态代码。

`ExceptionFlags` 字段用来记录异常标志，它的每一位代表一种标志，目前已经定义的标志位有。

- `EH_NONCONTINUABLE` (1)，该异常不可恢复继续执行。
- `EH_UNWINDING` (2)，当因为执行栈展开而调用异常处理函数时，会设置此标志。
- `EH_EXIT_UNWIND` (4)，也是用于栈展开，较少使用。
- `EH_STACK_INVALID` (8)，当检测到栈错误时，设置此标志。
- `EH_NESTED_CALL` (0x10)，用于标识内嵌的异常（参见第 24 章）。
- `EH_NONCONTINUABLE` 位用来表示该异常是否可以恢复继续执行，如果试图恢复运行一个不可继续的异常，便会导致 `EXCEPTION_NONCONTINUABLE_EXCEPTION` 异常。

`ExceptionRecord` 指针指向与该异常有关的另一个异常记录，如果没有相关的异常，

那么这个指针便为空。

表 11-2 Windows 的异常代码

异常代码	值	来源
EXCEPTION_ACCESS_VIOLATION	0xC0000005L	非法访问（访问违例）
EXCEPTION_DATATYPE_MISALIGNMENT	0x80000002L	CPU 的对齐检查异常, #AC (17)
EXCEPTION_BREAKPOINT	0x80000003L	CPU 的断点异常, #BP (3)
EXCEPTION_SINGLE_STEP	0x80000004L	CPU 的调试异常, #DB (1)
EXCEPTION_ARRAY_BOUNDS_EXCEEDED	0xC000008CL	CPU 的数组越界异常, #BR (5)
EXCEPTION_FLT_DIVIDE_BY_ZERO	0xC000008EL	CPU 的协处理器异常, #NM (7)
EXCEPTION_FLT_INEXACT_RESULT	0xC000008FL	CPU 的协处理器异常, #NM (7)
EXCEPTION_FLT_INVALID_OPERATION	0xC0000090L	CPU 的协处理器异常, #NM (7)
EXCEPTION_FLT_OVERFLOW	0xC0000091L	CPU 的协处理器异常, #NM (7)
EXCEPTION_FLT_STACK_CHECK	0xC0000092L	CPU 的协处理器异常, #NM (7)
EXCEPTION_FLT_UNDERFLOW	0xC0000093L	CPU 的协处理器异常, #NM (7)
EXCEPTION_INT_DIVIDE_BY_ZERO	0xC0000094L	CPU 的除零异常, #DE (0)
EXCEPTION_INT_OVERFLOW	0xC0000095L	CPU 的溢出异常, #OF (4)
EXCEPTION_PRIV_INSTRUCTION	0xC0000096L	CPU 的一般保护异常, #GP (13)
EXCEPTION_IN_PAGE_ERROR	0xC0000006L	CPU 的页错误异常, #PF (14)
EXCEPTION_ILLEGAL_INSTRUCTION	0xC000001DL	CPU 的无效指令异常, #UD (6)
EXCEPTION_NONCONTINUABLE_EXCEPTION	0xC0000025L	见上文
EXCEPTION_STACK_OVERFLOW	0xC00000FDL	CPU 的页错误异常, #PF (14)
EXCEPTION_GUARD_PAGE	0x80000001L	CPU 的页错误异常, #PF (14)
CONTROL_C_EXIT	0xC000013AL	仅适用于控制台程序, 当用户按 Ctrl+C 或 Ctrl+Break 键时

ExceptionAddress 字段用来记录异常地址, 对于硬件异常, 它的值因为异常类型不同而可能是导致异常的那条指令的地址, 或者是导致异常指令的下一条指令的地址。例如, 对于非法访问异常 (EXCEPTION_ACCESS_VIOLATION), 属于错误类异常 (Fault), ExceptionAddress 的值是导致异常的那条指令的地址。对于数据断点触发的调试异常, 属于陷阱类异常 (Trap), ExceptionAddress 的值是导致异常指令的下一条指令的地址。

NumberParameters 是附加参数的个数, 即 ExceptionInformation 数组中包含的有效参数个数, 该结构最多允许存储 15 个附加参数。

导致非法访问异常的原因主要来源于 CPU 的页错误异常#PF (14), 但也可能是系统检测到的其他违反系统规则的情况。

11.2.2 登记 CPU 异常

对于 CPU 异常, KiTrapXX 例程在完成针对本异常的特别动作后, 通常会调用 CommonDispatchException 函数, 并通过寄存器将如下信息传递给这个函数。

- 将唯一标识该异常的一个异常代码 (表 11-2) 放入 EAX 寄存器。
- 将导致异常的指令地址放入 EBX 寄存器。
- 将其他信息作为附带参数 (最多 3 个) 分别放入 EDX (参数 1)、ESI (参数 2) 和 EDI (参数 3) 寄存器, 并将参数个数放入 ECX 寄存器。

CommonDispatchException 被调用后, 它会在栈中分配一个 EXCEPTION_RECORD 结构, 并把以上异常信息存储到该结构中。在准备好这个结构后, 它会调用内核中的 KiDispatchException 函数来分发异常。

11.2.3 登记软件异常

下面看看软件异常的产生和登记过程。简单来说, 软件异常是通过直接或间接调用内核服务 NtRaiseException 而产生的。

```
NTSTATUS NtRaiseException (IN PEXCEPTION_RECORD ExceptionRecord,
    IN PCONTEXT ContextRecord, IN BOOLEAN FirstChance )
```

用户模式中的程序可以通过 RaiseException() API 来调用这个内核服务。RaiseException API 是由 KERNEL32.DLL 导出的 API, 供应用程序产生“自定义”的异常, 其原型如下:

```
void RaiseException( DWORD dwExceptionCode, DWORD dwExceptionFlags,
    DWORD nNumberOfArguments, const DWORD* lpArguments );
```

其中 dwExceptionCode 是异常代码, 可以是表 11-2 中的代码, 也可以是应用程序自己定义的代码。lpArguments 和 nNumberOfArguments 用来定义异常的常数, 相当于 EXCEPTION_RECORD 结构中的 ExceptionInformation 和 NumberParameters。事实上, RaiseException 的实现也很简单, 它只是将参数放入一个 EXCEPTION_RECORD 后便调用 NTDLL 中的 RtlRaiseException()。RtlRaiseException 会将当前的执行上下文 (通用寄存器等) 放入 CONTEXT 结构, 然后通过 NTDLL 中的系统服务调用机制调用内核中的 NtRaiseException。

NtRaiseException 内部会调用另一个内核函数 KiRaiseException:

```
NTSTATUS KiRaiseException (IN PEXCEPTION_RECORD ExceptionRecord,
    IN PCONTEXT ContextRecord, IN PKEXCEPTION_FRAME ExceptionFrame,
    IN PKTRAP_FRAME TrapFrame, IN BOOLEAN FirstChance )
```

ExceptionRecord 是指向异常记录的指针, ContextRecord 是指向线程上下文结构 (CONTEXT) 的指针, ExceptionFrame 对于 x86 平台总是为空, TrapFrame 就是栈帧的基址, FirstChance 表示这是该异常的第一轮 (TRUE) 还是第二轮 (FALSE) 处理机会。

内核中的代码可以通过 `RtlRaiseException`（相当于 NTDLL 中的版本）来调用 `NtRaiseException` 和 `KiRaiseException`。也就是说，不论是从用户模式调用 `RaiseException` API，还是从内核模式调用相应的函数，最后都会转到 `KiRaiseException`。

`KiRaiseException` 内部会通过 `KeContextToKframes` 例程把 `ContextRecord` 结构中的信息复制到当前线程的内核栈，然后把 `ExceptionRecord` 中的异常代码的最高位清 0，以便把软件产生的异常与 CPU 异常区分开来。接下来 `KiRaiseException` 会调用 `KiDispatchException` 开始分发该异常。

对于 Visual C++ 程序抛出的异常，比如 MFC 中从 `CException` 派生来的各个异常类对应的异常，`throw` 关键字直接对应的是 `CxxThrowException` 函数，`CxxThrowException` 会调用 `RaiseException`，并将 `ExceptionCode` 参数固定为 `0xe06d7363`（对应的 ASCII 码为.msc）。接下来的过程与上面直接调用 `RaiseException` 的情况相同。因为 C++ 异常的实现与编译器有关，所以本书只讨论使用 Visual C++ 编译器的情况。

.NET 程序抛出的异常（CLR 异常）也是通过 `RaiseException` API 产生的，其异常代码固定为 `0xe0434f4d`（对应的 ASCII 码为.COM）。

综上所述，不论是 CPU 异常还是软件异常，尽管产生的原因不同，但最终都会调用内核中的 `KiDispatchException` 来分发异常，也就是说，Windows 是使用统一的方法来分发 CPU 异常和软件异常的。

11.3 异常分发过程

根据前面两节的介绍，当有异常发生时，CPU 会通过 IDT 表找到异常处理函数，即内核中的 `KiTrapXX` 系列函数，然后转去执行。但是，`KiTrapXX` 函数通常只是对异常作简单的表征和描述，为了支持调试和软件自己定义的异常处理函数，系统需要将异常分发给调试器或应用程序的处理函数。对于软件异常，Windows 系统采用的策略是以和 CPU 异常统一的方式来分发和处理的，本节我们将介绍分发异常的核心函数 `KiDispatchException` 和它的工作过程。

11.3.1 KiDispatchException 函数

Windows 内核中的 `KiDispatchException` 函数是分发各种 Windows 异常的枢纽。其函数原型如下：

```
VOID KiDispatchException ( IN PEXCEPTION_RECORD ExceptionRecord,
                         IN PKEXCEPTION_FRAME ExceptionFrame, IN PKTRAP_FRAME TrapFrame,
                         IN KPROCESSOR_MODE PreviousMode, IN BOOLEAN FirstChance )
```

其中，参数 `ExceptionRecord` 指向的是我们上一节介绍的 `EXCEPTION_RECORD` 结

构，用来描述要分发的异常。参数 `ExceptionFrame` 对于 x86 系统总是为 `NULL`。参数 `TrapFrame` 指向的是 `KTRAP_FRAME` 结构，用来描述异常发生时的处理器状态，包括各种通用寄存器、调试寄存器、段寄存器等。参数 `PreviousMode` 是一个枚举类型的常量，DDK 的头文件中有这个枚举类型的定义：

```
typedef enum _MODE { KernelMode, UserMode, MaximumMode} MODE;
```

也就是说，`PreviousMode` 等于 0 表示前一个模式（通常是触发异常代码的执行模式）是内核模式，1 表示用户模式。`FirstChance` 参数表示是否是第一轮分发这个异常。对于一个异常，Windows 系统会最多分发两轮。

图 11-3 画出了 `KiDispatchException` 分发异常的基本过程（示意图）。

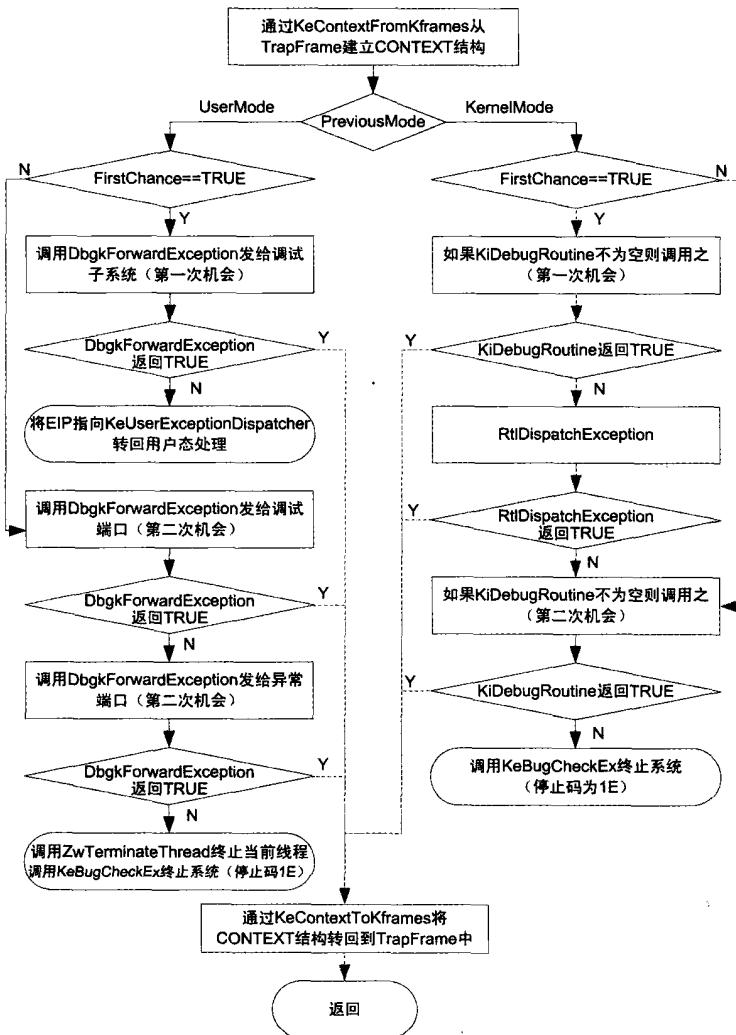


图 11-3 `KiDispatchException` 分发异常的流程图

从图 11-3 中可以看到, KiDispatchException 会先调用 KeContextFromKframes 函数, 目的是根据 TrapFrame 参数指向的 KTRAP_FRAME 结构产生一个 CONTEXT 结构, 以供向调试器和异常处理器函数报告异常时使用。

接下来, 根据前一个模式(异常发生的模式)是内核模式还是用户模式, KiDispatchException 会选取左右两个流程之一来分发异常, 下面我们分别作进一步说明。

11.3.2 内核态异常的分发过程

如果前一模式是内核模式, 也就是 PreviousMode 参数等于 KernelMode(0), 那么 KiDispatchException 会执行图 12-3 中右侧的路线。

具体来说, 对于第一轮处理机会(FirstChance==TRUE), KiDispatchException 会试图先通知内核调试器来处理该异常。内核变量 KiDebugRoutine 用来标识内核调试引擎交互的接口函数。当内核调试引擎被启用时, KiDebugRoutine 指向的是内核调试引擎的 KdpTrap, 这个函数会进一步把异常信息封装为数据包发送给内核调试器。当内核调试引擎没有启用时, KiDebugRoutine 指向的是 KdpStub 函数。KdpStub 的实现非常简单, 作一些简单的处理后便返回 FALSE。

如果 KiDebugRoutine 返回为 TRUE, 也就是内核调试器处理了该异常, 那么 KiDispatchException 便停止继续分发, 准备返回。如果 KiDebugRoutine 返回 FALSE, 也就是没有处理该异常, 那么 KiDispatchException 会调用 RtlDispatchException, 试图寻找已经注册的结构化异常处理器(SEH)。清单 11-4 给出了 RtlDispatchException 函数的原型和伪代码。

清单 11-4 RtlDispatchException 函数的伪代码

```

BOOLEAN RtlDispatchException(PEXCEPTION_RECORD ExceptionRecord,
    PCONTEXT ContextRecord)
{
    for (RegistrationPointer = RtlpGetRegistrationHead(); // 取异常登记链表的头指针
        RegistrationPointer != EXCEPTION_CHAIN_END;           // 结束条件
        RegistrationPointer = RegistrationPointer->Next)   // 遍历
    {
        // 检查异常登记记录的有效性, 如果有效, 则调用执行其中记录的处理函数
        switch RtlpExecuteHandlerForException()                // 判断函数的执行结果
        {
            case ExceptionContinueExecution: return TRUE      // 异常已处理, 返回真
            case ExceptionContinueSearch: continue             // 没有处理, 继续寻找
            case ExceptionNestedException: ...                  // 内嵌异常, 特别处理
            default: return FALSE                            // 默认返回没有处理
        }
    }
    return FALSE;                                         // 默认返回没有处理异常
}

```

首先, RtlDispatchException 会调用 RtlpGetRegistrationHead 取得异常注

册链表（Exception Registration List）的首节点地址，以下是这个函数的汇编代码：

```
1kd> uf nt!RtlpGetRegistrationHead
nt!RtlpGetRegistrationHead:
80541ffc 64a100000000    mov     eax,dword ptr fs:[00000000h]
80542002 c3              ret
```

在 x86 架构中，FS 段存储的是线程信息块，即 TIB。其偏移 0 开始的 DWORD 指向的便是异常登记链表的首节点地址。接下来，RtlDispatchException 会遍历异常登记链表，依次执行每个异常处理器，如果某一个处理器返回了 ExceptionContinueExecution，那么 RtlDispatchException 便会返回 TRUE，表示已经处理了该异常。我们将在下一节和第 24 章更详细讨论 SEH 和异常处理器。

如果 RtlDispatchException 返回 FALSE，也就是没有找到处理该异常的异常处理器，那么 KiDispatchException 会试图给内核调试器第二轮处理机会。如果这次 KiDebugRoutine 仍然返回 FALSE，那么 KiDispatchException 会认为这是个“无人”处理的异常，简称为未处理异常（unhandled exception）。对于发生在内核态中的未处理异常，Windows 认为这是一个严重的错误，会调用 KeBugCheckEx 引发蓝屏（BSOD），报告错误并终止系统运行。KeBugCheckEx 的第一个参数被置为 KMODE_EXCEPTION_NOT_HANDLED（0x1E），代表未处理的内核异常。异常代码和异常地址会作为参数传给 KeBugCheckEx，并显示在蓝屏界面上。

11.3.3 用户态异常的分发过程

如果前一模式是用户模式，即 PreviousMode 参数等于 UserMode(1)，那么 KiDispatchException 会按左侧的流程执行。首先，KiDispatchException 会判断是否需要发给内核调试器。判断的条件包括，这个异常是否是内核调试器触发的，以及内核调试的设置选项中是否接收用户态异常（参见 18.3.2 节关于/noumex 选项的描述）。如果判断的结果是需要发送，那么则通过内核调试会话发送给主机上的内核调试器。但内核调试器通常不处理用户态的异常，直接返回不处理。因此，大多数时候，KiDispatchException 都会继续执行以下分发过程。

对于第一轮处理机会，KiDispatchException 会试图先将该异常分发给用户态的调试器，方法是调用用户态调试子系统的内核例程 DbgkForwardException。我们在第 9 章（9.2 节）介绍过这个函数，它有三个参数，第一个是异常结构，第二个是一个布尔值，用来指定要发给调试端口，还是异常端口，第三个参数用来指定是否是第二轮处理机会。因此，KiDispatchException 首先会这样调用 DbgkForwardException：DbgkForwardException(ExceptionRecord, TRUE, FALSE)；

也就是发给调试端口，而且指明是第一轮处理机会。DbgkForwardException 会检查当前进程的 DebugPort 字段是否为空，如果不为空，则调用 DbgkpSendApiMessage 将异常发给调试子系统，后者又将异常发给调试器。如果 DbgkpSendApiMessage 返回成功

(STATUS_SUCCESS)，而且调试器处理了该异常(ReturnedStatus == DBG_CONTINUE)，那么 DbgkForwardException 会返回 TRUE，该异常的分发过程也就结束了。

如果 DbgkpSendApiMessage 返回的结果不成功，或者调试器没有处理该异常 (ReturnedStatus == DBG_EXCEPTION_NOT_HANDLED)，那么 DbgkpSendApiMessage 会返回 FALSE，KiDispatchException 下一步的动作是试图寻找异常处理块来处理该异常，因为异常发生在用户态代码中，异常处理块也应该在用户态函数中，KiDispatchException 会准备转回到用户态去执行。内核变量 KeUserExceptionDispatcher 记录了用户态中的异常分发函数。在目前的 Windows 中，它指向的是 NTDLL 中的 KiUserExceptionDispatcher 函数。

如何转回到用户态执行呢？其过程是这样的，KiDispatchException 先确认用户态栈有足够的空间容纳 CONTEXT 结构和 EXCEPTION_RECORD 结构，然后将这两个结构复制到用户态栈中。而后，KiDispatchException 会将 TrapFrame 所指向的 KTRAP_FRAME 结构中的状态信息调整为在用户态执行所需的合适值，包括段寄存器和栈指针。最后 KiDispatchException 将 KeUserExceptionDispatcher 的值赋给 KTRAP_FRAME 结构中的程序指针 (EIP) 字段，目的是让这个线程返回用户态后从 KiUserExceptionDispatcher 函数处开始执行。以上工作做好后，KiDispatchException 便返回。对于软件异常，当前线程会返回到 KiRaiseException，再返回到 NtRaiseException，而后通过系统服务返回流程返回到用户态。因为 KTRAP_FRAME 信息已经被修改了，所以当前线程回到用户态后会从 KiUserExceptionDispatcher 函数开始执行，而不是本来调用系统服务的地方。对于 CPU 异常，KiDispatchException 会返回到 CommonDispatchException 函数，然后执行 KiExceptionExit 并根据 TrapFrame 恢复 CPU 状态，而后执行异常返回指令 (IRETD)。因为 TrapFrame 信息被修改过了，所以异常返回指令执行后，当前线程便转到用户态的 KiUserExceptionDispatcher 函数处开始执行。

回到用户态后，KiUserExceptionDispatcher 会通过调用 RtlDispatchException 来寻找异常处理器，具体细节我们将在下一节及第 24 章讨论。如果 RtlDispatchException 返回 TRUE，那么表示已经有异常处理器处理了该异常，KiUserExceptionDispatcher 会调用 ZwContinue 系统服务继续执行原来发生异常的代码。该调用如果成功，便不会再返回到 KiUserExceptionDispatcher 函数中，如果返回，则说明 Ccontinue 失败，KiUserExceptionDispatcher 会通过调用 RtlRaiseException 来抛出异常。

与前面介绍的内核态的情况一样，多个异常处理器是以链表形式连接在一起的，其头节点地址保存在线程信息块的开始处 (FS:[0])。RtlDispatchException 从表头开始遍历这个链表，如果前面的异常处理器不处理这个异常，那么它会找下一个。值得特别说明的是，在这个链表的尾部总是保存着系统注册的一个默认的异常处理器，

这个异常处理器的过滤函数是 Kernel32.DLL 中的 UnhandledExceptionFilter 函数。如果前面的异常处理器都没有处理异常，那么 RtlDispatchException 最后便会找到这个默认的异常处理器并执行 UnhandledExceptionFilter 函数。

UnhandledExceptionFilter 函数的细节我们将在第 12 章讨论，目前我们可以简单地这样认为，如果当前程序没有正在被调试，那么该函数会将当前异常当作未处理异常来处置，然后启动系统对未处理异常的处置措施，包括弹出“应用程序错误”对话框后终止这个进程。也就是在没有调试的情况下，用户态异常不会经历第二轮分发过程。如果当前进程在被调试，那么 UnhandledExceptionFilter 会返回 EXCEPTION_CONTINUE_SEARCH，这会导致 RtlDispatchException 返回 FALSE。

如果 RtlDispatchException 返回 FALSE，也就是没有找到哪个异常处理块愿意处理该异常，而且当前进程在被调试，那么 KiUserExceptionDispatcher 会调用 ZwRaiseException 并将 FirstChance 参数设为 FALSE，发起对这个异常的第二轮分发。ZwRaiseException 会通过内核服务 NtRaiseException 把该异常传递给 KiDispatchException 来进行分发。此时调用 KiDispatchException 时，其最后一个参数 FirstChance 为假，所以 KiDispatchException 会按照第二轮机会分发该异常。

从图 12-3 可以看到，KiDispatchException 会先将第二轮处理机会送给调试子系统的 DbgkForwardException 函数。如果该函数返回 TRUE，那么分发结束；如果返回 FALSE，也就是该进程不在被调试，或者调试器没有处理该异常，那么 KiDispatchException 会尝试将异常分发给该进程的 ExceptionPort 字段指定的端口，通常环境子系统会在创建进程时将该字段设置为子系统监听的一个 LPC 端口（名为 ApiPort）。也就是说，对于第二轮分发，KiDispatchException 会把异常分发到该进程的异常端口（ExceptionPort），给那里的监听者一次处理异常的机会。向 ExceptionPort 发送异常使用的也是 DbgkForwardException 函数，只是把该函数的第二个参数 DebugException 设为 FALSE，此时 DbgkForwardException 会检测该进程的 ExceptionPort 字段，而且随后的 DbgkpSendApiMessage 不会挂起当前进程，但是当前线程还是被堵塞的。

如果向 ExceptionPort 发送异常，DbgkForwardException 再次返回 FALSE，那么 KiDispatchException 会终止当前线程。

11.3.4 归纳

至此，我们介绍了 Windows 操作系统的异常分发机制和有关的内核函数与 API，总结如下。

内核服务 NtRaiseException 是产生软件异常的主要方法，用户态代码可以通过 RaiseException API 来调用此内核服务。NtRaiseException 内部会调用 KiRaiseException，

KiRaiseException 再调用内核函数 KiDispatchException 进行异常分发。

对于 CPU 级的异常，CPU 会通过 IDT 表寻找异常的处理函数入口，也就是 KiTrapXX 例程，对于需要按异常流程分发的异常，KiTrapXX 例程会调用 CommonDispatchException 准备必要的参数，然后调用 KiDispatchException 进行异常分发。CPU 级的异常可能是由于执行用户态代码导致的，也可能是在执行内核态代码时导致的，但是以上处理逻辑是一致的，只不过是通过参数来区分出异常是发生在内核态的还是用户态的。对于内核代码触发的除零异常，系统会有特殊的处理，不会使用上面介绍的分发流程。

无论是来自用户态的异常，还是内核态的异常，（如果需要分发）系统都会使用 KiDispatchException 函数来分发异常。对于每个异常系统会最多给它两轮被处理机会，对于每轮机会，KiDispatchException 又都是尝试最先让调试器来处理，如果调试器没有处理，那么 KiDispatchException 会寻找代码中的异常处理块来处理该异常。对于来自内核态的异常，KiDispatchException 会直接调用 RtlDispatchException 来寻找异常处理块；对于来自用户态的异常，KiDispatchException 会通过设置 TrapFrame 让 KiUserExceptionDispatcher 来寻找用户空间中的异常处理块。尽管 KiUserExceptionDispatcher 也是通过调用 RtlDispatchException 来具体枚举和执行异常处理块的，但是这个 RtlDispatchException 是位于 NTDLL 中的函数，是位于用户态的。用户态的 RtlDispatchException 和内核态 RtlDispatchException（位于 NTOSKRNL 中）要实现的功能和执行的逻辑是基本一致的，不过由于“服务”对象不同，所以它们分别存在于两个模块中。我们讨论的时候也会把它们区别开来。

尽管 KiDispatchException 分发内核态异常和用户态异常的流程有所不同，但也有类似之处，比如都会试图先交给调试器来处理，而且会最多给每个异常两轮处理机会。

11.4 结构化异常处理（SEH）

常言道，天有不测风云，再周密的计划在执行时都可能遇到没有估计到的意外情况。同样，再严谨的代码在执行时也可能遇到没有考虑到的情况而出错。因此，好的计划和好的代码都应该考虑针对意外情况的应对措施，也就是要准备好处理异常的设施。为了增强操作系统和应用程序的健壮性，Windows 将异常处理融入到操作系统的总体设计之中，使异常处理机制成为操作系统的一个不可分割的部分。从前两节中关于异常的描述和分发流程的介绍中，大家可能已经意识到了这一点。

11.4.1 SEH 简介

为了让系统和应用程序代码都可以简单方便地支持异常处理，Windows 定义了一套标准的机制来规范异常处理代码的设计（对程序员）和编译（对编译器），这套机制被称为结构化异常处理（Structured Exception Handling），简称为 SEH。

从系统的角度来看（广义），SEH 是对 Windows 操作系统中的异常分发和处理机制的总称，其实现遍布在 Windows 系统的很多模块和数据结构中。比如，KiDispatchException 函数和 NtRaiseException 函数是位于内核模块中的，KiUserExceptionDispatcher 是位于 NTDLL.DLL 中的，异常注册链表的表头是登记在每个线程的线程信息块（TIB）中的。

从编程的角度来看（狭义），SEH 是一套规范，利用这套规范，程序员可以编写处理代码来复用系统的异常处理设施。可以将其理解为是操作系统的异常机制的对外接口，也就是如何在 Windows 程序中使用 Windows 系统的异常处理机制。

本节我们将着重从编程的角度来讨论结构化异常处理机制的原理，介绍它是如何与上一节的异常分发机制联系起来的。

从使用角度来看，结构化异常处理为程序员提供了终结处理（Termination Handling）和异常处理（Exception Handling）两种功能。终结处理用于保证终结处理块始终可以得到执行，无论被保护的代码块如何结束。异常处理用于接收和处理被保护块中的代码所发生的异常。下面通过实例分别加以介绍。

11.4.2 SEH 的终结处理

终结处理的语法结构如下（以微软的 Visual C++ 编译器为例，下同）：

```
__try
{
    // 被保护体 (guarded body), 也就是要保护的代码块。
}
__finally
{
    // 终结处理块
}
```

其中 __try 和 __finally 是 Visual C++ 编译器为支持 SEH 专门定义的关键字。按照惯例，没有下画线的关键字通常是编程语言所定义的，加双下画线的关键字是编译器定义的。需要指出的是，不同编译器支持 SEH 机制的关键字可能不同，如不特别说明，我们都是以 Visual C++ (VC) 编译器为例的。

显而易见，终结处理由两个部分组成，使用 __try 关键字定义的被保护体和使用 __finally 关键字定义的终结处理块。终结处理的目标是只要被保护体被执行，那么终结处理块就也会被执行，除非被保护体中的代码终止了当前线程（比如使用

ExitThread 或 ExitProcess 退出线程或整个进程)。因为终结处理块的这种特征, 终结处理非常适合做状态恢复或资源释放等工作。比如释放被保护块中获得的信号量以防止被保护块内发生意外时因没有释放这个信号量而导致线程死锁。

根据被保护块的执行路线, SEH 把被保护块的退出(执行完毕)分为正常结束(normal termination)和非正常结束(abnormal termination)两种。如果被保护块得到自然执行并顺序进入到终结处理块, 就认为被保护块是正常结束的。如果被保护块是因为发生异常或由于 return、goto、break 或 continue 等流程控制语句离开被保护块的, 就认为被保护块是非正常结束的。在终结处理块中可以调用 AbnormalTermination() 函数来知道被保护块的退出方式。

```
BOOL AbnormalTermination(void);
```

如果被保护块正常结束, 那么 AbnormalTermination() 函数返回 FALSE, 否则返回 TRUE。只能在终结块中调用 AbnormalTermination() 函数, 否则编译器会提示如下编译错误:

```
error C2707: '_abnormal_termination' : bad context for intrinsic function
```

除了上面出现的__try 和 __finally 关键字, 终结处理还有一个关键字 __leave。__leave 关键字的作用是立即离开(停止执行)被保护块, 或者理解为立即跳转到被保护块的末尾(__try 块的右大括号)。__leave 关键字只能出现在被保护体中。使用 __leave 关键字的退出属于正常退出。

下面通过清单 11-5 所列出的 SEH_Trmt 程序来加深对终结处理的理解。

清单 11-5 终结处理的例子

```

1 // SEH_Trmt.cpp : Termination Handling of SEH Demonstration
2 // Raymond April 16th, 2006
3 //
4
5 #include "stdafx.h"
6 #include <stdlib.h>
7 #include <excpt.h>
8
9 void PrintHelp()
10 {
11     printf("seh_trmt <number>\n");
12 }
13
14 int main(int argc, char* argv[])
15 {
16     int nNum=0, nRet=0;
17     const char* Week_Days[]={ "Sunday", "Monday", "Tuesday",
18         "Wednesday", "Thursday", "Friday", "Saturday" };
19
20     printf("Termination Handling of SEH Demonstration!\n");
21     if(argc<2)
22     {
23         PrintHelp(); return -1;
24     }
25     __try
26     {

```

```

27     printf("You entered: %s.\n", argv[1]);
28     nNum=atoi(argv[1]);
29
30     printf("It's %d in number.\n", nNum);
31     if(nNum<=0)
32     {   nRet=-3; __leave; }
33
34     //a test for goto in guarded block
35     if(nNum==6666)
36         goto EXIT_BYE;
37
38     printf("It's %s in a week.\n", Week_Days[nNum-1]);
39
40     //a test for return in guarded block
41     if(nNum==6)
42         return -2;
43
44     __finally
45     {
46         printf("Termination/Cleanup block is executed with %d.\n",
47             AbnormalTermination());
48     }
49
50 EXIT_BYE:
51     printf("Exit, Bye!\n");
52     return nRet;
53 }
```

以上代码中，第 26~43 行是被保护块，第 45~48 行是终结处理块。SEH_Trmt 是个命令行程序，它接受一个整数参数。下面便以不同的参数来执行 SEH_Trmt，并分析其结果。

首先，输入 seh_trmt 2 得到的结果是（//后是说明）：

```

Termination Handling of SEH Demonstration!
You entered: 2. It's 2 in number.
It's Monday in a week.
Termination/Cleanup block is executed with 0. //被保护块正常结束
Exit, Bye!
```

然后，输入 seh_trmt 6 得到的结果是：

```

Termination Handling of SEH Demonstration!
You entered: 6. It's 6 in number.
It's Friday in a week.
Termination/Cleanup block is executed with 1. //被保护块非正常结束
```

这次第 41 行的条件被满足，被保护块中的 return 语句被执行，而且这条 return 语句意味着退出程序。这时该如何保证在退出前终结处理块仍得到执行呢？答案是，为了做到这一点，编译器加入了额外的代码。在编译时，如果编译器扫描到终结处理对应的被保护块中包含有 return 语句，那么它便会定义一个局部变量用来保存 return 的值，并在目标代码中插入指令调用名为__local_unwind2 的局部展开函数。从第 41 和 42 行对应的汇编语句可以清楚地看到这些处理。

```

41:           if(nNum==6)
0040116E 83 7D E4 06      cmp      dword ptr [ebp-1Ch],6
00401172 75 1A            jne      main+11Eh (0040118e)
00401174 6A FF            push    0FFh
```

```

00401176 C7 45 C4 FE FF FF FF mov      dword ptr [ebp-3Ch],0FFFFFFFEh
// 该行将-2 放入专门用来存储返回值的局部变量[ebp-3Ch]。
42:           return -2;
0040117D 8D 55 F0          lea      edx,[ebp-10h]
00401180 52              push     edx
00401181 E8 68 04 00 00    call    __local_unwind2 (004015ee)
00401186 83 C4 08          add     esp,8
00401189 8B 45 C4          mov     eax,dword ptr [ebp-3Ch]
// 将局部变量的值放入 EAX，EAX 寄存器用来保存函数的返回值。
0040118C EB 31          jmp     EXIT_BYE+0Fh (004011bf)
// 跳到第 53 行退出。这种情况下，第 51 和 52 行不会得到执行。

```

单步跟踪`__local_unwind2` 函数可以知道，其中调用了终结处理块，为了能够被作为函数调用和返回，编译器将`finally` 块末尾插入了`ret` 指令。

为了保持一致，即使正常进入`finally` 块，使用的也是`call` 指令，这从清单 11-6 所示的第 43~48 行代码所对应的汇编代码中可以看到。

清单 11-6 终结处理的汇编代码

```

43:   }
0040118E C7 45 FC FF FF FF FF mov      dword ptr [ebp-4],0FFFFFFFh
00401195 E8 02 00 00 00    call    $L1062 (0040119c) // 调用终结块
0040119A EB 14          jmp     EXIT_BYE (004011b0)
44:   __finally
45:   {
46:       printf("Termination/Cleanup block is executed with %d.\n",
47:              AbnormalTermination());
0040119C E8 B5 04 00 00    call    __abnormal_termination (00401656)
004011A1 50              push     eax
004011A2 68 44 20 42 00    push offset string "Termination/Cleanup ..." (00422044)
004011A7 E8 F4 00 00 00    call    printf (004012a0)
004011AC 83 C4 08          add     esp,8
$L1063:
004011AF C3              ret      // 从终结块返回
48:   }

```

通过以上分析我们知道，在保护块中使用`return` 语句会使程序调用`__local_unwind2` 函数做所谓的局部展开，导致额外的开销。其实，`goto` 语句也有同样的效果（输入 `seh_trmt 6666` 可以跟踪该种情况），所以在编程时应该尽可能避免在被保护块中使用`goto` 和`return` 语句。因为保护块正常结束时，不需要执行局部展开，所以对于第 36 行和第 42 行可以先设置合适的变量，然后使用`_leave` 关键字退出被保护块，也就是采用第 32 行的做法。

还有一点要说明的是，如果在终结处理块中也使用了`return` 语句，比如在第 47 行加一行“`return -5;`”，那么，当执行`SEH_trmt 6` 时退出值会是多少呢？答案是-5。原因是该`return` 语句会被编译成一个赋值语句和一个无条件跳转语句，直接跳转到终结块后面的代码（`EXIT_BYE`），不会返回到`__local_unwind2` 了。

11.4.3 SEH 的异常处理

异常处理的语法结构如下：

```

__try
{
// 被保护体 (guarded body), 也就是要保护的代码块。
}
__except(过滤表达式)
{
// 异常处理块 (exception-handling block)
}

```

为了便于讨论，我们把过滤表达式和它下面的异常处理块合称为__except 块。除了__try 和__except 关键字，VC 编译器还提供了以下两个宏来辅助编写异常处理代码。

- `DWORD GetExceptionCode()`: 返回异常代码。只能在过滤表达式或异常处理块中使用这个宏。
- `LPEXCEPTION_POINTERS GetExceptionInformation()`: 返回一个指向 `EXCEPTION_POINTERS` 结构的指针，该结构包含了指向 `CONTEXT` 结构和异常记录 (`exception record`) 结构的指针。只能在过滤表达式中使用这个宏。

其中，`EXCEPTION_POINTERS` 结构的定义如下。

```

typedef struct _EXCEPTION_POINTERS {
    PEXCEPTION_RECORD ExceptionRecord;           // 异常记录
    PCONTEXT ContextRecord;                      // 异常发生时的线程上下文
} EXCEPTION_POINTERS, *PEXCEPTION_POINTERS;

```

`ExceptionRecord` 指向的就是我们在第 11.2 节所介绍的异常结构，通过它可以得到异常的详细信息，通过 `ContextRecord` 指针可以得到发生异常时的线程上下文，包括寄存器取值等。

11.4.4 过滤表达式

过滤表达式既可以是常量、函数调用，也可以是条件表达式或其他表达式，只要表达式的结果为 0, 1, -1 这三个值之一，它们的含义如下。

`EXCEPTION_CONTINUE_SEARCH(0)`: 本保护块不处理该异常，让系统继续寻找其他异常保护块。

`EXCEPTION_CONTINUE_EXECUTION(1)`: 已经处理异常，让程序回到异常发生点继续执行，如果导致异常的情况没被消除，那么很可能还会发生异常。

`EXCEPTION_EXECUTE_HANDLER(-1)`: 这是本保护块预计到的异常，让系统执行本块中的异常处理代码，执行完后会继续执行本异常处理块下面的代码，即 `except` 块之后的第一条指令。

下面介绍几种常用的编写过滤表达式的方法。

1. 直接使用常量，比如`__except(EXCEPTION_EXECUTE_HANDLER)`，或者直接写为`__except(-1)`等。

2. 使用条件运算符，比如 __except (GetExceptionCode() == EXCEPTION_ACCESS_VIOLATION ? EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH)，其含义是，如果发生的异常是非法访问异常，那么就执行异常处理块，否则就继续搜索其他异常保护块。
3. 使用逗号表达式实现一系列操作，比如在 2.2 节我们给出的例子中，便在过滤表达式中执行了打印变量、判断等多个操作，__except (printf("In __except block: "), VAR_WATCH(), ...)。
4. 调用其他函数，通常将 GetExceptionCode() 得到的异常代码或 GetExceptionInformation() 得到异常信息作为参数传给该函数。例如 __except (ExcptFilter (GetExceptionInformation()))。

可见，可以根据具体需要设计不同复杂度的表达式，短可到一个常数，长可到使用逗号运算符或编写过滤函数进行一系列操作。

下面给出一个通过过滤函数修正错误情况后再恢复执行的例子（参见清单 11-7）。

清单 11-7 SEH 的异常处理功能演示

```

1 // SEH_Excp.cpp : Exception Handling of SEH Demonstration
2 // Raymond April 16th, 2006
3 //
4 #include "stdafx.h"
5 #include <excpt.h>
6 #include <windows.h>
7 #include <stdlib.h>
8
9 char g_szDefPara[]="0123456789";
10 int ExcptionFilter(LPEXCEPTION_POINTERS pException,char***ppPara)
11 {
12     PEXCEPTION_RECORD pER=pException->ExceptionRecord;
13     PCONTEXT pContext=pException->ContextRecord;
14
15     printf("Exception Info: code=%08X, addr=%08X, flags=%X.\n",
16            pER->ExceptionCode, pER->ExceptionAddress,
17            pER->ExceptionFlags);
18
19     printf("Context Info: EIP=%08X, ECX=%08X.\n",
20            pContext->Eip, pContext->Ecx);
21
22     if(*ppPara==NULL && pER->ExceptionCode==STATUS_ACCESS_VIOLATION)
23     {
24         *ppPara=g_szDefPara;
25         pContext->Eip-=3;
26         printf("New EIP=%08X and *ppPara=%s.\n",
27                pContext->Eip,*ppPara);
28         return EXCEPTION_CONTINUE_EXECUTION;
29     }
30
31     return EXCEPTION_EXECUTE_HANDLER;
32 }
33 void FuncA(char* lpsz)
34 {
35     printf("Entering FuncA with lpsz=%s.\n",lpsz);
36     __try

```

```

37     {
38         *lpsz='2';
39     }
40     __except(ExcptionFilter(GetExceptionInformation(),&lpsz))
41     {
42         printf("Exexcuting handling block in FuncB.\n");
43     }
44     printf("Exiting from FuncA with lpsz=%s.\n",lpsz);
45 }
46 int FuncB(int nPara)
47 {
48     printf("Entering FuncB with Para=%d.\n",nPara);
49     __try
50     {
51         nPara=1/nPara;
52
53         *(int*)0=1;
54     }
55     __except (GetExceptionCode() == EXCEPTION_ACCESS_VIOLATION?
56             EXCEPTION_EXECUTE_HANDLER:EXCEPTION_CONTINUE_SEARCH)
57     {
58         printf("Executing handling block in FuncB [%X].\n",
59             GetExceptionCode());
60     }
61     printf("Exiting from FuncB with Para=%d.\n",nPara);
62     return nPara;
63 }
64 int main(int argc, char* argv[])
65 {
66     int nRet=0;
67
68     FuncA(argv[1]);
69
70     __try
71     {
72         nRet=FuncB(argc-1);
73     }
74     __except (EXCEPTION_EXECUTE_HANDLER)
75     {
76         printf("Executing exception handling block in main [%X].\n",
77             GetExceptionCode());
78     }
79
80     printf("Exit from main with nRet=%d.\n",nRet);
81     return nRet;
82 }

```

以上代码主要用来演示 SEH 的异常处理功能，FuncA 旨在说明过滤函数的用法，FuncB 用来说明异常处理块的处理逻辑（稍后讨论）。

首先尝试不带任何参数运行 SEH_Excp 程序，其结果如下：

```

Entering FuncA with lpsz=(null).
Exception Info: code=C0000005, addr=00401180, flags=0.
Context Info: EIP=00401180, ECX=00000000.
New EIP=0040117D and *ppPara=0123456789.
Exiting from FuncA with lpsz=2123456789.
Entering FuncB with Para=0.
Executing exception handling block in main [C0000094].
Exit from main with nRet=0.

```

下面让我们来一起分析以上的执行结果。因为我们没有使用任何命令行参数，所

以 argv[1] 为空，进入 FuncA 时 lpsz=NULL。第 38 行企图为 lpsz 指向的字符串的第一字节赋值为字符'2'，因为此时 lpsz 没有指向有效的内存，所以当 CPU 执行该语句时会导致页错误异常（参见 3.3 节）。页错误异常的内核处理例程是 KiTrap0E，这是一个非常重要而且繁忙的内核例程，因为它也是虚拟内存机制的重要部分，当一个不在物理内存中的内存地址被引用时，CPU 会产生页错误异常，然后执行 KiTrap0E。KiTrap0E 会从 CR2 寄存器中取出 CPU 是在访问哪个内存地址时发生了异常，并根据堆栈中的错误代码取出发生模式、访问内存的方式（读、写或取指）等信息。然后 KiTrap0E 会调用 MmAccessFault 函数检查异常的发生原因。当 MmAccessFault 发现导致异常的内存地址指向的是不允许访问的内存空间时，MmAccessFault 便会返回 STATUS_ACCESS_VIOLATION（值为 C0000005），即非法访问异常。接下来此异常会被送到 KiExceptionDispatch 函数进行分发，由于这是一个发生在用户态的异常，所以 KiExceptionDispatch 会把它交给 KiUserExceptionDispatch 函数进行分发。而后，KiUserExceptionDispatch 会调用用户态的 RtlDispatchException 寻找异常处理器，其寻找细节我们将在第 24 章详细介绍，目前大家可以认为是在栈中寻找异常处理器所注册的 EXCEPTION_REGISTRATION 结构，执行它的过滤函数，然后判断过滤函数的返回值，并据此决定是继续寻找其他异常处理器或是恢复执行程序，还是执行异常处理块中的代码。

回到我们的例子，因为第 40~44 行的异常处理器是与发生的异常距离最近的异常处理器，所以 RtlDispatchException 首先会找到这个异常处理器，评估其过滤表达式，也就是执行 ExceptionFilter 函数。ExceptionFilter 函数首先会打印出异常的信息（注意异常代码）和发生异常时的 CONTEXT（上下文）信息。这些信息都是内核中的函数在异常发生后收集并存储在 EXCEPTION_RECORD 和 CONTEXT 这两个结构中的。当要恢复执行发生异常的程序时，系统会使用 CONTEXT 结构来恢复当时的寄存器内容，以保证程序的状态不会有损失。

当 ExceptionFilter 函数检测到异常是非法访问异常，而且字符串指针为空时 (*ppPara==NULL)，它便意识到发生的异常是因为没有把字符串指针指向有效的内存而导致的，或者说是由于用户没有指定命令行参数导致的，因此它便把字符串指针指向到默认的参数字符串。按道理现在 ExceptionFilter 就可以让 CPU 回去重新执行发生异常的指令了，但是因为高级语言和汇编语言的差异，现在直接返回还会再次导致异常。看了第 38 行对应的汇编代码大家便会明白了。

```
38:          *lpsz='2';
0040117D 8B 4D 08          mov      ecx,dword ptr [ebp+8]
00401180 C6 01 32          mov      byte ptr [ecx],32h
```

首先，*lpsz='2'; 这条语句被编译成了两条汇编指令，第一条是把 lpsz 指针的内容（它指向的地址）放到 ECX 寄存器当中，第二条是把字符 '2' (ASCII 码 0x32) 放到 ECX 代表的地址中。根据执行结果打印出的信息（Context Info: EIP=00401180,

EAX=00000021.)，是第二条指令 (mov byte ptr [ecx],32h) 导致了异常。这很合乎情理，因为 lpsz 是个指针，所以取 lpsz 指针内容的第一条指令显然不会导致异常，但由于 lpsz 指向为空，所以 ecx 等于 0，那么向地址 0 写内容的第二条指令会导致异常。从前面的执行结果 (Context Info: EIP=00401180, ECX=00000000) 可以看到异常发生时的 EIP 指针为 00401180，确实是第二条指令导致了异常。那么回过头来，如果 ExceptionFilter 里修正了字符串指针的值，就返回 EXCEPTION_CONTINUE_EXECUTION 让程序恢复执行，那么 CPU 会回到第二条指令处执行，由于 ECX 的值仍然为 0，所以还是会导异常。也就是说，因为高级语言和汇编语言的差异，尽管我们改变了 lpsz 指针的值，但是因为第一条汇编指令没有被重新执行，所以我们的修正没有起到作用。那么如何让它起到作用呢？一种简单的方法就是将 CONTEXT 结构中的程序指针减小，使其指向第一条汇编指令。

看了上面修改程序指针的方法，大家也许会问，是不是总要观察反汇编的结果才能知道如何调整 EIP 的值呢？这样做有通用性么？这些问题很好，可以说我们的例子只是为了演示恢复执行的原理给大家看，对于实际的问题，需要设计更系统周密的方法。不过类似这样的问题，确实比较难以找到尽善尽美的方案，因此 VC 编译中把所有 C++ 异常都强制规定为不可恢复执行 (non-continuable)，也就是说，对于这样的异常，异常过滤函数不准返回 EXCEPTION_CONTINUE_EXECUTION。如果强行返回，那么会导致 EXCEPTION_NONCONTINUABLE_EXCEPTION 异常。

11.4.5 异常处理块

在现实的软件工程中，刚才介绍的先修正错误情况，然后恢复执行的办法被应用的并不多。主要原因是多方面的，包括错误情况难以判断，错误难以修正，程序员这方面的知识有限等。因此更多时候，过滤表达式都是直接或间接指定 EXCEPTION_EXECUTE_HANDLER，也就是执行异常处理块，对异常来做“善后”工作。异常处理块处理完成后，程序便沿着异常处理块下面的流程继续执行了，也就是不会再返回到发生异常的位置了。

仍然以清单 11-7 所示的 SEH_Excp 程序为例，我们来分析函数 B。函数 B 的被保护块 (第 49~54 行) 中共有两行代码，是我们特别设计的，分别用来产生除零异常 (第 51 行) 和非法访问异常 (第 53 行)。第 53 行直接向空指针赋值，所以只要执行到这里就会产生异常。第 51 行用参数 (nPara) 做除数，所以只有当 nPara 等于 0 时才会产生异常。当我们不带参数执行 SEH_Excp 时，argc 等于 1，因此第 72 行对 FuncB (argc-1) 的调用相当于用 0 做参数调用 FuncB。这会导致 CPU 执行到第 51 行对应的汇编指令时产生除零异常 (#DE)，接下来的过程大家应该已经非常熟悉：CPU 从 IDT 表中根据向量 0 处的门描述符找到 KiTrap00 的函数地址，然后开始执行 KiTrap00，KiTrap00 调用 CommonExceptionDispatch 建立异常记录结构，然后调用 KiExceptionDispatch 分

发异常。由于异常发生在用户态，所以 KiExceptionDispatch 会把它交给用户态的 KiUserExceptionDispatch 函数进行分发，KiUserExceptionDispatch 会通过调用 RtlDispatchException 来寻找异常处理器，RtlDispatchException 在栈中依次寻找各个异常处理器的 EXCEPTION_REGISTRATION 结构，评估各个异常处理器的过滤表达式，并根据过滤表达式的返回值决定是要继续寻找其他异常处理器或是恢复执行程序，还是执行该异常处理器的异常处理块。因为 RtlDispatchException 是按由近到远的顺序来评估各个异常处理块的（详见第 24 章）。因此第 55 行和第 56 行定义的过滤表达式会最先得到评估（执行）。

```
_except (GetExceptionCode() == EXCEPTION_ACCESS_VIOLATION?
    EXCEPTION_EXECUTE_HANDLER:EXCEPTION_CONTINUE_SEARCH)
```

显而易见，该过滤表达式的含义是：如果所发生异常的异常代码为 EXCEPTION_ACCESS_VIOLATION，那么就执行异常处理块，否则就继续搜索。因为我们现在讨论的是第 51 行引发的除零异常（异常代码 EXCEPTION_INT_DIVIDE_BY_ZERO），所以过滤表达式会返回 EXCEPTION_CONTINUE_SEARCH 给 RtlDispatchException，意思是“我不处理该异常，你继续找其他人吧”。RtlDispatchException 收到此回复后会在栈中继续追溯，寻找其他异常处理器，因为 main 函数在调用 FuncB 时也使用了 SEH，因此 RtlDispatchException 接下来会评估第 74 行所定义的过滤表达式。

```
_except (EXCEPTION_EXECUTE_HANDLER)
```

由于该过滤表达式是常量 EXCEPTION_EXECUTE_HANDLER，意思是“不论发生什么异常我都处理，请执行我的处理块！”。RtlDispatchException 接到此回复后，首先会进行全局展开（global unwinding）和局部展开（local unwinding）（通称栈展开，我们将在第 24 章讨论），然后跳转到异常处理块的起始地址，开始执行异常处理块。观察异常处理块（第 74~78 行）所对应的汇编代码（清单 11-8），可以看到没有任何跳转和返回指令（与前面讲的 finally 块不同），因此 CPU 执行完异常处理块后，便会继续执行异常处理块下面的代码。

清单 11-8 __except 块的编译

```
74:    {
75:        printf("Executing exception handling block in main [%X].\n",
76:               GetExceptionCode());
0040DAB1 8B 4D E0          mov      ecx,dword ptr [ebp-20h]
0040DAB4 51                push     ecx
0040DAB5 68 DC 31 42 00    push offset string "Executing exception ..." (004231dc)
0040DABA E8 61 39 FF FF    call    printf (00401420)
0040DABF 83 C4 08          add     esp,8
78:    }
0040DAC2 C7 45 FC FF FF FF mov      dword ptr [ebp-4],0FFFFFFFh
79:
80:    printf("Exit from main with nRet=%d.\n",nRet);
```

回味上面的执行过程，由于 FuncB 中的第 51 行在执行时发生了异常，系统（异常分发逻辑）在 main 函数中找到了异常处理器，执行完 main 中的异常处理块后，系

统便继续执行 main 中的代码了。从函数调用关系看,由于发生异常,CPU 执行完 FuncB 函数的第 51 行后便继续执行 main 函数了,仿佛是从 FuncB 函数中间“飞”了出来,这也是在执行 main 函数的异常处理块之前要进行栈展开的原因,简单来说,栈展开就是将栈恢复成适合异常处理块执行的栈状态。在我们现在的例子中,就是将栈恢复成 main 函数所对应的栈。

从执行流程角度来看,由于发生异常,程序从 FuncB 函数中间“飞”回到了 main 函数,相当于在 FuncB 函数中多了一个额外的“函数出口”,而且该出口的位置是不固定的。这种不可预测的“出口”是违背结构化编程理念的,会使软件的执行流程变得更加复杂,也会给软件调试(错误定位)带来困难。为了降低异常的负面影响,应该及早捕捉和处理异常,也就是某一段代码可能发生异常,那么就在这段代码附近设置能够处理该异常的异常处理器。

11.4.6 嵌套使用终结处理和异常处理

一个异常保护块(`_try`块)不能同时配有终结处理块(`_finally`块)和异常处理块(`_except`块),但是可以通过嵌套使一段代码同时得到终结处理和异常处理(见清单 11-9)。

清单 11-9 嵌套使用终结处理和异常处理

```

1 // SEH_Mix.cpp : Demonstrate nested termination handling
2 // and exception handling of SEH.
3 // Raymond April 22th, 2006
4 //
5
6 #include "stdafx.h"
7 #include <excpt.h>
8
9 void main(void)
10 {
11     __try
12     {
13         __try
14         {
15             int n=0;
16             int i=1/n;
17         }
18         __finally
19         {
20             printf("Executing terminating block.\n");
21         }
22     }
23     __except (printf("Exceuting ExcpFilter.\n"),
24             EXCEPTION_EXECUTE_HANDLER)
25     {
26         printf("Executing exception handling block.\n");
27     }
28 }
```

清单 11-9 所示的代码演示了如何使用终结处理和异常处理。第 13~17 行定义了内

层的被保护块，第 18~21 行定义了一个终结处理块，第 11~22 行定义了外层的被保护块，第 23~27 行定义了外层的异常处理器。第 15~16 行故意作了个除零操作，CPU 执行到此一定会产生一个除零异常 (#DE)。接下来的执行逻辑会怎么样呢？会先执行内层的 `__finally` 块还是外层的 `__except` 块呢？答案是先执行 `__except` 块的过滤表达式，该逗号表达式的结果是 `EXCEPTION_EXECUTE_HANDLER`，在异常分发函数收到此结果后，会进行全局展开和局部展开，局部展开的过程中会调用终结处理块。因此以上代码的执行结果是：

```
Executing ExcpFilter.  
Executing terminating block.  
Executing exception handling block.
```

以上对 Windows 的结构化异常处理 (SEH) 机制进行了初步介绍。因为异常处理机制这一内容确实比较复杂，有些部分也不太容易理解，所以读完该节后大家心中可能还有一些疑问，比如 `RtlDispatchException` 内部是如何工作的？它是如何找到各个异常处理器的？因为理解这些问题需要对栈结构有比较深的了解，所以我们把这一内容留到第 24 章来讨论。

11.5 向量化异常处理 (VEH)

除了结构化异常处理 (SEH)，从 Windows XP 开始，Windows 还支持一种名为向量化异常处理 (Vectored Exception Handling) 的异常处理机制，简称为 VEH。与 SEH 既可以用在用户态又可以用在内核态不同，VEH 只能用在用户态程序中。

11.5.1 登记和注销

VEH 的基本思想是通过注册以下原型的回调函数来接收和处理异常：

```
LONG CALLBACK VectoredHandler(PEXCEPTION_POINTERS ExceptionInfo);
```

其中 `ExceptionInfo` 是指向 `EXCEPTION_POINTERS` 结构的指针，与 `GetExceptionInformation()` 函数的返回值是相同类型的。`VectoredHandler` 的返回值应该为 `EXCEPTION_CONTINUE_EXECUTION(-1, 恢复执行)` 或者 `EXCEPTION_CONTINUE_SEARCH(0, 继续搜索)`。

相应的，Windows 公布了两个 API，`AddVectoredExceptionHandler` 和 `RemoveVectoredExceptionHandler` 分别用来注册和注销回调函数 (`VectoredHandler`)。

```
PVOID AddVectoredExceptionHandler(ULONG FirstHandler,  
    PVECTORED_EXCEPTION_HANDLER VectoredHandler);
```

参数 `FirstHandler` 用来指定该回调函数的被调用顺序，为 0 表示希望最后被调用，为 1 表示希望最先被调用。如果注册了多个回调函数，而且 `FirstHandler` 都为非零值，那么最后注册的会最先被调用。如果注册成功，返回值指向的是系统为该异

常处理器分配的一个结构 (VEH_REGISTRATION) 指针, 应用程序应该保存这个指针, 以便注销该 VEH 时使用; 如果注册失败, 返回值为 0。

在 AddVectoredExceptionHandler 内部, 它会为每个 VEH 处理器分配一个类似如下结构的结构体 (长为 12 字节):

```
typedef struct _VEH_REGISTRATION
{
    _VEH_REGISTRATION* next;
    _VEH_REGISTRATION* prev;
    PVECTORED_EXCEPTION_HANDLER pfnVeh;
} VEH_REGISTRATION, * PVEH_REGISTRATION;
```

其中 next 指针指向下一个 VEH, prev 指向前一个 VEH, pfnVeh 指向该 VEH 的回调函数。当有多个 VEH 时, 这些 VEH 的 VEH_REGISTRATION 结构组成一个环状链表。NTDLL 中的全局变量 RtlpCalloutEntryList 指向该链表的头。

RemoveVectoredExceptionHandler 函数用来注销 VEH 处理器, 也就是将一个 VEH 的注册结构从 RtlpCalloutEntryList 所指向的链表中移除。

```
ULONG RemoveVectoredExceptionHandler( PVOID VectoredHandlerHandle);
```

11.5.2 调用 VEH

前面几节我们介绍过, 在用户态下发生的异常, KiUserExceptionDispatch 会调用 RtlDispatchException 来寻找异常处理器, 在支持 VEH 的系统中, 在寻找结构化异常处理器之前, RtlDispatchException 会先调用 RtlCallVectoredExceptionHandlers 给 VEH 优先的处理机会。RtlCallVectoredExceptionHandlers 会从前面讲的 RtlpCalloutEntryList 开始遍历 VEH 记录列表。

- 如果 RtlpCalloutEntryList 的 next 成员指向的是 RtlpCalloutEntryList 自身, 则说明没有注册的 VEH 需要调用, RtlCallVectoredExceptionHandlers 会返回 FALSE (0);
- 如果 RtlpCalloutEntryList 的 next 成员指向了另一个 VEH_REGISTRATION 结构, 则 RtlCallVectoredExceptionHandlers 会先调用 RtlEnterCriticalSection 函数防止其他线程访问链表, 然后调用该 VEH 的回调函数, 如果回调函数返回 EXCEPTION_CONTINUE_SEARCH(0), 那么 RtlCallVectoredExceptionHandlers 就继续循环下一个准备被调用的 VEH 处理器, 如果这是最后一个 VEH 处理器, 那么 RtlCallVectoredExceptionHandlers 便返回 FALSE (0);
- 如果一个 VEH 处理器返回了 EXCEPTION_CONTINUE_EXECUTION(-1), 那么 RtlCallVectoredExceptionHandlers 便返回 TRUE。

如果 RtlCallVectoredExceptionHandlers 返回 FALSE, 那么 RtlDispatchException 会继续寻找 SEH 处理器; 如果 RtlCallVectoredExceptionHandlers 返回 TRUE, 那么 RtlDispatchException 便认为 VEH 处理器已经处理了该异常, 便

直接返回了(返回值为真)。清单 11-10 给出了 RtlCallVectoredExceptionHandlers 函数的伪代码，供大家参考。

清单 11-10 RtlCallVectoredExceptionHandlers 函数的伪代码

```

bool RtlCallVectoredExceptionHandlers(
    PEXCEPTION_RECORD pExcptRec,
    CONTEXT * pContext )
{
    bool bRet = false;

    if(RtlpCalloutEntryList->Next==RtlpCalloutEntryList)
        return bRet;

    RtlEnterCriticalSection( &RtlpCalloutEntryLock );

    // 指向 VEH 链表的第一个节点
    PVEH_REGISTRATION pCurrentNode = RtlpCalloutEntryList->next;

    // 遍历注册的所有 VEH 节点
    while ( pCurrentNode != RtlpCalloutEntryList )
    {
        EXCEPTION_POINTERS pExceptionPointers =
            (EXCEPTION_POINTERS)&pExcptRec;
        LONG disposition = pCurrentNode->pfnVeh
            ( pExceptionPointers );

        if ( disposition == EXCEPTION_CONTINUE_EXECUTION )
        {
            bRet = true;
            break;
        }

        pCurrentNode = pCurrentNode->next;
    }

    RtlLeaveCriticalSection( &RtlpCalloutEntryLock );
}

return bRet;
}

```

11.5.3 示例

下面通过一个程序实例来加深大家对 VEH 的理解(参见清单 11-11)。

清单 11-11 演示向量化异常处理的实例

```

1 // VEH.cpp : Code to demonstrate Vectored Exception Handling.
2 // Raymond April 22th, 2006
3 //
4
5 #include <windows.h>
6 #include <stdio.h>
7 #include <stdlib.h>
8 #include <ctype.h>
9
10 typedef struct _VEH_REGISTRATION
11 {
12     _VEH_REGISTRATION* next;

```

```

13     _VEH_REGISTRATION* prev;
14     PVECTORED_EXCEPTION_HANDLER pfnVeh;
15 } VEH_REGISTRATION, * PVEH_REGISTRATION;
16
17 #define TRACE(szWhere,ExceptionInfo) \
18     printf("Exceuting %s: code=%X, flags=%X\n",szWhere,\n\
19     ExceptionInfo->ExceptionRecord->ExceptionCode,\n\
20     ExceptionInfo->ExceptionRecord->ExceptionFlags)
21
22 #define SHOW_VEH(v) printf("Node [%X]: next=%8X, prev=%8X, PFN=%X.\n",\n\
23 v, v->next, v->prev, v->pfnVeh)
24
25 LONG WINAPI VEH1( struct _EXCEPTION_POINTERS *ExceptionInfo)
26 {
27     TRACE("VEH1",ExceptionInfo);
28     return EXCEPTION_CONTINUE_SEARCH;
29 }
30
31 LONG WINAPI VEH2(struct _EXCEPTION_POINTERS *ExceptionInfo )
32 {
33     TRACE("VEH2",ExceptionInfo);
34     PCONTEXT Context = ExceptionInfo->ContextRecord;
35
36     Context->Eip++;
37     return EXCEPTION_CONTINUE_EXECUTION;
38 }
39 LONG WINAPI VEH3( struct _EXCEPTION_POINTERS *ExceptionInfo )
40 {
41     TRACE("VEH3",ExceptionInfo);
42     return EXCEPTION_CONTINUE_SEARCH;
43 }
44
45 void VehTest1()
46 {
47     PVOID h1,h2,h3;
48
49     h2 = AddVectoredExceptionHandler(1,VEH2);
50     h3 = AddVectoredExceptionHandler(0,VEH3);
51     h1 = AddVectoredExceptionHandler(1,VEH1);
52     SHOW_VEH(((PVEH_REGISTRATION)h1));
53     SHOW_VEH(((PVEH_REGISTRATION)h2));
54     SHOW_VEH(((PVEH_REGISTRATION)h3));
55     _asm {cli};
56     RemoveVectoredExceptionHandler(h1);
57     RemoveVectoredExceptionHandler(h2);
58     RemoveVectoredExceptionHandler(h3);
59 }
60
61 void VehTest2()
62 {
63     PVOID h1;
64
65     h1 = AddVectoredExceptionHandler(1,VEH1);
66
67     __try
68     {
69         _asm {int 1};
70     }
71     __except(TRACE("SEH Filter in VehTest2",
72         ((PEXCEPTION_POINTERS)GetExceptionInformation())),
73         EXCEPTION_EXECUTE_HANDLER)
74     {
75         printf("Exceuting SEH handling block in VehTest2.\n");

```

```

76      }
77
78     RemoveVectoredExceptionHandler(h1);
79 }
80
81 void main( )
82 {
83     printf("VehTest1:\n");
84     VehTest1();
85     printf("VehTest2:\n");
86     VehTest2();
87 }

```

在清单 11-11 所示的小程序中，第 25~43 行的代码定义了 3 个用于向量化异常处理的回调函数，或者说 VEH 处理器。第 45~79 行设计了两个函数用于测试以上 VEH 处理器。

让我们先看 VehTest1，首先 VehTest1 调用 AddVectoredExceptionHandler API 对 3 个 VEH 进行了注册，第 49 行的调用将 FirstHandler 参数设为 1，这告诉系统将 VEH2 注册为第一个 VEH。第 50 行的调用将 FirstHandler 参数设为 0，这告诉系统将 VEH3 注册为最后一个 VEH。第 51 行的调用将 FirstHandler 参数也设为 1，这告诉系统将 VEH1 注册为第一个 VEH。因为 VEH1 的注册在后，所以 VEH1 会取代 VEH2 成为第一个 VEH。因为 AddVectoredExceptionHandler 的返回值就是系统中用来记录该 VEH 的链表项指针，所以可以很容易地打印出（第 52~54 行）这 3 个 VEH 表项（VEH_REGISTRATION 结构）的各个成员值：

```

Node [1430E0]: next= 1430B0, prev=77FC4880, PFN=401014.      // VEH1 的节点
Node [1430B0]: next= 1430C8, prev= 1430E0, PFN=401005.      // VEH2 的节点
Node [1430C8]: next=77FC4880, prev= 1430B0, PFN=40100F.      // VEH3 的节点

```

显而易见，VEH1 节点（1430E0）的 next 值为 1430B0，即 VEH2 节点的地址；VEH2 节点（1430B0）的 next 值为 1430C8，这正是 VEH3 节点的地址；VEH1 节点的 prev 和 VEH3 的 next 指向的都是 77FC4880，这正是 NTDLL 中全局变量 RtlpCalloutEntryList 的值。

在第 55 行，我们故意插入了一条特权指令（清除允许中断标志），这会触发一个 EXCEPTION_PRIV_INSTRUCTION 异常（0xC0000096）。如前面所讨论的，系统会遍历 VEH 链表，执行 VEH 处理器的回调函数。最先被调用的是 VEH1，但是 VEH1 返回 EXCEPTION_CONTINUE_SEARCH，让系统继续搜索。于是 VEH2 得到了处理机会，VEH2 将 CONTEXT 结构中的指令指针（EIP）递增 1，因为 EIP 本来是指向导致异常的 CLI 指令的，而且 CLI 指令的长度只有一个字节，所以递增 1 相当于使程序指针跳过了导致异常的 CLI 指令。做了这个处理后，VEH2 返回 EXCEPTION_CONTINUE_EXECUTION，意思说“我已经处理好了这个异常，请恢复执行吧”。因为 VEH2 返回了“好消息”（EXCEPTION_CONTINUE_EXECUTION），所以系统停止继续搜索其他的 VEH 和 SEH 处理器，RtlDispatchException 返回 TRUE 给 KiUserExceptionDispatch，KiUserExceptionDispatch 调用 NtContinue 服务恢复程序继续执行。

第 56 行到第 58 行注销了 VEH 处理器, 注意注销的顺序可以与注册的顺序不一致。

下面再来看 VehTest2, VehTest2 先注册了一个 VEH 处理器 (VEH1), 然后使用结构化异常处理机制对第 69 行的“危险操作”进行了保护。当执行第 69 行发生异常后, 系统会找到并执行 VEH1, 但由于 VEH1 没有处理该异常, 所以系统会继续搜索 SEH 处理器。执行 VEH 小程序, 得到的执行结果与以上的分析是一致的:

```
VehTest1:  
Node [1430E0]: next= 1430B0, prev=77FC4880, PFN=401014.  
Node [1430B0]: next= 1430C8, prev= 1430E0, PFN=401005.  
Node [1430C8]: next=77FC4880, prev= 1430B0, PFN=40100F.  
Exceuting VEH1: code=C0000096, flags=0  
Exceuting VEH2: code=C0000096, flags=0  
VehTest2:  
Exceuting VEH1: code=C0000005, flags=0  
Exceuting SEH Filter in VehTest2: code=C0000005, flags=0  
Exceuting SEH handling block in VehTest2.
```

最后, 我们归纳 VEH 与 SEH 的区别与联系。

从应用范围来讲, SEH 既可以用在用户态(应用程序)代码中, 也可以用在内核态(比如驱动程序)代码中, 但是 VEH 只能用在用户态代码中。另外, VEH 只有在 Windows XP 或更高版本的 Windows 中才能使用, 在编译使用 VEH 的应用程序时必须将要求的 Windows 版本定义为 0x500 或更高的值(_WIN32_WINNT=0x0500), 而 SEH 没有这一限制。

从优先级的角度来看, 对于同时注册了 VEH 和 SEH 的代码所触发的异常, VEH 比 SEH 先得到处理权。

从登记方式来看, SEH 的注册信息是以固定的结构存储在线程栈中的, 不同层次的各个 SEH 的注册信息依次被压入到栈中, 分布在栈的不同位置上, 依靠结构体内的指针相联系, 因为人们经常将一个函数所对应的栈区域称为栈帧(Stack Frame), 所以 SEH 的异常处理器又经常被称为基于帧的异常处理器(frame-based exception handler); VEH 的注册信息是存储在进程的内存堆中的。

从作用域的角度来看, VEH 处理器相对于整个进程都有效, 具有全局性; SEH 处理器是动态建立在所在函数的栈帧上的, 会随着函数的返回而被注销, 因此 SEH 只对当前函数或这个函数所调用的子函数有效。

从编译的角度来讲, SEH 的登记和注销是依赖编译器编译时所生成的数据结构和代码的(详见第 24 章), VEH 的注册和注销都是通过调用系统的 API 显式完成的, 不需要编译器的特别处理。

11.6 本章总结

本章分两个部分比较详细地介绍了 Windows 操作系统的中断和异常管理，前半部分（前 3 节）介绍了用以描述异常的基本数据结构和分发异常的基本过程。后半部分介绍了结构化异常处理机制和向量化异常处理机制。下一章我们将继续介绍 Windows 系统对未处理异常的处置方法和详细过程。

参考文献

1. Jeffrey Richter. Programming Applications for Microsoft Windows. Microsoft Press, 1999

未处理异常和 JIT 调试

本章的前半部分将详细介绍 Windows 系统对“未处理异常”的处置方法和过程，包括默认的异常处理函数（第 12.2 节）、UnhandledExceptionFilter 函数（第 12.3 节）和应用程序错误对话框（第 12.4 节）。后半部分将介绍与未处理异常密切相关的 JIT 调试（第 12.5 节）、顶层过滤函数（第 12.6 节）、系统自带的 JIT 调试器——Dr. Watson 程序（第 12.7 节）、Dr.Watson 产生的日志文件（第 12.8 节）及用户态转储文件（第 12.9 节）。

12.1 简介

从上一章的介绍我们知道，Windows 系统定义了非常周密的逻辑来分发异常，会给每个异常最多两轮被处理的机会。对于第一轮处理机会，异常分发函数会先尝试分发给调试器，如果没有调试器或调试器没有处理，就会分发给异常处理器来处理。

当开发软件时，我们可以使用结构化异常处理（SEH）机制，在不同层次上定义多个异常处理器。当有异常需要处理时，系统会把处理机会依次交给各个异常处理器。每个处理器可以根据系统传给它的参数，来了解异常的详细信息，以此判断自己能否处理该异常，并把结果返回给系统。如果一个处理器返回 EXCEPTION_CONTINUE_EXECUTION，就表示它已经处理了该异常，并让系统恢复执行因为发生异常而中断的代码；如果一个处理器返回 EXCEPTION_CONTINUE_SEARCH，那么表示它不能处理该异常，并让系统继续寻找其他的异常处理器。那么，系统有没有可能“问”遍了所有的异常处理器，大家都说“处理不了”呢？

以清单 12-1 所示的 UEF（HelloWorld）小程序为例，我们在 main 函数中没有编写设计任何异常处理代码，既没有 SEH 处理器，也没有 VEH 处理器，但是却在第 10 行设置了一个“炸弹”。

清单 12-1 演示未处理异常的 UEF 小程序

```
1 // UEF.cpp : Demonstrate Unhandled Exceptions.  
2 // Raymond April 22th 2006  
3 //  
4
```

```

5  #include "stdafx.h"
6
7  int main(int argc, char* argv[])
8  {
9      printf("Going to assign value to null pointer!!\n");
10     *(int*)0=1;
11     return 0;
12 }

```

如果运行这个程序，当 CPU 执行到第 10 行时，一定会产生一个非法访问异常（CPU 级是页错误异常）。因为在我们的程序代码中，根本没有任何异常处理器来处理异常，所以我们编写的代码不会处理这个异常，像这样的异常被称为未处理异常（Unhandled Exception）。未处理异常的另一种情况是，尽管我们设计了一个或多个异常处理器，但是它们都没有处理某个异常。

相对于操作系统的代码而言，系统中其他软件的代码又被称为用户代码，因此，可以把未处理异常定义为在用户代码范围内发生或触发的但用户代码没有处理的异常。

根据用户的运行模式，发生在驱动程序等内核态模块中的未处理异常通常被称为内核态的未处理异常。类似的，发生在应用程序中的未处理异常被称为用户态的未处理异常。

认真回忆我们在上一章介绍的异常分发流程（参见上一章的图 11-3），系统只有在第一轮分发时，才会把异常分发给用户代码注册的异常处理器，第二轮分发时并不会这么做。因此，对 SEH 或 VEH 异常处理器来说，只有一轮处理异常的机会。也就是说，如果在第一轮异常分发的过程中，一个异常没有被处理，那么它便成为未处理异常。进入到第二轮分发的异常都属于未处理异常。

对于用户态的未处理异常，Windows 的策略是使用系统登记的默认异常处理器来处理。事实上，Windows 为应用程序的每个线程都设置了默认的 SEH 异常处理器。当应用程序内的代码没有处理异常时，系统会使用这些默认的异常处理器来处理异常。

对于内核态的未处理异常，如果有内核调试器存在，则系统（KiExceptionDispatch）会给调试器第二轮处理机会，如果调试器没有处理该异常，或者根本没有内核调试器，则系统便会调用 KeBugCheckEx 发起蓝屏机制，报告错误并停止整个系统，其停止码为 KMODE_EXCEPTION_NOT_HANDLED。Windows 这样做的理由是，它把内核态执行的代码看作是系统信任的代码，这些代码应该是经过缜密设计和认真测试过的，因此，一旦在信任代码中发生未处理异常，那么“一定”是发生了事先没有估计到的严重问题，蓝屏终止可让系统以可控的方式停止工作，防止其继续运行造成更大的损失。在第 13 章中我们将详细讨论蓝屏机制。

考虑内核态未处理异常的机制比较简单，因此本章将集中讨论用户态的未处理异常，下一节将介绍默认的异常处理器。

12.2 默认的异常处理器

对于典型的 Windows 应用程序，系统会为它的每个线程登记默认的结构化异常处理器（SEH）。此外，编译器在编译时插入的启动函数通常也会注册一个 SEH 处理器。对于使用 C 运行库的程序，C 运行库包含了基于信号的异常处理器机制。下面分别加以介绍。

12.2.1 BaseProcessStart 函数中的 SEH 处理器

为了更好地理解系统是如何设置默认的异常处理器的，我们先来看看 Windows 创建进程（启动一个程序）的大体过程，不论是通过双击程序文件、键入程序命令，还是在程序中调用 CreateProcess API 来启动一个程序，其内部过程都是类似的。*Windows Internals, 4th Edition*（《深入解析 Windows 操作系统，第 4 版》，电子工业出版社，2007 年）一书将创建进程的整个过程分为如下 6 个阶段。

1. 打开要执行的程序映像文件，创建 section 对象用于将文件映射到内存中。
2. 建立进程运行所需的各种数据结构（EPROCESS、KPROCESS 及 PEB）和地址空间。
3. 调用 NtCreateThread，创建处于挂起状态的初始线程，将用户态的起始地址存储在ETHREAD 结构中。
4. 通知 Windows 子系统注册新的进程。
5. 开始执行初始线程。
6. 在新进程的上下文（context）中对进程做最后的初始化工作。

以上第 3 点决定了初始线程的起始地址，也就是新线程开始在用户态正式运行时的起始地址。大家知道，Windows 程序的 PE 文件中登记了程序的入口地址，即 IMAGE_OPTIONAL_HEADER 结构的 AddressOfEntryPoint 字段（参见 25.4.2 节）。但是在创建 Windows 子系统进程时，系统通常并不把这个地址用作新线程的起始地址，而是把起始地址指向 kernel32.dll 中的进程启动函数 BaseProcessStart。这样做的一个原因，是要注册一个默认的 SEH 处理器。从清单 12-2 所示的 BaseProcessStart 函数的伪代码中可以清楚地看到这一点。

清单 12-2 BaseProcessStart 函数的伪代码

```

1  VOID BaseProcessStart(PPROCESS_START_ROUTINE lpfnEntryPoint)
2  {
3      __try
4      {
5          NtSetInformationThread(GetCurrentThread(),
6                               ThreadQuerySetWin32StartAddress,
7                               &lpfnEntryPoint, sizeof(lpfnEntryPoint) );
8          ExitThread((lpfnEntryPoint)());
9      }
10     __except(UnhandledExceptionFilter(GetExceptionInformation()))

```

```

11      {
12          ExitThread(GetExceptionCode());
13      }
14  }
15

```

在以上代码中，`lpfnEntryPoint` 是指向入口函数的指针，它的值就是从 PE 中读取到的入口地址。第 8 行中调用这个入口地址，也就是跳到入口函数，让应用程序代码开始运行。该函数返回后，便执行 `ExitThread` API。

从第 3、10 行的 `_try` 和 `_except` 关键字可以清楚地看出，在应用程序的入口函数被调用前，`BaseProcessStart` 会为其先设置一个结构化的异常处理器，它是初始化线程中最早注册的异常处理器。因为当分发异常时，系统（`RtlDispatchException`）是从最晚注册的异常处理器来查找的，所以这个最早注册的 SEH 处理器是最后得到处理机会的。这样便保证了只有当应用程序自己设计的代码没有处理异常时，这个默认的 SEH 处理器才会得到处理机会。

12.2.2 编译器插入的 SEH 处理器

`BaseProcessStart` 函数（清单 12-2）中的 `lpfnEntryPoint` 指向的是是否为应用程序的 `main` 函数或 `WinMain` 函数的地址呢？答案是否定的，至少对于大多数程序来说不是这样的。

因为在执行这些函数前，还有很多准备工作要做，比如初始化 C 运行库（runtime library）、初始化全局变量、初始化 C 运行库所使用的堆、准备命令行参数等。为了做这些准备工作，编译器通常会把自身提供的一个启动函数登记为程序的入口，让系统先执行这个启动函数，这个启动函数内部再调用用户编写的 `main` 或 `WinMain` 函数。

以 VC 编译器为例（下同），它总是将自己提供的以下函数之一登记为程序的入口。

- `mainCRTStartup`: 非 UNICODE 的控制台程序，内部会调用 `main` 函数。
- `wmainCRTStartup`: UNICODE 的控制台程序，内部会调用 `main` 函数。
- `WinMainCRTStartup`: 非 UNICODE 的 Win32 程序，内部会调用 `WinMain` 函数。
- `WinMainCRTStartup`: UNICODE 字符的 Win32 程序，内部会调用 `WinMain` 函数。

以上函数是共享同一套源代码的，其源程序文件名叫 `crt0.cpp`，位于 VC 编译器的 C 运行库源程序目录中（`crt\src`）。为了用一套代码定义 4 个函数，`crt0.cpp` 中使用了很多条件编译选项。清单 12-3 列出了简化后的代码。

清单 12-3 编译器插入的入口函数（部分）

```

1  #ifdef _WINMAIN_
2  #ifdef WPRFLAG
3  void wWinMainCRTStartup( // 用于宽字符 Win32 应用程序
4  #else /* WPRFLAG */
5  void WinMainCRTStartup( // 用于普通的 Win32 应用程序

```

```

6      #endif /* WPRFLAG */
7  #else /* _WINMAIN_ */
8      #ifdef WPRFLAG
9          void wmainCRTStartup( // 用于宽字符的 Win32 控制台程序
10         #else /* WPRFLAG */
11             void mainCRTStartup( // 用于普通的 Win32 控制台程序
12             #endif /* WPRFLAG */
13         #endif /* _WINMAIN_ */
14             void // 4 种形式的启动函数都没有参数，也都没有返回值（不需要）
15         )
16     {
17         int mainret;
18
19         // 取 Windows 的版本号，代码略
20         // 调用 _heap_init(1 或 0) 初始化堆，代码略
21         // 如果定义了 _MT，则做与多线程有关的初始化 (_mtinit())
22
23         __try {
24             // 初始化命令行参数，代码略
25
26             _cinit(); /* 执行 C 的数据初始化函数 */
27
28 #ifdef _WINMAIN_
29             StartupInfo.dwFlags = 0;
30             GetStartupInfo( &StartupInfo );
31             #ifdef WPRFLAG
32                 lpszCommandLine = _wwincmdln();
33                 mainret = wWinMain(
34             #else /* WPRFLAG */
35                 lpszCommandLine = _wincmdln();
36                 mainret = WinMain(
37             #endif /* WPRFLAG */
38                 GetModuleHandleA(NULL),
39                 NULL,
40                 lpszCommandLine,
41                 StartupInfo.dwFlags & STARTF_USESHOWWINDOW
42                     ? StartupInfo.wShowWindow
43                     : SW_SHOWDEFAULT
44             ); // 调用 WinMain 函数
45 #else /* _WINMAIN_ */
46             #ifdef WPRFLAG
47                 _winitenv = _wenviron;
48                 mainret = wmain(__argc, __argv, _wenviron);
49             #else /* WPRFLAG */
50                 _initenv = _environ;
51                 mainret = main(__argc, __argv, _environ); // 调用 main 函数
52             #endif /* WPRFLAG */
53
54             #endif /* _WINMAIN_ */
55             exit(mainret); // 正常退出
56         }
57         __except (_XcptFilter(GetExceptionCode(),GetExceptionInformation()) )
58     {
59         /*
60         * Should never reach here
61         */
62         _exit( GetExceptionCode() ); // 异常退出
63     } /* end of try - except */
64 }

```

从第 23 行和第 57 行的 `_try` 和 `_except` 关键字可以看到, C 运行库的入口函数中也包含了一个 SEH 处理器, 它的保护范围内包含了对 `main` 或 `WinMain` 函数的调用。这个 SEH 是应用程序初始线程的第二个异常处理器 (倒数第二个得到处理机会)。

12.2.3 基于信号的异常处理

在编译器入口函数 (见清单 12-3) 的 SEH 处理器的 `_except` 中 (第 57~64 行), 过滤表达式 (第 57 行) 的内容是调用 `_XcptFilter` 函数。`winxfltr.cpp` (`crt\src` 目录) 文件中包含了这个函数的源代码, 其函数原型如下:

```
int __cdecl _XcptFilter (unsigned long xcptnum, PEXCEPTION_POINTERS pxcptinfoptrs)
```

要搞清 `_XcptFilter` 函数的用途和工作原理, 我们必须先介绍一些背景资料。在 UNIX 或类似 UNIX 系统中, 通常使用 C 风格的信号处理机制 (C-style signal handling) 来处理异常。当有异常发生时, 系统通过检索一张专门的异常应对表 (exception action table) 来寻找异常处理函数。为了便于把使用这种机制的软件移植到 Windows 系统中, Windows 的 C 运行库包含了对这种异常处理机制的支持。在 C 运行库的源文件 `winxfltr.cpp` 中, 我们可以看到一张名为 `_XcptActTab` 的结构数组 (见清单 12-4)。

清单 12-4 C 运行库中用于支持基于信号的异常处理机制的结构数组

```
struct _XCPT_ACTION _XcptActTab[] = {
/*
 * Exceptions corresponding to the same signal (e.g., SIGFPE) must be grouped
 * together.
 *   XcptNum           SigNum     XcptAction
 *-----
 */
{ (unsigned long)STATUS_ACCESS_VIOLATION,      SIGSEGV,  SIG_DFL },
{ (unsigned long)STATUS_ILLEGAL_INSTRUCTION,    SIGILL,    SIG_DFL },
{ (unsigned long)STATUS_PRIVILEGED_INSTRUCTION, SIGILL,    SIG_DFL },
// { (unsigned long)STATUS_NONCONTINUABLE_EXCEPTION, NOSIG,    SIG_DIE },
// { (unsigned long)STATUS_INVALID_DISPOSITION,    NOSIG,    SIG_DIE },
{ (unsigned long)STATUS_FLOAT_DENORMAL_OPERAND, SIGFPE,   SIG_DFL },
{ (unsigned long)STATUS_FLOAT_DIVIDE_BY_ZERO,    SIGFPE,   SIG_DFL },
{ (unsigned long)STATUS_FLOAT_INEXACT_RESULT,    SIGFPE,   SIG_DFL },
{ (unsigned long)STATUS_FLOAT_INVALID_OPERATION, SIGFPE,   SIG_DFL },
{ (unsigned long)STATUS_FLOAT_OVERFLOW,          SIGFPE,   SIG_DFL },
{ (unsigned long)STATUS_FLOAT_STACK_CHECK,        SIGFPE,   SIG_DFL },
{ (unsigned long)STATUS_FLOAT_UNDERFLOW,          SIGFPE,   SIG_DFL },
// { (unsigned long)STATUS_INTEGER_DIVIDE_BY_ZERO, NOSIG,    SIG_DIE },
// { (unsigned long)STATUS_STACK_OVERFLOW,           NOSIG,    SIG_DIE }
};
```

简单地说, 以上数组 (表) 定义了对各个异常的处理方式。每一行对应一种异常。第一列是 Windows 中的异常代码 (状态代码); 第二列是为这个异常指定的信号量, 其中 `SIGILL` 代表无效指令 (Illegal instruction), `SIGFPE` 代表浮点错误 (Floating-point error), `SIGSEGV` 代表非法访问存储器 (Illegal storage access); 第三列代表处理异常的动作, 其中 `SIG_DFL` 表示默认的动作, `SIG_DIE` 表示终止进程。

显而易见，清单 12-4 中只定义了为数很少的 Windows 异常，这或许是因为只需要兼容这些异常就足够了。`siglookup` 函数可以用来查找一个信号（`SigNum`）所对应的处理动作是什么，`xcptlookup` 函数可以用来根据异常代码（`ExcptNum`）查找一个异常所对应的处理动作是什么。这两个函数的源代码分别被包含在 `crt\src\winsig.c` 文件和 `crt\src\winxflt.c` 中。

有了以上基础后，我们来看一看 `_XcptFilter` 是如何工作的。`_XcptFilter` 首先会调用 `xcptlookup` 函数来检索上面列出的 `_XcptActTab` 表格，并根据返回的动作采取不同的行为。

- 如果返回的动作为空（没有找到对该异常的处理动作），或者为 `SIG_DFL`，那么 `_XcptFilter` 会调用 `UnhandledExceptionFilter` 函数，然后返回，即 `return (UnhandledExceptionFilter(pxcptinfoptrs))`。
- 如果返回的动作为 `SIG_DIE`，那么 `_XcptFilter` 会返回 `EXCEPTION_EXECUTE_HANDLER`，让系统执行异常处理块（清单 12-3 的第 59~64 行）。
- 如果返回的动作为 `SIG_IGN` (`ignore` 忽略)，那么 `_XcptFilter` 会返回 `EXCEPTION_CONTINUE_EXECUTION`，让系统恢复执行发生异常的程序。

从清单 12-4 中可以看到，除了被注释的四行，其他行的处理动作都为 `SIG_DFL`，这意味着今天的 `_XcptFilter` 函数已经简化为对 `UnhandledExceptionFilter` 函数的简单调用。

至此，我们知道，一个典型的 Win32 程序的初始线程（主线程）会有两个 SEH 异常处理器和一个 C 运行库的基于信号的异常处理器。基于信号的异常处理器主要是为了兼容来自 UNIX 系统的软件才存在的，对于大多数 Win32 程序已经不起什么作用了。两个 SEH 异常处理器：一个是 `BaseProcessStart` 函数设置的，另一个是 C 运行库的启动函数设置的。两个 SEH 异常处理器的过滤表达式都直接或间接调用了 `UnhandledExceptionFilter` 函数。

12.2.4 试验：观察默认的异常处理器

下面通过一个试验（调试清单 12-1 的 UEF 程序）来加深大家的理解。启动 WinDBG，通过 `File>Open Executable` 菜单打开 `UEF.exe`，建立调试对话。键入 `bp kernel32! UnhandledExceptionFilter` 命令，设置一个断点。键入 `g` 命令让程序运行。

由于 `main` 函数中向空指针赋值导致异常，因此执行到这里时程序会中断到调试器。

```
(f88.11a8): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
```

从提示信息中知道，这是第一次处理机会。键入 `k` 命令观察函数调用栈：

```
0:000> k
ChildEBP RetAddr
0012ff80 00401299 UEF!main+0x25 [c:\dig\dbg\author\code\chap03\uef\uef.cpp @ 22]
0012ffc0 77e8141a UEF!mainCRTStartup+0xe9 [crt0.c @ 206]
0012ffff0 00000000 kernel32!BaseProcessStart+0x23
```

可以看到线程的起始函数为 `BaseProcessStart`, 它调用了 `mainCRTStartup`, `mainCRTStartup` 又调用了 `main`, 这与我们的介绍完全一致。

键入 `g` 命令让程序继续运行, 让系统继续分发异常, 接下来应该是 VEH 处理器和 SEH 处理器。因为在我们的代码(见清单 12-1)中没有任何的异常保护器, 所以系统会找到默认的异常保护器。前面我们介绍有两个异常保护器都调用了 `UnhandledExceptionFilter` 函数, 而且已对 `UnhandledExceptionFilter` 设置了断点, 所以系统在评估过滤表达式时会触发这个断点:

```
Breakpoint 0 hit
eax=00424ce8 ebx=0012ffb0 ecx=00424ce8 edx=0012fb5c esi=00000000 edi=00422138
eip=77e99b95 esp=0012fb14 ebp=0012fb30 iopl=0 nv up ei pl zr na po nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000246
kernel32!UnhandledExceptionFilter:
77e99b95, 6800050000 push 0x500
```

那么, 这个过滤表达式是位于 `BaseProcessStart` 还是 `mainCRTStartup` 中的呢? 键入 `k` 命令便知道了。

```
0:000> k
ChildEBP RetAddr
0012fb10 00402efe kernel32!UnhandledExceptionFilter
0012fb30 004012bf UEF!_XcptFilter+0x2e [winxfltr.c @ 228]
0012ffc0 77e8141a UEF!mainCRTStartup+0x10f [crt0.c @ 212]
0012ffff0 00000000 kernel32!BaseProcessStart+0x23
```

从对 `_XcptFilter` 的调用我们知道, 这是系统在执行 `mainCRTStartup` 中的过滤表达式。

我们在后面会仔细讨论 `UnhandledExceptionFilter` 的内部机制, 现在不妨先看它返回什么。键入 `gu` (`Go Up`) 命令让这个函数执行完:

```
0:000> gu
eax=00000000 ...
```

从 `EAX` 寄存器的值可以看到, `UnhandledExceptionFilter` 函数返回的是 0。因为在过滤表达式中 0 代表的是继续搜索, 所以这个返回值意味着是让系统继续搜索其他异常处理器。此处需要说明的是, `UnhandledExceptionFilter` 函数的返回值是与当前进程是否在被调试有关, 我们将在下一节介绍其细节。现在, 键入 `g` 命令继续运行。接下来, WinDBG 提示断点 0 再次被命中:

```
Breakpoint 0 hit ...
```

这次命中的是 `UnhandledExceptionFilter` 函数上的断点。这是因为系统在收到刚才的返回值后, 继续寻找其他异常处理器, 因此找到了 `BaseProcessStart` 函数中的 SEH 处理器, 再执行它的过滤表达式。键入 `k` 命令, 可以看到这一点:

```
0:000> k
ChildEBP RetAddr
0012fb34 77e9a2ad kernel32!UnhandledExceptionFilter
0012ffff0 00000000 kernel32!BaseProcessStart+0x39
```

再次执行 `gu` 命令并通过 `EAX` 寄存器观察 `UnhandledExceptionFilter` 函数的返回值，发现仍然是 0，看来它还是声明“不处理该异常”。

键入 `g` 继续运行，这回用户态的异常分发函数调用 `RaiseException`，发起对这个异常的第二轮分发。果然 WinDBG 接下来显示：

```
(1438.f24): Access violation - code c0000005 (!!! second chance !!!)
eax=0000000d ebx=7ffd000 ecx=00424a60 edx=00424a60 esi=000391c8 edi=0012ff80
```

这是因为当系统在分发第二轮异常时，会先分发给调试器，对于第二轮处理机会，WinDBG 默认会返回 `DBG_CONTINUE`（参见第 11.3.3 节），即告诉系统，调试器已经处理了异常，这会导致系统结束异常分发，恢复执行刚才发生异常的代码。但因为错误条件并没有消除，所以还是会再次发生异常，于是 WinDBG 又得到第一轮处理机会，开始重复前面的过程。

通过前面的讨论，大家已经清楚对于典型的 Win32 进程的初始线程（主线程），系统的 `BaseProcessStart` 函数和编译器嵌入的启动函数（`[w]mainCRTStartup` 或 `[w]WinMainCRTStartup`）分别会加入一个 SEH 处理器，来处理用户代码中的未处理异常。那么，如果是其他线程出现了未处理异常，情况又如何呢？

12.2.5 BaseThreadStart 函数中的 SEH 处理器

对于初始线程之外的其他线程，其用户态的起始地址是系统提供的另一个位于 `kernel32.dll` 中的函数 `BaseThreadStart`。清单 12-5 给出了 `BaseThreadStart` 函数的伪代码，显而易见，系统在该函数也提供了一个 SHE 处理器。

清单 12-5 BaseThreadStart 函数的伪代码

```

1  VOID BaseThreadStart(
2      IN LPTHREAD_START_ROUTINE lpStartAddress,
3      IN LPVOID lpParameter)
4  {
5      __try
6      {
7          ExitThread((lpStartAddress)( lpParameter));
8      }
9      __except (UnhandledExceptionFilter(GetExceptionInformation()))
10     {
11         if ( !BaseRunningInServerProcess )
12         {
13             ExitProcess(GetExceptionCode());
14         }
15         else
16         {
17             ExitThread(GetExceptionCode());
18         }
19     }
20 }
```

下面通过一段实际代码来讨论非主线程的未处理异常。清单 12-6 列出了 UefSndThrd 程序的源代码, UefSndThrd 的意思是第二个线程中的未处理异常(Unhandled Exception in Second Thread)。

清单 12-6 UefSndThrd (第二个线程中的未处理异常) 程序的源代码

```

1 // UefSndThrd.cpp : Demonstrate Unhandled Exceptions in non-startup thread.
2 // Raymond May 1st 2006
3 //
4
5 #include <windows.h>
6 #include <stdio.h>
7
8 DWORD WINAPI UefThreadProc(
9     LPVOID lpParameter
10 )
11 {
12     int n=0;
13     n=1/n;
14     return S_OK;
15 }
16
17 int main(int argc, char* argv[])
18 {
19     printf("Hello Advanced Debugging!\n");
20     CreateThread(NULL, 0, UefThreadProc,
21                 NULL, 0, NULL);
22     getchar();
23     return 0;
24 }
```

UefSndThrd 程序的 main 函数创建了第二个线程(第 20 和 21 行), 然后等待一个用户按键(第 22 行), 以便给线程 2 运行机会, 如果没有第 22 行的等待, 那么主线程会立即退出(第 23 行), 而线程 2 可能还没有得到执行。线程 2 在没有设置任何异常处理器的情况下作了除 0 操作, 这显然会导致一个未处理异常。

建议读者模仿上文的试验步骤使用 WinDBG 来调试 UefSndThrd 程序, 亲自观察和比较发生在非初始线程中的未处理异常, 以及发生在初始线程中的有什么不同。我们把这个问题留给读者实践。

12.3 未处理异常过滤函数

BaseProcessStart 和 BaseThreadStart 的 SEH 异常处理器的过滤表达式都调用了 UnhandledExceptionFilter 函数, 编译器的启动函数([w]mainCRTStartup 或 [w]WinMainCRTStartup) 的异常处理器也间接调用了 UnhandledExceptionFilter 函数(在_XcptFilter 中调用)。因此, 如果说默认的异常处理器是系统“处理”“未处理异常”的地方, 那么 UnhandledExceptionFilter 就是选择处理办法的决策者。因为这个函数在 Windows 系统中有着非常重要的地位, 所以我们将在本节对其做深入

的介绍。我们先简要介绍 Windows XP 之前的版本，然后详细介绍 Windows XP 中的版本。下一节将简要介绍 Windows Vista 版本的主要变化。

12.3.1 Windows XP 之前

清单 12-7 给出了 Windows 2000 和 NT 中的 UnhandledExceptionFilter 函数的伪代码，描述这个函数所执行的主要动作。该代码参考了 Matt Pietrek 在关于 SEH 工作原理一文（参考文献 1）中给出的代码。

清单 12-7 UnhandledExceptionFilter 函数的伪代码（Windows XP 之前）

```

1  LONG UnhandledExceptionFilter(STRUCT _EXCEPTION_POINTERS * ExInfo)
2
3  {
4      PEXCEPTION_RECORD pExcptRec = ExInfo->ExceptionRecord;
5      // 检查是否是因为写资源区而导致的访问违例
6      if ( (pExcptRec->ExceptionCode == EXCEPTION_ACCESS_VIOLATION)
7          && (pExcptRec->ExceptionInformation[0]) )
8      {
9          retValue = _BasepCheckForReadOnlyResource(
10              pExcptRec->ExceptionInformation[1]);
11          // 如果已经处理（改为可以写），则恢复执行
12          If ( EXCEPTION_CONTINUE_EXECUTION == retValue )
13              return EXCEPTION_CONTINUE_EXECUTION;
14      }
15
16      // 检查当前进程是否正在被调试，如果在被调试，则返回不处理，让系统继续分发
17      retValue = NtQueryInformationProcess(GetCurrentProcess(),
18                                         ProcessDebugPort, &hDebugPort, sizeof(hDebugPort), 0 );
19      if ( (retValue >= 0) && hDebugPort != NULL )
20          return EXCEPTION_CONTINUE_SEARCH;
21
22      // 是否有通过 SetUnhandledExceptionFilter 注册的顶层过滤函数，如果有，则调用
23      if ( _BasepCurrentTopLevelFilter )
24      {
25          retValue = _BasepCurrentTopLevelFilter( ExInfo );
26          If ( EXCEPTION_EXECUTE_HANDLER == retValue
27              || EXCEPTION_CONTINUE_EXECUTION == retValue )
28              return retValue;
29          // 如果顶层过滤函数已经处理则返回，否则继续
30      }
31
32      // 检测错误模式中是否禁止显示 GPF 对话框 (SEM_NOGPFAULTERRORBOX)
33      if ( (GetErrorMode() & SEM_NOGPFAULTERRORBOX) )
34          return EXCEPTION_EXECUTE_HANDLER;
35      // 从注册表读取 AeDebug 选项，参见 12.5.1 节
36      retValue = _GetProfileStringA( "AeDebug", "Debugger", 0,
37                                   szDbgCmdFmt, sizeof(szDbgCmdFmt)-1 );
38      if ( retValue ) // 如果存在 Debugger 键值，则增加用于开始 JIT 调试器的按钮
39          dwDlgOptionFlag = DlgOptionOKCancel;
40      // 读取 AeDebug 中的 Auto 键值
41      retValue = GetProfileStringA( "AeDebug", "Auto", "0",
42                                   szAeDebug, sizeof(szAeDebug)-1 );
43      if ( retValue && ( 0 == strcmp(szAeDebug, "1" ) )
44          && ( DlgOptionOKCancel == dwDlgOptionFlag )
45          fAeDebugAuto = TRUE;
46

```

```

47     if ( FALSE == fAeDebugAuto ) // 如果不需要自动启动 JIT 调试器
48     {
49         returnValue = NtRaiseHardError(
50             STATUS_UNHANDLED_EXCEPTION | 0x10000000, 4, 0, Parameters,
51             _BasepAlreadyHadHardError ? 1 : dwDlgOptionFlag, &dwResponse );
52     }
53     else // 把变量设置成和用户选择调试按钮一样的值
54     {
55         dwResponse = ResponseCancel; // Cancel 按钮代表调试
56         returnValue = STATUS_SUCCESS;
57     }
58 // 如果需要自动启动 JIT 调试或用户选择了 Cancel 按钮，则发起 JIT 调试
59 if ( NT_SUCCESS(returnValue) && ( dwResponse == ResponseCancel )
60     && ( !_BasepAlreadyHadHardError )
61     && ( !_BasepRunningInServerProcess ))
62 {
63     _BasepAlreadyHadHardError = TRUE;
64     returnValue = CreateProcessA(...); // 启动 JIT 调试器，参见 12.5 节
65     if ( returnValue && hEvent )
66     {
67         do { // 等待 JIT 调试器设置事件对象
68             returnValue = NtWaitForSingleObject(
69                 EventHandle, TRUE, NULL);
70             } while (Status == STATUS_USER_APIC || Status==STATUS_ALERTED);
71
72         return EXCEPTION_CONTINUE_SEARCH;
73     }
74 }
75 if ( _BasepAlreadyHadHardError )
76     NtTerminateProcessGetCurrentProcess(), pExcptRec->ExceptionCode);
77
78 return EXCEPTION_EXECUTE_HANDLER;
79 }

```

下面结合以上代码来讨论 UnhandledExceptionFilter 函数(XP 之前)的工作原理。

首先，UnhandledExceptionFilter 的参数就是我们前面介绍的 EXCEPTION_POINTERS 结构，其返回值为 EXCEPTION_CONTINUE_SEARCH(0)、EXCEPTION_CONTINUE_EXECUTION(1) 和 EXCEPTION_EXECUTE_HANDLER(-1) 这 3 个常量之一。可以把 UnhandledExceptionFilter 的处理措施分为如下几个步骤。

第一步(第 6~14 行)，如果异常代码为 EXCEPTION_ACCESS_VIOLATION(非法访问异常)，而且异常信息数组的元素 0 (ExceptionInformation[0]) 不是 0，那么调用 BasepCheckForReadOnlyResource 函数进行处理。对于非法访问异常，ExceptionInformation[0] 用来表示导致异常的访问类型，0 代表读，1 代表写，ExceptionInformation[1] 表示导致异常的访问地址。在 BasepCheckForReadOnlyResource 函数中，它首先会根据参数所指定的地址查找到它所在的内存块，然后判断其类型属性，如果这个内存块的类型是执行映像 (MEM_IMAGE, 0x01000000)，那么它会返回 EXCEPTION_CONTINUE_SEARCH；如果这个地址属于一个资源块 (resource)，那么 BasepCheckForReadOnlyResource 会尝试将该内存块的属性修改为可以写，如果成功，则 BasepCheckForReadOnlyResource 返回 EXCEPTION_CONTINUE_EXECUTION，

让程序恢复继续执行。

第二步（第 17~20 行），如果当前进程正在被调试（DebugPort 非空），那么返回 EXCEPTION_CONTINUE_SEARCH。当 KiUserExceptionDispatcher 收到此返回值后，会通过调用 ZwRaiseException 并将 FirstChance 参数设为假，让内核中的 KiExceptionDispatch 开始对该异常进行第二轮分发。因为该进程在被调试，所以第二轮机会先分发给调试器。对于第二轮处理机会，调试器的典型处理是返回 DBG_CONTINUE，也就是恢复执行。但因为错误情况没有被消除，所以异常通常会再次发生，我们看到的就是程序在有问题的代码处反复执行。

第三步（第 22~30 行），如果函数指针 BasepCurrentTopLevelFilter 不为空，便调用它。BasepCurrentTopLevelFilter 是系统（kernel32.dll）定义的全局变量（当前进程范围），用来记录用户通过 SetUnhandledExceptionFilter API 设置的顶层（top-level）异常过滤函数。利用这一机制，应用程序可以设置自己的一个用于处理未处理异常的调用函数。我们稍后对其做详细讨论。

第四步（第 33~34 行），如果当前进程的 ErrorMode 中包含 SEM_NOGPFAULTERRORBOX 标志，就返回 EXCEPTION_EXECUTE_HANDLER，也就是去执行异常处理块。根据前面的介绍，异常处理块的处理方法通常是退出进程或线程。因此，如果这里返回 EXCEPTION_EXECUTE_HANDLER，那么当前进程或线程便退出了。

第五步（第 36~45 行），整理异常信息和读取注册表中的 AeDebug 设置。我们将在下一节详细介绍 AeDebug 设置。

第六步（第 47~57 行），如果不需要自动启动 JIT 调试器，那么通过系统的 HardError 机制弹出“应用程序错误”对话框，以征询用户的处理意见。如果要，则不显示对话框，直接将代表用户响应结果的变量设置为需要启动 JIT 调试器。

第七步（第 59~76 行），如果成功弹出“应用程序错误”对话框，并得到用户回应，而且用户选择了调试按钮，或者注册表中设置的是自动启动 JIT 调试器，那么启动 JIT（Just In Time）调试器，其细节我们将在第 12.5 节介绍。否则，如果 BasepAlreadyHadHardError 全局变量（在 kernel32.dll 中）为真，则终止当前进程。对于每个进程，BasepAlreadyHadHardError 都被初始化为 FALSE，UnhandledExceptionFilter 函数在启动 JIT 调试器前将其置为 TRUE（第 59 行）。这意味着每个进程都只有一次被 JIT 调试的机会。

关于以上代码还有以下两点值得说明。首先，全局变量 BaseRunningInServer-Process（也定义在 KERNEL32 中）是用来标志当前进程是否为 Windows 子系统进程的服务器进程（CSRSS.EXE），大家不要误以为它标志的是 Windows 中的服务进程（NT Service）。第二，UnhandledExceptionFilter 函数总是在默认桌面（WINST0Default）（第 61 行）创建 JIT 调试器。这意味着其他桌面将无法看到 JIT 调试器。

12.3.2 Windows XP

Windows XP 对未处理异常的处理办法作了部分改进和增强，引入了新的报告应用程序错误的界面和方法，支持通过网络发送错误报告，并加入了对应用程序验证机制的支持。清单 12-8 给出了 Windows XP 的 UnhandledExceptionFilter 函数的伪代码，尽管该代码是根据 Windows XP SP1 中版本号为 5.1.2600.1560 的 KERNEL32.DLL 模块编写的，但是，其中大部分内容对于所有 Windows XP 版本和 Windows Vista 都是适用的。

需要说明的是，编写这段伪代码的难度和所花费的时间远远超出了笔者最初的估计，尽管笔者已经尽了很大努力使其接近真实代码，但是仍然可能与实际情况存在差异。

清单 12-8 Windows XP 的 UnhandledExceptionFilter 函数的伪代码

```

1  LONG UnhandledExceptionFilter( struct _EXCEPTION_POINTERS *ExInfo )
2  {
3      NTSTATUS Status;
4      ULONG_PTR Parameters[ 4 ];
5      ULONG Response; // [ebp-39Ch]
6      HANDLE DebugPort;
7      CHAR AeDebuggerCmdLine[256];
8      CHAR AeAutoDebugString[8];
9      BOOLEAN AeAutoDebug; // [ebp-171h]
10     ULONG ResponseFlag;
11     LONG FilterReturn;
12     PRTL_CRITICAL_SECTION pPebLock;
13     JOBOBJECT_BASIC_LIMIT_INFORMATION BasicLimit;
14     HMODULE hFaultRepDll; // ebp-68h
15     EFaultRepRetVal nFaultRepRetVal=4; // [ebp-1ch]
16     TCHAR szFaultReportModuleName[MAX_PATH];
17     HANDLE hRealProcessHandle; // [ebp-3A0h]
18     HANDLE hRealThreadHandle; // [ebp-3A4h]
19     Pfn_REPORTFAULT pfnReportFault;
20
21     if(ExInfo->ExceptionRecord->ExceptionCode ==
22         STATUS_ACCESS_VIOLATION
23         && ExInfo->ExceptionRecord->ExceptionInformation[0] )
24     {
25         FilterReturn = BasepCheckForReadOnlyResource(
26             (PVOID)ExInfo->ExceptionRecord->ExceptionInformation[1]);
27         if ( FilterReturn == EXCEPTION_CONTINUE_EXECUTION )
28             return FilterReturn;
29     }
30     Status = NtQueryInformationProcess(
31         GetCurrentProcess(), ProcessDebugPort,
32         (PVOID)&DebugPort, sizeof(DebugPort), NULL);
33     if ( NT_SUCCESS(Status) && DebugPort )
34     {
35         if( (NtCurrentPeb()->NtGlobalFlag & 1) == 0)
36             return EXCEPTION_CONTINUE_SEARCH;
37         else
38         {
39             if(ExInfo->ExceptionRecord->ExceptionCode ==
40                 STATUS_ACCESS_VIOLATION)
41             {
42                 if(InterlockedExchange(77ED9328,1)==0)
43                 {
44                     RtlApplicationVerifierStop(20000002h,
45                     "access violation exception for current stack trace",

```

```

46     ExInfo->ExceptionRecord->ExceptionInformation[1],
47     "Invalid address being accessed",
48     ExInfo->ExceptionRecord->ExceptionAddress,
49     "Code performing invalid access",
50     ExInfo->ExceptionRecord,
51     ".exr (exception record)",
52     ExInfo->ContextRecord,
53     ".cxe (context record)");
54 }
55 if(ExInfo->ExceptionRecord->ExceptionCode ==
56     STATUS_INVALID_HANDLE)
57 {
58     if(InterlockedExchange(77ED9328,1)==0)
59     {
60         RtlApplicationVerifierStop(20000300h,
61             "invalid handle exception for current stack trace",
62             NULL,NULL,NULL,NULL,
63             NULL,NULL,NULL,NULL);
64     }
65 }
66 }
67 return EXCEPTION_CONTINUE_SEARCH;
68 }
69 if ( BasepCurrentTopLevelFilter )
70 {
71     FilterReturn = (BasepCurrentTopLevelFilter)(ExInfo);
72     if ( FilterReturn == EXCEPTION_EXECUTE_HANDLER || 
73         FilterReturn == EXCEPTION_CONTINUE_EXECUTION )
74         return FilterReturn;
75 }
76 // Check ErrorMode of current process
77 if ( GetErrorMode() & SEM_NOGPFAULTERRORBOX )
78     return EXCEPTION_EXECUTE_HANDLER;
79
80 Status = NtQueryInformationJobObject(
81     NULL, JobObjectBasicLimitInformation,
82     &BasicLimit, sizeof(BasicLimit), NULL);
83 if ( NT_SUCCESS(Status) && (BasicLimit.LimitFlags &
84     JOB_OBJECT_LIMIT_DIE_ON_UNHANDLED_EXCEPTION) )
85     return EXCEPTION_EXECUTE_HANDLER;
86
87 Parameters[0]=ExInfo->ExceptionRecord->ExceptionCode;
88 Parameters[1]=ExInfo->ExceptionRecord->ExceptionAddress;
89 if(ExInfo->ExceptionRecord->ExceptionCode== STATUS_IN_PAGE_ERROR)
90     Parameters[ 2 ] = ExInfo->ExceptionRecord->ExceptionInformation[ 2 ];
91 else
92     Parameters[ 2 ] = ExInfo->ExceptionRecord->ExceptionInformation[ 0 ];
93 Parameters[ 3 ] = ExInfo->ExceptionRecord->ExceptionInformation[ 1 ];
94
95 ResponseFlag = OptionOk;
96 AeAutoDebug = FALSE;
97
98 pPebLock = NtCurrentPeb()->FastPebLock;
99 // Check whether this thread owns PebLock
100 if (pPebLock ->OwningThread != NtCurrentTeb()->ClientId.UniqueThread )
101 {
102     __try
103     {
104         if ( GetProfileString("AeDebug", "Debugger", NULL,
105             AeDebuggerCmdLine, sizeof(AeDebuggerCmdLine)-1))
106             ResponseFlag = OptionOkCancel;
107
108

```

```

109         if ( GetProfileString("AeDebug", "Auto", "0",
110             AeAutoDebugString, sizeof(AeAutoDebugString)-1) )
111     {
112         if ( !strcmp(AeAutoDebugString, "1") )
113         {
114             if ( ResponseFlag == OptionOkCancel )
115                 AeAutoDebug = TRUE;
116         }
117     }
118     __except(EXCEPTION_EXECUTE_HANDLER)
119     {
120         ResponseFlag = OptionOk;
121         AeAutoDebug = FALSE;
122     }
123
124     PebLockPointer = NtCurrentPeb()->LoaderLock;
125     if ( !_BasepAlreadyHadHardError && PebLockPointer->OwningThread
126         != NtCurrentTeb()->ClientId.UniqueThread )
127     {
128         if(!AeAutoDebug || StrFunc_StartWithIgnoreCase(
129             AeDebuggerCmdLine, "drwtsn32")!=NULL)
130         {
131             szFaultReportModuleName[0]=0;
132             if(GetSystemDirectory(szFaultReportModuleName,MAX_PATH))
133                 strcat(szFaultReportModuleName,"\\faultrep.dll");
134             else
135                 szFaultReportModuleName[0]=0;
136             LdrLockLoaderLock(&ulMagic, &ulResult, 2 );
137             if(ulResult==1)
138             {
139                 hFaultRepDll=LoadLibraryEx(szFaultReportModuleName,
140                     NULL,NULL)
141                 LdrUnlockLoaderLock(0, ulMagic);
142             }
143
144             if(hFaultRepDll)
145             {
146                 if(ResponseFlag!=ResponseOkCancel)
147                 {
148                     dwReportFaultMode=0;
149                 }
150                 pfnReportFault = (pfn_REPORTFAULT)GetProcAddress(
151                     hFaultRepDll, "ReportFault" ) ;
152                 if ( pfnReportFault )
153                 {
154                     nFaultRepRetVal = pfnReportFault
155                         ( pExceptionPointers,dwReportFaultMode) ;
156                 }
157                 FreeLibrary(hFaultRepDll );
158                 hFaultRepDll=NULL;
159             }
160             if(nFaultRepRetVal == frrvLaunchDebugger)
161                 AeAutoDebug=TRUE;
162             } // !AeAutoDebug || JIT debugger is drwtsn32
163             } // ( !_BasepAlreadyHadHardError && not holding LoaderLock
164             } // Not holding the PebLock
165
166             if(!AeAutoDebug
167                 && (nFaultRepRetVal==frrvErrNoDW
168                     || (ResponseFlag==ResponseOkCancel &&
169                         (nFaultRepRetVal==frrvErrTimeout
170                             || nFaultRepRetVal==frrvOkQueued
171                             || nFaultRepRetVal==frrvOkHeadless

```

```

172
173
174
175
176     Status=NtRaiseHardError( STATUS_UNHANDLED_EXCEPTION
177     | HARDERROR_OVERRIDE_ERRORMODE,
178     4, 0, Parameters,
179     BasepAlreadyHadHardError ? OptionOk : ResponseFlag,
180     &Response);
181 }
182 else
183 {
184     Status = STATUS_SUCCESS;
185     Response = ResponseCancel;
186 }
187
188 if ( NT_SUCCESS(Status) && Response == ResponseCancel
189     && BasepAlreadyHadHardError == FALSE)
190 {
191     if(_BaseRunningInServerProcess!=0)
192         _BasepAlreadyHadHardError=TRUE;
193     else
194     {
195         BOOL b;
196         STARTUPINFO StartupInfo;
197         PROCESS_INFORMATION ProcessInformation;
198         CHAR CmdLine[256];
199         NTSTATUS Status;
200         HANDLE EventHandle; // [ebp-3B4h]
201         SECURITY_ATTRIBUTES sa;
202
203         // Convert pseudo handle returned by GetCurrentProcess
204         // to a real handle to the process.
205         if(!DuplicateHandle(GetCurrentProcess(), //SourceProcessHandle
206             GetCurrentProcess(), //SourceHandle
207             GetCurrentProcess(), //TargetProcessHandle
208             &hRealProcessHandle,
209             NULL,TRUE,DUPPLICATE_SAME_ACCESS))
210     {
211         hRealProcessHandle=NULL;
212     }
213     if(!DuplicateHandle(GetCurrentProcess(), //SourceProcessHandle
214             GetCurrentProcess(), //SourceHandle
215             0xFFFFFFF, //TargetProcessHandle, that's current thread
216             &hRealThreadHandle,
217             NULL,TRUE,DUPPLICATE_SAME_ACCESS))
218     {
219         hRealThreadHandle=NULL;
220     }
221     sa.nLength = sizeof(sa);
222     sa.lpSecurityDescriptor = NULL;
223     sa.bInheritHandle = TRUE;
224     EventHandle = CreateEvent(&sa,TRUE,FALSE,NULL);
225     RtlZeroMemory(&StartupInfo,sizeof(StartupInfo));
226     sprintf(CmdLine,AeDebuggerCmdLine,
227             GetCurrentProcessId(),EventHandle);
228     StartupInfo.cb = sizeof(StartupInfo);
229     StartupInfo.lpDesktop = "Winsta0\\Default";
230     CsrIdentifyAlertableThread();.text:77E99EE6
231     b=CreateProcess(NULL, CmdLine, NULL,      NULL,
232                     TRUE, 0, NULL, NULL, &StartupInfo,
233                     &ProcessInformation);
234     if(hRealProcessHandle)

```

```

235             CloseHandle(hRealProcessHandle);
236             if(hRealThreadHandle)
237                 CloseHandle(hRealThreadHandle);
238
239             if(b && EventHandle)
240             {
241                 do
242                 {
243                     ahHandles[0]=EventHandle;
244                     ahHandles[1]=ProcessInformation.hProcess;
245                     Status = NtWaitForMultipleObjects(
246                         2, ahHandles,
247                         TRUE, // WaitAnyObject
248                         TRUE, // Alertable
249                         FALSE);
250                 }while (Status == STATUS_USER_APIC
251                         || Status == 0x101); //STATUS_ALERTED
252
253             if(Status== WAIT_OBJECT_1 /*1*/)
254             {
255                 DebugPort = (HANDLE)NULL;
256                 Status = NtQueryInformationProcess(
257                     GetCurrentProcess(), ProcessDebugPort,
258                     (PVOID)&DebugPort, sizeof(DebugPort),
259                     NULL);
260
261                 if ( !NT_SUCCESS(Status) || DebugPort==NULL )
262                     _BasepAlreadyHadHardError=TRUE;
263             }
264             CloseHandle(EventHandle);
265             CloseHandle(ProcessInformation.hProcess);
266             CloseHandle(ProcessInformation.hThread);
267         }
268         else//(b && EventHandle)
269             _BasepAlreadyHadHardError=TRUE;
270     }//(!_BasepRunningInServerProcess)
271 // need to spawn Jit debugger
272 if ( _BasepAlreadyHadHardError )
273 {
274     NtTerminateProcess(NtCurrentProcess(),
275                     ExInfo->ExceptionRecord->ExceptionCode);
276 }
277
278     return EXCEPTION_EXECUTE_HANDLER;
279 }

```

如果您完全跳过前面的代码直接翻到本页，那么，笔者建议您在我分析下面的每一点时，应该按照我提示的行号返回去阅读相关的代码。因为有些时候，阅读代码是理解软件流程（工作原理）的最好办法。

上面的代码与清单 12-7 所列出的 Windows XP 之前的 UnhandledExceptionFilter 函数相比，主要的不同之处有以下几点。

第一，新增了对应用程序验证机制的支持（第 39~67 行）。对 STATUS_ACCESS_VIOLATION 和 STATUS_INVALID_HANDLE 异常，如果当前进程处于被调试状态（DebugPort!=0），而且当前进程块的 NtGlobalFlag 包含 0x100 标志时（即 bit 8 为 1），新的函数会调用 RtlApplicationVerifierStop 函数，分别打印出类似清单 12-9 和

清单 12-10 所示的信息供调试使用。

清单 12-9 RtlApplicationVerifierStop 函数为非法访问 (ACCESS VIOLATION) 异常打印的调试信息

```
=====
VERIFIER STOP 00000002: pid 0xCAC: access violation exception for current stack trace
00000000 : Invalid address being accessed
00401440 : Code performing invalid access
0012F9BC : .exr (exception record)
0012F9D8 : .cxr (context record)
=====
```

清单 12-10 RtlApplicationVerifierStop 函数为 STATUS_INVALID_HANDLE 异常打印的调试信息

```
=====
VERIFIER STOP 00000300: pid 0xCAC: invalid handle exception for current stack trace
00000000 : (null)
00000000 : (null)
00000000 : (null)
00000000 : (null)
=====
```

概言之，Windows XP 的 UnhandledExceptionFilter 函数对调试 STATUS_ACCESS_VIOLATION 和 STATUS_INVALID_HANDLE 异常提供了特别支持。

第二，增加了新的方法来提示应用程序错误（第 131~161 行）。新的函数会首先通过动态加载 faultrep.dll 模块并调用其中的 ReportFault 函数启动 DWWIN.EXE，来提示“应用程序错误”（第 150~156 行）。如果失败，再使用旧的方法通过 HardError 机制提示错误（第 176~180 行）。Windows XP 以前的 UnhandledExceptionFilter 函数只能使用 HardError 机制提示错误。其中还有一个细微的差别是，当注册表中设置了自动启动 JIT 调试器（AeDebug 的 Auto 等于“1”）时，Windows XP 的 UnhandledExceptionFilter 函数会判断 JIT 调试器是否是 DRWTSN32.EXE（第 129 行），如果是，仍会弹出错误提示对话框。以前的 UnhandledExceptionFilter 函数没有这个判断，也就是一旦 Auto 等于“1”（且 Debugger 项不为空），就直接启动 JIT 调试器而不再出现错误提示对话框。

第三，对启动 JIT 调试器作了增强。新的函数会先复制当前进程和线程的句柄（第 205~220 行），在等待调试器的循环中（第 241~251 行），也改为使用 NtWaitForMultipleObjects 来等待两个对象（进程句柄和 JIT 调试事件）。以前的实现是使用 NtWaitForSingleObject，它只等待一个事件对象，这样，当建立 JIT 调试对话过程中出错时，当前进程（发生未处理异常的进程）便会死锁在这里。

本节我们使用了较大的篇幅分析了 Windows 系统的未处理异常的过滤函数（UnhandledExceptionFilter），介绍了这个函数的工作流程和它执行的主要动作，接下来我们将分别介绍应用程序错误对话框和启动 JIT 调试器。

12.4 应用程序错误对话框

当 Windows 系统检测到应用程序内发生了未处理异常或其他严重错误时，系统的策略是将其终止，在终止前，系统通常会弹出一个对话框来通知用户这个程序即将被关闭，这个对话框通常被称为应用程序错误对话框（Application Fault Dialog）或者叫 GPF 错误框，GPF 是 General Protection Fault（通用保护错误）的缩写。

系统弹出“应用程序错误”对话框的目的有二，一是告知用户，该应用程序已经由于发生严重错误（未处理异常）而无法继续运行即将被终止，二是征求用户的处理意见：是立刻终止、启动 JIT 调试器调试该程序，还是先发送错误报告然后再终止（Windows XP 引入）。

根据上一节的介绍，应用程序错误对话框是由系统（Kernel32.dll）的未处理异常过滤器函数（UnhandledExceptionFilter）触发，由操作系统的其他程序弹出并维护（与用户交互）的。考虑到需要显示应用程序错误对话框时，应用程序本身已经发生了严重的错误，再在该进程中执行新的任务可能已经不安全，所以系统是使用其他进程来显示“应用程序错误”对话框的。具体使用哪个进程及其具体方式因为 Windows 版本的不同而有所不同，在 Windows XP 之前使用的是 HardError 机制，Windows XP 和 Windows Vista 分别引入了使用 ReportFault API 和 WER 2.0 API 的方法，但也支持以前的方法。下面分别加以介绍。

12.4.1 用 HardError 机制提示应用程序错误

在 Windows 2000 或更早的 Windows 中，系统是通过 HardError 提示机制来提示“应用程序错误”的。简单地说，就是调用 NtRaiseHardError 内核服务。

在上一节 UnhandledExceptionFilter 函数的伪代码中（见清单 12-7），我们可以看到它对 NtRaiseHardError 系统服务的调用：

```
returnValue = NtRaiseHardError(
    STATUS_UNHANDLED_EXCEPTION | 0x10000000, 4, 0, Parameters,
    _BasepAlreadyHadHardError ? 1 : dwDlgOptionFlag, &dwResponse );
```

其中，第一个参数 ErrorStatus 被指定为 STATUS_UNHANDLED_EXCEPTION|0x10000000，STATUS_UNHANDLED_EXCEPTION 代表未处理异常，与 0x10000000 进行或运算是为了设置 HARDERROR_OVERRIDE_ERRORMODE 标志（13.1.3 节将详细介绍）。Parameters 参数是一个数组，包含了关于异常的详细信息。倒数第二个参数用来指定错误对话框中包含哪些按钮。最后一个参数用于存放用户的选择（响应），它的值是如下枚举常量之一：

```
typedef enum_HARDERROR_RESPONSE {
    ResponseReturnToCaller, //0
    ResponseNotHandled, ResponseAbort, ResponseCancel, ResponseIgnore,
    ResponseNo, ResponseOk, ResponseRetry, ResponseYes
} HARDERROR_RESPONSE, *PHARDERROR_RESPONSE;
```

经过多个负责 HardError 处理和分发的内核函数的依次处理(13.1 节将详细介绍),系统把 HardError 的提示请求通过 LPC 端口发给 Windows 子系统进程(CSRSS.EXE)。CSRSS 接到请求后,根据参数中描述的要求,弹出图 12-1 所示的应用程序错误对话框。

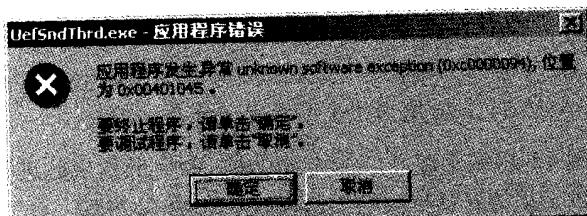


图 12-1 两种选项的“应用程序错误”对话框

对话框标题中包含了应用程序的可执行文件名称并带有“应用程序错误”字样,这是“应用程序错误”对话框这一名称的来由。

对话框的文字中包含了对错误的简单描述(未知的软件异常,异常代码为 0xC0000094,对应于 STATUS_INTEGER_DIVIDE_BY_ZERO)和错误的发生位置(即导致错误的指令地址,0x00401045)。这些信息是通过 Parameters 参数传递给 CSRSS 的。

如果点击确定(OK)按钮,UnhandledExceptionFilter 函数会返回 EXCEPTION_EXECUTE_HANDLER 给默认的异常处理器,接下来系统执行异常处理块,退出应用程序。

如果点击取消(Cancel)按钮,UnhandledExceptionFilter 会启动注册在系统中的 JIT 调试器(下一节讨论)。如果系统内没有注册任何 JIT 调试器,那么便不会有取消按钮,如图 12-2 所示。

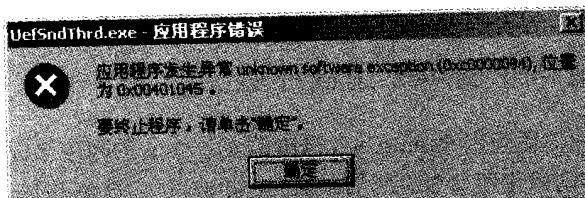


图 12-2 只包含确定选项的“应用程序错误”对话框

在用户选择了某个按钮后,NtRaiseHardError 通过 dwResponse 参数将选择结果返回给 UnhandledExceptionFilter 函数。如果当前进程设置了禁止弹出“应用程序错误”对话框(比如前面的 ErrorMode 程序),那么 dwResponse 会被设置为 ResponseReturnToCaller(返回给调用者)。如果在分发和处理过程中遇到错误,那么 NtRaiseHardError 的函数返回值会指示调用失败。

如果 NtRaiseHardError 返回成功,而且以下 3 个条件都满足后,UnhandledExceptionFilter 便准备启动 JIT 调试器。

- 用户选择的是 Cancel 按钮(dwResponse== ResponseCancel);

- 当前进程的 _BasepAlreadyHadHardError 全局变量为假 (`!_BasepAlreadyHadHardError`);
- 当前进程不是服务进程 (`!_BaseRunningInServerProcess`)。

如果不满足启动调试器的条件，那么 `UnhandledExceptionFilter` 便调用 `NtTerminateProcess(GetCurrentProcess(), pExcptRec->ExceptionCode)` 终止当前进程，`pExcptRec->ExceptionCode` 用来指定退出码（异常代码）。

使用 Spy++ 工具观察如图 12-1 或图 12-2 所示的对话框，找到其所属的进程 ID，然后再打开任务管理器，可以发现该对话框的宿主进程就是 CSRSS 进程。

因为 HardError 对话框是由 CSRSS 进程弹出的，所以，即使当系统被锁定（lock）或无人登录时，应用程序错误对话框也是可见的（参见图 12-3）。

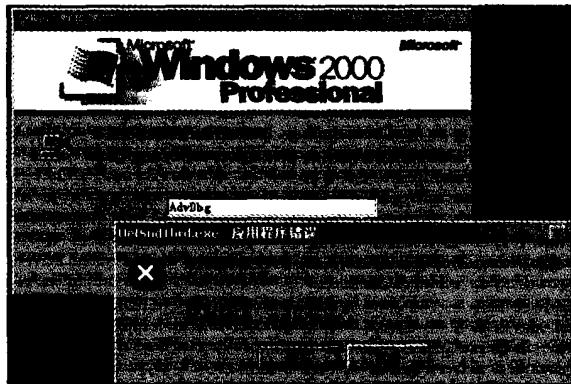


图 12-3 通过 HardError 机制提示的“应用程序错误”对话框在桌面被锁定后仍然可见

12.4.2 使用 ReportFault API 提示应用程序错误

Windows XP 系统会优先考虑使用 `ReportFault` API 启动一个新的程序来提示应用程序错误，如果该机制失败，则仍使用前面讲的 HardError 机制。下面介绍其具体过程，我们将重点介绍与 Windows 2000 不同的地方。

在 `UnhandledExceptionFilter` 函数决定了需要弹出应用程序错误对话框后，它会首先通过 `GetSystemDirectory` 取得系统目录 (%WINNT%\SYSTEM32)，然后将“FAULTREP.DLL”追加到此路径上得到 FAULTREP.DLL 的全路径，FAULTREP.DLL 是 Windows XP 引入的专门用于错误提示的系统模块。它受到系统文件保护机制（System File Protection，简称 SFP）的保护，因此，试图用一个同名的文件替代它时，尽管当时似乎成功了，但是系统会很快把它恢复回来。FAULTREP.DLL 的模块文件名称是硬编码到 `UnhandledExceptionFilter` 函数中的，这也是为了避免配置在注册表中可能被篡改。

而后，UnhandledExceptionFilter 函数便调用 LoadLibraryEx 来动态加载 FAULTREP.DLL，加载成功后通过 GetProcAddress 得到 ReportFault 函数（API）的地址。ReportFault 函数的原型如下：

```
EFaultRepRetVal APIENTRY ReportFault(LPEXCEPTION_POINTERS pep, DWORD dwOpt);
其中，返回值是一个枚举类型（EFaultRepRetVal）的常量，其定义如下，我们稍后再详细讨论其含义。参数 dwOpt 用于指定报告方式、对话框界面等选项。
```

```
typedef enum tagEFaultRepRetVal
{
    frrvOk = 0,      frrvOkManifest,     frrvOkQueued,      frrvErr,
    frrvErrNoDW,     frrvErrTimeout,     frrvLaunchDebugger,   frrvOkHeadless
} EFaultRepRetVal;
```

接下来，UnhandledExceptionFilter 函数通过动态取得的函数指针调用 ReportFault 函数。ReportFault 函数读取以下注册表项，以获取当前的错误提示设置，生成记录文件，然后判断当前进程的可执行文件是否是 dwwin.exe 和 dumprep.exe，因为这两个进程都是错误报告机制的成员，所以 ReportFault 如果检测到自己是在这两个进程中时，便会返回退出。

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\PCHealth\ErrorReporting

如果一切顺利，ReportFault 会调用另一个位于 FAULTREP.DLL 中的函数 Start-DWException。StartDWException 首先创建几个用于同步的事件对象，然后在临时目录中生成一个名为 appcompat.txt 的文件（例如，C:\DOCUME~1\%UserID%\LOCALS~1\Temp\WER77A.tmp.dir00\appcompat.txt）。之后，ReportFault 函数取得系统目录，然后生成 DWWIN.EXE 程序的全路径，再调用 CreateProcess 函数在 WinStation 0 的默认桌面(Winsta0\Default)中启动 DWWIN 程序。于是便出现了图 12-4 所示的 Windows XP 风格的应用程序错误对话框。

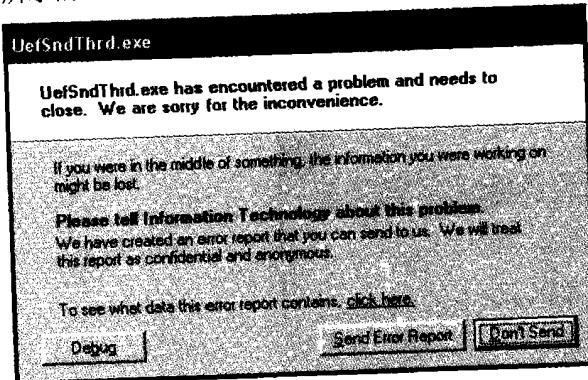


图 12-4 由 DWWIN 程序弹出的应用程序错误对话框（Windows XP 引入）

与前面介绍的 CSRSS 进程弹出的应用程序错误对话框相比，DWWIN 程序弹出的对话框有以下相同点和不同点。

首先，DWWIN 仍然是读取注册表中的 AeDebug 键决定是否包含 Debug 按钮，如果用户选择 Debug 按钮，那么 StartDWException 函数返回 frrvLaunchDebugger(6) 给 ReportFault 函数，ReportFault 函数再将这个值返回给 UnhandledExceptionFilter。UnhandledExceptionFilter 函数再启动 JIT 调试器进程。

第二，新的对话框增加了 Send Error Report（发送错误提示）按钮，如果用户选择该按钮，程序会将错误信息发送给微软公司或本企业的用来收集错误信息的专用服务器。不论错误报告是否发送成功，StartDWException 函数都会返回 frrvOk 给 ReportFault 函数，ReportFault 函数再将这个返回值传递给 UnhandledExceptionFilter。UnhandledExceptionFilter 收到 frrvOk 后，会返回 EXCEPTION_EXECUTE_HANDLER(1)，即执行异常处理块。如果用户选择 Don't Send（不发送）按钮，那么 StartDWException 便直接返回 frrvOk（不调用发送错误报告的函数）。可见，不论是选择 Send Error Report 还是 Don't Send 按钮，ReportFault 函数都会返回 frrvOk，也就是说，这两个按钮的效果都相当于以前的 OK 按钮（对于 UnhandledExceptionFilter 函数）。

另一点不同的是，DWWIN 的对话框中没有直接显示异常信息，如果要了解错误的进一步信息，需要点击对话框上的“click here”链接，查看错误报告。我们将在第 14 章继续介绍这一内容。

当 ReportFault 函数返回的结果表明启动 DWWIN 程序失败(frrvErrTimeout(5)、frrvErrNoDW(4)、没有启动(frrvOkQueued(2)) 或 DWWIN 程序以静默方式工作(frrvOkHeadless(7))时，UnhandledExceptionFilter 会使用旧的方法通过 HardError 机制提示“应用程序错误”（参见图 12-5）。

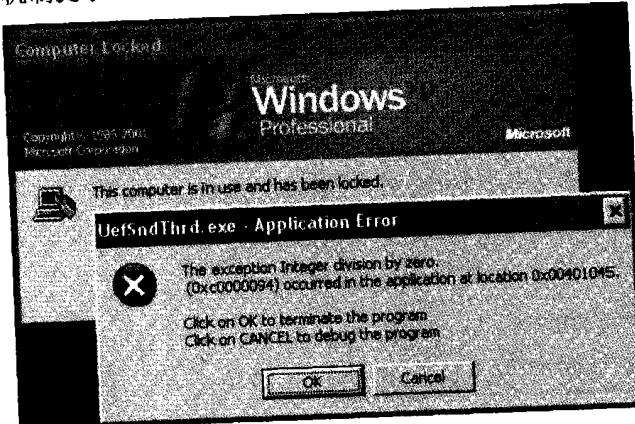


图 12-5 Windows XP 使用旧的方式提示应用程序错误

要说明的一点是，frrvErr(3)的含义是已经成功启动了 DWWIN 程序，所以 UnhandledExceptionFilter 收到该返回值后不会再调用 NtRaiseHardError。表

12-1 归纳了 ReportFault 函数各个返回值的含义及 UnhandledExceptionFilter 函数所采取的动作。

表 12-1 ReportFault 函数的返回值

符号	值	含义	UnhandledExceptionFilter 的动作
frrvOk	0	成功	返回 1**
frrvOkManifest	1	函数成功, DWWIN 程序以清单 (manifest) 模式启动	返回 1
frrvOkQueued	2	函数成功, 错误报告被插入队列, 以后发送	调用 NtRaiseHardError
frrvErr	3	函数失败, 但是启动了 DWWIN 程序	返回 1
frrvErrNoDW	4	函数失败, 没有启动 DWWIN 程序	调用 NtRaiseHardError
frrvErrTimeout	5	函数超时	调用 NtRaiseHardError
frrvLaunchDebugger	6	用户选择了调试按钮	启动 JIT 调试器
frrvOkHeadless	7	函数成功, DWWIN 以静默 (silent) 模式启动	调用 NtRaiseHardError

**UnhandledExceptionFilter 返回 1 的含义是 EXCEPTION_EXECUTE_HANDLER, 即让异常分发函数执行异常处理块 (通常是退出进程)。

Windows XP 允许禁止错误报告发送功能, 既可以在整个系统的范围内禁止错误报告发送功能, 也可以定义把某些或某个程序加入到允许或排除列表中。这时, DWWIN 程序弹出的界面会略有不同, 不再有 Send Error Report 按钮和 Don't Send 按钮, 取而代之的是 Close 按钮 (参见图 12-6)。

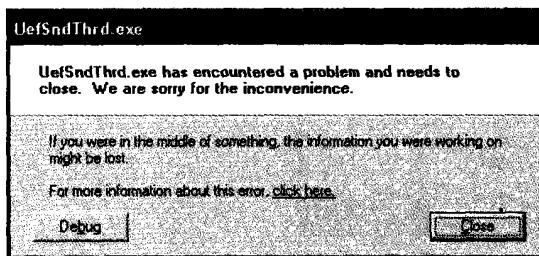


图 12-6 禁止发送错误报告后, DWWIN 程序弹出的应用程序错误对话框

Windows Vista 改为通过新引入的 WER 系统服务来提示应用程序错误。WER 系统服务收到请求后会启动一个名为 WerFault.exe 的程序来显示错误对话框。如果使用 WER 系统服务失败, 那么 UnhandledExceptionFilter 会使用 Kernel32.dll 中的 WER 函数来启动 WerFault.exe。如果这样做也失败, 那么会使用最原始的 HardError 方法。我们将在第 14 章讨论 WER (Windows Error Reporting) 时介绍 WER API, 并介绍如何配置错误报告发送方案和其内部工作过程。

12.5 JIT 调试和 Dr. Watson

所谓 JIT 调试 (Just-In-Time Debugging)，就是指在应用程序出现严重错误后而启动的紧急调试。因为 JIT 调试建立时，被调试的应用程序内已经发生了严重的错误，通常都无法再恢复正常运行，所以 JIT 调试又被称为事后调试 (Postmortem Debugging)。

JIT 调试的主要目的是分析和定位错误原因，或者收集和记录错误发生时的现场数据供事后分析。很多调试器都可以作为 JIT 调试器来使用，比如 WinDBG、CDB、NTSD、Visual C++ IDE (msdev) 和 Dr. Watson 等。其中 Dr. Watson 是 Windows 系统中默认的 JIT 调试器。

12.5.1 配置 JIT 调试器

关于 JIT 调试器的配置信息被保存在注册表的如下表键 (KEY) 中：

HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\AeDebug

因为该表键的键名为 AeDebug (笔者认为 AE 代表的是 Application Error)，所以本书中便将该表键包含的设置选项简称为 AeDebug 选项。

如图 12-7，AeDebug 表键下通常包含 3 个键值：Auto、Debugger 和 UserDebuggerHotKey。先简单介绍如下。

- Debugger 用来定义启动 JIT 调试器的命令行。
- Auto 决定是否自动启动 JIT 调试器。也就是当有未处理异常发生时，是先询问用户，还是直接启动 JIT 调试器。
- UserDebuggerHotKey 用来定义终止到调试器的热键（默认为 F12）。

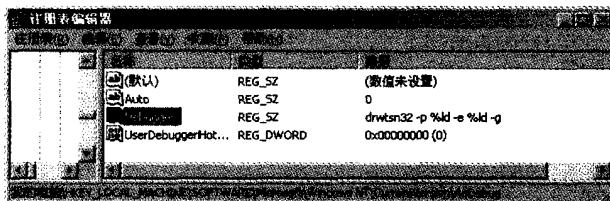


图 12-7 AeDebug 选项

下面我们分别详细讨论每个选项的用途和设置方法。

Debugger 选项用来定义供 UnhandledExceptionFilter 函数启动 JIT 调试器的命令行。如果 UnhandledExceptionFilter 函数成功读到该项，那么，在应用程序错误对话框内就会包含“调试”(Windows XP) 或“取消”按钮 (Windows 2000 或 NT)，供用户选择是否调试发生错误的程序。如果该项不存在或为空，那么弹出的对话框中就不包含调试 (或取消) 按钮 (见图 12-8)。

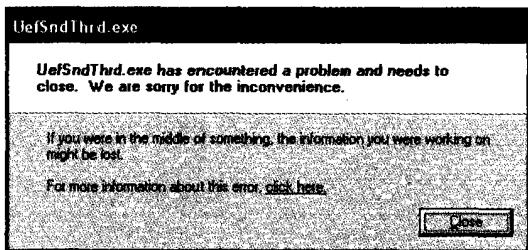


图 12-8 只包含关闭按钮的应用程序错误对话框

尽管 Auto 选项的类型也是 REG_SZ (字符串) 类型, 但是它的有效值只有 0 和 1 两种 (事实上, UnhandledExceptionFilter 只在乎它是否等于 “1”)。如果该值等于 “1”, 而且 Debugger 选项非空, 那么系统 (UnhandledExceptionFilter 函数) 就会直接启动 JIT 调试器。否则, 系统会先显示 “应用程序错误” 对话框, 等用户选择调试选项 (如果存在) 后, 再启动 JIT 调试器。但有一个例外情况是, 对于 Windows XP, 如果 JIT 调试器是 drwtsn32.exe (Dr. Watson), 即使 Auto 项为 1, 系统仍会先显示 “应用程序错误” 对话框, 而且不包含 “调试” 按钮。Auto 选项的默认设置为 “1”。

UserDebuggerHotKey 选项用来定义从被调试程序中断到调试器的热键。大家知道, 使用 WinDBG 调试时, 如果向 WinDBG 输入 (当 WinDBG 在前台) Ctrl+Break, 那么 WinDBG 会将被调试进程中断到 WinDBG 中。其实还有另一种反方向的做法可以达到这个目的, 那就是向被调试程序 (当被调试程序在前台时) 输入调试热键, 使其中断到调试器。默认的调试热键是 F12 (扫描码 0), 通过 UserDebuggerHotKey 选项可以将其设为其他按键, 方法是将该项的内容改为所希望按键的扫描码。表 12-2 列出了标准键盘的扫描码与虚拟键码的对应关系。

表 12-2 标准键盘的扫描码与虚拟键码的对应关系

扫描码	虚拟键码	Scan code	Virtual key code
0x0	VK_SUBTRACT (F12)	0x1	VK_LBUTTON
0x2	VK_RBUTTON	0x3	VK_CANCEL
0x4	VK_MBUTTON	0x8	VK_BACK
0x9	VK_TAB	0xC	VK_CLEAR
0xD	VK_RETURN	0x10	VK_SHIFT
0x11	VK_CONTROL	0x12	VK_MENU
0x13	VK_PAUSE	0x14	VK_CAPITAL
0x15	VK_KANA, VK_HANGEUL, VK_HANGUL	0x17	VK_JUNJA
0x18	VK_FINAL	0x19	VK_HANJA, VK_KANJI
0x1B	VK_ESCAPE	0x1C	VK_CONVERT
0x1D	VK_NONCONVERT	0x1E	VK_ACCEPT
0x1F	VK_MODECHANGE	0x20	VK_SPACE

续表

扫描码	虚拟键码	Scan code	Virtual key code
0x21	VK_PRIOR	0x22	VK_NEXT
0x23	VK_END	0x24	VK_HOME
0x25	VK_LEFT	0x26	VK_UP
0x27	VK_RIGHT	0x28	VK_DOWN
0x29	VK_SELECT	0x2A	VK_PRINT
0x2B	VK_EXECUTE	0x2C	VK_SNAPSHOT
0x2D	VK_INSERT	0x2E	VK_DELETE
0x2F	VK_HELP	0x30-0x39	VK_0-VK_9 (ASCII 0-9)
0x41-0x5A	VK_A-VK_Z (ASCII A-Z)		

关于调试热键还有两点需要说明。首先，只有在当前应用程序已经处于被调试状态下，调试热键才有效。第二，该方法不适用于在命令行窗口运行控制台程序，事实上，当我们向任何命令行窗口输入调试按键时，系统都会以为是要调试 Windows 子系统进程（CSRSS）。

12.5.2 启动 JIT 调试器

下面，我们看看 `UnhandledExceptionFilter` 是如何启动 JIT 调试器的。在第 10 章中，我们讨论了建立调试会话的两种情况，在调试器中创建被调试进程和将调试器附加到已经运行的进程。JIT 调试显然属于后一种，这意味着，`UnhandledExceptionFilter` 函数必须将当前进程（被调试进程）的进程 ID（PID）告诉 JIT 调试器，JIT 调试器才能与其建立调试对话。`UnhandledExceptionFilter` 函数采取的做法是通过命令行参数来传递这一信息。因此，`UnhandledExceptionFilter` 函数要求 `Debugger` 选项的值应该为如下格式：

```
jitdebugger.exe -p %ld -e %ld [调试器的其他参数]
```

其中 `%ld` 是可变域，`UnhandledExceptionFilter` 会使用类似如下的代码填充这些变量域，生成真正的命令行：

```
HANDLE hEvent = CreateEventA( &secAttr, TRUE, 0, 0 );
sprintf(szDbgCmdLine,szDbgCmdFmt,GetCurrentProcessId(), hEvent);
```

也就是说，`UnhandledExceptionFilter` 会把当前进程的进程 ID 和一个同步事件的事件句柄通过命令行参数传递给 JIT 调试器。

接下来，`UnhandledExceptionFilter` 函数会调用 `CreateProcess` 函数来使用格式化好了的命令行启动 JIT 调试器。

`CreateProcess` 成功返回后，`UnhandledExceptionFilter` 调用 `NtWaitForSingleObject` 函数无限期等待 `hEvent` 事件。其目的是给 JIT 调试器足够的时间进行初始化和完成各种准备工作。

当 JIT 调试器成功附加到应用程序并准备就绪后，JIT 调试器就会设置同步事件（SetEvent），这会导致 UnhandledExceptionFilter 函数中的等待函数（NtWaitForSingleObject 或 NtWaitForMultipleObject）返回，接下来 UnhandledExceptionFilter 返回 EXCEPTION_CONTINUE_SEARCH，让系统继续搜索其他异常处理器，如果当前的异常处理器不是最后一个 SEH，那么系统会评估另一个 SEH 的过滤函数，通常也是 UnhandledExceptionFilter 函数，而且也会返回 EXCEPTION_CONTINUE_SEARCH，这样第一轮异常分发便结束了，并且会发起第二轮异常分发。而此时，因为 JIT 调试器已经准备好，所以系统会把异常分发给 JIT 调试器。

Debugger 选项的默认值为“drwtsn32 -p %ld -e %ld -g”，也就是使用 Dr. Watson 作为 JIT 调试器，-g (go) 参数的作用是忽略初始断点。12.7 节将会详细讨论 Dr. Watson 程序。

最后要说明的一点是，如果 JIT 调试器程序文件不在公共路径中，那么应该使用带有完整路径的程序文件名。

很多调试器在安装时会将自己注册为系统的 JIT 调试器，并且提供快捷的方式来将自己注册为 JIT 调试器。例如，如果要将 WinDBG 调试器注册为 JIT 调试器，那么只要在命令行执行 WinDBG-I 即可，其中，I 需要大写并需要指定 WinDBGexe 的完整路径，比如 c:\windbg\windbg -I。执行这个命令后，WinDBG 会将注册表中 AeDebug 表键下的 Debugger 键值修改为如下内容：

```
"c:\windbg\windbg.exe" -p %ld -e %ld -g
```

其中-g 表示忽略初始断点。同时，WinDBG 还会将 Auto 键值改为 1，也就是当有应用程序错误发生时就自动启动 JIT 调试器。

将 WinDBG 注册为 JIT 调试器后，再执行会导致未处理异常的 UEF 小程序，我们就会看到 WinDBG 被自动运行了（见图 12-9）。

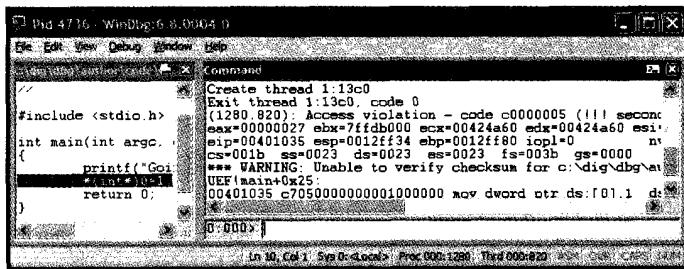


图 12-9 以 JIT 调试器方式启动的 WinDBG

启动后，WinDBG 先显示一系列模块加载信息（图 12-9 中未包含），而后可能显示远程中断线程的创建和退出信息（右侧的第 1、2 行信息）。说“可能”有两个原因，一是需要将这两个事件的处理方式（Debug>Event Filters）设置为输出信息（Output）

才能看到这样的信息。二是远程中断线程的创建和退出时间是不确定的。因为在 WinDBG 的命令行中带有-g 开关，WinDBG 一收到远程中断线程导致的断点事件就会立刻恢复运行，不会报告给用户，所以我们不会看到远程线程的断点，如果希望看到，那么可以去掉-g 开关。

而后，WinDBG 显示接收到非法访问(0xC0000005)的第二轮处理机会(!!! second chance)，即图 12-9 中右侧的第 3 行。接着 WinDBG 会根据调试事件中的信息寻找导致事件的模块和符号信息，因此我们可能需要几秒钟。等待后，WinDBG 会呈现出如图 12-9 所示的界面。在右侧，是调试事件发生时的寄存器取值，最下面一行是触发调试事件（也就是导致非法访问异常）的程序指令：

```
00401035 c70500000000001000000 mov dword ptr ds:[0],1 ds:0023:00000000=????????
```

从这条指令可以看出，向地址 0 写入 1 时导致了非法访问，即典型的访问空指针。

在左侧，WinDBG 显示出了以上指令所对应的源程序文件，并加亮了这条指令所对应的源代码行，也就是我们故意放置的向地址 0 赋值的语句。

通过这个小实验，我们看到了利用 JIT 调试机制，可以很容易地定位到导致应用程序错误的根源。需要说明的一点是，以上实验使用的 UEF.exe 就是在同一台机器上编译出的，因此不需要做任何设置，WinDBG 就能根据 EXE 文件中内嵌的符号信息 (IMAGE_DEBUG_TYPE_CODEVIEW, 第 25 章将详细讨论)找到符号文件(UEF.PDB)，然后再根据符号文件中的源代码行信息找到源文件和错误指令所对应的代码行。如果 EXE 程序不是在同一台机器上编译的，或者符号文件和源文件改变了位置，那么 WinDBG 启动后，可能只显示右侧的内容。这时可以手工设置符号路径 (File>Symbol File Path) 和源文件路径 (File>Source File Path)，设置后执行.reload 命令，WinDBG 也就可以自动打开源文件和加亮错误指令所对应的代码行。

12.5.3 自己编写 JIT 调试器

在了解了 JIT 调试器的基本原理之后，我们可以很容易地将前面设计的小调试器修改为支持 JIT 调试功能的调试器。清单 12-11 显示了修改后的 main() 主要代码。

清单 12-11 JIT 调试器 (JitDbgr) 的核心代码

```

1 void Help()
2 {
3     printf ( "JitDbgr -p <PID of Program to Debug> -e <event handle>\n" ) ;
4 }
5 int main(int argc, char* argv[])
6 {
7     if(argc<=4)
8     {
9         Help();    return -1;
10    }

```

```

11     printf("JitDbgr got parameters: ");
12     for(int i=1;i<argc;i++)
13         printf("%s ",argv[i]);
14     printf("\n");
15
16     if(argc>5)
17     {
18         SetEvent((HANDLE) atoi(argv[4]));
19         return -2;
20     }
21     if(!DebugActiveProcess(atoi(argv[2])))
22     {
23         printf("Failed in DebugActiveProcess() with %d.\n",GetLastError());
24         return -3;
25     }
26     printf("Successfully attached debugger, any key to continue.\n");
27     getchar();
28     SetEvent((HANDLE) atoi(argv[4]));
29     return DbgMainLoop();
30 }

```

从上面的代码可以看到，JIT 调试器在将参数中包含的进程 ID 转变为整数后，便调用 DebugActiveProcess API 来建立调试对话，如果成功，则通过设置参数中指定的事件来通知 UnhandledExceptionFilter 函数可以返回了。

修改注册表中的 AeDebug 选项便可以将我们的 JitDbgr 设为 JIT 调试器了。也就是将 AeDebug 键下的 Debugger 选项设置为：“<示例代码路径>\code\bin\release\jitdbgr.exe -p %ld -e %ld”（如果路径中包含空格，那么需要将路径和程序名用引号包围）。作了如上设置后，当系统内再有某个应用程序出现未处理异常而且需要调试时，系统便会启动 JitDbgr 程序了。为了实验，大家可以执行本章前面给出的 UEF 程序（uef.exe），执行后窗口中应该得到如清单 12-12 所示的内容（//后为注释）。

清单 12-12 JIT 调试举例

```

Going to assign value to null pointer! // UEF 程序在向空指针赋值前打印的信息
// 如果 Auto 为 0，那么会有应用程序错误对话框弹出，请选择调试
JitDbgr got parameters: -p 3280 -e 44
// 未处理异常导致系统启动 JIT 调试器，JitDbgr 程序打印出 UnhandledExceptionFilter 函数传递给
// 它的参数，3280 是 UEF 程序的进程 ID，44 是事件句柄
Successfully attached debugger, any key to continue.
// JitDbgr 使用 DebugActiveProcess API 与 UEF 进程成功建立调试对话
// 按任意键，下面是 JitDbgr 收到并打印出的调试事件
Debug event received from process 3280 thread 3268: CREATE_PROCESS_DEBUG_EVENT.
Debug event received from process 3280 thread 3268: LOAD_DLL_DEBUG_EVENT.
Debug event received from process 3280 thread 3268: LOAD_DLL_DEBUG_EVENT.
Debug event received from process 3280 thread 3268: LOAD_DLL_DEBUG_EVENT.
Debug event received from process 3280 thread 3268: LOAD_DLL_DEBUG_EVENT.
Debug event received from process 3280 thread 528: CREATE_THREAD_DEBUG_EVENT.
Debug event received from process 3280 thread 3268: EXCEPTION_DEBUG_EVENT.
-Debuggee breaks into debugger; press any key to continue.
// 按任意键
Debug event received from process 3280 thread 3268: EXCEPTION_DEBUG_EVENT.
-Debuggee breaks into debugger; press any key to continue.
// 因为导致异常的条件没有消除，所以会循环不断地产生异常，分发异常.....按 Ctrl+C 可以退出

```

本节我们介绍了 JIT 调试的基本原理, JIT 调试器的启动过程, 以及如何使用 WinDBG 作为 JIT 调试器, 关于 WinDBG 调试器的更多内容将在第 30 章介绍。

12.6 顶层异常过滤函数

前面三节我们介绍了 Windows 系统对未处理异常的处理方法。如果应用程序不希望使用这些默认逻辑来处理未处理异常, 那么可以注册一个自己的未处理异常过滤函数, 并通过 SetUnhandledExceptionFilter API 进行注册。因为这个过滤函数只有在有未处理异常发生时才可能被调用, 所以它通常被称为顶层异常过滤函数(Top Level Exception Filter), 本节简称为顶层过滤函数。

12.6.1 注册

SetUnhandledExceptionFilter API 用来注册顶层过滤函数, 这个 API 的原型如下:

```
LPTOP_LEVEL_EXCEPTION_FILTER SetUnhandledExceptionFilter(
    LPTOP_LEVEL_EXCEPTION_FILTER lpTopLevelExceptionFilter);
```

其中参数 lpTopLevelExceptionFilter 用来指定用户自己编写的顶层过滤函数的地址, 这个函数应该具有如下函数原型:

```
LONG WINAPI TopLevelExceptionFilter(struct_EXCEPTION_POINTERS* ExceptionInfo);
```

可以看到, 这个函数原型与系统的 UnhandledExceptionFilter 函数的原型是完全一样的。

在 Windows XP 之前, SetUnhandledExceptionFilter API 的实现非常简单(如清单 12-13 所示), 它先把 BasepCurrentTopLevelFilter 的当前值保存起来作为返回值(EAX), 然后把参数指定的新函数地址赋给全局变量 BasepCurrentTopLevelFilter。BasepCurrentTopLevelFilter 是定义在 KERNEL32.DLL 中的一个全局变量, 其初始值为空, 也就是说, 默认情况下, 系统没有设置顶层过滤函数。

清单 12-13 SetUnhandledExceptionFilter 函数的实现(XP 之前)

```
0:001> uf kernel32! SetUnhandledExceptionFilter
kernel32!SetUnhandledExceptionFilter:
77e7e4f4 8b4c2404    mov    ecx,[esp+0x4]
77e7e4f8 a1b473ed77  mov    eax,[kernel32!BasepCurrentTopLevelFilter (77ed73b4)]
77e7e4fd 890db473ed77 mov    [kernel32!BasepCurrentTopLevelFilter (77ed73b4)],ecx
77e7e503 c20400       ret    0x4
```

为了防止 BasepCurrentTopLevelFilter 变量被轻易篡改, 从 Windows XP 开始, 以上函数的实现不再这样简单, BasepCurrentTopLevelFilter 变量的值也不再是顶层过滤函数地址的原始值, 而是经过编码的值。具体的编码规则是不公开的, Kernel32.DLL 中的 RtlDecodePointer 和 RtlEncodePointer 函数分别用来解码和编码。

12.6.2 C 运行时库的顶层过滤函数

如果当前进程直接或间接使用了微软的 C 运行时库 (MSVCRT.DLL)，那么 C 运行时库在初始化期间会调用 SetUnhandledExceptionFilter 将当前进程的顶层过滤器设置为 msvcrt!__CxxUnhandledExceptionFilter 函数。清单 12-14 所示的栈回溯记录显示了 MSVCRT 模块注册顶层过滤函数的过程。

清单 12-14 MSVCRT 模块注册顶层过滤函数的过程

```
0:000> k
ChildEBP RetAddr
0012f95c 77c26eca kernel32!SetUnhandledExceptionFilter
0012f964 77c37a20 msvcrt!__CxxSetUnhandledExceptionFilter+0xb
0012f970 77c1ea3b msvcrt!_cinit+0x2b
0012fa14 77f5b42c msvcrt!_CRTDLL_INIT+0xec
0012fa34 77f56771 ntdll!LdrpCallInitRoutine+0x14
0012fb30 77f649a8 ntdll!LdrpRunInitializeRoutines+0x32f
0012fc90 77f55349 ntdll!LdrpInitializeProcess+0xe70
0012fd1c 77f75d87 ntdll!LdrpInitialize+0x186
00000000 00000000 ntdll!KiUserApcDispatcher+0x7
```

观察 __CxxUnhandledExceptionFilter 的汇编指令，可以看出它主要是针对 C++ 异常的，清单 12-15 给出了这个函数的伪代码。

清单 12-15 微软 C 运行时库的顶层过滤函数

```
#define CXX_FRAME_MAGIC 0x19930520
#define CXX_EXCEPTION 0xe06d7363
LONG WINAPI CxxUnhandledExceptionFilter (struct _EXCEPTION_POINTERS*
ExceptionInfo)
{
    PEXCEPTION_RECORD pER;

    pER=ExceptionInfo->ExceptionRecord;

    if(pER->ExceptionCode==CXX_EXCEPTION
        && pER->NumberParameters==3
        && pER->ExceptionInformation[0]==CXX_FRAME_MAGIC)
    {
        terminate();
    }
    if(UnDecorator::fGetTemplateArgumentList)
    {
        if(_ValidateExecute(UnDecorator::fGetTemplateArgumentList)!=0)
            return FUNC_UNK(ExceptionInfo);
    }
    return EXCEPTION_CONTINUE_SEARCH;
}
```

因为 VC 编译器实现的 C++ 异常都具有统一的异常代码 0xe06d7363 (即 msc 的 ASCII 代码)，且参数 0 为 0x19930520 (看似日期)，所以 CxxUnhandledExceptionFilter 很容易判断出发生的未处理异常是否是 C++ 异常。如果是，那么 CxxUnhandledExceptionFilter 会调用 VC 运行库自己的 terminate 函数进行必要的清理工作。我们会在第 24 章详细讨论 C++ 异常的细节。

对于其他异常，`CxxUnhandledExceptionFilter` 会返回 `EXCEPTION_CONTINUE_SEARCH`，这会使 `UnhandledExceptionFilter`（`KERNEL32.DLL` 中）继续向下执行，如果 `CxxUnhandledExceptionFilter` 返回 `EXCEPTION_EXECUTE_HANDLER` 或 `EXCEPTION_CONTINUE_EXECUTION`，那么 `UnhandledExceptionFilter` 便会立即返回（参见清单 12-7 的第 26~28 行和清单 12-8 的第 73~75 行）。

12.6.3 执行

在第 12.3 节我们介绍过，系统的 `UnhandledExceptionFilter` 函数会在判断当前进程是否处于被调试状态之后（清单 12-7 的第 17~20 行，清单 12-8 的第 70~76 行），检查全局变量 `BasepCurrentTopLevelFilter` 是否为空，如果不为空，则会调用该指针所指向的函数。因此顶层过滤函数被调用的前两个条件是，有未处理异常发生，而且所在程序不再被调试。

在以上两个条件都满足后，是不是我们注册的顶层过滤函数就一定会被调用呢？答案是否定的。因为系统是使用一个全局变量而不是一个链表来记录顶层过滤函数的，所以，前面注册的地址会被后面注册的所覆盖。这意味着，系统（`UnhandledExceptionFilter` 函数）只会调用最后注册成功的那个顶层过滤函数。

因为应用程序调用 `SetUnhandledExceptionFilter` API 成功后，会返回前一个顶层过滤函数的地址。所以理论上，如果每个顶层过滤函数都记录下前一个过滤函数，并在自己被调用后调用前一个过滤函数，那么每个过滤函数都被调用是有可能的。但是事实上，很多顶层过滤函数都没有这么做，包括我们前面介绍的 C 运行库的过滤函数。为了确保自己的过滤函数能被调用，某些软件使用了非常不好的做法，比如反复调用 `SetUnhandledExceptionFilter` 注册自己的过滤函数以确保其是最后注册的一个，甚至有些软件在自己注册成功后便修改 `SetUnhandledExceptionFilter` API，使其他模块再也无法成功调用这个 API。

需要说明的是，在 Windows XP SP2 和 Windows Vista 等版本中，`SetUnhandledExceptionFilter` 函数会先将要设置的顶层过滤函数地址进行编码，然后再保存到 `BasepCurrentTopLevelFilter` 变量中，`UnhandledExceptionFilter` 函数在调用顶层过滤函数时会先对其进行解码。

12.6.4 调试

关于顶层过滤函数还有一点要说明，就是如何调试顶层过滤函数。从 `UnhandledExceptionFilter` 函数的伪代码可以看到，如果当前进程处于被调试状态，那么 `UnhandledExceptionFilter` 函数会返回 `EXCEPTION_CONTINUE_SEARCH`，让系统把接下来的处理交给调试器。也就是说，这时根本执行不到调用顶层过滤函数的地方。

尽管以上逻辑给调试顶层过滤函数带来了不便，但是笔者认为这样做是有道理的，因为很多顶层过滤函数会做内存清理甚至退出等操作，等调用完这些函数再交给调试器带来的不便也许会更多。但是微软知识库（KB）还是承认这是一个 Bug，并给出了解决建议 (<http://support.microsoft.com/kb/173652/en-us>)。

其实，有一个简单有效的办法来调试顶层过滤函数，那就是仍然事先就把调试器附加到要调试的程序上，当 `UnhandledExceptionFilter` 函数判断 `DebugPort` 是否为空时，通过调试器动态修改标记进程是否在被调试的变量的值。具体来说，`UnhandledExceptionFilter` 函数是调用 `NtQueryInformationProcess` 内核服务来查询当前进程是否在被调试的（`DebugPort` 的值），相关的汇编指令如下：

```
77e99bd6 ff15ac10e677 call dword ptr [kernel32!_imp__NtQueryInformationProcess]
77e99bdc 85c0      test    eax,eax ;检查 NtQueryInformationProcess 的返回值
77e99bde 7c09      j1     kernel32!UnhandledExceptionFilter+0xfe (77e99be9)
77e99be0 3975e0    cmp     [ebp-0x20],esi ;检查 DebugPort 的值，esi 为 0
```

根据上面的代码，通过 `NtQueryInformationProcess` 查询到的 `DebugPort` 值存储在局部变量 `[ebp-0x20]` 中，这个值不为 0，便代表当前进程在被调试，`UnhandledExceptionFilter` 函数就很快返回了。因此，我们只要在执行这个比较指令之前，将其设为 0 便可以了。如果使用的是 WinDBG 调试器，那么可以在这个比较指令的位置设个断点：

```
bp 77e99be0
```

当该断点命中时，再输入如下命令：

```
ed [ebp-20] 0
```

这样，便绕过了下面的检查，可以继续跟踪调用和执行顶层过滤函数的过程了。

12.7 Dr. Watson

Dr. Watson（华生医生）本来是小说《福尔摩斯探案集》中的人物，他是福尔摩斯的得力助手。在 Windows 中，Dr. Watson 是以下几个程序的别称。

- DRWATSON.EXE：16 位版本的 Windows 中收集和记录错误的小工具。今天的 Windows 目录仍然有这个文件（兼容目的）。
- DRWTSN32.EXE：32 位版本的 Dr. Watson 程序。是系统中默认的 JIT 调试器，具有生成错误报告、产生内存转储文件等功能。
- DWWIN.EXE：Windows XP 引入的提示应用程序错误和发送错误报告的工具。Windows XP 中的“应用程序错误”对话框便是由该程序弹出的（参见 12.4 节），该程序还负责通过网络将错误报告发送到服务器，system32\1033 目录下的 dwintl.dll 中包含了 DWWIN 程序所使用的对话框、图标、字符串资源，从这些资

源中可以很容易地发现，很多熟悉的错误报告和收集有关的对话框（包括 OFFICE 程序所专用的错误提示对话框）都来自 DWWIN 程序。

DRWATSON 已经过时，DWWIN 程序的主要目的是提示和报告错误（尽管很多时候也将其称为 Dr. Watson），所以如无特别说明，本书中的 Dr. Watson 是指与调试关系更为密切的 DRWTSN32 程序。本节我们将集中介绍 DRWTSN32 程序。

DRWTSN32 程序的核心功能是当应用程序出现错误时，以 JIT 调试器的身份收集错误信息产生记录文件和错误报告。围绕这一核心功能，DRWTSN32 程序有以下几种运行模式。

12.7.1 配置和察看模式

当不带任何参数直接运行 DRWTSN32 程序时，它会显示出如图 12-10 所示的界面，通过该界面可以查看 DRWTSN32 程序记录下来的工作选项。

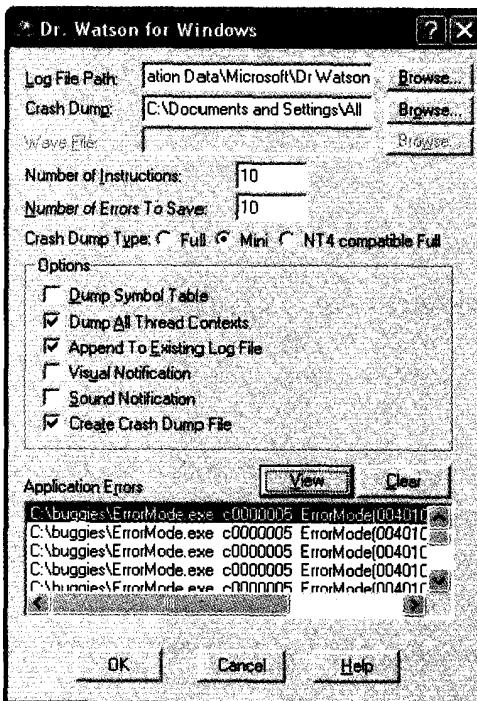


图 12-10 DRWTSN32 程序的配置和察看界面

DRWTSN32 程序的配置信息被存储在注册表的如下表键中：

\HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\DrWatson

表 12-3 列出了各个键值的含义和设置方法。

表 12-3 DRWTSN32 程序的配置选项

键值	类型	含义	默认值
LogFilePath	REG_SZ	存放日志文件 (drwtsn32.log) 的文件夹	默认时没有该键值, 此时的日志文件目录为 Documents and Settings\All Users\Application Data\Microsoft\Dr Watson
CrashDumpFile	REG_SZ	故障转储文件 (dump file)	默认时没有该键值, 此时的故障转储文件为 Documents and Settings\All Users\Application Data\Microsoft\Dr Watson\user.dmp
WaveFile	REG_SZ	声音提示波形文件	默认时没有该键值, 也不播放声音
Instructions	REG_DWORD	指定在日志中为每个线程记录的最多汇编指令条数	10, 即最多包含当前指令的前10条和后10条指令
NumberOfCrashes	REG_DWORD	要记录的最多错误次数(日志文件和系统日志)	10
CrashDumpType	REG_DWORD	故障转储的类型 (1-mini, 2-full, 0-NT4 兼容)	1 (mini)
DumpSymbols	REG_DWORD	是否转储符号表, 选择该项会导致转储文件非常庞大	0
DumpAllThreads	REG_DWORD	转储所有线程的上下文状态 (1), 还是只转储导致错误的线程 (0)	1
AppendToFile	REG_DWORD	附加到现有的日志文件 (1), 还是覆盖现有的日志文件 (0)	0
VisualNotification	REG_DWORD	是否显示图 3-40 所示的提示对话框	0
SoundNotification	REG_DWORD	是否播放声音提示	0
CreateCrashDump	REG_DWORD	是否产生故障转储文件	0

DRWTSN32 程序界面中, 下部的列表框显示了 DRWTSN32 以前产生的错误记录, 选中其中的一行, 然后点击“View”按钮, 可以观察其详细信息。

如果 VisualNotification 项被设置为 1, 那么, 在 DRWTSN32 被作为 JIT 调试器启动后, 它会显示图 12-11 所示的对话框。

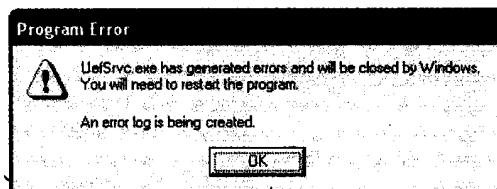


图 12-11 当 VisualNotification 选项为 1 时，DRWTSN32 程序显示的提示对话框

笔者发现，DRWTSN32 程序本身还有 Bug 存在，比如，LogFile Path 选项一旦被设置为其他目录，便无法再设置回默认目录（按 OK 后再启动，看到的仍是旧的）。解决的办法是直接到注册表中将 LogFilePath 键值删除。

12.7.2 安装为 JIT 调试器模式

如果在命令行参数中指定 -i (install) 选项，那么 DRWTSN32 程序只是简单地将自己设置为系统的 JIT 调试器，然后显示出如图 12-12 所示的对话框，按 OK 后 DRWTSN32 程序便退出了。

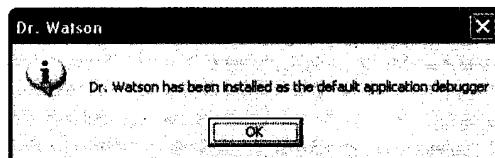


图 12-12 DRWTSN32 程序提示已经成功地将自己设置为默认的 JIT 调试器

观察注册表中的 AeDebug 表项，可以看到 DRWTSN32 程序把 Debugger 键值设为 drwtsn32 -p %ld -e %ld -g，Auto 键值设为 1。

12.7.3 JIT 调试模式

当 DRWTSN32 程序被注册为系统的 JIT 调试器时，如果再有应用程序错误（未处理异常）发生，系统便会自动启动 DRWTSN32 程序。也就是通过如下命令行来启动它：

```
drwtsn32 -p %ld -e %ld -g
```

-p 开关后面跟的是发生错误的进程 ID，-e 开关后面是事件句柄，也就是 UnhandledExceptionHandler 函数（位于 KERNEL32.DLL 中）创建的用于等待 JIT 调试器的事件对象。当 DRWTSN32 成功附加到-p 开关指定的进程并作好了接受调试事件的准备后它会设置此事件，这样 UnhandledExceptionHandler 函数会退出等待状态，返回 EXCEPTION_CONTINUE_SEARCH，让系统的异常分发函数继续工作，将异常送给调试器来处理。-g 参数是调试器常用的一个开关，其含义是让调试器忽略初始断点。通常在调试器附加被调试进程后，会通过初始断点将被调试进程中断到调试器中。

在 JIT 调试器模式下，DRWTSN32 程序会执行以下动作。

第一，如果 VisualNotification 选项为 1，则显示图 12-11 所示的对话框，提示某应用程序内发生错误，系统（DRWTSN32）正在产生错误日志。该对话框刚弹出时，按钮的内容为“取消”，待处理完毕后变为“确定”。

第二，如果 SoundNotification 选项为 1，则播放 WaveFile 选项所指定的波形文件。

第三，向系统日志中写入类似如下内容的记录。使用系统日志观察工具可以观察这些记录（我的电脑>管理，然后选系统工具>事件观察器>应用程序）。

```
Faulting application excel.exe, version 11.0.8012.0, stamp 43e2ab74, faulting module kernel32.dll, version 5.1.2600.1560, stamp 40d1dbc, debug? 0, fault address 0x000138b2.
```

第四，如果 CreateCrashDump 选项为 1，则生成故障转储文件（dump），该文件的路径和文件名是由 CrashDumpFile 选项指定的。默认为 Documents and Settings\All Users\Application Data\Microsoft\Dr Watson\user.dmp。使用 WinDBG 或 Visual Studio 2003（或更高版本）可以打开故障转储文件。使用 MiniDumpReadDumpStream API 也可以读取故障转储文件中的信息。

第五，生成日志文件。文件的名称为 drwtsn32.log，位置是由 LogFilePath 选项定义的，默认值为 Documents and Settings\All Users\Application Data\Microsoft\Dr Watson。如果 AppendToFile 选项为 1，而且 drwtsn32.log 已经存在，那么 DRWTSN32 会将新的记录附加在文件的末尾。否则的话，DRWTSN32 会覆盖现有的文件。下一节，我们将详细介绍 DRWTSN32 程序日志文件的格式和阅读方法。

12.8 DRWTSN32 的日志文件

DRWTSN32 程序的日志文件（drwtsn32.log）是对应用程序错误发生现场的详实记录，对分析和诊断软件问题有着重要的参考价值，特别是在分析发生在用户电脑上无法直接调试的问题时，DRWTSN32 程序的日志文件就更加宝贵，其实这也正是 Windows 系统将其作为默认的 JIT 调试器和最初设计这个程序的目的。

下面我们以一个真实的 drwtsn32.log 文件为例，详细介绍如何阅读和分析 DRWTSN32 程序的日志文件。该文件放在高端调试网站（<http://advdbs.org/books/swdbg/>）的\Data 目录中，建议大家打开该文件（记事本程序便可）对照阅读。在同一目录下还有一个为同一次应用程序错误产生的 user.dmp 文件，大家可以使用 WinDBG 打开该文件（Open Crash Dump）。

做好以上准备工作后，下面我们按照 drwtsn32.log 文件的组成部分来逐一介绍。

12.8.1 异常信息

该部分描述了应用程序的名称（UEF.exe）、进程 ID（3408）、错误（异常）发生时间和导致应用程序错误的未处理异常的异常代码（c0000005）。

```

Application exception occurred:
  App: c:\dig\dbg\author\code\bin\release\UEF.exe (pid=3408)
  When: 2006-6-3 @ 17:30:35.372
  Exception number: c0000005 (access violation)

```

这部分最有价值的信息便是异常代码，通过它我们可以知道“事故”的基本类型。c0000005 是非常常见的一个异常代码，其含义是非法访问（STATUS_ACCESS_VIOLATION）。导致非法访问的情况很多，比如访问空指针、执行特权指令等。SDK 中的 WinNT.H 和 DDK 中的 NTSTATUS.H 中定义了所有异常代码。本书第11章的11.2节的表11-2列出了常见的异常代码。除了 Windows 系统定义的异常代码，编译器和应用程序本身也可以定义自己的异常代码，例如，VC 编译器中所有 C++ 异常的异常代码都是 0xe06d7363（即'msc'的ASCII 代码）。

12.8.2 系统信息

该部分描述了发生应用程序错误的系统的基本软硬件信息，包括 CPU 的型号和数量，Windows 的版本号和 Service Pack 的版本号，以及正在使用的 Windows 内核文件（NTOSKRNL.EXE）的类型（Uniprocessor Free）等。

```

*----> System Information <----*
  Computer Name: ADV_DBG           // 计算机名
  User Name: DBGR                  // 用户名
  Terminal Session Id: 0          // 会话 ID
  Number of Processors: 1          // 处理器数量
  Processor Type: x86 Family 6 Model 9 Stepping 5 // 处理器的类型和版本
  Windows Version: 5.1             // Windows 系统的版本，5.1 即 Windows XP
  Current Build: 2600              // Windows 系统的构建号
  Service Pack: 1                 // Service Pack 号码
  Current Type: Uniprocessor Free // Windows 系统文件的构建选项

```

12.8.3 任务列表

该部分描述了错误发生时系统运行的所有进程（任务）的进程 ID 和可执行文件名称。

```

*----> Task List' <----*
  0 System Process      // IDLE 进程
  4 System               // 系统进程
  1216 smss.exe         // 会话管理器进程
  ....                  // 为节约篇幅，其他从略

```

12.8.4 模块列表

该部分描述了错误发生时进程中所有模块的位置和名称（完整路径）。比如下面的第一行便是 EXE 文件的完整文件名和它的加载地址：00400000-0040c000，这是 VC 编译器为普通 EXE 程序指定的默认加载地址。

```

*----> Module List <----*
(0000000000400000 - 000000000040c000:
c:\dig\dbg\author\code\bin\release\UEF.exe
....      // 为节约篇幅，其他从略

```

12.8.5 线程状态

该部分用于描述错误发生时的线程状态，如果 DumpAllThreads 选项为 1，那么这部分会包含进程内所有线程的信息，否则只包含发生错误的线程。

```
*----> State Dump for Thread Id 0x1128 <----*
```

副标题中包含了线程 ID，下面是当时的寄存器状态：

```
eax=00000027 ebx=7ffd000 ecx=00408088 edx=00000001 esi=00000007 edi=77f944a8
eip=0040100d esp=0012ff84 ebp=0012ffc0 iopl=0 nv up ei pl nz na po nc
cs=001b ss=0023 ds=0023 es=0023 fs=0038 gs=0000 efl=00000206
```

接下来是错误发生位置附近的反汇编结果，通常会包含导致错误的那条指令的前 10 条和后 10 条汇编指令，如果希望观察更多的指令可以通过修改 Instructions 选项，或者产生完整的 dump 文件，然后分析 dump 文件。

```
*** WARNING: Unable to verify checksum for
c:\dig\dbg\author\code\bin\release\UEF.exe
function: UEF!main
.....
00401005 e816000000      call    UEF!printf (00401020)
0040100a 83c404          add     esp,0x4
FAULT ->0040100d c7050000000001000000 mov dword ptr [0],0x1
ds:0023:00000000=???????
00401017 33c0            xor    eax, eax
00401019 c3              ret
.....
```

带有 FAULT -> 标志的是导致错误的那条指令，本例中它是一条赋值指令 (MOV)，其源操作数为 1，目标操作数为指向地址 0 的指针。显然，这是一个典型的访问空指针的操作，会导致非法访问。事实上这正是源代码中向空指针赋值的语句。

```
*(int *)0=1;
```

12.8.6 函数调用序列

这部分信息用来描述事故发生时，线程栈所记录的函数调用序列 (Calling Stack)。

```
*----> Stack Back Trace <----*
*** ERROR: Symbol file could not be found. Defaulted to export symbols for
C:\WINNT\system32\kernel32.dll -
WARNING: Stack unwind information not available. Following frames may be wrong.
ChildEBP RetAddr  Args to Child
0012ff80 00401105 00000001 003715a0 003715b8 UEF!main+0xd (FPO: [2,0,0])
0012ffc0 77e8141a 77f944a8 00000007 7ffd000 UEF!mainCRTStartup+0xb4
0012ffff 00000000 00401051 00000000 78746341 kernel32!GetCurrentDirectoryW+0x44
```

因为从栈回溯生成函数调用序列需要了解每个函数的函数原型和调用规范等信息，所以，如果没有找到合适的调试符号文件，那么日志文件中会包含一或多条“Symbol file could not be found”这样的错误信息。上面“ERROR”行显示的错误信息表示没有找到 kernel32.dll 的符号文件，只能使用 DLL 文件本身的导出信息。接下来的警告信息提醒我们其下的函数调用序列可能是错误的。在这种情况下，我们应该格外谨慎。比如在本例中，函数调用序列的最后一行给出的函数名是 kernel32!GetCurrent-

DirectoryW，这很容易让人误以为是 GetCurrentDirectoryW 函数调用了 mainCRTStartup 函数，但这显然是不可能的。我们前面曾经介绍过，通常是 kernel32.dll 中的启动函数调用编译器生成的入口函数。但因为 kernel32.dll 中的启动函数没有导出，所以这里便使用了最靠近的导出函数。在 WinDBG 中，使用 kv 命令得到的结果如下：

```
0:000> kv
*** Stack trace for last set context - .thread/.cxr resets it
ChildEBP RetAddr Args to Child
0012ff80 00401105 00000001 003715a0 003715b8 UEF!main+0xd (FPO: [2,0,0])
0012ffc0 77e8141a 77f944a8 00000007 7ffd000 UEF!mainCRTStartup+0xb4
0012fff0 00000000 00401051 00000000 00000000 kernel32!BaseProcessStart+0x23 (FPO:
[Non-Fpo])
```

显然，WinDBG 给出的结果是与我们的推测是相符的。事实上，因为 kernel32!BaseProcessStart+0x23 与 kernel32!GetCurrentDirectoryW+0x44 指向的是同一地址，所以我们也不能说上面日志中的信息是错误的，因为对那一行的确切解释应该是 UEF!mainCRTStartup 函数的返回地址是 GetCurrentDirectoryW 函数的地址加上 0x44。

DRWTSN32 的帮助文档中建议通过 _NT_SYMBOL_PATH 环境变量设置调试符号文件的路径。事实上在笔者的机器上，已经将该环境变量设置为：

```
SRV*c:\symbols*http://msdl.microsoft.com/download/symbols
```

这是使用 WinDBG 的典型设置，WinDBG 通过该设置可以找到正确的符号文件，但是 DRWTSN32 程序不能这样，其原因是 DRWTSN32 程序使用 DBGHELP.DLL 不支持以符号服务器的方式搜索符号，改进的办法是将系统目录下（system32）的 DBGHELP.DLL 更新为 WinDBG 目录中的 DBGHELP.DLL。关于函数调用序列的更详细讨论我们将在第 22 章中给出。

12.8.7 原始栈数据

这一部分包含了栈中的少量原始数据，其起点是 ESP（栈指针）寄存器的值（0012ff84）。每一行左侧是地址，中间是数据的十六进制表示，右侧是数据的 ASCII 码表示。因为局部变量通常是在栈中分配的，所以有时可以在原始栈数据中找到局部变量的值。

```
*----> Raw Stack Dump <----*
000000000012ff84 05 11 40 00 01 00 00 00 - a0 15 37 00 b8 15 37 00 ..@.....7...7.
.....
```

通过以上分析，我们可以了解到这次应用程序（UEF.exe）崩溃是由于其 main 函数的入口附近（偏移 13 个字节）做了一次空指针访问，导致了一个非法访问异常。

Windows Vista 引入了新的错误报告和解决方案，不再预装 DRWTSN32.exe 程序，但是如果把以前版本的 DRWTSN32.exe 复制过来，那么它仍能正常工作（参考文献 2）。

12.9 用户态转储文件

简单地说，用户态转储文件（User Mode Dump）就是用于保存应用程序在某一时刻运行状态的二进制文件。为了支持调试，Windows 系统定义了用户态转储文件的格式并提供了 API 来创建和读取用户态转储文件。与描述整个系统状态的系统转储文件相比，用户态转储文件的描述范围仅限于用户进程，二者的格式也是不同的。

在 Windows 的很多文档中，用户态转储文件被称为 MiniDump，意思是小型的转储文件，但这个名字并不总是确切的，因为我们也可能产生包含完整内存数据的非常庞大的转储文件。本节我们将介绍用户态转储文件的文件格式、产生方法及如何读取和使用其中的信息。因为本节只讨论用户态转储文件，所以下文将其简称为转储文件。本节中的示例采用的是 data 目录中的 user.dmp。

12.9.1 文件格式概览

为了方便生成和读取，转储文件的格式非常简单，主要由四个部分组成。第一部分是位于文件的开始处的文件头，它是一个固定格式的数据结构（MINIDUMP_HEADER）。第二部分是目录表，目录表中的每个目录项是一个固定长度的名为 MINIDUMP_DIRECTORY 的结构，用来描述一个数据流。在目录表之后（第三部分）便是一个个的数据流。第四部分是不定数量的内存块。图 12-13 画出了以上三种数据的布局示意图。

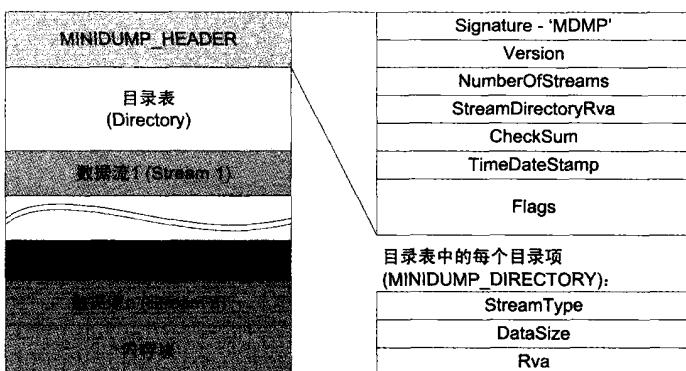


图 12-13 用户态转储文件的布局

WinDBG 和 Windows Platform 的开发工具包（SDK）中包含了描述转储文件的各个数据结构。下面我们分别对其进行介绍。先来看文件头，即 MINIDUMP_HEADER 结构：

```
typedef struct _MINIDUMP_HEADER {
    ULONG32 Signature;           // 文件签名
    ULONG32 Version;            // 版本号
    ULONG32 NumberOfStreams;     // 包含的数据流个数
    RVA StreamDirectoryRva;      // 目录表的偏移 (RVA)
```

```

    ULONG32 CheckSum;           // 校验和
    union {
        ULONG32 Reserved;      // 
        ULONG32 TimeStamp;     // time_t 格式的时间戳
    };
    ULONG64 Flags;             // 标志
} MINIDUMP_HEADER, *PMINIDUMP_HEADER;

```

其中, Signature 字段固定为 0x504D444D, 即 MDMP 的 ASCII 码, MDMP 是 MiniDump 的简写。Version 字段用来标识文件格式的版本号, 低字 (Low-order WORD) 是常量 MINIDUMP_VERSION, 即 0xa793, 高字是与实现有关的, 在 user.dmp 中是 0x5128。NumberOfStreams 字段用来描述文件中所包含的数据流的个数, 因为每个目录项描述一个数据流, 所以它的值也代表了目录表中的目录项个数。StreamDirectoryRva 字段用来指示目录表的起始位置, 即相对于文件头的偏移地址 (RVA)。尽管目前版本的转储文件的文件头是 32 个字节长, 而且 StreamDirectoryRva 的值就是 0x20, 因此可以使用文件头的长度来推算目录表的位置, 但是考虑到文件头的格式因为版本不同可能长度不同, 所以始终应该使用这个字段来定位目录表。Flags 标志用来描述文件信息的类型选项, 每一个二进制位代表一个标志, 枚举类型 MINIDUMP_TYPE 包含了目前已经定义的所有标志位。

目录表中以线性方式存储着目录项, 每个目录项是一个固定的 MINIDUMP_DIRECTORY 结构, 其长度为 12 个字节, 分别是这个目录项所描述数据流的类型 (StreamType)、长度 (DataSize) 和起始地址 (RVA)。

12.9.2 数据流

目录表之后便是数据流, 每个数据流的格式和长度是与类型有关的。表 12-4 列出了目前已经定义的数据流类型, 以及每种数据流的格式。

表 12-4 转储文件的数据流类型

数据流类型	值	格式	备注
ReservedStream0	1	N/A	保留
ReservedStream1	2	N/A	保留
ThreadListStream	3	MINIDUMP_THREAD_LIST	线程列表
ModuleListStream	4	MINIDUMP_MODULE_LIST	模块列表
MemoryListStream	5	MINIDUMP_MEMORY_LIST	内存列表
ExceptionStream	6	MINIDUMP_EXCEPTION_STREAM	异常信息
SystemInfoStream	7	MINIDUMP_SYSTEM_INFO	系统信息
ThreadExListStream	8	MINIDUMP_THREAD_EX_LIST	增强线程列表
Memory64ListStream	9	MINIDUMP_MEMORY64_LIST	内存列表
CommentStreamA	10	单字节 (ANSI) 的字符串, 以 0 结束	注释
CommentStreamW	11	以 0 结束的宽字节字符串	注释
HandleDataStream	12	MINIDUMP_HANDLE_DATA_STREAM	句柄数据

续表

数据流类型	值	格式	描述
FunctionTableStream	13	MINIDUMP_FUNCTION_TABLE_STREAM	函数表
UnloadedModuleListStream	14	MINIDUMP_UNLOADED_MODULE_LIST	卸载模块表
MiscInfoStream	15	MINIDUMP_MISC_INFO	零散信息
MemoryInfoListStream	16	MINIDUMP_MEMORY_INFO_LIST	内存块信息
ThreadInfoListStream	17	MINIDUMP_THREAD_INFO_LIST	线程信息
HandleOperationListStream	18	MINIDUMP_HANDLE_OPERATION_LIST	句柄操作记录
LastReservedStream	0xffff	MINIDUMP_USER_STREAM	用户数据

表中第二列为每个数据流的类型 ID，其中，类型 1~18 是系统支持的数据流类型，我们将其简称为系统类型。除了系统类型，用户也可以通过注册回调函数来将其他格式的信息写入到转储文件中（稍后介绍），用户类型的 ID 值应该从 LastReservedStream 开始，比如 LastReservedStream+1、LastReservedStream+2 等。

12.9.3 产生转储文件

尽管利用我们前面介绍的文件格式和数据结构可以自己调用文件 I/O 函数（如 WriteFile）来产生转储文件，但这是不必要的，因为系统已经为我们提供了一个 API 来做这件事，即 MiniDumpWriteDump。

```
BOOL MiniDumpWriteDump( HANDLE hProcess, DWORD ProcessId,
    HANDLE hFile, MINIDUMP_TYPE DumpType,
    PMINIDUMP_EXCEPTION_INFORMATION ExceptionParam,
    PMINIDUMP_USER_STREAM_INFORMATION UserStreamParam,
    PMINIDUMP_CALLBACK_INFORMATION CallbackParam);
```

其中，hProcess 和 ProcessId 用来指定转储文件所描述的进程，hFile 是转储文件的句柄，通常是调用 CreateFile API 所创建的。DumpType 用来指定写入信息的类型和选项，表 12-5 列出了目前已经定义的选项和它们的含义。

表 12-5 转储文件的类型标志

标志（常量）	取值（位）	含义
MiniDumpNormal	0x00000000	普通
MiniDumpWithDataSegs	0x00000001	包含数据段
MiniDumpWithFullMemory	0x00000002	包含完整的内存
MiniDumpWithHandleData	0x00000004	包含句柄数据
MiniDumpFilterMemory	0x00000008	做过过滤处理，去除私有信息
MiniDumpScanMemory	0x00000010	做过扫描处理以包含引用内存
MiniDumpWithUnloadedModules	0x00000020	包含系统维护的卸载模块
MiniDumpWithIndirectlyReferencedMemory	0x00000040	包含未直接引用的内存
MiniDumpFilterModulePaths	0x00000080	过滤掉模块路径中的私有信息

续表

标志(常量)	取值(位)	含义
MiniDumpWithProcessThreadData	0x00000100	包含完成进程和线程信息
MiniDumpWithPrivateReadWriteMemory	0x00000200	包含更多类型的内存数据
MiniDumpWithoutOptionalData	0x00000400	不包含可选数据
MiniDumpWithFullMemoryInfo	0x00000800	包含内存区信息
MiniDumpWithThreadInfo	0x00001000	包含线程状态信息
MiniDumpWithCodeSegs	0x00002000	包含所有代码段和有关的内存段
MiniDumpWithoutAuxiliaryState	0x00004000	不使用辅助的数据收集器
MiniDumpWithFullAuxiliaryState	0x00008000	使用所有辅助的数据收集器

通过表 12-5，我们了解到转储文件可以包含丰富的信息。不过有必要说明的是，以上某些选项是与系统的版本，特别是用于实现转储文件功能的 DNGHELP.DLL 模块的版本有关的，例如，MiniDumpWithUnloadedModules 开始的选项都需要 DBGHELP 模块的版本不能低于 5.1，MiniDumpWithThreadInfo 开始的选项要求 DBGHELP 模块的版本不能低于 6.1。WinDBG 工具包所附带的 DNGHELP.DLL 模块通常比 Windows 系统所附带的版本更新。

参数 UserStreamParam 用来指定希望写入到转储文件中的用户数据，它是一个 MINIDUMP_USER_STREAM_INFORMATION 结构，用来描述一个数组，每个数组元素是一个 MINIDUMP_USER_STREAM 结构，用来指定每个用户数据流的长度和内容(缓冲区地址)。

参数 CallbackParam 用来指定一个回调函数，以了解和进一步定制产生转储文件的过程。

根据 MSDN 的介绍，MiniDumpWriteDump API 对操作系统的最低要求是 Windows XP 或 Server 2003。如果希望在更早的 Windows 系统中使用这个 API，那么需要复制或安装 DBGHELP.DLL 文件。

12.9.4 读取转储文件

MiniDumpReadDump API 用来读取用户态转储文件，其原型如下：

```
BOOL MiniDumpReadDumpStream( PVOID BaseOfDump, ULONG StreamNumber,
    PMINIDUMP_DIRECTORY* Dir, PVOID* StreamPointer, ULONG* StreamSize);
```

在调用这个 API 前，应该先调用 OpenFile 或 CreateFile 打开要读取的转储文件，然后使用 CreateFileMapping API 创建一个文件映射对象，再调用 MapViewOfFile API 将文件映射到当前进程的内存空间中，而后把 MapViewOfFile 函数返回的基址用作调用 MiniDumpReadDumpStream 函数的第一个参数，即 BaseOfDump。这样做的好处是，不用在调用 MiniDumpReadDumpStream 时为要返回的内容分配内存，只返回要读取内容的虚拟地址就可以了。以下是打开一个转储文件的简化代码。

```

HANDLE hDumpFile=CreateFile(lpszFileName,
    GENERIC_READ, FILE_SHARE_READ, NULL, OPEN_EXISTING, 0, NULL);
HANDLE hDumpMapFile=CreateFileMapping(m_hDumpFile,
    NULL, PAGE_READONLY, 0, 0, NULL);
PVOID pBaseofDump=MapViewOfFile(m_hDumpMapFile, FILE_MAP_READ, 0, 0, 0);
PMINIDUMP_HEADER pMdpHeader=(PMINIDUMP_HEADER)m_pBaseofDump;

```

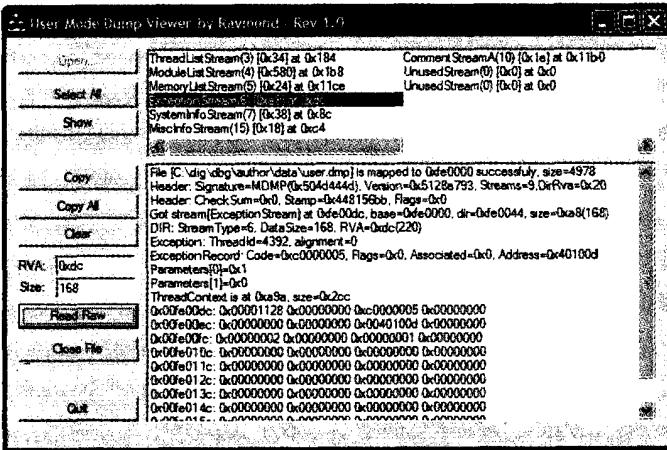
而后便可以这样调用 MiniDumpReadDumpStream:

```

PMINIDUMP_DIRECTORY pMdpDir=0;
PVOID pStream=0; ULONG ulStreamSize=0;
HRESULT hRet=MiniDumpReadDumpStream(pBaseofDump,nStreamType,
    &pMdpDir, &pStream, &ulStreamSize);

```

为了演示以上过程，我们编写了一个名为 `MdmpView` 的小程序（`code\chap12\udmpview`），它全面地演示了读取转出文件的过程。图 12-14 显示了使用这个小工具读取我们上一节介绍的 `user.dmp` 文件的情景。



清单 12-16 所示的信息。

清单 12-16 WinDBG 显示的转储文件信息

```
Loading Dump File [C:\dig\dbg\author\data\user.dmp]
User Mini Dump File: Only registers, stack and portions of memory are available
Comment: 'Dr. Watson generated MiniDump'
...[省略关于符号搜索路径和转储产生时间的若干行]
This dump file has an exception of interest stored in it.
The stored exception information can be accessed via .ecxr.
(d50.1128): Access violation - code c0000005 (first/second chance not available)
eax=00000027 ebx=7ffd000 ecx=00408088 edx=00000001 esi=00000007 edi=77f944a8
eip=0040100d esp=0012ff84 ebp=0012ffc0 iopl=0 nv up ei pl nz na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=0038 gs=0000
*** WARNING: Unable to verify checksum for UEF.exe
UEF+0x100d:
0040100d c7050000000001000000 mov dword ptr ds:[0],1 ds:0023:00000000=????????
```

第 3 行显示的便是注释 (CommentStreamA) 数据流的内容，其内容告诉我们这个转储文件是 Dr. Watson 程序所产生的。第 5 行开始显示了转储文件中所包含的异常数据流 (ExceptionStream)，这看起来与调试器收到异常的情景类似。转储文件通常是在应用程序因为未处理异常而被系统关闭前产生的，因此，分析转储文件中所记录的异常信息对了解应用程序崩溃的原因非常重要。对于本例，从异常代码那一行可以看到应用程序 (进程号 0xd50) 的 0x1128 号线程内发生了访问违例异常。从寄存器信息中的 EIP 值可以知道导致异常的代码位置是 0x40100d，因为应用程序的 EXE 文件通常被加载在进程中 0x400000 开始的位置，所以出现这个问题的指令可能是属于 EXE 模块的，倒数第 2 行中 WinDBG 帮我们定位到的符号位置 (UEF+0x100d) 可以证实这一点。其中，UEF 是 EXE 模块名，+0x100d 表示发生异常的位置相对于 UEF 模块基地址的偏移是 0x100d。最后一行是导致异常的指令，我们前面介绍过，这是在向地址 0 写入常量 1。至此，我们可以判断出这个转储文件所对应的应用程序因为 EXE 模块中执行了空指针访问而导致了异常。

那么，如何进一步定位异常的发生位置呢？比如在哪个函数里？这需要符号文件的帮助，因此，我们应该将 UEF 程序的符号文件所在的目录（同样是 data 目录）设置到 WinDBG 的符号路径中 (File>Symbol File Path...), 然后执行.reload 命令。

```
0:000> .reload
.....
*** WARNING: Unable to verify checksum for UEF.exe
```

上面的警告信息可以忽略，接下来可通过.ecxr 命令让 WinDBG 重新显示异常信息：

```
0:000> .ecxr
...
UEF!main+0xd:
```

这次，WinDBG 便给出了更确切的异常发生位置，即距离 UEF 模块的 main 函数入口偏移 0xd 字节的位置。0xd 是个较小的偏移，于是我们可以断定异常是发生在 main 函数的入口附近的。

那么能否定位到异常对应的源代码呢？答案是可能的。要做到这一点，需要符号文件中包含源代码行信息，通常是调试版本的符号文件才包含这个信息，以上转储文件对应的程序文件是发布版本的，它的符号文件中不包含源代码行信息，所以做不到这一点。如果是调试版本的，那么设置好源文件路径后，再执行.ecxr命令或.reload命令，WinDBG便会自动打开对应的源程序文件并加亮对应的代码行。

下面再介绍如何使用WinDBG观察转储文件中的其他信息。首先可以使用!cpuid命令观察系统信息（SystemInfoStream）中的处理器信息，例如：

```
0:000> !cpuid
CP F/M/S Manufacturer
6,9,5 GenuineIntel
```

可以使用~命令来显示线程信息，并通过~<线程号>s命令切换当前线程。

```
0:000> ~
. 0 Id: d50.1128 Suspend: -1 Teb: 7ffd000 Unfrozen
```

另外，使用lm命令可以显示模块信息，使用kv命令可以观察栈回溯信息，使用!handle命令可以观察句柄信息（需要转储文件中包含HandleDataStream），在此不再详细介绍。

12.10 本章总结

本章比较详细地介绍了Windows系统对未处理异常的处置方法和基础设施。这些内容不仅与应用程序的开发和调试有关，而且与普通用户的使用体验也关系密切。下一章我们将深入介绍用于报告系统中严重错误（包括未处理异常）的HardError机制，以及报告系统崩溃的蓝屏机制。

参考文献

1. Matt Pietrek. A Crash Course on the Depths of Win32™ Structured Exception Handling. Microsoft System Journal, 1997
2. Dmitry Vostokov. Resurrecting Dr. Watson on Vista. <http://www.dumpanalysis.org/>



硬错误和蓝屏

“人非圣贤，孰能无过”，这句古语是说人总是有可能犯错的。把这句话用在软件上也很恰当，任何软件都可能因为自身设计缺欠或外部环境变化等而发生错误。既然错误是不可避免的，那么制定完善的错误处理机制显然要比试图避免发生错误更明智。“我们的软件不能有任何错误，因此也不需要花时间设计什么错误处理机制”，类似这样的思想害了很多软件。

错误处理是项复杂的任务，迄今没有也永远不会有万能的单一解决方案可以应对所有软件的所有错误。必须根据软件产品的具体特点和用户需求设计相应的错误处理方案。一套好的错误处理方案通常应该考虑以下 3 个方面。

- **即时提示 (instant notification)**。即当错误情况需要立刻提示给用户时，将错误情况以可靠的方式、以用户可以理解的语言及时提示给用户。
- **永久记录 (persistent recording)**。即当错误情况满足需要永久记录的条件时，将错误描述永久记录在文件或数据库中供事后分析。
- **自动报告 (automatic reporting)**。即自动收集错误现场的详细情况并生成错误报告，让用户可以通过简单的方式（比如网络）发送到专门用来收集错误报告的服务器。从 Windows XP 开始，Windows 引入了一整套设施来实现这一目标，称为 Windows Error Reporting，简称为 WER。

从本章开始的 4 章将分别介绍 Windows 操作系统的错误提示机制（第 13 章）、WER（第 14 章）和两种错误记录机制（第 15 和 16 章）。

本章先介绍用于报告严重错误的硬错误机制（HardError）（第 13.1 节）和蓝屏机制（BSOD）（第 13.2 节）。然后介绍系统转储文件的产生方法（第 13.3 节）和分析方法（第 13.4 节）。第 13.5 节将介绍声音和闪动窗口等辅助的错误提示机制，第 13.6 节介绍如何配置错误提示机制。第 13.7 节将介绍使用错误提示机制时应该注意的问题。

13.1 硬错误提示

消息对话框（message box）是 Windows 中最常见的即时错误提示方法。利用系统

提供的 MessageBox API，弹出一个图形化的消息框对程序员来说真是唾手可得。而且不论是程序本身带有消息循环的 Win32 GUI 程序，还是用户代码中根本没有消息循环的控制台程序，都可以调用这个 API。

使用 MessageBox API 来实现错误提示的优点是简单易用，但是这种方法存在如下局限。首先，MessageBox 是一个用户态的 API，内核代码无法直接使用。第二，MessageBox 是工作在调用者（通常是错误发生地）的进程和线程上下文中的，如果当前进程/线程的数据结构（比如消息队列）已经由于严重错误而遭到损坏，那么 MessageBox 可能无法工作。第三，对于系统启动关闭等特殊情况，MessageBox 是无法工作的。

为了适应复杂（恶劣）环境下的错误提示需要，Windows 定义了一种比 MessageBox 更复杂也更强大的错误提示机制，称为硬错误（Hard Error）提示。

Hard Error 的本意是指与硬件有关的严重错误，是与重新启动便可以恢复的软件错误（Soft Error）相对而言的。逐渐地，这个词被用来泛指比较严重的错误情况。

13.1.1 缺盘错误

下面先举一个常见的 Hard Error 实例，让大家有个感性的认识。图 13-1 所示的对话框是大家熟悉的缺盘错误对话框，它是因为光盘上的安装程序（setup.exe）正在运行时我们把光盘取出而导致的。这个对话框便是使用硬错误提示机制弹出的。

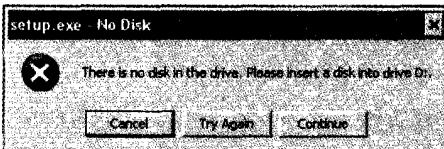


图 13-1 因缺盘而导致的 Hard Error 对话框

尽管这个对话框的外观和消息框是一样的，而且事实上它就是使用 MessageBox API 弹出的。但是调用 MessageBox API 只是硬错误提示机制的一小部分，此前还经历了很复杂的发起和分发过程。

硬错误提示既可以在用户态使用，也可以在内核态使用。在用户态使用的方法是调用 NtRaiseHardError 内核服务。

13.1.2 NtRaiseHardError.

在第12章我们介绍未处理异常和 UnhandledExceptionFilter 函数时曾经提到过 NtRaiseHardError 服务，并且提到了即使在系统没有登录时也可以使用它提示应用程序错误（见图 12-3）。下面我们仔细看看它的函数原型：

```
NTSYSAPI NTSTATUS NTAPI
NtRaiseHardError( /* ExRaiseHardError 和 ExpRaiseHardError 的原型也如此 */
    IN NTSTATUS           ErrorStatus,
    IN ULONG              NumberOfParameters,
```

```

IN ULONG                      UnicodeStringParameterMask,
IN PVOID *                    Parameters,
IN HARDERROR_RESPONSE_OPTION ResponseOption,
OUT PHARDERROR_RESPONSE      Response );

```

其中, `ErrorStatus` 参数用来传递错误代码, 它的值通常是定义在 `NTSTATUS.H` 中的常量。`NumberOfParameters` 指定了 `Parameters` 指针数组所包含的指针个数。`UnicodeStringParameterMask` 参数的各个二进制位与 `Parameters` 指针数组一一对应, 如果某一位为 1, 则说明对应的参数指针指向的是一个 `UNICODE_STRING`, 否则是个整数。`ResponseOption` 参数的作用与 `MessageBox` API 的 `uType` 参数很类似, 用来定义错误消息的按钮个数、响应方式等选项, 其类型类似如下形式的枚举常量:

```

typedef enum _HARDERROR_RESPONSE_OPTION {
    OptionAbortRetryIgnore,   OptionOk,     OptionOkCancel,
    OptionRetryCancel,       OptionYesNo,   OptionYesNoCancel,
    OptionShutdownSystem,   OptionOkNoWait, OptionCancelTryContinue
} HARDERROR_RESPONSE_OPTION, *PHARDERROR_RESPONSE_OPTION;

```

`Response` 参数用来返回错误提示的响应结果, 这个结果可能是用户选择的, 也可能是因为超时而导致的默认值, 或者是处理失败, 它的值为如下常量之一:

```

typedef enum _HARDERROR_RESPONSE {
    ResponseReturnToCaller,   ResponseNotHandled,
    ResponseAbort,           ResponseCancel, ResponseIgnore,
    ResponseNo,              ResponseOk,     ResponseRetry,
    ResponseYes
} HARDERROR_RESPONSE, *PHARDERROR_RESPONSE;

```

`ExRaiseHardError` 和 `NtRaiseHardError` 的工作主要是参数检查和预处理, 在它们把所有错误信息都复制到一个用户态可访问的内存结构中后便调用 `ExpRaiseHardError`。

`NtRaiseHardError` 内部会对参数信息进行预处理, 而后便调用内核中用于实现硬错误提示机制的核心函数 `ExpRaiseHardError`。

13.1.3 ExpRaiseHardError

`ExpRaiseHardError` 函数与 `NtRaiseHardError` 有着相同的函数原型, 它是分发硬错误的枢纽。首先, 对 `ResponseOption` 等于 `OptionShutdownSystem` (关机) 的请求, `ExpRaiseHardError` 会检查调用者是否具有 `SeShutdownPrivilege` 权限, 如果没有, 则返回错误 (0xc0000061)。

而后 `ExpRaiseHardError` 会检查全局变量 `nt!ExReadyForErrors`, 如果该变量等于 0 (FALSE), 那么表示用户态的 HardError 提示系统还没有准备好, 不能向其发送提示请求, `ExpRaiseHardError` 进一步检查 `ErrorStatus`, 看其代表的是否是一个错误, 如果是, 而且当前线程的 `ETHREAD` 结构的 `HardErrorsAreDisabled` 标志 (`CrossThreadFlags` 的一位) 为 0 (即没有禁止 Hard Error), 那么 `ExpRaiseHardError` 便调用 `ExpSystemErrorHandler` 在内核态处理和提示这个 Hard Error。

`ExpSystemErrorHandler` 函数是以蓝屏的形式来提示 Hard Error 的。它在准备好蓝屏所需的参数后, 便调用 `KeBugCheckEx` 或 `PoShutdownBugCheck` 函数来启动蓝屏, 蓝

屏的停止码 (Stop Code) 为 FATAL_UNHANDLED_HARD_ERROR (0x0000004C)，停止码的第一个参数就是要提示 Hard Error 的状态码。下一节我们会进一步讨论与蓝屏有关的问题。

接下来，`ExpRaiseHardError` 开始按如下规则寻找用来发送 Hard Error 的错误端口 (`ErrorPort`)，并将寻找结果赋给一个指针变量。

首先，如果当前进程的 `EPROCESS` 结构的 `DefaultHardErrorProcessing` 字段的位 0 等于 1 (即当前进程的 `ErrorMode` 设置不包含 `SEM_FAILCRITICALERRORS`)，那么便使用 `EPROCESS` 结构的 `ExceptionPort` 字段所指定的端口作为错误端口，如果 `ExceptionPort` 字段的值为空，那么便使用全局变量 `nt!ExpDefaultErrorPort` 所指定的端口。在一个典型的 Windows 系统中，每个 Windows 进程的 `ExceptionPort` 字段和全局的 `ExpDefaultErrorPort` 变量的值是相同的，代表的都是 `CSRSS` 进程的 `\Windows\ApiPort` 端口。

```
kd> dd nt!ExpDefaultErrorPort 11
8054ed04 e13774a0
kd> !lpc port e13774a0 //观察 LPC 端口对象
Server connection port e13774a0 Name: ApiPort
Handles: 1 References: 57
Server process : 815c7020 (csrss.exe) //Windows 子系统服务进程
Queue semaphore : 816a9158
Semaphore state 0 (0x0)
The message queue is empty
The LpcDataInfoChainHead queue is empty
```

第二，如果 `DefaultHardErrorProcessing` 字段的位 0 等于 0，说明当前进程的 `ErrorMode` 设置禁止了 Hard Error 提示 (包含 `SEM_FAILCRITICALERRORS` 标志位)，那么 `ExpRaiseHardError` 会检查 `ErrorStatus` 参数中是否设置了 `HARDERROR_OVERRIDE_ERRORMODE` 标志。`ErrorStatus` 是标准的 NT 状态码格式，`HARDERROR_OVERRIDE_ERRORMODE` 标志使用的是 NT 状态码的保留位 (位 28)。如果 `ErrorStatus` 中包含 `HARDERROR_OVERRIDE_ERRORMODE` 标志 (位 28 为 1)，那么说明这是一个不受 `ErrorMode` 设置限制的错误，所以 `ExpRaiseHardError` 会忽略该设置，如果当前进程的 `ExceptionPort` 不为空，则将其作为错误端口，如果为空，则使用 `ExpDefaultErrorPort` 端口。

第三，检查当前线程的 `ETHREAD` 结构的 `HardErrorsAreDisabled` 标志是否为 1，如果为 1，则将错误端口设为空。这说明线程的 `HardErrorsAreDisabled` 标志具有比 `DefaultHardErrorProcessing` 更高的优先级。

经过以上过程，如果代表查找结果的错误端口指针为空，那么 `ExpRaiseHardError` 函数便将 `Response` (返回型参数) 结果赋值为 `ResponseReturnToCaller`，然后返回成功。如果错误端口指针不为空，那么 `ExpRaiseHardError` 会检查当前进程是否是全局变量 `nt!ExpDefaultErrorPortProcess` 所代表的进程。`ExpDefaultErrorPortProcess` 代表的是监听 `ExpDefaultErrorPort` 端口的进程，如果当前进程就是 `ExpDefaultErrorPortProcess` 进程，那么说明负责提示 HardError 的进程本身出了错误，所以再向其发送错误提示请求可能是不可靠的。对于这种情况，

`ExpRaiseHardError` 会调用 `ExpSystemErrorHandler` 来通过蓝屏提示错误。

使用 WinDBG 观察 `ExpDefaultErrorPortProcess` 变量，可以发现它的值就是 `CSRSS` 进程的 `EPROCESS` 结构的地址，也就是说，`CSRSS` 是系统中默认的硬错误提示进程。

```
kd> dd nt!ExpDefaultErrorPortProcess 11
8054ed08 815c7020
kd> dt _eprocess 815c7020
ntdll!_EPROCESS
...
+0x0c0 ExceptionPort : (null) //普通进程的这个地段不为空
...
+0x174 ImageFileName : [16] "csrss.exe" //Windows 子系统服务进程
...
```

如果当前进程不是 `ExpDefaultErrorPortProcess` 进程，那么 `ExpRaiseHardError` 便调用 `LpcRequestWaitReplyPort` 函数将 Hard Error 信息发送到这个端口。因为 `LpcRequestWaitReplyPort` 函数是同步的，所以 `ExpRaiseHardError` 会一直等待应答。如果 `LpcRequestWaitReplyPort` 成功返回，`ExpRaiseHardError` 会将答复消息中的响应值赋给 `Response` 参数。

13.1.4 CSRSS 中的分发过程

通常，Windows 子系统进程 `CSRSS` 负责监听提示 Hard Error 的 LPC 端口，该端口的名字其实就是 `\Windows\ApiPort`。前面我们介绍过，在 Windows XP 以前，调试事件默认也是发到这个端口的。Windows XP 使用专门的 `DebugObject` 来传递调试事件，但因为这个 LPC 端口还有其他用途，比如它是普通 Windows 进程的默认异常接收端口（`ExceptionPort`）；它是我们正在介绍的 Hard Error 的传递端口；另外它还是 `CSRSS` 对外服务的窗口，用来接收各种其他服务请求，所以它依然存在。

清单 13-1 显示了 `CSRSS` 进程的工作线程处理硬错误提示的过程。

清单 13-1 Hard Error 在用户态（`CSRSS`）中的处理过程

```
0:004> kn
# ChildEBP RetAddr
00 0069fdac 75b7bf38 USER32!MessageBoxTimeoutW
01 0069fe80 75b7c11e winsrv!HardErrorHandler+0x2d1
02 0069fea0 75b7cf11 winsrv!ProcessHardErrorRequest+0x99
03 0069fec0 75b7cf40 winsrv!UserHardErrorEx+0x232
04 0069fed0 75b44545 winsrv!UserHardError+0xf
05 0069fff4 00000000 CSRSRV!CsrApiRequestThread+0x355
```

其中，最下面的（栈帧#05）`CSRSRV!CsrApiRequestThread` 是 `CSRSS` 进程中专门监听 `\Windows\ApiPort` 端口的工作线程，它负责接收、分发和回复发到 `\Windows\ApiPort` 端口的 LPC 消息。每个 LPC 消息的开头都是一个 `PORT_MESSAGE` 结构：

```
typedef struct _PORT_MESSAGE {
    USHORT DataSize;           // 数据长度
    USHORT MessageSize;        // 消息的总长度
    USHORT MessageType;        // 消息类型
```

```

USHORT VirtualRangesOffset; // 数据的偏移
CLIENT_ID ClientId; // 服务请求者(客户)的进程和线程 ID
ULONG MessageId; // 消息 ID
ULONG SectionSize; // 
// UCHAR Data[]; // 与消息类型相关的用户数据
} PORT_MESSAGE, *PPORT_MESSAGE;

```

其中消息类型字段 (MessageType) 是一个枚举类型的常量:

```

typedef enum _LPC_TYPE {
    LPC_NEW_MESSAGE, // 新的消息
    LPC_REQUEST, // 请求服务的消息
    LPC_REPLY, // 对请求消息的回复
    LPC_DATAGRAM, // 数据报(Datagram)消息
    LPC_LOST_REPLY, // 与正等待消息不匹配的回复消息
    LPC_PORT_CLOSED, // 端口关闭消息
    LPC_CLIENT_DIED, // 向线程的终止端口发送的消息
    LPC_EXCEPTION, // 向异常端口发送的消息
    LPC_DEBUG_EVENT, // 向调试端口发送的消息
    LPC_ERROR_EVENT, // 提示硬错误的消息
    LPC_CONNECTION_REQUEST // 用于建立连接的消息
} LPC_TYPE;

```

CsrApiRequestThread 根据消息类型字段来分发消息，在它检测到类型为 LPC_ERROR_EVENT 的消息后，它会先将消息结构强制转换为 HARDERROR_MSG 类型:

```

typedef struct _HARDERROR_MSG {
    LPC_MESSAGE LpcMessageHeader;
    NTSTATUS ErrorCode;
    ErrorTime;
    HARDERROR_RESPONSE_OPTION ResponseOption;
    HARDERROR_RESPONSE Response;
    ULONG NumberOfParameters;
    PVOID UnicodeStringParameterMask;
    ULONG Parameters[MAXIMUM_HARDERROR_PARAMETERS];
} HARDERROR_MSG, *PHARDERROR_MSG;

```

然后，HARDERROR_MSG 枚举进程中已经注册的所有服务模块，查询它是否设置了 HardErrorRoutine 指针。如果一个模块的 HardErrorRoutine 指针不为空，那么便调用该函数。如果函数的返回值不等于 0 (0 代表没有处理该消息)，那么便中止，否则继续寻找下一个服务模块中的 HardErrorRoutine。

在典型的 Windows XP 系统中，CSRSS 进程通常有几个服务模块：BASESRV、WINSRV 和 CSRSRV，但输出 HardErrorRoutine 的只有 WINSRV，且指向的是 WINSRV 中的 UserHardError 函数。

UserHardError 不做任何操作便调用 UserHardErrorEx。UserHardErrorEx 首先建立一个数据结构用来记录与该 Hard Error 相关的各种信息，除了包含 HARDERROR_MSG 结构的副本，该结构还包含线程、端口、消息文字等信息。然后 UserHardErrorEx 把指向这个结构的指针追加到一个专门用来记录此类结构的全局链表的末尾，全局变量 winsrv!gphiList 记录了这个链表的表头，gphiList 的含义是全局的硬错误信息链表(gloable hard error information pointer list)，我们简称其为 PHI 链表。随后 UserHardErrorEx 调用 ProcessHardErrorRequest 让其继续处理。

对于需要等待回复的情况，`ProcessHardErrorRequest` 会在当前线程中调用 `HardErrorHandler` 来处理 PHI 链表，否则会启动一个新的线程（名为 `winsrv!HardErrorWorkerThread`）来调用 `HardErrorHandler` 函数。

`HardErrorHandler` 函数首先会调用 `NtUserHardErrorControl` 内核服务（位于 `WIN32K` 中），其作用是根据参数中指定的命令在内核态中执行相应的任务。`WIN32K` 的全局变量 `gHardErrorHandler` 记录了系统中当前的 `HardError` 处理器。因此，`HardErrorHandler` 函数会通过向 `NtUserHardErrorControl` 发出一个初始化命令（命令值等于 0）将自己设为当前的 `HardError` 处理器（使用线程指针作为标识）。

接下来，`HardErrorHandler` 函数会依次从 PHI 链表取出要处理的 `HardError` 任务。对于每项任务，依次执行如下操作。

第一，向 `NtUserHardErrorControl` 发出连接桌面命令（命令值等于 2 或其他），`NtUserHardErrorControl` 接到此命令后，会调用 `win32k!xxxSetCsrssThreadDesktop` 将当前线程设置到合适的桌面。全局变量 `win32k!grpdeskRitInput` 记录了拥有原始输入线程（Raw Input Thread）的桌面；`win32k!gspdeskDisconnect` 记录了用户登录前（名为 Disconnect）的桌面；`win32k!gspdeskShouldBeForeground` 记录了当 Terminal Service 活动时的前台桌面。如果当前活动桌面不是当前硬错误希望显示的桌面，比如对设置了 `MB_DEFAULT_DESKTOP_ONLY` 标志的硬错误，只需要将其显示在默认（Default）桌面上，如果当前桌面不是默认桌面，那么 `NtUserHardErrorControl` 会返回错误，`HardErrorHandler` 函数收到这样的回复后，会对这个硬错误设置一个特殊标志，等当前桌面切换到默认桌面时再处理这个错误。

第二，准备窗口风格（TOPMOST）、按钮等信息，然后调用 `MessageBoxTimeoutW` 函数（或其他消息框函数）弹出消息对话框。

第三，向 `NtUserHardErrorControl` 发出与桌面分离命令（值等于 4），`NtUserHardErrorControl` 接到此命令后，会调用 `xxxRestoreCsrssThreadDesktop` 将当前桌面恢复到调用 `win32k!xxxSetCsrssThreadDesktop` 前的桌面。

第四，将响应结果（用户选取的或超时后的预设值）通过 `ReplyHardError` 回复给等待线程。

当 PHI 链表中的所有任务都处理完毕后，向 `NtUserHardErrorControl` 发出清理命令（命令号为 1），注销本线程在全局变量 `gHardErrorHandler` 中的记录。

图 13-2 勾勒出了发起硬错误和系统分发与处理硬错误的过程。大家可以从左上角的矩形开始看，它代表某个应用程序由用户态发起硬错误提示请求。该请求先通过 `NTDLL` 中的残根函数提交给未公开的 `NtRaiseHardError` 内核服务。对于来自用户的调用，`NtRaiseHardError` 在把参数数组中的所有字符串数据都复制到一个内核空间中的内存区后，便调用 `ExpRaiseHardError` 进行分发。

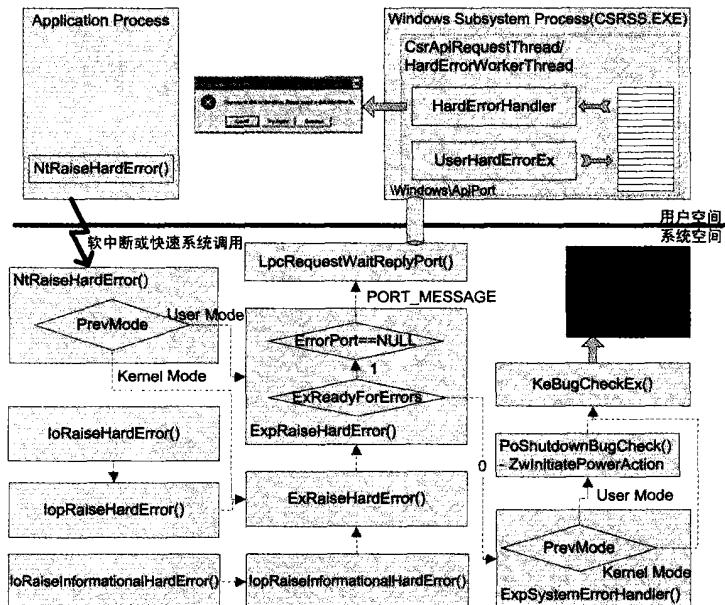


图 13-2 HardError 提示流程

用户态发起硬错误提示请求的一个典型例子是在 Windows XP 之前，当应用程序内出现未处理异常（Unhandled Exception）时，系统的 UnhandledExceptionFilter 函数会调用 NtRaiseHardError 函数来显示应用程序错误对话框。除此以外，某些应用程序也会调用这种机制来报告严重错误，比如图 13-3 所示的错误对话框，便是 VC6 的集成开发环境 MSDEV 程序通过硬错误机制所弹出的。

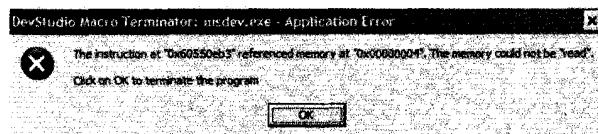


图 13-3 MSDEV 程序通过硬错误机制报告的严重错误

不仅可以从用户态发起硬错误提示请求，内核代码（比如驱动程序）也可以发起硬错误提示请求，比如我们前面介绍的因为缺盘而导致的硬错误，就是内核态的代码调用 IopRaiseHardError() 函数发起的提示请求。除了 IopRaiseHardError，DDK 中还公开了 IoRaiseHardError 和 IoRaiseInformationalHardError 两个函数用以发起 HardError 提示请求。图 13-2 的左下角区域描述了这几个函数的相互关系。

13.2 蓝屏终止 (BSOD)

蓝屏（Blue Screen）是 Windows 中用于提示严重的系统级错误的一种方式，因其出现时整个屏幕都被涂以蓝色背景而得名（参见图 13-4）。

因为蓝屏一旦出现，Windows 系统便宣告终止，只有重新启动才能恢复到桌面环境，所以蓝屏又被称为蓝屏终止（Blue Screen Of Death），简称为 BSOD。

从软件调试的角度来看，蓝屏机制的设计思想是将系统终止在导致错误的第一现场，并且把这个现场的信息显示给用户或永远保存下来，比如保存到转储文件，这样有利于更快地发现问题根源，去除软件错误（Bug）。所以蓝屏的另一种名称是错误检查（Bug Check），很多用于实现蓝屏机制的内核函数都带有 BugCheck 字样。

13.2.1 简介

因为产生蓝屏会终止整个系统的运行，所以，蓝屏是 Windows 中代价最高的错误提示方式。通常，只有发生极其严重的错误，或者其他错误提示方式无法工作时才使用这种方式。以下是采用蓝屏方式提示错误的几种常见情况。

第一，系统捕捉到内核代码中的未处理异常，或者检测到违反操作系统规则的情况。内核代码（包括运行在内核态的驱动程序）是操作系统的信任代码，对于这些代码中发生的错误，Windows 认为只有发生了严重的意外才会导致这些代码出错，所以通过蓝屏提示错误，并让系统以可控的方式终止运行。

第二，系统检测到操作系统的数据结构、模块或进程遭到破坏。

第三，在系统安装、启动或退出等边缘状态时发生错误，这时由于还不具备以其他方式提示错误的能力，所以只能以蓝屏提示。

例如，图 13-4 所示的蓝屏就是由于系统检测到 Windows 子系统的服务器进程 (CSRSS) 被终止而产生的。

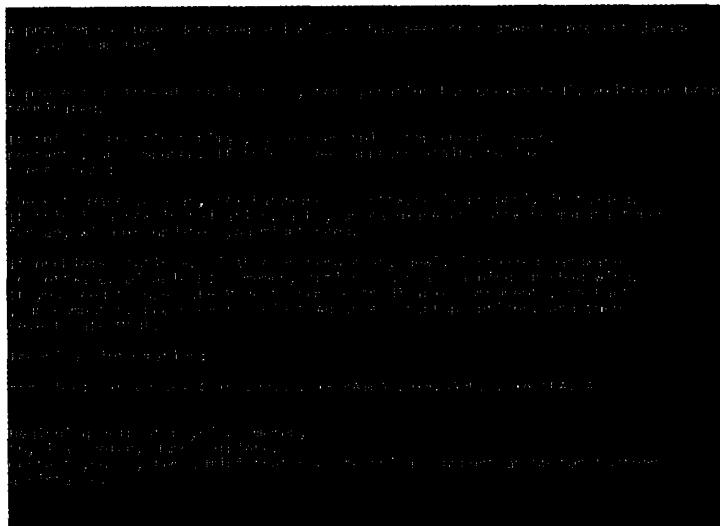


图 13-4 因 CSRSS 进程被终止而提示的蓝屏

通常，蓝屏中会包含以下几部分内容。

第一，错误信息。用以描述错误情况和错误原因的文字信息，比如图 13-4 中屏幕顶部的两段文字。

第二，解决错误的建议。对于同一版本的 Windows，这段信息只要出现就是相同的，所以通常没有太大的实际意义。

第三，技术信息（Technical information）。其格式为“STOP：停止码（参数 1，参数 2，参数 3，参数 4）”。其中，停止代码代表了导致蓝屏的根本原因，是诊断蓝屏故障的重要技术资料。停止代码后面括号中的参数用来进一步描述错误原因，其含义因停止码的不同而不同，代表了更深层次的错误信息。

第四，显示内存转储（Dump）的过程和结果。即图 13-4 中倒数第 3 行和第 4 行的信息。

在对蓝屏有了基本的了解后，下面我们先介绍发起蓝屏的方法，然后介绍蓝屏的产生过程。

13.2.2 发起和产生过程

DDK 中公开了两个 DDI（Device Driver Interface）用以提示蓝屏错误，KeBugCheck 和 KeBugCheckEx，其原型如下：

```
VOID KeBugCheck(IN ULONG BugCheckCode);
VOID KeBugCheckEx(IN ULONG BugCheckCode,
    IN ULONG_PTR BugCheckParameter1, IN ULONG_PTR BugCheckParameter2,
    IN ULONG_PTR BugCheckParameter3, IN ULONG_PTR BugCheckParameter4);
```

其中，BugCheckCode 便是出现在蓝屏上的停止码，可以是系统定义的停止码，也可以是驱动程序自己定义的代码。

因为 KeBugCheck 内部只是简单调用 KeBugCheckEx（后面 4 个参数都设为 0），所以，以上两个函数使用的是同一套实现。在 Windows XP 以前，其实现就在 KeBugCheckEx 中，从 Windows XP 开始，其实现位于内核中的 KeBugCheck2 函数中。考虑到 KeBugCheck、KeBugCheckEx 和 KeBugCheck2 这 3 个函数只是调用形式的差异，为了行文简洁，我们就使用 BugCheck 函数来泛指它们。

为了更好地理解蓝屏机制，我们把 BugCheck 函数的工作过程分为以下 11 个步骤。

第一，初始化和准备。包括将全局变量 nt!KeBugCheckActive 设置为真，标志着系统已经进入到特殊的错误检查（Bug Check）状态；产生描述系统状态的上下文结构（CONTEXT）。

第二，根据参数中的停止代码（Stop Code）寻找合适的错误提示信息。即蓝屏的第一部分内容。对于某些停止代码，KeBugCheck2 会根据 BugCheckParameterX 参数获取对应的模块名，并将其赋给全局变量 KiBugCheckDriver。例如，如果内核代码

在当前中断级别 (IRQL) 不低于 DISPATCH_LEVEL 时访问分页内存，那么系统检测到后，就会调用 BugCheck 函数发起蓝屏，并将停止码设置为 IRQL_NOT_LESS_OR_EQUAL (0xA)，BugCheckParameter1 是被访问的内存地址，BugCheckParameter2 是不当的 IRQL 值，BugCheckParameter3 是访问方式，0 为读，1 为写，BugCheckParameter4 是执行访问的指令地址。对于这个停止码，BugCheck 函数会根据 BugCheckParameter4 寻找对应的模块名称（利用 KiPcToFileHeader 或 MmLocateUnloadedDriver 函数）。如果找到了模块名称，那么它会被显示在蓝屏的提示信息中。这是为什么对于某些蓝屏错误，我们可以看到模块名的原因。

第三，如果启用了内核调试，那么调用 KdPrint 打印出停止码和蓝屏参数信息。而后判断内核调试器是否真正连接，如果是，则调用 KiBugCheckDebugBreak(3) 中断到内核调试器，其中参数 3 代表这是因为错误检查而第一次中断到调试器，类似于异常的第一次通知。在内核调试器中可以看到如下信息：

```
*** Fatal System Error: 0x00000050
          (0xF50E42F4,0x00000000,0xF8BA866D,0x00000000)
Driver at fault:
*** RealBug.SYS - Address F8BA866D base at F8BA8000, DateStamp 47ad2dfb
Break instruction exception - code 80000003 (first chance)
A fatal system error has occurred.
Debugger entered on first try; Bugcheck callbacks have not been invoked. [...]
```

第四，使系统进入到单纯的错误检查状态，停止其他一切活动。具体来说，BugCheck 函数会调用 KeDisableInterrupts 函数禁止中断，调用 KeRaiseIrql 函数将中断级别设置为最高 (HIGH_LEVEL)，对于多处理器版本，调用 KiSendFreeze 冻结其他 CPU。执行以上操作后，系统中只有当前的 CPU 在执行，而且它只会执行当前的线程，即继续处理错误检查，不做别的事。系统这样做是为了防止继续执行其他任务可能会有其他错误发生，破坏了最初错误环境，掩盖了真正的错误根源。而且以上过程是不可逆的，这意味着，系统一旦执行到这一步，所有用户态代码和当前线程之外的其他所有线程都不会被执行了，只有重新启动后才能恢复到正常的执行状态。

第五，绘制蓝屏画面。首先是启用用于启动过程的简单显示驱动 (BootVid)，然后将显示模式重置为 640×480 的低分辨率模式，将整个屏幕填充为蓝色，将文字设为白色。而后，逐步地绘制出蓝屏的错误信息和建议措施。在 Windows Vista 之前，这一步就是在 BugCheck 函数中实现的，Windows Vista 将其转移到一个单独的名为 KiDisplayBlueScreen 的内核函数中。

第六，调用错误检查回调函数。即驱动程序通过 KeRegisterBugCheckCallback 函数注册的回调函数。这一步的目的是通知驱动程序，系统已经进入错误检查阶段，驱动程序可以执行必要的清理工作，或者通知自己的硬件。

第七，如果内核调试引擎没有被启用，那么判断是否需要启用，如果需要，则调用启用内核引擎的函数来启用内核调试引擎。如果启用成功，会中断到内核调试器。

这种在系统蓝屏时动态启用内核调试的思想与上一章介绍的用户态的 JIT 调试很类似。需要启用内核调试引擎的一个典型例子就是在启用选项中指定了崩溃调试选项（/CRASHDEBUG）。启用内核调试引擎的方法在 Windows Vista 之前是调用 KdInit 函数，Windows Vista 是调用 KdEnableDebuggerWithLock 函数。

第八，准备系统转储（System Dump）数据，然后调用 IoWriteCrashDump 函数来将转储信息写入到存储器（通常为硬盘）中。在 IoWriteCrashDump 函数中，它会调用通过 KeRegisterBugCheckReasonCallback 函数注册的回调函数，目的是让驱动程序可以附加自己的转储数据（BugCheckSecondaryDumpDataCallback）或将数据写入到其他存储介质中（BugCheckDumpIoCallback）。

第九，再次扫描并调用通过 KeRegisterBugCheckCallback 函数注册的回调函数。

第十，判断是否需要自动重新启动，如果是，则调用 HalReturnToFirmware（HalRebootRoutine）重启系统，否则，系统便会长期停留在蓝屏画面。可以通过系统的计算机属性对话框配置这一选项（My Computer>Properties>Advanced>Startup and Recovery > Automatically Restart）。从 Windows XP 开始，这个选项是默认启用的，所以系统发生蓝屏后会自动重新启动。值得说明的是，这个重启选项只会影响本步骤，不会影响前面的步骤，也就是说，蓝屏画面总是被绘制的，只不过如果自动重启，而且系统运行速度较快，那么蓝屏画面可能一闪而过或根本看不到。

第十一，调用 KiBugCheckDebugBreak(4) 试图中断到调试器。其中参数 4 的含义是告诉内核调试器这是因为发生错误检查而第二次中断到调试器。

在以上过程中，我们可以看到很多调试支持，比如动态启用调试引擎，两次通知调试器，向调试器打印调试信息等。特别当第一次通知调试器时，BugCheck 函数尚未屏蔽中断和提升中断级别，这是为了可以让调试人员更好地了解错误情况。

13.2.3 诊断蓝屏错误

对于蓝屏错误，可以通过如下步骤逐步分析其原因。

首先，根据蓝屏的停止码和蓝屏参数作初步的判断。在 WinDBG 的帮助文件中（Debugging Techniques > Bug Checks (Blue Screens) > Bug Check Code Reference）以停止码为顺序列出了系统定义的所有蓝屏错误，包括错误码和每个参数的含义，以及解决问题的建议，这个建议是针对停止码的，要比蓝屏画面中的建议有用得多。

第二，可以在微软的知识库（support.microsoft.com）中或使用其他搜索引擎搜索蓝屏的停止码和参数，以了解更多信息。

第三，分析转储文件。系统会默认为蓝屏产生小型转储文件（Small memory dump），

默认位置为 Windows 系统目录的 MiniDump 文件夹, 文件名是 MiniMMDDYY-XX.dmp 的格式, 其中 MMDDYY 是发生蓝屏的日期 (月日年), XX 是序号, 从 01 开始, 依次递增。我们将在下一节详细介绍转储文件有关的内容。

第四, 如果经过以上步骤还没有找到问题的原因, 那么应该考虑通过内核调试做进一步的调试和分析。使用内核调试, 可以设置断点, 跟踪内核代码的执行过程, 这样更容易准确地定位到错误根源。我们将在第 18 章深入讨论与内核调试有关的内容。

13.2.4 手工触发蓝屏

可以通过以下方法之一来手工触发蓝屏。方法一是在内核调试会话中, 执行 WinDBG 的.crash 命令。

第二种方法不需要调试器, 但要事先在如下注册表键下加入一个 REG_DWORD 类型的键值, 并取名为 CrashOnCtrlScroll, 将其值设为 1。

位置: HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\i8042prt\Parameters

这个设置需要重新启动系统后才生效。当需要触发蓝屏时, 只要在键盘上按住 Ctrl 键, 再按 ScrollLock 键。因为是 Windows 系统的 PS2 端口驱动程序 (i8042prt.sys) 检查这组按键并调用 BugCheck 函数发起蓝屏的, 所以使用这种方法触发蓝屏机制需要系统安装有 PS2 键盘。大多数笔记本电脑的自带键盘都是 PS2 键盘。

第三种方法不要做任何设置也不需要使用调试器, 但是需要有硬件经验, 其做法是使用一个探针似的金属工具将内存条的某两个数据线短路或接地。使用这种方法时需要格外小心, 建议只有在没有其他方法时才在硬件工程师的帮助下使用。

13.3 系统转储文件

顾名思义, 系统转储文件就是将系统的状态从内存转储到磁盘文件中。相对于保存应用程序状态的用户态转储文件来说, 系统转储文件描述的目标是整个系统, 包括操作系统内核, 内核态的驱动程序和各个用户进程。

13.3.1 分类

为了满足不同情况的需要, Windows 系统定义了 3 种类型的系统转储文件, 分别是完整转储 (Complete memory dump)、内核转储 (Kernel memory dump) 和小型内存转储 (Small memory dump)。以上 3 种类型的文件其大小依次递减, 其包含的信息量也是如此。完整转储包含产生转储时物理内存中的所有数据, 其文件大小通常比物理内存的容量还要大。内核转储去除了用户进程所使用的内存页, 因此文件大小要比完

整转储的小得多，对于典型的 Windows XP 系统，其大小为 200MB 左右。小型转储的文件大小默认为 64KB，如果包含用户数据（通过 BugCheckSecondaryDumpDataCallback 回调函数写入），那么可能略大。

小型转储文件的默认位置为 Windows 系统目录的 MiniDump 文件夹，因为系统会按照日期加序号的方式为其命名，所以这个文件夹中可以保存很多个转储文件。另两种转储文件的默认存放位置是 Windows 系统的根目录（如 C:\Windows），其文件名称是固定的，默认为 memory.dmp，因此产生的新的会覆盖以前的。

13.3.2 文件格式

与用户态转储文件不同，系统转储文件的格式是不公开的。因此，以下内容是作者根据可以搜索到的资料加上个人的分析而得到的，仅供读者参考，目的是更好地理解和使用转储文件分析问题。

首先系统转储文件是以内存页面（4KB）为单位来组织数据的，因此它的大小是内存页大小（4096B）的整数倍。

对于所有的转储类型，转储文件中的第一页（即头页，Header Page）内容的格式是一样的，其开始处是一个名为 DUMP_HEADER 的结构，其布局如表 13-1 所示。

表 13-1 内核转储文件的头信息

偏移	长度	内容
0	8	文件签名，固定为 0x50 41 47 45 44 45 4D 50，即“PAGEDUMP”
8	4	0xC 表示内核是检查版本，0xF 表示自由版本，其他值代表系统主版本号
0xC	4	子版本号或构建号，XP 系统的典型值为 0xA28，即 2600
0x10	4	系统内核的页目录地址，即 CR3 寄存器的值
0x14	4	系统 PFN（Page Frame Number）数组的地址，即 MmPfnDatabase
0x18	4	模块列表的头节点地址，即全局变量 PsLoadedModuleList
0x1C	4	进程列表的头节点地址，即全局变量 PsActiveProcessHead
0x20	4	系统架构类型，0x14C 代表 x86，0x184 为 Alpha，0x1F0 为 PowerPC
0x24	4	处理器个数
0x28	4	蓝屏的停止码
0x2C	16	蓝屏的 4 个停止参数
0x60	4	内调试数据块，即 KdDebuggerDataBlock 的地址
0x64	32	文件中所包含物理内存块的概要描述
0x320		上下文结构，即 CONTEXT
0x7D0		异常结构，即 EXCEPTION_RECORD
0xF88	4	转储类型，1 为完整转储，2 为内核转储，4 为小型转储
0xF90	4	执行回调函数的结果，即 NT_STATUS 的值
0xF94	4	产品类型，1 代表 NT，3 代表服务器
0xF98	4	KUSER_SHARED_DATA 结构的 SuiteMask 字段
0xFA0	8	需要转储的空间
0xFB8	8	中断时间，即 KUSER_SHARED_DATA 结构的 InterruptTime 字段
0xFC0	8	系统时间，即 KUSER_SHARED_DATA 结构的 SystemTime 字段

从表 13-1 中我们可以看到，其中已经包含了系统的很多关键信息，包括页目录基地址、模块列表地址、进程列表地址、异常结构、上下文结构等。特别是偏移 0x60 处的 KdDebuggerDataBlock 地址，KdDebuggerDataBlock 结构是用来支持内核调试的，是内核调试器与内核调试引擎之间的重要数据接口，有了这个地址，调试器就可以通过它来与转储文件建立调试会话，使用类似活动内核调试的方式来分析转储文件。

头页之后的内容便是各个物理内存页的数据，包括实现内存管理的特殊内存页，如保存页目录和页表的内存页等。转储文件读取工具会利用其中的页目录和页表结构模拟地址过程来读取内存页的数据。举例来说，某个转储文件的 0x60 偏移处的值是 818f3c40，那么我们在 WinDBG 里就可以使用如下命令来观察 KdDebuggerDataBlock 结构的值：

```
0: kd> db 818f3c40
818f3c40  ec 7f b0 81 00 00 00 00-ec 7f b0 81 00 00 00 00  .....
818f3c50  4b 44 42 47 28 03 00 00-00 00 80 81 ff ff ff ff  KDBG(.....)
```

如果某个虚拟地址对应的内存页不在转储文件中，那么 WinDBG 会将其内容显示为“？”，与调试活动内核时观察不在物理内存中的内存块时的结果一样。

因为系统在产生转储文件的头页数据时，会先用 0x45474150 (PAGE 的 ASCII 码) 来填充整个页面，所以，如果直接观察转储文件，可以看到以上字段的间隙会保留着这些填充字符。

13.3.3 产生方法

可以通过以下两种方法之一来产生系统转储文件。

一是使用 WinDBG 调试器 .dump 命令。当我们在内核调试会话中执行这条命令时，WinDBG 调试器会通过内核调试会话来读取被调试系统的状态信息和内存数据，并按照我们前面介绍的格式写到指定的文件中。例如，执行命令 .dump c:\krnldmp.dmp，便可以让调试器为目标系统产生一个名为 krnlidmp.dmp 的小型内核转储文件。如果希望得到完全内存转储，那么只要在文件名前加上 /f 选项。

第二种方法是让系统来产生。因为当系统发生蓝屏崩溃时，默认情况下系统会产生系统转储文件。首先，在系统每次启动时都会为产生转储做好准备工作 (IoInitializeCrashDump)，包括初始化数据结构 (IopDumpControlBlock 等)，将磁盘端口驱动程序 (ATAPI) 在内存中复制一份，并命名为 dump_XXX，比如 dump_atapi。

当蓝屏发生时，系统的 BugCheck 函数在绘制蓝屏后会调用 IoWriteCrashDump 函数，这个函数会检查 dump_XXX 驱动程序的完好性，然后利用它将转储信息写入到位于系统盘上的虚拟内存页面文件 (pagefile.sys) 中。

而后，当系统再次启动时，WinLogon 程序会启动系统目录中的 SaveDump 程序，

后者会将转储数据从页面文件另存到转储文件中。当转储文件的类型是内核转储或完全转储时，系统会为其产生一个对应的小型转储文件。

13.4 分析系统转储文件

本节将通过一个实例来介绍如何使用 WinDBG 调试器来分析系统转储文件。

13.4.1 初步分析

首先启动 WinDBG，并点击 File > Open Crash Dump...，然后浏览到本书配套资料的 Dump 目录并选择其中的 Mini121206-01.dmp 文件。点击确定后，WinDBG 会显示类似清单 13-2 所示的信息。

清单 13-2 使用 WinDBG 打开系统转储文件

```

1 Loading Dump File [C:\dig\dbg\author\data\Mini121206-01.dmp]
2 Mini Kernel Dump File: Only registers and stack trace are available
3 Symbol search path is: SRV*d:\symbols*http://msdl.microsoft.com/download/symbols;
4 Executable search path is:
5 Windows XP Kernel Version 2600 (Service Pack 1) UP Free x86 compatible
6 [目标系统概况和加载符号信息, 省略]
7 ****
8 *
9 *           Bugcheck Analysis
10 *
11 ****
12 Use !analyze -v to get detailed debugging information.
13 BugCheck 7F, {0, 0, 0, 0}
14 Unable to load image RealBug.SYS, Win32 error 2
15 *** WARNING: Unable to verify timestamp for RealBug.SYS
16 *** ERROR: Module load completed but symbols could not be loaded for RealBug.SYS
17 Probably caused by : RealBug.SYS ( RealBug+4e1 )

```

第 2 行告诉我们，转储文件的类型是小型转储，只包含寄存器和栈回溯信息。事实上其中还包含模块列表信息，可以通过 lm 命令来显示。第 3、4 行显示了 WinDBG 的路径设置，包括符号文件路径和可执行文件路径。

第 14 行显示未能加载 RealBug.SYS 文件，这是因为目前的可执行文件搜索路径为空（第 4 行），所以 WinDBG 没有找到这个文件。值得强调的是，当分析转储文件时，因为本机的环境和目标系统可能差异很大，所以正确设置可执行文件的搜索路径是很重要的。微软的符号服务器不仅包含符号文件，而且包含各种版本的系统文件，因此应该将其设置到符号搜索路径中（执行 .symfix 命令）。这样，当缺少合适版本的 Windows 系统文件时，WinDBG 会自动从符号服务器下载。

第 7~17 行是 WinDBG 对转储文件的初步分析，第 7~11 行是标题，第 12 行建议使用 !analyze 命令得到更多信息。第 13 行显示的是停止码和参数，可以通过 WinDBG 帮助文件找到这些代码的含义（Debugging Techniques > Bug Checks（Blue Screens）> Bug

Check Code Reference)。查阅后，我们知道 7F 代表 UNEXPECTED_KERNEL_MODE_TRAP，即意外的内核模式异常，其中第一个参数为 0，代表异常的具体类型是除零异常（即#DE）。

第 17 行显示了 WinDBG 的初步判断结果：崩溃可能是由 RealBug.SYS 模块导致的，导致问题的代码距离这个模块的起始地址 0x4e1 个字节。使用 `lm m real*` 命令可以列出这个模块的起始地址。

```
kd> lm m real*
start end module name
fa18b000 fa18bd00 RealBug T (no symbols)
```

其中 T 代表这个模块的时间戳 (Timestamp) 信息缺失，这是因为 WinDBG 还未能加载到这个模块的映像文件。

13.4.2 线程和栈回溯

可以通过栈回溯信息来进一步了解转储时的线程状态和崩溃原因。根据我们上一节的介绍，一旦发起蓝屏后，系统便不再作线程切换或执行其他任务，因此，通常作蓝屏绘制和转储的线程就是崩溃时的线程，这也是 WinDBG 打开转储文件后的默认线程。可以通过 `.thread` 和 `.process` 命令来显示当前线程 (KTHREAD) 和进程 (EPROCESS) 的结构地址。

```
kd> .thread
Implicit thread is now 815ef3f0
kd> .process
Implicit process is now 8160eb98
```

而后，可以通过 `!thread 815ef3f0` 和 `!process 8160eb98` 命令来显示线程和进程的更多信息。

```
kd> !thread 815ef3f0
...
Owning Process 8160eb98 Image: ImBuggy.exe
```

这说明崩溃发生在名为 `ImBuggy` 的进程中。也可以使用 `dt nt!_KTHREAD 815ef3f0` 和 `dt nt!_EPROCESS 8160eb98` 命令来观察这两个数据结构。

使用 `k` 命令可以查看当前线程的栈回溯信息：

```
kd> k
ChildEBP RetAddr
f6869ac0 80607339 nt!KeBugCheck+0x10
f6869b20 804dac8f nt!Ki386CheckDivideByZeroTrap+0x23
f6869b20 fa18b4e1 nt!KiTrap00+0x6d
WARNING: Stack unwind information not available. Following frames may be wrong.
f6869b9c 815ef600 RealBug+0x4e1
...
```

上面的警告信息是告诉我们 WinDBG 没有找到 `RealBug` 模块的符号文件。通过 `File > Symbol File Path...` 对话框将 Dump 目录加入到符号文件路径列表中，选中 `Reload` 后

点击 OK 按钮。再次输入 k 命令，会得到清单 13-3 所示的信息。

清单 13-3 模块加载错误的栈回溯

```
kd> k
ChildEBP RetAddr
f6869ac0 80607339 nt!KeBugCheck+0x10
f6869b20 804dac8f nt!Ki386CheckDivideByZeroTrap+0x23
f6869b20 fa18b4e1 nt!KiTrap00+0x6d
Unable to load image RealBug.SYS, Win32 error 2
*** WARNING: Unable to verify timestamp for RealBug.SYS
*** ERROR: Module load completed but symbols could not be loaded for RealBug.SYS
WARNING: Stack unwind information not available. Following frames may be wrong.
f6869b9c 815ef600 RealBug+0x4e1
f6869bcc fa18b54b 0x815ef600
```

以上信息说明 WinDBG 因为未能找到 RealBug.sys 文件并因此未能加载符号文件。这是由于 Windows 系统的符号文件加载函数（DbgHelp 函数）总是通过模块文件来组织其符号的。将包含 RealBug.SYS 文件的 Dump 目录设置到映像文件路径（File > Image File Path）并再次执行.reload 命令。这时，如果再执行前面的 lm 命令，RealBug 模块后的 T 标志便消失了。

再次输入 k 命令，这次应该得到没有任何警告和错误信息的栈回溯（清单 13-4）。

清单 13-4 正确的栈回溯

```
kd> k
ChildEBP RetAddr
f6869ac0 80607339 nt!KeBugCheck+0x10
f6869b20 804dac8f nt!Ki386CheckDivideByZeroTrap+0x23
f6869b20 fa18b4e1 nt!KiTrap00+0x6d
f6869bcc fa18b54b RealBug!PropDivideZero+0x3f [c:\...\realbug\realbug.c @ 63]
f6869bdc fa18b62e RealBug!DivideZero+0xb [c:\...\realbug\realbug.c @ 77]
f6869be8 fa18b73e RealBug!RealBugDeviceControl+0x44 [c:\...\realbug\realbug.c @ 114]
f6869c34 804eca36 RealBug!RealBugDispatch+0x8e [c:\...\realbug\realbug.c @ 175]
f6869c44 8058b076 nt!IopfCallDriver+0x31
f6869c58 8058bc62 nt!IopSynchronousServiceTail+0x5e
f6869d00 805987ec nt!IopXxxControlFile+0x5ec
f6869d34 804da140 nt!NtDeviceIoControlFile+0x28
f6869d34 7ffe0304 nt!KiSystemService+0xc4
0012f8a8 00000000 SharedUserService!SystemCallStub+0x4
```

从上面的栈回溯信息我们可以看到，WinDBG 已经给出了有关的源程序文件和代码行信息。打开这些源文件，可以非常清楚地看到导致这次蓝屏的原因就是 realbug.c 的第 63 行中执行了除零操作（n=n/m;）。

13.4.3 陷阱帧

当有异常发生时，系统会将当时的状态保存到一个 KTRAP_FRAME 结构中，称为陷阱帧。因为很多崩溃与异常有关，所以转储文件中经常包含着陷阱帧数据。首先可以通过 kv 命令搜索当前栈回溯序列中是否有关联的陷阱帧。

```
kd> kv
ChildEBP RetAddr Args to Child
f6869ac0 80607339 ... nt!KeBugCheck+0x10 (FPO: [1,0,0])
```

```
f6869b20 804dac8f ... nt!Ki386CheckDivideByZeroTrap+0x23 (FPO: [Non-Fpo])
f6869b20 fa18b4e1 ... nt!KiTrap00+0x6d (FPO: [0,0] TrapFrame @ f6869b2c)...
f6869d34 7ffe0304 ... nt!KiSystemService+0xc4 (FPO: [0,0] TrapFrame @ f6869d64)
0012f8a8 00000000 ... SharedUserData!SystemCallStub+0x4 (FPO: [0,0,0])
```

第 5 行和第 6 行末尾的 TrapFrame 表明这个线程中有两个陷阱帧，@ 符号后面便是它们的地址。使用 .trap 命令加上陷阱帧的地址便可以切换到一个陷阱（异常）发生时的状态，好像把时间倒退到那一点一样。例如：

```
kd> .trap f6869b2c
ErrCode = 00000000
eax=00000001 ebx=814c5368 ecx=00000004 edx=00000000 esi=8156aae8 edi=815ef600
eip=fa18b4e1 esp=f6869ba0 ebp=f6869bcc iopl=0 nv up ei pl zr na pe nc
cs=0008 ss=0010 ds=0023 es=0023 fs=0030 gs=0000 efl=00000346
RealBug!PropDivideZero+0x3f:
0008:fa18b4e1 f77de0 idiv    eax,dword ptr [ebp-20h] ss:0010:f6869bac=00000000
```

以上寄存器值便是这个异常发生时的状态，第 2 行的错误码是异常的错误码。最后 1 行的汇编指令便是导致这个异常的指令，可见它是一条除法指令。指令后面的 ss:0010:f6869bac 便是 ebp-0x20，等号后面是这个地址的值，即 00000000，这说明当时除数是 0，因此触发了除零异常。

13.4.4 自动分析

为了简化蓝屏分析，WinDBG 将很多可以自动完成的分析工作实现在 !analyze 命令中了。因此，分析蓝屏的第一步通常是先执行这个命令，让 WinDBG 自动作分析，然后再作手工分析。清单 13-5 显示了执行 !analyze-v 命令的结构。

清单 13-5 使用 !analyze -v 命令分析系统转储

```
1 kd> !analyze -v
2 UNEXPECTED_KERNEL_MODE_TRAP (7f)
3 [省略了多行关于停止码的详细解释]
4 Arguments:
5 Arg1: 00000000, EXCEPTION_DIVIDED_BY_ZERO
6 Arg2: 00000000
7 Arg3: 00000000
8 Arg4: 00000000
9
10 Debugging Details:
11 -----
12 BUGCHECK_STR: 0x7f_0
13 TRAP_FRAME: f6869b2c -- (.trap 0xfffffffff6869b2c)
14 ErrCode = 00000000
15 eax=00000001 ebx=814c5368 ecx=00000004 edx=00000000 ...
16 [陷阱帧信息，与前面 .trap f6869b2c 的结果相同]
17 Resetting default scope
18
19 CUSTOMER_CRASH_COUNT: 1
20 DEFAULT_BUCKET_ID: INTEL_CPU_MICROCODE_ZERO
21 PROCESS_NAME: ImBuggy.exe
22 LAST_CONTROL_TRANSFER: from 80607339 to 805266bf
23
24 STACK_TEXT:
25 f6869ac0 80607339 0000007f 815ef600 8156aae8 nt!KeBugCheck+0x10
```

```

26 [栈回溯, 从略]
27 SharedUserData!SystemCallStub+0x4
28
29 STACK_COMMAND: kb
30
31 FOLLOWUP_IP:
32 RealBug!PropDivideZero+3f [c:\...\realbug\realbug.c @ 63]
33 fa18b4e1 f77de0 idiv eax,dword ptr [ebp-20h]
34
35 FAULTING_SOURCE_CODE:
36     59:    n=1;
37     60:    m=0;
38     61:    __try
39     62:    {
40 >   63:        n=n/m;
41     64:    }
42     65:    __except(EXCEPTION_EXECUTE_HANDLER)
43     66:    {
44     67:        DBGOUT(("Caught divide by zero safely."));
45     68:    }
46 SYMBOL_STACK_INDEX: 3
47 SYMBOL_NAME: RealBug!PropDivideZero+3f
48 FOLLOWUP_NAME: MachineOwner
49 MODULE_NAME: RealBug
50 IMAGE_NAME: RealBug.SYS
51 DEBUG_FLR_IMAGE_TIMESTAMP: 457ea5ed
52 FAILURE_BUCKET_ID: 0x7f_0_RealBug!PropDivideZero+3f
53 BUCKET_ID: 0x7f_0_RealBug!PropDivideZero+3f
54 Followup: MachineOwner
55 -----

```

第 1 行中的-v 开关用来启动最详尽的分析方式。其他各行是!analyze 命令输出的分析结果。可以将其分为如下几个部分。

第一部分是蓝屏的停止码和参数（第 2~8 行）。其中包含了停止码所对应的常量、详细的解释说明和每个参数的含义。事实上，这部分内容来源于我们前面提到的 WinDBG 帮助文件中关于每个停止码的描述，摘取了其中的关键内容。

第二部分是 BUGCHECK_STR（第 12 行）。是赋给这次崩溃的一个分类代码，通常是蓝屏的停止码或停止码加上一个子类号，比如本例中的 0x7f_0，0x7f 是停止码，0 是这个停止码的第一个参数，是异常的向量号，代表除零异常。

第三部分是陷阱帧信息（第 13~15 行）。描述了导致这次蓝屏的意外异常发生时的状态，主要是当时的寄存器值和异常的错误代码。第 13 行中的地址就是用来保存这些状态信息的_KTRAP_FRAME 结构的地址，使用.trap 命令加上这个地址就可以切换到当时的状态。事实上，!analyze 命令就是先切换到这个陷阱帧后才显示出从第 14 行开始的这些信息的，第 15 行的提示信息表示将上下文切换回默认状态，也就是产生转储时的状态。

第四部分是关于这次蓝屏的几个基本属性（第 18~22 行）。第 19 行是系统的崩溃计数。第 20 行显示了这个蓝屏所属的大类（General Category），因为产生这个蓝屏的原因是操作系统感觉到了不该发生的异常（事实上是除零，稍后还会讨论），所以被归入 INTEL_CPU_MICROCODE_ZERO 类，另一种常见的大类是 DRIVER_FAULT，即

驱动程序错误。第 21 行显示的是导致蓝屏发生的进程名。第 22 行显示了最后一次执行转移的源和目标地址。使用 `ln` 命令可以检索每个地址附近的函数：

```
kd> ln 80607339
(80607316) nt!Ki386CheckDivideByZeroTrap+0x23  | (80607563)
    nt!NtQueryInformationPort
kda> ln 805266bf
(805266af)  nt!KeBugCheck+0x10   | (805266c2)  nt!KeBugCheckEx
```

由此判断，最后一次转移很可能是在 `Ki386CheckDivideByZeroTrap` 跳转到 `KeBugCheck` 函数，也就是 `Ki386CheckDivideByZeroTrap` 函数调用 `KeBugCheck`，发起蓝屏。

第五部分是栈回溯（第 24~29 行）。显示了于可疑线程栈上所记录的执行记录，包括函数调用及因为中断或异常而发生的转移。这部分信息对于深入了解导致蓝屏的原因很多时候是非常有价值的，为了保证它的准确性，应该先设置好映像文件路径和符号文件路径。第 29 行显示了产生这个栈回溯所使用的命令，重复这个命令应该可以得到同样的结果。本例中是 `kb` 命令，有时可能是先执行 `.tss` 或 `.trap`，然后再执行 `kb` 命令，目的是先切换到可疑的线程或陷阱帧。

第六部分是与错误相关的指令位置。即程序指针（IP）值（第 31~34 行）。其中的 FOLLOWUP 是“贯彻”和“进一步追查”的意思。FOLLOWUP_IP 就是要进一步追查和分析的程序指针。这一部分内容有时也被称为 FAULTING_IP，即导致错误的程序指针位置。第 33 行显示了 IP 地址、该地址的机器码和对应的汇编指令。第 32 行显示了这个 IP 地址所对应的符号（模块和函数）。

第七部分是导致错误的源代码（FAULTING_SOURCE_CODE）（第 35~45 行）。即上一部分的汇编指令所对应的源程序片段。前面带有>号的（第 40 行）是导致错误的那一行。

第八部分（第 46~53 行）是自动产生的统计字段。以供错误分析软件对大量的转储文件进行自动分析、统计和归档。微软的 OCR（Online Crash Analysis）服务器（<http://oca.microsoft.com/>）就是用来做这一工作的。

最后一个部分显示的是谁应该来进一步追踪和负责这个问题。`MachineOwner` 是默认的负责人。通过修改 WinDBG 的 triage 目录中的 `trage.ini` 文件可以定义模块或函数的负责人，其基本格式是 `Module[!Function]=Owner`。例如，如果在这个文件的末尾加入新的一行：

```
RealBug!PropDivideZero=AaFoo
```

重新启动 WinDBG 后再分析这个转储文件，我们就会看到 `Followup: AaFoo`。可以通过 `!owner` 命令来查询指定模块或函数的负责人。

本节通过一个实例介绍了如何分析系统转储文件。因为 WinDBG 将分析转储文件看作调试一个特殊的被调试系统（非活动的），然后使用各种内核调试命令对其进行分

析，除了跟踪和运行类命令外，其他大多数命令都是可以执行的，所以在我们熟练掌握了内核调试的原理和命令后，便很自然地学会了分析转储文件。因此，尽管只有本节是专门针对系统转储分析的，但是本书后面介绍的很多内容（特别是第18章和第30章）都是可以应用到转储分析上的。

13.5 辅助的错误提示方法

无论是消息框还是蓝屏都是以图像形式让用户通过视觉感知错误情况。除此以外，Windows也提供了通过声音和闪动窗口等形式来提示错误的方法，我们将其称为辅助的错误提示方法。

13.5.1 MessageBeep

与调用 MessageBox 类似，应用程序可以非常方便地调用 MessageBeep API 来以声音形式提示错误，其函数原型如下：

```
BOOL MessageBeep(UINT uType);
```

其中，uType 参数用来指定声音的种类，可以是以下常量之一：MB_ICONASTERISK（0x00000040L）、MB_ICONEXCLAMATION（0x00000030L）、MB_ICONHAND（0x00000010L）、MB_ICONQUESTION（0x00000020L）、MB_OK（0x00000000L）和-1。

每一种声音对应的波形文件（wave file）都是可配置的，通过控制面板中的声音管理对话框（Sound And Audio Devices）可以修改这些配置，配置结果被保存在如下注册表表项中：

```
HKEY_CURRENT_USER\AppEvents\Schemes\Apps\.Default
```

通常，系统会通过系统中的声卡设备来播放指定的波形文件，如果失败，则会尝试播放默认的声音（.Default 键下的声音）文件，如果仍旧失败，那么会使用主板上或机箱的蜂鸣器（PC 喇叭）来播放简单的声音，这相当于调用 Beep 函数（稍后讨论）。

MessageBeep API 是从 USER32.DLL 中导出的，其内部工作过程是典型的 Windows 子系统的用户态 DLL（客户）调用内核驱动（WIN32K.SYS）中的子系统服务的过程。具体来说，MessageBeep API 会通过 USER32.DLL 中的残根函数 NtUserCallOneParam 调用 WIN32K.SYS 中的内核服务 NtUserCallOneParam。NtUserCallOneParam 是用来中转子系统服务调用的一个常用函数，其原型如下（USER32.DLL 中的残根和 WIN32K.SYS 中的真正函数具有同样的原型）：

```
DWORD STDCALL NtUserCallOneParam (DWORD dwParam, DWORD dwRoutine);
```

其中 dwRoutine 用来指定要调用目标服务的序号，dwParam 用来传递请求服务的参数。类似的，win32k!NtUserCallTwoParam、win32k!NtUserCallNoParam 和

`win32k!NtUserCallHwndParam` 分别用于调用两个参数、没有参数和以窗口句柄作为参数的子系统服务。

也就是说，`MessageBeep` API 只是简单地通过 `NtUserCallOneParam` 来调用 `WIN32K.SYS` 中的子系统服务的，其伪代码如下：

```
BOOL MessageBeep(UINT uType)
{
    return (BOOL)NtUserCallOneParam(uType, 0x31);
}
```

其中 `0x31` 是 `WIN32K.SYS` 中提供 `MessageBeep` 服务的那个内核例程在内核服务表中的序号，对于每个特定的 Windows 版本，它是一个常数，我们是从 `MessageBeep` 的汇编清单中得到这个值的：

```
0:000> uf user32!MessageBeep
USER32!MessageBeep:
77d69083 6a31          push   0x31 // MessageBeep 服务的索引号
77d69085 ff742408      push   dword ptr [esp+0x8] // uType 参数
77d69089 e8ffffcfdf    call   USER32!NtUserCallOneParam (77d48d8d)
77d6908e c20400         ret    0x4
```

`WIN32K` 中的 `xxxMessageBeep` 函数是用来提供 `MessageBeep` 服务的。它是如何实现（播放波形文件或 `beep`）的呢？简单来说，它又把这项任务通过 Windows 消息发给了 `WinLogon`。

```
PostMessage(gspwndLogonNotify, 0x4C, 0x9, uSoundID);
```

`WinLogon` 是 `SMSS` 在启动 `CSRSS` 后启动的另一个重要的系统进程。它负责与系统登录有关的很多重要任务，比如 `GINA` 模块就是被加载在 `WinLogon` 进程中的。`WinLogon` 通过一个专门的窗口（"SAS window"）来接收各种请求，并在初始化阶段将该窗口的句柄通过 `win32k!NtUserSetLogonNotifyWindow` 函数告诉给 Windows 子系统的内核部分（`WIN32K`），`WIN32K` 将该窗口句柄记录在名为 `win32k!gspwndLogonNotify` 的全局变量中。因为该窗口属于 `WinLogon` 专用的桌面\`WinLogon`，所以在登录到 Windows 系统中的普通桌面（\Default）后是看不见这个窗口的，使用 `FindWindow (Ex)` API 也找不到它。

SAS 窗口收到消息后，会调用 `PlaySound` API 来播放声音文件。

```
BOOL WINAPI PlaySound( LPCSTR pszSound, HMODULE hmod, DWORD fdwSound);
```

当 `fdwSound` 参数中带有 `SND_ALIAS` 标志时，`pszSound` 可以指定一种声音代号（alias）的名称。在早期的 Windows 中（3.X），`WIN.INI` 文件中定义了这些别名对应的声音波形文件名称。现在这些设置都定义在注册表中，即我们前面介绍的表项：

```
HKEY_CURRENT_USER\AppEvents\Schemes\Apps\.Default
```

该表项下的键名（key）即声音代号的名称，如图 13-5 所示。

每个键下通常有两项，一项为当前设置，一项为默认定义。如果用户保存过其他 `scheme`，那么还有以该 `scheme` 名称定义的一项。

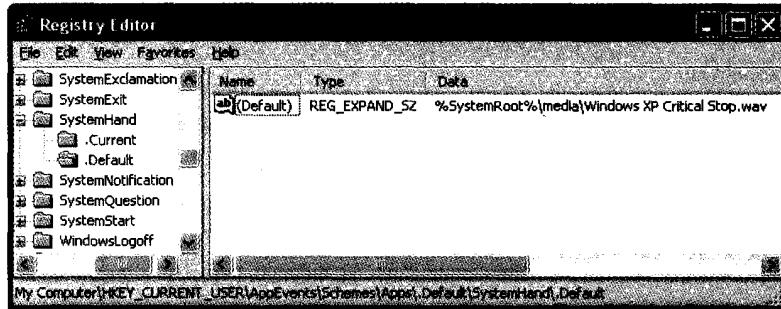


图 13-5 注册表中的声音事件定义

不同版本的 Windows 包含的声音事件类型可能不同,如果播放一个不存在的别名,那么系统便会播放默认的声音 (.default)。为了演示如何直接播放这些声音,我们编写了一个名为 MsgBeep 的小程序,清单 13-6 列出了源代码。

清单 13-6 Windows 的声音事件别名和直接播放方法

```

1 #include <windows.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 static CONST LPCTSTR UserSoundsAlias[] = {
6     ".Default",    "AppGPFault",      "CCSelect",      "Close",
7     "CriticalBatteryAlarm",    "DeviceConnect",    "DeviceDisconnect",
8     "DeviceFail",     "FaxBeep",        "LowBatteryAlarm",  "MailBeep",
9     "Maximize",       "MenuCommand",    "MenuPopup",     "Minimize",
10    "Open",          "PrintComplete",   "RestoreDown",   "RestoreUp",
11    "ShowBand",      "SnapShot",       "SystemAsterisk",
12    "SystemExclamation", "SystemExit",    "SystemHand",
13    "SystemQuestion",  "SystemStart",    "WindowsLogoff",
14    "WindowsLogon",   "WindowsUAC",     "TestForNotExisted"};
15 void PlayEventSound()
16 {
17     for(int i=0;i<sizeof(UserSoundsAlias)/sizeof(LPCTSTR);i++)
18     {
19         printf("Any key to play [%s] sound...\n",UserSoundsAlias[i]);
20         getchar();
21         if(!PlaySound(UserSoundsAlias[i],
22                         NULL,
23                         SND_ALIAS|SND_SYNC))
24         {
25             printf("Failed to play the sound with error %d.\n",
26                   GetLastError());
27             getchar();
28         }
29     }
30 }
31 void main()
32 {
33     printf("Any key beep... \n"); getchar();
34     Beep( 750, 300 );
35
36     printf("Any key to call MessageBeep... \n");getchar();
37     MessageBeep(MB_ICONEXCLAMATION);
38     PlayEventSound();
39 }
```

第 5~14 行的常量列出了 Windows Vista 中定义的所有声音事件别名。在第 21~23 行调用 PlaySound API 时我们指定了 SND_SYNC 标志，目的是让这个函数等待声音播放完毕后再返回。这与由 MessageBeep API 触发的 SAS 窗口过程中的调用不同，后者使用的是 SND_ASYNC 标志，即将声音任务放入队列后便立即返回。

最后要指出的是，如果 SAS 窗口过程调用 PlaySound 失败，那么对于某些重要的事件，它会调用 Beep 函数发出简单的声音。最后归纳 MessageBeep API 的整个工作过程：(1) 通过 USER32.DLL 中的残根函数 NtUserCallOneParam 调用 NtUserCallOneParam 内核服务，其目的是间接调用 WIN32K 中的 xxxMessageBeep 服务。(2) xxxMessageBeep 内核例程先将 uType 参数翻译为相应声音事件所对应的 ID，然后将播放任务发给用户态下 WinLogon 进程的 SAS 窗口。(3) SAS 窗口通过声音事件的序号找到对应的声音事件别名，然后调用 PlaySound API 发出播放命令。(4) PlaySound API 在注册表 HKEY_CURRENT_USER\AppEvents\Schemes\Apps\Default 表项下寻找声音别名对应的表项，然后打开其.Current 键值所定义的声音文件并播放。

13.5.2 Beep 函数

也可以通过另 Beep API 产生声音提示。

```
BOOL Beep( DWORD dwFreq, DWORD dwDuration);
```

Beep 函数总是通过 PC 喇叭来播放简单的声音，dwFreq 指定声音的频率（以 Hz 为单位，其数值必须在 37 到 32 767 之间），dwDuration 参数用来指定声音的时间长度（以毫秒为单位）。

Beep 函数是同步的，也就是说，只有当声音播放完毕后，该函数才返回。MessageBeep 函数是异步的，它将要播放的声音放入队列后便返回。通过控制面板或直接修改注册表可以取消因为调用 MessageBeep 而发出的声音。但是要取消 Beep 函数发出的声音，只能停止 beep 驱动程序。这可以通过在命令行中输入如下命令实现：

```
net stop beep
```

事实上，Windows 中有一个专门的驱动程序 beep.sys 负责 beep 功能。Beep 函数（在 KERNEL32.DLL 中实现）内部就是通过 NtCreateFile 打开 beep 驱动程序所创建的设备，然后调用 NtDeviceIoControl 把参数通过 IOCTL_BEEP_SET 命令发给 beep 驱动程序，beep 再通过 IO 命令反复驱动硬件打开或关闭蜂鸣器，产生声音。上面的 net stop 命令就是停止这个驱动程序（在系统中，驱动程序也被看作是服务）。

Beep 是个用户态的 API，供应用程序使用。对于内核模块，可以使用 UserBeep 函数。UserBeep 函数的功能和实现与 Beep() API 几乎一样，只不过它是内核中的，可以供其他内核例程调用。

因为 ASCII 码 7 代表的含义是 beep，所以在控制台窗口执行 echo <Ctrl+G>也可听到 PC 喇叭的蜂鸣声。

13.5.3 闪动窗口

Windows 还提供了一种让窗口闪动 (flash) 的提示机制。可以使用 FlashWindow 或 FlashWindowEx 来调用这一功能。

```
BOOL FlashWindow( HWND hWnd, BOOL bInvert );
BOOL FlashWindowEx( PFLASHINFO pfw );
```

其中 hWnd 是要闪动的窗口，bInvert 是一个反转标志，pfwi 是一个结构，其定义如下：

```
typedef struct {
    UINT cbSize;           // 该结构的大小
    HWND hWnd;             // 要闪动的窗口句柄
    DWORD dwFlags;         // 标志
    UINT uCount;           // 次数
    DWORD dwTimeout;       // 闪动的间隔时间，以毫秒为单位。0 表示使用默认值
} FLASHINFO, *PFLASHINFO;
```

通过 dwFlags 可以指定。

- 闪动范围。是闪动窗口标题栏 (FLASHW_CAPTION) 还是任务条上的按钮 (FLASHW_TRAY) 或是全部 (FLASHW_ALL)。
- 连续闪动 (FLASHW_TIMER 或 FLASHW_TIMERNOFG)。不停地闪动直到调用 FLASHW_STOP。
- 停止闪动 (FLASHW_STOP)。

为了方便大家尝试 FlashWindowEx 和 FlashWindow 的工作方式，我们特意编制了一个名为 FlashWin 的小程序，其界面如图 13-6 所示。

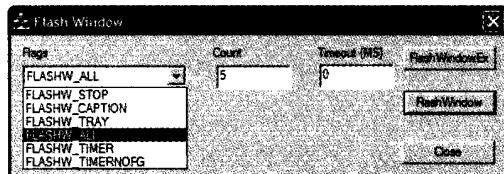


图 13-6 试验 FlashWindowEx API 的小程序

FlashWin 程序的完整源代码在 code\chap13\falshwin 目录中。

13.6 配置错误提示机制

前面几节我们介绍了 Windows 系统提供的错误提示机制，为了使这些机制具有非常好的灵活性和可控性，Windows 提供了配置界面和相应的 API 来控制这些机制。

13.6.1 SetErrorMode API

SetErrorMode API 用来设置（定制）当前进程的错误提示方式，其原型如下：

```
UINT SetErrorMode( UINT uMode );
```

其中，**uMode** 参数可以是多个标志位的组合，每个标志位用来开启或关闭某一个选项。MSDN 中定义了如下几个标志。

SEM_FAILCRITICALERRORS (0x0001)：不显示严重错误处理（critical-error handler）消息框，即我们前面介绍的硬错误（HardError）提示。如果当前进程设置了这个标志，那么系统就会取消分发来自这个进程的硬错误提示请求，返回 ResponseReturnToCaller，除非发生的 HardError 的 ErrorStatus 参数中带有 HARDERROR_OVERRIDE_ERRORMODE 标志。

SEM_NOALIGNMENTFAULTEXCEPT (0x0004)：系统自动处理内存对齐异常，不通知应用程序。

SEM_NOGPFAULTERRORBOX (0x0002)：不显示一般保护错误（General Protection Fault）消息框，即“应用程序错误”对话框。在系统（kernel32.dll）的未处理异常过滤函数（UnhandledExceptionFilter）中会判断这个标志（参见清单 12-7 的第 52 行和清单 12-8 的清单 78~79 行）。

```
if ( GetErrorMode() & SEM_NOGPFAULTERRORBOX )
    return EXCEPTION_EXECUTE_HANDLER;
```

也就是说，如果当前进程的 ErrorMode 包含 SEM_NOGPFAULTERRORBOX 标志，便返回执行默认的异常保护块退出线程或进程，这样便看不到“应用程序错误”对话框了。

SEM_NOOPENFILEERRORBOX (0x8000)——不显示无法找到文件对话框（参见图 13-7）。也就是当类似如下所示代码中的 OpenFile API 无法找到参数中指定的文件时，不要弹出错误提示对话框。

```
OpenFile("c:\\abc.abc",&of,OF_PROMPT);
```

参数中应该包含 OF_PROMPT 标志系统才可能自动显示图 13-7 所示的 Retry、Cancel 对话框。

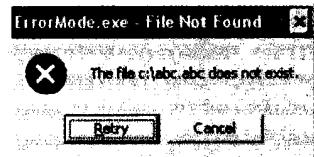


图 13-7 OpenFile API 自动弹出的消息框

事实上，图 13-7 的对话框是通过本章第 13.1 节介绍的硬错误机制弹出的。具体来说，当 OpenFile API 调用 CreateFile API 时以失败返回之后，它会调用

NtRaiseHardError 内核服务。因为 OpenFile API 在调用 NtRaiseHardError 函数时，在 ErrorStatus 参数中设置了 HARDERROR_OVERRIDE_ERRORMODE 标志位，所以单纯将 ErrorCode 设置为 SEM_FAILCRITICALERRORS 不能起到设置 SEM_NOOPENFILEERRORBOX 标志的作用。

每个进程的错误提示模式被记录在进程的 EPROCESS 结构的 DefaultHardErrorProcessing 字段中，上面讲的 4 种标志各对应 1 位。值得说明的是，SEM_FAILCRITICALERRORS 标志为 0 时有效，为 1 时无效，与其他 3 个相反。

SetErrorMode API 的实现（位于 kernel32.dll 中）并不复杂，其内部其实就是调用系统服务 NtSetInformationProcess 来设置 DefaultHardErrorProcessing 字段。

讲到这里大家可能会问，有 SetErrorMode API 来设置错误提示模式，是不是也有个 API 来获取当前的错误提示模式呢？在 WinDBG 中检查 KERNEL32.DLL 的符号，可以看到其中确实包含了 GetErrorMode 函数。但不知是什么原因，这个函数并没有被输出，头文件中也没有它的定义。

```
0:001> x kernel32!*ErrorMode*
77e7eed6 kernel32!GetErrorMode = <no type information>
77e7ee9f kernel32!SetErrorMode = <no type information>
```

因此我们是没有办法像调用 SetErrorMode API 那样直接调用 GetErrorMode 函数的。解决办法有两个。一是 SetErrorMode 会返回旧的 ErrorCode 值，因此可以通过以下代码间接取得当前的 ErrorCode：

```
UINT nCurMode = SetErrorMode(0);
SetErrorMode(nCurMode);
```

第二种方法是，既然我们已经知道 ErrorCode 保存在 EPROCESS 结构的 DefaultHardErrorProcessing 字段里，所以可以通过调用系统服务 NtQueryInformationProcess 取得该字段，然后稍微加工（SEM_FAILCRITICALERRORS 标志是反逻辑存储的）便得到自己的 GetErrorMode 函数了（code\chap13\ErrorMode）。NTDDK 中包含了 NtQueryInformationProcess 服务的原型。

关于 SetErrorMode API 最后要说明的一点是，对于普通的 WIN32 GUI 程序，ErrorCode 的默认设置为 0，即启用所有错误提示对话框，这与 MSDN 中的说明是一致的（0 - Use the system default, which is to display all error dialog boxes.）。但对于 MFC 程序，默认设置为 32773，即 SEM_NOOPENFILEERRORBOX | SEM_NOALIGNMENTFAULTEXCEPT | SEM_FAILCRITICALERRORS，这意味着不显示打开文件失败对话框，系统自动处理内存对齐异常，忽略一般的硬错误提示（ErrorCode 中没有设置 HARDERROR_OVERRIDE_ERRORMODE 标志）。在 MFC 的源程序文件中搜索 SetErrorMode 可以看到，在 MFC 程序的入口函数 AfxWinInit（mfc\src\appinit.cpp）中一开头就调用了 SetErrorMode 函数：

```
BOOL AFXAPI AfxWinInit(HINSTANCE hInstance, HINSTANCE hPrevInstance,
    LPTSTR lpCmdLine, int nCmdShow)
```

```
{
...
// handle critical errors and avoid Windows message boxes
SetErrorMode(SetErrorMode(0) |
    SEM_FAILCRITICALERRORS|SEM_NOOPENFILEERRORBOX);
...
}
```

在 DLLINIT.CPP 的 RawDllMain 函数中也可以找到同样的调用。

13.6.2 IoSetThreadHardErrorMode

ErrorMode 是针对整个进程的设置，也就是说一个进程中的所有线程都共享同样的 ErrorMode 设置。那么可不可以使某个线程具有特别的设置呢？答案是肯定的，通过 DDK 中公开的 IoSetThreadHardErrorMode 函数可以禁止当前线程的硬错误提示。

```
BOOLEAN IoSetThreadHardErrorMode( IN BOOLEAN EnableHardErrors );
```

在内部，IoSetThreadHardErrorMode 其实就是设置线程 ETHREAD 结构中的 HardErrorsAreDisabled 标志的。

```
nt!IoSetThreadHardErrorMode:
804e3302 64a124010000  mov  eax,fs:[00000124] // 取得ETHREAD结构
804e3308 8d8848020000  lea   ecx,[eax+0x248] // 指向32位的CrossThreadFlags
804e330e 8b01           mov   eax,[ecx]          // 将CrossThreadFlags的值放入EAX
804e3310 c1e805         shr   eax,0x5           // HardErrorsAreDisabled在bit5
804e3313 f6d0           not   al               // 取反
804e3315 2401           and   al,0x1           // 去除多余位
// 至此返回值已经准备好(al中便是旧的HardErrorsAreDisabled标志值)
...
}
```

根据第 13.1 节的介绍，线程的 HardErrorsAreDisabled 标志具有比 DefaultHardErrorProcessing 更高的优先级。因此一旦设置了 HardErrorsAreDisabled 标志，即使进程的 ErrorMode 是允许提示硬错误的（没有设置 SEM_FAILCRITICALERRORS 位），来自这个线程的硬错误提示请求也会被（ExpRaiseHardError 函数）忽略。

13.6.3 蓝屏后自动重启

尽管显示蓝屏对于调试有着很多重要的意义，但对有些情况它是不必要的。对于没有用户值守的系统，比如服务器或将显示输出到大屏幕或其他终端设备上的自动展示系统，让系统停滞在蓝屏状态是不必要的。对于普通用户使用的系统，某些用户可能不希望看到蓝屏这样的界面。因此，从 Windows XP 开始，发生蓝屏后，系统默认会自动重启。但这是可以配置的，如果希望不要自动重启，那么操作步骤为 My Computer > Properties > Advanced > Settings for Startup and Recovery，然后清除对话框中的“Automatically restart”选项。

是否自动重启及与蓝屏有关的系统转储选项保存在注册表的如下位置中：

HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\CrashControl

图 13-8 显示了该注册表键下的各个键值，大家可以很容易地把它们与对话框中的选项对应起来，在此不再赘述。

需要说明的是，Windows 并不是到了导致蓝屏的严重错误已经发生了之后才到注册表中读取以上表项的，这是因为那时系统可能不够稳定而无法支持这样的操作。作为系统初始化的一个步骤，这些选项很早就被读取到一个名为 IopDumpControlBlock 的全局变量之中，完成该任务的内核函数名为 IoInitializeCrashDump。

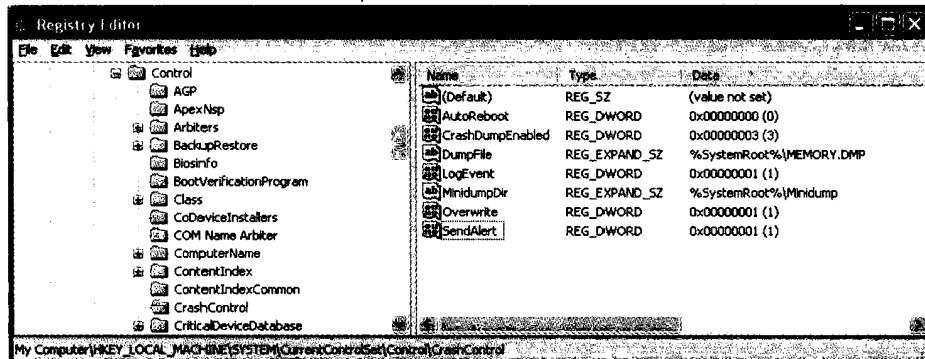


图 13-8 蓝屏和故障转储（DUMP）有关的注册表表项

当用户通过系统属性对话框修改系统失败选项时，系统先更改相应的注册表项，然后通过未公开的 zwSetSystemInformation 服务间接调用内核中的 IoConfigureCrashDump 函数、IoConfigureCrashDump 函数接到调用后，再调用 IoInitializeCrashDump，IoInitializeCrashDump 除了先调用 IopFreeDCB（DCB 的含义是 Dump Control Block）释放旧的结构，再调用 IopInitializeDCB 函数重新初始化 DCB 结构外，还要根据新设置决定是否需要重新分配转储使用的内存页面（例如转储类型从 Small memory dump 改为 Complete memory dump）。IopInitializeDCB 会调用 IopReadDumpRegistry 从注册表中重新读取各个表项，然后再将结果放到 IopDumpControlBlock 变量所指向的结构中。清单 13-7 显示了从用户按下 OK 按钮到调用 IopReadDumpRegistry 函数的整个过程。

清单 13-7 修改“系统失败”选项的执行过程

```

kd> k
ChildEBP RetAddr
f60b4b58 805ab54e nt!IopReadDumpRegistry      // 读取注册表
f60b4b80 805abd78 nt!IopInitializeDCB        // 初始化 DCB 结构
f60b4b8c 805ff3bb nt!IoInitializeCrashDump    // 初始化内核转储
f60b4ba0 805f0d67 nt!IoConfigureCrashDump     // 配置转储的内核函数
f60b4d50 804da140 nt!NtSetSystemInformation+0x256 // 系统信息设置内核服务
f60b4d50 7ffe0304 nt!KiSystemService+0xc4       // 内核服务分发函数
0006e6ac 77f7654b SharedUserData!SystemCallStub+0x4 // 调用内核服务

```

```

0006e6b0 5876cc9e ntdll!ZwSetSystemInformation+0xc // 调用设置系统信息的内核服务
0006e6f4 5876cf95 SYSDM!CoreDumpHandleOk+0xf4 // OK 按钮处理函数
0006e70c 5876a863 SYSDM!CoreDumpDlgProc+0x9f // 转储页的过程函数
0006e76c 77d43a68 SYSDM!StartupDlgProc+0x5b // 系统属性对话框的窗口过程函数
...
    
```

其中的 SYSDM 用于显示系统属性对话框的系统模块，其文件名为 SYSDM.CPL，CPL 即控制板程序所惯用的扩展名。

13.7 防止滥用错误提示机制

滥者，泛也；滥用就是任意（超过限度）或胡乱地（用错地方）使用。任何好的工具一旦被滥用都会适得其反，形成危害，错误提示机制也不例外。因此，在使用错误提示机制时应该格外重视合理使用，防止产生副作用。

最常见一个误区就是很多时候错误提示机制被误用为显示调试信息的工具，比如，有些程序员大量使用 MessageBox 来追踪（trace）执行轨迹或函数的返回值。这种通过插入 MessageBox 之类的语句显示调试信息的方法看似简单，但这是一种很不好的做法：首先，这样做效率很低。每个消息框通常只能显示很有限的调试信息，要在多个地方插入多个消息框也未必能解决问题；其次，不具可控性。消息框是硬编码到程序中的，每次改动都要修改源代码然后重新编译；最后，一旦有这样的消息框残留到产品之中就会造成比较严重的后果。因为用户根本不希望看到这种莫名其妙的调试信息，一些普通用户看到突然弹出的对话框（尤其是带有错误标志的），还会紧张和不安，影响使用体验。

辅助调试是错误提示机制的目的之一，但绝不能用错误提示机制取代其他正常的调试手段。应该采用合适的调试机制来帮助调试，比如使用 ASSERT 语句进行参数检查，使用 TRACE 语句或调用 OutputDebugString 之类的调试信息打印函数来追踪变量和执行路径。事实上，在开发期间应该把调试器作为最主要的调试手段。在调试器中运行，可以大大缩短我们熟悉代码执行情况和发现问题的时间。

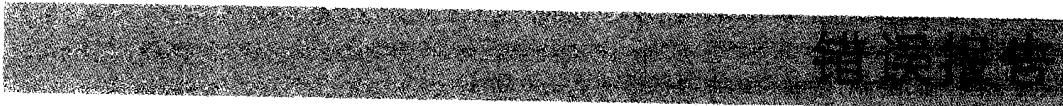
设计优秀的错误提示机制应该具有可控性，可以按严重程度、错误类别等属性定制错误提示的方式和次数。举例来说，对于正常发布的产品，应该通过配置选项保证只有严重的需要用户知晓的错误才提示给用户；但是当客户报告了问题，技术支持人员来到现场需要查找错误原因时，他们可以降低错误提示的级别，以便看到更多的错误提示，更快发现有价值的线索。

13.8 本章总结

本章介绍了 Windows 系统的错误提示机制，包括消息框和硬错误提示（第 13.1 节），蓝屏（第 13.2 节），声音提示（第 13.5），以及错误提示的配置方法（第 13.6 节）和注意事项（第 13.7 节）。第 13.3 节和第 13.4 节介绍了与蓝屏机制机制密切相关的系统转储机制，第 13.5 节介绍了分析转储文件的基本方法。我们介绍这些内容的目的一方面是让大家全面地理解 Windows 的错误提示机制，另一方面是让大家学习这种方法和思想，提高对错误处理设施的重视，并应用到软件开发实践中。

参考文献

1. David B. Probert. Windows Kernel Internals Lightweight Procedure Calls.
2. MSDN Library for Visual Studio 2005. Microsoft Corporation



发布到最终用户手中的软件仍然可能发生错误。当不同地域、不同语言的用户通过电话或信件向我们抱怨他们遇到了某个错误时，任何看似简单的信息收集工作都可能变得异常困难，因为用户可能根本听不进我们的指示，不愿意按照我们认为已经足够简单易懂的操作步骤一步一步地做下去。这时，一种解决方法是亲自跑到用户那里去，但如果有成千上万的用户或位于阿拉斯加的用户遇到了问题，那么这种做法就很难实行了。另一种做法便是让我们的软件自动收集错误信息，生成报告，并在征求用户同意后通过网络自动发送到一台专门用来收集错误报告的服务器。相对而言，第二种办法方便快捷，逐渐被越来越多的软件所使用。

尽管很难把错误提示（notification）和错误报告（reporting）这两个经常混合使用的词语截然分开，但是本书是用它们分别代表不同含义的，请从以下几个方面来区分。

- 从时间角度来看。错误提示是即时的，其目的是让用户立刻知晓所提示的信息；错误报告往往没有如此强的时间要求。
- 从目的性来看。错误提示的是让用户得知错误情况，选择处理方法；错误报告是记录错误详情以便找到错误原因。
- 从信息（数据）量角度来看。错误报告往往包含更全面、更多的信息。

从 Windows 3.0 开始，Windows 中就包含了 Dr. Watson 程序（drwatson.exe）用来收集错误信息并生成错误报告，但是不能自动地将报告发送给软件开发者。Windows XP 引入了自动发送错误报告的（DWWIN.EXE 程序）能力，可以根据系统设置将错误报告发送到指定的服务器；并且公开了 API（ReportFault 和 AddERExcludedApplication）供应用程序使用。Windows Vista 进一步加强和完善了错误报告机制，加入了更多的 API（以 Wer 开头），并正式地将错误报告机制定义为 Windows Error Reporting，简称 WER。因为 Windows Vista 的错误报告机制和 Windows XP 的有较大不同，所以在 SDK 的文档中，Windows XP 的错误报告机制被称为 WER 1.0，Windows Vista 扩充后的机制被称为 WER 2.0。

14.1 WER 1.0

从宏观来看，Windows Error Reporting（WER）采用的是比较典型的客户端/服务器（C/S）结构。客户端负责搜集、生成和发送错误报告；服务器端负责接收、存储、分类和自动寻找解决方案等任务。

14.1.1 客户端

WER 1.0 的客户端主要包括以下几个部分。

第一，FAULTREP.DLL，这是 WER 新引入的模块，位于 system32 目录中。WER 1.0 的两个公开的 API（ReportFault 和 AddEREExcludedApplication）都是在这个 DLL 中实现的。除了这两个 API，该 DLL 还导出了如下函数：CreateMinidumpA、CreateMinidumpW、ReportEREEvent、ReportEREEventDW、ReportFaultDWM（使用 DWWIN 程序报告应用程序错误）、ReportFaultFromQueue、ReportFaultToQueue、ReportHang、ReportKernelFaultA、ReportKernelFaultDWW（使用 DWWIN 程序报告系统错误）和 ReportKernelFaultW。

第二，DWWIN.EXE，WER 的客户端主程序，负责显示 WER 风格的“应用程序错误”对话框和发送错误报告，包括应用程序崩溃后的错误报告和系统崩溃后的错误报告。DWWIN.EXE 的对话框资源存放在 system32\<语言 ID>\DWINTL.DLL 文件中。如默认的英语资源 DLL 是 system32\1033\DWINTL.DLL。

第三，DW.EXE 和 DW20.EXE，这是 DWWIN.EXE 的两个变体，分别用于报告 Visual Studio .NET 2003 程序（devenv.exe）和 Office 程序（winword.exe、excel.exe 等）的应用程序错误。它们通常是通过 SetUnhandledExceptionHandler 注册自己的顶层未处理异常过滤器来启动的。事实上，只要观察 DWWIN.EXE 的文件属性就可以发现它的本来名称（Perpropties>Version>Original File Name）就是 DW.EXE。DW20.EXE 显然是 DW.EXE 的下一个版本。以下是在笔者机器上有关文件的版本、位置和描述的信息。

- DWWIN.EXE：版本号 10.0.4024.0, Original Name DW.EXE，位于 c:\winnt\system32 目录。
- DW.EXE：版本号 10.0.4109.0，位于 c:\Program Files\Microsoft Visual Studio .NET 2003\Common7\IDE 目录。
- Dw20.exe：版本号 11.0.6401.0，位于 c:\Program Files\Common Files\Microsoft Shared\DW 目录。

第四，DUMPREP.EXE，用于检查是否有等待发送的错误报告。如果有，则会通过动态加载 FAULTREP.DLL 中的 ReportFault 函数启动 DWWIN.EXE 程序发送报告。详细情况将在下文描述。

第五, 修改了的 KERNEL32.DLL, WER 修改了 KERNEL32.DLL 中的 UnhandledExceptionFilter 函数。当应用程序中出现未处理异常时, 调用 FAULTREP.DLL 中的 ReportFault API。这是 WER 与系统的一个重要接入点。

第六, 配置界面, 即图 14-1 所示的配置界面。它具有如下功能: 禁止和启用 WER; 启用针对 Windows 操作系统的 WER 功能; 启用针对应用程序的 WER 功能; 选择应用 WER 功能的应用程序 (可以定义包含列表和排除列表)。

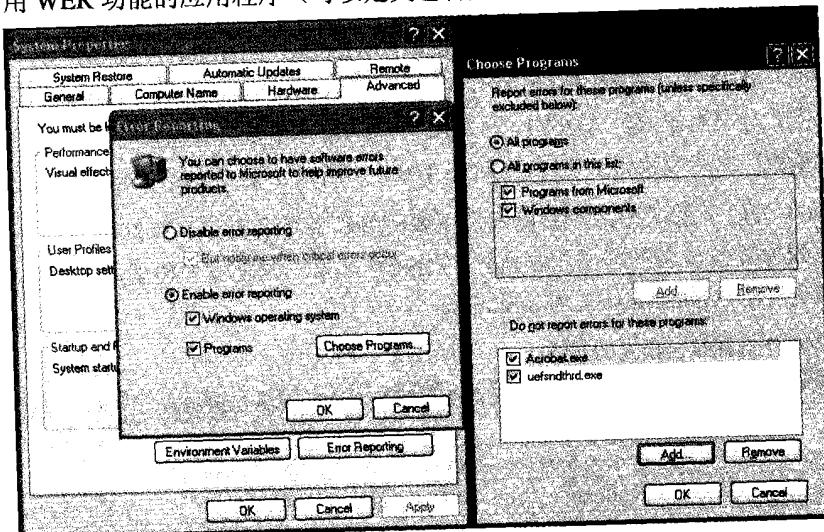


图 14-1 WER 设置界面

以上设置存储在如下注册表键中 (见图 14-2):

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\PCHealth\ErrorReporting

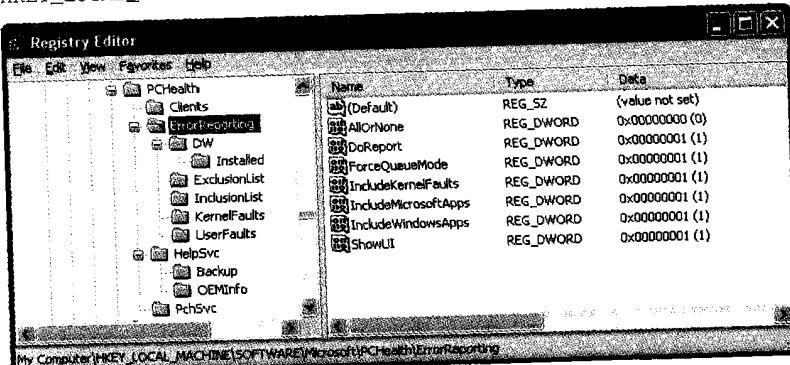


图 14-2 注册表中的 WER 设置

14.1.2 报告模式

WER 定义如下几种报告模式 (Reporting Modes)。

共享内存模式 (Shared memory mode): DWWIN 程序直接从发生错误的应用程序（内存）中抓取信息并产生故障转储文件。它只适用于应用程序由于未处理异常而导致崩溃的情况，所以该模式又被称为异常模式。使用该模式的另一个条件是应用程序的安全上下文（Security Context）与登录用户的安全上下文一致。

清单模式 (Manifest mode): DWWIN 程序根据清单文件发送错误报告。既适用于应用程序错误（异常、僵死），也适用于内核错误。登录用户必须具有管理员权限。

排队模式 (Queued mode): 默认情况下，Windows .NET Server 使用该模式。当有错误发生时，仍会调用 ReportFault API，但不会显示 UI，而是把要处理的任务放入一个队列中。当管理员登录时，这些 UI 会弹出，并询问管理员是否发送报告。该模式适用于所有错误情况。因为当错误发生时，没有 UI 显示，所以该模式又被称为 Headless Mode。

表 14-1 列出了系统选用报告模式中主要的判断条件和方法。其中第一列是登录用户，第二列为发生错误的应用程序所使用的账号，第三列是发生错误的应用程序是否运行在可交互的（interactive）Win-Station。

表 14-1 WER 选择报告模式的方法

登录用户	应用程序运行时的账号	Interactive?	报告模式
None	All Cases	All Cases	Queue
Administrator	System Service Account	All Cases	Manifest
Administrator	Interactively Logged-On User	All Cases	Exception
Administrator	Other	Yes	Exception
Administrator	Other	No	Manifest
Non-Administrator	System Service Account	All Cases	Queue
Non-Administrator	Interactively Logged-On User	All Cases	Exception
Non-Administrator	Other	Yes	Exception
Non-Administrator	Other	No	Queue

14.1.3 传输方式

WER 1.0 支持两种传输方式来发送报告文件。

互联网方式 (Internet Mode): 通过互联网直接发送到微软的错误报告服务器，网址为 <https://watson.microsoft.com>。该网址是硬编码到 DWWIN.EXE 程序中的，这样可以防止被不良程序篡改。另外从 URL 中包含的“https”可以看出，发送时使用的是安全的（Secured）HTTP 链接，可以防止数据被中途截取和盗用。

企业方式 (Corporate Mode): 先发送到企业内的一个共享文件夹，然后经企业 IT 部门审查后再发送到微软的服务器。共享文件夹是通过一个符合 UNC（Universal Naming Convention）规范的文件夹路径指定的。该模式下的错误报告又被称为 CER（Corporate Error Reporting），我们将在第 14.4 节详述。

Windows 系统默认的方式是互联网方式。

14.2 系统错误报告

上一节介绍了 WER 1.0 的客户端概况，本节将继续介绍系统错误是如何发送的。在系统发生崩溃并且再次启动后，WER 的客户端程序开始工作，准备发送错误报告。

首先，在注册表的自动运行表键下，默认会包含一条关于 DUMPREP.EXE 的记录以便每次 Windows 启动时都会自动运行 DUMPREP.EXE。

```
KEY: HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Run
Value Name: KernelFaultCheck
Type: REG_EXPAND_SZ
Value: %systemroot%\system32\dumprep 0 -k
```

-k 参数的含义是让 dumprep 检查是否有内核（kernel）错误需要处理。在 DUMPREP.EXE 看到此开关后，会检查以下注册表项，看是否有要处理的任务：

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\PCHealth\ErrorReporting\KernelFaults
```

上一章我们介绍系统转储文件的产生过程时提到过，当分页文件中包含了有效的转储数据时，系统会启动 SaveDump.exe 来将这些数据保存到文件中，并且会向 KernelFaults 表键中写入一条记录（见图 14-3）。

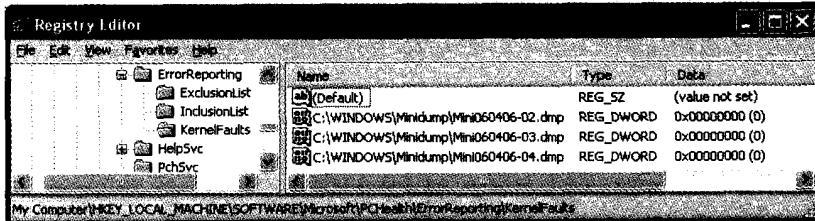


图 14-3 KernelFaults 表键记录了要报告的内核错误

在 DUMPREP.EXE 程序启动后会枚举 KernelFaults 表键下的所有键值，如果发现有效的键值，则会依次执行如下动作。首先是创建一个临时目录（参见图 14-4），并向此目录放入两个文件，一个名为 sysdata.xml 的 XML 文件，其中包含了系统的基本信息（Windows 名称、详细版本号、语言 ID），系统内设备信息和驱动程序信息。另一个名为 manifest.txt 的文本文件，用来作为清单提交给 DWWIN 程序。

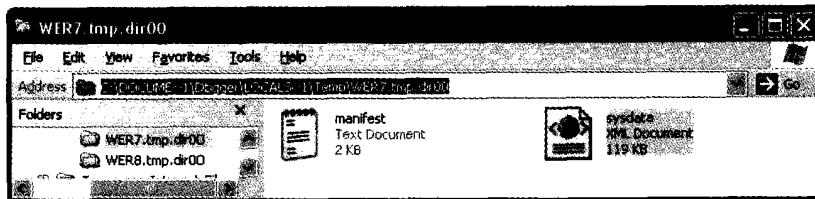


图 14-4 DUMPREP 生成的临时目录和文件

清单 14-1 给出了一个清单文件的内容。

清单 14-1 WER 使用的 manifest.txt 文件

```
Server=watson.microsoft.com
UI LCID=1033
Flags=123152
Brand=WINDOWS
TitleName=Microsoft Windows
DigPidRegPath=HKLM\Software\Microsoft\Windows
NT\CurrentVersion\DigitalProductId
RegSubPath=Microsoft\PCHealth\ErrorReporting\DW
ErrorText=A log of this error has been created.
HeaderText=The system has recovered from a serious error.
Stage2URL=
Stage2URL=/dw/bluetwo.asp?BCCode=deaddead&BCP1=00000000&BCP2=00000000&BCP3=00
000000&BCP4=00000000&OSVer=5_1_2600&SP=1_0&Product=256_1
DataFiles=C:\WINDOWS\Minidump\Mini060406-03.dmp|C:\DOCUME~1\Dbgger\LOCALS~1\T
emp\WER7.tmp.dir00\sysdata.xml
ErrorSubPath=blue
```

从清单 14-1 中，我们可以看到服务器地址、发送数据的 URL，以及要发送的文件（以 | 分隔多个文件）。

在完成以上准备工作后，DUMPREP 会通过动态加载 FAULTREP.DLL，并调用其中的 ReportFault 函数来启动 DWWIN 程序并以清单方式发送错误报告。

图 14-3 所示的窗口显示了 KernelFaults 表项下有 3 次内核崩溃记录（dump 文件）需要处理。DUMPREP 程序会依次处理它们，而且每处理完一项便删除一项。

DWWIN 程序启动后，便会显示出图 14-5 所示的系统错误对话框。

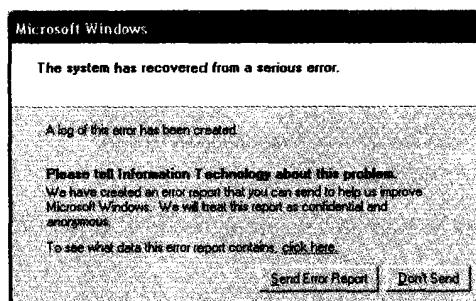


图 14-5 系统崩溃再重启后，DUMPREP.EXE 触发 DWWIN.EXE 显示系统错误对话框

点击图 14-5 中的“click here”链接可以看到错误报告的错误概况和一些说明信息。错误概况中的 BC 代表的是 Bug Check（错误检查），即蓝屏。BCCode 即错误检查代码。BCP 代表的是错误检查参数（Bug Check Parameter），即蓝屏错误中跟在错误检查代码后面的整数，通常有 4 个。继续点击对话框中的第一个“click here”链接，可以看到 WER 要发送的错误报告所包含的文件。

如果选择了图 14-5 中的“Send Error Report”按钮，那么 DWWIN 程序便会与错误报告服务器连接，并向其发送数据（见图 14-6）。

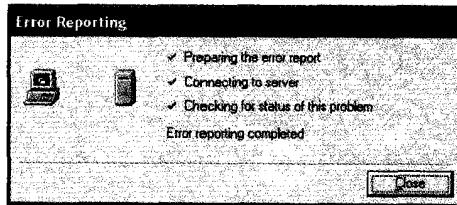


图 14-6 WER 向错误报告服务器发送错误报告

成功发送错误报告后，WER 服务器会检查系统中是否为该错误定义了回应信息。如果是，那么 DWWIN 会显示出图 14-7 所示的对话框，提示用户通过链接获取进一步信息，该对话框又被称为 WER 的结局对话框（final dialog）。

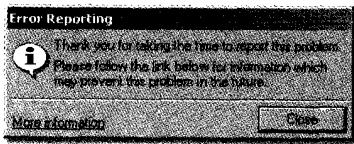


图 14-7 WER 的结局对话框

在用户点击 More Information 链接后，DWWIN 程序会使用浏览器打开一个图 14-8 所示的错误报告回应（response）页面，其中包含错误原因、可能的解决方案等信息。

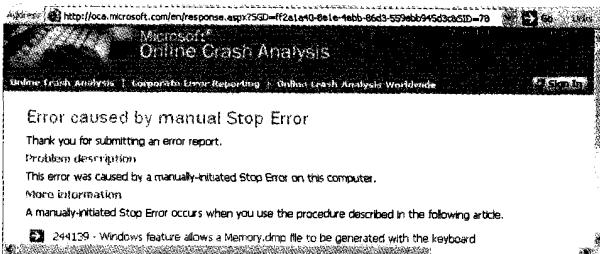


图 14-8 WER 的错误回应（response）页面

以上页面显示这次发送的系统错误是手工触发的系统崩溃。事实上，就是使用我们在上一章（第 13.2.4 节）介绍的按 Ctrl+ScrollLock 键所触发的蓝屏。

应用程序错误报告的发送过程与以上介绍的基本类似，我们不再做详细介绍。下一节将介绍 WER 1.0 的服务端。

14.3 WER 服务器端

WER 的服务器端主要包括以下几个 HTTP/HTTPS 站点。

- <https://watson.microsoft.com>: 负责接收错误报告。
- <https://oca.microsoft.com>: Online Crash Analysis (OCA) 门户站点，用于检查错误报告状态等。

- <https://winqual.microsoft.com>: Windows Quality Online Services 门户站点，负责注册用户，提交数字签名请求和下载错误报告等。

不仅微软自己的软件（操作系统和应用程序）可以使用 WER，其他软件开发商（ISV）也可以使用 WER。微软称其为 WER 服务（WER Service），或者 Windows Feedback 服务。

14.3.1 WER 服务

微软曾经公布过几组数据用来说明 WER 机制的重要性和取得的效果。第一组数据是：通过 WER 报告的关于 Office 产品组的错误有 50% 被 Office 产品的 Service Pack 所修正；Visual Studio 产品组报告的 74% 错误都被 Visual Studio 2005 Beta1 所修正；Windows 产品组中关于 Windows XP 的错误有 29% 被 Windows XP 的 SP1 所修正。这些数据表明 WER 机制成功地收集到了很多有效的错误报告。

另一组数据与著名的 20/80 规则很类似。

- 50% 的失败情况是因为数量上占 1% 的那些严重缺欠导致的。
- 80% 的失败情况是因为数量上占 20% 的那些严重缺欠导致的。

这组数字说明了软件开发商了解到用户频繁遇到的问题或大多数用户所遇到的问题的重要性。因为大多数用户遇到的问题都是由于软件中少数的几个严重缺欠（可能是很容易修正的）所导致的，了解到这一信息并修正了这些错误有着事半功倍的效果。

然而对于很多 ISV 来说，收集大量用户的错误报告并不是件容易的事，需要在软件中加入代码，建立收集错误的服务器，号召用户在遇到错误时发送错误报告，等等。

基于以上两点原因，微软将 WER 免费提供给其他 ISV 使用，具体做法是。

- 申请数字证书（digital certificate）。
- 通过 Windows Quality Online Services 门户站点（<https://winqual.microsoft.com/SignUp/>）注册用户。注册时需要接受协议。
- 定义关联关系，将你的公司与你公司的程序文件关联起来。这样 WER 便会把与所定义程序文件有关的错误报告发送到你公司所对应的“账号”中。

经过以上步骤后，便可以使用 WER 服务来收集和管理软件的错误报告了。

14.3.2 错误报告分类方法

WER 按照 Bucket 来分类和组织错误报告，我们在上一章分析转储文件时便看到了!analyze 命令为转储文件产生的 Bucket ID。

当有新发送来的报告时，WER 会先检查该报告所描述的问题是否已经存在 Bucket，如果存在，便把该报告加到这个 Bucket 中，如果不存在，则建立一个新的 Bucket。

对于发生在内核模式中的错误，WER 首先按停止码（Stop Code）进行分组，对

于某些频繁出现的停止码，再按停止码的参数进一步分组。Bucket 名称通常是由错误类型和模块名称来定义的。表 14-2 给出了一些 Bucket 名称的示例。

表 14-2 WER 的 Bucket 示例

Bucket 名称	含义
OLD_IMAGE_SAMPLE.SYS_DEV_3577	因为旧版本的 sample.sys (设备 ID 3577) 导致的崩溃
0x44_BUGCHECKING_DRIVER_SAMPLE	可能是驱动程序 sample.sys 导致的停止码为 0x44 的崩溃
POOL_CORRUPTION_SAMPLE	可能是由于 sample.sys 导致的 pool corruption
0xBE_sample!bar+1a	驱动程序 sample.sys 在 bar 函数中崩溃

对于用户态错误，WER 是按应用程序名称（如 winword.exe）、应用程序版本号、错误指令所在的模块名称（如 mso.dll）、模块版本号和错误指令在模块中的偏移地址来分类的。

WER 的错误报告中包含了每个 Bucket 的 ID 和所包含的错误报告数量（也就是该类错误的发生次数）及其对应的模块名称。

14.3.3 报告回应 (Response)

在软件开发者解决了某个 Bucket 所对应的缺欠后，可以通过 WER 的 Winqual 站点定义一个回应信息和一个获取更多信息的链接。在 WER 服务端接受完一个错误报告后，会检查给报告所对应的 Bucket 是否已经定义了回应信息，如果是，则会将这些信息显示给用户（参见图 14-7 和图 14-8）。

回应信息中可以包含：指向 ISV 自己公司的网站链接，让用户从那里得到更多信息或下载软件的更新版本；也可以指向微软的软件更新站点。

关于定义报告回应的具体细节，大家可以参阅微软网站的帮助页面 (https://winqual.microsoft.com/help/wer_help/bucket_response.aspx)。

14.4 WER 2.0

Windows Vista 对 WER 作了进一步加强，引入了一系列新的模块，并定义了更为丰富的 API，使 ISV 也可以非常方便地使用 WER 并给自己的软件加入强大的错误报告能力。为了与 Windows XP 的 WER 功能相区分，Vista 引入的 WER 被称为 WER 2.0。下面我们先介绍 WER 2.0 新引入的模块，然后讨论如何通过新的 WER API 来以程序方式发送错误报告。

14.4.1 模块变化

WER 2.0 引入了如下一些新模块，它们都位于 system32 目录下。

- Wer.dll: 新引入的 WER API 是实现在这个 DLL 中的。
 - Wercon.exe: WER 控制台界面, 用来观察错误报告、寻找解决方案及配置 WER 的选项, 取代了 WER 1.0 使用的简单界面(见图 14-1)。
- WER 2.0 使用如下注册表键来保存设置。
- HKEY_CURRENT_USER\Software\Microsoft\Windows\Windows Error Report。
 - Werclsupport.dll: 控制面板模块, 允许用户通过控制面板中的“问题报告和解决方案”图标启动 WER 控制台。
 - Werdiagcontroller.dll: WER 诊断控制器。
 - WerFault.exe 和 WerFaultSecure.exe: WER 2.0 错误报告程序。
 - Wermgr.exe: 负责管理和调度错误报告发送任务, 当应用程序使用进程外方式提交报告时, Wermgr.exe 会负责安排错误报告发送工作。
 - Wersvc.dll: WER 的系统服务, 以 SVCHOST 作为宿主进程运行。

WER 2.0 去除了用于发起系统错误报告的 DUMPREP.exe 程序, 改由 WER 系统服务来负责这一任务。

14.4.2 创建报告

首先, 应该调用 WerReportCreate API 创建一个 WER 报告, 其原型如下。

```
HRESULT WINAPI WerReportCreate(
    PCWSTR pwzEventType, WER_REPORT_TYPE repType,
    PWER_REPORT_INFORMATION pReportInformation, HREPORT* phReportHandle);
```

其中 pwzEventType 用来指定事件名称, 对于应用程序报告可以使用 WERAPI.H 中定义的 APPCRASH_EVENT。

```
#define APPCRASH_EVENT L"APPCRASH"
```

参数 repType 用来指定报告的类型, 可以为以下枚举常量之一:

```
typedef enum _WER_REPORT_TYPE
{
    WerReportNonCritical = 0,           // 非致命错误报告
    WerReportCritical = 1,             // 致命错误报告
    WerReportApplicationCrash = 2,     // 应用程序崩溃
    WerReportApplicationHang = 3,       // 应用程序僵死
    WerReportKernel = 4,               // 内核报告
    WerReportInvalid                // 无效值
} WER_REPORT_TYPE;
```

参数 pReportInformation 用来指定报告的应用程序名称等信息, 指向的是一个 WER_REPORT_INFORMATION 结构:

```
typedef struct _WER_REPORT_INFORMATION
{
    DWORD dwSize;                      // 本结构的大小
    HANDLE hProcess;                   // 要为其生成报告进程的进程句柄
}
```

```

WCHAR wzConsentKey[64];           // 用于查找征询设置的关键字，可以为 NULL
WCHAR wzFriendlyEventName[128];   // 友好的事件名称
WCHAR wzApplicationName[128];     // 应用程序名称
WCHAR wzApplicationPath[MAX_PATH]; // 应用程序的完整路径
WCHAR wzDescription[512];         // 问题描述
HWND hwndParent;                 // 父窗口句柄
} WER_REPORT_INFORMATION, *PWER_REPORT_INFORMATION;

```

如果 WerReportCreate 成功（返回 S_OK），那么 phReportHandle 参数会传回一个 HREPORT 句柄，供其他 API 使用。

接下来可以通过以下 API 丰富报告的信息或定制 UI 选项，包括。

- 使用 WerReportAddDump 产生并加入 DUMP 信息；
- 使用 WerReportAddFile 向报告中加入文件；
- 使用 WerReportSetParameter 设置最多 10 个参数（WER_P0 到 WER_P9）；
- 使用 WerReportSetUIOption 定制 UI 选项，比如对话框中的按钮文字和提示信息等。

14.4.3 提交报告

在准备好错误报告的内容后，可以使用 WerReportSubmit API 提交报告：

```

HRESULT WINAPI WerReportSubmit(
    HREPORT hReportHandle, WER_CONSENT consent,
    DWORD dwFlags, PWER_SUBMIT_RESULT pSubmitResult);

```

其中，consent 用来指示征询用户同意的情况，可以为以下枚举常量之一。

```

typedef enum _WER_CONSENT
{
    WerConsentNotAsked = 1,           // 尚未征求用户同意
    WerConsentApproved = 2,          // 用户已经同意提交本报告
    WerConsentDenied = 3,            // 用户拒绝提交本报告
    WerConsentMax                // 本常量的最大值
} WER_CONSENT;

```

参数 dwFlags 用来定制报告的界面和方式选项，可以是表 14-3 所列出的各种标志的组合。

表 14-3 提交 WER 报告的标志位

常量	位	说明
WER_SUBMIT_HONOR_RECOVERY	1	显示恢复选项
WER_SUBMIT_HONOR_RESTART	2	显示重新启动应用程序选项
WER_SUBMIT_QUEUE	4	直接将报告放入队列
WER_SUBMIT_SHOW_DEBUG	8	显示调试按钮
WER_SUBMIT_ADD_REGISTERED_DATA	16	向报告中加入注册数据
WER_SUBMIT_OUTOFPROMESS	32	产生一个新的进程来提交报告并等待结束
WER_SUBMIT_NO_CLOSE_UI	64	对于致命错误报告不显示关闭对话框
WER_SUBMIT_NO_QUEUE	128	不要将这个报告放入队列

续表

常量	位	说明
WER_SUBMIT_NO_ARCHIVE	256	不要对这个报告做归档处理
WER_SUBMIT_START_MINIMIZED	512	初始化界面是最小化的
WER_SUBMIT_OUTOFPOLICY_ASYNC	1024	产生一个新的进程来提交报告并立刻返回

其中 WER_SUBMIT_ADD_REGISTERED_DATA 标志用来在报告中包含使用 WerRegisterMemoryBlock 和 WerRegisterFile 所注册的内存块和文件。

14.4.4 典型应用

使用 WER 的一种情况是应用程序自己调用上面介绍的 API (WerReportCreate 和 WerReportSubmit 等) 来提交报告。对于这种情况，如果当提交时指定了进程外标志 (WER_SUBMIT_OUTOFPOLICY 或 WER_SUBMIT_OUTOFPOLICY_ASYNC)，那么系统会使用 WERMGR.EXE 来执行实际的报告发送工作。如果没有指定进程外标志，那么便会在当前进程的上下文中执行报告发送工作。

使用 WER 的另一种典型情况是当有未处理异常发生时，系统的 WER 服务 (WERSVC) 收到请求后会启动 WerFault.exe，然后在这个进程中产生并发送报告。清单 14-2 显示了 WerFault 进程的主线程产生和提交报告的过程。

清单 14-2 WerFault 进程产生和提交报告的过程

```
kd> kn 50
# ChildEBP RetAddr
00 001fd090 6ab7b0cc dbghelp!MiniDumpWriteDump+0xf2
01 001fd528 6ab8b0c5 wer!CProcessDump::CollectDump+0x33b
02 001fdde4 6ab8b28d wer!CDataCollection::AddDump+0x2d3
03 001fde00 6ab8227c wer!CDataCollection::AddDumps+0x55
04 001fde70 6ab8252d wer!CWatson::AddReportToQueue+0x1e0
05 001fde90 6ab834ae wer!CWatson::SubmitReportToQueue+0xb7
06 001fe128 6ab79fff wer!CWatson::ReportProblem+0x587
07 001fe138 6ab6aba1 wer!WatsonReportSend+0x1e
08 001fe154 6ab6af6d wer!CDWInstance::WatsonReportStub+0x17
09 001fe178 6ab65ee3 wer!CDWInstance::SubmitReport+0x21e
0a 001fe19c 6b576d4a wer!WerReportSubmit+0x6d
0b 001ff0c8 6b5773fe faultrep!CCrashWatson::GenerateCrashReport+0x5e3
0c 001ff360 6b574e31 faultrep!CCrashWatson::ReportCrash+0x374
0d 001ff860 008bda68 faultrep!WerInitiateCrashReporting+0x304
0e 001ff898 008b620d WerFault!UserCrashMain+0x14e
0f 001ff8bc 008b658a WerFault!wmain+0xbff
10 001ff900 7752436e WerFault!_initterm_e+0x163
11 001ff90c 778297bf kernel32!BaseThreadInitThunk+0xe
12 001ff94c 00000000 ntdll!_RtlUserThreadStart+0x23
```

栈帧#0d 中的 WerInitiateCrashReporting 函数是 Vista 版本的 FaultRep.DLL 新输出的函数，通过 GetProcAddress 可以取得这个函数的地址然后调用它。栈帧#05 的 AddReportToQueue 方法是将报告放在本地的队列中，事实上就是放在磁盘的如下目录中：Users\<name>\AppData\Local\Microsoft\Windows\wer\ReportQueue。

除了前面介绍的 API，WER 2.0 还定义了如下 API。

- WerGetFlags 和 WerSetFlags，取得或修改一个进程的错误报告设置。
- WerUnregisterMemoryBlock，注销错误发生时要收集的数据区（内存块）。
- WerUnregisterFile，注销错误发生时要收集的文件。
- WerAddExcludedApplication 和 WerRemoveExcludedApplication，增加或移除要进行错误报告的程序。

总而言之，WER 2.0 比 1.0 更加灵活和强大，允许程序员做更多定制，以用于我们自己开发的软件。

14.5 CER

CER 是 Corporate Error Reporting 的缩写，即企业错误报告。使用微软提供的 CER 工具（需要是微软的 Software Assurance 客户），企业可以建立自己的错误报告服务器（简称 CER 服务器）。具体包括。

- 定义用于存放错误报告的共享目录。
- 定义和发布错误报告搜集策略，CER 工具中包含了模板文件以简化这一操作。
- 决定将错误报告转发给微软的规则，比如是否将应用程序错误（或系统错误）发送给微软。
- 定制报告回应消息，可以定义一个 URL，在用户发送报告后都会被重新定向到这个 URL 指向的内部网页。

可以针对不同的软件定义不同的错误报告收集策略，表 14-4 给出了用于定制 Windows XP 和微软的常用软件时要使用的注册表表项（简称为 CER 策略表项）。

表 14-4 常用软件的 CER 策略表项

软件	定义 CER 策略的注册表表项
Windows XP	HKLM\Software\Policies\Microsoft\PCHealth>ErrorReporting\DW
Windows Server 2003	HKLM\Software\Policies\Microsoft\PCHealth>ErrorReporting\DW
Microsoft Office XP	HKCU\Software\Policies\Microsoft\Office\10.0\Common
MSN Explorer	HKCU\Software\Policies\Microsoft\msn6\watson
Microsoft Project 2002	HKCU\Software\Policies\Microsoft\Office\10.0\Common
Windows Media Player	HKCU\Software\Policies\Microsoft\MediaPlayer\Player\ExceptionHandling
Visio 2002	HKCU\Software\Policies\Microsoft\Office\10.0\Common
Internet Explorer 6 (Non Windows XP)	HKCU\Software\Policies\Microsoft\Office\10.0\Common
MapPoint, Streets&Trips, AutoRoute	HKLM\Microsoft\ErrorReporting\DW
SharePoint Portal Server	HKLM\Software\Policies\Microsoft\PCHealth>ErrorReporting\DW
SQL 2000 SP3	HKLM\Software\Microsoft\ErrorReporting\DW

在每个 CER 表项下，可以通过表 14-5 中列出的键值定义错误报告策略。

表 14-5 定义 CER 策略的注册表键值

键值名称	类型	含义
DWNeverUpload	DWORD	如果非零，则根本不上传错误报告数据，为 0 会向用户询问
DWAllowHeadless	DWORD	如果为 1 允许以静默方式 (Headless) 报告错误
DWNoFileCollection	DWORD	如果为 1，则取消所有与 WER 服务器间的数据会话
DWNoSecondLevelCollection	DWORD	如果为 1，则取消所有与 WER 服务器间的次级数据会话
DWFileTreeRoot	SZ	错误报告共享目录的 UNC 路径
DWTracking	DWORD	如果为 1，WER 会向日志文件中写入 tracking 信息
DWNoExternalURL	DWORD	如果为 1，WER 不会向用户显示微软发送的外部 URL
DWURLLaunch	SZ	如果存在并且为有效的 URL，WER 会在错误报告后的对话框 (图 14-7) 中显示这个 URL，这个 URL 比 WER 服务器发送的任何外部 URL 有更高的优先级
DWReporteeName	SZ	如果该项定义了有效的字符串，那么 WER 会使用它作为接受报告方的名字。否则会使用“Microsoft”

关于 CER 的具体实施方法，可以参考微软网站的帮助信息 (<http://www.microsoft.com/resources/satech/cer/new.asp>) 或参考文献 2。

14.6 本章总结

本章介绍 Windows 系统的错误报告发送机制，即 WER。前 3 节介绍的是 Windows XP 所引入的 WER 1.0 版本，第 14.1 节介绍了 WER 1.0 的客户端，第 14.2 节以发送系统错误报告为例介绍了 WER 1.0 客户端发送错误报告的完整过程，第 14.3 节介绍了 WER 的服务端，这个内容尽管我们是以 WER 1.0 为例来介绍的，但是大多数内容都可以推广到 WER 2.0，因为 WER 1.0 和 WER 2.0 之间的变化主要发生在客户端。第 14.4 节介绍了 WER 2.0，尤其是新引入的 API。第 14.5 节介绍了用于较大企业环境的 CER 机制。

参考文献

1. Sheryl Canter. Windows Error Reporting Under the Covers.
<http://www.windowsdevcenter.com/pub/a/windows/2004/03/16/wer.html>
2. Microsoft Visio 2002 Resource Kit (Chapter 11: Corporate Error Reporting).
<http://www.microsoft.com/technet/prodtechnol/Visio/visio2002/reskit/CH11.mspx>

日志

对于很多需要长时间不间断运行的服务器程序（service），日志（log）是管理员了解系统运行情况的最重要途径之一，如果有问题出现，事件日志也是用来发现和定位故障（troubleshooting）的第一手资料。对于普通的应用程序，也可以使用事件日志来记录重要的运行信息，满足软件调试和客户支持等需要。为了让系统中运行的各种软件可以简单方便地生成和管理日志数据，操作系统通常会建立一套公共的机制来存储、记录、浏览和维护事件日志，并将这些设施以 API 的形式公开给应用软件来使用。

概括来说，Windows 操作系统建立了两套日志机制。一套是 Windows Vista 引入的名为 CLFS（Common Log File System）的机制，另一套是从 NT 3.5 就支持的 Event Logging 机制，因为其内部函数大多是以 Elf（Event Log File）开头的，所以我们将其简称为 ELF。

本章将首先介绍日志机制的一般概念（15.1 节），而后分两个部分分别介绍 ELF 和 CLFS。第 15.2 节介绍 ELF 机制的架构，第 15.3 节介绍 ELF 组织数据的方法，第 15.4 节介绍如何查看和使用 ELF 日志。第 15.5 节介绍 CLFS 的组成和基本原理，第 15.6 节介绍 CLFS 的 API 和使用方法。

15.1 日志简介

简单来说，日志就是软件为自己写的日记，每一条日志记录用来记述一件事。基于这一基本原则，一条日志记录通常包含如下几个要素。

- 时间：所记录事件的发生时间，通常至少精确到分钟级别。
- 地点：用来定位所记录事件发生时的“位置信息”，通常包括机器名、进程 ID、线程 ID 等。
- 主体（来源）：即该事件的实施者，根据需要可以是服务名称、模块名称或类名和函数名。

- 事件：对所发生事件的描述。
- 类型(严重程度)：该事件的严重程度，可以分为信息(information)、警告(warning)和错误(error)3种，也可以根据需要分得更细。

那么软件应该如何记录事件日志呢？对于这个问题，最简单的回答是编写一个函数，在这个函数中打开一个文件，并把事件记录写入到这个文件中。然后在需要记录日志的地方调用这个函数。我们说这是最简单原始的事件日志记录方式。这种方式对于某些要求较低的情况或许是可以的。但是这种简单的做法存在着很多局限性，还有很多问题需要解决。

- 如何存放和命名日志文件，使用一个还是多个文件？让用户（管理员）如何找到这个/些文件？如何删除过时的文件和记录？
- 如何阅读日志文件？如何解释每一条事件日志的含义？
- 在软件本身已经出现错误的情况下，比如当前进程中已经无法再分配任何内存（分配请求总是失败），这种机制此时是否还能工作？如果不能，那么重要的日志信息就会丢失。
- 如何控制日志的安全性？是否允许所有用户都可以阅读所有日志记录？
- 如何同步来自多个线程的请求，如果有十几个线程同时发出记录请求，那么是否能在合理的时间内完成所有请求？

事件日志的重要地位和特殊职责迫使我们必须慎重考虑上面列出的每个问题，然而要满足所有这些要求，确实又不是件容易的事。这就使得让操作系统实现统一而完善的日志设施成为必要。

15.2 ELF 的架构

图 15-1 画出了 ELF 的基本架构，左侧是调用日志记录服务的应用程序进程，中间是承担日志服务的服务进程，右侧代表的是用来集中存储日志记录的日志文件。应用程序进程通过 RPC（Remote Procedure Call）机制与服务进程进行通信。但是 Windows 已经将通信的细节隐藏在 API 的内部实现中，相对应用程序只要调用 ELF 的 API 就可以了。

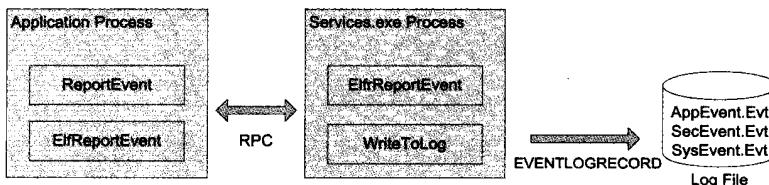


图 15-1 ELF 架构示意图

下面分几个部分来分别介绍 ELF，我们先从它的文件谈起。

15.2.1 ELF 的日志文件

ELF 使用磁盘文件来记录事件日志，每一类事件放在一个文件中。以 Windows XP 系统为例，系统中共定义了 3 类日志，分别是应用程序（Application）日志、安全（Security）日志和系统（System）日志，它们的文件分别是 AppEvent.Evt、SecEvent.Evt 和 SysEvent.Evt，这些文件都位于用于存储注册表文件和配置信息的 CONFIG 目录中：
%SystemRoot%\SYSTEM32\CONFIG\

Windows Vista 增加了 HardwareEvents 和 DFS Replication 等日志类别，并且为所有日志文件建立了一个单独的目录，即%SystemRoot%\SYSTEM32\winevt\Logs 目录，日志文件的扩展名也由.EVT 改为.EVTX。

每一类日志的配置信息存储在以下注册表键下：

HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\Eventlog

图 15-2 显示了应用程序日志的注册表设置，可以看到其中的 File 键值指向的就是 AppEvent.Evt 文件。

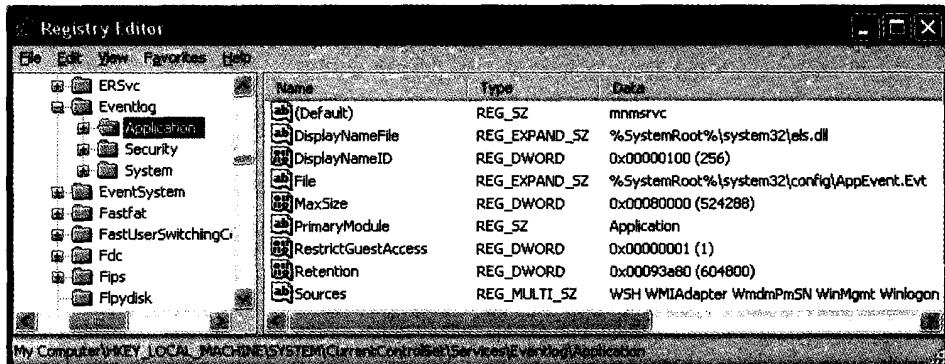


图 15-2 应用程序日志的注册表设置

MaxSize 键值用来指定日志文件的最大值，它的单位是字节，大小必须是 64KB (0x00010000) 的倍数。默认设置为 512KB (0x00080000)。PrimaryModule 键值用来指定使用该类日志的模块类别。

RestrictGuestAccess 键值如果为 1，则禁止 Guests 和普通用户账号访问该类日志。在 Windows 2000 中安全类日志的默认设置为 1，应用和系统类日志默认值为 0（允许访问）。在 Windows XP 和 Vista 中，这 3 类日志的默认值都为 1。

Retention 键值用来指定自动覆盖旧日志的期限，0x00000000 代表根据需要覆盖，0xffffffff 代表不覆盖。其它值代表可以覆盖超过该值所指定秒数的日志记录。默认值为 0x00093a80 (7 天)。

Windows Vista 以前，Sources 键值用来保存注册的事件源（Event Source）列表。

事件源用来标识日志记录的来源（报告者），ELF 在显示日志记录时需要根据事件源来格式化日志信息。Sources 中的每个字符串代表一个事件源，在同一个表键下，会有一个以该事件源命名的子键来进一步描述这个事件源（稍后介绍），Vista 直接枚举这些子键，不再使用 Sources 键值。

以上键值修改后需要重新启动才能生效。另外，从启动到关闭，系统都以独占方式（Exclusive）使用日志文件，因此删除或任何其它写访问都会被拒绝。

15.2.2 事件源

在向一个事件日志中写日志前，应该先注册一个事件源，事实上，就是在事件日志的注册表键下建立一个子键，并加入合适的键值。例如，以下是在应用程序日志中注册的 Application Hang 事件源：

```
位置: HKLM\SYSTEM\CurrentControlSet\Services\EventLog\Application\Application
Hang
键值: EventMessageFile      %SystemRoot%\System32\faultrep.dll
      TypesSupported     0x00000007
```

其中 EventMessageFile 键值（REG_EXPAND_SZ）用来指定这个事件源的消息文件（Message File）的位置和名称。消息文件就是包含消息资源（message resource）的文件，消息资源与我们熟悉的字符串资源和对话框资源的性质是相同的，只不过它的作用是充当模板来格式化日志事件或其它消息。与对话框和字符串资源一样，消息资源既可以存储在 EXE 文件中，也可以存储在 DLL 或其它有效的 PE 文件中。消息资源是由消息源文件（通常以.MC 为后缀）通过专门的消息编译器（Message Compiler）编译成二进制格式 (.BIN)，然后再链接到 EXE 或 DLL 文件中的。如果一个事件源有多个消息文件，那么用分号分开。每个事件源必须至少有一个消息文件，否则 ELF 在显示它的日志记录时会给出类似如下内容的错误提示：

```
The description for Event ID ( 0 ) in Source ( AdvDbgEvtLogger ) cannot be found. ....
The following information is part of the event: Hello ELF.
```

TypesSupported 键值（REG_DWORD）用来指定该事件源所支持的事件类型，可以包含如下标志位：EVENTLOG_ERROR_TYPE（0x0001），EVENTLOG_WARNING_TYPE（0x0002），EVENTLOG_INFORMATION_TYPE（0x0004），EVENTLOG_AUDIT_SUCCESS（0x0008），EVENTLOG_AUDIT_FAILURE（0x0010）。

除了以上键值，对于使用事件类属的事件源还可以通过 CategoryCount 键值（REG_DWORD）定义该事件源支持的事件类属（categories）个数，通过 CategoryMessageFile 键值（REG_EXPAND_SZ）指定类属消息文件位置和名称。对于使用参数文件的事件源，可以通过 ParameterMessageFile 键值（REG_EXPAND_SZ）指定参数文件位置和名称（多个文件用分号分开）。

15.2.3 ELF 服务

在 Windows 系统默认启动的服务中包含了负责事件日志的服务，其名称是 Event Log，我们将其简称为 ELF 服务。在 Windows 2000 之前，ELF 服务是一个单独的执行文件 EventLog.EXE。从 Windows 2000 开始，ELF 服务运行在 SERVICES.EXE 进程中。日志服务是自动启动的，而且是不可停止的。在任务管理器中也不允许杀掉 Services.EXE 进程，如果使用其它工具（如 WinDBG 或 kill.exe）强行杀掉它，那么系统检测到后会自动关机。

ELF 服务的职责是管理和维护事件日志文件，并通过 RPC 机制向应用程序提供各种日志服务，包括添加删除日志记录，获取日志信息，备份日志文件等，下一节我们将介绍其细节。

15.3 ELF 的数据组织

在理解 ELF 组织数据的方法之前，有必要先思考一个问题。作为一种通用的日志记录机制，必须要考虑到不同的事件需要记录的信息量和格式可能是大相径庭的，简单的可能只是一句话（一个字符串），复杂的可能包含多个文本、多个数值和数据附件。

那么如何以一种统一的方式来组织和存储不同信息量和不同结构的事件记录呢？大家可以很容易想出两种做法。

- 方法 A：将所有不同结构的事件格式化为字符串后统一存储，也就是将变化的内容通过格式模板格式化后以统一的字符串形式写入到日志文件中。
- 方法 B：将每个事件的格式信息抽象出来单独存储，在日志文件中只要记录每个事件实例的具体数据。也就是将事件的格式与实际数据分别存储，二者通过一个 ID 联系起来。以下事件为例：Faulting application msdev.exe, version 6.0.8168.2, faulting module devprj.pkg, version 6.0.8447.0, fault address 0x00003b42。其格式模板为“Faulting application %1, version %2, faulting module %3, version %4, fault address 0x%5.”。

方法 A 是比较简单做法，但是这样做的一个缺点是要重复记录格式信息，使日志文件中存储大量冗余数据。ELF 采用的是方法 B。事件的格式信息存储在消息文件（message file）中，日志文件只存储每个事件描述中的变化部分。

15.3.1 日志记录

ELF 是以表格的形式来存储日志记录的，表格的每一行对应于一条日志记录，存储着这条记录的基本信息和附属信息的偏移地址。可以使用 EVENTLOGRECORD 结构来描述 ELF 文件中的数据表结构，每个字段相当于表格的一列（见清单 15-1）。

清单 15-1 EVENTLOGRECORD 结构

```

typedef struct _EVENTLOGRECORD {
    DWORD Length;           // 结构长度, 以字节为单位
    DWORD Reserved;         // 保留
    DWORD RecordNumber;     // 记录号
    DWORD TimeGenerated;    // 产生本结构的时间
    DWORD TimeWritten;      // 写入时间, 即日志服务在写入日志文件前的时间
    DWORD EventID;          // 事件 ID
    WORD EventType;         // 事件类型
    WORD NumStrings;        // 在 StringOffset 偏移处包含的字符串个数
    WORD EventCategory;     // 事件类属
    WORD ReservedFlags;     // 保留标志
    DWORD ClosingRecordNumber; // 保留
    DWORD StringOffset;     // 字符串偏移
    DWORD UserSidLength;    // 用户 SID 长度
    DWORD UserSidOffset;    // 用户 SID 偏移
    DWORD DataLength;        // 数据附件 (raw data) 的长度
    DWORD DataOffset;        // 数据附件的偏移
} EVENTLOGRECORD, *PEVENTLOGRECORD;

```

以上数据结构没有包含事件源和计算机名信息，这是因为计算机名和事件源是在上一层次记录的。当打开一个事件日志（句柄）时就指定了这些信息，之后的所有操作都是相对这一日志的。

15.3.2 添加日志记录

下面介绍如何向一个事件日志中添加日志记录，也就是写日志数据。

第一步应该在注册表中注册事件源，方法是调用操作注册表 API 并按照上文介绍的规则来添加子键和键值，本章的示例项目 EventLog（code\chap15\eventlog）将这些操作归纳到 AddEventSource 函数中。

第二步是调用 ELF 的 RegisterEventSource API 来取得事件源句柄。

```
HANDLE RegisterEventSource( LPCTSTR lpUNCServerName, LPCTSTR lpSourceName);
```

参数 lpUNCServerName 用来指定按 UNC 规范（Universal Naming Convention）表示的机器名，如果是操作本机的日志文件，那么只要指定 NULL。lpSourceName 参数用来指定事件源名称，应该与第一步中注册的子键名一致。如果事件源名称在指定机器的注册表中不存在，那么系统会使用默认的应用类日志下的 Application 事件源。但由于 Application 事件源没有定义消息文件，所以当察看该类事件时，会得到前面介绍过的错误提示。

完成以上两步后，便可以调用 ELF 的 ReportEvent API 来添加日志记录了。

```
BOOL ReportEvent( HANDLE hEventLog, WORD wType, WORD wCategory,
    DWORD dwEventID, PSID lpUserSid, WORD wNumStrings,
    DWORD dwDataSize, LPCTSTR* lpStrings, LPVOID lpRawData);
```

其中，hEventLog 是使用 RegisterEventSource 得到的事件源句柄。wType 用

来指定事件的类型，可以为如下常量之一：EVENTLOG_SUCCESS(0x0000)、EVENTLOG_AUDIT_FAILURE(0x0010)、EVENTLOG_AUDIT_SUCCESS(0x0008)、EVENTLOG_ERROR_TYPE(0x0001)、EVENTLOG_INFORMATION_TYPE(0x0004)、EVENTLOG_WARNING_TYPE(0x0002)。参数 wCategory 用来指定事件在事件源中的类属号，其分类规则是应用程序所定义的。参数 dwEventID 用来指定事件的 ID 号，ELF 通过这个 ID 值来定位这条日志记录所对应的格式信息。ID 号的编排也是应用软件定义的。参数 lpUserSid 用来指定用户的安全标识，可以为 NULL。参数 wNumStrings 用来指定 lpStrings 所指向的字符串数组所包含的字符串指针个数。类似的，参数 dwDataSize 用来指定参数 lpRawData 所指向的原始数据缓冲区的长度。

那么，ReportEvent 是否直接把日志数据写到日志文件中的呢？答案是否定的，事实上，它只是通过 RPC 机制将调用转发给 ELF 服务。清单 15-2 显示了 ReportEvent 申请添加事件日志的内部过程。

清单 15-2 ReportEvent API 的工作过程

```
0:000> kn
# ChildEBP RetAddr
00 0012f0b0 78002fb5 RPCRT4!OSF_CCALL::SendReceive+0x32 // 标准协议层
01 0012f0b8 78002fdd RPCRT4!I_RpcSendReceive+0x20 // 与通信层的接口
02 0012f0c8 7807955e RPCRT4!NdrSendReceive+0x28 // 准备发送
03 0012f494 77de05a1 RPCRT4!NdrClientCall2+0x1ca // 将函数调用打包
04 0012f4a4 77de057e ADVAPI32!ElfrReportEventA+0x14 // 发起远程调用
05 0012f51c 77de04c4 ADVAPI32!ElfReportEventA+0x5a // ELF 函数
06 0012f588 00401a05 ADVAPI32!ReportEventA+0xce // 调用 API
```

栈帧#06 中的 ReportEventA 是 ReportEvent API 的 ANSI 版本，“!”号前的模块名说明它是实现 ADVAPI32.DLL 中的。ReportEventA 函数内部先调用了 ElfReportEventA 函数，ADVAPI32.DLL 输出了这个函数，但是没有文档化。ElfReportEventA 内部又调用 ElfrReportEventA 函数将所有请求信息封装为 RPC 消息，然后使用 RPC 函数发起调用，其中的 NDR 是 Network Data Representation 的缩写，OSF 是 Open Software Foundation 组织的简称，是 RPC 标准的制定组织。

清单 15-3 中的函数调用序列显示了 ELF 服务收到调用后的工作过程。首先，RPC 机制会将函数调用请求分发给 eventlog.dll 中的 ElfrReportEventA 函数，经过一系列操作后 WriteToLog 函数真正将事件日志写入到日志文件中，其细节从略，感兴趣的读者可以使用 WinDBG 附加到 services.exe 进程上，然后对 eventlog!WriteToLog 函数设置断点。值得提醒的是，最好在测试机器上作这样的调试，防止 Services 进程被中断后导致系统重启而造成损失。

清单 15-3 事件日志服务内部将日志写入文件的过程

```
0:009> kn
# ChildEBP RetAddr
00 009af708 758a1f39 ntdll!NtWriteFile+0xa // 真正写文件
01 009af754 758a1def eventlog!WriteToLog+0x46 // 写入日志
02 009af818 758a1cad eventlog!PerformWriteRequest+0x4db // 执行写请求
```

```

03 009af824 758a223a eventlog!ElfPerformRequest+0x7c      // 分发请求
04 009af888 758a2490 eventlog!ElfrReportEventW+0x2cf      // UNICODE 版本
05 009af8dc 780038f7 eventlog!ElfrReportEventA+0xc1      // 执行日志函数
06 009af928 780791a5 RPCRT4!Invoke+0x30                  // 调用目标函数
07 009afdf3c 780795aa RPCRT4!NdrStubCall2+0x1fb            // 
08 009afdf58 78002d28 RPCRT4!NdrServerCall2+0x17          // NDR 的服务端函数
09 009afdf8c 78002ca5 RPCRT4!DispatchToStubInC+0x38        // RPC 工作线程的分发函数

```

栈帧#07 和#08 中的 NDR 是 Network Data Representation 的缩写，是 RPC 技术中用来将函数调用封装成数据包的引擎（Marshalling Engine）。NDR 的客户端函数用来将函数调用打包成一个适合在网络上传输的消息包，比如清单 15-2 中的栈帧#03。NDR 的服务端函数负责将数据包解开还原为函数调用，如清单 15-3 中的栈帧#08。

因为应用程序进程和日志服务进程之间使用的是 RPC 方式，所以它们可以分别位于不同的机器上，这使得 ELF 可以很自然地支持跨机器操作，比如向另一台机器的日志文件中写记录，或者读取和备份其它机器上的日志数据。

15.3.3 API 一览

除了增加日志，ELF 还提供了其它几个 API 来读取、查询和清除日志记录，以及用于备份日志文件的 API。表 15-1 列出了 ELF 的所有 API。

表 15-1 ELF 的 API

名称	用途
RegisterEventSource	取得注册事件源的句柄
ReportEvent	添加日志记录
OpenEventLog	打开一个事件日志（文件）
ReadEventLog	从事件日志中读取记录
BackupEventLog	将指定的事件日志备份到指定的文件
ClearEventLog	清除事件日志
OpenBackupEventLog	打开一个备份的日志文件
NotifyChangeEventLog	注册一个用于接受日志变化事件的事件句柄
GetNumberOfEventLogRecords	取得指定事件日志所包含的记录数
GetEventLogInformation	读取事件日志的信息，比如是否已经写满
CloseEventLog	关闭事件日志
DeregisterEventSource	注销事件源
GetOldestEventLogRecord	取得日志中最老的那条记录的记录号

ELF 的 API 都是从 ADVAPI32.DLL 模块输出的。

除了供用户态使用的 API 外，驱动程序等内核代码可以调用 DDK 中公开的内核函数来使用 ELF 服务，比如调用 IoAllocateErrorLogEntry 分配日志表项，调用 IoWriteErrorLogEntry 来添加日志记录等。系统进程中专门负责错误日志的工作线程会将这些日志记录通过 LPC 端口发送给 ELF 系统服务。

15.4 察看和使用 ELF 日志

Windows 自带了事件察看器 (Event Viewer) 程序用来查看和管理事件日志记录。可以使用以下任一方法启动事件察看器。

- 在运行对话框或命令行执行 eventvwr.exe;
- 在运行对话框或命令行执行 eventvwr.msc;
- 在计算机管理 (Computer Management) 程序中, 选择事件察看器。

通过事件察看器的图形界面可以浏览所有事件, 对事件排序、过滤或搜索事件 (View 菜单下的 Filter 和 Find), 清除或将事件导出到文本文件中。双击某一条记录, 事件察看器会弹出类似图 15-3 所示的对话框来显示日志记录的详细信息。

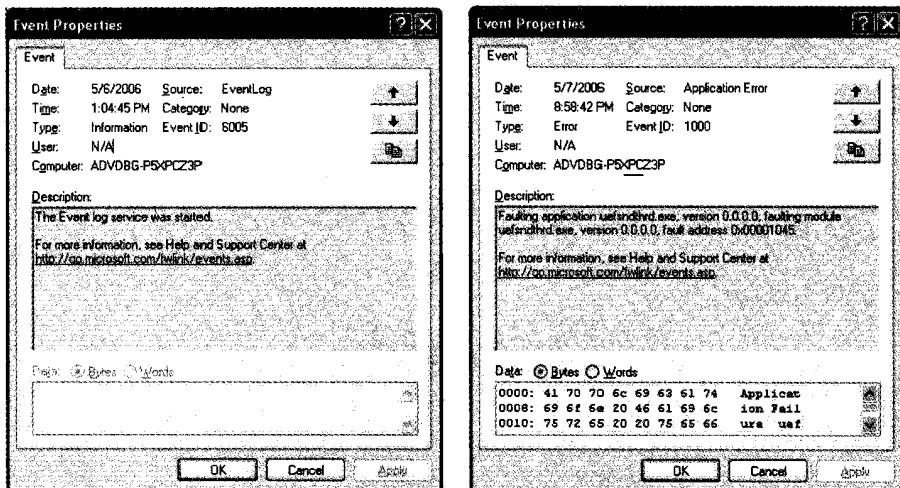


图 15-3 6005 号 (左) 和 1000 号 (右) 事件日志

图 15-3 左边是 6005 号事件 (Event ID 为 6005), 它是日志服务自己写入的。日志服务每次启动后, 都会写入该条日志, 因此可以通过这条记录来了解系统的启动时间等信息。6005 号事件的描述只包含一句话: The Event log service was started (事件日志服务启动), 而且该事件没有数据附件 (数据区是灰的)。右边是 1000 号事件, 描述的是应用程序错误。在 Windows XP 中, DWWIN.EXE 在弹出错误对话框前会调用 ReportEvent 写入这一事件。

除了事件本身的信息外, 在系统事件的描述栏中, 通常会有一个链接:

```
For more information, see Help and Support Center at  
http://go.microsoft.com/fwlink/events.asp.
```

点击该链接并同意发送信息后, 系统会启动“帮助与支持中心”程序连接网络,

以获取关于这一事件的更多信息。

当调试时，可以使用!evlog 命令来执行各种日志操作，包括显示日志文件的概要信息(!evlog info)、增加事件源(!evlog addsource)、备份(!evlog backup)和清理(!evlog clear)日志文件、读取(!evlog read)和添加(!evlog report)日志记录等。另外，使用!elog_str 命令可以非常方便地向应用程序日志中加入一条错误类型的日志。当内核调试时，可以使用!errlog 命令检查还没有写入到文件中的日志请求。

15.5 CLFS 的组成和原理

CLFS 是 Common Log File System 的简称，它是 Windows Vista 新引入的一种日志机制。与前面介绍的 ELF 相比，CLFS 的速度更快，可靠性也更高。本节将介绍 CLFS 的组成和基本原理，下一节将介绍在应用软件和驱动程序中如何使用 CLFS。

15.5.1 组成

CLFS 主要实现在两个模块中，一个是位于内核态的 CLFS.SYS，另一个是位于用户态的 CLFSW32.DLL。图 15-4 画出了这两个模块在系统中的位置和相互关系。

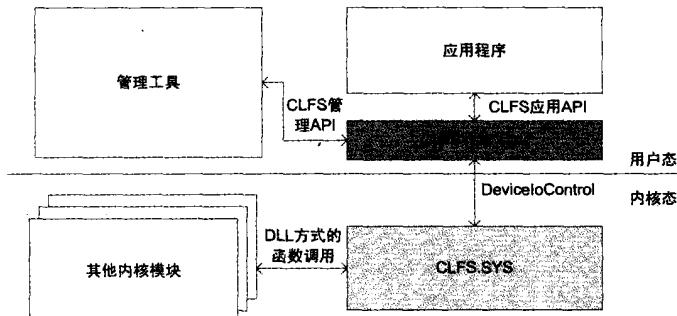


图 15-4 CLFS 的主要部件和与系统的交互方式

CLFS 的核心功能都是实现 CLFS.SYS 模块中的。CLFS.SYS 与系统的内核文件(NTOSKRNL.EXE)有着直接的相互依赖关系，因此是随着内核文件一起由系统的加载程序(WinLoad)在系统启动的早期加载到内存中的。CLFS.SYS 以 DLL 方式输出了一系列函数，都是以 Clfs 开头的，例如 ClfsCreateLogFile、ClfsAddLogContainer 等，为了行文方便，我们将它们称为 CLFS 公开函数。内核模块可以通过 DLL 方式直接调用 CLFS 公开函数，WDK 的文档中详细介绍了这些函数的原型和用法。

CLFSW32.DLL 是一个简单的用户态 DLL，它的主要作用是向用户态程序输出 CLFS 的应用程序编程接口(CLFS API)。CLFSW32.DLL 输出两类 API，一类是供管理工具来定义 CLFS 管理策略和注册回调函数参与 CLFS 事务的，另一类是供各种应

应用程序使用 CLFS 的日志服务的。前一类 API 的头文件是 Clfsmgmtw32.h，后一类是 Clfsw32.h。

大多数 API 都是通过 DeviceIoControl 来调用内核态的 CLFS.SYS 公开函数，只有少量 API 是实现在 CLFSW32.DLL 中的。

15.5.2 存储结构

为了具有高可靠性和好的性能，CLFS 的日志记录与它在物理介质（磁盘）中的存储结构有着直接的映像关系。这种映像关系也反映在顶层的 API 中。因此，要理解 CLFS API，必须对 CLFS 的底层存储结构有所了解。

首先，一个 CLFS 日志（Log）是由一个 BLF 文件和多个容器文件（Container）组成的。BLF 文件用来存储日志的全局信息，其大小通常为 64KB。容器文件是真正用来存储日志数据的，整个文件对应于物理介质上的一段连续存储空间，称为 Extent，其大小一定是存储介质的分配单位的整数倍。以硬盘为例，一个容器文件对应于磁盘上的一系列连续的磁盘扇区，其大小是扇区大小（512 字节）的整数倍。这样做的好处是使用原始的磁盘读写方法就可以将日志数据写到磁盘上，以便当系统崩溃或文件系统出现故障时也可以将日志保存下来。一个日志可以包含多个容器文件，它们的大小应该是相同的。图 15-5 画出了一个 CLFS 日志的示意图，图中左侧是 BLF 文件，右侧是两个容器文件。容器文件中均匀排列的那些矩形代表了存储介质中的最小分配单位（如扇区）。

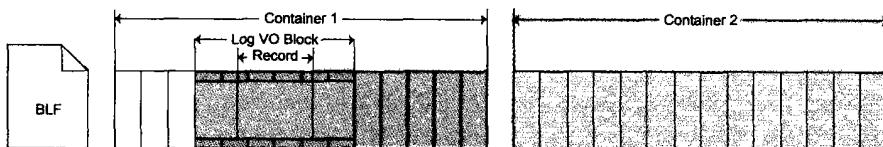


图 15-5 CLFS 的存储结构

BLF 文件是通过 `ClfsCreateLogFile` 函数或 `CreateLogFile` API 来创建的，其大小通常是 64KB。容器文件是通过 `ClfsAddLogContainer` 函数或 `AddLogContainer` API 来创建的，其大小是可以指定的。CLFS 要求一个日志至少要有两个容器文件才可以向其添加日志记录。例如，以下是系统的 KTM（Kernel Transaction Manager）日志的 BLF 文件和配套的两个容器文件：

08/16/2007	12:11 PM	65,536	KtmRmTm.blf
08/16/2007	12:11 PM	524,288	KtmRmTmContainer0000000000000000000000000001
08/16/2007	12:11 PM	524,288	KtmRmTmContainer0000000000000000000000000002

以上文件位于 Windows 系统的 system32\msdtc 目录中。

为了提高性能，在向 CLFS 日志写记录前，必须先在内存中分配一个编组区（Marshalling Area）。一个编组区对应于容器文件中的一组存储区，称为一个日志 IO

块 (Log I/O Block)。在成功创建编组区后，便可以向日志中写日志记录了，每个日志记录的长度可以是不同的。

15.5.3 LSN

CLFS 使用一个 64 位的整数 (ULLONG) 来唯一标识一条日志记录，称为日志序列号 (Log Sequence No)，简称 LSN。一个 LSN 由以下 3 部分组成。

- 容器标识，使用的是 LSN 的高 32 位，范围为 0~0xFFFFFFFF。
- IO 块偏移，必须是 512 的倍数，位于 LSN 的 9 到 32 位 (共 23 位)。
- 日志记录在 IO 块内的顺序号，位于 LSN 的低 9 位，范围为 0~512。

CLFS.SYS 中输出了一系列函数用来操纵 LSN，比如可以调用 ClfsLsnCreate 将以上 3 部分合成一个 LSN，使用 ClfsLsnContainer、ClfsLsnBlockOffset 和 ClfsLsnRecordSequence 分别取得 LSN 的 3 个部分之一。用户态的 CLFSW32.DLL 也实现并输出了类似的函数，并且函数名与驱动中的对应函数相比只是不带 Clfs。

15.6 CLFS 的使用方法

上一节我们介绍了 CLFS 的基本特征、模块组成和数据结构。本节将继续介绍如何在软件开发中使用 CLFS。用户态程序和内核态程序都可以使用 CLFS 来读写日志，前者使用 CLFSW32.DLL 所输出的用户态 API，后者使用 CLFS.SYS 以 DLL 方式输出的公开函数。用户态的 API 与内核态的 CLFS 公开函数非常类似，一个基本的规律是内核态的函数都是以 Clfs 开头的，而用户态的 API 去掉了这个前缀。例如创建 CLFS 日志的内核函数是 ClfsCreateLogFile，而用户态的 API 是 CreateLogFile。下面将以用户态的情况为例来介绍 CLFS 的用法，其中绝大多数内容可以很容易推广到内核态。

考虑到 MSDN 中已经描述了 CLFS API 的基本用法，但是目前还没有提供代码实例，因此作者编写了一个名为 HiClfs 的小程序，用来演示操作 CLFS 日志的重要步骤，包括创建日志文件，添加容器文件，创建编组区，添加日志记录和读取日志记录。下面便结合这个小程序的代码来讲解操作 CLFS 的重要步骤。

15.6.1 创建日志文件

应该调用 CreateLogFile API 来创建或打开一个 CLFS 日志。例如以下便是 HiCLFS 小程序所使用的代码：

```
hClfsLog=CreateLogFile(szLogFile, GENERIC_WRITE|GENERIC_READ,
0,NULL,OPEN_ALWAYS,0);
```

其中 szLogFile 用来指定要创建或打开的 CLFS 日志，其格式为：

```
LOG:<log name>[::<log stream name>]
```

中间的 log name 应该为一个有效的文件路径和日志文件的主文件名，系统会根据这个参数来决定日志的 BLF 文件名称和位置。举例来说，如果将这个参数指定为 LOG:c:\logs\advdbg，那么系统会在 c:\logs 目录中创建一个 advdbg.blf。

按照一个日志中所包含的日志流个数，CLFS 日志分为专用（dedicated）日志和复合（multiplexd）日志两种，前者只包含一个数据流，后者可以包含多个数据流，可以让多个应用程序各用一个日志流而共享一个 CLFS 日志。如果要创建复合日志，那么在创建日志时就要指定日志流的名称（log stream name）。

HiCLFS 小程序使用自己的程序模块路径和名称作为日志的名称，因此它会在和自己的 EXE 文件同位置的地方创建一个名为 HiCLFS.BLF 的文件。

15.6.2 添加 CLFS 容器

CreateLogFile 函数成功返回后，应该创建容器文件。CLFS 要求一个日志至少要有两个容器文件后，才能向其写入日志记录。添加容器的方法有两种，一种是调用 AddLogContainer 或 AddLogContainerSet API，另一种是调用 CLFS 管理 API，SetLogFileSizeWithPolicy。使用前一种方法时应该指定容器文件的路径和名称，例如：

```
lpszContainerPath=_T("%BLF%\CLFSCON01");
if(!AddLogContainer(hClfsLog,&cbContainer,lpszContainerPath,NULL))
```

其中%BLF%是 CLFS 定义的一个别名，用来指代 BLF 文件所在的路径。以上调用成功后，HiCLFS.BLF 所在的目录中会增加一个 CLFSCON01 的文件，其大小是由 cbContainer 参数所指定的。我们指定的是 512*1024*2，因此创建好的文件是 1048576 字节，二者相等。CLFS 要求每个容器的大小至少是 512KB。

如果使用后一种方法，那么可以这样调用 SetLogFileSizeWithPolicy API：

```
ULLONGLONG ullContainers=2;
ULLONGLONG ullResultingSize=0;
if(!SetLogFileSizeWithPolicy(hClfsLog, &ullContainers, &ullResultingSize))
```

系统会自动为容器文件命名，其名称通常类似如下的形式：

```
HiCLFSContainer00000000000000000000
```

15.6.3 创建编组区

在向 CLFS 日志写日志记录前，还需要创建一个特殊的内存缓冲区，称为编组区（Marshalling Area）。编组区的主要作用是可以缓存多条日志记录，以便减少访问外部存储器（磁盘）的次数。

以下是 HiCLFS 程序用来创建编组区的代码：

```
if(!CreateLogMarshallingArea(hClfsLog,NULL,NULL,NULL,
512, 2,2,&pMarshalContext))
```

其中 3 个 NULL 参数用于指定内存分配和释放回调函数, 以及回调函数的上下文参数。512 是这个缓冲区可以编组的最大日志记录大小, 后面的两个 2 分别是读写缓冲区个数。以上调用成功后, 系统会将一个上下文结构的地址存入到 pMarshalContext 参数中, 有了这个结构后, 就可以向日志中添加日志记录了。

15.6.4 添加日志记录

添加日志的方法是调用 ReserveAndAppendLog API, 调用前应该把要写入日志的数据登记到 CLFS_WRITE_ENTRY 结构中。这个 API 允许一次写入多条记录。以下是 HiCLFS 程序中的相关代码:

```
_snprintf(szLogBuffer, MAX_PATH, _T("A testing log record at tick 0x%x"),
          GetTickCount());
ClfsEntry.Buffer= szLogBuffer;
ClfsLSN=CLFS_LSN_INVALID;
ClfsEntry.ByteLength=(_tcslen(szLogBuffer)+1)*sizeof(TCHAR);
if(!ReserveAndAppendLog(
    pMarshalContext,
    &ClfsEntry,
    1,
    &ClfsLSN,
    &ClfsLSN,
    0,
    0,
    CLFS_FLAG_NO_FLAGS,
    &ClfsLSN,
    NULL
    ))
{
    // PVOID pvMarshal,
    // PCLFS_WRITE_ENTRY rgWriteEntries,
    // ULONG cWriteEntries,
    // PCLFS_LSN plsnUndoNext,
    // PCLFS_LSN plsnPrevious,
    // ULONG cReserveRecords,
    // LONGLONG rgcbReservation[],
    // ULONG fFlags,
    // PCLFS_LSN plsn,
    // LPOVERLAPPED pOverlapped
}
```

以上代码将 szLogBuffer 所指向的字符串写到日志中。如果函数成功返回, 倒数第二个参数保存了新添加记录在日志中的 LSN。

向 CLFS 添加日志记录后, 系统会在达到预先定义的上限后自动将缓冲区中的数据冲转到磁盘中, 应用程序也可以调用 FlushLogBuffers API 来强制冲转。

结束所有日志操作后, 应该调用 CloseHandle API 来关闭日志。

15.6.5 读日志记录

读 CLFS 日志前也应该先打开日志文件和创建编组区。然后调用 ReadLogRecord 来开始读取日志记录:

```
if(!ReadLogRecord(pMarshalContext, // 编组缓冲区上下文结构
                  &(li.BaseLsn), // 要读取的起始记录的 LSN
                  ClfsContextForward, // 后续的读取模式
                  &pReadBuffer, &ulSize,&ulRecordType, &lsnUndoNext,&lsnPrevious,
                  &pReadContext,NULL))
```

其中 ClfsContextForward 和 pReadContext 参数都是用于继续读取其它记录的, ClfsContextForward 的意思是接下来会继续向前读取其它记录, 其它允许值还

有 ClfsContextPrevious（向后读取）和 ClfsContextUndoNext（读取 Undo 链表的下一个记录）。

接下来，可以循环调用 ReadNextLogRecord API 以刚才指定的读取模式（方向）遍历当前日志：

```
while(ReadNextLogRecord(pReadContext, &pReadBuffer,
    &ulSize,&ulRecordType, NULL, &lsnUndoNext,&lsnPrevious,&lsnRecord,NULL))
```

读取完成后，应该调用 TerminateReadLog (pReadContext) 来释放与 pReadContext 相关联的资源。

15.6.6 查询信息

可以调用 GetLogFileInformation API 来查询日志的详细信息。

```
CLFS_INFORMATION li;
ULONG ulSize=sizeof(li);
if(!GetLogFileInformation(hClfsLog, &li,&ulSize))
```

表 15-2 列出了 HiCLFS 程序在不同阶段执行查询调用而得到的结果，每一行代表一种属性，第二列开始的每一列代表一个时间点。

表 15-2 执行各种操作后的 CLFS 日志信息（示例）

属性/时间	新创建后	添加容器后	创建保护区后	附加一记录后	冲转后
BaseFileSize	0x10000	0x10000	0x10000	0x10000	0x10000
ContainerSize	0	0x100000	0x100000	0x100000	0x100000
TotalContainers	0	2	2	2	2
FreeContainers	0	1	1	1	1
TotalAvailable	0	0x200000	0x200000	0x200000	0x200000
CurrentAvailable	0	0x200000	0x1ff800	0x1ff800	0x1ff600
TotalReservation	0	0	0x800	0x800	0x800
LastLsn	0	0	0	0	0x200
LastFlushedLsn	0	0	0	0	0x200

表 15-2 中 BaseFileSize 的含义是基础文件（即 BLF 文件）的大小，ContainerSize 是指容器文件的大小，这里列出的是调用 AddLogContainer 创建了两个容器后的取值，TotalContainers 是总的容器数，FreeContainers 是空闲的容器数，TotalAvailable 是总的可用空间（字节），CurrentAvailable 是当前的可用空间，TotalReservation 是总的保留空间（字节），LastLsn 代表小于指定 LSN 的日志记录已经被添加到日志记录，LastFlushedLsn 代表小于指定 LSN 的日志记录已经被冲转到磁盘。除了表 15-2 列出的属性外，还有以下几个属性的值是稳定的，比如 SectorSize=512、FlushThreshold=40000、MinArchiveTailLsn=0x0、BaseLsn=0 等。

15.6.7 管理和备份

管理工具软件可以通过 CLFS 的管理类 API 来注册回调函数参与 CLFS 日志的管理，注册回调函数的方法是调用 `RegisterManageableLogClient` API，注销的方法是 `DeregisterManageableLogClient`。使用 `ReadLogNotification` API 可以读取关于日志的通知消息。

可以调用 `InstallLogPolicy`、`RemoveLogPolicy` 和 `QueryLogPolicy` 来安装、删除或查询日志策略，包括占用空间的上限，是否允许自动增长等。

如果要对 CLFS 日志进行备份，那么应该先调用 `PrepareLogArchive` API，这个 API 会为日志产生一个快照（snapshot），并建立一系列备份描述符，然后返回一个上下文结构。有了这个结构后就可以反复调用 `GetNextLogArchiveExtent` API 来备份日志记录了。备份结束后应该调用 `TerminateLogArchive` 来释放有关的资源。

15.7 本章总结

日志是一种简单而有效的辅助调试手段，利用日志信息可以追溯软件的执行历史，分析其执行路线，这对于寻找软件故障的根源和分析系统的运行情况都有着重要意义。

本章的前半部分（15.2~15.4 节）介绍了 Windows 操作系统中传统的事件日志机制（ELF），后半部分介绍了最近引入的 CLFS 机制。CLFS 是对文件系统的扩展，其支持来源于 Windows 内核和驱动程序，因此具有更好的可靠性和更高的性能。

Windows Vista 引入了一套代号为 Crimson 的 API，尽管 SDK 中将其称为 Windows Event Log，但实际上它并不是建立在本章介绍的事件日志基础上的。而是基于下一章将介绍的事件追踪机制（ETW）构建的，因此我们将在下一章介绍它。

参考文献

1. Andreas Schuster . Introducing the Microsoft Vista event log file format.
2. Seth Livingston. Fast and Flexible Logging with Vista's Common Log File System.

事件追踪

简单来说，事件追踪（Event Tracing）要解决的问题就是记录软件运行的动态轨迹，包括代码的执行轨迹和变量的变化轨迹。本章将先介绍事件追踪的目标和基本特征（16.1 节），而后详细介绍 Windows 系统的事件追踪机制（ETW），包括架构（16.2 节）、组成（16.3 节至 16.5 节）、格式文件（16.6 节）和应用（16.7 节至 16.9 节）。

16.1 简介

概括来讲，事件追踪的目标就是要把软件执行的踪迹以一种可以观察的方式输出出来。与上一章介绍的日志机制相比，事件追踪机制更关心软件的“变化和运动过程”。如果说日志只要记录下软件中重要事件的结果，那么事件追踪则要记录下导致这一结果的完整过程。因此要求事件追踪机制必须能够适应频繁的数据输出和庞大的数据量，这通常是普通的日志机制所难以胜任的。由于这一原因，事件追踪机制通常是以二进制方式而不是文本来传输和记录信息的。另外，事件追踪机制的目标“读者”主要是开发人员，而事件日志的主要对象还包括系统管理员。这一差异导致了事件追踪信息通常会包含更多的技术细节，比如函数名称、变量取值等。最后一个差异是，日志机制通常是始终开启的，而事件追踪机制在软件正常运行时通常是关闭的，只在观察和分析时才会开启。

事件追踪机制不依赖于软件的调试版本，也不需要使用调试器。因此是了解软件执行情况的既简单而又有效的途径。特别在以下任务（领域）中有重要的应用。

性能分析：分析程序的执行轨迹，寻找热点和瓶颈，确定优化目标。

产品期调试：在用户环境中动态开启事件追踪机制，观察执行过程，寻找故障原因。因为在产品期调试中，通常难以建立基于调试器的调试环境进行跟踪——用户可能无法接受你在他/她的机器上安装任何调试器或不愿意看到漫长而又根本无法理解的调试过程。这使得事件追踪成为产品期调试的一个有效手段。

单元测试或自动测试：记录下测试时的执行情况，为分析异常或错误提供依据。

为了使事件追踪机制能更好地满足以上应用的需要，好的事件追踪机制应该满足如下要求。

高效性：开销小 (low overhead)，只要占用少量的 CPU 时间和内存资源就可以支持频繁地输出信息。

动态性：可以动态开启或关闭，不需要重新启动系统或软件本身。

灵活性：可以方便定义信息的输出目标，比如重定向到不同位置的文件、调试器或其他观察工具。

选择性：可以选择性开启要追踪的模块，而且最好可以定义过滤条件来控制信息的详细程度。

易用性：首先要让程序员可以非常容易地在代码中嵌入用于追踪的代码，其次就是在观察追踪信息时要简单易行，不需要复杂的工具或繁琐的步骤。

C 标准库中的 `printf()` 函数、Windows SDK 中的 `OutputDebugString()` 函数和 DDK 中的 `DbgPrint() / DbgPrintEx()` 函数为实现事件追踪提供了简单的支持，可满足简单的应用需要，但是它们都有明显的局限性：`printf()` 通常只适用于控制台程序；`OutputDebugString()` 的效率很低，不适用于输出频繁而且大量的事件（参见 10.7 节）；`DbgPrint() / DbgPrintEx()` 需要使用中断 (INT 3) 发起请求（调用 `DbgBreakPointWithStatus`），效率也很低。

为了更好地支持事件追踪，从 Windows 2000 开始，Windows 操作系统提供了一套完整的事件追踪机制（包括内部实现、API 和辅助工具），称为 ETW (Event Tracing for Windows)。使用 ETW，程序员可以非常简单地将这一强大的事件追踪机制插入到自己的软件应用中。ETW 很好地满足了前面提到的动态性、灵活性、选择性、高效性和易用性要求。

- ETW 将追踪消息先输出到由系统来管理和维护的缓冲区中，然后异步写入到追踪文件或送给观察器。如果系统崩溃，崩溃前没有写入到文件中的信息会记录到转储文件 (dump) 中。
- ETW 的追踪消息是以二进制形式传输和存储的，所有格式信息存放在私有的消息文件中，这样可以防止软件本身的保密技术泄露。
- ETW 机制支持动态开启，没有开启时，开销几乎可以忽略（只需判断一个标志）。

那么，ETW 是如何实现的呢？下面我们就从 ETW 的基本架构开始探索它的组成和工作机理。

16.2 ETW 的架构

从架构上看，ETW 使用了经典的 Provider/Consumer/Controller 设计模式：使用 ETW

技术输出追踪消息的目标程序是提供器（Provider），接收和察看追踪消息的工具（如 TraceView）或文件是消耗器（Consumer），负责控制追踪会话（Session）的工具软件是控制器（Controller）。图 16-1 画出了以上 3 个角色的协作模型。

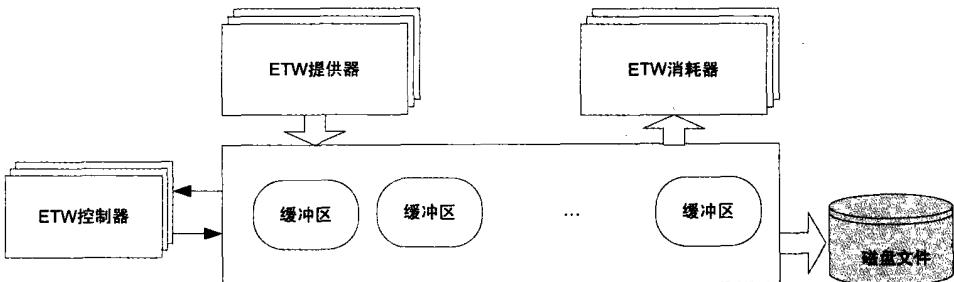


图 16-1 ETW 协作模型

用于支持 ETW 传输追踪信息的通信连接被称为 ETW 会话（Session）。系统会为每个 ETW 会话维护一定数量的缓冲区用以缓存发送到该会话的 ETW 消息。ETW 控制器可以定制缓冲区的大小。系统会定期地（每秒钟一次）将缓冲区中的信息发送给 ETW 消耗器或冲转（flush）到追踪文件（Trace Files）中。当缓冲区已满或会话停止时，系统会提前冲转缓冲区的内容。ETW 控制器也可以显式要求冲转缓冲区。当系统崩溃时，系统会将缓冲区的内容放入到 DUMP 文件中。ETW 会话可以在没有提供器或消耗器的情况下存在。Windows 2000 最多支持 32 个 ETW 会话，从 Windows XP 开始，最多支持 64 个 ETW 会话。但有两个是系统专用的，它们分别是启动早期使用的 Global Logger Session 和记录系统预定义事件的 NT Kernel Logger Session。

从实现角度来看，ETW 基础设施的核心部分是实现在内核模块中的，它是作为 WMI 的一个部分来实现的。在内核调试会话中执行以下命令，便可以看到 ETW 的内核函数。

```

1kd> x nt!Wmi*trace*
805f394e nt!WmiFlushTrace = <no type information>           // 冲转追踪信息
80530d8c nt!WmiTraceMessage = <no type information>          // 输出追踪信息
805f7bcc nt!WmipEnableDisableTrace = <no type information>   // 启动或停止追踪
805f719e nt!WmipTraceRegistry = <no type information>         // 追踪注册表调用
805f7080 nt!WmipTraceLoadImage = <no type information>        // 追踪模块加载
8065a9c2 nt!WmiTraceUserMessage = <no type information>       // 接收用户态的信息
...

```

以上函数有一部分是在 DDK 中公开的，用来供内核态的驱动程序直接调用。对于用户态的应用程序代码，ADVAPI32.DLL 模块输出了一系列 API，SDK 中包含了详细描述。因此，既可以在应用程序中使用 ETW，也可以在驱动程序中使用 ETW，而且其用法是基本一致的。下面 3 节我们将以应用程序的情况为例来分别介绍提供、控制和消耗 ETW 事件（消息）的方法。

16.3 提供 ETW 消息

ETW 通过 GUID 来标识系统中的 ETW 提供器。因此，ETW 提供器应该通过 RegisterTraceGuids API 向系统注册自己的 GUID。这样，ETW 控制器才可以通过 GUID 找到该提供器。

```
ULONG RegisterTraceGuids(
    WMIDPREQUEST RequestAddress,           // 回调函数
    PVOID RequestContext,                  // 回调函数的参数
    LPCGUID ControlGuid,                  // 标识此提供器的 GUID
    ULONG GuidCount,                      // TraceGuidReg 数组中包含的元素个数
    PTRACE_GUID_REGISTRATION TraceGuidReg, // 标识追踪事件 GUID 的数组
    LPCTSTR MofImagePath,                 // 保留未用, NULL
    LPCTSTR MofResourceName,               // 保留未用, NULL
    PTRACEHANDLE RegistrationHandle      // 返回句柄
);
```

注册成功后，如果有 ETW 控制器启动或停止该提供器，那么系统会调用 RequestAddress 参数指定的回调函数。在回调函数中，ETW 提供器可以通过参数中的请求代码来判断被调用的原因，如果是被启用，那么应该调用 GetTraceLoggerHandle API 取得 ETW 会话的句柄。清单 16-1 给出了 ETW 提供器回调函数的典型写法。

清单 16-1 ETW 提供器的回调函数

```
ULONG WINAPI MyControlCallback( WMIDPREQUESTCODE RequestCode, PVOID Context,
                                ULONG* Reserved, PVOID Buffer )
{
    if( RequestCode == WMI_ENABLE_EVENTS )           // 启用追踪
    {
        g_hTrace = GetTraceLoggerHandle( Buffer ); // 取得会话句柄
        g_dwFlags = GetTraceEnableFlags( Buffer ); // 读取控制器设置的启用标志
        g_dwLevel = GetTraceEnableLevel( Buffer ); // 读取控制器设置的启用级别
    }
    else if ( RequestCode == WMI_DISABLE_EVENTS ) // 禁止追踪
    {
        g_hTrace = NULL;                         // 将句柄设置为空
    }
    return 0;                                     // 返回值应该设置为 0
}
```

有了 ETW 会话句柄，便可以调用 TraceEvent API 来向 ETW 会话输出信息了。

```
ULONG TraceEvent( TRACEHANDLE SessionHandle,
                  PEVENT_TRACE_HEADER EventTrace);
```

其中，参数 SessionHandle 用来指定追踪会话句柄，即上面的回调函数所取得的值 (g_hTrace)，参数 EventTrace 用来指定存放追踪信息的内存缓冲区的起始地址，缓冲区应该以一个 EVENT_TRACE_HEADER 结构开始，后面跟随追踪事件的具体数据，即有效负载 (Payload)。清单 16-2 中的代码演示了如何组织缓冲区、填写头结构并调用 TraceEvent API。

清单 16-2 调用 TraceEvent API

```

typedef struct _MyEvent {           // 定义一个结构来描述追踪消息缓冲区
    EVENT_TRACE_HEADER m_Header;   // 标准的追踪信息头
    ULONG             m_ulDataInfo; // 要输出的事件数据
} MyEvent;
....                                // 以下是用于输出追踪信息的代码
MyEvent e;                          // 定义一个内存缓冲区

// 填写信息头
e.m_Header.Size = sizeof( e );      // 总的长度
e.m_Header.Guid = MyEventGUID;      // 所属事件类别的 GUID
e.m_Header.Class.Type = uType;      // 事件的具体类型
e.m_Header.Flags = WNODE_FLAG_TRACED_GUID; // 标志选项
e.m_Header.Level = TRACE_LEVEL_INFORMATION; // 事件的重要级别
e.m_ulDataInfo = uVarValue;         // 事件的负载数据
// 发送信息
Status = TraceEvent( g_hTrace, (PEVENT_TRACE_HEADER)&e );

```

以上函数中没有判断是否启用追踪，应该在这个代码段的外层加这样的检查。

除了 TraceEvent，Windows XP 引入的 TraceMessage 和 TraceMessageVa API 也可以向 ETW 会话发送追踪消息。

```

ULONG TraceMessage(
    TRACEHANDLE SessionHandle, // 会话句柄
    ULONG MessageFlags,       // 消息标志
    LPGUID MessageGuid,      // 消息所属分类的 GUID
    USHORT MessageNumber,     // 消息编号
    ...
);                           // 消息的负载数据

```

事实上，以上 3 个 API 内部都是调用 NtTraceEvent 内核服务，NtTraceEvent 又调用 ETW 的内核函数 WmiTraceMessage。

Windows Platform SDK 中提供了一个简单的 ETW 提供器例子，其位置是 Samples\WinBase\eventtrace\tracedp。

16.4 控制 ETW 会话

ETW 控制器程序应该调用 StartTrace API 来开始一个 ETW 追踪会话，并取得这个会话的信息和句柄。

```
ULONG StartTrace( [OUT] PTRACEHANDLE SessionHandle,
    [IN] LPCTSTR SessionName, [IN, OUT] PEVENT_TRACE_PROPERTIES Properties);
```

其中，参数 SessionName 用来指定 ETW 会话的名称，参数 Properties 指向一个 EVENT_TRACE_PROPERTIES 结构，调用时用来描述会话选项，函数返回时，它包含会话的属性，其定义如下：

```

typedef struct _EVENT_TRACE_PROPERTIES {
    WNODE_HEADER Wnode;          // 头架构，包含提供器的 GUID 值等属性
    ULONG BufferSize;            // 缓冲区大小
    ULONG MinimumBuffers;        // 缓冲区的最小个数

```

```

ULONG MaximumBuffers;           // 缓冲区的最大个数
ULONG MaximumFileSize;          // 对于文件模式，追踪文件的最大容量
ULONG FileMode;                // ETW 消息传递模式（文件、实时等）
ULONG FlushTimer;              // 多久冲转缓冲区一次，以秒为单位
ULONG EnableFlags;             // 仅适用于 NT Kernel Logger 会话，选择要输出的事件
LONG AgeLimit;                 // 见下文
ULONG NumberOfBuffers;         // [输出] 已分配的缓冲区个数
ULONG FreeBuffers;             // [输出] 分配但未使用的缓冲区个数
ULONG EventsLost;              // [输出] 未能记录（丢失）的事件个数
ULONG BuffersWritten;          // [输出] 已写缓冲区个数
ULONG LogBuffersLost;          // [输出] 文件模式中未能写入日志的缓冲区个数
ULONG RealTimeBuffersLost;     // [输出] 实时模式时未能发送给消耗器的缓冲区个数
HANDLE LoggerThreadId;         // [输出] ETW 提供器的线程 ID
ULONG LogFileNameOffset;        // 追踪文件 (.etl) 的文件名偏移，0 代表不使用文件
ULONG LoggerNameOffset;         // ETW 会话名称的偏移
} EVENT_TRACE_PROPERTIES, *PEVENT_TRACE_PROPERTIES;

```

其中，AgeLimit 只适用于 Windows 2000 系统，用来指定释放未使用缓冲区的时间（单位为分钟），默认为 15 分钟，Windows 2000 之后的系统不会释放缓冲区。LogFileMode 字段用来描述消息的投递（Deliver）模式，分为两种，一种是将 ETW 消息写入到文件中，简称文件模式。ETW 要求文件的后缀名是 ETL（Event Tracing Log，即事件追踪日志）或 ETL 加进程 ID（对于 EVENT_TRACE_PRIVATE_LOGGER_MODE），我们将这种文件简称为 ETL 文件。如果使用文件模式，那么应该将包含完整路径的文件名跟在 EVENT_TRACE_PROPERTIES 之后，并将其相对于结构开始处的偏移赋给 LogFileNameOffset 成员。另一种模式是实时地将 ETW 消息递送给 ETW 消耗器，简称为实时模式（Real time mode）。

如果调用 StartTrace 成功，那么 SessionHandle 参数中会返回 ETW 会话的句柄。使用该句柄和 ETW 提供器的 GUID 便可以启动 ETW 提供器，使其向这个 ETW 会话输出追踪消息。

```

ULONG EnableTrace(
    ULONG Enable,                      // TRUE 启动，否则停止指定的 ETW 提供器
    ULONG EnableFlag,                  // 启动标志
    ULONG EnableLevel,                // 追踪信息的级别
    LPCGUID ControlGuid,             // ETW 提供器的 GUID
    TRACEHANDLE SessionHandle        // StartTrace 返回的 ETW 会话句柄
);

```

启动会话后，ETW 控制器可以通过 ControlTrace API 来查询 ETW 会话的状态或执行其他控制动作，其原型如下：

```

ULONG ControlTrace(
    TRACEHANDLE SessionHandle,        // 会话句柄
    LPCTSTR SessionName,             // 会话名称
    PEVENT_TRACE_PROPERTIES Properties, // 属性结构
    ULONG ControlCode               // 控制代码，见表 16-1
);

```

其中 SessionHandle 和 SessionName 用来指定要控制的 ETW 会话，指定其一即可，参数 ControlCode 用来指定控制动作，可以为表 16-1 中的常数之一。

表 16-1 ETW 的控制码

常量	值	含义
EVENT_TRACE_CONTROL_FLUSH	3	冲转(Flush)会话的缓冲区, Windows 2000 不支持此命令
EVENT_TRACE_CONTROL_QUERY	0	读取会话的属性和统计信息
EVENT_TRACE_CONTROL_STOP	1	停止会话
EVENT_TRACE_CONTROL_UPDATE	2	更新会话属性

除了以上 API, ETW 控制器可以使用 QueryAllTraces API 查询当前系统内启动的所有 ETW 会话, 使用 EnumerateTraceGuids API 枚举系统内注册的所有 ETW 提供器。

SDK 的 TraceLog 程序 (*Samples\WinBase\eventtrace\tracelog*) 实现了一个命令行方式的 ETW 控制器。

16.5 消耗 ETW 消息

ETW 消耗器在接收 ETW 消息前, 必须使用 OpenTrace API 打开一个 ETL 文件(文件模式)或实时的 ETW 会话(实时模式), 同时注册自己用于接收消息的回调函数。

```
TRACEHANDLE OpenTrace( PEVENT_TRACE_LOGFILE Logfile );
```

参数 Logfile 用来描述要打开的会话和回调函数地址, 是一个 EVENT_TRACE_LOGFILE 结构, 其定义如下:

```
typedef struct _EVENT_TRACE_LOGFILE {
    LPTSTR LogFileName; // 如果使用文件模式, 则指向 ETL 文件名, 否则为 NULL
    LPTSTR LoggerName; // 如果使用实时模式, 则指向 ETW 控制器定义的会话名称
    LONGLONG CurrentTime; // [输出] 当前时间
    ULONG BuffersRead; // [输出] 已读缓冲区个数
    ULONG FileMode; // 消息递送方式
    EVENT_TRACE CurrentEvent; // [输出] 指向被处理的最后一个事件的 EVENT_TRACE 结构
    TRACE_LOGFILE_HEADER LogfileHeader; // [输出] ETL 文件信息
    PEVENT_TRACE_BUFFER_CALLBACK BufferCallback; // 要注册的 BufferCallback 函数, 可选
    ULONG BufferSize; // [输出] 每个 ETW 会话缓冲区的大小
    ULONG Filled; // [输出] 缓冲区中包含的有效信息字节数
    ULONG EventsLost; // 未使用
    PEVENT_CALLBACK EventCallback; // 要注册的 EventCallback 函数
    ULONG IsKernelTrace; // [输出] 如果 ETW 会话是 NT Kernel Logger, 则为 1, 否则为 0
    PVOID Context; // 保留
} EVENT_TRACE_LOGFILE, *PEVENT_TRACE_LOGFILE;
```

如果调用 OpenTrace 成功, 那么系统会返回一个句柄。但是此时系统还没有开始递送消息, 待 ETW 消耗器做好接收消息的所有准备后, 应该调用 ProcessTrace API 通知系统启动消息递送。

```
ULONG ProcessTrace(
    PTRACEHANDLE HandleArray, // 会话句柄数组
    ULONG HandleCount, // HandleArray 数组的元素个数
    LPFILETIME StartTime, // 希望开始接收事件的时间, 可以为 NULL
    LPFILETIME EndTime // 希望结束接收事件的时间, 可以为 NULL
);
```

其中，HandleArray 是一个数组，包含了使用 OpenTrace 打开的 ETW 句柄，每个句柄代表了一个 ETW 实时会话或一个已经打开的 ETL 文件。

因为使用文件时，系统会重放（playback）其中包含的追踪信息，所以对于 ETW 消耗器来说，实时模式和文件模式没有大的区别，以下讨论我们只以实时会话情况为例。HandleCount 指定了数组中包含的元素个数。StartTime 和 EndTime 用来指定希望接收事件的时间范围，如果不限定，则使用 NULL。

在成功调用 ProcessTrace 后，如果在指定的 ETW 会话中有输出事件，那么系统便会调用消耗器注册的回调函数。ETW 消耗器可以最多注册 3 种回调函数：BufferCallback、EventCallback 和 EventClassCallback。第一个是用于接收关于会话缓冲区的统计信息，后两个都是用来接收追踪事件的。EventCallback 函数的原型如下：

```
VOID WINAPI EventCallback( PEVENT_TRACE pEvent );
```

默认情况下，系统会将调用 ProcessTrace 时指定的所有会话的所有事件都发送给 EventCallback 函数，但是 ETW 允许通过 SetTraceCallback API 通知系统按事件所属的分类发给不同的回调函数。

```
ULONG SetTraceCallback(
    LPCGUID pGuid,           // 一类事件的 GUID
    PEVENT_CALLBACK EventCallback // 接收该类事件的回调函数
);
```

在成功调用 SetTraceCallback 后，当有与指定 GUID 相符的事件发生时，系统便会调用与其对应的 EventClassCallback 函数。但需要注意的是，系统始终还会把这个事件也发给 EventCallback 函数。也就是说，EventCallback 函数总会接收到所有事件。

BufferCallback 回调函数用来接收 ETW 递送缓冲区中事件的统计信息。其函数原型如下：

```
ULONG WINAPI BufferCallback( PEVENT_TRACE_LOGFILE Buffer );
```

其中 Buffer 参数指向一个我们前面介绍的 EVENT_TRACE_LOGFILE 结构，包含已读缓冲区个数、缓冲区大小等信息。BufferCallback 回调函数是可选的，ETW 消耗器可以根据需要决定是否注册这个回调函数。

SDK 的 TraceDmp 程序（Samples\WinBase\eventtrace\tracedmp）实现了一个命令行方式的 ETW 消耗器，它可以把二进制 ETL 文件中的追踪消息转化为 CSV 文件。在转化时它需要格式文件的帮助，下一节将详细介绍产生格式文件的不同方法。

16.6 格式描述

为了提高效率和节省空间，ETW 提供器是以二进制格式来输出信息的，ETL 文件也是以二进制形式来存储信息的。只有当 ETW 消耗器显示 ETW 消息时，才将其格式化为文本格式。目前 ETW 提供了 3 种方式来描述格式信息，第一种是使用 MOF 文件；

第二种是使用 WPP 写在源文件中，然后利用工具从编译好的符号文件中提取出来；第三种是 Windows Vista 引入的使用 Manifest 文件和 TDH API。下面我们分别介绍前两种方法，第 16.9 节将介绍第三种方法。

16.6.1 MOF 文件

MOF（Managed Object Format）是使用文本形式来描述 CIM 模型（Common Information Model）的程序语言，是 WBEM（Web Based Enterprise Management）标准中的一部分。Windows 系统中的 WMI（Windows Management Instrumentation）是 WBEM 的一种实现，也应用了 MOF。MOF 文件的主要内容是对类、属性、方法和实例声明的描述，其思想与 C++ 等面向对象的语言很类似。清单 16-3 列出了 SDK 中的 TraceDP 程序所使用的 MOF 文件的一部分内容。

清单 16-3 MOF 文件示例（部分）

```
[Dynamic,                                     // 指示下面的类是动态的
Description("Sample String Data") : amended,   // 描述，括号中为取值
EventType{1, 2},                                // 事件类型编号数组，括号中为取值
EventTypeName{"Start", "End"} : amended,         // 事件类型名称数组
locale("MS\0x409")                             // 语言和国别
]
class TraceDPData_Ulong:TraceDPData           // 以上块称为描述符 (Qualifiers)
{
    [WmiDataId(1),                           // WMI 数据 ID
    Description("SampleULONG") : amended,     // 名称
    read]                                    // 只读
    uint32 Data;                            // 类的成员
};
```

以上代码中，方括号中的部分被称为描述符（Qualifiers），第 1~6 行的描述符是用来描述其后的 TraceDPData 类的，第 9~11 行的描述符是用来描述 Data 成员的。每组描述符中的多个描述用逗号分隔开来。每个描述又可以分为名称、取值和 Flavor 3 个部分。如果描述符的类型是一个数组，那么取值用大括号包围起来（第 4 行），如果是单一值，则用小括号包围（第 2、5、9 和 10 行），如果是布尔型，那么只要这个描述符出现便代表对应的属性值为真（第 1 行和第 12 行）。

使用 Windows 系统自带的 mofcomp 程序可以编译 MOF 文件。Mofcomp.exe 是个命令行程序，位于 c:\<WINDOWS 根目录>\system32\wbem 目录中。清单 16-4 列出了编译 TraceDP 程序的 tracedp.mof 所用的命令和执行结果。

清单 16-4 编译 MOF 文件

```
C:\SDK60\Samples\WinBase\eventtrace\tracedp>mofcomp -N:root\wmi tracedp.mof
Microsoft (R) 32-bit MOF Compiler Version 5.1.2600.2180
Copyright (c) Microsoft Corp. 1997-2001. All rights reserved.
Parsing MOF file: tracedp.mof
MOF file has been successfully parsed
Storing data in the repository...
Done!
```

说是编译器，其实 mofcomp 不仅对 mof 文件进行解析和检查，如果没有错误，mofcomp 还会将该类定义加到 CIM 库中，清单 16-4 中最后两行的提示表明 mofcomp 在将 MOF 程序中的信息加到命令行参数所指定的命名空间（root\wmi）中。

以上操作成功后，执行系统自带的 WbemTest 程序会连接到 root\wmi 命名空间，然后点击 Open Classes 按钮，输入 TraceDPData_Ulong 便可以看到 TraceDPData_Ulong 类了（见图 16-2）。

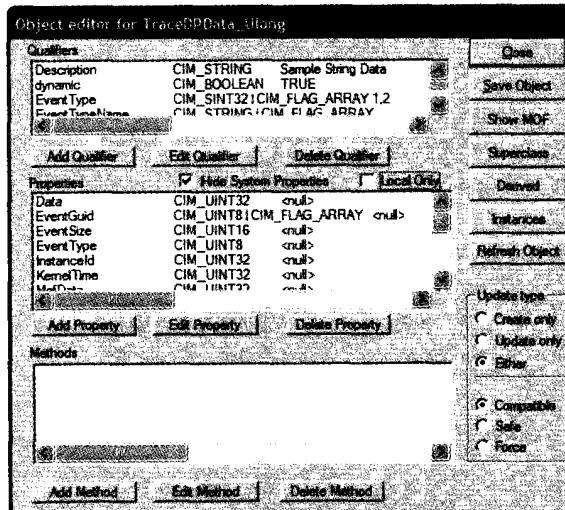


图 16-2 使用 WbemTest 工具观察 MOF 文件所定义的类

图 16-2 所示对话框的上部列表框就是清单 16-3 中的修饰符，中间是类的成员，其中除了 Data 外都是从父类继承来的，选择 Local Only 按钮可以只显示当前类的属性。

使用 WMI 的编程接口（COM 接口）可以枚举系统中注册的类信息，然后根据事件 GUID 寻找匹配的类，TraceDmp 程序演示了其细节，在此不详细介绍。

学习 MOF 的一种简单方法就是阅读 CIM Schema 中已经定义好的各个类，可以从 DMTF 网站 (<http://www.dmtf.org/standards/cim/>) 下载包含所有类定义的 MOF 文件压缩包。MSDN 中关于 WMI 的参考资料中介绍了 MOF 的描述符、数据类型和语法。

16.6.2 WPP

WPP 是 Windows Software Trace Preprocessor（追踪预处理器）的缩写，是驱动程序开发工具（DDK 和 WDK）中用来支持 ETW 而设计的一整套工具和方法的统称，其目标是让驱动程序可以很方便地实现 ETW 提供器，通过 ETW 输出追踪事件。简单来说，WPP 就是通过宏定义和编译指令来让编译工具自动产生实现 ETW 的代码和格式文件。下面介绍其主要步骤。

首先应该在源程序文件 (.c) 中定义 ETW 提供器的 GUID 和追踪标志位，例如：

```
#define WPP_CONTROL_GUIDS \
    WPP_DEFINE_CONTROL_GUID(CtlGuid, {d58c126f, b309, 11d1, 969e, 0000f875a5bc}), \
        WPP_DEFINE_BIT(TRACELEVELONE) \
        WPP_DEFINE_BIT(TRACELEVELTWO) )
```

标志位定义至少要有 1 个，最多可以有 31 个，其目的是对信息进行过滤。

而后应该在驱动程序的入口函数 (DriverEntry) 中加入 WPP 的初始化宏，即：

```
WPP_INIT_TRACING(DriverObject, RegistryPath);
```

在卸载函数中调用 WPP 的清理宏，即：

```
WPP_CLEANUP(DriverObject);
```

并在需要输出信息的地方使用 DoTraceMessage 宏来输出信息，例如：

```
DoTraceMessage(TRACELEVELONE, "Hello, %d %s", i, "Hi" );
```

为了通过编译，应该在文件中包含一个自动产生的追踪消息头 (Trace Message Header) 文件，简称 TMH 文件，例如：

```
#include "tracedrv.tmh" // 这是编译格式文件时自动产生的头文件
```

TMH 文件是自动产生的，要让 DDK/WDK 产生 TMH 文件，应该在用于构建驱动程序的 sources 文件中加入一行 RUN_WPP 指令 (directive)，格式为：

```
RUN_WPP= $(SOURCES) -km
```

当 DDK 编译这样的驱动程序时，会先运行 TraceWpp 工具 (位于 DDK 的 /bin/x86 目录中)，这个工具会自动产生一个与驱动程序同名的 TMH 文件，并在其中加入一些代码和前面使用的宏定义。

使用 WPP 的好处不仅可以自动产生代码，还可以将自动提取的追踪事件的格式信息放入到符号文件 (PDB) 中，再使用 DDK 中的 TracePDB 工具，便可以将这些信息提取到一个文本形式的 TMF 文件中，清单 16-5 列出了根据 DDK 中的 TraceDrv 驱动程序的符号文件而产生的 TMF 文件的内容。

清单 16-5 TMF 文件示例

```
// PDB: tracedrv.pdb
// PDB: Last Updated :2008-2-2:14:27:52:328 (UTC) [tracepdb]
37753236-c81f-505e-d40a-128d3bb2b5ff tracedrv // SRC=tracedrv.c MJ= MN=
#typev tracedrv_c264 11 "%0Hello, %10!d! %11!s!" // LEVEL=TRACELEVELONE
FUNC=TracedrvDispatchDeviceControl
{
    i, ItemLong -- 10
    'Hi', ItemString -- 11
}
#typev tracedrv_c258 10 "%0IOCTL = %10!d!" // LEVEL=TRACELEVELONE
FUNC=TracedrvDispatchDeviceControl
{
    ioctlCount, ItemLong -- 10
}
```

可见 WPP 会根据驱动程序代码中的事件输出语句自动提取格式信息，第 4~9 行

是根据我们在上面给出的 DoTraceMessage 行产生的，第 10~14 行是根据源代码中的以下源代码行产生的：

```
DoTraceMessage(TRACELEVELONE, "IOCTL = %d", ioctlCount);
```

TraceDrv 例子的完整路径是 *src\general\tracedrv\tracedrv*，它是学习 WPP 的一个很好的起点。

16.7 NT Kernel Logger

前面两介绍了 ETW 的模型和工作原理。本节将介绍 ETW 在 Windows 系统中的一个应用——用于追踪内核事件的 NT Kernel Logger（NT 内核记录器），简称 NKL。

NKL 是 Windows 内建的用于记录 Windows 内核事件的系统组件，其核心是一个根据 ETW 规范实现的 ETW 提供器（Provider）。通过这个 ETW 提供器，ETW 消耗器程序可以接收来自内核的追踪事件，包括进程、线程、磁盘、文件 IO、网络等。

上一节我们讲到过，系统中可以最多有 32 个（Windows 2000）或 64 个（Windows XP 开始）EW 会话，有两个是系统专用的，其中之一就是 NKL 会话（NT Kernel Logger Session）。也就是说，系统总是为 NKL 会话保留了位置，并为其定义了专用的名字，“NT Kernel Logger”。

NKL 提供器的 GUID 是 9e814aad-3204-11d2-9a82-006008a86939，*evntrace.h* 头文件中定义的量 *SystemTraceControlGuid* 就是这个 GUID。

16.7.1 观察 NKL 的追踪事件

NKL 是内建在 Windows 2000 开始的所有 Windows 内核中的，只要使用一个简单的 ETW 消耗器程序就可以接收到它的事件输出。下面我们以 DDK 中包含的 TraceView 工具为例介绍如何启动并观察 NKL 的追踪事件。

首先找到 DDK 中的 *TraceView.exe*。以 Windows Server 2003 SP1 的 DDK 为例（3790.1830 版），其位置是 *c:\WINDDK\3790.1830\tools\tracing\i386*。然后运行这个程序，选择文件菜单和创建新会话（Create New Log Session）命令，点击加入提供器（Add Providers）按钮，会弹出图 16-3 所示的选择 ETW 提供器对话框。

TraceView 是一个通用的 ETW 消耗器程序，支持从系统中注册的各种 ETW 提供器接收信息，因为我们要接收 NKL 事件，所以选择 Kernel Logger，并选中希望看到的事件类型。图 16-3 中列出了 9 类事件，包括进程/线程管理、文件、磁盘、内存、网络、注册表操作等。尽管可以同时选择多项，但是为了便于观察和分析，我们只选择进程事件一项。因为 ETW 的原始消息是二进制的，*TraceView* 需要格式化信息将 ETW 消息格式化为可读的文本消息，所以接下来它会弹出一个打开文件对话框，让我们选择 *TMF* 文件。在 DDK 的同一个文件夹中，DDK 为我们准备了一个 *system.tmf* 文件，

其中包含了上述系统事件所使用的格式信息。因此只要选择该文件就可以了。

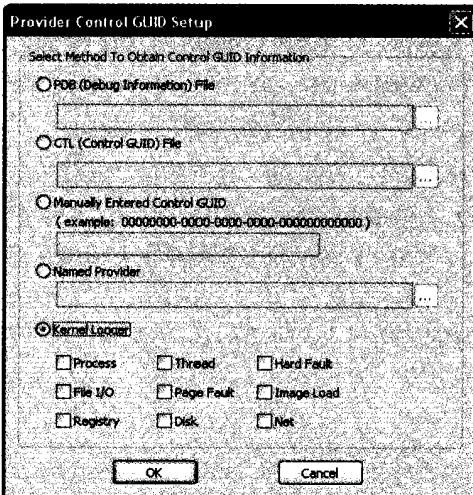


图 16-3 选择 ETW 提供器对话框

点击继续，出现 Log Session Options 对话框，其中可以指定会话的工作模式（实时显示和写入文件，可以同时选中）及其他选项，我们暂时使用默认设置，直接点击完成按钮。至此，一个 NKL 会话便建立了。

图 16-4 是 TraceView 的工作画面，上边的子窗口显示 NT Kenerl Logger 会话正在运行，共收到了 74 条消息，丢失 0 条。下面的子窗口显示了第 66 条到第 74 条消息的内容。编号为 66 的事件描述的是笔者以 Dbgger 用户身份启动计算器程序，第 67 条描述的是退出计算器程序。第 68 条到第 70 条描述的是切换到已经登录的另一个用户（Dbgee）（Session 1），第 71 条描述的是以 Dbgee 用户启动记事本程序。

建议大家按照以上步骤亲自动手做一做，加深对 ETW 和 NKL 的理解，这样才有可能把它们应用到实践中。

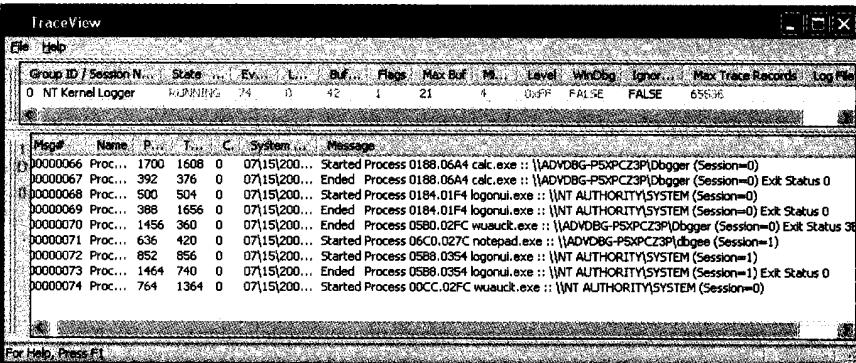


图 16-4 TraceView 程序的界面

16.7.2 编写代码控制 NKL

除了 TraceView，还有很多其他工具也可以与 NKL 建立会话，比如命令行方式的 TraceLog（参见表 16-5）。也有很多工具是基于 NKL 设计的，比如 SysInternals 的 TCPView 和 DiskMon 就是以 NKL 会话输出的网络和磁盘类事件为基础的。

事实上，通过 SDK 中公开的 ETW API (StartTrace)，只需要很少的代码就可以建立 NKL 会话，清单 16-6 所示的 StartNKL 方法演示了如何使用 StartTrace API 建立 NKL 会话，并将指定的事件（参数 dwEnableFlags）写入到参数指定的记录文件中。

清单 16-6 启动 NKL 会话

```

1  BOOL CNKLMgr::StartNKL(LPCWSTR lpszLogFile , DWORD dwEnableFlags)
2  {
3      ULONG BufferSize = 0;
4      ULONG rc = 0;
5
6      BufferSize = sizeof(EVENT_TRACE_PROPERTIES) +
7          sizeof(TCHAR)*(_tcslen(lpszLogFile)+1) + sizeof(KERNEL_LOGGER_NAME);
8      if(m_pNklProperties==NULL)
9          m_pNklProperties = (EVENT_TRACE_PROPERTIES*) malloc(BufferSize);
10
11     if (NULL == m_pNklProperties)
12     {
13         wprintf(L"Unable to allocate %d bytes for properties structure.\n",BufferSize);
14         return FALSE;
15     }
16
17     ZeroMemory(m_pNklProperties, BufferSize);
18     m_pNklProperties->Wnode.BufferSize = BufferSize;
19     m_pNklProperties->Wnode.Flags = WNODE_FLAG_TRACED_GUID;
20     m_pNklProperties->Wnode.ClientContext = 1; //QPC clock resolution
21     m_pNklProperties->Wnode.Guid = SystemTraceControlGuid;
22     m_pNklProperties->EnableFlags = dwEnableFlags;
23     m_pNklProperties->LogFileMode = EVENT_TRACE_FILE_MODE_CIRCULAR
24         | EVENT_TRACE_USE_PAGED_MEMORY;
25     m_pNklProperties->MaximumFileSize = 20;
26     m_pNklProperties->LoggerNameOffset = sizeof(EVENT_TRACE_PROPERTIES);
27     m_pNklProperties->LogFileNameOffset = sizeof(EVENT_TRACE_PROPERTIES)
28         + sizeof(KERNEL_LOGGER_NAME);
29     _tcscpy((LPTSTR)((char*)m_pNklProperties+m_pNklProperties->LogFileNameOffset),
30             lpszLogFile);
31
32     rc = StartTrace((PTRACEHANDLE)&m_hNklSessionHandle,
33                      KERNEL_LOGGER_NAME, m_pNklProperties);
34     if (ERROR_SUCCESS != rc)
35     {
36         if (ERROR_ALREADY_EXISTS == rc)
37             OUTMSG(_T("The NT Kernel Logger session is already in use.\n"));
38         else
39             OUTMSG(_T("StartTrace() failed, %d.\n"), rc);
40
41         return FALSE;
42     }
43     OUTMSG(_T("StartTrace() succeeded, %s.\n"), lpszLogFile);
44
45     return TRUE;
46 }
```

其中 dwEnableFlags 参数可以包含一个或多个代表预定义内核事件的标志。表 16-2 列出了不同事件的标志位。

表 16-2 内核记录器（NKL）使用的事件标志位

标志	含义	值
EVENT_TRACE_FLAG_DISK_FILE_IO	文件读写	0x00000200
EVENT_TRACE_FLAG_DISK_IO	磁盘访问	0x00000100
EVENT_TRACE_FLAG_IMAGE_LOAD	映像加载	0x00000004
EVENT_TRACE_FLAG_MEMORY_HARD_FAULTS	硬错误	0x00002000
EVENT_TRACE_FLAG_MEMORY_PAGE_FAULTS	页错误异常	0x00001000
EVENT_TRACE_FLAG_NETWORK_TCPIP	网络 (TCP/IP 和 UDP)	0x00010000
EVENT_TRACE_FLAG_PROCESS	进程	0x00000001
EVENT_TRACE_FLAG_REGISTRY	注册表访问	0x00020000
EVENT_TRACE_FLAG_THREAD	线程	0x00000002
EVENT_TRACE_FLAG_DBGPRINT	调试输出 (DbgPrint)	0x00040000

因此，只要使用类似如下的语句调用 StartNKL 方法，就可以将进程类内核事件写到指定的记录文件中。

```
m_NklMgr.StartNKL("C:\\\\nkl.etl", EVENT_TRACE_FLAG_PROCESS);
```

本章示例项目（code\\chap16\\etw）中包含了 CNklMgr 类的完整代码和使用该类的一个小程序。可以使用 TraceFmt 工具（与 TraceView 在同一文件夹中）将 ETL 文件格式化为文本文件。以下是使用执行 TraceFmt 工具的命令行和部分执行结果：

```
c:\\WINDDK\\3790.1830\\tools\\tracing\\i386>tracefmt c:\\nkl.etl -tmf system.tmf
Setting log file to: c:\\nkl.etl
Examining system.tmf for message formats, 14 found.
```

TraceFmt 会产生两个文件，一个是 FmtSum.txt，存储的是概要信息，另一个是 FmtFile.txt，存储的是一个个追踪事件。

16.7.3 NKL 的实现

与调试子系统的 Dbgk 例程类似，ETW 也公开了一系列函数，供内核的其他部件向 ETW 报告事件。包括 WmiTraceProcess、WmiTraceThread、WmipTraceIo、WmipTraceNetwork、WmipTraceFastMutex、WmipTracePageFault、WmipTraceFile、WmipTraceLoadImage、WmipTraceRegistry、WmiTraceContextSwap 等。

Vista 将这些函数改名为以 Etw 开头，例如以下栈回溯显示的是当系统删除进程对象时调用 ETW 的进程退出函数（EtwExitProcess）的执行过程。

```
kd> kn
# ChildEBP RetAddr
00 877a7c48 81e1f290 nt!EtwExitProcess
01 877a7c7c 81de1f7b nt!PspProcessDelete+0x2ac
...
```

以上函数被调用后，会检查 NKL 是否被启用，如果是，则准备输出追踪信息，调用 WmiReserveWithSystemHeader 取得缓冲区，而后向缓冲区中写入数据，最后调用 WmipReleaseTraceBuffer 完成信息输出。

16.8 Global Logger Session

上一节我们介绍了 Windows 系统内建的 ETW 提供器 NKL。通过它可以了解内核部件的行为。但是我们知道，ETW 提供器需要有 ETW 会话启动它后才会输出追踪事件。在系统启动后，我们可以运行 TraceView 或 TraceLog 来启动 NKL，但是，如果我们希望在系统启动早期就启用 NKL 的输出，这两个工具就不适用了。为了支持这一需求，除了内建的 ETW 提供器外，Windows 系统还内建了一个名为 Global Logger Session（全局记录器会话）的全局 ETW 会话，简称 GLS。

概而言之，GLS 就是系统预先定义的一个 ETW 会话，通过设置注册表便可以启动这个会话，定义这个会话要启用的 ETW 提供器。因此，通过 GLS 可以在系统登录前启用系统中的 ETW 提供器，GLS 可以帮助我们把提供器所输出的事件保存到文件中。利用 ETW 的缓冲功能，也可以用 GLS 来追踪文件系统还没有准备好（无法写文件）的事件。

GLS 既可以控制内建的 NKL 提供器，也可以启用驱动程序或用户态的服务程序所定义的 ETW 提供器。

16.8.1 启动 GLS 会话

因为 GLS 会话的主要目的是记录启动早期的事件。所以，GLS 会话是通过注册表选项来启动和配置的，而不是 API 调用。控制 GLS 的注册表键的路径如下：

HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\WMI\GlobalLogger

如果 GlobalLogger 键不存在，那么加上即可。只要在以上表键下加入一个名为 Start 的键值（REG_DWORD 类型），并取值为 1，当 Windows 下次启动时就会启动 GLS 会话。

也可以使用 tracelog 来自动添加和删除这些注册表选项。

```
tracelog -start GlobalLogger      // 加入启动选项
tracelog -stop GlobalLogger      // 停止
tracelog -remove GlobalLogger    // 删除
```

GLS 会话启动后，其行为与普通的 ETW 会话是一样的，不过它不支持实时模式，只支持文件模式。GLS 会将追踪事件写入到注册表中定义的文件中（稍后介绍）。

系统会把启动 GLS 的结果写到 GlobalLogger 表键下，键值名为 Status。如果成功，那么 Status 的值为 0 (STATUS_SUCCESS)；如果失败，那么系统会把错误的状态码转换为 WIN32 错误码，然后写到注册表中，可以使用查询 Last Error 的工具（比如 VC 中的 Error Lookup）来查看错误码的含义。GLS 报告的一种常见错误码是 87，意思是参数错误，这种错误的一个典型原因就是把 REG_BINARY 类型的 EnableKernelFlags 键值创建成 REG_DWORD（见下文）。

16.8.2 配置 GLS

除了 Start 键值，在 GlobalLogger 表键下还可以通过其他键值来配置 GLS 所使用的缓冲区和文件路径，表 16-3 列出了配置 GLS 的各个注册表键值。

表 16-3 配置全局记录器会话（GLS）的注册表选项

Start	REG_DWORD	为 1 时，系统启动时会启动 Global Logger 会话
BufferSize	REG_DWORD	每个缓冲区的大小(单位 KB)，默认值为 0x40 (64 KB)
ClockType	REG_DWORD	选择要使用的定时器(timer), 1 = 性能计数器, 2 = 系统定时器, 3 = CPU 时钟(cycle clock)
EnableKernelFlags	REG_BINARY	将 GLS 转变为 NKL 会话，并定义要包含的内核事件，见下文
FileName	REG_SZ	事件追踪记录文件的名称和路径(可选)，默认值为 %SystemRoot%\System32\LogFiles\WMI\trace.log
FlushTimer	REG_DWORD	强制冲转(flush)ETW 缓冲区的时间(以秒为单位)，强制冲转是对自动冲转的补充。当缓冲区已满或追踪会话停止时，系统会自动冲转缓冲区。默认值为 0，含义是仅当缓冲区满时冲转
LogFileMode	REG_DWORD	指定 ETW 记录的模式，从 Vista 开始支持
MaximumBuffers	REG_DWORD	指定该会话可以分配的最多缓冲区个数，默认值是 0x19 (25)
MaximumFileSize	REG_DWORD	指定记录文件的最大大小，默认没有限制
MinimumBuffers	REG_DWORD	会话启动时分配的缓冲区个数，默认值为 0x3
Status	REG_DWORD	用于存储启动 GLS 的返回值。如果启动失败，这里记录的是 Win32 错误代码；如果成功启动，那么是 ERROR_SUCCESS (0)
FileCounter	REG_DWORD	用于存储 GLS 所产生的记录文件数，系统会自动递增该值直到达到 FileMax 项所定义的最大值。达到最大值后，会复位为 0。该计数器的目的是防止系统覆盖记录文件
FileMax	REG_DWORD	系统中允许的最多记录文件数。当达到该值时，系统会覆盖以前的记录文件(从最老的开始)。默认值为 0，代表没有限制

首先解释 EnableKernelFlags 键值，通过设置这个键值可以让 GLS 会话追踪 NKL 事件。特别需要指出的是，尽管 WDK 和微软网站上的文档都将这个键值描述为 REG_DOWRD 类型，但是根据笔者的反复试验，最终发现这个键值应该为 REG_BINARY 类型，其内容应该是一个 DWORD 数组，每个数组元素是表 16-2 中的一种标志值。

通过表 16-3 中的 FileName 键值可以指定用来存储追踪事件的日志文件路径和名称，如果没有指定，那么默认的文件是：

%SystemRoot%\System32\LogFiles\WMI\Trace.log

或者对于 XP 之前的系统可能是：

```
%SystemRoot%\System32\LogFiles\WMI\GlobalLogger.etl
```

如果定义了 EnableKernelFlags 键值，那么在 Vista 中的默认文件名为 NT Kernel Logger.etl，XP 的默认文件名依然是 Trace.log。

LogFileMode 键值用来指定 GLS 处理文件的方式，可以包含 EVENT_TRACE_FILE_MODE_SEQUENTIAL(0x1)、EVENT_TRACE_FILE_MODE_CIRCULAR(0x2)、EVENT_TRACE_FILE_MODE_APPEND(0x4)、EVENT_TRACE_FILE_MODE_NEWFILE(0x8)、EVENT_TRACE_FILE_MODE_PREALLOCATE(0x20)、EVENT_TRACE_KD_FILTER_MODE (0x0x80000) 等标志位。

GLS 产生的文件都是二进制的，可以使用 TraceView 工具打开这些文件(File>Open Existing Log File) 进行观察。

16.8.3 在驱动程序中使用 GLS

驱动程序也可以通过 GLS 会话输出 ETW 消息。首先，应该在源代码中 (WPP_CONTROL_GUIDS 宏与包含 TMH 头文件之间) 加入如下定义：

```
#define WPP_GLOBALLOGGER
```

有了这个标志后，WPP 自动产生的代码便会检查 GLS 会话是否启动，如果启动，那么便会向其输出信息。因为 GLS 会话不会主动调用 StartTrace 函数来启用 ETW 提供器。

第二步是在注册表的 GlobalLogger 表键下建立一个子键，其名称是驱动程序定义的提供器 GUID。比如，以下是为 TraceDrv 驱动程序而建立的子键：

```
HKEY\SYSTEM\CurrentControlSet\Control\WMI\GlobalLogger\d58c126f-b309-11d1-969e-0000f875a5bc
```

在这个子键下可以加入 Flags 和 Level 键值，用来过滤要输出的事件。

做好以上准备后，只要按前面介绍的内容配置 GLS 的其他选项并将 Start 键值设置为 1 便可以了，当下次启动时，驱动程序的追踪信息便可以通过 GLS 输出到追踪文件中。

16.8.4 Autologgers

Windows Vista 引入了一种新的机制来记录启动期间的事件，称为 Autologgers (自动记录器)。Autologgers 的工作原理和 GLS 非常相似，但是可以定义多个会话，与 GLS 只能有一个会话不同。

自动记录器也是通过注册表来配置的。首先在以下表键下注册一个自动记录器会话，以会话名为子键名：

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\WMI\Autologger
```

然后在 WMI 的 AutologgerProvider 表键下以 ETW 提供器的 GUID 为子键名注册自动记录器会话所使用的 ETW 提供器。只有 Vista 及其以后的 Windows 操作系统才支持自动记录器会话。

16.8.5 BootVis 工具

微软公司曾经发布过一个名为 BootVis 的自由工具，使用这个工具可以追踪和记录系统启动（Reboot）和从睡眠模式（S1、S3）恢复过程中的详细过程，包括发生的事件和所使用的时间，并且能以图形化的方式显示出来（见图 16-5）。

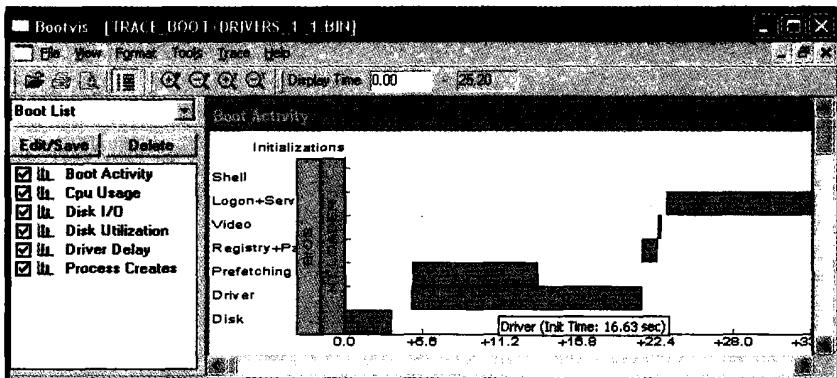


图 16-5 使用 BootVis 追踪启动过程

下面以追踪启动过程为例介绍 BootVis 工具的工作过程。当我们发起追踪（Trace>Bext Boot + Driver Delays）时，BootVis 会向当前目录中动态创建一个可执行文件，名为 bootvis_sleep.exe，并且为这个可执行程序创建一个快捷方式（Shortcut）放在开始菜单的自动启动目录中，通常为：

```
C:\Documents and Settings\All Users\Start Menu\Programs\Startup
```

因为 BootVis 是使用 GLS 来记录启动事件的，所以它会设置注册表来启用 GLS，包括将 Start 键值设为 1，把 EnableKernelFlags 设置为如下值（十六进制）：

```
07 23 00 00 16 00 00 00 <24 个字节 00> 17 23 00 00
```

做好以上准备工作后，BootVis 会重新启动系统。

待系统重新启动后，放在自动启动菜单中的 bootvis_sleep.exe 会被启动，bootvis_sleep 在屏幕上显示一段正在处理的提示信息后会调用 Sleep API 使进程休眠一段时间，其目的是等待 NKL 输出的事件全部冲转到文件中，等待期满后 bootvis_sleep 会启动 BootVis.exe，自己退出。BootVis.exe 启动后会从 C:\WINDOWS\System32\LogFiles\WMI\trace.log 文件中读取追踪事件并保存到一个新创建的文件中，文件名为 TRACE_BOOT_x_x.BIN，其中 X 为数字。

16.9 Crimson API

Windows Vista 引入了一套代号为 Crimson 的 API，旨在复用 ETW 设施来加强原有的事件日志机制。因为 Crimson 是建立在 ETW 技术之上的，所以在我们理解了 ETW 后就可以比较容易地理解 Crimson。SDK 文档将 Crimson 称为 Windows Event Log，为了明确起见，本书中仍使用 Crimson 这一开发代号。因为是基于 ETW 的，所以 ETW 中的提供器、控制器和消耗器概念仍适用于 Crimson，使用 Crimson API 输出事件的应用程序是提供器，在 Crimson 中，又被称为事件发布器（Event Publisher）。订阅和接收事件的应用程序或系统工具被称为事件消耗器。下面我们将分别介绍在 Crimson 中发布和消耗消息的方法。

16.9.1 发布事件

表 16-4 列出了用于发布事件的 Crimson API（部分）。它们的名字大多暗示出了它们的用途，简单说来，应用程序应该先调用 EventRegister 向系统注册，并获得一个用于后续操作的句柄。注册成功后，便可以使用 EventWriteString 或 EventWrite 函数来输出信息了。在输出信息前，发布器可以通过 EventEnabled API 来判断要输出的事件是否被启用，如果不判断，那么系统函数会进行判断。

表 16-4 供事件发布器使用的 Crimson API

API	说明
EventRegister	注册事件发布器，成功后会获得一个句柄
EventEnabled	判断是否被启用
EventWrite	输出事件
EventWriteString	输出字符串类型的事件信息
EventUnregister	注销登记

为了便于分类和管理，事件提供器在输出事件时可以指定事件的严重级别（Level）、所属频道（Channel）和供搜索使用的关键字（Keyword）等属性。如果使用 EventWrite API，那么应该通过 EVENT_DESCRIPTOR 结构来指定这些属性。

```
typedef struct _EVENT_DESCRIPTOR {
    USHORT      Id;           // 事件的标识代码 (ID)
    UCHAR       Version;       // 版本
    UCHAR       Channel;       // 频道号
    UCHAR       Level;         // 严重级别
    UCHAR       Opcode;        // 操作代码
    USHORT      Task;          // 模块或子任务代码
    ULONGLONG   Keyword;       // 关键字掩码
} EVENT_DESCRIPTOR, *PEVENT_DESCRIPTOR;
```

如果使用 EventWriteString API，那么可以直接在参数中指定信息的级别和关键字属性。

```
ULONG EventWriteString( REGHANDLE RegHandle,
    UCHAR Level, ULONGLONG Keyword, PCWSTR String);
```

本章的示例程序（code\chap16\Crimson）演示了以上 API 的用法。

16.9.2 消耗事件

供事件消耗器使用的 Crimson API 都是以 Evt 开头的，共有 30 多个。可以分为如下几类。

查询类：包括开始查询的 EvtQuery，成功后，它会返回一个结果集合（Result Set），然后可以使用 EvtNext 和 EvtSeek 遍历这个集合。

远程访问：使用 EvtOpenSession API 可以登录和访问网络上其他机器的 Crimson 日志。

订阅：即 EvtSubscribe，使用它可以注册回调函数以便以实时方式接收感兴趣的事件。

格式化：因为 Crimson 也是以二进制方式来组织数据的，所以在将事件显示给用户前应该将其格式化，这一过程被称为绘制（Render）。EvtRender API 用于帮助事件消耗器程序实现这一功能。

维护：包括 EvtOpenLog（打开）、EvtClearLog（清除数据）、EvtGetLogInfo（读取信息）、EvtExportLog（输出到文件）、EvtArchiveExportedLog（归档）等。

频道枚举和配置：频道是比发布器小一级的单位，一个事件发布器可以建立多个频道以便输出针对不同受众的信息，好像电视台通过不同的频道输出不同类型的节目一样。EvtOpenChannelEnum API 用于开始枚举系统中的可用频道，EvtNextChannelPath 取得当前频道的名字并递进枚举位置，EvtGetChannelConfigProperty 和 EvtSetChannelConfigProperty 分别用来读取和设置频道的配置属性，EvtSaveChannelConfig 用于保存频道的配置属性。

其他：包括用于创建和更新书签的 EvtCreateBookmark 和 EvtUpdateBookmark，以及用于读取事件信息的 EvtGetEventInfo 和用于读取查询信息的 EvtGetQueryInfo。

Windows Platform SDK 6.0 版本（以下简称 SDK60）中包含了几个示例程序，分别演示了查询事件（QueryLog）、以拉方式实时接收事件（SubscriberPull）和以推方式实时接收事件（SubscriberPush），它们位于 Samples\WinBase\eventtrace 目录下。

16.9.3 格式描述

Crimson 使用 XML 格式的 manifest 文件来描述事件的格式信息和附加属性。SDK 中把这样的文件称为 Instrumentation Manifest，我们将其简称为清单文件。概而言之，

清单文件以树形结构依次描述了提供器、频道、事件模板及事件，清单 16-7 描述了一个简单的清单文件的基本内容。

清单 16-7 Crimson 使用的格式文件示例（部分）

```

1   <provider name=...>
2     <channels>
3       <importChannel child="C1" name="Application" />
4       <channel child="MyChannel" name=.../>
5       <channel child="MyChannel2" name=.../>
6     <channels>
7     <templates>
8       <template tid="MyEventTemplate">
9         <data name="" "Prop_UnicodeString" inType="win:UnicodeString" />
10        ...
11      </template>
12    </templates>
13    <events>
14      <event value="1"
15        level="win:Informational"
16        template="MyEventTemplate"
17        opcode="win:Info"
18        channel="MyChannel"
19        symbol="PROCESS_INFO_EVENT"
20        message="${string.Publisher.EventMessage}" />
21    </events>
22 <provider name=...>
```

我们可以清楚地看到以上内容分为 3 大块，第 2~6 行用来描述提供器的每个频道，第 7~12 行用来定义事件格式模板，第 13~21 行用来描述事件。一个清单文件可以描述多个提供器，也就是可以有多个并列的<provider>元素。

使用 SDK60 所附带的消息文件编译器 MC.exe 可以编译清单文件，例如：

```
C:\dig\dbg\code\chap16\Crimson>mc -W c:\sdk60\include\winmeta.xml Crimson.man
```

VS2005 的 MC 程序还不支持这一功能。为了帮助事件消耗器工具查询和处理事件信息，Vista 提供了一系列 API，称为 TDH (Trace Data Helper) API，这些 API 都是以 Tdh 开头的，例如 TdhGetProperty、TdhGetEventInformation 等。

16.9.4 收集和观察事件

根据我们前面对 ETW 的介绍，ETW 提供器所要输出的事件信息在被 ETW 控制器启用后才会真正输出，使用 Crimson API 的事件发布器也不例外。可以使用系统中附带的 logman 和 tracerpt 工具来完成这些操作。

例如，以下命令可以启动一个名为 mysession 的会话，将本章示例程序 (Crimson.exe) 所输出的事件输出到文件 (mytest.etc) 中：

```
logman start mysession -p AdvDbg-Book-WELPublisher -o mytest.etc -ets
```

其中-p 开关后面是发布器的名字，它是 Crimson 程序的清单文件中所定义的，在运行以上命令前，我们已经使用了 wevtutil 工具将 crimson.man 注册到系统中，执行

`wevtutil im crimson.man`。也可以通过发布器的 GUID 来注册。

执行以上命令后，就会发现当前目录中多了一个 `mytest.etl` 文件，但其长度可能是 0 字节。执行 `crimson.exe` 后，停止会话（`logman stop mysession-ets`），就会发现这个文件的长度不再为 0 了。

使用 `tracert` 工具可以把事件文件中的二进制信息格式化成文本形式，例如以下命令会读取 `mytest.etl` 中的事件信息，并保存到 `dumpfile.xml` 和 `summary.txt` 中。

16.9.5 Crimson API 的实现

首先，Vista 的 NTDLL 中新增了一系列以 `Etw` 开头的函数，发布器使用的 API 大多是这些函数的别名。比如，`EventWriteString` API 就是直接转发到 NTDLL 中的 `EtwEventWriteString` 函数的，类似的，`EventRegister` 是转发到 `EtwEventRegister` 函数的，等等。

消耗器使用的以 `Evt` 开头的函数是从 `wevtapi.dll` 输出的，TDH 函数是实现在 `TDH.DLL` 中的，它们是 Vista 为了支持 Crimson API 而引入的新模块。

除了以上模块外，为了支持事件订阅功能，Vista 还引入了一个事件收集器服务，全称为 Windows Event Collector。这个服务通常运行于 `svchost` 进程中，其功能实现在 `wevtsvc.dll` 中。

16.10 本章总结

自从 Windows 2000 引入 ETW 后，基于 ETW 的应用也在不断增加。表 16-5 归纳出了与 ETW 密切相关的一些工具的名称、来源和用途。

表 16-5 ETW 工具

名称	来源	用途
Tracelog	DDK/SDK	命令行方式的 ETW 控制器，可以启动、配置、停止、更新实时或文件模式的 ETW 会话。SDK 中有源代码： <code>c:\Program Files\Microsoft SDK\Samples\winbase\eventtrace\TraceLog</code>
TracePDB	DDK	命令行方式的支持工具，可以从 PDB 文件生成 TMF 文件。TMF 文件可用于 TraceView 和 TraceFmt 工具
TraceFmt	DDK	命令行方式的 ETW 消耗器，可以格式化来自实时 ETW 会话或 ETW 记录文件的消息，格式化后的信息可以输出到控制台窗口或文本文件中
TraceWPP	DDK	扫描驱动程序源文件，产生头文件供 WPP 使用
TraceView	DDK	图形界面的 ETW 控制器和消耗器。集成并扩展了 Tracepdb、Tracelog 和 Tracefmt 的功能
Logman	Windows XP/Vista	Windows 自带的 ETW 控制器

续表

名称	来源	用途
TraceRpt	Windows XP/Vista	Windows 自带的命令行方式 ETW 消耗器, 可以将 ETW 消息格式化到 CSV 或 XML 文件中
RATT	微软网站	用来监视 ISR 和 DPC 的执行时间, 并生成报告
ETWProvider	微软网站	ETW 提供器(用户态)示例, 包含完整源代码
BootVis	微软网站 (已停)	利用 GlobalLogger 追踪系统启动(恢复)过程, 并以可视化的形式呈现出来
PIX for Windows	DirectX SDK	用于监视 3D 图形有关的性能(Performance Investigator for DirectX)

除了以上工具外, 像 VS 2005 和 WinDBG 这样的调试器也实现了对 ETW 的支持, 可以控制 ETW 会话和显示 ETW 消息。WinDBG 的扩展命令模块 Wmitrace.dll 和 Traceprt.dll 提供了一系列命令 (!wmitrace.XXX) 控制 ETW 会话和消息, 大家可以参考 WinDBG 帮助文档中关于这些扩展命令的帮助说明, 以了解其详细用法。

参考文献

1. Getting Started with Software Tracing in Windows Drivers. Microsoft Corporation
2. Insung Park and Ricky Buch. Event Tracing: Improve Debugging And Performance Tuning With ETW.
3. Matt Pietrek. Fun With Crimson Eventing in Vista. http://blogs.msdn.com/matt_pietrek/

WHEA 是 Windows Hardware Error Architecture 的缩写，即 Windows 硬件错误架构。概括来说，WHEA 是 Windows 操作系统中用于处理硬件错误的基本框架，它定义了系统中的硬件、固件（Firmware），以及软件应该如何相互协作来报告、处理和记录各种硬件错误，并且提供了一系列基础设施和机制来实现以上目标。

微软在 2005 年的硬件技术大会上介绍了 WHEA 的设计背景和基本架构。而后 WHEA 得到了包括英特尔和 Dell 等在内的计算机硬件厂商的支持。2006 年底推出的 Windows Vista 是实现了 WHEA 的第一个 Windows 客户端（Client）版本。Windows Server 2008 是 WHEA 在 Windows 服务器版本中的第一个实现。

作为介绍 Windows 错误机制的最后一个部分，本章我们将分两部分介绍 WHEA。前半部分（前两节）将介绍 WHEA 的架构（17.1 节）和错误源（17.2 节）。后半部分（17.3~17.5 节）介绍 WHEA 处理硬件错误的过程（17.3 节），WHEA 如何通过固件来保存错误记录（17.4 节），以及模拟硬件错误的错误注入机制（17.5 节）。

17.1 目标和架构

与软件错误相比，硬件错误的总量要少得多，但是一旦发生，导致的后果通常都比较严重，经常会导致整个系统的崩溃。根据微软在线崩溃分析系统（OCA）的统计结果，在用户发送的崩溃报告中有 7%~10% 是与硬件错误有关的。

一个计算机系统中有很多种硬件，每种硬件产生的错误也是不同的，有些错误是可以纠正的（correctable），而有些错误是不可以纠正的（uncorrectable）。

17.1.1 目标

在 WHEA 之前，Windows 系统对硬件错误的最典型处理方法就是立即发起 Bug Check，即蓝屏崩溃（BSOD）。这样做有两个明显的不足，首先是对于可以纠正的错

误，蓝屏方法显然处置过度，导致不必要的系统终止。第二，蓝屏崩溃通常只能记录崩溃前的 CPU 和内存状态，无法记录与硬件错误直接相关的硬件状态，这让发现硬件错误的根源和修正错误很困难。

WHEA 正是为了解决以上问题而设计的，其初衷就是在 Windows 系统中建立一套基础设施，来更好地记录、处理和报告硬件错误。进一步说，WHEA 框架所定义的基础设施包括。

- 通用的错误来源（Error Source）发现机制。
- 统一的硬件错误记录格式。
- 统一的硬件错误处理流程。
- 可靠的错误记录持久化机制。
- 基于 ETW 的硬件错误事件模型，管理程序可以通过这种模型接收到硬件错误事件并采取进一步的措施。

通过以上设施，WHEA 旨在实现以下目标。

- 借助 WHEA 的错误记录机制所记录下的错误信息，可以更快地发现错误根源，缩短系统从硬件错误中恢复运行的平均时间。
- 通过纠正可纠正错误和健康状况监视（Health Monitoring），可减少因为硬件错误而导致的系统崩溃。
- 为应用软件开发者提供支持，以便可以开发出强大的硬件错误报告和管理软件。
- 更好地利用硬件已经提供的和将来可能提供的错误报告机制，比如 CPU 的 MCA 机制（详见第 6 章），以及 PCI Express 总线标准中定义的 AER（Advance Error Reporting）机制。

因为 WHEA 旨在处理硬件错误，所以它不仅仅是操作系统自身实现的问题，还需要系统硬件和固件的支持。因此 WHEA 的实现涉及系统软件、固件和硬件。这也是 WHEA 名字中架构一词的内涵。

17.1.2 架构

图 17-1 画出了构成 WHEA 的主要部件和它们在系统中的位置。浏览全图可以看到，从系统的固件层到硬件抽象层，再到内核和驱动程序，都有 WHEA 的部件。我们先分别概述如下。

固件（Firmware）层：在这一层中可以定义两类 WHEA 设施，一类是使用 ASL（ACPI Source Language）脚本编写的系统描述表（System Description Table），包括描述错误源的 HEST（Hardware Error Source Table），描述错误记录持久化的 ERST（Error Record Serialization Table）和描述引导错误的 BERT（Boot Error Record Table）；另一

类是使用本地代码编写的实现在固件中的硬件错误处理函数，即硬件错误固件处理器（Firmware Hardware Error Handler，简称 FHEH）。

平台相关层：首先在传统的硬件抽象层（HAL）模块中包含了两个内建的硬件错误底层处理器（Low Level Hardware Error Handler，简称 LLHEH），一个是用来处理 CPU 的机器检查错误（MCE），另一个是用来处理已经纠正了的平台错误（Corrected Platform Error，简称 CPE），硬件或固件在纠正了某些错误后，会通过中断或设置标志位来通知系统软件。除了 HAL 模块中的以上部件，这一层还有一个专门用于 WHEA 的 DLL 模块，称为平台相关硬件错误驱动程序（Platform Specific Hardware Error Driver，简称 PSHED），文件名为 PSHED.DLL，我们稍后会详细介绍这个模块。

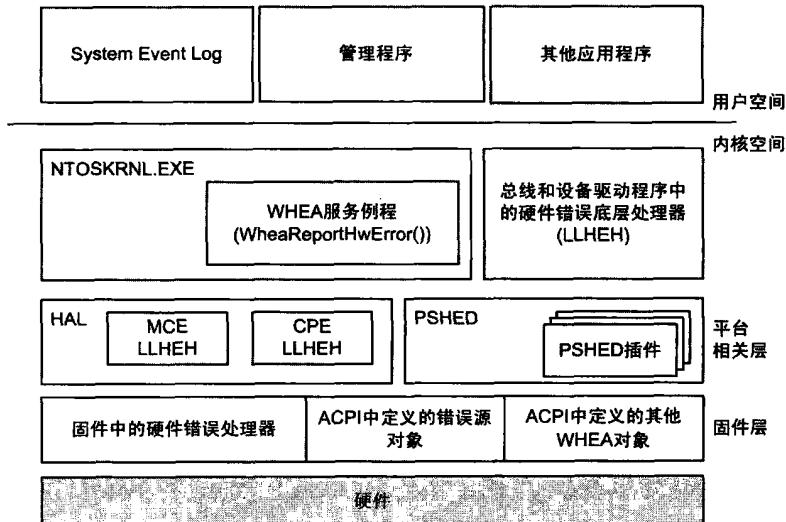


图 17-1 WHEA 架构示意图

内核和驱动层：在系统的内核文件（NTOSKRNL.EXE）中定义了一系列以 'whea' 开头的函数，用来实现 WHEA 的基本功能，我们将它们称为 WHEA 的内核例程。其中最重要的一个函数就是 WheaReportHwError，它是内核模块输出的一个公共函数，可以被 HAL.DLL、PSHED.DLL 及其他驱动程序通过 DLL 方法所调用。从模块设计角度来看，wheareportHwError 函数是位于不同模块中的硬件错误处理器向系统内核报告硬件错误的统一接口。或者说 WheaReportHwError 是内核接收硬件错误的统一入口。在驱动程序中，也可以包含底层硬件错误处理器，比如 Vista 系统中的 PCI Express 总线驱动程序就实现了一个标准的错误处理器（LLHEH），用于处理通过 AER 机制所报告的硬件错误，包括总线控制器的错误，以及总线上的各种 PCI Express 设备的错误。

应用程序层：WHEA 支持通过 ETW 机制来报告硬件错误，因此应用程序可以方便地订阅和查询硬件错误，从而得到关于硬件错误的通知和报告。服务器管理程序可以在得到硬件错误通知后采取各种措施，比如，通过网络或寻呼机通知服务器管理员。

如果把位于 HAL 模块和驱动程序中的 LLHEH 放在一起，那么图 17-1 就演变成图 17-2 的样子，这样就更容易看出 WHEA 中各种部件的角色和它们之间的交互关系。LLHEH 是硬件错误的接收者，它通过 PSHED 检测和读取错误信息，然后报告给内核。内核在处理硬件错误时也可以通过 PSHED 来访问硬件和固件。内核通过 ETW 机制将硬件错误通知给用户态的应用程序。

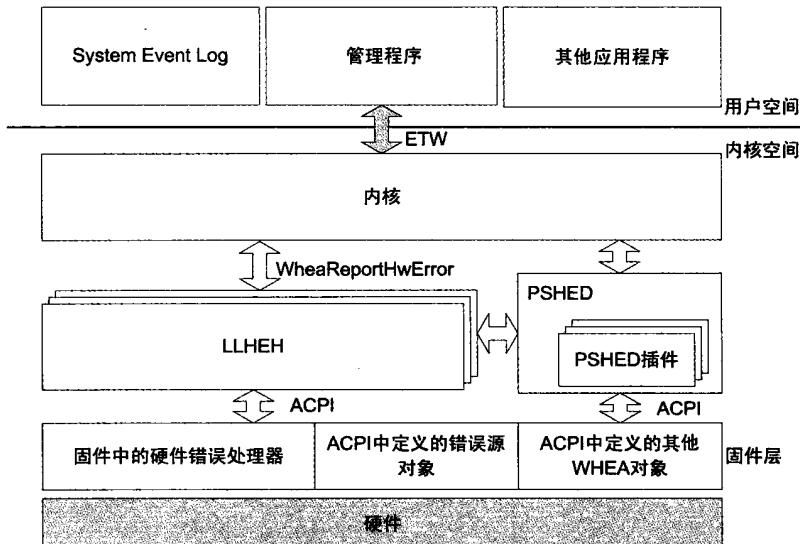


图 17-2 WHEA 各部件之间交互关系示意图

举例来说，清单 17-1 所示的栈回溯记录代表了 WHEA 处理一个 MCE (Machine Check Exception) 错误的过程，在 HAL 中的 MCE LLHEH 扫描到硬件错误后，调用 HAL 中的 HalpMcaExceptionHandler 函数，后者通过 HalpReportMachineCheck 调用内核的 WheaReportHwError 函数，将硬件错误报告给内核。内核在经过一系列处理和分析后，最终通过 KeBugCheckEx 函数发起蓝屏，如清单 17-1 所示。

清单 17-1 WHEA 报告错误的典型过程

```
STACK_TEXT:
82571464 81b9ae93 00000124 00000000 84db6310 nt!KeBugCheckEx+0x1e
82571480 818a201c 84db6310 851b6bc8 00000002 hal!HalBugCheckSystem+0x37
825714a0 81b9ae52 851b6bc8 851b6ce0 825714d4 nt!WheaReportHwError+0x10c
825714b0 81b9afac 00000003 851b6bc8 00000000 hal!HalpReportMachineCheck+0x28
825714d4 81b9789f 8256c0c0 00000000 00000000 hal!HalpMcaExceptionHandler+0xfc
825714d4 00000000 8256c0c0 00000000 00000000 hal!HalpMcaExceptionHandlerWrapper +0x77
```

栈帧 0 中的第一个参数 0x124，是常量 WHEA_INTERNAL_ERROR，是专门为 WHEA 增加的错误检查代码。相关的还有 WHEA_UNCORRECTABLE_ERROR，值为 0x127。

17.1.3 PSHED

PSHED 是 Windows Vista 为了实现 WHEA 而新引进的一个内核模块 (DLL)，其

作用是与固件（BIOS 或者 EFI）中的硬件错误处理脚本和代码相交互。因为固件是与计算机系统的设计相关的，因此这个模块的名字叫平台相关的硬件错误驱动（Platform Specific Hardware Error Driver）。PSHED 与内核文件（NTOSKRNL.EXE）有着直接的相互依赖关系（图 17-3），因此，当 WinLoad 加载内核文件时就会加载它，也就是说，PSHED.DLL 是作为内核文件的依赖模块而被 OS Loader 加载的。

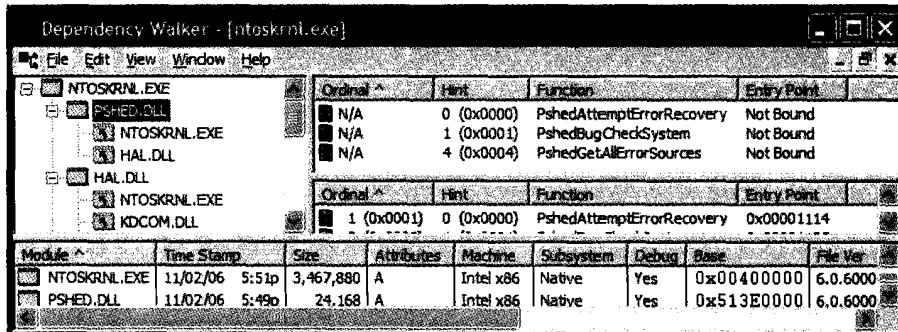


图 17-3 PSHED 与内核文件的相互依赖关系

进一步讲，PSHED.DLL 是 WHEA 的一个底层模块，它负责枚举错误源，读取、清除和持久化错误记录。表 17-1 列出了 RTM 6000 版本的 PSHED.DLL 的所有输出函数，通过这些函数我们可以更具体地感受到 PSHED 模块的作用。

表 17-1 PSHED.DLL 的输出函数

输出函数	用途
PshedAttemptErrorRecovery	错误恢复
PshedBugCheckSystem	发起蓝屏（BSOD）
PshedClearErrorRecord	清除错误记录
PshedFinalizeErrorRecord	将错误记录填写完整，或增加补充信息
PshedGetAllErrorSources	枚举所有错误源
PshedGetErrorHandlerInfo	读取错误源信息
PshedInitialize	初始化
PshedRetrieveErrorInfo	读取错误信息
PshedRetrieveErrorRecord	读取错误记录
PshedSaveErrorRecord	保存错误记录，即错误记录持久化

值得说明的是，因为 Vista 是针对客户端的 Windows 版本，而且它是 WHEA 在 Windows 中的第一个实现，所以某些 WHEA 功能在目前的 Vista 中没有完全实现，比如通过 ACPI 表枚举错误源（详见下一节）。Windows Server 2008 更全面地实现了 WHEA。

PSHED 支持插件（Plug-in），允许 OEM 或 IHV/ISV 开发自己的模块来定制或扩展 PSHED 的功能。

17.2 错误源

系统中有很多部件，如果要将某一个部件纳入到 WHEA 的支持范围，那么必须通过一种机制来描述这个部件的属性和特征，并告诉系统应该如何与它交互以得到错误通知和错误信息。WHEA 使用错误源（Error Source）的概念来解决这个问题。简单来说，错误源是 WHEA 管理和组织硬件错误来源的基本单位，每个 WHEA 错误都属于某个错误源。WHEA 通过错误源来了解硬件单元的基本属性，决定如何来检测该硬件是否发生错误（轮询还是中断等），以及应该如何处理和记录这类错误。

那么应该如何定义错误源呢？首先，对于标准化的硬件错误，WHEA 预定义了默认的错误源，简称为标准的错误源。对于非标准化的错误，可以在系统的固件中通过 ACPI 表来进行描述，也可以通过编写 PSHED 插件程序来报告，下面分别做简单介绍。

17.2.1 标准的错误源

对于已经存在明确标准的硬件错误，WHEA 可以使用默认的错误源信息来处理这类错误。这意味着即使系统固件（BIOS）没有提供对这类错误的错误源描述，那么操作系统仍然可以处理这类错误。典型的例子便是 CPU 的机器检查异常，以及通过 PCI Express 总线的 AER 机制所报告的错误。

我们在第 6 章详细介绍了 CPU 的 MCA 机制。Windows 2000 引入了 MCA 支持，在 HAL 中加入了一系列包含 MCA 字样的函数，比如 `hal!HalpMcaExceptionHandler` 等。Windows Vista 将以上支持纳入到 WHEA 框架中。

AER（Advanced Error Reporting）是 PCI Express 总线定义的一种高级的错误报告方式，比基本（baseline）的错误报告方式具有更高的可靠性。感兴趣的读者可以阅读参考文献 2 和 3。Windows Vista 的 PCI Express 总线驱动程序中包含了对 AER 的支持，实现了一个用于处理 AER 的 LLHEH。

除了 MCE 和 AER 错误源，针对 x86 和 x64 系统的默认错误源，还有已经纠正的机器检查（Corrected Machine Check）和已经纠正的平台错误（Corrected Platform Error）。在 HAL 中内建了处理这两类错误源的 LLHEH。操作系统对于已纠正错误的处理工作主要是将其记录下来。

17.2.2 通过 ACPI 表来定义错误源

如果要为系统中的非标准设备定义错误源，或者希望定制和扩展默认的标准错误源，那么可以通过系统固件来向系统报告这些信息。ACPI 是操作系统与系统固件之间的接口规范。在 ACPI 3.0b 版本中新增了用于支持 WHEA 的一系列系统描述表。其中之一就是用来描述错误源的硬件错误源表（Hardware Error Source Table），简称 HEST。

图 17-4 画出了 HEST 表的布局，简单来说，开始处是一个表头，表头的第一个字段是表的 ACPI 签名，长度为一个 DWORD，对应的 ASCII 字符内容就是 HEST，接下来是表的总长度（4 字节），表的版本，校验和，OEM ID，用来创建 HEST 表的工具的厂商 ID 和版本等信息。表头的末尾是 4 字节的错误源个数值。这个值代表了表头后跟随的错误源结构个数，我们使用 N 来表示。

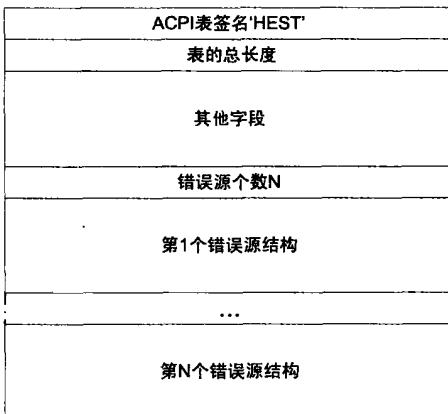


图 17-4 HEST 表的布局

在表头结构之后，便是 N 个错误源结构。每个错误源结构的内容和长度因为所描述的错误类型不同而不同。但是每个结构的开头两个字节（WORD）都是这个结构的类型 ID，操作系统可以根据这个类型值了解这个结构的其他字段的含义和结构的长度。接下来是两个字节的错误源 ID，用来唯一标识这个错误源，以便与这个系统中定义的其他错误源相区别。

每个错误源结构通常还包含如下字段。

- 标志。多个标志位的组合，其中一个很重要的标志就是用来声明处理方式的 FIRMWARE_FIRST 标志（位 0），我们将在 17.3.4 节详细介绍。
- 是否启用。用来启用和禁止这个错误源。
- 希望操作系统为这个错误源预先分配错误记录的个数。
- 操作系统初始化和控制与这个错误源相关的硬件设备的参数信息，具体内容因错误源类型的不同而不同。

对 HEST 表表头和各个错误源结构的详细定义包含在目前尚未公开的《WHEA 平台设计指南》（*WHEA Platform Design Guide*）中，以上信息来源于参考文献中所列出的资料。

17.2.3 通过 PSHED 插件来报告错误源

WHEA 通过 PSHED（平台相关的硬件错误驱动程序）来枚举错误源。因此也可

以通过编写 PSHED 插件来向系统报告错误源。

当系统初始化时，内核会调用 PSHED.DLL 的 `PshedGetAllErrorSources` 函数来枚举错误源。PSHED 会建立一个错误源描述表，并加入标准的错误源信息，然后调用每个 PSHED 插件（如果存在）的 `GetErrorSourceInfo` 函数，让其报告错误源，清单 17-2 描述了系统内核调用 `PshedGetAllErrorSources` 函数的过程。

清单 17-2 枚举错误源

```
kd> kn
# ChildEBP RetAddr
00 86404bf0 81f76025 PSHED!PshedGetAllErrorSources
01 86404c10 81f76397 nt!WheapQueryPshedForErrorSources+0x1f
02 86404c3c 81f7fa66 nt!WheapInitialize+0x3d
03 86404c8c 81f75859 nt!IoInitSystem+0x572
04 86404d7c 81c20013 nt!Phase1InitializationDiscard+0xc1d
05 86404d84 81e2bc93 nt!Phase1Initialization+0xd
06 86404dc0 8194202a nt!PspSystemThreadStartup+0x18f
07 00000000 00000000 nt!KiThreadStartup+0x16
```

从以上栈回溯序列来看，系统是在执行体的阶段 1 初始化期间来枚举错误源的。

当 LLHEH 或内核需要获取某个错误源的信息时，它会调用 PSHED 的 `PshedGetErrorSourceInfo` 函数。这时 PSHED 也会调用 PSHED 插件的 `GetErrorSourceInfo` 函数。

根据参考文献 4，Vista 系统的 WHEA 实现不会从 ACPI 表来读取错误源信息，也就是说，即使固件中存在 HEST 表，那么 Vista 也不会读取它。Windows Server 2008 的 WHEA 实现会读取 HEST 表。

17.3 错误处理过程

本节我们将介绍 WHEA 处理硬件错误的过程。简单来说，是 LLHEH 通过轮询或接收到系统通知或中断而感知到错误，然后调用 PSHED 的函数读取错误信息，而后根据错误的严重程度采取处理措施。因为在整个流程中，涉及多个模块中的多个函数相互配合，所以大家必须借助一定的数据结构来共享信息，WHEA 使用了两个数据结构，一个是 `WHEA_ERROR_PACKET` 结构，另一个是 `WHEA_ERROR_RECORD` 结构。

17.3.1 WHEA_ERROR_PACKET 结构

当一个错误源的 LLHEH（底层硬件错误处理器）检测到一个错误（或者得到通知）时，它会创建并初始化一个 `WHEA_ERROR_PACKET` 结构。它使用这个结构来收集错误信息，并使用这个结构向内核报告硬件错误。表 17-2 列出了 `WHEA_ERROR_PACKET` 结构的布局和主要字段。

表 17-2 WHEA_ERROR_PACKET 结构

字段名	偏移	类型	描述
Signature	+0x000	Uint4B	结构签名
Flags	+0x004	Uint4B	标志
Size	+0x008	Uint8B	结构体长度
RawDataLength	+0x010	Uint8B	状态数据的长度
Context	+0x018	Uint8B	错误有关的上下文结构
ErrorType	+0x020	_WHEA_ERROR_TYPE	见下文
ErrorSeverity	+0x024	_WHEA_ERROR_SEVERITY	见下文
ErrorSourceId	+0x028	Uint4B	错误源 ID
ErrorSourceType	+0x02C	_WHEA_ERROR_SOURCE_TYPE	见下文
Reserved1	+0x030	Uint4B	保留
Version	+0x034	Uint4B	版本号
Cpu	+0x038	Uint8B	
u	+0x040	<unnamed-tag>	见下文
RawDataFormat	+0x110	_WHEA_ERROR_STATUS_FORMAT	见下文
Reserved2	+0x114	Uint4B //	保留
RawData	+0x118	[1] UChar	状态数据

总的来看，WHEA_ERROR_PACKET 是个不确定长度的数据结构，前面 0x40 个字节是概要信息，从 0x40 到 0x110 是一个枚举结构，会因错误的不同而不同。从偏移 0x118 开始是与这个错误相关的原始数据（RawData），其长度可以由上面的 RawDataLength 字段的值来确定，格式可以通过 RawDataFormat 字段来确定。RawDataFormat 字段的值可以为如下枚举常量之一：

```

WheaErrorStatusFormatIPFSalRecord = 0 // 安腾 CPU 的 SAL 记录
WheaErrorStatusFormatIA32MCA = 1      // IA32 处理器的 MCA 记录
WheaErrorStatusFormatEM64TMCA = 2     // Intel64 位处理器的 MCA 记录
WheaErrorStatusFormatAMD64MCA = 3     // AMD64 处理器的 MCA 记录
WheaErrorStatusFormatPCIExpress = 4    // PCI Express
WheaErrorStatusFormatNMIPort = 5       // NMI 端口
WheaErrorStatusFormatOther = 6        // 其他

```

其中 ErrorType 字段用来指定错误类型，其值是枚举常量，可以为以下值之一：

```

WheaErrTypeProcessor = 0 // 处理器
WheaErrTypeMemory = 1   // 内存
WheaErrTypePCIExpress = 2 // PCI Express
WheaErrTypeNMI = 3      // 不可屏蔽中断，一些致命错误是通过 NMI 来报告的
WheaErrTypePCIXBus = 4  // PCIX 总线
WheaErrTypePCIXDevice = 5 // PCIX 设备

```

ErrorSeverity 字段用来标识错误的严重程度，可以为如下值之一：

```

WheaErrSevRecoverable = 0 // 可恢复的错误
WheaErrSevFatal = 1      // 致命错误
WheaErrSevCorrected = 2   // 已经纠正的错误
WheaErrSevNone = 3        // 未知

```

ErrorSourceType 字段用来指定错误源的类型，可以为如下枚举值之一：

```

WheaErrSrcTypeMCE = 0           // 机器检查异常
WheaErrSrcTypeCMC = 1           // 纠正了的机器检查
WheaErrSrcTypeCPE = 2           // 纠正了的平台错误
WheaErrSrcTypeNMI = 3           // 不可屏蔽中断
WheaErrSrcTypePCIe = 4          // PCI Express
WheaErrSrcTypeOther = 5          // 其他

```

在了解了 WHEA_ERROR_PACKET 结构后，接下来我们将介绍 WHEA 是如何使用这个结构的。

17.3.2 处理过程

图 17-5 显示了 WHEA 处理硬件错误的基本过程。在 LLHEH 侦测到错误情况后，它首先会构建一个 WHEA_ERROR_PACKET 结构，然后调用 PSHED 的 PshedRetrieveErrorInfo 函数，并把这个结构传递给它。PshedRetrieveErrorInfo 函数会根据 WHEA_ERROR_PACKET 结构中的错误源 ID (ErrorSourceId) 来读取错误信息（如果需要，则可能调用 PSHED 插件模块中的错误读取函数），并把读取到的信息填写到 WHEA_ERROR_PACKET 结构中。

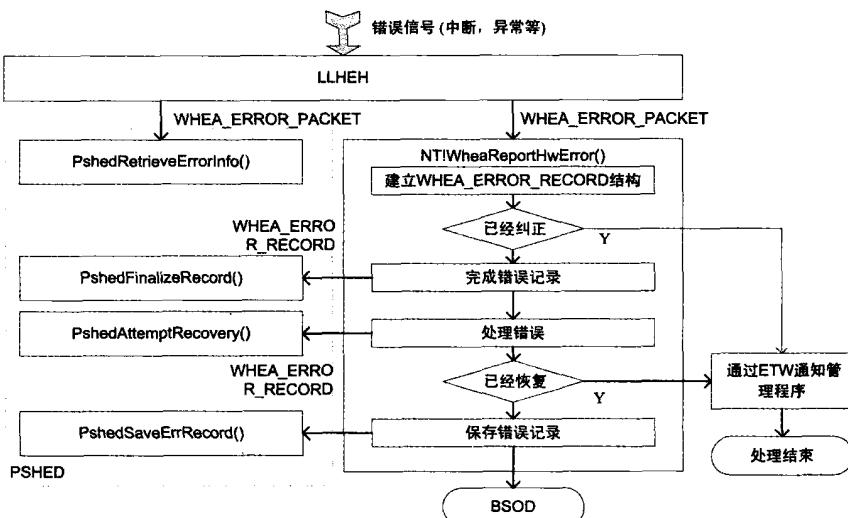


图 17-5 WHEA 处理硬件错误的过程

在 PshedRetrieveErrorInfo 函数返回后，LLHEH 调用内核的 WheaReportHwError 函数，向系统报告这个错误，调用时将填写好的 WHEA_ERROR_PACKET 结构传递给内核。

```
NTSTATUS WheaReportHwError (PWHEA_ERROR_PACKET Packet);
```

内核 (WheaReportHwError 函数) 接到报告后，首先会构建一个 WHEA_ERROR_RECORD 结构（稍后介绍）。接下来，WheaReportHwError 函数会根据 ErrorSeverity 字段判断这个错误是否为已经纠正的错误，如果是，那么便通过 ETW 将这个错误通知给管理程序，而后处理结束。如果不是已经纠正的错误，那么 WheaReportHwError 函数

数会调用 PSHED 的 `PshedFinalizeRecord` 来将错误记录填写完整，PSHED 也可以借此机会增加补充信息。

对于可以恢复的错误（`ErrorSeverity` 等于 `WheaErrSevRecoverable`），`WheaReportHwError` 函数会协同系统的其他模块来恢复错误，比如对 PCI Express 的完成超时（Completion time-out），可以让驱动程序重试（retry）这个事务。PSHED 的 `PshedAttemptRecovery` 函数也是用于恢复错误的，通过这个函数，可以进一步调用 PSHED 插件的错误恢复函数。如果错误被成功恢复，那么 `WHEA_ERROR_RECORD` 结构的对应字段会得到更新。

接下来，内核会判断错误是否已经恢复，如果是，那么便通过 ETW 通知管理程序，并且处理结束，如果没有，那么内核开始保存错误记录，即调用 `PshedSaveErrorRecord`，而后 `WheaReportHwError` 函数调用发起蓝屏，终止整个系统。

17.3.3 WHEA_ERROR_RECORD 结构

在 LLHEH 通过 `WHEA_ERROR_PACKET` 将一个硬件错误报告给操作系统内核后，内核会构建一个 `WHEA_ERROR_RECORD` 结构，并且从此使用这个结构来记录和处理这个硬件错误。

总的来说，`WHEA_ERROR_RECORD` 也是一个可变长度的数据结构，开始部分是一个固定长度的 `WHEA_ERROR_RECORD_HEADER` 结构，而后是一个或多个（由结构头的 `SectionCount` 字段决定）固定长度的节描述符（`WHEA_ERROR_RECORD_SECTION_DESCRIPTOR`），每个节描述符描述一个节。在节描述符后便是各个节。每个节又由两部分组成，第一部分是节的头，而后是节的数据，参见图 17-6。

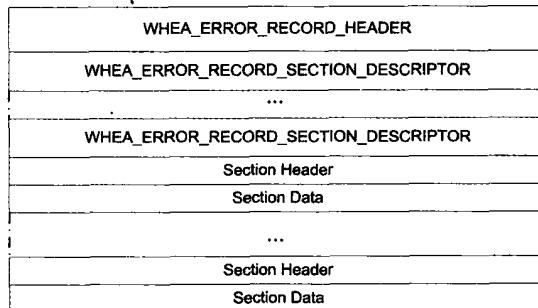


图 17-6 WHEA_ERROR_RECORD 结构和节数据布局

以下是 WinDBG 显示的 `WHEA_ERROR_RECORD` 结构：

```

kd> dt nt!_WHEA_ERROR_RECORD
+0x000 Header      : _WHEA_ERROR_RECORD_HEADER
+0x088 SectionDescriptor : [1] _WHEA_ERROR_RECORD_SECTION_DESCRIPTOR
  
```

其中，`Header`（头）是一个 `WHEA_ERROR_RECORD_HEADER` 结构，表 17-3 列出了它的各个字段。

表 17-3 WHEA 错误记录结构头的各个字段

字段名	偏移	类型	含义
Signature	+0x000	Uint4B	结构签名, 'REPC'
Revision	+0x004	Uint2B	版本
Reserved1	+0x006	Uint2B	保留
Reserved2	+0x008	Uint2B	保留
SectionCount	+0x00a	Uint2B	节的个数
Severity	+0x00c	_WHEA_ERROR_SEVERITY	严重程度
ValidationBits	+0x010	Uint4B	用来标识 Timestamp、PlatformId 和 PartitionId 的有效性
Length	+0x014	Uint4B	结构长度, 以字节为单位
Timestamp	+0x018	_LARGE_INTEGER	时间戳
PlatformId	+0x020	_GUID	发生错误的平台 GUID 值
PartitionId	+0x030	_GUID	发生错误的分区 GUID 值
CreatorId	+0x040	_GUID	创建这个错误记录的实体
NotifyType	+0x050	_GUID	通知类型
RecordId	+0x060	Uint8B	记录 ID
Flags	+0x068	Uint4B	标志, 见下文
PersistenceInfo	+0x070	_WHEA_PERSISTENCE_INFO	持久化信息
Reserved3	+0x078	[12] UChar	保留

其中 Flags 字段可以包含以下标志。

- WHEA_ERROR_RECORD_FLAGS_RECOVERED: 成功恢复这个错误。
- WHEA_ERROR_RECORD_FLAGS_PREVIOUSERROR: 这个错误发生在上次系统启动时。
- WHEA_ERROR_RECORD_FLAGS_SIMULATED: 这个错误是模拟出来的 (用于测试)。

每个节描述符是一个 WHEA_ERROR_RECORD_SECTION_DESCRIPTOR 结构:

```
kd> dt nt!_WHEA_ERROR_RECORD_SECTION_DESCRIPTOR
+0x000 SectionOffset      : Uint4B          // 节的偏移
+0x004 SectionLength     : Uint4B          // 节的长度
+0x008 Revision         : Uint2B          // 版本
+0x00a ValidationBits   : UChar           // 验证位
+0x00b Reserved         : UChar           // 保留
+0x00c Flags             : Uint4B          // 标志
+0x010 SectionType       : _GUID           // 节类型
+0x020 FRUID             : _GUID           // 见下文
+0x030 SectionSeverity  : _WHEA_ERROR_SEVERITY // 严重程度
+0x034 FRUText           : [20] Char        // 见下文
```

其中 SectionOffset 用来指定这个节的节数据的偏移, SectionLength 是节数据的长度。FRUID 和 FRUText 字段用来描述与错误相关的可更换单元 (Field Replaceable Unit), 简称 FRU。FRUID 是 FRU 的 ID 号, FRUText 是 FRU 的名称。FRU 一词用来泛指计算机系统中可以简单更换的电路板或零部件。

WDK 描述了 WHEA 错误记录结构, 包括 WHEA_ERROR_RECORD_HEADER 和 WHEA_ERROR_RECORD_SECTION_DESCRIPTOR 结构的各个字段。

17.3.4 Firmware First Model

在 WHEA 的一些文档中，经常可以看到所谓的 Firmware First Model，即固件优先模式。其含义是让系统固件先得到错误通知，固件的代码先对错误进行处理后，再报告给操作系统。要实现这种方式，首先在系统电路设计方面，需要将硬件设备的错误信号与系统芯片连接，使其能够触发 SMI (System Management Interrupt) 中断。因为南桥的很多 GPIO 管脚可以触发 SMI，所以可以将错误信号连接到这些 GPIO 管脚，这样，当有错误发生时，便会触发 SMI 让 CPU 进入系统管理模式执行固件中的 SMI 处理例程。然后，固件中的 SMI 例程对错误进行处理，并填写错误记录。对于使用固件优先模式的错误源，在其描述中便声明了这一特征，并且要求操作系统为其预先分配好内存来记录错误。

在固件完成了错误处理后，它会根据错误的严重程度选择以下方法之一来通知操作系统。一种是系统控制中断 (System Control Interrupt，即 SCI)，另一种是 NMI。对于致命错误，固件会使用 NMI，对于其他错误会使用 SCI。

并不是所有硬件错误都可以使用固件优先模式，比如 CPU 的机器检查异常就不能使用这种方式。

17.4 错误持久化

所谓错误持久化就是将错误信息记录到可长久保存的介质中，供以后调查使用。对于大多数情况，操作系统可以使用磁盘 (Disk) 来将错误信息保存到文件中。但是在某些情况下，这种方法可能不可行，比如磁盘硬件或磁盘端口驱动程序出现故障。

为了在以上特殊情况下也能够将错误信息保存下来，WHEA 设计了一种比磁盘文件更可靠的方式。这种方式要求系统的固件提供一个接口，操作系统可以调用这个接口将错误信息保存到系统的 NVRAM(Non-volatile RAM)中。NVRAM 是对闪存 (Flash) 等非易失性存储介质的通称。在系统的主板上，通常配备了容量从 1MB 到几十 MB 不等的 NVRAM，用来存储固件程序和系统的配置信息。WHEA 定义了一套规范让固件将错误记录保存到 NVRAM 中，以便实现错误持久化，其中最核心的就是 ERST 表。

17.4.1 ERST

ERST 是 Error Record Serialization Table 的缩写，即错误记录序列化表。它是 ACPI 标准 3.10b 版本引入的用来支持 WHEA 的一个系统描述表。ERST 表的起始部分是一个标准的 ACPI 描述表表头，长度为 32 个字节，而后是序列化头结构和用于序列化的指令表项，如表 17-4 所示。

表 17-4 ERST 表的数据布局

字段	长度	值	说明
Header Signature	4	0x0	“ERST”
Serialization Header Size	4	0x24	序列化头结构的长度
Reserved	4	0x28	保留，必须为 0
Instruction Entry Count	4	0x2c	后面的指令项个数
序列化指令项	32*Count	0x30	一系列指令项

序列化动作表用来告诉操作系统执行序列化操作时所需采取的动作，由一系列指令项组成，每个指令项是一个 32 字节的数据结构，第一个字节是动作编号，例如动作 0 代表开始写操作 (BEGIN_WRITE_OPERATION)，1 代表开始读操作 (BEGIN_READ_OPERATION)，2 代表开始清除操作 (BEGIN_CLEAR_OPERATION)，3 代表结束操作 (END_OPERATION) 等。表 17-5 列出了指令项的所有字段。

表 17-5 ERST 表的指令项

字段	长度	偏移量	说明
Serialization Action	1	N	本结构所描述的序列化动作
Instruction	1	N+0x1	要执行动作的代码
Flags	1	N+0x2	与动作代码有关的标志
Reserved	1	N+0x3	保留
Register Region	12	N+0x4	寄存器地址，其格式使用的是 ACPI 标准中定义的通用地址结构 (Generic Address Structure)
Value	8	N+0x10	要写的数据
Mask	8	N+0x18	掩码

例如以下指令项定义的动作是将 Value 字段指定的数值 (0x80) 写到 GAS 指定的地址位置。

```
{
0x00,                                     // BEGIN_WRITE_OPERATION
0x03,                                     // WRITE_REGISTER_VALUE
0x00,                                     // Flags
0x00,                                     // Reserved
0x00, 0x01, 0x00, 0x03, 0x00000000AAFF0000, // GAS
0x0000000000000080,                      // Value
0x00000000000000FF                        // Mask
}
```

当操作系统需要执行 BEGIN_WRITE_OPERATION 动作时，它会依次查找 ERST 表中的动作表项，在找到上面的表项后，便会根据其内容执行相应的动作。

17.4.2 工作过程

为了实现错误持久化，操作系统需要固件执行的任务（操作）有如下 3 个。

- 当错误发生时，接受系统调用将错误记录写入到固件的 NVRAM 中，而后系统将因为蓝屏（BSOD）而终止运行。我们将这个任务简称为写错误记录。
- 在系统重新启动后，操作系统需要将错误记录从 NVRAM 中读出。我们将这个任务简称为读错误记录。
- 在错误记录读出后，操作系统需要将这个错误记录清除，以防止多次读取和报告。我们将这个任务简称为清除错误记录。

对于以上每个任务（操作），操作系统都需要执行一系列动作。WHEA 的设计指南详细定义了每项任务所需执行的动作。当执行每个动作时，操作系统会查询 ERST 中的指令项定义，决定如何执行这个动作。概而言之，WHEA 定义了一系列动作，但是每个动作的具体行为和参数可以由系统固件通过 ERST 表来定制。当操作系统需要执行某项任务时，它会将这项任务分解为若干个动作，然后依次来执行。

操作系统执行一项任务的一般过程是：首先执行开始动作（如 BEGIN_WRITE_OPERATION），启动这项任务。第二步是执行设置辅助信息的动作，比如设置错误记录所在的内存位置。第三步是执行 EXECUTE_OPERATION 动作，让固件开始执行这项任务。接下来等待并反复执行 CHECK_BUSY_STATUS 命令，直到固件指示操作已经完成，系统再执行 GET_COMMAND_STATUS 动作查询任务完成的结果，最后执行 END_OPERATION 动作结束整个任务。

17.5 注入错误

为了测试 WHEA 的各个模块和系统对 WHEA 的支持情况，WHEA 定义了一种模拟产生硬件错误的方法，称为错误注入（Error Injection）。

WHEA 设计了两种错误注入机制，一种是编写 PSHED 扩展模块，另一种是在固件中提供 EINJ 表。编写 PSHED 扩展模块的方法是不鼓励使用的。我们在这里简要介绍使用 EINJ 表的方法。

EINJ 表的结构与前面介绍的 ERST 表非常相似，其起始部分也是一个标准的 ACPI 表头，而后是一个简单的用于描述错误注入信息的头结构，接下来是一系列指令项。

当操作系统需要模拟硬件错误时，它会根据需要执行 EINJ 的指令项所定义的操作。

17.6 本章总结

被确认为是故障或错误的“拥有者（Owner）”，这对于任何公司和个人似乎都不是一件愉快的事情。尤其是当这个故障发生在最终用户手中时。当用户不清楚错误的

根源时，他们大多时候首先想到的是软件的瑕疵。当他们搞不清楚是哪个软件的瑕疵时，可能会首先怀疑操作系统的稳定性。

WHEA 建立了一种统一的机制来处理和记录硬件错误。这有利于真正发现每个错误的根源，并提高硬件的质量和整个系统的稳定性。最终有利于改善用户的使用体验。

通过本章的介绍，我们知道要完全发挥 WHEA 的效果需要得到系统固件和硬件的支持，也就是系统开发商和固件开发商的支持。支持 WHEA 的计算机系统被称为 WHEA-Enabled Platform。为了让更多的系统都支持 WHEA，目前 Windows 的徽标测试（WHQL）定义了专门针对 WHEA 的要求。目的是强制系统，特别是服务器系统，要实现 WHEA 所需的支持，其细节可以参阅参考文献 8。

参考文献

1. John Strange. Windows Hardware Error Architecture. WinHec2005
2. PCI Express Base Specification. <http://www.pcisig.com>
3. Ravi Budruk, Don Anderson, Tom Shanley. PCI Express System Architecture. Mindshare INC., 2004
4. Windows Hardware Error Architecture (whea_overview.doc). Microsoft Corporation, 2006
5. John Strange. Developing For The Windows Hardware Error Architecture. WinHec2006
6. John Strange. WHEA System Design and Implementation. WinHec2007
7. John Strange. WHEA Platform Implementation. WinHec2007
8. Windows Server code named “Longhorn” Logo Program for Systems Version 3.09. Microsoft Corporation

内核调试引擎

简单来说，内核调试就是分析和调试位于内核空间中的代码和数据。运行在内核空间的模块主要有操作系统的内核、执行体和各种驱动程序。从操作系统的角度来看，可以把驱动程序看作是对操作系统内核的扩展和补充。因此可以把内核调试简单地理解为调试操作系统的广义内核。

使用调试器调试的一个重要特征就是可以把调试目标中断到调试器中，换言之，当我们在调试器中分析调试目标时，调试目标是处于冻结状态的。当我们进行用户态调试时，操作系统会在报告调试事件时自动将被调试进程挂起，使其停止运行。内核调试的调试目标就是操作系统的内核，所以对于内核调试而言，将调试目标中断到调试器就意味着将操作系统的内核中断，让其停止运行以接受调试器的分析和检查。但我们知道，内核负责着整个系统的调度和执行，一旦它被停止，那么系统中的所有进程和线程也都停止运行了。或者说，内核一旦停止，那么整个系统也就停止了。如果调试器运行在系统中，那么它也无法运行了。如果让调试器运行在另一个系统中，那么调试器又如何与这个静止的内核通信和联系呢？

目前，主要有 3 种方法来解决以上问题。第一种是使用硬件调试器，它通过特定的接口（如 JTAG）与 CPU 建立连接并读取它的状态，比如我们在第 7 章介绍的 ITP 调试器。第二种是在内核中插入专门用于调试的中断处理函数和驱动程序，当操作系统内核被中断时，这些中断处理函数和驱动程序接管系统的硬件，营造一个供调试器可以运行的简单环境，这个环境使用自己的驱动程序来接收用户输入，显示输出（窗口）。SoftICE 和 Syser 调试器使用的是这种方法。第三种方法是在系统内核中加入调试支持，当需要中断到调试器中时，只保留这部分支持调试的代码还在运行，内核的其他部分都停止了，包括负责任务调度、用户输入和显示输出的部分。因为正常的内核服务都已经停止，所以调试器程序是不可能运行在同一个系统中的。因此这种方法需要调试器运行在另一个系统中，二者通过通信电缆交流信息。

Windows 操作系统推荐的内核调试方式使用的是第三种方法。内建在操作系统内核中负责调试的那个部分通常被称为内核调试引擎（Kernel Debug Engine）。

本章的前半部分将介绍内核调试引擎的概况（18.1节），内核调试所使用的通信连接（18.2节），如何启用内核调试（18.3节），以及内核调试引擎的初始化（18.4节）。后半部分将介绍内核调试协议（18.5节），调试引擎如何与内核进行交互（18.6节），建立和维持调试连接（18.7节），最后介绍本地内核调试（18.8节）。

18.1 概览

从NT系列Windows操作系统的第一个版本（NT 3.1）开始，内核调试引擎就是系统的一个固有部分。而且它是这个系统最先开始工作的部件之一，很多其他部件都是在它的帮助下开发出来的。根据不同的用途，Windows操作系统分为很多个发行版本，如家庭版本、专业版本、服务器版本等，根据构建选项的不同，每个版本又分为Free版本和Checked版本。但无论哪种版本，内核调试引擎都包含在其中。

18.1.1 KD

Windows操作系统的每个系统部件都有一个简短的名字，通常为两个字符，比如MM代表内存管理器，OB代表对象管理器，PS代表进程和线程管理，等等。同样，内核调试引擎也有这样一个双字母的名字，叫KD（Kernel Debug）。内核模块中用来支持内核调试的函数和变量大多都是以这两个字母开头的。

18.1.2 角色

从调试会话的角度来看，内核调试引擎是调试器和被调试内核之间的桥梁（参见图18-1左图），调试器通过内核调试引擎来访问和控制被调试内核，被调试内核通过调试引擎向调试器报告调试事件。对于内核的其他部分，调试引擎代表着调试器。对于调试器而言，调试引擎是它访问调试目标的媒介。

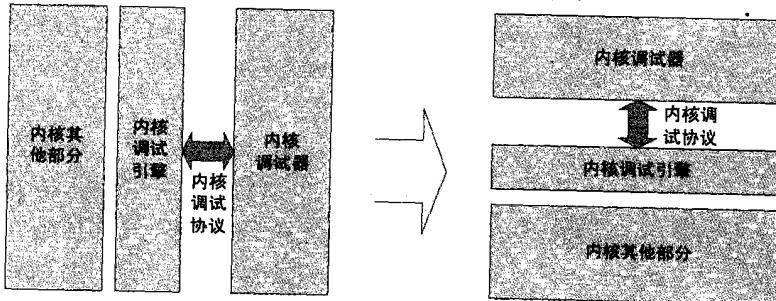


图18-1 内核调试引擎在内核调试中的角色示意图

内核调试引擎和调试器之间通过内核调试协议进行通信。通过这个协议，调试器可以请求调试引擎帮助它访问和控制目标系统，调试引擎也会主动地将目标系统的状态报告给调试器。

从访问内核的角度来看，内核调试引擎为内核调试器提供了一套特殊的 API，我们将其称为内核调试 API，简称 KdAPI。使用 KdAPI，调试器可以以一种类似远程调用的方式访问到内核，这与应用程序通过 Win32 API 访问内核（服务）很类似。图 18-1 中的右图更好地显示了内核调试器、调试引擎、内核其他部分三者间的这种关系。

18.1.3 组成

可以把内核调试引擎分为以下几个部分。

与系统内核的接口函数，是内核调试引擎向内核其他部件公开的一系列函数，供内核的其他部分调用，以便让内核调试引擎得到执行机会，包括初始化的机会、处理异常的机会和检查中断命令的机会等。比如，Windows 启动过程中会调用 KdInitSystem 函数让内核调试引擎初始化；当系统分发异常时会调用 KiDebugRoutine 变量所指向的函数（KdpTrap 或 KdpStub）；另外，系统的时间更新函数 KeUpdateRunTime 会调用 KdCheckForDebugBreak 来检查调试器是否发出了中断命令。

与调试器的通信函数，这部分负责与位于另一个系统（通常位于另一台机器，称为主机）中的调试器进行通信，包括建立和维护通信端口，收发数据包等。

断点管理，负责记录所有断点，如插入、移除断点等。内核调试引擎使用一个数组来记录断点，其名称为 KdpBreakpointTable。

内核调试 API，这是内核调试引擎与调试器之间的逻辑接口，通过这个接口向调试器提供各种观察和分析服务，包括读写内存、读写 IO 空间、读取和设置上下文、设置和恢复断点等。因为调试器与调试引擎并不在同一台机器上，所以不可以直接调用这些 API。实际的做法是调试器通过数据包将要调用的 API 号码和参数传递给调试引擎，调试引擎收到后调用对应的函数，然后再将函数的执行结果以数据包的形式发回给调试器。例如，读虚拟内存的 API 号码是 0x3130。

系统内核控制函数，包括负责将系统内核中断到调试器的 KdEnterDebugger 函数和恢复系统运行的 KdExitDebugger 函数，我们将在 18.6 节详细讨论。

管理函数，包括启用和禁止内核调试引擎的 KdEnableDebugger 和 KdDisableDebugger，以及修改选项的 KdChangeOption。WinDBG 工具包中的 kdbgctrl 工具就是使用这些函数来工作的。

ETW 支持函数（Windows 2000 开始），与 ETW 机制配合将追踪数据通过内核调试通信输出到调试器所在的主机上。负责这一功能的主要函数是 KdReportTraceData，其内部又调用另一个函数 nt!KdpSendTraceData 进行真正的数据操作。

驱动程序更新服务（XP 开始），即从主机上读取驱动程序文件来更新被调试系统中的驱动程序。WinDBG 的 .kdfiles 命令就是依赖这一服务而工作的。当系统内存管理

器的 MiCreateSectionForDriver 为一个驱动程序分配内存节（Section）时，如果检测到当前处于内核调试状态，就会调用调试引擎的 KdPullRemoteFile 函数，后者再使用 KdpCreateRemoteFile、KdpReadRemoteFile 等函数完成文件更新工作（详见 18.6.9 节）。

本地内核调试支持（XP 开始），包括 NtSystemDebugControl 和 KdSystemDebugControl（从 Server 2003 开始），我们将在 18.8 节详细讨论。

图 18-2 画出了以上各部分与系统内核其他部分和与调试器之间的关系示意图。

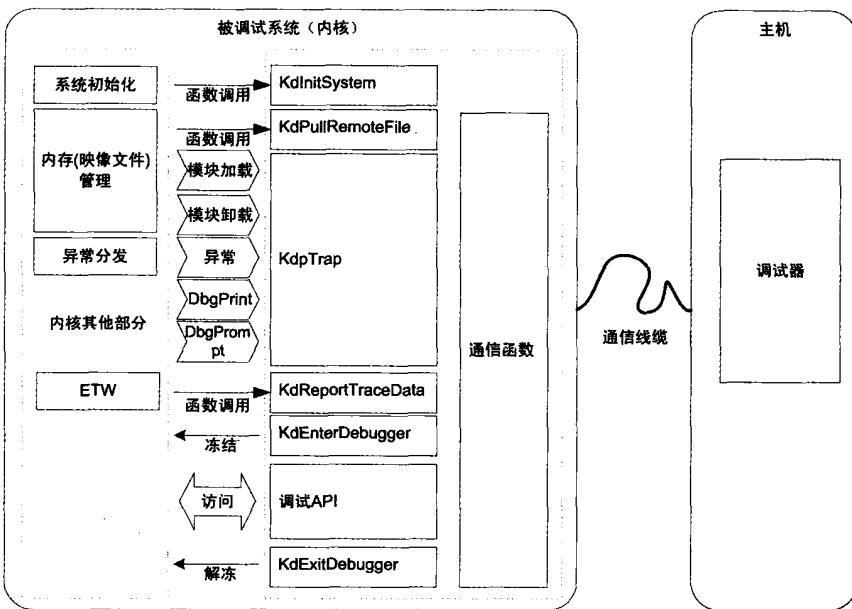


图 18-2 内核调试引擎示意图

18.1.4 模块文件

在 Windows XP 之前，内核调试引擎的所有函数都是位于 NT 内核文件中的，即 NTOSKRNL.EXE。从 Windows XP 开始，内核调试引擎中的通信部分被拆分到一个单独的 DLL 模块中，名为 KDCOM.DLL。

KDCOM 是一个典型的动态链接库（DLL），它输出了一系列函数供内核调试引擎调用。另外，KDCOM 也会调用 NTOSKRNL 输出的符号和函数。因此二者之间是相互依赖的。使用 depends 工具可以清楚地看到这一点（见图 18-3）。

因为 NTOSKRNL 对 KDCOM 直接依赖，所以 KDCOM 是作为 NTOSKRNL 的依赖模块由 NTLDR（对 Vista 是 WinLoad）加载到内存中的。

图 18-3 的右侧窗口显示了 NTOSKRNL 使用的 KDCOM 函数。其中最重要的就是 KdSendPacket 和 KdReceivePacket，分别用来发送和接收数据包。KdD0Transition 和 KdD3Transition 用来“接收”电源状态的变化，当系统进入休眠状态时，内核中的

KdPowerTransition 函数会调用 KdD3Transition 通知 KDCOM，当系统被唤醒时，KdD0Transition 会被调用。KdDebuggerInitialize0、KdDebuggerInitialize1 是 KDCOM 输出的初始化函数，供 Windows 启动过程调用。KdSave 和 KdRestore 用来保存和恢复状态，目前没有使用。

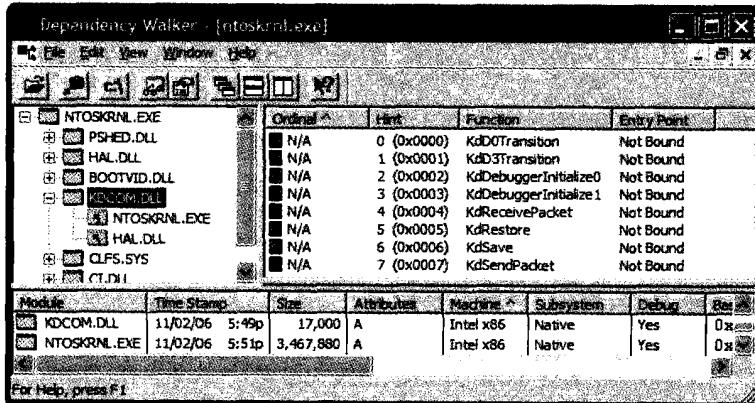


图 18-3 KDCOM 与 NTOSKRNL 之间的相互依赖关系

18.1.5 版本差异

内核调试引擎存在于所有版本的 NT 系列 Windows 操作系统中。在这些版本中，内核调试引擎的基本数据结构和工作方式一直没有大的变化，保持着非常好的稳定性和兼容性。举例来说，使用今天的 WinDBG 调试器，仍然可以非常顺利地调试 NT 4 版本的系统。反过来，使用较老版本的 WinDBG 调试器也可以调试最新的 Vista 系统。

不过，在保持核心结构和工作方式不变的同时，内核调试引擎也在逐步增加一些新的功能。例如，Windows XP 开始支持使用 1394 作为通信方式，将支持内核调试通信的部分独立成 DLL，KDCOM.DLL 用来支持串口通信，KD1394.DLL 用来支持 1394 通信，同时引入本地内核调试支持，以及更新驱动程序的功能。Windows Vista 开始支持使用 USB 2.0 作为通信方式，通信模块为 KDUSB.DLL。

本节简要介绍了内核调试引擎的概况和组成，后面将介绍更多的细节。因为本地内核调试可以看作是双机内核调试的一个特例，所以本章将在最后一节专门讨论它，在其他内容中，除非特别指出“内核调试”，都是指使用两个系统的真正内核调试。

18.2 连接

当使用内核调试引擎进行内核调试时，调试器是运行在与被调试内核不同的 Windows 系统中的。因此二者之间需要通过通信电缆进行通信。迄今为止，共有 3 种连接方式：串行口、1394 和 USB 2.0，下面将分别介绍它们。

18.2.1 串行口

串行通信（Serial Communication）是 Windows 内核调试的本位（native）通信方式。所有版本的内核引擎都支持这种通信方式，它也是 Windows XP 之前所支持的唯一方式。因为内核调试一开始就是针对串行通信的特征而设计的，所以直至今天，串行通信仍然是进行 Windows 内核调试的最稳定方式。这其中的一个主要原因就是内核调试通信协议是面向字节而不是数据包定义的，而串行通信是最适合按字节来读写数据和同步的（第 18.5 节将更详细讨论），其他通信方式都是以数据包（packet 或 message）的形式来组织数据的。

使用串行通信进行内核调试需要一根收发信号对接的串行通信电缆，通常称为 Null-Modem Cable。同时要求目标系统和主机都至少有一个可用的串行端口（COM Port）。串行端口有 9 管脚和 25 管脚两种，大多数个人电脑上使用的是 9 管脚端口。表 18-1 列出了串行端口的信号名称和使用 Null-Modem 电缆进行通信时的连接方式。

表 18-1 串行端口的信号名称和 Null-Modem 电缆连接方式

信号名称	一端		连接	另一端		信号名称		
	9 管脚			9 管脚				
	管脚号	管脚名		管脚号	管脚名			
FG (Frame Ground)	1	—	X	—	1	FG		
TD (Transmit Data)	2	3	—	2	3	RD		
RD (Receive Data)	3	2	—	3	2	TD		
RTS (Request To Send)	4	7	—	8	5	CTS		
CTS (Clear To Send)	5	8	—	7	4	RTS		
SG (Signal Ground)	7	5	—	5	7	SG		
DSR (Data Set Ready)	6	6	—	4	20	DTR		
CD (Carrier Detect)	8	1	—	4	20	DTR		
DTR (Data Terminal Ready)	20	4	—	1	8	CD		
DTR (Data Terminal Ready)	20	4	—	6	6	DSR		

从表 18-1 可以看到，Null-Modem 电缆是把一端的发送信号接到另一端的接收信号，也就是把一端的数据发送管脚（TD）连接到另一端的数据接收管脚（RD）。

串行通信的速度可以为 110bps（位每秒）到 115200 bps 之间的一系列值。但内核调试只支持如下值之一：19200、38400、57600 和 115200。通常设置为 115200。内核调试使用的其他串行通信参数是，8 个数据位，不带奇偶校验，一个停止位，即典型的 8n1 模式。

因为串行通信的最大速度为 115200 bps，所以使用串行通信进行内核调试时，如果进行频繁的单步跟踪和要传递较大的文件（比如.kdfiles 和.dump 命令），那么会

感觉到速度有些慢。但是大多数时候我们都不会感觉到明显的等待。或者可以利用等待的时间来思考，通常经过深思熟虑有的放矢要比盲目地试来试去高效得多。

串行通信电缆的长度通常为 1~2 米，最长一般不能超过 10 米。

随着 USB 等通信方式的流行，串行通信的应用慢慢变少，以至于今天的很多计算机系统已经不再配备串行口。但是为了支持调试，笔者建议至少在开发版本的系统中应该配备串行口。

18.2.2 1394

1394 又称火线，是一种高性能的串行总线通信标准。其研究工作始于 1986 年，1995 年正式成为 IEEE 标准，全称为 IEEE 1394—1995。1394 的第一个产品化应用是 1995 年的索尼数字摄像机(DV Camcorder)，之后逐步被推广。2000 年和 2002 年 IEEE 分别发布了 1394a 和 1394b，它们对最初的 1394 标准作了更新。1394b 主要是提高了数据传输速度，将原来的最高通信速度 400Mbps 提高到最高为 3.2Gbps。

1394 使用 64 位地址，其中高 10 位代表总线号，随后的 6 位表示节点号，之后的 48 位称为节点内偏移，用来寻址设备节点内的空间。因此，一个 1394 网络最多可以有 1023 个总线，总线号为 0 到 1023；一个总线上最多可以有 63 个节点（Nodes），号码为 0 到 63。其中，总线号 1023 代表本地总线，节点号 63 用作广播。

1394 协议栈（Protocol Stack）定义了以下 3 个层次（layer）：事务层（Transaction Layer）、链路层（Link Layer）和物理层（Physical Layer）。事务层负责从系统总线（通常为 PCI）接收读写请求，然后向链路层发出请求。链路层把请求封装为数据包，然后发给物理层。物理层负责总线初始化和仲裁（arbitration），保证同一时间只有一个节点发送数据。

1394 支持异步数据传输（Asynchronous Data Transport）和同步数据传输（Isochronous Data Transport）两种数据传输方式。所有的同步传输数据包是通过通道号（Channel Number）来标识的。通道号为 0 到 63 之间的数字之一，是设备节点请求同步通信带宽时分配的。

1394 标准电缆的最大长度是 4.5 米，其内部包含三对双绞线，两对用来传递数据，一对用来为外设供电。

Windows 操作系统从 XP 版本开始支持 1394 总线和设备。配备的驱动程序有 1394bus.sys、ohci1394.sys、arp1394.sys、enum1394.sys、nic1394.sys，前两个分别代表总线驱动和端口驱动，是 1394 驱动程序栈的核心。后三个主要用于使用 1394 来建立网络连接。其中 OHCI 是 Open Host Controller Interface 的缩写，是 1394 总线链路层协议的一种实现方式。

WinDBG 工具包共设计了 3 个用于 1394 调试的驱动程序：1394dbg1.sys、1394dbg2.sys 和 1394udbg.sys。其中，最后一个用于通过 1394 来进行远程用户态调试，前两个都用于内核调试，1394dbg1 用于最初的 Windows XP（Build 2600），1394dbg2 用于 XP SP1 及之后的系统。当第一次使用 WinDBG 来通过 1394 调试目标系统时，WinDBG 会自动安装驱动程序（在主机上，而不是被调试的目标机）。在设备管理器中可以看到这些驱动程序（图 18-4）。其中，名为 1394 Windows Debug Driver（Kernel Mode）的设备使用的驱动程序是 1394dbg2.sys。

在目标系统一侧，内核调试引擎使用 KD1394.DLL 来进行通信。这个模块实现了以下基本功能。

- 初始化 1394 控制器。
- 在 1394 的配置空间（config ROM）中公开调试支持，以便让调试器发现所在的调试目标。
- 启用 1394 对系统内存的物理访问。
- 将内核调试用的数据包映射到物理内存，让调试器读取和处理。

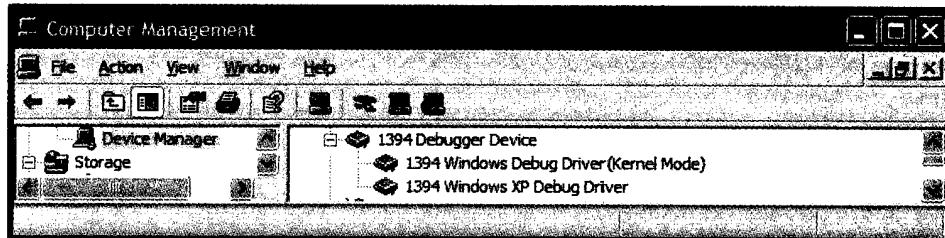


图 18-4 WinDBG 安装的用于通过 1394 进行内核调试的驱动程序（主机端）

KD1394.DLL 是 Windows 的一个系统文件（从 XP 开始），位于 system32 目录下，它的全称是 Kernel Debugger IEEE 1394 HW Extension DLL，即内核调试器的 IEEE1394 硬件扩展 DLL。它的角色与 KDCOM.DLL 是一样的。而且它输出的函数名称与顺序也都是与 KDCOM.DLL 完全一致的。这种一致性使得 KD 可以使用一套代码来使用不同 DLL 所提供的通信服务，而不必关心通信模块内部的细节。

图 18-5 画出了使用 1394 进行内核调试时目标系统和主机间的通信示意图。左侧为目标系统，内核调试引擎（KD）通过它的通信模块 KD1394.DLL 来发送和接收数据。右侧是主机端，调试器将通信要求发送给自己的 1394 驱动程序（1394DBG1.SYS 或 1394DBG2.SYS），再使用系统的 1394 驱动程序栈来执行真正的数据传输工作。因为主机端需要使用系统的 1394 驱动栈，所以当使用 1394 来进行内核调试时，目标系统和主机端的系统版本都必须至少是 Windows XP。

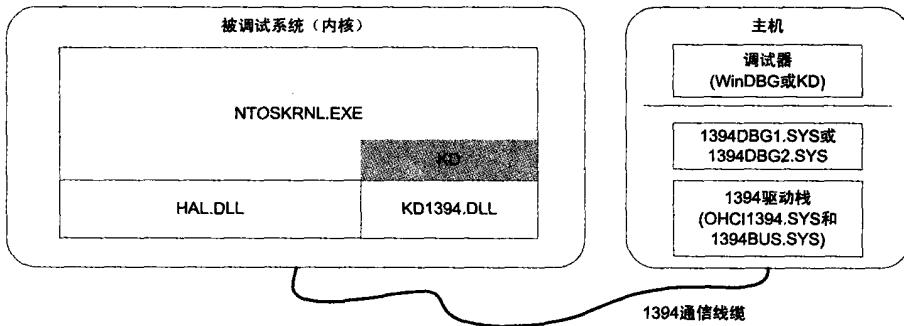


图 18-5 使用 1394 进行内核调试时的通信示意图

与串口方式相比，使用 1394 作为内核调试这种通信方式还比较新。因此还存在稳定性问题，有些时候目标系统和主机的调试器无法建立连接，在关于内核调试的讨论组中，经常可以看到有人因为使用 1394 调试失败而困惑不解，即使经验丰富的调试高手也偶尔会遇到这样的问题。一个老生常谈的救命稻草就是禁止系统中使用 1394 的其他部件。

- 对于目标系统，禁止系统中的 1394 Host Controller，以防止 Windows 系统中常规的 1394 栈驱动程序(1394bus.sys 等)与 KD1394.DLL 争抢资源，发生冲突，Windows XP SP2 和 Server 2003 SP1 加入了支持，会自动完成这一动作(见下文)。
- 对于主机系统，禁止使用 1394 网络适配器(1394 Net Adapter，在网络适配器栏目下)，防止它与 1394 调试驱动程序冲突。

此外，可以尝试的方法还有。

- 检查目标系统的注册表中是否存在如下键值：HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\PCI\Debug，如果这个键值不存在，那么说明 1394 控制器初始化失败。
- 更换 1394 适配卡的插槽位置，或者如果 1394 卡有多个插口，那么更换插口。
- 尽可能让目标系统和主机使用相同的 1394 卡或控制芯片。

但以上方法并不总能解决问题，因此，更好的措施是改进内核调试引擎中关于 1394 调试的设计和实现，使其变得和串行方式一样稳定。事实上，Windows 的设计者们一直在做这样的努力，从 Windows XP 引入 1394 调试后，之后的几乎每次版本升级都包含了对这部分的改进，以下是简要的归纳。

- Windows XP (Build 2600)，支持 1394 调试的第一个版本，使用 1394 的同步数据传输方式。支持调试最多 4 个目标系统。
- Windows XP SP1，使用 1394 的异步数据传输方式，支持最多 62 个目标系统。
- Windows Server 2003，使用更好的方法来初始化 1394 Host Controller。
- Windows XP SP2，自动完成前面提到过的禁止 1394 Host Controller。

因为 Windows XP SP2 及之后的 Windows (Server 2003 SP1, Vista) 发现启用 1394 调试后会自动禁止 1394 Host Controller, 因此, 如果目标系统是这些版本, 那么不应该再手动禁止 1394 Host Controller, 否则可能适得其反。

18.2.3 USB 2.0

内核调试的一种更新的通信方式是 USB 2.0。USB 是 Universal Serial Bus 的缩写, 是一种低成本高性能的串行总线标准。USB 1.0 规范是在 1996 年发布的, 支持低速 (Low Speed) 和全速 (Full Speed) 两种通信速度, 分别为 1.5Mbps 和 12Mbps。2000 年发布的 USB 2.0 规范支持 480Mbps 的高速通信速度。

USB 总线的拓扑结构是一种星形的分层结构。最上端(第一层)是主机控制器(Host Controller), 第二层是根集线器 (Root Hub), 而后是设备或下一层集线器。USB 1.0 的总线控制器通常被称为 UHCI (USB Host Controller Interface)。相应的, USB 2.0 的总线控制器被称为 EHCI (Enhanced Host Controller Interface), 即增强的主机控制器接口。从硬件角度来看, UHCI 和 EHCI 通常都实现于系统芯片组的南桥芯片 (ICH) 中。尽管信号的传输速度有较大差异, 但是 USB 2.0 和 USB 1.0 的连接端口 (Port) 是一样的。支持 USB 2.0 设备的端口完全可以插入 USB 1.0 设备, 反之亦然。一个新插入的设备总是先工作在 USB 1.0 的通信模式, 由 UHCI 与之通信, UHCI 会检测它是否为 USB 2.0 设备, 如果是, 那么会移交给 EHCI 来管理。这意味着同一个 USB 端口同时被映射到 UCHI 和 EHCI (参见图 18-6), 它可能成为 USB 2.0 端口, 也可能成为 USB 1.0 端口, 取决于插入的设备类型。

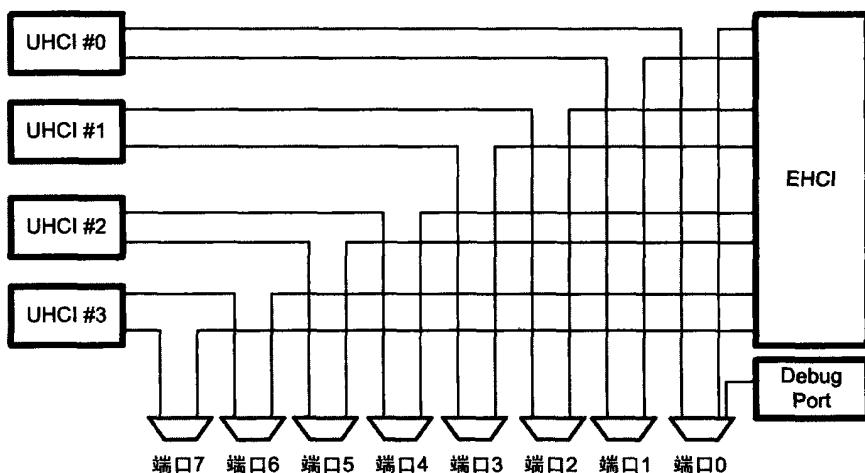


图 18-6 USB 端口与 UHCI 和 EHCI 的关系

因为 USB 总线的节点连接具有方向性，所以 USB 端口分为用于连接设备的上游（upstream）端口和用来连接主机的下游（downstream）端口。而一般个人电脑系统上的 USB 端口都是所谓的上游（upstream）端口。因为两个上游端口是无法简单连接而进行通信的，所以使用 USB 2.0 方式进行内核调试时，首先需要有一根特殊的 USB 2.0 主机到主机（Host to Host）的电缆。

另外，因为 USB 通信协议是比较复杂的，所以需要编写较为复杂的驱动程序才能使其工作，这显然不适合内核调试的目的。为了支持调试，USB 2.0 专门定义了支持调试的调试端口（Debug Port）。并不是所有 USB 端口都是调试端口，通常只有 0 号端口（Port 0）是调试端口。可以使用 DDK 中附带的 USBView 工具来观察哪个端口是 0 号端口。

Windows Vista 增加了 KDUSB.DLL（Kernel Debugger USB 2.0 HW Extension DLL）模块，用于支持使用 USB 2.0 的调试端口进行内核调试。在主机端需要 Windows 2000 或更高版本，因为 NT4 不支持 USB 2.0。对调试器而言，WinDBG 的 6.5.3.8 版本开始支持 USB 2.0 方式，配备的驱动程序名为 usb2dbg.sys，6.6.3.5 版本开始在界面上增加 USB 2.0 连接页（图 18-9）。

图 18-7 画出了使用 USB 2.0 进行内核调试的通信示意图。其中 USB 线缆的目标机一侧一定不能有集线器，主机一侧可以有集线器，并且可以通过这个集线器与几个调试目标系统相连，以实现同时调试多个目标系统。

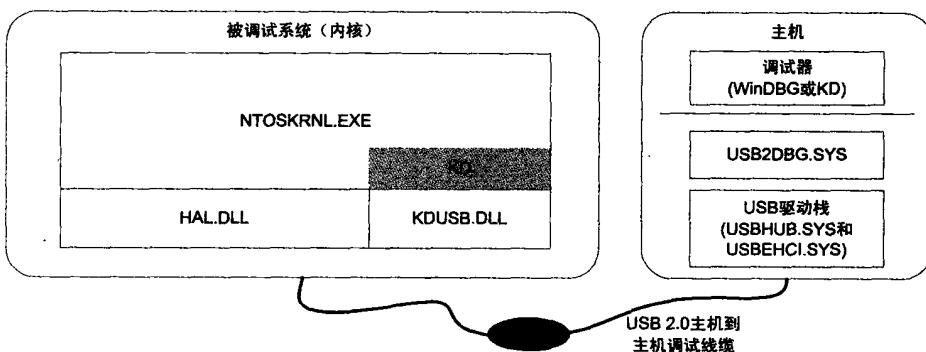


图 18-7 使用 USB 2.0 进行内核调试时目标系统和主机间的通信示意图

因为只有 0 号端口支持内核调试，所以建议系统硬件设计师们尽可能地将这个端口引出到外部，以用于调试，不要用这个端口来连接内部 USB 设备，或者不输出这个端口。举例来说，在笔者使用的笔记本电脑中，ICH 包含了 4 个 UHCI，1 个 EHCI。每个 UHCI 下的根集线器（Root Hub）有 2 个端口，EHCI 下的根集线器有 8 个端口，对应于 UHCI 下的所有端口。在这 8 个端口中，0 号端口是调试端口，但是在笔记本外部可见的 2 个 USB 端口是 3 号和 4 号端口。这意味着要通过 USB 方式调试这台笔记本电脑需要拆卸系统才能找到调试端口。

18.2.4 管道

虚拟机（Virtual Machine）技术可以在一台物理系统中构建出多个虚拟机，每个虚拟机可以安装和运行一个操作系统。这样，一个物理系统中就可以运行多个软件（操作）系统。通常把提供和管理虚拟机环境的软件称为虚拟机管理软件。常用的虚拟机管理软件有 Virtual PC 和 VMWare 等。虚拟机管理软件通常不能直接管理硬件，也需要运行在一个操作系统中。为了便于区别，运行在虚拟机中的操作系统一般被称为寄宿系统（Guest），虚拟机管理软件所运行的操作系统被称为宿主系统（Host）。

虚拟机技术的流行使人们很自然地想到利用它来进行内核调试。因为通过虚拟机技术就可以用一台物理计算机来模拟两台计算机，一台作为运行调试器的主机，另一台作为运行被调试系统的目标机。因为宿主 OS 一旦被中断，那么虚拟机也将无法继续运行，所以调试器应该运行在宿主系统上，被调试内核运行在虚拟机中。

接下来要考虑的是如何建立两个系统间的通信连接。这有两种方法，一种是使用支持回路（Loop back）的真正物理端口，比如串行口，将系统的一个串行端口通过虚拟机管理软件映射给虚拟机，另一个给调试器使用，然后使用 Null-Modem 电缆将这两个端口连接起来。

因为两个系统都是在同一个物理系统中，所以再使用物理连接方法有些没有必要。因此，另一个方法就是使用软件的通信方式来模拟硬件通信端口，比如使用命名管道（Named Pipe）模拟串行端口。其做法是在虚拟机管理软件中使用命名管道虚拟出一个 COM 口，图 18-8 显示了在 Virtual PC 中的设置界面（选中要设置的虚拟机，然后选择 Settings 调出此对话框）。

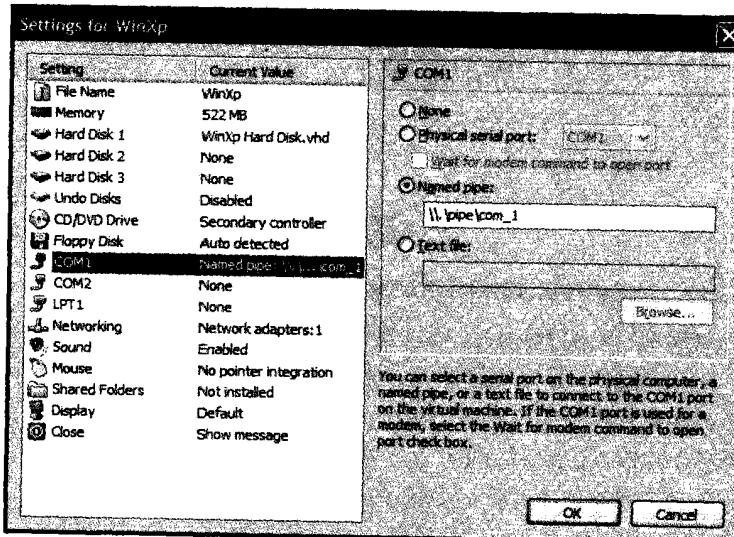


图 18-8 在虚拟机管理软件使用命名管道来模拟串行端口

经过以上设置后，虚拟机中所有对串行端口 1 的读写操作都会被虚拟机管理软件转换为对宿主系统中的命名管道的读写。因此，运行在宿主系统中的调试器便可以通过这个命名管道来与虚拟机中的内核调试引擎进行通信了。可以用以下两种方式之一来设置调试器，一种是通过命令行参数：

```
windbg [-y SymPath] -k com:pipe,port=\\.\pipe\PipeName[, resets=0][,reconnect]
```

其中 *PipeName* 应该替换为虚拟机管理软件中设置的名称，即 *com_1*, *resets=0* 告诉调试器可以无限制地向管道发送复位命令（Reset）。调试器使用复位命令来与调试引擎建立连接（握手），在启动内核调试后，调试器会不停地发送复位命令，直到与目标系统建立起连接。真正的物理端口会自动抛弃过剩的数据，Virtual PC 也会自动丢弃发到虚拟串行端口中的过量数据，因此可以将 *resets* 设置为 0，但对于 VMWare 等不会自动丢弃过量数据的虚拟机系统，不应该指定这个参数。如果指定 *reconnect* 参数，那么，如果读写管道失败，调试器会自动断开连接，然后再重新连接。因为当每次虚拟机重新启动时，虚拟机管理软件会重新构建用来虚拟串行端口的管道，所以，如果不选中指定的 *reconnect* 选项，那么，如果虚拟机重新启动，就需要重新开启调试器，否则在虚拟机下次启动后将无法建立连接。

另一种是通过图形界面，也就是使用图 18-9 所示的内核调试属性对话框。选中 Pipe 复选框，然后在 Port 中指定命名管道的全路径，即 \\.\pipe\com_1，再选中 Reconnect 复选框。

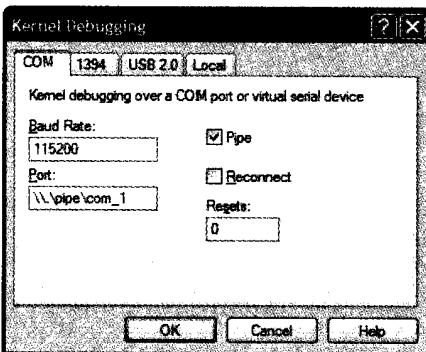


图 18-9 通过图形界面来指定使用管道作为内核调试的连接方式

使用虚拟机进行内核调试的优点是简单方便，但也有如下缺点：一是难以调试硬件相关的驱动程序。二是当对某些涉及底层操作（中断、异常或者 I/O）的函数或指令设置断点时，可能导致虚拟机意外重新启动。三是当将目标系统中断到调试器中时，目前的虚拟机管理软件会占用非常高的 CPU，超过 90%。

18.2.5 选择连接方式

上面我们介绍了内核调试的几种连接方式，可以说每一种方式都有它的优点和局

限。因此，我们需要针对具体的调试环境和调试任务来选择合适的通信连接方式。

选择好连接方式后，还需要设置目标系统和调试器，让它们使用选定的连接方式。我们将在下一节介绍如何设置目标系统。本节先介绍如何设置调试器。有以下3种方法。

- 通过环境变量设置连接方式和参数。
- 通过命令行参数，使用-k开关后面跟一个连接串来指定连接方式和参数。
- 通过图形界面，即图18-9所示的对话框。

以下是使用命令行方式的命令格式：

```
windbg ... -k com:port=ComPort,baud=BaudRate ...
windbg ... -k 1394:channel=1394Channel[,symlink=1394Protocol] ...
windbg ... -k usb2:targetname=String ...
windbg ... -k com:pipe,port=\VMHost\pipe\PipeName[, resets=0][,reconnect]
```

从上至下，各行依次为串行口方式、1394方式、USB2.0方式和管道方式。如果进行本地内核调试，那么使用-kl开关。

18.2.6 解决连接问题

内核调试中一个很令人郁闷的问题就是无法建立通信连接，在解决真正的问题前，人们往往不得不花时间解决连接问题。

这时，首先应该认真检查实际连接方式与软件设置是否一致，包括目标系统的设置和调试器中的设置。

然后检查选定的连接方式是否可以真正在目标系统与主机之间传递数据。如果使用串行端口方式，那么可以使用Windows中的附属工具Hyper Terminal（超级终端）来进行测试。

如果还没有解决问题，第三步就要根据选择的连接方式，利用我们前面讲解的知识来核对或调整物理电缆的连接端口和参数设置。

在进行以上尝试的过程中，可以通过Ctrl+Alt+D（WinDBG）或Ctrl+D（KD）让调试器输出内部的通信信息。

以使用1394方式为例，选择以1394方式启动WinDBG调试器后，它显示如下信息：

```
Using 1394 for debugging
Opened \\.\DBG1394_CHANNEL22
Waiting to reconnect...
```

以上信息表明调试器在等待与内核调试引擎建立连接。但如果连接不成功，它通常没有任何信息提示。此时按Ctrl+Alt+D，WinDBG便会显示通信记录：

```
READ: Data ByteCount error (short read) 0, 10.
READ: Wait for type 7 packet
READ: Data ByteCount error (short read) 0, 10.
READ: Wait for type 7 packet
...
...
```

以上信息表明 WinDBG 在等待类型 7（状态变化）数据包。

因为建立通信连接是内核调试的关键一步，所以本节比较详细地介绍了各种连接方式的原理和特点。但仍无法覆盖每种通信方式的详细原理和细节，感兴趣的读者可以阅读参考文献中列出的资料。本章后面的各节会介绍调试器与调试引擎的通信协议和各种对话过程，这将有助于大家更好地理解通信连接和解决有关问题。

18.3 启用

尽管内核调试引擎已经包含在每个 Windows 系统中，但出于安全和性能的考虑，它默认处于禁止状态。因此在进行内核调试前需要先启用内核调试引擎。这需要修改系统的启动配置文件，对于 Vista 之前的 Windows，需要修改 BOOT.INI 文件，对于 Vista，需要修改启动配置数据（Boot Configuration Data）。

18.3.1 BOOT.INI

BOOT.INI 是传统 INI 格式的文本文件，位于 Windows 系统所在盘的根目录下，通常为 C:\。因为 BOOT.INI 默认具有系统、隐藏和只读属性，所以在编辑前应该去掉这些属性，编辑后再恢复这些属性。如果是在 DOS 命令行下编辑 BOOT.INI，那么执行的典型步骤为。

1. 切换到 Windows 启动盘的根目里，如 cd \。
2. 执行 attrib -s -h -r Boot.ini 去除保护属性。
3. 使用编辑器进行编辑，如 edit boot.ini。
4. 执行 attrib +s +h +r Boot.ini，恢复文件属性。

以下是一个典型的 BOOT.INI 文件的内容：

```
[boot loader]
timeout=30
default=multi(0)disk(0)rdisk(0)partition(1)\WINDOWS
[operating systems]
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS="Microsoft Windows XP
Professional" /noexecute=optin /fastdetect
```

[operating systems] 节下面的每一行对应一个启动入口（boot entry），目前只有一行（由于行宽所限显示为两行）。等号左侧是使用 ARC（Advanced RISC Computing）规范描述的 Windows 系统目录的完整路径。等号右侧是启动选项，其中 /noexecute 用来配置 DEP（Data Execution Prevention）功能，即当执行数据属性的内存页时是否产生异常，optin 表示只对操作系统部件（内核和驱动程序）启用 DEP。/fastdetect 用来告诉 NTDETECT 程序（启动前 NTLDL 用来检测硬件的简单程序）不必枚举串行和并行端口上的设备。因为 NT4 需要 NTDETECT 来枚举串行和并行设备，所以在 NT4

系统的启动配置中，不应该有这个开关，这也是这个开关存在的原因。

要启用内核调试，通常是将已有的启动入口复制一份，粘贴到下面，然后在新添加的启动入口中增加调试选项。例如，以下是使用串行口 1 作为内核调试通信端口的典型写法：

```
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS="Debug with Serial Port"
/fastdetect /debug /debugport=COM1 /baudrate=115200
```

其中双引号内的字符串会出现在启动菜单中，/debugport，用来指定通信端口，COM1 代表串行口 1，/baudrate 用来指定串行通信的通信速率（bps）。值得说明的是，主机与被调试机使用的 COM 口号码可以不同，只要各自与串行电缆所插入的实际端口一致即可。

如果使用 1394 通信，那么可以配置为：

```
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS="Debug with 1394" /fastdetect
/debug /debugport=1394 /channel=22
```

其中/channel 用来指定 1394 通信的通道号，需要与主机上的一致。

以上配置中的/debug 可以省略，因为有了/debugport，就起到了启用内核调试引擎的作用。如果只有/debug 而没有指定/debugport，那么 Windows 会使用默认的通信端口和通信参数，对于 x86 系统通常为 COM2（可以枚举到的号码最大的 COM 口），对于 RISC 系统为 COM1，通信速率为 19200。

除了以上启动选项，与内核调试有关的选项还有。

- /NODEBUG：禁止内核调试，它比/DEBUG 和/DEBUGPORT 的优先级都高。
- /CRASHDEBUG：加载内核调试引擎，但是只有当系统发生崩溃时才会被激活，否则一直处于不活动状态。这一选项也导致内核调试引擎不会占用通信端口，使其可以被系统所使用。如果启用了内核调试引擎但没有指定这个选项，那么系统中是看不到/debugport 中所指定的端口的，因为内核调试引擎在使用它。

对于 Windows XP 和 Server 2003，Windows 系统的 system32 目录自带了一个名为 bootcfg 的命令行工具，用来帮助修改启动文件。对于这两个系统也可以通过计算机属性对话框来启动 notepad 编辑 BOOT.INI 文件，其操作步骤为：My Computer → Properties → Advanced → Startup and Recovery → Settings → System Startup → Edit。

最后说明一点，从 Windows XP 开始才支持使用 1394 进行内核调试，Windows 2000 和它以前的 Windows 只支持串行口方式。

18.3.2 BCD

Windows Vista 不再使用 BOOT.INI 文件，改为使用 Boot Configuration Data（BCD）。改动的一个原因是 BOOT.INI 文件容易被恶意软件所修改。

首先使用命令行工具 `bcdedit` 来编辑 BCD。它需要启动一个管理员权限（Run As Administrator）的命令行窗口。然后执行如下命令将当前的启动入口复制一份：

```
bcdedit /copy {current} /d "Vista Debug with Serial"
```

其中双引号中的字符串为新启动入口的名称。如果执行成功，会显示类似如下的信息：

```
The entry was successfully copied to {49916baf-0e08-11db-9af4-000bdbd316a0}.
```

其中大括号中的内容是新启动入口的 GUID，用来唯一标识这个启动入口。

接下来执行如下命令对这个启动入口启用内核调试：

```
bcdedit /debug {49916baf-0e08-11db-9af4-000bdbd316a0} on
```

BCD 中总有一套全局的调试设置，使用 `bcdedit/dbgsettings` 可以观察和修改这套设置：

```
bcdedit/dbgsettings[[connect type][connect para]/start startpolicy/noemux]
```

其中 `connect type` 和 `connect para` 用来定义内核调试的连接方式和参数，可以为以下 3 种之一。

- **serial:** 串行通信，使用 `DEBUGPORT:port` 来指定串行口号，`BAUDRATE:baud` 来指定传输速度。
- **1394:** 1394（火线）通信，使用 `CHANNEL:channel` 来指定通道号。
- **USB:** USB2.0 通信，使用 `TARGETNAME:name` 来指定调试目标的名称。

例如，以下命令将内核调试的全局设置定义为串行口 1，波特率为 115200：

```
bcdedit /dbgsettings serial DEBUGPORT:1 BAUDRATE:115200
```

以下命令使用 USB 2.0，目标名称为 RTM：

```
bcdedit /dbgsettings usb targetname:RTM
```

使用 `/start` 开关可以指定内核调试引擎的启动策略（`startpolicy`），可以为以下选项之一。

- **ACTIVE:** 使内核调试引擎（始终）处于活动状态，这是默认的选项。
- **AUTOENABLE:** 当发生异常或蓝屏崩溃时，自动启用内核调试引擎，在这类事件发生前内核调试引擎处于非活动状态。
- **DISABLE:** 通过 `kdbgctrl` 命令来启用。

开关 `/noumex` 告诉内核调试引擎忽略用户态异常。如果不指定这个开关，当某个进程的用户态代码触发了断点或单步异常，而且这个进程又不在调试状态时，那么内核调试引擎会中断到内核调试器。

如果希望为某个启动项设置单独的调试选项，那么可以使用 `bcdedit/set` 命令，例如：

```
bcdedit /set {18b123cd-2bf6-11db-bfae-00e018e2b8db} debugtype serial
bcdedit /set {18b123cd-2bf6-11db-bfae-00e018e2b8db} debugport 1
bcdedit /set {18b123cd-2bf6-11db-bfae-00e018e2b8db} baudrate 115200
```

使用 WinDBG 工具包中的 KDbgCtrl 工具 (KDbgCtrl.exe) 可以观察或调整以上部分参数，比如调试引擎的启动策略和对用户态异常的中断策略。使用时可以参考这个工具的帮助信息 (?)。

18.3.3 高级启动选项

如果被调试系统已无法正常启动，不能登录并修改 BOOT.INI 或使用 BCDEdit，那么可以通过 Windows 高级启动选项菜单 (Windows Advanced Options Menu) 中的 Debugging Mode 选项来启用内核调试。

调出高级选项的方法是在当系统固件 (BIOS) 将控制权交给 Windows 的加载器 (NTLDR 或 BootMgr/WinLoad) 时，按 F8 热键。

通过这种方法启用内核调试引擎时，内核调试引擎会从默认的调试设置中读取通信连接方式。对于 Windows Vista 之前的版本，系统会使用可以枚举到的最大序号 COM 口和 19200 波特率。

18.4 初始化

本节我们将介绍内核调试引擎初始化的过程。因为这一过程是穿插在 Windows 系统的启动过程中的，所以我们先来简要介绍 Windows 的启动过程。

18.4.1 Windows 启动过程概述

计算机开机后，先执行的是系统的固件 (firmware)，即 BIOS (Basic Input/Output System，基本输入输出系统) 或 EFI (Entexed Firmware Interface)。BIOS 或 EFI 在完成基本的硬件检测和平台初始化工作后，将控制权移交给磁盘上的引导程序。磁盘引导程序再执行操作系统的加载程序 (OS Loader)，即 NTLDR (Vista 之前) 或 WinLoad.exe (Vista)。

系统加载程序首先会对 CPU 做必要的初始化工作，包括从 16 位实模式切换到 32 位保护模式，启用分页机制等，然后通过启动配置信息 (Boot.INI 或 BCD) 得到 Windows 系统的系统目录并加载系统的内核文件，即 NTOSKRNL.EXE。当加载这个文件时，会检查它的 PE 文件头导入节中所依赖的其他文件，并加载这些依赖文件，其中包括用于内核调试通信的硬件扩展 DLL (KDCOM.DLL、KD1394.DLL 或 KDUSB.DLL)。加载程序会根据启动设置加载这些 DLL 中的一个，并将其模块名统一称为 KDCOM。

而后系统加载程序会读取注册表的 System Hive，加载其中定义的启动 (boot) 类型 (SERVICE_BOOT_START (0)) 的驱动程序，包括磁盘驱动程序。

在完成以上工作后，系统加载程序会从内核文件的 PE 文件头找到它的入口函数，即 KiSystemStartup 函数，然后调用这个函数。调用时将启动选项以一个名为 LOADER_PARAMETER_BLOCK 的数据结构传递给 KiSystemStartup 函数。于是，NT 内核文件得到控制权并开始执行。

可以把接下来的启动过程分为图 18-10 所示的 3 个部分。左侧是发生在初始启动进程中的过程，这个初始的进程就是启动后的 Idle 进程。中间是发生在系统进程（System）中的所谓的执行体阶段 1 初始化过程。右侧是发生在会话管理器进程（SMSS）的过程。

首先我们来看 KiSystemStartup 函数的执行过程，它所做的主要工作有。

第一，调用 HalInitializeProcessor() 初始化 CPU。

第二，调用 KdInitSystem 初始化内核调试引擎，我们稍后将详细介绍这个函数。

第三，调用 KiInitializeKernel 开始内核初始化，这个函数会调用 KiInitSystem 来初始化系统的全局数据结构，调用 KeInitializeProcess 创建并初始化 Idle 进程，调用 KeInitializeThread 初始化 Idle 线程，调用 ExpInitializeExecutive() 进行所谓的执行体阶段 0 初始化。ExpInitializeExecutive 会依次调用执行体各个机构的阶段 0 初始化函数，包括调用 MmInitSystem 构建页表和内存管理器的基本数据结构，调用 ObInitSystem 建立名称空间，调用 SeInitSystem 初始化 token 对象，调用 PsInitSystem 对进程管理器做阶段 0 初始化（稍后详细说明），调用 PpInitSystem 让即插即用管理器初始化设备链表。

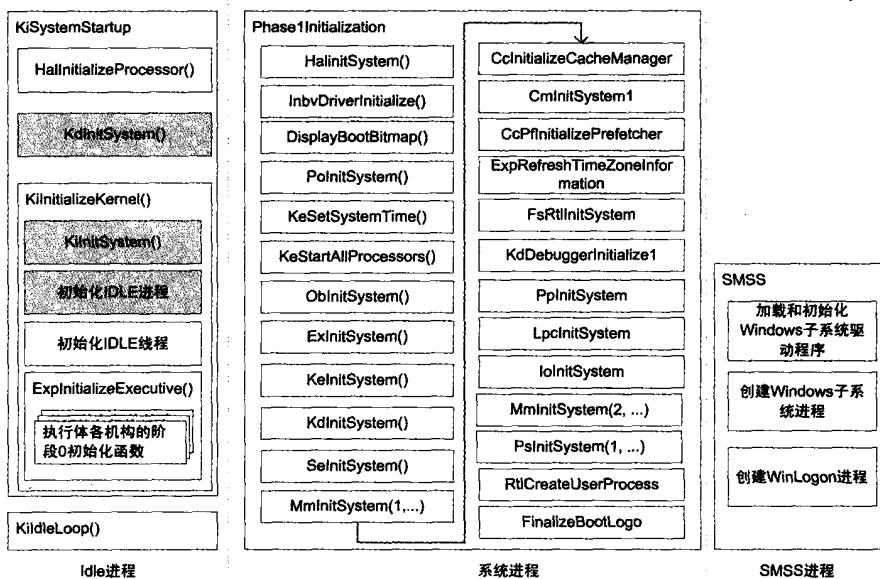


图 18-10 Windows 启动过程概览

在 KiInitializeKernel 函数返回后, KiSystemStartup 函数将当前 CPU 的中断请求级别 (IRQL) 降低到 DISPATCH_LEVEL, 然后跳转到 KiIdleLoop(), 退化为 Idle 进程中的第一个 Idle 线程。

对于多 CPU 的系统, 每个 CPU 都会执行 KiInitializeKernel 函数, 但只有第一个 CPU 才执行其中的所有初始化工作, 包括全局性的初始化, 其他 CPU 只执行 CPU 相关的部分。比如只有 0 号 CPU 才调用和执行 KiInitSystem, 初始化 Idle 进程的工作也只有 0 号 CPU 执行, 因为只需要一个 Idle 进程。但是, 由于每个 CPU 都需要一个 Idle 线程, 所以每个 CPU 都会执行初始化 Idle 线程的代码。KiInitializeKernel 函数使用参数来了解当前的 CPU 号。全局变量 KeNumberProcessors 标志着系统中的 CPU 个数, 其初始值为 0, 因此, 当 0 号 CPU 执行 KiSystemStartup 函数时, KeNumberProcessors 的值刚好是当前的 CPU 号。当第二个 CPU 开始运行时, 这个全局变量会被递增 1, 因此 KiSystemStartup 函数仍然可以从这个全局变量了解到 CPU 号, 依此类推, 直到所有 CPU 都开始运行。ExpInitializeExecutive 函数的第一个参数也是 CPU 号, 在这个函数中有很多代码是根据 CPU 号来决定是否执行的。

下面我们仔细看看进程管理器的阶段 0 初始化, 它所做的主要动作有。

- 定义进程和线程对象类型。
- 建立记录系统中所有进程的链表结构, 并使用 PsActiveProcessHead 全局变量指向这个链表。此后 WinDBG 的!process 命令才能工作。
- 为初始的进程创建一个进程对象 (PsIdleProcess), 并命名为 Idle。
- 创建系统进程和线程, 并将 Phase1Initialization 函数作为线程的起始地址。

注意上面的最后一步, 因为它衔接着系统启动的下一个阶段, 即执行体的阶段 1 初始化。但是这里并没有直接调用阶段 1 的初始化函数, 而是将它作为新创建系统线程的入口函数。此时由于当前的 IRQL 很高, 所以这个线程还得不到执行, 只有当 KiInitializeKernel 返回, KiSystemStartup 将 IRQL 降低后, 内核下次调度线程时, 才会开始执行这个线程。

阶段 1 初始化占据了系统启动的大多数时间, 其主要任务就是调用执行体各机构的阶段 1 初始化函数。有些执行体部件使用同一个函数作为阶段 0 和阶段 1 初始化函数, 用参数来区分。图 18-10 中列出了这一阶段所调用的主要函数, 下面简要说明其中几个。

- 调用 KeStartAllProcessors() 初始化所有 CPU。这个函数会先构建并初始化好一个处理器状态结构, 然后调用硬件抽象层的 HalStartNextProcessor 函数将这个结构赋给一个新的 CPU。新的 CPU 仍然从 KiSystemStartup 开始执行。
- 再次调用 KdInitSystem 函数, 并且调用 KdDebuggerInitialize1 来初始化内核调试通信扩展 DLL (KDCOM.DLL 等)。

- 在这一阶段结束前，它会创建第一个使用映像文件创建的进程，即会话管理器进程（SMSS.EXE）。

会话管理器进程会初始化 Windows 子系统，创建 Windows 子系统进程和登录进程（WinLogon.EXE），后者会创建 LSASS（Local Security Authority Subsystem Service）进程和系统服务进程（Services.EXE）并显示登录画面，至此启动过程基本完成。

18.4.2 第一次调用 KdInitSystem

从图 18-10 可以看到，系统在启动过程中会两次调用内核调试引擎的初始化函数，KdInitSystem。第一次是在系统内核开始执行后由入口函数 KiSystemStartup 调用。

调用时，KdInitSystem 会执行以下动作。

- 初始化调试器数据链表，使用变量 KdpDebuggerDataListHead 指向这个链表。
- 初始化 KdDebuggerDataBlock 数据结构，该结构包含了内核基址、模块链表指针、调试器数据链表指针等重要数据，调试器需要读取这些信息以了解目标系统。
- 根据参数指针指向的 LOADER_PARAMETER_BLOCK 结构寻找调试有关的选项，然后保存到变量中（见下文）。
- 对于 XP 之后的系统，调用 KdDebuggerInitialize0 来对通信扩展模块进行阶段 0 初始化。对于 XP 之前的版本，调用 KdPortInitialize 来初始化 COM 端口。如果使用的是串行通信方式，那么 KDCOM 中的 KdDebuggerInitialize0 函数会调用模块内的 KdCompInitialize 函数来初始化串行口。不论是 KdPortInitialize 还是 KdCompInitialize 函数，成功初始化 COM 口后，都会设置 HAL 模块中所定义的 KdComPortInUse 全局变量，记录下已被内核调试使用的 COM 端口，此后的串行口驱动程序（serial.sys）会检查这个变量，并跳过用于调试用途的串行端口，因此，在目标系统的设备管理器中我们看不到用于内核调试的串行端口。

此外，KdInitSystem 会初始化以下全局变量：

- KdPitchDebugger：布尔类型，用来标识是否显式抑制内核调试。当启动选项中包含/NODEBUG 选项时，这个变量会被设置为真。
- KdDebuggerEnabled：布尔类型，用来标识内核调试是否被启用。当启动选项中包含/DEBUG 或/DEBUGPORT 而且不包含/NODEBUG 时，这个变量会被设置为真。
- KiDebugRoutine：函数指针类型，用来记录内核调试引擎的异常处理回调函数，当内核调试引擎活动时，它指向 KdpTrap 函数，否则指向 KdpStub 函数。
- KdpBreakpointTable：结构数组类型，用来记录代码断点，每个元素为一个 BREAKPOINT_ENTRY 结构，用来描述一个断点，包括断点地址。

对于 Windows XP 之前的版本，还会初始化如下变量。

- KdpNextPacketIdToSend：整数类型，用来标识下一个要发送的数据包 ID，KdInitSystem 将这个变量初始化为 0x80800000 | 0x800，即 INITIAL_PACKET_ID（初始包）或 SYNC_PACKET_ID（同步包）。
- KdpPacketIdExpected：整数类型，用来标识期待收到的下一个数据包 ID，初始化为 0x80800000，即 INITIAL_PACKET_ID。

对于 XP 和之后的系统，当使用串行通信方式时，在 KDCOM.DLL 中定义了两个同样用途的变量 KdCompNextPacketIdToSend 和 KdCompPacketIdExpected。

对于 Windows Vista，会初始化如下变量。

- KdAutoEnableOnEvent：布尔类型，如果调试设置中的启动策略为 AUTOENABLE，则设为真。
- KdIgnoreUmExceptions：布尔类型，代表了处理用户态异常的方式，如果调试设置中的启动策略包含/noumex，则设为真，含义是忽略用户态异常。

关于 KdInitSystem 第一次被调用，还有如下两点值得说明。第一，只有当 0 号 CPU 执行 KiSystemStartup 函数时才会调用 KdInitSystem，所以它不会被 KiSystemStartup 多次调用。第二，不管系统是否启用内核调试，调用都会发生。

18.4.3 第二次调用 KdInitSystem

在执行体的阶段 1 初始化过程中，系统会第二次调用 KdInitSystem。让调试子系统做进一步的初始化工作。以下栈回溯显示这次调用的过程：

```
kd> kn
# ChildEBP RetAddr
00 f8958840 8068b0ee nt!KdInitSystem           // 内核调试引擎初始化
01 f8958dac 8057c73a nt!Phase1Initialization+0x410 // 执行体的阶段 1 初始化
02 f8958ddc 805124c1 nt!PspSystemThreadStartup+0x34 // 系统线程启动函数
03 00000000 00000000 nt!KiThreadStartup+0x16    // 内核态的线程启动函数
```

从 Windows XP 开始，KdInitSystem 函数的第一个参数为阶段号，0 代表阶段 0，即第一次调用，1 代表阶段 1，即第二次调用，第二个参数为指向 LOADER_PARAMETER_BLOCK 结构的指针。在 XP 之前，KdInitSystem 的第一个参数是 LOADER_PARAMETER_BLOCK 结构指针。

在目前的实现中，KdInitSystem 的阶段 1 初始化（第二次被调用）只是简单地调用 KeQueryPerformanceCounter 来对变量 KdPerformanceCounterRate（性能计数器的频率）初始化，然后返回。

18.4.4 通信扩展模块的阶段 1 初始化

在阶段 1 初始化中，系统会调用通信扩展模块的 KdDebuggerInitialize1 函数

来让通信扩展模块得到阶段 1 初始化的机会。

```
kd> kn
# ChildEBP RetAddr
00 f8958844 8068b313 kdcom!KdDebuggerInitialize1      // 内核调试通信扩展模块
01 f8958dac 8057c73a nt!Phase1Initialization+0x69a    // 执行体的阶段 1 初始化
02 f8958ddc 805124c1 nt!PspSystemThreadStartup+0x34   // 系统线程的启动函数
03 00000000 00000000 nt!KiThreadStartup+0x16         // 内核态的线程启动函数
```

目前的 KDCOM 实现会调用 KdCompInitialize1，但是只执行很少的操作就返回了。

18.5 内核调试协议

当使用内核调试引擎进行调试时，位于被调试系统中的内核调试引擎与位于主机中的调试器相互协作共同完成各种调试任务。为了实现这种合作，双方需要按照一定的规则进行通信和交互，这些规则被称为内核调试协议（NT Kernel Debugging Protocol）。

尽管没有详细的文档，但是从 NT3.51 到 Windows 2000 的 DDK 中都包含了一个名为 Windbgkd.h 的头文件，其中包含了内核调试协议所使用的所有数据结构、常量和简单说明。通过这个头文件和 WinDBG 所输出的通信记录（按 Ctrl+Alt+D，见 18.3 节），可以归纳出内核调试协议的基本规则和原理。

18.5.1 数据包

内核调试引擎和调试器是以数据包的形式来通信的，根据包中的内容，可以把数据包分成如下 3 大类。

- 中断包（Breakin Packet）：供调试器通知内核调试引擎中断到调试器。
- 信息包（Information Packet）：用来传递调试信息或调试命令。
- 控制包（Control Packet）：用来建立通信连接或控制通信流程，例如确认收到数据，要求重新发送数据，或者请求重新建立连接。

中断包的格式最简单，只有一种，而且内容固定为 1 到 4 个字节的 0x62，即字符 b。

信息包用来在内核调试引擎和调试器之间传递状态信息或调试命令，其长度是不固定的，但都是以一个 KD_PACKET 结构开始，然后跟随不定长度的数据，最后以字符 0xAA 结束。

```
typedef struct _KD_PACKET {
    ULONG PacketLeader;           // 导引
    USHORT PacketType;            // 包类型
    USHORT ByteCount;             // 跟随在本结构后数据的长度
    ULONG PacketId;               // 包的 ID 号
    ULONG Checksum;               // 数据的校验和
} KD_PACKET, *PKD_PACKET;
```

我们把上面的结构称为信息包的头结构，其中，`PacketLeader` 是整个包的导引，长度为 4 个字节，对所有信息包，`PacketLeader` 都等于 0x30303030。`PacketType` 用来标识信息的子类型，可以为如下值之一。

- `PACKET_TYPE_KD_STATE_CHANGE` (1): 供内核调试引擎向调试器报告状态变化，包头后是一个 `DBGKD_WAIT_STATE_CHANGE` 结构。调试器收到后应该发送确认控制包。
- `PACKET_TYPE_KD_STATE_MANIPULATE` (2): 供调试器调用内核调试引擎的各种调试服务（称为 KdAPI）。在内核调试引擎收到后，先发送确认控制包，然后再通过这个类型的信息包回复给调试器，调试器收到后再发送确认信息。
- `PACKET_TYPE_DEBUG_IO` (3): 供内核调试引擎向调试器输出字符串或通过调试器征询（Prompt）用户输入。
- `PACKET_TYPE_KD_STATE_CHANGE64` (7): 与 `PACKET_TYPE_KD_STATE_CHANGE` 类似，但是头结构后跟随的是一个 `DBGKD_WAIT_STATE_CHANGE64` 结构。这个子类型是为了支持 64 位而增加的。
- `PACKET_TYPE_KD_POLL_BREAKIN` (8): 在 Windows XP 以前，内核内次更新系统时间时，直接调用通信函数（`KdPortPollByte`）查询通信队列中是否有中断字节（0x62）。XP 将通信函数独立到扩展模块后，改为调用通信函数的 `KdReceivePacket` 来轮询（Poll）是否有中断包（Breakin），为了与接收正常包相区别，引入了这个包类型（8）作为参数，其他 4 个参数都为 0。
- `PACKET_TYPE_KD_TRACE_IO` (9): 用于将 ETW 信息输出到调试器。
- `PACKET_TYPE_KD_CONTROL_REQUEST` (10): 用法不详。
- `PACKET_TYPE_KD_FILE_IO` (11): 用来从调试器读取和更新内核文件（驱动程序），WinDBG 的 `.kdfiles` 命令就是依靠这种机制实现的。

以上类型中，8 到 11 是 Windows XP 引入的。

`ByteCount` 为 `KD_PACKET` 结构后所跟随的数据长度（字节数）。`PacketId` 用来标识数据包，对于内核调试引擎所发送的信息包，其编排方式与通信连接方式有关，使用 1394 连接时，它从 0 开始，不断递增。使用串行口连接时，ID 为 8080XXXX，一轮对话后会复位。对于调试器所发送的包，其 ID 与调试器的实现有关。`Checksum` 是数据包中所有字节的代数和，供接收方来验证数据。

控制包就是一个 `KD_PACKET` 结构，因此其长度固定为 16 字节。包中的 `PacketLeader` 固定等于 0x69696969，因为没有额外的数据，所以 `ByteCount` 总是等于 0，`PacketType` 用来标识想发送的控制命令，可以为以下 3 个值之一。

- `PACKET_TYPE_KD_ACKNOWLEDGE` (4): 确认收到对方的信息包，`PacketId` 为收到包的包 ID。

- PACKET_TYPE_KD resend (5): 要求对方重新发送正在发送的信息包或尚未得到确认的上一个信息包。PacketId 似乎总为 0x12062F。
- PACKET_TYPE_KD_RESET (6): 请求重新建立连接，用于同步。当调试器在等待与内核调试引擎建立连接时，通常会反复发送这样的控制包。在恢复调试目标运行后，WinDBG 也会定期地发送这样的控制包。

以上简要介绍了数据包的结构和 3 种数据包的概况，下面我们将以典型的操作为例来说明这些数据包的应用。

18.5.2 报告状态变化

内核调试引擎使用 PacketType 等于 PACKET_TYPE_KD_STATE_CHANGE (1) 或 PACKET_TYPE_KD_STATE_CHANGE64 (7) 的信息包向调试器报告状态变化。在 KD_PACKET 结构后跟随的是一个 DBGKD_WAIT_STATE_CHANGE32 结构或 DBGKD_WAIT_STATE_CHANGE64。因为内核调试引擎和 WinDBG 调试器关于支持 64 位的一个重要设计思想，就是用同一个调试器既可以调试 32 位系统，又可以调试 64 位目标系统，所以 Windows 2000 开始的内核调试引擎（不论是 32 位还是 64 位）都是使用 64 位的数据结构的。对调试器来说，WinDBG 既有 32 位版本，又有 64 位版本，32 位版本的会把 64 位格式的数据结构转换为 32 位格式。考虑到 32 位的数据结构与 64 位的数据结构之间主要是字段长度的变化，因此我们仍以 32 位为例来说明。

```
typedef struct _DBGKD_WAIT_STATE_CHANGE32 {
    ULONG NewState; // 状态代号
    USHORT ProcessorLevel; // 处理器级别
    USHORT Processor; // CPU 号
    ULONG NumberProcessors; // 活动的处理器个数
    ULONG Thread; // ETHREAD 结构地址
    ULONG ProgramCounter; // 程序指针
    union { // 枚举类型，见下文
        DBGKM_EXCEPTION32 Exception; // 异常信息
        DBGKD_LOAD_SYMBOLS32 LoadSymbols; // 映像文件信息
    } u;
    DBGKD_CONTROL_REPORT ControlReport; // 附属报告
    CONTEXT Context; // 上下文状态
} DBGKD_WAIT_STATE_CHANGE32, *PDBGKD_WAIT_STATE_CHANGE32;
```

其中，NewState 等于 DbgKdExceptionStateChange (0x00003030L) 或 DbgKdLoadSymbolsStateChange (0x00003031L)，分别代表异常类状态变化和加载符号类状态变化。ProcessorLevel 为与 CPU 架构相关的处理器级别（并非是 IRQL），通常为 0。Processor 为与此状态相关的 CPU 号。NumberProcessors 为当时系统中的 CPU 个数，值得注意的是，即使对于多 CPU 的系统，此数字也可能为 1，比如在启动过程中，其他 CPU 是在执行体的阶段 1 初始化时才被唤醒的。Thread 为发生状态变化线程的 ETHREAD 结构地址。ProgramCounter 为触发状态变化的程序指令地址。联合结构 u 的内容根据

NewState 决定, 如果 NewState 等于 DbgKdExceptionStateChange, 那么 u 联合中便是一个 DBGKM_EXCEPTION32 结构, 否则便是一个 DBGKD_LOAD_SYMBOLS32 结构。接下来的 ControlReport 和 Context 都是与 CPU 架构相关的, 对于 x86 架构, 它们的定义如下:

```
#define DBGKD_MAXSTREAM 16
typedef struct _DBGKD_CONTROL_REPORT {
    ULONG Dr6;                                // 调试状态寄存器
    ULONG Dr7;                                // 调试控制寄存器
    USHORT InstructionCount;                   // 附带指令数
    USHORT ReportFlags;                        // 标志
    UCHAR InstructionStream[DBGKD_MAXSTREAM]; // 指令
    USHORT SegCs;                             // 代码段寄存器 CS 的值
    USHORT SegDs;                             // 数据段寄存器 DS 的值
    USHORT SegEs;                             // 段寄存器 ES 的值
    USHORT SegFs;                             // 段寄存器 FS 的值
    ULONG EFlags;                            // 标志寄存器的值
} DBGKD_CONTROL_REPORT, *PDBGKD_CONTROL_REPORT;
```

以上两个结构的设计思想是把调试器可能需要的常用数据也顺带发送过去, 以减少数据通信的次数。举例来说, 调试器可以通过 DBGKD_CONTROL_REPORT 结构中的指令数据 (InstructionStream) 来了解触发状态变化的指令, 通过 Dr6、Dr7 和 EFlags (标志寄存器) 来判断硬件断点, 通过 CONTEXT 结构来显示寄存器值。

在发送状态变化信息包前, 内核调试引擎 (KdpTrap) 已经调用 KdEnterDebugger 函数将内核的其他部分冻结。发送状态变化信息包后, 内核调试引擎 (kdpSendWaitContinue 函数) 一直等待来自调试器的回复。调试器收到状态变化数据包后, 首先应该发送确认控制包, 然后根据调试器的调试事件设置选项 (Event Filters) 来判断是应该中断给用户开始交互式调试, 还是立刻发送恢复执行 (Continue) 命令。在发送恢复执行命令前, 调试器可以发送各种分析和诊断命令, 来访问目标系统的数据和对象。

18.5.3 访问目标系统

调试器通过发送 PacketType 等于 PACKET_TYPE_KD_STATE_MANIPULATE (2) 的信息包, 请求内核调试引擎的服务以访问目标系统。在 KD_PACKET 结构后跟随的是一个 DBGKD_MANIPULATE_STATE32 结构或 DBGKD_MANIPULATE_STATE64 结构。当调试引擎回复时使用的也是这样的结构。

```
typedef struct _DBGKD_MANIPULATE_STATE32 {
    ULONG ApiNumber;                          // API 代号, 见表 18-2
    USHORT ProcessorLevel;                    // 处理器级别
    USHORT Processor;                         // 处理器编号
    NTSTATUS ReturnStatus;                   // 供内核调试引擎使用, 来指示访问结构, 成功还是失败
    union {                                  // 枚举结构, 与 API 代号相关
        DBGKD_READ_MEMORY32 ReadMemory;       // 读内存, 32 位
        DBGKD_WRITE_MEMORY32 WriteMemory;     // 写内存, 32 位
        DBGKD_READ_MEMORY64 ReadMemory64;     // 读内存, 64 位
    };
}
```

```

DBGKD_WRITE_MEMORY64 WriteMemory64;           // 写内存, 64 位
DBGKD_GET_CONTEXT GetContext;                 // 读取上下文
DBGKD_SET_CONTEXT SetContext;                 // 设置上下文
DBGKD_WRITE_BREAKPOINT32 WriteBreakPoint;     // 设置断点
DBGKD_RESTORE_BREAKPOINT RestoreBreakPoint;   // 恢复断点
DBGKD_CONTINUE Continue;                      // 恢复执行
DBGKD_CONTINUE2 Continue2;                    // 恢复执行
DBGKD_READ_WRITE_IO32 ReadWriteIo;            // IO 读写, 32 位
DBGKD_READ_WRITE_IO_EXTENDED32 ReadWriteIoExtended; // IO 读写
DBGKD_QUERY_SPECIAL_CALLS QuerySpecialCalls; // 查询特殊调用
DBGKD_SET_SPECIAL_CALL32 SetSpecialCall;      // 设置特殊调用
DBGKD_SET_INTERNAL_BREAKPOINT32 SetInternalBreakpoint; // 设置内部断点
DBGKD_GET_INTERNAL_BREAKPOINT32 GetInternalBreakpoint; // 读取内部断点
DBGKD_GET_VERSION32 GetVersion32;             // 读取版本结构
DBGKD_BREAKPOINTEX BreakPointEx;              // 增强的断点操作
DBGKD_READ_WRITE_MSR ReadWriteMsr;            // 读写 MSR 寄存器
DBGKD_SEARCH_MEMORY SearchMemory;             // 搜索内存
} u;
} DBGKD_MANIPULATE_STATE32, *PDBGKD_MANIPULATE_STATE32;

```

从服务和被服务的角度来看，调试器通过内核调试引擎来访问目标系统，是服务的使用者，内核调试引擎是服务的提供者。如果使用函数调用来作比喻，那么内核调试引擎提供了一系列可供调用的 API，调试器便是这些 API 的调用者。因为这个因素，内核调试引擎所提供的服务有时也被称为内核调试 API，简称 KdAPI。只不过，内核调试 API 是通过编号来调用的。在上面的结构中，第一个字段 ApiNumber 就是指内核调试 API 的编号。表 18-2 列出了 Windows 2000 的内核调试引擎所定义的所有内核调试 API。

表 18-2 内核调试 API (KdAPI)

ApiName	ApiNumber	功能描述
DbgKdReadVirtualMemoryApi	0x00003130L	读虚拟内存
DbgKdWriteVirtualMemoryApi	0x00003131L	写虚拟内存
DbgKdGetContextApi	0x00003132L	取上下文结构 (CONTEXT)
DbgKdSetContextApi	0x00003133L	设置上下文结构 (CONTEXT)
DbgKdWriteBreakPointApi	0x00003134L	向指定地址写中断指令
DbgKdRestoreBreakPointApi	0x00003135L	恢复指定地址的中断指令
DbgKdContinueApi	0x00003136L	恢复目标系统继续运行
DbgKdReadControlSpaceApi	0x00003137L	读控制空间
DbgKdWriteControlSpaceApi	0x00003138L	写控制空间
DbgKdReadIoSpaceApi	0x00003139L	读 I/O 空间
DbgKdWriteIoSpaceApi	0x0000313AL	写 I/O 空间
DbgKdRebootApi	0x0000313BL	重新启动目标系统
DbgKdContinueApi2	0x0000313CL	设置并恢复目标系统继续运行
DbgKdReadPhysicalMemoryApi	0x0000313DL	读物理内存
DbgKdWritePhysicalMemoryApi	0x0000313EL	写物理内存
DbgKdQuerySpecialCallsApi	0x0000313FL	查询需要特殊对待的函数调用
DbgKdSetSpecialCallApi	0x00003140L	设置需要特殊对待的函数调用
DbgKdClearSpecialCallsApi	0x00003141L	清除所有需要特殊对待的函数调用列表

续表

代号	取值	功能
DbgKdSetInternalBreakPointApi	0x00003142L	设置内部断点
DbgKdGetInternalBreakPointApi	0x00003143L	读取内部断点
DbgKdReadIoSpaceExtendedApi	0x00003144L	读 ALPHA 系统的 I/O 空间
DbgKdWriteIoSpaceExtendedApi	0x00003145L	写 ALPHA 系统的 I/O 空间
DbgKdGetVersionApi	0x00003146L	读取版本信息
DbgKdWriteBreakPointExApi	0x00003147L	写多个断点
DbgKdRestoreBreakPointExApi	0x00003148L	恢复多个断点
DbgKdCauseBugCheckApi	0x00003149L	触发错误检查（蓝屏崩溃）
DbgKdSwitchProcessor	0x00003150L	切换当前处理器
DbgKdPageInApi	0x00003151L	过时不用
DbgKdReadMachineSpecificRegister	0x00003152L	读 CPU 的 MSR 寄存器
DbgKdWriteMachineSpecificRegister	0x00003153L	写 CPU 的 MSR 寄存器
OldVlm1	0x00003154L	功能不详
OldVlm2	0x00003155L	功能不详
DbgKdSearchMemoryApi	0x00003156L	搜索内存
DbgKdGetBusDataApi	0x00003157L	读总线数据（如 PCI 设备的配置空间）
DbgKdSetBusDataApi	0x00003158L	写总线数据（如 PCI 设备的配置空间）
DbgKdCheckLowMemoryApi	0x00003159L	用来实现扩展命令!chklowmem

在表 18-2 中，DbgKdReadMachineSpecificRegister 到 DbgKdCheckLowMemoryApi（包括这两个）是 Windows 2000 引入的。

当内核调试引擎接收到调试器发送的服务请求（操纵状态 API）时，会先发送确认包，然后再以同样子类型（PacketType 等于 PACKET_TYPE_KD_STATE_MANIPULATE）的数据包回复给调试器，头结构后依然是一个 DBGKD_MANIPULATE_STATE 结构，并且 ApiNumber 等于调试器请求的号码。调试器收到后，也是先发送确认包，然后要么发送下一个服务请求，要么发送继续运行 API（DbgKdContinueApi），让目标系统恢复运行。

18.5.4 恢复目标系统执行

当完成了访问目标系统的各种操作时，或者当用户在调试器中发出了恢复目标系统继续执行的命令（比如 g）时，调试器会通过 DbgKdContinueApi 或 DbgKdContinueApi2 来让目标系统恢复运行，也就是发送 ApiNumber 等于这两个 API 号码的操纵状态包。DBGKD_MANIPULATE_STATE 结构的 u 联合中应该是一个 DBGKD_CONTINUE 结构或 DBGKD_CONTINUE2 结构：

```
typedef struct _DBGKD_CONTINUE {
    NTSTATUS ContinueStatus;
} DBGKD_CONTINUE, *PDBGKD_CONTINUE;
```

其中 ContinueStatus 可以为以下几个值之一：

- DBG_CONTINUE (0x00010002L): 恢复执行。
- DBG_EXCEPTION_HANDLED (0x00010001L): 恢复执行，并告知系统调试器已经处理异常状态，让异常分发函数结束异常分发。
- DBG_EXCEPTION_NOT_HANDLED (0x80010001L): 恢复执行，并告知系统调试器没有处理异常。

```
typedef struct _DBGKD_CONTINUE2 {
    NTSTATUS ContinueStatus;
    DBGKD_CONTROL_SET ControlSet;
} DBGKD_CONTINUE2, *PDBGKD_CONTINUE2;
```

`ContinueStatus` 的含义与上面一样，不过使用这个结构还可以同时向内核调试引擎传递一个 `DBGKD_CONTROL_SET` 结构，让它先设置好结构中指定的信息后再恢复执行。

```
typedef struct _DBGKD_CONTROL_SET {
    ULONG TraceFlag;           // 追踪标志，用于单步跟踪
    ULONG Dr7;                 // 调试控制寄存器，用于硬件断点
    ULONG CurrentSymbolStart;  // 希望调试引擎进行本地跟踪的起始地址
    ULONG CurrentSymbolEnd;   // 希望调试引擎进行本地跟踪的结束地址
} DBGKD_CONTROL_SET, *PDBGKD_CONTROL_SET;
```

因为 `DBGKD_CONTROL_SET` 结构中的内容是调试器每次恢复执行前都需要设置的内容，如果要使用 `DbgKdContinueApi`，就需要发送数据包做这些设置，然后再发送 `DbgKdContinueApi`。而使用 `DbgKdContinueApi2` 时，就可以在一个数据包中同时做这两件事，这也就是加入后者的好处。

18.5.5 版本

随着 Windows 内核的发展，调试引擎也在逐渐发展和改进，相应的内核调试协议也需要随之发展。当建立内核调试对话时，调试器会通过 `DbgKdGetVersionApi` 来查询对方的版本号，以确定目标系统中调试引擎和调试协议的版本号。

Windows 2000 内核调试引擎的实用的调试协议版本号是 5，Windows XP 和 Vista 使用的都是版本 6。NT4 及以前的 NT 系统使用的版本号为 1 到 4。

从版本 5 开始，不管目标系统是 32 位的还是 64 位的，所有信息包中的数据结构都使用 64 位版本。调试器可以通过 `DbgKdGetVersionApi` 来读取版本信息。尽管这个 API 的数据结构也有 32 位和 64 位两种定义，但是，因为这两种定义的前 3 个字段都一样，所以调试器可以得到确切的协议版本号，并据其判断出这个结构到底是 32 位的还是 64 位的。

```
typedef struct _DBGKD_GET_VERSION64 {
    USHORT MajorVersion;        // 内核的主版本号
    USHORT MinorVersion;        // 内核的子版本号
    USHORT ProtocolVersion;     // 内核调试协议的版本号
}

typedef struct _DBGKD_GET_VERSION32 {
```

```

USHORT MajorVersion;           // 内核的主版本号
USHORT MinorVersion;          // 内核的子版本号
USHORT ProtocolVersion;       // 内核调试协议的版本号
...

```

18.7 节会详细介绍建立内核调试连接的过程和这个数据结构的全部字段。

18.5.6 典型对话过程

深入理解内核调试协议的一种有效办法，就是使用串行口通信监视工具记录下调试器与内核调试引擎之间的所有通信记录，然后分析它们之间的对话过程。表 18-3 列出了使用 WinDBG 通过串行口调试一个 Windows XP SP2 系统时二者之间的对话过程。第一列为步骤编号，只是为了便于引用。

表 18-3 内核调试的典型对话过程

1	启动，选择 File → Kernel Debug 选 COM 方式，OK	(配置好 BOOT.INI，但尚未启动)
2	打开 COM 口，设置通信参数	
3	发送复位控制包 (PACKET_TYPE_KD_RESET, 6)	
4	读取 COM 口	
5	每隔 10 秒重复以上两步一次 (仍在) 循环 3、4 步	以调试选项启动 内核调试引擎初始化
6	(仍在) 循环 3、4 步	接收到复位包后，将全局变量 KdDebuggerNotPresent 设置为 0
7	(仍在) 循环 3、4 步	冻结内核，发送类型 7 (KD_STATE_CHANGE64) 信息包， NewState = 0x3031，报告加载 NTOSKRNL.EXE 的符号
8		收到复位包后，重新发送类型 7 信息包
9	收到类型 7 信息包后，发送确认控制包， 显示已经与内核调试引擎建立连接	
10	发送操纵类信息包“调用” DbgKdGetVersionApi，取目标版本	收到 DbgKdGetVersionApi 请求后，发送确认控制包
11		通过操纵类信息包返回版本信息结构
12	收到版本信息结构后，先发送确认控制包， 然后根据版本信息初始化调试器引擎	
13	请求 KdReadVirtualMemoryApi	确认
14		回复 KdReadVirtualMemoryApi*
15	多次请求 KdReadVirtualMemoryApi	确认和回复 KdReadVirtualMemoryApi
16	请求 DbgKdRestoreBreakPointApi	确认和回复 DbgKdRestoreBreakPointApi
17	重复 15 步，共执行 32 次，对应于 32 个 断点位置	确认和回复 DbgKdRestoreBreakPointApi
18	请求 DbgKdClearAllInternalBreakpointsApi， 清除所有内部断点	确认，不需要额外回复

续表

#	WinDBG 传递给 (主机)	内核调试引擎 (目标机)
19	请求 KdReadVirtualMemoryApi	确认和回复 KdReadVirtualMemoryApi
20	请求 DbgKdGetContextApi	确认和回复 DbgKdGetContextApi
21	请求 DbgKdReadControlSpaceApi	确认和回复 DbgKdReadControlSpaceApi
22	请求 KdReadVirtualMemoryApi	确认和回复 KdReadVirtualMemoryApi
23	请求 DbgKdSetContextApi	发送重发控制包
24	重新发送 DbgKdSetContextApi 请求	确认和回复 DbgKdSetContextApi
25	请求 DbgKdWriteControlSpaceApi	确认和回复 DbgKdWriteControlSpaceApi
26	请求 DbgKdGetContextApi	确认和回复 DbgKdGetContextApi
27	请求 DbgKdReadControlSpaceApi	确认和回复 DbgKdReadControlSpaceApi
28	请求 DbgKdContinueApi2	确认, 然后恢复内核执行
29		发送类型 7 信息包, 通知加载符号[2627]
30	确认类型 7 信息包	
31	3 次调用 DbgKdReadVirtualMemoryApi	确认和回复 DbgKdReadVirtualMemoryApi
32	请求 DbgKdContinueApi	确认, 不需回复, 恢复内核执行
33	第 29 到 32 步重复十几次, 对应于 NTLDR 所加载的第一批内核模块中的每一个	
34		发送类型 11 信息包 (内核文件 IO) [10768]
35	确认和回复类型 11 信息包	接到类型 11 回复包后, 确认并恢复内核运行
36	内核加载第 34 行所对应的驱动程序, 发送类型 7 信息包通知调试器加载符号, 即重复第 31 和 32 行的操作	
37	对于每个非启动类型的驱动程序重复第 34 到 36 行 [10768~22606]	
38	发送中断包, 一个字节的 0x62[22661]	
39		发送状态变化信息包, NewState=DbgKdExceptionStateChanged
40	请求 DbgKdReadVirtualMemoryApi	确认和回复 DbgKdReadVirtualMemoryApi
41	请求 DbgKdGetContextApi	确认和回复 DbgKdGetContextApi
42	多次请求 DbgKdReadVirtualMemoryApi	确认和回复 DbgKdGetContextApi
43	省略多次恢复断点和读取控制空间的操作	
44	调试器进入命令模式, 允许用户输入各种调试命令, 进行交互式调试。 试验中时设置断点	
45	多次请求 DbgKdWriteBreakPointApi	确认和回复 DbgKdWriteBreakPointApi
46	请求 DbgKdContinueApi2	
47		(断点命中) 发送类型 7 信息包
48	多次请求 DbgKdRestoreBreakPointApi	确认和回复 DbgKdRestoreBreakPointApi
49	调试器进入命令模式, 过程与第 43 到 46 步类似 [25260~26106]	
50	反复读取 COM 口, 超时 [26119~26177]	(内核恢复运行, 继续启动)
51		发送类型 3 信息包 (PACKET_TYPE_KD_DEBUG_IO) 输出调试信息
52	确认收到类型 3 信息包	

* 对于内核调试引擎的每个回复, 调试器正确接收后都会发送确认控制包, 为节约篇幅, 下面各行省略了这一步。

表 18-3 中概括了使用一个预先启动好的调试器与启用了内核调试选项后的 Windows XP 系统之间的交互过程, 包含了以下几个重要动作。

- 建立连接，即第 4~9 行。
- 调试器读取目标系统信息，初始化调试引擎的过程，第 10~28 行。其中第 23~24 行包含了一次调试引擎要求重新发送的操作。
- 内核调试引擎通过状态变化信息包通知调试器加载初始模块的调试符号，第 29~37 行。
- 调试器端发送中断包，将目标系统中断到调试器，交互调试后又恢复执行的过程，第 38~46 行。
- 因为断点命中，目标系统中断到调试器的过程，第 47~49 行。
- 内核中的模块输出调试字符串（DbgPrint）到调试器，第 51~52 行。

表中的内容来自于使用 PortMon 工具产生的通信记录，为了便于处理，我们把这些信息放入到 Excel 文件中，表格中方括号中的数字是对应 Excel 行号的。

18.5.7 KdTalker

在理解了内核调试协议后，就可以编写一个简单的程序与内核调试引擎进行对话。KdTalker 便是这样一个小程序。点击界面上的 Start 按钮，KdTalker 便启动一个线程，按照我们前面介绍的内核调试协议，试图与内核调试引擎建立连接。一旦成功接收到调试引擎的数据包并建立连接后，它会以最简单的方式应付对方，让目标内核继续运行。KdTalker 支持发送中断包，将运行着的目标系统中断，但是中断后，KdTalker 并不支持很有用的调试命令，只支持恢复目标继续执行。因此，KdTalker 可以帮助我们学习和理解内核调试协议和内核调试的对话过程。

本节比较详细地介绍了内核调试协议，讨论了内核调试引擎如何与调试器对话，特别是对话的流程及使用的数据结构和常量。下一节我们将介绍调试引擎如何与操作系统内核进行交互，以实现各种调试功能。

18.6 与内核交互

我们在第 18.1 节中介绍内核调试引擎的角色时曾说，对位于另一台系统中的调试器来说，内核调试引擎是它派驻在目标系统中的代理，调试引擎代表调试器来访问和控制目标系统。本节我们将介绍内核调试引擎与系统内核其他部分之间进行交互的一些重要过程。

为了使语言更简洁，在本节中，我们将内核调试引擎简称为调试引擎，将内核的其他部分简称为内核。

18.6.1 中断到调试器

在调试引擎向调试器报告状态变化信息包之前, 它会调用 KdEnterDebugger 函数来冻结内核。直到收到调试器的恢复继续执行命令 (如 DbgKdContinueApi 和 DbgKdContinueApi2) 后, 再调用 KdExitDebugger 恢复内核运行。

KdEnterDebugger 执行的动作主要有。

- 调用内核的调试支持函数 KeFreezeExecution()。KeFreezeExecution 会调用 KeDisableInterrupts 禁止中断, 对于多处理器的系统, 它会将当前 CPU 的 IRQL 升高到 HIGH_LEVEL, 并且冻结所有其他的 CPU。
- 锁定调试通信端口, 即获取全局变量 KdpDebuggerLock 所代表的锁对象。
- 调用 KdSave (或 Windows 2000 及以前为 KdPortSave) 让通信扩展模块保存通信状态。
- 将全局变量 KdEnteredDebugger 设置为真。

在 KdEnterDebugger 被执行后, 整个系统进入一种简单的单任务状态, 当前的 CPU 只执行当前的线程, 其他 CPU 处于冻结状态。

当前 CPU 接下来要做的是执行状态变化报告函数, 如果要报告的是异常类状态变化, 那么会执行 KdpReportExceptionStateChange 函数, 如果要报告的是符号加载类状态变化, 那么会执行 KdpReportLoadSymbolsStateChange 函数。

这两个状态变化报告函数在准备好状态变化信息包的内容(即 DBGKD_WAIT_STATE_CHANGE64 或 DBGKD_WAIT_STATE_CHANGE32 数据结构)后, 会调用 KdpSendWaitContinue 函数来发送信息包, 并与调试器进行对话, 直到收到恢复执行命令。

18.6.2 KdpSendWaitContinue

KdpSendWaitContinue 函数是调试引擎中与调试器进行交互式对话的主要函数。它先将状态变化信息包发送给调试器, 然后开始一个循环反复等待和处理来自调试器的操纵状态信息包 (PACKET_TYPE_KD_STATE_MANIPULATE)。其伪代码如清单 18-1 所示。

清单 18-1 KdpSendWaitContinue 函数的伪代码

```

1  KdpSendWaitContinue()
2  {
3      ULONG Length;
4      STRING DataBody;
5      STRING MoreData;
6      DBGKD_MANIPULATE_STATE ManipulateState;
7      DataBody.MaximumLength=sizeof(ManipulateState);
8      DataBody.Buffer=(PUCHAR)&ManipulateState;
9      MoreData.MaximumLength=0x1000;

```

```

10         MoreData.Buffer=(PUCHAR)KdpMessageBuffer;
11
12         KdSendPacket();
13
14         do
15         {
16             KdReceivePacket(
17                 PACKET_TYPE_KD_STATE_MANIPULATE,
18                 &DataBody,
19                 &MoreData,
20                 &Length);
21             switch(ManipulateState.ApiNumber)
22             {
23                 case DbgKdReadVirtualMemoryApi:
24                     KdpReadVirtualMemory();
25                     break;
26                 case DbgKdWriteVirtualMemoryApi:
27                     KdpWriteVirtualMemory();
28                     break;
29                 // 代表其他内核调试 API 的 DbgKdXXXApi 常量 ...
30                 //
31                 case DbgKdContinueApi:
32                 case DbgKdContinueApi2:
33                     return;
34                 default:
35                     DataBody.Length=0;
36                     ManipulateState.ReturnStatus=STATUS_UNSUCCESSFUL;
37                     KdSendPacket(PACKET_TYPE_KD_STATE_MANIPULATE,
38                                 &DataBody, &MoreData);
39                     break;
40             }
41         }while(TRUE);
42     }

```

一个完整的操纵状态信息包分为 4 个部分，最前面是 KD_PACKET 结构，而后是 PACKET_TYPE_KD_STATE_MANIPULATE32 或 PACKET_TYPE_KD_STATE_MANIPULATE64 结构（我们使用 PACKET_TYPE_KD_STATE_MANIPULATE 泛指它们中的一个），其次是与 ApiNumber 有关的额外数据，最后一个代表整个包的末尾字节 0xAA。因为开头和最后的末尾字节主要是用来控制通信过程的，所以像 KdpSendWaitContinue 这样的函数只关心中间两个部分。当它们调用 KdReceivePacket 函数时向其传递了两个指针来接收这两部分的信息（第 16~20 行）。

第 21~40 行是一个庞大的 switch...case 结构，用来分发来自调试器的 KdAPI 请求。对于大多数 API，它都是将其转发给调试引擎的内部函数 KdpXXX，这些内部函数会真正提供 API 所要求的服务，并将处理结果通过 KdSendPacket 发送给调试器。对于未知的 API 号码，这个函数会直接回复，在返回的 ManipulateState 结构中指定 STATUS_UNSUCCESSFUL。在收到 DbgKdContinueApi 或 DbgKdContinueApi2 后，这个函数会返回到调用它的函数，后者再调用 KdExitDebugger 退出调试器，结束本次交互式对话。

18.6.3 退出调试器

在调试引擎收到调试器的恢复继续执行命令（如 DbgKdContinueApi 和 DbgKdContinueApi2）后，会调用 KdExitDebugger 恢复内核运行。

KdExitDebugger 所执行的动作主要有。

- 调用 KdRestore（或 Windows 2000 及以前为 KdPortRestore）让通信扩展模块恢复通信状态。
- 对锁定的调试通信端口解锁，即释放全局变量 KdpDebuggerLock 所代表的锁对象。
- 调用 KeThawExecution 来恢复系统进入正常的运行状态，包括恢复中断，降低当前 CPU 的 IRQL，对于多 CPU 系统，恢复其他 CPU。

系统执行 KdExitDebugger 函数后，又恢复到正常运行的状态。这时，如果想让系统再中断到调试器中，调试器可以发送中断包（Breakin Packet）。那么，调试引擎是如何检测到调试器发送的中断包的呢？

18.6.4 轮询中断包

为了支持内核调试，系统的 KeUpdateSystemTime 函数在每次更新系统时间时会检查全局变量 KdDebuggerEnabled 来判断内核调试引擎是否被启用，如果这个变量为真，便调用 KdPollBreakIn 函数来查看调试器是否发送了中断命令，如果是，便调用 DbgBreakPointWithStatus 触发断点异常，以中断到调试器。其伪代码如下：

```
KeUpdateSystemTime()
{
    // ...
    if (KdDebuggerEnabled
        && KdPollBreakIn())
    {
        DbgBreakPointWithStatus(DBG_STATUS_CONTROL_C/*(1)*/');
    }
    // ...
}
```

当系统的时钟（System Timer）中断（一般为 0 号中断）发生时，中断处理函数（通常为 hal!HalpClockInterrupt）会跳转到 KeUpdateSystemTime 函数。时钟中断是频繁发生的硬件中断。当系统正常运行时，这个中断会一直按一定的时间间隔不断触发，也可以说它是推动系统运动的脉搏。系统在每次响应这个中断时来检查是否需要中断到调试器，可以达到非常高的可靠性和灵敏性。

当我们在调试器端按 Ctrl+C（KD）或 Ctrl+Break（WinDBG）目标系统中断到调试器后，观察栈回溯序列，可以看到 KeUpdateSystemTime 函数：

```
kd> kn
# ChildEBP RetAddr
00 80541ebc 805120f8 nt!RtlpBreakWithStatusInstruction
01 80541ebc 806ccefa nt!KeUpdateSystemTime+0x142
02 80541f40 804eed89 hal!HalProcessorIdle+0x2
03 80541f50 804f1d65 nt!PopIdle0+0x47
04 80541f54 00000000 nt!KiIdleLoop+0x10
```

以上栈序列的 2~4 帧说明，当时钟中断发生时，当前 CPU 正在 Idle 线程中执行 hal!HalProcessorIdle 函数。

有的读者可能会想到为什么没有使用一个系统线程来检查是否有中断包。这主要有两个原因：第一，如果使用固定的线程，那么每次中断的上下文都是那个固定的系统线程，而使用现有的方法，中断会在当时 CPU 正在执行的线程发生，这显然更符合调试的需要。第二，时钟中断具有较高的优先级（参见 3.3 节），即使系统中发生了比较严重的故障，只要时钟中断的响应和处理还正常，就可以把系统中断到调试器中，以观察系统中的情况。但是，如果使用一个专门的系统线程，那么这个线程此时可能已经得不到执行的机会。

18.6.5 接收和报告异常事件

异常与调试有着密不可分的关系，首先很多软件问题与代码中的异常有关，其次，包括断点、单步跟踪等很多调试机制也是依靠异常机制来实现的。在第 9 章我们介绍了用户态调试器是如何接收发生在目标程序中的异常事件的，概括来说，是系统的异常分发函数（KiDispatchException）通过用户态调试子系统（DbgSS）转发给用户态调试器的。

那么，内核调试器是如何接收到目标系统中的异常事件的呢？与用户态的情况有些类似，简单来说，就是系统的异常分发函数通过内核调试引擎（KD Engine）转发给内核调试器。

我们在第 11 章介绍 KiDispatchException 分发异常的过程时提到过，对于发生在内核态的异常，KiDispatchException 函数会调用全局变量 KiDebugRoutine 所指向的函数。当调试引擎被启用时，这个变量的值是函数 KdpTrap 的地址。这意味着，当内核态异常发生时，系统会调用 KdpTrap 函数。调试引擎就是通过这个函数从系统内核接收异常事件的。

```
BOOLEAN KdpTrap (
    IN PKTRAP_FRAME TrapFrame,
    IN PKECEPTION_FRAME ExceptionFrame,
    IN PEXCEPTION_RECORD ExceptionRecord,
    IN PCONTEXT ContextRecord,
    IN KPROCESSOR_MODE PreviousMode,
    IN BOOLEAN SecondChance)
```

从 KdpTrap 函数的原型可以看出，它与 KiDispatchException 函数的原型很类似，或者说，后者是把与异常有关的所有信息通过参数传递给了 KdpTrap 函数。

`KdpTrap` 函数根据 `ExceptionRecord` 判断所发生的异常，特别是标志异常类型的 `ExceptionRecord->ExceptionCode` 字段，对于断点指令异常、调试异常（`STATUS_SINGLE_STEP`）和所有第二轮异常（`SecondChance` 为 `TRUE`），`KdpTrap` 会调用 `KdpReport` 向调试器报告异常，也就是执行以下动作。

1. 调用 `KdEnterDebugger` 冻结内核的其他部分。
2. 调用 `KiSaveProcessorControlState` 保存处理器的控制状态。
3. 调用 `KdpReportExceptionStateChange` 向调试器报告异常状态变化，在这个函数发送状态变化信息包后，会接收和处理调试器的各种调试命令（操纵状态 API），直到收到恢复执行命令。
4. 调用 `KiRestoreProcessorControlState` 恢复 CPU 状态。
5. 调用 `KdExitDebugger` 恢复内核运行。

`KdpTrap` 函数的返回值代表了内核调试器是否处理了该异常，如果为真，那么表示处理，否则表示没有处理。`KdpTrap` 函数根据调试器的恢复执行信息包（`DbgKdContinueApi`）中的 `ContinueStatus` 来决定返回真或假。

18.6.6 调试服务

上面介绍的通过异常分发函数和 `KdpTrap/KdpReport` 函数向内核调试器发送信息的方法，也被复用来实现以下调试功能。

- 打印调试信息，例如，内核态代码通过调用 `DbgPrint` 函数所打印的字符串。
- 征求用户输入（`Prompt`）。
- 报告模块加载事件。
- 报告模块卸载事件。

当需要执行以上任务时，系统会触发一个软件异常，编号为 `0x2D`。通常把这个异常称为调试服务（`Debug Service`）异常。观察 IDT 表，可以看到这个异常的处理函数是 `KiDebugService`：

```
1kd> !idt 2d
Dumping IDT:
2d:      8053db50 nt!KiDebugService
```

`KiDebugService` 做了一些预处理工作后，便跳转到 `KiTrap03`。我们知道，`KiTrap03` 是断点异常的处理函数。因此，从 `KiDebugService` 跳转到 `KiTrap03` 后，要传递的调试信息会被当作断点异常的参数信息而传递给异常分发函数，再传递给 `KdpTrap` 函数，最后传递给调试器。

`ExceptionRecord` 结构的 `ExceptionInformation[0]` 字段标识了这个异常结构记

录的是一个真正的断点异常，还是调试服务请求，它可以为以下几个值之一。

- BREAKPOINT_BREAK (0): 真正的断点异常。
- BREAKPOINT_PRINT (1): 打印调试信息，ExceptionInformation [1] 为要打印的字符串 (STRING 指针)。
- BREAKPOINT_PROMPT (2): 提示并请求用户输入，ExceptionInformation [1] 指向提示字符串，ExceptionInformation [2] 指向的是用来存放用户输入的缓冲区。
- BREAKPOINT_LOAD_SYMBOLS (3): 加载符号文件，ExceptionInformation [1] 指向模块文件的名称，ExceptionInformation [2] 是模块的基地址。
- BREAKPOINT_UNLOAD_SYMBOLS (4): 卸载符号文件，ExceptionInformation [1] 指向模块文件的名称，ExceptionInformation [2] 是模块的基地址。

根据上面的信息，尽管断点指令 (INT 3) 和调试服务所发起异常的异常代码都是 STATUS_BREAKPOINT (0x80000003)，但是系统可以根据 ExceptionInformation 数组来区分它们。KdpTrap 函数正是如此做的：对于 BREAKPOINT_PRINT，KdpTrap 会调用 KdpPrint 函数。稍后我们再介绍其细节。对于 BREAKPOINT_PROMPT，KdpTrap 会调用 KdpPrompt 函数，后者先调用 KdEnterDebugger 冻结内核，再调用 KdpPromptString，待返回后调用 KdExitDebugger 恢复内核运行。KdpPromptString 内部会使用 KdSendPacket 把提示信息发送给调试器，然后使用 KdReceivePacket 接收用户输入。对于 BREAKPOINT_LOAD_SYMBOLS 和 BREAKPOINT_UNLOAD_SYMBOLS，KdpTrap 会调用 KdpSymbol 函数，稍后我们介绍其细节。

为了更方便调用调试服务 (INT 2D)，内核中设计了两个简单的函数 DebugService 和 DebugService2，这两个函数的实现是先把参数传递给寄存器，然后执行 INT 2D 指令。例如 DebugService2 函数的反汇编代码如下：

```
nt!DebugService2:
8052da3c 8bff        mov    edi,edi
8052da3e 55          push   ebp
8052da3f 8bec        mov    ebp,esp
8052da41 8b4510      mov    eax,dword ptr [ebp+10h]
8052da44 8b4d08      mov    ecx,dword ptr [ebp+8]
8052da47 8b550c      mov    edx,dword ptr [ebp+0Ch]
8052da4a cd2d        int    2Dh
8052da4c cc           int    3
8052da4d 5d           pop    ebp
8052da4e c20c00      ret    0Ch
```

因为调试服务是当作断点异常来统一处理的，而处理断点异常时，系统会自动跳过断点指令，因此倒数第 3 行是必需的。

DebugService2 有 3 个参数，第 3 个是服务类型，其值就是上面介绍的 BREAKPOINT_PRINT 等常量，它被放在 EAX 寄存器中，第一个和第二个是与服务类型有关的附件信息，分别放在 ECX 和 EDX 中。

`DbgLoadImageSymbols`、`DbgUnLoadImageSymbols`、`DebugPrint` 和 `DebugPrompt` 函数是内核中使用调试服务的几个主要函数，观察它们的实现可以看到，这些函数都非常简单，整理好参数后便调用 `DebugService`/`DebugService2` 函数。Vista 中的 `DebugService`/`DebugService2` 函数在编译时被 `inline` 了，所以看到的是以上函数直接执行 INT 2D 指令。

18.6.7 打印调试信息

与用户态的 `OutputDebugString` API 类似，DDK 中公开了 3 个函数供内核代码打印调试信息，一个是 `DbgPrint`，另两个是 Windows XP 引入的 `DbgPrintEx` 和 `vDbgPrintEx`。它们的原型为。

- `ULONG DbgPrint (PCHAR Format, ...)。`
- `ULONG DbgPrintEx (ULONG ComponentId, ULONG Level, PCHAR Format, ...)。`
- `ULONG vDbgPrintEx(IN ULONG ComponentId, IN ULONG Level, IN PCCH Format, IN va_list arglist)。`

其中 `ComponentId` 用来指定组件 ID，头文件 `Dpfilter.h` 中定义了 Windows 的各个系统组件所使用的 ID，其他程序可以使用 DDK 中规定的几个 ID（`DPFLTR_IHVDRIVER_ID` 等）之一。`Level` 用来指定信息的严重程度，0 通常代表最严重。

从上面的原型可以看出，`DbgPrintEx` 和 `vDbgPrintEx` 只是表达可变数量参数的方式不同。

事实上这 3 个函数的内部实现是非常类似的，都是经过简单的处理后就调用 `vDbgPrintExWithPrefix` 函数。`DbgPrint` 会使用默认的组件 ID (`DPFLTR_DEFAULT_ID`) 和严重级别 (`DPFLTR_INFO_LEVEL`)。

`vDbgPrintExWithPrefix` 内部会先检查是否应该输出这个信息。检查时既会考虑全局设置，又会考虑组件 ID。在 XP 开始的系统中，系统中定义了很多名为 `KD_XXX_MASK` 的 `ULONG` 类型全局变量，用来标识 XXX 部件的调试信息输出过滤掩码 (Debug Print Filter Mask)，这些变量的地址被以一个数组的形式存放在一起，全局变量 `KdComponentTable` 记录着这个数组的地址，`KdComponentTableSize` 记录了数组的元素个数。系统可以根据组件 ID，找到它在 `KdComponentTable` 中的掩码变量地址，然后根据 `Level` 值和掩码值决定是否应该输出这个信息。

对于使用默认组件 ID 的信息输出，XP 会将其送给调试器。但是 Vista 不会，如果希望改变这一状况，可以修改全局变量 `Kd_DEFAULT_MASK`，将它的值设置为 `0xF`。

`vDbgPrintExWithPrefix` 在确认应该输出调试信息后，它会调用 `DebugPrint` 请求调试服务。后者执行 INT 2D 触发异常，根据前面的介绍，异常分发函数会调用 `KdpTrap`，

KdpTrap 会调用 KdpPrint 函数。

KdpPrint 内部先调用 KdLogDbgPrint 将要打印的字符串记录到环形缓冲区中，然后根据系统的设置判断是否需要发送给内核调试器（再检查一次 KdComponentTable），如果需要，则先调用 KdEnterDebugger 冻结内核，然后调用 KdpPrintString 发送 PACKET_TYPE_KD_DEBUG_IO 类型的信息包给调试器，最后再调用 KdExitDebugger 恢复内核运行。冻结内核的目的是，调试器也可以在接收到这类信息包时进入交互式调试。但因为每次打印信息时要执行冻结和解冻操作，所以频繁的信息打印会严重影响系统的运行速度。

18.6.8 加载调试符号

系统的模块加载函数 MmLoadSystemImage 在成功加载一个模块后会调用 MiDriverLoadSucceeded 函数，后者会调用 DbgLoadImageSymbols 函数，而 DbgLoadImageSymbols 函数内部就是调用 DebugService2 触发 0x2D 号异常，请求调试服务。

如果启用了调试引擎，那么系统的异常分发函数会将调试服务异常转交给 KdpTrap 函数来处理。KdpTrap 函数根据异常的附加信息检测到加载符号请求后调用 KdpSymbol 函数。后者先调用 KdEnterDebugger 冻结内核，然后调用 KdpReportLoadSymbolsStateChange 报告调试器，最后调用 KdExitDebugger。

KdpReportLoadSymbolsStateChange 会通过状态变化（类型 2）信息包向调试器报告需要加载的符号文件。这个信息包的头部依然是一个 KD_PACKET 结构，而后是一个 DBGKD_WAIT_STATE_CHANGE64 或 DBGKD_WAIT_STATE_CHANGE32 结构，其中的 NewState 等于 0x00003031，即 DbgKdLoadSymbolsStateChange，联合 u 中存放的是一个 DBGKD_LOAD_SYMBOLS64 或 DBGKD_LOAD_SYMBOLS32 结构：

```
typedef struct _DBGKD_LOAD_SYMBOLS32 {
    ULONG PathNameLength;
    ULONG BaseOfD11;
    ULONG ProcessId;
    ULONG CheckSum;
    ULONG SizeOfImage;
    BOOLEAN UnloadSymbols;
} DBGKD_LOAD_SYMBOLS32, *PDBGKD_LOAD_SYMBOLS32;
```

其中 PathNameLength 代表的是在 DBGKD_WAIT_STATE_CHANGE 结构所跟随的模块文件名称的长度，即这个信息包的第三部分的长度。在整个包的末尾是包结束字节，即 0xAA。

调试器收到以上信息包后，会根据其中的信息更新自己的模块列表。并根据事件设置选项（Event Filters）决定是立刻发送继续命令，还是需要进入命令模式让用户开始交互式调试。

18.6.9 更新系统文件

当进行内核调试时，特别是调试驱动程序时，经常需要使用主机上的文件来替换被调试系统中的文件。Windows XP 所引入的利用内核调试连接来更新系统文件的机制，正是为了满足这种需求而设计的。

当 IO 管理器加载一个驱动程序文件时，内存管理器的映像加载函数 MmLoadSystemImage 会判断内核调试引擎是否被激活 (KdDebuggerEnabled = TRUE)，以及是否有内核调试器已经连接 (KdDebuggerNotPresent = FALSE)，如果这两个条件都成立，便调用内核调试引擎的 KdPullRemoteFile 函数，典型调用序列如下：

```
kd> kn
# ChildEBP RetAddr
f898c690 805d9f35 nt!KdPullRemoteFile+0x52
f898c838 80558b1e nt!MmLoadSystemImage+0x1fc
f898c904 80555417 nt!IoPnPLoadDriver+0x311
...
```

其中 KdPullRemoteFile 的第一个参数就是准备加载的驱动程序文件名称：

```
kd> ds f898c8f8
e1aea30  "\SystemRoot\system32\drivers\wdm"
e1aea30  "aud.sys"
```

KdPullRemoteFile 被调用后，调试引擎会通过 PACKET_TYPE_KD_FILE_IO(11) 信息包（以下简称类型 11 信息包）将要加载的文件名报告给调试器。调试器端通常会维护一个映射表，这个映射表记录了需要更新的驱动文件在远程的全路径及在本地的全路径。在 WinDBG 调试器中，可以使用.kdfiles 命令向这个映射表中增加或减少映射。具体方法是运行记事本程序并输入：

```
map
\??\c:\windows\system32\drivers\realbug.sys
c:\DbgLabs\realbug\objchk_wxp_x86\i386\realbug.sys
```

第 2 行是要更新文件在目标系统中的全路径，第 3 行是新文件的全路径，将以上内容保存到一个文本文件中，例如 c:\windbg\maps\realbug.sys，然后在 WinDBG 调试器中执行.kdfiles 命令：

```
kd> .kdfiles c:\windbg\maps\realbug.txt
KD file associations loaded from 'c:\windbg\maps\realbug.txt'
```

调试器收到调试引擎的类型 11 信息包后，会检查这个文件是否在自己的文件映射表中，如果在，那么会通过回复包告诉调试引擎需要从远程读取这个文件。调试引擎收到回复后，会开始与调试器继续对话读取这个文件。整个文件读取结束后，KdPullRemoteFile 函数返回，此时磁盘上的文件已经是更新后的文件，系统继续运行开始加载更新后的文件。例如，以下便是被调试系统加载 realbug.sys 文件时，WinDBG 调试器所显示的信息：

```
KD: Accessing 'c:\DbgLabs\realbug\objchk_wxp_x86\i386\realbug.sys'
(\??\C:\WINDOWS\system32\drivers\RealBug.SYS)
File size 4K.
MmLoadSystemImage: Pulled \??\C:\WINDOWS\system32\drivers\RealBug.SYS from kd
```

需要说明的是，映射文件中的文件路径和名称是不区分大小写的，但必须严格匹配，可以按 Ctrl+Alt+D 热键让 WinDBG 显示通信细节以了解需要匹配的文件路径，例如当我们使用.kdfiles -c 清除映射关联后再加载 RealBug 驱动程序时，WinDBG 会输出：

```
KdFile request for '\??\C:\WINDOWS\system32\drivers\RealBug.SYS' returns C000000F
```

返回值 C000000F 代表没有匹配的更新文件，如果有，则返回 0。

当启用调试引擎时，内存管理器在加载每个系统文件时，都会重复以上步骤。因此使用这种方法可以更新大多数工作在内核态的程序模块。但是这种方法不能更新启动（BOOT）类型的驱动程序，或者说由操作系统的加载器（OS Loader，如 NTLDLR）加载的驱动程序，因为那时调试引擎还没有开始工作。

18.7 建立和维持连接

通过前面几节的介绍，大家应该对内核调试引擎和内核调试协议有了比较深入的了解。为了巩固这些内容，本节我们将介绍建立和维护内核调试连接的细节。

本节的内容尽管基本适用于 NT 系列的所有操作系统，但是，某些细节仍然与 Windows 内核的版本有关，特别是 Windows XP 之前和之后的差异更大些。所以本节我们将以 Windows Vista RTM 6000 版本为例，讨论的目标系统具有一个双核的 CPU。

18.7.1 最早的调试机会

与 ITP 这样的硬件调试工具相比，使用内核调试引擎进行调试的一个不足就是无法调试引擎初始化之前所出现的问题。比如，无法调试操作系统加载程序（NTLDLR 或 WinLoad 程序）加载内核的过程，也无法调试内核开始工作的最初过程。那么，到底是从内核初始化的哪一阶段开始调试呢？或者说，可以把内核中断到调试器的最早时机是什么时候呢？

简单来说，是在内核的入口函数 KiSystemStartup 调用 HalInitializeProcessor 初始化启动 CPU 后，在调用 KiInitializeKernel 之前。也就是 KdInitSystem 函数第一次被调用的时候。

在 KdInitSystem 第一次被调用并完成基本的初始化工作后，会特意调用 DbgLoadImageSymbols 函数向调试器报告内核模块和 HAL 模块的加载事件。DbgLoadImageSymbols 内部调用通过 INT 2D 请求调试服务，这个异常的处理例程做一些处理后会跳转到断点异常的处理例程，然后系统的异常分发函数调用 KdpTrap，KdpTrap 又调用 DbgKdLoadSymbolsStateChange 向调试器发送类型 7 信息包，即我们在上一节所描述的过程。

因此，在启动过程中，调试引擎向调试器发送的最早数据包就是类型 7 信息包。

这时调试器通常还在发送复位控制包。在调试器收到这个信息包后，会先发送确认包，然后通过类型 2（操纵状态）信息包请求 DbgKdGetVersionApi 服务，以读取调试引擎和目标系统的版本。

调试引擎会使用一个 DBGKD_GET_VERSION64 结构来响应调试器的请求，WinDBG 工具包 SDK 目录的 wdbgexts.h 中包含了这个结构的定义。在调试器中也可以使用 dt 命令来显示这个结构：

```
kd> dt nt!_DBGKD_GET_VERSION64
+0x000 MajorVersion : Uint2B      // 0xf
+0x002 MinorVersion : Uint2B      // 0x1770 即 6000
+0x004 ProtocolVersion : Uint2B   // 6
+0x006 Flags : Uint2B      // 3
+0x008 MachineType : Uint2B      // 0x14c
+0x00a MaxPacketType : UChar     // 12
+0x00b MaxStateChange : UChar    // 3
+0x00c MaxManipulate : UChar    // 0x2e
+0x00d Simulation : UChar
+0x00e Unused : [1] Uint2B
+0x010 KernBase : Uint8B        // 0x81800000
+0x018 PsLoadedModuleList : Uint8B // 0x81908ab0
+0x020 DebuggerDataList : Uint8B // 0x81aefbfec
```

每个字段右侧的注释是我们实验中得到的值。其中 ProtocolVersion 代表内核调试协议的版本，版本 6 代表所有信息包中使用的数据结构都是 64 位版本的。其中 Flags 字段可以为如下几个标志的组合：

DBGKD_VERS_FLAG_MP	0x0001	// 内核是多处理器 (MP) 版本
DBGKD_VERS_FLAG_DATA	0x0002	// DebuggerDataList 字段有效
DBGKD_VERS_FLAG_PTR64	0x0004	// 本地指针 (native pointers) 是 64 位
DBGKD_VERS_FLAG_NOMM	0x0008	// 未启用页机制，不要使用 PTE 解码
DBGKD_VERS_FLAG_HSS	0x0010	// Hardware Stepping Support

因为我们实验中的系统是 32 位的 Vista 系统，双核 CPU，安装程序选用的是适用于多处理器的内核文件，所以上面的 Flags 字段的取值为 3。

在得到版本信息后，WinDBG 的内核调试目标类 LiveKernelTargetInfo 会执行初始化方法 InitFromKdVersion，并在窗口中显示建立连接信息：

```
Connected to Windows Vista 6000 x86 compatible target, ptr64 FALSE
Kernel Debugger connection established.
Symbol search path is:
  SRV*d:\symbols*http://msdl.microsoft.com/download/symbols
Executable search path is:
```

这意味着调试器与内核调试引擎已经成功建立了连接。接下来，调试器会使用 DbgKdReadVirtualMemoryApi 读取目标系统的其他信息，并显示类似如下的信息：

```
Windows Vista Kernel Version 6000 MP (1 procs) Free x86 compatible
Built by: 6000.16386.x86fre.vista_rtm.061101-2205
Kernel base = 0x81800000 PsLoadedModuleList = 0x81908ab0
System Uptime: not available
```

括号中的 1 procs 表示目前系统中有一个处理器在工作，因为尽管是双核的系统，但是另一个 CPU 还没有被唤醒。在做好以上工作后，WinDBG 已经收集到了目标系统

的足够信息，明确知道它的系统类型是 x86 架构，于是会准备加载用于调试 x86 系统的工作空间（Workspace）（详见 30.1 节）。在切换到新的工作空间之前，WinDBG 通常会弹出图 18-11 所示的对话框询问是否保存目前实用的“基础（base）”工作空间。

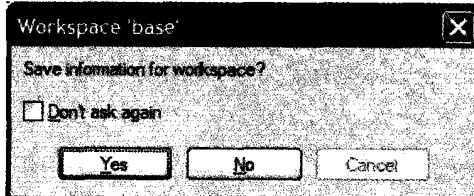


图 18-11 WinDBG 询问是否保存当前的工作空间

特别要提一下这个对话框，在关闭这个对话框前，目标系统是不会继续运行的，因为当调试引擎在发送类型 7 信息包时，它就已经把内核冻结起来了。只有在它收到调试器的恢复执行命令后，才会恢复内核执行。而这个对话框不关闭，WinDBG 就不会发送继续执行信息包，因此内核就会一直被冻结在那里。

当选择 Yes 或 No 时，默认情况下，WinDBG 就会发送继续执行信息包，于是目标系统继续其启动过程。如果希望 WinDBG 进入命令模式，以便可以设置断点或继续分析目标系统，那么需要事先启用 WinDBG 的初始中断（Initial Break）功能。这可以通过 Debug 菜单的 Kernel Connection > Cycle Initial Break 项来操作，也可以按 Ctrl+Alt+K 热键，每发出一次命令，WinDBG 在以下 3 种初始中断设置中切换一次。

- 请求初始断点，即在与调试引擎建立通信后就自动发送一个中断控制包（0x62）。切换到这种模式时，WinDBG 会提示 Will request initial breakpoint at next boot。
- 当得到第一个加载符号报告时中断，即上面讨论的情况。当切换到这种模式时，WinDBG 会提示 Will breakin on first symbol load at next boot。
- 不中断，WinDBG 提示 Will NOT breakin at next boot。

因为第二种方式对应于调试引擎初始化后第一次向调试器发送信息，所以，以这种方式产生的中断要比第一种更早，这也就是我们上面讨论的情况。当选择这种模式时，图 18-11 所示的对话框关闭后，WinDBG 便会进入到命令模式，显示：

```
nt!DbgLoadImageSymbols+0x47:  
8180c533 cc          int     3
```

输入 kv 命令，其结果为：

```
ChildEBP RetAddr Args to Child  
00120dc4 81aa93f8 00120df0 00120db4 ffffffff nt!DbgLoadImageSymbols+0x47  
00120f00 818df2be 00000000 80806b10 818efac0 nt!KdInitSystem+0x37a  
00120f30 00000000 80806b10 00251544 80806b10 nt!KiSystemStartup+0x2be
```

显而易见，以上栈回溯印证了我们前面的介绍，内核的入口函数 KiSystemStartup 调用 KdInitSystem 来初始化调试引擎，前面的第一个参数 0 代表这是阶段 0 初始化，

也就是第一次调用这个函数。观察 `DbgLoadImageSymbols` 函数的第一个参数：

```
kd> ds 00120df0
00120df8  "\SystemRoot\system32\ntoskrnl.exe"
00120e18  "e"
```

可见这个参数正是内核文件的完整名称，也就是说，`KdInitSystem` 在调用 `DbgLoadImageSymbols` 函数通知调试器加载内核文件的符号。而 `DbgLoadImageSymbols` 会通过异常来请求调试服务。请求调试服务不是执行 INT 2D 么，那么上面为什么显示 INT 3 呢？这是因为调试器显示的是 INT 2D 后的那条 INT 3 指令。因为 INT 2D 已经执行过了。

上面的栈回溯还有一点值得注意，那就是栈帧基地址，可以看到它们都是小于 0x80000000 的，也就是此时的栈还在使用 OS Loader 程序创建的低于 2GB 的内存空间。

如果目标系统为 Windows XP，那么此时的栈回溯为类似如下的情形：

```
kd> kv
ChildEBP RetAddr  Args to Child
0005ff80 804d88e7 000600c4 0005ff94 00000003 nt!DebugService2+0xe (FPO: [3,0,0])
0005ffa4 80656c46 000600c4 804d4000 ffffffff nt!DbgLoadImageSymbols+0x40
000600cc 80691ac8 00000000 80087000 80542268 nt!KdInitSystem+0x23e ([Non-Fpo])
00060100 00420e53 80087000 00467904 00438ab7 nt!KiSystemStartup+0x264
WARNING: Frame IP not in any known module. Following frames may be wrong.
00060e3c 0041ec40 00000007 00060e5c 00000000 0x420e53
00060ecc 004014fe 004678e0 0047164f 00051d68 0x41ec40
00061ff0 10101010 00000002 00000000 04e4000d 0x4014fe
00061ff4 00000000 00000000 04e4000d 003f0001 0x10101010
```

首先，第 1 行中的 `DebugService2` 是用来请求调试服务的，Vista 系统中，这个简单的函数被 `inline` 了，所以前面的栈回溯中没有这一行。另外，注意以上信息中的警告，这句话的意思是下一个栈帧（Frame）的程序指针（IP）不属于任何模块，因此接下来的栈帧信息可能是错误的。事实上，下面的栈帧是发生在 `NTLDR` 中的。我们知道，系统加载程序负责加载内核和启动类型的驱动程序，也就是系统的第一批文件，然后调用内核文件的入口函数，即 `KiSystemStartup`。以上栈帧显示了这个调用过程。这也是在内核调试器中能看到的关于 `NTLDR` 的较少痕迹之一。

当通过以上方法将目标系统中断到调试器中时，目标系统处于初始化的起步阶段（内核初始化函数 `KiInitializeKernel` 尚未被调用），只有启动 CPU 才运行，而且它是以单任务方式工作的，也就是说，此时系统中还只有这一个初始的“线程”。此时屏幕还处于简单的黑白界面，所有执行体尚未初始化，内核对象和进程列表尚未建立，因此包括进程枚举（`!process 0 0`）在内的一些命令都会返回错误。

18.7.2 初始断点

如果不使用上面介绍的收到符号加载事件时中断，那么将目标系统中断到调试器的最早机会就是初始断点（Initial Breakpoint）。其做法是在目标系统启动前启动

WinDBG 的内核调试，并按 Ctrl+Alt+K 切换到初始中断模式。

启用这种模式后，当 WinDBG 收到调试引擎的信息后，就发送一个中断控制包，即 0x62。清单 18-2 所包含的 WinDBG 输出反映了这一过程。

清单 18-2 WinDBG 发送初始的中断命令

```
SYNCTARGET: Timeout.
Throttle 0x10 write to 0x1
Throttle 0x10 write to 0x1
SYNCTARGET: Received KD_RESET ACK packet.
SYNCTARGET: Target synchronized successfully...
Done.
READ: Wait for type 7 packet
Attempting to get initial breakpoint.
Send Break in ...
```

前 3 行反映了 WinDBG 启动后反复发送复位控制包，企图与调试引擎同步。第 4~6 行表示收到了调试引擎的确认包，同步成功，第 7 行表示等待类型 7 信息包，第 8、9 行表示 WinDBG 为了获得初始中断而发送中断包。

在收到类型 7 信息包后，WinDBG 会显示略微不同的建立连接信息，表示已经发出中断请求：

```
Kernel Debugger connection established. (Initial Breakpoint requested)
```

尽管从以上过程来看这个中断包的发送时机与第一次收到符号事件差不多，但是调试引擎并不会立刻响应这个中断包。

我们知道每次时钟中断时，中断的处理函数会调用 KdPollBreakIn 函数检查是否有中断包。但是在启动的早期（阶段 0）中断是被禁止的。所以时钟更新函数在此阶段还不会被调用。为了检测这一阶段的中断命令，KdInitSystem 在使用 DbgLoadImageSymbols 函数报告好内核文件和 HAL 文件的符号加载事件之后，会调用 KdPollBreakIn 来读取是否有中断包，并且把结果保存在变量 KdBreakAfterSymbolLoad 中：

```
call _KdPollBreakIn@0
mov _KdBreakAfterSymbolLoad, al
```

而后 KdInitSystem 函数返回，启动过程继续，KiSystemStartup 调用 KiInitializeKernel 开始内核初始化。KiInitializeKernel 在执行好 KiInitSystem 和初始化 Idle 进程和线程后，调用 ExpInitializeExecutive 准备初始化执行体。ExpInitializeExecutive 的主要任务除了调用系统中各个执行体的阶段 0 初始化函数外，还有一个任务就是扫描模块列表加载调试符号。这里加载调试符号的含义就是调用 DbgLoadImageSymbols 通知调试器加载符号（对内核而言，很多时候就简称为加载符号文件）。此时模块列表中存放的是 OS Loader 程序所加载的第一批模块文件的基本信息。因为在 KdInitSystem 中已经对内核和 HAL 文件调用过 DbgLoadImageSymbols，所以在这一阶段会跳过这两个文件。

在处理好链表中的所有文件后，系统会判断 KdBreakAfterSymbolLoad 变量，如

果它为真，就会调用 `DbgBreakPointWithStatus` 函数（参数为 1）执行断点指令，中断到调试器，清单 18-3 显示了 Vista 系统中的函数调用情况。

清单 18-3 Vista 检测和发起初始断点的过程

```
# ChildEBP RetAddr
00 81966b44 81f74454 nt!RtlpBreakWithStatusInstruction // 触发断点异常
01 81966cdc 81f00019 nt!InitBootProcessor+0x3d8 // 启动 CPU 要执行的初始化工作
02 81966ce8 81f01d07 nt!ExpInitializeExecutive+0x13 // 执行体的阶段 0 初始化
03 81966d3c 81959321 nt!KiInitializeKernel+0x5cf // 初始化内核
04 00000000 00000000 nt!KiSystemStartup+0x319 // 内核文件的入口函数
```

其中 `InitBootProcessor` 是 Vista 新增的内核函数，将只需要启动 CPU 才执行的阶段 0 初始化工作放在这个函数中，包括扫描模块列表和加载符号文件。在此之前，`ExpInitializeExecutive` 函数直接做这个工作，清单 18-4 显示了 XP 系统中的情况。

清单 18-4 XP 检测和发起初始断点的过程

```
# ChildEBP RetAddr
00 80541d68 8069208b nt!RtlpBreakWithStatusInstruction // 触发断点异常
01 80541ee8 8068624c nt!ExpInitializeExecutive+0x302 // 执行体的阶段 0 初始化
02 80541f3c 80691b23 nt!KiInitializeKernel+0x29e // 初始化内核
03 00000000 00000000 nt!KiSystemStartup+0x2bf // 内核文件的入口函数
```

通过初始断点将目标系统中断到调试器中的时机比收到符号加载事件就中断要晚，二者之间的差异主要有 `KiInitSystem`、`Idle` 进程和线程的初始化及某些执行体的阶段 0 初始化。但是当初始断点发生时，目标系统仍处于执行体的阶段 0 初始化过程中，此时进程管理器仍没有初始化，仍然只有一个 CPU 在执行，耗时最长的 IO 管理器也还没有开始初始化，所有启动类型之外的驱动程序还没有加载，因此，这个时机对于大多数调试任务都是来得及的。

如果目标系统是 Windows 2000，那么初始断点的中断时间会更晚一些。Windows 2000 没有定义 `KdBreakAfterSymbolLoad` 变量和上面所说的检查机制，因此要等执行体的阶段 1 初始化开始，中断被启用后，再有时钟中断发生时 `KeUpdateSystemTime` 得到执行，它调用 `KdPollBreakIn` 时才能发现中断包并发起中断，如清单 18-5 所示。

清单 18-5 Windows 2000 检测和发起初始断点的过程

```
# ChildEBP RetAddr
00 f241b99c 804654be nt!RtlpBreakWithStatusInstruction // 触发断点异常
01 f241b99c 8006f19c nt!KeUpdateSystemTime+0x13e // 时钟中断发生后更新系统时间
02 f241ba28 8006ff15 hal!HalpEnableInterruptHandler+0x34 // 启用中断
03 f241ba54 8054acfa hal!HalInitSystem+0x25f // HAL 的阶段 1 初始化
04 f241bda8 804524f6 nt!Phase1Initialization+0x54 // 执行体的阶段 1 初始化
05 f241bddc 80465b62 nt!PspSystemThreadStartup+0x69 // 线程线程启动函数
06 00000000 00000000 nt!KiThreadStartup+0x16 // 线程起始函数
```

我们在 18.6.4 节介绍了 `KeUpdateSystemTime` 调用 `KdPollBreakIn` 的细节。可以在启动选项中指定/BREAK 选项，让系统在初始化硬件抽象层时中断到内核调试器。

清单 18-6 显示了以这种方式中断后的栈回溯。

清单 18-6 通过/BREAK 启动选项中断到内核调试器

```
kd> kn
# ChildEBP RetAddr
00 80541d48 806d77b4 nt!DbgBreakPoint
01 80541d50 806d78d8 hal!HalpGetParameters+0x3e
02 80541d64 8067f728 hal!HalInitSystem+0x30
03 80541ee8 8068624c nt!ExpInitializeExecutive+0x13c
04 80541f3c 80691b23 nt!KiInitializeKernel+0x29e
05 00000000 00000000 nt!KiSystemStartup+0x2bf
```

与清单 18-4 和清单 18-5 相比较, 可以发现使用/BREAK 选项中断的时间与初始断点基本相同。

18.7.3 断开和重新建立连接

以下几种情况可能需要断开内核调试连接。

- 运行调试器的主机因为某种原因突然重新启动, 比如掉电或崩溃。
- 调试器僵死或一个命令很久没有返回, 按 Ctrl+Break 也无法中断, 而且我们没有耐心继续等待。
- 运行调试器的主机是一台笔记本电脑, 因为参加会议或下班必须将其关机带走。

无论哪种情况, 大多数时候不需要重新启动目标系统就可以再建立内核调试连接。通常只要启动调试器, 使用同样的设置便可以恢复调试会话。根据断开时目标系统的状态, 我们分两种情况来讨论。

第一种情况是断开时目标系统处于被中断到调试器状态。这种情况下, 目标系统只有调试引擎在运行, 内核的其他部分被冻结着。根据我们前面的介绍, 此时 CPU 通常是在执行调试引擎的 KdpSendWaitContinue 函数, 也就是在与调试器的对话循环中。因此这时调试引擎是在等待调试器的信息。而且我们知道, 调试器开始内核调试后就会不停地发送复位控制包。因此这种情况下, 只要接好通信电缆, 并使用上次同样的参数, 在开启调试器后, 调试引擎会马上收到复位包, 并回复确认包, 于是连接建立, 而后 WinDBG 会读取对方的版本信息, 像第一次建立连接那样初始化。而后显示的信息与第一次建立连接时类似, 而且会显示出系统已经运行的时间 (System Uptime) 和上次调试会话的结束时间。

```
Debug session time: Sat Aug 18 10:27:39.386 2007 (GMT+8)
System Uptime: 0 days 0:03:52.964
```

第二种情况是断开时目标系统处于运行状态。这种情况下, 调试引擎处于被动状态, 只有当有内核调试事件发生或时钟中断发生时它的函数才可能被调用。因此如果开启调试器后没有自动建立连接, 那么需要发出一个中断命令, 也就是按热键 Ctrl+Break 或选择菜单中的 Debug > Break。

全局变量 nt!KdDebuggerNotPresent（布尔类型，一字节长）用来标识是否存在内核调试器。它的初始值为 0，也就是假定有调试器存在，在数据通信函数等待调试器的回复多次失败后会将这个变量设置为 1。在通信函数收到调试器的通信包后会将其置为 0。当我们在调试器中观察时，它总为 0：

```
kd> db nt!KdDebuggerNotPresent 11
80544740 00
```

当 KdDebuggerNotPresent 为 1 时，内核调试引擎便不再主动向调试器发送数据。但是时钟更新函数还是会调用 KdCheckForDebugBreak 来检查是否用中断命令。因此，当我们把调试器重新连接到一个曾经调试过的系统（未重启过）时，需要将被调试系统中断一次才开始收到对方的信息。

18.8 本地内核调试

因为使用内核调试引擎进行内核调试需要两个系统，设置和使用需要较多的步骤，所以，如果只希望执行一些观察变量或检查符号之类的简单任务，那么可以使用本节介绍的本地内核调试方法。

18.8.1 LiveKD

LiveKD 是 Mark Russinovich 编写的一个小工具，最初包含在 *Inside Windows 2000* 一书的配套光盘上，目前可以从微软的网站上免费下载。

运行 LiveKD.exe 后，它会先从自身的资源中提取一个名为 LiveKdD.SYS 的驱动程序，然后安装这个驱动程序，并使用它在保持系统工作的情况下产生一个故障转储文件（Dump），名为 LiveKD.DMP，再启动 WinDBG 或 KD 来“调试”这个转储文件。

因为是让调试器调试一个动态的转储文件，所以当使用 LiveKD 进行本地内核调试时，WinDBG 会显示如下信息：

```
Loading Dump File [C:\WINDOWS\system32\livekd.dmp]
Kernel Complete Dump File: Full address space is available
Comment: 'LiveKD live system view'
```

此后，可以使用调试内核转储文件时的各种命令来观察和分析系统。

18.8.2 Windows 自己的本地内核调试支持

从 Windows XP 开始，Windows 操作系统内建了本地内核调试支持。使用 WinDBG 就可以进行本地内核调试（File > Kernel Debug > Local）。

简单来说，Windows 的本地内核调试主要是通过未公开的内核服务 ZwSystemDebug-

Control (内核函数为 NtSystemDebugControl) 来提供的。

ZwSystemDebugControl 是 Windows 操作系统的一个内核服务，在 NT 的最初版本中就存在，其作用是让用户态的程序可以通过这个服务来在本地调用系统的调试 API，即内核调试函数。其函数原型为：

```
NTSTATUS NTAPI ZwSystemDebugControl( DEBUG_CONTROL_CODE ControlCode,
    PVOID InputBuffer, ULONG InputBufferLength,
    PVOID OutputBuffer, ULONG OutputBufferLength, PULONG ReturnLength);
```

其中第一个参数被称为控制代码，或者命令代码，用来指定要调用的内核调试服务。后面几个参数用来指定输入和输出缓冲区和返回值长度 (ReturnLength)。

在 Windows XP 之前，这个内核服务支持的命令非常有限。Windows XP 为了支持本地内核调试，对这个函数的功能做了大量扩充，支持的操作由原来的不到 10 个增加到多达 21 个，覆盖了本地调试所需的几乎一切功能。Windows Server 2003 进一步增强了这个函数。下面的枚举类型定义了用来代表操作类型的所有控制代码。从中我们可以看到，它涵盖了内核调试的各个方面，包括内存、IO、MSR 寄存器、进程列表、模块列表等。

```
typedef enum _SYSDBG_COMMAND
{
    SysDbgQueryModuleInformation = 0, SysDbgQueryTraceInformation = 1,
    SysDbgSetTracepoint = 2, SysDbgSetSpecialCall = 3, SysDbgClearSpecialCalls
    = 4, SysDbgQuerySpecialCalls = 5, SysDbgBreakPoint = 6, SysDbgQueryVersion
    = 7, SysDbgReadVirtual = 8, SysDbgWriteVirtual = 9, SysDbgReadPhysical = 10,
    SysDbgWritePhysical = 11, SysDbgReadControlSpace = 12,
    SysDbgWriteControlSpace = 13, SysDbgReadIoSpace = 14, SysDbgWriteIoSpace = 15,
    SysDbgReadMsr = 16, SysDbgWriteMsr = 17, SysDbgReadBusData = 18,
    SysDbgWriteBusData = 19, SysDbgCheckLowMemory = 20,
    SysDbgEnableKernelDebugger = 21, SysDbgDisableKernelDebugger = 22,
    SysDbgGetAutoKdEnable = 23, SysDbgSetAutoKdEnable = 24,
    SysDbgGetPrintBufferSize = 25, SysDbgSetPrintBufferSize = 26,
    SysDbgGetKdUmExceptionEnable = 27, SysDbgSetKdUmExceptionEnable = 28,
} SYSDBG_COMMAND;
```

WinDBG 的本地内核调试功能就是基于以上系统服务而实现的。可以理解为，ZwSystemDebugControl 为 WinDBG 提供了一种在本地访问内核调试 API 的功能。例如，清单 18-7 描述了在本地内核调试会话中执行!pci 扩展命令时，WinDBG 通过本地内核服务访问 IO 空间的过程。

清单 18-7 本地内核调试中访问 IO 空间的执行过程

```
0:001> kn
# ChildEBP RetAddr
00 00f1d714 0222395f ntdll!NtSystemDebugControl      // 调用内核服务
01 00f1d754 020d87d6 dbgeng!LocalLiveKernelTargetInfo::DebugControl+0xaf
02 00f1d7a0 0213f885 dbgeng!LocalLiveKernelTargetInfo::WriteIo+0x66
03 00f1e0a0 01618c30 dbgeng!ExtIoctl+0x3f5        // 调试引擎的 IOCTL 函数
04 00f1e0c4 01619dfc kext!WriteIoSpace64+0x30       // 包含!pci 命令的扩展模块
05 00f1e110 0161c391 kext!ReadPci+0x18c           // 读 PCI 配置空间
06 00f1e1e8 0161d09a kext!pcidump+0x101            // 显示 PCI 配置空间
07 00f1e248 02144ffa kext!pci+0x15a                // !pci 命令的入口函数
```

```

08 00f1e2d4 02145239 dbgeng!ExtensionInfo::CallA+0x2da      // 调用扩展命令
09 00f1e464 02145302 dbgeng!ExtensionInfo::Call+0x129        //
0a 00f1e480 02143bf1 dbgeng!ExtensionInfo::CallAny+0x72       //
0b 00f1e8f8 021875bc dbgeng!ParseBangCmd+0x661                // 解析扩展命令
0c 00f1e9d8 021889a9 dbgeng!ProcessCommands+0x4ec              // 分发命令
0d 00f1ea1c 020cbe9 dbgeng!ProcessCommandsAndCatch+0x49        //
0e 00f1eeb4 020cc12a dbgeng!Execute+0x2b9                   // 调试引擎的命令入口函数
0f 00f1eee4 01028553 dbgeng!DebugClient::ExecuteWide+0x6a
10 00f1ef8c 01028a43 windbg!ProcessCommand+0x143            // WinDBG 的命令处理函数
11 00f1ffa0 0102ad06 windbg!ProcessEngineCommands+0xa3        // WinDBG 的调试命令处理函数
12 00f1ffb4 7c80b6a3 windbg!EngineLoop+0x366                 // WinDBG 调试循环
13 00f1ffec 00000000 kernel32!BaseThreadStart+0x37           // WinDBG 调试工作线程的起点

```

其中栈帧 0 的前 3 个参数为：0000000f 00f1d778 00000020，第一个是控制码，0xf 对应于枚举常量 `SysDbgWriteIoSpace`，即写 IO 空间。第二个参数指向一个数据结构，第三个参数 0x20 是输入缓冲区的长度，即参数 2 结构的长度。清单中的 `LocalLiveKernelTargetInfo` 是 WinDBG 调试器引擎模块中负责本地内核调试目标类，我们将在 29.5 节详细介绍。

18.8.3 安全问题

尽管调用 `ZwSystemDebugControl` 服务需要所在进程具有调试特权 ('SeDebug-Privilege')，但是，恶意软件也很容易得到这个特权，所以在 Windows XP 推出后不久，这个扩充后的内核服务便成为恶意程序攻击系统的一个捷径。

为了解决这个问题，Windows Vista 的本地内核调试只有在启用了内核调试选项后才能使用。

本地内核调试尽管使用比较方便，但是它只支持有限的调试命令，不支持设置断点、单步执行和将系统中断到调试器这样的高级调试功能。因此，通常只使用这种方法来观察系统的模块、函数、内存、进程和内核对象等。

18.9 本章总结

内核调试是软件调试中比较复杂的部分，比用户态调试的难度要大很多。但是，当开发内核态的模块或解决系统一级的软件问题时，内核调试通常又是最有效的方法。本章比较系统地介绍了 Windows 的内核调试引擎，目的是为大家理解内核调试打下一个坚实的基础。在第 6 篇介绍调试器时，我们还会介绍与内核调试有关的内容，特别是调试器端的更多细节。

参考文献

1. CMU 1394 Digital Camera Driver
<http://www.cs.cmu.edu/~iwan/1394/downloads/index.html>
2. Roger Jennings. Fire on the Wire: The IEEE 1394 High Performance Serial Bus
3. 1394 Open Host Controller Interface Specification. Microsoft Corporation
4. Tom Green. 1394 Kernel Debugging Tips And Tricks
5. Enhanced Host Controller Interface Specification for Universal Serial Bus (Appendix C. Debug Port). Intel Corporation
6. John Keys. USB2 Debug Device A Functional Device Specification. Intel Corporation
7. How to Set Up a Remote Debug Session Using a Modem. Microsoft Corporation,
<http://support.microsoft.com/kb/148954>

Windows 的验证机制

因为软件中的错误是不可避免的，所以，如何尽早地发现和修正错误就成了软件开发中最关键的目标之一。软件调试的主要任务是寻找软件瑕疵（defect）的根源，其前提通常已经是知道了有瑕疵存在。

发现软件瑕疵的最普遍方法就是测试。常见的测试手段有以下几种。

- 黑盒测试（Black box testing），是指测试人员根据软件需求规约和测试文档对软件的运行行为进行检查。其基本思想是将被测试软件当作一个不透明的黑盒子，给其一个输入，看其输出是否符合要求，只要输出结果正确，便认为测试通过，不检查盒子内部的变化过程。
- 白盒测试（White box testing），是指根据程序的结构和代码逻辑来编写测试用例并进行测试。比黑盒测试更有针对性，但是对测试人员的要求更高。
- 内建自检，又称为 BIST（Built-In Self-Test），是指在软件代码内部构建一些测试功能，这些功能（函数）可以在某些情况下执行，或者被自动测试工具所调用以发现问题。
- 压力测试（Stress testing），用于测试目标程序在高负载（如频繁的访问和大量要处理的任务）和低资源（如低可用内存和低硬件配置）情况下的工作情况。

虽然以上每种测试都有它的优势和侧重点，但即使使用了以上所有测试手段，也不能保证会发现所有问题。原因之一是测试时的运行环境和条件不足以将错误触发并暴露出来。举例来说，如果某个程序有轻微的内存泄漏，通常是较难发现的。再比如，如果一个程序调用系统的 API 时参数使用不当，而且它没有检查系统返回了的错误值，于是这个错误就被掩盖了。对于这样的问题，当测试时，我们通常希望系统做严格的检查，发现问题就立刻报告，并且最好能模拟极端的和苛刻的运行环境，以便让错误更容易暴露出来。Windows 操作系统的验证机制（Verifier）就是为了满足这个需求而设计的。

本章将先介绍 Windows 验证机制的概况（19.1 节），而后介绍驱动程序验证机制

的实现原理（19.2 节）和用法（19.3 节），最后介绍应用程序验证的实现原理（19.4 节）和用法（19.5 节）。

19.1 简介

我们知道，从编译和构建（Build）角度来看，Windows 系统的映像文件有 Checked 版本和 Free 版本之分。二者的主要差别就是 Checked 版本中包含断言，而 Free 版本中不包含。尽管断言也是用于检查软件错误的，但其主要的检查目标是软件自身。这与本章介绍的验证机制是不同的。验证机制的主要目标是检查被测试软件，或者说是为被测试软件提供一个验证器（Verifier）。

19.1.1 驱动程序验证器

Windows 2000 最先引入了驱动程序验证器（Driver Verifier），用于验证各种设备驱动程序和内核模块。为了行文简洁，我们将驱动程序验证器简称为驱动验证器。

驱动验证器的主体是实现在内核文件（NTOSKRNL.EXE）中的一系列内核函数和全局变量，其名字中大多都包含 Verifier 字样，或者是以 Vi 和 Vf 开头的。例如用于验证内存池的 nt!VerifierFreePool，用于验证降低 IRQL 操作的 nt!VerifierKeLowerIrql。

为了配合驱动验证器工作，Windows 2000 还自带了一个名为 Driver Verifier Manager 的管理程序，即 Verifier.exe，位于 Windows 系统目录的 system32 子目录中。我们将这个程序简称为驱动验证管理程序。

Windows XP 和 Windows Vista 都对驱动程序验证器做了增强。驱动验证器包含在 Windows 2000 开始的所有 NT 系列操作系统中，不论是 Checked 版本还是 Free 版本。而且内建在系统中，不需要额外安装。我们将在第 19.2 节详细介绍驱动程序验证器。

19.1.2 应用程序验证器

与驱动验证器类似，为了支持应用程序验证，Windows XP 引入了验证应用程序的机制。其实现分为两个部分，一部分是实现在 NTDLL.DLL 中的一系列函数，这些函数都是以 AVrf 开头的，所以，我们将它们简称为 AVrf 函数。另一部分是一个名为应用程序验证器（Microsoft Application Verifier）的工具包，包含在 Windows XP 安装光盘的 support\tools 目录下，也可以从微软网站免费下载。

应用程序验证器最初是为了测试应用程序与 Windows XP 操作系统的兼容性而设计的，其设计思想是通过监视应用程序与操作系统之间的交互来发现应用程序中隐藏的设计问题，比如内存分配、内核对象使用和 API 调用等。我们将在第 19.3 节详细介绍应用程序验证器。

19.1.3 WHQL 测试

通过 WHQL (Windows Hardware Quality Labs) 测试是驱动程序取得 Windows 徽标和得到数字签名的必要条件, 而通过驱动验证器的各种验证是驱动程序 WHQL 测试的必不可少的内容。因此, 一个驱动程序如果想取得 Windows 徽标和签名, 则一定要通过驱动验证器的验证。这种强制性有利于提高内核模块的质量和整个系统的稳定性。

使用验证器有利于更快地发现软件错误。因此, 在软件开发和测试过程中, 我们应该积极使用它, 以便尽早发现程序中的错误和设计缺欠。

19.2 驱动验证器的工作原理

本节我们将比较深入地介绍 Windows 驱动验证器的组成、原理和工作过程。

19.2.1 设计原理

驱动验证器的基本设计思想是在驱动程序调用设备驱动接口 (DDI) 函数时, 对驱动程序执行各种检查, 看其是否符合系统定义的设计要求, 特别是 DDK 文档所定义的调用条件和规范。

那么, 如何来监视驱动程序对 DDI 的调用呢? 一种很容易想到的方法是在每个内核函数的入口加一段代码, 如果驱动验证器被启用, 就调用这个函数对应的验证函数, 否则, 就直接继续执行这个函数。以 KeLowerIrql 为例, 如果它的验证函数是 VerifierKeLowerIrql, 那么应用这种方法后的 KeLowerIrql 函数便是下面的样子:

```
VOID KeLowerIrql( IN KIRQL NewIrql )
{
    if(/*VerifierEnabled*/)
    {
        VerifierKeLowerIrql(NewIrql);
        return;
    }
    // ...
}
```

但是 Windows 并没有采用这种方法, 原因是需要对大量内核函数的实现进行修改, 而且由于加入了判断和分支, 不仅会影响这些函数的简洁, 而且会始终影响这些函数的执行速度。这种修改方式的另一个缺点是它的全局性, 不能针对某个被测试驱动程序决定是否执行验证函数。

因为以上因素, Windows 实际上采用的是通过修改被验证驱动程序的输入地址表 (Import Address Table, 简称 IAT) 来挂接 (Hook) 驱动程序的 DDI 调用, 即通常所说的 IAT Hook 方法。简单来说, 系统会将被验证驱动程序 IAT 表中的 DDI 函数地址替换为验证函数的地址。这样, 当这个驱动程序调用 DDI 函数时, 便会调用对应的验证函数。

验证函数与原来的函数具有完全一致的函数原型，所以不会影响被验证程序的执行。

在验证函数得到调用后，它执行的典型操作如下。

- 更新计数器，或者全局变量。
- 检测调用参数，或者做其他检查，如果检测到异常情况，那么调用 KeBugCheckEx (DRIVER_VERIFIER_DETECTED_VIOLATION, ...) 函数，即通过蓝屏 (Bug Check) 机制来报告验证失败。
- 如果没有发现问题，那么验证函数会调用原来的函数，并返回原函数的返回值（如果有）。

下面我们分几个部分详细介绍驱动验证器的工作过程。

19.2.2 初始话

在 Windows 系统启动的早期，确切地说是在执行体进行阶段 0 初始化期间，内存管理器的初始化函数 (MmInitSystem) 会调用驱动验证器的初始化函数，来初始化驱动验证器。其中重要的工作就是创建并初始化一个用来存放被验证驱动程序信息的链表，Windows 使用全局变量 MiSuspectDriverList 来记录这个链表，因此，我们便把这个链表简称为可疑驱动链表。

以下栈回溯显示了内存管理器的 MmInitSystem 函数调用初始化可疑驱动链表的 MiInitializeDriverVerifierList 函数的过程：

```
# ChildEBP RetAddr
00 80541d0c 80683000 nt!MiInitializeDriverVerifierList // 初始化可疑驱动链表
01 80541d64 8067f84e nt!MmInitSystem+0x8e6 // 内存管理器阶段 0 初始化
02 80541ee8 8068624c nt!ExpInitializeExecutive+0x272 // 执行体阶段 0 初始化
03 80541f3c 80691b23 nt!KiInitializeKernel+0x29e // 内核初始化
04 00000000 00000000 nt!KiSystemStartup+0x2bf // 系统的入口函数
```

可疑驱动链表的每个节点是一个 MI_VERIFIER_DRIVER_ENTRY 结构，用来记录一个被验证的驱动程序，以下是这个结构的定义和关于被验证驱动程序 dbgmsg.sys 的取值：

```
Kd> dt nt!_MI_VERIFIER_DRIVER_ENTRY 82374c78
+0x000 Links : _LIST_ENTRY [ 0x8054c128 - 0x82374cf8 ]
+0x008 Loads : 1 // 加载计数
+0x00c Unloads : 0 // 卸载计数
+0x010 BaseName : _UNICODE_STRING "dbgmsg.sys" // 名称
+0x018 StartAddress : 0xf53fd000 // 起始内存地址
+0x01c EndAddress : 0xf5401000 // 结束内存地址
+0x020 Flags : 1 // 标记，1 代表直接方式
+0x024 Signature : 0x98761940 // 这个结构的签名，始终为这个值
+0x028 Reserved : 0 // 保留
+0x02c VerifierPoolLock : 0 // 用做同步，ExAcquireSpinLock
+0x030 PoolHash : 0x821898c0 _VI_POOL_ENTRY // 记录内存池使用情况
+0x034 PoolHashSize : 0x10 //
+0x038 PoolHashFree : 0xe //
+0x03c PoolHashReserved : 0 //
```

```
+0x040 CurrentPagedPoolAllocations : 5 // 在分页内存池中的分配数
+0x044 CurrentNonPagedPoolAllocations : 9 // 在非分页内存池中的分配数
+0x048 PeakPagedPoolAllocations : 6 // 分页内存分配数峰值
+0x04c PeakNonPagedPoolAllocations : 9 // 非分页内存分配数峰值
+0x050 PagedBytes : 0x100 // 分配的分页内存数量(字节数)
+0x054 NonPagedBytes : 0x412c // 分配的非分页内存数量
+0x058 PeakPagedBytes : 0x17c // 分配的分页内存数量峰值
+0x05c PeakNonPagedBytes : 0x412c // 分配的非分页内存数量峰值
```

Windows Vista (RTM 6000) 对以上结构做了少量修改，新的结构为：

```
kd> dt nt!_MI_VERIFIER_DRIVER_ENTRY 83dc95f0
+0x000 Links : _LIST_ENTRY [ 0x81b7d610 - 0x81b7d610 ]
+0x008 Loads : 1 // 加载计数
+0x00c Unloads : 0 // 卸载计数
+0x010 BaseName : _UNICODE_STRING "mrxxvpc.sys"
+0x018 StartAddress : 0xa161b000 // 起始内存地址
+0x01c EndAddress : 0xa163a000 // 结束内存地址
+0x020 Flags : 1 // 标志
+0x024 Signature : 0x98761940 // 结构签名
+0x028 PoolPageHeaders : _SLIST_HEADER // 记录使用内存池情况的链表表头
+0x030 PoolTrackers : _SLIST_HEADER //
+0x038 CurrentPagedPoolAllocations : 0x20 // 目前在分页内存池中的分配数
+0x03c CurrentNonPagedPoolAllocations : 0x1f // 目前在非分页内存池中的分配数
+0x040 PeakPagedPoolAllocations : 0x22 // 分页内存分配数峰值
+0x044 PeakNonPagedPoolAllocations : 0x21 // 非分页内存分配数峰值
+0x048 PagedBytes : 0x278b4 // 分配的分页内存数量(字节数)
+0x04c NonPagedBytes : 0x5454 // 分配的非分页内存数量
+0x050 PeakPagedBytes : 0x284f4 // 分配的分页内存数量峰值
+0x054 PeakNonPagedBytes : 0x5ae4 // 分配的非分页内存数量峰值
```

函数 ViInsertVerifierEntry 用于向可疑驱动链表中插入表项。其参数就是一个指向 MI_VERIFIER_DRIVER_ENTRY 结构的指针。

当插入表项时，MI_VERIFIER_DRIVER_ENTRY 结构的某些字段已经填充内容了，但是某些还没有。

在可疑链表初始化后，可以在调试器中通过 !list 命令来显示这个链表的内容：

```
kd> !list "-t _MI_VERIFIER_DRIVER_ENTRY.Links.Flink -e -x \'dd @$extret 14; dt _MI_VERIFIER_DRIVER_ENTRY @$extret -y BaseName\'" poi(nt!MiSuspectDriverList)"
dd @$extret 14; dt _MI_VERIFIER_DRIVER_ENTRY @$extret -y BaseName
82374c78 8054c128 82374cf8 00000001 00000000
nt!_MI_VERIFIER_DRIVER_ENTRY
+0x010 BaseName : _UNICODE_STRING "dbgmsg.sys"
...
```

关于 !list 命令的详细用法，我们将在第 30 章（30.16 节）讲述。

系统启动完成后，也可以在命令行执行 verifier/query 命令来显示被验证驱动的信息，其主要内容就是来自以上链表。

19.2.3 挂接验证函数

前面介绍了用来记录所有被验证驱动的可疑驱动列表，接下来我们将介绍系统是

如何将验证函数挂接到被验证驱动程序的 IAT 表中的。简单来说，当系统加载一个内核模块时，它会调用 MiApplyDriverVerifier 函数，这个函数的任务就是查询要加载的模块是否在可疑驱动列表中，如果在，则表示这是一个被验证的驱动，并调用 MiEnableVerifier 函数，对它的 IAT 表进行修改。

对于 OS Loader (NTLDR) 加载的模块，当驱动验证器初始化时，也就是在内存管理器的阶段 0 初始化期间，驱动验证器的 MiInitializeVerifyingComponents 函数会遍历所有已经加载的模块，并依次对其调用 MiApplyDriverVerifier，清单 19-1 中的栈回溯显示了这个函数被调用的过程。

清单 19-1 对 OS Loader 加载的驱动程序应用验证机制的过程

# ChildEBP RetAddr		
00 80541cb8	805d8a84 nt!MiEnableVerifier	// 挂接验证函数
01 80541cd0	80694746 nt!MiApplyDriverVerifier+0x128	// 应用驱动验证逻辑
02 80541d0c	80683101 nt!MiInitializeVerifyingComponents+0x239	// 初始化验证器
03 80541d64	8067f84e nt!MmInitSystem+0xad0	// 内存管理器阶段 0 初始化
04 80541ee8	8068624c nt!ExpInitializeExecutive+0x272	// 执行体阶段 0 初始化
05 80541f3c	80691b23 nt!KiInitializeKernel+0x29e	// 初始化内核
06 00000000	00000000 nt!KiSystemStartup+0x2bf	// 系统的入口函数

对于以后加载的驱动程序，内存管理器的工作函数 (MiLoadSystemImage) 会调用 MiApplyDriverVerifier 以给驱动验证器以检查机会。例如，清单 19-2 显示了阶段 1 初始化期间，IO 管理器调用 MiApplyDriverVerifier 和 MiEnableVerifier 的过程。

清单 19-2 对 IO 管理器加载的驱动程序应用验证机制的过程

ChildEBP RetAddr		
ec41b250	804f02b0 nt!MiEnableVerifier	// 挂接验证函数
ec41b268	804a5fe4 nt!MiApplyDriverVerifier+0x128	// 应用驱动验证逻辑
ec41b4dc	8048fcda nt!MiLoadSystemImage+0x722	// 加载系统映像文件
ec41b500	804a4218 nt!MmLoadSystemImage+0x1c	// 加载系统映像文件
ec41b5d8	80426f6d nt!IopLoadDriver+0x3a3	// 加载驱动程序
..		// 省略多个栈帧
ec41ba58	8054b35a nt!IoInitSystem+0x644	// IO 子系统初始化
ec41bda8	804524f6 nt!Phase1Initialization+0x71b	// 阶段 1 初始化
ec41bddc	80465b62 nt!PspSystemThreadStartup+0x69	// 线程启动函数
00000000	00000000 nt!KiThreadStartup+0x16	// 线程起始函数

那么，MiEnableVerifier 函数是如何修改驱动 IAT 表的呢？

简单来说，系统定义了几个包含被验证函数和验证函数的数组，记录在 MixxxThunks 这样的全局变量中，称为 Thunk 数组。下面我们以其中名为 MiVerifierThunks 的一个为例。

```
kd> dd nt!MiVerifierThunks
80636708 804ebf17 80647ed0 805128bc 8064815f
..
```

使用 dds 命令显示每个数组元素的对应符号：

```
kd> dds nt!MiVerifierThunks
80636708 804ebf17 nt!KeSetEvent
```

```

8063670c 80647ed0 nt!VerifierSetEvent
80636710 805128bc nt!ExAcquireFastMutexUnsafe
80636714 8064815f nt!VerifierExAcquireFastMutexUnsafe
...

```

可以看到，数组双号元素的内容就是验证函数的地址，单号元素就是被验证函数的地址。理解了这个数组后，就可以想象得到，MiEnableVerifier 函数就是先根据参数中指定的_LDR_DATA_TABLE_ENTRY 结构指针，得到 IAT 表的地址，然后依次遍历 IAT 表的每个表项，看其函数地址是否存在于上面的 Thunk 数组中（单号元素）的，如果存在，就使用相邻的双号元素的地址将原来的地址替换掉。

19.2.4 验证函数的执行过程

成功挂接验证函数后，当被验证驱动程序再调用被挂接的内核函数时，对应的验证函数就会被调用。例如，VerifierAllocatePoolWithTag 是 ExAllocatePoolWithTag 函数的验证函数，当驱动程序调用 ExAllocatePoolWithTag 时，VerifierAllocatePoolWithTag 便会被调用。下面以 VerifierAllocatePoolWithTag 为例介绍验证函数的执行过程。

VerifierAllocatePoolWithTag 在建立好自己的栈帧（`mov ebp,esp`）后，便调用 `RtlGetCallersAddress` 从栈上读取本函数的返回地址，这个地址也就是被验证驱动程序发起调用验证函数的地址（`CALL` 指令的下一条指令）。接下来，VerifierAllocatePoolWithTag 将得到的父函数地址作为参数调用 `ViLocateVerifierEntry` 函数，后者会在可疑驱动链表中根据每个驱动的起始地址（`StartAddress`）和结束地址（`EndAddress`）查找被验证驱动程序所对应的 MI_VERIFIER_DRIVER_ENTRY 结构，如果找到，则返回结构的地址，否则返回 `NULL`。

如果 `ViLocateVerifierEntry` 返回 `NULL`，则表明情况异常，VerifierAllocatePoolWithTag 会中止验证，调用原来的被验证函数 `ExAllocatePoolWithTag`。否则会调用 `VeAllocatePoolWithTagPriority`，执行验证动作，包括：

- 对调用 `ExAllocatePoolSanityChecks` 进行检查。
- 调用 `ViInjectResourceFailure` 函数判断是否模拟分配失败，如果是，则模拟资源不足，返回 `NULL` 表示分配失败。如果允许内存池的选项设置中指定分配失败时抛出异常，那么在返回 `NULL` 前会调用 `ExRaiseStatus(STATUS_INSUFFICIENT_RESOURCES)` 发起异常。
- 调用 `ExAllocatePoolWithTagPriority` 实际分配内存。
- 调用 `ViPostPoolAllocation` 对分配情况进行记录，如利用 `ViInsertPoolAllocation` 将本次分配记录到 MI_VERIFIER_DRIVER_ENTRY 结构的 `PoolHash` 字段指向的哈希表中。

最后返回 ExAllocatePoolWithTagPriority 函数。以上所描述的函数名和具体过程是以 Windows XP SP1 为例的。

19.2.5 报告验证失败

在驱动验证器的验证函数检测到错误情况后，它会通过蓝屏机制（BSOD）进行报告，它通常是调用 KeBugCheckEx 函数，并使用以下停止码（Stop Code）作为标识。

- DRIVER_VERIFIER_IOMANAGER_VIOLATION (0xC9)，关于 IO 的验证错误。
- DRIVER_VERIFIER_DMA_VIOLATION (0xE6)，DMA，关于 DMA 的验证错误，Windows XP 引入。
- SCSI_VERIFIER_DETECTED_VIOLATION (0xF1)，关于 SCSI 的验证错误，Windows XP 引入。
- DRIVER_VERIFIER_DETECTED_VIOLATION (0xC4)，其他验证错误。

停止码后的第一个参数表示该类错误中的子错误类型，其他参数用来进一步说明错误的详细情况。例如，图 19-1 显示的是一个驱动验证器报告的蓝屏错误，停止码为 0xC4，停止码后的第一个参数是子错误代码，0 表示被验证程序试图分配 0 字节的内存，第二个参数是当时的 IRQL，第三个参数是要分配的内存池类型（Pool Type），0 代表 NonPagedPool。第四个参数未用为 0。



图 19-1 通过蓝屏报告验证失败

清单 19-3 中的栈回溯描述了被验证的驱动程序 verifier.sys 调用内存分配函数和验证器检测到错误后发起蓝屏的过程。

清单 19-3 验证内存分配和报告验证失败的执行过程

# ChildEBP RetAddr	
00 f890a124 805258ca	nt!RtlpBreakWithStatusInstruction // 触发中断异常
01 f890a170 805266aa	nt!KiBugCheckDebugBreak+0x19 // 向内核调试器汇报
02 f890a534 805266db	nt!KeBugCheck2+0x9b7 // 蓝屏处理函数
03 f890a554 80650e5a	nt!KeBugCheckEx+0x19 // 发起 Bug Check
04 f890a574 806489c5	nt!ExAllocatePoolSanityChecks+0x5a // 分配检查
05 f890a594 80649354	nt!VeAllocatePoolWithTagPriority+0x14 // 验证逻辑的入口函数
06 f890a5b8 f8903e4b	nt!VerifierAllocatePoolWithTag+0x44 // 验证函数
07 f890a600 8067dcbb	verifier+0xe4b // 被验证的驱动程序
...	// 以下省略

栈帧#0 和#1 表明蓝屏处理函数在绘制蓝屏后尝试中断到内核调试器。

在 WinDBG 的帮助文件中列出了关于 0x000000C4 停止码的说明及其他验证失败时蓝屏参数的含义。

19.3 使用驱动验证器

上一节我们介绍了驱动验证器的工作原理，本节将简要介绍它的用法，特别是如何使用它来辅助测试和调试。

19.3.1 验证项目

驱动验证器将要验证的功能分成若干个项目。Windows 2000 引入的驱动验证器包含了一些基本的验证项目，包括。

- **自动检查**，这是唯一一个不可以单独禁止和取消的项目。一旦一个驱动程序被加入到被验证列表中，那么系统便会对它自动进行本项目所定义的各种检查。包括是否在合适的 IRQL 级别使用内存，是否有不恰当的切换栈，是否释放包含活动计时器(Timer)的内存池，是否在合适的 IRQL 级别获取和释放自旋锁(spin lock)。此外，当驱动程序卸载时，系统也会检测它是否已经释放了所有资源。
- **特殊内存池**，从特殊的内存池来为驱动程序分配内存。系统对这个内存池具有增强的监视功能，可以发现上溢(overrun)、下溢(underrun)和释放后又访问等错误情况。
- **强制的 IRQL 检查**，强制让驱动程序使用的分页内存失效，并监视驱动程序是否在错误的 IRQL 级别访问分页内存或持有自旋锁。
- **低资源模拟**，对驱动程序的内存分配请求或其他资源请求，随机地返回失败。
- **内存池追踪(Pool Tracking)**，记录驱动程序分配的内存，释放时看其是否全部释放。
- **I/O 验证**，从特殊内存池分配 IRP(I/O Request Packet)，并监视驱动程序 I/O 的处理。

Windows XP 引入了一些新的验证项目，包括。

- **死锁探测**，监视驱动程序对各种同步资源(自旋锁、互斥量和快速互斥量等)的使用情况，如果发现可能导致死锁的情况，则报告验证失败。
- **增强的 I/O 验证**，监视驱动程序对 I/O 例程的调用，并对 PnP、电源和 WMI 有关的 IRP 进行压力测试。
- **SCSI 验证**，监视 SCSI 小端口驱动程序(SCSI miniport driver)，如果发现它不恰当地使用 SCSI 端口例程，过分地延迟，或者处理 SCSI 请求不当，则报告验证失败。

Windows Server 2003 引入了。

- **IRP 记录 (IRP Logging)**，监视并记录驱动程序使用 IRP 的情况。

Windows Vista 引入了：

- 驱动滞留探测(Driver Hang Detection)，监视驱动程序的 I/O 完成例程(Completion Routine) 和取消例程(Cancellation Routine) 的执行时间，如果超出限制，则报告验证失败。
- 安全检查，寻找可能威胁安全的一般错误，比如内核态的函数引用用户态地址等。
- 强制 I/O 请求等待解决(Pending I/O Requests)，对驱动程序的 IoCallDriver 调用随机返回 STATUS_PENDING，看其是否能正确处理这种需要等待的情况。
- 零散检查(Miscellaneous Checks)，检查可能导致驱动程序崩溃的典型诱因，比如错误的处理已经释放的内存等。

19.3.2 启用驱动验证

尽管驱动验证器是内建在 Windows 系统中的，但是默认它是处于禁用状态的。因此，在使用前需要首先使用驱动验证管理器程序(Verifier.exe)来启用它。

驱动验证管理器有图形界面和命令行两种工作方式，在运行对话框中输入 Verifier 并点击 OK 按钮就可以启动它的图形界面。在 Windows 2000 中，其界面是包含多个页表(Tab)的对话框，Windows XP 版本改为向导方式。无论是哪种界面，其实质无非是选择要验证的驱动程序和验证时要做的验证项目。图 19-2 显示了 Windows XP 版本的验证管理器的“显示当前设置”界面。右侧显示了目前已经选择的被验证驱动程序，左侧是要执行的验证测试，Yes 表示启用了这项测试，No 表示没有启用。

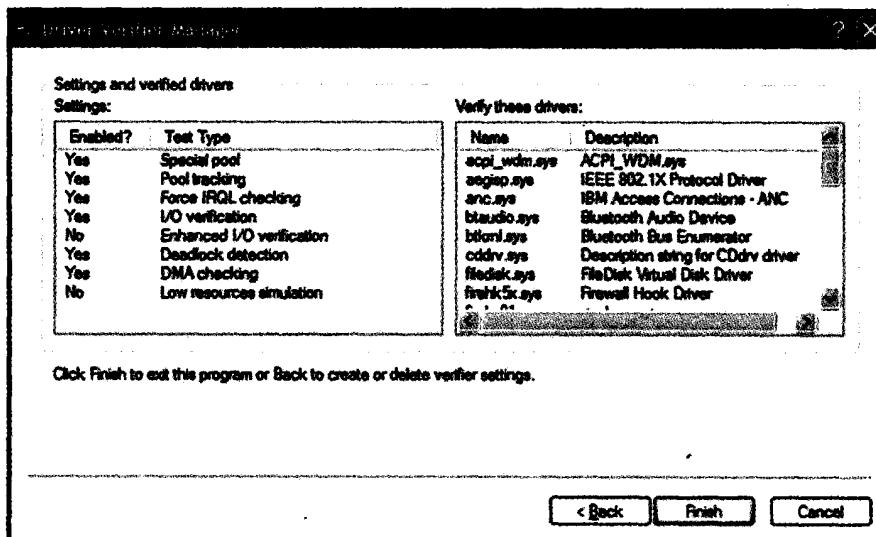


图 19-2 Windows XP 版本的验证管理器的“显示当前设置”界面

启动一个命令行窗口，然后输入 verifier /? 可以看到一个关于如何通过命令行方式使用验证管理器的简单帮助。例如可以通过以下方式来增加被验证驱动程序：

```
verifier [ /flags FLAGS ] /driver NAME [NAME ...]
```

其中 Name 是被验证驱动程序的名称，FLAGS 是一个整数，每一位代表一个要验证的项目，其位定义如下。

- bit 0: special pool checking, 即内存池检查。
- bit 1: force irql checking, 即强制 IRQL 检查。
- bit 2: low resources simulation, 即低资源模拟。
- bit 3: pool tracking, 即内存池记录。
- bit 4: I/O verification, 即 I/O 验证。
- bit 5: deadlock detection, 即死锁探测。
- bit 6: enhanced I/O verification, 即增强的 I/O 验证。
- bit 7: DMA verification, 即 DMA 验证。
- bit 8: security checks, 即安全检查。
- bit 9: force pending I/O request, 即强制的 IO 请求等待。
- bit 10: IRP logging, IRP 记录。
- bit 11: miscellaneous checks, 零散检查。

其中 bit 8~11 需要 Windows Vista 或更高版本的 Windows 才支持。

无论是图形界面还是命令行方式，验证管理器都是将要验证的驱动程序和验证项目记录在如下注册表表键中（见图 19-3）。

HLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management

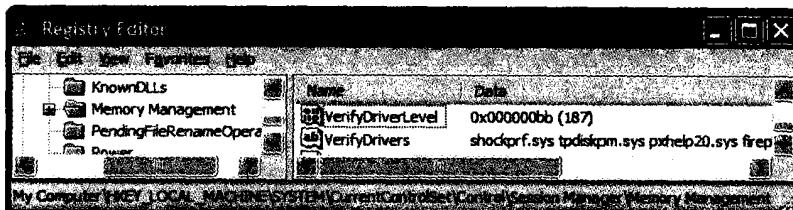


图 19-3 用来记录要验证驱动程序和验证项目的注册表表键

其中 VerifyDirverLevel 为 DWORD 类型，用来记录验证项目，目前的值 0xBB 翻译为二进制是 0b10111011，即 0、1、3、4、5 和 7 位为 1，也就是启用了内存池检查、IRQL 检查、内存池记录、I/O 验证、死锁探测和 DMA 验证，与上面图形界面中显示的结果是一致的。VerifyDrivers 为 Multi-String 类型，用来记录要验证的驱动程序名。

19.3.3 开始验证

在 Windows 2000 和 XP 中，启用了驱动验证后，需要重新启动系统，设置才会生效，因为系统只有在启动期间才会初始化可疑驱动链表。Windows Vista 对驱动验证器

做了增强，可以在增加验证驱动程序后，不重启系统便可以使验证生效。

在系统启动过程中，如果有内核调试器存在，那么可以看到驱动验证器所输出的信息：

```
*****
*
* This is the string you add to your checkin description
* Driver Verifier: Enabled for MRxVPC.sys on Build 6000 r3gQeWTUpmjwDLIQ4zNm1B
*
*****
```

无论是哪一种方法，验证过程都是被动的，或者说是在被验证驱动调用系统的函数时，验证逻辑才会真正执行。所以在验证一个驱动程序时应该使用它。并且让其中尽可能多的执行路径都得到执行。

正如上一节所介绍的，在验证器检测到错误情况后，它会通过蓝屏进行报告。反之，如果没有出现蓝屏，那么说明还没有检测到错误情况。在蓝屏发生前，可以通过下面介绍的方法来了解验证情况。

19.3.4 观察验证情况

驱动验证器运行过程中会将很多信息记录下来，比如关于每个驱动程序的信息会被记录到可疑链表中这个驱动程序对应的 MI_VERIFIER_DRIVER_ENTRY 结构中。此外，全局信息会被记录到全局变量 MmVerifierData 中。

MmVerifierData 的类型为 MM_DRIVER_VERIFIER_DATA，在 WinDBG 中，可以看到这个结构的定义。清单 19-4 显示了 Windows XP 中的结构定义和在某一时刻的取值。

清单 19-4 观察 MM_DRIVER_VERIFIER_DATA 结构和取值

```
kd> x nt!MmVerifierData
8054c140 nt!MmVerifierData = <no type information>
kd> dt nt!_MM_DRIVER_VERIFIER_DATA 8054c140
+0x000 Level : 0xbb // 验证项目
+0x004 RaiseIrqls : 2 // 对 KeRaiseIrql 的调用次数
+0x008 AcquireSpinLocks : 7 // 获得自旋锁的次数
+0x00c SynchronizeExecutions : 0 // 同步的总次数
+0x010 AllocationsAttempted : 0x18 // 请求内存分配的次数
+0x014 AllocationsSucceeded : 0x18 // 内存分配成功的次数
+0x018 AllocationsSucceededSpecialPool : 0x18 // 从特殊内存池分配成功的次数
+0x01c AllocationsWithNoTag : 0 // 不到标记的分配次数
+0x020 TrimRequests : 9 // 请求修剪(trim)系统内存的次数
+0x024 Trims : 6 // 实际修剪系统内存(即 Unmap)的次数
+0x028 AllocationsFailed : 0 // 分配失败计数
+0x02c AllocationsFailedDeliberately : 0 // 故意的分配失败计数
+0x030 Loads : 5 // 加载次数
+0x034 Unloads : 0 // 卸载次数
+0x038 UnTrackedPool : 0 // 没有追踪的内存次数
+0x03c UserTrims : 0 // 整理用户态内存的次数
+0x040 CurrentPagedPoolAllocations : 5 // 目前在分页内存池中的分配数
```

```
+0x044 CurrentNonPagedPoolAllocations : 0xc // 目前在非分页内存池中的分配数
+0x048 PeakPagedPoolAllocations : 6 // 分配分页内存的累计次数
+0x04c PeakNonPagedPoolAllocations : 0xc // 分配非分页内存的累计次数
+0x050 PagedBytes : 0x100 // 分配的分页内存字节数
+0x054 NonPagedBytes : 0x7168 // 分配的非分页内存字节数
+0x058 PeakPagedBytes : 0x17c // 累计分配的分页内存字节数
+0x05c PeakNonPagedBytes : 0x7168 // 累计分配的非分页内存字节数
+0x060 BurstAllocationsFailedDeliberately : 0 // 故意失败的突发性内存分配次数
+0x064 SessionTrims : 0 // 会话修剪(Trim)次数
+0x068 Reserved : [2] 0 // 保留
```

需要说明的是，以上统计计数都是对于所有被验证驱动程序的累计值。

除了使用 WinDBG 来观察验证状态外，也可以使用验证管理器程序（verifier.exe）来查询验证情况。例如以下是在被调试的 Windows XP 系统中通过 verifier.exe 得到的结果：

```
C:\>verifier /query // 以命令行方式查询验证情况
9/16/2007, 11:24:15 AM
Level: 000000BB // 验证项目
RaiseIrqls: 2 // 对 KeRaiseIrql 的调用次数
...
Verified drivers: // 以下是关于每个被验证驱动程序的信息
Name: verifiee.sys, loads: 1, unloads: 0 // 驱动程序文件名, 加载和卸载次数
CurrentPagedPoolAllocations: 5 // 在分页内存池中的分配次数
...
PeakNonPagedPoolUsageInBytes: 16684 // 省略多行
...
//省略关于其他驱动的显示
```

容易看出，信息的前半部分就是 MmVerifierData 的值，后半部分是可疑驱动列表中关于每个驱动程序的信息。事实上，验证管理器是调用 NtQuerySystemInformation 系统服务，后者又调用 MmGetVerifierInformation 将 MmVerifierData 结构复制给参数指定的缓冲区。

19.3.5 WinDBG 的扩展命令

为了更方便地在调试器中观察驱动验证信息，了解驱动验证器收集到的驱动程序状态，WinDBG 工具包中设计了专门的扩展命令，名称是!verifier。有 3 个扩展 DLL 提供了这个命令，分别是 kdexts.dll、kdextx86.dll 和 gdikdx.dll。可以使用![DLL 名称] verifier 的方式执行其中的某一个，比如，!gdikdx.verifier 是执行 gdikdx.dll 中的 verifier 命令，它专门用于显示类驱动程序。另两个用于其他驱动程序。清单 19-5 给出了!verifier 命令的典型执行结果（目标系统是 Windows Vista）。

清单 19-5 使用 WinDBG 扩展命令观察验证信息

```
kd> !verifier 1
Verify Level 9bb ... enabled options are: [启用的验证项目]
Special pool [...省略其他项]
Summary of All Verifier Statistics [以下是统计数据]
```

RaiseIrqls	0x0
AcquireSpinLocks	0x8
[...省略多行]	
Peak nonpaged pool allocations	0x4 for 0000448C bytes
Driver Verification List	[以下是被验证的驱动程序列表]
Entry State	NonPagedPool PagedPool Module
83dc95f0 Loaded	00004144 00012000 mrxvpc.sys

改变命令的参数，可以得到更多其他信息，但由于篇幅所限，我们不再一一介绍，读者使用时可以在 WinDBG 中输入!Verifier /?获得帮助，或者查看 WinDBG 关于这个命令的帮助信息。

19.4 应用程序验证器的工作原理

前面两节我们介绍了用于验证 Windows 系统中驱动程序（内核模块）的驱动验证器，本节和下一节将介绍用于验证 Windows 应用程序的应用程序验证器（Application Verifier），我们将其简称为应用验证器。

19.4.1 原理和组成

与驱动验证器类似，应用验证器的设计原理也是通过挂接应用程序模块的 IAT 表来截取应用程序对编程接口（API）的调用，然后验证它是否符合 Windows SDK 所定义的设计规范。

概括来说，应用验证器由以下 3 个部分组成。

位于 NTDLL 中的支持例程，这些函数大多以 AVrf 开头，位于系统的 NTDLL.DLL 模块中，我们将它们统称为 AVrf 函数。我们知道，NTDLL.DLL 是 Windows 系统中具有非常特殊意义的一个 DLL 模块，它是所有 Windows 程序执行时都依赖的一个模块，它会被映射到所有 Windows 进程之中。因此，AVrf 系统函数是应用验证器进入到被验证程序的登录器和事件探测器，位于 NTDLL 中的进程初始化函数和模块加载函数在进程初始化期间会调用 AVrf 系统函数，让应用验证器得到执行机会。AVrf 函数会检测当前程序是否启用了验证选项，如果是，则加载其他验证模块并作初始化（稍后讨论）。当应用程序加载其他模块时，NTDLL 中的系统函数也会调用 AVrf 函数，使其获得通知。

验证提供器模块，是安装应用程序验证工具时复制到系统 system32 目录下的多个 DLL 文件，每个文件为一个动态链接库（DLL），用于完成某一方面的验证任务。每个提供器都包含若干个验证函数和 Thunk 表，前者用来替代系统 API，后者用来描述要挂接的函数信息。目前版本（3.4）的应用验证工具包包含了如下几个提供器模块：Verifier.dll（基础模块）、VRFCore.dll（基础模块）、VFBasics.dll（基本验证项目）、VFCompat.dll（兼容性验证项目）、VFLuaPriv.dll（用户账号）、VFPrint.dll（打印 API）、VFPrintPthelper.dll（打印驱动）、VFSvc.dll（系统服务）。

应用验证管理器，用于管理（添加、删除）被验证的应用程序和选择验证项目的工具，也是应用程序验证工具包时需安装的，位于 system32 目录下，文件名为 appverif.exe，可以以图形界面工作，也可以以命令行方式工作。我们将在下一节介绍它的用法。

下面我们将分几个部分介绍应用验证器的工作过程，以进一步理解应用验证器的工作原理。在以下分析中，我们以验证一个 MFC 小程序 Verifiee.exe 为例。

19.4.2 初始化

当系统执行一个 Windows 程序时，系统完成内核态的进程创建工作后，会在新进程中开始加载程序所依赖的动态链接库模块。负责加载工作的部分通常被称为加载器（Loader），实际上就是位于 NTDLL.DLL 中的一系列以 Ldr 开头的函数，如 LdrLoadDll、LdrpMapDll 和 LdrInitializeThunk 等。当加载器开始工作时，进程中已经有两个模块，一个是可执行程序（EXE），另一个是 NTDLL.DLL。

```
0:000> lm
start      end        module name
00400000  0041b000  verifiee  (deferred)
7c900000  7c9b0000  ntdll    (pdb symbols)
d:\symbols\ntdll.pdb\...\ntdll.pdb
```

当加载器初始化时，它会从以下注册表键下寻找当前程序的执行选项：

```
HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Image File Execution Options\<程序名>
```

关于一个程序的应用验证选项也是保存在这个表键下的。图 19-4 显示的是注册表中 verifiee.exe 的执行选项。其中 VerifierDlIs 和 VerifierFlags 便是应用验证设置。事实上，当我们使用应用验证管理器对 verifiee.exe 启用验证时，这个工具执行的主要动作就是将我们选择的验证设置保存到注册表中。

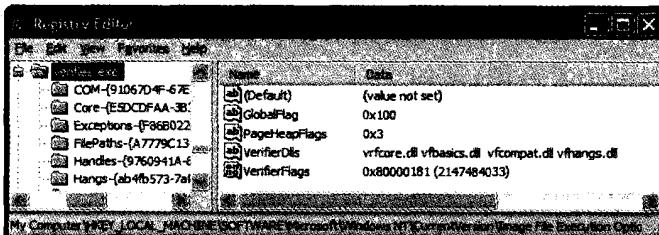


图 19-4 保存在注册表中的应用验证设置

当加载器初始化执行选项时，它会判断程序的全局标志（GlobalFlag），如果它的位 8（0x100）为 1，那么表示这个程序启用了应用验证，于是加载器就会调用验证器的初始化函数 LdrpInitializeApplicationVerifierPackage，后者又会调用 AVrf 函数 AVrfInitializeVerifier。清单 19-6 显示了实际的调用过程。

清单 19-6 加载器调用应用验证器的初始化函数

```
0:000> kn
# ChildEBP RetAddr
00 0012fc20 7c951779 ntdll!AVrfInitializeVerifier //初始化应用验证器
01 0012fc3c 7c93df07 ntdll!LdrpInitializeApplicationVerifierPackage+0x35
02 0012fc00 7c922d2d ntdll!LdrpInitializeExecutionOptions+0xfe //初始化执行选项
03 0012fd1c 7c90eac7 ntdll!_LdrpInitialize+0x60 //加载器初始化
04 00000000 00000000 ntdll!KiUserApcDispatcher+0x7 //用户态 APC 分发函数
```

AVrfInitializeVerifier 被调用后，会做一些基本的初始化工作，包括调用 AVrfpEnableVerifierOptions 判断用户所启用的验证项目等。

接下来，加载器会开始针对当前进程的初始化工作，也就是调用 LdrpInitializeProcess 函数。LdrpInitializeProcess 函数是个比较复杂的函数，它执行的任务包括建立当前进程的加载器数据表（Ldr Data Table），加载当前 EXE 静态链接（依赖）的 DLL，创建堆等。我们熟悉的初始断点也是在这个函数接近退出前发起的（调用 DbgBreakPoint）。

在建立了加载器数据表之后，在加载静态依赖的 DLL 之前，LdrpInitializeProcess 会调用 AVrfInitializeVerifier 函数，如清单 19-7 所示。

清单 19-7 LdrpInitializeProcess 函数调用应用验证器的初始化函数

```
0:000> kn
# ChildEBP RetAddr
00 0012fb10 7c93eb7b ntdll!AVrfInitializeVerifier //应用验证器初始化函数
01 0012fc94 7c921639 ntdll!LdrpInitializeProcess+0xc73 //加载器的进程初始化函数
02 0012fd1c 7c90eac7 ntdll!_LdrpInitialize+0x183 //加载器初始化函数
03 00000000 00000000 ntdll!KiUserApcDispatcher+0x7 //用户态 APC 分发函数
```

这一次，AVrfInitializeVerifier 会输出如下信息：

AVRF: verifiee.exe: pid 0x16D0: flags 0x80000181: application verifier enabled

接下来，AVrfInitializeVerifier 会加载 verifier.dll，这个 DLL 加载后，AVrfInitializeVerifier 会根据注册表中 VerifierDlls 项的内容依次加载和初始化其中的每个验证提供器 DLL。此时，VerifierDlls 项的内容已经被放入到一个名为 AVrfpVerifierDllsString 的全局变量中：

```
0:000> du ntdll!AVrfpVerifierDllsString
7c97f760 "vrfcore.dll vfbasics.dll vfcomp"
7c97f7a0 "at.dll vfhangs.dll"
```

VerifierFlags 项的值保存在 AVrfpVerifierFlags 变量中：

```
0:000> dd ntdll!AVrfpVerifierFlags
7c97fc18 80000181 00000000 00000000 00000000
```

加载了 DLL 后，LdrpInitializeProcess 会初始化每个 DLL，也就是执行 DllMain 中的初始化代码。当 verifier.dll 初始化时，会输出以下信息：

AVRF: verifier.dll provider initialized for verifiee.exe with flags 0x80000181

AVrf 函数会将初始化好的验证提供器信息保存到一个链表中，并使用全局变量 AVrfpVerifierProvidersList 指向这个链表。

19.4.3 挂接 API

在加载了所有静态链接的 DLL 之后，加载器要做的下一件事就是遍历进程中每个模块（包括 EXE 模块）的 IAT（导入地址表），将其中的输入函数地址字段修改为目標函数的实际地址，即完成所谓的动态链接（绑定）。

举例来说，在 Verifiee 程序中，BugDoubleFree 函数调用了 VirtualFree API。对应的汇编代码为：

```
call    dword ptr [verifiee!_imp__VirtualFree (00417404)]
```

也就是读取 00417404 地址处的内容作为函数地址来调用它。在完成了前面介绍的加载工作后，这个地址处的值为 000178b0：

```
0:000> dd 00417404  
00417404 000178b0 000178be 000178ce 000178de
```

其中 178b0 是输入函数的 IMAGE_IMPORT_BY_NAME 结构偏移 (RVA)：

```
0:000> dt IMAGE_IMPORT_BY_NAME  
vrfcore!IMAGE_IMPORT_BY_NAME  
+0x000 Hint : Uint2B  
+0x002 Name : [1] UChar
```

使用 db 命令观察：

```
0:000> db 400000+178b0  
004178b0 f1 02 56 69 72 74 75 61-6c 46 72 65 65 00 ee 02 ..VirtualFree...
```

其中 0x02F1 为 Hint 值，后面的 11 个字节便是这个导入函数的名称，即 VirtualFree。最后的两个字节是下一个函数的 Hint 值。

加载器使用 LdrpWalkImportDescriptor 函数来遍历每个模块的输入表，清单 19-8 显示了这个函数的工作过程。

清单 19-8 加载器处理 IAT 表的过程

```
# ChildEBP RetAddr  
00 0012fd0 7c91d690 ntdll!LdrpSnapThunk+0xe3  
01 0012fa54 7c91d9cb ntdll!LdrpSnapIAT+0x20e  
02 0012fa80 7c91d944 ntdll!LdrpHandleOneOldFormatImportDescriptor+0xcc  
03 0012fa98 7c91c8a6 ntdll!LdrpHandleOldFormatImportDescriptors+0x1f  
04 0012fb14 7c922370 ntdll!LdrpWalkImportDescriptor+0x19e  
05 0012fc94 7c921639 ntdll!LdrpInitializeProcess+0xe02  
06 0012fd1c 7c90eac7 ntdll!_LdrpInitialize+0x183  
07 00000000 00000000 ntdll!KiUserApcDispatcher+0x7
```

LdrpWalkImportDescriptor 的第一个参数是 DLL 加载路径，第二个参数是一个 LDR_DATA_TABLE_ENTRY 结构。它处理的第一个模块便是 EXE 模块，使用 dt 命令观察第二个参数，结果如清单 19-9 所示。

清单 19-9 观察 LDR_DATA_TABLE_ENTRY 结构

```
0:000> dt ntdll!_LDR_DATA_TABLE_ENTRY 01428fb0  
+0x000 InLoadOrderLinks : _LIST_ENTRY [ 0x142afb0 - 0x1426fe4 ]  
+0x008 InMemoryOrderLinks : _LIST_ENTRY [ 0x142afb8 - 0x1426fec ]
```

```
+0x010 InInitializationOrderLinks : _LIST_ENTRY [ 0x0 - 0x0 ]
+0x018 DllBase           : 0x00400000
+0x01c EntryPoint        : 0x004024a0
+0x020 SizeOfImage       : 0x1b000
+0x024 FullDllName      : _UNICODE_STRING "C:\...\bin\Debug\verifiee.exe"
+0x02c BaseDllName       : _UNICODE_STRING "verifiee.exe"
+0x034 Flags              : 0x4000
+0x038 LoadCount         : 0xfffff
+0x03a TlsIndex          : 0
+0x03c HashLinks         : _LIST_ENTRY [ 0x1430fec - 0x7c97c268 ]
+0x03c SectionPointer    : 0x01430fec
+0x040 CheckSum          : 0x7c97c268
+0x044 TimeStamp          : 0x46eccd2f
+0x044 LoadedImports      : 0x46eccd2f
+0x048 EntryPointActivationContext : (null)
+0x04c PatchInformation   : (null)
```

经过 LdrpSnapThunk 处理后，00417404 处的值变为 7c809b04：

```
0:000> dd 00417404
00417404 7c809b04 000178be 000178ce 000178de
```

使用 ln 命令可以看到这正是位于 Kernel32 模块中的 VirtualFree 函数的实际地址：

```
0:000> ln 7c809b04
(7c809b04) KERNEL32!VirtualFree  |  (7c809b22) KERNEL32!VirtualFreeEx
Exact matches:
KERNEL32!VirtualFree = <no type information>,
```

以上便是动态链接库在运行期的动态绑定过程。为了支持应用程序验证，LdrpWalkImportDescriptor 在完成正常的处理工作后，会调用 AVrfDllLoadNotification，给应用验证器以处理机会。应用验证器正是利用这个机会来修改 IAT 而实现 API 挂接的，具体完成这项任务的函数是 AVrfpSnapDllImports（见清单 19-10）。

清单 19-10 调用应用验证器的 IAT 挂接函数

```
ChildEBP RetAddr
0012fa8c 7c95672a ntdll!AVrfpSnapDllImports+0x11e
0012faa4 7c93ab9f ntdll!AVrfDllLoadNotification+0x3f
0012fb14 7c922370 ntdll!LdrpWalkImportDescriptor+0x1f0
0012fc94 7c921639 ntdll!LdrpInitializeProcess+0xe02
0012fd1c 7c90ea7 ntdll!_LdrpInitialize+0x183
00000000 00000000 ntdll!KiUserApcDispatcher+0x7
```

经过 AVrfpSnapDllImports 处理后，00417404 处的值变为 00398af0：

```
0:000> dd 00417404
00417404 00398af0 7c809a71 7c812d76 7c80b6c1
```

使用 ln 命令，可以看到新的值是 AVrfpVirtualFree 函数的地址，位于验证提供器模块 vfbasics 中：

```
0:000> ln 00398af0
Exact matches: vfbasics!AVrfpVirtualFree (void *, unsigned long, unsigned long)
```

那么，AVrfpSnapDllImports 是怎么知道要把 00417404 处的值替换为 AVrfpVirtualFree 函数的地址呢？前面我们提到过，每个验证提供器除了包含一系列验证函数（代码）外，还包含一个或多个 Thunk 描述表，这些描述表的作用就是用

来描述系统 API 和验证函数的对应关系，比如通过 `x` 命令可以看到 `vfbasics` 模块中包含的 Thunk 描述表（数组）：

```
0:000> x vfbasics!*thunks*
003a7920 vfbasics!AVrfpOleaut32Thunks = struct
    _RTL_VERIFIER_THUNK_DESCRIPTOR [6]
003a75b8 vfbasics!AVrfpKernel32Thunks = struct
    _RTL_VERIFIER_THUNK_DESCRIPTOR [41]
...
```

可见，每个描述表是一个数组，每个数组元素是一个 `_RTL_VERIFIER_THUNK_DESCRIPTOR` 结构。下面是使用 `dt` 命令显示 `AVrfpKernel32Thunks` 数组的部分结果：

```
dt -a41 vfbasics!_RTL_VERIFIER_THUNK_DESCRIPTOR 003a75b8
...
[31] @ 003a772c
-----
+0x000 ThunkName      : 0x00391668 "VirtualFree"
+0x004 ThunkOldAddress : 0x7c809b04
+0x008 ThunkNewAddress : 0x00398af0
...
```

上面显示的正是关于 `VirtualFree` API 的转发（Thunk）信息，`0x7c809b04` 是原来的函数地址，`0x00398af0` 是验证函数的地址。

至此，我们可以想象出，`AVrfpSnapDllImports` 函数是通过每个验证提供器的 Thunk 表中的描述来挂接 API 的。它遍历 IAT 表中的导入函数地址，对于每个地址，然后在 Thunk 表中寻找与 `ThunkOldAddress` 匹配的元素，找到了便将其替换为 `ThunkNewAddress` 字段的值。

19.4.4 验证函数的执行过程

应用验证器成功初始化和挂接 API 后，当被验证的程序再调用这些 API 时，便会执行验证函数。比如，当 `Verifiee` 的 `BugDoubleFree` 函数执行到本来调用 `VirtualFree` API 的位置时，它便会进入到 `AVrfpVirtualFree` 函数中，也就是在本来调用被验证函数的地方实际会调用验证函数，因为 IAT 表被偷梁换柱了，清单 19-11 所示的栈回溯显示了实际的调用过程。

清单 19-11 调用验证函数

```
# ChildEBP RetAddr
00 0012f478 0040129a vfbasics!AVrfpVirtualFree+0x54 [e:\...\sics\vspace.c @ 1442]
01 0012f504 0040149f verifiee!BugDoubleFree+0x8a [C:\...\erifiee\Bugs.cpp @ 45]
02 0012f560 00402169 verifiee!CBugs::FireBug+0x2f [C:\...\Bugs.cpp @ 83]
03 0012f5bc 5f43749c verifiee!CVerifierDlg::OnBang+0x59 [C:\...\eeDlg.cpp @ 182]
04 0012f5f4 5f437bcb MFC42D!_AfxDispatchCmdMsg+0xa2
[省略多个栈帧]
```

与上一节介绍的内核态验证函数类似，`AVrfpVirtualFree` 函数执行的主要动作有：

- 调用 `vfbasics!AVrfpVirtualFreeSanityChecks`，执行健全性检查。

- 调用 vfbasics!VerifierGetAppCallerAddress 函数得到父函数的调用地址，这个地址属于被验证模块中的调用函数，对于本例，即 BugDoubleFree。
- 调用 vfbasics!AVrfpGetThunkDescriptor 函数得到 Thunk 描述表中的关于本函数的 _RTL_VERIFIER_THUNK_DESCRIPTOR 结构，然后调用结构中的 ThunkOldAddress 内容，即原来的 API。

执行以上动作后，AVrfpVirtualFree 返回。

19.4.5 报告验证失败

在 BugDoubleFree 函数中，我们故意设计了一个错误，即对同一个内存地址，调用两次 VirtualFree API。因此，当第二次调用 VirtualFree 时，验证函数会检测到这个错误情况，打印出清单 19-12 所示的信息。

清单 19-12 应用验证器报告验证失败

```
=====
VERIFIER STOP 60B:pid 0xA94:Trying to free virtual memory block that is already free.
03970000 : Memory block address.
00000000 : Not used.
00000000 : Not used.
00000000 : Not used.

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

观察栈回溯信息，其执行过程如清单 19-13 所示。

清单 19-13 报告应用验证失败

```
0:000> knL
# ChildEBP RetAddr
00 0012f134 00363933 ntdll!DbgBreakPoint
01 0012f338 003a3087 vrfcore!VerifierStopMessageEx+0x4bd
02 0012f35c 00398182 vfbasics!VfBasicsStopMessage+0x157
03 0012f3c4 00397dc5 vfbasics!AVrfpFreeVirtualMemNotify+0xa2
04 0012f3f0 7c809b4f vfbasics!AVrfpNtFreeVirtualMemory+0xf5
05 0012f410 7c809b19 KERNEL32!VirtualFreeEx+0x37
06 0012f428 00398bc2 KERNEL32!VirtualFree+0x15
07 0012f478 004012b4 vfbasics!AVrfpVirtualFree+0xd2
08 0012f504 0040149f verifiee!BugDoubleFree+0xa4
09 0012f560 00402169 verifiee!CBugs::FireBug+0x2f
0a 0012f5bc 5f43749c verifiee!CVerifieeDlg::OnBang+0x59
...
```

其中，AVrfpNtFreeVirtualMemory 是 ZwFreeVirtualMemory 的验证函数。VirtualFreeEx 中原本是调用 ZwFreeVirtualMemory 来通过系统服务释放虚拟内存，但因为验证器挂接了这个函数，所以它实际调用的是 AVrfpNtFreeVirtualMemory。AVrfpNtFreeVirtualMemory 被调用后，会执行 AVrfpHandleSanityChecks 函数进行检查，将要释放的地址（句柄）与记录的内存分配和释放记录进行核对，如果发现问题，就调用 VfBasicsStopMessage 报告错误信息。

19.4.6 验证停顿

应用验证器把因为验证失败而中断到调试器称为验证停顿(Verifier Stop)。为了便于分析和报告，系统为不同的验证失败情况定义了一个代码，称为验证停顿代码，其作用与蓝屏错误中的错误检查停止代码(Bug Check Stop Code)类似。每次验证停顿时，验证器通常还会报告4个参数，用来进一步描述验证失败的具体情况。对于清单19-12所示的验证停顿，其代码为0x60B，它的第一个参数是被多次释放的内存块地址，另外3个参数没有使用。

某些验证停顿是可以继续的，也就是说，在调试器中结束分析后，可以使用恢复执行命令让程序继续执行，比如上面所示的0x60B停顿。但某些验证停顿是不可以继续的，分析后继续执行，程序便会终止。

19.5 使用应用程序验证器

上一节我们介绍了应用程序验证器的工作原理，这一节将从应用的角度对其做进一步介绍。

19.5.1 应用验证管理器

应用验证管理器(简称AppVerifier)包含在应用验证工具包中，可以从Windows XP的安装光盘或者微软网站得到这个工具包。安装后，安装程序会将应用验证器的文件复制到system32目录中，并在开始菜单建立一个程序组。启动其中的appverif.exe，便可以看到图19-5所示的应用验证管理器界面。

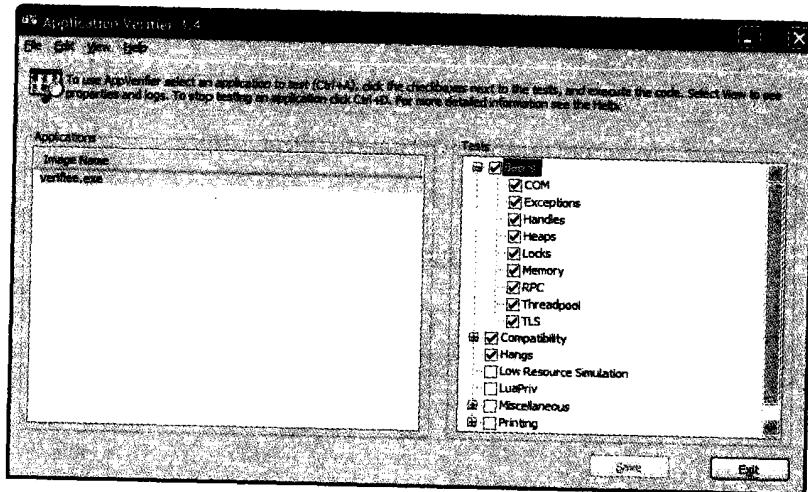


图 19-5 应用验证管理器的主界面

选择文件菜单的 Add Application 命令或按 Ctrl+A 热键后，便可以利用文件浏览对话框寻找要验证的应用程序文件（EXE），选择后便会在主界面的左侧列表中。如果要验证多个程序，那么重复这个过程可以加入多个程序。在图 19-5 显示的界面中我们加入了一个程序，即 verifiee.exe。

选中左侧的程序文件，在右侧可以配置对其要做的验证项目（稍后讨论）。与驱动验证中所有驱动程序共享一套验证项目不同，应用程序验证时，每个应用程序可以有自己的验证项目。

选择好后，可以点击 Save 按钮保存设置。此时，可能弹出图 19-6 所示的对话框，提示我们需要在调试器下运行被验证程序。这是因为验证函数需要调试器报告验证信息和验证停顿。

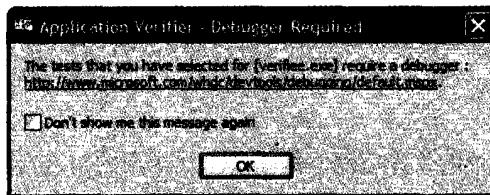


图 19-6 验证管理器提示需要在调试器下运行被验证程序

除了图形界面，AppVerif 程序也支持以命令行方式工作。先启动一个控制台窗口，然后输入 appverif /?便可以看到一个简单的帮助。表 19-1 列出了 appverif 所支持的命令行开关和每一种开关所需要的参数。

表 19-1 应用验证管理器的命令行选项

开关	描述	
-enable	TEST ... -for TARGET ... [-with [TEST.]PROPERTY=VALUE ...]	启用验证项目
-disable	TEST ... -for TARGET ...	禁止验证项目
-query	TEST ... -for TARGET ...	查询验证信息
-configure	STOP ... -for TARGET ... -with PROPERTY=VALUE...	配置验证停顿
-verify	TARGET [-faults [PROBABILITY [TIMEOUT [DLL ...]]]]	添加被验证程序
-export	log -for TARGET -with To=XML_FILE [Symbols=SYMBOL_PATH] [StampFrom=LOG_STAMP] [StampTo=LOG_STAMP][Log=RELATIVE_TO_LAST_INDEX]	将验证信息输出到文件
-delete	[logs]settings] -for TARGET ...	删除日志记录
-stamp	log -for TARGET -with Stamp=LOG_STAMP [Log=RELATIVE_TO_LAST_INDEX]	向日志信息中增加戳标记
-logtoxml	LOGFILE XMLFILE	将日志文件转为 XML 格式
-installprovider	PROVIDERBINARY	安装验证提供模块

在使用时，参数中的大写部分应该替换为实际需要的内容，例如 TARGET 应该为要验证应用程序的可执行文件名（如 verifiee.exe）或进程 ID；TEST 应该验证项目的名字，即表 19-1 中第二列的名称；STOP 是验证停顿代码，可以为十进制形式或以 0x 开始的十六进制形式；PROPERTY 为验证项目的属性名称（详见下文）；VALUE 为属性的值。

19.5.2 验证项目

目前版本的验证器设计了 19 个验证项目，分为 6 个类别。表 19-2 列出了这些项目。

表 19-2 应用验证项目

Basics (基本)	COM	正确使用组件对象模型 (COM)	0x400 ~ 0x410
	Exceptions	检测非法访问异常 (Access Violation)	0x650
	Handles	正确使用句柄	0x300 ~ 0x305
	Heaps	正确使用堆	0x1 ~ 0x14
	Locks	正确使用同步对象	0x200 ~ 0x215
	Memory	合理使用虚拟内存	0x600 ~ 0x61E
	RPC	正确使用远程过程调用 (RPC)	0x500
	Threadpool	正确使用线程池	0x700 ~ 0x709
	TLS	正确使用线程局部存储 API	0x350 ~ 0x352
Compatibility (兼容性)	FilePaths	正确读取公共目录和使用有关 API，如 SHGetFolderPath	0x2400 ~ 0x240A
	HighVersionLie	读取版本信息的方式	0x2200 ~ 0x2204
	Interactive-Services	验证交互式系统服务	0x2800 ~ 0x2807
	KernelMode-DriverInstall	正确安装内核态驱动程序	0x2305 ~ 0x230D
Hangs	Hangs	检测可能导致程序僵死的情况	0x2000 ~ 0x2002
低资源模拟	LowRes	模拟低资源情况，对分配资源请求返回失败，又称为错误注入 (Fault Injection)	无
LuaPriv	LuaPriv	验证账号和安全有关的设计要求*	0x3300 ~ 0x3339
杂项	DangerousAPIs	检查使用危险 API 的情况	0x100 ~ 0x104
	DirtyStacks	探测使用未初始化局部变量的情况	无
	TimeRollOver	强制 GetTickCount API 快速重新计数，以考验应用程序是否处理此情况	无
打印	PrintAPI	正确使用打印 API	0xA0000 ~ 0xA01E
	PrintDriver	验证打印驱动程序	0xD0000 ~ 0xD027
服务	Service	验证系统服务	0x4000 ~ 0x4007

*LUA 是 Limited User Account 的缩写，这个验证的目的是判断在管理员权限可以运行的程序是否也可以在受限用户账号下正常运行。

对于表 19-2 中没有验证停顿代码的验证项目，它们的主要目的是监视应用程序在某一方面的函数调用，然后输出信息。

19.5.3 配置验证属性

可以通过指定属性值来定制某些验证项目的行为。其操作方法是选中一个验证项目，然后选择 Edit 菜单中的 Properties，而后通过弹出的对话框来输入属性的值。也可以选择 View 菜单中的 Property Window 使主窗口中出现属性栏（默认隐藏）。或者，也可以通过命令行方式的-enable 子命令来定制属性。

19.5.4 配置验证停顿

对于可能产生验证停顿的验证项目，可以配置每个验证停顿的属性。其操作方法是选中一个验证项目，然后选择 Edit 菜单中的 Verifier Stop Options。接下来可以在图 19-7 所示的对话框中选择验证停顿选项。

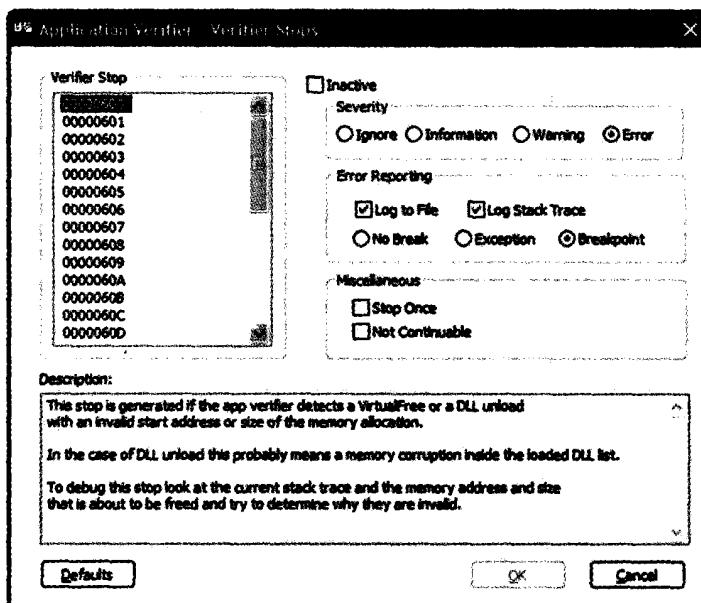


图 19-7 配置验证停顿选项

也可以使用命令方式的-configure 命令来配置验证停顿选项。

19.5.5 编程调用

为了方便的与其他测试工具集成，应用验证器提供了 COM 组件和一组接口。通过这组接口，用户可以很方便地以编程方式启用验证器和配置验证选项。应用验证器的安装程序安装了编程所需的文件，包括头文件 vrfauto.h，接口定义文件 vrfauto.idl 和 COM 组件的实现模块 vrfauto.dll。这些文件被安装在程序文件目录中（c:\Program Files\Application Verifier）。

19.5.6 调试扩展

为了支持应用程序验证，WinDBG 专门配备了一个扩展命令!avr。使用这个命令可以控制应用验证选项并查询和显示验证信息。例如，!avr -ex 可以显示已经发生的所有异常的信息；!avr -threads 可以显示所有线程的信息；不带任何开关可以显示已经发生的验证停顿的详细信息，等等。执行!avr -?可以得到一个简单的帮助。WinDBG 的帮助文件包含了关于!avr 扩展命令的详细用法。

本节简要介绍了应用验证器的使用方法，更详细的介绍请参阅应用验证器的帮助文件（appverif.chm）。

19.6 本章总结

本章介绍了 Windows 操作系统所提供的验证机制，第 19.1 节为概括性的介绍，第 19.2、19.3 节介绍了用于验证内核态模块的驱动验证器，第 19.4、19.5 节介绍了用于验证用户态程序的应用验证器。介绍这些内容的目的主要有两个。

第一，验证机制是内建在 Windows 操作系统中的一个固定部分，被包含在 XP 开始的所有 NT 系列 Windows 系统中。因为验证机制通过调试信息来报告验证信息、使用异常来报告验证停顿，所以验证机制与软件调试有着密切的关系，很多时候，我们在调试时，就会遇到验证器的函数或看到验证器输出的信息。或者，我们调试的问题是通过验证器而发现的。

第二，验证机制是 Windows 操作系统提供的用来提高测试效率和帮助提高软件质量的一种重要手段。自从 Windows 2000 和 XP 分别引入驱动验证和应用程序验证机制后，它便得到了广泛的认可，逐步成为测试 Windows 系统中系统软件和应用软件的一种标准方法，驱动程序验证是 WHQL 测试的一个必不可少的内容。从这个角度来看，验证机制也是 Windows 系统中用来支持软件开发，提高软件开发、测试和调试效率的众多设施中的一个部分。

很多人喜欢系统验证器来帮助测试和开发，因为这样可以更快地发现问题，把问题消灭在萌芽状态。但是也有人不相信使用验证器发现的问题，认为验证器触发的问题在现实中不会发生，或者怀疑是验证器本身的问题。笔者认为应该重视验证器的验证结果，在怀疑验证器之前，应该认真分析自己的代码。

本章介绍的验证机制有时也被称为运行期验证。它是运行期检查的一种方法。所谓运行期检查就是指在软件运行的过程中，监视、分析和探测软件中存在的问题。与运行期检查相对应的是编译期检查，即在软件的编译期间，分析和检查软件中存在的错误。二者的最大区别是，编译器检查是静态的检查，通常只能发现静态问题。而运行期检查可以发现运行过程中才体现出来的动态问题。

我们将在第 20 章介绍编译期检查。第 21 章介绍编译器提供的运行期检查功能，包括自动插入的检查和程序员利用断言等机制手工插入的检查。第 23 章我们将介绍有关堆的检查和辅助调试设施。

参考文献

1. Driver Verifier in Windows Vista. Microsoft Corporation, 2006
2. Patrick Garvan. Debugging with NTSD and Application Verifier. Dr. Dobb's Portal, 2007

编译器的调试支持



在计算机科学尤其是软件的发展历史中，Grace Murray Hopper (1906—1992) 是一位做出了重大贡献的杰出女科学家。她的主要成就之一就是发明了编译器——可以将程序语言翻译成机器码的自动编码工具。Grace 于 1951 年到 1952 年间开发了 A-0 编译器，A-0 编译器被公认为是计算机历史上的第一个编译器。

编译器的出现为编程语言的繁荣和软件产业的形成奠定了基础。有了编译器，人们把繁琐的编码工作交给机器来完成，编码时间一下子减小到几乎可以忽略的地步，以至于今天我们很少再关心机器码的产生过程，就连编码这个词的主要含义也由本来的编写机器码演变为编写源代码。

编译器的出现，对软件调试也有着极其重要的意义。在 1955 年题为《数字计算机自动编码》的著名演讲中，Grace 在谈到自动编码的优势时，首先讲的是降低了调试时间，因为编译器编码比人工编码更加准确，而且可以做很多自动的检查。可见，Grace 在设计第一个编译器时便考虑到了检查编程错误和支持软件调试。今天的编译

器尽管与 50 多年前 Grace 所发明的编译器有了很大的变化，但是有一点没有变，那就是支持调试仍然是编译器设计中的一项基本任务，很多调试技术都是建立在编译器的有关支持基础上的。可以把编译器的调试支持概括为以下几个方面。

编译期检查：编译器在编译过程中，除了检查代码中的语法错误外，还会检查可能存在的逻辑错误和设计缺欠，并以编译错误或警告的形式报告出来。

运行期检查：为了帮助发现程序在运行阶段所出现的问题，编译器在编译时可以产生并加入检查功能，包括内存检查、栈检查等。

调试符号：今天的大多数软件都是使用更易于人类理解的中高级编程语言（如 C/C++，Pascal 等）编写的，然后由编译器编译为可执行程序交给 CPU 执行。当调试这样的程序时，我们可以使用源程序中的变量名来观察变量，跟踪或单步执行源程序语句，仿佛 CPU 就是在直接执行高级语言编写的源程序。我们把这种调试方式称为源代码级调试（Source Code Level Debugging）。要支持源代码级调试，调试器必须有足够的信息将 CPU 使用的二进制地址与源程序中的函数名、变量名和源代码行联系起来，起到这种桥梁作用的便是编译器所产生的调试符号（Debugging Symbols）。调试符号不仅在源代码级调试中起着不可缺少的作用，在没有源代码的汇编级调试和分析故障转储文件时，也是非常宝贵的资源。有了正确的调试符号，我们便可以看到要调用的函数名称或要访问的变量名（只要调试符号中包含它）。当对冗长晦涩的汇编指令进行长时间跟踪时，这些符号的作用就好像是黑夜中航行时所遇到的一个个灯塔。

内存分配和释放：使用内存的方法和策略关及软件的性能、稳定性、资源占用量等诸多指标，很多软件问题也都与内存使用不当有关。因此，如何降低内存使用的复杂度，减少因为内存使用所导致的问题便很自然地成为编译器设计中的一个重要目标。例如，在编译调试版本时，编译器通常会使用调试版本的内存分配函数，加入自动的错误检查和报告功能。

异常处理：Windows 操作系统和 C++ 这样的编程语言都提供了异常处理和保护机制。编译包含异常保护机制的代码需要编译器的支持。

映射（MAP）文件：很多时候，我们可以得到程序崩溃或发生错误的内存地址，这时我们很想得到的信息就是这个地址属于哪个模块，哪个源文件，甚至哪个函数更好。调试符号中包含了这些信息，但是调试符号通常是以二进制的数据库文件形式存储的，适合调试器使用，不适合人工直接查阅。映射（MAP）文件以文本文件的形式满足了这一需要。

本篇将先介绍编译有关的基本概念和编译期检查（第 20 章），然后介绍运行库和运行期检查（第 21 章），第 22 章和第 23 章将分别介绍栈和堆的原理和有关问题，第 24 章将介绍编译器编译异常处理代码的方法，第 25 章将详细讨论调试符号。

编译和编译期检查

除了基本的编译功能，检查并报告被编译软件中的错误是编译器设计中的另一个主要目标。为了实现这一目标，编译器在编译源代码和链接目标程序时会做很多检查工作，这些检查有些是为了实现编译目标所必需的，比如不完整的语句会直接阻碍编译过程；有些是为了发现设计缺欠的，比如数据类型检查。

本章的前半部分将介绍编译器的基本常识，包括构建程序（Build）的一般过程（20.1 节）、编译器的组成（20.2 节）及 VC 编译器（20.3 节）。后半部分将介绍编译错误（20.4 节）和编译期检查（20.5 节），最后讨论用于增强编译期检查效果的标准标注语言（SAL）（20.6 节）。

20.1 程序的构建过程

软件是程序及其文档的集合，软件开发的核心任务就是生产出可以满足用户需要的程序。图 20-1 勾勒出了构建并执行一个 C 语言程序的典型过程。最左侧是源程序，它经过编译器（Compiler）被编译为等价的汇编语言模块，再经过汇编器（Assembler）产生出与目标平台 CPU 一致的机器码模块。尽管机器码模块中包含的指令已经可以被目标 CPU 所执行，但其中可能还包含没有解决（unresolved）的名称和地址引用，因此需要链接器（Linker）解决这些问题，并产生出符合目标平台上的操作系统所要求格式的可执行模块。当用户执行程序时，操作系统的加载器（Loader）会解读链接器记录在可执行模块中的格式信息，将程序中的代码和数据“布置”在内存中，成为真正可以运行的内存映像。

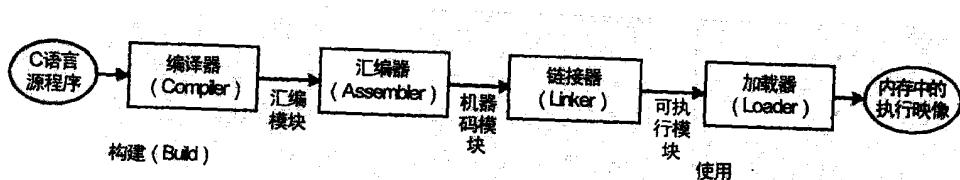


图 20-1 构建和运行程序的典型过程

编译器和汇编器所做的主要都是翻译工作，因此以目标代码（机器码）为分界，可以把程序的构建过程分为编译和链接两个阶段，我们将在下一节介绍编译器的更多内容，本节只简要讨论链接器和加载器。

20.1.1 链接器

链接器的主要职责是将编译器产生的多个目标文件合成为一个可以在目标平台下执行的执行映像。这里说的目标平台是指程序的运行环境，包括 CPU 和操作系统。举例来说，如果我们要在 Windows 操作系统下运行的应用程序，那么链接器应该根据 Windows 操作系统定义的可执行文件格式来产生可执行文件，也就是产生 PE (Portable Executable) 格式的执行映像文件。要产生一个 PE 格式的可执行文件，链接器要完成的典型任务如下。

- 解决目标文件中的外部符号，包括函数调用和变量引用。如果调用的函数是 Windows API 或其他位于 DLL 模块中的函数，那么须要为这些调用建立输入目录表（Import Directory Table）和输入地址表（Import Address Table，简称 IAT）。输入目录表用来描述被引用的文件，IAT 表用来记录或重定位被引用函数的地址。链接器会把输入目录表和 IAT 表放在 PE 文件的输入数据段 (.idata) 中。
- 生成代码段 (.text)，放入已经解决了外部引用的目标代码。
- 生成包含只读数据的数据段 (.data)。
- 生成包含资源数据的资源段 (.rsrc)。
- 生成包含基地址重定位表（Base Relocation Table）的.reloc 段。当链接器产生 PE 文件时，它会假定一个地址作为本模块的基地址，比如 VC6 编译器为 EXE 模块定义的默认基地址是 0x00400000。当程序运行时，如果加载器将一个模块加载到与默认值不同的基地址，那么这时便须要用重定位表来进行重定位。可以通过链接器的链接选项来指定模块的默认基地址，也可以使用 Visual Studio 所附带的 Rebase 工具来修改 DLL 文件的默认基地址。
- 如果定义了输出函数和变量，则产生包含输出表的.edata 段。输出表通常出现在 DLL 文件中，EXE 文件一般不包含.edata 段，但 NTOSKRNL.EXE 是个例外。
- 生成 PE 文件头，文件头描述了文件的构成和程序的基本信息。

理解链接器和 PE 文件细节的一种极好的方法是使用 PEView 工具 (<http://www.magma.ca/~wjr/>)，或者 SDK 附带的 dumpbin 工具来观察和分析链接器所产生的 PE 文件。在第 19 章介绍驱动验证和应用程序验证时讨论过 IAT 表，将在第 25 章介绍 PE 文件的文件头格式及使用 dumpbin 工具的一些例子。

20.1.2 加载器

加载器 (Loader) 是操作系统的一个部分，它负责将可执行程序从外部存储器 (如硬盘) 加载到内存中，并做好执行准备，包括遍历输入目录表加载依赖模块，遍历 IAT 表绑定动态调用的函数，对基地址发生冲突的模块执行调整工作等。NTDLL 中包含了一系列以 Ldr 开头的函数，用于完成以上任务，第 19 章简要介绍了其中的部分函数。

20.2 编译

编译器的基本功能就是将使用一种语言编写的程序（源程序）翻译成用另一种语言表示的等价程序（目标）。现实中，通常是从高层次的语言翻译到低层次的语言，比如将使用 C++ 语言编写的源程序编译成可以被 CPU 理解的汇编语言程序。

尽管编译器的种类很多，实现也千差万别，但是其基本结构通常都是由图 20-2 所示的前端和后端两个部分组成。前端主要负责理解源代码的含义，即分析 (analysis) 功能；后端负责产生等价的目标程序，即合成 (synthesis) 功能。

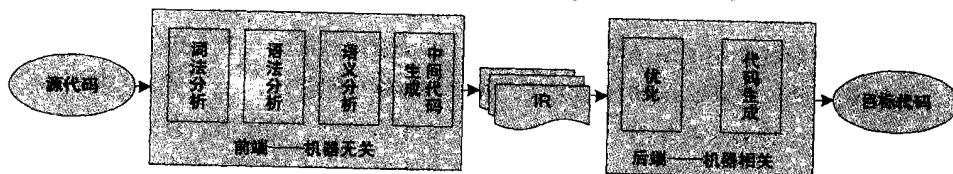


图 20-2 编译器的基本结构

前端和后端之间的媒介是中间代码，又称为中间表示 (Intermediate Representation)，即 IR。编译器前端对源程序进行词法分析、语法分析、语义分析，并将其映射到中间表示，后端对中间表示进行优化处理，再将其映射到用机器码表示的目标程序中。而后，链接器再将目标程序经链接成为可以执行的执行映像。

20.2.1 前端

编译器的前端 (Front End) 负责扫描和分析源程序并产生中间表示 (IR)，它主要完成如下几项任务。

词法分析 (Lexical Analysis): 读入并扫描源程序文件的字符流，剔除其中的空格和注释内容，并根据构词规则识别出单词，将源代码中的字节流转换成记号流 (token stream)。实现该功能的部分通常被称为扫描器 (Scanner)。

语法分析 (Syntax Analysis): 对词法分析产生的记号流进行层次分析，根据语法规则 (syntax rules) 把单词序列组成语法短语，并表示为语法树 (syntax tree) 或推导树 (derivation tree) 的形式。

语义分析 (Semantic Analysis): 对语法树中的语句进行语义处理，审查数据类型的正确性，以及运算符使用是否符合语言规范。因为编译阶段的语义分析无法分析程序运行时才确定的动态语义，所以编译时的语义分析又被称为静态语义分析 (Static Semantic Analysis)，简称 SSA。

中间代码生成: 产生编译器前端和后端交流所使用的中间表示 (IR)，有时也称为中间代码。中间表示的具体形式因编译器的不同而不同，常见的有三元式 (three-address code)、四元式 (4-tuple code) 等。

20.2.2 后端

编译器的后端 (Back End) 负责对前端产生的中间代码进行优化处理，并产生使用目标代码表示的目标程序。优化后的代码仍然是使用中间代码表示的。目标程序经过链接最终成为可以在特定平台上运行的可执行程序。

编译器的前端-后端设计使它们分工明确，前端与被编译的语言相关，不必关心目标平台 (CPU)，而后端与目标平台相关，不必关心源程序语言。这样的好处是编译器的前端和后端松耦合，更容易支持新的语言和新的目标平台。

本节简要介绍了编译器的基本构成，以及编译过程的 6 个主要步骤：词法分析、语法分析、语义分析、中间代码生成、优化和目标代码生成。下一节我们将以 VC 编译器为例进一步理解这些步骤。除了完成以上 6 个步骤外，编译器还有另外两个重要任务，那就是符号表 (Symbol Table) 管理和错误处理。这两项功能与以上 6 个步骤几乎都有关系。我们将在第 25 章讨论符号表有关的内容，第 20.4 节将介绍编译器的错误处理。

20.3 Visual C++ 编译器

微软的 Visual C++ 编译器 (简称 MSVC 或 VC) 是开发 C 和 C++ 程序的最常用编译器之一。因为本书的大多数内容都是以 MSVC 为例来讨论的，几乎所有演示代码也都是使用它来编译的，所以本节将概要性地介绍 VC 编译器。

20.3.1 MSVC 简史

MSVC 的前身是微软的 C 编译器 (Microsoft C Compiler)，简称 MSC。MSC 的主要用途是开发 DOS 程序和早期 16 位版本的 Windows (Windows 1.0~3.x) 程序。1992 年发布的 MSC 7.0 加入了很多新的功能，包括支持 C++ 语言、MFC 框架 (1.0)、预编译头文件、P-Code、函数 inline 和远程调试等。

为了写作本节的内容，笔者特意安装了一份 MSC 7.0，看到了很多亲切的名字。比如，它的调试器叫做 CodeView，此名字还存在于今天的很多文档中。它的集成环境

叫 Programmer's WorkBench，简称 PWB。PWB 是运行在 DOS 下的一个“窗口”程序。

因为支持 C++ 语言，所以 MSC 7.0 的正式名称为 Microsoft C/C++ Compiler 7.0，但事实上它所包含的 C++ 编译器还是第一个版本。因此当 MSC 8.0 发布时，便更名为 Microsoft Visual C++ 1.0 (MSVC 1.0)。

MSVC 1.0 分为 16 位版本和 32 位版本。32 位版本可以开发 Windows NT 程序和 Windows 3.x 下的 Win32s 程序。MSVC 1.0 的第一个版本也是在 1992 年发布的。MSVC 1.0 包含了 MFC 2.0 框架。表 20-1 列出了 MSC 和 MSVC 的主要版本、发布时间和它们的概要信息。

表 20-1 微软 C/C++ 编译器的主要版本

版本	发布时间	新增主要功能
MSC 7.0	1992	C++、MFC 1.0
MSVC 1.0	1992	32 位支持，MFC 2.0
MSVC 2.0	1994	MFC 3.0
MSVC 4.0	1995	MFC 4.0，Developer Studio IDE (MSDEV.exe)，ClassView
MSVC 4.2	1996	MFC 4.2
MSVC 5.0	1997	MFC 4.21，COM
MSVC 6.0	1998	运行时错误检查，Edit and Continue，MFC 6.0
MSVC 7.0	2002	Visual Studio .NET 2002 (又称 VS7.0) 的一部分，.NET 支持，link time code generation，MFC 7.0
MSVC 7.1	2003	Visual Studio .NET 2003 (又称 VS7.1) 的一部分，Visual J#，MFC 7.1
MSVC 8.0	2005	Visual Studio 2005 (又称 VS8.0) 的一部分，C++/CLI，OpenMP，MFC 8.0
MSVC 9.0	2007	Visual Studio 2008 (又称 VS9.0) 的一部分

VC 编译器的发展可以分为两个主要阶段，第一阶段从 VC1.0 开始到 VC6.0 结束，这一阶段的核心功能是使用 C/C++ 语言开发本地的 Windows 应用程序。VC7 开始引入了.NET 支持，可以开发托管 (managed) 程序。下面分别以 VC6 和 VC8 作为代表加以介绍。

20.3.2 MSVC6

MSVC6 (简称 VC6) 是使用最广泛的 C/C++ 编译器之一，今天运行在 Windows 系统上的很多软件都是由它编译产生的。尽管它发布于 1998 年，但至今仍被广泛地应用着。

VC6 是 Microsoft Visual Studio 6.0 软件集合的一个部分，同集合的还有 VB6、Visual Foxpro 等。图 20-3 显示了 VC6 在磁盘中的目录布局 (左侧) 和它的部分文件 (右侧)。

图中 Common 目录用来存放 Visual Studio 的公共组件，其中 MSDev98 目录下存放了集成开发环境 (IDE) 所使用的各种文件，MSDev98 下的 Bin 目录中的 MSDEV.EXE

是 IDE 环境的主程序。MSDEV 是 Microsoft Developer Studio 的缩写，这是从 VC4.0 便开始使用的 IDE 名称。在 Bin 目录下还有一些重要的文件，如实现 IDE 主要界面逻辑（shell）的 devshl.dll、负责生成 PDB 文件的 msfdb60.dll、用于反汇编的 msdis110.dll、实现 Edit and Continue（简称 EnC）功能的 msenc10.dll、编译资源文件的 RC.exe、用于远程调试的 MSVCMON.EXE，以及供集成调试器使用的 eecxx.dll（评估 C++ 表达式）、dm.dll（Debuggee Module）和 em.dll（Execution Model）等。

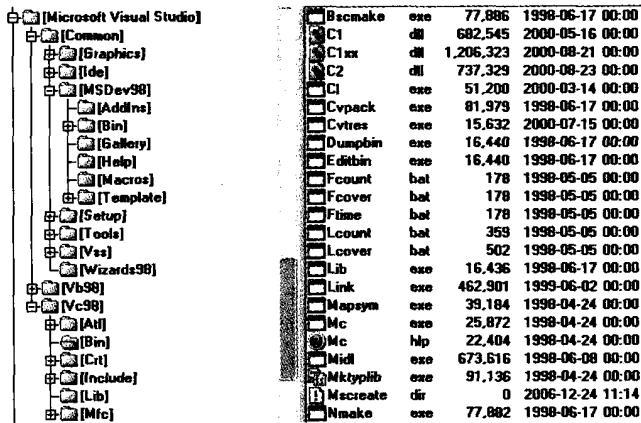


图 20-3 Visual Studio 6.0 (VC6) 的目录布局

Vc98 目录中存放着 VC6 专用的各种文件，其下的 Bin 目录包含了 C/C++ 编译器的核心模块（右侧）。例如，C1.DLL 和 C1XX.DLL 分别是 C 编译器和 C++ 编译器的前端（Front End），C2.DLL 是编译器的后端（Back End）。CL.EXE 是所谓的 Compiler Driver（编译推动器），当编译一个源程序时，我们只要执行 CL.EXE，它会依次调用前端后端，并在没有编译错误的情况下调用 Link.exe 以进行链接操作。这个目录中的重要文件还有用于编译消息文件的 MC.EXE、编译接口描述文件（IDL）的 MIDL.EXE，以及处理调试信息的 cvpack.exe。

在 MSDev98\bin 目录下还有一个重要的名为 IDE 的子目录，其下面有一系列以 PKG 为后缀的文件，这些实际上是 DLL 格式的程序模块，每个 PKG 模块都至少输出了 InitPackage 和 ExitPackage 两个函数。以下是几个重要的 PKG 模块：Devbld.pkg 包含了 CBuildEngine 类的实现，是执行 Build 功能的主要模块；Devdbg.pkg 是实现调试功能的主要模块；Devclvw.pkg 是用于显示类视图（ClassView）的模块。

从图 20-3 中可以看到有些文件的日期是 1998 年的，有些是 2000 年的，后者是在安装 VC6 的 SP5（Service Pack 5）时更新的。

20.3.3 VS7 和 VS8

Visual Studio .NET 2002 是 Visual Studio 6.0 (VS6) 的下一个版本，因此又称为

Visual Studio 7.0，简称 VS7。VS7 引入的一个最重要功能就是支持开发.NET 程序，包括 ASP.NET 网络应用和 Windows Form。下面列出了 VS7 引入的其他变化。

- 支持 C#语言，称为 Visual C#。
- 真正把不同语言所使用的 IDE 统一在一起，主程序为 devenv.exe。此前，不同的语言使用的 IDE 程序是不同的，如 VC6 的 IDE 是 MSDEV.EXE，而 VB6 的 IDE 是 VB6.exe。
- 实现了对使用 VB、VC 和 C#语言开发的多个模块的跨语言调试。
- 引入了混合调试（Interop Debugging）的概念，支持在一个调试会话中同时调试托管代码和本地代码。
- VC 编译器增加了检查缓冲区溢出的安全 Cookie，又称 GS 编译开关（/GS）。

2003 年推出的 Visual Studio .NET 2003（又被称为 VS7.1）修正 2002 版本存在的问题，除增加了 Visual J#语言外，新增的功能不多。在调试方面，VS7.1 开始支持自动从符号服务器下载调试符号。

2005 年推出的 Visual Studio 2005（VS2005 或 VS8）是微软开发工具产品中在 VS6 后的最重要版本，它完善了 Visual Studio .NET 2002 和 2003 的各种功能，并引入了很多新的功能，如 C++/CLI。

图 20-4 显示了 VS8 在磁盘上的目录结构和部分文件。从图中可以看出，它的结构仍保留了很多 VS6 的痕迹，但增加了很多新的内容，如用于访问符号文件的 DIA SDK、支持团队开发和手机开发的各种工具及.NET SDK 等。

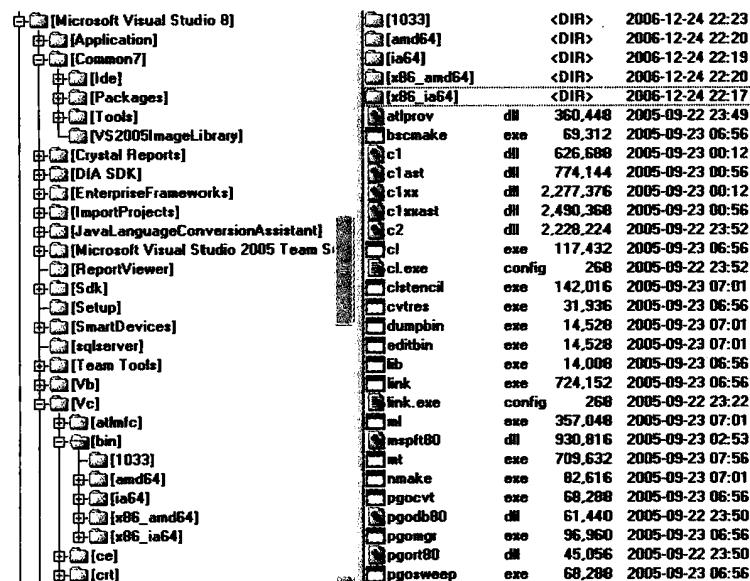


图 20-4 VS2005（VS8）的目录结构和部分文件

Common7 的 IDE 目录存放了集成开发环境的各个模块，包括主程序 devenv.exe、实现 IDE 主要逻辑的 msenv.dll、用于反汇编的 msdis150.dll、用于产生 PDB 文件的 mspdb80.dll 和 mspdbccore.dll、实现 EnC 的 msenc80.dll 等。

Common7 的 Packages 目录用于存放 VS8 所使用的组件包，包括用来以所见即所得的方式（WYSIWYG）设计 HTML 页面的 Vswebdesign.dll、编辑 HTML 源文件的 srredit.dll 等。在这个目录下的 Debugger 子目录包含了 VS8 的集成调试器的大多数模块，如 vsdebug.dll（调试器主模块，CDebugger 类）、NatDbgDE.dll（本地调试引擎）、NatDbgEE.dll（表达式评估）、vcencbld.dll（用于 EnC 的构建器）和 msdia80.dll（访问符号）等。

20.3.4 构建程序

下面通过以 VC8 为例介绍构建一个应用程序的过程。清单 20-1 显示了 VC8 的一个工作线程在执行构建动作时的函数调用序列。最下面是 CVCBuildThread 类的 BuildThread 方法，CConfiguration 是读取项目配置信息的类，而后是 CDynamicBuildEngine 开始执行真正的 Build 动作，调用 ExecuteCmdLines 方法产生命令行，再使用 CConsoleSpawner 根据命令行创建在后台运行的编译进程。观察 SpawnCommandLine 方法的第二个参数，可以看到执行的命令行为：

```
cl.exe @c:\...\HiWorld\Debug\RSP00000252485908.rsp /nologo /errorReport:prompt
```

其中 CL.EXE 正是我们前面提到过的编译推动器程序。以上命令行中的第一个参数是 VS8 动态产生的文件，用于向 CL 传递要执行的任务，以下是其中的内容：

```
/Od /Oi /D "WIN32" /D "_DEBUG" /D "_WINDOWS" /D "_UNICODE" /D "UNICODE" /Gm /EHsc
/RTC1 /MDd /Yu"stdafx.h" /Fp"Debug\HiWorld.pch" /Fx /FAs /Fa"Debug\""
/Fo"Debug\" /Fd"Debug\vc80.pdb" /W3 /c /Wp64 /Z7 /TP .\HiWorld.cpp
```

可见，尽管是在 IDE 中执行 Build 操作，IDE 还是会以命令行的方式来执行真正的编译任务。启动 CL 程序后，VC8 会一直等待这个进程退出，清单最上面的两个栈帧即 CConsoleSpawner 对象是读取 CL 进程输出的执行结果。

清单 20-1 VS8 执行构建动作（Build）的过程

```
0:016> k
ChildEBP RetAddr
0746e8c0 5aedfdd9 kernel32!ReadFile+0x16c
0746f950 5aee0245 VCProjectEngine!CConsoleSpawner::ReadChildProcessOutput+...
0746fa3c 5aedf79b VCProjectEngine!CConsoleSpawner::SpawnCommandLine+0x9e4
0746faf4 5aedfc4f VCProjectEngine!CConsoleSpawner::SpawnCommandLines+0x334
0746fb34 5aedfb6c VCProjectEngine!CDynamicBuildEngine::ExecuteCmdLines+0x5b
0746fb60 5aedfb45 VCProjectEngine!CDynamicBuildEngine::ExecuteCommandLines+...
0746fbac 5aedfa90 VCProjectEngine!CVCToolImpl::ExecuteBuild+0x9a
0746fbe4 5aedef65 VCProjectEngine!CVCToolImpl::PerformBuildActions+0x7f
0746fc3c 5aedeef6 VCProjectEngine!CBldToolWrapper::PerformBuildActions+0x20d
0746fc94 5aee3774 VCProjectEngine!CDynamicBuildEngine::PerformActualBuild+...
0746fdb4 5aee3833 VCProjectEngine!CDynamicBuildEngine::DoBuild+0x47b
0746fde4 5aeee3157 VCProjectEngine!CDynamicBuildEngine::DoBuild+0x36
```

```
0746fec0 5aeee32c3 VCProjectEngine!CConfiguration::DoPreparedBuild+0x6fe
0746ff40 5aeee2ad3 VCProjectEngine!CConfiguration::TopLevelBuild+0x112
0746ffb4 7c80b683 VCProjectEngine!CVCBuildThread::BuildThread+0x2c9
0746ffec 00000000 kernel32!BaseThreadStart+0x37
```

在编译通过后，VC8 会通过类似的过程执行 Link.exe，其命令行为：

```
link.exe @c:\...\\HiWorld\\Debug\\RSP00000452485908.rsp /NOLOGO
/ERRORREPORT:PROMPT
```

VC6 构建程序的过程与此非常类似，也是使用一个临时文件 (.tmp) 将要执行的任务传递给 CL.EXE 或 Link.exe。

20.3.5 调试

当我们在 IDE 环境中调试程序时，会加载集成的调试器模块来进行调试。清单 20-2 显示了 VC8 集成调试器调试工作线程的执行状态，此时这个线程正在调用 NtWaitForDebugEvent 内核服务等待调试事件。

清单 20-2 VC8 的调试器工作线程在等待调试事件

```
0:018> k
ChildEBP RetAddr
0746fe34 7c90e996 ntdll!KiFastSystemCallRet
0746fe38 7c95072b ntdll!NtWaitForDebugEvent+0xc
0746fe50 7c85a621 ntdll!DbgUiWaitStateChange+0x1e
0746fecc 5be11783 kernel32!WaitForDebugEvent+0x21
0746fed8 5be119c5 NatDbgDE!Win32Debugger::RawWaitForDebugEvent+0x28
0746fecf 5be116a5 NatDbgDE!ProcessCreateProcessEvent+0x270
0746ff74 781329aa NatDbgDE!DmPollLoop+0x27
0746ffac 78132a36 MSVCR80!_callthreadstartex+0x1b
0746ffb4 7c80b683 MSVCR80!_threadstartex+0x66
0746ffec 00000000 kernel32!BaseThreadStart+0x37
```

上面清单中包含的 NatDbgDE 模块是 VC8 的本地代码调试引擎（Native Debug Engine）。

20.4 编译错误和警告

在编译过程中，编译器如果发现被编译代码中的问题，或者编译器自身出现了故障，它会以错误或警告的形式报告出来。为了更好地标志错误以便了解错误的来源和含义，编译器会为每个错误或警告信息赋予一个唯一的标识符（ID），这些标识符在编译器的各个版本间保持兼容。下面以 Visual C++ 编译器为例，介绍常见错误种类和错误标识符的编排方式。

20.4.1 错误 ID 和来源

首先，每个标识符是以 1 个或多个大写字母开头，后面跟 4 位阿拉伯数字，字母用来表示报告错误信息的组件，数字用来表示错误或警告的原因。表 20-2 列出了各类

错误的错误码范围。

表 20-2 错误 ID 的格式及来源

起始字符	主机	错误来源
Cxxxx	Compiler	编译器（前端和后端组件）
LNKxxxx	Linker	链接器
Dxxxx	Driver	CL.EXE
PRJxxxx	Project	项目构建工具，或者项目构建工具调用 custom build step 时发生错误
RCxxxx	Resource Compiler	资源编译器
Uxxxx		NMAKE

另外，对于同一来源的错误，处于不同范围的 ID 通常对应不同的严重级别，分为致命错误、错误和警告等。警告还可能分成多个级别，表 20-3 列出了各类错误的 ID 范围划分方法。

表 20-3 错误码的范围划分

错误来源	错误码范围
编译器	C999~C1999 为致命错误（fatal error），C2001~C3999 为错误，C4xxx 为警告
链接器	LNK4xxx 为警告（除 LNK4096 外），其他为链接错误
CL.EXE	D8xxx 为错误，D9xxx 为警告
NMAKE	U1xxxx 和 U2xxx 为错误，U4xxx 为警告
资源编译器	RC1xxx 为致命错误，RC2xxx 为错误，C4xxx 为警告
项目构建工具	没有规律

对于每个错误或警告，在编译器的手册中（或 MSDN）有详细的描述，通过 ID 号，就可以查找到这些信息。

20.4.2 编译警告

下面我们重点看看编译器的警告信息，这里的编译器是指严格意义上的编译器，即编译器的前端和后端。为了区分不同程度的警告，VC 编译器把警告信息分为 4 个级别（level），从 1 级到 4 级，严重程度逐渐递减。可以通过设置编译器的编译选项来配置如何显示编译警告。表 20-4 列出了这些设置所对应的命令行开关。

表 20-4 控制编译警告

开关	含义
/w	禁止所有编译警告
/Wn	显示严重程度最高为 n（n 为 0 到 4）的警告，W0 禁止所有警告，W4 显示所有警告
/Wall	启用所有警告

续表

开关	意义
/WIn	将 ID 值为 n 的编译警告的级别设为 1, 1 应该为 1 到 4 的整数
/wdn	禁止 ID 值为 n 的编译警告
/wen	启用 ID 值为 n 的编译警告
/WX	将所有编译警告视做错误
/won	只报告一次 (only once) ID 值为 n 的编译警告

通过表 20-3, 可以看到我们根据需要很方便地开启或关闭某一级别或某一个编译警告。

除了编译选项, 也可以在源代码中通过编译器的 #pragma warning 指令 (compiler directive) 来控制编译警告。比如, 如下两条语句分别会禁止 4705 号警告和将其恢复为默认设置:

```
#pragma warning( disable : 4705 )
#pragma warning( default : 4705 )
```

如下语句会禁止 4507 号和 4034 (小于 1000 的 ID 号会自动加上 4000) 号警告, 只报告一次 4385 号警告, 将 4164 号警告当作错误:

```
#pragma warning( disable : 4507 34; once : 4385; error : 164 )
```

可以用 #pragma warning(push) 将所有编译警告的当前设置保存起来, 而后再用 #pragma warning(pop) 恢复保存的值。

```
#pragma warning( push )
#pragma warning( disable : 4705 4706 4707 )
// Some code
#pragma warning( pop )
```

编译警告主要来源于编译器前端的语义分析组件, 语义分析可以从上下文的角度来检查代码的类型匹配情况和其他潜在的问题。为了及早发现代码中的设计问题, 程序员应该认真对待每个错误和警告信息。尤其是在项目的早期或寻找难以发现的错误时, 应该启用所有编译警告, 对于确认确实没有问题的警告, 可以在导致该警告的代码前使用 #pragma 指令暂时禁止警告, 在这段代码后再立即恢复该警告。

20.5 编译期检查

根据上一小节的介绍, 编译器在编译过程中会把被编译代码中所包含的问题以编译错误和编译警告的形式报告出来, 我们把这种编译器在编译阶段所作的检查称为编译期检查。编译是软件生产中必不可少的一步, 而且是比较早的一步。如果能在编译期发现软件设计中的问题, 那么可以节约测试时间, 降低开发成本, 对于软件工程有着重要意义。因为这个因素, 如何在编译期进行更有效的检查并将更多的问题消灭在“摇篮阶段”, 一直是编译器设计者和很多研究者的努力方向。

在编译的 6 个主要阶段中，词法分析可以发现非法字符、语句不完整等明显的错误。语法分析可以检查语句或表达式是否符合编程语言的语法规则。语义检查可以结合语句的上下文环境进一步检查变量的有效性（是否定义）、变量或参数的类型是否匹配等更深层次的问题。

综上所述，编译期检查可以发现代码中的词法、语法及少量语义方面的问题。为了发现更多的问题，可以通过编译器指令（compiler directives）手动加入检查，或者使用 VS2005 引入的标准标注语言（SAL）向代码中加入标注，再一种方法就是使用静态分析工具。

下面我们先介绍两种常见的编译期检查：未初始化的局部变量和类型不匹配，然后讨论后 3 种方法。

20.5.1 未初始化的局部变量

局部变量是指定义在某个函数内的变量，当编译器编译函数时会自动为其分配空间，分配的方法有两种，一种是在当前线程的栈上为其分配空间，另一种是将其映射到某个寄存器（更多讨论见第 22.4 节）。因为大多数局部变量是分配在栈上的，所以我们这里也只讨论这种情况。

在函数的入口处，编译器产生的代码会通过调整栈指针来为局部变量分配空间，在函数返回前，编译器产生的代码恢复栈指针的位置，从而释放这些空间。也就是说，从栈上分配和释放空间的过程，基本上就是调整栈指针的简单过程（不包括栈检查，我们将在第 22 章讨论）。从这个意义上来看，局部变量具有简单高效的优点。

当编译器编译调试版本时，会自动将所分配的局部变量区域全部初始化为一个固定的值，对于 x86 系统，这个值是 0xCC，即 INT 3 指令的机器码。当编译发布版本时，考虑到初始化工作需要空间和时间开销，编译器不会加入这些初始化代码。这样，如果程序员自己的代码也没有初始化某个局部变量，那么这个变量的值便是分配给它的栈空间中的本来值，这个值可能是前一个函数的局部变量所留下的，也可能是栈上的其他数值，或者说是随机的。这便导致了调试版本和发布版本的一个重要差异，这个差异也是导致调试版本和发布版本的运行行为不一致的一个原因。

使用一个随机的值很可能会埋下隐患，因此，使用局部变量前应该先将其初始化。如果没有这样做，编译器会发现并给出警告。例如，VC 编译器在编译以下函数时，便会给出 C4700 号警告。

```

int func()
{
    int i;
    return i;
}
C:\...\chap20\CompChk\mfc.c(10) : warning C4700: local variable 'i' used without
having been initialized

```

在 VS2005 中，除了给出 C4700 号警告，对于调试版编译器还会插入指令在运行到使用未初始化的局部变量的代码时，跳出警告对话框。因为理解该功能的工作原理需要对栈有比较深刻的理解，所以我们将在第 22.11 节详细讨论这个功能。

20.5.2 类型不匹配

编译器在进行语义分析时会检查变量比较、赋值等操作，目的是发现潜在的问题。比如，当有符号整数和无符号整数比较大小时，VC 编译器会给出如下警告：

```
warning C4018: '>' : signed/unsigned mismatch
```

当赋值语句可能丢失信息时，也会给出警告，比如当把一个双精度的浮点数赋给一个单精度类型的变量时，VC 编译器会给出如下警告：

```
warning C4244: '=' : conversion from 'double' to 'float', possible loss of data
```

当编译函数调用时，编译器会根据函数原型逐一检查每个参数的类型。另外，当向指针类型的变量赋值时，编译器也会做类型检查。

20.5.3 使用编译器指令

增加编译期检查的最简单做法是直接在头文件或源文件中通过编译器指令（compiler directives）加入特别的检查语句。当编译器编译这些语句时，会评估这些检查语句，如果检查的条件满足，那么编译器便会执行所定义的动作，通常是以编译错误的形式将异常情况报告出来。例如，MFC 类库的 AFX.H 头文件一开始便包含了如下检查语句：

```
// <afx.h>
#ifndef __cplusplus
    #error MFC requires C++ compilation (use a .cpp suffix)
#endif
```

该检查语句的含义是如果符号 __cplusplus 没有定义，那么便“执行”下面的#error 指令，显示指令后的错误信息。因为 C/C++ 编译器是根据被编译源文件的扩展名来决定是使用 C 还是 C++ 语言规范来编译，而且 MFC 是一套用 C++ 语言编写的库，所以，如果要使用 MFC，则必须使用 C++ 编译器，也就是要求包含 AFX.H（使用 MFC）的源文件必须具有.CPP 扩展名。如果一个扩展名为.c 的源文件包含了 AFX.H（示例见 code\chap20\compchk\mfc.c），那么编译时便会得到如下致命错误：

```
Compiling...
mfc.c
c:\program files\microsoft visual studio\vc98\mfc\include\afx.h(15) : fatal
```

```
error C1189: #error : MFC requires C++ compilation (use a .cpp suffix)
Error executing cl.exe.
```

上面的`#ifndef`、`#error` 和`#endif` 都是编译器指令，MSDN 描述了更多编译器指令及它们的详细用法。以下是 DDK 的 wdm.h 文件中使用编译器指令检查编译器版本的例子：

```
// <wdm.h>
#if _MSC_VER < 1300
#error Compiler version not supported by Windows DDK
#endif
```

`#if` 指令判断的条件表达式中可以使用通过`#define` 语句定义的符号，也可以使用在项目属性（Preprocessor Definitions）中定义的符号。

20.5.4 标注

增加编译期检查的另一种方法是向代码中加入标注信息（annotation）为编译器提供帮助，使其可以检查出更多的问题。标注的一个典型应用就是在声明函数原型时使用特定的符号标注函数的参数和返回值。这种方法很早便用在描述组件接口的 IDL（Interface Definition Language）语言中。VS2005 引入了一种名为标准标注语言（Standard Annotation Language，简称 SAL）的标注方法，定义了一整套标注符号和标注规范。我们将在下一节详细介绍。

20.5.5 驱动程序静态验证器（SDV）

增加编译期检查的第四种方法是使用更复杂的分析和检查算法，通过模式识别、数据挖掘等技术找出代码中满足知识库中所定义规则或与已知错误模式相匹配的“嫌疑”代码。例如，Windows Vista 的 WDK 中就包含了一个名为 SDV 的工具用于对驱动程序代码进行各种静态检查。

SDV 的全称是 Static Driver Verifier，即静态的驱动程序验证工具，说其静态一是因为 SDV 的功能是对驱动程序的源代码做编译期的静态检查；二是与对驱动程序进行运行期检查的驱动验证器（Driver Verifier）（第 19 章）相区别。

与驱动验证器相比，SDV 更复杂，整个工具由程序文件、规则文件、示例程序等近 300 个文件组成。SDV 的主目录位于 WDK 的 tools 目录下（例如：c:\winddk\6000\tools\sdv）。SDV 的主目录中包含了如下几个子目录。

- bin：SDV 的程序文件、C 编译器和分析引擎。
- data：用于存放 XML 数据文件。
- osmodel：存放了一些头文件和 C 代码用于模拟操作系统的函数。
- rules：用于存放规则文件，SDV 配备了 7 套针对驱动程序的规则。

- samples: 示例, 内部包含一个存在问题的驱动程序。

使用 SDV 进行检查的方法与编译驱动程序很类似, 启动一个带有 WDK 环境变量的命令行窗口后, 只要输入类似如下的命令, SDV 便会对当前目录下的驱动程序运行所有检查规则:

```
staticcdv /rule:*
```

如果只运行某个或某些规则, 那么可以指定规则的名称。

```
staticcdv /rule:Pnp*
```

也可以把规则放在一个文件中, 让 SDV 运行文件中所定义的所有规则, 这样的文件被称为规则列表文件。Samples 目录下包含了多个适用于不同用途的规则列表文件。例如如下命令会运行 IRQL.SDV 中定义的所有规则, 以对当前目录的驱动程序进行 IRQL (中断级优先级) 有关的各种检查:

```
staticcdv /config:%WDK%\tools\sdv\samples\rule_sets\wdm\irql.sdv
```

检查结束后, 可以使用 staticcdv/view 命令来观察 SDV 的检查报告。

PREFast 和 FxCop 是微软开发的另两个常用的静态分析工具。使用 PREFast, 可以分析 C/C++ 代码 (应用程序和驱动程序) 中的典型错误, 比如空指针引用、缓冲区溢出等。FxCop 可以分析托管代码中的潜在错误。PREFast 和 FxCop 都可以与 Visual Studio IDE 集成, 更多的信息请大家查阅 MSDN。

20.6 标准标注语言

VS2005 还引入了一种名为标准标注语言 (Standard Annotation Language, 简称 SAL) 的机制来帮助编译器和其他分析工具发现源代码中的安全问题, 提高代码的安全性。SAL 定义了一套标注 (Annotation) 符号和标注方法, 通过 SAL, 可以准确地描述出函数参数的特征及函数使用参数的方法。

VC2005 的头文件 sal.h 包含了所有 SAL 标注符号的定义, 以及每个符号的基本用法和示例。可以把 SAL 的标注符号分为两大类: 一类是用来描述函数参数和返回值的, 称为缓冲区标注符; 另一类称为高级标注符。

20.6.1 缓冲区标注符

缓冲区标注符用来描述函数的参数或返回值, 包括指针特征、缓冲区长度, 以及函数使用该参数 (或设置返回值) 的方法。一个缓冲区标注符可以描述一个参数, 而且每个参数也只能有一个缓冲区标注符。但一个缓冲区标注符可以包含多个元素, 分别用来描述参数的某个方面的特征。可以使用连接符 (_) 将多个元素连接在一起构成一个完整的缓冲区标注符。表 20-5 列出了目前版本的 SAL (VS2005) 所定义的用来

组成缓冲区标注符的基本选项标记（元素），以及每个选项的含义。

表 20-5 缓冲区标志符的组成元素

类别	选项	含义
指针的间接级别 (Indirection Level)	(none) <code>_deref</code> <code>_deref_opt</code>	用来描述参数或返回值的间接层次 (indirection level)。比如参数 p，如果不加限定，那么 p 便是缓冲区指针；如果加了 <code>_deref</code> ，那么 *p 是指向缓冲区的指针，所以 p 不该为空；如果加了 <code>_deref_opt</code> ，那么 *p 可能是指向缓冲区的指针，p 可以为空，也可以不为空
用法 (Usage)	(none) <code>_in</code> <code>_out</code> <code>_inout</code>	用来描述参数使用缓冲区的方法。如果不加修饰，则函数不会访问该缓冲区，如果加了 <code>_in</code> ，那么函数仅会读该参数，因此调用函数必须提供并初始化该缓冲区；如果加了 <code>_out</code> ，那么函数仅会写该缓冲区，如果与 <code>_deref</code> 一起用，或者用在返回值时，那么被调用函数应该提供并初始化缓冲区，否则调用者必须提供缓冲区，被调用函数对其初始化；如果有 <code>_inout</code> 修饰，那么函数可以自由读写该缓冲区，调用者必须提供并初始化缓冲区，如果与 <code>_deref</code> 合用，那么函数可以重新分配缓冲区
大小 (Size)	(none) <code>_ecount</code> <code>_bcount</code>	用来描述缓冲区的总大小。如果不加修饰，则不指定大小；如果加了 <code>_ecount</code> 描述，则缓冲区大小是以元素为单位来计数的，如果加了 <code>_bcount</code> 描述，则缓冲区大小是以字节为单位来计数的
输出 (Output)	(none) <code>_full</code> <code>_part</code>	描述函数初始化缓冲区的程度。 <code>_full</code> 表示完全初始化， <code>_part</code> 表示部分初始化；如果不加描述，则使用类型隐含表示初始化程度，比如对于 LPWSTR 类型，那么它一定是以 0 结束的
Null-Termination	<code>_z</code> <code>_nz</code>	描述是否使用 '\0' 来表示缓冲区中的最后一个有效元素
可选性 (Optional)	(none) <code>_opt</code>	描述缓冲区本身是否是可选的，如果不加修饰，那么指针一定要指向一个有效的缓冲区；如果加了 <code>_opt</code> ，那么指针可能为空
Parameters	(none) (size) (size, length)	描述参数所代表缓冲区的总长度 (size) 和已经初始化了的长度 (length)

下面举几个例子来帮助大家理解，先来看简单的用法：

```
void MyPaintingFunction(
    __in HWND hwndControl,           // 调用者初始化该参数，函数只会读这个参数
    __in_opt HDC hdcOptional,        // 调用者初始化的只读参数，可能为空
    __inout IPropertyStore *ppsStore // 调用者初始化该参数，函数可以读写
);
```

以下是使用 `deref` 和 `bcount` 的一个例子：

```
HRESULT SHLocalAllocBytes(size_t cb, __deref_bcount(cb) T **ppv);
```

其中，`__deref_bcount(cb)` 的含义是这个函数会把参数 `ppv` 的提领（dereference，即 `*ppv`）当作 `cb` 字节长的缓冲区来使用，而且使用前不会初始化这个缓冲区。如果初始化，则标注符中会包含 `_full` 或 `_part`。

函数声明是函数编写者和函数调用者之间的重要契约，SAL 无疑为更清楚地定义这个契约提供了一种简单有效的方法，VC2005 已经把大部分库函数的声明都加上了 SAL 标注。建议大家在设计函数时也尽可能地使用 SAL，这不仅可以提高代码的可读性，而且还为使用编译器或其他分析工具来扫描代码的潜在问题奠定了基础。

20.6.2 高级标注符

除了用来描述函数声明的缓冲区标注符外，SAL 还定义了一些高级的标注符（见表 20-6），以用来描述缓冲区标注符无法表达的约束或限定，实现更高级的编译期检查功能。

表 20-6 SAL 的高级标注符

<code>__success(expr) f</code>	修饰函数 <code>f</code> ， <code>expr</code> 为函数成功的条件。如果有此修饰，那么函数声明中所描述的承诺只有在函数成功时才是有保证（guarantees）的。如果函数前没有此修饰，那么函数应该始终保证其承诺。对于使用标准方式（如 <code>HRESULT</code> ）指示是否成功的函数，会被自动加上这个修饰
<code>__nullterminated p</code>	指针 <code>p</code> 是以 0 结束的缓冲区
<code>__nullnullterminated p</code>	指针 <code>p</code> 是以两个连续的 0 结束的缓冲区
<code>__reserved v</code>	保留， <code>v</code> 的值为 0 或 <code>NULL</code>
<code>__checkReturn v</code>	调用者必须检查函数的返回值 <code>v</code>
<code>__typefix(ctype) v</code>	将值 <code>v</code> 的类型当作 <code>ctype</code> ，而不是声明的类型
<code>__override f</code>	指定 C# 式的 <code>'override'</code> 行为，重载虚函数
<code>__callback f</code>	函数 <code>f</code> 可以用作函数指针
<code>__format_string p</code>	字符串 <code>p</code> 包含 <code>printf</code> 风格的 % 标记
<code>__blocksOn(resource) f</code>	资源 <code>resource</code> （如事件、信号量等）会阻塞函数 <code>f</code>
<code>__fallthrough</code>	修饰希望穿过（fall-through）效果的 <code>case</code> 语句，以便与忘记 <code>break</code> 语句的 <code>case</code> 情况相区别

以下是使用 SAL 的高级标注符来标注 Windows 的 `PathCanonicalize` API 的例子：

```
__success(return == TRUE) BOOL
PathCanonicalizeA(__out_ecount(MAX_PATH) LPSTR pszBuf, LPCSTR pszPath);
```

以上标注的含义是，`PathCanonicalizeA` 函数的成功返回值是 `TRUE`，只有当该函数返回 `TRUE` 时，才保证 `pszBuf` 指向的缓冲区是以 0 结束的合法字符串。`__out_ecount(MAX_PATH)` 的含义是，`pszBuf` 是包含 `MAX_PATH` 个元素（字符）的缓冲区，因为没有 `_deref` 修饰，所以是调用者提供该缓冲区，被调用函数只会写该缓冲区。

以下是 VS2005 的 malloc.h 中关于 malloc 函数的声明。

```
_CRTIMP _CRT_JIT_INTRINSIC _CRTNOALIAS _CRTRESTRICT // CRT 函数声明
__checkReturn                                // 调用者应该检查函数的返回值。
__bcount_opt(_Size)                          // 返回值的长度为 _Size 个字节，但可能为空。
void * __cdecl malloc(__in size_t _Size); // _Size 参数是只读的。
```

在笔者写作此内容时，尽管普通 VS2005 也能识别 SAL 标注(不会有未定义错误)，但是只有使用 Vistal Studio 2005 Team System 或 Visual Studio 2005 Team Edition for Developers 并使用/analyze 开关进行编译，SAL 标注才会真正被评估和检查。

20.7 本章总结

本章介绍了编译器的基本部件，编译的基本过程，并比较深入地讨论了编译错误和编译期检查。下一章我们将介绍编译器的运行库和运行期检查。

参考文献

1. 张素琴, 吕映芝等编著. 编译原理. 北京: 清华大学出版社, 2005
2. Major Changes from Visual C++ 4.2 to 5.0. Microsoft Corporation
3. Major Changes from Visual C++ 5.0 to 6.0. Microsoft Corporation
4. Major Changes from Visual C++ 6.0 to Visual C++ .NET. Microsoft Corporation
5. Microsoft Minimizes Threat of Buffer Overruns, Builds Trustworthy Applications. Microsoft Corporation
6. Microsoft Announces Visual C++ 5.0, Professional Edition. Microsoft Corporation
<http://www.microsoft.com/presspass/press/1997/feb97/vc5pr.mspx>

运行库和运行期检查

上一章我们介绍了编译期检查，利用编译期检查可以发现代码的词法、语法和部分语义错误。因为编译期检查分析的主要的是程序的静态特征，所以对于程序运行过程中才体现出的错误编译期检查通常是难以发现的。为了发现只有在运行期才显露出的问题，编译器通常还设计了运行期检查功能。编译器的运行库（Run-Time Library）是支持运行期检查的载体。

本章的前 3 节将简要地介绍运行库的基本概念（21.1 节）、链接运行库的方式（21.2 节）、运行库的加载和清理（21.3 节）。第 21.4 节将介绍基本的运行期检查，第 21.5 节介绍报告运行期检查所发现错误的方法。与栈和堆密切相关的运行期检查功能将分别在第 22 章和第 23 章介绍。

21.1 C/C++ 运行库

当编译器在将高级语言编译到低级语言的过程中时，因为高级语言中的某些比较复杂的运算符要对应比较多的低级语言指令，为了防止这样的指令段多次重复出现在目标代码中，编译器通常将这些指令段封装为函数，然后将高级语言的某些操作翻译为函数调用。比如 VC 编译器通常把 `n = f` (`n` 为整型，`f` 为浮点型) 这样的赋值编译为调用 `_ftol` 函数，把 `new` 和 `delete` 操作符编译为对 `malloc` 和 `free` 函数的调用。同时，为了增强编程语言的能力，加快软件开发速度，几乎所有编程语言都定义了相配套的函数库或类库，比如 C 标准定义的标准 C 函数、C++ 标准定义的 C++ 标准类库，这些库通常被称为支持库（support library）。支持库是编程语言和编译器的不可分割的部分，实现支持库是实现编译器的一项重要任务。

对于使用了某一支持库编译的程序来说，支持库是它们运行的必要条件，这些程序在运行时必须可以以某种方式找到支持库。出于这个原因，支持库有时也被称为运行库（Run-time Library）。以 VC 编译器为例，它同时提供了支持 C 语言的 C 运行库和支持 C++ 语言的 C++ 标准库，下面分别介绍之。

21.1.1 C 运行库

为了提高使用 C 语言的开发软件的效率，C 语言标准（ANSI/ISO C）定义了一系列常用的函数，称为标准 C 库函数，简称 C 库函数。C 标准定义了 C 库函数的原型和功能，但没有提供实现，把这个任务留给了编译器。每个编译器实现的通常是标准 C 函数库的一个超集，一般称为 C 支持库或 C 运行库（C Run-Time Library），简称 CRT。表 21-1 列出了微软编译器所实现的 C 运行库所包含的主要函数。

表 21-1 微软编译器的 C 运行库函数

参数访问	va_arg, va_start, va_end	访问函数的参数
浮点运算	powf, modf, acosf, ...	浮点计算
缓冲区处理	memcmp, memmove, memset, memcpy, ...	处理内存缓冲区
输入输出	vsnprintf, fwrite, fgetc, _read, _write, _commit, _getch, _outp, ...	读写文件，读写控制台及端口，低级 I/O
Byte classification	isleadbyte, mbsinit, ...	测试多字节字符的指定字节是否满足某一条件
国际化	localeconv, setlocale, ...	多语言和国际化支持
内存分配	new[], malloc, free, calloc, _set_new_mode, ...	内存分配、释放，我们将在第 23 章详细讨论内存有关的内容
数据对齐	_aligned_malloc, _aligned_free, ...	按照指定的数据对齐边界分配内存
进程和执行环境控制	abort, exit, _execvpe, _pipe, raise, signal, _spwenle, system, ...	创建终止进程，执行系统命令等
数据类型转换	_itoa, tolower, strtol, atof, mbtowc, labs, _gcvt, _fcvt, ...	数据转换
鲁棒性	set_terminate, set_unexpected, _set_new_handler, _set_se_translator	设置内存分配错误、异常终止处理函数，提高程序的鲁棒性
调试	_ASSERT, _CrtCheckMemory, _CrtDbgBreak, _RPT, _free_dbg	断言、内存分配检查、报告运行期检查错误，以及其他调试支持，我们将在下一节详细讨论这些函数
运行期错误检查(Run-time Error Checking)	_RTC_GetErrDesc, _RTC_NumErrors, _RTC_SetErrorFunc, _RTC_SetErrorHandler	读取运行期检查发现的错误，定制检查方式
目录控制	_chdrive, _getcwd, _getdrive, _mkdir, _chdir	获取目录信息，切换目录(即文件夹)
搜索及排序	bsearch, _lfind, qsort, ...	搜索及排序
错误处理	_set_error_mode, _set_purecall_handler, clearer, ...	设置错误处理函数和错误报告模式
操纵字符串	strcoll, strcat, strcspn, wcscspn, _mbscspn, ...	处理字符串，包括单字节字符串、多字节字符串和宽字符串
异常处理	_set_se_translator, set_unexpected, terminate, unexpected, ...	设置终止和异常处理函数，处理异常
系统调用	_findfirst, _findnext, ...	调用操作系统的服务
文件处理	_filelength, _fstat, _set_mode, _fullpath, _remove, _makepath, ...	创建、删除、操纵文件，设置和检查文件访问权限
时间管理	ctime, clock, asctime, difftime, strftime, ...	读取时间，转换，格式化时间

通过上表可以看出，C 支持库包含了从内存分配、错误处理、字符串处理、浮点计算、数据类型转换、文件和输入输出等方面大量的函数，这些函数有些是 C 标准所定义的标准 C 函数，有些是针对 Windows 操作系统而特别设计的，并不属于标准 C 函数。MSDN 更全面地列出了每类函数，并详细描述了每个函数。

在 VC 编译器的安装目录中可以找到 CRT 函数的源程序文件，包括头文件和.c 文件。对于 VC6，其默认目录是 c:\Program Files\Microsoft Visual Studio\VC98\CRT\SRC\，对于 VC8，其默认位置是 c:\Program Files\Microsoft Visual Studio 8\VC\crt\src\。

21.1.2 C++标准库

与 C 标准类似，C++的国际标准（ISO C++）也是既包含对 C++语言本身的定义，又规定了 C++标准库的内容。C++标准库是为了方便使用 C++语言编程而设计的一套函数、常量、类和对象库，包括标准输入输出、字符串、容器（列表、队列、map 等）、排序和搜索算法、数学运算等。

C++标准库由三大部分组成，第一部分是 C 标准库，基本上是表 21-1 所列出的 C 标准支持函数；第二部分是 IO 流(iostream)；第三部分是标准模板库(Standard Template Library)，即 STL。

清单 21-1 给出了一个使用 IO 流和 STL 容器类的 C++程序 CppSLib（完整源代码位于\code\chap21\cppslib 文件夹）。

清单 21-1 使用 C++标准类库的示例程序 CppSLib

```
// CppSLib.cpp : 用于探索支持库的示例程序
//
#include "stdafx.h"
#include <iostream>
#include <string>
#include <vector>
using namespace std;
typedef vector<int> INTVECTOR;
int main(int argc, char* argv[])
{
    INTVECTOR iv;
    string s="Explore C++ Standard Library";
    int i;

    for (i = 0; i < s.length(); i++)
        iv.push_back(s.at(i));

    INTVECTOR::iterator iter;
    // 输出 iv 的内容
    cout << "[" ;
    for (iter = iv.begin(); iter != iv.end();
         iter++)
    {
        cout << (char)*iter;
        if (iter != iv.end()-1)
            cout << "-";
    }
}
```

```

    cout << " ]" << endl ;
    cin>>i;
    return i;
}

```

使用 depends 工具观察 VC6 编译器编译好的 CppSLib.exe（见图 21-1），从图的左侧可以看到 CPPSLIB 程序依赖于 MSVCRT.DLL 和 MSVCP60.DLL，其中 MSVCRT.DLL 是 VC6 编译器的 C 运行库 DLL，MSVCRT 是 MS Visual C Runtime 的缩写，MSVCP60.DLL 是 VC6 编译器的 IO 流和 STL 库的 DLL，MSVCP60 是 MS Visual C Plus Plus (C++) 6.0 的缩写。图中右侧的列表显示了 MSVCP60 所输出的 basic_string、basic_istream 和 basic_ostream 类的部分成员和方法。

使用 VC8 编译上面的代码，可以看到与图 21-1 非常类似的情形，只不过是两个 DLL 变为 VC8 版本的两个支持库：MSVCR80.DLL 和 MSVCP80.DLL。

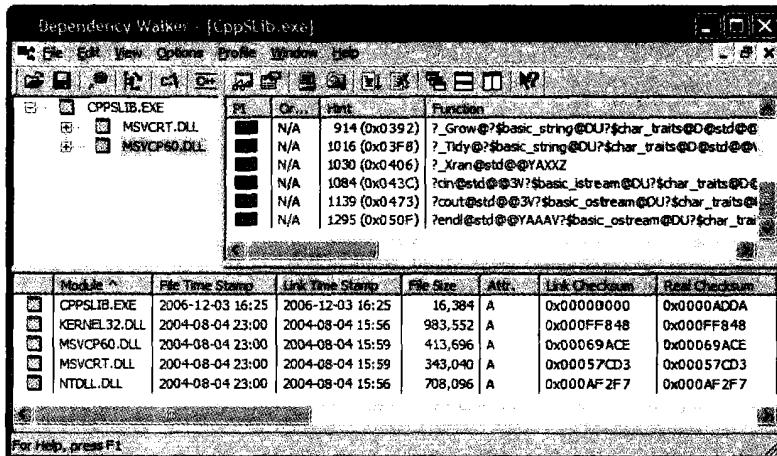


图 21-1 观察 VC6 编译器的支持库 DLL

通过以上分析，可以看出 VC 编译器是将 C++ 编译器所使用的 C 标准库与 C 编译器所使用的 C 运行库一起实现的（即 MSVCRT 或 MSVCR80），而把 IO 流和标准模板库单独来实现（即 MSVCP60 或 MSVCP80）。为了行文方便，我们用 MSVCR 来泛指 VC 编译器的 C 运行库，用 MSVCP 来泛指 VC 编译器的标准 C++ 类库。

21.2 链接运行库

为了满足不同情况的需要，编译器的运行库通常有多个版本。举例来说，为了辅助软件调试，编译器的支持库包含了很多供调试使用的函数和变量，比如表 21-1 所列出的断言、内存检查、RTC 函数等。为了使这些专门用于调试的函数不影响软件发布后的性能，编译器通常将大多数调试支持只放在调试版本的支持库中。调试版的支持库除了增加一些发布版中没有用于调试的函数之外，调试版还在很多同样存在于发布版的函数中加入了更多的错误检查和辅助调试代码。

调试版本和发布版本的支持库是共享源代码的，但是.lib 文件和 DLL 文件是不同的。表 21-2 列出了 VC6、VC7 和 VC8 的 C 运行库 DLL、标准 C++类库的发布版本 DLL，以及对应的调试版本 DLL。

表 21-2 VC 编译器的支持库 DLL

文件名	发布版	调试版
VC6 编译器的 C 运行库 DLL	MSVCRT.DLL	MSVCRTD.DLL
VC6 编译器的 C++类库 DLL	MSVCP60.DLL	MSVCP60D.DLL
VC7 编译器的 C 运行库 DLL	MSVCR71.DLL	MSVCR71D.DLL
VC7 编译器的 C++类库 DLL	MSVCP71.DLL	MSVCP71D.DLL
VC8 编译器的 C 运行库 DLL	MSVCR80.DLL	MSVCR80D.DLL
VC8 编译器的 C++类库 DLL	MSVCP80.DLL	MSVCP80D.DLL

我们将在第 23 章进一步讨论如何使用调试版支持库的内存（堆）检查功能。

21.2.1 静态链接和动态链接

为了使编译好的程序可以顺利运行，使用运行库的程序在运行时必须可以找到库中的函数。实现这一目的有两种方法，一种是静态链接，另一种是动态链接。

简单来说，静态链接就是将程序所使用的支持库中的函数复制到程序文件中。这样一来，这些支持库函数的实现就位于程序模块中，成为本模块中的代码。

当在 VC6 中创建一个控制台类型的项目时，默认的配置是静态链接 C 运行库。对于这样的程序，调试时我们可以在程序模块中看到静态链接的支持库函数。例如，清单 21-2 中显示出的符号便都是静态链接到可执行程序模块（HelloVC6）的支持库变量和函数。

清单 21-2 链接到可执行文件中的支持库函数

```
0:000> x hellovc6!*crt*
00428114 HelloVC6!_crtheap = 0x00370000
00424e20 HelloVC6!_crtDbgFlag = 1
00424e28 HelloVC6!_crtBreakAlloc = -1
00424cc0 HelloVC6!_crtAssertBusy = -1
...
```

因为静态链接必须把支持库中的函数复制到目标程序中，所以产生的程序文件会比较大。对于由多个模块组成的较大型软件来说，如果属于同一个进程的几个模块都选择了静态链接 C 运行库，那么在这个进程中，某些 C 运行库的函数便会重复存在于多个模块中。

动态链接是利用动态链接库技术，在程序运行时再动态地加载包含支持函数的动态链接库（DLL），并更新程序的 IAT（Import Address Table）表，使程序可以顺利地调用 DLL 中的支持函数。使用 depends 工具观察到的依赖模块就是这个程序选择与之动态链接的那些模块。当运行一个程序时，如果操作系统的加载器找不到它所依赖的模块，那么会显示一个类似图 21-2 的消息框。

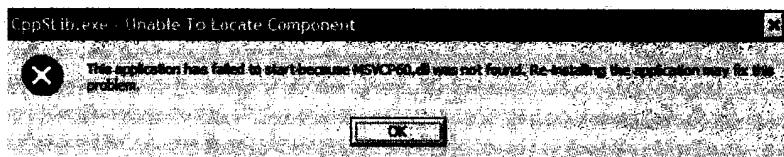


图 21-2 操作系统的加载器找不到依赖模块 (MSVCP60.DLL) 时显示的对话框

可以通过编译器选项来设置链接支持库的方式。`/MT` 开关代表静态链接，`/MD` 代表动态链接。如果要使用调试版本的支持库，则在后面加一个小写的 `d`，即`/MTd` 或 `/MDd`。VC6 的支持库还配有不带多线程支持的版本，如果要使用这个版本，则使用选项`/ML` 或 `/MLd`，二者都是静态链接。VC8 不再支持单线程版本的支持库和选项，也就是说，VC8 只提供了带有多线程支持的支持库。

图 21-3 显示了 VC 编译器的项目属性中关于链接支持库的所有选项，左侧是 VC6 编译器所使用的，右侧是 VC8 的。值得注意的是，这个选项是在 C/C++ 编译选项下的 Code Generation 页中，并不在链接器 (Linker) 选项下。另一点要说明的是，界面上显示的是使用运行库 (Use run-time library) 的方式，这里的运行库既包括 C 运行库，又包括标准 C++ 类库，也就是说，只能以同一种方式来使用这两个库，不可以为其中某一个定义不同的链接方法。

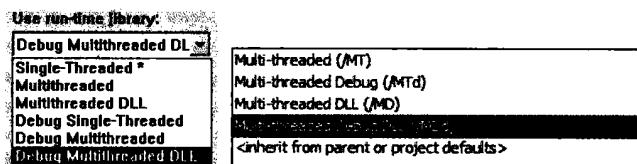


图 21-3 VC6 (左) 和 VC8 (右) 的 CRT 链接选项

以上选项都是对于 C/C++ 本地程序的。对于.NET 程序，VC8 定义了`/clr` 和 `/clr:pure` 两个选项，前者输入本地的 C 运行库，产生的程序为本地代码和托管代码相混合；后者将 C 运行库也编译为.NET 字节码，产生的程序为 100% 的托管代码。

21.2.2 Lib 文件

无论是静态链接还是动态链接，都离不开 lib 文件。VC 编译器的 lib 目录（如 c:\Program Files\Microsoft Visual Studio 8\VC\lib\）包含了链接运行库所需的各种 lib 文件。表 21-3 显示了 VC8 所使用的用来链接运行库的各个 lib 文件和它们的典型大小及用途。

表 21-3 链接运行库的 lib 文件 (VC8)

<code>libcmt.lib</code>	7,716,360	静态链接 C 运行库，mt 的含义是多线程支持
<code>libcmtd.lib</code>	9,758,380	静态链接调试版本的 C 运行库
<code>libcpmt.lib</code>	4,991,534	静态链接标准 C++ 类库
<code>libcpmtd.lib</code>	6,703,474	静态链接调试版本的标准 C++ 类库
<code>msvcrt.lib</code>	939,484	动态链接 C 运行库

续表

文件名	大小(字节)	用途
msvcrt.lib	974,050	动态链接调试版本的 C 运行库
msvcprt.lib	1,480,526	动态链接标准 C++ 类库
msvcprtd.lib	1,623,976	动态链接调试版本的标准 C++ 类库
msvcmr.lib	967,418	以混合方式链接 C 运行库, m 的含义是 mixed 的
msvcmttd.lib	1,004,566	以混合方式链接调试版本的 C 运行库
msvcurt.lib	5,958,948	生成纯托管代码, urt 的含义是 Universal Runtime
msvcurtd.lib	6,632,868	生成调试版的纯托管代码

表 21-3 中的文件大小只是想让大家看出不同用途的 lib 文件的大小差异, 其具体数值会因为 VC 编译器的版本不同而不同。

21.3 运行库的初始化和清理

在前面两节, 我们讨论了运行库的作用和链接方式。本节将继续讨论运行库是如何工作的, 包括介入到应用程序中的方法, 以及如何进行初始化和清理。

21.3.1 介入方法

首先, 在什么时候执行初始化运行库的代码呢? 或者说在哪里调用运行库的初始化函数呢? 因为很多运行库函数和设施是运行其他代码的基础, 比如 CRT 堆必须在其他代码调用 CRT 的内存分配函数 (malloc 函数等) 之前被初始化好。这意味着, 必须在执行用户代码 (应用程序开发者所编写的代码, 相对于编译器的运行库代码) 之前初始化运行库。我们知道程序是从入口函数开始执行的, 这也就是说, 要在入口函数的开始处或之前调用运行库的初始化函数。要求每个应用程序开发者调用运行库的初始化函数虽然可以, 但是不够好。

事实上的做法是, 编译器会为每个模块自动插入一个“编译器编写”的入口函数, 在这个入口函数中进行好各种初始化工作后再调用用户的入口函数, 在用户的入口函数返回后再运行自己的清理函数。我们把编译器插入的这个入口函数称为 CRT 入口函数。其伪代码示意如下:

```
CRT 入口函数()
{
    CRT 初始化
    调用用户的入口函数
    CRT 清理
}
```

对于 EXE 模块, 因为程序类型的不同, 用户的入口函数名也有所不同, 通常为表 21-4 第一列中的某一个, 相应的, 编译器为每种类型的用户入口函数准备了 CRT 入口函数 (第二列)。

表 21-4 EXE 模块的入口函数一览

用户入口函数	CRT 入口函数	应用
main	mainCRTStartup	控制台程序
wmain	wmainCRTStartup	宽字符的控制台程序
WinMain	WinMainCRTStartup	Win32 应用程序
wWinMain	wWinMainCRTStartup	宽字符的 Win32 应用程序

DLL 模块的用户入口函数是 DllMain，因为 DllMain 没有字符类型的参数，所以对于宽字符（Unicode）的 DLL 模块，其入口函数也是 DllMain。

```
BOOL WINAPI DllMain(
    HANDLE hDllHandle,           //模块句柄，即地址
    DWORD dwReason,             //调用原因
    LPVOID lpreserved );
```

编译器为 DLL 模块准备的 CRT 入口函数是_DllMainCRTStartup 函数，其函数原型与 DllMain 完全相同。

在链接阶段，模块的入口函数是被注册到目标程序文件（PE 文件）中的，默认情况下，链接器会将 CRT 入口函数注册为模块的入口，这样模块执行时，首先执行的是 CRT 的入口函数，它可以完成运行库的初始化工作后再调用用户的入口函数。等用户的入口函数返回后，CRT 的入口函数可以执行清理工作。不过，链接器支持通过 /ENTRY:function 选项指定其他函数作为入口。如果在项目链接属性中（Output）将 Entry-point symbol 指定为 WinMain 函数名，或者直接在命令行参数中加入 /ENTRY:"WinMain"，那么便将 WinMain 函数设置为程序的入口了。此时，就须要用户自己考虑如何初始化和清理运行库了，这种做法是不推荐的。

21.3.2 初始化

在 CRT 源代码的 crt0.c 中包含了 CRT 入口函数的源代码，从中我们可以看到执行各种初始化的代码。概括来说，CRT 入口函数执行的初始化工作主要有如下几项。

- 调用 __security_init_cookie() 初始化安全 Cookie。
- 初始化全局变量，包括环境变量和标识操作系统版本号的全局变量等。
- 调用 _heap_init() 函数创建 C 运行库所使用的堆，简称 CRT 堆，程序中调用 malloc 等函数时所得到的内存便是从这个堆中分配的。_heap_init 函数是专门用来创建和初始化 CRT 堆的。
- 调用 _mtinit() 函数初始化多线程支持。
- 如果使用 VC7 引入的运行期检查功能（简称 RTC，下一节详细介绍），那么需要调用 _RTC_Initialize()。
- 调用 _ioinit() 函数初始化低级 IO。
- 调用 _cinit() 函数初始化 C 和 C++ 数据，包括初始化浮点计算包、除 0 向量（已

经过时), 以及调用 _initterm(__xi_a, __xi_z) 执行注册在 PE 文件数据区的初始化函数, 调用 _initterm(__xc_a, __xc_z) 执行 C++ 的初始化函数, 例如全局 C++ 对象的构造函数。

我们将在第 22.9 节详细讨论安全 Cookie, 第 23 章讨论 CRT 堆, 下面将集中介绍 CRT 是如何初始化数据(全局变量)的。

在 crt0dat.c 中包含了 _cinit() 函数的源代码, 其中最主要的代码便是以下两次函数调用:

```
_initterm( __xi_a, __xi_z ); // do initializations
_initterm( __xc_a, __xc_z ); // do C++ initializations
```

其中 __xi_a、__xi_z 和 __xc_a、__xc_z 是两对特殊的全局变量, 分别指向两个函数指针表的起始点和结束点。_initterm 函数的实现也非常简单, 它只是遍历参数中指定的函数指针表, 调用其中的每个函数。编译器在编译时, 会为需要初始化的数据产生一个函数, 为这个函数定义一个函数指针, 并将这个函数指针做一个类似下面的段声明:

```
CRT$XCU SEGMENT
$_$S3 DD FLAT:_$E2
CRT$XCU ENDS
```

这样, 当链接器链接可执行文件时便会将这个函数指针(\$_\$S3)放入到 PE 文件的特定位置, 使其位于上面介绍的函数指针表中, 也就是将变量 \$_\$S3 放在 __xc_a 和 __xc_z 之间, 值得说明的是, __xc_a 和 __xc_z 只是用来标识函数指针表的起至, 并不代表这个表中只能有 a~z 个元素。

下面通过一个实例来加深印象。在 HelloVC6 程序(code\chap21\HelloVc6)中, 我们定义了一个全局对象, Cat 类的实例 c。清单 21-3 中的栈回溯(由 VC6 调试器产生)显示了实例 c 的构造函数被调用的过程。

清单 21-3 构造 C++ 全局对象的过程

Cat::Cat()	line 11 + 22 bytes	// 对象 c 的构造函数
\$E1()	line 14 + 34 bytes	// 编译器为全局对象 c 产生的初始化函数
\$E2()	+ 29 bytes	// 编译器产生的变量初始化函数
_initterm(void ** 0x00424104 \$S3, void ** 0x00424208 __xc_z)	line 525	
_cinit()	line 192 + 15 bytes	// 运行库函数
mainCRTStartup()	line 205	// 编译器插入的入口函数
KERNEL32! 7c816ff7()		// 系统的进程启动函数 kernel32!BaseProcessStart

其中, \$E2() 是编译器编译时产生的简单函数, \$S3 是指向函数 \$E2 的函数指针。函数 \$E1() 也是编译器产生的, 它对应的就是定义全局对象的源代码行, 清单 21-4 包含了 \$E1() 的汇编代码。

清单 21-4 编译器为全局对象产生的函数 \$E1()

14: Cat c;		// 对应的源代码(全局对象定义)
00401090 push	ebp	// 保存父函数的栈帧地址
00401091 mov	ebp, esp	// 建立本函数的栈帧地址
...		// 省略保存寄存器的多行
004010A8 mov	ecx, offset c (00427f68)	// 将对象地址(this 指针)赋给 ECX

```

004010AD  call      @ILT+15(Cat::Cat) (00401014) //调用构造函数
...
004010B8  cmp      ebp,esp //省略恢复寄存器的多行
004010BA  call      __chkesp (00401370) //调用栈指针检查函数
004010BF  mov      esp,ebp //恢复栈指针寄存器
004010C1  pop      ebp //恢复父函数的栈帧地址
004010C2  ret      //返回

```

在 CRT 源代码的 cinitexe.c 中，我们可以看到函数指针表的起至变量（`_xi_a` 等）的定义（见清单 21-5）。

清单 21-5 向 PE 文件的数据段注册初始化函数（cinitexe.c）

```

#pragma data_seg(".CRT$XIA")
PFV _xi_a = 0; /* C initializers */
#pragma data_seg(".CRT$XCA")
PFV _xc_a = 0; /* C++ initializers */
#pragma data_seg(".CRT$XPA")
PFV _xp_a = 0; /* C pre-terminators */
#pragma data_seg(".CRT$XTA")
PFV _xt_a = 0; /* C terminators */
...
#pragma data_seg() /* reset */

```

以 `data_seg(".CRT$XIA")` 为例，点之后的三个字符是数据段名称，即 CRT；\$ 是分隔符，其后的第二位代表类别，“I”的含义是 C 初始化（C init），'C'代表 C++ 初始化，'P'代表 Pre-terminators（在终结器之前执行），'T'代表终结器（Terminators）。

理解了 initterm 函数的原理后，也可以在自己的程序文件中通过以上机制来注册初始化和清理函数。比如如下代码便注册了 `myinit` 函数在 C 初始化阶段执行。

```

int myinit()
{
    MessageBox(NULL, "my init function test", "_initterm ", 0);
    return 0;
}
typedef int cb(void);
#pragma data_seg(".CRT$XIU")
static cb *autostart[] = { myinit };
#pragma data_seg()

```

将以上代码插入到一个 HelloVC6 程序中（code\chap21\HelloVc6）执行时，`myinit` 便会在 `main` 函数执行前被执行。

21.3.3 多个运行库实例

下面我们考虑一个进程中的多个模块都使用了运行库的情况。在 Windows 程序中，这是很普遍的。

对于静态链接运行库的程序模块，不论是 EXE 还是 DLL，每个模块内部都复制了一份运行库的变量和代码（部分）。也就是说，在这样的每个模块中都有一个运行库的实例。对于 DLL 项目，VC6 的默认设置也是静态链接运行库。清单 21-6 显示了静态链接 CRT 的 DLL 模块 HpDllVC6 所包含的 CRT 变量和函数。

清单 21-6 观察静态链接 CRT 的 DLL 模块所包含的 CRT 变量和函数

```
0:000> x HpDllVC6!*crt*
10034464 HpDllVC6!_crtheap = 0x00370000
1002ea5c HpDllVC6!_crtDbgFlag = 1
1002ea64 HpDllVC6!_crtBreakAlloc = -1
1002ec74 HpDllVC6!_crtAssertBusy = -1
...
```

当进程中多个运行库实例时，每个运行库实例会各自使用自己的数据（变量）和资源（堆），理论上是可以正常工作的。但有时也可能引发问题，其中之一就是从一个 CRT 堆分配的内存被送给另一个 CRT 实例来释放，在调试版本中，CRT 的内存检查功能会发现这一问题并报告错误，但在发布版本中，这便会导致严重的问题。

21.4 运行期检查

顾名思义，运行期检查（Run-time Check）就是在程序运行期间对其进行的各种检查。更多时候，运行期检查是与编译期检查相对的，二者都是与编译器密切相关的概念。特别是在本章中，我们讨论的只是编译器所提供的运行期检查功能。

运行期检查的目的是发现程序在运行时所暴露出的各种错误，即运行期错误（Run-time Errors）。因为程序运行是编译过程结束后才发生的事情，所以要实现运行期检查，编译器通常采取如下几种措施。

1. 使用调试版本的支持库和库函数，调试版本的库函数包含了更多的调试支持和检查功能。比如调试版本的内存分配和释放函数会插入额外的信息来支持各种内存检查功能（详见第 23 章）。
2. 在编译时插入额外代码对栈指针、局部变量等进行检查。
3. 提供断言（ASSERT）、报告（_RPT）等机制让程序员在编写程序时加入检查代码和报告运行期错误。

前两种检查是由编译器的运行库和编译过程自动提供的，我们称其为编译器的自动（运行期）检查。第三种检查需要程序员插入代码来调用编译器提供的宏或函数，我们称其为手工插入的（运行期）检查。

运行期检查需要额外的空间和时间开销，所以，为了避免影响软件产品期的性能，大多数运行期检查只用于调试版本。比如_ASSERT 和_RPT 等宏定义只有在调试版本中才有作用。

不同的编译器所提供的运行期检查功能有所不同，下面我们便以 VC8 编译器为例，介绍它的运行期检查功能。我们首先介绍自动的运行期检查，然后介绍断言。下一节将介绍报告运行期检查所发现错误的方法。

21.4.1 自动的运行期检查

VC8 编译器可以自动检查以下运行期错误。

1. 栈指针被破坏 (Stack pointer corruption.), 负责栈指针检查的函数是 _RTC_CheckEsp。
2. 分配在栈上的局部变量越界 (Overruns), 以及因此而导致的栈被破坏 (Stack corruption.)。
3. 依赖未初始化过的局部变量, 负责该功能的 RTC 函数是 _RTC_UninitUse。
4. 因为赋值给较短的变量导致数据丢失, 负责该项检查的 RTC 函数是 _RTC_Check_x_to_y, 其中 x 和 y 可以是 8、4、2、1 几个值的组合, 如 _RTC_Check_2_to_1 (short)、_RTC_Check_4_to_1 (int)、_RTC_Check_4_to_2 (int)、_RTC_Check_8_to_1 (int64)、_RTC_Check_8_to_2 (int64)、_RTC_Check_8_to_4 (int64)。
5. 使用堆有关的错误。

以下几个编译器开关用来控制运行期检查功能。

/RTCs 栈检查, 包括栈指针检查 (VC6 的/GZ 开关), 栈上的变量是否被破坏等。负责栈上变量检查的函数是 _RTC_CheckStackVars。

/RTCu 局部变量检查, 如果程序中使用了没有初始化过的局部变量, 那么编译器在编译期会给出 C4700 或 C4701 号警告。但因为 C4700 或 C4701 号警告属于第 4 级别的警告 (Level 4 Warning), 所以有可能被屏蔽掉或被忽视。为此, 如果启用了/RTCu 开关, 那么 VC8 编译器会插入检查代码, 当程序运行到使用未初始化过的局部变量时, 会弹出错误报告对话框。

/RTCc 数据赋值时的截断检查, 比如把较大的数值赋给一个较短的变量。如果编译器的警告级别为 W4 或设置了/WX 开关 (将警告视为错误), 那么编译器在编译时会给出 C4244 号警告。

/RTC1, 相当于同时设置了/RTCs 和/RTCu。

/GS, 缓冲区溢出, 该检查在发布版本中仍起作用。

/GZ, VC6 中用来启动栈指针检查功能。VC8 将该开关的功能合并到/RTCs。

为了描述方便, 我们经常使用以上编译器开关来指代它所对应的检查功能, 并将 /RTC 开头的几种检查泛称为 RTCx。

如果指定了 RTCx 中的任一个选项, 那么编译器便会定义宏 __MSVC_RUNTIME_CHECKS。因此可以通过检查这个宏是否定义来判断是否启用了 RTCx 中的功能。

因为要理解 RTCx 和 GS 检查功能的工作原理, 需要对栈布局有较深的理解, 所以我们将在第 22 章再讨论该内容, 并在第 23 章中讨论堆有关的错误检查。

21.4.2 断言 (ASSERT)

断言 (ASSERT) 是程序员手工插入运行期检查的一种常用方法。通常用来检查某一条件是否成立。要断言的条件以参数的形式传递给断言宏，如果该参数表达式的结果为真，那么断言成功并直接返回，否则断言失败，会弹出类似图 21-4 所示的断言消息框 (assert message box)。

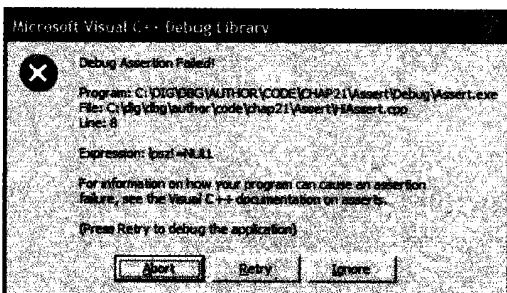


图 21-4 断言消息框

断言消息框通常包含断言发生的源文件名称 (HiAssert.cpp)、行号 (Line: 8) 和被检查的表达式 (lpsz!=NULL)，使程序员可以一目了然地看到哪里出了问题。

VC 运行库 (CRT) 定义了两个宏来提供断言功能，分别是 _ASSERT 和 _ASERTE，名字中的下画线表示这是 VC 编译器的扩展，不属于 C/C++ 标准。在头文件 <crtdbg.h> 中可以找到这两个宏的定义，如清单 21-7 所示。

清单 21-7 _ASSERT 和 _ASERTE 的宏定义 (调试版本) (crtdbg.h)

```
#define _ASSERT(expr) \
    do { if (!(expr) && \
        (1 == _CrtDbgReport(_CRT_ASSERT, __FILE__, __LINE__, NULL, NULL))) \
        _CrtDbgBreak(); } while (0)

#define _ASERTE(expr) \
    do { if (!(expr) && \
        (1 == _CrtDbgReport(_CRT_ASSERT, __FILE__, __LINE__, NULL, #expr))) \
        _CrtDbgBreak(); } while (0)
```

在上面的定义中，do{}while(0)是为了将大括号中的多条语句封装成一个整体，使这个宏可以在各种环境下（比如 if 语句后）都可以正确使用，并不是为了实现循环。因此 _ASSERT 宏与清单 21-8 中的代码片段是等价的。

清单 21-8 _ASSERT 宏展开后的等价代码

```
if (!(expr) )
{
    int nRet = _CrtDbgReport(_CRT_ASSERT, __FILE__, __LINE__, NULL, NULL);
    if (nRet==1)
        _CrtDbgBreak();
}
```

也就是说，如果被断言的表达式 (expr) 的结果为 0，那么便调用 _CrtDbgReport 函数。_CrtDbgReport 是 VC 运行库中用来报告 CRT 事件的一个重要函数，可以把事

件信息输出到调试文件、调试器的监视窗口，或者弹出一个消息窗口（类似图 21-4），我们将在下一节详细讨论这个函数。

比较这两个宏的定义，可以看到它们只是断言失败后调用 `_CrtDbgReport` 的方式不同，`ASSERTE` 会将断言表达式传递给 `CrtDbgReport` 函数，而 `ASSERT` 不会。因此，使用 `ASSERTE` 宏产生的断言消息框中包含断言表达式（图 21-4），而 `ASSERT` 宏产生的不包含。下面通过一个示例程序（`code\chap21\Assert`）来进一步说明断言宏的原理和用法。先来看下面的 `Foo` 函数：

```
void Foo(char * lpsz)
{
    _ASSERTE(lpsz!=NULL);
    //...
}
```

在 `main` 函数中我们使用 `NULL` 为参数调用这个函数，使断言失败。在 VC 调试器中执行以上调用，对于弹出的断言对话框，我们选择重试。于是调试器停在 `_ASSERTE(lpsz!=NULL)` 行。显示反汇编，可以看到 `ASSERTE` 宏被展开后所对应的汇编语句（见清单 21-9）。

清单 21-9 `_ASSERTE` 宏所对应的汇编语句

1	11:	<code>_ASSERTE(lpsz!=NULL);</code>	
2	00401048	cmp	dword ptr [ebp+8],0
3	0040104C	jne	Foo+45h (00401075)
4	0040104E	push	offset string "lpsz!=NULL" (00423058)
5	00401053	push	0
6	00401055	movsx	eax,word ptr [`Foo'::`2':__LINE__Var (00425a30)]
7	0040105C	add	eax,1
8	0040105F	push	eax
9	00401060	push	offset string "C:\...\chap21\As"... (0042301c)
10	00401065	push	2
11	00401067	call	<code>_CrtDbgReport</code> (004013c0)
12	0040106C	add	esp,14h
13	0040106F	cmp	eax,1
14	00401072	jne	Foo+45h (00401075)
15	00401074	int	3
16	00401075	xor	ecx,ecx
17	00401077	test	ecx,ecx
18	00401079	jne	Foo+18h (00401048)

第 2 行是判断被断言的条件（`[ebp+8]` 即参数 `lpsz`），如果不等，则跳到第 16 行，第 16~18 行对应于 `while(0)`，因此相当于继续向下执行。如果第 2 行的判断结果为相等，那么断言失败，于是开始执行第 4 行，第 4~11 行是调用 `_CrtDbgReport` 函数，报告断言失败，默认情况是弹出断言消息框。因为 `_CrtDbgReport` 函数采用的是 C 调用协议，需要调用它的函数清理栈上的参数，所以第 12 行是为了清理前面压入栈的 5 个参数（共 20 个字节）。第 13 行判断 `_CrtDbgReport` 函数的返回值（在 `Eax` 中），如果等于 1（即用户选择了重试），那么便执行第 15 行的 `int 3`。如果用户选择忽略，那么返回值为 0，第 14 行的条件转移语句会使 CPU 跳转到第 16 行，使程序继续执行。

`ASSERT` 和 `ASSERTE` 宏只在调试版本中起作用，在发布版本中，这两个宏被定义为空（清单 21-10）。因此使用这两个宏的断言语句不会被编译到发布版本的目标程序中。

清单 21-10 _ASSERT 和 _ASERTE 的宏定义（发布版本）(crtdbg.h)

```
#define _ASSERT(expr) ((void)0)
#define _ASERTE(expr) ((void)0)
```

这也意味着 _ASSERT 和 _ASERTE 宏不能代替常规的错误检查和处理。例如，下面的代码是需要改进的：

```
char * lpsz=new char[BIG_ARRAY];
	ASSERT(lpsz!=NULL);
	memset(lpsz,0,BIG_ARRAY);
//...
```

在发布版本中如果内存分配失败，那么没有任何检查，会导致非法访问。正确的做法是应该在 _ASSERT 语句下加入常规的错误检查代码。

对断言机制的另一种误用就是在断言表达式中执行计算。比如 _ASSERT (*lpsz++!='0')，因为参数中的表达式在发布版本中根本不存在，所以对应的指针递增操作在发布版本中根本不会执行。

标准 C 也提供了一个名为 assert（全小写）的宏来实现断言，使用这个宏的好处是代码具有更好的移植性。

```
void assert( int expression );
```

VC 编译器对 assert 宏的实现与 _ASSERT 宏非常类似，不再赘述。

对于 MFC 程序，MFC 框架定义了 ASSERT 宏和 VERIFY 宏（见清单 21-11）。

清单 21-11 MFC 框架中用于断言的宏 (afx.h)

```
#ifdef _DEBUG
#define ASSERT(f) \
    do \
    { \
        if (!(f) && AfxAssertFailedLine(THIS_FILE, __LINE__)) \
            AfxDebugBreak(); \
    } while (0) \

#define VERIFY(f)           ASSERT(f)
#define ASSERT_VALID(pOb)   (::AfxAssertValidObject(pOb, THIS_FILE, __LINE__))
#else // !_DEBUG
#define ASSERT(f)           ((void)0)
#define VERIFY(f)           ((void)(f))
#define ASSERT_VALID(pOb)   ((void)0)
#endif // !_DEBUG
```

在 afxassert.cpp 文件中（位于 mfc\src\文件夹），我们可以看到 AfxAssertFailedLine 函数在没有定义 _AFX_NO_DEBUG_CRT 标志时会调用 _CrtDbgReport 函数。而 AfxDebugBreak 宏与 _CrtDbgBreak 是等价的。

```
#define AfxDebugBreak() _CrtDbgBreak()
```

因此，可以说 MFC 的 ASSERT 宏与 CRT 的 _ASSERT 宏几乎是相同的。

VERIFY 宏在调试版本中等同于 ASSERT 宏，但在发布版本中，其表达式仍会被编译进目标代码，所以 VERIFY 宏检查的条件表达式中可以包含函数调用或计算。因此像下面这样使用 VERIFY 宏是可以的，但是不可以使用 ASSERT 宏：

```
CWnd * pWndParent;
VERIFY(pWndParent=GetParent());
```

ASSERT_VALID 宏用于检查从 **CObject** 派生出的类实例指针，**AfxAssertValidObject** 函数（源代码文件为 **mfc\src\objcore.cpp**）除了检查指针是否为空外，还会检查对象的方法表（**VTable**）是否有效，并调用对象的 **AssertValid()** 方法。

MFC 还定义了一个 **ASSERT_KINDOF**，用来检查某个对象是否是某个类的实例。

```
#define ASSERT_KINDOF(class_name, object) \
ASSERT((object)->IsKindOf(RUNTIME_CLASS(class_name)))
```

被检查的类必须具有 **DECLARE_DYNAMIC** 和 **DECLARE_SERIAL** 声明。

21.4.3 _RPT 宏

在 VC8 中，可以通过 **_RPT** 系列宏调用 **_CrtDbgReport** 函数报告调试信息。**_RPT** 宏有以下两个系列。

一个系列是 **_RPT0**, **_RPT1**, **_RPT2**, **_RPT3**, **_RPT4**，简称 **_RPTn**。它们都具有如下原型：

```
_RPTn( reportType, format, ...[args] );
```

其中，**reportType** 可以是 **_CRT_WARN**、**_CRT_ERROR** 和 **_CRT_ASSERT** 三个常量之一；**n** 的值为 0~4，代表可变参数的个数。比如 **_RPT1** 宏的定义如下：

```
#define _RPT1(rptno, msg, arg1) \
    do { if ((1 == _CrtDbgReport(rptno, NULL, 0, NULL, msg, arg1))) \
        _CrtDbgBreak(); } while (0)
```

另一个系列是 **_RPTF0**, **_RPTF1**, **_RPTF2**, **_RPTF3**, **_RPTF4**，简称 **_RPTFn**。它们的原型和使用方法与 **_RPTn** 宏都相同，唯一的差别是 **_RPTFn** 的报告中包含源文件名和行号。比如 **_RPTF1** 宏的定义如下：

```
#define _RPTF1(rptno, msg, arg1) \
    do { if ((1 == _CrtDbgReport(rptno, __FILE__, __LINE__, NULL, msg, arg1))) \
        _CrtDbgBreak(); } while (0)
```

在发布版本中，以上宏都被定义为空，因此这些宏输出的信息也只有在调试版才有作用。

21.5 报告运行期检查错误

本节将介绍 CRT 报告运行期检查错误的方法，包括 **_CrtDbgReport** 函数、如何配置报告方式，以及如何编写回调函数参与报告过程。

21.5.1 _CrtDbgReport

_CrtDbgReport 是报告运行期检查信息的一个主要函数，根据我们在上一节的介绍，断言失败和 **_RPT** 宏都是通过调用 **_CrtDbgReport** 函数来报告调试信息的。

_CrtDbgReport 函数的函数原型为：

```
int _CrtDbgReport( int reportType, const char *filename,
    int linenumber, const char *moduleName, const char *format [, argument] ... );
```

其中，第一个参数是报告类型，可以为以下几个常量之一：_CRT_WARN（0）、_CRT_ERROR（1）和_CRT_ASSERT（2），filename 是要报告的文件名，linenumber 是要报告的行号，moduleName 是模块名，format 是一个格式串，与 printf 函数的格式串类似，format 后是数量不定的具体数据。如果 reportType 参数为_CRT_ASSERT，那么 format 参数会被看作断言表达式的内容，在其前面加上“Expression:”后显示出来，这也是_ASSERT 和_ASSERTE 间的唯一区别。

_CrtDbgReport 函数的返回值逻辑是：

```
if (MessageBox) // 如果信息输出到消息框
{
    Abort -> aborts // 如果用户选择终止，那么退出程序，不再返回
    Retry -> return TRUE // 如果用户选择重试，则返回 1
    Ignore-> return FALSE // 如果用户选择忽略，则返回 0
}
else
    return FALSE // 如果输出到文件或调试器，那么总返回 0
```

如果在处理过程中发生了异常情况，那么_CrtDbgReport 函数会返回-1。

在 CRT 的源文件目录中我们可以找到_CrtDbgReport 函数的源代码，对于 VC6，源文件是 VC98\crt\src\dbgrpt.c。对于 VC8，_CrtDbgReport 函数的实现分布在三个文件中，分别是 Vc\crt\src\dbgrpt.c、dbgrpt.c 和 dbgrptw.c，其中 dbgrptw.c 包含的是宽字符版本，即_CrtDbgReportW 函数，dbgrpt.c 包含了_VCrtDbgReportW 和_VCrtDbgReportA 两个内部函数，dbgrpt.c 包含了_CrtDbgReportT 函数和_CrtDbgReportTV 函数。这些函数的调用关系是，_CrtDbgReportT 将不定个数的参数放入到 arglist 数组（使用 va_start），然后调用_CrtDbgReportTV，_CrtDbgReportTV 会根据需要决定调用_VCrtDbgReportW 或_VCrtDbgReportA。清单 21-12 中的栈回溯显示了 CRT 的变量检查函数发现错误并调用错误报告函数的执行过程。

清单 21-12 报告 RTC 检查到的错误

001218c8 0041ad77	USER32!MessageBoxW+0x45	// MessageBox API
00121910 004079b9	rtcsample!__crtMessageBoxW+0x257	
00123b7c 00419304	rtcsample!__crtMessageWindowW+0x3b9	
0012bc1c 004075e8	rtcsample!_VCrtDbgReportW+0xb4	// 执行实际的报告逻辑
0012ec48 004072a9	rtcsample!_CrtDbgReportWV+0x328	// 数组方式的_CrtDbgReport
0012ec70 004030d2	rtcsample!_CrtDbgReportW+0x29	// UNICODE 版本
0012fadc 0040336c	rtcsample!failwithmessage+0x152	// 见下文
0012ff04 00401595	rtcsample!_RTC_StackFailure+0x10c	// 报告栈失败的入口函数
0012ff24 00401171	rtcsample!_RTC_CheckStackVars+0x45	// 检查栈变量
0012ff4c 00401082	rtcsample!TrashStackVariable+0x51	[rtcsamp.cpp @ 59]
0012ff54 00402d13	rtcsample!main+0x12	[rtcsamp.cpp @ 24] // 用户的入口函数
0012ffb8 00402add	rtcsample!__tmainCRTStartup+0x233	[crt0.c @ 318]
0012ffc0 7c816fd7	rtcsample!mainCRTStartup+0xd	// CRT 的入口函数
0012fff0 00000000	kernel32!BaseProcessStart+0x23	// 系统的进程启动函数

其中 failwithmessage 函数是 VC8 的 RTC 系列函数（即_RTC_CheckXXX），用来报告错误的枢纽，其函数原型为：

```
void __cdecl failwithmessage(void * retaddr, int crttype,
                           int errnum, const char * msg);
```

failwithmessage 内部会检查当前程序是否在被调试。它先调用 DebuggerProbe 函数来判断是否在 IDE 环境的集成调试器中执行。DebuggerProbe 函数的检测方法是调用 RaiseException API 抛出一个代码为 406D1388h 的特殊异常。当使用 VC 的集成调试器进行调试时，调试器会处理这个异常，使 DebuggerProbe 返回 1，否则 DebuggerProbe 会返回 0。

在 DebuggerProbe 返回 1 后，failwithmessage 会像下面这样调用 DebuggerRuntime 函数：

```
DebuggerRuntime(unsigned long dwErrorNumber=0x2, int bRealBug=0x1,
               void * pvReturnAddr=0x00401171, const wchar_t * pwMessage=0x0012ecb0)
```

其中 0x00401171 是 TrashStackVariable 函数中调用 RTC 检查函数（_RTC_CheckStackVars）的下一条指令地址：

```
0040116C call      _RTC_CheckStackVars (401540h)
00401171 pop       eax
```

参数 pwMessage 的内容是"Stack around the variable 'stackvar' was corrupted."，其中 stackvar 是 TrashStackVariable 函数中的变量。

DebuggerRuntime 函数内部会再次通过 406D1388h 号异常的方式与 VC 调试器通信，这次，调试器会显示图 21-5 所示的对话框。

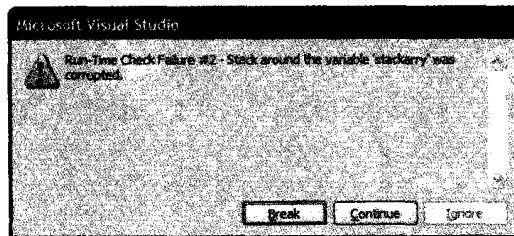


图 21-5 VC8 报告的 RTC 错误

如果当前的调试器不是 IDE 环境的集成调试器，那么 failwithmessage 会调用 IsDebuggerPresent API 判断是否是在其他调试器中运行，如果是，那么 failwithmessage 函数会调用 DbgBreakPoint 触发断点将当前程序直接中断到调试器中（清单 21-13）。

清单 21-13 在有调试器时先报告给调试器

```
0:000> k
ChildEBP RetAddr
0012ec8c 004031a4 ntdll!DbgBreakPoint
0012fadc 0040336c rtcsample!failwithmessage+0x224
0012ff04 00401595 rtcsample!_RTC_StackFailure+0x10c
0012ff24 00401171 rtcsample!_RTC_CheckStackVars+0x45
0012ff4c 00401082 rtcsample!TrashStackVariable+0x51 [rtcsamp.cpp @ 59]
0012ff54 00402d13 rtcsample!main+0x12 [rtcsamp.cpp @ 24]
0012ffb8 00402add rtcsample!__tmainCRTStartup+0x233 [crt0.c @ 318]
0012ffc0 7c816fd7 rtcsample!mainCRTStartup+0xd [crt0.c @ 187]
0012fff0 00000000 kernel32!BaseProcessStart+0x23
```

如果 IsDebuggerPresent 也返回 0,便如清单 21-12 所示那样使用_CrtDbgReport 函数来报告错误, 那个清单是在 CRT 对话框弹出后才附加调试器而产生的。

21.5.2 _CrtSetReportMode

_CrtDbgReport 函数可以把参数中指定的调试信息送往三个目的地: 调试文件, 调试器的监视窗口和图 21-4 所示的调试消息对话框。

可以调用_CrtSetReportMode 和_CrtSetReportFile 函数来配置 CRT 的报告方式, 默认的输出是调试消息对话框。我们先来看一下_CrtSetReportMode 函数, 其原型如下:

```
int _CrtSetReportMode(int reportType, int reportMode);
```

其中 reportType 是信息的三种类型: _CRT_WARN、_CRT_ERROR 和_CRT_ASSERT, reportMode 可以为表 21-5 所列出的三种模式中的零或多种。

表 21-5 运行期检查的信息输出模式

报告模式	值	信息输出目的地
_CRTDBG_MODE_DEBUG	2	调试器的输出 (output) 窗口
_CRTDBG_MODE_FILE	1	通过_CrtSetReportFile 函数设置的文件或其他输出终端 (例如 stdout 或 stderr)
_CRTDBG_MODE_WNDW	4	包含 Abort、Retry 和 Ignore 按钮的消息框

如果想获取某种报告类型的当前输出设置, 那么可以在调用_CrtSetReportMode 时把 reportMode 参数设为_CRTDBG_REPORT_MODE (-1)。

在 VC 运行库内部,CRT 使用一个名为_CrtDbgMode 的全局数组来记录报告模式。

```
int _CrtDbgMode[_CRT_ERRCNT] = {
#ifndef _WIN32
    _CRTDBG_MODE_DEBUG, _CRTDBG_MODE_WNDW,
    _CRTDBG_MODE_WNDW
#else /* _WIN32 */
    _CRTDBG_MODE_DEBUG, _CRTDBG_MODE_DEBUG,
    _CRTDBG_MODE_DEBUG
#endif /* _WIN32 */
};
```

也就是对于 WIN32 程序, 警告信息的默认输出是调试窗口, 其他两类信息输出到消息对话框, 对于控制台程序, 所有信息都输出到调试器。

如果要输出到消息对话框, 那么_CrtDbgReport 函数会调用另一个名为 CrtMessageWindow 的内部函数来弹出消息对话框:

```
static int CrtMessageWindow( int nRptType, const char * szFile, const char * szLine,
                            const char * szModule, const char * szUserMessage)
```

CrtMessageWindow 函数弹出的消息对话框包含三个按钮: 终止 (Abort)、重试 (Retry) 和忽略 (Ignore)。如果用户选择终止, 那么 CrtMessageWindow 便调用 ExitToShell() 函数(或_exit(3))退出进程了。如果选择重试, 那么 CrtMessageWindow 函数会返回 1 给_CrtDbgReport,_CrtDbgReport 函数再将这个值返回到它的调用者。

如果调用者是 ASSERT 宏，那么从清单 21-8 可以看出，接下来会执行 _CrtDbgBreak，_CrtDbgBreak 也是个宏定义，对于 x86 平台，_CrtDbgBreak 被定义为 int 3 指令。

```
#define _CrtDbgBreak() __asm { int 3 }
```

因此，如果选择重试，接下来便会执行 INT 3 指令。如果该程序正在被调试，便会中断到调试器，否则会导致一个断点异常，如果应用程序没有捕捉这个异常，那么系统的默认异常处理会启动应用程序错误对话框，这时，如果系统中安装了 JIT 调试器，那么可以开始 JIT 调试，如果不进行 JIT 调试，那么应用程序错误对话框被关闭后，应用程序也就被强行停止了，如果进行调试，那么应用程序会停在 int 3 指令的位置。

如果用户选择忽略按钮，则 _CrtDbgReport 函数返回 0，程序会继续执行。但如果导致断言失败的错误条件仍然存在，则程序继续运行很可能导致其他错误。

21.5.3 _CrtSetReportFile

可以通过 _CrtSetReportFile 函数为每一种报告类型指定一个文件句柄，当该报告类型所对应的模式（_CrtDbgMode）中包含 _CRTDBG_MODE_FILE 标志时，_CrtDbgReport 函数便会将调试信息写到该文件中。_CrtSetReportFile 函数的原型为：

```
_HFILE _CrtSetReportFile( int reportType, _HFILE reportFile );
```

其中 reportFile 参数可以是一个文件句柄（比如使用 CreateFile API 创建的），也可以用 _CRTDBG_FILE_STDOUT ((_HFILE)-4) 或 _CRTDBG_FILE_STDERR ((_HFILE)-5) 来指定标准输出（stdout）和标准错误输出（stderr）。在参数 reportFile 中指定 _CRTDBG_REPORT_FILE 可以读取当前的设置。

在 VC 运行库内部，CRT 使用一个名为 _CrtDbgFile 的全局数组来记录各个文件句柄。

```
_HFILE _CrtDbgFile[_CRT_ERRCNT] = { _CRTDBG_INVALID_HFILE,
    _CRTDBG_INVALID_HFILE, _CRTDBG_INVALID_HFILE };
```

在调试时可以通过观察这个全局数组来直接读取这个设置。

21.5.4 _CrtSetReportHook

除了可以通过以上两个 CRT 函数来控制运行期检查的信息输出外，还可以通过调用 _CrtSetReportHook 函数来设置一个回调函数。这样，每当 _CrtDbgReport 函数被调用时，_CrtDbgReport 都会先调用这个回调函数。回调函数应该具有如下原型：

```
int OurReportHook( int reportType, char *message, int *returnValue );
```

定义好了这样一个函数后，便可以调用 _CrtSetReportHook 函数将其注册给 CRT。

```
_CRT_REPORT_HOOK _CrtSetReportHook(_CRT_REPORT_HOOK reportHook);
```

其中 reportHook 便是指向回调函数的函数指针。在内部，CRT 使用一个名为 _pfnReportHook 的全局变量来记录用户注册的回调函数。因此 _CrtSetReportHook

的操作其实很简单，只是将参数指定的函数指针赋给 `_pfnReportHook` 变量，并将原来的值返回。

`_CrtSetReportHook` 函数没有直接提供卸载回调函数的方法；因此一种可能的间接做法是将 `_CrtSetReportHook` 函数的返回值保存起来，当须要卸载自己的回调函数时将这个保存的值设置回去。但是这样做在多个 DLL 被动态加载和卸载时可能导致问题，因为如果多个 DLL 卸载顺序不严格是 LIFO（后进先出）的，那么 `_pfnReportHook` 指针就可能指向一个已经卸载的模块中的函数，这样当 `_CrtDbgReport` 再次被调用时就会导致应用程序崩溃。为了避免这个问题，对于 DLL 模块，应该使用下面介绍的 `_CrtSetReportHook2` 函数来注册回调函数。

21.5.5 `_CrtSetReportHook2`

为了克服 `_CrtSetReportHook` 存在的不足，VC7 引入了 `_CrtSetReportHook2` 函数和新的以链表机制来管理 CRT 报告的回调函数。`_CrtSetReportHook2` 函数的原型如下：

```
int _CrtSetReportHook2( int mode, _CRT_REPORT_HOOK pfnNewHook);
```

`_CrtSetReportHook2` 使用一个名为 `_pReportHookList` 的链表来记录注册的回调函数。链表的每个节点都是一个指向 `ReportHookNode` 结构的指针。该结构定义在 `vc\crt\src\dbgint.h` 文件中。

```
typedef struct ReportHookNode {
    struct ReportHookNode *prev;      // 指向前一个节点
    struct ReportHookNode *next;      // 指向后一个节点
    unsigned refcount;                // 引用计数
    _CRT_REPORT_HOOK pfnHookFunc;     // 本节点的回调函数
} ReportHookNode;
```

新版的 `_CrtDbgReport` 函数会先扫描 `_pReportHookList` 链表并依次调用其中包含的每个回调函数，然后调用通过 `_CrtSetReportHook` 设置的回调函数，感兴趣的读者可以阅读 `vc\crt\src\dbgrptt.cpp` 中的 `_VCrtDbgReportW` 函数的源代码，观察其详细过程。

21.5.6 使用其他函数报告 RTC 错误

对于 VC8，如果不使用 CRT 的 `_CrtDbgReport` 函数报告 RTC 错误，那么可以自己定义一个与 `_CrtDbgReport` 函数具有相同函数原型的函数，然后调用 `_RTC_SetErrorFuncW` 将其设置为 RTC 报告函数。

```
_RTC_error_fnW _RTC_SetErrorFuncW( _RTC_error_fnW function );
```

此设置会影响到所有 `_RTC_CheckXXX` 函数的输出，但不会改变 `ASSERT` 类宏的输出。从清单 21-12 中可以看到，`_RTC_CheckXXX` 函数如果检测到错误，那么会调用 `failwithmessage` 函数，`failwithmessage` 会调用 `_RTC_GetErrorFuncW` 函数来取当前的 RTC 错误报告函数指针。

```
_RTC_GetErrorFuncW:
00401D2E    mov        eax,dword ptr [_RTC_ErrorReportFuncW (4071A8h)]
00401D33    ret
```

从以上 _RTC_GetErrorFuncW 函数的反汇编指令可以看出，_RTC_GetErrorFuncW 函数只是简单地读取全局变量 _RTC_ErrorReportFuncW。

_RTC_ErrorReportFuncW 是在 _CRT_RTC_INITW 函数中被初始化的，_CRT_RTC_INITW 函数的默认实现是将 _CrtDbgReportW 函数赋给 _RTC_ErrorReportFuncW 变量。

```
0:000> u _CRT_RTC_INITW
rtcsample!_CRT_RTC_INITW:
004036a0 55          push    ebp
004036a1 8bec        mov     ebp,esp
004036a3 b880724000  mov     eax,offset rtcsample!_CrtDbgReportW (00407280)
004036a8 5d          pop    ebp
004036a9 c3          ret
```

在不使用 C 运行库的情况下使用 RTC 机制

如果不使用 C 运行库，即在链接选项中指定忽略/NODEFAULTLIB 开关，或者指定忽略 C 运行库的 lib 文件 (msvcrt.lib msvcrt.lib libc.lib libcd.lib libmt.lib libmtd.lib)，那么是否还可以使用 RTC 机制呢？答案是肯定的。MSDN 的 rtcsample 例子便演示了这种用法。简单说来，要完成以下几个步骤。

首先，向上面介绍的那样定义一个自己的 RTC 错误报告函数。然后，实现一个 _CRT_RTC_INITW 函数，在这个函数中返回自己定义的 RTC 错误报告函数，即类似如下：

```
extern "C" _RTC_error_fnW _CRT_RTC_INITW(void *, void **, int , int , int )
{
    return &Catch_RTC_Failure;
}
```

第三步是在程序初始化和退出前分别调用 _RTC_Initialize() 和 _RTC_Terminate() 函数。最后需要在链接选项中加入 RunTmChk.lib，以便链接器链接 RTC 函数。更多细节可以参阅 rtcsample 示例，对于使用特殊 C 运行库的项目，如果还希望使用 RTC 机制，那么本技术是很有用的。另外，使用 CRT 和没有 CRT 产生的可执行文件大小有着非常明显的差异。

21.6 本章总结

本章前半部分（前 3 节）详细介绍了编译器的运行库，包括 C 运行库和 C++ 的运行库。后半部分介绍了编译器所提供的运行期检查功能，包括自动插入的检查和程序员利用断言等机制手工插入的检查，以及报告检查错误的方法。接下来的两章将分别详细介绍栈和堆有关的运行期检查。

参考文献

1. Jac Goudsmit. Running Code Before and After Main (or WinMain or DllMain) .
<http://www.codeguru.com>
2. Using Run-Time Checks Without the C Run-Time Library. Microsoft Corporation

栈和函数调用

在当今无法计数的众多软件当中，不同软件所使用的技术大多也千差万别，但可以肯定地说，很难找到哪个软件没有使用栈（stack）和函数调用（function call）这两项技术。可以说，这两项技术是支撑软件大厦的两块重要基石。也许因为这两项技术太基本了，所以大多数文档和书籍在论及相关的内容时都默认读者已经熟悉它们了。但事实上，很多人对这两项技术还存在着很多疑问。这两项技术与软件调试也有着密不可分的联系，是很多调试技术的基础。

本章将先介绍栈的基本概念（22.1 节）和栈的创建过程（22.2 节），然后分别介绍栈在函数调用（22.3 节）和局部变量分配（22.4 节）方面所起的作用，以及栈帧的概念、帧指针省略（22.5 节）和栈指针检查（22.6 节）。第 22.7 节将详细介绍调用协定（Calling Convention）。第 22.8 节介绍栈空间的分配和自动增长机制，第 22.9 节至第 22.12 节将介绍栈有关的安全问题，以及编译器所提供的检查和保护机制。

22.1 简介

什么是栈（stack）？可以从以下几个角度来回答这个问题。

从数据结构角度看，栈是一种用来存储数据的容器（container）。放入数据的操作被称为压入（push），从栈中取出数据的操作被称为弹出（pop）。存取数据的一条基本规则是后进先出（last-in-first-out，简称 LIFO），即最后放入的数据先被取出。

对基于栈的计算机系统而言，栈是存储局部变量和进行函数调用所必不可少的连续内存区域。编译器、操作系统和 CPU 按照规范各尽其责，保证正确合理地使用栈。编译器在编译时会将函数调用和局部变量存取编译为合适的栈操作（第 20.4 节和第 20.5 节详细讨论）。操作系统在创建线程时，会为每个线程创建栈，包括分配栈所需的内存空间和初始化有关的数据结构及寄存器。以 x86 系统为例（如不特别说明，本章的讨论均基于 x86 系统），SS（Stack Segment）寄存器用来描述栈所在的内存段，ESP（Extended Stack Pointer）寄存器用来记录栈的栈顶地址。CPU 在执行程序时，会假定 SS 和 ESP 寄存器已经指向一个设置好的栈，执行 PUSH 指令时便向 ESP 所指向的内

存地址写入数据，然后调整 ESP 的值，使其指向新的栈顶（也就是新压入的数据）；执行 POP 指令时便从栈顶弹出数据，并会向栈底方向调整 ESP 寄存器的值，保证它始终指向栈的顶部。当 CPU 执行 CALL 和 RET 这样的函数调用指令时，它也会使用栈（第 20.3 节详细讨论）。

从线程角度看，栈是每个 Windows 线程的必备设施。在 Windows 系统中，每个线程至少有一个栈，系统线程之外的每个线程都有两个栈，一个供该线程在用户态下执行时使用，称为用户态栈；另一个供该线程在内核态下执行时使用，称为内核态栈。在一个运行着多任务的系统中，因为有很多个线程，所以会有多个栈存在。尽管有如此多的栈，但是对于 CPU 而言，它只使用当前栈，即 SS 和 ESP 寄存器所指向的栈。当进行不同任务间的切换，以及同一任务内的内核态与用户态之间的切换时，系统会保证 SS 和 ESP 寄存器始终指向合适的栈。

另外值得说明的是，在 x86 系统中，栈是朝低地址方向生长的。也就是说，压栈操作会导致 ESP 寄存器（栈指针）的值减小，弹出操作会导致 ESP 值变大。这也意味着，后压入数据的内存地址比先压入数据的地址更小。

22.1.1 用户态栈和内核态栈

一个线程可能在不同的特权级（privilege）下执行，比如用户态的代码调用系统服务时，该线程便会被切换到系统模式下执行，待所调用的系统服务执行完毕后再切换回用户态执行（这个切换过程通常被称为 Context Switch）。如果让用户态的代码和系统服务使用同一个栈，那么必然会产生安全问题（参见第 22.10 节关于栈溢出的内容）。为了保证不同优先级的代码和数据的安全，线程在不同优先级下运行时会使用不同的栈。以 x86 系统为例，系统中共有四种特权级，CPU 在执行跨越特权级的代码转移时，会自动切换到不同的栈。那么 CPU 是如何找到每个特权级应该使用的栈的呢？答案是每个任务的任务状态段（TSS）记录了不同优先级所使用的栈的基本信息。在 TSS 中，偏移 4 到偏移 28 的 24 个字节是用来记录栈的段信息（SS）和栈指针（ESP）值的（参见第 11.1.3 节和图 11-2）。

Windows 系统只使用了 x86 CPU 定义的四种特权级中的两种，所以每个普通的 Win32 线程都有两个栈。一个供线程在内核态下执行时使用，称为内核态栈（kernel-mode stack），另一个供线程在用户态下执行时使用，称为用户态栈（user-mode stack）。这里之所以说普通的 Win32 线程，是因为那些只运行在内核模式下的系统线程没有用户态栈，因为它们不需要在用户模式下运行。

在 Windows 系统为每个线程所维护的基本数据结构中，记录了内核态栈和用户态栈的基本信息。内核态栈记录在_KTHREAD 结构中，用户态栈记录在_TEB 结构中。

每个 Windows 线程都拥有一个名为_KTHREAD 的数据结构，该数据结构位于内核空间中，是 Windows 系统管理和记录线程信息和进行线程调度的重要依据。在

_KTHREAD 结构中，有几个成员是专门用于记录栈信息的。

- **StackBase:** 内核态栈的基地址，即栈的起始地址。
- **StackLimit:** 内核态栈的边界，因为栈是向下生长的，所以其值等于 StackBase 减去内核态栈的大小。
- **LargeStack:** 是否已经切换成大内核态栈（参见下文）。
- **KernelStack:** 内核态栈的栈顶地址，用于保存栈顶地址（ESP）。
- **KernelStackResident:** 内核态栈是否位于物理内存中。
- **InitialStack:** 供内核态代码逆向调用用户态代码（参见第 8.3.4 节）时记录本来的栈顶位置。
- **CallbackStack:** 也是在逆向调用时使用，Vista 之前用来记录栈指针值，KiCallUserMode 在将执行权交给用户态代码之前将当时的内核态栈指针（ESP）保存在这个字段中，待用户态函数执行完毕用户态的代码通过触发 INT 2B 返回时，对应的处理例程 KiCallbackReturn 再将这里保存的值恢复到 ESP 寄存器中。

可见，通过以上字段足以了解内核态栈的各种基本信息，基地址、边界、栈顶等。可以通过 WinDBG 的.thread 命令得到一个线程的_ETHREAD 地址：

```
kd> .thread
Implicit thread is now 81af8bf0
```

因为_ETHREAD 结构的第一个成员 Tcb 就是_KTHREAD 类型的，所以上地址也就是_KTHREAD 结构的地址，因此可以使用 dt 命令来观察上面介绍的各个字段：

```
kd> dt nt!_KTHREAD 81af8bf0
+0x018 InitialStack      : 0xf4d2bb90
+0x01c StackLimit        : 0xf4d26000
+0x028 KernelStack       : 0xf4d2bcb4
+0x12a KernelStackResident : 0x1 ''
+0x12c CallbackStack     : 0xf4d2bb98
+0x142 LargeStack        : 0x1 ''
+0x168 StackBase          : 0xf4d2c000
```

其中 StackBase 的值是 0xf4d2c000，可以看出这个地址是一个内核空间的地址，StackLimit 的值是 0xf4d26000，StackLimit- StackBase 等于 0x6000，即 24576，说明这个线程的内核态栈的大小是 24KB。此时 LargeStack 字段已经为 1，意味着这已经是大内核态栈，普通的内核态栈通常为 12KB。

因为速度和支持多种处理器等，Windows 没有采用 x86 CPU 的硬件级任务切换机制（以 TSS 为核心），但是为了让 CPU 在模式切换时可以找到合适的栈信息（参见下一节对跨优先级调用的讨论），Windows 会为系统内的普通线程建立一个共享的 TSS 段，当进行软件方式的任务切换时，Windows 会把当前任务的内核态栈信息复制到 TSS 段中。

用户态栈的基本信息记录在线程信息块 (_NT_TIB) 结构中，NT_TIB 是线程环境块 (TEB) 结构的第一个部分，因此可以根据 TEB 的地址来显示_NT_TIB 结构（见清单 22-1）。

清单 22-1 观察线程的_NT_TIB 结构

```

0:001> ~
  0  Id: 10d4.140  Suspend   : 1 Teb: 7ffdf000 Unfrozen
  . 1  Id: 10d4.175c Suspend   : 1 Teb: 7ffdde000 Unfrozen
0:001> dt _NT_TIB 7ffdf000
ntdll!_NT_TIB
+0x000 ExceptionList      : 0x0007ff10 _EXCEPTION_REGISTRATION_RECORD
+0x004 StackBase          : 0x00080000
+0x008 StackLimit         : 0x0007d000
+0x00c SubSystemTib       : (null)
+0x010 FiberData          : 0x00001e00
+0x010 Version            : 0x1e00
+0x014 ArbitraryUserPointer : (null)
+0x018 Self               : 0x7ffdf000 _NT_TIB

```

其中 StackBase 即用户态栈的基地址，StackLimit 即栈的边界。用户态栈具有按照需要自动增长的特性，我们将在第 22.6 节详细讨论。

从功能和使用方式来看，用户态栈与内核态栈是相同的。但因为它们一个是为用户态代码服务的，另一个是为内核态代码服务的，所以它们的特征还是有差异的。首先，用户态栈是被分配在其所在进程的用户空间中的，而内核态栈是被分配在系统空间中的。我们知道用户空间是共享的，而系统空间是全局性的。换句话说，一个系统内的用户空间有很多个，而系统空间只有一个。以典型的 32 位 Windows 系统为例，系统内每个进程都拥有自己的 2GB 用户空间，不过这看似独有的空间使用的都是 0~0x7FFFFFFF 这一段相同的地址空间；而系统空间是全局性的，整个系统只有 0x80000000~0xFFFFFFFF 这 2GB 的空间，被所有进程所共享。因为线程的内核态栈都是分配在系统空间中的，所以系统内每个线程的内核态栈占用的都是宝贵的系统空间。笔者在写作此内容时，所使用的 Windows 系统中共有 660 个线程（略有浮动，可以使用 PerfMon 工具观察 System 的 Threads 计数器）。这意味着即使每个线程的内核态栈只占用较少的空间，加起来也就很大了。这很自然地产生了内核态栈和用户态栈的一个重要区别，用户态栈是可指定大小的，其默认大小为 1MB；而内核态栈是完全由系统来控制大小的，其大小因处理器结构不同而不同，但通常在十几 KB 到几十 KB 之间。以下是基于不同处理器的 Windows 系统的默认内核态栈大小。

- 在基于 x86 CPU 的系统中，内核态栈的初始大小是 12KB。
- 在基于 x64 CPU（英特尔 64 和 AMD64）的系统中，内核态栈的初始大小是 24KB。
- 在基于安腾（Itanium）处理器的系统中，内核态栈的初始大小是 32KB。

考虑 GUI 线程在调用 GDI 等内核服务时通常需要更大的内核态栈，所以在一个线程被转变为 GUI 线程后（创建时，所有线程都是非 GUI 线程），Windows 会为其创建一个较大的可增长的内核态栈来替换掉原来的栈，称为大内核态栈，下一节将详细讨论。

22.1.2 函数、过程和方法

在软件工程中，函数（function）、过程（procedure 或 subroutine）和方法（method）这三个术语经常被替换使用，因为它们都可以用来指代一段可被调用的程序代码。它

们有什么区别吗？简单来说，函数可以返回一个结果值（result value），而严格意义上的过程不能，这一点在 VB 语言中体现得最为明显。C 语言中，只有函数，它可以返回值也可以不返回值（返回类型为 void）。在面向对象的语言（如 C++）中，通常使用方法这一术语来指代类中的各个函数。

尽管以上三个术语有着细微的差异，且不可轻易说它们是一回事，但是因为本章主要以 C/C++ 和汇编语言为例来讨论调试技术，所以除非特别指出，我们统一使用函数这一术语来泛指以上三个概念。

22.2 栈的创建过程

对于每个 Windows 线程来说，栈是线程内的代码进行函数调用和变量分配的必备设施，因此拥有可使用的栈是线程能够运行的前提条件。操作系统在创建线程时，就为其创建了栈。或者说，创建栈是创建线程的一个必不可少的步骤。我们先来看每个线程的内核态栈是如何创建的。

22.2.1 内核态栈的创建

PspCreateThread 是 Windows 内核中用于创建线程的一个重要内部函数。无论是创建系统线程（PsCreateSystemThread）还是用户线程（NtCreateThread 服务），都离不开这个函数。除了创建重要的 ETHREAD 结构，PspCreateThread 函数的另一个重要任务就是创建内核态栈。

前面我们说过，对于 GUI 线程，Windows 会为其创建大内核态栈。不过，当创建线程时，所有线程都不是 GUI 线程。Windows 是在线程第一次调用 Windows 子系统的内核服务（Win32K）时将其转变为 GUI 线程的。因此当创建线程时，PspCreateThread 函数总是调用 MmCreateKernelStack 函数创建一个默认大小的内核态栈，这个栈的大小是固定的，而且是不可增长的。

当一个线程被转化为 GUI 线程时，系统的 PsConvertToGuiThread 函数会为该线程重新创建一个栈，然后使用 KeSwitchKernelStack 切换到新的栈。新的栈是可以改变大小的，被称为大内核态栈（Large Kernel Stack）。大内核态栈的最大值记录在名为 MmLargeStackSize 的全局变量中。

```
kd> dd MmLargeStackSize 11
80542720 0000f000
```

另两个全局变量 MmLargeStacks 和 MmSmallStacks 分别用来记录系统中大型栈和小型栈的总个数：

```
kd> dd nt!MmLargeStacks 11
8054be18 0000004a
kd> dd nt!MmSmallStacks 11
8054be14 000000d1
```

在一个线程被转变为 GUI 线程后，其 KTHREAD 结构的 LargeStack 字段会被改为 1，同时其 Win32Thread 字段也会由 0 变为非 0（这可以作为判断是否是 GUI 线程的一个依据）。转换后，新的栈通常不会立即增大，而是当需要时，调用 MmGrowKernelStack 函数来增长栈（参见下文），每次增长的幅度至少为一个页面的大小（x86 中为 4KB）。内核态的代码（如驱动程序）可以调用 IoGetStackLimits 函数和 IoGetRemainingStackSize 函数分别得到当前栈的边界和剩余大小。

22.2.2 用户态栈的创建

Windows 线程的用户态栈是由 KERNEL32.DLL 中的 BaseCreateStack 函数创建的。对于每个进程的初始线程，用户态栈是由 CreateProcess 函数调用 NtCreateProcess 创建好新进程的，包括地址空间、进程句柄等在内的各种基本结构后（调用 NtCreateThread 创建初始线程前）调用 BaseCreateStack 函数创建的。

初始线程之外的其他线程通常是调用 CreateThread 或 CreateRemoteThread 函数创建的。事实上，CreateThread 函数只是简单地调用 CreateRemoteThread 函数。CreateRemoteThread 函数在调用内核服务 NtCreateThread 前，会调用 BaseCreateStack 函数来创建用户态栈。

因此，无论是系统创建的初始线程还是用户代码创建的其他线程，都是在使用 BaseCreateStack 创建好用户态栈后再调用 NtCreateThread 来创建线程的。

BaseCreateStack 是位于 KERNEL32.DLL 中的一个未公开函数，其原型大致如下：

```
NTSTATUS BaseCreateStack(IN HANDLE hProcess,
    IN DWORD dwCommitStackSize, IN DWORD dwReservedStackSize,
    OUT PINITIAL_TEB pInitialTeb)
```

简单来说，BaseCreateStack 函数就是在 hProcess 所指定的进程空间中根据 dwReservedStackSize 参数保留一段内存区域，并在这个区域中按照 dwCommitStackSize 参数所指定的大小提交出一部分作为栈的初始空间。BaseCreateStack 函数将所保留和提交内存区域的参数保存到 pInitialTeb 指向的结构中。而后这些参数会传递给 NtCreateThread 内核服务，最终被保存到线程的环境块（TEB）结构中。当用户态调试时，可以使用 !teb 命令来观察 TEB 中包含的栈信息：

```
0:002> !teb
TEB at 7ffdc000
  ExceptionList:          00f7ffe4
  StackBase:              00f80000
  StackLimit:             00f7f000 ...
```

也可以使用 dt 命令来观察：

```
0:002> dt _Teb 7ffdc000 -b
ntdll!_TEB
+0x000 NtTib           : _NT_TIB
+0x000 ExceptionList   : 0x00f7ffe4
+0x004 StackBase        : 0x00f80000
+0x008 StackLimit       : 0x00f7f000 ...
```

下面我们来看一下参数 dwReservedStackSize 和 dwCommitStackSize，它们分别用来指定要创建栈的保留内存区大小和已经提交的内存区大小。也就是为栈保留的最大内存地址空间，以及初始提交的内存空间大小，后者属于前者的一部分。保留空间实际上只是保留一个地址范围，在使用前还必须进行提交，提交时系统才真正进行内存分配。

对于使用 CreateThread 或 CreateRemoteThread 创建的线程，可以通过参数 dwStackSize 指定栈的保留大小（在 dwCreationFlags 中设置 STACK_SIZE_PARAM_IS_A_RESERVATION 标志）和初始提交大小。那么初始线程的栈大小是如何指定的呢？答案是可执行文件（PE 文件）的文件头信息。在 PE 文件的 IMAGE_OPTIONAL_HEADER 中，SizeOfStackReserve 和 SizeOfStackCommit（都为 DWORD 类型）字段就是分别用来指定栈的默认保留大小和提交大小的。当我们调用 CreateThread 时用 0 作 dwStackSize 参数时，系统也会使用 IMAGE_OPTIONAL_HEADER 中指定的大小。SizeOfStackReserve 和 SizeOfStackCommit 字段的值是链接程序（Linker）在生成 EXE 文件时根据链接选项写入的。对于 VC 编译器，可以通过如下开关进行设定：

```
/STACK:reserve[,commit]
```

其中 reserve 和 commit 分别是保留空间和提交空间的字节数（如果不是 4 的倍数，会自动取整为 4 的倍数）。也可以在 DEF 文件中使用 STACKSIZE 语句来设定：

```
STACKSIZE reserve[,commit]
```

对于链接好的 EXE 文件，还可以通过 EDITBIN 工具来修改这两个值。

进一步来说，保留和提交内存都是通过系统的虚拟内存分配函数来完成的，SDK 中公开了 VirtualAlloc 和 VirtualAllocEx API，事实上它们都是调用内核服务 NtAllocateVirtualMemory：

```
NTSTATUS NtAllocateVirtualMemory( IN HANDLE hProcessHandle,
    IN OUT PVOID lpBaseAddress, IN ULONG ZeroBits, IN OUT PULONG plRegionSize,
    IN ULONG flAllocationType, IN ULONG flProtect );
```

其中，lpBaseAddress 是地址指针；plRegionSize 是区域大小；flAllocationType 为要分配的内存类型：MEM_RESERVE 是保留内存，MEM_COMMIT 为提交内存；flProtect 用来指定所分配内存的保护属性，如 PAGE_READONLY（只读）和 PAGE_READWRITE（读写）等。

可以把 BaseCreateStack 创建用户态栈的过程归纳为如下几个重要步骤。

1. 将提交空间大小取为内存页大小的倍数，将总保留大小取整为内存分配的最小粒度（4 字节）。
2. 调用内存分配函数（NtAllocateVirtualMemory）保留内存地址空间，内存分配类型为 MEM_RESERVE，分配的大小为栈空间保留大小。
3. 调用 NtAllocateVirtualMemory 在保留空间的高地址端提交初始栈空间，内存分配类型为 MEM_COMMIT，分配的大小为初始提交大小（参见下一条）。

4. 如果保留空间大于初始提交空间，则第 3 步会多提交一个页面用作栈保护页面。保护的方法是调用虚拟内存保护函数（VirtualProtect）对这个页面（已提交栈低地址端的一个页面）设置 PAGE_GUARD 属性。

其中第 4 步创建的栈保护页面是实现栈自动增长功能所必需的，我们将在第 22.8 节详细介绍栈增长机制。

22.2.3 跟踪用户态栈的创建过程

下面通过一个实验来加深大家对用户态栈创建过程的理解。为了跟踪 BaseCreateStack 函数的工作过程，我们特意编写了一个名为 AllocStk 的小程序，在这个程序中调用 CreateProcess API 创建另一个进程（EvtLog.Exe）。

启动 WinDBG，选择 File>Open Executable...，浏览到 code\bin\debug 目录，选择 AllocStk.exe，并在命令行参数中指定 EvtLog.exe。然后用 bp 命令对 kernel32!BaseCreateStack 函数设置一个断点，输入 g 命令让程序执行，断点会随即命中，键入 kv 命令观察函数调用过程（清单 22-2）。

清单 22-2 在创建新进程的初始线程前为其创建用户态栈

```
0:000> knL
# ChildEBP RetAddr
00 0012f1fc 7c819d88 kernel32!BaseCreateStack      // 为新进程创建栈
01 0012fc38 7c81d5af kernel32!CreateProcessInternalW+0x19d5 // UNICODE 版本
02 0012fd24 7c802393 kernel32!CreateProcessInternalA+0x29c // 内部函数
03 0012fd5c 004010e7 kernel32!CreateProcessA+0x2c    // 创建新进程 (EvtLog.exe)
04 0012ff30 00401463 AllcStk!WinMain+0x67        // 用户的入口函数
05 0012ffc0 7c816ff7 AllcStk!WinMainCRTStartup+0x113 // CRT 的入口函数
06 0012ffff0 00000000 kernel32!BaseProcessStart+0x23 // 系统的进程启动函数
```

此时任务管理器中已经可以看到新进程，但是线程数是 0，因为初始线程尚未创建。观察栈上的参数：

```
0:000> dd 0012f1fc+8 14
0012f204 000007bc 00001000 00100000 0012f484
```

其中 7bc 是新进程的句柄，1000 即 4K 为栈初始提交大小，0x100000 为栈保留大小，即 1MB。

接下来对 ntdll!ZwAllocateVirtualMemory 设置一个断点，因为 BaseCreateStack 会多次调用它来创建栈。当 ntdll!ZwAllocateVirtualMemory 处的断点命中时，观察其参数。因为其参数较多，所以我们直接使用 dd esp 命令来观察栈中的参数：

```
0:000> dd esp 18
0012f1c8 7c810327 000007bc 0012f1f8 00000000
0012f1d8 0012f20c 00002000 00000004 00000000
```

其中 7c810327 是返回地址，000007c0 是进程句柄（参数 1），0012f1f8 是参数 lpBaseAddress，用来存放分配到的地址，其目前值为 0。0012f20c 是参数 plRegionSize，其内容是要申请的内存区大小：

```
0:000> dd 0012f20c 11
0012f20c 00100000
```

接下来的 00002000 和 00000004 分别代表分配类型 MEM_RESERVE 和内存页保护属性 PAGE_READWRITE, WinNT.h 中定义了这些常量。

#define MEM_RESERVE	0x2000	//保留
#define PAGE_READWRITE	0x04	//读写
#define MEM_COMMIT	0x1000	//提交
#define PAGE_GUARD	0x100	//保护

键入 gu 命令让程序继续, 完成这个系统调用, 再观察 0012f1f8 处的值:

```
0:000> dd 0012f1f8 11
0012f1f8 00030000
```

这说明系统分配给栈的 1MB 空间是从 0x30000 开始的, 截止地址应该为 0x130000。

按 F5 继续执行, 应该再次命中 ntdll!ZwAllocateVirtualMemory 处的断点。再次显示其参数:

```
0:000> dd esp
0012f1c8 7c810552 000007bc 0012f1f8 00000000
0012f1d8 0012f208 00001000 00000004 00000000
```

参数 5 等于 00001000 表示这是在提交初始栈空间, 观察参数 2 指定的地址值:

```
0:000> dd 0012f1f8 11
0012f1f8 0012e000
```

这说明 BaseCreateStack 在提交从 0x0012e000 开始到 0x130000 的 0x2000=8KB 空间, 第一步中的保留参数是 0x1000, 即 4KB, 这里提交 8KB 是为栈保护页面多提交 4KB (一个页面)。

接下来, BaseCreateStack 会调用 VirtualProtectEx API 对栈保护页面加 PAGE_GUARD 属性, 这个 API 会调用 NtProtectVirtualMemory 系统服务, 因此我们对 ntdll!ZwProtectVirtualMemory 设下断点, 然后按 F5 继续执行。随即断点待命中后, 观察栈:

```
0:000> dd esp 18
0012f1cc 7c8103ab 000007bc 0012f1f8 0012f1f4
0012f1dc 00000104 0012f1f0 00000000 00000003
```

其中 000007bc 是第一个参数 hProcess, 0012f1f8 是第二个参数 lpAddress, 它的内容是要保护页面的地址, 使用 dd 命令观察可以看到其值为 0012e000。0012f1f4 中包含的是要保护的内存区大小, 其值为 0x1000, 即 4KB (1 个内存页)。参数 4 (00000104) 为新的内存页属性, 0x100 即 PAGE_GUARD, 最后一个参数 0012f1f0 是个指针, 用来接收旧的页保护属性。

图 22-1 (左) 画出了刚创建好的栈的示意图, 此时栈的基地址为 0x130000, 栈的边界为 0x12f000, 栈中可使用空间为 4KB, 栈的总大小为 1MB。

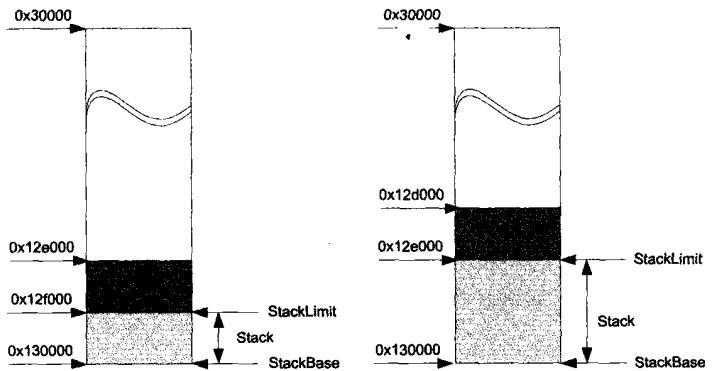


图 22-1 刚创建好的用户态栈（左）和自动增长一个页面后的栈（右）

地址 0x12f000 和 0x12e000 之间的一个页是保护页，当已经提交的栈空间用完触及到保护页时，系统的栈增长机制会提交更多空间并移动保护页，右侧的图表示栈自动增长一个内存页后的情形。

22.3 CALL 和 RET 指令

在基于 x86 处理器的系统中，CALL 和 RET 指令是专门用来进行函数调用和返回的。理解这两条指令有助于我们深刻理解函数调用的内部过程和栈的使用方法。

22.3.1 CALL 指令

CALL 指令是 x86 CPU 中专门用作函数调用的指令，简单来说，它的作用就是将当前的程序指针（EIP 寄存器）值保存到栈中（称为 linking information），然后转移到（branch to）目标操作数所指定的函数（被调用过程）继续执行。

根据被调用过程是否位于同一个代码段，CALL 调用被分为近调用（Near Call）和远调用（Far Call）两种。对于近调用，CPU 所执行的操作如下。

1. 将 EIP 寄存器的当前值压入到栈中供返回时使用。
2. 将被调用过程的偏移（相对于当前段）加载到 EIP 寄存器中。
3. 开始执行被调用过程。

对于远调用，CPU 所执行的操作如下。

1. 将 CS 寄存器的当前值压入到栈中供返回时使用。
2. 将 EIP 寄存器的当前值压入到栈中供返回时使用。
3. 将包含被调用过程的代码段的段选择子加载到 CS 寄存器中。
4. 将被调用过程的偏移加载到 EIP 寄存器中。

5. 开始执行被调用过程。

易见，近调用和远调用的差异在于是否处理段寄存器，因为近调用是发生在一个代码内的调用，因此不需要向栈中压入和切换代码段，而远调用因为发生在不同代码段间，因此需要保存和切换代码段。近调用和远调用的机器码是不一样的，因此，编译器在编译时决定使用何种调用。在编写 16 位的 Windows 程序（Windows 3.x）时，在某些函数声明中会带有 FAR 声明，表示调用这样的函数时应该使用远调用。对于 NT 系列的 Windows，因为使用了平坦内存模型，同一进程内的代码都在一个大的 4GB 段中，因此不必再考虑段的差异，几乎所有时候使用的都是近调用。

22.3.2 RET 指令

RET 指令用于从被调用过程返回到发起调用的过程。RET 指令可以有一个可选的参数 n，用于指定 ESP 寄存器要递增的字节数，ESP 递增 n 个字节相当于从栈中弹出 n 个字节，经常用来释放压在栈上的参数。相对于近调用的返回被称为近返回（Near Return），类似，相对于远调用的被称为远返回（Far Return）。

对于近返回，CPU 所执行的操作如下。

1. 将位于栈顶的数据弹出到 EIP 寄存器。这个值应该是发起近调用时 CALL 指令压入的返回地址。
2. 如果 RET 指令包含参数 n，那么便将 ESP 寄存器的字节数递增 n。
3. 继续执行程序指针所指向的指令，通常就是父函数中调用指令的下一条指令。

对于远返回，在第 1 和第 2 步间，CPU 会弹出执行远调用时压入的 CS 寄存器。从以上过程我们看到，RET 指令只是单纯地返回到执行这条指令时栈顶所保存的地址，如果栈寄存器（ESP）没有指向合适的位置或栈上的地址被破坏了，那么 RET 指令就会返回到其他地方，这也正是缓冲区溢出攻击的基本原理，我们将在第 22.10 节讨论更多细节。

22.3.3 观察函数调用和返回过程

下面通过一个程序实例来加深大家的理解。清单 22-3 所示的代码是我们特意编制的一个名为 HiStack 的控制台程序的源代码（完整代码位于 code\chap22\histack）。

清单 22-3 HiStack 程序的源代码

```

1 #include <stdio.h>
2
3 int __stdcall Proc(int n)
4 {
5     int a=n;
6     printf("A test to inspect stack, n=%d,a=%d.",n,a);
7     return n*a;
8 }
9 int main()
10 {

```

```

11     return Proc(122);
12 }

```

因为编译器会为调试版本分配额外的变量和加入栈检查功能（将在第 22.11 节讨论），所以我们使用发布版本（release）来进行观察。为了可以在调试器中看到函数名信息，我们让编译器为发布版本产生调试符号。其操作过程是：选项目属性，切换到对发布版本（Win32 Release），然后在链接设置中选中 Generate debug info（生成调试信息）复选框。

接下来，使用 WinDBG 打开（Open Executable...）编译好的发布版本的 HiStack.exe 程序。使用 bp histack!main 命令对 main 函数设置一个断点，然后让程序执行到这个断点。这时从反汇编窗口（disassembly）可以看到以上源代码所对应的汇编代码（清单 22-4）。

清单 22-4 HiStack 程序的 main 和 Proc 函数所对应的汇编指令

```

1  HiStack!Proc:
2  00401000 56          push    esi
3  00401001 8b742408    mov     esi,[esp+0x8]
4  00401005 56          push    esi
5  00401006 56          push    esi
6  00401007 6830804000  push    0x408030
7  0040100c e81f000000  call    HiStack!printf (00401030)
8  00401011 83c40c      add     esp,0xc
9  00401014 8bc6        mov     eax,esi
10 00401016 0fafc6     imul   eax,esi
11 00401019 5e          pop    esi
12 0040101a c20400     ret    0x4
13 0040101d 90          nop
14 0040101e 90          nop
15 0040101f 90          nop
16 HiStack!main:
17 00401020 6a7a        push    0x7a
18 00401022 e8d9ffff    call    HiStack!Proc (00401000)
19 00401027 c3          ret

```

键入 r eip, esp 命令（显示寄存器值），可以看到此时的 EIP（程序指针）和 ESP（栈指针）寄存器的值：

```
0:000> r eip, esp
eip=00401020 esp=0012ff84
```

也就是说，目前栈顶的地址是 0012ff84；CPU 即将执行 00401020 处的指令，即 17 行的 push 0x7a，0x7a 即要传递给 Proc 函数的参数 122。

键入 p 命令单步执行一次，再显示 EIP 和 ESP 寄存器的值：

```
0:000> r eip, esp
eip=00401022 esp=0012ff80
```

可见程序指针指向下一条指令（18 行的 CALL），ESP 也指向了新的地址 0012ff80。新的地址与刚才的地址相差 4 个字节，这正好是用于存放刚刚压入的 0x7a 所需的空间。使用内存显示命令可以观察到地址 0012ff80 处的内容就是 0x7a。这也验证了 ESP 寄存器总是指向位于栈定的数据。

```
0:000> dd 0012ff80 11
0012ff80 0000007a
```

键入 t 命令跟踪执行 18 行的 CALL 指令，会发现执行光标移动到函数 Proc 的第一条指令，即清单 20-2 的第 2 行，再次观察 EIP 和 ESP 寄存器的值：

```
0:000> r esp, eip
esp=0012ff7c eip=00401000
```

可以发现 EIP 指向即将执行的函数 Proc 的第一条指令，ESP 又递减了 4 个字节，根据我们前面对 CALL 指令的介绍，这应该是 CALL 指令压入的函数返回地址。因为是近调用，因此只须要压入偏移地址，不须要压入段寄存器。使用 dd 命令观察栈顶的数据，可以发现其值为 00401027，这正是 19 行的指令的地址，即执行 Proc 函数后应该返回到的地址。

```
0:000> dd 0012ff7c 11
0012ff7c 00401027
```

我们在介绍 CALL 指令时说，CPU 压入的是 EIP 寄存器的当前值，也就是说 CPU 在执行 CALL 指令时，EIP 指针已经指向了其后的那条指令，这与 EIP 寄存器总是指向即将执行的下一条指令相吻合。

第 2 到第 11 行是 Proc 函数内部的代码，我们不再详细介绍，将光标移到第 12 行，然后按 Ctrl+F10 直接执行到这一行。输入 r esp, eip 命令看此时的 EIP 和 ESP 寄存器的值：

```
0:000> r esp, eip
esp=0012ff7c eip=0040101a
```

可见接下来要执行的是位于第 12 行的 ret 4 指令，而此时的 ESP 值与进入 Proc 函数时的值相同，使用 DD 命令显示 ESP 地址的值，仍然是函数的返回地址，这便是所谓的保持栈平衡。保持栈平衡是判断一个函数是否正确使用栈的一个基本标准。只有做到保持栈平衡，函数才可能返回到正确的位置。

```
0:000> dd 0012ff7c 11
0012ff7c 00401027
```

按 p 命令再次单步执行，会发现执行光标如预期的移动到第 19 行，即从 Proc 函数中返回到 main 函数中，准备执行 CALL 指令之后的下一条指令。再次观察 ESP 指针：

```
0:000> r esp, eip
esp=0012ff84 eip=00401027
```

可见，ESP 值从 0012ff7c 变为 0012ff84，递增了 8 个字节。根据我们前面对 ret 指令的介绍，其中 4 个字节是由于弹出返回地址到 EIP 寄存器导致的，另 4 个字节是按参数 n (n=4) 指定的值释放参数所占用的空间而导致的。另外大家可以发现，此时的 ESP 值与进入 main 函数时的值是完全一致的，也就是说调用 Proc 函数没有影响 ESP 的值。这样看来，尽管单纯从 main 函数看，第 17 行的压栈操作 push 0x7a 似乎没有对应的弹出操作，但是因为被调用函数进行了清理参数的操作，所以栈还是平衡的。这种清理栈中参数的方法就是所谓的被调用者清理栈，我们将在 22.7 节讨论函数调用协定时进一步介绍各种清理栈的方法。

顺便说一下，第 13 到第 15 行的 nop 指令是用来填补空位进行内存对齐的，CPU 执行 nop 指令时除了递增 EIP 寄存器外不做任何其他操作。

22.3.4 跨特权级调用

通常，发起调用的函数和被调用的函数都是位于同一个特权级的代码中的，这种调用叫同特权级调用。另一种情况是位于不同特权级代码段中的代码相互调用，被称为跨特权级调用。跨特权级调用通常是通过一个所谓的调用门（Call Gate）来完成的。调用门的全称是调用门描述符（Call-Gate Descriptor），其结构（图 22-2）与中断描述符非常类似。调用门描述符可以出现在 GDT 和 LDT 表中，不可以出现在 IDT 表中。

图中，Segment Selector（段选择子）用来指定被调用代码所在的段，Offset in Segment 用来指定被调用代码的偏移地址。DPL 代表了这个段描述符的特权级别。下面以低特权级的代码调用高特权级的代码为例介绍 CPU 执行跨特权级调用的过程。

1. 进行访问权限检查，如果检查失败，则产生保护性异常。
2. 将 SS、ESP、CS、EIP 寄存器的值临时保存到 CPU 内部。

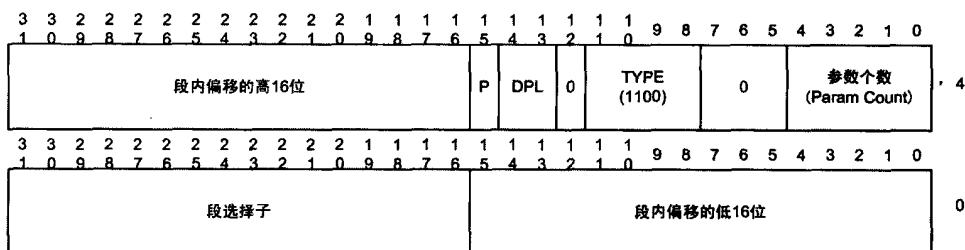


图 22-2 调用门描述符（Call-Gate Descriptor）(32 位)

3. 从任务状态段（TSS）中找到目标代码所处特权级的栈信息，并将段选择子和栈指针加载到 SS 和 ESP 寄存器中。这一步进行的动作经常被称为栈切换。
4. 将第二步保存的 SS 和 ESP 寄存器值依次压入新的栈。这一步的目的是将发起调用的代码的栈信息压入到被调用代码所使用的栈。
5. 将参数从发起调用的栈复制到新的栈。调用门中的 Param Count（参数个数）字段描述了要复制的参数个数，这里是以 DWORD 为单位的，最多可以复制 32 个 DWORD。
6. 将第二步保存的 CS 和 EIP 值压入到新的栈中。
7. 将要调用的代码段的段选择子和函数偏移分别加载到 CS 和 EIP 寄存器中。
8. 开始执行被调用的代码。

当被调用的代码执行完毕时，可使用 RET 指令返回到发起调用的函数，其过程从略。

CPU 在处理中断或异常时，如果 CPU 当前正在执行用户代码段中的低特权级代码，

而中断处理例程位于高特权级别的内核代码段中，那么 CPU 所做的动作与上面的跨特权级调用非常类似，只不过使用的是中断描述符。Windows 操作系统使用 INT 2E 或专门的快速系统调用指令来实现从用户态（低特权）到内核态（高特权）的系统调用，没有使用调用门。但是某些根件（Rootkit）使用调用门来从用户态调用内核空间的代码。

22.4 局部变量和栈帧

所谓局部变量（Local Variables）就是指作用域和生命周期都局限于所在函数或过程范围内的变量，它是相对全局可见的全局变量（Global Variable）而言的。编译器在为局部变量分配空间时通常有两种做法：使用寄存器和使用栈。

从性能上看使用寄存器来分配局部变量是最好的，因为访问寄存器比访问内存还要快许多倍，但由于寄存器的空间和数量都非常有限（尤其对于 x86 系统，参见第 2.1 节），所以字符串或数组这样的局部变量是不适合分配在寄存器中的。通常编译器只会把频繁使用的临时变量分配在寄存器中，比如 for 循环中的循环变量。当编译器的优化选项打开时，编译器会充分利用可用的寄存器来给临时变量使用，以提高程序的性能。对于调试版本，优化选项默认是关闭的，编译器会在栈上分配所有变量。在 C/C++ 程序中，可以在声明变量时加上 register 关键字，请求编译器在可能的情况下将该变量分配在寄存器中。但也只是“尽可能”，并不能保证所描述的变量一定被分配在寄存器中。大多数时候，编译器还是根据全局设置和编译器自身的逻辑来决定是否把一个变量分配在寄存器中。

编译器在编译阶段根据变量特征和优化选项为每个局部变量选择以上两种分配方法之一。使用栈来分配局部变量是最主要的做法，大多数局部变量都是分配在栈上的。

根据分配方法，我们把分配在寄存器中的局部变量称为寄存器变量（Register Variable），把分配在栈上的局部变量又称为栈变量（Stack Variable）。本章讨论的重点是分配在栈上的局部变量。因此以下如不特别说明，我们说的局部变量都是指栈变量。因为分配在栈上的变量和对象会随着函数的调用和返回而自动分配和释放，所以栈有时也被称为自动内存。

22.4.1 局部变量的分配和释放

概言之，局部变量的分配和释放是由编译器插入的代码通过调整栈指针（Stack Pointer）的位置来完成的。编译器在编译时，会计算当前代码块（如函数或过程）中所声明的所有局部变量所需要的空间，并将其按照内存对齐规则取为满足对齐要求的最接近整数值。在 32 位系统中，内存分配是按 4 字节对齐的，这意味着不满 4 字节的空间分配会按 4 字节来分配。举例来说，如果某个 ANSI 类型的字符数组的长度是 13 个字节，那么编译器会分配 16 个字节。

计算好所需空间后，编译器会插入适当的指令来调整栈指针，为局部变量挪动（分配）出空间来。对于 x86 系统，栈指针保存在 ESP 寄存器中，所以调整栈指针实际上也就是调整 ESP 寄存器的值。

编译器有几种方法来调整 ESP 寄存器的值，一种方法是直接对其进行加减运算，比如：

```
sub esp, 10h ; 将 ESP 寄存器中的值减去 16。
add esp, 10h ; 将 ESP 寄存器中的值加上 16。
add esp, 0FFFFFFFCC ; 加上一个负数 (-34)，相当于减去 34。
```

另一种方法是使用 PUSH 和 POP 指令，因为这两个指令也会改变 ESP 寄存器的值。PUSH 和 POP 指令比 SUB 和 ADD 指令执行得更快，所以当只需要一两个 PUSH 或 POP 指令就能达到目的时，编译器就会使用 PUSH 或 POP 指令，而不用加减指令。另外 enter、leave 和 ret 指令也可以改变 ESP 寄存器的值。

因为栈是向低地址方向生长的，所以分配空间时，是使 ESP 寄存器递减，释放空间时是使 ESP 寄存器递增。下面举个例子来加深大家的印象。清单 22-5 列出了本节的示例程序 LocalVar（完整代码位于 code\chap22\LocalVar 目录）中的 FuncA 函数的源代码。

清单 22-5 LocalVar 中的 FuncA 函数的源代码

```
1 int FuncA()
2 {
3     int l,m,n;
4     char sz[]="Advanced SW Debugging";
5     l=sz[0];
6     m=sz[4];
7     n=sz[8];
8     return l*m*n;
9 }
```

从 FuncA 的源代码我们可以看到，该函数中共定义了四个局部变量，三个整数和一个字符串。三个整数需要 12 个字节的空间，可以分配在寄存器中，也可以分配在栈上。字符串中包含了 21 个字符，加上末尾的结束符 0，共需 22 个字节，字符串变量是无法分配在寄存器中的，因此一定要分配在栈上。下面我们通过观察汇编代码来看编译器是如何分配这些变量的。清单 22-6 列出了 FuncA 函数的反汇编代码，它是使用 WinDBG 调试器针对 LocalVar 程序的发布版本（优化选项是速度最大化）产生的。

清单 22-6 FuncA 函数的反汇编代码（发布版本）

```
1 LocalVar!FuncA:
2 00401000 83ec18      sub    esp,0x18
3 00401003 b905000000  mov    ecx,0x5
4 00401008 56          push   esi
5 00401009 57          push   edi
6 0040100a be30704000  mov    esi,0x407030
7 0040100f 8d7c2408    lea    edi,[esp+0x8]
8 00401013 f3a5        rep    movsd
9 00401015 66a5        movsw
10 00401017 0fbe442410  movsx  eax,byte ptr [esp+0x10]
11 0040101c 0fbe4c240c  movsx  ecx,byte ptr [esp+0xc]
```

```

12 00401021 0fbe542408    movsx   edx,byte ptr [esp+0x8]
13 00401026 0fafc1        imul    eax,ecx
14 00401029 5f             pop     edi
15 0040102a 0fafc2        imul    eax,edx
16 0040102d 5e             pop     esi
17 0040102e 83c418        add    esp,0x18
18 00401031 c3             ret

```

下面通过分析清单 22-6 来看一看编译器到底是如何为以上四个变量分配空间的。从第 2 行可以看到，ESP 寄存器被递减 24（个字节），也就是分配了 24 个字节的栈空间。这个空间是为字符串变量分配的，尽管字符串实际需要 22 个字节，但因为内存对齐的需要，所以编译器会实际分配 24 个字节。看来编译器是想使用寄存器来存储三个整数变量了，没有为它们在栈中分配空间。第 4 和第 5 行是保存 ESI 和 EDI 寄存器，因为接下来要使用它们操纵字符串将变量 sz 初始化为字符串常量"Advanced SW Debugging"。第 6 行将字符串常量的地址（源地址）赋给 ESI 寄存器，第 7 行将目标字符串的地址加载到 EDI 寄存器中。目标字符串也就是栈上的 sz 变量。因为第 4 和 5 行压入了两个 4 字节的寄存器，所以现在 sz 变量是在栈顶加 8 的位置。第 8 行是条循环指令，循环的次数在 ECX 中，第 3 行将 ECX 寄存器设为 5，这就意味着 CPU 会执行 MOVS 指令 5 次，每次将 ESI 开始的 4 个字节赋给 EDI 指定的地址，然后将 ESI 和 EDI 递增 4，ECX 递减 1。循环 5 次可以复制 20 个字节，因为字符串的长度是 22 个字节（包括末尾的 0），所以第 9 行又复制了 2 个字节（一个 WORD）。运行到这里时，我们先使用寄存器命令查看栈指针寄存器 ESP 的值，然后再使用内存观察变量观察栈内的详细情况（清单 22-7）。

清单 22-7 观察栈上的原始数据

```

0:000> r esp
esp=0012ff60
0:000> dd /c1 esp
0012ff60 00090000 <-这是栈内最接近栈顶的四个字节，即第 5 行压入的 EDI 寄存器的以前值
0012ff64 0171fa9c <-这是第 4 行压入的 ESI 寄存器的以前值
0012ff68 61766441 <-从这个地址开始就是局部变量 sz 的空间了，这四个字节分别是 avda
0012ff6c 6465636e <-字符 decn
0012ff70 20575320 <-字符 ws (ws 前后各有一个空格)
0012ff74 75626544 <-字符 ubeD
0012ff78 6e696767 <-字符 nigg
0012ff7c 00400067 <-字符 g 和结束符 0, 0040 是因为内存对齐而多分配的两个字节
0012ff80 00401065 <-这是本函数的返回地址，即 main 函数中调用 FuncA 的下一条指令的地址
0012ff84 00401134 <-这是 main 函数的返回地址，这下面是 main 函数的参数

```

要说明的是，因为上面我们是按四字节（DWORD）格式来显示的，所以对于字符串数据来说，其次序是反的，以第 3 行为例，这 4 个字节的起始地址是 0012ff68，首字节是 0x41 (A)，然后依次是 0x64 (d)、0x76 (v) 和 0x61 (a)。

接下来的第 10~12 行，是将字符串数组的元素赋给分配在寄存器中的局部变量，EDX 保存的是变量 l，ECX 是变量 m，EAX 是变量 n。这三行的顺序和源代码的顺序是相反的，但这不会影响计算的结果。第 13 和 15 行是做乘法运算，计算的结果在 EAX

寄存器中，刚好作为返回值返回。第 17 行，ESP 寄存器的值被加 0x18，释放了第 2 行所分配的空间。

22.4.2 EBP 寄存器和栈帧

对于分配在栈上的局部变量，编译器是如何来引用它们的呢？这是软件调试中经常遇到的问题，当我们跟踪反编译过来的汇编指令时，如何知道当前指令是否使用了局部变量呢（如果使用，又是哪个）？

对于前面 FuncA 函数中的局部变量 sz，第 7 行的指令是通过 `esp+8` 来引用这个变量的，也就是使用 ESP 作为参照物来引用局部变量 sz。但我们知道所有压栈和弹出操作都会影响 ESP 的值，因此使用 ESP 寄存器来引用变量的缺点是它不稳定，ESP 的值变化了，引用变量的偏移值也要变化。为了更好地说明这个问题，我们先看一下清单 22-8 所示的 FuncB 函数（仍属于 LocalVar 程序）。

清单 22-8 FuncB 函数的源代码

```
1 void FuncB(char * szPara)
2 {
3     char szTemp[5];
4     strncpy(szTemp,szPara,sizeof(szTemp)-1);
5     printf("%s;Len=%d.\n",szTemp,strlen(szTemp));
6 }
```

清单 22-9 是 FuncB 函数编译后的汇编代码（发布版本）。为了便于理解，每条指令后面都加上了简明的解释。

清单 22-9 FuncB 函数的反汇编代码（发布版本）

```

1 LocalVar!FuncB:
2 00401040 8b442404    mov    eax,[esp+0x4] ; 将参数 szPara 赋给 EAX 寄存器
3 00401044 83ec08    sub    esp,0x8 ; 为变量 szTemp 分配空间
4 00401047 8d4c2400    lea    ecx,[esp] ; 将 szTemp 的有效地址放入 ECX 寄存器
5 0040104b 57         push   edi ; 保存 EDI 寄存器的以前值
6 0040104c 6a04         push   0x4 ; 准备调用 strncpy, 压入参数 4
7 0040104e 50         push   eax ; 压入 szPara
8 0040104f 51         push   ecx ; 压入 szTemp
9 00401050 e88b000000    call   LocalVar!strncpy (004010e0) ; 调用函数 strncpy
10 00401055 8d7c2410    lea    edi,[esp+0x10] ; 将 szTemp 的地址放入 EDI
11 00401059 83c9ff    or     ecx,0xffffffff ; 将 ECX 寄存器设为 -1
12 0040105c 33c0         xor    eax,eax ; 将 EAX 置为 0
13 0040105e 83c40c    add    esp,0xc ; 调整栈指针, 释放 6、7、8 行压入的参数
14 00401061 f2ae         repne scasb ; 在 EDI 开始的字符串中寻找 0 (AL), 即求长度
15 00401063 f7d1    not    ecx ; 对 ECX 取反, 使其由 -4 变为 4*
16 00401065 49         dec    ecx ; 遂减 ECX, 变为 3
17 00401066 8d542404    lea    edx,[esp+0x4] ; 将变量 szTemp 的地址放入 EDX
18 0040106a 51         push   ecx ; 压入 ECX, 即 strlen(szTemp)
19 0040106b 52         push   edx ; 压入 EDX, 即 szTemp
20 0040106c 6848804000    push   0x408048 ; 压入字符串常量, 即 "%s;Len=%d.\n"
21 00401071 e82a000000    call   LocalVar!printf (004010a0) ; 调用 printf 函数
22 00401076 83c40c    add    esp,0xc ; 释放 18~20 行压入的参数

```

```

23 00401079 5f      pop    edi ; 弹出第 5 行压入的 EDX 寄存器, 恢复其以前值
24 0040107a 83c408   add    esp, 0x8 ; 释放分配给局部变量的空间
25 0040107d c3      ret    ; 返回

```

*以使用参数“Dbg”来调用 FuncB 为例。

在清单 22-9 的汇编代码中，共有三次访问局部变量 szTemp，分别是第 4、10 和 17 行。尽管都是引用同一个局部变量，但是我们看到，三次引用的表达就不同，分别是 esp、esp+0x10 和 esp+0x4。那么这是为什么呢？原因是 ESP 值在变化。第 4 行时，刚刚为 szTemp 分配好栈空间，因此它就位于栈顶，所以 ESP 寄存器的值就是 szTemp 变量的起始地址。到了第 10 行时，由于 5、6、7、8 四行各有一条 PUSH 语句，压入了 16 个字节，szTemp 离栈顶的距离变为 16(0x10)个字节，因此这时就要用 esp+0x10 来引用它了。第 13 行因为清理 6~8 行压入的参数向 ESP 加 0xC，这又使 szTemp 离栈顶近了，因此第 17 行是用 esp+4 来引用 szTemp。

通过这个例子，我们看到，尽管可以使用相对于栈顶（ESP 寄存器）的偏移来引用局部变量，但是因为 ESP 寄存器经常变化，所以用这种方法引用同一个局部变量的偏移值是不固定的。这种不确定性对于 CPU 来说不成什么问题，但在调试时，如果要跟踪这样的代码，那么很容易就被转得头晕眼花，因为现实的函数大多有多个局部变量，可能还有层层嵌套的循环，栈指针变化非常频繁。

为了解决以上问题，x86 CPU 设计了另一个寄存器，这就是 EBP 寄存器。EBP 的全称是 Extended Base Pointer，即扩展的基址指针。使用 EBP 寄存器，函数可以把自己将要使用的栈空间的基准地址记录下来，然后使用这个基准地址来引用局部变量和参数。在同一函数内，EBP 寄存器的值是保持不变的，这样函数内的局部变量便有了一个固定的参照物。

通常，一个函数在入口处将当时的 EBP 值压入堆栈，然后把 ESP 值（栈顶）赋给 EBP，这样 EBP 中的地址就是进入本函数时的栈顶地址，这一地址上面（地址值递减方向）的空间便是这个函数将要使用的栈空间，它下面（地址值递增方向）是父函数使用的空间。如此设置 EBP 后，便可以使用 EBP 加正数偏移来引用父函数的内容，使用 EBP 加负数偏移来引用本函数的局部变量，比如 EBP+4 指向的是 CALL 指令压入的函数返回地址；EBP+8 是父函数压在栈上的第一个参数，EBP+0xC 是第二个参数，依次类推；EBP-n 是第一个局部变量的起始地址（n 为变量的长度）。

因为在将栈顶地址（ESP）赋给 EBP 寄存器之前先把旧的 EBP 值保存在栈中，所以 EBP 寄存器所指向的栈单元中保存的是前一个 EBP 寄存器的值，这通常也就是父函数的 EBP 值。类似的父函数的 EBP 所指向的栈单元中保存的是更上一层函数的 EBP 值，依此类推，直到当前线程的最顶层函数。这也正是栈回溯的基本原理。

下面再以刚才的 LocalVar 程序为例看看 EBP 寄存器的实际用法，我们将 FuncB 函数复制并改名为 FuncC，然后在该函数前面加上#pragma optimize("", off) 告

告诉编译器不要对此函数进行优化，在其后面再加上#pragma optimize("", on)恢复优化功能。这样编译后的 FuncC 函数所对应的反汇编代码如清单 22-10 所示。

清单 22-10 FuncC 函数的反汇编代码（发布版本，关闭优化）

1	LocalVar!FuncC:	
2	00401080 55	push ebp ; 压入 EBP 寄存器的当前值
3	00401081 bbec	mov esp,ebp ; ESP 寄存器的值（栈顶）赋给 EBP
4	00401083 83ec08	sub esp,0x8 ; 为变量 szTemp 分配空间
5	00401086 57	push edi ; 保存 EDI 寄存器的以前值
6	00401087 6a04	push 0x4 ; 准备调用 strcpy, 压入参数 4
7	00401089 b84508	mov eax,[ebp+0x8] ; 将 szPara 赋给 EAX
8	0040108c 50	push eax ; 压入 szPara
9	0040108d 8d4df8	lea ecx,[ebp-0x8] ; 将 szTemp 的有效地址放入 ECX 寄存器
10	00401090 51	push ecx ; 压入 szTemp
11	00401091 e88aa000000	call LocalVar!strcpy (00401120) ; 调用函数 strcpy
12	00401096 83c40c	add esp,0xc ; 调整栈指针，释放第 6、8、10 行压入的参数
13	00401099 8d7df8	lea edi,[ebp-0x8] ; 将 szTemp 放入 EDI
14	0040109c 83c9ff	or ecx,0xffffffff ; 将 ECX 寄存器设为 -1
15	0040109f 33c0	xor eax,eax ; 将 EAX 置为 0
16	004010a1 f2ae	repne scasb ; 在 EDI 开始的字符串中寻找 0 (AL)，即求长度
17	004010a3 f7d1	not ecx ; 对 ECX 取反
18	004010a5 83c1ff	add ecx,0xffffffff ; 对 ECX 减 1
19	004010a8 51	push ecx ; 压入 ECX，即 strlen(szTemp)
20	004010a9 8d55f8	lea edx,[ebp-0x8] ; 将 szTemp 的有效地址放入 EDX 寄存器
21	004010ac 52	push edx ; 压入 EDX，即 szTemp
22	004010ad 6848804000	push 0x408048 ; 压入字符串常量，即 "%s;Len=%d.\n"
23	004010b2 e829000000	call LocalVar!printf (004010e0) ; 调用 printf 函数
24	004010b7 83c40c	add esp,0xc ; 释放第 19、21、22 行压入的参数
25	004010ba 5f	pop edi ; 弹出第 5 行压入的 EDX 寄存器，恢复其以前值
26	004010bb 8be5	mov esp,ebp ; 将 EBP 寄存器的值赋给 ESP
27	004010bd 5d	pop ebp ; 恢复 EBP 寄存器的以前值
28	004010be c3	ret ; 返回
29	004010bf cc	int 3 ; 补位用的断点指令

比较清单 22-9 和清单 22-10，尽管清单 22-10 中也有三次对局部变量 szTemp 的引用（第 9、13 和 20 行），但是使用的都是[ebp-0x8]，而不是像清单 22-9 中那样三次各使用不同的偏移值。显然，清单 22-10 中的代码比清单 22-9 中的更容易理解，更容易辨识出局部变量和参数。

图 22-3 画出了当 CPU 执行 FuncC 函数时栈的状态，确切地说，是 CPU 执行完 PUSH EAX 指令（第 8 行）后的状态，此时的 EBP 和 ESP 寄存器的值分别为：

```
0:000> r ebp,esp
ebp=0012ff74 esp=0012ff60
```

图 22-3 中的柱体代表栈，表面的数字是栈中的内容，左侧的数字是地址，右侧是对栈中数据的说明。图中详细画出了 FuncC 函数、main 函数所使用的栈空间，包括局部变量、返回地址和参数。因为局部变量 szTemp 没有被初始化，所以它目前的值是随机的。使用 da 可以显示栈中参数 szPara 的值：

```
0:000> da 408054
00408054 "Dbg"
```

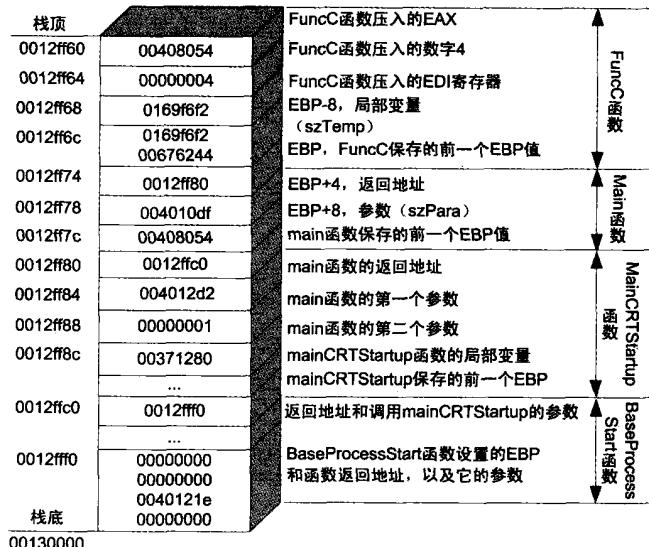


图 22-3 执行 LocalVar!FuncC 函数时栈中的数据布局

在 szPara 参数下面（更高地址）依次是 main 函数保存的 EBP 值、main 函数的返回地址和 main 函数的参数。我们知道，main 函数的第一个参数代表命令行参数的个数，此时为 1，代表第二个参数所指向的字符串数组有一个元素，使用如下命令可以看到其内容：

```
0:000> dd 00371280 12
00371280 00371288 00000000
0:000> da 00371288
00371288 "C:\dig\dbg\author\code\bin\relea"
003712a8 "se\LocalVar.exe"
```

再下面是 mainCRTStartup 函数的数据、局部变量、保存的 EBP 值、返回地址和参数。最下面是系统函数 BaseProcessStart 函数的栈内容。因为 BaseProcessStart 函数是这个线程的起始，它的函数返回地址（0x12fff4 处）是 0，它保存的 EBP 值也是 0，0040121e 是它的参数，即这个程序的入口地址（mainCRTStartup）。清单 22-11 给出了栈中的原始数据，用来帮助大家理解图 22-3。

清单 22-11 直接观察栈上的数据（对应图 22-3 中的状态）

```
0:000> dd esp 150
0012ff60 00408054 00000004 0169f6f2 0169f6f2
0012ff70 00676244 0012ff80 004010df 00408054
0012ff80 0012ffc0 004012d2 00000001 00371280
0012ff90 003712d8 0169f6f2 0169f76c 7ffd0000
0012ffa0 00000001 00000006 0012ff94 80616337
0012ffb0 0012ffe0 004028f0 004070e0 00000000
0012ffc0 0012fff0 7c816fff7 0169f6f2 0169f76c
0012ffd0 7ffd000 805441fd 0012ffc8 87681468
0012ffe0 ffffffff 7c839a30 7c817000 00000000
0012fff0 00000000 00000000 0040121e 00000000
```

从上面的分析我们看到，尽管栈中的数据是连续存储的，好像所有数据都混作一团（清单 22-11），但事实上，它们是按照函数调用关系依次存放的，而且这种顺序关系非常严格。为了更好地描述和指代栈中的数据，我们把每个函数在栈中所使用的区域称为一个栈帧（Stack Frame），有时也简称为帧。在图 22-3 中，栈内共有 4 个栈帧，分别属于 FuncC 函数、main 函数、mainCRTStartup 函数和 BaseProcessStart 函数。

关于栈帧还有以下几点值得说明。

1. 在一个栈中，依据函数调用关系，发起调用的函数（Caller）的栈帧在下面（高地址方向），被调用的函数的栈帧在上面。
2. 每发生一次函数调用，便产生一个新的栈帧，当一个函数返回时，这个函数所对应的栈帧被消除（eliminated）。
3. 线程正在执行的那个函数所对应的栈帧被位于栈的最顶部，它也是栈内仍然有效的最年轻（建立时间最晚）栈帧。

从清单 22-10 中，我们可以归纳出建立栈帧的典型指令序列：

```
00401080 55      push    ebp ; 压入 EBP 寄存器的当前值
00401081 8bec    mov     ebp,esp ; ESP 寄存器的值（栈顶）赋给 EBP
00401083 83ec08  sub    esp,0XXX ; 调整栈顶，为局部变量分配空间
```

在函数的出口，通常有对应的消除栈帧的指令序列：

```
004010bb 8be5    mov     esp,ebp ; 将 EBP 寄存器的值赋给 ESP
004010bd 5d      pop    ebp ; 恢复 EBP 寄存器的以前值
```

以上两段指令通常分别出现在一个函数的开始和结束处，分别被称为函数的 **prolog**（序言）和 **epilog**（结语）。在 Windows 的很多系统函数中，我们可以看到函数入口附近会调用 **_SEH_prolog**，出口附近调用 **_SEH_epilog**，这两个函数便是用来建立栈帧和消除栈帧的，同时还具有登记和注销结构化异常处理器（SEH）的功能。

22.4.3 帧指针和栈帧的遍历

指向每个栈帧的指针被称为帧指针（Frame Pointer），因为在 x86 系统中，通常使用 EBP 寄存器作为帧指针使用，所以经常使用 ChildEBP 或 EBP 来代替帧指针。

当一个函数建立一个新的栈帧时，它会将当时的 EBP 值压入栈保存起来，然后立即把当时的栈指针值赋给 EBP 寄存器。这样，EBP 的值便是新栈帧的基础地址，而这个地址在栈中的值便是 EBP 的以前值，也就是前一个栈帧的基础地址。以图 22-3 所示的情况为例，此时的 EBP 寄存值是 0x0012ff74，栈内这个地址的值是 0x0012ff80，这正是 main 函数的栈帧地址；再观察栈地址 0x0012ff80 处的值是 0x0012ffc0，这正是 mainCRTStartup 的栈帧地址，依次类推，我们就可以遍历整个栈中的所有栈帧。这也正是调试器的 Calling Stack 功能（显示函数调用序列）的基本原理。

下面通过一个试验来加深大家的印象。使用 WinDBG 打开 Release 目录下的 LocalVar.exe，使用 bp LocalVar!FuncC+9 设置一个断点，然后键入 g 命令执行。程序应该停在清单 22-10 所示的第 7 行处。此时键入 r ebp, esp, eip 命令，显示出当前的 EBP（帧指针）、ESP（栈指针）和 EIP（程序指针）寄存器值：

```
0:000> r ebp, esp, eip
ebp=0012ff74 esp=0012ff64 eip=00401089
```

EBP 的值就是当前栈帧的地址，根据图 22-3，EBP+4 应该是返回地址，EBP+8 是第一个参数。使用内存观察命令可以看到这些值：

```
0:000> dd ebp+4 14
0012ff78 004010df 00408054 0012ffc0 004012d2
```

使用 ln eip（列出与 EIP 值最近的符号）命令可以找到即将执行的下一条指令：

```
0:000> ln eip
(00401080) LocalVar!FuncC+0x9 | (004010c0) LocalVar!main
```

因为 eip=00401089，所以 LocalVar!FuncC 离得更近。将以上内容放在一起便得到当前栈帧的基本信息如下：

帧指针	返回地址	参数	程序指针
0012ff74	004010df	00408054	0012ffc0 004012d2 LocalVar!FuncC+0x9

那么如何得到父函数栈帧的情况呢？根据刚才的介绍，每一帧的帧指针指向的就是其外层帧的基址。也就是说，EBP 所代表地址处保存的就是外层帧的基础地址。

显示 EBP 的值：

```
0:000> dd ebp l1
0012ff74 0012ff80
```

这说明父函数栈帧的基地址是 0012ff80，这个地址+4 偏移便是返回地址，+8 便是参数：

```
0:000> dd 0012ff80+4 14
0012ff84 004012d2 00000001 00370ec0 00370f28
```

使用 ln 命令寻找这个返回地址对应的符号：

```
0:000> ln 004010df
(004010c0) LocalVar!main+0x1f | (004010e6) LocalVar!printf
```

这说明 FuncC 的父函数是 main 函数，将以上信息放在一起，便得到了 main 函数栈帧的数据，依次类推，我们可以得到再上一级栈帧的情况，重复这一过程直到帧指针指向的地址为空，说明已经到达栈底。最后将所有数据放在一起便得到清单 22-12 所示的函数调用序列。

清单 22-12 手工产生的栈回溯信息

帧指针	返回地址	参数	程序指针
0012ff74	004010df	00408054	0012ffc0 004012d2 LocalVar!FuncC+0x9
0012ff80	004012d2	00000001	003d0ec0 003d0f28 LocalVar!main+0x1f
0012ffc0	7c816d4f	00090000	08ebfa9c 7ffd7000 LocalVar!mainCRTStartup+0xb4
0012fff0	00000000	0040121e	00000000 78746341 kernel32!BaseProcessStart+0x23

以上数据每一行描述一个栈帧，由上至下分别描述了栈中由顶到底的各个栈帧。这样从当前栈帧层层追溯而得到函数调用记录的过程被称为栈回溯 (Stack Backtrace)。栈回溯信息对软件调试有着非常重要的意义，比如可以通过参数值核对参数的正确性，根据帧指针观察局部变量，根据程序指针信息了解函数调用关系。因此，大多数调试器都提供了显示栈回溯信息的功能，比如 WinDBG 提供了以 k 开头的一系列命令 (k, kb, kd, kp, kP, kv) 来显示各种格式的栈回溯信息（30.14 节将详细介绍）。清单 22-13 给出了使用 WinDBG 的 kv 命令显示的栈回溯信息。

清单 22-13 WinDBG 显示的栈回溯信息

```
0:000> kv
ChildEBP RetAddr Args to Child
0012ff74 004010df 00408054 0012ffc0 004012d2 LocalVar!FuncC+0x9
0012ff80 004012d2 00000001 003d0ec0 003d0f28 LocalVar!main+0x1f
0012ffc0 7c816d4f 00090000 08ebfa9c 7ffd7000 LocalVar!mainCRTStartup+0xb4
0012fff0 00000000 0040121e 00000000 78746341 kernel32!BaseProcessStart+0x23 (FPO:
[Non-Fpo])
```

比较以上两个清单，我们看到其结果几乎是一样的，只是 WinDBG 显示的最后一行多了(FPO: [Non-Fpo])信息，下一节我们将介绍 FPO 的含义。

22.5 帧指针省略 (FPO)

上一节我们介绍了栈帧的概念和用于标志栈帧位置的帧指针。帧指针不仅对函数中的代码起到定位变量和参数的参照物作用，而且将栈中的一个个栈帧串联在一起，形成一个可以遍历所有栈帧的链条。但是并不是所有函数都会建立帧指针，某些优化过的函数省去了建立和维护帧指针所需的指令，所以这些函数所对应的栈帧不再有帧指针，这种情况被称为帧指针省略 (Frame Pointer Omission)，简称 FPO。

清单 22-9 所示的 FuncB 函数便使用了 FPO，从其反汇编代码中我们找不到建立和恢复帧指针的指令。从理论上讲，使用 FPO 可以省略一些指令，减小目标文件，提高运行速度，因此 FPO 成为速度优化和空间优化的一种方法。VC6 编译器在编译发布版本时默认会开启 FPO 选项。

在使用 FPO 的函数中，因为没有固定位置的帧指针可以参考，所以必须使用其他参照物来引用局部变量，如我们前面所讨论的，在 x86 系统中，通常是使用 ESP 寄存器。因为 ESP 寄存器的值是经常变化的，所以对同一个局部变量的引用所需的偏移值也是变化的，这给跟踪这样的代码增加了难度。使用 FPO 的另一个副作用就是对于采用 FPO 优化的函数，因为没有了帧指针，所以给生成栈回溯信息带来了不便。

当执行使用 FPO 优化过的函数时，EBP 寄存器指向的仍然是前一个栈帧。这使得我们很难为这样的函数产生帧信息。为了解决这一问题，编译器在生成调试符号时，会为使用 FPO 的函数产生 FPO 信息。利用调试符号中的 FPO 信息，调试器可以为省略帧指针的函数生成回溯信息。

我们先来做个试验，在 WinDBG 中加载发布版本的 LocalVar 程序，使用 `bp LocalVar!FuncB+0x7` 设置一个断点，然后执行到这个位置，发出 `kv` 命令看 WinDBG 产生的栈回溯信息：

```
0:000> kv
ChildEBP RetAddr  Args to Child
0012ff74 004010d2 00408054 0012ffc0 004012d2 LocalVar!FuncB+0xc (FPO: [1,2,1])
0012ff80 004012d2 00000001 00371280 003712d8 LocalVar!main+0x12
0012ffc0 7c816ff7 0169f6f2 0169f76c 7ffdde000 LocalVar!mainCRTStartup+0xb4
0012fff0 00000000 0040121e 00000000 78746341 kernel32!BaseProcessStart+0x23 ...
```

我们看到，WinDBG 产生了非常好的栈回溯信息，也为经过 FPO 优化的 FuncB 函数产生了完整的帧信息，观察此时的 EBP 和 ESP 寄存器：

```
0:000> r ebp, esp
ebp=0012ff80 esp=0012ff70
```

可以看到 EBP 寄存器指向的仍然是 main 函数的栈帧。那么 WinDBG 是如何生成 FuncB 函数的栈帧信息的呢？答案是依靠调试符号中的附加信息。我们先来看一下 FuncB 函数那一行的末尾括号中的信息（FPO: [1,2,1]）是什么含义。FPO 代表对应的函数采用了 FPO 优化，方括号中的三个数字的含义分别是：

- 1：FuncB 函数有一个参数；
- 2：为局部变量分配的栈空间是 2 个 DWORD 长（即 8 个字节）；
- 1：使用栈保存的寄存器个数为 1，即有一个寄存器（第 5 行压入的 EDI）被压入栈中保存。

以上信息来源于符号文件中的帧数据（FrameData）表，下面我们介绍一下 WinDBG 是如何利用这些信息来产生 FuncB 的栈帧记录的。仍然是根据 EIP 寄存器的值寻找最靠近的函数符号，也就是找到 FuncB。在 FuncB 的符号信息中有一个字段是这个函数的 RVA，也就是这个函数的入口相对于模块起始地址的偏移，其值为 0x1040。根据 FuncB 的 RVA，WinDBG 可以在符号文件中搜索到这个函数的 FPO 信息，在符号文件中 FPO 信息是按照它所描述函数的 RVA 来组织的。根据 FPO 信息，WinDBG 可以知道函数的参数长度（0x4）、局部变量的长度（0x8）、代码块的长度（0x62）等信息。根据这些信息和当前的程序指针位置以及栈指针值，结合反汇编，调试器可以推算出当前函数的帧指针值。比如对于我们目前分析的执行点，CPU 执行到 FuncB 的偏移 7 字节处，根据反汇编分析 WinDBG 可以知道已经执行了局部变量分配操作，因此 ESP 的值加上局部变量的长度便是当前栈帧的边界，即 `ESP+8=0012ff78`。根据惯例帧指针指向的应该是当前栈帧的第一个 DWORD，因此应该把这个值减去 4，即 `0x0012ff74` 是当前函数的帧指针（ChildEBP）。得到帧指针后，便可以像处理普通帧那样显示参数和返回值了。

为了证明调试符号对产生栈回溯信息的重要性，停止调试并将 LocalVar.PDB 文件改名为 LocalVar.BAK，然后再重复以上过程（可以直接使用地址来设置断点 `bp 0040104b`）。这时 `kv` 命令显示的信息如下：

```
0:000> kv
ChildEBP RetAddr Args to Child
WARNING: Stack unwind information not available. Following frames may be wrong.
0012ff80 004012d2 00000001 00371280 003712d8 LocalVar+0x104b
0012ffc0 7c816ff7 0169f6f2 0169f76c 7ffdde000 LocalVar+0x12d2
0012fff0 00000000 0040121e 00000000 78746341 kernel32!BaseProcessStart+0x23 ...
```

首先 WinDBG 显示了一行警告信息，告诉我们下面的信息可能是错误的。另外，这时只显示了三个栈帧的信息，而不是刚才的 4 个。也就是说失去了调试符号的帮助后，调试器已经没有办法再为省略了帧指针（FPO）的函数生成栈帧信息。

接下来对 FuncC 函数设置一个断点（bp 00401087）并执行至此，然后使用 kv 命令显示栈回溯信息：

```
0:000> kv
ChildEBP RetAddr Args to Child
WARNING: Stack unwind information not available. Following frames may be wrong.
0012ff74 004010df 00408054 0012ffc0 004012d2 LocalVar+0x1087
0012ff80 004012d2 00000001 00371280 003712d8 LocalVar+0x10df
0012ffc0 7c816ff7 0169f6f2 0169f76c 7ffdde000 LocalVar+0x12d2
0012fff0 00000000 0040121e 00000000 78746341 kernel32!BaseProcessStart+0x23...
```

尽管仍然有警告信息，但是此时正确显示出了四个栈帧的情况。可见，同样是没有调试符号的情况，对于采用 FPO 的 FuncB 函数，调试器无法显示出它的帧信息；而对于未采用 FPO 的 FuncC 函数调试器仍然可以显示出帧信息。也就是说没有采用 FPO 优化的函数具有更好的可调试性，因此，编译器在编译调试版本时通常会禁止包括 FPO 在内的所有优化选项。

最后要说明的是，处理 FPO 对于调试器来说是一件比较复杂的操作，有时也可能出现错误。例如，如果当前位置在 FuncB 的入口处，那么 WinDBG 会将 FuncB 函数显示到 main 函数所在的栈帧，成为清单 22-14 所示的样子。

清单 22-14 WinDBG 产生的包含错误的栈回溯

```
0:000> kv
ChildEBP RetAddr Args to Child
0012ff80 004012d2 00000001 00371280 003712d8 LocalVar!FuncB (FPO: [1,2,1])
0012ffc0 7c816ff7 0169f6f2 0169f76c 7ffdde000 LocalVar!mainCRTStartup+0xb4
0012fff0 00000000 0040121e 00000000 78746341 kernel32!BaseProcessStart+0x23 ...
```

总的来说，FPO 是不利于调试的，因此 Windows Vista 的很多系统模块在编译时禁止了 FPO 选项。

22.6 栈指针检查

栈的有序性是依靠每个函数都遵守栈的使用规则，保证函数返回时栈指针（ESP）的值与进入函数时一致，即保持栈平衡。否则的话，栈数据就可能错位甚至面目全非。

下面通过一个例子来说明，在 CheckEsp 程序（完整源代码位于 \code\chap22\CheckESP 目录）中我们编写了一个 BadEsp 函数，通过嵌入式汇编向栈中压入了 4 个字节的内容，但是却没有对应的弹出操作。

```
void BadEsp()
{
    _asm push eax;
}
```

当执行到这个函数时，栈顶存放的是返回地址，执行 PUSH EAX 指令后，栈顶存放的内容变为 EAX 寄存器的值。当执行 RET 指令时，因为 RET 指令总是返回到栈顶所存放的地址处，所以执行 RET 后程序会跳转到 EAX 寄存器所包含的值。以下是笔者跟踪执行 RET 指令后的结果：

```
0:000> p
eax=0000000d ebx=7ffdff000 ecx=00408070 edx=7c90eb94 esi=00000000 edi=00000000
0000000d esp=0012ff80 ebp=0012ffc0 iopl=0 nv up ei pl nz na po nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000
0000000d ?? ??
```

可见程序指针指向了 0000000d 的位置，这里根本不是有效的代码区。“??”代表 WinDBG 无法显示这个地址的内容。

为了及时发现以上问题，编译器设计了专门的栈指针检查函数来帮助发现问题。我们先来介绍 VC6 的做法，然后再推广到 VC8。

当编译调试版本时，VC6 会自动在每个函数的末尾插入指令来调用一个名为 `_chkesp` 的函数。在清单 22-15 所示的 BadEsp 函数的调试版本反汇编代码中我们可以看到这个调用。

清单 22-15 调试版本的 BadEsp 函数的反汇编代码

```

1   6:          void BadEsp()
2   7:          {
3   0040D630      push    ebp ; 保存 EBP 寄存器的本来值
4   0040D631      mov     ebp,esp ; 将进入函数时的栈指针值保存到 EBP 寄存器
5   0040D633      sub     esp,40h ; 调试版本为函数分配的默认局部变量，64 字节长
6   0040D636      push    ebx ; 保存 EBX 寄存器的本来值
7   0040D637      push    esi ; 保存 ESI 寄存器的本来值
8   0040D638      push    edi ; 保存 EDI 寄存器的本来值
9   0040D639      lea     edi,[ebp-40h] ; 取局部变量的起始地址
10  0040D63C      mov     ecx,10h ; 设置循环次数
11  0040D641      mov     eax,0CCCCCCCCCh ; CC 即 INT 3 指令的机器码
12  0040D646      rep     stos dword ptr [edi] ; 循环，将缓冲区初始化为 CC
13  8:          _asm push eax; ; 源代码中的嵌入式汇编语句
14  0040D648      push    eax ; 编译后的嵌入式汇编语句
15  9:          }
16  0040D649      pop     edi ; 弹出 EDI，与第 8 行对应
17  0040D64A      pop     esi ; 弹出 ESI，与第 7 行对应
18  0040D64B      pop     ebx ; 弹出 EBX，与第 6 行对应
19  0040D64C      add     esp,40h ; 释放分配的局部变量，与第 5 行相对应
20  0040D64F      cmp     ebp,esp ; 比较 EBP 和 ESP
21  0040D651      call    __chkesp (0040d6a0) ; 调用栈指针检查函数
22  0040D656      mov     esp,ebp ; 将 EBP 寄存器的值赋给 ESP，与第 4 行相对应
23  0040D658      pop     ebp ; 弹出保存的 EBP，与第 3 行相对应
24  0040D659      ret     ; 返回

```

我们看到，编译器为只包含一条汇编指令的 BadEsp 函数生成了 20 多条指令。这

是因为在调试版本中，编译器插入了很多指令来帮助检查错误和支持调试。比如，尽管我们没有定义局部变量，但是编译器仍会分配一段空间，并使用0xCC（即INT 3指令）来填充，以便万一CPU意外执行到该区域时，可以中断到调试器。

观察上面的代码，我们可以看出函数入口和出口附近的代码具有着非常好的对应性，这些相互呼应的操作是为了使栈保持平衡。但因为我们插入了一个额外的不对称的PUSH操作，所以可以想见第16到18行的弹出操作都错位了，EDI会被恢复成EAX的值，ESI会被恢复为本来EDI的值……而且第19行执行后的ESP值也会比预期的小4。本来从第5到19行的所有栈操作相互抵消，ESP值应该保持不变，也就是与保存的EBP值相等。但是现在二者不等了，这会导致_chkesp函数弹出图22-4所示的错误对话框。

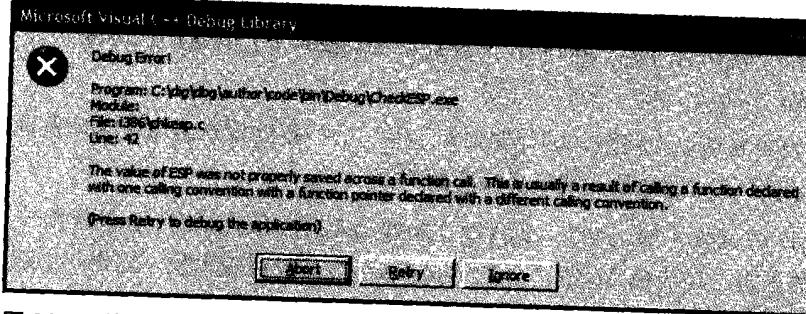


图22-4 栈指针检查函数(_chkesp)报告的错误对话框

_chkesp是C运行时库(CRT)中的一个函数，用来检查栈指针的完好性。检查方法是比较ESP和EBP寄存器的值(第20行)，看其是否相等，如果相等则通过，否则就准备参数调用_CrtDbgReport函数报告错误。

```
_chkesp:
004010B0 jne      _chkesp+3 (004010b3) ; 检查比较结果
004010B2 ret      ; 相等则返回
...
004010D1 call     _CrtDbgReport (00401410) ; 不相等则准备参数
                                                ; 报告错误
```

因为_chkesp函数需要借助建立栈帧时保存的EBP寄存器值，所以不适用于使用FPO优化的函数。编译器的编译选项/GZ用来控制是否插入栈指针检查函数。该选项被包含在调试版本的默认设置中，但不在发布版本中，因为/GZ选项与发布版本中的优化选项(/O)是矛盾的。

VC8的_RTC_CheckEsp函数与_chkesp原理和工作方法是一样的，只是改变了函数名称和报告错误的方式。例如以下是调用_RTC_CheckEsp的一个例子：

```
004010E7 cmp      ebp, esp
004010E9 call     _RTC_CheckEsp (401510h)
```

_RTC_CheckEsp的实现也与以前非常类似：

```
_RTC_CheckEsp:
00401510 jne      esperror (401513h) ; 检查比较结果
00401512 ret      ; 相等则返回
```

```

esperror:
...
00401524 call      _RTC_Failure (402E60h) ;报告错误

```

也就是检查比较结构，如果不等则转到 esperror 处准备参数调用_RTC_Failure 报告 RTC 错误。那么为什么没有把比较 ESP 和 EBP 那条指令放在_chkesp 和 _RTC_CheckEsp 内部呢？原因是函数调用会影响 ESP 寄存器的值。

22.7 调用协定

早在 20 世纪 40 年代，计算机科学家们就开始将一些具有通用性的代码整理出来，以便可以很方便地重复使用这些代码。他们将这些能完成一定功能而且相对独立的代码片断称为子过程(subroutine)。在名为《数字计算机自动编码》的论文中，Grace Hopper 提到了使用子过程所带来的好处：复用编写好而且已经测试过的代码。这也许是为实现软件复用这一目标所作的最早努力之一。尽管今天已经出现了 COM/DCOM、Web Service 等模块一级或更高层次的软件复用技术，但是函数和过程一级的代码复用仍是最基本而且最重要的。

要复用现有函数和过程的一个基本问题就是如何调用这些函数，包括如何传递参数，如何接收计算结果，如何维持程序的上下文状态(context)和清理栈，等等。考虑到要调用的函数可能是不同人使用不同语言和不同编译器开发的，所以这个看似简单的问题并不简单。通过上一节对栈的介绍，我们知道参数传递或栈操作中的任何误差都可能导致严重的错误。

要解决以上问题，在发起调用的一方和被调用函数之间必须建立一种契约，这便是函数调用协定(Calling Convention)。通常在设计一个函数时，将调用协定写在函数声明中，其一般格式为：

```
return-type [调用协定] function-name[(argument-list)]
```

例如：BOOL WINAPI IsDebuggerPresent(VOID);

其中 BOOL 是返回值类型，WINAPI 是调用协定名称，如果没有指定调用协定，那么编译器会根据语言环境使用默认的调用协定。

在长期的软件开发实践中，人们设计并归纳出了很多种函数调用协定以适应不同的需求。其中，在 x86 系统中应用比较广泛有以下几种：C 调用协定、标准调用协定、快速调用协定和 C++ 成员函数使用的 this 调用协定。对于非 x86 系统(如 Itanium、PowerPC 等)，调用协定通常比较单一。本节将分别讨论 x86 和 x64 系统所使用的各种调用协定。

22.7.1 C 调用协定

C 调用协定是 C/C++ 程序的普通函数所使用的默认调用协定，完全使用栈来传递

参数，函数调用者（caller）在发起调用前将参数按从右到左的顺序依次压入到栈中，并且负责在函数返回后调整栈指针清除压入栈的参数。从右到左压入参数的好处是对于被调用函数来说第一个参数相对于栈顶的偏移总是固定的，而且是调用者负责清理参数，所以 C 调用协定支持可变数目的参数。例如，常用的 printf 函数使用的就是 C 调用协定，可以使用不同个数的参数来调用这个函数。

```
int printf(const char* format [, argument]... );
```

C 调用协定所使用的关键字是 `_cdecl`。

22.7.2 标准调用协定

标准调用协定与 C 调用协定很类似，也是用栈来传递参数，传递顺序也是从右到左。但不同的是，标准调用协定规定被调用函数清理栈中的参数。对于被频繁调用的函数，这样做好处是可以减小目标程序的大小，因为清理栈的指令不必反复出现在调用函数中。标准调用协定使用的关键字是 `_stdcall`。

大多数 Windows 系统函数和 API 使用的都是标准调用协定，只不过通常是使用 `_stdcall` 关键字的别名。搜索 Windows SDK 的头文件 `windef.h`，我们很容易发现，常见的 WINAPI 和 CALLBACK 关键字其实就是 `_stdcall`。

```
#define CALLBACK      __stdcall
#define WINAPI       __stdcall
#define WINAPIV        __cdecl
#define APIENTRY      WINAPI
#define APIPRIVATE    __stdcall
#define PASCAL         __stdcall
```

大多数 Windows API 和回调函数使用的都是标准调用协议。所有使用标准调用协定的函数必须有函数原型。

22.7.3 快速调用协定

因为 CPU 访问寄存器的速度要比访问内存快很多，所以使用寄存器传递参数比使用栈速度更快，因此很多编译器都设计了使用寄存器传递参数的调用协定。在 x86 平台上，比较典型的便是所谓的快速调用（fastcall）协定。对于 32 位程序，快速调用协定使用 ECX 和 EDX 来传递前两个（左起）长度不超过 32 位的参数，其他参数使用栈来传递（从右到左）。举例来说，如果第一个参数（左起）是 `_int64`，第二个参数是 `int`，第三个参数是 `char`，那么第一个参数会使用栈传递，第二个参数会用 ECX 传递，第三个参数会用 EDX（DL）来传递。类似的，对于 16 位程序，使用 CX 和 DX 传递前两个不超过 16 位的参数。快速调用协定的关键字是 `_fastcall`。举例来说，以下是一个声明使用快速调用协定的函数 `FastCallFunc`，它共有 4 个参数，分别是浮点类型、整数类型、类和整数类型。

```
int __fastcall FastCallFunc(float f, int a, Cat c, int b)
{
```

```

printf("FastCallFunc:%f,%d,%s,%d", f, a, c.Name(), b);
return 30;
}

```

清单 22-16 给出了调用这个函数的源代码和反汇编代码。

清单 22-16 调用使用快速调用协定的函数

51:	FastCallFunc(1.5,10,cat,20);	
004012A6	mov	edx,14h ; 使用 EDX 寄存器来传递常数 20
004012AB	sub	esp,14h ; 为参数 cat 分配空间
004012AE	mov	ecx,5 ; 设置复制 cat 要循环的次数
004012B3	lea	esi,[ebp-14h] ; 将复制操作的源指向 cat 对象
004012B6	mov	edi,esp ; 将复制目标指向栈 (刚刚分配的参数 cat)
004012B8	rep movs	dword ptr [edi],dword ptr [esi] ; 开始复制
004012BA	mov	ecx,0Ah ; 使用 ECX 来传递参数 10
004012BF	push	3FC00000h ; 使用栈来传递浮点参数 1.5
004012C4	call	@ILT+25(FastCallFunc) (0040101e) ;

因为参数 f 是浮点类型，所以使用了栈来传递；参数 a 是第一个适合使用寄存器传递的参数，所以使用 ECX 来传递；参数 cat 是 C++ 类的实例，是被复制到栈上传递的；参数 d 是第二个适合使用寄存器传递的参数，所以使用 EDX 来传递。

在调试时，kv 这样的栈回溯命令只显示放在栈上的参数，不会显示使用寄存器传递的参数，因此使用快速调用协定是不利于调试的。Windows 系统 IO 管理器的某些函数使用了快速调用协定，如 NTSTATUS FASTCALL IofCallDriver(IN PDEVICE_OBJECT DeviceObject, IN OUT PIRP Irp)。

22.7.4 This 调用协定

C++ 程序中的成员函数（类方法）默认使用的调用协定是 this 调用协定。这种调用协定的最重要特征就是 this 指针会被放入 ECX 寄存器（Borland 编译器使用 EAX）传递给被调用的方法。This 调用协定也是要求被调用函数负责清理栈，因此不支持可变数量的参数。当我们在 C++ 类中定义了可变数量参数的成员函数时，编译器（VC）会自动改为使用 C 调用协定。当调用这样的方法时，编译器会将所有参数压入栈之后，再将 this 指针压入栈。举例来说，下面的 Cat 类包含了一个可变数目参数的方法 ChooseFood：

```

enum MEAL {BREAKFAST, LUNCH, SUPPER};
class Cat
{
public:
    char* ChooseFood(MEAL i, ...);
};

```

清单 22-17 给出了调用以上方法的一个实例和对应的汇编代码。

清单 22-17 调用可变数量参数的类方法

```

Cat cat;
printf("Cat choose food %s.",cat.ChooseFood(BREAKFAST,"meat","beaf","rice"));
// 以下是对应的汇编指令

```

00401011 6854804000	push	0x408054	; 压入指向字符串"rice"的指针
00401016 684c804000	push	0x40804c	; 压入指向字符串"beaf"的指针
0040101b 6844804000	push	0x408044	; 压入指向字符串"meat"的指针
00401020 8d44240c	lea	eax, [esp+0xc]	; 取 cat 实例的地址
00401024 6a00	push	0x0	; 压入枚举常量 BREAKFAST
00401026 50	push	eax	; 压入 cat 实例的地址, 即 this 指针
00401027 e8d4fffff	call	CallConv!Cat::ChooseFood (00401000)	; 发起调用
0040102c 83c414	add	esp, 0x14	; 清理栈, 前面共压入 20 个字节
0040102f 50	push	eax	; 将刚才的返回值压入栈
00401030 6830804000	push	0x408030	; 压入字符串常量
00401035 e806000000	call	CallConv!printf (00401040)	; 发起调用 printf 函数

当执行到 ChooseFood 方法中时, 使用 kv 命令显示栈回溯信息, 其结果如下:

```
0:000> kv
ChildEBP RetAddr Args to Child
0012ff64 0040102c 0012ff80 00000000 00408044 CallConv!Cat::ChooseFood+4 ...
0012ff80 00401125 00000001 00370ec0 00370f28 CallConv!main+0x1c (FPO: [0,1,0])
...
```

其中 0012ff80 是 this 指针的值, 其后的 00000000 是枚举常量 BREAKFAST, 00408044 是字符串常量 “meat” 的地址。

VC8 以前的 VC 编译器没有为 this 调用协定定义关键字。因此程序员不能显式地指定 this 调用协定 (通常也没有这个必要)。VC8 加入了 __thiscall 作为 this 调用协定的关键字, 原因是托管的 C++ 类默认使用新定义的 CLR 调用协定, 如果要改为使用传统的 this 调用协定, 那么就需要显式在声明中加上 __thiscall 关键字。

22.7.5 CLR 调用协定

在 VC8 以前, 编译器会为每个托管函数生成两个入口点, 一个是托管 (managed) 入口点, 另外一个是本地 (native) 入口点。这样当托管代码使用本地入口点调用该函数时 (虚拟函数必须总使用本地入口点调用), 本地入口点需要将调用转给托管入口点, 这样便会造成不必要的托管/非托管上下文切换和参数/返回值的复制, 这种现象被称为双重转换 (double thunking)。因为双重转换会导致性能损失, 所以 Visual Studio 2005 引入了一个新的调用约定关键字, 如果一个托管函数不会被非托管代码使用指针调用, 那么可以在声明此函数时用新增的 __clrcall 修饰符阻止编译器生成两个入口。

因为使用传统的 DLL 输出方法 (dllexport) 输出一个托管函数也会导致编译器为其产生两个入口点, 而且任何通过 DLL 方法对该函数的调用都会使用本地入口点 (再转给托管入口点)。为了防止这种情况的双重转换, SDK 的文档建议使用 .Net 模块引用方法, 不要用传统的 DLL 方法。

22.7.6 X64 调用协定

X64 系统通常只使用一种派生自快速调用的调用协定, 使用寄存器传递前四个参

数，其他参数使用栈来传递。RCX、RDX、R8、R9 四个 64 位的寄存器用于 64 位或短于 64 位的整数类型和指针类型，XMM0~XMM3 用于浮点类型。结构类型会被自动当作指针类型（引用）进行传递。比如对于如下函数：

```
int Func64(__m64 a, __m128 b, struct c, float d);
```

当调用这个函数时，参数 a 会被放入 RCX 寄存器，指向 b 的指针会被放入 RDX 寄存器，指向结构 c 的指针会被放入 R8 寄存器，参数 d 会被放入 XMM3 寄存器。

须要指出的是，64 位的快速调用固定使用 ECX 或 XMM0 来传递第一个参数，EDX 或 XMM1 传递第二个参数，依次类推，最多可能有 4 个寄存器用来传递（前四个）参数。这与 32 位的快速调用有所不同，在 32 位的快速调用中，ECX 和 EDX 被用来传递前两个适合使用它们传递的参数，所以它们实际可能传递第 3 个或第 5 个参数（参见前面的 FastCallFunc 函数示例）。

与 32 位的快速调用协定的另一个重要差异是，x64 调用协定规定调用者（而不是被调用函数）来清理栈，这使得 x64 调用协定也可以支持可变数目的参数。

22.7.7 通过编译器开关改变默认调用协定

对于函数声明中没有指定调用协定的函数，编译器会根据情况使用合适的默认调用协定。可以通过编译器选项来定义默认的调用协定。以 VC 编译器为例，它支持如下三个选项：

/Gd，默认设置，使用 C 调用协定（`_cdecl`）作为默认调用协定。

/Gr，使用快速调用协定（`_fastcall`）作为默认调用协定。

/Gz（注意 z 小写），使用标准调用协定（`_stdcall`）作为默认调用协定。

以上选项仅适用于目标平台是 x86 的情况，因为其他平台通常都只使用一种调用协定，无需设置。

22.7.8 函数返回值

如何传递函数的返回值也是调用协定要定义的一个重要内容。要说明的是，我们这里说的返回值就是指使用 `return` 语句或其他等价方式显式返回的计算结果，不包括通过全局变量或参数指针来传递计算结果的情况。

传递返回值的方法远不像传递参数那样有很多种。大多数编译器通常都使用统一的一种方法。因此在以上介绍每个调用协定时，我们没有一一介绍传递返回值的方法，目的是留在这里统一介绍。

通常编译器会根据以下原则来选择传递返回值的方法（以 x86 平台的 32 位编译器为例）。

- 如果返回值是 EAX 寄存器能够容纳的整数、字符或指针（4 字节或少于 4 字节），那么使用 EAX 寄存器。
- 如果返回值是超过 4 字节但是少于 8 字节的整数，那么使用 EDX 寄存器来存放四字节以上的值。
- 如果要返回一个结构或类，那么分配一个临时变量作为隐含的参数传递给被调用函数，被调用函数将返回值复制到这个隐含参数之中，并且将其地址赋给 EAX 寄存器。
- 浮点类型的返回值和参数通常是通过专门的浮点指令使用栈来传递的，其细节从略。

下面通过一个例子来说明返回结构或类的情况。我们首先将刚才使用的 Cat 类扩充成如下形式：

```
#define NAME_LENGTH 20
class Cat
{
    char m_szName[NAME_LENGTH];
public:
    Cat(){m_szName[0]=0;}
    Cat(char* sz);
    char* ChooseFood(MEAL i, ...);
    Cat GetChild(int n);
    char * Name(){return m_szName;}
};
```

其中的 GetChild 方法的实现如下：

```
Cat Cat::GetChild(int n)
{
    Cat c("Caty");
    return c;
}
```

也就是这个方法返回一个 Cat 类的实例，乍一看，有些读者可能会说这样编写的方法直接返回定义在栈上的局部变量有明显的错误。其实不然，看了下面的解释大家就会明白其实返回的并非局部变量本身。

接下来我们在 main 函数中定义两个实例，并调用 GetChild 方法：

```
Cat cat("Looi"),cat2("");
cat2=cat.GetChild(2);
```

上面的第二条语句对应的反汇编指令如下：

```
45:      cat2=cat.GetChild(2);
004011BD  push      2          ; 压入参数 2
004011BF  lea       ecx, [ebp-3Ch]   ; 取用作隐含参数的局部变量的地址
004011C2  push      ecx          ; 将这个隐含参数压入栈
004011C3  lea       ecx, [ebp-14h]   ; 取 cat 变量的地址，传递 this 指针
004011C6  call     @ILT+35(Cat::GetChild) (00401028) ; 调用 GetChild 函数
004011CB  mov       esi, eax        ; 将返回值（即隐含参数指针）赋给 ESI
004011CD  mov       ecx, 5          ; 设置循环次数，Cat 类的大小是 20 字节
004011D2  lea       edi, [ebp-28h]   ; 将 cat2 的地址赋给 EDI
004011D5  rep      movs dword ptr [edi],dword ptr [esi] ; 开始复制
```

也就是说，编译器会定义一个额外的临时对象，并将它传递给函数，当函数返回时，EAX 寄存器指向的是这个临时对象。

从 WinDBG 显示的栈回溯信息中我们也可以看到隐含参数：

```
0:000> kb
ChildEBP RetAddr  Args to Child
0012ff30 004010bd 0012ff70 00000002 ffffffff CallConv!Cat::GetChild+0x3
0012ff80 004012d3 00000001 00370ec0 00370f28 CallConv!main+0x5d
```

其中 0012ff70 便是指向临时对象的指针，下面再看看 GetChild 方法内部的 return 语句所对应的操作：

```
38:      return c;
0040112A  mov      ecx,5          ; 设置循环次数, Cat 类的大小是 20 字节
0040112F  lea      esi,[ebp-18h]   ; 将局部变量 c 的地址赋给 ESI
00401132  mov      edi,dword ptr [ebp+8] ; 将参数 1(隐含参数) 的地址赋给 EDI
00401135  rep      movs    dword ptr [edi],dword ptr [esi]; 开始复制
00401137  mov      eax,dword ptr [ebp+8] ; 将参数 1(隐含参数) 的地址赋给 EAX
```

从以上汇编代码可以明显看出，尽管源代码中的 return c 很容易让人误认为是直接返回局部变量 c，但事实上，编译器在编译时会自动生成代码将局部变量 c 复制给传入的隐含参数，实际上返回的是指向隐含参数的指针。因此从函数调用的角度来看 Cat Cat::GetChild(int n) 函数是被作为 Cat* Cat::GetChild(Cat *, int n) 的形式来编译的。

在上面的调用过程中，共发生了两次对象复制操作，一次是 GetChild 函数中将局部变量 c 赋给隐含参数，另一次是隐含参数赋给 main 函数中的局部变量 cat2。所以，尽管这样的设计方法是 C++ 语言和编译器所支持的，但是因为须要定义额外的局部变量和承担对象复制所需的开销，所以其效率是比较低的。一种改进的方法是将 cat2 对象以指针或引用的形式直接传递给被调用函数。

22.7.9 归纳和补充

前面我们介绍了有关调用协定的重要内容，除了以上内容，还有以下几点值得说明。

- 对于 32 位程序，所有长度小于 32 位的参数会被自动加宽到 32 位，换句话来说，每个参数所占的空间至少是 32 位，不论是在栈中，还是寄存器中。类似的，在 16 位程序和 64 位程序中，长度小于 16 位或 64 位的参数也会被加宽为 16 位或 64 位。
- 在函数调用过程中，被调用函数必须保证某些寄存器的内容在函数返回时和进入函数时是一样的，这些寄存器被称为 nonvolatile 寄存器，如果一个函数要使用 nonvolatile 寄存器，那么它应该先将寄存器的本来内容保存起来，待函数返回时，再将这些寄存器的值恢复成原来的值。在 32 位的 x86 程序中，ESI、EDI、EBX 和 EBP 寄存器是 nonvolatile 寄存器，在 x64 中，RDI、RSI、RBX、RBP、R12、R13、R14 和 R15 寄存器是 nonvolatile 寄存器。

表 22-1 归纳了 Windows 系统（微软编译器）下常用的调用协定，以方便查阅。

表 22-1 调用协定一览

代码段	调用协定	用于传递参数的寄存器	参数入栈顺序	调用者
16 位	cdcall	无	从右向左	调用者
	pascal	无	从左向右	被调用者
	fastcall	AX、DX、BX	从右向左	被调用者
32 位	cdcall	无	从右向左	调用者
	stdcall	无	从右向左	被调用者
	fastcall	EAX、EDX	从右向左	被调用者
	thiscall	ECX (this 指针)	从右向左	被调用者
64 位	-	rcx/xmm0、rdx/xmm1、r8/xmm2、r9/xmm3	从右向左	调用者

本节的内容对调试，特别是理解栈回溯信息是很有帮助的。在 `kv` 或 `kb` 等栈回溯命令的结果中，返回地址右侧的三列被称为 `Args to child`，通常解释为函数的前三个参数。这种解释其实是不准确的，以下面的调用 `FastCallFunc` 函数栈回溯信息为例：

```
0:000> kb
ChildEBP RetAddr  Args to Child
0012fed8 004012c9 3fc00000 696f6f4c 00000000 CallConv!FastCallFunc+0x20
0012ff80 00401669 00000001 00370f20 00370fa8 CallConv!main+0xb9
...
```

`3fc00000` 确实是第一个参数（浮点数 1.5），但是 `696f6f4c` 和 `00000000` 却不是第二个和第三个参数，它们是第三个参数 `cat` 对象的前 8 个字节。而真正的第二个和第四个参数分是在 `ECX` 和 `EDX` 寄存器中的。使用 `r` 命令可以看到它们：

```
0:000> r ecx,edx
ecx=0000000a edx=00000014
```

因此准确的说法是，`kv` 或 `kb` 命令显示的是栈帧中参数区域的前三个 `DWORD`，这三个 `DWORD` 的确切含义要视函数所采用的调用协定而定。因为大多数的 Windows 系统函数使用的都是标准调用，所以很多时候这前三个 `DWORD` 确实对应于前三个参数。

22.8 栈空间的增长和溢出

栈为函数调用和定义局部变量提供了一块简单易用的内存空间，定义在栈上的变量根本不需要考虑内存申请和释放这样的麻烦事，因为编译器会生成代码来完成这些任务。从栈上分配空间意味着栈指针下移（向更低地址移动），腾出更多空间；释放空间意味着栈指针移回到原来的位置（图 22-5）。也就是说，只要调整栈指针就可以分配和释放栈中的空间，这比从堆上分配空间要高效得多。

那么栈空间到底有多大呢？是不是取之不尽用之不竭呢？栈空间被用完了会怎样呢？本节我们就来回答这些问题。

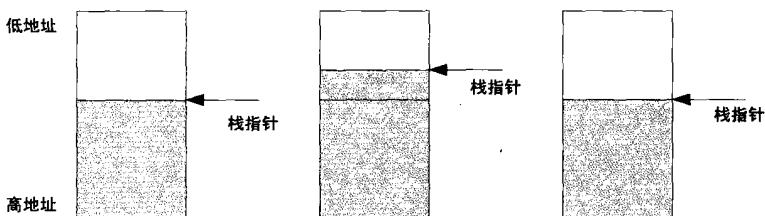


图 22-5 从栈上分配空间（中）和释放空间（右）

22.8.1 栈空间的自动增长

VC 编译器在构建一个程序（x86）时，为栈设置的默认参数是保留 1MB，初始提交 4KB，这意味着系统会为这个进程的初始线程创建一个 1MB 大小的栈，并先提交其中的一小部分（8KB，其中 4KB 为保护页）供程序使用。提交一小部分的目的是为了节约内存。但是，只有已经提交的内存才是可以访问的，访问保留的内存仍然会导致访问违例，提交的内存空间用完了怎么办呢？答案是触发栈增长机制来扩大提交区域。

第 22.2 节介绍栈创建过程时，我们提到过系统在提交栈空间时会故意多提交一个页面，我们称这个页面为栈保护页面（Stack Guard Page）。栈保护页面具有特殊的 PAGE_GUARD 属性，当具有如此属性的内存页被访问时，CPU 会产生页错误异常并开始执行系统的内存管理函数，在内存管理函数检测到 PAGE_GUARD 属性后，会先清除对应页面的 PAGE_GUARD 属性，然后调用一个名为 MiCheckForUserStackOverflow 的系统函数，这个函数会从当前线程的 TEB 中读取用户态栈的基本信息并检查导致异常的地址，如果导致异常的被访问地址不属于栈空间范围，则返回 STATUS_GUARD_PAGE_VIOLATION；否则，MiCheckForUserStackOverflow 函数会计算栈中是否还有足够的剩余保留空间可以创建一个新的栈保护页面。如果有，则调用 ZwAllocateVirtualMemory 从保留空间中再提交一个具有 PAGE_GUARD 属性的内存页。新的栈保护页与原来的紧邻。经过这样的操作后，栈的保护页向低地址方向平移了一位，栈的可用空间增大了一个页面的大小，这便是所谓的栈空间自动增长（参见图 22-1 右）。

22.8.2 栈溢出

当提交的栈空间再被用完，栈保护页又被访问时，系统便会重复以上过程，直到当栈保护页距离保留空间的最后一个页面只剩一个页面的空间时（图 22-6 左），MiCheckForUserStackOverflow 函数会提交倒数第二个页面，但不再设置 PAGE_GUARD 属性（图 22-6 右），因为最后一个页面永远保留不可访问，所以这时栈增长到它的最大极限。为了让应用程序知道栈即将用完，MiCheckForUserStackOverflow 函数会返回 STATUS_STACK_OVERFLOW，触发栈溢出异常。

栈溢出异常是个警告性的异常，在其发生后，栈的倒数第二个页面通常还没有真正用完，所以应用程序还可以继续运行，但当这个页面用完，访问到最后一个页面时，便会引发访问违例异常。

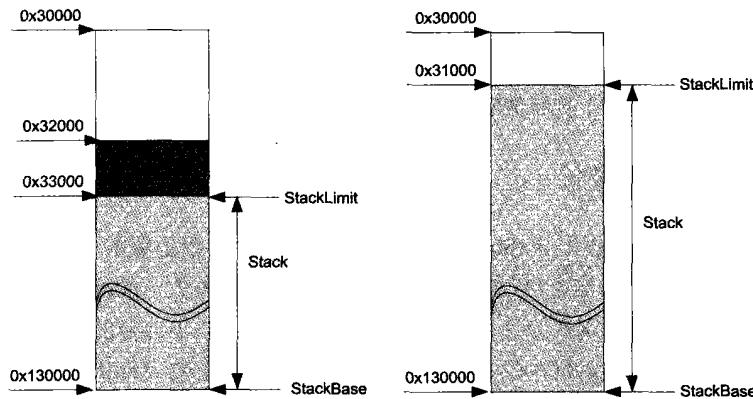


图 22-6 栈保护页可能出现的最后位置（左）和没有保护页的栈（右）

下面通过清单 22-18 所示的 StackOvr 小程序来演示栈的增长和溢出过程。

清单 22-18 演示栈溢出的 StackOvr 程序

```

1 #include "stdafx.h"
2 #include <windows.h>
3 int g_nCount=0;                                // 用来记录递归次数的全局变量
4 int g_ESP;                                     // 用来记录栈顶位置的全局变量
5 #pragma optimize("",off)                         // 关闭编译器的优化功能
6 void deadloop()
7 {
8     g_nCount++;                                 // 递增递归次数
9     char szMsg[512]="";                          // 定义一个局部变量，消耗栈空间
10    _asm mov g_ESP,esp;                          // 记录栈顶地址
11    deadloop();                                // 递归调用本函数
12 }
13 #pragma optimize("",on)                         // 开启编译器的优化功能
14 int main(int argc, char* argv[])
15 {
16     printf("Stack overflow example!\n");
17     deadloop();                                // 调用上面的死循环函数
18     return g_nCount;
19 }
```

为以上程序的发布版本生成调试符号后，在 WinDBG 中打开发布版本的 StackOvr.exe，使用 bp deadloop 命令设下一个断点，然后让程序执行。

当断点第一次命中时，观察 ESP 寄存器值和栈信息：

```

0:000> r esp
esp=0012ff80
0:000> !teb
  StackBase:      00130000
  StackLimit:     0012e000
```

此时栈的状态正是图 22-1 右图所示的情形，栈保护页位于栈的第三个页面，栈边界位于栈的第二个页面起始处。这说明栈已经增长过一次，也就是栈指针曾经指向过栈的第二个页面，尽管目前它指向栈的第一个页面，其原因是 main 函数之前的 CRT 启动函数曾经调用过使用栈较多的某些函数，随着这些函数的返回，栈指针又回落了。

使用 !address 命令可以观察不同栈区域的内存属性，例如观察当前栈顶的地址：

```
0:000> !address esp
00030000 : 0012e000 - 00002000
    Type      00020000 MEM_PRIVATE
    Protect   00000004 PAGE_READWRITE
    State     00001000 MEM_COMMIT
    Usage     RegionUsageStack
    Pid.Tid  cf0.158c
```

第 2 行的含义是这个地址属于 00030000 开始的一个大内存区中 0012e000 开始的一个内存块，这个块的长度是 00002000，即 8KB。从状态字段的 MEM_COMMIT 标志可以看到这个内存块已经提交，保护属性为 PAGE_READWRITE（可读写）。再指定一个属于保护页范围的地址来观察：

```
0:000> !address 12d800
00030000 : 0012d000 - 00001000
    Type      00020000 MEM_PRIVATE
    Protect   00000104 PAGE_READWRITE | PAGE_GUARD
    State     00001000 MEM_COMMIT
    Usage     RegionUsageStack
    Pid.Tid  cf0.158c
```

可见这个内存块的大小是 0x1000 (4KB)，具有 PAGE_GUARD 属性。再选择一个属于未提交区域的地址，可以看到其状态为保留 (MEM_RESERVE)：

```
0:000> !address 12c800
00030000 : 00030000 - 000fd000
    Type      00020000 MEM_PRIVATE
    State     00002000 MEM_RESERVE
    Usage     RegionUsageStack
    Pid.Tid  cf0.158c
```

接下来，禁止断点 (bd 0) 并让程序执行，因为 deadloop 函数递归调用自身，所以会使用越来越多的栈空间，促使栈不断增长，直到导致栈溢出。以下 WinDBG 显示的栈溢出异常信息 (节选)：

```
(e34.894): Stack overflow - code c00000fd (first chance)
eip=00401009 esp=00032f60 ebp=00033160 iopl=0          nv up ei pl nz na po nc
```

其中，c00000fd 是栈溢出异常的异常代码，即 STATUS_STACK_OVERFLOW，从 ESP 寄存器的值可以看出此时的栈指针指向的是栈的倒数第 3 页，即当栈处于图 22-5 左侧所示的状态时，程序触及了栈保护页。

观察栈边界信息，可以看到栈边界已经递减到倒数第二个页面的起始位置：

```
0:000> !teb
StackBase:      00130000
StackLimit:     00031000
```

观察内存地址 00031000 和 00030000 的属性：

```

0:000> !address 00031000
00030000 : 00031000 - 000ff000
    Type      00020000 MEM_PRIVATE
    Protect   00000004 PAGE_READWRITE
    State     00001000 MEM_COMMIT
0:000> !address 00030000
00030000 : 00030000 - 00001000
    Type      00020000 MEM_PRIVATE
    State     00002000 MEM_RESERVE

```

可见此时最后一个页面的属性仍是 **MEM_RESERVE** (保留), 倒数第 2 个页面已经提交, 而且不具有保护属性, 正是我们前面介绍的图 22-6 右侧的状态。

观察全局变量 `g_nCount` (`dd StackOvr!g_nCount 11`), 它的值为 `0x79b`, 说明函数 `deadloop` 已经循环了这么多次。

按 F5 让程序继续执行, 系统会给调试器第二次处理机会, 再按 F5, 系统会恢复程序继续执行。于是 `deadloop` 函数继续循环, 继续消耗栈空间, 当访问到栈的最后一个页面时, 会触发访问异常:

```
(e34.894): Access violation - code c0000005 (first chance)
eip=00401009 esp=000300e20 ebp=00031020 iopl=0          nv up ei pl nz na pe nc
```

从 `ESP` 寄存器的值可以看到栈指针已经指向了没有提交的最后一个页面。

观察全局变量 `g_ESP`, 其值为 `00031028`, 这说明函数 `deadloop` 最后一次成功执行时 `ESP` 寄存器的值为 `00031028`, 也就是当时程序使用的是栈的倒数第二个页面, 也就是栈的最后一个可用页面。

再次观察全局变量 `g_nCount`, 其值为 `0x7ab`, 可见函数 `deadloop` 在发生栈溢出异常后, 又运行了 $7ab-79b=16$ 次。

C 运行库提供了一个名为 `_resetstkoflw` 的函数用来帮助应用程序从栈溢出异常中恢复, 以防止应用程序悄无声息地退出, VC 安装目录的 `crt\src\resetstk.c` 包含了 `_resetstkoflw` 函数的源代码。Windows XP 的 x64 版本支持通过名为 `SetThreadStackGuarantee` 的 API 设置报告栈溢出异常时仍然可用的栈空间。调用这个 API 可以预留更大的栈空间来处理栈溢出异常。

22.8.3 分配检查

前面我们介绍了利用栈保护页实现栈自动增长的工作原理。根据该原理, 在提交的栈空间用完后, 对栈空间的继续分配自然延伸到栈保护页, 当访问位于栈保护页的内容时, CPU 产生页错误异常, 然后系统平移栈保护页增大栈, 如此往复。但如果某一次栈分配需要的空间特别大, 超过了一个页面, 这时这个栈帧的某些部分就有可能一下子被分配到了保护页之外的未提交空间, 这便会导致访问违例。为了防止这种情况发生, 对于要分配较大 (超过一个页面) 栈空间的情况, 编译器会自动调用一个分配检查函数来把超过一个页面的分配分成很多次。

具体来说，对于超过一个页面的栈分配，分配检查函数会按照每次不超过一个页面大小逐渐调整栈指针，而且每调整一次，它会访问一次新分配空间中的内容，称为 Probe，目的是触发栈保护页平移，让栈增长。页面大小因处理器不同可能有所变化，x86 和 x64 都是 4KB，IA64 是 8KB。

VC 编译器使用的分配检查函数名叫 _chkstk，VC 安装目录下的 crt\src\intel\子目录中的 chkstk.asm 文件包含了这个函数的汇编语言源代码。编译器通常会对这个函数进行静态链接，也就是 _chkstk 函数通常直接被包含在使用它的模块中。

为了观察分配检查函数的工作方法，我们特意编写了一个名为 StackChk 的小程序（项目文件位于\code\chap22\StackChk 目录），其源代码如清单 22-19 所示。

清单 22-19 StackChk 程序的源代码

```
#include "stdafx.h"

#define BUFF_SIZE 1024*4           // 定义常量，大小为 x86 系统的内存页面大小 (4K)
void Func()
{
    char sz[BUFF_SIZE];          // 定义一个在线上分配的局部变量
    for(int i=0;i<BUFF_SIZE;i++)   // 循环操作局部变量
        sz[i]=(i+1)%127;          // 赋值
    printf("%s",sz);              // 打印显示
}
void main()
{
    Func();                      // 调用 Func 函数
}
```

在 Func 函数中，我们定义了一个比较大的局部变量 sz，其大小为 4KB。笔者的系统是基于 x86 的，因此 Func 函数满足了编译器插入栈检查函数的条件。在 VC6 中开始调试，然后使用反汇编窗口观察 Func 函数，在函数入口附近，可以看到有一个函数调用：

```
7: void Func()
8:
00401020 push    ebp
00401021 mov     ebp,esp
00401023 mov     eax,1044h // 将要分配的栈空间大小放入 EAX
00401028 call    $$000001 (004011e0)
```

单步跟踪进入这个 \$\$000001 函数，看到的便是 _chkstk 函数（见清单 22-20）。_chkstk 函数是使用 EAX 寄存器作为输入参数，在函数内部直接调整 ESP 寄存器，在线上分配出 EAX 长的栈帧。

清单 22-20 栈检查函数

```
1  _chkstk:
2  004011E0 push    ecx ; 保存 ECX 寄存器的值 (0)，第 17 行恢复
3  004011E1 cmp     eax,1000h ; 比较要分配的栈帧是否大于 4KB
4  004011E6 lea     ecx,[esp+8] ; 取调用本函数前的栈顶 (TOS)
5  004011EA jb     lastpage (00401200) ; 如果#3 的比较结果为小于，则转到第 12 行
```

```

6   probepages:          ; 这个代码块的作用是分配整页 (4KB) 的部分
7   004011EC  sub        ecx,1000h ; 对 ECX 递减 4KB, ECX 是用来记录新的栈顶值
8   004011F2  sub        eax,1000h ; 对 EAX (未分配大小) 递减 4KB
9   004011F7  test       dword ptr [ecx],eax ; 访问 (Probe) 新分配空间的最顶部一次
10  004011F9  cmp        eax,1000h ; 比较还未分配的空间是否大于 4KB
11  004011FE  jae       probepages (004011ec) ; 如果大于或等于, 则到第 6 行
12  lastpage:           ; 这个代码块的作用是分配不足 4KB 的零头部分
13  00401200  sub        ecx,eax ; 递减 ECX, 即将 EAX 剩余的零头部分分配完
14  00401202  mov        eax,esp ; 将 ESP 值放入 EAX 保存, 现在 EAX=12FF24
15  00401204  test       dword ptr [ecx],eax ; 访问 (Probe) 新分配空间的最顶部一次
16  00401206  mov        esp,ecx ; 将新的栈顶值赋给 ESP = 调用本函数前的 ESP - EAX
17  00401208  mov        ecx,dword ptr [eax] ; 恢复第 1 行压入的 ECX 寄存器值 (0)
18  0040120A  mov        eax,dword ptr [eax+4] ; 将函数返回地址赋给 EAX
19  0040120D  push      eax ; 将返回地址压到栈顶, RET 指令使用后会自动将其弹出
20  0040120E  ret

```

在 Func 函数调用 _chkstk 前 ESP 的值是 0x12FF2C，又返回到 Func 函数时，ESP 寄存器为 0x12EEE8，二者的差恰好为 0x1044，即要分配的栈帧大小。

默认情况下，编译器在编译程序时，会对超过一个内存页大小的栈帧自动插入 _chkstk 函数，可以通过编译器选项来改变这个阀值：

/Gs[size]

例如，我们可以在项目属性的 C++ 编译选项中加入 /Gs2048，然后将 #define BUFF_SIZE 1024*4 改为 #define BUFF_SIZE 1024*2，此时尽管 Func 函数要分配空间不足一个内存页大小，但是编译器还是会插入 _chkstk 函数。如果去掉 /Gs2048，那么对于修改后的程序，编译器就不会插入 _chkstk 函数了。

也可以通过 #pragma 指令来控制是否插入分配检查函数：

```
#pragma check_stack([ {on | off}])
#pragma check_stack(+ | -)
```

但是这只能控制栈帧大小低于规定阀值的函数，不会阻止编译器为栈帧超过规定阀值的函数插入 _chkstk 函数。比如下面的 FuncVia 函数，尽管其前面有 check_stack -，但因为其栈帧有 98K，所以编译器还是会为其加入 _chkstk 函数的。

```
#pragma check_stack -
void FuncVia()
{
    char sz[1024*98];
    sz[0]=66,sz[1]=0;

    printf("%s\n",sz);
}
#pragma check_stack()
```

强制编译器不要加入 _chkstk 函数的方法是，在项目属性中设置一个很大的阀值，如 /Gs102400。这样设置并编译后，再执行这个函数时便会导致访问异常。这是因为当 FuncVia 函数对新分配的局部变量进行访问时 (sz[0]=66)，由于该内存页尚未提交而导致异常。观察此时的 FuncVia 函数的反汇编，可以看到不再有 _chkstk 函数：

StackChk!FuncVia:

```

00401050 81ec00880100    sub      esp,0x18800
00401056 8d442400          lea      eax,[esp]
0040105a c644240042        mov     byte ptr [esp],0x42

```

观察发生异常时的 ESP 和 EIP 寄存器，以及栈的边界：

```

0:000> r esp,eip
esp=00117784 eip=0040105a
0:000> !teb
StackBase:           00130000
StackLimit:          0012e000

```

可以看到栈顶的地址 0x117784 已经越过栈边界 12e000 很多个页面，这便是我们前面说的直接跳过栈保护页而访问未提交空间的情况，从这个例子我们可以理解 chkstk 函数的必要性。

22.9 栈下溢

通常把访问栈边界以上的空间称为栈溢出（Stack Overflow），比如上一节介绍的 StackOvr 程序由于 deadloop 函数无限循环而导致了栈溢出。类似的，把访问栈底以下的空间称为栈下溢（Stack Underflow）。这里的“以上”和“以下”都是相对于栈顶（stack top）在上栈底（stack base）以下的栈布局图而言的，而不是指地址值大小。因为栈是向低地址方向生长的，栈顶的地址小，栈底的地址大，所以栈底以下是指比栈底的地址还大的内存区域。

以下是一个用来演示栈下溢的小程序 StkUFlow 的源代码：

```

#pragma optimize("",off)
void main()
{
    char sz[20];
    sz[2000]=0;
}

```

执行以上程序会导致访问违例（Access Violation）异常，异常发生在赋值语句中：

```

(12b4.ed8): Access violation - code c0000005 (first chance)
StkUFlow!main+0x6:
00401006 c685bc07000000    mov     byte ptr [ebp+0x7bc],0x0 ss:0023:0013073c=00

```

也就是向地址 `ebp+0x7bc` (0013073c) 赋值时触发了异常，使用 `!address` 命令观察这个地址：

```

0:000> !address 0013073c
00130000 : 00130000 - 00003000
              Type      00040000 MEM_MAPPED
              Protect   00000002 PAGE_READONLY
              State     00001000 MEM_COMMIT
              Usage     RegionUsageIsVAD

```

可见，这个内存地址所在的内存块具有只读属性（PAGE_READONLY），所以上面的操作导致了访问违例。观察 EBP 和 ESP 寄存器的值：

```

0:000> r ebp,esp
ebp=0012ff80 esp=0012ff6c

```

也就是 0012ff6c 到 0012fff8 之间的 20 个字节是分配给局部变量 sz 的。因此可以用 esp 寄存器来引用它(esp 指向 sz 数组的第一个元素), 也可以用 ebp-0x14 来引用它。这样算来 $ebp+0x7bc=ebp+1980$ 也就是等于 $ebp-0x14+2000$, 即 $ebp+0x7bc$ 正是 $sz[2000]$ 。

使用 !teb 命令观察栈的基地址:

```
0:000> !teb
StackBase:          00130000
StackLimit:         0012e000
```

可见 $ebp+0x7bc=0013073c$, 已经超越了栈底, 也就是指向了栈之外的区域, 形成了栈下溢。在 Windows 98 中, 为了防止栈下溢, 系统会在栈底之下保留额外的 64KB, 用于捕捉栈下溢错误。但是在 NT 系列的 Windows 系统中没有这种保护。

22.10 缓冲区溢出

缓冲区是程序用来存储数据的连续内存区域, 一旦分配完成, 其起止地址(边界)和大小便固定下来。当使用缓冲区时, 如果使用了超出缓冲区边界的区域, 那么便称为缓冲区溢出(Buffer Overflow), 或缓冲区越界(Buffer Overrun)。如果缓冲区是分配在栈上的, 那么又称为栈缓冲区溢出, 如果是分配在堆上的, 又称为堆缓冲区溢出, 本节我们将讨论发生在栈上的缓冲区溢出。

22.10.1 感受缓冲区溢出

下面我们通过一个非常简单的 BufOvr 小程序来感受由于缓冲区溢出所导致的程序崩溃(见清单 22-21)。

清单 22-21 BufOvr 程序的源代码

```
1 int main(int argc, char* argv[])
2 {
3     char szInput[5];
4     printf("Enter string:\n");
5     gets(szInput);
6     printf("You Entered:\n%s\n", szInput);
7     return 0;
8 }
```

在命令行窗口运行编译好的 BufOvr 程序, 然后根据提示输入一串字符 987654321012345, 再按回车结束输入。

```
Enter string:
987654321012345
```

而后程序显示出已经输入的字符串:

```
You Entered:
987654321012345
```

但紧接着便出现应用程序错误对话框, 说明该程序内发生了崩溃性错误, 观察错误的

异常代码，是 0xC0000005，即访问违例。

如果在调试器（VC6）中运行，输入同样的一串数字，则会接受到访问违例异常的第一次处理机会。观察此时的程序指针寄存器，其值为 0x00353433，把这个地址与进程内的各个模块起止地址相匹配，发现它不属于任何模块。这是缓冲区溢出的一个典型症状，由于缓冲区溢出，存放在栈上的函数返回地址被破坏，因此函数返回到错误的地方。

分析源代码，问题应该出现调用 gets 函数读取用户输入这一行，即 gets(szInput)。重新运行程序，对 gets 函数设置断点，当断点命中时，观察栈顶（ESP 寄存器）的值为 0x12ff2c，使用内存窗口显示出栈内容（图 22-7 左），对照 main 函数的反汇编代码，可以看出栈顶的三行分别是保存的 EDI、ESI 和 EBX 寄存器的值，接下来从 12FF38 到 12FF78 的 64 个字节是调试版为每个函数分配的默认局部变量。从 12FF78 开始的 8 个字节便是分配给 szInput 变量的了。尽管我们定义了 5 个字节的字符串数组，但因为 32 位系统下栈分配的最小单位是 4 字节，所以为 szInput 变量分配了 8 个字节。因为我们没有对缓冲区进行初始化，所以编译器插入的代码帮我们用 0xCC（INT 3 指令的机器码）填充整个缓冲区。

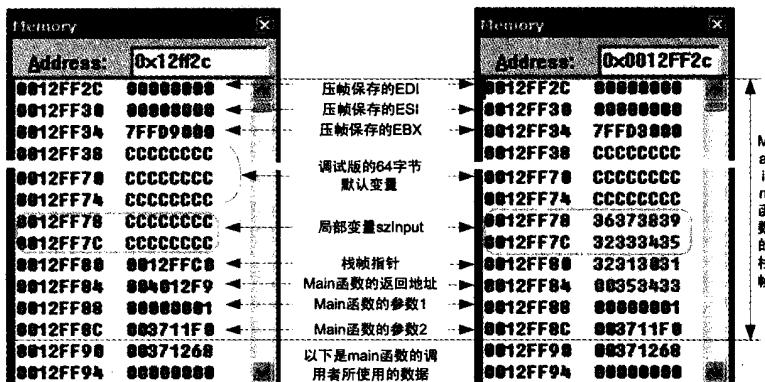


图 22-7 缓冲区溢出前（左）后（右）的栈内数据

单步执行 gets(szInput) 语句，输入 987654321012345，然后按回车，程序会再次中断调试器。此时的栈指针依然为 0x12ff2c。再观察栈内数据（图 22-7 右），红色区域（0x12FF78~0x12FF88）是已经发生变化的数据。我们看到 szInput 变量的内容变化了，其前四个字节的数据为 0x36373889，如果显示为字符则为“9876”接下来的 4 个字节（地址为 0x12FF7C）为“5432”，这正是我们输入的字符。但因为我们只定义了 5 个字节的缓冲区，尽管系统多分配给我们 3 个字节，现在还是已经用完了。我们共输入了 15 个字符，现在保存在 szInput 变量所申请的缓冲区中 8 个，剩下的 7 个呢？继续向下（高地址方向）看，在本来用来保存栈帧指针的地方（地址为 0x12FF80）我们发现了“1012”4 个字符，在本来保存 main 函数返回地址的地方可以看到“345”

三个字符，另一个字节是 0，即字符串的结束符。

可见，由于 `gets` 函数向本来只申请 5 个字节长度的缓冲区存放了 15 个字符（实际上是 16 个，包括结束的 0），这导致了缓冲区溢出，将栈中的栈帧指针和函数返回地址都意外地覆盖掉了。因为被覆盖掉的是 `main` 函数的返回地址，`get()` 函数的返回地址（在地址小于 0x12ff2c 的更上方）没有受到影响，所以 `gets()` 函数还可以顺利返回到 `main` 函数中，接下来 `main` 函数可以继续执行一段时间，所以我们可以看到 `printf` 函数打印出的输入字符串。这是缓冲区溢出错误的另一个特征，即错误不会立刻体现出来，程序可能继续执行一段时间才暴露出异常。

显示出反汇编窗口，继续跟踪 `main` 函数中的 `return` 语句，尽管编译器自动加入了 `_chkesp` 函数，但由于该函数只是检查栈指针（ESP）与保存在 EBP 寄存器中的进入函数时的栈指针值是否相同，因此它并不能发现缓冲区溢出所导致的栈破坏。当执行到 `ret` 指令时，栈指针的值为 0x0012FF84，即刚好指到本来存放函数返回地址的地方。但是系统不知道现在的返回地址已经是错误的了，继续执行，程序便返回到错误的 0x00353433 处。因为该地址不属于任何模块，对应的内存区不具有合适的类型和保护属性，所以当 CPU 读取并执行这里的指令时产生了访问违例异常。包含代码的内存块通常具有 `MEM_IMAGE` 类型和 `PAGE_EXECUTE_READ` 保护属性。

如果再执行一遍这个程序，仍然在调试器中执行调试版本的 `BufOvr.exe`，并撤除一切断点，而且这次将输入换为“987654321012i4@”（不包括引号，共 15 个字符）。我们发现这次程序打印出输入的字符串后中断到调试器，停在 00403469 处，此处的指令为 `INT 3 (CC)`。

```
(c64.ccc): Break instruction exception - code 80000003 (first chance)
eax=00000000 ebx=7ffd7000 ecx=00425a58 edx=00425a58 esi=00000000 edi=00191f18
eip=00403469 esp=0012ff88 ebp=32313031 iopl=0 nv up ei pl zr na po nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000246
BufOvr!_setenvp+0x149:
00403469 cc int 3
```

可以看出，返回到的地址值（00403469）就是我们输入的最后三个字符“i4@”的 ASC 代码。之所以输入这三个字符，是我们“精心”设计的。是通过浏览反汇编后的程序代码，发现这个地址处有编译器补位用的断点指令，于是我们将这个地址换为对应的字符。

推而广之，只要我们巧妙设计要输入给缓冲区的数据，就可以控制该缓冲区所在函数的返回位置。如果我们加长输入的内容，使其包含一段代码，然后再将返回地址指向这段代码，那么就可以让这个程序执行我们动态注入的代码了。这便是利用缓冲区溢出进行恶意攻击的基本原理。

22.10.2 缓冲区溢出攻击

我们再来看一个例子，清单 22-22 是我们用来演示缓冲区攻击的 `BoAttack` 程序的

主要代码。其中的 `HandleInput` 函数是用来处理用户输入的一个函数，它接受一个字符串类型的参数，其内部首先将参数所指定的字符串复制到一个缓冲区 `szBuffer` 中，然后再进行解析和处理（省略）。类似这样的代码在不少软件中都可以看到，但其中却隐藏着严重的安全漏洞，如果参数指定的字符串长度大于 32 个字节，那么便会导致缓冲区溢出。

清单 22-22 BoAttack 程序的主要代码

```
1 void HandleInput(LPCTSTR lpszInput)
2 {
3     TCHAR szBuffer[31];
4     strcpy(szBuffer,lpszInput);
5     OutputDebugString(szBuffer);
6     //process the content in buffer...
7 }
8
9 int APIENTRY WinMain(HINSTANCE hInstance,
10                      HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow)
11 {
12     MessageBox(NULL, "A sample to demo buffer overflow attack.",
13                "AdvDbg", MB_OK);
14     HandleInput(SZ_INPUT);
15     return 0;
16 }
```

出于演示方便，我们用一个设计好的字符串（常量 `SZ_INPUT`，稍后分析）作为输入来调用 `HandleInput` 函数，现实中的攻击可能是通过网页或窗口界面输入给程序的。执行程序，我们会发现，除了源代码中第 12 行和第 13 行弹出的消息框之外，还会有一个带有 `OK` 和 `CANCEL` 按钮的空白对话框弹出（图 22-8），点击 `OK` 后该消息框会再次弹出，直到点击 `CANCEL`，而后程序崩溃结束。

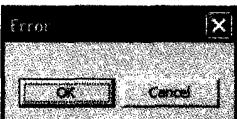


图 22-8 通过缓冲区溢出弹出的消息框

这个空白对话框就是因为缓冲区溢出而执行字符串中的“恶意”代码而弹出的，“恶意”代码都包含在传递给 HandleInput 函数的参数字符串中，即 SZ_INPUT:

上面的字符串包含三部分内容，前三行是“恶意”代码的机器码，最后一行是用来覆盖函数返回地址的新地址，即“恶意”代码的内存地址，中间部分是用来调整位置的。前三行的机器码所对应的汇编指令如下：

TAG_LOOP:
push MB_OKCANCEL ; 压入 MessageBoxA 的最后一个参数 uType

```

xor eax,eax          ; 将 EAX 清 0
push eax             ; 压入 MessageBoxA 的参数 lpCaption
push eax             ; 压入 MessageBoxA 的参数 lpText
push eax             ; 压入 MessageBoxA 的参数 hWnd
mov edx, 77d8050bh   ; 将函数 MessageBoxA 的地址赋给 EDX
call edx             ; 调用 MessageBoxA 函数
dec eax              ; 对 MessageBoxA 函数的返回值递减
test eax,eax         ; 测试返回值
jz TAG_LOOP          ; 如果选择的是 OK (1) 则循环, 选 CANCEL 的返回值为 2

```

看了以上代码，大家就明白了空白消息框是因为调用 `MessageBox` API 弹出的，`lpText` 和 `lpCaption` 参数都设为 `NULL`，所以它没有标题和提示信息。

因为是演示，所以我们的“恶意”代码并无恶意，只是显示一个丑陋的消息框。但通过这段代码我们知道，既然可以执行代码调用 `MessageBox` API，那么便可以调用其他函数，执行其他操作。通过这个例子，我们应该认识到缓冲区溢出攻击的危险性。

22.11 变量检查

为了及时发现缓冲区溢出，Visual Studio 的 C/C++ 编译器逐步引入了一系列功能来帮助应用程序检查缓冲区溢出，包括用于精确检查每个局部变量是否溢出的变量检查功能，以及用于保护函数返回地址的安全 Cookie 功能。本节和下一节将分别介绍这两项功能。

我们通过一个例子来感受变量检查功能。使用 VC8 创建一个空白的 C++ 项目（Windows 本地程序），取名为 SecChk，然后把上一节的 `HandleInput` 函数和有关的常量复制过来。编译并运行该程序的调试版本，我们会得到图 22-9 所示的对话框。如果在 VS2005 集成环境中执行，Visual Studio 的集成调试器会显示一个包含类似提示信息和 `Break/Continue` 按钮的对话框。

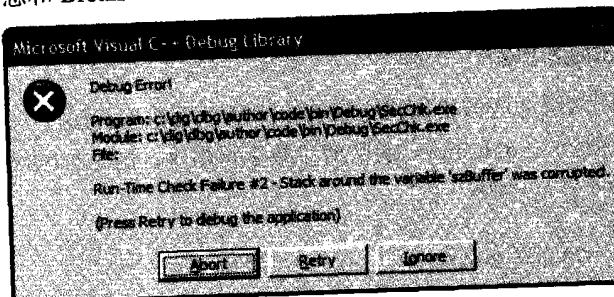


图 22-9 编译器插入的变量检查代码检测到缓冲区溢出错误

看来编译器插入的代码检测到了缓冲区溢出，而且还指出了发生溢出的变量名是“`szBuffer`”。那么编译器插入了什么样的代码，如何进行检查的呢？我们先来看一看 VC8 编译的 `HandleInput` 函数的反汇编代码（见清单 22-23），以了解其中的奥秘。

清单 22-23 VC8 编译出的 HandleInput 函数的反汇编代码（调试版本）

```

1 void HandleInput(LPCTSTR lpszInput)
2 {
3     004113C0 push    ebp ; 保存发起调用函数的栈帧指针
4     004113C1 mov     ebp,esp ; 建立栈帧
5     004113C3 sub    esp,0ECh ; 分配局部变量，ECh=236 字节
6     004113C9 push    ebx ; 保存 nonvolatile 寄存器，EBX
7     004113CA push    esi ; 保存 nonvolatile 寄存器，ESI
8     004113CB push    edi ; 保存 nonvolatile 寄存器，EDI
9     004113CC lea     edi,[ebp-0ECh] ; 取局部变量的起始地址
10    004113D2 mov     ecx,3Bh ; 设置循环次数
11    004113D7 mov     eax,0CCCCCCCCCh ; 将 INT 3 机器码放入 EAX
12    004113DC rep stos dword ptr es:[edi] ; 将 EAX 中的值循环复制到 EDI 指向的区域
13    004113DE mov     eax,dword ptr [__security_Cookie (417030h)] ;
14    004113E3 xor     eax,ebp ; 将 EAX 中的安全 Cookie 与 EBP 做异或运算，增大随机性
15    004113E5 mov     dword ptr [ebp-4],eax ; 将安全 Cookie 赋给局部变量
16    TCHAR szBuffer[31];
17    strcpy(szBuffer,lpszInput);
18    004113E8 mov     eax,dword ptr [ebp+8] ; 参数 1 的值赋给 EAX
19    004113EB push    eax ; 将 EAX 压入栈，作为 strcpy 的参数
20    004113EC lea     ecx,[ebp-28h] ; 取局部变量 szBuffer 的地址
21    004113EF push    ecx ; 压入 szBuffer，作为 strcpy 的参数
22    004113F0 call    @ILT+170(_strcpy) (4110AFh) ;
23    004113F5 add     esp,8 ; 释放因为调用 strcpy（使用 C 调用协定）而压入的参数
24    OutputDebugString(szBuffer);
25    004113F8 mov     esi,esp ; 将当前栈指针的值赋给 ESI
26    004113FA lea     eax,[ebp-28h] ; 取局部变量 szBuffer 的地址
27    004113FD push    eax ; 压入 szBuffer
28    004113FE call    dword ptr [__imp_OutputDebugStringA@4 (41825Ch)] ; 调用
29    00411404 cmp     esi,esp ; 比较现在的 ESP 值是否与第 25 行保存起来的值一样
30    00411406 call    @ILT+315(__RTC_CheckEsp) (411140h) ; 调用栈指针检查函数
31    //process the content in buffer...
32 }
33    0041140B push    edx ; 保存 EDX 寄存器
34    0041140C mov     ecx,ebp ; 将栈帧指针的值赋给 ECX，供变量检查函数使用
35    0041140E push    eax ; 保存 EAX 寄存器的值
36    0041140F lea     edx,[ (41143Ch)] ; 取变量信息表的地址，放入 EDX
37    00411415 call    @ILT+145(@__RTC_CheckStackVars@8) (411096h) ; 变量检查函数
38    0041141A pop    eax ; 弹出第 35 行保存的 EAX
39    0041141B pop    edx ; 弹出第 33 行保存的 EDX
40    0041141C pop    edi ; 弹出第 8 行保存的 EDI
41    0041141D pop    esi ; 弹出第 7 行保存的 ESI
42    0041141E pop    ebx ; 弹出第 6 行保存的 EBX
43    0041141F mov     ecx,dword ptr [ebp-4] ; 将第 15 行保存的安全 Cookie 赋给 ECX
44    00411422 xor     ecx,ebp ; 再次与 EBP 异或，与第 14 行相呼应
45    00411424 call    @ILT+30(@__security_check_Cookie@4) (411023h) ; 安全检查
46    00411429 add     esp,0ECh ; 释放局部变量，与第 5 行对应
47    0041142F cmp     ebp,esp ; 比较 EBP 和 ESP
48    00411431 call    @ILT+315(__RTC_CheckEsp) (411140h) ; 调用栈指针检查函数
49    00411436 mov     esp,ebp ; 将 EBP 赋给 ESP
50    00411438 pop    ebp ; 弹出保存的发起调用函数的栈帧
51    00411439 ret     ; 返回到发起调用的函数

```

观察上面的反汇编代码，我们可以看出 VC8 与 VC6 编译产生的代码有以下几点明显的不同。

- VC8 为每个函数的调试版本所分配的默认局部变量比 VC6 更长, VC6 分配的长度通常是 64 字节, VC8 分配的是 192 字节。即使一个函数内部没有任何语句, 编译器也会在栈帧中为这个默认局部变量分配空间。
- VC8 会在紧邻栈帧指针的位置分配 4 个字节用于存放安全 Cookie。安全 Cookie 好似一个安全护栏, 用于保护其下的栈帧指针和函数返回地址。安全 Cookie 一旦损坏则说明函数内已经发生了缓冲区溢出, 栈帧数据可能已经受损。在函数的出口, VC8 会插入代码 (第 43~45 行) 检查安全 Cookie 的完好性, 下一节将详细讨论。
- 在给每个局部变量分配空间时, 除了补齐内存对齐所需的零头部分, VC8 还会给每个变量多分配 8 个字节, 放在前面 4 个字节、后面 4 个字节, 这前后各 4 个字节总被初始化为 0xCCCCCCCC。这样, 每个变量都相当于有了前后两个屏障, 一旦这两道屏障受损了, 则说明该变量附近发生了溢出。为了行文方便, 我们将这两个字段称为变量的屏障字段。比如上面的代码, szBuffer 的定义长度是 31 个字节, 按内存对齐要求应该分配 32 个字节, 再加上前后各 4 个 4 节则 szBuffer 真正被分配了 40 个字节。加上安全 Cookie 占 4 个字节, 默认分配 192 个字节, 所以第 5 行总共分配 $192+40+4=236$ 字节。
- VC8 默认使用 _RTC_CheckEsp 函数来检查栈指针的平衡性, 其原理与 VC6 使用的 _chkesp 完全相同。不过除了像 VC6 那样在函数末尾 (第 47 和 48 行) 检查栈指针外, 当函数调用其他函数时, 如果要调用的函数采用的是被调用函数清理栈的调用协定时, VC8 会在调用这个函数前记录下栈指针的值, 调用函数后进行检查。在上面的代码中, strcpy 使用的是 C 调用协定, 启发调用的函数负责清理栈 (第 23 行), 所以没有对这个函数插入栈指针检查代码。而 OutputDebugString 函数使用的是标准调用协定, 被调用函数负责清理栈, 所以 VC8 在调用这个函数前后插入了用于进行栈指针检查的代码 (第 25 行和第 29、30 行), 确保调用这个函数前后栈指针没有发生变化。
- 对于有可能发生缓冲区溢出的函数 (如上面的函数), 在函数返回前, VC8 会调用 _RTC_CheckStackVars 做变量检查, 也就是检查每个变量前后的屏障字段是否完好无损。如果发现屏障字段的内容发生变化 (不再全是 0xCC), 则调用错误报告函数报告错误, 前面图 22-2 所示的对话框就是 _RTC_CheckStackVars 函数检查到 szBuffer 变量后面 (高地址端) 的屏障字段被破坏而发出的错误报告。

在以上措施中, 我们把插入和检查安全 Cookie 称为安全检查 (Security Check)。检查栈指针和针对每个变量的检查是运行期检查 (Runtime Check) 的一部分, 我们在第 21 章和第 22.6 节已经介绍了一些运行期检查功能。

下面继续介绍变量检查的工作原理, 我们将其分为三个部分, 首先, 在分配局部变量时编译器会为每个局部变量多分配 8 个字节的额外空间 (前后各 4 个字节), 用作

屏障字段，在填充局部变量区域时，这些屏障字段，以及变量尾部的因为内存对齐而分配的补足字节都会被 0xCC（INT 3 指令的机器码）所填充，我们把这些 0xCC 称为栅栏字节。第二，为了在运行期仍能够准确知道每个变量的长度、位置和名称，编译器会产生一个变量描述表，用来记录局部变量的详细信息。第三，在函数返回前，调用 _RTC_CheckStackVars 函数，根据变量描述表逐一检查其中包含的每个变量，如果发现变量前后的栅栏字节发生变化则报告检查失败。

为了更好地理解以上内容，我们再来看一个包含更多变量的函数。在清单 22-24 所示的 VarCheck 函数（也位于 SecChk 项目中）中，我们一共定义了 7 个局部变量、三个字符串数组、三个整数、一个字符串指针。

清单 22-24 用于演示变量检查功能的 VarCheck 函数

```

1 void VarCheck(LPCTSTR lpszInput)
2 {
3     int n;
4     TCHAR szFstBuffer[3];
5     LPTSTR lpsz;
6     TCHAR szSndBuffer[5];
7     int m=9;
8     TCHAR szThdBuffer[3];
9     int j;
10    j=n=m=strlen(lpszInput);
11    sprintf(szFstBuffer, "%d\n", n);
12    OutputDebugString(szFstBuffer);
13
14    lpsz=szThdBuffer;
15    strcpy(lpsz,lpszInput);
16    strcat(szSndBuffer,lpsz);
17    OutputDebugString(szSndBuffer);
18 }

```

在 WinMain 函数中使用 VarCheck("DB") 调用该函数并使用 VC8 的集成调试器单步跟踪到函数末尾（18 行）。通过寄存器窗口可以看到此时的 EBP 和 ESP 寄存器的值为：

ESP = 0012FD0C EBP = 0012FE34

打开一个内存观察窗口，观察 0012FD0C 开始的内存区域，便可以看到栈中的原始数据（图 22-10）。

在图 22-10 中，我们可以清楚地看到每个局部变量的位置、内容，以及它们前后的栅栏字节。除了屏障字段，在字符串变量的补齐位置，我们也可以看到编译器填充的 0xCC，例如 szFstBuffer，它的定义长度是 3 个字节，补齐后为 4 个字节，第 4 个字节被初始化为 0xCC。因为图 22-10 是按 4 字节（DWORD）显示的，所以对于字符串型数据，我们应该颠倒过来观察。以变量 szFstBuffer 为例，图中的显示 DWORD 格式，内容为 0xcc000a32，恢复成字节顺序是 320a00cc，其中 32 是 ASCII 码 ‘2’，0a 是换行符 ‘\n’，00 是字符串的结束符，cc 是填充的断点指令。

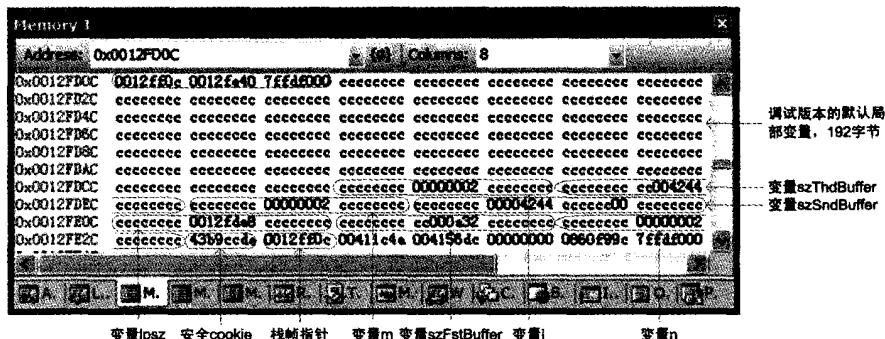


图 22-10 观察局部变量在栈上的分配情况

在图中，我们也可以看到安全 Cookie，它位于局部变量和重要的栈帧信息（栈帧指针及函数返回地址）之间。这样，如果 Cookie 之上的局部变量发生溢出，在漫延到栈帧信息前会先覆盖 Cookie，因此检查 Cookie 的完整性便可以探测到可能危及栈帧信息的溢出情况。

靠近栈顶的三个 DWORD 是保存的 non-volatile 寄存器，即 EDI, ESI 和 EBX，因为根据函数调用协定，每个函数退出时都应保证这三个（其实还有 EBP）寄存器的值保持不变。中间的大块区域是调试版本的默认布局变量（192 字节）。

在对变量分配有了比较深入的理解之后，我们看一下编译器插入的 _RTC_CheckStackVars 函数，该函数的原型为：

```
void __fastcall _RTC_CheckStackVars(void *_Esp, _RTC_framedesc *_Fd);
```

其中第一个参数是栈指针，但它并不是调用此函数时的栈顶位置，而是建立函数栈帧时的栈顶位置，也就是 EBP 寄存器的值。第二个参数是一个指向 _RTC_framedesc 结构的指针。_RTC_framedesc 是用来描述栈帧内的每个变量，也就是我们前面说的变量描述表， rtcapi.h 文件中包含了这个结构的定义：

```
typedef struct _RTC_framedesc {
    int varCount;
    _RTC_vardesc *variables;
} _RTC_framedesc;
```

可以把 _RTC_framedesc 结构理解成一个表格的表头，其中 varCount 是这个表格所描述变量的个数（行数），variables 指向被描述的每个变量，它是一个数组结构，每个元素是一个 _RTC_vardesc 结构：

```
typedef struct _RTC_vardesc {
    int addr;           // 变量的偏移地址
    int size;           // 变量的大小，以字节为单位
    char *name;         // 变量的名称
} _RTC_vardesc;
```

其中 addr 是变量的地址，它是用相对于栈帧地址（即 EBP 寄存器）的偏移来表示的，

因此都是负数，size 是变量的长度（以字节为单位），这里的长度是指定义长度，不是实际分配长度，因此不包括栅栏字节，name 指向包含变量名称的字符串。

编译器在编译需要加入变量检查的函数时，会以静态变量的形式来定义变量描述表，观察的编译器产生的汇编文件可以清楚地看到这些定义。首先在项目属性中打开编译器的汇编输出功能（C/C++>Output Files>Assembler Output>选择 Assembly with Source Code (/FAs)），重新编译后打开 seccheck.asm 文件（位于项目的 Debug 目录中），清单 22-25 给出了 VarCheck 函数的主要代码。

清单 22-25 编译器产生的包含变量检查功能的汇编代码

```
?VarCheck@@YAXPBDD@Z PROC ; VarCheck, COMDAT
;...函数的汇编语言代码，省略多行
$LN7@VarCheck:
    DD   3           ; 变量个数，即 varCount 字段
    DD   $LN6@VarCheck ; variables 字段，以下是连续的_RTC_vardesc 结构
$LN6@VarCheck:
    DD   -24          ; 变量 1 的偏移地址，相对于帧的基地址 (EBP)
    DD   3             ; 变量 1 (szFstBuffer) 的长度 3
    DD   $LN3@VarCheck
    DD   -52          ; 变量 2 的偏移地址，相对于帧的基地址 (EBP)
    DD   5             ; 变量 2 (szSndBuffer) 的长度 5
    DD   $LN4@VarCheck
    DD   -76          ; 变量 3 的偏移地址，相对于帧的基地址 (EBP)
    DD   3             ; 变量 3 (szThdBuffer) 的长度 3
    DD   $LN5@VarCheck
$LN5@VarCheck:
    DB   115          ; 变量 1 名称
    ; ...省略多行
    DB   0
$LN4@VarCheck:
    DB   115          ; 变量 2 名称
    ; ...省略多行
    DB   0
$LN3@VarCheck:
    DB   115          ; 变量 3 名称
    DB   0             ; 00000073H, 字符 s
    ; ...省略多行
    DB   0
?VarCheck@@YAXPBDD@Z ENDP ; VarCheck
```

这些静态变量按照上面的布局存储在目标文件的静态变量区中，形成了一个 _RTC_framedesc 结构，图 22-11 所示的内存观察窗口显示了它们在内存中的排列方式。

图 22-11 中，先是 _RTC_framedesc 结构的头信息，而后是多个相邻的表项，每一项是一个 _RTC_vardesc 结构。而后是变量名字符串。图中共包含了 3 个变量的描述，依次为 szFstBuffer、szSndBuffer 和 szThdBuffer。我们知道 VarCheck 实际上共有 7 个局部变量，这里只有 3 个，因为 VC8 目前只检查更容易发生溢出错误的字符串数组变量，因此没有为整型变量和指针变量建立描述信息。

表格地址	表项数	表项起始地址	表项1. 对szFstBuffer变量的描述
Memory 3	1	0x004137F4	
Address	0x004137F4	00000003 004137F4 0fffffa8 00000003 7A	表项2. 对szSndBuffer变量的描述
0x00413604	00413638 0fffffc8 00000005 00413624 06A 6A	表项3. 对szThdBuffer变量的描述	
0x00413614	0fffffb4 00000003 00413620 06547a79 8A		
0x00413624	86754264 00726566 84537a73 86754264 dBuffer . szSndBuf		
0x00413634	00726564 73487a73 86754274 00726566 fer . szFstBuf		
0x00413644	00726564 73487a73 86754274 00726566 fer . szFstBuf		

图 22-11 编译器为 VarCheck 函数产生的变量描述表

在调用 _RTC_CheckStackVars 函数时，编译器会将变量信息表的地址传递给 _RTC_CheckStackVars 函数，以下是编译器插在 VarCheck 函数末尾插入的代码：

```
004137C2 push      edx
004137C3 mov       ecx,ebp
004137C5 push      eax
004137C6 lea       edx,[ (4137F4h)]
004137CC call     @ILT+145(@_RTC_CheckStackVars@8) (411096h)
```

因为 _RTC_CheckStackVars 函数使用的是快速调用协定，所以我们看到 EBP 寄存器的值被放入 ECX 寄存器，变量描述表的地址（4137F4h）被放入 EDX 寄存器。

有了以上基础后，我们就可以很容易地理解 _RTC_CheckStackVars 函数的工作方法了。清单 22-26 给出了 _RTC_CheckStackVars 函数的伪代码。

清单 22-26 _RTC_CheckStackVars 函数的伪代码

```
1 void __fastcall RTC_CheckStackVars(void *_Frame, _RTC_framedesc *_Fd)
2 {
3     DWORD* pdwFense, _RetAddr;
4
5     _RTC_vardesc * pVarDesc= _Fd->variables;
6
7     for( int i=0; i< _Fd->varCount;i++)
8     {
9         pdwFense=(DWORD*) ((DWORD)_Frame+
10             (DWORD)pVarDesc->addr-offsetof(DWORD));
11         if(*pdwFense!=0xFFFFFFFF)
12             goto TAG_CORRUPT_FOUND;
13         pdwFense=(DWORD*) ((DWORD)_Frame+
14             (DWORD)pVarDesc->addr+pVarDesc->size);
15         if(*pdwFense!=0xFFFFFFFF)
16         {
17             TAG_CORRUPT_FOUND:
18                 _asm mov edx,dword ptr [ebp+4] ;
19                 _asm mov _RetAddr, edx;
20                 _RTC_StackFailure((void *)_RetAddr,pVarDesc->name);
21         }
22
23         pVarDesc++;
24     }
25 }
```

以上代码循环检查参数 _Fd 所指定的变量描述表中的每个变量，检查方法就是根据变量的偏移和参数 _Frame 传入的栈帧基址得到变量的起始地址，然后再向前偏移 4 个字节得到变量前的屏障字段地址（9~10 行），然后检查其值是否为 0xFFFFFFFF

(11行)。类似的，栈帧地址加上变量的偏移地址再加上变量长度，便得到变量后的栅栏字节的地址。需要说明的是，如果有用于补齐的字节，那么_RTC_CheckStackVars 函数检查的是从第一个补齐字节开始的 4 个字节，也就是说，即使小于 4 字节的少量溢出只是破坏了补齐用字节，也是可以检测到的。

如果_RTC_CheckStackVars 函数发现变量前后的栅栏被破坏，那么便会调用_RTC_StackFailure 函数报告错误，_RTC_StackFailure 函数的原型如下：

```
void _RTC_StackFailure(void * _RetAddr,char const* szVarName);
```

其中第一个参数是从栈上取得的_RTC_CheckStackVars 函数的返回地址，它是位于被检查函数中的。当我们在 VC8 的集成调试中调试时，如果选择错误报告窗口中的中断(Break)按钮，那么就是中断到这个地址。如果是在其他调试器中运行，那么选择图 22-9 中的 Retry 按钮，也可以中断到这个地址。_RTC_StackFailure 函数的第二个参数是检测出问题的变量名称，错误报告对话框中显示的变量名就是这个参数传递过去的。

VC8 编译器新引入的/RTCs(s 代表 stack) 选项用来启用包括变量检查功能在内的栈检查机制。不过在 VC8 生成的项目文件中，为调试版本设置的默认选项使用的通常是/RTC1，RTC1 是 RTCsu 的等价形式，即同时启用 RTCs 和 RTCu。RTCu 的作用是对使用没有初始化过的变量发出警告错误。例如对于类似如下代码：

```
int n,j;
j=n;
```

如果启用了/RTCu 选项，编译器会为变量 n 定义一个标志变量，位于栈帧中调试版默认局部变量上方。标志变量为 bool 类型，但其前后也会加上屏障字段，因此每个标志要占用 12 字节的栈空间。标志变量用来记录变量是否已经被初始化，在函数入口处，编译器会插入指令将该标志赋值为 0。在对该变量赋值的地方，编译器会加入代码将该标志设为 1。在访问该变量的地方会加入如下代码：

```
00411503 cmp    byte ptr [ebp-12Dh],0
0041150A jne   VarCheck+59h (411519h) ;
0041150C push   offset (411668h) ;
00411511 call   @ILT+200(__RTC_UninitUse) (4110CDh) ;
```

其中第 1 行是判断初始化标志是否为 0，以避免当这个变量被赋了值后还发出警告。第 3 行是压入指向变量名的地址指针，而后是调用_RTC_UninitUse 函数，发出警告错误。_RTC_UninitUse 函数弹出的错误报告窗口与发现变量溢出的错误窗口外观一样，只是错误信息是类似如下的描述：

```
Run-Time Check Failure #3 - The variable 'n' is being used without being defined.
```

编译器会判断变量在函数中的使用情况，只为可能被“取值”的变量定义标志和加入检查逻辑，以在上面的代码为例，编译器不会为变量 j 定义标志变量。

因为变量检查功能(溢出检查和初始化检查)须要使用变量信息描述表和插入较多的额外指令和函数调用，对可执行文件的大小和运行性能都会产生影响，所以该功

能不能用于启用了优化选项的发布版本，当试图向发布版本的编译选项中加入这些选项时，会得到如下错误信息：

```
cl : Command line error D8016 : '/O2' and '/RTCu' command-line options are
incompatible
cl : Command line error D8016 : '/O2' and '/RTCs' command-line options are
incompatible
```

22.12 基于 Cookie 的安全检查

上一节介绍的变量检查功能可以检测变量溢出，而且能报告出发生溢出（或受损）的位置和变量名称。但是由于其开销较大，所以通常只用于调试版本。为了可以在发布版本中也能检测到缓冲区溢出，防止程序因缓冲区而受到攻击，VS2005 还支持基于 Cookie 的安全检查机制。

在计算机领域，Cookie 一词最早出现在网站开发中，是指网站的服务程序通过浏览器保存到客户端的少量数据，这些数据是二进制的，或者经过加密的，通常用来记录用户身份和登录情况等信息。后来人们用这个词泛指一方签发给另一个的认证或标志信息。

22.12.1 安全 Cookie 的产生、植入和检查

VC8 编译器在编译可能发生缓冲区溢出的函数时，会定义一个特别的局部变量，该局部变量会被分配在栈帧中所有其他局部变量和栈帧指针与函数返回地址之间（参见图 22-10），这个变量专门用来保存 Cookie，因此我们将其称为 Cookie 变量。Cookie 变量是一个 32 位的整数，它的值是从全局变量 `_security_cookie` 得到的。该全局变量是定义在 C 运行库中的，在 CRT 源代码目录（默认安装后的路径为 `c:\Program Files\Microsoft Visual Studio 8\VC\crt\src`）中的 `gs_cookie.c` 文件中我们可以看到其定义：

```
#define DEFAULT_SECURITY_COOKIE 0xBB40E64E
DECLSPEC_SELECTANY UINT_PTR __security_cookie = DEFAULT_SECURITY_COOKIE;
DECLSPEC_SELECTANY UINT_PTR __security_cookie_complement =
~(DEFAULT_SECURITY_COOKIE);
```

可见，这里已经为 `__security_cookie` 变量赋了默认值。在 VC8 编译器的启动函数中（如 `WinMainCRTStartup`）还会调用 `__security_init_cookie` 函数对该变量进行正式的初始化。在 `crt0.c` 文件中，我们可以看到启动函数 `_tmainCRTStartup` 的源代码：

```
int _tmainCRTStartup( void )
{
    /*
     * The /GS security Cookie must be initialized before any exception
     * handling targeting the current image is registered. No function
     * using exception handling can be called in the current image until
     * after __security_init_Cookie has been called.
     */
    __security_init_cookie(); // 初始化安全 Cookie
```

```

        return __tmainCRTStartup(); // 调用以前的启动函数
}

```

`_security_init_Cookie()` 函数的源代码包含在 `gs_support.c` 文件中。其中为 `Cookie` 取值的代码如（去除了针对 64 位的部分）清单 22-27 所示。

清单 22-27 初始化安全 Cookie

```

GetSystemTimeAsFileTime(&systime.ft_struct);
Cookie = systime.ft_struct.dwLowDateTime;
Cookie ^= systime.ft_struct.dwHighDateTime;

Cookie ^= GetCurrentProcessId();
Cookie ^= GetCurrentThreadId();
Cookie ^= GetTickCount();

QueryPerformanceCounter(&perfctr);
Cookie ^= perfctr.LowPart;
Cookie ^= perfctr.HighPart;
__security_cookie = Cookie;
__security_cookie_complement = ~Cookie;

```

为了达到较好的随机性，先取当前时间值作为基础，而后再与其他具有随机性的数据（进程 ID、线程 ID、系统的 TickCount 和性能计数器）进行逐位异或。

因为是在启动函数中初始化全局变量 `__security_Cookie`，所以在该进程以后的运行过程中，`Cookie` 值是保持不变的。如果使用 VC8 的集成环境进行调试，可以通过 `Immediate` 窗口来观察 `__security_cookie` 的值。

```
? __security_cookie
0x3f4b9fee
```

类似的，在 WinDBG 中可以使用 DD 命令来观察：

```
0:000> dd SecChk!__security_init_cookie 11
00401fd6 83ec8b55
```

编译器在为一个函数插入安全 `Cookie` 时，它还会与当时的 `EBP` 寄存器的值做一次异或操作，然后再保存到 `Cookie` 变量中，这些指令通常出现在函数的序言（Prolog）之后，清单 22-28 所示的反汇编代码显示了初始化栈帧和存放安全 `Cookie` 的典型过程。

清单 22-28 初始化栈帧和存放安全 Cookie 的典型过程

SecChk!VarCheck:		
00401040 55	push ebp	; 保存栈帧指针的旧值
00401041 8bec	mov ebp,esp	; 建立栈帧
00401043 83ec18	sub esp,0x18	; 为局部变量在栈上分配空间
00401046 a12c404000	mov eax,[SecChk!__security_Cookie (0040402c)]	;
0040104b 33c5	xor eax,ebp	; 与 EBP 异或
0040104d 8945fc	mov [ebp-0x4],eax	; 存入 Cookie 变量

与 `EBP` 异或有两个好处，一是可以进一步提高安全 `Cookie` 的随机性，尽可能使每个函数的安全 `Cookie` 都不同，另一个好处是还可以起到检查 `EBP` 值不被破坏的作用。因为在检查 `Cookie` 变量时，先要再与 `EBP` 寄存器的值做一次异或，如果 `EBP` 的值没有变化，那么两次异或后 `Cookie` 变量的值就应该恢复成全局变量 `__security_Cookie` 的值。

清单 22-29 显示了函数末尾检查 Cookie 变量的过程。

清单 22-29 检查安全 Cookie 的完好性

004010e5 8b4dfc	mov	ecx, [ebp-0x4] ; 将 Cookie 变量的值赋给 ECX 寄存器
004010e8 5f	pop	edi ; 弹出栈中保存的 EDI 寄存器（与 Cookie 无关）
004010e9 33cd	xor	ecx, ebp ; 将 Cookie 变量的值与 EBP 异或
004010eb 5e	pop	esi ; 弹出栈中保存的 ESI 寄存器（与 Cookie 无关）
004010ec e8b5000000	call	SecChk!__security_check_cookie (004011a6) ;
004010f1 8be5	mov	esp, ebp ; 恢复栈顶，释放分配的局部变量，即撤除栈帧
004010f3 5d	pop	ebp ; 将 EBP 寄存器的值恢复成上一栈帧的值
004010f4 c3	ret	; 返回

以上代码将 Cookie 变量的值与 EBP 异或后传递给 __security_check_cookie 函数，crt\src\intel\secchk.c 文件中包含了 __security_check_cookie 函数的源代码：

```
void __declspec(naked) __fastcall __security_check_Cookie(UINT_PTR Cookie)
{
    /* x86 version written in asm to preserve all regs */
    __asm {
        cmp ecx, __security_Cookie
        jne failure
        rep ret /* REP to avoid AMD branch prediction penalty */
failure:
        jmp __report_gsfailure
    }
}
```

为了降低对可执行文件大小和运行性能的影响，__security_check_cookie 函数是直接使用汇编语言编写的，整个函数只有 4 条汇编指令，没有使用任何变量和改变任何寄存器（标志寄存器除外）。因为使用的是快速调用协定，所以唯一的参数存放在 ECX 寄存器中，第一条指令比较 ECX 的值是否与全局变量记录的值相同，如果相同则说明被检查函数的 Cookie 变量（以及 EBP 寄存器）的值完好，于是返回。如果不同，则跳转到 __report_gsfailure 函数，报告错误。这样做的目的是单独处理异常情况，尽可能降低正常情况所需的开销。

22.12.2 报告安全检查失败

因为缓冲区溢出可能覆盖掉函数的本来返回地址，使函数返回到未知的地方，这会使程序失去控制。为了避免程序继续运行可能造成的不可预期的后果，安全检查失败被看作是致命的错误（fatal error）。一旦捕捉到这种错误，该应用程序就该被终止。但为了支持调试，在终止程序前，__report_gsfailure 函数会记录下错误发生时的重要信息，并提供 JIT 调试机会。

在 crt\src\gs_report.c 文件中，我们看到 __report_gsfailure 函数的完整源代码，可以将其工作过程分为如下几个步骤。

首先，__report_gsfailure 函数将当前的寄存器值保存到一个名为 GS_ContextRecord 的全局变量中。GS_ContextRecord 是一个 CONTEXT 类型的结

构，用来描述 CPU 的寄存器值和上下文信息。类似的，还有两个全局变量用来记录错误信息，分别是 GS_ExceptionPointers 和 GS_ExceptionRecord。

```
static EXCEPTION_RECORD      GS_ExceptionRecord;
static CONTEXT              GS_ContextRecord;
static const EXCEPTION_POINTERS GS_ExceptionPointers = {
    &GS_ExceptionRecord,
    &GS_ContextRecord
};
```

在将重要寄存器的值都保存到 GS_ContextRecord 后，`_report_gsfailure` 函数会设置 GS_ExceptionRecord 结构的字段：

```
GS_ExceptionRecord.ExceptionCode = STATUS_STACK_BUFFER_OVERRUN;
GS_ExceptionRecord.ExceptionFlags = EXCEPTION_NONCONTINUABLE;
```

可以把以上过程看作是模拟一个发生异常的现场，异常的代码是 STATUS_STACK_BUFFER_OVERRUN。这样做的目的是可以使用处理异常的方法来处理安全检查失败了。

接下来，如果定义了 _CRTBLD（编译 CRT 库时此符号是定义了的），会判断当前程序是否在被调试，并将这一信息记录到局部变量 DebuggerWasPresent 中。然后调用 _CRT_DEBUGGER_HOOK，给 CRT 的调试挂钩函数一次处理机会。

```
#if defined (_CRTBLD) && !defined (_SYSCRT)
    DebuggerWasPresent = IsDebuggerPresent();
    _CRT_DEBUGGER_HOOK(_CRT_DEBUGGER_GSFFAILURE);
#endif /* defined (_CRTBLD) && !defined (_SYSCRT) */
```

VC8 的默认 _CRT_DEBUGGER_HOOK 函数不采取任何动作。函数中唯一的一行只是为了防止该函数在编译发布版本时被优化器删除。

```
_declspec(noinline)
void __cdecl _CRT_DEBUGGER_HOOK(int _Reserved)
{
    /* assign 0 to _debugger_hook_dummy so that the function is not folded in retail */
    (_Reserved);
    _debugger_hook_dummy = 0;
}
```

在完成以上工作后，`_report_gsfailure` 函数会通过调用 `UnhandledExceptionFilter` 函数，以显示应用程序错误对话框。我们在第 12 章详细讨论过 `UnhandledExceptionFilter` 函数和应用程序错误对话框。在调用 `UnhandledExceptionFilter` 函数前，`_report_gsfailure` 函数首先调用 `SetUnhandledExceptionFilter` 清除顶层过滤函数。

```
SetUnhandledExceptionFilter(NULL);
UnhandledExceptionFilter((EXCEPTION_POINTERS *)&GS_ExceptionPointers);
```

如果用户在应用程序错误对话框中选择了关闭程序，那么 `UnhandledExceptionFilter` 函数就会调用 `NtTerminateProcess` 终止进程，也就是再也不会返回到 `_report_gsfailure` 函数中。

如果用户选择了调试，那么 `UnhandledExceptionFilter` 函数会启动注册表中登记的 JIT 调试器。启动 JIT 调试器后，`UnhandledExceptionFilter` 函数开始等待 JIT

调试器设置通过命令行传递给调试器的调试器就绪事件 (Event)。等待成功后, UnhandledExceptionFilter 函数返回 EXCEPTION_CONTINUE_SEARCH。这使得 CPU 又返回继续执行 __report_gsfailure 函数, 即如下代码:

```
if (!DebuggerWasPresent)
{
    _CRT_DEBUGGER_HOOK(_CRT_DEBUGGER_GSFALLURE);
}
TerminateProcess(GetCurrentProcess(), STATUS_STACK_BUFFER_OVERRUN);
```

这段代码的含义是如果 DebuggerWasPresent 标志为假, 则再次调用 _CRT_DEBUGGER_HOOK 函数, 并让 _CRT_DEBUGGER_HOOK 函数触发调试器让调试器将该进程中断下来。因为继续执行下面就是 TerminateProcess, 但 _CRT_DEBUGGER_HOOK 函数什么都不做就返回了, 所以如果不及时中断到调试器还是会继续执行 TerminateProcess, 导致进程终止。

如果使用 WinDBG 做 JIT 调试器, 因为 WinDBG 不会立即中断被调试进程, 而是让目标进程继续执行, 这会使得未处理的异常被再次分发 (第二轮处理机会)。于是 WinDBG 得到第二轮处理机会, 将进程中断到调试器。

如果使用 VC8 做 JIT 调试器, 因为 VC8 总是询问用户是否中断进程 (图 22-12), 因此如果选择中断, 可以使进程停止在 _CRT_DEBUGGER_HOOK() 函数的位置。

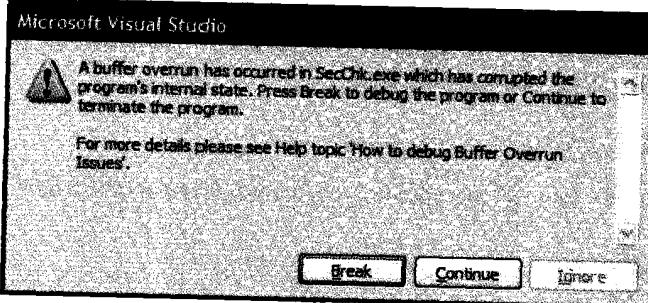


图 22-12 使用 VS2005 作为 JIT 调试器调试安全检查失败

在 VC8 中, 缓冲区安全检查是默认启用的, 但也可以在编译选项加入 /GS (S 大写) 显式的启用安全检查。如果要禁止该功能, 则应在编译选项中加入 /GS-。

22.12.3 编写安全的代码

编译器为降低因为缓冲区溢出而导致的安全问题做了很多努力, 变量检查功能可以帮助我们及时发现问题, 自动的安全 Cookie 检查可以及时终止出问题的进程, 防止进一步的危害。然而比这些措施更好的解决方案是在设计和编写代码时合理地使用缓冲区, 避免在代码中埋下缓冲区溢出隐患, 特别应该注意以下几点。

- 在向缓冲区写入数据前应该检查源和目标的长度, 防止向小的缓冲区写入超出其

容量的数据。

- 在设计函数时，如果需要使用缓冲区作为参数来回传数据时，应该设计一个参数供调用者指定缓冲区长度。以 GetWindowText (HWND hWnd, LPTSTR lpString, int nMaxCount) API 为例，第三个参数就是用来指定第二个参数所指向的缓冲区可以容纳的最多字符数。
- 尽可能避免使用不安全的库函数或 API，某些早期设计的 C 运行库中的函数为了性能考虑缺少必要的安全检查，比如 sprintf(char *buffer, const char *format [, argument] ...) 函数和 strcpy(char *strDestination, const char *strSource) 函数都不会检查目标缓冲区的长度，很容易发生溢出。最好避免使用这些函数，如果一定要使用，最好在调用这些函数前后做必要的检查。VS2005 的 C 运行库为以上存在安全问题的函数提供了新的安全版本，这些新的函数在原来的函数名后加上了_s 后缀，如 sprintf_s(char *buffer, size_t sizeOfBuffer, const char *format [, argument] ...) 和 strcpy_s(char *strDestination, size_t sizeInBytes, const char *strSource)。新的函数与原来的函数相比除了加上用来指定缓冲区长度的参数外，其他参数和用法都保持了与以前版本的兼容性。

使用我们在第 20 章介绍的标准标注语言（Standard Annotation Language, SAL）可以帮助编译器和其他分析工具发现代码的安全问题，提高代码的安全性。

举例来说，尽管函数 strcpy 设计中考虑到了使用第三个参数 count 来指定 strDest 缓冲区的大小以防止 strSource 长度大于 strDest 缓冲区时发生溢出。但如果调用者错误地传递了 count 参数，那么还是可能发生缓冲区溢出，例如下面的代码：

```
char szBuffer[20];
strcpy(szBuffer, szInput, 100);
```

既然在 strcpy 的函数声明中就假定了第三个参数应该是第一个参数的内存空间长度，那么编译器是不是可以检测出上面的调用代码存在问题呢？答案是肯定的，如果使用 SAL 技术，并且在编译选项中指定/analyze 进行编译，那么就会得到如下警告信息：

```
c:\xx\xx.cpp(xx) : warning C6203: Buffer overrun for non-stack buffer 'szBuffer'
in call to 'strcpy': length '100' exceeds buffer size '20'
```

编译器是利用函数声明中的 SAL 信息发现以上问题的。在 VC8 的 C 运行库头文件中，所有函数原型都加上了 SAL 标注，如带有 SAL 标注后的 strcpy 函数声明为：

```
_CRTIMP char * __cdecl strcpy(__out_ecount (_MaxCount) char * _Dst,
__in_z const char * _Src,__in size_t _MaxCount);
```

其中，__out_ecount (_MaxCount)、__in_z 和 __in 都是 SAL 标注符号，__out_ecount (_MaxCount) 的含义是其后的参数是输出类型的，其长度为 _MaxCount，这个函数会向这个参数写入数据，因此它不能为 NULL；__in_z 表示其后的参数是输入类型的以 0 结尾的字符串，该函数不会改变它的值；__in 表示其后的

参数是输入类型的。

22.13 本章总结

本章围绕着栈和函数调用这两个相互联系的主题比较深入地讨论了编译器在这两方面的调试支持。全章可以分为前后两大部分，前半部分（22.1~22.7 节）讨论了用户态栈和内核态栈（22.1 节）、栈的创建过程（22.2 节）、CALL 和 RET 指令（22.3 节）、局部变量和栈帧（22.4 节）、栈指针省略（22.5 节）、栈指针检查（22.6 节）、函数调用协定（22.7 节）等基本内容。后半部分（22.8~22.12 节）讨论了栈空间的分配和自动增长机制（22.8 节）、栈下溢（22.9 节）、缓冲区溢出（20.10 节），以及编译器提供的检查措施（22.11 节和 22.12 节）。

表 22-2 归纳了编译器在产生栈和函数调用有关的代码时所加入的支持软件调试的机制，以及这些机制在调试版本与发布版本中的可用性。

表 22-2

操作	操作的意义	可用性	章节
建立栈指针	产生栈回溯信息的基础	视优化选项而定	22.4
使用 0xCC 填充局部变量区	如果 CPU 因为缓冲区溢出等原因意外执行到这些区域，可以立即中断	仅应用于调试版本	22.11
为局部变量分配和建立屏障字段，并在运行期进行变量检查（RTCu）	检测缓冲区溢出，可以报告出发生溢出的局部变量名称	仅应用于调试版本	22.11
栈指针检查（RTCs）	检测不匹配的函数调用，及时发现栈指针异常	仅应用于调试版本	22.6
对可能发生缓冲区溢出的函数自动加入基于安全 Cookie 的检查机制（GS Switch）	及时发现缓冲区溢出和栈受损	调试版本和发布版本	22.12

参考文献

1. Grace Murray Hopper. Automatic Coding for Digital Computers
2. Agner Fog. Calling conventions for different C++ compilers and operating systems
3. Microsoft Minimizes Threat of Buffer Overruns, Builds Trustworthy Applications. Microsoft Corporation
4. Michael Howard. Writing Secure Code. Microsoft Press, 2002

堆和堆检查

内存是软件工作的舞台，当用户启动一个程序时，系统会将程序文件从外部存储器（硬盘等）加载到内存中。当程序工作时，须要使用内存空间来放置代码和数据。在使用一段内存之前，程序须要以某种方式（库函数或 API 等）发出申请，接收到申请的一方（C 运行库或各种内存管理器）根据申请者的要求从可用（空闲）空间中寻找满足要求的内存区域分配给申请者。当程序不再需要该空间时应该通过与申请方式相对应的方法归还该空间，即释放。

在软件开发实践中，尤其是在较大型的软件项目中，合理地分配和释放内存是非常重要的，由于内存使用不当而导致的各种问题经常成为软件项目的严重障碍。提高内存的使用效率，降低内存分配和释放过程的复杂性一直是软件产业中的永恒话题。Java 和 .NET 语言的一个共同优势就是可以自动回收不再需要的内存，使程序员可以不用编写释放内存的代码。上一章介绍的通过栈来分配局部变量也可以看作是简化内存使用的一种方法。

堆（Heap）是组织内存的另一种重要方法，是程序在运行期动态申请内存空间的主要途径。与栈空间是由编译器产生的代码自动分配和释放不同，堆上的空间需要程序员自己编写代码来申请（如 HeapAlloc）和释放（如 HeapFree），而且分配和释放操作应该严格匹配，忘记释放或多次释放都是不正确的。

与栈上的缓冲区溢出类似，如果向堆上的缓冲区写入超过其大小的内容，也会因为溢出而破坏堆上的其他内容，可能导致严重的问题，包括程序崩溃。

为了帮助发现堆使用方面的问题，堆管理器、编译器和软件类库（如 MFC、.NET Framework 等）提供了很多检查和辅助调试机制。比如，Win32 堆支持参数检查、溢出检查及释放检查等功能。VC 编译器设计了专门的调试堆并提供了一系列用来追踪和检查堆使用情况的函数，在编译调试版本的可执行文件时，可以使用这些调试支持来解决内存泄漏等问题。

本章将先介绍堆的基本概念（23.1 节），然后分两大部分分别介绍 Win32 堆和 CRT

堆，前半部分（23.2~23.10 节）将介绍 Win32 堆的创建方式（23.2 节）、使用方法（23.3 节）、内部结构（23.4 节）、低碎片堆（23.5 节）和调试支持（23.6~23.10 节）。后半部分（23.11~23.15 节）将介绍 CRT 堆的概况（23.11 节）、堆块结构（23.12 节）和它的调试支持（23.13~23.15 节）。

23.1 理解堆

栈是分配局部变量和存储函数调用参数及返回位置的主要场所，系统在创建每个线程时会自动为其创建栈。对于 C/C++ 这样的编程语言，编译器在编译阶段会生成合适的代码来从栈上分配和释放空间，不需要程序员编写任何额外的代码，出于这个原因栈得到了“自动内存”这样一个美名。

不过从栈上分配内存也有不足之处，首先，栈空间（尤其是内核态栈）的容量是相对较小的，为了防止栈溢出，不适合在栈上分配特别大的内存区。其次，由于栈帧通常是随着函数的调用和返回而创建和消除的，因此分配在栈上的变量只是在函数内有效，这使栈只适合分配局部变量，不适合分配需要较长生存期的全局变量和对象。第三，尽管也可以使用 `_alloca()` 这样的函数来从栈上分配可变长度的缓冲区，但是这样做会给异常处理（EH）带来麻烦，因此，栈也不适合分配运行期才能决定大小（动态大小）的缓冲区。

堆（Heap）克服了栈的以上局限，是程序申请和使用内存空间的另一种重要途径。应用程序通过内存分配函数（如 `malloc` 或 `HeapAlloc`）或 `new` 操作符获得的内存空间都来自于堆。

从操作系统的角度来看，堆是系统的内存管理功能向应用软件提供服务的一种方式。通过堆，内存管理器（Memory Manager）将一块较大的内存空间委托给堆管理器（Heap Manager）来管理。堆管理器将大块的内存分割成不同大小的很多个小块来满足应用程序的需要。应用程序的内存需求通常是频繁而且零散的，如果把这些请求都直接传递给位于内核中的内存管理器，那么必然会影响系统的性能。有了堆管理器，内存管理器就只需要处理大规模的分配请求。这样做不仅可以减轻内存管理器的负担，也可以大大缩短应用程序申请内存分配所需的时间，提高程序的运行速度。从这个意义上来说，堆管理器就好像是经营内存零售业务的中间商，它从内存管理器那里批发大块的内存，然后零售给应用程序的各个模块。图 23-1 画出了 Windows 系统中实现的这种多级内存分配体系。

在图 23-1 中，我们使用不同类型的箭头来代表不同层次的内存分配方法。具体来说，用户态的代码应该调用虚拟内存分配 API 来从内存管理器分配内存。虚拟内存 API 包括 `VirtualAlloc`、`VirtualAllocEx`、`VirtualFree`、`VirtualFreeEx`、`VirtualLock`、`VirtualUnlock`、`VirtualProtect`、`VirtualQuery` 等。内核态的代码可以调用以上 API 所对应的内核函数，

比如 `NtAllocateVirtualMemory`、`NtProtectVirtualMemory` 等。

为了满足内核空间中的驱动程序等内核态代码的内存分配需要，Windows 的内核模块中实现了一系列函数来提供内存“零售”服务，为了与用户空间的堆管理器相区别，我们把这些函数统称为池管理器（Pool Manager）。池管理器公开了一组驱动程序接口（DDI）以向外提供服务，包括 `ExAllocatePool`、`ExAllocatePoolWithTag`、`ExAllocatePoolWithTagPriority`、`ExAllocatePoolWithTagQuota`、`ExFreePool`、`ExFreePoolWithTag` 等。

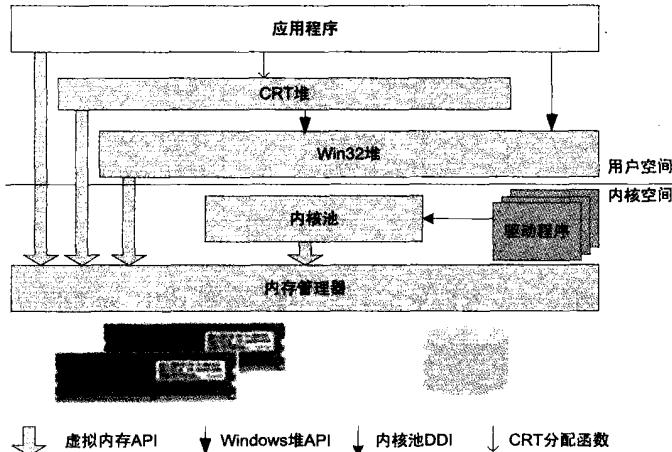


图 23-1 多级的内存分配体系

与内核模块中的池管理器类似，在 `NTDLL.DLL` 中实现了一个通用的堆管理器，目的为用户态的应用程序提供内存服务，通常被称为 Win32 堆管理器。SDK 中公开了一组 API 来访问 Win32 堆管理器的功能，比如 `HeapAlloc`、`HeapFree` 等。本章后文将详细介绍 Win32 堆管理器。

为了支持 C 的内存分配函数和 C++ 的内存分配运算符（`new` 和 `delete`）（以下统称 CRT 内存分配函数），编译器的 C 运行库会创建一个专门的堆供这些函数所使用，通常称为 CRT 堆。根据分配堆块的方式不同，CRT 堆有三种工作模式：SBH（Small Block Heap）模式、旧 SBH（Old SBH）模式和系统模式（System Heap），当创建 CRT 堆时，会选择其中的一种。对于前两种模式，CRT 堆会使用虚拟内存分配 API 从内存管理器批发大的内存块过来，然后分割成小的堆块满足应用程序的需要。对于系统模式，CRT 堆只是把堆块分配请求转发给它所基于的 Win32 堆，因此处于系统模式的 CRT 堆只是对 Win32 堆的一种简单封装，在原来的基础上又增加了一些附加的功能。我们将在第 23.10~23.12 节详细介绍 CRT 堆和它的调试支持。

应用程序开发商也可以实现自己的堆管理器，只要通过虚拟内存 API 从内存管理器“批发”内存块过来后提供给自己的客户代码使用，但这超出了本书的讨论范围。

从实现角度来讲，内核态的池管理器和用户态的 Win32 堆管理器是共享一套基础代码的，它们以运行时库（Run Time Library）的形式分别存在于 NTOSKRNL.EXE 和 NTDLL.DLL 模块中。使用 WinDBG 的检查符号命令分别列出 NTDLL.DLL 和 NTOSKRNL.EXE 模块中包含 Heap 单词的符号，便可以看到很多相同的函数，表 23-1 列出了其中的一部分。

表 23-1 NTDLL.DLL 和 NTOSKRNL.EXE 中的堆管理器函数

	用户态的堆管理函数	内核态的堆管理函数	功能
位置	NTDLL.DLL	NTOSKRNL.EXE	
函数列表	ntdll!RtlAllocateHeap ntdll!RtlCreateHeap ntdll!RtlDestroyHeap ntdll!RtlExtendHeap ntdll!RtlFreeHeap ntdll!RtlSizeHeap ntdll!RtlZeroHeap	nt!RtlAllocateHeap nt!RtlCreateHeap nt!RtlDestroyHeap nt!RtlExtendHeap nt!RtlFreeHeap nt!RtlSizeHeap nt!RtlZeroHeap	从堆上分配内存 创建堆 销毁堆 扩展堆大小 释放堆块 取堆块大小 清零或填充空闲堆块

从表 23-1 中可以看出，用户态和内核态中负责管理和操作堆的基本函数都是相同的，因此接下来我们将集中讨论用户态的 Win32 堆。如不特别指出，后面内容中的堆就是指 Win32 堆。

23.2 堆的创建和销毁

本节将介绍 Windows 进程创建、使用和维护堆的方法，我们从系统为每个进程所自动创建的默认堆谈起。

23.2.1 进程的默认堆

Windows 系统在创建一个新的进程时，在加载器函数执行进程的用户态初始化阶段，会调用 RtlCreateHeap 函数为新的进程创建第一个堆，称为进程的默认堆，有时简称进程堆（Process Heap）。清单 23-1 中的栈回溯记录包含了 LdrpInitializeProcess 函数调用 NTDLL 中的堆创建函数的过程。此时，新进程中还只有应用程序的 EXE 模块和 NTDLL.DLL 模块，尚未加载其他任何模块，EXE 模块中的用户代码也还没有执行。

清单 23-1 为新进程创建第一个堆

```
0:000> kb
ChildEBP RetAddr  Args to Child
0012fb04 7c921e0a 00000002 00000000 00100000 ntdll!RtlCreateHeap
0012fc94 7c921639 0012fd30 7c900000 0012fce0 ntdll!LdrpInitializeProcess+0x4b7
0012fd1c 7c90eac7 0012fd30 7c900000 00000000 ntdll!_LdrpInitialize+0x183
00000000 00000000 00000000 00000000 ntdll!KiUserApcDispatcher+0x7
```

创建好的堆句柄会被保存到进程环境块（PEB）的 ProcessHeap 字段中。可以使用 WinDBG 的 dt 命令来观察 PEB 结构，以下是与进程堆有关的几个字段：

```
0:000> dt _PEB 7ffd5000
+0x018 ProcessHeap          : 0x00150000 //进程(默认)堆句柄
+0x078 HeapSegmentReserve  : 0x1000000 //堆的默认保留大小,字节数,1MB
+0x07c HeapSegmentCommit   : 0x2000    //堆的默认提交大小,8KB
```

其中 `ProcessHeap` 即进程堆的句柄, `HeapSegmentReserve` 是进程堆的保留大小, 其默认值为 1MB (即 0x100000), `HeapSegmentCommit` 是进程堆的初始提交大小, 其默认值为两个内存页大小, x86 系统中普通内存页的大小为 4KB, 因此是 0x2000, 即 8KB。可以通过链接选项/`HEAP` 来改变进程堆的保留大小和初始提交大小。

```
/HEAP:reserve[,commit]
```

使用 `GetProcessHeap API` 可以取得当前进程的进程堆句柄:

```
HANDLE GetProcessHeap(void);
```

事实上, `GetProcessHeap` 函数只是简单地找到 PEB 结构, 然后读出 `ProcessHeap` 字段的值。得到进程堆的句柄后, 就可以使用 `HeapAlloc API` 从这个堆上申请空间, 如:

```
CStructXXX * pStruct = HeapAlloc(GetProcessHeap(), 0, sizeof(CStructXXX));
```

23.2.2 创建私有堆

除了系统为每个进程创建的默认堆, 应用程序也可以通过调用 `HeapCreate API` 创建其他堆, 这样的堆只能被发起调用的进程自己访问, 通常被称为私有堆 (`Private Heap`)。

```
HANDLE HeapCreate( DWORD f1Options, SIZE_T dwInitialSize, SIZE_T dwMaximumSize);
```

其中 `dwInitialSize` 用来指定堆的初始提交大小, `dwMaximumSize` 用来指定堆空间的最大值 (保留大小), 如果为 0, 则创建的堆可以自动增长。尽管可以使用任意大小的整数作为 `dwInitialSize` 和 `dwMaximumSize` 参数, 但是系统会自动将其取整为大于该值的临近页边界 (即页大小的整数倍)。`f1Options` 参数可以为如下标志中的零个或多个。

`HEAP_GENERATE_EXCEPTIONS(0x00000004)`, 使用异常来报告失败情况, 如果没有该标志则使用返回 `NUL` 报告错误。

`HEAP_CREATE_ENABLE_EXECUTE(0x00040000)`, 允许执行堆中内存块上的代码。

`HEAP_NO_SERIALIZE(0x00000001)`, 当堆函数访问这个堆时, 不需要进行串行化控制 (加锁)。指定这一标志可以提高堆操作函数的速度, 但应该在确保不会有多个线程操作同一个堆时才这样做, 通常是在将某个堆分给某个线程专用时这么做。也可以在每次调用堆函数时指定该标志, 告诉堆管理器不需要对那次调用进行串行化控制。

`HeapCreate` 内部主要是调用 `RtlCreateHeap` 函数, 因此私有堆与默认堆并没有本质的差异, 只是创建的用途不同。`RtlCreateHeap` 内部会调用 `ZwAllocateVirtualMemory` 系统服务从内存管理器申请内存空间, 初始化用于维护堆的数据结构 (第 23.4 节详细介绍), 最后将堆句柄记录到进程的 PEB 结构中, 确切说是我们将要介绍的 PEB 结构的堆列表中。

23.2.3 堆列表

每个进程的 PEB 结构以列表的形式记录了当前进程的所有堆句柄，包括进程的默认堆。具体来说，PEB 中有三个字段用于记录这些句柄。NumberOfHeaps 字段用来记录堆的总数，ProcessHeaps 字段用来记录每个堆的句柄，它是一个数组，这个数组可以容纳的句柄数记录在 MaximumNumberOfHeaps 字段中。如果 NumberOfHeaps 达到 MaximumNumberOfHeaps，那么堆管理器会增大 MaximumNumberOfHeaps 值，并重新分配 ProcessHeaps 数组。

```
0:000> dt _peb 7ffd5000          //显示进程的 PEB 结构
ntdll!_PEB
+0x018 ProcessHeap      : 0x00150000 //进程默认堆句柄
+0x088 NumberOfHeaps   : 3           //进程中堆的总数
+0x08c MaximumNumberOfHeaps : 0x10    //ProcessHeaps 数组的目前大小
+0x090 ProcessHeaps     : 0x7c97de80 -> 0x00150000 //存放堆句柄的数组
```

使用 dd 命令观察 ProcessHeaps 字段的值便可以看到当前进程的所有堆指针。

```
0:000> dd 7c97de80 13
7c97de80 00150000 00250000 00260000
```

使用 WinDBG 的!heap 命令可以列出当前进程的所有堆：

```
0:000> !heap -h
Index Address Name      Debugging options enabled
1: 00150000 Segment at 00150000 to 00250000 (00003000 bytes committed)
2: 00250000 Segment at 00250000 to 00260000 (00006000 bytes committed)
3: 00260000 Segment at 00260000 to 00270000 (00003000 bytes committed)
```

可以看到!heap 命令显示的结果与我们用 dd 命令直接观察到的是一致的。另外，堆句柄的值实际上就是这个堆的起始地址。

23.2.4 销毁堆

应用程序可以调用 HeapDestroy API 来销毁进程的私有堆。HeapDestroy 内部主要是调用 NTDLL 中的 RtlDestroyHeap 函数。后者会从 PEB 的堆列表中将要销毁的堆句柄移除，然后调用 NtFreeVirtualMemory 向内存管理器归还内存。

应用程序不须要也不应该销毁进程的默认堆，因为进程内的很多系统函数会使用这个堆。不必担心这会导致内存泄漏，因为当进程退出和销毁进程对象时，系统会两次调用内存管理器的 MmCleanProcessAddressSpace 函数来释放清理进程的内存空间。具体来说，第一次是在退出进程中执行的，当 NtTerminateProcess 函数调用 PspExitThread 退出线程时，如果退出的是最后一个线程，那么 PspExitThread 会调用 MmCleanProcessAddressSpace，后者会先删除进程用户空间中的文件映射和虚拟地址，释放虚拟地址描述符，然后删除进程空间的系统部分，最后删除进程的页表和页目录设施。这是在退出进程上下文中执行的最后几项任务之一。

而后，当系统的工作线程删除进程对象时会再次调用 MmCleanProcessAddressSpace 函数，清单 23-2 所示的栈回溯序列显示了这次调用的经过。

清单 23-2 系统在删除进程对象时会再一次清理进程的内存空间

```

kd> kn
# ChildEBP RetAddr
00 f86c0c58 805922ba nt!MmCleanProcessAddressSpace //清理进程的地址空间
01 f86c0c74 805921d1 nt!PspExitProcess+0x1fe //进程退出函数
02 f86c0ca8 8057e49d nt!PspProcessDelete+0x11a //删除进程对象
03 f86c0cc4 804ecc07 nt!ObpRemoveObjectRoutine+0xdd //调用对象的删除函数
04 f86c0ce8 80581110 nt!ObfDereferenceObject+0x5d //减少引用次数
05 f86c0d00 8058132d nt!ObpCloseHandleTableEntry+0x153 //处理句柄表的一个表项
06 f86c0d48 8058136e nt!ObpCloseHandle+0x85 //对象管理器的关闭句柄函数
07 f86c0d58 804da140 nt!NtClose+0x19 //关闭句柄内核服务
08 f86c0d58 7ffe0304 nt!KiSystemService+0xc4 //分发系统服务
09 0051ff40 00000000 SharedUserData!SystemCallStub+0x4 //调用系统服务关闭进程句柄

```

因此，应用程序退出时没有必要一一销毁每个私有堆，系统在清理进程空间时会自动释放。

23.3 分配和释放堆块

当应用程序调用堆管理器的分配函数向堆管理器申请内存时，堆管理器会从自己维护的内存区中分割出一个满足用户指定大小的内存块，然后把这个块中允许用户访问部分的起始地址返回给应用程序，堆管理器把这样的块叫做一个 Chunk，本书将其译为堆块。应用程序用完一个堆块后，应该调用堆管理器的释放函数归还堆块。本节将介绍分配和释放堆块的典型方法，并讨论这些方法的区别和联系。

23.3.1 HeapAlloc

在 Windows 系统中，从堆中分配空间的最直接方法就是调用 `HeapAlloc` API，其原型如下：

```
LPVOID HeapAlloc( HANDLE hHeap, DWORD dwFlags, SIZE_T dwBytes);
```

其中 `hHeap` 是要从中分配空间的内存堆的句柄，`dwBytes` 为所需内存块的字节数，`dwFlags` 可以为如下标志位的组合。

`HEAP_GENERATE_EXCEPTIONS(0x00000004)`，使用异常来报告失败情况，如果没有此标志，则使用 `NULL` 返回值来报告错误。异常的代码可能为 `STATUS_ACCESS_VIOLATION` 或 `STATUS_NO_MEMORY`。

`HEAP_ZERO_MEMORY(0x00000008)`，将所分配的内存区初始化为 0。

`HEAP_NO_SERIALIZE(0x00000001)`，不需要对该次分配实施串行化控制（加锁）。如果希望对该堆的所有分配调用都不需要串行化控制，那么可以在创建堆时指定 `HEAP_NO_SERIALIZE` 选项。对于进程堆，调用 `HeapAlloc` 时永远不应该指定 `HEAP_NO_SERIALIZE` 标志，因为系统的代码（比如 `Ctrl+C` 处理函数）可能随时会调用堆函数访问进程堆。

如果 HeapAlloc 函数成功，那么它会返回指向所分配内存块的指针，也就是程序可以使用的内存块的起始地址。如果失败，而且 dwFlags 中没有包含 HEAP_GENERATE_EXCEPTIONS 标志，那么返回 NULL，如果包含，则会产生异常。

观察调用 HeapAlloc API 的代码所对应的汇编指令，我们可以看到实际上调用的是 NTDLL 中的 RtlAllocateHeap 函数：

```
0040129e ff1510704000 call dword ptr [HiHeap!_imp__HeapAlloc (00407010)]
ds:0023:00407010={ntdll!RtlAllocateHeap (7c9105d4)}
```

因此，HeapAlloc API 只不过是 RtlAllocateHeap 的一个别名，二者实际上是等价的。

可以使用 HeapReAlloc API 来改变一个从堆中分配的内存块的大小，其原型如下：

```
LPVOID HeapReAlloc(HANDLE hHeap, DWORD dwFlags, LPVOID lpMem, SIZE_T dwBytes);
```

其中 lpMem 指向以前分配的内存块，dwBytes 为重新分配的大小，dwFlags 除了可以包含上面讲的 HeapAlloc API 的三种标志之外，还可以指定 HEAP_REALLOC_IN_PLACE_ONLY (0x00000010)，含义是不改变原来内存块的位置。

23.3.2 CRT 分配函数

也可使用标准 C 所定义的内存分配函数来从堆中分配内存，如 malloc 和 calloc。

```
void *malloc( size_t size );
void *calloc( size_t num, size_t size );
```

这两个函数会从我们在第 23.1 节提到的 CRT 堆中分配内存。编译器的运行时库在初始化阶段会创建 CRT 堆，创建前会选择三种模式之一，大多时候选择的都是系统分配模式，在这种模式下，CRT 堆是建立在 Win32 堆之上的，所以以上内存分配函数最终也是调用 HeapAlloc 函数来分配内存块的。清单 23-3 所示的栈回溯显示了在本节的示例程序 HiHeap 中 malloc 函数间接调用 HeapAlloc（即 RtlAllocateHeap）的过程。

清单 23-3 malloc 函数的工作过程（VC6 发布版本）

0:000> k	ChildEBP RetAddr	
0012ff10	004012a4 ntdll!RtlAllocateHeap+0x1a	//NTDLL 中的 Win32 堆分配函数
0012ff24	00401216 HiHeap!_heap_alloc+0x72	//CRT 堆的分配函数
0012ff2c	00401203 HiHeap!_nh_malloc+0x10	//支持 new handler 的分配函数
0012ff38	004010fa HiHeap!malloc+0xf	//标准 C 函数
0012ff40	0040116c HiHeap!TestMalloc+0xa	//测试 malloc 函数
0012ff80	00401393 HiHeap!main+0x3c	//用户入口
0012ffc0	7c816d4f HiHeap!mainCRTStartup+0xb4	//CRT 的入口函数
0012fff0	00000000 kernel32!BaseProcessStart+0x23	//系统的进程启动函数

其中，_heap_alloc 是 CRT 堆块的分配函数，它内部调用 Win32 堆的分配函数实际分配堆块，然后封装成 CRT 格式的堆块，_nh_malloc 是支持分配处理器（new handler）的中间函数，它内部除了调用_heap_alloc 外，会调用_callnewh 来检查是否有注册的分配处理器，如果有，则调用。

在 C++ 语言中，经常使用 new 操作符来创建对象和分配内存，例如：

```
char * lpsz=new char[2048];
```

跟踪以上语句的执行过程，我们可以看到，事实上，编译器产生的代码仍是调用 CRT 堆的内存分配函数，后者再调用 RtlAllocateHeap 来从堆上分配内存，清单 23-4 显示了其调用过程。

清单 23-4 new 操作符间接调用 HeapAlloc（即 RtlAllocateHeap）的过程（VC6 发布版本）

0:000> k	ChildEBP RetAddr	
0012ff14 004012a4	ntdll!RtlAllocateHeap	// NTDLL 中的 Win32 堆分配函数
0012ff28 00401216	HiHeap!_heap_alloc+0x72	// CRT 堆的分配函数
0012ff30 004011f1	HiHeap!_nh_malloc+0x10	// 支持 new handler 的分配函数
0012ff3c 004010da	HiHeap!operator new+0xb	// new 运算符
0012ff44 00401165	HiHeap!TestNew+0xa	// 测试 new 运算符
0012ff80 00401393	HiHeap!main+0x35	// 用户入口
0012ffc0 7c816d4f	HiHeap!mainCRTStartup+0xb4	// CRT 的入口函数
0012fff0 00000000	kernel32!BaseProcessStart+0x23	// 系统的进程启动函数

比较清单 23-3 和清单 23-4，可以看到，无论源程序中使用的是 malloc 函数还是 new 运算符，编译器所产生的代码都是通过 CRT 堆的分配函数间接调用 RtlAllocateHeap 函数来从堆上分配内存的。那么，为什么要通过 C 运行库的中间函数来间接调用系统的堆分配函数呢？首先是为了降低编译器与操作系统间的耦合度，另一个好处是可以借助这些中间函数加入内存检查功能来辅助调试。

清单 23-3 和清单 23-4 显示的都是 VC6 编译器产生的代码（发布版本），在 VC2005（VC8）所产生的发布版本代码中，new 操作符调用的是 malloc 函数，malloc 函数直接调用 RtlAllocateHeap。但在调试版本中，仍会调用 _nh_malloc_dbg 和 _heap_alloc_dbg 等中间函数，我们将在介绍 CRT 堆时详细讨论这些中间函数的作用。

23.3.3 释放从堆中分配的内存

应该调用 HeapFree API 来释放使用 HeapAlloc API 所分配的内存。HeapFree API 的原型为：

```
BOOL HeapFree( HANDLE hHeap, DWORD dwFlags, LPVOID lpMem);
```

其中 dwFlags 可以包含 HEAP_NO_SERIALIZE (0x00000001) 标志，用来告诉堆管理器不需要对该次调用进行防止并发的串行化操作以提高速度。lpMem 用来指定要释放内存块的地址，也就是使用 HeapAlloc 函数申请内存时所得到的返回值。

与 HeapAlloc 被链接到 NTDLL.DLL 中的 RtlAllocateHeap 类似，HeapFree 函数实际上总是被链接到 NTDLL.DLL 中的 RtlFreeHeap 函数。

通过 malloc 和 calloc 分配的内存应该使用 free() 函数来释放，通过 new 操作符分配的内存应该使用 delete 操作符来释放，但二者内部最终都是调用 HeapFree

(RtlFreeHeap) 函数来真正从堆上释放指定的内存块。在发布版本中，为了提高执行效率，`delete` 操作符通常被编译为跳转，而不是函数调用，即：

```
004010f8 e9c5000000      jmp      HiHeap!operator delete (004011c2)
```

而 `delete` 操作符只是简单地跳转到 `free` 函数：

```
HiHeap!operator delete:
004011c2 e959010000      jmp      HiHeap!free (00401320)
```

在 `free` 函数中，它会调用 `HeapAlloc`，即 `RtlFreeHeap`，如清单 23-5 所示。

清单 23-5 `delete` 操作符释放从堆上申请的内存块的执行过程（VC8 发布版本）

0:000> k	ChildEBP RetAddr	
0012fef4 0040138e	ntdll!RtlFreeHeap	//Win32 堆的释放函数
0012ff34 00401184	HiHeap!free+0x6e	//C 标准函数
0012ff70 00401772	HiHeap!main+0x34	//程序的用户代码入口
0012ffc0 7c816d4f	HiHeap!_tmainCRTStartup+0x15f	//CRT 代码的入口函数
0012fff0 00000000	kernel32!BaseProcessStart+0x23	//系统的进程启动函数

在调试版本中，为了支持内存检查功能，其过程会复杂得多，我们将在以后各节中逐步讨论。

23.3.4 GlobalAlloc 和 LocalAlloc

16 位的 Windows (Windows 3.x) 支持所谓的全局堆和局部堆，简单来说，局部堆是进程内的，全局堆是系统提供给所有进程来共享使用的，从全局堆和局部堆分配内存的 API 是不同的，全局堆应该使用 `GlobalAlloc` 和 `GlobalFree`，局部堆应该使用 `LocalAlloc` 和 `LocalFree`。

NT 系列的 Windows 不再支持全局堆，但为了保持与 16 位的 Windows 程序兼容，以上 API 仍保留着，不过无论是 `GlobalAlloc` 还是 `LocalAlloc` 实际上都是从进程的默认堆上来分配内存。也就是说，今天这两个函数已经没有大的差异，新的程序也不该再使用它们。清单 23-6 中的栈回溯显示了 `GlobalAlloc` 和 `LocalAlloc` 函数调用 `RtlAllocateHeap` 的过程。

清单 23-6 `GlobalAlloc` 和 `LocalAlloc` 函数的工作过程

0:000> kbn	# ChildEBP RetAddr Args to Child	
00 0012fef4 7c80fe8f	00140000 00100000 0000006f	ntdll!RtlAllocateHeap+0xf
01 0012ff40 0040114a	00000000 0000006f	0040119c kernel32!GlobalAlloc+0x66
02 0012ff4c 0040119c	00408058 00001000 00010000	HiHeap!TestGlobal+0xa
// 以下是 LocalAlloc 的情况		
00 0012fef4 7c809ff	00140000 00140000 0000006f	ntdll!RtlAllocateHeap+0x5
01 0012ff40 0040115b	00000000 0000006f	0040119c kernel32!LocalAlloc+0x58
02 0012ff4c 0040119c	00408058 00001000 00010000	HiHeap!TestGlobal+0x1b

其中 00140000 是 `RtlAllocateHeap` 的堆句柄参数 (`hHeap`)，实际上就是进程的默认堆句柄，Kernel32.DLL 中用一个名为 `BaseHeap` 的全局变量来记录这个句柄值。栈帧

0 第 4 列的 00100000 和 00140000 是 RtlAllocateHeap 的第二个参数 dwFlags，我们在调用 GlobalAlloc 和 LocalAlloc 函数时没有指定任何标志位，因此这两个标志位是 GlobalAlloc 和 LocalAlloc 加入的。栈帧 0 第 5 列的 0000006f 是要分配的内存块长度。

23.3.5 解除提交

释放从堆上分配的内存并不意味着堆管理器会立刻把这个内存块所对应的空间立刻归还给系统的内存管理器。考虑到应用程序可能很快还会申请内存和减少与内存管理器的交互次数，堆管理器只在以下两个条件都满足时才会立即调用 ZwFreeVirtualMemory 函数向内存管理器释放内存，通常称为解除提交（Decommit）。第一个条件是本次释放的堆块大小超过了堆参数中的 DeCommitFreeBlockThreshold 所代表的阈值，第二个条件是累积起来的总空闲空间（包括本次）超过了堆参数中的 DeCommitTotalFreeThreshold 所代表的阈值。DeCommitFreeBlockThreshold 和 DeCommitTotalFreeThreshold 是放在堆管理区的参数，创建堆时会使用 PEB 结构中的 HeapDeCommitFreeBlockThreshold 和 HeapDeCommitTotalFreeThreshold 字段的值来初始化这两个参数。以下是使用 dt_PEB 命令观察到的这两个字段和它们的取值：

```
+0x080 HeapDeCommitTotalFreeThreshold : 0x10000
+0x084 HeapDeCommitFreeBlockThreshold : 0x1000
```

也就是说，当要释放的堆块超过 4KB 并且堆上的总空闲空间达到 64KB 时，堆管理器才会立即向内存管理器执行解除提交操作真正释放内存，否则，堆管理器会将这个块加到空闲块列表中，并更新堆管理区的总空闲空间值（TotalFree）。清单 23-7 所示的栈回溯反映了 RtlFreeHeap 函数调用 RtlpDeCommitFreeBlock 函数来归还内存的过程。

清单 23-7 堆管理器向内存管理器归还内存

```
0:000> kn
# ChildEBP RetAddr
00 0012fdfc 7c918331 ntdll!ZwFreeVirtualMemory          //释放内存的系统调用
01 0012fe20 7c9183dc ntdll!RtlpSecMemFreeVirtualMemory+0x1b
02 0012fe6c 7c910eca ntdll!RtlpDeCommitFreeBlock+0x1fb //堆管理器的解除提交函数
03 0012ff3c 004011df ntdll!RtlFreeHeap+0x3a2           //释放堆块
04 0012ff54 00401283 HiHeap!TestDecommit+0x6f          //测试解除提交的示例函数
05 0012ff80 00401835 HiHeap!main+0x53                 //用户代码的入口函数
06 0012ffc0 7c816fff7 HiHeap!mainCRTStartup+0xb4        //CRT 插入的入口函数
07 0012ffff0 00000000 kernel32!BaseProcessStart+0x23    //系统的进程启动函数
```

以上栈回溯中释放的堆块的用户区大小为 65536，满足上面说的两个条件。值得说明的是，堆管理器内部是以分配粒度为单位来表示上面说的那两个阈值和计算堆块大小的。调用 GetSystemInfo API 可以取得分配粒度值，通常为 8。在 WinDBG 中使用 !heap -v 命令可以观察堆的分配粒度和解除提交阈值（见清单 23-8）。

清单 23-8 观察堆的参数信息

```

0:000> !heap 140000 -v                                //140000 即进程堆句柄
Index Address Name      Debugging options enabled // 未启用任何调试选项
1: 00140000                                         //这个命令支持列出多个堆的信息
Segment at 00140000 to 00240000 (00015000 bytes committed)
                                                //堆的内存段范围和提交字节数
Flags:          00000002                           //堆标志
ForceFlags:     00000000                           //强制标志
Granularity:   8 bytes                            //堆块分配粒度
Segment Reserve: 00100000                         //堆的保留空间
Segment Commit: 00002000                          //每次向内存管理器提交的内存大小
DeCommit Block Thres: 00000200                    //解除提交的单块阈值
DeCommit Total Thres: 00002000                    //解除提交的总空闲阈值
Total Free Size: 00000000                         //堆中空闲块的总大小
Total Commit Size: 00015000                        //省略其他信息
...

```

其中，单块阈值参数等于 0x200，因为它是以分配粒度（8）为单位的，所以将其乘以 8 便是 0x1000，与 PEB 中的 HeapDeCommitFreeBlockThreshold 是一致的。

也可以通过任务管理器来观察堆管理器是否执行了解除提交动作，在执行了这个动作后，进程的 VM Size 数会变小。以本节的 HiHeap 程序为例，在命令行指定堆块的大小，在分配堆块前观察这个进程的 VM Size 值（180KB），然后按任意键让程序分配堆块，观察 VM Size 值，再按键让程序释放堆块，观察 VM Size 值。如果指定的堆块参数大于 64KB，那么释放动作会触发解除提交动作，可以看到 VM Size 值减小。表 23-2 记录了笔者所作的两次试验结果。

表 23-2 使用任务管理器观察 HiHeap 程序所使用的虚拟内存量（VM Size 值）

试验序号	参数（堆块大小）	分配前	分配后	释放后
#1	65535	180KB	248KB	180KB
#2	60000	180KB	240KB	240KB

可见两次分配都触发了堆管理器向内存管理器提交内存，因此进程的虚拟内存使用量变大。但是只有第一次的释放动作触发了堆管理器立刻向内存管理器归还内存。

23.4 堆的内部结构

上一节我们介绍了堆的基本概念，并从应用角度介绍了从堆上分配与释放内存的方法。本节将介绍堆的内部结构，以及堆管理器组织堆中数据的方法。

23.4.1 结构和布局

正如我们在第 23.1 节所介绍的，可以把堆管理器想象为经营内存业务的零售商，它从 Windows 内核的内存管理器那里批发内存块过来，然后零售给应用程序。图 23-2 画出了一个 Win32 堆所拥有的内存块。左侧的大矩形是这个堆创建之初所批发过来的第一个内存段（Segment），我们将其简称为 0 号段（Segment00）。每个堆至少拥有一个段，

即 0 号段，最多可以有 64 个段。堆管理器在创建堆时会建立一个段（Segment），在一段用完后，如果这个堆是可增长的，也就是堆标志中含有 HEAP_GROWABLE(2) 标志，那么堆管理器会再分配一个段。

在 0 号段的开始处存放着堆的头信息，是一个 HEAP 结构，其中定义了很多个字节用来记录堆的属性和“资产”状况，比如 Segments 数组记录了这个堆拥有的所有段。每个段都有一个 HEAP_SEGMENT 结构来描述自己，对于 0 号段，这个结构位于 HEAP 之后，对于其他段，这个结构就在段的起始处。例如，图 23-2 中右侧上方的矩形代表了这个堆的 1 号段，它的最上方就是一个 HEAP_SEGMENT 结构。

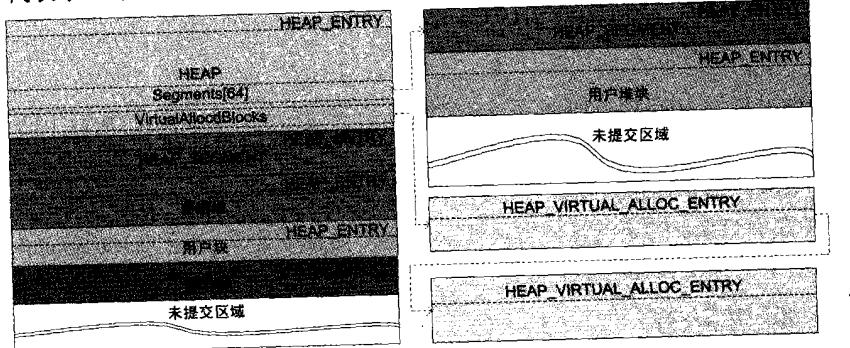


图 23-2 堆的数据布局

堆中的内存区便被分割为一系列不同大小的堆块。每个堆块的起始处一定是一个 8 字节的 HEAP_ENTRY 结构，后面便是供应用程序使用的区域，通常称为用户区。HEAP_ENTRY 结构的前两个字节是以分配粒度表示的堆块大小。分配粒度通常为 8，这意味着每个堆块的最大值是 2 的 16 次方乘以 8，即 $0x10000 * 8 = 0x80000 = 524288$ 字节=512KB，因为每个堆块至少要有 8 字节的管理信息，因此应用程序可以使用的最大堆块便是 $0x80000 - 8 = 0x7FFF8$ ，这也正是 SDK 文档中所给出的数值（位于 HeapAlloc 函数 dwMaximumSize 参数的说明中）。不过这并不意味着不可以从 Win32 堆上分配到更大的内存块。当一个应用程序要分配大于 512KB 的堆块时，如果堆标志中包含 HEAP_GROWABLE (2)，那么堆管理器便会直接调用 ZwAllocateVirtualMemory 来满足这次分配，并把分得的地址记录在 HEAP 结构的 VirtualAllocdBlocks 所指向的链表中。这意味着，堆管理器批发过来的大内存块，有两种形式，一种形式是段，另一种形式就是直接的虚拟内存分配，我们将后一种形式称为大虚拟内存块。图 23-2 右侧下方的两个矩形代表了这个堆所拥有的两个大虚拟内存块。因为对管理器是以链表方式来管理大虚拟内存块的，因此其数量是没有限制的。每个大虚拟内存块的起始处是一个 HEAP_VIRTUAL_ALLOC_ENTRY 结构（32 字节）。HiHeap 程序中的 TestVirtualAlloc 函数演示了如何触发堆管理器分配大虚拟内存块。考虑到大虚拟内存块是属于堆块的特例，而且是少数情况，所以接下来的内容将集中讨论段中的普通堆块。

HEAP_SEGMENT 结构后面是一个特殊的堆块，它用来存放已经释放堆块的信息，

主要是一个旁视列表。当应用程序释放一个普通的小型堆块时，堆管理器可能把这个堆块的信息加入到旁视列表中，然后就返回。在分配新的堆块时，堆管理器会先搜索旁视列表看是否有合适的堆块。因为从旁视列表中分配堆块是优先于其他分配逻辑的，所以它又叫前端堆（Front End Heap），前端堆主要用来提高释放和分配堆块的速度。

段中的所有已经提交的空间都属于某一个堆块，即使是 HEAP 结构和 HEAP_SEGMENT 结构所占用的空间也是分别属于一个单独的堆块，因此这两个结构的起始处都是一个 HEAP_ENTRY 结构。下面我们分别介绍 HEAP、HEAP_SEGMENT 和 HEAP_ENTRY 这三种管理堆的重要数据结构。

23.4.2 HEAP 结构

堆管理器使用 HEAP 结构来记录和维护堆的管理信息，因此我们把这个结构称为堆的管理结构，因为这个结构总是在每个堆的开始处，因此有时也被称为堆的头结构。事实上，HeapCreate 返回的句柄便是指向 HEAP 结构的指针。SDK 没有详细介绍 HEAP 结构，但是在 NTDLL 的调试符号中包含了 HEAP 结构的定义。清单 23-9 显示了 HEAP 结构的各个字段及在进程堆中的取值。

清单 23-9 堆的管理结构

0:000> dt ntdll!_HEAP 00140000	//00140000 为进程堆句柄
+0x000 Entry	: _HEAP_ENTRY //用于存放管理结构的堆块结构
+0x008 Signature	: 0xeffffeff //HEAP 结构的签名，固定为这个值
+0x00c Flags	: 2 //堆标志，2 代表 HEAP_GROWABLE
+0x010 ForceFlags	: 0 //强制标志
+0x014 VirtualMemoryThreshold	: 0xfe00 //最大堆块大小，见下文
+0x018 SegmentReserve	: 0x100000 //段的保留空间大小
+0x01c SegmentCommit	: 0x2000 //每次提交内存的大小
+0x020 DeCommitFreeBlockThreshold	: 0x200 //解除提交的单块阈值（粒度为单位）
+0x024 DeCommitTotalFreeThreshold	: 0x2000 //解除提交的总空闲块阈值（粒度数）
+0x028 TotalFreeSize	: 7 //空闲块总大小，以粒度为单位
+0x02c MaximumAllocationSize	: 0x7ffdefff //可分配的最大值
+0x030 ProcessHeapsListIndex	: 1 //本堆在进程堆列表中的索引
+0x032 HeaderValidateLength	: 0x608 //头结构的验证长度，实际占用 0x640
+0x034 HeaderValidateCopy	: (null)
+0x038 NextAvailableTagIndex	: 0 //下一个可用的堆块标记索引
+0x03a MaximumTagIndex	: 0 //最大的堆块标记索引号
+0x03c TagEntries	: (null) //指向用于标记堆块的标记结构
+0x040 UCRSegments	: (null) //UnCommittedRange Segments
+0x044 UnusedUnCommittedRanges	: 0x140598 _HEAP_UNCOMMITTED_RANGE
+0x048 AlignRound	: 0xf
+0x04c AlignMask	: 0xffffffff8 //用于地址对齐的掩码
+0x050 VirtualAllocdBlocks	: _LIST_ENTRY [0x140050 - 0x140050]
+0x058 Segments	: [64] 0x00140640 _HEAP_SEGMENT //段数组
+0x158 u	: __unnamed
+0x168 u2	: __unnamed
+0x16a AllocatorBackTraceIndex	: 0 //用于记录回溯信息
+0x16c NonDedicatedListLength	: 0
+0x170 LargeBlocksIndex	: (null)
+0x174 PseudoTagEntries	: (null)

+0x178 FreeLists	:	[128] _LIST_ENTRY [0x140178 - 0x140178] //空闲块
+0x578 LockVariable	:	0x00140608 _HEAP_LOCK //用于串行化控制的同步对象
+0x57c CommitRoutine	:	(null)
+0x580 FrontEndHeap	:	0x00140688 //用于快速释放堆块的“前端堆”
+0x584 FrontHeapLockCount	:	0 //“前端堆”的锁定计数
+0x586 FrontEndHeapType	:	0x1 '' //“前端堆”的类型
+0x587 LastSegmentIndex	:	0 '' //最后一个段的索引号

解释一下 VirtualMemoryThreshold 字段的含义，它是以分配粒度为单位的堆块阈值，即我们前面提到过的可以在段中分配的堆块最大值，它是以用户数据区的大小来衡量的， $0xfe00 * 8 = 0x7f000 = 508KB$ ，这个值小于真正的最大值，为堆块的管理信息区保留了 4KB 的空间，是一个很稳妥的上限。这个阈值意味着这个堆中最大的普通堆块的用户数据区是 508KB，对于超过这个数值的分配申请，堆管理器会直接调用 ZwAllocateVirtualMemory 来满足这次分配，并把分得的地址记录在 VirtualAllocdBlocks 所指向的链表中。这样做的前提是标志中包含 HEAP_GROWABLE(2)。如果堆标志中不包含 HEAP_GROWABLE，这样的分配就会失败。或者说，如果一个堆是不可增长的，那么可以分配的最大用户数据区便是 512KB，即使堆中空闲空间远大于这个值。

Segments 字段用来记录堆中包含的所有段，它是一个数组，每个元素是一个指向 HEAP_SEGMENT 结构的指针。LastSegmentIndex 字段用来标志目前堆中最后一个段的序号，其值加一便是段的总个数。

FreeLists 是一个包含 128 个元素的数组，用来记录堆中空闲堆块链表的表头。当有新的分配请求时，堆管理器会遍历这个链表寻找可以满足请求大小的最接近堆块，如果找到了，便将这个块分配出去，否则，便要考虑为这次请求提交新的内存页和建立新的堆块。当释放一个堆块时，除非是这个堆块满足解除提交的条件要直接释放给内存管理器，大多数情况也都是将其修改属性并加入到空闲链表中。稍后我们还会详细介绍堆块的分配和释放过程。

23.4.3 HEAP_SEGMENT 结构

清单 23-10 给出了用来描述堆中内存段 HEAP_SEGMENT 结构的各个字段，其中字段的取值是针对进程堆的第一个段的，即清单 23-9 中 Segments 数组的第一个元素。

清单 23-10 堆的段结构

0:000> dt _HEAP_SEGMENT 0x00140640		
ntdll!_HEAP_SEGMENT		
+0x000 Entry	:	_HEAP_ENTRY //段中存放本结构的堆块
+0x008 Signature	:	0xfffffff //段结构的签名，固定为这个值
+0x00c Flags	:	0 //段标志
+0x010 Heap	:	0x00140000 _HEAP //段所属的堆
+0x014 LargestUnCommittedRange	:	: 0xfc000 //段的地址
+0x018 BaseAddress	:	//段的地址
+0x01c NumberOfPages	:	//段的内存页数

+0x020 FirstEntry	:	0x00140680 _HEAP_ENTRY //第一个堆块
+0x024 LastValidEntry	:	0x00240000 _HEAP_ENTRY //堆块的边界值
+0x028 NumberOfUnCommittedPages	:	0xfc //尚未提交的内存页数
+0x02c NumberOfUnCommittedRanges	:	1 //UnCommittedRanges 数组元素数
+0x030 UnCommittedRanges	:	0x00140588 _HEAP_UNCOMMITTED_RANGE
+0x034 AllocatorBackTraceIndex	:	0 //初始化段的 UST 记录序号
+0x036 Reserved	:	0
+0x038 LastEntryInSegment	:	0x00143128 _HEAP_ENTRY //最末一个堆块

上面显示的是堆的第一个段的信息，在这段的开头存放的是堆的管理结构，其地址范围是 0x140000~0x140640，从 0x00140640 开始是 0x40 字节长的段结构，即上面的结构，之后便是段中的第一个用户堆块，FirstEntry 字段用来直接指向这个堆块。堆管理器使用 HEAP_ENTRY 结构来描述每个堆块。

23.4.4 HEAP_ENTRY 结构

描述堆块的最重要数据结构是 HEAP_ENTRY。同样，SDK 文档没有介绍过这个结构，但是 NTDLL 的调试符号中包含了这个结构的描述，表 23-3 列出了使用 WinDBG 的 dt ntdll!_HEAP_ENTRY 命令观察到的 HEAP_ENTRY 结构的各个字段。

表 23-3 HEAP_ENTRY 结构

字段	含义
+0x000 Size	堆块的大小，以分配粒度为单位
+0x002 PreviousSize	前一个堆块的大小
+0x004 SmallTagIndex	用于检查堆溢出的 Cookie
+0x005 Flags	标志
+0x006 UnusedBytes	因为补齐而多分配的字节数
+0x007 SegmentIndex	这个堆块所在段的序号

其中，Flags 字段代表堆块的状态，其值是表 23-4 列出的标志位的组合，这些标志位是根据 !heap -a 命令的输出结果整理得到的。

表 23-4 堆块的部分状态标志 (Flags)

标志	值	含义
HEAP_ENTRY_BUSY	01	该块处于占用 (Busy) 状态
HEAP_ENTRY_EXTRA_PRESENT	02	这个块存在额外 (Extra) 描述
HEAP_ENTRY_FILL_PATTERN	04	使用固定模式填充堆块
HEAP_ENTRY_VIRTUAL_ALLOC	08	虚拟分配 Virtual Allocation
HEAP_ENTRY_LAST_ENTRY	0x10	这是该段的最后一个块

HEAP_ENTRY 结构的长度固定为 8 字节长，位于堆块起始处，其后便是堆块的用户数据。也就是说，将 HeapAlloc 函数返回的地址减去 8 便是这个堆块的 HEAP_ENTRY 结构的地址。

23.4.5 分析堆块的分配和释放过程

下面举个例子来说明从堆上分配和释放内存块的过程。在 WinDBG 中打开

bin\debug 目录下的 HiHeap 程序，然后在 TestAlloc 函数的如下语句设置断点。

```
void * pStruct = HeapAlloc(GetProcessHeap(), 0, MAX_PATH);
```

易见这条语句是从进程堆中申请一个长度为 MAX_PATH (260) 字节长的内存块。单步执行完这条语句，观察到函数返回的地址为 00142a80，这个地址指向的是分配给应用程序的内存区的起始地址，也就是用户数据区的起始地址。使用 dd 00142a80 观察该地址，会发现整个内存区都被填充为 0xBAADF00D (英文 Bad Food)，这是因为我们在调试器中运行程序，系统自动启用了堆的调试支持（第 23.6 节详细讨论）。

```
0:000> dd 00142a80
00142a80 baadf00d baadf00d baadf00d baadf00d
```

在此地址的前 8 个字节便是 HEAP_ENTRY 结构，可以使用 dt 命令观察。

```
0:000> dt ntdll!_HEAP_ENTRY 00142a80-8
+0x000 Size : 0x24 //以粒度表示的块大小，不包括本结构
+0x002 PreviousSize : 9 //前一个堆块的大小
+0x000 SubSegmentCode : 0x00090024 //子段代码
+0x004 SmallTagIndex : 0x8a '' //堆块的标记序号
+0x005 Flags : 0x7 '' //堆块的状态标志
+0x006 UnusedBytes : 0x1c '' //用户数据区后的未使用字节数
+0x007 SegmentIndex : 0 '' //所在段序号
```

其中 0x24 是以粒度为单位的块大小， $0x24*8 = 288 =$ 用户请求大小 (260) + UnusedBytes(28)。Flags 为 7，表明该块处于占用 (Busy) 状态，这个块存在额外 (Extra) 描述 (位于块尾)，而且进行过模式填充 (baadf00d)。SegmentIndex 为 0，表示这个堆块是从这个堆的 0 号段 (Segment00) 中分配的。

继续跟踪 TestAlloc，单步执行下面的 HeapFree 语句。

```
HeapFree(GetProcessHeap(), 0, pStruct);
```

单步执行好以上语句后，再次观察刚才的两个地址。

```
0:000> dd 00142a80
00142a80 00140178 feffffee feffffee
00142a90 feffffee feffffee feffffee
```

也就是用户区除前 8 个字节外的区域都被填充为 feffffee，看起来像英文的 free，这正是堆管理器对已经释放堆块所自动填充的内容。

再观察 HEAP_ENTRY 结构：

```
0:000> dt ntdll!_HEAP_ENTRY 00142a80-8
+0x000 Size : 0xb1 //以粒度表示的块大小
+0x002 PreviousSize : 9 //前一个堆块的大小
+0x000 SubSegmentCode : 0x000900b1
+0x004 SmallTagIndex : 0x8a '' //堆块的标记序号
+0x005 Flags : 0x14 '' //堆块的状态标志
+0x006 UnusedBytes : 0x1c '' //残留的信息
+0x007 SegmentIndex : 0 '' //所在段序号
```

大小从刚才的 0x24 变为 0xb1，即 $0xb1*8=1416$ 字节，这是堆管理器将刚才释放

的块与其后的空闲区域合并成了一个大的空闲块，留作以后使用，合并的过程叫拼接（Coalesce）。RtlpCoalesceFreeBlocks 和 RtlpCoalesceHeap 函数便是用来完成空闲堆块拼接任务的。

另外，堆块的标志字段从刚才的 0x7 变为 0x14，表示这个块已经空闲，是这个段中的最后一个块，而且进行过模式填充。

事实上，对于已经释放的堆块，堆管理器专门定义了 HEAP_FREE_ENTRY 结构来描述。这个结构的前 8 个字节与 HEAP_ENTRY 一样，但增加了 8 个字节用于存放空闲链表（Free List）的节点。因此，可以使用 dt ntdll!_HEAP_FREE_ENTRY 来观察刚才的地址：

```
0:000> dt ntdll!_HEAP_FREE_ENTRY 00142a80-8
+0x000 Size          : 0xb1           //堆块的大小，粒度为单位
+0x002 PreviousSize : 9              //上一堆块的大小，粒度为单位
+0x000 SubSegmentCode : 0x000900b1   //子段代码
+0x004 SmallTagIndex : 0x8a ''       //堆块的标记序号
+0x005 Flags          : 0x14 ''       //堆块标志
+0x006 UnusedBytes   : 0x1c ''       //残留信息
+0x007 SegmentIndex  : 0 ''           //所在段序号
+0x008 FreeList       : _LIST_ENTRY [ 0x140178 - 0x140178 ] //空闲链表节点
```

刚才我们观察释放后的内存地址，看到的起始 8 个字节便是 FreeList 字段。堆结构中包含了一个 FreeLists 数组（128 项）用来存放各个空闲链表的表头。因为这个堆块是最后一个块，所以 LIST_ENTRY 结构的 FLink 和 Blink 字段指向的都是这个空闲链表的头节点。

23.4.6 使用!heap 命令观察堆块信息

使用 WinDBG 的!heap 命令加上-h 开关和堆句柄可以显示出堆的堆块信息。清单 23-11 给出了针对 HiHeap 程序进程堆的执行结果和笔者的注释。

清单 23-11 使用!heap 命令观察堆的堆块信息

```
0:000> !heap 140000 -hf
Index Address Name      Debugging options enabled
1: 00140000             //堆句柄，!heap 命令支持显示多个堆
Segment at 00140000 to 00240000 (000f6000 bytes committed) //0 号段
Segment at 00410000 to 00510000 (0007a000 bytes committed) //1 号段
Flags:                 00000002           //段标志
Total Free Size:      000009ae           //空闲堆块的总大小（粒度为单位）
Virtual Alloc List:   00140050           //大虚拟内存块链表
UCR FreeList:          001405a8           //空闲的 UnCommittedRange 链表表头
FreeList Usage:        00000000 00000000 00000000 00000000
FreeList[ 00 ] at 00140178: 00233140 . 00488160           //0 号空闲链表
    00488158: 00110 . 01ea8 [10] - free                  //空闲堆块
    00233138: 78008 . 02ec8 [10] - free                  //空闲堆块
Heap entries for Segment00 in Heap 00140000               //0 号段中的堆块
    00140640: 00640 . 00040 [01] - busy (40)            //段结构所占的堆块
    00140680: 00040 . 01808 [01] - busy (1800) (Tag d0) //“前端堆”占的堆块
```

```

...
00143128: 00078 . 78008 [01] - busy (78000) (Tag 25) //用户堆块
001bb130: 78008 . 78008 [01] - busy (78000) (Tag 26) //用户堆块
00233138: 78008 . 02ec8 [10] //段中的最后一个块, 是空闲块
00236000: 0000a000 - uncommitted bytes. //未提交区
Heap entries for Segment01 in Heap 00140000 //1号段中的堆块
00410000: 00000 . 00040 [01] - busy (40) //段结构所占的堆块
00410040: 00040 . 78008 [01] - busy (78000) (Tag 8) //用户堆块
00488048: 78008 . 00110 [01] - busy (104) (Tag 9) //用户堆块
00488158: 00110 . 01ea8 [10] //段中的最后一个块, 是空闲块
0048a000: 00086000 - uncommitted bytes. //未提交区

```

我们解释一下空闲堆块的描述, 第7行显示的是空闲链表中堆块的总大小(0x9ae), 它是以粒度为单位的, 乘以8可以换算为字节数, 即 $0x9ae * 8 = 0x4d70$ 。第13和14行显示出了两个空闲堆块的信息, 第一个的大小是0x1ea8字节, 第二个的大小是0x2ec8字节, 加起来正好是0x4d70字节。

以上列表中关于堆块的显示格式是, “堆块的起始地址: 前一个堆块的字节数.本堆块的字节数[堆块标志] - 堆块标志的文字表示 (堆块的用户数据区字节数)(堆块的标记序号)”。其中堆块的起始地址就是HEAP_ENTRY结构的地址, 用户区字节数不包含多分配的未使用字节(UnusedBytes), 后两个部分是可选的。

以倒数第3行为例, 其节点地址为00488048, 接下来的78008是前一个块的大小, 00110是当前块的大小(字节数)。01是标志, 代表繁忙(Busy), 104是用户数据区的字节数, 也就是调用HeapAlloc时应用程序申请的大小。

23.5 低碎片堆 (LFH)

在堆上的内存空间被反复分配和释放一段时间后, 堆上的可用空间可能被分割得支离破碎, 当再试图从这个堆上分配空间时, 即使可用空间加起来的总额大于请求的空间, 但是因为没有一块连续的空间可以满足要求, 那么分配请求仍会失败, 这种现象被称为堆碎片(Heap Fragmentation)。堆碎片与磁盘碎片的形成机理是一样的, 但却比磁盘碎片的影响更大, 因为多个磁盘碎片加起来仍可以满足磁盘分配请求, 但是堆碎片是无法通过累加来满足内存分配要求的, 因为堆函数返回的必须是地址连续的一段空间。

针对堆碎片问题, Windows XP 和 Windows Server 2003 引入了低碎片堆(Low Fragmentation Heap), 简称LFH。那么LFH是如何来降低碎片的呢?首先, LFH将堆上的可用空间划分成128个桶位(Bucket), 编号为1~128, 每个桶位的空间大小依次递增, 1号桶为8个字节, 128号桶为16384字节(即16KB)。当需要从LFH上分配空间时, 堆管理器会根据堆函数参数中所请求的字节将满足要求的最小可用桶分配出去。例如, 如果应用程序请求分配7个字节, 而且1号桶空闲, 那么便将1号桶分配给它, 如果1号桶已经分配出去了(busy), 那么便尝试分配2号桶。另外, LFH为不同编号区域的桶规定了不同的粒度, 桶的容量越大, 分配桶时的粒度也越大, 比如1~32号桶的粒度是8字节, 这意味着这些桶的最小分配单位是8字节, 不足8字节的分配请求也至少会

分配给 8 个字节。表 23-5 列出了 LFH 各个桶位的分配粒度和最佳适用范围。

表 23-5 低碎片堆中不同桶位的粒度和适用范围

Buckets	Min Fragmentation	Size Range
1~32	8	1~256
33~48	16	257~512
49~64	32	513~1024
65~80	64	1025~2048
81~96	128	2049~4096
97~112	256	4097~8192
113~128	512	8193~16384

通过 `HeapSetInformation` API 可以对一个已经创建好的 Win32 堆启用低碎片堆支持。例如，下面的代码对当前进程的进程堆启用 LFH 功能。

```
ULONG HeapFragValue = 2;
BOOL bSuccess = HeapSetInformation(GetProcessHeap(),
    HeapCompatibilityInformation, &HeapFragValue, sizeof(HeapFragValue));
```

调用 `HeapQueryInformation` API 可以查询一个堆是否启用了 LFH 支持。

23.6 堆的调试支持

为了帮助发现内存有关的问题，堆管理器提供了一系列功能来辅助调试。

- 堆尾检查 (Heap Tail Checking)，简称 HTC，是在堆块的末尾附加额外的标记信息（通常为 8 个字节），用于检查堆块是否发生溢出，其原理与我们上一章介绍的防止栈上缓冲区溢出的栅栏字节类似。
- 释放检查 (Heap Free Checking)，简称 HFC，是在释放堆块时对堆进行各种检查，可以防止多次释放同一个堆块。
- 参数检查，对传递给堆管理器的参数进行更多的检查。
- 调用时验证 (Heap Validation on Call)，简称 HVC，即每次调用堆函数时都对整个堆进行验证和检查。
- 堆块标记 (Heap Tagging)，即为堆块增加附加标记 (Tag)，以记录堆块的使用情况或其他信息。
- 用户态栈回溯 (User mode Stack Trace)，简称 UST，即将每次调用堆函数的函数调用信息 (Calling Stack) 记录到一个内存数据库中。
- 专门用于调试的页堆 (Debug Page Heap)，简称 DPH 堆。

HTC、HVC 和 DPH 主要用来检测堆上的缓冲区溢出，我们将在第 23.8 节和第 23.9 节介绍，下一节介绍 UST 功能，本节接下来将介绍启用调试支持的方法和 HFC 功能。

23.6.1 全局标志

创建堆时，堆管理器根据当前进程的全局标志 (Global Flags) 来决定是否启用堆

的调试功能。操作系统的进程加载器在加载一个进程时会从注册表中读取进程的全局标志值，具体来说是在 HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options 表键下寻找以该程序名（如 MyApp.EXE，不区分大小写）命名的子键，如果存在这样的子键，那么读取下面的 GlobalFlag 键值（REG_DWORD 类型）。

可以使用 GFlags 工具（gflags.exe）来编辑系统的全局标志或某个程序文件的全局标志。WinDbg 帮助文件中列出了所有全局标志。表 23-6 列出了与堆有关的全局标志，其中缩写列是可以传递给 gflags 工具的命令行参数，比如 gflags/i HiHeap.exe +ust 便为 HiHeap.exe 程序增加了 FLG_USER_STACK_TRACE_DB 标志。事实上，gflags 工具只是将标志信息保存在上面所说的注册表表键下（如果不存在，会先创建），所以使用 gflags 工具与直接手工向注册表中加入键值是等价的。

表 23-6 堆有关的全局标志（Global Flags）

标志	值	缩写	说明
FLG_HEAP_ENABLE_FREE_CHECK	0x20	hfc	释放检查
FLG_HEAP_VALIDATE_PARAMETERS	0x40	hpc	参数检查
FLG_HEAP_ENABLE_TAGGING	0x800	htg	附加标记
FLG_HEAP_ENABLE_TAG_BY_DLL	0x8000	htd	通过 DLL 附加标记
FLG_HEAP_ENABLE_TAIL_CHECK	0x10	htc	堆尾检查
FLG_HEAP_VALIDATE_ALL	0x80	hvc	全面验证
FLG_HEAP_PAGE_ALLOCS	0x02000000	hpa	Debug Page Heap
FLG_USER_STACK_TRACE_DB	0x1000	ust	用户态栈回溯
FLG_HEAP_DISABLE_COALESCING	0x00200000	dhc	禁用合并空闲块

如果是在调试器中运行一个程序，而且注册表中没有设置 GlobalFlag 键值，那么操作系统的加载器会默认将全局标志设置为 0x70，也就是启用 htc、hfc 和 hpc 三项堆调试功能。如果注册表中有设置，那么会使用注册表中的设置。例如，在 WinDbg 中打开 HiHeap.exe 后，执行 !gflag 命令就可以观察当前进程的全局标志取值：

```
0:000> !gflag
Current NtGlobalFlag contents: 0x00000070
    htc - Enable heap tail checking
    hfc - Enable heap free checking
    hpc - Enable heap parameter checking
```

如果是附加到一个已经运行的进程，那么它的全局标志值就是它的本来值（来自注册表和程序文件），对于普通程序默认为 0。

23.6.2 释放检查

很多堆损坏的情况是由于错误的释放动作所导致的，比如多次释放同一个堆块，或者从一个堆释放本不属于这个堆的堆块。堆释放检查（HFC）功能可以比较有效地发现这类问题。为了便于说明，我们编写了一个名为 HeapHFC 的控制台程序，其代码

如清单 23-12 所示。

清单 23-12 演示释放检查的 HeapHFC 程序

```
#include <windows.h>
int main(int argc, char* argv[])
{
    void * p;
    BOOL bRet;
    HANDLE hHeap; //堆句柄

    printf("Heap Free Check (HFC)!\n");
    hHeap=HeapCreate(0, 4096, 0); //创建一个新的堆
    p = HeapAlloc(hHeap, 0, 20); //分配 20 个字节的堆块
    bRet=HeapFree(hHeap, 0, p); //第一次释放
    printf("Free pointer p first time, %d\n", bRet);
    bRet=HeapFree(hHeap, 0, p); //第二次释放同一堆块
    printf("Free pointer p second time, %d\n", bRet);
    bRet=HeapValidate(hHeap, 0, NULL); //验证堆
    printf("HeapValidate returned %d\n", bRet);
    bRet=HeapDestroy(hHeap); //销毁堆
    printf("HeapDestroy returned %d\n", bRet);
    return getchar();
}
```

HeapHFC 会创建一个新的堆，然后从这个堆上分配一个堆块，并两次调用 HeapFree 释放这个堆块。多次释放是导致堆损坏的一个重要因素。

直接执行调试版本或发布版本的 HeapHFC 程序，在输出如下内容后，HeapHFC 程序开始占用非常高（99%）的 CPU，表现出死循环的症状。

```
c:\dig\dbg\author\code\bin\release>HeapHFC
Heap Free Check (HFC) !
Free pointer p first time, 1
Free pointer p second time, 3743233
^C
```

看来是在执行 HeapValidate 函数时出了问题，按 Ctrl+C 可以将其强行停止。上面第 3 行中的信息表明第一次释放返回真，是成功的。第二次释放返回的是 3743233，一个较大的非零整数，根据 SDK 文档，非零即表示 HeapFree 成功，看来第二次释放尽管是明显错误的，但是堆管理器并没有发现和通过返回值报告这个错误。但当 HeapValidat 函数对堆做全面检查时，该函数遇到了问题。

下面我们启用释放检查功能，即在 HeapHFC.exe 所在目录执行：

```
gflags -i HeapHFC.exe +hfc
```

此时再执行 HeapHFC，得到的结果是：

```
C:\dbg\author\code\bin\release>HeapHFC
Heap Free Check (HFC) !
Free pointer p first time, 1
Free pointer p second time, 0 // 返回-代表释放失败
HeapValidate returned 1
HeapDestroy returned 1
```

看来这次堆管理器发现了错误，并通过返回值报告了这个错误。HeapValidate 返回

1，表示堆仍然是完好的。堆完好，表明第二次释放被及时制止了。

下面我们看看在调试器中执行的情况。在 WinDBG 中打开发布版本的 HeapHFC.exe。为了说明 HFC 功能对于调试版本也适用，我们以发布版本为例，事先已经为发布版本的 HeapHFC.exe 生成了调试符号（项目链接属性中选中 Generate debug info）。

当初始断点发生时，使用 !gflag 扩展命令确认全局标志中已经包含了 hfc（释放检查）标志。

按 F5 让程序执行，在控制台窗口打印两行消息（第一次释放）后，程序中断到调试器中，WinDBG 显示清单 23-13 中的信息。

清单 23-13 堆管理器检测到释放错误后打印的调试信息和发起的断点异常

```
HEAP[HeapHFC.exe]: Invalid Address specified to RtlFreeHeap( 00390000, 00391EA0 )
(41c.1688): Break instruction exception - code 80000003 (first chance)
eax=00391e98 ebx=00391e98 ecx=7c91eb05 edx=0012fb16 esi=00390000
...
ntdll!DbgBreakPoint:
7c901230 cc          int     3
```

第 1 行的调试信息表明堆管理器检测到了 HeapHFC.exe 程序在调用 RtlFreeHeap 从堆 00390000 上释放堆块 00391EA0（用户指针）时指定的地址参数非法。第 2 行的信息表明应用程序是因为断点异常而中断到了调试器，其中 (41c.1688) 是 HeapHFC 的进程 ID 和线程 ID。事实上，这是堆管理器检测到错误情况并判断当前程序是在被调试后故意触发了断点异常。使用 k 命令可以看到完整的执行过程（见清单 23-14）。

清单 23-14 堆释放检查功能报告错误

0:000> k				
ChildEBP RetAddr				
0012fd1c 7c96c943 ntdll!DbgBreakPoint				// 触发段断点异常
0012fd24 7c96cd80 ntdll!RtlpBreakPointHeap+0x28				// 堆管理器的触发断点函数
0012fd38 7c96df66 ntdll!RtlpValidateHeapEntry+0x113				// 验证堆块
0012fdac 7c94a5d0 ntdll!RtlDebugFreeHeap+0x97				// 支持调试功能的释放函数
0012fe94 7c9268ad ntdll!RtlFreeHeapsSlowly+0x37				// 慢速的堆块释放函数
0012ff64 0040104e ntdll!RtlFreeHeap+0xf9				// HeapFree API
0012ff80 00401355 HeapHFC!main+0x4e				// 程序的用户入口
0012ffc0 7c816fd7 HeapHFC!mainCRTStartup+0xb4				// 编译器插入的入口函数
0012fff0 00000000 kernel32!BaseProcessStart+0x23				// 系统的进程启动函数

RtlpBreakPointHeap 函数会检查当前进程是否处于被调试状态，如果不在被调试，那么便不会发起断点异常。

对于某些版本的堆实现（NTDLL），需要同时启用 hpc 标志，否则可能不会自动触发断点异常。

本节介绍了启用 Win32 堆调试支持的方法和释放检查功能，要说明的一点是，一旦启用了堆的调试功能，那么堆管理器会把安全检查和调试支持放在第一位，会使用带有全面检查功能的分配和释放函数，这会导致程序的执行速度下降。例如，

RtlAllocateHeap 会调用 RtlAllocateHeapSlowly 执行真正的堆分配功能，相应的 RtlFreeHeap 函数也会将调用转给 RtlFreeHeapSlowly 来执行。

23.7 栈回溯数据库

当调试内存问题时，很多时候我们希望知道每个内存块是由哪段代码或哪个函数分配的，而且最好有这个函数被调用的完整过程，这样便可以大大提高定位错误代码的速度。堆管理器所实现的用户态栈回溯（User-Mode Stack Trace，简称 UST）机制就是为了实现这个目的而设计的。

23.7.1 工作原理

如果当前进程的全局标志中包含了 UST 标志（FLG_USER_STACK_TRACE_DB，0x1000），那么堆管理器会为当前进程分配一块大的内存区，并建立一个 STACK_TRACE_DATABASE 结构来管理这个内存区，然后使用全局变量 ntdll!RtlpStackTraceDataBase 指向这个内存结构。这个内存区被称为用户态栈回溯数据库（User-Mode Stack Trace Database），简称栈回溯数据库或 UST 数据库。清单 23-15 显示了 UST 数据库的头结构。

清单 23-15 UST 数据库的 STACK_TRACE_DATABASE 结构

```
0:001> dt ntdll!_STACK_TRACE_DATABASE 00410000
+0x000 Lock          : __unnamed           //同步对象
+0x038 AcquireLockRoutine : 0x7c901005  ntdll!RtlEnterCriticalSection+0
+0x03c ReleaseLockRoutine : 0x7c9010ed  ntdll!RtlLeaveCriticalSection+0
+0x040 OkayToLockRoutine : 0x7c952080  ntdll!NtDllOkayToLockRoutine+0
+0x044 PreCommitted   : 0 ''               //数据库提交标志
+0x045 DumpInProgress : 0 ''               //转储标志
+0x048 CommitBase     : 0x00410000         //数据库的基地址
+0x04c CurrentLowerCommitLimit : 0x00422000
+0x050 CurrentUpperCommitLimit : 0x0140f000
+0x054 NextFreeLowerMemory : 0x00421acc  ""      //下一空闲位置的低地址
+0x058 NextFreeUpperMemory : 0x0140f4fc  "???"    //下一空闲位置的高地址
+0x05c NumberOfEntriesLookedUp : 0x3fb
+0x060 NumberOfEntriesAdded : 0x2c1          //已加入的表项数
+0x064 EntryIndexArray : 0x01410000 -> (null)
+0x068 NumberOfBuckets : 0x89              //Buckets 数组的元素数
+0x06c Buckets : [1] 0x00410a50 _RTL_STACK_TRACE_ENTRY // Buckets 数组
```

其中 dt 命令中的地址就是 UST 数据库的起始地址，是通过观察全局变量 RtlpStackTraceDataBase 得到的：

```
0:001> dd ntdll!RtlpStackTraceDataBase 11
7c97c0d0 00410000
```

Buckets 是个指针数组，数组的每个元素指向的是一个桶位。堆管理器在存放栈回溯记录时，先计算这个记录的哈希值（Hash），然后对桶位数（NumberOfBuckets）求

余 (%), 将得到的值作为这个记录所在的桶位。位于同一个桶位的多个记录是以链表方式链接在一起的。每个栈回溯记录是一个 RTL_STACK_TRACE_ENTRY 结构。清单 23-16 列出了 RTL_STACK_TRACE_ENTRY 结构的各个字段, 以及针对第一个回溯记录的取值。

清单 23-16 UST 数据库的回溯记录

```
0:001> dt _RTL_STACK_TRACE_ENTRY 0x00410a50
+0x000 HashChain      : 0x00410e9c _RTL_STACK_TRACE_ENTRY
+0x004 TraceCount     : 1           //本回溯的发生次数
+0x008 Index          : 0x23        //记录的索引号
+0x00a Depth          : 0xe         //栈回溯的深度, 即 BackTrace 的元素数
+0x00c BackTrace      : [32] 0x7c96d6dc //从栈帧中得到的函数返回地址数组
```

其中 HashChain 字段指向的是属于同一桶位的下一个记录的地址。因为 BackTrace 数组的长度是 32, 所以栈回溯的最大深度为 32。

建立了 UST 数据库后, 当堆块分配函数再被调用的时候, 堆管理器便会将当前的栈回溯信息记录到 UST 数据库中, 其过程如下。

第一, 堆分配函数调用 RtlLogStackBackTrace 发起记录请求。

第二, RtlLogStackBackTrace 判断 ntdll!RtlpStackTraceDataBase 指针是否为 NULL, 如果是, 则返回; 否则调用 RtlCaptureStackBackTrace。

第三, RtlCaptureStackBackTrace 调用 RtlWalkFrameChain 遍历各个栈帧并将每个栈帧中的函数返回地址以数组的形式返回。

第四, RtlLogStackBackTrace 将得到的信息放入一个 RTL_STACK_TRACE_ENTRY 结构中。然后根据新数据的哈希值搜索是否已记录过这样的回溯记录。如果搜索到, 则返回该项的索引值, 如果没有找到, 则调用 RtlpExtendStackTraceDataBase 将新的记录加入到数据库中, 然后将新加入项的索引值返回。每个 UST 记录都有一个索引值, 我们将其称为 UST 记录索引号。RTL_STACK_TRACE_ENTRY 结构中的 TraceCount 字段用来记录这个栈回溯的发生次数, 如果它的值大于 1, 便说明这样的函数调用过程发生了多次。

第五, 堆分配函数 (RtlDebugAllocateHeap) 将 RtlLogStackBackTrace 函数返回的索引号放入堆块末尾一个名为 HEAP_ENTRY_EXTRA 的数据结构中, 这个数据结构是在分配堆块时就分配好的, 它的长度是 8 个字节, 依次为 2 字节的 UST 记录索引号, 2 字节的堆块标记号 (Tag), 最后 4 字节用来存储用户设置的数值。

可以使用 gflags 工具来配置 UST 数据库的大小, 如下命令便将 heappmfc.exe 程序的 UST 数据库设置为 24MB。

```
gflags /i heappmfc.exe /tracedb 24
```

也可以在注册表中 HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options\heapmfc.exe 表键下直接修改 StackTraceDatabaseSizeInMb 表项 (REG_DWORD) 达到同样的目的。

可以使用 WinDBG 的 dds 命令来观察 UST 回溯记录所对应的调用过程，例如：

```
0:001> dds 0x00410a50
00410a50 00410e9c          // HashChain 字段，不对应任何符号
00410a54 00000001          // TraceCount 字段
00410a58 000e0023          // Index 和 Depth 字段
00410a5c 7c96d6dc ntdll!RtlDebugAllocateHeap+0xe1 // BackTrace 数组的元素 0
00410a60 7c949d18 ntdll!RtlAllocateHeapSlowly+0x44 // BackTrace 数组的元素 1
00410a64 7c91b298 ntdll!RtlAllocateHeap+0xe64    // BackTrace 数组的元素 2
... //省略 11 行 BackTrace 数组所对应的符号
```

但是这样做只适合分析少量的 UST 记录，如果希望自动转储和分析大批的 UST 记录，那么可以使用下面将介绍的软件工具。

23.7.2 DH 和 UMDH 工具

可以使用 DH.EXE (Display Heap) 和 UMDH.EXE (User-Mode Dump Heap) 工具来查询包括 UST 数据库在内的堆信息。尽管这两个工具的用法不尽相同，但它们的基本功能和工作原理是基本一致的。都是利用堆管理器的调试功能将堆信息显示出来或转储 (dump) 到文件中。

这两个工具都是在命令行运行的，通过-p 开关指定要观察的进程。如果要转储 UST 数据库，那么应该先设置好符号文件的路径。

例如，通过以下命令可以将进程 5622 的堆信息转储到文件 DH_5622.dmp 中。

```
C:\>set _NT_SYMBOL_PATH=D:\symbols
C:\>dh -p 5622
```

尽管以.dmp 为后缀，但 DH 生成的文件就是文本文件。内部包含了进程中所有堆的列表和 UST 数据库中的所有栈回溯记录（称为 Hogs）。

可以针对应用程序运行的不同时间点生成多个转储文件，然后使用 dhcmp.exe 工具比较这些文件的差异，利用这种方法可以为定位内存泄漏提供线索。WinDBG 工具包中的 UMDH 将转储和比较的功能都集成在一个工具中，因此更适合使用它来定位内存泄漏。

23.7.3 定位内存泄漏

下面介绍使用 UMDH 来定位内存泄漏的基本步骤。我们以本节的示例程序 HeapMfc 程序 (code\chap23\heapmfc) 为例。

首先，使用 gflags 工具启用 ust 功能，也就是在 HeapMfc.exe 所在的目录中执行

```
gflags /i HeapMfc.exe +ust.
```

然后运行 HeapMfc 程序，并使用 UMDH 工具对其进行第一次采样，即执行 c:\windbg\umdh -p:1228 -d -f:u1.log -v。

点击 HeapMfc 对话框中的 New 按钮，这会导致 HeapMfc 程序分配内存，但是并不释放，也就是模拟一个内存泄漏情况。

第三步，再次执行 UMDH 对程序进行采样：c:\windbg\umdh -p:1228 -d -f:u2.log -v。

最后，使用 UMDH 比较两个采样文件：c:\windbg\umdh -d u1.log u2.log -v，清单 23-17 列出了命令的执行结果和笔者的注释。

清单 23-17 利用 UMDH 工具定位内存泄漏

```
c:\dig\dbg\author\code\bin\release>c:\windbg\umdh -d u1.log u2.log -v
// Debug library initialized ... //加载和初始化符号库，即 DBGHELP.DLL
DBGHELP: HeapMfc - private symbols & lines //加载 HeapMfc 程序的符号文件
      .\HeapMfc.pdb //符号文件路径和名称
DBGHELP: ntdll - public symbols //加载 NTDLL 的符号文件
      d:\symbols\ntdll.pdb\36515FB5D04345E491F672FA2E2878C02\ntdll.pdb
...
// 以下是 UMDH 发现的两次采样间的差异，即可能的内存泄漏线索
+   100 ( 11308 - 11208)    20 allocs    BackTraceA2
+   1 (     20 -      19)    BackTraceA2    allocations
  ntdll!RtlDebugAllocateHeap+000000E1
  ntdll!RtlAllocateHeapSlowly+00000044
  ntdll!RtlAllocateHeap+000000E64
  msvcrt!_heap_alloc+000000E0
  mservt!_nh_malloc+00000013
  mservt!malloc+00000027
  MFC42!operator new+00000031
//归纳结果
Total increase == 100 requested + 28 overhead = 128
```

UMDH 会比较两次采用中的每个 UST 记录，必将存在差异的记录以如下格式显示出来：

- + 字节差异（新字节数 - 旧字节数）新的发生次数 allocs BackTrace UST 记录的索引号
- + 发生次数差异（新次数值 - 旧次数值） BackTrace UST 记录的索引号 allocations 栈回溯列表

在上面的结果中，UMDH 共发现了一个差异，第 9~18 行报告了这一差异的详情。第 9 行的含义是，索引号为 A2 (BackTraceA2) 的 UST 记录在两次采样中新增 100 字节（用户数据区大小），新的字节数为 11308，上次的字节数为 11208。这一记录所代表函数调用过程的发生次数是 20 次。第 10 行的含义是，BackTraceA2 所代表的调用过程在两次采样间新增 1 次，新的发生次数是 20 次，旧的发生次数是 19 次。最后一行的含义是，第 2 次采样比第一次总增加 128 字节，其中 100 字节属于用户数据区（请求长度），28 字节属于堆的管理信息，8 字节为 HEAP_ENTRY 结构，另 20 字节为堆块末尾的自动填充和 HEAP_ENTRY_EXTRA 结构。

23.8 堆溢出和检测

我们在第22章讨论过，如果对分配在栈上的缓冲区写入超出其容量的数据，就可能将存放在栈上的栈帧指针、函数返回地址等信息覆盖掉，即所谓的栈缓冲区溢出。类似的，对于分配在堆上的缓冲区也可能因为访问其分配空间以外的区域而导致溢出，即所谓的堆缓冲区溢出，有时也简称为堆溢出。

23.8.1 堆缓冲区溢出

根据我们前两节的介绍，堆被划分为很多个堆块（chunk），每个堆块又可分为用于管理该堆块的控制数据和该堆块的用户数据区两个部分。对于已经分配（即处于 busy 状态）的堆块，用户数据前面是 HEAP_ENTRY 结构，后面可能有 HEAP_ENTRY_EXTRA 结构。对于空闲的堆块，起始处是一个 HEAP_FREE_ENTRY 结构。因为堆上的用户数据和控制数据是混合存放的，并不是把所有的控制数据放在一个单独的地方特殊保护起来，因此如果访问用户数据区之前或之后的空间都可能将控制数据覆盖掉。

以图 23-3 所示的情况为例，`p0` 是堆块的起始处，`p1` 到 `p2` 是用户数据区，堆分配函数（`HeapAlloc`）会将地址 `p1` 以返回值的形式返回给应用程序，让应用程序通过它来使用用户数据区，假设应用程序使用指针 `pMem` 来记录这个地址。从 `p2` 到 `p3` 是可能存在的放在堆块末尾的附属信息，`p3` 开始便是下一个堆块。正常情况下，应用程序只读写用户数据区，也就是 `pMem` 只在 `p1` 和 `p2` 之间变化。但是因为指针操作不当等原因，`pMem` 可能指向小于 `p1` 空间，这时应用程序便可能破坏当前堆块的控制结构或上一个堆块的数据，更严重时可能破坏堆的段结构（HEAP_SEGMENT）或整个堆的管理结构（HEAP），这种情况通常被称为下溢（Underflow）。如果 `pMem` 指向大于 `p2` 的空间，应用程序便可能破坏放在堆尾的管理信息和下一个堆块的数据，这种情况通常称为上溢（Overflow）。因为上溢的情况发生得更多，因此通常说的溢出都是指上溢，我们也只讨论这种情况。

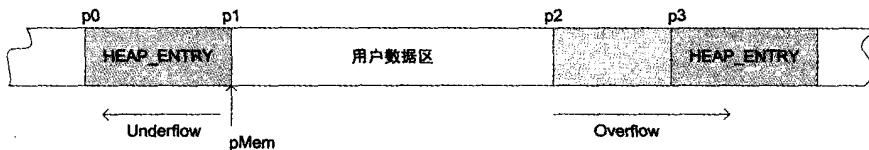


图 23-3 堆缓冲区溢出示意图

下面通过一个小程序来进一步说明，程序的名字叫 `HeapOver`，清单 23-18 给出了它的主要代码。

清单 23-18 演示堆缓冲区溢出的 `HeapOver` 程序

```
1 #include <windows.h>
2
```

```

3  int main(int argc, char* argv[])
4  {
5      char * p1,*p2;
6      HANDLE hHeap;
7      hHeap = HeapCreate(0, 1024, 0);           // 创建一个私有堆
8      p1=(char*)HeapAlloc(hHeap, 0, 9);        // 分配一个用户区长度为 9 字节的堆块
9      for(int i=0;i<50; i++)                  // 循环访问堆块，存在溢出
10         *p1++=i;
11
12     p2=(char*)HeapAlloc(hHeap, 0, 1);        // 再分配一个堆块
13     printf("Allocation after overflow got 0x%x\n",p2);
14     HeapDestroy(hHeap);
15     return 0;
16 }

```

使用 WinDBG 打开 HeapOver.exe（位于 bin\debug\目录），然后执行到第 8 行（停在此行）。观察 hHeap 句柄，其值为 0x00390000。

```
0:000> dd hHeap 11
0012ff74 00390000
```

事实上，这个值就是堆内存区的起始地址，即 HEAP 结构的地址。将这个值作为参数传递给 !heap 便可以观察这个堆的信息。

```

0:000> !heap -a 00390000
...
    Heap entries for Segment00 in Heap 00390000
00390000: 00000 . 00640 [01] - busy (640)    // 用于存放 HEAP 结构
00390640: 00640 . 00040 [01] - busy (40)     // 用于存放段结构
00390680: 00040 . 01818 [07] - busy (1800), tail fill // “前端堆”
00391e98: 01818 . 01168 [14] free fill       // 空闲堆块
00393000:      0000d000 - uncommitted bytes.

```

从上面的结果可以看出堆中已经有四个堆块，三个是占用堆块 (busy)，一个是空闲的堆块。三个占用堆块是堆管理器用于存放管理数据的。空闲堆块的起始地址为 0x00391e98，使用 dt 命令可以得到该块的更多信息：

```
0:000> dt ntdll!_HEAP_FREE_ENTRY 0x00391e98
+0x000 Size          : 0x22d
+0x002 PreviousSize : 0x303
+0x000 SubSegmentCode : 0x0303022d
+0x004 SmallTagIndex : 0xee ''
+0x005 Flags          : 0x14 ''
+0x006 UnusedBytes   : 0xee ''
+0x007 SegmentIndex   : 0 ''
+0x008 FreeList       : _LIST_ENTRY [ 0x390178 - 0x390178 ]
```

如果直接观察该地址所在的内存，那么其内容如图 23-4 的左图所示。也就是除了 16 字节的控制数据外，用户数据区都被填充为 0xFEEEFEEE。

单步执行第 8 行的语句，调用 HeapAlloc 申请分配 9 个字节的缓冲区。执行完该行后，观察返回的指针 p2，其值为 0x00391ea0。

```
0:000> dd p1 11
0012ff7c 00391ea0
```

Memory	样	Memory	样	Memory	样
00391e98	0303022d	00391e98	03030005	00391e98	03030005
00391e9c	00ee14ee	00391e9c	001f075c	00391e9c	001f075c
00391ea0	00390178	00391ea0	bbaad00d	00391ea0	03020100
00391ea4	00390178	00391ea4	bbaad00d	00391ea4	07060504
00391ea8	feeee0ee	00391ea8	abababee	00391ea8	0bba0908
00391eac	feeee0ee	00391eac	abababab	00391eac	0f0e0d0c
00391eb0	feeee0ee	00391eb0	feeee0ab	00391eb0	13121110
00391eb4	feeee0ee	00391eb4	feeee0ee	00391eb4	17161514
00391eb8	feeee0ee	00391eb8	00000000	00391eb8	1b1a1918
00391ebc	feeee0ee	00391ebc	00000000	00391ebc	f1f1ed1c
00391ec0	feeee0ee	00391ec0	00050228	00391ec0	23222120
00391ec4	feeee0ee	00391ec4	00ee14ee	00391ec4	27262524
00391ec8	feeee0ee	00391ec8	00390178	00391ec8	2b2a2928
00391ecc	feeee0ee	00391ecc	00390178	00391ecc	2f2e2d2c
00391ed0	feeee0ee	00391ed0	feeee0ee	00391ed0	feee3130
00391ed4	feeee0ee	00391ed4	feeee0ee	00391ed4	feeee0ee
00391ed8	feeee0ee	00391ed8	feeee0ee	00391ed8	feeee0ee
00391ecd	feeee0ee	00391ecd	feeee0ee	00391ecd	feeee0ee
00391eed	feeee0ee	00391eed	feeee0ee	00391eed	feeee0ee
00391ee4	feeee0ee	00391ee4	feeee0ee	00391ee4	feeee0ee
00391ee8	feeee0ee	00391ee8	feeee0ee	00391ee8	feeee0ee

图 23-4 堆块的空闲状态（左）、已分配状态（中）和溢出后的状态（右）

此时内存窗口的显示的内容变为图 23-4 中间所示的情况，其中，第 1~2 行的前 8 个字节是 HEAP_ENTRY 结构，第 3~4 行的 8 个字节加上第 5 行的第一个字节（0xee）是对管理器分配给应用程序的 9 个字节，之后（第 5~7 行中）是 8 个字节的 0xab，这是堆管理器为了支持溢出检测而多分配的，我们稍后再详细介绍，第 7~8 行中的 0xfee 是堆尾补齐用的未使用字节（Unused Bytes），第 9~10 行的 8 字节是 HEAP_ENTRY_EXTRA 结构，因为没有启用 UST 功能，所以它的值都是 0。第 11 行开始是新的空闲快，也就是说，堆管理器将刚才的空闲块分配一部分满足我们的需要，然后把空闲块的位置向后调整了，再使用!heap 命令观察，也可以看到这一点：

```
0:000> !heap -a 00390000
...
00390680: 00040 . 01818 [07] - busy (1800), tail fill
00391e98: 01818 . 00028 [07] - busy (9), tail fill
00391ec0: 00028 . 01140 [14] free fill
00393000: 00000d000 - uncommitted bytes.
```

倒数第 3 行即刚刚分配的堆块，p1 值指向该块的数据区，00391e98 是该块的 HEAP_ENTRY 结构的地址。这一块的大小为 0x28（40）个字节，即图 23-3 中间靠上方的 40 个字节。倒数第 2 行是新的空闲快信息，其起始位置由前面的 00391e98 变为 00391ec0，大小由 0x01168 变为 0x01140，二者的差刚好是 40（0x28）个字节。

执行到第 12 行，再观察内存窗口，其内容变成了图 23-4 右图所示的样子。因为我们向申请空间为 9 个字节的缓冲区写入了 50 个字节，所以该缓冲区严重溢出，不仅覆盖掉了本堆块堆尾的的内容，而且将本堆块后面空闲堆块的 HEAP_FREE_ENTRY 结构完全覆盖了。此时再执行!heap 命令：

```
0:000> !heap -a 00390000
...
FreeList[ 00 ] at 00390178: 00391ec8 . 00391ec8
00391ec0: 11910 . 10900 [25] - free
Unable to read nt!_HEAP_FREE_ENTRY structure at 2b2a2920
Heap entries for Segment00 in Heap 00390000
00390000: 00000 . 00640 [01] - busy (640)
...
00391ec0: 11910 . 10900 [25] - busy (108da), tail fill, user flags (1)
```

因为空闲块的控制结构被覆盖了，所以关于空闲块的描述完全混乱了。使用 dt 命令观察位于 00391ec0 地址处的 HEAP_FREE_ENTRY（清单 23-19）。

清单 23-19 被破坏了的 HEAP_FREE_ENTRY 结构

0:000> dt ntdll!_HEAP_FREE_ENTRY 00391ec0	
+0x000 Size	: 0x2120
+0x002 PreviousSize	: 0x2322
+0x000 SubSegmentCode	: 0x23222120
+0x004 SmallTagIndex	: 0x24 '\$'
+0x005 Flags	: 0x25 'Unknown format character... character'
+0x006 UnusedBytes	: 0x26 '&'
+0x007 SegmentIndex	: 0x27 ''''
+0x008 FreeList	: _LIST_ENTRY [0x2b2a2928 - 0x2f2e2d2c]

可见，所有信息都失常了，特别是最后 FreeList 的两个指针也被覆盖为无效的值。

单步执行第 12 行，试图从堆上再分配一段内存，会得到访问异常。

```
0:000> p
(d78.1768): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00391ec8 ebx=2f2e2d2c ecx=00002120 edx=00390168 esi=00391ec0 edi=2b2a2928
eip=7c91b3fb esp=0012f80c ebp=0012fa28 iopl=0 nv up ei pl nz ac po nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00010212
ntdll!RtlAllocateHeapSlowly+0x6aa:
7c91b3fb 8b0b          mov    ecx,dword ptr [ebx] ds:0023:2f2e2
```

观察导致异常的指令和寄存器的当前值，ebx 的值为 2f2e2d2c，这正是由于缓冲区溢出而覆盖到 HEAP_FREE_ENTRY 中的值。因为 0x2f2e2d2c 不是一个有效的内存地址，所以访问该内存时发生保护性违例。

事实上，上面的异常是这样导致的。当第 12 行的代码又调用 HeapAlloc 时，堆管理器接到调用后会遍历空闲块链表，寻找满足要求的空闲块，因为这个块的 Flags 字段被错误地标记为占用块（位 0 为 1），所以堆管理器需要通过 Flink 指针访问下一个节点，因此访问到了这个地址。

值得说明的是，如果溢出时覆盖到链表指针处的数据是有效的内存地址，而且堆块的标志仍为空闲，那么堆管理器下次分配堆块时便会从该块分割出一个堆块，然后调整空闲块的位置，这时需要更新链表指针，但因为此时的指针已经指向了被修改过的地址，所以这时就会导致堆管理器向新的地址写入数据。如果精心设计新的地址和写入内容，那么便可能意外修改系统的数据，导致系统执行意外的代码，所谓的堆溢出攻击便是基于类似的原理实施的。

23.8.2 调用时验证

为了及时发现堆中的异常情况，可以让堆管理器在堆函数每次被调用时对堆进行检查，这便是堆的调用时验证（Heap Validation on Call）功能，简称 HVC。

因为验证会影响执行速度，所以 HVC 功能默认是关闭的。如本章 23.6.1 节所介绍的，可以使用 gflags 工具来启用 HVC 功能。例如，如果要对前面的 HeapOver 程序启用 HVC 功能，那么只要执行命令 gflags/i HeapOver.exe+hvc 便可以了。也可以在 WinDBG 调试中通过 !gflag +hvc 命令来启用 HVC 功能。

启用 HVC 功能后，再在 WinDBG 中打开并执行 HeapOver 程序，会得到如下信息：

```
HEAP[HeapOver.exe]: dedicated (0000) free list element 00391EC0 is marked busy
(864.1688): Break instruction exception - code 80000003 (first chance)
eax=00391ec0 ebx=00000000 ecx=7c91eb05 edx=0012f846 esi=00391ec0 edi=00390000
eip=7c901230 esp=0012fa50 ebp=0012fa54 iopl=0 nv up ei pl nz na po nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000
ntdll!DbgBreakPoint:
7c901230 cc int 3
```

这是因为发生堆溢出后又调用 HeapAlloc 函数（第 12 行）时触发了堆管理器的验证功能，该功能报告检测到专用的空闲链表中位于 00391EC0 的元素被标志为占用后，触发断点异常中断到调试器中。使用 kn 命令可以观察到详细的执行过程（清单 23-20）。

清单 23-20 堆的 HVC 功能发现并报告错误

```
0:000> kn
# ChildEBP RetAddr
00 0012fa4c 7c96c943 ntdll!DbgBreakPoint          //触发断点异常
01 0012fa54 7c96d208 ntdll!RtlpBreakPointHeap+0x28 //调用堆的断点触发函数
02 0012fa84 7c96d6a0 ntdll!RtlpValidateHeap+0x43f   //验证堆
03 0012fb04 7c949d18 ntdll!RtlDebugAllocateHeap+0xa5 //支持调试的分配函数
04 0012fd34 7c91b298 ntdll!RtlAllocateHeapSlowly+0x44 //调用慢速的分配函数
05 0012ff68 00401032 ntdll!RtlAllocateHeap+0xe64    //堆块分配函数
06 0012ff80 00401106 HeapOver!main+0x32           //应用程序的入口函数
07 0012ffc0 7c816ff7 HeapOver!mainCRTStartup+0xb4  //编译器插入的入口函数
08 0012fff0 00000000 kernel32!BaseProcessStart+0x23 //系统的进程启动函数
```

其中 RtlpValidateHeap 函数便是用来验证堆的函数，它会执行一系列动作，对堆的头信息、段信息和堆块进行全面检查。

如果不是在调试器中执行，而且启用了 HVC，那么验证函数仍会发现错误，但不会触发断点异常，本次分配会失败，因此 HeapOver 程序会执行到第 13 行（清单 23-18）打印出 Allocation after overflow got 0x0 这样的信息。如果没有启用 HVC 功能，那么执行第 12 行时便会触发访问异常而被操作系统关闭，不会执行到底（第 13 行）。

23.8.3 堆尾检查 (Tail Check)

除了使用 HVC 功能，也可以使用堆尾检查 (Heap Tail Check, HTC) 功能来发现堆溢出。该功能的原理是在每个堆块的用户数据后附加 8 个字节（与粒度值相同）的固定内容模式。如果该内容被破坏，便说明发生了溢出。全局变量 CheckHeapFillPattern 定义了附加在堆块末尾的模式常量，通常为连续的 8 个字节的 0xAB。

```
0:000> dd ntdll!CheckHeapFillPattern 12
7c95dbac abababab abababab
```

一旦启用堆尾检查，那么堆管理器在分配堆块时就会附加 8 字节的 CheckHeapFillPattern。如果要触发堆管理器检查这个模式是否被破坏，那么还要启用其他两种调试检查：释放检查和参数检查。其目的是让堆管理器使用支持调试检查的 Slowly 系列的堆函数，如 RtlFreeHeapSlowly 和 RtlAllocHeapSlowly，我们不妨将它们称为慢速堆函数。慢速堆函数在执行时会调用包含检查功能的释放和分配函数，如 RtlDebugFreeHeap 和 RtlDebugAllocHeap。

如果是在调试器中打开被检查的程序，操作系统的进程加载器会自动启用堆尾检查 (htc)、释放检查 (hfc) 和参数检查功能 (hpc)。但值得说明的是，如果注册表的 Image File Execution Options 子键下存在对该程序的设置，尤其是只有一个空的子键（即存在以程序名命名的子键，但是没有任何键值），那么加载器便可能不再自动启用默认的三项检查。所以应该在调试器中通过 !gflag 扩展命令进行检查确认。如果没有设置，那么可以通过 !gflag 命令进行设置。如 !gflag +htc +hfc +hpc 便为当前进程增加了堆尾检查、释放检查和参数检查。如果要去除某项检查，只要使用减号。注意应该在创建堆之前作修改。

下面通过名为 FreCheck 的小程序为例做进一步说明。该程序的源代码如清单 23-21 所示。

清单 23-21 FreCheck 程序的源代码

```

1 #include <windows.h>
2
3 int main(int argc, char* argv[])
4 {
5     char * p;
6     HANDLE hHeap;
7     hHeap = HeapCreate(0, 1024, 0);
8     p=(char*)HeapAlloc(hHeap, 0, 9);
9     for(int i=0;i<50; i++)
10        *(p+i)=i;
11
12    if(!HeapFree(hHeap, 0, p))
13        printf("Free %x from %x failed.", p, hHeap);
14
15    if(!HeapDestroy(hHeap))
16        printf("Destroy heap %x failed.", hHeap);
17
18    printf("Exit with 0");
19    return 0;
20 }
```

在 WinDBG 中打开 FreCheck.exe (bin\debug 目录)，输入 !gflag 命令检查当前的检查选项。默认应包含 htc、hfc 和 hpc 三项。

```

0:000> !gflag
Current NtGlobalFlag contents: 0x00000070
    htc - Enable heap tail checking
    hfc - Enable heap free checking
    hpc - Enable heap parameter checking
```

单步执行到第 9 行，然后观察返回的用户地址，即指针 p 的值。

```
0:000> dd p l1
0012ff7c 00391ea0
```

使用 dd 命令观察整个堆块：

```
0:000> dd 00391ea0-8
00391e98 03030005 001f078a baadf00d baadf00d
00391ea8 abababee abababab feefeeab feefefeee
00391eb8 00000000 00000000 00050228 00ee14ee
00391ec8 00390178 00390178 feefefeee feefefeee
```

因为申请了 9 个字节，所以从 0x00391ea0 开始的 9 个字节是分配给我们的用户数据区。从 00391ea9 开始的 8 个字节（abababee abababab）便是用于堆尾检查功能的 CheckHeapFillPattern。因为我们是以 DWORD 格式显示的，所以某些字节的顺序是反的，如果是字节格式显示便可以看得更加清楚：

```
0:000> db 00391ea9 l10
00391ea9 ab ab ab ab ab ab ab fe ee fe ee fe ee fe 00 .....
```

从 00391eb1 (00391ea9+8) 开始到 00391eb8 间的 7 个字节是为了满足分配粒度要求而多分配的内容。从 00391eb8 开始的 8 个字节即所谓的额外信息区，即 HEAP_ENTRY_EXTRA 结构，从 00391ec0 开始便是下一个空闲堆块的内容了。

现在使用 !heap 命令观察堆，那么可以看到堆块的归纳信息。

```
0:000> !heap -a 00390000
00391e98: 01818 . 00028 [07] - busy (9), tail fill - unable to read heap entry
extra at 00391eb8
```

按 F5 继续执行，会得到堆管理器的一个调试信息：

```
HEAP[FreCheck.exe]: Heap block at 00391E98 modified at 00391EA9 past requested
size of 9
```

并且程序因为断点异常中断到调试器：

```
(c38.1168): Break instruction exception - code 80000003 (first chance) ...
```

堆管理器打印出的调试信息的意思是堆块 00391E98 的 00391EA9 处被意外篡改了，这超出了用户数据区的长度 9。00391EA9 是用户数据区后的第一个字节，即附加在用户数据区后的 CheckHeapFillPattern 模式的第一字节。观察此地址处的内存值：

```
0:000> db 00391ea9 l10
00391ea9 09 0a 0b 0c 0d 0e 0f 10-11 12 13 ee fe ee fe 00 .....
```

可见，本来的 ab ab ab ab ab... 已经被覆盖成其他值了。

打印栈回溯信息（清单 23-22），可以看到检查的完整过程和发起释放堆块的源程序位置（FreCheck.cpp 的第 16 行）。

清单 23-22 堆的 HTC 功能发现并报告错误

```
0:000> k
ChildEBP RetAddr
0012fcbb 7c96c943 ntdll!DbgBreakPoint
```

```

0012fcbc 7c95db9c ntdll!RtlpBreakPointHeap+0x28
0012fcda 7c96cd11 ntdll!RtlpCheckBusyBlockTail+0x76
0012fce8 7c96df66 ntdll!RtlpValidateHeapEntry+0xa4
0012fd5c 7c94a5d0 ntdll!RtlDebugFreeHeap+0x97
0012fe44 7c9268ad ntdll!RtlFreeHeapSlowly+0x37
0012ff14 00401094 ntdll!RtlFreeHeap+0xf9
0012ff80 004012f9 FreCheck!main+0x84 [C:\...\FreCheck.cpp @ 16]
0012ffc0 7c816fd7 FreCheck!mainCRTStartup+0xe9 [crt0.c @ 206]
0012fff0 00000000 kernel32!BaseProcessStart+0x23

```

即 main 函数调用 HeapFree API，该 API 直接被链接到 NTDLL 中的 RtlFreeHeap。RtlFreeHeap 检查到堆标志中的调试选项，将调用转给 RtlFreeHeapSlowly 函数。接下来 RtlDebugFreeHeap 调用 RtlpValidateHeapEntry 对堆块的完整性进行验证，接下来 RtlpCheckBusyBlockTail 检查到堆块末尾的固定模式被修改，调用 DbgBreakPoint 函数（即 INT 3）发起断点异常，中断到调试器。

除了上面介绍的 HVC 和 HTC，下一节将介绍另一种更强大的堆溢出检查方法。

23.9 页堆

利用堆尾检查可以在释放堆块时发现堆溢出，或者在调用其他函数（比如再次分配内存）时，检查到堆结构被破坏，但这些检查都是滞后的，是在堆溢出发生之后下次再调用堆函数时才检查到的。这样虽然知道了被破坏的堆块，但是仍然很难知道堆块是何时和如何被破坏的，不容易追查出是执行哪段代码时导致的溢出。为了解决这一问题，Windows 2000 引入了用于支持调试的页堆（Debug Page Heap），简称 DPH。一旦启用该机制，那么堆管理器会在堆块后增加专门用于检测溢出的栅栏页（Fence Page），这样一旦用户数据区溢出触及栅栏页便会立刻触发异常。DPH 被包含在 Windows 2000 之后的所有 Windows 版本中，也被加入到 NT 4.0 的 Service Pack 6 中。检查 NTDLL 的调试符号，我们可以看到很多包含 dph 字样的函数，这些函数便是用于实现 DPH 功能的。

23.9.1 总体结构

图 23-5 画出了页堆的结构，其中的地址是以 x86 系统中的一个典型页堆为例的。左侧的矩形是页堆的主体部分，右侧是附属的普通堆。创建每个页堆时，堆管理器都会创建一个附属的普通堆，其主要目的是用来满足系统代码的分配需要，以节约页堆上的空间。

页堆上的空间大多是以内存页来组织的。第一个内存页（起始 4KB）用来伪装普通堆的 HEAP 结构，但大多空间被填充为 0xFFFFFFFF，只有少数字段（Flags 和 ForceFlags）是有效的，这个内存页的属性是只读的，因此可以用于检测到应用程序意外写 HEAP 结构的错误。第二个内存页的开始处是一个 DPH_HEAP_ROOT 结构，该

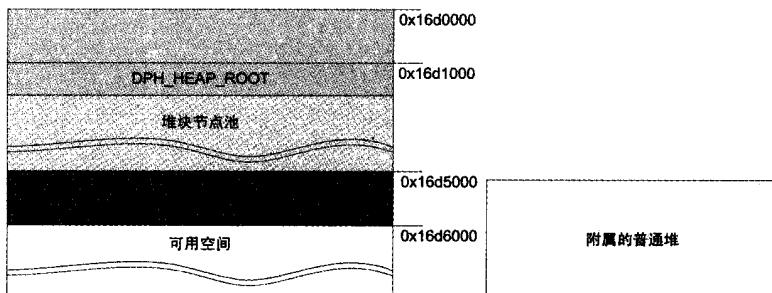


图 23-5 页堆的布局

结构包含了 DPH 堆的基本信息和各种链表，是描述和管理页堆的重要资料。它的第一个字段是这个结构的签名（Signature），固定为 0xffeeddcc，与普通堆结构的签名 0xeeffff 不同。它的 NormalHeap 字段记录着附属普通堆的句柄。

DPH_HEAP_ROOT 结构之后的一段空间用来存储堆块节点，称为堆块节点池（Node Pool）。为了防止堆块的管理信息被覆盖，除了在堆块的用户数据区前面存储堆块信息外，页堆还会在节点池为每个堆块记录一个 DPH_HEAP_BLOCK 结构，简称 DPH 节点结构。多个节点是以链表的形式链接在一起的。DPH_HEAP_BLOCK 结构的 pNodePoolListHead 字段用来记录这个链表的表头，pNodePoolListTail 字段用来记录链表的结尾。它的第一个节点描述的是 DPH_HEAP_ROOT 结构和节点池本身所占用的空间。节点池的典型大小是 4 个内存页（16KB）减去 DPH_HEAP_ROOT 结构的大小。

节点池后的一个内存页用来存放同步用的关键区对象，即 _RTL_CRITICAL_SECTION 结构。这个结构之外的空间被填充为 0。DPH_HEAP_BLOCK 结构的 HeapCritSect 字段记录着关键区对象的地址。

23.9.2 启用和观察页堆

可以全局启用，也可以对某个应用程序启用页堆。以上一节使用过的 FreCheck 程序为例，在命令行中键入命令 gflags /p /enable frecheck.exe /full 或 gflags /i frecheck.exe +hpa 便对这个程序启用了 DPH。以上两个命令都会在注册表中建立子键 HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options\frecheck.exe，并加入如下两个键值：

```
GlobalFlag (REG_SZ) = 0x02000000
PageHeapFlags (REG_SZ) = 0x00000003
```

如果使用第一个命令，那么还会加入以下键值：

```
VerifierFlags (REG_DWORD) = 1
```

在 WinDBG 中打开 frecheck.exe (bin\debug 目录)，输入 !gflag 命令确认已经启用完全的 DPH。

```
0:000> !gflag /p
```

```
Current NtGlobalFlag contents: 0x02000000
hpa - Place heap allocations at ends of pages
```

也可以通过观察全局变量 ntdll!RtlpDebugPageHeap 的值来了解当前进程的 Page Heap 机制是否被启用，如果启用，那么它的值应该为 1。执行到 FreCheck 程序的第 8 行，即创建好堆，观察 hHeap 句柄，记录下它的值 (016d0000)，然后使用 !heap -p 命令显示当前进程的堆列表（见清单 23-23）。

清单 23-23 观察启用 DPH 后的堆概况

```
0:000> !heap -p
Active GlobalFlag bits:
    hpa - Place heap allocations at ends of pages
StackTraceDataBase @ 00430000 of size 01000000 with 00000011 traces
PageHeap enabled with options:
    ENABLE_PAGE_HEAP COLLECT_STACK_TRACES
active heaps:
+ 140000  ENABLE_PAGE_HEAP COLLECT_STACK_TRACES      // DPH 堆
    NormalHeap - 240000                                // 附属的普通堆
        HEAP_GROWABLE
....[省略数行]
+ 16d0000  ENABLE_PAGE_HEAP COLLECT_STACK_TRACES
    NormalHeap - 17d0000
        HEAP_GROWABLE HEAP_CLASS_1
```

“+”号后面的是 Page Heap 句柄，对于每个 DPH 堆，堆管理器还会为其创建一个普通的堆，比如 16d0000 堆的普通堆是 17d0000。如果在 !heap 命令中不包含 /p 参数，那么列出的堆中只包含每个 DPH 的普通堆，不包含 DPH 堆。如果要观察某个 DPH 堆的详细信息，那么应该在 !heap 命令中加入 -p 开关，并用 -h 来指定 DPH 堆的句柄（见清单 23-24）。

清单 23-24 观察页堆的详细信息

```
0:000> !heap -p -h 16d0000
_DPH_HEAP_ROOT @ 16d1000          // DPH_HEAP_ROOT 结构的地址
Freed and decommitted blocks     // 释放和已经归还给系统的块列表
DPH_HEAP_BLOCK : VirtAddr VirtSize // 列表的标题行，目前内容为空
Busy allocations                 // 占用（已分配）的块
DPH_HEAP_BLOCK : UserAddr UserSize - VirtAddr VirtSize // 列表的标题行
_HEAP @ 17d0000                  // 普通堆的句柄，亦即 HEAP 结构的地址
_HEAP_LOOKASIDE @ 17d0688         // 旁视列表（“前端堆”）地址
_HEAP_SEGMENT @ 17d0640           // 段结构地址
CommittedRange @ 17d0680          // 已提交区域的起始地址
HEAP_ENTRY Size Prev Flags      UserPtr UserSize - state // 普通堆上的堆块列表
* 017d0680 0301 0008 [01] 017d0688 01800 - (busy)
  017d1e88 022f 0301 [10] 017d1e90 01170 - (free)
VirtualAllocdBlocks @ 17d0050     // 直接分配的大虚拟内存块列表表头
```

可见此时页堆上还没有分配任何用户堆块，普通堆上只有一个管理堆块。

23.9.3 堆块结构

与普通堆块相比，页堆的堆块结构也有很大不同。首先，每个堆块至少占用两个

内存页，在用于存放用户数据的内存页后面，堆管理器总会多分配一个内存页，这个内存页是专门用来检测溢出的，我们将其称为栅栏页（Fence Page）。栅栏页的工作原理与我们在第 22 章介绍的用于实现栈自动增长的保护页相似。栅栏页的页属性被设置为不可访问（PAGE_NOACCESS），因此，一旦用户数据区发生溢出触碰到栅栏页时便会引发异常，如果程序在被调试，那么调试器便会立刻收到异常，使调试人员可以在第一现场发现问题，从而迅速定位到导致溢出的代码。为了及时检测溢出，堆块的数据区是按着紧邻栅栏页的原则来布置的，以一个用户数据大小远小于一个内存页的堆块为例，这个堆块会占据两个内存页，数据区在第一个内存页的末尾，第二个内存页紧邻在数据区的后面，图 23-6 画出了一个这样的一个页堆堆块的数据布局。

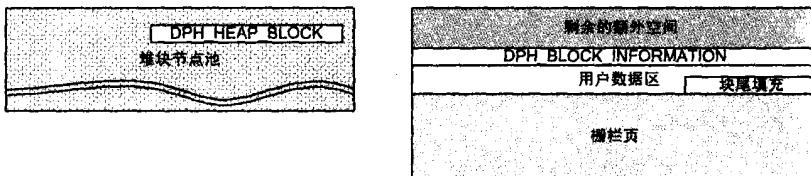


图 23-6 页堆堆块的数据布局

页堆堆块的数据区由三个部分组成，起始处是一个固定长度的 DPH_BLOCK_INFORMATION 结构，我们将其称为页堆堆块的头结构；中间是用户数据区；最后是用于满足分配粒度要求而多分配的额外字节。如果应用程序申请的长度（即用户数据区的长度）正好是分配粒度的倍数，比如 16 字节，那么第三部分就不存在了。除了以上三个部分，对于每个页堆堆块，在页堆的堆块节点池中还会有一个 DPH_HEAP_BLOCK 结构，即我们前面曾提到的 DPH 节点结构。

下面以一个实际的页堆堆块为例来详细描述以上结构。仍然是用前面使用的 FreCheck 程序，单步执行第 8 行代码，从堆上分配一段内存：

```
p=(char*)HeapAlloc(hHeap, 0, 9);
```

观察返回的指针，其值为 016d6ff0，把这个地址与页堆和它所配套的普通堆的地址（分别为 0x16d0000 和 0x17d0000）比较，可以推测出这个堆块是从页堆上分配的。把用户数据区的地址减去 DPH_BLOCK_INFORMATION 结构的大小（32 字节）便得到页堆堆块的头结构地址，然后便可以使用 dt 命令来观察这个结构的内容（清单 23-25）。

清单 23-25 页堆堆块的头结构

0:000> dt ntdll!_DPH_BLOCK_INFORMATION 016d6ff0-20
+0x000 StartStamp : 0xabcdabbb //头结构的起始签名，固定为这个值
+0x004 Heap : 0x016d1000 //DPH_HEAP_ROOT 结构的地址
+0x008 RequestedSize : 9 //堆块的请求大小（字节数）
+0x00c ActualSize : 0x1000 //堆块的实际字节数，不包括栅栏页
+0x010 FreeQueue : _LIST_ENTRY [0x12 - 0x0] //释放后使用的链表结构
+0x010 TraceIndex : 0x12 //在 UST 数据库中的追踪记录序号
+0x018 StackTrace : 0x00346a60 //指向 RTL_TRACE_BLOCK 结构的指针
+0x01c EndStamp : 0xdcbabbbb //头结构的结束签名，固定为这个值

为了方便地检验 DPH_BLOCK_INFORMATION 结构的完好性，其起始 4 个字节（StartStamp）和最后 4 个字节（EndStamp）都是固定的模式值，分别称为 Start Magic 和 End Magic。表 23-7 列出了堆管理器对页堆堆块不同区域的填充数据，第 2 列是针对占用堆块的，第 3 列是针对空闲堆块的。

表 23-7 页堆堆块的填充模式

	占用堆块	空闲堆块
头结构的 Start Magic	ABCDDBBB	ABCDBBBA
头结构的 End Magic	DCBABBBBB	DCBABBBBA
用户区	C0(或者 00, 如果用户要求初始化为 0)	F0
用户数据后的填充部分	D0	N/A

使用 dd 命令直接观察堆块附近的数据，可以看到堆管理器所填充的信息：

```
0:000> dd 016d6fc0
016d6fc0 00000000 00000000 00000000 00000000
016d6fd0 abcdbbb 016d1000 00000009 00001000
016d6fe0 00000012 00000000 00346a60 dcba bbbb
016d6ff0 c0c0c0c0 c0c0c0c0 d0d0d0c0 d0d0d0d0
016d7000 ??????? ??????? ??????? ??????? ????????
```

其中，第 1 行是内存页中没有使用的空闲数据，因为页堆要保证用户数据位于内存页的末尾，所以前面通常会空闲一些空间，第 2 行开始是 32 字节的 DPH_BLOCK_INFORMATION 结构，即清单 23-25 中所显示的数据。第 4 行的前 9 个字节是用户数据区，被填充为 c0，如果我们在调用 HeapAlloc 时指定 HEAP_ZERO_MEMORY 标志，那么用户区会被初始化为 0。第 4 行后面的 7 个字节是为了满足分配粒度（8 字节）而用于补齐的数据，被填充为 d0。第 5 行属于不可访问的栅栏页，所以被显示为问号。

再次执行 !heap -p -h 16d0000 命令，可以看到页堆的占用堆块列表中出现了一项：

```
0:000> !heap -p -h 16d0000
.....
Busy allocations
DPH_HEAP_BLOCK : UserAddr UserSize - VirtAddr VirtSize
016d110c : 016d6ff0 00000009 - 016d6000 00002000
....
```

这一行正对应于刚才所申请的内存，第 1 列是专门用于描述页堆堆块的 DPH_HEAP_BLOCK 结构地址，第 2 列是用户地址，即 HeapAlloc 返回的地址，第 3 列是用户数据区的长度，即 9 个字节，第 4 列是该堆块的起始地址（虚拟地址）。第 5 列是该堆块所占用的虚拟内存大小，0x2000 即 8KB。可见，为了满足 9 个字节的内存需求，页堆实际使用了 8KB，外加存放在堆块节点池中的（地址 016d110c 处）DPH_HEAP_BLOCK 结构所占的空间。堆块的前 4KB（016d6000~016d6FFF）的大多数空间没有使用，后 4KB（016d7000~016d7FFF）用作栅栏页。如果使用 dd 命令显示栅栏页的内容，会发现都是??，这是因为栅栏页具有不可访问属性，调试器无法访问这个空间。如果使用内存观察窗口观察，那么会得到如下错误信息：

Unable to retrieve information, Win32 error 30: The system cannot read from the specified device.

使用 !address 命令观察后栅栏页所对应空间的状态和属性信息，可以看到：

```
0:000> !address 016d7000
016d0000 : 016d7000 - 000f9000
    Type      00020000 MEM_PRIVATE
    Protect   00000001 PAGE_NOACCESS      // 保护属性
    State     00001000 MEM_COMMIT        // 已经提交
    Usage     RegionUsagePageHeap      // 用于 DPH
    Handle    016d1000 //_DPH_HEAP_ROOT 结构地址
```

其中第一行的含义是，第一个数字是所观察地址所属的较大内存区，第二个数字是这个较大内存区的较小区域，第三个数字是区域的总大小。

下面我们再观察 DPH_HEAP_BLOCK 结构，从刚才列出的 Busy allocations 列表中取出第一列的地址值便可以使用 dt 命令来观察了（见清单 23-26）。

清单 23-26 页堆堆块的节点结构

```
0:000> dt ntdll!_DPH_HEAP_BLOCK 016d110c -r
+0x000 pNextAlloc      : (null)
+0x004 pVirtualBlock   : 0x016d6000 ""      // 用于该堆块的内存页起始地址
+0x008 nVirtualBlockSize : 0x2000          // 用于该堆块的总空间大小 8KB
+0x00c nVirtualAccessSize : 0x1000          // 可访问区域大小 4KB
+0x010 pUserAllocation : 0x016d6ff0 "???" // 用户区起始地址
+0x014 nUserRequestedSize : 9              // 用户请求大小，字节为单位
+0x018 nUserActualSize   : 0x10            // 用户区实际大小，字节为单位
+0x01c UserValue        : (null)           // 供应用程序使用的用户数据
+0x020 UserFlags         : 0                // 供应用程序使用的用户标志
+0x024 StackTrace       : 0x00346a30 _RTL_TRACE_BLOCK // 栈回溯信息
```

其中 StackTrace 指向的是用于记录分配这个堆块的栈回溯信息（函数调用序列）的 _RTL_TRACE_BLOCK 结构，可以使用 dds 命令将 Trace 数组中的函数返回地址值翻译为符号显示出来：

```
0:000> dds 0x00346a50 14
00346a50 7c91b298 ntdll!RtlAllocateHeap+0xe64
00346a54 00401053 FreCheck!main+0x43 [C:\...\FreCheck.cpp @ 12]
00346a58 00401309 FreCheck!mainCRTStartup+0xe9 [crt0.c @ 206]
00346a5c 7c816fd7 kernel32!BaseProcessStart+0x23
```

在前面的堆块头结构中也有一个 StackTrace 字段，它指向的也是 _RTL_TRACE_BLOCK 结构。二者的差异是记录的时间不同，节点结构中记录的是创建节点时的栈回溯，栈顶的函数是 RtlAllocateHeap；头结构记录的是创建堆块时的栈回溯，栈顶的函数是 RtlAllocateHeapSlowly。

23.9.4 检测溢出

对页堆有了比较深刻的理解后，我们来看一下使用它检测堆溢出的效果。我们知道在 FreCheck 程序中包含了故意设计的溢出，因此按 F5 继续执行 FreCheck 程序，会

发现 g 命令执行后 FreCheck 程序立刻又中断回调试器中：

```
0:000> g
(172c.14f8): Access violation - code c0000005 (first chance) ...
eax=016d6f10 ebx=7ffd9000 ecx=00000010 edx=016d7000 ...
FreCheck!main+0x6b:
0040107b 8802        mov     byte ptr [edx],al      ds:0023:016d7000=??
```

从以上信息可以看出，应用程序中发生了访问异常（Access violation）。导致异常的指令是最后一行所示的 MOV 指令，位于 main 函数的偏移 0x6b 处。MOV 指令的目标操作数是 edx 所代表的地址，即 016d7000，这正是位于我们前面分析看到的栅栏页面的起始地址。

于是可以推测出，是由于 main 函数中的 for 循环越界访问到了用户数据区后的栅栏页面，从而导致了异常并中断到调试器中。

```
for(int i=0;i<20; i++)
    *p+++=i;
```

从 EAX 寄存器的值，可以知道 al 的值为 0x10，即 16，也就是变量 i 此时的值为 16。这说明 for 循环在执行到第 17 次时触及到了栅栏页，而引发了异常。可见，通过页堆的栅栏页我们可以在堆缓冲区发生溢出的第一时间得到通知，观察到发生溢出的现场，这对于定位导致溢出的根本原因是非常有效的。

23.10 准页堆

因为使用页堆要为每个堆块至少多分配一个内存页，因此要多使用大量的内存，这对于内存密集型的应用程序来说可能是不可行的。对于这样的情况，可以让堆管理器从页堆的附属普通堆中分配堆块，这样可以大大减少内存占用量，又可以部分发挥页堆的调试功能。因为这样分配的堆块不再有专门的栅栏页，因此我们将这种页堆称为准页堆。MSDN 的文档将准页堆称为常规页堆（Normal Debug Page Heap），简称常规 DPH，将页堆称为完全 DPH。

常规 DPH 不再为每个堆块分配栅栏页，只是在堆块前后增加类似于安全 Cookie 的附加标记。当释放堆块时，DPH 会检测这些标记的完好性，如果这些标记被破坏，那么这个堆块便发生过溢出。检测出溢出后，DPH 会产生一个断点异常，试图报告给调试器。因此，与完全的 DPH 一样，为了接收到 DPH 报告的信息，被检查的应用程序也应该在调试器中运行。调试器可以是任何类型的调试器，如 WinDBG、NTSD、VS 等。

23.10.1 启用准页堆

可以使用 gflags 命令对指定的程序启用准页堆，如 gflags /p /enable frecheck.exe。

这样执行后，观察注册表，会发现注册表中 `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options\fcheck.exe` 表键下存在如下键值：

```
GlobalFlag (REG_SZ) = 0x02000000
PageHeapFlags (REG_SZ) = 0x00000002
VerifierFlags (REG_DWORD) = 0x8000
```

注意 `GlobalFlag` 的值与启用完全页堆是一样的，但是 `PageHeapFlags` 的值由 3 变为 2。也可以使用 `gflags /r +hpa` 或 `gflags /k +hpa` 对整个系统内的所有应用程序启用常规 DPH 机制（需要重新启动后才能生效）。

使用 `gflags /p` 可以列出当前的 DPH 设置信息：

```
C:\>gflags /p
path: SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options
      fcheck.exe: page heap enabled with flags (traces)
      heapover.exe: page heap enabled with flags (full traces)
```

其中，(traces) 表示启用了准页堆 (Normal DPH)，(full traces) 表示启用了完全页堆 (Full DPH)。

在 WinDBG 中打开 `FreCheck` 程序，执行 `!gflag -p` 命令或观察全局变量 `ntdll!RtlpDebugPageHeap`，其结果与前面完全 DPH 时一样。但是 `!heap -p` 命令显示出的 `PageHeap` 选项中没有了 `ENABLE_PAGE_HEAP`：

```
PageHeap enabled with options:
    COLLECT_STACK_TRACES
```

因此可以通过这一差异来在 WinDBG 中判断当前使用的是完全的 DPH 还是常规的 DPH。

23.10.2 结构布局

准页堆的总体布局与页堆是一样的，也就是仍保持着图 23-5 中的结构。但是因为要从常规堆分配堆块，所以准页堆的堆块结构变化了，而且节点池中不再为其分配节点结构。图 23-7 画出了准页堆堆块的数据布局。首先是普通堆块所需要的 `HEAP_ENTRY` 结构，而后是页堆堆块的头结构，即 `DPH_BLOCK_INFORMATION`，然后是用户数据区。用户数据区后是用来检测缓冲区溢出的栅栏字节 (8 或 16 个字节长)，栅栏字节的内容固定为 `0xA0`。栅栏字节后是满足分配粒度要求的补位字节，最后是用于存放附加信息的 `HEAP_ENTRY_EXTRA` 结构。

为了与页堆堆块相区别，准页堆堆块的填充模式有所不同。表 23-8 列出了两种页堆使用的填充模式。

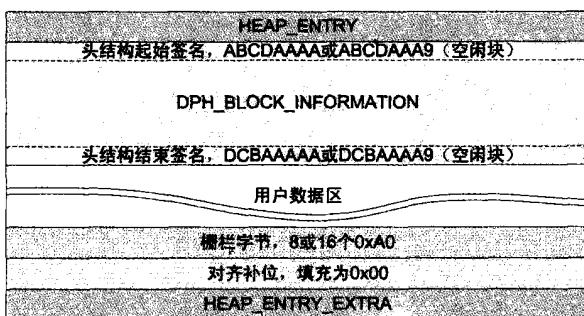


图 23-7 准页堆堆块的数据布局

表 23-8 页堆堆块和准页堆堆块的填充模式

	占用堆块	空闲堆块	页堆	准页堆
所属	页堆	准页堆	页堆	准页堆
头结构的起始签名	ABCDBBBB	ABCDAAAA	ABCDAAA9	ABCDBBBA
头结构的结束签名	DCBABBBB	DCBABBBB	DCBAAAA9	DCBABBBB
用户区	C0	E0	F0	F0
栅栏字节	N/A	A0	N/A	N/A
补齐字节	D0	00	N/A	N/A

如果分配（调用 `HeapAlloc`）时指定了 `HEAP_ZERO_MEMORY`，那么用户区会被填充为 0。

下面仍以 `FreCheck` 程序为例来介绍准页堆的细节。在 WinDBG 中打开这个程序让其执行到第 8 行，创建一个新的堆，然后使用 `!heap -p` 命令观察堆的概况，其结果与清单 23-23 非常相似，每个堆仍由页堆和附属的普通堆两部分组成。

```
0:000> !heap -p
...
+ 16d0000 //省略了概要信息和其他堆的信息
          //我们创建的页堆，即 hHeap 变量的值
  COLLECT_STACK_TRACES //堆的启用选项
  NormalHeap - 17d0000 //附属的普通堆句柄（基址）
  HEAP_GROWABLE HEAP_CLASS_1 //附属堆的属性
```

执行完第 9 行，即分配一个堆块，然后观察指针 `p` 的值：

```
0:000> dd p 11
0012ff7c 017d1eb0
```

这便是新堆块的用户数据区起始地址，把这个地址值和页堆的基址（`0x16d0000`）和普通堆的基址（`0x17d0000`）相比较，可以分析出这个堆块是从普通堆上分配的。这正体现了完全 DPH 和常规 DPH 的最主要区别：完全 DPH 从它的页堆分配用户堆块，常规 DPH 从它的普通堆中分配用户堆块，因此常规 DPH 比完全的 DPH 的开销要小得多。使用 `!heap -p -h 16d0000` 命令显示堆信息，也可以看出这一差异：

```
0:000> !heap -p -h 16d0000
  _DPH_HEAP_ROOT @ 16d1000
...
  HEAP_ENTRY Size Prev Flags     UserPtr UserSize - state
* 017d0680 0301 0008 [01]    017d0688 01800 - (busy) // "前端堆"
```

```
017d1e88 0009 0301 [03] 017d1eb0 00009 - (busy) // 用户堆块
017d1ed0 0226 0009 [10] 017d1ed8 01128 - (free) // 空闲堆块
VirtualAllocdBlocks @ 17d0050
```

倒数第3行所代表的堆块就是我们刚才所分配的，017d1e88是这个堆块的起始地址，也就是HEAP_ENTRY结构的地址。第2列的0009是本堆块的大小（以粒度为单位），0301是上一堆块的大小，017d1eb0是用户数据区的起始地址，也就是用户指针的值，第6列的00009是用户请求的大小（字节数）。使用dt命令观察017d1e88地址可以得到同样的信息：

```
0:000> dt ntdll!_HEAP_ENTRY 017d1e88
+0x000 Size : 9 // 本堆块大小，以粒度为单位，因此是72字节
+0x002 PreviousSize : 0x301 // 前一堆块大小，以粒度为单位
+0x000 SubSegmentCode : 0x03010009
+0x004 SmallTagIndex : 0x7b '{' // 堆块标记
+0x005 Flags : 0x3 '' // 标志
+0x006 UnusedBytes : 0x17 '' // 未使用字节数
+0x007 SegmentIndex : 0 '' // 所在段序号
```

其中未使用字节数（UnusedBytes）等于0x17（23）是因为用户数据后有8个用于检测溢出的栅栏字节，补位用的7字节，再加上8字节的额外信息，即HEAP_ENTRY_EXTRA结构。使用dd命令观察堆块的附近内存区，可以看到每一部分的原始数据：

```
0:000> dd 017d1e88
017d1e88 03010009 0017037b abcdaaaa 816d1000
017d1e98 00000009 00000031 00000012 00000000
017d1ea8 00346728 dcbaaaaa e0e0e0e0 e0e0e0e0
017d1eb8 a0a0a0e0 a0a0a0a0 000000a0 00000000
017d1ec8 00000000 00000000 00090226 00001000
017d1ed8 017d0178 017d0178 00000000 00000000
```

第1行的前8个字节即HEAP_ENTRY结构，之后的32字节是DPH_BLOCK_INFORMATION结构。从017d1eb0开始的9个字节是用户数据区，被使用e0填充。017d1eb9开始的8个字节a0是栅栏字节，又称后缀模式（Suffix Pattern）。而后的7字节00是补位使用的。017d1ec8开始的8个字节是额外信息区（HEAP_ENTRY_EXTRA），从017d1ed0开始是后面的空闲块。可以使用dt命令来观察堆块的头结构（见清单23-27）。

清单23-27 准页堆堆块的头结构

```
0:000> dt ntdll!_DPH_BLOCK_INFORMATION 017d1e90
+0x000 StartStamp : 0xabcdaaaa // 堆块头结构的起始签名
+0x004 Heap : 0x816d1000 // 堆的根结构地址
+0x008 RequestedSize : 9 // 请求大小
+0x00c ActualSize : 0x31 // 实际大小
+0x010 FreeQueue : _LIST_ENTRY [ 0x12 - 0x0 ] // 释放后使用的链表结构
+0x010 TraceIndex : 0x12 // 回溯记录在UST数据库中的序号
+0x018 StackTrace : 0x00346728 // 分配过程的栈回溯
+0x01c EndStamp : 0xdcdaaaa // 堆块头结构的结束签名
```

其中的实际大小（ActualSize）字段是指HEAP_ENTRY结构之后到后缀模式结束间的总字节数。以本块为例，即地址017d1e90到017d1ec1间一共有49个字节，分别是：

DPH_BLOCK_INFORMATION结构（32字节）+ 用户请求大小（9）+ 栅栏字节（8） = 49

StackTrace 指向的是 _RTL_TRACE_BLOCK 结构，因此可以使用 dt NTDLL!_RTL_TRACE_BLOCK 0x00346728 命令显示它的各个字段值，使用 dds 0x00346728 命令可以显示它所记录的函数调用过程。

23.10.3 检测溢出

如果分配在准页堆上的缓冲区发生溢出，那么用户数据后的栅栏字节就会被破坏，因此可以通过检查栅栏字节的完好性来检查是否发生溢出。当页堆的释放函数释放一个堆块时，它会检查栅栏字节的完好性，如果发现其中的任一个字节发生变化，那么便触发断点异常，向调试器报告。

以 FreCheck 程序为例，for 循环对 9 字节的缓冲区写入 20 个字节的数据，显然导致了溢出，但因为准页堆不能像页堆那样立刻检测到溢出，所以 for 循环可以全部执行完，但当调用 HeapFree 释放这个发生溢出的堆块时，调试器中会出现如下信息：

```
=====
VERIFIER STOP 00000008: pid 0x220: corrupted suffix pattern
 016D1000 : Heap handle
 017D1EB0 : Heap block
 00000009 : Block size
 00000000 :
=====
(220.1778): Break instruction exception - code 80000003 (first chance)...
```

并且程序因为断点异常中断到调试器。使用 k 命令观察栈回溯信息（清单 23-28），可以看出是在 main 函数调用 RtlFreeHeap 释放堆块时，DPH 检查到了堆中的溢出错误。

清单 23-28 准页堆的工作函数检测和报告溢出的过程

```
0:000> k
ChildEBP RetAddr
0012fbe0 7c9551ad ntdll!DbgBreakPoint          // 执行断点指令，触发断点异常
0012fbf8 7c969b7e ntdll!RtlApplicationVerifierStop+0x160 // 验证器的停止函数
0012fc74 7c96ac57 ntdll!RtlpDphReportCorruptedBlock+0x17c // 报告损坏堆块
0012fc98 7c96ae5a ntdll!RtlpDphNormalHeapFree+0x2e // 释放准页堆堆块的 DPH 函数
0012fce8 7c96defb ntdll!RtlpDebugPageHeapFree+0x79 // 堆释放函数
0012fd5c 7c94a5d0 ntdll!RtlDebugFreeHeap+0x2c      // 支持调试的堆释放函数
0012fe44 7c9268ad ntdll!RtlFreeHeapSlowly+0x37     // 慢速的堆块释放函数
0012ff14 00401094 ntdll!RtlFreeHeap+0xf9           // 释放堆块的入口函数
0012ff80 004012f9 FreCheck!main+0x84 [C:\...\FreCheck.cpp @ 16] // 应用程序入口
0012ffc0 7c816fd7 FreCheck!mainCRTStartup+0xe9 [crt0.c @ 206]
0012fff0 00000000 kernel32!BaseProcessStart+0x23    // 系统的进程启动函数
```

也可以用 PageHeap 工具（pageheap.exe）来启用页堆和准页堆。目前，PageHeap 工具已经成为应用程序验证工具（Application Verifier）的一部分。事实上，启用应用程序验证器中的内存验证功能，就会启用页堆机制，我们在第 19 章详细讨论过应用程序验证的工作原理。

23.11 CRT 堆

软件开发中，我们经常使用 C 库函数中的内存分配函数（`malloc` 等），或者 C++ 的 `new` 运算符来分配内存。在第 23.1 节我们简要介绍过，这两种方式实际上都是从 CRT 堆上分配内存空间的。顾名思义，CRT 堆就是指 C/C++ 的运行库（CRT）所使用的堆。因为 `new` 操作符也会被编译为对 CRT 函数的调用，所以下文将使用 CRT 内存分配函数来泛指 C 的内存分配函数和 C++ 的 `new` 运算符。

23.11.1 CRT 堆的三种模式

根据分配堆块的方式不同，CRT 堆有三种工作模式：系统模式、SBH 模式和旧 SBH 模式。CRT 使用以下三个常量（定义在 `winheap.h`）分别代表这三种模式：

```
#define __SYSTEM_HEAP      1    //系统模式
#define __V5_HEAP           2    //旧 SBH 模式
#define __V6_HEAP           3    //SBH 模式
```

系统模式（`__SYSTEM_HEAP`）的含义是创建一个标准的 Win32 堆，并从中分配堆块，也就是将堆块分配和释放请求一一转发给系统的堆管理器。SBH 模式（`__V6_HEAP`）的含义是使用 VC6（Vistaul C++ 6.0）改进过的 Small Block Heap（简称 SBH）方式来分配和管理堆块。旧 SBH 模式（`__V5_HEAP`）的含义是使用 VC++ 5.0 的 Small Block Heap 方式来分配和管理堆块，为了与 VC6 的 SBH 相区别，VC5 的 SBH 被称为旧 SBH，其声明和函数名中通常都带有 `old_sbh` 字样。

CRT 初始化时，会选择以上三种模式之一，并将其保存在全局变量 `_active_heap` 中。

为了隐藏堆块管理模式的差异，CRT 定义了一组基础函数把堆的底层管理方式与上层隔离开来。例如，用来分配堆块的基础函数是 `_heap_alloc`（调试版本使用别名 `_heap_alloc_base`），释放堆块的基础函数是 `_free_base`，重新分配堆块的是 `_realloc_base`，为了行文方便，我们将这些函数简称为 CRT 堆基础函数。

CRT 堆基础函数会根据 `_active_heap` 变量来判断当前的工作模式，并将应用程序的请求分发给合适的底层函数。如果使用系统模式，那么基础函数会将应用程序的分配和释放请求转发给系统堆的 API，即 `HeapAlloc`、`HeapFree` 等。这时的 CRT 分配函数相当于是对系统堆 API 的一种封装。如果使用 SBH 模式，那么内存分配和释放请求会被转发给专门设计的分配和释放函数，这些函数都以 `_sbh` 开头，如 `_sbh_alloc_block`、`_sbh_free_block` 等。如果使用旧 SBH 模式，那么堆工作函数都是以 `_old_sbh` 开头的，如 `_old_sbh_alloc_block`、`_old_sbh_free_block` 等。清单 23-29 显示了 CRT 堆基础函数将应用程序对 `malloc` 函数的调用分发给 SBH 的堆块分配函数 `_sbh_alloc_block` 的过程。

清单 23-29 CRT 堆基础函数将分配请求转发给 SBH 函数

```

0:000> kn //VC6 调试版本
# ChildEBP RetAddr
00 0012fef0 00405381 HiHeap!__sbh_alloc_block [sbheap.c @ 522] //SBH 分配函数
01 0012ff00 004018c2 HiHeap!_heap_alloc_base+0x21 [malloc.c @ 158] //堆基础函数
02 0012ff28 004016c9 HiHeap!_heap_alloc_dbg+0x1a2 [dbgheap.c @ 378]
03 0012ff44 0040167f HiHeap!_nh_malloc_dbg+0x19 [dbgheap.c @ 248] //支持 New Handler
04 0012ff64 004092d0 HiHeap!_malloc_dbg+0x1f [dbgheap.c @ 165] //调试版本的 malloc
05 0012ff8c 0040378f HiHeap!_setenvp+0xe0 [stdenvp.c @ 122] //设置环境变量
06 0012ffc0 7c816ff7 HiHeap!mainCRTStartup+0xb [crt0.c @ 178] //CRT 入口函数
07 0012ffff 00000000 kernel32!BaseProcessStart+0x23 //进程启动函数

```

第 23.3 节的清单 23-3 和清单 23-4 描述了将 `malloc` 函数和 `new` 运算符分发给系统堆 API (`RtlAllocHeap`) 的过程。

23.11.2 SBH 简介

图 23-8 画出了一个以 SBH 模式工作的 CRT 堆（以下简称 SBH 堆）的结构布局。概括说来，每个 SBH 堆由一个附属的普通 Win32 堆和不确定数量的 SBH 区域（Region）组成。Win32 堆用来存储区域的管理信息和分配大小超过 SBH 阈值的堆块。SBH 区域用来分配小于 SBH 阈值的堆块。SBH 阈值是由全局变量 `_sbh_threshold` 所记录的，初始值为 `0x3F8`（1016）。每个 SBH 区域又划分为 32 个组（Group），每个组包含 8 个内存页，每个内存页分为 256 个段落（Paragraph），每个段落的大小是 16 个字节。段落是 SBH 中的最小分配单位。以上常量定义在 `winheap.h` 文件中，可以调整，但调整后需要重新编译运行库。

```

#define BYTES_PER_PARA    16      //每个段落的字节数
#define PARAS_PER_PAGE    256     //每个页的段落数
#define PAGES_PER_GROUP   8       //每个组的页数，可以调整
#define GROUPS_PER_REGION 32      //每个区域的组数，可以调整，但最大为 32

```

每个 SBH 区域在普通堆中都有一个 HEADER 结构和一个 REGION 结构。多个区域的 HEADER 结构是以数组形式存放的，数组的初始元素个数是 16 个，但用光后可以重新分配并增大（调用 `HeapReAlloc`）。全局变量 `_sbh_pHeaderList` 用来记录这个数组的起始位置，`_sbh_sizeHeaderList` 记录着目前的总元素数，`_sbh_cntHeaderList` 记录着已经使用的元素数。HEADER 结构的定义如下：

```

typedef struct tagHeader
{
    BITVEC          bitvEntryHi;      //可用空间位向量的高位部分
    BITVEC          bitvEntryLo;      //可用空间位向量的低位部分
    BITVEC          bitvCommit;       //区域中各个组的提交情况
    void *          pHeapData;        //堆块数据区的起始地址
    struct tagRegion * pRegion;      //REGION 结构的指针
} HEADER, *PHEADER;

```

其中，`pHeapData` 用来记录该区域的堆块数据区地址，这个地址是通过 `VirtualAlloc` 从虚拟内存管理器申请的。`pRegion` 指向这个区域的 REGION 结构，REGION 结构是

使用 HeapAlloc API 从普通堆上分配的。

bitvEntryHi 和 bitvEntryLo 用来标志区域中可用的空闲空间多少，这两个字段共有 64 个二进制位，一个非零位代表一个空闲块，例如，bitvEntryHi 的位 0 为 1 代表有长度为 1 个段落的空闲块，位 1 为 1 代表有两个段落长的空闲块，依此类推，bitvEntryLo 的位 0 为 1，代表有 33 个段落的空闲快，位 31 为 1，代表有至少为 64 个段落长的空闲块。对于一个新建的区域，bitvEntryHi 等于 0，bitvEntryLo 等于 1，表示区域中有一个长度大于 64 个段落的空闲块。当分配堆块时，SBH 会根据堆块的总大小（按段落大小对齐）计算出堆块的索引号，然后根据这个索引号来寻找可以满足需要的区域（见图 23-8）。

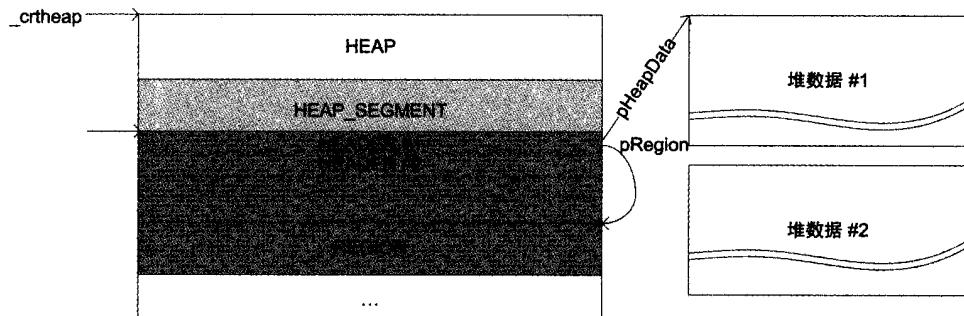


图 23-8 SBH 模式的 CRT 堆

考虑篇幅限制，我们不再深入讨论 SBH 的更多细节，CRT 源代码目录中的 sbheap.c 包含了 SBH 管理函数的具体实现，winheap.h 中包含了结构声明和常量定义，感兴趣的读者可以自己去阅读这些源文件，本章的示例程序 SBHeap (code\chap23\sbheap) 演示了如何启用 SBH 模式和从中分配堆块。

23.11.3 创建和选择模式

在第 21 章介绍 CRT 初始化时，我们提到了其中一个重要的步骤就是调用 `_heap_init()` 函数创建和初始化 CRT 堆。CRT 的源程序文件 heapinit.c 包含了 `_heap_init()` 函数的完整代码。概括来说，`_heap_init()` 主要是执行以下三个动作。

第一，调用 `HeapCreate` 创建一个普通的 Win32 堆，将其句柄保存在全局变量 `_crtheap` 中。

```
_crtheap = HeapCreate( mtflag ? 0 : HEAP_NO_SERIALIZE,
                      BYTES_PER_PAGE, 0 )
```

第二，调用 `_heap_select` 函数选择使用三种模式中的一种，并将选择结果记录在全局变量 `_active_heap` 中。

```
_active_heap = _heap_select();
```

第三，如果选择的是 SBH 或旧 SBH 模式（`__active_heap` 等于 2 或 3），便调用 `__sbh_heap_init` 或 `__old_sbh_new_region` 来初始化 SBH。

那么 `__heap_select` 函数是如何选择工作模式的呢？简单来说，它会做以下三个判断。

第一，读取操作系统的版本号，如果是 Windows 2000 或更高，那么便返回 `__SYSTEM_HEAP(1)`，即选择 `SYSTEM_HEAP` 模式。

```
if ( (osplatform == VER_PLATFORM_WIN32_NT) && (winmajor >= 5) )
    return __SYSTEM_HEAP;
```

第二，尝试读取 "`__MSVCRT_HEAP_SELECT`" 和 "`__GLOBAL_HEAP_SELECTED`" 这两个环境变量，如果读取到了有效设置，那么便返回读到的设置。

第三，调用 `_GetLinkerVersion` 函数取链接器的版本号，如果取得的主版本号大于等于 6 (VC6)，那么便返回 `V6_HEAP`，否则返回 `V5_HEAP`。

VC6 版本的 `__heap_select` 函数和 VC2005 的实现略有差异，以上是 VC6 的做法。对于 VC2005，第二步被放在检查 `CRTDLL` 标志的条件编译块中，而且被移到最前面。这意味着，CRT 运行库 DLL 中的 `__heap_select` 函数会先检查环境变量，然后再检查系统版本。因此，对于 VC2005 编译的程序，如果使用的是动态链接运行库，那么设置环境变量可以配置它的 CRT 堆工作模式。

因为大多数系统都没有设置上面所说的环境变量，而且系统的版本大多都是 Windows 2000 以上，所以，大多数情况下，CRT 堆的工作模式都是系统模式 (`SYSTEM_HEAP`)。调用 `_set_sbh_threshold` 函数可以强制启用 SBH 模式，本章的 `SBHeap` 例子使用的就是这种方法。观察全局变量 `__active_heap` 可以知道 CRT 堆的当前工作模式。如果是静态链接 CRT 库，那么这个变量定义在应用程序的模块中，如果是动态链接 CRT 库，那么它定义在 VC 的运行库 DLL 中。以 `SBHeap` 程序为例，它是使用 VC2005 编译的，而且使用的是动态链接 CRT (默认)，因此可以使用如下命令：

```
0:000> dd MSVCR80D!__active_heap 11
10313980 00000003 // 3 代表 SBH 模式
```

值得注意的是，一个进程中可能有多个运行库的实例，它们可以使用不同的工作模式。例如 `SBHeap` 程序中也加载了 `msvcrt.dll` 模块，它的 CRT 堆使用的就是系统模式：

```
0:000> dd msvcrt!__active_heap 11
77c6241c 00000001 // 1 代表系统模式
```

23.11.4 CRT 堆的终止

与进程的其他堆一样，销毁 CRT 堆是作为系统销毁进程的最后动作之一进行的。换句话说，尽管 `heapinit.c` 中存在一个名为 `__heap_term` 的函数，该函数中有销毁 CRT 堆的调用：

```
HeapDestroy(_crtheap);
_crtheap=NULL;
```

但是正常情况下，该方法是不会被执行的。事实上，当进程退出，WinDBG 得到最后的控制机会时，进程的主线程已经退出，因此不能再执行任何恢复执行命令，但是我们还可以使用!heap 命令观察进程的各个堆（包括 CRT 堆）。这时观察任务管理器中的进程列表，仍然可以看到这个进程，这是因为程序的初始线程还在等待调试器的回复，而且调试器还在引用进程句柄，所以进程还没有完全退出。正如第 23.2 节所介绍的，系统在清理进程的地址空间时会自动销毁进程的所有堆，因此__heap_term 不被调用也不会有资源泄漏问题。

23.12 CRT 堆的调试堆块

为了支持调试内存分配有关的问题，CRT 特别设计了供调试使用的堆块结构和函数，分别简称为 CRT 堆的调试堆块和调试函数。与普通堆块相比，调试堆块中增加了很多用于调试的信息，因此堆块所占用的空间更大了。

当编译调试版本的 C/C++ 应用程序时，编译器会默认使用 CRT 堆的调试函数和调试堆块结构。CRT 的源代码文件 dbgheap.c 包含了 CRT 堆调试函数的实现细节，头文件 crtdebug.h (INCLUDE 目录) 中定义了 CRT 调试函数所公开的常量、数据结构和函数原型，头文件 dbgint.h (CRT\SRC 目录) 中定义了 CRT 调试函数内部使用 (internal) 的数据结构和函数。本节将着重介绍 CRT 堆的调试堆块结构，下一节介绍 CRT 堆的调试函数和应用。

23.12.1 _CrtMemBlockHeader 结构

可以把 CRT 堆的调试堆块分为四个部分，最前面是管理信息区，即一个固定长度的_CrtMemBlockHeader 结构；中间是用户数据区，其长度是根据应用程序请求分配的长度而定的；数据区后是用来检测堆溢出的栅栏字节，又被称为“不可着陆区”(No Mans Land)，长度为 4 个字节；最后一个部分是用于满足分配粒度要求的填充字节，如果前三个部分的长度恰好已经满足分配粒度要求，那么就不存在第四个部分。因为_CrtMemBlockHeader 结构的最后一个字段也是 4 字节的不可着陆区，所以实际上用户数据区的前后都有四个字节的“栅栏”保护。

头文件 dbgint.h 中定义了_CrtMemBlockHeader 结构：

```
typedef struct _CrtMemBlockHeader
{
    struct _CrtMemBlockHeader * pBlockHeaderNext;      // 指向下一个块
    struct _CrtMemBlockHeader * pBlockHeaderPrev;      // 指向上一个块
    char *                  szFileName;                // 源文件名
    int                     nLine;                    // 源代码行号
    size_t                  nDataSize;                // 用户区的字节数
```

```

int          nBlockUse;           //块的类型
long         lRequest;           //块序号
unsigned char gap[nNoMansLandSize]; //不可着陆区
} _CrtMemBlockHeader;

```

其中 `pBlockHeaderNext` 和 `pBlockHeaderPrev` 分别指向下一个块和前一个块的 `_CrtMemBlockHeader` 结构。`szFileName` 用来指向使用该块的源程序文件文件名，`nLine` 用来记录源代码的行号，这两个字段的值不总是有效的，我们将在下一节介绍启用它们的方法。`nDataSize` 是用户数据区的字节数，`nBlockUse` 代表该块的类型，稍后介绍。`nNoMansLandSize` 被定义为 4，即 `gap` 字段的长度是 4 个字节，通常被填充为 `0xFD`。`lRequest` 是堆块分配的一个流水号。

因为每个堆块的 `_CrtMemBlockHeader` 结构和用户数据是紧邻的，所以可以通过如下两个宏来从用户指针得到 `_CrtMemBlockHeader` 结构指针，或者相反：

```
#define pbData(pblock) ((unsigned char *)((_CrtMemBlockHeader *)pblock + 1))
#define pHdr(pbData) (((_CrtMemBlockHeader *)pbData)-1)
```

可见，两个宏都是先将源指针强制转换为指向 `_CrtMemBlockHeader` 结构的指针，然后向前或向后偏移一个结构便是目标指针。

23.12.2 块类型

CRT 堆的调试堆块可以为如下五种类型之一（定义在 `crtdbg.h` 中）：

```

#define _FREE_BLOCK      0 // 空闲块
#define _NORMAL_BLOCK    1 // 常规块
#define _CRT_BLOCK        2 // CRT 内部使用的块
#define _IGNORE_BLOCK     3 // 堆检查时应该忽略的块
#define _CLIENT_BLOCK     4 // 客户块，高 16 位可以进一步定义子类型

```

其中，`_IGNORE_BLOCK` 用于屏蔽 CRT 的检查功能，比如在做堆块转储（dump）时，这样的块会被跳过。常规块是应用程序调用 CRT 分配函数时所默认使用的块。对于较大型的程序，或者为了追踪不同模块的内存分配情况，可以使用 `_CLIENT_BLOCK` 类型，并使用高 16 位来记录模块 ID 或其他标志信息，因此高 16 位又被称为子类型。块类型信息被记录在堆块的 `_CrtMemBlockHeader` 结构的 `nBlockUse` 字段中。`nBlockUse` 字段为 32 位的整数，低 16 位为块的基本类型，即以上 5 个常量之一，高 16 位用于存储子类型。

使用 `_CrtReportBlockType` 函数可以取得一个用户地址（`pvData`）所在堆块的堆块类型，也就是返回该堆块的 `_CrtMemBlockHeader` 结构的 `nBlockUse` 字段。如果要进一步判断块的基本类型和子类型，那么可以使用如下两个宏：

```
#define _BLOCK_TYPE(block)          (block & 0xFFFF)
#define _BLOCK_SUBTYPE(block)        (block >> 16 & 0xFFFF)
```

下面以 MFC 类库为例介绍如何使用子类型来追踪堆块的用途。MFC 的根类 `CObject` 将 `new` 操作符作了如下重载：

```
void* PASCAL CObject::operator new(size_t nSize) //位于 afxmem.cpp 中
```

```

{
#ifndef _AFX_NO_DEBUG_CRT           //如果不使用 CRT 的调试支持
    return ::operator new(nSize);   //使用普通的 new 运算符
#else                            //不然则使用专用的
    return ::operator new(nSize, _AFX_CLIENT_BLOCK, NULL, 0);
#endif                           // _AFX_NO_DEBUG_CRT
}

```

`_AFX_NO_DEBUG_CRT` 标志的含义是不使用 CRT 的调试功能，其作用是为了满足某些特别情况下在调试版本中禁用 CRT 的调试支持。对于大多数情况，不会定义该标志，所以，`CObject` 的 `new` 操作符是使用全局的 `new` 操作符从 CRT 堆上分配一个 `_AFX_CLIENT_BLOCK` 类型的块。

`_AFX_CLIENT_BLOCK` 的定义如下：

```
#define _AFX_CLIENT_BLOCK (_CLIENT_BLOCK|(0xc0<<16))
```

即低 16 字节为 `_CLIENT_BLOCK`，高 16 字节为 `0xC0`。因为 `CObject` 是 MFC 类库中几乎所有其他类的基类，MFC 程序中的类也大多间接或直接从该类派生，所以这些派生类都继承了上面的 `new` 操作符。这样，当使用 `new` 操作符动态创建这些类对象时，这些类对象在堆上的堆块便都具有 `_AFX_CLIENT_BLOCK` 类型。换句话说，当遍历堆时，可以很容易地辨别出该堆块是否是 `CObject` 类或其派生类的实例所使用的。事实上，用于转储 (Dump) MFC 对象的 `_AfxCrtDumpClient` 函数，就是通过检查堆块类型是否等于 `_AFX_CLIENT_BLOCK` 来判断一个指针是否是指向 `CObject` 对象。

```
void __cdecl _AfxCrtDumpClient(void * pvData, size_t nBytes) //位于
dumpinit.cpp 中
{
    if(_CrtReportBlockType(pvData) != _AFX_CLIENT_BLOCK)           //判断块类型
        return;                                                     //不是 CObject 对象
    CObject* pObject = (CObject*)pvData;                          //类型强制转化
    ...
}
```

23.12.3 分配堆块

下面我们来看 CRT 调试堆块的分配过程并观察一个实际的堆块。调试版本的 CRT 分配函数大多实现在 `dbgheap.c` 文件中，阅读该文件，可以看到各个函数间的调用关系如图 23-9 所示。

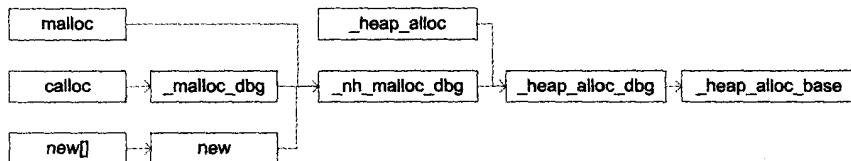


图 23-9 CRT 分配函数（调试版本）的调用关系图

其中，`new[]` 代表数组形式的 `new` 运算符，它会调用普通的 `new` 操作符，然后调用 `_nh_malloc_dbg` 函数。以上函数名中的“`_nh`”是 `new handler`（分配处理器）的缩写，

意味着这些函数分配内存失败时，它们会调用已经注册的分配处理器函数，通过 `_set_new_handler` 函数可以设置分配处理器。

通过以上调用关系可以看到这些分配函数最终都调用 `_heap_alloc_base` 函数。在 VC6 的 CRT 源文件 `malloc.c` 中可以直接看到这个函数的实现，在 VC8 中，`_heap_alloc_base` 被定义为 `_heap_alloc` (`malloc.c` 中) 函数的别名，也就是说，`malloc.c` 中的 `_heap_alloc` 函数在调试版本中会被编译为 `_heap_alloc_base`。`_heap_alloc_base` 所作的事情也并不复杂，它只是根据 CRT 堆的当前工作模式（记录在 `__active_heap` 中），调用各个模式自己的分配函数。例如，如果分配模式是 `__SYSTEM_HEAP`，就调用 `HeapAlloc API`。

事实上，`_heap_alloc_base` 函数也是调试版本的分配函数和非调试版本函数的汇合处。到这里，调试版本和发布版本的工作方式就一样了。调试版本的主要差异是会调用调试版本的中间函数，比如 `_nh_malloc_dbg`、`_heap_alloc_dbg` 等，而 `_heap_alloc_dbg` 函数是分配调试堆块的中心，它的原型为：

```
extern "C" void * __cdecl _heap_alloc_dbg( size_t nSize, int nBlockUse,
    const char * szFileName, int nLine )
```

其中，`nSize` 是要分配的字节数（应用程序请求的字节数）；`nBlockUse` 为要分配的堆块类型；`szFileName` 和 `nLine` 用来指定发起调用的源文件名称和行号。

以 VC8 为例，`_heap_alloc_dbg` 函数的执行过程如下。

第一，调用 `_mlock(_HEAP_LOCK)` 锁定堆。

第二，进入一个 `__try` 块，对应的 `__finally` 块中有 `_munlock(_HEAP_LOCK)`，保证退出此函数前会释放堆的锁信号。

第三，如果当前请求计数 `_lRequestCurr` 等于 `_crtBreakAlloc`，而且 `_crtBreakAlloc!=-1`，那么调用 `_CrtDbgBreak()` 中断到调试器。`_lRequestCurr` 和 `_crtBreakAlloc` 是 CRT 调试版本定义的两个全局变量。`_lRequestCurr` 用来记录从调试堆分配内存的流水号，`_crtBreakAlloc` 用来设置所谓的内存分配断点（详见 23.13.1）。

第四，如果用来记录内存分配挂钩函数的全局函数指针 `_pfnAllocHook` 不为空，那么便调用该指针所指向的函数。如果该函数返回 0 (FALSE)，便调用 RPT0 或 RPT2 报告错误，如果 `szFileName` 不为空，报告的内容为：Client hook allocation failure at file %hs line %d，如果 `szFileName` 为空，则报告的内容为：Client hook allocation failure。报告返回后，`_heap_alloc_dbg` 函数也随即返回了。

第五，检查 `nBlockUse` 参数，如果指定的不是合法的堆块类型，那么调用 `_RPT0` 报告错误。如果用户选择忽略，那么该函数会继续执行，内存分配也会成功。但是使用 CRT 的内存检查函数或作堆块转储时，CRT 会发现错误类型的堆块，给出错误信息：

Bad memory block found at XXX。

第六，计算堆块的大小，其算法是在请求大小的基础上先加上_CrtMemBlockHeader 结构所占的空间和隔离区的大小：

```
blockSize = sizeof(_CrtMemBlockHeader) + nSize + nNoMansLandSize;
```

然后，如果定义了 WINHEAP 标志，那么再取整为最接近的粒度（x86 平台为 8）倍数：

```
#ifndef WINHEAP
    blockSize = _ROUND2(blockSize, _GRANULARITY);
#endif /* WINHEAP */
```

第七，调用_heap_alloc_base 函数分配 blockSize 字节的内存，将返回值赋给局部变量 pHead，如果 pHead 为空，那么将错误号码设置为 errno = ENOMEM，而后函数返回。

第八，递增记录在全局变量_lRequestCurr 中的堆块分配次数计数。

第九，设置_CrtMemBlockHeader 结构的各个字段。如果该块需要忽略（避开 CRT 的检查功能，比如 CRT 自己分配的块），那么便将 pBlockHeaderNext 和 pBlockHeaderPrev 设置为空，nBlockUse 设置为_IGNORE_BLOCK，lRequest 设置为 IGNORE_REQ。否则便将该块插入到由全局变量_pFirstBlock 和 _pLastBlock 所记录的链表的表头，并更新用于统计内存分配长度的全局变量_lTotalAlloc（曾经分配的所有用户数据的总长度，包括已经释放的）、_lCurAlloc（已经分配的用户数据的长度净值，扣除了已经释放的）和 _lMaxAlloc（迄今为止的最大用户数据长度）。

第十，使用静态变量_bNoMansLandFill 的值（0xFD）填充用户数据区前后的隔离区。

第十一，使用静态变量_bCleanLandFill 的值（0xCD）填充整个用户数据区。这可以解释为什么 CRT 堆转储出的用户数据中经常包含很多个 CD，下一节将介绍转储 CRT 堆块的方法。

第十二，使用 pbData(pHead) 宏，返回指向用户数据区起始字节的指针。

从上面的过程我们知道，当分配 CRT 调试堆块时，CRT 总会在用户数据前增加一个_CrtMemBlockHeader 结构，默认增加 4 字节的保护字段。因为_CrtMemBlockHeader 结构的长度是 32 字节，这意味着，每次都至少要多分配 36 字节，这是使用 CRT 调试堆块的固定额外开销。

在上面的第六步中，_heap_alloc_base 函数会调用底层的堆块分配函数来实际分配堆块，根据 CRT 堆的工作模式，可能调用 Win32 堆的分配函数，也可能调用 SBH 的分配函数，但是无论使用哪一种，这些底层的分配函数都需要为新的堆块再做一次“包装”，在原来的基础上增加新的管理结构。这就好像是网络通信中发送一个数据包时每个通信层都会附加一个报头结构。以系统模式为例，每个 Win32 堆块前都需要一

个 HEAP_ENTRY 结构, 因此一个 CRT 调试堆块在内存中的布局便是 Win32 的管理结构(HEAP_ENTRY)+CRT 调试堆块的管理结构(_CrtMemBlockHeade)+用户数据+CRT 调试堆块的栅栏字节+ [CRT 调试堆块的补位字节] + [Win32 堆块的栅栏字节]+[Win32 堆块的额外信息]。

下面观察一个真实的堆块, 该堆块是执行如下语句得到的:

```
void * p=malloc(5);
```

显示 p 的值, 得到 003707b8, 向前偏移 40 个字节 (32+8) 得到该堆块的起始位置, 然后显示从这个位置开始的内存区, 其内容为:

```
0:000> dd 003707b8-20-8
00370790 00290009 0018070f 00372fb8 00000000
003707a0 00000000 00000000 00000005 00000001
003707b0 00000039 fdfdfdfda cddcdcd fdfdfdcdb
003707c0 baadf0fd baadf00d abababab abababab
003707d0 00000014 00000000 00090114 00ee04ee
003707e0 00374000 00370178 feffffee feffffee
```

第一行的前 8 个字节是 HEAP_ENTRY 结构, 可以使用 dt 命令显示其各个字段的值:

```
0:000> dt _HEAP_ENTRY 003707b8-20-8
+0x000 Size : 9 //以粒度为单位的堆块大小
+0x002 PreviousSize : 0x29 //前一个堆块的大小 (粒度为单位)
+0x000 SubSegmentCode : 0x00290009 //子段代码
+0x004 SmallTagIndex : 0xf '' //堆块标记
+0x005 Flags : 0x7 '' //堆块标志
+0x006 UnusedBytes : 0x18 '' //未使用字节
+0x007 SegmentIndex : 0 '' //所属段
```

注意这里的 Size (9) 是以粒度为单位的, 也就是等于 72 字节, 这是整个堆块的大小, 这意味着我们请求的 5 个字节所占用的堆块大小为 72 字节, 即上面 dd 结果中的前 4 行加上第 5 行的 8 个字节。

HEAP_ENTRY 结构后面是 32 字节的_CrtMemBlockHeade 结构, 我们可以看到 pBlockHeaderNext 字段的值为 00372fb8, 即下一个块的_CrtMemBlockHeade 结构地址。pBlockHeaderPrev 字段的值为空, 即当前块是第一个块, 也就是新创建的块总是被放在链表的表头 (参见前面第八步)。

第 3 行的 8 到 12 字节是用户数据区, 即 5 个字节的 0xCD, 其前面是 4 个字节的隔离区, 即_CrtMemBlock- Heade 结构的 gap 字段, 其后面也是 4 个字节的隔离区 (从 00372fb0 到 003707c0)。隔离区后是 7 个字节长的补齐字节。这些字节仍保留着 Win32 堆填充的 baadf00d 模式。而后的 8 个字节 0xab 是 Win32 堆填充的栅栏字节 (CheckHeapFillPattern)。第 5 行的开始 8 个字节是 Win32 堆块的附加信息, 即 HEAP_ENTRY_EXTRA 结构, 它后面便是下一个堆块的内容了。

追踪_heap_alloc_dbg 函数的执行过程, 可以看到, _heap_alloc_dbg 函数计

算出的 blockSize 为 41，也就是说，CRT 向 Win32 堆请求的长度是 48 字节，即从第 1 行的后两个 DWORD 到第 4 行前两个 DWORD。相对于 Win32 堆，这 48 个字节都是数据区，会使用 baadf00d 来填充，但是返回后，CRT 又对这个区域进行了填充，因此只有最后 7 个字节保留着原来的填充内容。对于 CRT 堆和应用程序来说，从 003707b8 开始的 5 个字节是真正的数据区，因此被填充为 CD。从这中数据布局我们可以清楚地看到 CRT 的调试堆块被包裹在 Win32 的堆块之中，这与 OSI 模型中网络包结构很类似，下面的层总是会对上层的数据再包装，使用户数据被层层包裹起来。

23.13 CRT 堆的调试功能

上一节介绍了 CRT 堆的调试堆块格式和堆块分配过程，本节和下一节将介绍 CRT 堆所支持的各种调试功能。

23.13.1 内存分配序号断点

根据上一节的介绍，CRT 堆会维护一个名为 _lRequestCurr 的全局变量，用来记录每次分配堆块的序号。为了支持调试，CRT 还定义了另一个全局变量，名为 _crtBreakAlloc。当每次从 CRT 调试堆分配内存时，CRT 的内存分配函数 (_heap_alloc_dbg) 会检查 _lRequestCurr 变量是否等于 _crtBreakAlloc，如果相等，便调用 _CrtDbgBreak() 触发断点事件，试图中断到调试器。这种针对内存分配序号所设置的断点被称为内存分配序号断点（Breakpoint on an Allocation Number）。

每个 CRT 调试堆块的 _CrtMemBlockHeader 结构中记录了这个堆块的分配序号。因此如果希望下次分配这个堆块时中断到调试器中，那么可以将这个序号设置到 _crtBreakAlloc 变量中，并按照上次的路线重新执行程序。可以通过以下三种方法之一来设置 _crtBreakAlloc 变量。

第一，如果使用的是 Visual Studio 集成环境，那么可以在变量观察窗口输入 _crtBreakAlloc（静态连接）或 {,,msvcrXXd.dll} _crtBreakAlloc（动态链接），然后输入要中断的序号。msvcrXXd 中的 XX 是 C 运行库的版本号。可以通过观察当前进程中的模块列表来了解当前使用的 CRT 版本。

第二，如果使用的是 WinDBG 调试器，那么可以使用 ed 命令编辑 _crtBreakAlloc 变量的值。

第三，如果想在程序中通过代码设置内存分配序号断点，那么可以直接向 _crtBreakAlloc 变量赋值，或者调用 _CrtSetBreakAlloc 函数。

_crtBreakAlloc 变量的默认值是 -1，即禁用内存分配序号断点。

23.13.2 分配挂钩

如果希望对内存操作做更多的跟踪和记录，那么可以定义一个内存分配挂钩函数（allocation hook function）。然后调用 CRT 的 _CrtSetAllocHook 函数将其保存到全局变量 _pfnAllocHook 中。分配挂钩函数应该具有如下原型：

```
int YourAllocHook( int allocType, void *userData, size_t size, int
    blockType, long requestNumber, const unsigned char *filename, int lineNumber);
```

其中 allocType 用来指定内存操作的类型，可以为 _HOOK_ALLOC(1)、_HOOK_REALLOC(2) 和 _HOOK_FREE(3) 三个常量之一。当 CRT 即将分配一个堆块时，会调用已经注册的分配挂钩函数，并将 allocType 参数设置为 _HOOK_ALLOC，userData 参数设为空（因为内存尚未分配）。类似的，当 CRT 即将重新分配一个堆块时，会使用 _HOOK_REALLOC 参数调用分配挂钩函数，此时 userData 指向前一次分配的用户数据。当 CRT 即将释放一个堆块时会使用 _HOOK_FREE 参数调用分配挂钩函数，userData 参数指向要释放堆块的用户数据。

对于任一种情况，如果分配挂钩函数返回 0 (FALSE)，那么 CRT 会报告错误并终止操作。因此可以利用分配挂钩函数强制内存分配申请失败（Forced Failure）。

23.13.3 自动和手动检查

调试版本的 CRT 设计了一个名为 _CrtCheckMemory 的函数用于检查 CRT 堆的完好性。该函数的源代码也位于 dbgheap.c 文件中，它执行的主要步骤如下。

1. 检查全局变量 _crtDbgFlag (CRT 调试标志) 中是否包含 _CRTDBG_ALLOC_MEM_DF (1) 标志，如果不包含，那么返回。_crtDbgFlag 变量的默认值中包含了 _CRTDBG_ALLOC_MEM_DF 标志。
2. 调用 _mlock (_HEAP_LOCK) 锁定 CRT 堆，然后进入 __try 块，对应的 __finally 块包含了解锁操作。
3. 调用 _heapchk() 函数，该函数的源代码位于 heapchk.c 中，其作用是根据 CRT 堆的工作模式调用堆本身的检查函数，例如，如果使用的是系统模式 (__SYSTEM_HEAP)，便调用 HeapValidate API 进行检查。如果 _heapchk() 返回的结果有问题，便报告错误，然后返回 FALSE。
4. 根据 _pFirstBlock 指针遍历堆中的所有堆块，逐一进行检查。对于每个堆块，首先检查块头中的 nBlockUse 字段是否是有效的堆块类型。然后检查用户数据区前后的隔离区是否完好。另外，对于空闲堆块会检查自动填充的 0xDD (变量 _bDeadLandFill 的值) 是否被破坏。如果被破坏，那么就说明释放的堆块又被访问过，会通过 RPT 宏报告 (_CRT_WARN)。如果检查到任何问题，_CrtCheckMemory 便会返回 FALSE，否则返回 TRUE。

可以在程序中插入代码调用 `_CrtCheckMemory` 函数，触发内存检查操作。事实上，CRT 堆的工作函数在被调用时也会自动调用 `_CrtCheckMemory` 函数，其调用频率可以通过调用 `_CrtSetDbgFlag()` 函数来设置。对于 VC6，可以在参数中指定 `_CRTDBG_CHECK_ALWAYS_DF (4)`，告诉 CRT 堆每次执行分配、重新分配、计算大小 (`_msize_dbg`) 或释放操作时都进行内存检查。在 VC8 中，可以通过这个函数设置全局变量 `check_frequency` 的值，指定每多少次堆操作做一次检查，比如 `_CRTDBG_CHECK_EVERY_16_DF (0x00100000)` 代表每 16 次内存操作会执行一次检查，`_CRTDBG_CHECK_EVERY_128_DF (0x00800000)` 代表每 128 次执行一次检查，`_CRTDBG_CHECK_EVERY_1024_DF (0x04000000)` 代表每 1024 次执行一次检查，默认值是 1024。

当释放内存时，除了可能调用 `_CrtCheckMemory` 进行上述检查，CRT 还会进行一项很有意义的检查，那就是调用 `_CrtIsValidHeapPointer(pUserData)` 检查要释放的堆块是否属于当前堆。如果 CRT 堆的工作模式是 `_V6_HEAP` 或 `_V5_HEAP`，那么 `_CrtIsValidHeapPointer` 函数会在当前堆中寻找是否可以找到与用户指针相对应的堆块（块头地址等于 `pHdr(pUserData)`），如果找不到，那么这个用户指针就是非法的。如果 CRT 堆的工作模式是 `_SYSTEM_HEAP`，那么 `_CrtIsValidHeapPointer` 函数会调用 `HeapValidate API` 来检查 `pHdr(pUserData)` 是否属于有效的 Win32 堆块。我们知道一个 CRT 堆块是由块头和数据区两个部分组成的，返回给应用程序的是指向用户区的指针。在 CRT 内部，很多时候是使用 `pHdr` 宏直接根据用户指针推算出块头指针，即：`pHead = pHdr(pUserData);`；如果用户指针因为意外操作改变了，也就是 `pUserData` 偏离了分配函数本来返回的值，那么便会导致推算出的块头指针也是无效的，对这样的指针继续执行任何操作都可能导致非法访问错误。

23.14 堆块转储

所谓堆块转储，就是将堆中的所有或一部分堆块以某种方式输出来供分析和审查。CRT 提供了一系列函数来实现不同功能堆块转储，包括转储所有堆块、转储从某一时刻开始的堆块等。本节将介绍这些功能的工作原理和使用方法。

23.14.1 内存状态和检查点

CRT 使用如下结构来描述某一时刻 CRT 堆的状态：

```
typedef struct _CrtMemState
{
    struct _CrtMemBlockHeader * pBlockHeader;      //堆中第一个堆块的地址
    unsigned long lCounts[_MAX_BLOCKS];            //每种类型堆块的总数
    unsigned long lSizes[_MAX_BLOCKS];              //每种类型堆块的总大小
    unsigned long lHighWaterCount;                  //分配的最大堆块大小
```

```

        unsigned long lTotalCount; //分配的所有堆块大小之和
} _CrtMemState;

```

其中, `pBlockHeader` 指向堆中最近分配的一个堆块 (忽略块除外), 即采样时全局变量 `_pFirstBlock` 的值。`lCounts` 和 `lSizes` 数组的一个元素对应于一种类型的堆块。`lCounts` 数组用来记录每种类型的堆块的总块数, `lSizes` 数组用来记录每种类型的所有堆块的用户数据 (`nDataSize`) 的总和。`lHighWaterCount` 是采样时 `_lMaxAlloc` 的值, 即曾经分配的最大堆块的大小。`lTotalCount` 是采样时 `_lTotalAlloc` 变量的值, 即曾经分配的总内存数。`lTotalCount` 和 `_lTotalAlloc` 的值都不包括已经分配的忽略块。

只要调用 `_CrtMemCheckpoint` 函数便可以将当时的 CRT 堆的情况统计到 `_CrtMemState` 结构中。在程序运行的不同时刻调用该函数便可以得到不同时刻的堆状态, 比较这些状态可以帮助我们了解堆的变化信息, 为寻找内存泄漏和发现其他问题提供参考。为了方便比较多个内存状态, CRT 设计了函数 `_CrtMemDifference`, 将两个状态的比较结果放入到参数 `stateDiff` 指向的第三个 `_CrtMemState` 结构中。

```
int _CrtMemDifference( _CrtMemState *stateDiff, const _CrtMemState *oldState,
    const _CrtMemState *newState );
```

`_CrtMemDumpStatistics` 函数可以把 `_CrtMemState` 结构中的信息以文字的形式通过 `RPT` 宏输出到 `CRT_WARN` 信息所对应的目的地 (调试器窗口或文件, 参见第 21.5.2 节)。

23.14.2 `_CrtMemDumpAllObjectsSince`

可以使用 `_CrtMemDumpAllObjectsSince` 函数将从某一检查点以来的所有堆块转储到 `CRT_WARN` 信息所对应的目标输出中, 它的函数原型为:

```
void _CrtMemDumpAllObjectsSince( const _CrtMemState *state );
```

如果 `state` 参数等于 `NULL`, 那么便转储所有的堆块, 该函数的工作过程如下。

第一, 调用 `_mlock(_HEAP_LOCK)` 锁定堆, 然后进入 `__try` 块, 对应的 `__finally` 块中有 `_munlock(_HEAP_LOCK)`, 保证退出此函数前会释放堆的锁信号。

第二, 打印提示信息: “`Dumping objects ->`”。

第三, 如果 `state` 不为空, 则将其中包含的头指针 (`pBlockHeader`) 赋给局部变量 `pStopBlock`。`pStopBlock` 用来控制转储循环的结束点, 默认为 `NULL`, 即一直循环到链表的最后一个节点。

第四, 开始一个 `for` 循环, 从当前堆列表的头开始, 结束点为 `pStopBlock`。

```
for (pHead = _pFirstBlock; pHead != NULL && pHead != pStopBlock;
    pHead = pHead->pBlockHeaderNext)
```

以上指针都是 `_CrtMemBlockHeader` 类型, 即 CRT 调试堆块的头结构。

第五, 在 `for` 循环中, 对于每个节点 (堆块), 执行如下操作。首先检查是否是需

要忽略的块，如果是，则开始下一轮循环。满足如下三个条件之一的块都会被跳过：块类型为`_IGNORE_BLOCK`；块类型为`_FREE_BLOCK`；块类型为`_CRT_BLOCK`，而且全局变量`_crtDbgFlag`中不包含`_CRTDBG_CHECK_CRT_DF`标志。

对于须要转储的块，首先判断块头的`szFileName`字段是否为空，如果不为空，先调用`_CrtIsValidPointer`检查是否是有效的指针。如果不是，则显示：`#File Error#(%d)`，其中`%d`会被替换为块头的`nLine`字段的值。如果`szFileName`有效，则显示其内容和`nLine`字段的值，即执行`_RPT2(_CRT_WARN, "%hs(%d) : ", pHead->szFileName, pHead->nLine);`；然后打印堆块的分配序号，即`lRequest`字段的值。

```
_RPT1(_CRT_WARN, "{%ld} ", pHead->lRequest);
```

接下来根据堆块类型，分别报告堆块的详细信息。如果堆块类型为`_CLIENT_BLOCK`，则显示“client block at 0x%p, subtype %x, %Iu bytes long.\n”。如果堆块类型为`_NORMAL_BLOCK`，则显示“normal block at 0x%p, %Iu bytes long.\n”。如果堆块类型为`_CRT_BLOCK`，则显示“crt block at 0x%p, subtype %x, %Iu bytes long.\n”。

最后，转储堆块中的用户数据。如果块类型为`_NORMAL_BLOCK`和`_CRT_BLOCK`，则调用`_printMemBlockData`函数打印出前 16 个字节，例如：

```
Data: <This is the p2 s> 54 68 69 73 20 69 73 20 74 68 65 20 70 32 20 73
```

尖括号中是字符格式，后面是原始的十六进制值。

如果块类型为`_CLIENT_BLOCK`，则先判断用来记录转储挂钩函数的全局指针`_pfnDumpClient`是否为空，如果不为空，则调用该函数。如果为空，则调用`_printMemBlockData`函数打印出前 16 个字节。

第六，循环结束后，打印提示信息：“Object dump complete.\n”，随即函数返回。

清单 23-30 显示了`_CrtMemDumpAllObjectsSince`函数转储出的两个堆块信息，上面两行报告的是 59 号块，类型为`_CLIENT_BLOCK`，在块号前是分配该块的源文件名（完整路径，做了省略）和行号（第 127 行）。后面两行报告的是 55 号块，类型为`_NORMAL_BLOCK`。

清单 23-30 CRT 转储出的堆块信息

```
c:\...\memchk.cpp(127) : {59} client block at 0x00372430, subtype 0, 40 bytes long.
Data: <p2 points to a C> 70 32 20 70 6F 69 6E 74 73 20 74 6F 20 61 20 43
{55} normal block at 0x00371038, 34 bytes long.
Data: <This is the p2 s> 54 68 69 73 20 69 73 20 74 68 65 20 70 32 20 73
```

我们将在下一节介绍如何让转储信息中包含确切的源文件名和行号。

23.14.3 转储挂钩

上面介绍的堆块转储信息中，只包含用户数据前 16 字节的十六进制值和对应的 ASCII 码。那么可不可以将用户数据以结构化的方式显示出来呢？也就是按照用户数

据的本来结构将其显示出来。答案是可以的，其做法如下。

第一，为对象结构设计一个子类型，以便可以通过子类型号确认一个堆块中存储的是这个数据类型。

第二，在类中重载 new 操作符，使得创建该类的对象时会分配指定子类型的用户块 (_CLIENT_BLOCK)，参见前面对 CObject 类的 new 运算符的介绍。

第三，定义并实现一个转储挂钩函数，用于将用户数据以结构化的形式显示出来。该函数应该具有如下原型和类似实现：

```
void DumpClientFunction( void *userPortion, size_t blockSize )
{
    CxxxClass * pObject;                                //判断块类型
    if(_CrtReportBlockType(pvData) != _MY_CLIENT_BLOCK) //如果不是则返回
        return;                                         //强制类型转换
    pObject = (CxxxClass *) pUserData;                 //显示 pObject 对象
    ...
}
```

可以参考 MFC 类库源程序 dumpinit.cpp 中的 _AfxCrtDumpClient 函数。

第四，调用 _CrtSetDumpClient 函数，注册上一步所定义的转储挂钩函数。

_CrtSetDumpClient 函数会将参数中指定的转储挂钩函数记录在全局变量 _pfnDumpClient 中。CRT 的堆块转储函数 (_CrtMemDumpAllObjectsSince) 在转储_CLIENT_BLOCK 类型的堆块时，如果检测到 _pfnDumpClient 不为空，那么便会调用这个函数。于是便可以实现结构化的显示用户数据了。MFC 程序便是通过以上方法来转储 MFC 对象的。具体来说，在 dumpinit.cpp 中定义了一个全局变量 afxDebugState：

```
PROCESS_LOCAL(_AFX_DEBUG_STATE, afxDebugState)
```

在该对象实例的构造函数中，会将 _AfxCrtDumpClient 函数设置为转储挂钩函数。在 afxDebugState 变量被析构时，_AFX_DEBUG_STATE 的析构函数会调用 _CrtDumpMemoryLeaks() 函数，后者在转储用户类型的堆块时会调用 _AfxCrtDumpClient 函数。例如以下是一个 CDlg 对象的转储信息：

```
Dumping objects ->
C:\...\HeapMfc\HeapMfc.cpp(70) : {97} client block at 0x00374F80, subtype 0, 100
bytes long.
a CDlg object at $00374F80, 100 bytes long
Object dump complete.
```

清单 23-31 显示了一个 MFC 程序（HeapMfc）退出时 _AFX_DEBUG_STATE 类的析构函数调用 _CrtDumpMemoryLeaks 函数的过程。

清单 23-31 调试版本的 MFC 程序退出时会自动做内存泄漏检测

```
0:000> k
ChildEBP RetAddr
0012cc4c 10215139 kernel32!OutputDebugStringA      //输出调试信息
0012fc88 10214b6b MSVCRTD!_CrtDbgReport+0x2f9     //CRT 报告函数
```

```

0012fce8 5f404f28 MSVCRTD!_CrtDumpMemoryLeaks+0x4b //转储堆块
0012fcf8 5f404fff MFC42D!_AFX_DEBUG_STATE:::_AFX_DEBUG_STATE+0x18
0012fd04 5f49141f MFC42D!_AFX_DEBUG_STATE::`scalar deleting destructor'+0xf
0012fd24 5f40504f MFC42D!CProcessLocalObject::~CProcessLocalObject+0x37
0012fd30 5f404fa4 MFC42D!CProcessLocal<_AFX_DEBUG_STATE>::~CProcessLocal...+0xf
0012fd38 5f403ece MFC42D!_AFX_DEBUG_STATE:::_AFX_DEBUG_STATE+0x94
0012fd44 5f403fb8 MFC42D!_CRT_INIT+0xde //CRT 初始化和清理函数
0012fd5c 7c9011a7 MFC42D!_DllMainCRTStartup+0xbb //DLL 模块中的 CRT 入口函数
0012fd7c 7c923f31 ntdll!LdrpCallInitRoutine+0x14 //调用初始化和清理例程
0012fe00 7c81ca3e ntdll!LdrShutdownProcess+0x14f //加载器的关闭进程函数
0012fef4 7c81cab6 kernel32!_ExitProcess+0x42 //进程退出
0012ff08 1020acf6 kernel32!ExitProcess+0x14 //进程退出 API
0012ff18 1020abd0 MSVCRTD!_c_exit+0xd6 //CRT 的内部函数
0012ff2c 00402180 MSVCRTD!exit+0x10 //CRT 的退出程序函数
0012ffc0 7c816d4f HeapMfc!WinMainCRTStartup+0x1c0 [crtexe.c @ 345]
0012fff0 00000000 kernel32!BaseProcessStart+0x23 //进程启动函数

```

本节我们介绍了 CRT 的堆块转储功能，下一节我们将介绍如何利用堆块转储来定位内存泄漏的根源。

23.15 泄漏转储

内存泄漏（Memory Leak）是 C/C++ 程序经常遇到的一个棘手问题。简单来说，内存泄漏就是没有释放本来应该释放的内存。我们知道，内存资源是有限的，即使今天的计算机系统大多都安装了几个 GB 的物理内存，而且有足够大的磁盘空间来提供非常多的虚拟内存，但是地址空间仍是有限的，因此大量的泄漏或累积起来的少量泄漏都可能把内存资源用完。

可以把解决内存泄漏问题分为两个步骤，第一步是定位到泄漏的堆块，第二步是定位到泄漏堆块是哪一段代码分配的。本节介绍如何使用 CRT 堆的调试支持来实现这两个目标。

23.15.1 _CrtDumpMemoryLeaks

CRT 设计了一个名为 _CrtDumpMemoryLeaks 函数来检测和报告发生在 CRT 堆中的内存泄漏。举例来说，下面便是这个函数所报告的内存泄漏信息：

```

Detected memory leaks!
Dumping objects ->
(60) normal block at 0x00372488, 10 bytes long.           //报告的标题
Data: <          > CD CD CD CD CD CD CD CD CD          //转储泄漏的堆块
Object dump complete.                                     //堆块#60 的概要信息
                                                        //用户数据区的前 16 个字节
                                                        //转储完成

```

易见，除了第一行的标题信息外，其他信息与 _CrtMemDumpAllObjectsSince 函数的输出是一样的。观察位于 `dbgheap.c` 文件中的 `_CrtDumpMemoryLeaks` 函数源代码（清单 23-32）就可以发现它是基于 `_CrtMemDumpAllObjectsSince` 函数实现的。

清单 23-32 _CrtDumpMemoryLeaks 函数的源代码（格式略微调整）

```

extern "C" _CRTIMP int __cdecl _CrtDumpMemoryLeaks( void )
{
    _CrtMemState msNow;           // 定义一个堆状态结构
    _CrtMemCheckpoint(&msNow);   // 将堆的当前状态记录到 msNow 结构中
    if (msNow.lCounts[_CLIENT_BLOCK] != 0 || msNow.lCounts[_NORMAL_BLOCK] != 0 || 
        (_crtDbgFlag & _CRTDBG_CHECK_CRT_DF) && msNow.lCounts[_CRT_BLOCK] != 0)
    {
        // 如果堆中仍有指定类型的堆块
        _RPT0(_CRT_WARN, "Detected memory leaks!\n"); // 则认为是泄漏，开始报告
        _CrtMemDumpAllObjectsSince(NULL); // 转储堆块
        return TRUE; // 返回真代表检测到泄漏
    }
    return FALSE; // 返回假代表没有检测到泄漏
}

```

从源代码也可以看出，_CrtDumpMemoryLeaks 只是先调用_CrtMemCheckpoint 取得当前堆的统计信息，然后检查以下三个条件：(1) 用户类型(_CLIENT_BLOCK) 的堆块数不等于 0；(2) 常规类型(_NORMAL_BLOCK) 的堆块数不等于 0；(3) CRT 类型(_CRT_BLOCK) 的堆块数不等于 0，而且 CRT 调试标志变量(_crtDbgFlag) 含有_CRTDBG_CHECK_CRT_DF 标志（用于调试 CRT 库本身）。如果以上三个条件之一满足，则认为有内存泄漏，打印提示信息后，调用_CrtMemDumpAllObjectsSince 函数将以上类型的所有堆块转储出来。

23.15.2 何时调用

从上面对_CrtMemDumpAllObjectsSince 函数（以下简称泄漏转储函数）的分析我们知道，无论何时调用这个函数，它都会将当时堆中满足三种条件的块认为是泄漏的块而转储出来。为了防止把能正常释放的块误当作内存泄漏，我们必须选择合适的时机来调用这个函数。如果调用得太早，那么释放内存的应用程序代码还没有执行，很可能导致“误判”。这意味着要等可能导致内存泄漏的代码全部执行结束后再调用泄漏转储函数，那时堆中剩余的堆块如果仍是这段代码所分配的，那么便是内存泄漏了。

因为很多时候我们是在一个进程（应用程序）的范围内来寻找内存泄漏的，所以通常要等应用程序的所有用户代码都执行完毕再调用泄漏转储函数，这时堆中剩余的用户类型块和常规类型块便是内存泄漏了。那么何时才能算是用户代码执行完毕呢？这我们很自然地想到 main（或 WinMain）函数的末尾，因为执行到这里时，整个程序就要退出了。但是这样做仍会将全局对象的析构函数中会释放的内存也报为内存泄漏。因为全局对象的析构函数是在 Main 函数之后才执行的。举例来说，在本章的示例程序 memchk 中定义一个 MemLeakTest 类的全局实例，相关代码如下：

```

class MemLeakTest // 演示内存泄漏的 C++类
{
public: // 公共方法和属性
    MemLeakTest() {m_lpszName=new char[20];} // 构造函数，动态分配内存
    ~MemLeakTest() {delete m_lpszName;} // 析构函数，释放内存
}

```

```

    char * m_lpszName;           //属性
};

MemLeakTest g_MemLeakTest;   //以全局变量形式定义一个类实例

```

在 main 函数的出口处，我们调用泄漏转储函数。正如我们所预料到的，因为此时析构函数~MemLeakTest 还没有执行，m_lpszName 所在的堆块还未释放，所以这个块被当作泄漏的内存而报告出来。避免这个问题的方法是再晚些调用泄漏转储函数，也就是等全局对象也都被析构后，再调用。为了简单地做到这一点，CRT 的设计中已经包含了很好的支持。方法是调用_CrtSetDbgFlag 函数设置_CRTDBG_LEAK_CHECK_DF 标志。一旦设置了该标志，那么 CRT 的退出函数（exit 和 doexit）会在执行完终结器（包含对全局对象的析构）后，再调用_CrtDumpMemoryLeaks。清单 23-33 列出了位于 crt0dat.c 文件中的有关代码。

清单 23-33 CRT 退出函数的泄漏转储支持（摘自 crt0dat.c 中的 doexit 函数）

```

#ifndef _DEBUG           //条件编译，意味着以下代码只对调试版本有效
    if (!fExit && _CrtSetDbgFlag(_CRTDBG_REPORT_FLAG) & //检查静态变量 fExit
        _CRTDBG_LEAK_CHECK_DF) //和调试标志变量中是否包含泄漏检查标志
    {
        fExit = 1;          //如果条件满足
        __freeCrtMemory(); //将 fExit 变量设置为 1，防止多次执行
        _CrtDumpMemoryLeaks(); //释放 CRT 库本身使用的内存块
    }
#endif /* _DEBUG */    //条件编译结束

```

因为 fExit 变量的初始值是 0，所以调用泄漏转储的关键条件是 CRT 的调试标志变量（_crtDbgFlag）中是否包含_CRTDBG_LEAK_CHECK_DF 标志。因为 _crtDbgFlag 变量中默认是不包含这个标志的，所以，如果使用 CRT 的泄漏转储功能，那么应该在程序中设置这个标志，也就是包含头文件 crtdbg.h，并加入如下代码：

```
_crtDbgFlag |= _CRTDBG_LEAK_CHECK_DF;
```

本章的示例程序 MemLeak（code\chap23\memleak）演示了这种方法并模拟了两次内存泄漏，读者可以运行这个程序观察实际的效果。

23.15.3 定位导致泄漏的源代码

了解了_CrtDumpMemoryLeaks 函数的工作原理和调用方法之后，下面我们看看如何利用该函数报告的信息定位导致内存泄漏的源代码。解决内存泄漏的根本方法是将泄漏的内存块在合适的时机释放，这就先要知道这块内存是何时分配的，或者说是哪段代码分配的。

解决这个问题的一个方法是根据转储信息中的堆块序号设置内存分配序号断点，然后重新执行程序，当再分配同样序号的堆块时程序便会中断到调试器，这样便可以利用栈回溯信息来定位分配这个堆块的代码。这种方法的缺点是，必须保证程序执行逻辑的稳定性，使两次执行分配堆块的顺序是一样的，否则触发断点的堆块和要分析

的堆块就会不一致，难以追踪。

第二种方法是让 CRT 转储出的堆块信息中包含源程序文件的文件名和行号。就像清单 23-30 中的第 59 块信息那样。回忆我们介绍调试堆块的头结构 (`_CrtMemBlockHeader`)，它的 `szFileName` 和 `nLine` 字段就是用来记录分配这个堆块的源代码位置的。观察 `_heap_alloc_dbg` 函数，它的最后两个参数就是 `szFileName` 和 `nLine`。也就是说，底层的数据结构和分配函数都包含了很好的支持。那么为什么有些堆块转储出来不包含源代码位置信息呢？答案是顶层的分配函数没有向下提供这样的信息。观察 `malloc` 函数 (`dbgheap.c`) 的源代码，我们就可以清楚地看到这一点：

```
_CRTIMP void * __cdecl malloc (size_t nSize)
{
    void *res = _nh_malloc_dbg(nSize, _newmode, _NORMAL_BLOCK, NULL, 0);
    return res;
}
```

显而易见，`malloc` 调用 `_nh_malloc_dbg` 时把最后两个参数都设置为 0，这两个参数便是 `szFileName` 和 `nLine`。也就是说，源代码信息在这一层就没有向下传递，于是 `_CrtMemBlockHeader` 结构中记录的便也都是 0，转储出来时当然看不到。

要解决以上问题，可以采取如下两种方法之一。

方法一：直接调用调试版本的分配函数或 `new` 运算符，把当前的源文件名和行号传递下去。幸运的是，可以始终使用 `_FILE_` 宏和 `_LINE_` 来表示当前的文件名和行号，因此只要这样写代码：

```
p1 = (char *)_malloc_dbg( 40, _NORMAL_BLOCK, __FILE__, __LINE__ );
p2 = (char *)::operator new(15, _NORMAL_BLOCK, __FILE__, __LINE__);
```

尽管以上函数和运算符只在调试版本中才有实现，但是 `crtdbg.h` 也为它们定义了发布版本的宏或 `inline` 函数，以保证可以顺利编译。

```
#ifndef _DEBUG
#define _malloc_dbg(s, t, f, l)           malloc(s)
    inline void* __cdecl operator new(unsigned int s, int, const char *, int)
    { return ::operator new(s); }
#endif
```

也就是说，在发布版本中，`size` 之外的参数会被丢弃，调试版本的函数会被替换为简单的发布版本。但以上方法的一个明显缺点便是书写比较麻烦，于是便有了使用宏定义的方法二。

方法二：定义 `_CRTDBG_MAP_ALLOC` 标志，可以在项目属性 (C/C++>Preprocessor definitions) 中定义，也可以在源文件中定义。如果在源文件中定义，那么应该在包含头文件 `crtdbg.h` 之前定义，也就是：

```
#define _CRTDBG_MAP_ALLOC
#include <crtdbg.h>
```

因为，如果定义了 `_CRTDBG_MAP_ALLOC`，那么 `crtdbg.h` 中的宏定义就会使编译器将应用程序中的 `malloc` 调用当作宏来解析，直接编译为对 `_malloc_dbg` 函数的调

用，而且参数中指定了当前的源程序文件名（`_FILE_`）和行号（`_LINE_`）。

```
#ifdef _CRTDBG_MAP_ALLOC
#define malloc(s) _malloc_dbg(s, _NORMAL_BLOCK, __FILE__, __LINE__)
#define calloc(c, s) _calloc_dbg(c, s, _NORMAL_BLOCK, __FILE__, __LINE__)
#define realloc(p, s) _realloc_dbg(p, s, _NORMAL_BLOCK, __FILE__, __LINE__)
#endif /* _CRTDBG_MAP_ALLOC */
```

清单 23-34 显示了在定义`_CRTDBG_MAP_ALLOC`标志的情况下，源代码中的`malloc`调用的执行情况。

清单 23-34 定义`_CRTDBG_MAP_ALLOC`标志后`malloc`函数的执行过程

```
_nh_malloc_dbg(unsigned int 10,int 0,int 1,const char* 0x00422558,int 125)
_malloc_dbg(unsigned int 10,int 1,const char * 0x00422558,int 125)line 165+27
main() line 125 + 25 bytes           //用户程序的入口函数
mainCRTStartup() line 206 + 25 bytes //编译器插入的入口函数
KERNEL32! 7c816fd7()               //系统的进程启动函数，即 BaseProcessStart
```

可见，源代码中的`malloc`被当作宏解释，直接编译为对`_malloc_dbg`函数的调用。如果没有定义`_CRTDBG_MAP_ALLOC`标志，那么应用程序中的`malloc`是调用`dbheap.c`中的`malloc`函数。那么既然这个函数也是专门为调试版本设计的，为什么不在这个函数调用`_nh_malloc_dbg`时使用`_FILE_`和`_LINE_`作参数呢？这是因为`_FILE_`和`_LINE_`是两个编译器宏，它们描述的总是当前的源文件。所以即使`malloc`函数在参数中使用这两个参数，堆块中记录的也永远是`dbheap.c`和这个文件中的行号，而不是调用`malloc`函数的源文件和行号。这个原因也导致了`_CRTDBG_MAP_ALLOC`标志的一个缺欠，那就是使用`new`运算符分配的内存堆块被报告出来时的源程序文件总是`crtdbg.h`，也就是下面的样子：

```
c:\program files\microsoft visual studio\vc98\include\crtdbg.h(552) : {55} normal block at 0x00372C58, 20 bytes long.
Data: <          > CD CD
```

微软知识库（KB）的 140858 号文章也描述了这个问题。导致这个问题的原因是`crtdbg.h`中将`new`运算符被定义为一个`inline`函数：

```
#ifdef _CRTDBG_MAP_ALLOC
inline void* __cdecl operator new(unsigned int s)
{ return ::operator new(s, _NORMAL_BLOCK, __FILE__, __LINE__); }
#endif /* _CRTDBG_MAP_ALLOC */
```

基于以上定义，编译器在编译用户代码中的`new`运算符时会将其编译为函数调用，而不是像前面处理`malloc`调用时做宏替换。而当编译器编译这个`inline`函数时，因为这个函数就是实现`crtdbg.h`中的，所以它会将`_FILE_`宏替换为`crtdbg.h`的完整文件名。

避免以上问题的一个方法是像前面讲的那样，直接在源代码中调用调试版本的`new`操作符。另外一种做法是在使用`new`运算符的源程序文件中加入如下宏定义：

```
#ifdef _DEBUG
#define MYDEBUG_NEW new( _NORMAL_BLOCK, __FILE__, __LINE__)
#define new MYDEBUG_NEW
#endif// _DEBUG
```

有了以上宏定义后，编译器会将程序中的 new 运算符替换为 new(_NORMAL_BLOCK, _FILE_, _LINE_), 从而间接实现方法一中的调用调试版本的 new 运算符。MFC 类框架定义了 DEBUG_NEW 宏，其原理与上面的 MYDEBUG_NEW 相同。

归纳一下，要利用方法二实现 CRT 内存分配函数和 new 操作符分配的堆块都含有源程序文件名，那么应该同时定义_CRTDBG_MAP_ALLOC 和类似 MYDEBUG_NEW 的宏。

最后要说明的是，本节介绍的定位内存泄漏的方法只适用于调试版本，而且检查的是 CRT 堆，并不检查应用程序自己创建的其他堆。本章前半部分介绍的 Win32 堆的调试支持没有这些局限。

23.16 本章总结

内存有关的软件问题是软件开发和维护中比较常见和棘手的一类问题。因为这类问题经常牵涉到方方面面的很多个要素，所以很多时候我们会觉得无从下手和无计可施。本章使用较大的篇幅详细介绍了 Win32 堆（23.1~23.10 节）和 CRT 堆（23.11~23.15 节）的工作原理和调试支持。Win32 堆是 Windows 操作系统的重要部分，大多数 Windows 应用程序都直接或间接地使用了 Win32 堆。CRT 堆是 C 运行库所使用的堆，大多数时候，CRT 堆是建立在 Win32 堆之上的。Win32 堆和 CRT 堆（调试版本）都提供了丰富的调试支持，以协助我们发现各种内存错误，表 23-9 归纳了 Win32 堆和 CRT 堆的常用调试功能。

表 23-9 Win32 堆和 CRT 堆的常用调试功能

调试机制	描述	典型应用	适用版本	启用方法
栈回溯数据库(UST)	记录堆分配函数的调用过程，详见第 23.7 节	检查内存泄漏	调试/发布	gflags /i +ust
堆尾检查(htc)	在堆块末尾设置模式字段，详见第 23.8.3 节	检测堆缓冲区溢出	调试/发布	gflags /i +htc
释放检查(hfc)	在释放堆块时进行检查	发现多次释放	调试/发布	gflags /i +hfc
参数检查(hpc)	检查调用堆函数时的参数	发现错误的参数	调试/发布	gflags /i +hpc
页堆(DPH)	为堆块分配栅栏页，详见第 23.9 节	检测堆缓冲区溢出	调试/发布	gflags /r +hpa
内存分配序号断点	针对 CRT 堆的堆块分配序号设置断点	检查某一次堆块分配的细节	调试	设置变量 crtBreakAlloc
内存状态快照(检查点)	记录 CRT 堆的统计状态	检查内存泄漏	调试	调用 CrtMemCheckpoint
堆块转储(Dump)	转储 CRT 堆中的堆块，详见第 23.14 节	检查堆块和内存泄漏	调试	设置 crtDbgFlag 标志或者调用转储函数

为了便于识别出堆块的不同区域，堆管理器会使用不同的字节模式来填充特定的区域。为了方便大家在调试时检索不同字节模式的含义，表 23-10 归纳了常见的字节模式和它们的用途。

表 23-10 Win32 堆和 CRT 堆的常用字节模式

0xFEEEFEEE	Win32 堆	填充空闲块的数据区	堆块数据区
0xBAADF00D	Win32 堆	填充新分配块的数据区	用户请求长度
0xAB	Win32 堆	填充在用户数据之后，用于检测堆溢出	8 或 16 字节
0xFD	CRT 调试堆	填充用户数据区前后的隔离区	各 4 个字节
0xDD	CRT 调试堆	填充释放的堆块 (dead land)	整个堆块大小 (包括块头、数据区和隔离区)
0xCD	CRT 调试堆	填充新分配的堆块 (clean land)	用户数据区大小

页堆和准页堆使用的填充模式列在表 23-8 中。

参考文献

1. Mark E. Russinovich and David A. Solomon. Microsoft Windows Internals 4th edition. Microsoft Press, 2005
2. Visual Studio 6.0 和 Visual Studio 2005 的 CRT 源代码. Microsoft Corporation

异常处理代码的编译

异常处理是软件开发和调试中的一个永恒话题。在第 2 篇和第 3 篇中我们分别从 CPU 和操作系统的角度介绍了异常有关的概念。本章我们将从编程语言和编译器的角度进一步探索 Windows 程序中的异常处理代码是如何被编译和执行的。

根据运行模式和编程语言的不同，Windows 系统中的程序可以选择使用不同的异常处理机制，比如驱动程序可以使用结构化异常处理（SEH）机制，C++语言编写的应用程序可以使用向量化异常处理和 C++ 标准定义的异常处理机制。其中，SEH 是其他异常处理机制的基础，因此，我们将围绕结构化异常处理代码的编译为中心展开讨论。

24.1 概览

结构化异常处理是 Windows 操作系统内建的异常处理机制。从理论上讲，各种编程语言和编译器都可以使用该机制。其主要完成两项任务。

第一，定义必要的关键字来表示异常处理逻辑，供编程人员使用。比如，VC 编译器定义了 `_try`、`_except`、`_finally` 三个扩展关键字，允许 C 和 C++ 程序使用这套关键字来编写异常处理代码。

第二，实现对以上关键字的编译，将使用这些关键字编写的异常处理代码与操作系统的 SEH 机制衔接起来。

我们在第 11 章（11.4 节）介绍过 SEH 的用法。概括来说，一段使用 SEH 的异常代码由被保护体、过滤表达式和异常处理块三部分组成，即：

```
_try
{
    //被保护体 (guarded body), 也就是要保护的代码块。
}
_except(过滤表达式)
{
    //异常处理块(exception-handling block)
}
```

从外部行为的角度来看，结构化异常处理的基本规则是，如果被保护体中的代码发生了异常，不论是 CPU 级的硬件异常还是软件发起的软件异常，系统都应该评估过滤表达式的内容，也就是执行过滤表达式中的代码。这意味着，程序的执行路线是从被保护体中飞跃到过滤表达式中的。要正确地飞跃到表达式中，显然不那么简单，需要准确地知道过滤表达式的位置，又要保持栈的平衡。

为了实现这样的飞跃，编译器在编译期间必须产生必要的代码和数据结构与系统的异常分发函数的密切配合。概括来说，首先要分析出异常处理代码的结构，并对每个部分进行必要的封装和标记。然后注册异常处理器函数，以便有异常发生时被调用。记录和注册异常处理器的方式（也就是系统分发异常时的寻找方式）主要有两种。

- **栈帧 (Stack Frame)**：将异常处理器注册在所在函数的栈帧中。使用这种方式注册的异常处理经常被称为基于帧的异常处理 (Frame Based Exception Handling)。32 位的 Windows 系统 (x86) 使用的是此种方式。
- **表格 (Table)**：将异常处理器的基本信息以表格的形式存储在可执行文件 (PE) 的数据段中，这种方式被简称为基于表的异常处理 (Table Based Exception Handling)。64 位 Windows 系统 (x64) 中的 64 位程序使用了这种方式。

第 11 章介绍了 Windows 系统中负责异常分发和处理的数据结构和函数，包括分发异常的中枢——KiDispatchException 函数，用来恢复继续执行的 ZwContinue 系统服务。这些函数有些是操作系统的工作函数，有些是系统提供给应用程序的编程接口。为了支持异常分发，在 Windows 的内核模块和 NTDLL 模块中，还包括了一系列负责将异常分发到异常处理器的 RTL 函数。RTL 即运行期库 (Runtime Library)，是操作系统为了满足自身代码的执行需要而集成到系统模块中的，它们大多都以 Rtl 开头，位于 NTOSKRNL.EXE 中是为了满足内核模块的需要，位于 NTDLL.DLL 中是为了满足用户态模块的需要。这两套 RTL 的函数名、参数和行为大多都一致，很多函数都是共享同一套源代码的。比如负责结构化异常分发的 RtlDispatchException 函数就是一个典型的例子，位于 NTOSKRNL.EXE 中的 RtlDispatchException 函数用于分发内核态代码的结构化异常，位于 NTDLL.DLL 中的 RtlDispatchException 函数用于分发用户态代码的结构化异常。这既满足了内核态代码和用户态代码都可以使用结构化异常，又保证了结构化异常的处理逻辑不论在内核态还是用户态都是一致的。

异常处理的另一个特征就是线程相关性。也就是说，异常的分发和处理是在线程范围内进行的，异常处理器的注册也是相对线程而言的。理解这一点对于理解系统寻找和调用异常处理器的方法很重要。

从下一节开始，我们将从记录异常注册信息的 FS:[0] 链条开始顺藤摸瓜地逐步理解寻找和执行异常处理代码的过程。

24.2 FS:[0]链条

Windows 系统中的每个用户态线程都拥有一个线程环境块 (Thread Environment Block), 简称 TEB。TEB 结构的具体定义因为 Windows 版本的不同会略有不同, 但可以确定的是, 在 TEB 结构的起始处总有一个被称为线程信息块 (Thread Information Block) 的结构, 简称 TIB。TIB 的第一个字段 ExceptionList 记录的就是用来登记结构化异常处理链表的表头地址。因为在 x86 系统中, 段寄存器 FS 总是指向线程的 TEB/TIB 结构, 也就是说, FS:[0]总是指向结构化异常处理链表的表头, 所以我们把这个链表称为 FS:[0]链条。

24.2.1 TEB 和 TIB 结构

尽管 SDK 和 DDK 的文档中没有描述过 _NT_TIB 结构, 但是在 DDK 和 SDK 的 winnt.h 头文件中都可以看到这个结构的定义 (见清单 24-1)。

清单 24-1 TIB (_NT_TIB) 结构 (winnt.h)

```
typedef struct _NT_TIB {
    struct _EXCEPTION_REGISTRATION_RECORD *ExceptionList; //指向异常登记链表
    PVOID StackBase;           //栈基址
    PVOID StackLimit;          //栈的边界
    PVOID SubSystemTib;        //指向描述子系统 (OS2 等) 相关信息的结构
    union {                   //从 NT 3.51 SP3 开始改为联合结构以存储纤程信息
        PVOID FiberData;       //指向描述纤程的信息结构
        DWORD Version;         //本结构的版本号
    };
    PVOID ArbitraryUserPointer; //某些时候用在内核态和用户态间传递信息
    struct _NT_TIB *Self;       //本结构的线性地址
} NT_TIB;
```

可见第一个地段就是与异常有关的, 我们稍后会对其做详细介绍。现在来看看是如何访问到 TIB 结构的。在 NTDLL 中有一个未公开的 NtCurrentTeb 函数, 用来取得当前线程的 TEB 地址。在 x86 系统中, 它的实现如下:

```
ntdll!__NtCurrentTeb:
7c901250 64a118000000      mov     eax,dword ptr fs:[00000018h]
7c901256 c3                 ret
```

在 x86 系统中, 当线程在用户态执行时, 段寄存器 FS 总是指向当前线程的 TEB 结构, 也就是说, FS:[0]的内容就是 TEB 结构的起始 4 字节的内容, 因为 TIB 位于 TEB 的起始处, 所以 FS:[0]的内容实际上就是 TIB 结构的最前 4 个字节, 依此类推, fs:[00000018h] 就是 NT_TIB 的 Self 字段, 也就是 TEB 和 TIB 结构在进程空间中的虚拟地址。

因为 NtCurrentTeb 函数的代码非常短, 所以它经常被编译器做 inline 处理了。例如 GetLastError API 就是这样的。以下是这个 API 的反汇编代码:

```
kernel32!GetLastError:
7c830699 64a118000000      mov     eax,dword ptr fs:[00000018h] // 得到逻辑地址
7c83069f 8b4034             mov     eax,dword ptr [eax+34h]   // GetLastErrorValue 字段
7c8306a2 c3
```

我们可以看到它是先通过 `fs:[00000018h]` 得到 TEB/TIB 结构的虚拟地址，然后返回 `LastErrorMessage` 字段的值。或者说是通过类似这样 `return NtCurrentTeb() ->LastErrorMessage;` 的源代码编译出来的。

使用 `dd fs:[0]` 命令可以观察 `fs` 段的内容：

```
0:001> dd fs:[0]
0038:00000000 0123ffe4 01240000 0123f000 00000000
0038:00000010 00001e00 00000000 7ffde000 00000000
```

其中 `0123ffe4` 是 `ExceptionList` 字段，`01240000` 是 `StackBase` 字段，`0123f000` 是 `StackLimit` 字段，`00000000` 是 `SubSystemTib` 字段，`0x7ffde000` 就是 `Self` 字段，即这个 TIB 结构的虚拟地址，有了这个地址后，就可以使用 `dt ntdll!_NT_TIB 7ffde000` 来结构化显示以上的字段了（见清单 24-2）。

清单 24-2 使用 dt 命令观察 TIB 结构

```
0:001> dt ntdll!_NT_TIB 7ffde000
+0x000 ExceptionList      : 0x0123ffe4 _EXCEPTION_REGISTRATION_RECORD
+0x004 StackBase          : 0x01240000
+0x008 StackLimit         : 0x0123f000
+0x00c SubSystemTib       : (null)
+0x010 FiberData          : 0x00001e00
+0x010 Version            : 0x1e00
+0x014 ArbitraryUserPointer: (null)
+0x018 Self               : 0x7ffde000 _NT_TIB
```

在内核模式中，内核态的代码仍然可以通过 `FS:[0]` 来引用到 TIB 结构，这个结构位于 CPU 的 KPCR 结构的开始处。KPCR 是与处理器相关的，系统在启动期间会为每个 CPU 创建一个 KPCR 结构和一个 KPRCB 结构。在内核调试中，可以使用 `!pcr` 命令查看当前处理器的 KPCR 结构地址：

```
kd> !pcr
KPCR for Processor 0 at ffdfff000:
    Major 1 Minor 1
        NtTib.ExceptionList: 805417d8
[...省略其他显示]
```

然后，可以使用 `dt nt!_KPCR ffdfff000` 命令来显示 KPCR 结构，使用 `dt nt!_NT_TIB ffdfff000` 命令来显示 NT_TIB 结构，结果从略。考虑到内核态的情况与用户态的情况类似，而且更简单些，因此接下来我们只讨论用户态的情况。

24.2.2 ExceptionList 字段

下面我们集中讨论 TIB 的 `ExceptionList` 字段，因为该字段位于 TIB 结构的起始处，所以 `FS:[0]` 的内容就是 `ExceptionList` 字段的内容。对于清单 24-2 所示的情况，它的值就是 `0123ffe4`。使用 `dt` 命令可以进一步观察这个字段的内容：

```
0:001> dt _EXCEPTION_REGISTRATION_RECORD 0x0123ffe4
+0x000 Next      : 0xffffffff _EXCEPTION_REGISTRATION_RECORD
+0x004 Handler   : 0x7c90ee18 ntdll!_except_handler3+0.
```

其中 Next 字段用来指向下一个 _EXCEPTION_REGISTRATION_RECORD 结构，0xffffffff(-1) 代表这是最后一个节点。Handler 字段用来指向这个异常处理器的处理函数，它的值 0x7c90ee18 指向的是 NTDLL.DLL 中的 _except_handler3 函数，并具有 SEH 处理函数的标准函数原型：

```
EXCEPTION_DISPOSITION SehHandler( _EXCEPTION_RECORD *ExceptionRecord,
    void * EstablisherFrame, _CONTEXT *ContextRecord, void * DispatcherContext);
```

其中第一个参数 ExceptionRecord 指向 EXCEPTION_RECORD 结构，用来描述要处理的异常，第 11.2 节我们介绍了这个结构的定义。EstablisherFrame 参数指向的是放在栈帧中的异常登记结构，之所以叫 Establisher Frame，是因为它描述的是建立（登记）异常处理器的那个函数栈帧，也就是包含 __try{}__except 代码的那个函数的栈帧。ContextRecord 参数是用来传递发生异常时的线程上下文结构，DispatcherContext 供异常分发函数来传递额外信息。

对 ExceptionList 的值和 StackBase 及 StackLimit 的值进行比较，可以看出，ExceptionList 的值是介于 StackLimit 和 StackBase 之间的。也就是说，FS:[0]链表的各个节点 (_EXCEPTION_REGISTRATION_RECORD) 是存储在栈上的。从 TIB 结构的地址（本例为 7fffdde000）可以看出 TIB 结构不是在栈上的，这意味着 ExceptionList 字段本身不在栈上，但是它指向位于栈上的 EXCEPTION_REGISTRATION_RECORD 结构。可以把 FS 段寄存器看作是索引 TEB/TIB 结构的一种便捷方式，而 FS:[0] 是索引结构化异常处理链的便捷方式。

介绍到这里，我们知道系统可以通过 FS:[0] 这种便捷方式引用到 ExceptionList 字段，获得一个链表的头指针，这个链表的每个节点是一个 _EXCEPTION_REGISTRATION_RECORD 结构，描述了一个结构化异常处理器的处理函数（Handler）。当有异常发生时，系统只要依次调用这个链表上的处理函数来分发异常。

概而言之，可以把结构化异常处理看作是操作系统与用户代码协同处理软硬件异常的一种模型，而 FS:[0] 链条便是这二者间协作的接口。当有异常需要处理时，操作系统通过 FS:[0] 链条寻找异常处理器，给用户代码处理异常情况的机会。接下来我们将介绍用户代码是如何登记异常处理器的。

24.2.3 登记异常处理器

在理解了 FS:[0] 链条的工作原理后，便可以手工编写代码来登记和注销异常处理器了。首先需要编写一个异常处理器函数，它应该具有标准的 SehHandler 原型。然后在栈上建立一个 EXCEPTION_REGISTRATION_RECORD 结构，并把这个结构的地址注册到 FS:[0] 链表中。以下汇编代码便实现了这一目标：

```
push    seh_handler          // 处理函数的地址
push    FS:[0]                // 前一个 SEH 处理器的地址
mov     FS:[0],ESP           // 登记新的结构
```

其中，第 1 行的作用是将编写好的异常处理函数（SehHandler）的地址压入栈，第 2 行是将 FS:[0]（也就是 ExceptionList 字段）的当前值压入栈，这样，在栈上便动态建立了一个 EXCEPTION_REGISTRATION_RECORD 结构，栈顶地址便是这个结构的地址，也就是 ESP 寄存器的值目前正指向的这个结构，因此，第 3 行便把 ESP 寄存器的内容写入到了 FS:[0]，实质上是将刚构建的 EXCEPTION_REGISTRATION_RECORD 结构的地址赋给了 ExceptionList 字段，即将新的异常处理器插到了 FS:[0] 链条的表头。到这里，一个新的异常处理器便登记（安装）完毕了，当 CPU 继续执行这段代码下面的代码时，如果有异常发生，那么当系统在遍历 FS:[0] 链条时便会首先找到这个异常处理器，给其处理机会。因此，以上代码片段一旦插入，那么它之后的代码便进入了它所安装的异常处理器的“保护”范围，直到这个处理器被注销为止。

可以使用如下代码来注销前面登记的异常处理器：

```
mov    eax, [ESP]           //从栈顶取得前一个异常登记结构的地址
mov    FS:[0], EAX          //将前一个异常结构的地址赋给 FS:[0]
add    esp, 8               //清理栈上的异常登记结构
```

在理解注销异常处理器的代码之前，有必要重申一下栈平衡原则。所谓栈平衡，就是指栈的压入（push）和弹出（pop）动作的对称性，让栈的每个使用者都觉得当它每次使用栈时，栈的状态和它上次使用时是一样的。这是不同函数或同一函数的不同部分共享栈的基本原则。登记和注销异常处理器是典型的对称动作，因此，无论在这两个操作之间执行了多少代码，当执行注销动作时，栈的状态（栈顶和栈内的内容）应该和安装动作后的状态是一样的，也就是说，栈顶存放的是一个 EXCEPTION_REGISTRATION_RECORD，前 4 个字节就是 Next 字段。因此，第 1 行是把这个指针的内容赋给 EAX，第 2 行再将其写到 FS:[0]（ExceptionList），这样，FS:[0] 的内容便又恢复到了安装这个结构化异常处理器前的状态。

使用 WinDBG 的 !exchain 命令可以列出当前线程的 FS:[0] 链条。第 24.4 节的清单 24-8 列出了使用该命令的一个实例。

24.3 遍历 FS:[0] 链条

上一节我们介绍了 FS:[0] 链条，这一节继续讨论当有异常发生时，系统是如何遍历这个链条来寻找和执行其中的异常处理器的，让我们从 RtlDispatchException 函数讲起。

24.3.1 RtlDispatchException

简单来说，RtlDispatchException 函数的工作过程就是找到注册在线程信息块（TIB）中异常处理器链表的头节点，然后依次访问每个节点，调用它的处理器函数，直到有人处理了异常，或者到了链表的末尾，图 24-1 画出了 RtlDispatchException 函数的工作流程。

图中的 `RtlCallVectoredExceptionHandlers` 为调用向量化异常处理器(VEH), 这一操作只存在于用户态版本中。`RtlpGetRegistrationHead` 用来取得结构化异常处理器链表的首节点, 从其反汇编代码可以看到就是返回 FS:[0]处的 `ExceptionList` 字段内容:

```
0:001> u ntdll!RtlpGetRegistrationHead
ntdll!RtlpGetRegistrationHead:
7c90392d 64a100000000    mov     eax,dword ptr fs:[00000000h]
7c903933 c3              ret
```

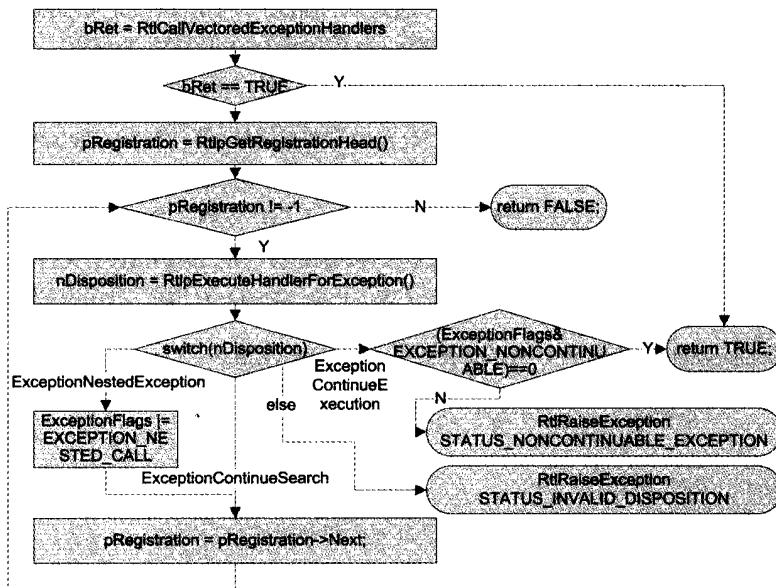


图 24-1 RtlDispatchException 函数的工作流程

顾名思义, `RtlpExecuteHandlerForException` 是执行 Handler 函数的。我们暂不考虑它的内部细节, 现在只须知道它会返回一个枚举类型 `EXCEPTION_DISPOSITION` 的常量, 表 24-1 列出了这个常量的可能值和含义。

表 24-1 EXCEPTION_DISPOSITION 常量和含义

常量	取值	含义
<code>ExceptionContinueExecution</code>	0	恢复执行触发异常的代码
<code>ExceptionContinueSearch</code>	1	继续寻找下一个异常处理器
<code>ExceptionNestedException</code>	2	在调用 Handler 函数过程中又发生了异常, 即嵌套异常

如果 `RtlpExecuteHandlerForException` 函数返回 `ExceptionContinueExecution`, 而且异常标志中包含了 `EXCEPTION_NONCONTINUABLE` (值为 1) 标志, 那么说明在试图恢复执行不可继续的异常, 因此 `RtlDispatchException` 会调用 `RtlRaiseException()` 再次抛出异常, 异常的代码为 `STATUS_NONCONTINUABLE_EXCEPTION`。如果异常标志中不包含 `EXCEPTION_NONCONTINUABLE` 标志, 那么 `RtlpExecuteHandlerForException` 会

返回 TRUE，对于用户态异常，这会导致 KiUserExceptionDispatcher 函数调用 ZwContinue 服务，让 CPU 返回到异常发生处继续执行。

对于无效返回值，RtlDispatchException 会抛出新的异常，异常代码为 STATUS_INVALID_DISPOSITION(0xC0000026)。RtlRaiseException 函数如果成功，便不会再返回到 RtlDispatchException 函数中。

如果 RtlDispatchException 遍历到了链表的最后一个节点，那么 RtlDispatchException 便返回 FALSE，这是它的最后一个出口。正如我们在第 12 章所介绍的，每个线程在启动时，系统函数会为其登记默认的异常处理器，这个异常处理器总是会处理异常，因此，一般情况下不会执行到这个出口。

微软没有公开过 RtlDispatchException 函数的内部细节，Matt Pietrek 在 1997 年发表的文章（参考文献 1）中给出了该函数当时版本的伪代码。随着 Windows 系统的发展，这个函数中也加入了一些新的功能，比如 VEH 支持、DEP（Data Execution Prevent）支持、Server 2003 SP0 和 Windows XP SP2 所引入的 SAFESEH 支持等。清单 24-3 给出了针对目前版本的 Windows 系统（Windows Vista）更新后的伪代码。

清单 24-3 RtlDispatchException 函数的伪代码（Windows Vista）

```

1  BOOLEAN RtlDispatchException( PEXCEPTION_RECORD pExcptRec,
2                               CONTEXT * pContext )
3  {
4      DWORD    dwStackUserBase;
5      DWORD    dwStackUserTop;
6      PEXCEPTION_REGISTRATION_RECORD pRegistration;
7      PEXCEPTION_REGISTRATION_RECORD pNestedRegistration=0;
8      DWORD hLog;
9      EXCEPTION_DISPOSITION nDisposition;
10     DISPATCHER_CONTEXT DispatcherContext;
11
12     if(RtlCallVectoredExceptionHandlers(pExcptRec, pContext))
13         return TRUE;
14
15     // 从 FS:[4]和 FS:[8]读取栈的基地址 (StackBase) 和边界 (Stack Limit)
16     RtIpGetStackLimits( &dwStackUserBase, &dwStackUserTop );
17
18     pRegistration = RtIpGetRegistrationHead();
19
20     while ( -1 != pRegistration)
21     {
22         PVOID justPastRegistrationFrame = &pRegistration + 8;
23         if ( dwStackUserBase > justPastRegistrationFrame )
24         {
25             pExcptRec->ExceptionFlags |= EH_STACK_INVALID;
26             return FALSE;
27         }
28
29         if ( dwStackUserTop < justPastRegistrationFrame )
30         {
31             pExcptRec->ExceptionFlags |= EH_STACK_INVALID;
32             return FALSE;
33         }
34
35         if ( pRegistration & 3 )

```

```

36      {
37          pExcptRec->ExceptionFlags |= EH_STACK_INVALID;
38          return FALSE;
39      }
40
41 // 异常处理器不应该是栈中的代码，这是为了支持 DEP 功能
42 if(pRegistration-> Handler> dwStackUserTop &&
43     pRegistration-> Handler < dwStackUserBase)
44     return FALSE;
45
46 // 为了支持 Server 2003 和 XP SP2 引入的 SafeSEH 功能
47 if(!RtlIsValidHandler(pRegistration-> Handler))
48     return FALSE;
49
50 if (NtGlobalFlag & FLG_ENABLE_EXCEPTION_LOGGING /*0x80*/)
51 {
52     hLog = RtlpLogExceptionHandler( pExcptRec, pContext, 0,
53                                     pRegistration, 0x10 );
54 }
55
56 nDisposition = RtlpExecuteHandlerForException(pExcptRec,
57                                               pRegistration,
58                                               pContext, & DispatcherContext,
59                                               pRegistration->Handler );
60
61 if (NtGlobalFlag & FLG_ENABLE_EXCEPTION_LOGGING)
62     RtlpLogLastExceptionDisposition( hLog, Disposition );
63
64 if (pNestedRegistration == pRegistration )
65 {
66     pExcptRec->ExceptionFlags &= ~ EXCEPTION_NESTED_CALL;
67     pNestedRegistration = NULL;
68 }
69
70 EXCEPTION_RECORD excptRec2;
71 switch ( nDisposition )
72 {
73 case ExceptionContinueExecution : // 0
74     if ( pExcptRec->ExceptionFlags & EXCEPTION_NONCONTINUABLE )
75     {
76         excptRec2.ExceptionRecord = pExcptRec;
77         excptRec2. ExceptionCode = STATUS_NONCONTINUABLE_EXCEPTION;
78         excptRec2.ExceptionFlags = EXCEPTION_NONCONTINUABLE;
79         excptRec2.NumberParameters = 0
80         RtlRaiseException( &excptRec2 );
81     }
82     else
83         return TRUE;
84 }
85 case ExceptionContinueSearch: // 1
86     break;
87 case ExceptionNestedException:
88     pExcptRec->ExceptionFlags |= EXCEPTION_NESTED_CALL;
89     if (DispatcherContext.RegistrationPointer >
90         pNestedRegistration)
91         pNestedRegistration = DispatcherContext .RegistrationPointer;
92     break;
93 default:
94     excptRec2.ExceptionRecord = pExcptRec;
95     excptRec2. ExceptionCode = STATUS_INVALID_DISPOSITION;
96     excptRec2.ExceptionFlags = EXCEPTION_NONCONTINUABLE;
97     excptRec2.NumberParameters = 0
98     RtlRaiseException( &excptRec2 );

```

```

98         }
99         pRegistration = pRegistration->Next;
100    }
101    return FALSE;
102 }

```

其中，第 12~13 行是调用 Windows XP 引入的向量化异常处理器（VEH）。因为执行这个调用时还没有遍历 FS:[0]链表，因此 VEH 处理器比 SEH 处理器先得到异常处理机会。第 18 行是得到 FS:[0]链表的表头，第 20~100 行是一个 while 循环，遍历链表中的每个异常处理器。

第 22~39 行是检查链表中的异常登记结构是否有效；第 42~44 行是检查异常处理函数是否位于栈空间中，以防止意外执行攻击程序动态布置在栈上的代码。第 46~48 行是调用 SAFESEH 机制来检查异常处理函数的有效性。第 56~59 行是执行异常处理函数，即 Handler 字段所代表的函数，我们将在下一节讨论其细节。第 71~98 行的 switch 结构针对执行异常处理函数的结果采取不同的动作。

24.3.2 KiUserExceptionDispatcher

对于发生在内核态代码中的异常，KiExceptionDispatcher 便直接调用内核版本的 RtlDispatchException 函数寻找异常处理器，也即从 FS:[0]处的链表头开始，不过此时 FS 指向的内容与用户态的不同。对于发生在用户态代码中的异常，KiExceptionDispatcher 需要把线程上下文的 EIP 字段指向 KeUserExceptionDispatcher 变量所记录的函数，也就是 NTDLL 中的 KiUserExceptionDispatcher 函数。然后 KiUserExceptionDispatcher 函数再调用 RtlDispatchException 函数（见清单 24-4）。

清单 24-4 KiUserExceptionDispatcher 函数的伪代码

```

1  VOID KiUserExceptionDispatcher( PEXCEPTION_RECORD pExcptRec, CONTEXT *
2   pContext )
3   {
4     DWORD dwRetVal;
5     if ( RtlDispatchException( pExcptRec, pContext ) )
6       dwRetVal = NtContinue( pContext, 0 );
7     else
8       dwRetVal = NtRaiseException( pExcptRec, pContext, 0 );
9
10    EXCEPTION_RECORD excptRec2;
11    excptRec2.ExceptionCode = dwRetVal;
12    excptRec2.ExceptionFlags = EXCEPTION_NONCONTINUABLE;
13    excptRec2.ExceptionRecord = pExcptRec;
14    excptRec2.NumberParameters = 0;
15    RtlRaiseException( &excptRec2 );
16 }

```

第 5~8 行的含义是如果 RtlDispatchException 返回真，就调用 NtContinue 系统服务，让程序继续执行，否则便调用 NtRaiseException 服务发起对该异常的第二轮分发。不论是 NtContinue 还是 NtRaiseException，如果执行成功，都不会再返回到

KiUserExceptionDispatcher 函数中，因此后半部分（第 10~15 行）的代码只有当调用 NtContinue 和 NtRaiseException 失败时才会执行。

24.4 执行异常处理函数

前面两节我们分别介绍了如何注册 SEH 处理函数，以及当有异常发生时系统是如何遍历 FS:[0]链条寻找 SEH 处理函数的。本节将介绍系统执行 SEH 处理函数的细节。

24.4.1 SehRaw 实例

为了便于理解，我们编写了一个简单的控制台程序 SehRaw（见清单 24-5），名字的含义是手工注册原始的 SEH 异常处理函数，目的是与后面将介绍的编译器产生的自动注册方法相区别。

清单 24-5 SehRaw 程序的源代码

```

1  #include "stdafx.h"
2  #include <windows.h>
3
4  EXCEPTION_DISPOSITION __cdecl _raw_seh_handler(
5      struct _EXCEPTION_RECORD *ExceptionRecord,
6      void * EstablisherFrame, struct _CONTEXT *ContextRecord,
7      void * DispatcherContext )
8  {
9      printf("_raw_seh_handler: code-0x%x, flags-0x%x\n",
10         ExceptionRecord->ExceptionCode,
11         ExceptionRecord->ExceptionFlags);
12
13     if(ExceptionRecord->ExceptionCode ==
14         STATUS_INTEGER_DIVIDE_BY_ZERO)
15     {
16         ContextRecord->Ecx = 10;
17         return ExceptionContinueExecution;
18     }
19     return ExceptionContinueSearch;
20 }
21
22 int main()
23 {
24     __asm
25     {
26         push    OFFSET _raw_seh_handler
27         push    FS:[0]
28         mov     FS:[0],ESP           //登记异常处理器
29
30         xor     edx, edx          //将 EDX 寄存器清 0
31         mov     eax, 100           //设置 EAX 寄存器，被除数
32         xor     ecx, ecx          //将 ECX 寄存器清 0
33         idiv   ecx              //EDX:EAX 除以 ECX
34
35         mov     eax,[ESP]          //取 Next 字段的值
36         mov     FS:[0], EAX          //写入到 TIB 的 ExceptionList 字段
37         add     esp, 8             //释放异常登记结构
38     }

```

```

39
40     printf("SehRaw exits\n");
41     return 0;
42 }

```

其中，第 4~20 行定义了一个符合 `SehHandler` 原型的异常处理函数 `_raw_seh_handler`，第 26~28 行使用前面介绍的手工方法将其注册到 FS:[0] 链条中。第 33~33 行故意执行了一个除零操作以引发 CPU 级的除零异常。在 `_raw_seh_handler` 被调用后，它如果检测到发生的是除零异常（第 13~14 行），便将上下文结构中的 ECX 寄存器的值改为非零（10），然后返回 `ExceptionContinueExecution` 让系统继续执行导致异常的代码。第二次执行除法指令时，因为除数不再为 0 所以便可以顺利执行了。

24.4.2 执行异常处理函数

下面我们来看一下异常发生后，系统调用和执行 `_raw_seh_handler` 函数的过程。清单 24-6 显示的是系统调用 `_raw_seh_handler` 函数的过程。

清单 24-6 系统调用 `_raw_seh_handler` 函数的过程

0:000> kn	
# ChildEBP RetAddr	
00 0012fb00 7c9037bf	SehRaw!_raw_seh_handler //异常处理函数
01 0012fb04 7c90378b	ntdll!ExecuteHandler2+0x26 //见正文
02 0012fc84 7c90eafa	ntdll!ExecuteHandler+0x24 //见正文
03 0012fc84 0040105c	KiUserExceptionDispatcher+0xe //参见 11.3.3 节
04 0012ff80 00401165	SehRaw!main+0x1c //入口函数
05 0012ffc0 7c816f7	SehRaw!mainCRTStartup+0xb4 //编译器插入的入口函数
06 0012fff0 00000000	kernel32!BaseProcessStart+0x23 //系统的进程启动函数

首先，在以上栈回溯中缺少了关于 `RtlDispatchException` 和 `RtlpExecuteHandlerForException` 函数的信息，它们应该位于栈帧 #02 和 #03 之间。也就是说，`KiUserExceptionDispatcher` 会调用 `RtlDispatchException`，`RtlDispatchException` 又调用 `RtlpExecuteHandlerForException`，然后再调用（实际上是跳转）`ExecuteHandler`。`ExecuteHandler` 函数的原型如下：

```

EXCEPTION_DISPOSITION ExecuteHandler( PEXCEPTION_RECORD pExceptionRecord,
                                         PEXCEPTION_REGISTRATION pExceptionRegistration,
                                         CONTEXT * pContext, DISPATCHER_CONTEXT * pDispatcherContext,
                                         SehHandler pfnHandler )

```

其中 `pDispatcherContext` 通常指向 `RtlDispatchException` 函数所定义的一个局部变量 `DispatcherContext`（参见清单 24-3 的第 10 行），其用途是供系统的异常分发函数来传递上下文信息。`pfnHandler` 就是 `_EXCEPTION_REGISTRATION_RECORD` 结构的 `Handler` 字段中的函数地址。

在 Windows XP 之前，`ExecuteHandler` 内部便调用 `pfnHandler` 所指向的异常处理函数。XP 引入了 `ExecuteHandler2` 函数（同样原型）来调用异常处理函数，而 `ExecutePPHandler` 退化为只是简单地调用 `ExecuteHandler2`。清单 24-7 给出了 `ExecuteHandler2` 函数的汇编代码。

清单 24-7 ExecuteHandler2 函数

```

1  ntdll!ExecuteHandler2:
2  7c903799 55          push   ebp
3  7c90379a 8bec        mov    ebp,esp
4  7c90379c ff750c      push   dword ptr [ebp+0Ch]
5  7c90379f 52          push   edx
6  7c9037a0 64ff3500000000 push   dword ptr fs:[0]
7  7c9037a7 64892500000000 mov    dword ptr fs:[0],esp
8  7c9037ae ff7514      push   dword ptr [ebp+14h]
9  7c9037b1 ff7510      push   dword ptr [ebp+10h]
10 7c9037b4 ff750c       push   dword ptr [ebp+0Ch]
11 7c9037b7 ff7508       push   dword ptr [ebp+8]
12 7c9037ba 8b4d18      mov    ecx,dword ptr [ebp+18h]
13 7c9037bd ffd1         call   ecx
14 7c9037bf 648b2500000000 mov    esp,dword ptr fs:[0]
15 7c9037c6 648f0500000000 pop    dword ptr fs:[0]
16 7c9037cd 8be5         mov    esp,ebp
17 7c9037cf 5d          pop    ebp
18 7c9037d0 c21400      ret    14h

```

从上面的清单可以看出，在调用用户的处理函数之前，ExecuteHandler2 会先登记一个异常处理器（第 4~7 行），待用户处理函数返回后（第 11 行），再将其注销（第 12 和 13 行）。这个异常处理器便是所谓的内嵌异常处理器（Nested Exception Handler），其目的是用来捕捉异常处理函数中可能引发的异常。内嵌异常处理器的处理函数是由 EDX 寄存器指定的（第 4 行），RtlpExecuteHandlerForException 在调用 ExecuteHandler 前会将合适的函数地址设置到 EDX 寄存器中：

```

ntdll!RtlpExecuteHandlerForException:
7c903753 bad837907c mov    edx,offset ntdll!ExecuteHandler2+0x3a (7c9037d8)
7c903758 eb0d        jmp    ntdll!ExecuteHandler (7c903767)

```

由此可见，这个函数只有两条指令，其中第一条就是准备 EDX 寄存器的。因为内嵌处理器函数的符号没有输出，所以这里显示的是最靠近的符号 ExecuteHandler2。当_raw_seh_handler 函数被调用时，使用 WinDBG 的!exchain 命令列出当前线程的 FS:[0]链表，可以看到最上面便是内嵌异常处理器（见清单 24-8）。

清单 24-8 观察包含内嵌处理器的 FS:[0]链条

```

0:000> !exchain
0012fbcc: ntdll!ExecuteHandler2+3a (7c9037d8)           //内嵌异常处理器
0012ff7c: SehRaw!_raw_seh_handler+0 (00401000)         //手工登记的异常处理器
0012ffb0: SehRaw!_except_handler3+0 (00402784)         //编译器入口函数登记的
    CRT scope 0, filter: SehRaw!mainCRTStartup+c0 (00401171) //过滤块地址
                func: SehRaw!mainCRTStartup+d4 (00401185)     //处理块地址
0012ffe0: kernel32!_except_handler3+0 (7c839a30)         //系统启动函数登记的
    CRT scope 0, filter: kernel32!BaseProcessStart+29 (7c8437b2) //过滤块地址
                func: kernel32!BaseProcessStart+3a (7c8437c8)     //处理块地址

```

因为内嵌的异常处理器是在 ExecuteHandler2 函数中动态注册的，并在用户的异常处理函数返回后立刻注销的，所以它并不会影响到 RtlDispatchException 函数遍历 FS:[0]链条。换句话说，当 RtlDispatchException 函数开始遍历 FS:[0]链条时，我们在 main 函数中注册的异常处理器是位于表头的。

24.5 __try{}__except()结构

上一节我们介绍了自己是如何编写代码登记异常处理器的。介绍这种方法是为了说明登记和注销 SEH 处理器的基本原理。如果将其应用在软件开发中，存在以下两个明显的不足：第一，需要编写符合 SehHandler 函数原型的处理函数，要理解_CONTEXT 等较为复杂的数据结构和概念。第二，因为需要直接操作栈指针，因此将清单 24-5 中的汇编代码简单地封装到普通的 C/C++ 函数中是不行的。这就需要在每个需要保护的程序块前后插入两段嵌入式汇编代码，这会影响程序的简洁性。以上不足为编译器提供了用武之地，VC 编译器的__try{}__except()结构便是一种更易理解和运用的解决方案。

24.5.1 与手工方法的对应关系

概括来说，__try{}__except()结构就是将手工方法中的函数和嵌入式汇编代码简化成高级语言中的标记符和表达式。具体来说，就是将被保护块使用__try关键字和大括号包围起来；使用__except 块将原本书写在 SehHandler 函数中的分支判断结构分解成过滤表达式（if 中的异常情况）和与之对应的异常处理块。图 24-2 显示了__try{}__except()结构与手工方法的对应关系。

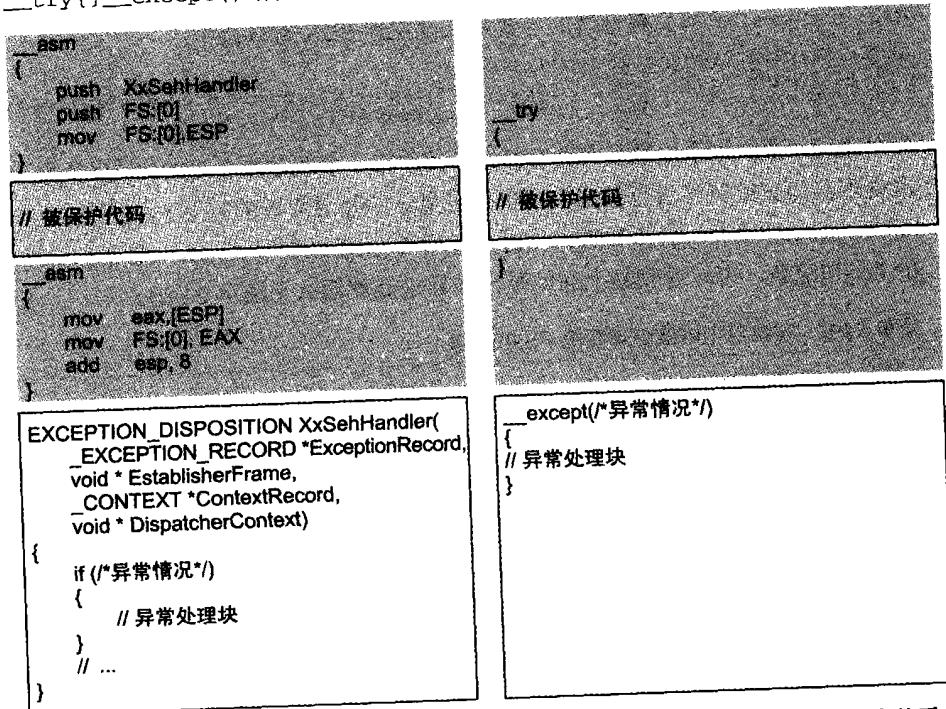


图 24-2 __try{}__except()结构（右）与手工登记和注销 SEH 处理器（左）的对应关系

观察左右两侧的代码，明显右侧的更加简洁，而且更具结构化。不过，我们应该

知道这种简洁性只是在源代码一级的，在汇编语言一级，左右两侧的代码是大同小异的。也就是说，编译器帮助我们隐藏了繁琐的细节。下面我们就看一下编译器是如何编译__try{}__except()结构的。

24.5.2 __try{}__except()结构的编译

清单 24-9 给出了一个使用 C/C++ 语言编写的 TrySeh 函数（位于 code\chap24\SehComp 项目中），在该函数中，我们使用了__try{}__except() 结构来保护可能发生异常的除法操作（ $n=1/n$ ）。

清单 24-9 TrySeh 函数的源代码

```

1 int TrySeh(int n)
2 {
3     __try
4     {
5         n=1/n;
6     }
7     __except(EXCEPTION_EXECUTE_HANDLER)
8     {
9         n=0x122;
10    }
11    return n;
12 }
```

清单 24-10 列出了对 VC6 编译器编译出的 TrySeh 函数（发布版本）及进行反汇编得到的汇编代码。

清单 24-10 TrySeh 函数（发布版本）的反汇编代码

```

1 00401000 55          push  ebp
2 00401001 8bec        mov   ebp,esp
3 00401003 6aff        push  offset SehComp!KERNEL32...+0x30 (004070e0)
4 00401005 68e0704000  push  offset SehComp!_except_handler3 (00401248)
5 0040100a 6848124000  push  offset SehComp!_except_handler3 (00401248)
6 0040100f 64a100000000 mov   eax,dword ptr fs:[00000000h]
7 00401015 50          push  eax
8 00401016 64892500000000 mov   dword ptr fs:[0],esp      ; 安装异常登记结构
9 0040101d 83ec08      sub   esp,8
10 00401020 53         push  ebx
11 00401021 56         push  esi
12 00401022 57         push  edi
13 00401023 8965e8      mov   dword ptr [ebp-18h],esp ; 保存栈指针
14 00401026 c745fc00000000 mov   dword ptr [ebp-4],0       ; 将 trylevel 置为 0
15 0040102d b801000000  mov   eax,1
16 00401032 99         cdq
17 00401033 f77d08      idiv  eax,dword ptr [ebp+8]
18 00401036 894508      mov   dword ptr [ebp+8],eax
19 00401039 eb0e        jmp   SehComp!TrySeh+0x49 (00401049) ; 正常结束
20 0040103b b801000000  mov   eax,1                   ; 过滤表达式
21 00401040 c3          ret
22 00401041 8b65e8      mov   esp,dword ptr [ebp-18h]      ; 异常处理块
23 00401044 b822010000  mov   eax,122h
24 00401049 c745fcfffffff mov   dword ptr [ebp-4],0FFFFFFFFFFh ; 函数的出口
25 00401050 8b4df0      mov   ecx,dword ptr [ebp-10h]
```

```

26 00401053 64890d00000000 mov dword ptr fs:[0],ecx
27 0040105a 5f pop edi
28 0040105b 5e pop esi
29 0040105c 5b pop ebx
30 0040105d 8be5 mov esp,ebp
31 0040105f 5d pop ebp
32 00401060 c3 ret

```

从上面的清单中，我们可以清楚地看到第 3~8 行是在登记异常处理器，与清单 24-5 中的汇编指令非常类似，但是有以下两点不同。

第一，使用 `_except_handler3` 函数作为异常处理函数。我们在上一节观察 FS:[0] 链条时（清单 24-8）也看到了这个函数。事实上，编译器编译 `__try{}__except()` 结构时总是使用统一的函数将其登记为异常处理函数，并不是为每段使用 SEH 的代码生成单独处理函数。这样做有利于代码复用和减小目标文件的大小，不同版本的编译器使用的异常处理函数可能不同。VC6 编译器使用的是 `_except_handler3` 函数，VC2005 使用的是 `_except_handler4`，因为它们的工作原理是类似的，所以我们仍以 `_except_handler3` 为例进行讨论。那么，使用统一的异常处理函数如何能够满足不同 SEH 异常代码块的需要呢？答案是通过传递给 `_except_handler3` 函数的参数来区分不同的情况。但是，我们知道，异常处理函数是由系统的异常分发函数来调用的，即 `RtlDispatchException>ExecuteHandler>ExecuteHandler2>_except_handler3`，而且这些函数传递给异常处理函数的参数个数是固定的。这意味着要增加新的参数是不可行的，解决的办法只有扩展现有的参数，通过类型转换将简单的类型转变为包含扩展字段的复杂类型，这正是 VC 所采用的方案，也就是下面要介绍的第二点差异。

第二，在栈上准备 `EXCEPTION_REGISTRATION_RECORD` 结构（第 5~7 行）前，编译器产生的代码会先压入一个被称为 `trylevel` 的整数（第 3 行）和一个指向 `scopetable_entry` 结构的 `scopetable` 指针（第 4 行），这样在栈上实际就形成了一个如下所示的 `_EXCEPTION_REGISTRATION` 结构。

```

struct _EXCEPTION_REGISTRATION{
    struct _EXCEPTION_REGISTRATION *prev;      //上一个结构的地址
    void (*handler)(PEXCEPTION_RECORD, PEXCEPTION_REGISTRATION,
                    PCONTEXT, PEXCEPTION_RECORD); //处理函数
    struct scopetable_entry *scopetable;        //范围表的起始地址
    int trylevel;                                //这个结构所对应__try块的编号
    int _ebp;                                     //栈帧的基地址
};

```

其中，前两个字段是操作系统规定的标准登记结构（`EXCEPTION_REGISTRATION_RECORD`），后三个字段是编译器扩展的。事实上，`_except_handler3` 函数正是依靠这几个扩展字段来寻找过滤表达式和异常处理块的。一些资料将以上结构称为 `MSVCRT_EXCEPTION_FRAME` 或类似的名字，其中 `FRAME` 的含义是这个结构在栈上动态构建的，是所在函数栈帧的一部分。

我们稍后再讨论 `scopetable` 和 `trylevel` 的含义，现在先使用 WinDBG 跟踪执行一次

TrySeh 函数，以加深大家的理解。当 CPU 执行到第 8 行时，靠近栈顶的数据为：

```
0:000> dd esp
0012ff3c 0012ff70 00401248 004070e0 ffffffff
0012ff4c 0012ff80 004010a9 00000001 00408048
```

此时，EBP 和 ESP 的值为：

```
0:000> r ebp, esp
ebp=0012ff4c esp=0012ff3c
```

其中，从 0012ff3c(栈顶)到 0012ff4c 的 20 个字节便是动态形成的_EXCEPTION_REGISTRATION 结构，此时，ESP 的值也正指向这个结构。具体讲，0012ff70 为 prev 字段（前一个异常处理器的注册记录），00401248 为_except_handler3 函数的地址，004070e0 是 scopetable 字段，fffffff(-1) 为 trylevel，0012ff80 为 ebp 的值，即当前函数的栈帧基址。

因为是在建立_EXCEPTION_REGISTRATION 结构之前将栈顶值放到 EBP 寄存器中（第 1 行）的，所以也可以使用 EBP 减去一个偏移来索引_EXCEPTION_REGISTRATION 结构的字段，EBP-4 为 trylevel 字段，EBP-8 为 scopetable 字段，EBP-0xC 为 handler 字段，EBP-0x10 为 prev 字段。例如，第 14 行和第 24 行就是使用 EBP-4 来引用 trylevel 字段的，第 25 行是使用 EBP-10 来索引 prev 字段的。从这个角度来看，_EXCEPTION_REGISTRATION 结构相当于定义在函数中的局部变量。第 25、26 行是注销结构化异常处理器的动作，其他各行的含义我们将在介绍_except_handler3 的执行过程时介绍。

24.5.3 范围表

为了描述应用程序代码中的__try{}__except 结构，编译器在编译每个使用此结构的函数时会为其建立一个数组，并存储在模块文件的数据区中，通常被称为异常处理范围表。数组的每个元素是一个 scopetable_entry 结构，用来描述一个__try{}__except 结构。

```
struct scopetable_entry
{
    DWORD     previousTryLevel;           //上一个__try{}结构的编号
    FARPROC   lpfnFilter;                //过滤表达式的起始地址
    FARPROC   lpfnHandler;               //异常处理块的起始地址
};
```

其中 lpfnFilter 和 lpfnHandler 分别用来描述__try{}__except 结构的过滤表达式和异常处理块的起始地址。以前面介绍的 TrySeh 函数为例，根据清单 24-10 中的第 4 行，它的异常处理范围表的地址为 004070e0，是显示这个地址开始的三个 DWORD：

```
0:000> dd 004070e0 13
004070e0 ffffffff 0040103b 00401041
```

其中 0040103b 是过滤表达式所对应代码块的地址，观察清单 24-10 中的汇编代码，这个地址对应的是第 20 行，对应的指令为 `mov eax,1`，下一条指令便是函数返回指令，这与过滤表达式中的 `EXCEPTION_EXECUTE_HANDLER(1)` 正好对应。00401041 为处理块的地址，即清单 24-10 中的第 22 行。

24.5.4 TryLevel

编译器是以函数为单位来登记异常处理器的，在函数的入口处进行登记，在出口处进行注销。那么，如何确定导致异常的代码是否在保护块中呢？如果有多个保护块，又如何判断属于哪个保护块呢？答案是对每个 `_try` 结构进行编号，然后使用一个局部变量来记录当前处在哪个 `_try` 结构中，这个局部变量被称为 `trylevel`，也就是在线上所形成的 `_EXCEPTION_REGISTRATION` 结构的 `trylevel` 字段。

编号是从 0 开始的，常量 `TRYLEVEL_NONE (-1)` 作为特殊值代表不在任何 `_try` 结构中。也就是说，`trylevel` 变量被初始化为 -1，然后执行到 `_try` 结构中时便将它的编号赋给 `TryLevel` 变量。以清单 24-10 为例，第 3 行将 `trylevel` 初始化为 -1，第 14 行将其设置为 0，因为进入了 0 号保护块，第 24 行将其又恢复为 -1，因为已经离开了保护块。

清单 24-11 所示的 `TryLevel` 函数显示了更复杂的情况，该函数中共有 4 个 `_try{}__except` 结构，顶层 2 个，内层 2 个，代码中的注释代表了它们的编号，我们将它们分别简称为 Try0~Try3。

清单 24-11 演示 `TryLevel` 变量用法的 `TryLevel` 函数

```

1 int TryLevel(int n)
2 {
3     __try{                      // Try0
4         n=n-1;
5         __try{                  // Try1
6             n=1/n;
7         }
8         __except(EXCEPTION_CONTINUE_SEARCH) {
9             n=0x221;
10        }
11        n++;
12        __try{                  // Try2
13            n=1/n;
14        }
15        __except(EXCEPTION_CONTINUE_SEARCH) {
16            n=0x222;
17        }
18    }
19    __except(EXCEPTION_EXECUTE_HANDLER) {
20        n=0x220;
21    }
22    __try{                      // Try3
23        n=1/n;
24    }
25    __except(EXCEPTION_EXECUTE_HANDLER) {

```

```

26         n=0x223;
27     }
28     return n;
29 }
```

观察 TryLevel 函数入口处的反汇编：

```

004010D0  push  ebp          // 保存调用函数的栈帧指针
004010D1  mov   ebp,esp      // 建立本函数的栈帧
004010D3  push  OFFh         // 压入-1, 初始化 trylevel
004010D5  push  offset string "B\n"+0FFFFFC4h (00422030) // 压入 scopetable
004010DA  push  offset __except_handler3 (00401464)       // 压入处理函数地址
```

第 4 行压入的便是异常处理范围表（scopetable）的地址 0x00422030，使用内存观察窗口观察该地址可以看到表中每个元素的值（见图 24-3）。

Address:	0x00422030	[X]
00422030	FFFFFFFFFF 00401187 0040118D	
0042203C	00000000 00401131 00401134	
00422040	00000000 0040116A 0040116D	
00422044	FFFFFFFFFF 0040118A 004011C8	
00422048	00000000 00000000 00000000	

图 24-3 TryLevel 函数的异常处理范围表（scopetable）

scopetable_entry 结构的大小是 12 个字节，即 3 个 DWORD，因此，图 24-3 中每行对应一个 scopetable_entry 结构，也就是 TryLevel 函数的一个__try 结构。因为一共有 4 个 Try 结构，所以范围表共有 4 个节点，即图 24-3 中的前 4 行，每行对应一个，依次为 Try0~Try3。纵向来看，第 1 列是每个结构的地址，第 2 列是 previousTryLevel 字段，即当前函数范围内的上一层次__try 结构在范围表（scopetable）中的序号，对于最外层的__try 结构，此字段为-1。比如 Try0 的 previousTryLevel 为-1，表示它没有上一层次的__try 结构。第 3 列是 lpfnFilter 字段，即过滤表达式的代码地址，第 4 列是 lpfnHandler 字段，即异常处理块的代码地址。

以 Try1 块为例，previousTryLevel 的值为 0，即这个 Try 块的上一级 Try 块的序号为 0 号，这意味着范围表（数组）的 0 号节点（即图 24-3 的第 1 行）是上一级__try 结构的 scopetable_entry。lpfnFilter 等于 00401131，根据清单 24-12 所示的反汇编结果，这正是过滤表达式所对应的代码（第 12~14 行），将 EAX 与自己异或，也就是将 EAX 设置为 0 (EXCEPTION_CONTINUE_SEARCH)，然后返回。lpfnHandler 的值为 00401134，这正是异常处理块所对应的代码（第 16~18 行）。注意观察第 18~21 行之间没有任何跳转语句，这是因为一旦系统开始执行异常处理块，那么异常处理块执行完毕后便继续执行其下面的代码，不需要做任何返回或转移。

清单 24-12 Try1 块的汇编代码

```

1   5:    __try{ // Try1
2   00401115  mov   dword ptr [ebp-4],1
3   6:           n=1/n;
```

```

4 0040111C mov eax,1
5 00401121 cdq
6 00401122 idiv eax,dword ptr [ebp+8]
7 00401125 mov dword ptr [ebp+8],eax
8 7: }
9 00401128 mov dword ptr [ebp-4],0
10 0040112F jmp $L31026+11h (00401145)
11 8: __except(EXCEPTION_CONTINUE_SEARCH){
12 00401131 xor eax,eax
13 $L31027:
14 00401133 ret
15 $L31026:
16 00401134 mov esp,dword ptr [ebp-18h]
17 9: n=0x21;
18 00401137 mov dword ptr [ebp+8],21h
19 10: }
20 0040113E mov dword ptr [ebp-4],0
21 11: n++;

```

第2行是将trylevel变量设置为1，标志着进入了1号__try结构的区域，如果接下来有异常发生，那么__except_handler3函数便根据trylevel字段的值在范围表中寻找scopetable_entry结构，找到后调用结构中lpfnFilter字段所指定的过滤表达式。第20行将trylevel设为0，因为这时已出了Try1块的范围，但还在Try0块的范围内。

24.5.5 __try{}__except()结构的执行

当位于__try{}结构保护块中的代码发生异常时，异常分发函数便会调用__except_handler3这样的处理函数。__except_handler3函数所执行的操作主要有。

第一，将第二个参数pRegistrationRecord从系统默认的EXCEPTION_REGISTRATION_RECORD结构强制转化为包含扩展字段的_EXCEPTION_REGISTRATION结构。

第二，先从pRegistrationRecord结构中取出trylevel字段的值并且赋给一个局部变量nTryLevel，然后根据nTryLevel的值从scopetable字段所指定的数组中找到一个scopetable_entry结构。

第三，从scopetable_entry结构中取出lpfnFilter字段，如果不为空，则调用这个函数，即评估过滤表达式，如果为空，则跳到第五步。

第四，如果lpfnFilter函数返回值不等于EXCEPTION_CONTINUE_SEARCH，则准备执行lpfnHandler字段所指定的函数，并且不再返回。如果过滤表达式返回的是EXCEPTION_CONTINUE_SEARCH，则自然进入(Fall Through)到第五步。

第五，判断scopetable_entry结构的previousTryLevel字段取值，如果它不等于-1，则将previousTryLevel赋给nTryLevel并返回到第二步继续循环。如果previousTryLevel等于-1，那么继续到第六步。

第六, 返回 DISPOSITION_CONTINUE_SEARCH, 让系统(RtlDispatchException)继续寻找其他异常处理器。

以上过程省略了较为复杂的全局展开和局部展开过程, 我们将在下一节专门讨论。Matt Pietrek 在他 1997 年发表的文章(参考文献 1)中给出了_except_handler3 函数的伪代码, 感兴趣的读者可以在网上找到并阅读。

24.5.6 _SEH_prolog 和_SEH_epilog

在清单 24-10 中, 建立栈帧和_EXCEPTION_REGISTRATION 结构的操作(第 1~8 行)就被插在使用__try{}__except() 结构的函数中。这样做的一个缺点是, 如果程序普遍使用了__try{}__except() 结构, 那么便会导致注册异常处理器的代码重复出现在所有函数中。解决这个问题的方法是将这段代码独立出来, 形成一个特殊的函数供包含__try{}__except() 结构的函数调用。之所以说特殊函数, 是因为这个函数不遵守栈平衡原则, _SEH_prolog 便是一个这样的函数(见清单 24-13)。

清单 24-13 _SEH_prolog

```

1  kernel32!_SEH_prolog:
2  7c8024c6 68009a837c      push   offset kernel32!_except_handler3 (7c839a00)
3  7c8024cb 64a100000000      mov    eax,dword ptr fs:[00000000h]
4  7c8024d1 50                push   eax
5  7c8024d2 8b442410      mov    eax,dword ptr [esp+10h]
6  7c8024d6 896c2410      mov    dword ptr [esp+10h],ebp
7  7c8024da 8d6c2410      lea    ebp,[esp+10h]
8  7c8024de 2be0                sub    esp,eax
9  7c8024e0 53                push   ebx
10 7c8024e1 56               push   esi
11 7c8024e2 57               push   edi
12 7c8024e3 8b45f8      mov    eax,dword ptr [ebp-8]
13 7c8024e6 8965e8      mov    dword ptr [ebp-18h],esp
14 7c8024e9 50                push   eax
15 7c8024ea 8b45fc      mov    eax,dword ptr [ebp-4]
16 7c8024ed c745fcfffffff  mov    dword ptr [ebp-4],0FFFFFFFFFh
17 7c8024f4 8945f8      mov    dword ptr [ebp-8],eax
18 7c8024f7 8d45f0      lea    eax,[ebp-10h]
19 7c8024fa 64a300000000      mov    dword ptr fs:[00000000h],eax
20 7c802500 c3                ret

```

其中, 第 2 行是压入_except_handler3 函数的地址, 第 3 行是压入 prev 字段, 第 18~19 行是将建立好的_EXCEPTION_REGISTRATION 结构的地址登记到 FS:[0] 链表。

以下是使用_SEH_prolog 的一个典型例子。

```

0:000> u kernel32!CreateProcessInternalW
kernel32!CreateProcessInternalW:
7c819704 68080a0000      push   0A08h
7c819709 68d899817c      push   offset kernel32!`string'+0xc (7c8199d8)
7c81970e e8b38dffef      call   kernel32!_SEH_prolog (7c8024c6)
...

```

与_SEH_prolog 配套的是_SEH_epilog(见清单 24-14)。

清单 24-14 _SEH_epilog

```

1  kernel32!_SEH_epilog:
2  7c802501 8b4df0      mov     ecx,dword ptr [ebp-10h]
3  7c802504 64890d00000000 mov     dword ptr fs:[0],ecx
4  7c80250b 59          pop    ecx
5  7c80250c 5f          pop    edi
6  7c80250d 5e          pop    esi
7  7c80250e 5b          pop    ebx
8  7c80250f c9          leave
9  7c802510 51          push   ecx
10 7c802511 c3         ret

```

显而易见，_SEH_prolog 执行的是_SEH_prolog 的逆向动作，用来恢复栈的平衡和注销_SEH_prolog 登记的异常处理器。

24.6 安全问题

前面各节我们介绍了 2 种处理异常的方法：一是编写结构化异常处理函数（SehHandler）并手工注册到 FS:[0]链条中。二是使用 VC 编译器的__try{}__except() 结构。这 2 种方法有一个共同点，那就是都需要在栈上动态建立一个登记结构并将这个结构注册到 FS:[0]链条中，因为这个登记结构是位于所在函数栈帧中的，所以这 3 种异常处理方法经常被通称为基于帧的异常处理（Frame Based Exception Handling）。

基于帧的异常处理的一个不足是，如果栈上发生缓冲区溢出，那么登记在栈上的信息可能被破坏，本来的异常处理函数地址可能意外指向新的地方，当再有异常发生时就可能执行未知的代码。这种不足已经被黑客和恶意软件用来作为实施攻击的一种途径。

24.6.1 安全 Cookie

为了应对上面提到的安全隐患，VC2005 在编译使用 try{}catch 结构的函数时生成的异常处理函数加入了一系列检查安全 Cookie 的指令，不再像以前那样只有两条指令。如果安全 Cookie 完好，那么会跳转到统一的异常处理函数_CxxFrameHandler3，如果安全 Cookie 已经被破坏，那么便报告 GS 失败（_report_gsfailure）终止程序运行。清单 24-15 显示了 VC2005 编译器为一个名为 TestTryOver2Bytes 的函数（位于 code\chap24\Vc8Win32 项目中）产生的异常处理函数。

清单 24-15 VC2005 产生的异常处理函数

```

text$ SEGMENT           //代码段声明
__ehandler$?TestTryOver2Bytes@0YAHPAUHWND__0@0@Z: //动态产生的异常处理函数
    mov     edx, DWORD PTR [esp+8]      //将参数 2 (异常登记结构) 赋给 EDX
    lea     eax, DWORD PTR [edx+12]      //通过登记结构的地址找到栈帧基址 EBP
    mov     ecx, DWORD PTR [edx-60]      //将保存在变量区顶部的 Cookie 赋给 ECX
    xor     ecx, eax                   //与保存的 EBP 值异或以还原 Cookie 种子

```

```

call    @__security_check_cookie@4      //检查完好性
mov     ecx, DWORD PTR [edx-8]        //将保存在变量区底部的Cookie赋给ECX
xor     ecx, eax                     //与EBP异或
call    @__security_check_cookie@4      //检查完好性
mov     eax, OFFSET __ehfuncinfo$?TestTryOver2Bytes@@YAHPAUHWND__@0H@Z
jmp     __CxxFrameHandler3           //转到统一的异常处理函数
text$X ENDS

```

为了便于理解以上代码，图 24-4 画出了 TestTryOver2Bytes 函数的栈帧，描述的时刻是在栈帧完全建立后用户代码执行前。从图中我们可以看到局部变量区域的底部和顶部分别放置了一个安全 Cookie，它们的值是一样的，都是通过全局的 Cookie 变量（种子）与 EBP 寄存器的值异或而得到的。底部的 Cookie（地址 0x12FD7C）位于异常登记结构的上面，局部变量发生溢出后会先破坏这个 Cookie，因此检查这个 Cookie 的完好性可以比较有效的探测到是否曾经发生过缓冲区溢出。

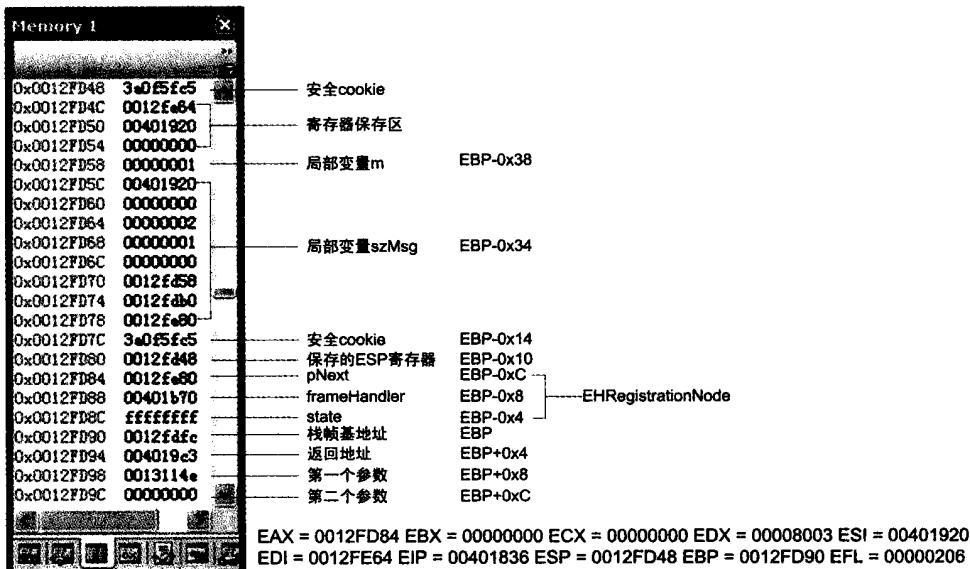


图 24-4 包含异常处理登记结构的栈帧

在 TestTryOver2Bytes 函数中我们故意向 32 字节长的 szMsg 数组中复制了 34 个字节的内容，制造了一个轻微的缓冲区溢出。而后，我们又做了一个除零操作，以引发异常。当异常发生后，系统的异常分发函数会通过 FS:[0] 链条调用清单 24-15 中的 __ehandler\$?TestTryOver2Bytes 函数。因为系统默认所有注册在 FS:[0] 链条中的异常处理函数都应该具有标准的 SehHandler 原型，因此系统会以如下形式调用时这个函数：

```

__ehandler$?TestTryOver2Bytes ( _EXCEPTION_RECORD *ExceptionRecord,
                               void * EstablisherFrame, _CONTEXT *ContextRecord, void * DispatcherContext);

```

其中第二个参数就是栈上的登记结构 EHRegistrationNode 的地址，即图 24-4 中的

地址 0x12FD84，从清单 24-15 看到，`_ehandler` 先是将这个地址放到 `EDX` 中，然后根据这个地址向高地址方向偏移 12 个字节 (`edx+12`) 找到栈帧地址 `EBP`，向低地址方向偏移 60 个字节 (`edx-60`) 找到保存在变量区顶部的 `Cookie`，而后将二者异或还原出全局 `Cookie` 值（种子），而后调用 `_security_check_cookie` 函数检查还原出的 `Cookie` 种子是否与全局变量中记录的一样。类似的，`_ehandler` 还会检查位于变量区底端的 `Cookie`。因为我们故意溢出了两个字节所以这个 `Cookie` 值由原来的 0x3a0f5fc5 变成了 0x3a0f0000（参见图 24-5）。

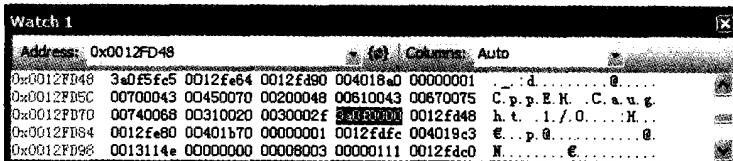


图 24-5 安全 `Cookie` 被破坏后的栈现场

`Cookie` 值的变化被 `_security_check_cookie` 函数发现后，它会调用 `_report_gsfailure()` 函数报告检查失败，如果是在 VC2005 的集成环境中执行，那么会弹出 CRT 报告对话框，选中断（Break）后，可以看到其执行过程如下：

```
VC8Win32.exe!__crt_debugger_hook() Unknown
> VC8Win32.exe!__report_gsfailure() Line 298 + 0x7 bytes C
  VC8Win32.exe!__CxxFrameHandler3() + 0x48 bytes C++
  ntdll.dll!ExecuteHandler@20() + 0x24 bytes
  ntdll.dll!_KiUserExceptionDispatcher@8() + 0xe bytes
  VC8Win32.exe!TestTryOver2Bytes(HWND__ * hWnd=0x00070f48, int n=0)
```

如果是在 WinDBG 中，那么程序会立刻结束。如果不是在调试器中执行，那么会触发应用程序错误对话框。总之对于轻微的缓冲区溢出，安全 `Cookie` 检查可以检测到溢出而停止继续执行异常处理函数，这样尽管程序也不能继续工作了，但是可以避免执行未知的代码导致更严重的后果。

对于比较严重的溢出，如果登记结构中的处理函数被覆盖了，那么放在 `_ehandler` 中的检查函数就得不到执行了。将刚才的 `TestTryOver2Bytes` 函数复制一份并略作修改，将溢出两个字节改为溢出 16 个字节，此时整个 `EHRegistrationNode` 结构都被覆盖了（图 24-6），本来的处理函数地址已经被替换成其他内容（002e0037），当异常发生时系统会执行新地址处的代码。因为这个地址没有有效指令的 002e0037（字符 s 和 . 的代码）所以当我们在调试器中执行时程序会因为访问违例而中断到调试器中，其执行过程如下：

```
> 002e0037()
  ntdll.dll!ExecuteHandler2@20() + 0x26 bytes
  ntdll.dll!ExecuteHandler@20() + 0x24 bytes
  ntdll.dll!_KiUserExceptionDispatcher@8() + 0xe bytes
```

可见，基于 `Cookie` 技术的检查机制不能完全防止程序意外执行通过缓冲区溢出而登记的异常处理函数。

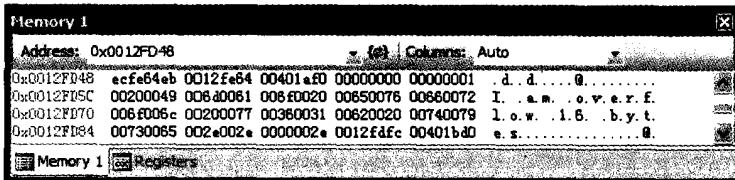


图 24-6 缓冲区溢出覆盖了整个 EHRegistrationNode 结构

24.6.2 SAFESEH

防止利用异常处理机制进行安全攻击的一种更有效方法是 SAFESEH (Secure Exception Handling) 技术。其基本思想是将模块中合法的异常处理函数登记在一个专用的被称为 SAFESEH 的表中，当有异常发生时，异常分发函数会根据异常处理函数的地址到它所对应的模块中查询这个函数是否在 SAFESEH 表中，如果在那么便说明这是个安全的异常处理函数可以执行它，如果不在那么就不去执行它。要启用 SAFESEH 技术，同时需要操作系统和编译器的支持。

- 需要使用 Visual C++ .NET 2003 或更高版本的编译器，在链接选项中加入 /SAFESEH 来链接准备使用 SAFESEH 的模块。链接器检测到此选项后，会将模块中的可执行模块中的异常处理函数登记在一个专用的 SAFESEH 表中，并自动创建以下两个变量（符号）用来描述这个表：

```
PVOID __safe_se_handler_table[];      // 异常处理器描述表的基地址
BYTE  __safe_se_handler_count;        // 异常描述表中所包含的表项数
```

- 需要操作系统支持 SAFESEH 技术。Windows 操作系统从 Windows Server 2003 和 Windows XP SP2 开始支持这一技术，主要体现在如下两个方面。

第一，操作系统的模块加载器 (Program Loader) 在加载一个可执行模块时，会检查这个模块是否包含 SAFESEH 表，如果有会对其进行解密，然后将异常处理函数存入全局变量 __safe_se_handler_table 所指向的地址，并将表项个数放入到 __safe_se_handler_count 变量中。

第二，RtlDispatchException 函数在遍历 FS:[0] 链条分发异常时，会调用 RtlIsValidHandler 函数对异常处理器的合法性进行检查，如果不合法 (RtlIsValidHandler 返回 FALSE)，那么 RtlDispatchException 函数不会执行这个处理器，而且会停止继续分发异常，返回 FALSE，使这个异常成为无人处理的异常。

需要说明的是，对于支持 SAFESEH 的 Windows 版本，RtlDispatchException 总是调用 RtlIsValidHandler，不管当前的模块是否是使用 /SAFESEH 选项链接的。RtlIsValidHandler 函数内部会考虑兼容没有使用 SAFESEH 的模块。其大致过程是调用 RtlLookupFunctionTable 函数，后者再调用 RtlCaptureImageException-Values 函数。

```
int RtlCaptureImageExceptionValues([IN] PVOID pHandler,
[OUT] PVOID pSafeSeHandlerTable)
```

RtlCaptureImageExceptionValues 函数会根据参数中指定的处理函数地址找到它所属的模块，然后再定位到这个模块的 _IMAGE_NT_HEADERS 和 _IMAGE_OPTIONAL_HEADER 结构，查找是否存在 SAFESEH 表，如果存在则将其地址存入 pSafeSeHandlerTable 参数，并返回表中所包含的表项个数。如果异常处理函数的地址所对应的模块中不包含 SAFESEH 表，那么 pSafeSeHandlerTable 将被设置为空，RtlIsValidHandler 将返回 TRUE 表示检查通过。

观察启用了 SAFESEH 支持的模块，比如 VC8Win32 程序，可以看到 __safe_se_handler_table 符号，即 SAFESEH 表的起始地址：

```
0:000> x vc8win32!*safe*
004023a0 VC8Win32!__safe_se_handler_table = void *[]
```

使用内存观察命令可以观察 SAFESEH 表的内容：

```
0:000> dd 004023a0
004023a0 00001675 00001d80 00001da5 00001dca
004023b0 00000000 00000000 00000000 00000000
```

第 1 行的 4 个 DWORD 值代表 4 个合法的异常处理器函数相对于模块基址的偏移。因为当前模块的基地址是 0x400000，所以第四个处理器函数的地址是 0x401dca，观察 TestTry 函数的序言代码，可以知道这就是 TestTry 函数的异常处理函数，对这个表项设置一个数据访问断点：

```
ba r1 0040235c
```

然后通过菜单中的 Test Try 命令触发执行 TestTry 函数，调试器收到除零异常通知后继续执行，数据访问断点就命中了，清单 24-16 显示了 RtlIsValidHandler 函数被调用的过程。

清单 24-16 验证异常处理函数的合法性

```
ChildEBP RetAddr Args to Child
0012f9d4 7c93783a 00401d9a 0012fe64 0012fa68 ntdll!RtlIsValidHandler+0x52
0012fa50 7c90eafa 00000000 0012fa7c 0012fa68 ntdll!RtlDispatchException+0x7e
0012fa50 004019e1 00000000 0012fa7c 0012fa68 ntdll!KiUserExceptionDispatcher+0xe
0012fd90 00401b80 00100782 00000000 00008003 VC8Win32!TestTry+0x41
```

进一步观察汇编指令，可以看到果然是 RtlIsValidHandler 函数在遍历 SAFESEH 表，验证参数中所指定的处理函数（00401d9a）是否在表中。因为要验证的函数是编译程序时产生的处理函数，与 SAFESEH 表中记录的吻合，所以 RtlIsValidHandler 会返回真，于是系统会执行这个处理函数来处理异常。

如果我们在收到除零异常时将栈中的异常处理函数地址略作改动，比如将 0x401dca 改为 0x401dcb，那么 RtlIsValidHandler 便会返回 FALSE，于是系统就不会执行位于新地址的处理函数，效果上相当于 TestTry 函数中的 Catch 块不存在。

当如果将 0x401dca 改动的非常大，比如改为 0x2e002e，那么系统还是会执行这个

新的函数地址，这是因为系统认为这地址所对应的模块不包含 SAFESEH 表，因此使用了兼容策略。这告诉我们，如果一个程序中有很多个模块，那么应该都支持 SAFESEH，否则不支持的 SAFESEH 模块便留下了安全漏洞。从 Windows Vista 开始，几乎所有 Windows 系统模块已经都使用了/SAFESEH 选项进行链接。

使用 !address 命令观察 SAFESEH 表所在地址的属性，可以看到它是只读的：

```
0:000> !address VC8Win32!__safe_se_handler_table
00400000 : 00402000 - 00001000
  Type      01000000 MEM_IMAGE
  Protect    00000002 PAGE_READONLY
  State     00001000 MEM_COMMIT
  Usage     RegionUsageImage
  FullPath  VC8Win32.exe
```

这意味着恶意软件是不能轻易篡改 SAFESEH 表中的内容的。

24.6.3 基于表的异常处理

增强异常处理机制安全性的一种更彻底方法是 x64 系统中的基于表的异常处理 (Table Based Exception Handling)。其基本思想是将异常处理器的描述和登记信息都以表格的形式存储在可执行文件中，当有异常发生时，系统根据异常的发生位置自动在这些表格中寻找匹配的处理函数，不需要在栈上做任何登记，也不再使用 FS:[0] 链条。基于表的异常处理机制与基于帧的异常处理机制是不兼容的。运行在 x64 CPU 的 64 位 Windows 使用了基于表的异常处理，编译运行在这样的目标系统中的 64 位应用程序时，编译器会自动使用新的编译方式产生合适的代码。由于篇幅所限，本书不再详细描述，感兴趣的读者可以阅读参考文献 5 和 6。

24.7 本章总结

本章从编译角度介绍了应用程序中异常处理代码的工作原理和执行过程。第 24.2~24.4 节介绍了系统中用于登记异常处理器的数据结构 (FS:[0] 链条) 和工作函数 (RtlDispatchException 等)。第 24.5 节介绍了用于支持结构化异常处理的 __try{}__except() 结构。最后一节介绍了与异常处理机制有关的安全问题和应对措施。

概括来讲，在 Windows 系统中，try{}catch() 结构与 __try{}__except() 结构都是建立在操作系统的结构化异常处理机制之上的。二者之间既有很多相同之处，也有明显的差异。它们是不可以出现在同一个函数中的，如果遇到这样的情况，编译器会给出如下错误：

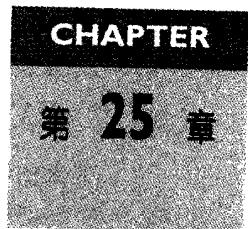
```
error C2713: Only one form of exception handling permitted per function
```

最后要说明的是应该合理地使用异常处理机制，不应该过度使用。因为无论是

基于帧的异常处理机制，还是基于表的异常处理机制，它们都是有明显的开销的。前者主要是时间开销，需要在函数中插入代码动态登记和注销异常处理器，即使不发生异常这些代码也需要执行；后者主要是空间开销，需要在可执行文件中存储很多描述信息。

参考文献

1. Matt Pietrek. A Crash Course on the Depths of Win32™ Structured Exception Handling. Microsoft System Journal, 1997
2. Ken Johnson (Skywing). Programming against the x64 exception handling support. <http://www.nynaeve.net/?p=99>
3. Exceptional Behavior: x64 Structured Exception Handling. The NT Insider, 2006



调试符号

在软件调试中，调试符号（Debug Symbols）是将被调试程序的二进制信息与源程序信息联系起来的桥梁。很多重要的调试功能都需要有调试符号才能工作，比如源代码级调试、栈回溯、按名称显示变量等。因此，正确地理解和使用调试符号是学习软件调试的一门必修课。

从软件编译的角度看，调试符号是编译器在将源文件编译为可执行程序的过程中，为支持调试而摘录的调试信息。这些信息以表格的形式记录在符号表中，是对源程序的概括。调试信息描述的目标主要有变量、类型、函数、标号（Label）和源代码行等。

调试信息是在编译过程中逐步收集和提炼出来的，最后由链接器或专门的工具保存到调试符号文件中。调试符号既可以存储在单独的文件中，也可以与目标代码共享一个文件。Visual Studio 编译器默认是将调试符号保存在单独的文件中，即 PDB 文件。PDB 是 Program Database 的缩写，这个名字非常好地体现了符号文件的性质——用于描述源程序的数据库。微软没有直接公开 PDB 文件的格式和内部细节，但提供了两种方式来访问调试符号文件中的符号，一种是 DbgHelp 函数库，另一种是 DIA SDK（Debug Interface Access Software Development Kit）。

本章将先介绍名称修饰（25.1 节）和存储调试信息的常用格式（25.2 节），然后分别介绍 OBJ 文件（25.3 节）、PE 文件（25.4 节）、DBG 文件（25.5 节）中的调试信息。第 25.6 节介绍 PDB 文件的概况和文件格式。第 25.7 节介绍与调试符号有关的编译器设置。第 25.8 节将深入介绍 PDB 文件中的各种数据表。

25.1 名称修饰

根据我们在第 22 章中的介绍，一个完整的函数声明包括返回值类型、调用协议名称、函数名称、参数信息等若干个部分。为了把函数的所有原型信息记录在单一的字符串中以便于标识和组织函数，VC 编译器使用了一种称为名称修饰（Name Decoration）的技术，其宗旨就是将函数的本来名称、调用协议、返回值等信息按照一定的规则编

排成一个新的名字，称为修饰名称（Decorated Name）。举例来说，以下两行分别是 TestTry 函数的原型和它的修饰名称：

```
int TestTry(HWND hWnd, int n)
?TestTry@@YAHPAUHWND__@@HEZ
```

与装饰前的名称相比，装饰后的名称不再包含空格和括号这些不利于存储的分隔符，将多个部分合并成一个连贯的单一整体，因此名称修饰有时也被称为名称碾平（Name Mangling）。观察编译器产生的汇编文件，可以看到编译器为每个函数所生成的修饰名称。这要先设置编译器的 /FA 选项使其输出汇编文件，对于 VC6，其操作步骤为：Settings > C/C++ > Category 中选择 Listing Files > 将 Listing file type 选为 No Listing 之外的一个选项。对于 VC2005，其操作步骤为：Properties > C/C++ > Output Files > Assembler Output。

使用 DbgHelp 库的 UnDecorateSymbolName 函数可以将一个修饰名翻译成本来的名字。

```
DWORD UnDecorateSymbolName(PCTSTR DecoratedName,
    PTSTR UnDecoratedName, DWORD UndecoratedLength, DWORD Flags);
```

但是该函数不能从修饰名中解析出函数原型中的其他信息，如参数和返回值等。下面我们将逐步介绍 VC 编译器产生修饰名称的方法。

25.1.1 C 和 C++

首先 C 编译器和 C++ 编译器使用的名称修饰规则是不同的，这意味着对 VC 这样的同时支持 C 和 C++ 语言的集成编译器来说，同一个函数可能产生不同的修饰名。举例来说，对于 void __cdecl test(void) 函数，按照 C 规范编译所产生的修饰名称是 _test，而按照 C++ 规范编译产生的修饰名称是 ?test@@ZAXXZ。因为链接器连接目标文件时是使用修饰名称来连接的，所以，如果调用方和实现方所使用的编译规范不同，那么连接时就会出现下面这样的错误：

```
error LNK2001: unresolved external symbol "void __cdecl Test(void)" (?Test@@YAXXZ)
```

VC 编译器默认按照源文件的扩展名来决定源文件的类型和选择编译器，.c 代表 C 文件，.cpp 和 .cxx 代表 C++ 文件，同时也支持使用如下方法来选择编译器（编译规范）：

- 使用编译选项/Tc 后跟文件名，强制此文件为 C 文件。
- 使用编译选项/Tp 后跟文件名，强制此文件为 C++ 文件。
- 使用/TC 选项，强制指定所有要编译的文件都是 C 文件。
- 使用/TP 选项，强制指定所有要编译的文件都是 C++ 文件。
- 在 C++ 文件中，使用 extern "C" 关键字声明使用 C 的名称修饰规则。

区分 C 和 C++ 修饰名的一种简单方法是，C++ 的修饰名称都是以问号 (?) 开始的。下面我们将分别介绍 C 和 C++ 编译器所使用的名称修饰规则。

25.1.2 C 的名称修饰规则

VC 的 C 编译器使用如下规则来产生修饰名称。

第一，对于使用 C 调用协议（`__cdecl`）的函数，在函数名称前加一下画线，不考虑参数和返回值。

第二，对于使用快速调用协议（`__fastcall`）的函数，在函数名称前后各加一@符号，后跟参数的长度，不考虑返回值。例如 `extern "C" int __fastcall Test(int n)` 的修饰名称为`@Test@4`。

第三，对于使用标准调用协议（`__stdcall`）的函数，在函数名称前加一下画线，名称后加一@符号，后跟参数的长度，不考虑返回值。例如 `extern "C" int __stdcall Test(int n, int m)` 的修饰名称为`_Test@8`。

25.1.3 C++ 的名称修饰规则

因为 C++ 编译器要支持类和命名空间等特征，其修饰名称中须考虑类名和命名空间信息，所以它的名称修饰规则要比 C 复杂一些。C++ 标准没有定义统一的名称修饰规则，因此，不同的编译器使用的规则是不一样的，即使是同一种编译器，不同版本间可能也存在差异。以 Visual C++ 编译器为例，一个 C++ 修饰名称依次由以下几部分组成。

1. 问号前缀。
2. 函数名称或不包括类名的方法名称。构造函数和析构函数具有特别的名称，分别是`?0` 和`?1`。运算符重载也具有特别的名称，例如 `new`、`delete`、`=`、`+`和`++`的名称分别为`?2`、`?3`、`?4`、`?H` 和`?E`，我们把这些特别的函数名称简称为特殊函数名。
3. 如果名称不是特殊函数名，那么加一个分隔符@。
4. 如果是类的方法，那么由所属类开始依次加上类名和父类名，每个类名后面跟一个@符号，所有类名加好后，再加一个@符号和字符 Q 或 S（静态方法）。如果不是类的方法，那么直接加上@符号和字符 Y。
5. 调用协议代码。对于不属于任何类的函数，C 调用协议（`__cdecl`）的代码为 A，`__fastcall` 协议的代码为 I，标准调用协议（`__stdcall`）的代码为 G。对于类方法，调用协议前会加一个字符 A，`this` 调用协议的代码为 E。
6. 返回值编码，例如字符 H 代表整数类型的返回值。
7. 参数列表编码，以@符号结束，其细节从略。
8. 后缀 Z。

概括来讲，C++ 的名称修饰有以下几条规律。

第一，都是以?开始，以字符 Z 结束，中间由@符号分割为多个部分。另外整个名

称的长度最长为 2048 个字符。

第二, 对于类的函数, 其基本结构为: ?方法名@类名@@调用协议 返回类型 参数列表 Z。

第三, 对不属于任何类的函数, 其基本结构为: ?函数名@@Y 调用协议 返回类型 参数列表 Z。

以 int __cdecl TestFunc(int, int) 为例, 它的修饰名称为?TestFunc@@YAHHH@Z, 其中@Y 表明这不是类的方法, 其后的 A 代表 C 调用协议, 第一个 H 代表返回值为整数类型, 后两个 HH 分别代表两个整型参数, 之后的@表示参数列表结束, 最后的 Z 是后缀。

以 C++ 的方法 public: int CTest::SetName(char *, ...) 为例, 它的修饰名称为?SetName@CTest@@QAAHPADZZ, 其中, 第一个?是前缀, SetName 是方法名, CTest 是类名, @Q 表示类名结束, 第一个 A 是 C++ 方法的调用协议前缀, 后面的 A 表示 C 调用协议 (因为声明中包含可变数量参数, 所以编译器会自动使用 C 调用协议), 随后的 H 表示返回值类型 (整数), PADZ 是参数编码, 最后的 Z 是后缀。

再举个构造函数的例子, 如 public: CTest::CTest(void), 它的修饰名称为??0CTest@@QAE@XZ, 其中?0 代表这是构造函数, CTest 表示类名, @Q 表示类名结束, 其后的 AE 表示 this 调用协议。再如 public:void CTest::operator delete(void *) 的修饰名称为??3CTest@@SAXPAX@Z, 其中@s 表明重载的 delete 运算符被自动编译为静态方法, 重载的 new 运算符也是这样。

25.2 调试信息的存储格式

因为缺乏统一的标准, 不同编译器存储调试信息的格式很可能是不同的, 本节将介绍几种常见的存储格式。

25.2.1 COFF 格式

COFF 是 Common Object File Format 的缩写, 是一种广为流传的二进制文件格式, 用来存储可执行映像文件、目标文件 (Object File)、库文件 (Library File) 等。COFF 格式历史悠久, 在很多早期的 Unix 系统中就使用广泛。Windows 操作系统中用来存储程序映像文件的 PE (Portable Executable) 格式也是源于 COFF 格式的。

因为 COFF 可以存储各种数据对象, 所以人们很自然地想到用它来存储调试信息。COFF 格式的调试信息可以和目标文件或映像文件保存在一起, 也可以单独存放, 取决于编译器的设置。举例来说, WinDBG 的 lm 命令在显示模块列表时会显示每个模块所使用的符号文件信息, 下面的显示说明 SymOption.exe 模块的调试信息是 COFF 格式的, 而且是与 EXE 文件存储在一起的:

```
start    end      module name
00400000 00415000  SymOption C (coff symbols)  C:\...\_Z7COFF\SymOption.exe
```

使用 rebase 工具可以将 exe 文件中的调试信息剥离到一个单独的.dbg 文件中，例如 rebase -x . -b 0x400000 symoption.exe，执行这一操作后再调试这个 EXE，lm 命令的显示变为：

```
00400000 00415000 symoption (coff symbols) C:\...\_Z7COFF\symoption.dbg
```

这说明，WinDBG 在使用存放在.dbg 文件中的 COFF 格式调试信息。

微软的 PE 规范（参考文献 6）定义了如何在 PE 文件中使用 COFF 格式来存储调试符号，可以说是对 COFF 格式的一种扩展，我们将在下一节对其作进一步介绍。

25.2.2 CodeView (CV) 格式

CodeView 是与 MSC 编译器一起使用的调试器（参见 28.3.3 节），它使用的调试符号格式被称为 CodeView 格式，简称 CV 格式。CV 格式是微软自己定义的调试符号存储格式，其详细定义没有完全公开。与 COFF 格式类似，CV 格式的调试信息既可以和目标文件或映像文件保存在一起，也可以单独存放。例如，以下分别是 WinDBG 使用存放在 exe 文件和 dbg 文件中的 CV 符号时的显示：

```
0:000> lm
start end module name
00400000 00415000 SymOption C (codeview symbols) C:\...\_Z7CV\SymOption.exe
0:000> lm
start end module name
00400000 00415000 symoption (codeview symbols) C:\...\_Z7CV\symoption.dbg
```

伴随着编译器和调试器产品的发展，CV 格式的调试符号又分为多种版本，不同版本的 CV 数据块使用的数据排放方法是有差异的。所有 CV 数据块的开始 4 个字节总是一个 32 位的签名，这个签名表示了其后数据所使用的格式版本。表 25-1 中列出了常见的 CV 版本签名和对它的简单介绍。

表 25-1 CodeView 调试符号的格式分类

版本签名	介绍
NB00	不再使用
NB01	不再使用
NB02	Microsoft LINK 5.10 所产生的数据块
NB05	Microsoft LINK 5.20 或更高版本的链接器所产生的压缩前数据块，CVPACK 会将其压缩为其他格式（NB09）
NB07	仅用于 Quick C for Windows 1.0
NB08	供 4.00 到 4.05 版本的 CodeView 调试器使用的数据块
NB09	供 4.10 版本的 CodeView 调试器使用的数据块，Windows NT 4.0 的符号文件使用这种格式
NB10	符号文件链接
NB11	CodeView 5.0 所引入的格式
RSDS	类似于 NB10，但它描述的是 7.0 版本的 PDB 符号文件

其中，NB10 和 RSDS 格式的数据块都是为了描述与当前文件（EXE 或 DLL）匹

配的 PDB 文件，指引调试器到那个文件中去加载调试符号。很多经常调试的朋友都有这样的经验，如果调试同一台机器编译出来的 EXE 程序，即使 EXE 文件的位置变化了，只要 PDB 文件仍在原来的地方，那么调试器还是可以找到合适的调试符号，事实上，调试器正是依靠 EXE 文件中的 NB10 或 RSDS 数据块寻找到符号文件的。二者的差异是，NB10 指引的是 2.0 版本的 PDB，RSDS 指引的是 7.0 版本的 PDB 文件。清单 25-1 给出了这两种格式的 CV 数据块所使用的数据结构。

清单 25-1 描述 NB10 和 RSDS 格式的数据结构

```
struct CV_INFO_PDB20
{
    CV_HEADER CvHeader;           //版本签名，即 NB10
    DWORD Signature;             //时间戳，是距离 1970 年 1 月 1 日的秒数
    DWORD Age;                   //年龄，PDB 文件支持增量修改，每次修改后这个值加 1
    BYTE PdbFileName[];          //以\0结束的 PDB 文件完整路径
};

struct CV_INFO_PDB70
{
    DWORD CvSignature;           //版本签名，即 RSDS
    GUID Signature;              //GUID 格式的文件签名
    DWORD Age;                   //年龄，PDB 文件支持增量修改，每次修改后这个值加 1
    BYTE PdbFileName[];          //以\0结束的 PDB 文件完整路径
};
```

可见这两个数据块的主要差异是因为 7.0 版本的 PDB 文件使用 GUID 格式的文件签名。我们将在介绍 PE 文件中的调试信息时介绍以上数据结构的实际应用。

Borland 公司的编译器和调试器所使用的调试信息格式与 CodeView 格式类似，但做了一些扩展，典型的版本签名有 FB09 和 FB0A。

25.2.3 PDB 格式

Visual C++ 1.0 引入了一种新的存储调试信息的文件格式，称为 Program Database，简称 PDB。PDB 格式的调试信息需要单独存储在一个文件中，通常以.pdb 为后缀。PDB 格式的调试符号文件支持 Edit and Continue (EnC) 等高级调试功能，这是 CV 和 COFF 格式所不支持的。

PDB 格式是微软自己定义的未公开格式，但是可以通过 DbgHelp 库或 DIA SDK 来间接访问 PDB 文件中的调试信息。

根据所包含符号的完整程度，PDB 格式的调试符号文件又分为私有 (Private) PDB 符号文件和公共 (Public) PDB 符号文件。私有 PDB 文件是由编译器和链接器产生的，公共 PDB 文件是在私有 PDB 文件的基础上使用工具产生的，产生时剔除了数据类型、函数原型和源代码行等与源程序文件关系密切的私有信息。公共 PDB 文件的典型用途是公开给客户或合作伙伴使用。举例来说，微软符号服务器中公开的符号文件大多都

是公共符号文件。使用 DDK 中的 binplace 工具可以成批地产生公共 PDB 文件。对于 VC2005 或更高版本的 VC 编译器，可以使用链接选项/PDBSTRIPPED:<公共 PDB 文件名>来产生公共 PDB 文件。

在 WinDBG 中显示模块列表时，如果一个模块使用的是私有 PDB 符号文件，那么关于这个模块的显示信息中会包含 private pdb symbols，如果是公共 PDB 文件，那么显示信息会包含 pdb symbols，例如：

```
00400000 0041d000  SymOption C (private pdb symbols)  C:\...\Z7PDB\SymOption.pdb
7c900000 7c9b0000  ntdll      (pdb symbols)  d:\symbols\ntdll.pdb\...\ntdll.pdb
```

公共调试符号不支持源代码级调试、观察局部变量和按类型显示变量等功能。为了能够观察某些重要的数据结构，Windows 内核和 NTDLL.DLL 的公共符号文件中保留了部分私有符号。

25.2.4 DWARF 格式

DWARF 是一种公开的调试信息格式规范，是由 DWARF 组织 (<http://dwarfstd.org>) 定义的。该格式目前主要应用在 Unix 和 Linux 系统中，如 gcc 和 gdb 都支持 DWARF 格式。DWARF 是 Debugging With Attributed Record Formats 的缩写，是其创始人 Brian Russell 所取的名字。从该组织的网站上可以自由下载描述 DWARF 格式的详细文档。

25.3 目标文件中的调试信息

简单来说，目标文件（Object File）就是编译器用来存放目标代码（Object Code）的文件。目标文件既是编译过程的输出结果，又是链接过程的输入材料。因为从文件角度来讲，编译器的任务就是将源文件中的源代码翻译成目标代码，然后存储在目标文件中；而链接器的任务是将多个目标文件中的目标代码连接成一个可以被操作系统加载和执行的映像文件（Image File）。

调试信息的主要作用是向调试器和调试人员提供源程序中的程序信息，包括类型、函数和各种数据（参数、局部变量、全局变量、静态变量等）。那么，如何把这些信息传递给调试过程呢？因为编译过程对源程序的知识（knowledge）最多，所以让编译过程收集调试信息是最佳的选择，那么把收集到的信息放在哪里呢？很自然的有两种方法，一种是和目标代码一同放在目标文件中；另一种是放在单独的文件中。本节我们将以 VC 编译器的目标文件为例，介绍存放在目标文件中的调试信息。

VC 编译器的目标文件使用的是 COFF 格式，整个文件分成多个数据块，图 25-1 画出了一个典型目标文件（SymOption.obj）的数据布局。

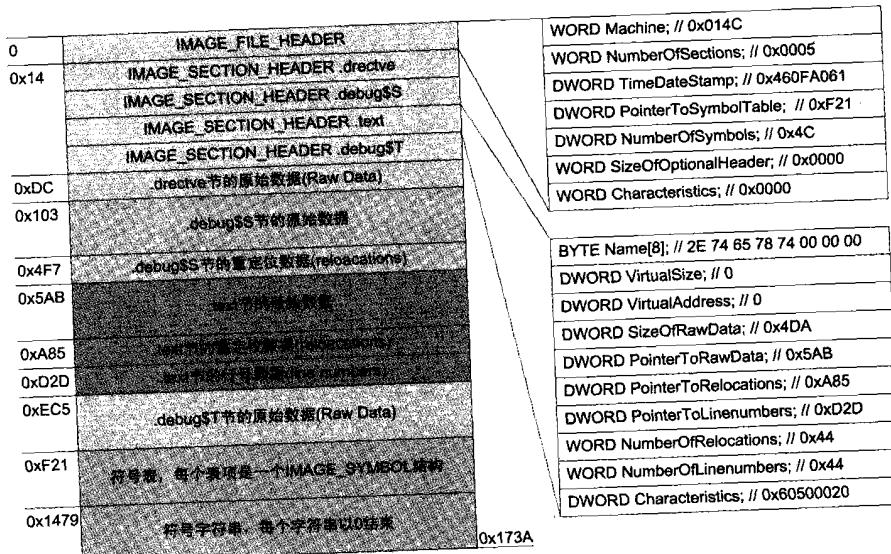


图 25-1 典型目标文件的数据布局

SymOption.obj 文件共有 5946 个字节，图中画出了文件中的所有数据块，最前面是一个 IMAGE_FILE_HEADER 结构，而后是 5 个 IMAGE_SECTION_HEADER 结构，分别描述了这个文件中的 5 个节的概要信息，分别为：

- .directive，其数据为一系列以空格分隔的链接选项（Linker Directives），如 -defaultlib:LIBCD -defaultlib:OLDNAMES。
- .debug\$S，调试符号信息（Symbolic Information）。
- .bss，自由格式的未初始化数据。
- .text，目标代码。
- .debug\$T，调试类型信息（Type Information）。

在节的头信息之后是节的数据，每个节可以最多有 3 种数据：原始数据（Raw Data）、重定位信息（Relocation）和行号信息（Line Number）。从图中可以看到，.directive 节只有原始数据，.debug\$S 有原始数据和重定位信息，.bss 节没有任何数据，.text 节有 3 种数据，.debug\$T 节只有原始数据。

在节数据之后是调试符号表和字符串表。

25.3.1 IMAGE_FILE_HEADER 结构

目标文件的开始处固定为一个 IMAGE_FILE_HEADER 结构，用来描述整个文件的基本信息，MSDN 中给出了这个结构的定义。

```
typedef struct _IMAGE_FILE_HEADER
{
    WORD Machine;                                //CPU 架构
```

```

WORD NumberOfSections;           //节数量
DWORD TimeDateStamp;           //时间戳
DWORD PointerToSymbolTable;    //符号表的偏移
DWORD NumberOfSymbols;          //符号表中的符号数量
WORD SizeOfOptionalHeader;     //可选文件头结构的大小
WORD Characteristics;          //特征
} IMAGE_FILE_HEADER, *PIMAGE_FILE_HEADER;

```

其中 Machine 字段表示这个目标文件所针对的 CPU 架构，常见的值有 IMAGE_FILE_MACHINE_I386 (0x014c) 代表 x86 CPU，IMAGE_FILE_MACHINE_IA64 (0x0200) 代表 Intel 安腾 CPU (IPF)，IMAGE_FILE_MACHINE_AMD64 (0x8664) 代表 x64 CPU。NumberOfSections 字段表示这个文件中所包含节 (Section) 的个数。TimeDateStamp 是文件的时间戳，其值为 1970 年 1 月 1 日至今的秒数。PointerToSymbolTable 是符号表起始字节相对于文件开头的偏移，如果没有符号表，这个值则为 0。对于 SymOption.obj，PointerToSymbolTable 为 0xF21，NumberOfSymbols 为 0x4C，这说明在文件的 0xF21 偏移处存放了 0x4C 个符号 (IMAGE_SYMBOL 结构)。SizeOfOptionalHeader 字段表示可选的文件头结构的长度，目标文件通常没有这个结构可选，因此这个值为 0。Characteristics 字段用来表示映像文件的属性，它的每一个二进制位对应于一种属性，WinNT.h 中的 IMAGE_FILE_XXX 系列常量定义了可能的属性标志。

25.3.2 IMAGE_SECTION_HEADER 结构

IMAGE_FILE_HEADER 结构之后，是连续多个 IMAGE_SECTION_HEADER 结构，其实际个数是由 IMAGE_FILE_HEADER 结构中的 NumberOfSections 字段决定的。

每个 IMAGE_SECTION_HEADER 结构用来描述一个节 (Section) 的概要信息，包括名称、长度和它所包含数据的偏移位置。图 25-1 右侧给出了 IMAGE_SECTION_HEADER 结构的定义，每个字段后的注释是.text 节所对应的值。例如，Name 字段的值即是 ASCII 代码 “.text”，空余的字节被填充为 0。PointerToRawData 字段用来指定节的原始数据的偏移，其长度由 SizeOfRawData 指定。以.text 节为例，这两个字段的值分别为 0x5AB 和 0x4DA，从图中可以看出偏移 0x5AB 处确实是.text 节原始数据，其长度为 0x4DA (0xA85-0x5AB=0x4DA)。类似的，PointerToRelocations 和 PointerToLinenumbers 分别用来描述节的另外两种数据——重定位信息和行号信息的偏移。这两种数据都由多个固定长度的结构组成，其具体个数分别由 NumberOfRelocations 字段和 NumberOfLine-numbers 的值来指定。Characteristics 字段用来描述节的属性，它的每个二进制位对应于一种属性，WinNT.h 中的 IMAGE_SCN_XXX 系列常量定义了可能的属性标志。

25.3.3 节的重定位信息和行号信息

每个节可以有 3 种数据，原始数据、重定位信息和行号信息。原始数据会因节的

不同而使用不同的格式。重定位信息用来描述链接和加载映像文件时应该如何修改节数据。每一条重定位信息是一个 IMAGE_RELOCATION 结构：

```
typedef struct _IMAGE_RELocation {
    UINT32 VirtualAddress;           //需要重定位项目的 RVA 地址
    UINT32 SymbolTableIndex;         //相关符号结构在符号表中的索引
    UINT16 Type;                   //重定位类型
} IMAGE_RELocation;
```

其中 VirtualAddress 字段用来指定要进行重定位项目 (Item) 的 RVA，即相对于文件开头的偏移地址，Type 用来指定要应用的重定位方法，其值为 IMAGE_REL_XXX 系列常量之一。SymbolTableIndex 用来指定与这条重定位信息相关的符号信息。符号信息中包含了用来重定位的地址。

行号信息用来描述源代码行与目标代码的对应关系，每条源代码行信息是一个 IMAGE_LINENUMBER 结构：

```
typedef struct _IMAGE_Linenumber {
    union { //联合结构，如果 Linenumber 等于 0，则 SymbolTableIndex 有效
        UINT32 SymbolTableIndex; //对应函数名称的符号索引
        UINT32 VirtualAddress; //目标代码相对于函数起始地址的偏移
    } Type; //相对于函数起始行号的偏移
    UINT16 Linenumber; //相对于函数基础行号的偏移
} IMAGE_Linenumber;
```

其中 Linenumber 是相对于函数基础行号的偏移。如果 Linenumber 为 0，那么 Type 联合字段应该取 SymbolTableIndex，用来索引符号表中的一个函数名称。如果 Linenumber 不为 0，那么 VirtualAddress 是可执行代码的地址。以 SymOption 程序 SymOption.cpp 为例，第一行的行号为 1，WinMain 函数的起始大括号的行号为 24，使用调试器的反汇编窗口观察它的目标地址是 0x00401040，这个函数中包含目标代码的第一个源代码行 (LoadString...) 是 30 行，它的目标代码地址为 0x00401064。观察描述这一行源代码的 IMAGE_LINENUMBER 结构，VirtualAddress 字段等于 0x24，Linenumber 字段等于 6，刚好分别是目标代码和源代码相对于函数起始地址和起始行号的偏移。

使用 dumpbin symoption.obj /LINENUMBERS 可以列出 symoption.obj 文件所包含的所有源代码行号信息，以下是关于 WinMain 函数的信息：

```
LINENUMBERS #4
Symbol index:      10 Base line number:   24
Symbol name = _WinMain@16
00000024( 30) 00000040( 31) 0000005C( 32) 00000068( 35)
0000007C( 37) 00000080( 40) 00000098( 43) 000000B5( 45)
000000D4( 47) 000000E7( 48) 000000FA( 50) 000000FC( 52)
000000FF( 53)
```

第 2 行的 10 (十六进制) 是这个函数的符号结构在符号表中的索引，24 (十进制) 是根据函数的.bf 符号读取的函数基础行号，稍后我们会介绍。

25.3.4 存储调试数据的节

PE 规范 (参考文献 6) 中定义了 4 种用于存储调试数据的节，分别如下。

.debug\$F: 用于存储函数的帧数据，包含多个 FPO_DATA 结构（见下文）。

.debug\$S: 用于存储 VC 编译器专用的符号（Symbolic）信息。

.debug\$P: 用于存储 VC 编译器专用的预编译（Precompiled）信息。

.debug\$T: 用于存储 VC 编译器专用的类型（Type）信息。

FPO_DATA 结构的定义如下：

```
typedef struct _FPO_DATA {
    DWORD      ulOffStart;           // 函数代码的第一条指令的偏移
    DWORD      cbProcSize;          // 函数的字节数
    DWORD      cdwLocals;           // 以 DWORD 为单位的局部变量长度
    WORD       cdwParams;           // 以 DWORD 为单位的参数长度
    WORD       cbProlog : 8;         // 函数序言的字节数
    WORD       cbRegs   : 3;         // 保存的寄存器数量
    WORD       fHasSEH  : 1;         // 如果栈帧中包含 SEH 登记结构，则为 1
    WORD       fUseBP   : 1;         // 如果使用 EBP 寄存器记录栈帧地址，则为 1
    WORD       reserved  : 1;        // 保留
    WORD       cbFrame   : 2;        // 帧类型
} FPO_DATA;
```

其中 cbFrame 是常量 FRAME_FPO (0)、FRAME_TRAP (1) 和 FRAME_TSS (2) 之一。因为 FPO 是一种优化措施，所以通常只有在发布版本中才会使用，这也意味着调试版本的 OBJ 文件中通常没有.debug\$F 节。使用 Visual Studio 附带的 dumpbin 工具，可以看到在发布版本的 SymOption.obj 文件中有 5 个.debug\$F 节。每个节都有 16 个字节的原始数据和 1 个重定位表。仔细观察，可以发现每个节描述了一个使用 FPO 的函数，例如以下是关于 WinMain 函数的.debug\$F 节的原始数据：

```
RAW DATA #4
00000000: 00 00 00 00 B9 00 00 00 07 00 00 00 04 00 00 14 ....1.....
```

使用 dumpbin symoption.obj /FPO 命令可以以更友好的方式显示 FPO 信息：

FPO Data (1)									
RVA	Proc	Size	Locals	Regs	Prolog	Use BP	Has SEH	Type	Frame Params
00000000		185	28	4	0	Y	N	fpo	16

使用 dumpbin symoption.obj /RELOCATIONS 可以显示重定位表的信息：

RELOCATIONS #4				Symbol	Symbol
Offset	Type	Applied To	Index	Name	
00000000	DIR32NB	00000000	E	_WinMain@16	

微软没有公开.debug\$S 和.debug\$T 数据所使用的结构，使用 dumpbin 工具可以打印出它们的概要信息和原始数据，以下是 SymOption.obj 的.debug\$T 节的原始数据：

```
RAW DATA #5
00000000: 02 00 00 00 56 00 16 00 61 A0 0F 46 02 00 00 00 ....V...a?.F...
00000010: 48 63 3A 5C 64 69 67 5C 64 62 67 5C 61 75 74 68 Hc:\dig\dbg\auth
00000020: 6F 72 5C 63 6F 64 65 5C 63 68 61 70 32 35 5C 73 or\code\chap25\s
00000030: 79 6D 6F 70 74 69 6F 6E 5C 73 79 6D 6F 70 74 69 ymoption\symopti
00000040: 6F 6E 5F 5F 5F 77 69 6E 33 32 5F 7A 69 70 64 62 on__win32_zipdb
00000050: 5C 76 63 36 30 2E 70 64 62 F3 F2 F1 \vc60.pdb??

```

根据以上数据所包含的关于 VC60.pdb 文件的全路径，可以猜测这是指向 VC60.pdb 的一个链接，第 25.6 节我们会介绍 VC 编译器默认是将类型符号存储在一个单独的 PDB 文件中，VC6 编译器使用的就是 VC60.pdb。也就是说，VC 编译器将类型符号放在 PDB 文件中，目标文件的.debug\$T 节只存储 PDB 文件的路径。

25.3.5 调试符号表

在目标文件中，通常有一个用于存储符号信息的符号表，因为它使用的是 COFF 格式，所以这个符号表经常被称为 COFF 符号表（COFF Symbol Table）。COFF 符号表的每个表项是一个固定长度（18 字节）的 IMAGE_SYMBOL（见清单 25-2）或 IMAGE_AUX_SYMBOL 结构。

清单 25-2 IMAGE_SYMBOL 结构

```
typedef struct _IMAGE_SYMBOL {
    union {
        BYTE   ShortName[8];           // 符号名称，联合结构
        struct {
            DWORD   Short;           // 不超过 8 字节的符号名称
            DWORD   Long;            // 描述长度大于 8 字节的符号名称
        } Name;
        DWORD   LongName[2];          // 如果不为 0，则 ShortName 为符号名
    } N;
    DWORD   Value;                // 长符号名的偏移地址
    SHORT   SectionNumber;         // 以 DWORD 方式来访问这个枚举结构
    WORD    Type;                 // 取值，与类型有关
    BYTE    StorageClass;          // 节的编号
    BYTE    NumberOfAuxSymbols;    // 符号类型，见下文
} IMAGE_SYMBOL, *PIMAGE_SYMBOL;
#define IMAGE_SIZEOF_SYMBOL 18
// 存储方式
// 附属符号的数量
```

如果名称不超过 8，那么起始 8 个字节便是符号的名称，此时联合结构中的 Short 字段不为 0，ShortName 字段即名称。如果 Short 字段为 0，那么说明名称超过 8 个字符，此时 Long 字段是这个名称在字符串表中的偏移，根据这个偏移可以在字符串表中找到符号的名称。SectionNumber 是与这个符号相关联节的序号（从 1 开始），以下值具有特殊的含义：

IMAGE_SYM_UNDEFINED (0) - 未定义的或者外部的符号
 IMAGE_SYM_ABSOLUTE (-1) - 这个符号是一个绝对的值，不与任何节关联，例如编译器 ID 等
 IMAGE_SYM_DEBUG (-2) - 供调试器使用的特殊符号。如.file 符号（见下文）

Type 字段表示符号的类型，VC 编译器通常只使用两个值，0x20 表示函数，0x0 表示不是函数。StorageClass 字段表示这个符号所描述对象的存储类型，WinNT 文件中的 IMAGE_SYM_CLASS_XXX 系列常量定义了可能的存储类型。

NumberOfAuxSymbols 字段用来指定跟随在当前符号之后的辅助符号的个数。例如一个.file 符号（名称为.file，存储类型为 IMAGE_SYM_CLASS_FILE）后会跟着一个或

多个用来描述源文件名称的辅助符号。比如 SymOption.obj 的第一个符号就是.file 符号，它的各字段值为：

```
ShortName[8] = 2E 66 69 6C 65 00 00 00 //即 ASC 代码.file
Value = 00000000 //取值，只对于静态变量等符号有效
SectionNumber = FFFE //即 IMAGE_SYM_DEBUG
Type = 0000 //非函数符号
StorageClass = 67 //IMAGE_SYM_CLASS_FILE
NumberOfAuxSymbols = 03 //附属符号数量
```

NumberOfAuxSymbols 等于 3，表示在这个符号后的 3 条记录都是这个符号的辅助信息，其值就是源程序文件的全路径，c:\...\SymOption\SymOption.cpp。

常见的辅助符号还有对函数和节的描述，WinNT.H 的 IMAGE_AUX_SYMBOL 结构定义了这些辅助符号的结构，IMAGE_AUX_SYMBOL 结构的长度也是 18 个字节，这保证了 COFF 符号表中的所有表项是等长的。

25.3.6 COFF 字符串表

在目标文件的 COFF 调试符号表之后通常是 COFF 字符串表。因为 IMAGE_FILE_HEADER 结构中没有记录字符串表的偏移地址，所以，寻找字符串表的方法是，使用调试符号表的地址加上调试符号个数乘以每个符号的长度。即使用如下公式：

$$\text{字符串表地址} = \text{PointerToSymbolTable} + \text{NumberOfSymbols} * \text{IMAGE_SIZEOF_SYMBOL}$$

以 SymOption.obj 为例：字符串表地址 = 0xF21 + 0x4C * 0x12 = 0x1479，这与使用 PEView 工具看到的结果是一致的。字符串表的起始 4 个字节是字符串表的长度，包括这 4 个字节本身。在长度之后便是以 0 结束的字符串。

字符串表中存放的字符串大多都是长度大于 8 的符号名，因为 COFF 符号表中的 IMAGE_SYMBOL 结构是固定长度的，只能存储长度不超过 8 的符号名。如果符号名称的长度大于 8，那么编译器会将其存储在字符串表中，并将其相对字符串表起始位置的偏移值保存在 IMAGE_SYMBOL 结构的 Long 字段中。以 SymOption.obj 中全局变量 hInst(struct HINSTANCE__ * hInst) 的符号为例，它的修饰名是?hInst@@3PAUHINSTANCE__@@A。因为这个名称超过了 8 个字符，所以它被保存在字符串表中，偏移位置是 4（第一个字符串），因此其 Short 字段的值是 0，表示其实际名称在字符串表中，Long 字段的值是 4，表明在字符串表中的偏移是 4。

25.3.7 COFF 符号例析

使用 DumpBin 工具可以列出目标文件符号表中所包含的所有符号。例如，使用如下命令可以将 SymOption.obj 中的符号信息写到文本文件 dump_sym.txt 中：

```
C:\...\SymOption\obj>dumpbin symoption.obj /symbols >dump_sym.txt
```

清单 25-3 列出了 dump_sym.txt 文件中所包含的主要信息。

清单 25-3 观察目标文件中的符号表（节选）

```

COFF SYMBOL TABLE
000 00000000 DEBUG notype      Filename    | .file
                           C:\dig\dbg\author\code\chap25\SymOption\SymOption.cpp
004 000B2306 ABS   notype      Static     | @comp.id
005 00000000 SECT1 notype      Static     | .drectve
                           Section length 27, #relocs 0, #linenums 0, checksum 0
007 00000000 SECT2 notype      Static     | .debug$S
                           Section length 3F4, #relocs 12, #linenums 0, checksum E9AFC280
00C 00000064 SECT3 notype      External   | ?szTitle@@3PADA (char * szTitle)
00E 00000000 SECT4 notype      Static     | .text
                           Section length 4DA, #relocs 44, #linenums 44, checksum 3E89ABBB
010 00000000 SECT4 notype ()  External   | _WinMain@16
                           tag index 00000019 size 00000110 lines 00000D2D next function 0000001E
012 00000000 UNDEF notype ()  External   | __imp_DispatchMessageA@4
019 00000000 SECT4 notype      BeginFunction | .bf
                           line# 0018 end 00000023
01B 0000000E SECT4 notype      .bf or.ef | .lf
01C 00000010 SECT4 notype      EndFunction | .ef
                           line# 0035
04A 00000000 SECT5 notype      Static     | .debug$T
                           Section length 5C, #relocs 0, #linenums 0, checksum 0

```

其中最左侧一列是符号的序号，然后依次是 Value 字段，SectionNumber 字段，Type 字段，StorageClass 字段和符号名称。

0 号符号的类型是文件名 (Filename)，其名称是.file，大多数 COFF 符号表都是以这样的符号开始的。接下来的 3 个符号是文件名符号的辅助符号，用来记录文件名，Dumpbin 便直接将文件名显示出来。

4 号符号是用来记录编译器 ID (版本) 的静态符号 (IMAGE_SYM_CLASS_STATIC) 的，它的 Value 字段的值是 000B2306，笔者猜想与编译器的版本号有关。SectionNumber 列是 ABS，代表 IMAGE_SYM_ABSOLUTE。

接下来的几个符号都是用来描述节 (Section) 的，而且都是一个主符号，后面跟一个辅助符号。5 号和 6 号符号描述的是.drectve 节，5 号描述了节的概况，6 号 (编号没有显示) 描述了它的细节，包括节的长度 (字节为单位)，节的重定位记录数 (0)，节的行号记录数和节的校验和。7 号和 8 号符号描述的是.debug\$S 节。

符号#00C 描述的是全局变量 szTitle，它前后的#00B 和#00D 描述的也是全局变量，清单 25-1 中省略了。

从#010 到#01D 的 14 个符号都是描述 WinMain 函数的，第一个符号 (#010) 描述了函数的名称，紧接其后的是描述函数的辅助符号，其中 0xD2D 是 IMAGE_AUX_SYMBOL 结构的 PointerToLineNumber 字段，即这个函数的行号信息 (IMAGE_LINENUMBER 结构) 的偏移地址，指向的是.text 节的行号信息部分，0xD2D 恰好是这个部分的起始地址 (参见图 25-1)。PointerToNextFunction 的值是 0x1E，这是下一个函数符号的序号。符号 0x12 到 0x18 都是 DT_FUNCTION 类型的符号，描述的是 WinMain 函数中调用的函数。符号 0x19 和 0x20 描述的是函数的序言，.bf 是 Begin of Function 的缩

写，其中 line# 是函数的基础行号，通常也就是函数的起始大括号所对应的行号（从 1 开始），其中 end 值 00000023 是下一个函数的 bf 符号的序号。Dumpbin 使用 end 作为这个字段的名称有些不妥，按照 PE 规范，它应该是 PointerToNextFunction。

符号 0x1B 的名称 lf 代表的是 lines in function，PE 规范中没有详细介绍这个符号，笔者推断这个符号的值（对于 WinMain 是 0xE）是函数中可跟踪的源代码行数，不包括空行和定义变量的行。符号 0x1C 的名称 ef 代表的是 end of function，它描述的是函数的结语，它的值 0x110 是函数的代码长度（指令的总字节数），其值与符号 0x11 中的 size 值相同。0x1D 是 ef 符号的辅助符号，它的 line# 值（0x35，即 53）是这个函数的结束行的行号，通常就是右大括号的位置。

本节我们介绍了目标文件中的调试信息，主要目的是让读者对调试符号的实际内容和存储方法有一个感性的认识，以便更好地理解后面的内容，而并不是让大家死记硬背其中的数据结构和细枝末节。并且，其中的某些细节会因为编译器的版本和编译选项的变化而不同，我们将在第 25.7 节介绍如何通过设置编译选项来定制调试信息的丰富程度和存储方式。

25.4 PE 文件中的调试信息

Windows 操作系统的可执行文件主要有两种格式，一种是 16 位 Windows 所使用的 NE (New Executable) 格式，另一种是 32 位和 64 位 Windows 所使用的 PE 格式。因为 16 位 Windows 已经很少使用，所以今天的大多数 Windows 可执行文件使用的都是 PE 格式。

PE 格式是 Windows NT 系列操作系统的第一个版本 NT 3.1 所引入的，其全称叫 Portable Executable，这个名字反映了 NT 3.1 的一个重要设计目标，那就是可移植性 (Portable)，因为设计 NT 时，期望可以很容易地把它移植到其他 CPU 架构中运行。

从微软网站可以下载名为 Microsoft Portable Executable and Common Object File Format Specification 的文档，它是了解 PE 格式的最好资料，这份文档同时定义了 PE 文件和 OBJ 文件的格式，通常被简称为 PE-COFF 规范。当笔者写作这个内容时，下载这个文档的 URL 为：

<http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx>

25.4.1 PE 文件布局

PE 文件由固定结构的文件头和不确定个数的若干个数据块构成，图 25-2 显示了 HiWorld.exe 程序的文件结构。HiWorld.exe 是由 VC2005 编译产生的一个典型的 Windows 可执行文件。

在文件的最前面是 IMAGE_DOS_HEADER 结构，WinNT.H 中有这个结构的详细

定义，但因为这个结构是为了兼容老的 DOS 操作而设计的，所以今天它的价值已经很小，值得我们注意的只有起始处的 e_magic 字段和最末的 e_lfanew 字段。e_magic 字段是所有 DOS 程序都使用的文件签名，其内容固定为字符“MZ”的 ASCII 代码 0x5A4D。e_lfanew 字段是新的 EXE 文件头的偏移地址，对于 HiWorld.exe，它的值为 0xE8。IMAGE_DOS_HEADER 结构后面是一段 16 位的代码，当这个 EXE 程序在 DOS 操作系统下执行时，这段程序会被执行，显示“This program cannot be run in DOS mode.” 后退出，因此这段代码通常被称为 DOS Stub Program。

IMAGE_NT_HEADERS 结构是 PE 文件的真正开始，由 3 个部分组成，最前面是 4 个字节的 PE 文件签名，其内容固定为 0x00004550，即字符串“PE\0\0”。PE 签名之后是一个 IMAGE_FILE_HEADER 结构，它与 OBJ 文件的头结构是相同的，接下来是一个 IMAGE_OPTIONAL_HEADER 结构（稍后介绍）。

在 IMAGE_NT_HEADERS 结构之后，是一系列 IMAGE_SECTION_HEADER 结构，用来描述 PE 文件中的各个节（section）的概况，常见的节有以下几种。

- .text 节，代码。
- .rdata 节，只读的已经初始化过的数据（Read-only initialized data）。
- .data 节，数据。
- .idata 节，输入数据（Import Data）。
- .rsrc 节，资源，比如 GUI 程序的菜单、图标、位图等，也可以存放 manifest 文件等。

图 25-2 中显示的是调试版本的 HiWorld.exe，如果是发布版本，其结构很类似（见图 25-3）。

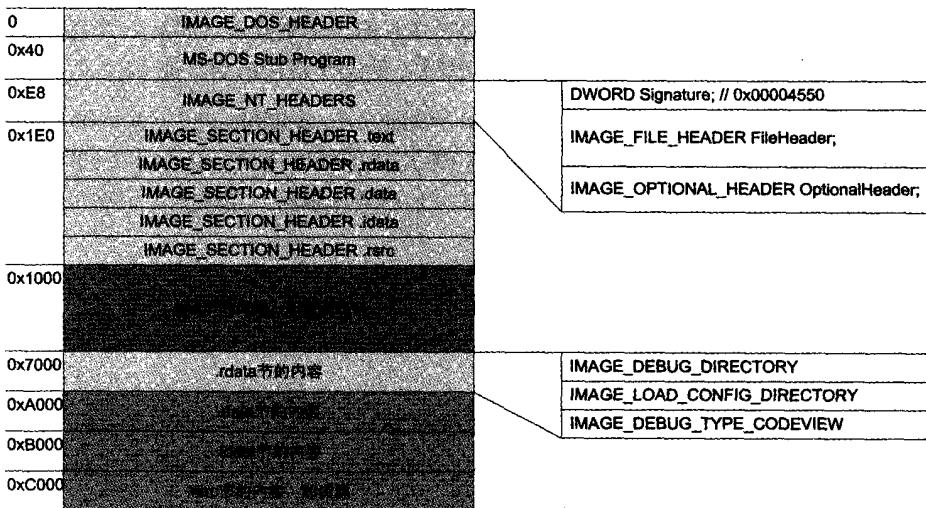


图 25-2 典型 PE 文件（HiWorld.exe 调试版本）的文件布局

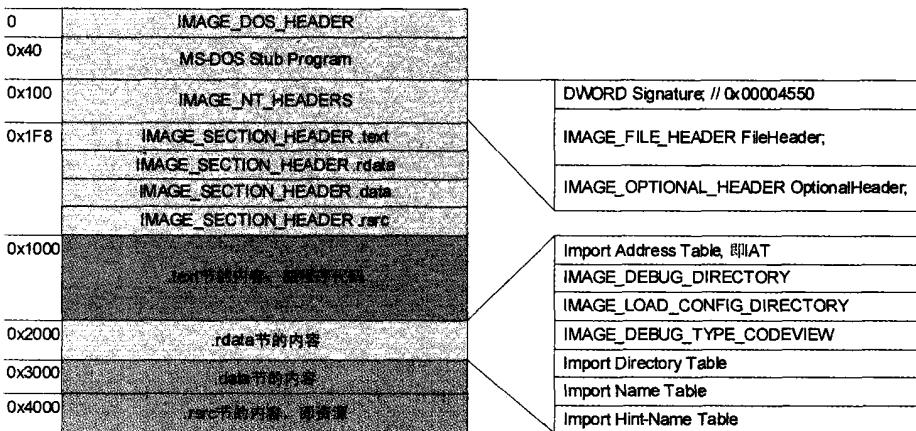


图 25-3 典型 PE 文件 (HiWorld.exe 发布版本) 的文件布局

比较图 25-2 和图 25-3，较大的差异就是发布版本将.image 节的内容合并到.rdata 中，这主要是为了使发布版本的 EXE 文件更小。

25.4.2 IMAGE_OPTIONAL_HEADER 结构

IMAGE_OPTIONAL_HEADER 结构是 PE 文件中的一个非常重要的结构，其中包含了 PE 文件的很多重要属性，这些属性对指导操作系统如何加载这个 PE 文件起着重要作用。MSDN 描述了这个结构和它的各个字段。表 25-2 列出了各个字段的定义和它们在 HiWorld.exe（调试版本）文件中的取值。

表 25-2 IMAGE_OPTIONAL_HEADER 结构

字段	取值	说明
WORD Magic;	010B	Magic 代码, 0x10B 代表 32 位的 PE 文件
BYTE MajorLinkerVersion;	08	链接器的主版本号
BYTE MinorLinkerVersion;	00	链接器的小版本号
DWORD SizeOfCode;	6000	代码节的长度 (字节)
DWORD SizeOfInitializedData;	15000	初始化数据节的长度
DWORD SizeOfUninitializedData;	0	未初始化数据节的长度
DWORD AddressOfEntryPoint;	0x22D0	入口函数的偏移地址
DWORD BaseOfCode;	1000	代码节的基地址
DWORD BaseOfData;	7000	数据节的基地址
DWORD ImageBase;	00400000	推荐的加载地址 (Preferred Address)
DWORD SectionAlignment;	1000	节的对齐尺寸
DWORD FileAlignment;	1000	文件的对齐尺寸
WORD MajorOperatingSystemVersion;	4	运行这个文件所需要 OS 的主版本号
WORD MinorOperatingSystemVersion;	0	运行这个文件所需要 OS 的小版本号
WORD MajorImageVersion;	0	本文件的主版本号
WORD MinorImageVersion;	0	本文件的小版本号
WORD MajorSubsystemVersion;	4	Windows 子系统的主版本号

续表

字段	取值*	说明
WORD MinorSubsystemVersion;	0	Windows 子系统的小版本号
DWORD Win32VersionValue;	0	保留未用
DWORD SizeOfImage;	1C000	内存映像大小(字节数)
DWORD SizeOfHeaders;	1000	文件头的大小(字节数)
DWORD CheckSum;	00000000	校验和
WORD Subsystem;	2	所需的 Windows 子系统, 2 代表 Windows GUI
WORD DllCharacteristics;	0	DLL 属性, 如果是 WDM 驱动程序, 则为 0x2000
DWORD SizeOfStackReserve;	00100000	栈的默认保留大小
DWORD SizeOfStackCommit;	1000	栈的默认初始提交大小
DWORD SizeOfHeapReserve;	00100000	堆的默认保留大小
DWORD SizeOfHeapCommit;	1000	堆的默认提交大小
DWORD LoaderFlags;	0	加载标志
DWORD NumberOfRvaAndSizes;	10	数据目录的个数
IMAGE_DATA_DIRECTORY DataDirectory[16];	见下文	数据目录

*所有数字使用的都是十六进制。

其中, DataDirectory 字段用于定位导入表、导出表、资源表、异常表、调试信息表、签名表、TLS 和延迟加载描述符表等其他数据表, 尽管它的默认元素个数是 16, 但其实际个数应该根据 NumberOfRvaAndSizes 字段来确定。它的每个元素是一个 IMAGE_DATA_DIRECTORY 结构:

```
typedef struct _IMAGE_DATA_DIRECTORY {
    DWORD VirtualAddress; // 表起始位置的 RVA
    DWORD Size;           // 表的大小, 字节数
} IMAGE_DATA_DIRECTORY, *PIMAGE_DATA_DIRECTORY;
```

这个定义中包含了该结构所描述的目录表的地址和大小, 但是并没有描述目录表的用途, 这是因为 PE 规范已经规定好了 DataDirectory 数组的元素内容。表 25-3 列出了 0~15 号元素所描述的目录表和它在 HiWorld.exe (调试版本) 文件中的取值。

表 25-3 PE 文件的数据表 (数值来自 HiWorld.exe 文件)

元素序号	所描述的数据表	表的 RVA	表的大小
0	输出表 (.edata)	0	0
1	导入表 (.idata)	B000	50
2	资源表 (.rsrc)	C000	F07C
3	异常表 (.pdata)	0	0
4	签名表	0	0
5	Base Relocation 表 (.reloc)	0	0
6	调试目录	7620	1C
7	架构相关的数据目录	0	0
8	全局指针目录	0	0
9	线程局部存储 (TLS)	0	0

续表

元素序号	所描述的数据表	表的 RVA	表的大小
10	加载配置 (Load Configuration) 目录	8170	40
11	Bound Import Directory	0	0
12	导入地址表 (IAT)	B260	210
13	延迟导入目录	0	0
14	COM/CLI 描述符表	0	0
15	保留	0	0

使用 dumpbin 工具可以观察一个 PE 文件的头信息，包括 IMAGE_OPTIONAL_HEADER 结构和目录表，例如以下命令可以显示 HiWorld.exe 的所有头信息：dumpbin /headers hiworld.exe。

25.4.3 调试数据目录

PE 文件可以有一个可选的调试目录 (Debug Directory)，用来描述 PE 文件中所包含的调试信息的种类和位置。调试目录可以位于专门的.debug 节中，也可以位于其他任何节中，或者不在任何节中。IMAGE_OPTIONAL_HEADER 结构的 DataDirectory[6] 字段用来记录调试目录的位置和长度，以表 25-3 所示的 HiWorld.exe 为例，DataDirectory[6] 的 RVA 地址是 0x7620，大小是 0x1C，说明这个文件的偏移 0x7620 处有一个长度为 0x1C 的调试目录。使用 PEView 工具观察，这个区域位于.rdata 节中。

调试目录可以包含若干个固定长度的 IMAGE_DEBUG_DIRECTORY 结构（见清单 25-4），每个结构描述一个调试信息数据块。

清单 25-4 描述调试数据块的 IMAGE_DEBUG_DIRECTORY 结构

```
typedef struct _IMAGE_DEBUG_DIRECTORY {
    DWORD Characteristics;           // 保留，必须为 0
    DWORD TimeDateStamp;            // 调试数据的产生时间
    WORD MajorVersion;              // 调试数据格式的主版本号
    WORD MinorVersion;              // 调试数据格式的次版本号
    DWORD Type;                     // 调试信息的类型，详见下文
    DWORD SizeOfData;               // 调试数据的长度，不包括本目录项
    DWORD AddressOfRawData;         // 调试数据的内存地址（相对于映像文件的基地址）
    DWORD PointerToRawData;         // 调试数据在文件中的偏移，即 RVA
} IMAGE_DEBUG_DIRECTORY, *PIMAGE_DEBUG_DIRECTORY;
```

其中 Type 字段可以为表 25-4 中的常量之一。

表 25-4 PE 文件中的调试数据类型

常量	值	含义
IMAGE_DEBUG_TYPE_UNKNOWN	0	未知
IMAGE_DEBUG_TYPE_COFF	1	COFF 格式的调试信息，包括行号、符号表和字符串表
IMAGE_DEBUG_TYPE_CODEVIEW	2	CodeView 格式的调试信息
IMAGE_DEBUG_TYPE_FPO	3	帧指针省略 (Frame pointer omission, FPO) 信息

续表

常量	值	含义
IMAGE_DEBUG_TYPE_MISC	4	描述.dbg 文件的位置
IMAGE_DEBUG_TYPE_EXCEPTION	5	异常信息,.pdata 节的拷贝
IMAGE_DEBUG_TYPE_FIXUP	6	保留
IMAGE_DEBUG_TYPE OMAP_TO_SRC	7	从内存映像的实际地址到 PE 文件中的地址的映射
IMAGE_DEBUG_TYPE OMAP_FROM_SRC	8	从 PE 文件中的地址到内存映像中实际地址的映射
IMAGE_DEBUG_TYPE_BORLAND	9	Borland 格式的调试信息

以刚才使用的 HiWorld.exe 为例, Type 字段等于 2, SizeOfData 字段等于 0x45, PointerToRawData 字段的值等于 0x84DC, 这意味着在文件的 0x84DC 处, 有一个 CodeView 格式的调试数据块, 长度为 0x45 个字节。事实上, 这个数据块就是我们曾经介绍过的 RSDS 信息(见清单 25-1), 即符号文件链接, 它的内容主要是 HiWorld.PDB 文件的全路径, 下面将详细描述。

25.4.4 调试数据

根据调试目录结构 (IMAGE_DEBUG_DIRECTORY) 中的信息可以找到各个调试数据块。其方法就是将 PointerToRawData 字段所指定的文件偏移加上 PE 模块的基地址。如果是直接观察文件, 那么基址便是 0, 如果是用 WinDBG 调试, 使用 `lm` 命令可以观察各个模块的基地址。以刚才介绍的 HiWorld.exe 为例, 偏移地址是 0x84DC, 模块地址是 0x00400000 (这是 EXE 模块的常用地址), 二者相加, 便得到调试数据的地址为 0x004084DC, 结合其长度信息 0x45, 便可以使用 WinDBG 的 `db` 命令直接观察对应的调试数据了 (见清单 25-5)。

清单 25-5 在调试器中观察 PE 文件的调试数据

```
0:000> db 0x004084DC 145h
004084dc 52 53 44 53 dc 34 d5 dd-a7 82 a8 41 9c 69 5b c4  RSDS.4.....A.i[.
004084ec 51 2b 1f 9b 02 00 00 00-63 3a 5c 64 69 67 5c 64 Q+.....c:\dig\d
004084fc 62 67 5c 61 75 74 68 6f-72 5c 63 6f 64 65 5c 62 bg\author\code\b
0040850c 69 6e 5c 44 65 62 75 67-5c 48 69 57 6f 72 6c 64 in\Debug\HiWorld
0040851c 2e 70 64 62 00 .pdb.
```

其中, 前 4 个字节是 CodeView 数据的版本签名, RSDS 代表这个数据块是 CV_INFO_PDB70 格式的 (清单 25-1)。RSDS 之后的 16 个字节是 GUID, 而后的 01 00 00 00 是 Age 字段, 即 DWORD 格式的 1, 剩下的便是字符串 PdbName。以下是 dumpbin 所显示的结果:

```
Format: RSDS, {DDD534DC-82A7-41A8-9C69-5BC4512B1F9B}, 2, c:\...\HiWorld.pdb
```

RSDS 或 NB10 类型的 CodeView 数据块是 PE 文件中最常见的调试数据, 使用 VC 编译器构建的调试版本的各种 PE 文件 (包括 DLL、SYS、EXE) 默认都包含这个数据块。对于发布版本, 只要项目的链接选项中设置了产生调试信息 (Generate debug info)

选项，那么也会包含这个数据块。

今天的 VC 编译器都默认使用单独的 PDB 文件来存储调试信息，因此 NB10 或 RSDS 类型的 CodeView 数据块通常也是 PE 文件中唯一的调试数据块，因为真正的调试数据都在 PDB 文件中，PE 文件中只留下这个小的数据块来索引 PDB 文件就够了。但是，尽管已经不流行，PE 规范确实支持在 PE 文件中存储各种调试数据，包括 COFF 格式的调试数据，RSDS/NB10 类型之外的 CodeView 数据，以及其他扩展格式的调试数据。为了与存储在单独文件中的调试数据相区别，以这种方式存储的调试信息通常被称为集成的调试信息（Integrated Debug Information）。

使用 PE 文件存储所有调试信息是旧的 MSC 编译器和很多 Borland 编译器的默认做法。VC 编译器在 VC7（Visual Studio .NET 2002）还一直支持这种做法，但是从 VC7 开始只支持使用单独的 PDB 文件来存储调试信息（PE 文件中只有一个 RSDS 或 NB10 类型的 CodeView 数据块）。

例如，图 25-4 所示的 SymOption.exe 文件中就包含了 COFF 格式的调试数据。这个 EXE 文件是使用 VC6 产生的，我们将在第 25.7 节介绍如何设置链接选项，以便在 PE 文件中存储集成的调试信息。

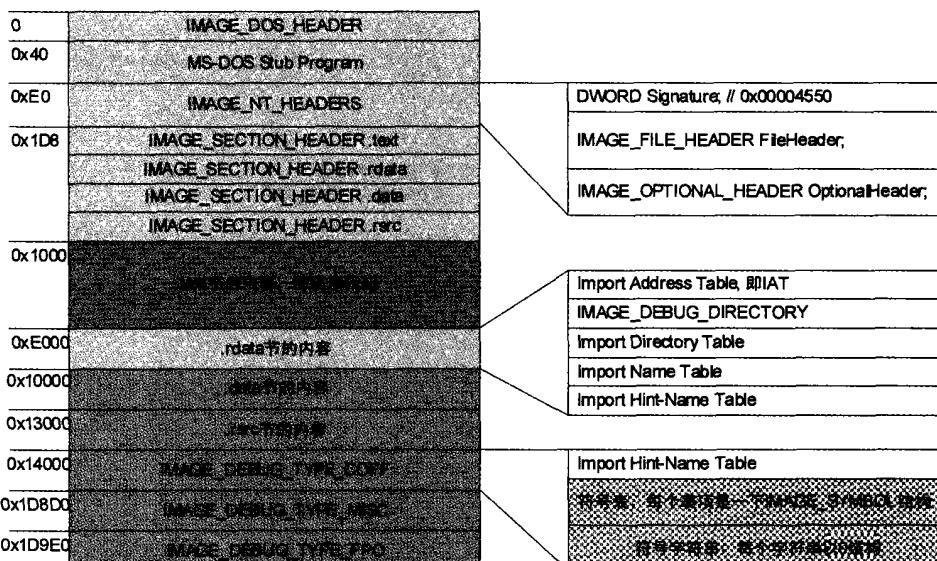


图 25-4 在 PE 文件中存储 COFF 格式的调试信息

从图 25-4 中我们可以看到，从 0x14000 开始的 3 个数据块都是用来存储调试信息的，分别为 IMAGE_DEBUG_TYPE_COFF、IMAGE_DEBUG_TYPE_MISC 和 IMAGE_DEBUG_FPO。在 IMAGE_DEBUG_TYPE_COFF 块中又包含了 IMAGE_COFF_SYMBOLS_HEADER、IMAGE_SYMBOL 符号表和字符串表 3 个部分，其中后两个部

分与上一节介绍的目标文件中的符号信息具有相同的格式。

有两种方法可以定位到 PE 文件中的调试数据块。第一种是通过 **IMAGE_OPTIONAL_HEADER** 结构的 **DataDirectory[6]** 字段找到调试目录的起始位置 (0xE130)，根据其长度 (0x54) 计算出调试目录中所包含的目录项个数 ($84/28 = 3$)，然后，就可以根据每个目录项 (**IMAGE_DEBUG_DIRECTORY** 结构) 中的 **PointerToRawData** 字段找到它所描述的调试数据块了。第二种方法是通过文件 **IMAGE_NT_HEADERS** 中的 **IMAGE_FILE_HEADER** 子结构的 **PointerToSymbolTable** 字段直接找到位于 **IMAGE_DEBUG_TYPE_COFF** 块中的符号表。

25.4.5 使用 WinDBG 观察 PE 文件中的调试信息

下面我们介绍如何使用 WinDBG 来观察 PE 文件中的调试信息。我们以使用 VC8 构建的发布版本 HiWorld.exe 为例。运行 WinDBG 并打开 HiWorld.exe(code\bin\release)，使用 **lm** 命令得到它的地址为 0x00400000。我们知道 PE 文件的开始处是一个 **IMAGE_DOS_HEADER** 结构，因此发出如下命令：

```
0:000> dt IMAGE_DOS_HEADER 00400000
+0x000 e_magic           : 0x5a4d
+0x03c e_lfanew          : 256
```

字段 **e_lfanew** 的值 256 (0x100) 是 **IMAGE_NT_HEADERS** 结构的偏移，因此：

```
0:000> dt IMAGE_NT_HEADERS 00400000+100
HiWorld!IMAGE_NT_HEADERS
+0x000 Signature        : 0x47d25acb
+0x004 FileHeader       : _IMAGE_FILE_HEADER
+0x018 OptionalHeader   : _IMAGE_OPTIONAL_HEADER
```

即 **OptionalHeader** 字段的偏移是 0x18，因此可以使用如下命令来显示 **IMAGE_OPTIONAL_HEADER** 结构中的 **DataDirectory** 字段：

```
0:000> dt _IMAGE_OPTIONAL_HEADER 00400000+100+18 -ny Data*
HiWorld!_IMAGE_OPTIONAL_HEADER
+0x060 DataDirectory : [16] _IMAGE_DATA_DIRECTORY
```

接下来要显示 **DataDirectory** 数组：

```
0:000> dt -ca16 _IMAGE_DATA_DIRECTORY 00400000+100+18+60
[6] @ 004001a8+0x000 VirtualAddress 0x2130 +0x004 Size 0x1c
【省去了无关的行】
```

DataDirectory 数组的 6 号元素是关于调试目录 (**IMAGE_DEBUG_DIRECTORY**) 的，**VirtualAddress** 是目录的偏移地址，**0x1c** 是目录的大小。因为每个目录项的大小是 28 (0x1c) 个字节，因此这个目录中只包含一个目录项。于是可以使用 **dt IMAGE_DEBUG_DIRECTORY 00402130** 命令来显示这个目录项，结果与使用 **dumpbin** 工具观察的完全一致，从略。

25.4.6 调试信息的产生过程

那么，PE文件中的调试信息是如何产生的呢？概括说来，分为如下三个阶段。

收集阶段：编译器在编译源文件的过程中收集调试信息，然后存放在目标文件中。

集成阶段：链接器在链接目标代码的过程中，将分布在各个目标文件中的调试信息集成到PE文件中。

可选的调整压缩阶段：为了使调试信息布局紧凑，并满足调试器的格式需要，如果调试信息的格式是CodeView格式，那么VC链接器会在链接的最后阶段执行一个名为CVPACK.exe的工具，这个工具的目标是对已经集成到PE文件中的调试信息做最后的整理和压缩。如果使用的是COFF格式的调试信息，那么不需要这一步骤。

图 25-5 画出了产生调试信息的基本过程。

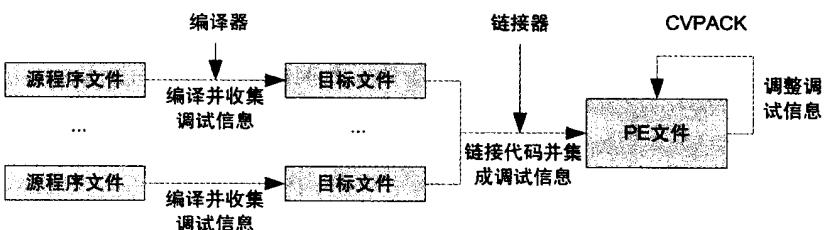


图 25-5 PE文件中的调试信息的产生过程

为了加深理解，可以通过一个简单的实验来观察链接器调用CVPACK工具的过程。在注册表的Image Execute Options表键下创建一个名为cvpack.exe的子键，并加入一个名为debugger的键值，使其包含WinDBGexe的完整文件名，例如c:\windbg\windbg.exe，这样当链接器执行cvpack.exe时，便会先启动WinDBG。

使用VC6打开SymOption项目，并重新编译链接Z7CV配置，在链接阶段，WinDBG便会被启动，其命令行为：

```
cvpack /nologo "SymOption__Win32_Z7CV/SymOption.exe"
```

这是链接器在执行cvpack，让其整理参数中指定的PE文件。此时观察PE文件，可以看到文件大小为577,707字节，使用dumpbin /HEADERS SymOption.exe可以看到，调试目录中列出了3个调试数据块：

Time	Type	Size	RVA	Pointer
4621DEB5	misc	110	00000000	14000
4621DEB5	fpo	50	00000000	14110
4621DEB5	cv	78F4B	00000000	14160

Image Name: SymO...32_Z7CV/SymOption.exe
Format: NB05

其中第三个是NB05格式的调试信息，其大小为0x78F4B字节。在WinDBG中输入g命令让cvpack执行，当其执行完毕时，再次观察，可以看到exe文件的大小变为506,256，比原来小了一些。再看dumpbin列出的调试目录，发现CodeView一行变为：

```
4621E057 cv      67830 00000000 14160 Format: NB11
```

可见, cvpack 将 NB05 格式的调试信息转化为 NB11 格式, 转换后的数据由原来的 0x78F4B 字节变为 0x67830 字节, 缩小了 71451 字节, 这正是 exe 文件所减少的尺寸。

使用 rebase 工具可以将 PE 文件中的集成调试信息提取出来, 保存到一个单独的(DBG 文件中, 我们将在下一节继续介绍。

25.5 DBG 文件

因为 PE 文件中的调试信息只有在调试时才有用, 为了节约内存空间和提高 PE 文件的加载速度, 人们自然地想到了将调试信息保存在单独的文件中, 当调试时再加载这些文件。

25.5.1 从 PE 文件产生 DBG 文件

使用 rebase 工具可以将集成在 PE 文件中的调试信息提取出来放在一个独立的(DBG 文件中。下面通过一个实例来说明。启动一个包含 VC6 环境变量的控制台窗口, 将当前目录切换到 Z7CV 配置的目标目录 (SymOption__Win32_Z7CV), 然后键入如下命令:

```
rebase -x . -b 0x400000 symoption.exe
```

命令执行后, 可以发现目录中新增了一个 symoption.dbg 文件, 其大小为 424,640, 同时 symoption.exe 文件缩小为 82,272。二者的和等于 506,912, 这与原来的 SymOption.exe 文件的大小 (506,256) 基本一致。值得说明的是, VC8 所带的 rebase 工具已经没有这个功能。

使用 dumpbin /HEADERS 观察提取调试信息后的 EXE 文件, 可以看到其中只剩下很少的调试数据:

Type	Size	RVA	Pointer	
misc	110	00000000	14000	Image Name: symoption.dbg
fpo	50	00000000	14110	

其中的 misc 数据, 是用来指示.dbg 文件名的, 它与 NB10 和 RSDS 数据块的功能很类似。

观察 SymOption.exe 的 IMAGE_FILE_HEADER 结构, 可以发现它的 Characters 字段由本来的 10F 改为 30F, 新的标志位 (0x200) 代表着调试信息已经剥离 (Debug information stripped), 即常量 IMAGE_FILE_DEBUG_STRIPPED。

25.5.2 DBG 文件的布局

DBG 文件的结构与 PE 文件非常类似 (见图 25-6), 其头部是一个固定的

IMAGE_SEPARATE_DEBUG_HEADER 结构，其后是零个或多个从 PE 文件中复制过来的 IMAGE_SECTION_HEADER 结构，但节的实际数据并不复制到 DBG 文件中，因此这些结构的作用主要是让调试器了解 PE 文件所包含各个节的概况。

在节描述之后是调试目录表，包含若干个 IMAGE_DEBUG_DIRECTORY 结构，用来描述 DBG 文件的调试数据块。调试数据目录之后便是真正的调试数据块。图 25-6 画出的是 symoption.dbg 文件的布局。

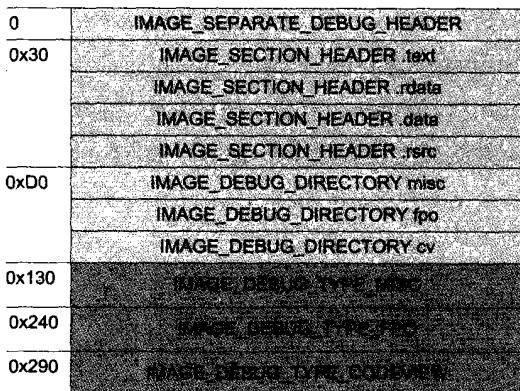


图 25-6 典型 DBG 文件的结构布局

尽管 MSDN 中没有介绍 IMAGE_SEPARATE_DEBUG_HEADER 结构，但是在 WinNT.h 中可以找到这个结构的定义。表 25-5 列出了它的各个字段和在 SymOption(DBG 文件中的取值。

表 25-5 IMAGE_SEPARATE_DEBUG_HEADER 结构

字段	类型	说明
WORD Signature;	4944	签名字符“DI”的 ASCII 码
WORD Flags;	0	标志
WORD Machine;	014C	即 IMAGE_FILE_MACHINE_I386
WORD Characteristics;	030F	属性特征
DWORD TimeDateStamp;	4621E10B	时间戳，Sun Apr 15 12:23:39 2007
DWORD CheckSum;	234A1	校验和
DWORD ImageBase;	400000	内存映像的基地址
DWORD SizeOfImage;	15000	内存映像的长度
DWORD NumberOfSections;	4	节个数
DWORD ExportedNamesSize;	0	输出名称表的长度
DWORD DebugDirectorySize;	54	调试目录表的长度
DWORD SectionAlignment;	1000	节对齐粒度
DWORD Reserved[2];	都为 0	保留未用

*在 SymOption(DBG 文件中的取值，数字全部为十六进制。

根据 NumberOfSections 字段指定的节个数可以找到调试目录的起始位置，然后根据 DebugDirectorySize 字段可以知道调试目录表中有多少个表项。因为每个调试

目录表项的长度是 0x1C，所以根据 DebugDirectorySize 的值 0x54 可以知道 SymOption(DBG) 文件中有 $0x54/0x1C=3$ 个目录项。以下是使用 dumpbin/HEADERS SymOption(DBG) 所列出的结果：

Type	Size	RVA	Pointer	
misc	110 00000000	130		Image Name: Sym..._Z7CV/SymOption.exe
fpo	50 00000000	240		
cv	67830 00000000	290		Format: NB11

可以看到 DBG 文件中的调试数据块与原来 EXE 文件中的数据块完全一样。

VC7 开始不再支持在 PE 文件中存储集成的调试信息，其 rebase 工具也去掉了从 PE 文件中提取调试信息和产生 DBG 文件的功能。但是在调试某些旧的驱动程序和应用程序时，可能还需要使用 DBG 文件格式的调试符号，这也是写作本节的目的。

25.6 PDB 文件

不论是 OBJ 文件、PE 文件还是 DBG 文件，它们存储调试信息的方式都有一个共同点，那就是将同一类信息组织成一个数据块，然后将这个数据块的位置和大小等信息登记在目录表中。例如在图 25-4 所示的 SymOption.EXE 文件中，从偏移 0x14000 处开始就是三个调试数据块，分别为 COFF、MISC 和 FPO 类型的调试信息。这 3 个数据块之间以及块中的子块之间都是相互紧邻的。这样做带来的一个明显问题就是，如果某个数据块需要增大，那么很可能要影响其他数据块的位置，这不仅会导致很多额外的移动工作，而且难以支持并发访问，不能让多个线程同时增删数据块的内容。为了摆脱以上局限，Visual C++ 1.0 引入了 PDB 文件，使用所谓的复合文件技术以数据流的方式来组织调试数据，每个数据流可以包含多个不连续的数据页。

25.6.1 复合文件

所谓复合文件（Compound File）就是使用管理磁盘的方式来组织文件中的不同类型数据，就好像是将包含多个文件的磁盘数据移植到一个文件中，所以有时又被称为包含文件系统的文件。复合文件中的每个子文件通常被称为一个数据流（stream）。使用复合文件的好处有。

- 可以方便地增加和减小每个数据流的大小，改变某个数据流的大小不会影响其他数据流。
- 可以使用类似读写文件的方式来读写数据流。
- 更好的并发支持，不同进程、线程可以访问文件的不同部分，互不干扰。

复合文件是随着 OLE 和 COM 技术的兴起而发展起来的，今天已经广泛应用在各个领域，包括 Office 软件使用的各种文件。复合文件技术的另一个名字是结构化存储

(Structured Storage)。Windows 的复合文件 API 和接口中的 Stg 便来源于此，例如 ReadClassStg、WriteClassStg、IStorage、StgCreateDocfile 等。

25.6.2 PDB 文件布局

PDB 文件是建立在复合文件技术之上的，但它不是典型的复合文档文件格式，因此使用 VC6.0 所附带的 DFView (DocFile Viewer) 工具是不能打开的。

目前使用的 PDB 文件主要有两种版本，一种是 VC6 所使用的 2.00 版本的 PDB 文件 (简称 PDB2)，另一种是 Visual Studio .Net 2002 (VS7) 所引入的 7.00 版本的 PDB 文件 (简称 PDB7)。Visual Studio .Net 2003 (VS7.1) 和 Visual Studio 2005 (VS8) 使用的也是 PDB7。

Sven B. Schreiber 在他的 *Undocumented Windows 2000 Secrets - The Programmers Cookbook* 一书中介绍了 PDB2 的文件格式。因为当时 PDB7 还没有出现，所以至今尚没有关于 PDB7 格式的完整公开描述。本小节将在 Sven 关于 PDB2 介绍的基础上根据笔者的分析统一介绍 PDB2 和 PDB7 的格式，供读者参考。以下内容如不特别说明，适用于两个版本。

图 25-7 画出了一个典型 PDB 文件的数据布局，可以看到，它主要由以下几个部分构成：(1) 一个代表 PDB 文件版本的签名字串。(2) 4 字节的 Magic 代码 (Magic DWORD)。(3) PDB 头结构 (PDB_HEADER)。(4) 紧跟在头结构之后的是用于存放流目录 (Stream Directory) 的根数据流 (Root Stream) 所在的页号数组 (PDB2) 或者这个数组所在的页号 (PDB7)。(5) 页分配表，又称为 bit allocation array，每个二进制位用于标识一个页的使用情况，0 代表已经使用，1 代表空闲。(6) 数据页。(7) 用于存放流目录的根数据流。

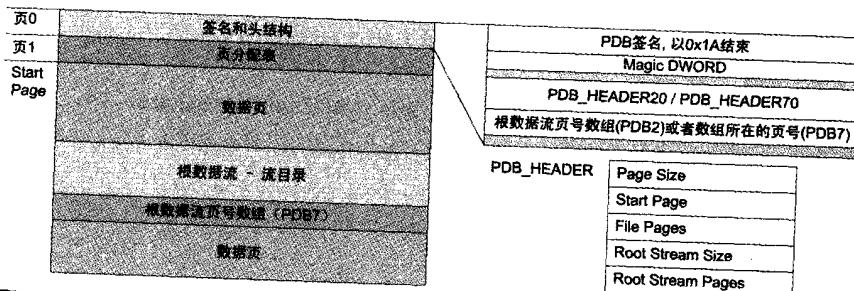


图 25-7 PDB 文件内容布局

图中右侧不同字段间的灰色区域代表为了实现数据对齐而没有使用的部分，通常被填充为全 0 或全 1。下面将分别介绍以上各个部分，首先看一下 PDB 签名。

25.6.3 PDB 签名

PDB 签名是一个可读的字符串，其长度不固定，但是一定以 0x1A 结束。在 ASCII

代码中，0x1A 代表文件结束（End Of File，即 EOF），因此很多处理文本文件的程序都使用 0x1A 作为文件结束的标志。比如在命令行窗口中可以使用 copy con <文件名> 的方法来创建一个文本文件，并将输入到控制台（con）的字符复制到文件中，当输入的内容完成时，应该按 Ctrl+Z 来结束，Ctrl+Z 实际上产生的输入字符就是 0x1A，copy 命令遇到这个字符后便会退出循环关闭文件（这个字符本身不会写到文件中）。类似的，用于显示文件文件内容的 type 命令遇到 0x1A 时，也会认为文件结束，停止继续显示后面的内容。根据这一原理，我们可以使用 type 命令来观察 PDB 文件的签名信息，比如 type symoption.pdb 显示的结果是：

```
Microsoft C/C++ program database 2.00
```

尽管这个 PDB 文件很长，后面还有很多内容，但是 type 命令只显示到字符 0x1A。

上面所显示的字符串就是所有 PDB2 格式的 PDB 文件的签名字串。PDB7 的签名字串是：

```
Microsoft C/C++ MSF 7.00
```

其中 MSF 代表 Microsoft Stream Format，流格式主要是用于存储和播放音视频文件的文件格式，比如微软 DirectMedia 中的音视频文件主要是基于所谓的 ASF（Advanced/Active Streaming Format）格式。PDB 文件的流格式与多媒体文件的流格式是有很大差异的，笔者的推测是 PDB7 在 PDB2 的基础上引入了 ASF 的某些特征，并因此在文件签名中加入了 MSF 来体现这一变化。另一点值得说明的是，签名中的 C/C++ 并不代表这个 PDB 文件一定对应的是 C/C++ 语言所编写的程序，例如，C# 项目产生的 PDB 文件的签名也是这样的。

事实上，VC 编译器所使用的很多其他文件也使用了类似的方法，比如 VC6 中用于存放类视图信息的 ncb（No Compile Browser 的缩写）文件的签名，与 PDB2 的签名是一样的：

```
C:\dig\dbg\author\code\chap25\SymOption>type symoption.ncb
Microsoft C/C++ program database 2.00
```

链接过程所使用的 ILK 文件的签名是 Microsoft Linker Database，即：

```
C:\dig\dbg\author\code\chap25\SymOption\Debug>type symoption.ilk
Microsoft Linker Database
```

25.6.4 Magic 代码

在 PDB 签名之后（0x1A）是 4 字节的 Magic 代码，其内容和含义如表 25-6 所示。

表 25-6 PDB 文件的 Magic 代码

PDB 版本	显示为字节	显示为字符	含义
PDB2	4A 47 00 00	JG	主要设计者 Jan Gray 的姓名缩写
PDB7	44 53 00 00	DS	主要设计者 Dan Spalding 的姓名缩写
PDB8*	52 53 00 00	RS	主要设计者 Richard Shupak 的姓名缩写

*PDB7 的下一个版本，笔者写作此内容时尚未正式发布，可能变化。VS2008 使用的仍然是 PDB7。

除了含蓄的纪念，Magic 代码的实际作用是供 PDB 文件的解析程序来判断这个文件所使用的格式版本，这样就不必从签名字符串中解析版本号了。

25.6.5 PDB_HEADER

在 PDB 的 Magic 代码之后是 PDB 文件的头结构，简称 PDB_HEADER。因为 PDB_HEADER 的起始地址是按 4 字节对齐的，所以在 PDB7 文件中，Magic DWORD 和 PDB_HEADER 之间有一个空闲的字节（0）。在 PDB2 中，因为刚好已经对齐，所以二者是紧邻的。以下是 PDB2 的头结构：

```
typedef struct _PDB_HEADER20
{
    DWORD      dwPageSize;           //页大小
    WORD       wStartPage;          //数据页的起始页号
    WORD       wFilePages;          //以页为单位的文件大小
    PDB_STREAM RootStream;         //流目录表信息
} PDB_HEADER20, *PPDB_HEADER20, **PPPDB_HEADER20;
```

其中 dwPageSize 是 PDB 文件中每个数据页的字节数，复合文件是以页为单位来分配空间的，可能的页大小为 1KB（0x400, 1024 字节）、2KB 和 4KB。wStartPage 用于指定第一个数据页的页号（以 0 开始），典型值为 2，即页 0 为字符串签名和头结构，页 1 为分配表，页 2 开始为数据页。wFilePages 是整个 PDB 文件的总页数，这个值乘以页大小便是文件的总字节数。观察一下 PDB 文件的大小，可以看到都是可以被 1024 所整除的，即都是整 KB 的。

PDB7 的头结构与 PDB2 的很相似，只是将 StartPage 和 FilePages 字段都由 WORD 改为 DWORD。

```
typedef struct _PDB_HEADER70
{
    DWORD      dwPageSize;           //页大小
    DWORD      dwStartPage;          //数据页的起始页号
    DWORD      dwFilePages;          //以页为单位的文件大小
    PDB_STREAM RootStream;         //流目录表信息
} PDB_HEADER70, *PPDB_HEADER70, **PPPDB_HEADER70;
```

图 25-8 分别显示了 PDB7 格式的 HiWorld.PDB 文件（左）和 PDB2 格式的 SymOption.PDB 文件（右）的头部信息（签名、Magic 代码和头结构），这些信息是使用本章的示例程序 PdbFairy 产生的。HiWorld.PDB 是使用 VC2005 编译器产生的（发布版本），SymOption.PDB 是使用 VC6 编译器产生的（项目配置为 ZiPDB）。

Signature: Microsoft C/C ++ MSF 7.00 - DS	Signature: Microsoft C/C ++ program database 200 - JG
Page Size: 0x400 [1024]	Page Size: 0x400 [1024]
File Pages: 0x33b [827]	File Pages: 0x169 [36]
Start Page: 0x2[2]	Start Page: 0x9 [9]
Root Stream Size: 0xbec [3052]	Root Stream Size: 0x540 [1344]
Root Stream Pages Pointer: 0x0[0]	Root Stream Pages Pointer: 0x3a0ef0 [3804912]

图 25-8 PDB7（左）和 PDB2（右）的头结构示例

25.6.6 根数据流——流目录

PDB 文件是以数据流的形式来组织数据的，每个数据流可以看作是复合文件中的一个子文件。类似于文件系统中的文件目录，PDB 文件的流目录用于记录文件中的各个数据流。流目录本身也是以一个数据流来存储的，这个数据流就是所谓的根数据流（Root Stream）。PDB_HEADER 结构的 RootStream 字段用来描述根数据流，它是一个 PDB_STREAM 结构：

```
typedef struct _PDB_STREAM
{
    DWORD dwStreamSize;           //流目录表的字节数
    PWORD pwStreamPages;         //页号数组的地址
} PDB_STREAM, *PPDB_STREAM, **PPPDB_STREAM;
```

其中 dwStreamSize 用来标识数据流的大小，pwStreamPages 指向一个数组，这个数组记录了这个数据流所拥有数据页的页号。但是对于根数据流，其页号数组是以特殊方法存放的，在 PDB2 中，页号数组就跟在 PDB_HEADER 结构之后，数组的每个元素代表一个页面，数组的元素个数可以根据头结构中的 RootStream. dwStreamSize 除以页大小而得到。数组的每个元素是一个 WORD，代表一个页号。比如在图 25-8 右侧所示的 SymOption.PDB 中，目录表的长度为 1344 字节，占两个页面，因此，这个数组共有两个元素分别是 0x161 和 0x162，转化为字节偏移，即 $0x161 * 0x400 = 0x58400$ 开始的 1344 个字节是根数据流。

在 PDB7 中，根数据流的页号数组改为存储在一个单独的页中，因此在 PEB_HEADER70 结构之后只有一个 DWORD，用来代表实际用来存储根数据流页数组的那个页的页号。比如在 HiWorld.PDB 中，这个 DWORD 值是 0x334，转化为字节偏移为 0xcd000。在这个偏移处，我们可以看到 3 个连续的 DWORD 分别为 0x331、0x332 和 0x333，这便是根数据流所在的 3 个页面的页号，这与头结构中的根数据流的长度 3052 字节（不满 3 个页面）也正好吻合。

根数据流的起始处是一个 PDB_ROOT 结构，用来标识文件中的文件流的个数，其定义如下：

```
typedef struct _PDB_ROOT
{
    WORD     wCount;          //文件流的总数
    WORD     wReserved;        //保留未用，等于 0
} PDB_ROOT, *PPDB_ROOT, **PPPDB_ROOT;
```

PDB_ROOT 结构之后便是对每个数据流的描述，而且描述的第一个数据流通常就是根数据流。描述数据流的方式因为 PDB 的版本不同而不同，在 PDB2 中，使用的是 PDB_STREAM 结构，在 PDB7 中有所变化，其细节没有公开，本书不再继续讨论。

25.6.7 页分配表

页分配表用来标识文件中数据页的使用情况，它的每个二进制位代表其所对应的

页是否已经使用，0 代表使用，1 代表空闲。页分配表通常位于 PDB 文件的第二个页（即页 1）。

页分配表所占的页数为头结构中的 StartPage 字段的值减去 1。比如上面的两个 PDB 文件的 StartPage 的值都是 2，那么页分配表的长度便是一个页，即 1024 字节，等于 $1024 \times 8 = 8192$ 个二进制位，这意味着这样的 PDB 文件可能包含的最多页数是 8192 个，或者说使用这种参数组合的 PDB 文件的最大可能为 8192KB。这对于大多数项目都够了，如果需要更大，那么可以通过调整页大小或者加长页分配表的长度（改变 StartPage 的值）来实现。

25.6.8 访问 PDB 文件的方式

微软没有公开 PDB 文件的详细格式，但提供了以下两种方式允许其他软件开发商间接访问 PDB 文件。

第一，使用 DBGHELP API。DBGHELP 是微软提供的用于支持调试的一套 API，包括处理映像文件（ImageXXX）、读取调试符号（SymXXX）和处理故障转储文件（MiniDumpXXX）等。所有 DbgHelp API 都是由 DbgHelp.dll 模块所输出的。Windows 的 system32 目录中预装了 DbgHelp.dll。但这个文件通常没有 WinDBG 所包含的版本高。

第二，使用 Visual Studio 所附带的 DIA（Microsoft Debug Interface Access）SDK。与 DBGHELP 使用的函数形式不同，DIA 使用了 COM 技术并以接口的方式提供服务。重要的接口有 IDiaDataSource、IDiaSymbol 和 IDiaEnumXXX 等。本书附带的 SymView 工具（见图 25-9）就是使用 DIA SDK 所开发的。

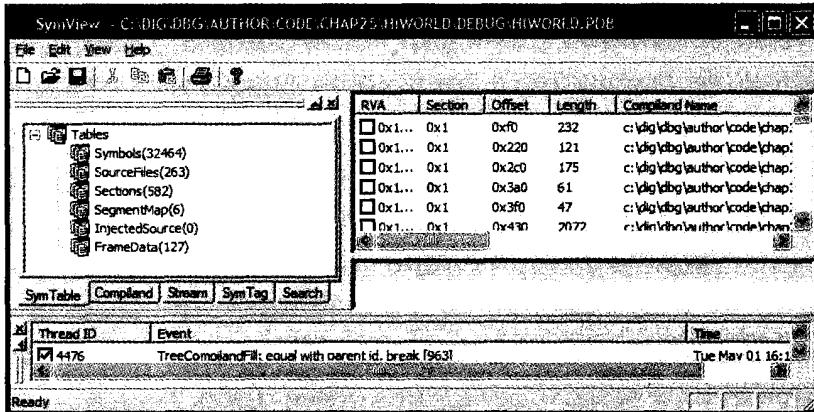


图 25-9 观察符号文件的 SymView 工具

除了以上公开的方式，Visual Studio 开发工具使用名为 MSPDBx0.DLL 的模块来产生和处理 PDB 文件，其中的 x 代表 Visual Studio 的主版本号，如 VS6 使用的是 MSPDB60.DLL，VS8 使用的是 MSPDB80.DLL。

25.6.9 PDB 文件的产生过程

默认情况下，使用 VC 编译器成功构建一个 C/C++ 程序后，我们可以发现在与可执行文件相同的位置中，有两个 PDB 文件，一个是与可执行文件具有相同主文件名的.PDB 文件，我们使用<project>.PDB 来表示它；另一个是 VCx0.PDB，其中 x 是 VC 编译器的主版本号，比如 VC6 产生的是 VC60.PDB，VC8 产生的 VC80.PDB。概言之，VCx0.PDB 是在编译期间由编译器产生的，<project>.PDB 是在链接阶段由链接程序（link.exe）产生的（参见下文）。

编译器在编译每个源文件时，在将调试信息以 COFF 格式保存在 obj 文件中的同时，会把类型有关的调试信息存储到 VCx0.PDB 中。这样做的一个好处是对于多个文件都使用的类型，例如定义在公共的头文件（如 windows.h）的类型，只需要存储一份。

在链接时，链接程序会根据命令行中的选项产生<project>.PDB。例如以下是 VC6 的集成环境链接 PdbFairy 程序时使用的命令行：

```
link.exe /nologo /subsystem:windows /incremental:yes /pdb:"Debug/PdbFairy.pdb"
/debug /machine:I386 /out:"Debug/PdbFairy.exe"
/pdbtype:sept .\Debug\PdbFairy.obj .\Debug\PdbFairyDlg.obj .\Debug\PdbMaster
.obj .\Debug\StdAfx.obj .\Debug\PdbFairy.res
```

其中 /pdb:"Debug/PdbFairy.pdb" 指定了要产生的 PDB 文件的名称，/pdbtype:sept 的含义是分散存储调试信息，也就是不要将 VCx0.PDB 文件的调试信息集成到 <project>.PDB 文件中。如果指定/PDBTYPE:CON，那么链接器会将 VCx0.PDB 文件的调试信息集成到 <project>.PDB 文件中。

以下是 VC8 执行链接程序的典型命令行：

```
link.exe @c:\dig\dbg\author\code\chap25\Vc8Win32\Debug\RSP00000459205488.rsp
/NOLOGO /ERRORREPORT:PROMPT
```

其中的.rsp 是一个用于存储参数的临时文件，使用记事本可以观察到其内容与上面介绍的命令行非常类似。

图 25-10 显示了产生 PDB 文件的典型过程，下一节我们将介绍如何通过编译和链接选项来定制 PDB 文件的内容和产生方式。

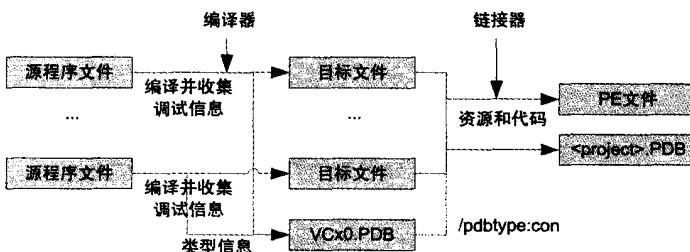


图 25-10 PDB 文件的产生过程

VC8 使用一个专门的进程来完成调试符号有关的任务，这个进程的 EXE 文件是

MSPDBSRV.EXE，因此我们将其称为 PDB 服务进程。当开始构建一个项目时，CL.EXE 会检测 PDB 服务进程是否已经运行，如果还没有，那么它会使用如下命令行启动 PDB 服务进程：

```
mspdbsrv.exe -start -spawn
```

而后，编译器和链接器会通过 RPC 调用把符号有关的任务提交给 PDB 服务进程去完成。因此，对于 VC8，PDB 文件都是 PDB 服务进程所创建的。以构建 HiWorld 程序为例，在编译过程中，编译器会调用 PDB 服务进程，后者会创建 VC80.PDB。当开始链接时，链接器（Link.EXE）会调用 PDB 服务进程，后者会创建 HiWorld.PDB。如果项目中指定了/PDBSTRIPPED:Private.PDB 选项，那么 PDB 服务进程会创建这个指定的 PDB 文件。

25.7 有关的编译和链接选项

本节我们将介绍与调试符号有关的编译和链接选项，分析不同选项所产生的符号文件的异同。

25.7.1 控制调试信息的编译选项

调试信息是由编译器在编译源文件的过程中收集起来的，VC 编译器设计了以下 4 个编译开关允许用户控制编译器收集信息的方式和内容。

/Z7 产生与 MSC (7.0) 相兼容的调试信息，包括变量的名称和类型、函数信息和行号等。在编译阶段，编译器把这些调试信息存储在.obj 文件中，链接器可以将这些信息连接到 PE 文件或单独的文件中。

/Zd 只产生行号信息和全局及外部符号，与/Z7 类似，编译器也是将这些信息放在.obj 文件中。

/Zi 生成符合 PDB 格式要求的调试信息，在编译阶段这些信息主要存储在.obj 和 VCx0.PDB 文件中。在链接时，链接器会根据链接选项将这些信息链接到 PE 文件中或单独的 PDB 文件中（见下文关于/PDB 选项的说明）。

/ZI 除了产生类似 Zi 的信息外，还生成 EnC 所需要的信息。这个选项只适用于 x86 平台。

可以在项目属性（VC7 开始）或项目设置（VC6）对话框中设置以上选项，位于 C++ 页面 General 栏目（Category）的 Debug info 子项中。

25.7.2 控制调试信息的链接选项

如下几个链接选项用来控制链接器生成调试文件的方式。

/DEBUG 这是控制链接器是否产生调试信息的一个根本选项，如果指定这个选项则产生调试信息。在调试版本的项目配置中，默认会包含这个选项。在发布版本的配置中，VC6 的默认配置不包含这个选项，但是 VC8 的包含。在项目属性对话框的链接页中选择 Generate Debug Info 实际上就是增加这个选项。

/debugtype:{CV|COFF|BOTH} 选择集成在 PE 文件 (EXE 或 DLL) 中调试信息的存储格式，CV 代表 CodeView 格式、BOTH 代表两种格式都需要。从 VC7 (Visual Studio .Net 2002) 开始不再支持在 PE 文件中集成存储 COFF 格式和非 NB10 类型的 CV 格式调试信息，所以这个链接选项也从 VC7 开始不再被支持。如果没有指定 /DEBUG 选项，那么这个选项会被忽略。

/pdb:{none|文件名} 如果指定文件名，链接器便会产生相应名称的 PDB 文件，如果指定 none，那么则不产生 PDB 文件。默认设置是使用与项目同名的 PDB 文件，例如 /pdb:"Debug/SymOption.pdb"。这个选项的优先级高于 /debugtype，这意味着，只要这里没有指定 none，那么编译器就会产生单独的 PDB 文件，/debugtype 选项会被忽略。因为 VC7 开始只支持将调试信息放在 PDB 文件中，所以不再支持 /PDB:NONE。

/PDBTYPE:{CON[SOLIDATE]|SEPT[YPES]} 这个选项告诉链接器是否将 VCx0.PDB 中的调试信息合并 (consolidate) 到一个 PDB 文件中 (/pdb:所指定的)。合并到一个文件的好处是调试器只需要加载一个符号文件，不需要再寻找和加载 VCx0.PDB。对于 WinDBG 这样的调试器，它是不支持自动加载一个模块对应的 VCx0.PDB 的，因此当我们使用 WinDBG 调试 VC6 产生的模块时，默认总是不能观察模块中的类型信息，即使是调试版本。对于 VC6 集成的调试器，如果它找不到合适的 VCx0.PDB，那么它便会显示图 25-11 所示的对话框。另外，因为不同模块的 VCx0.PDB 尽管内容完全不同，但是它们却是同名的，这为备份和管理大型项目的符号文件带来了不便。考虑以上因素，从 VC7 开始的链接器总是将 VCx0.PDB 文件中的调试信息合并到一个单一的 <project>.PDB 中，不再支持这个选项。对于 VC6，只有当编译选项中指定了 /Zi 或者 /ZI 选项后，/PDBTYPE 选项才有意义。

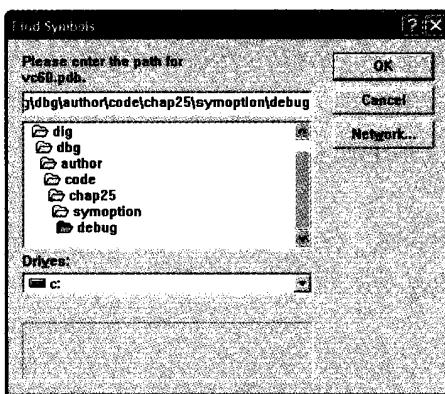


图 25-11 VC6 调试器寻找 Vc60.PDB

/PDBSTRIPPED:second_pdb_file_name 使用这个选项可以同时产生一个公共 PDB 文件，公共 PDB 文件省略了很多调试信息，有利于防止因为发布 PDB 文件而泄漏相关软件的技术秘密。

在项目属性（VC7 开始）或项目设置（VC6）对话框中可以设置以上选项，位于链接页面的调试栏目中。

25.7.3 不同链接和编译选项的比较

下面我们通过一个实际项目来分析和比较以上各个编译和链接选项的效果。因为从 VC7 开始很多选项不再被支持，所以我们以 VC6 为例进行讨论。VC6 可以产生 3 种格式的调试信息：COFF、CV 和 PDB。通过设置编译选项和链接选项可以选择产生其中的一种或多种。为了比较不同编译选项和链接选项的效果，我们使用 VC6 (SP5) 创建了一个 Win32 程序，取名为 SymOption。在这个项目中，除了两个默认的项目配置（Configuration）Debug 和 Release 外，我们还创建了若干个新的配置，分别采用了不同的编译和链接选项。表 25-7 列出了不同项目配置的名称（第 1 列），每种配置中使用的编译选项（第 2 列）和链接选项（第 3 列），以及产生的可执行文件（第 4 列）与符号文件（第 5、6 列）。

表 25-7 调试符号有关的编译选项和链接选项比较（VC6）

项目配置 名称	编译 选项	链接选项（所有配置中都 含有/debug开关）	SymOption. exe	SymOption. PDB	VS60.PDB
Z7COFF	/Z7	/debugtype:coff /pdb:none	121,392	不存在	不存在
Z7CV	/Z7	/debugtype:cv /pdb:none	506,256	不存在	不存在
Z7PDB	/Z7	/pdptype:sept /pdb:"x.pdb"*	110,686	680,960	不存在
ZiCOFF	/Zi	/debugtype:coff /pdb:none	121,628	不存在	208,896
ZiCV	/Zi	/debugtype:cv /pdb:none	499,496	不存在	208,896
ZiPDB	/Zi	/pdptype:sept /pdb:"x.pdb"*	110,686	369,664	208,896
Debug	/ZI	/pdptype:sept /pdb:"x.pdb"*	176,204	369,664	208,896
Release	无	/pdptype:sept /pdb:"x.pdb"*	45,134	115,712	不存在

*为排版方便作了省略，完整设置为/pdb：“<项目配置的目标文件目录>/ SymOption.pdb”，如/pdb：“SymOption_Win32_Z7PDB/SymOption.pdb”

我们先来分析 EXE 文件的差异。Z7CV 和 ZiCV 两种配置产生的 EXE 文件相对于其他配置大了很多，这是因为其中嵌入了 CodeView 格式的调试信息。使用 dumpbin 工具观察 PE 文件的内容可以看到其中的调试信息。具体步骤是，启动 VC6 的编译命令行，切换到 Z7CV 项目的输出目录（code\chap25\symoption\SymOption_Win32_Z7CV），然后键入 dumpbin symoption.exe /headers。而后在输出的结果中找到 Debug Directories 小节：

```
Debug Directories
Type      Size     RVA  Pointer
----- -----
misc      110 00000000  14000    Image Name: Sym..._Z7CV/SymOption.exe
```

Type	Size	RVA	Pointer
fpo	50 00000000	14110	
cv	67830 00000000	14160	Format: NB11

上面的显示表示这个 EXE 文件中包含了 3 个记录调试信息的数据块，分别为 misc、FPO 和 CV 格式的调试符号。第 2 列是每个块的大小，其中的数字都是十六进制的，因此，这 3 个块的长度分别为 $110 = 272$ 字节， 80 字节和 $0x67830 = 423,984$ 字节。可见 CV 格式的调试信息的长度占了整个文件（506,256）的绝大部分。对于 ZiCV 配置，相应的数据如下：

Type	Size	RVA	Pointer
misc	110 00000000	14000	Image Name: Sym..._ZiCV/SymOption.exe
fpo	50 00000000	14110	
cv	65DC8 00000000	14160	Format: NB11

可见仍是 3 个调试数据块，但第三个数据块是 CodeView 格式的，其大小略有变化，减少为 $67830 - 65DC8 = 0x1a68 = 6760$ 字节。这正是两个 EXE 文件的大小差异 $506,256 - 499,496 = 6760$ 。

对于 Z7COFF 配置所产生的 EXE 文件，其调试信息目录如下：

Type	Size	RVA	Pointer
coff	98D0 00000000	14000	
misc	110 00000000	1D8D0	Image Name: Sym..._Z7COFF/SymOption.exe
fpo	50 00000000	1D9E0	

COFF 数据块的长度为 $0x98D0 = 39120$ 字节。ZiCOFF 配置的 COFF 数据块的长度为 $99BA = 39354$ 字节，二者的差 234 与这两个 EXE 文件的大小差 ($121,628 - 121,392 = 236$) 也基本一致。

对于 Z7PDB 和 ZiPDB 两个配置，它们的 EXE 文件的大小相同。使用 dumpbin 观察，看到的 Debug Directories 信息如下 (Z7PDB)：

Type	Size	RVA	Pointer
cv	5E 00000000	1B000	Format: NB10, 460f8ec5, 1, C:\...\SymOption.pdb

也就是说，这个 EXE 文件中只包含 $0x5E$ (94) 个字节的 CV 数据，它的格式就是前面我们介绍过的 NB10 格式。观察其他几个附带有 PDB 文件的 EXE 文件，也可以看到有 NB10 格式的 CV 数据。

因为默认的 Debug 配置支持 EnC 功能，所以其 EXE 文件要比不带 EnC 支持的 (ZiPDB) 大一些。使用 dumpbin 比较二者的概要信息，发现不带 EnC 功能的 EXE 文件的.text 节的大小为 $0x11000$ ，带 EnC 的大小为 $0x21000$ ，二者的差 $0x21000 - 0x11000 = 0x10000 = 65536$ ，这与两个 EXE 文件的整个文件大小之差 ($176,204 - 110,686 = 65518$) 基本相当。使用工具直接观察带 EnC 支持的 EXE 文件的.text 节，可以看到有很多空间都是 $0xCC$ 或者 0 ，这显然是留给 EnC 功能使用的。

下面再分析一下 PDB 文件的差异。使用本书附带的 SymView 工具打开 3 个配置

所生成的 SymOption.PDB 文件。表 25-8 列出了每个 PDB 文件的几个主要数据表的表项数。我们将在下一节介绍每个数据表的具体内容和用途，在这里大家可以把表项数理解为数据库中的记录（record）数。

表 25-8 不同项目配置的 PDB 文件比较

	符号个数	源文件个数	帧数	模块数	行数
Z7PDB	21267	62	704	7	5
ZiPDB	6830	62	704	7	5
Debug	6922	62	708	7	5
Release	576	0	446	5	69

从上表可以看出，Z7PDB 版本的符号数最多，这也是它的文件最大的一个直接原因。尽管项目配置中包含了 /pdbtype:sept，但是观察 Z7PDB 的目标文件目录会发现不存在 vc60.pdb，这意味着 Z7PDB 文件集成了其他配置放在 VC60.PDB 中的类型符号信息。通过 SymView 工具可以看到 VC60.PDB 中有 9773 个符号。发布版本的 PDB 包含的符号数最少，不包含源文件描述和行号信息，因此其文件也最小。因为发布版本中起用的优化措施会对很多函数作 FPO，所以发布版本中包含的帧数据最多。

前面各节逐一对常见格式的符号文件做了比较全面的分析和归纳，从下一节开始我们将深入分析 PDB 文件的逻辑结构和 PDB 文件中所包含的调试符号。

25.8 PDB 文件中的数据表

PDB 文件是以表格的形式（关系数据库）来存储调试信息的，每条信息占据表格的一行，通过类型字段来区分不同种类的信息。一个典型的 PDB 文件中通常包含很多个数据表，用来存放不同类型的数据。通过 DIA SDK 的 IDiaEnumTables 接口可以枚举出 PDB 文件中的各个表。举例来说，以下是一个典型 PDB 文件内部所包含的 6 种数据表：

Symbols(29863)	// 符号表
SourceFiles(261)	// 源文件表
Sections(326)	// 节贡献表
SegmentMap(5)	// 段信息表
InjectedSource(13)	// 注入代码表
FrameData(53)	// 帧数据表

括号中的数字是示例文件中每个表所包含的数据行数。下面我们将分别介绍每一种数据表的作用和其中存储的内容。为了易于理解，我们将以 VC8 产生的 HiWorld.PDB 为例，如果讨论的内容与构建选项有关，我们将使用 HiWorld_REL.PDB 来指代发布版本，HiWorld_DBG.PDB 来指代调试版本。

25.8.1 符号表

符号文件的首要用途是支持调试，所以其中主要包含的是供调试器实现跟踪和分

析所使用的各种程序标志 (Identifier)，即通常所说的符号。符号表中的符号又被进一步分成很多种类型，分别用来描述不同类型的调试信息。每一种类型由一个类型 ID 来标识，称为 SymTag。所有的 SymTag 都定义在头文件 CVCONST.H 中的 SymTagEnum 枚举类型中。CVCONST 中的 CV 是 CodeView 的缩写。DIA SDK 中包含了 CVCONST.H 头文件，默认位于 VS 的 DIA SDK 目录中，如 c:\Program Files\Microsoft Visual Studio 8\Dia SDK\include。SymTagEnum 定义了 32 个常量，其中 1~30 用来代表 30 种调试符号，0 (SymTagName) 用来表示无类型的信息或者在搜索时用来通配所有类型，31 (SymTagMax) 用来记录这组常量的上限。表 25-9 列出了所有 SymTagEnum 常量和该类型符号的描述对象。

表 25-9 SymTagEnum 定义的调试符号类型

枚举常量	值	描述对象
SymTagName	0	不属于以下任何类型的信息，在搜索时表示所有类型
SymTagExe	1	可执行文件 (EXE、DLL、SYS 等) 的信息
SymTagCompiland	2	每个 Compiland 的概要信息
SymTagCompilandDetails	3	每个 Compiland 的详细信息
SymTagCompilandEnv	4	每个 Compiland 的环境信息
SymTagFunction	5	函数
SymTagBlock	6	内嵌的块
SymTagData	7	静态数据，常量，参数和各种变量
SymTagAnnotation	8	标注，参见 SAL
SymTagLabel	9	跳转标记
SymTagPublicSymbol	10	公共符号
SymTagUDT	11	用户定义的类型 (User Defined Type)
SymTagEnum	12	枚举类型
SymTagFunctionType	13	函数的原型信息，包括调用规范、返回值类型等
SymTagPointerType	14	指针类型
SymTagArrayType	15	数组类型
SymTagBaseType	16	基础类型，如 int、char 等
SymTagTypedef	17	通过 typedef 定义的类型别名
SymTagBaseClass	18	基类
SymTagFriend	19	友元 (friend) 信息
SymTagFunctionArgType	20	函数参数
SymTagFuncDebugStart	21	函数序言 (prolog) 代码的起始位置
SymTagFuncDebugEnd	22	函数序言 (prolog) 代码的结束位置
SymTagUsingNamespace	23	名称空间
SymTagVTableShape	24	类或接口的虚拟方法表信息
SymTagVTable	25	指向类或接口的虚拟方法表的指针
SymTagCustom	26	定制的内容
SymTagThunk	27	Thunk 代码
SymTagCustomType	28	编译器定制的类型
SymTagManagedType	29	使用元数据 (metadata) 描述的托管类型
SymTagDimension	30	FORTRAN 语言中的多维数组

可以通过 IDiaEnumSymbols 接口来枚举符号对象，然后通过 IDiaSymbol 接口来

读取每个符号的属性信息。由于不同类型符号的属性差异较大，所以 `IDiaSymbol` 接口定义了几十个方法，很多方法是只对某些类型有效的。大多数符号都具有如下几个属性。

- 索引 ID，PDB 文件会为每个符号分配一个整数 ID，用作它的唯一标识。通过 `IDiaSymbol` 的 `get_symIndexId` 方法可以读取符号的索引 ID。
- SymTag，符号的类型，其值就是表 25-9 中的常量之一。
- 名称，大多数符号还都有一个可读的字符类型的名称。`IDiaSymbol` 的 `get_name` 方法用于读取符号的名称。
- 父词条 ID，用来标识一个符号在词典编纂意义上的父符号，其值为父符号的索引 ID。`IDiaSymbol` 的 `get_lexicalParentId` 方法用于读取父词条 ID。

25.8.2 源文件表

源文件表中包含了编译可执行文件时所使用源文件的基本信息。这里说的源文件包括头文件、资源文件、预编译头文件，以及 CRT 库文件（Lib）的源文件和头文件等。以 `HiWorld.PDB` 符号文件为例，它的源文件表中包含了两百多条记录，每条记录描述一个文件，可分为如下几类。

- CPP 文件和头文件。
- 预编译头文件，如 `c:\..\hiworld\release\hiworld.pch`。
- 资源文件，列出的为编译后的 RES 文件，如 `c:\..\release\HiWorld.res`。
- 通过 `include` 所包含的头文件，如 `c:\program files\microsoft visual studio 8\vc\platformsdk\include\windef.h`。
- 库文件（Lib）的源程序文件，例如 `f:\rtm\vctools\crt_bld\self_x86\crt\src\ctype.h`，它是 `MSVCRT.lib` 文件的 `build\intel\ dll_obj\ newmode.obj` 所使用的源文件。

可以通过 `IDiaEnumSourceFiles` 来枚举 PDB 文件中的源文件对象，然后使用 `IDiaSourceFile` 接口获取每个源文件的属性信息，包括文件 ID，文件的完全路径和文件名，以及与这个文件有关的 Compiland（下一节介绍）。

25.8.3 节贡献表

一个映像文件是由很多个节组成的，每个节的数据又来自于不同的编译素材（Compiland）。节贡献表（Section Contribution Table）描述了构成节的各个数据块（称为 Section Contrib）的信息，包括：

- 数据块所属的节，RVA、偏移、长度。
- 数据块的来源，即来自于哪个编译素材（Compiland）。
- 这个块的内容特征，如是否包含代码，是否可以执行，是否包含初始化的（未初

始化) 的数据等。

可以通过 `IDiaEnumSectionContribs` 来枚举节贡献表中的各个 `Section Contrib` 对象，然后使用 `IDiaSectionContrib` 接口获取它的属性。

25.8.4 段信息表

当一个可执行映像被加载到内存中执行时，它的各个节 (Section) 会被映射到所在进程空间的某个段 (Segment)，段信息表便描述了这种映射关系。表 25-10 列出了 `HiWorld_REL.PDB` 的段信息表中的所有记录。

表 25-10 `HiWorld_REL.PDB` 的段信息表

ID	偏移	长度	页	写	执行	节号	RVA	VA
1	0x0	3342	1	0	1	1	0x1000	0x1000
2	0x0	2508	1	0	0	2	0x2000	0x2000
3	0x0	1488	1	1	0	3	0x3000	0x3000
4	0x0	48752	1	0	0	4	0x4000	0x4000
0	0x0	-1	0	0	0	5	0x0	0x0

其中 `RVA` 是 `Relative Virtual Address` 的缩写，即这个节的起始位置相对于模块起始地址的偏移值。`VA` 是 `Virtual Address` 的缩写，即节的起始位置在进程空间中的虚拟地址。因为 `SymView` 工具在加载符号文件时指定的基地址为 0，所以这里的 `RVA` 值和 `VA` 值相等，在实际调试程序时，不会是这样的。

可以通过 `IDiaEnumSegments` 来枚举段映射表中的各个段映射对象，然后使用 `IDiaSegment` 接口获取它的属性。

25.8.5 注入源代码表

编译器在编译某些内建函数 (Intrinsic) 时，会将这些函数所对应的代码动态注入 (`inject`) 到使用这些函数的源代码中，然后再按普通的方法进一步编译。

为了在调试时可以跟踪编译器注入的源代码，在生成符号文件时，编译器将这些代码插入到符号文件中。注入源代码表 (Injected Source Table) 便是用来存放注射代码的，表的每一行存放一段代码。

例如，当我们在 `HiWorld` 项目中的 `CBaseClass` 类中加入一个用 `_event` 关键字描述的事件函数后，便可在编译后生成的 PDB 文件中看到注入源代码表所包含的数据行不再为 0 了。

```
class CBaseClass
{
public:
    _event void f();
};
```

这种定义事件的方法使用的是 Visual C++ .NET 引入的新功能，称为 C++ 属性 (C++

Attributes)。使用 C++ 属性编程可以提高开发 COM 和.NET 程序的速度。在 C/C++ 编译选项中加入/Fx 开关 (VC2005 项目属性>C/C++ Output Files>Extend Attributed Source>选择 Yes) 后，便可以看到包含注入代码和原始代码混合在一起的 MRG 文件 (其文件名中包含.mrg)，如 baseclass.mrg.cpp 和 baseclass.mrg.h (位于 HiWorld 项目目录)。

可以通过 IDiaEnumInjectedSources 来枚举注入源代码表中的各个注入代码段对象，然后使用 IDiaInjectedSource 接口获取它的属性。每个代码段有以下几个属性 (列)。

- 源代码文件名，即被注入的源程序文件名，如.\BaseClass.cpp。
- 虚拟文件名，即注入代码段的虚拟名称，如*c:\...\hiworld\debug\baseclass.inj:3。:3 表示这是向 BaseClass.cpp 注入的第 3 段代码。
- 目标文件名，如 c:\...\hiworld\debug\baseclass.obj。
- 注入源代码的内容，清单 25-6 给出了一段注入源代码的例子。
- 用来校验源代码数据的 CRC 值。

清单 25-6 注入源代码示例

```

1 //+++ Start Injected Code For Attribute 'event'
2 #injected_line 15 "c:\\dig\\...\\chap25\\hiworld\\baseclass.h"
3 inline void CBaseClass::f()
4 {
5     __EventingCS.Lock();
6     __try
7     {
8         __eventNode_CBaseClass_f* node = __eventHandlerList_CBaseClass_f;
9         for (; node != 0; node = node->next) {
10             node->__invoke();
11         }
12     }
13     __finally
14     {
15         __EventingCS.Unlock();
16     }
17 }
18 }
19 //--- End Injected Code For Attribute 'event'

```

以上代码是通过 SymView 工具从 PDB 文件中读取出来的，在左侧的控件树中选择 InjectSource，然后在右侧上部的列表中选择要观察的注入源文件，它的源代码便会显示在右侧下方的列表中。

25.8.6 帧数据表

在第 22 章中我们详细介绍了栈帧 (有时简称帧) 的概念，以及它在局部变量分配和函数调用方面所起的作用。栈帧信息是调试器工作的一个重要依据，对生成函数调用序列，显示函数参数和局部变量都起着重要作用。

PDB 文件中的帧数据表是专门用来存储帧信息的。表 25-11 列出了 PDB 文件中包含的帧信息类型和描述对象。

表 25-11 栈帧信息的类型和描述对象

类型 (枚举变量)	值	描述对象
FrameTypeFPO	0	使用了 FPO (Frame pointer omitted) 的函数帧信息
FrameTypeTrap	1	用来保存中断和异常现场的陷阱帧
FrameTypeTSS	2	用来保存任务切换现场的 TSS 帧
FrameTypeStandard	3	标准的函数帧, EBP 值是帧的基地址, 又称为 FRAME_NONFPO
FrameTypeFrameData	4	VC6 之后引入的帧数据格式
FrameTypeUnknown	-1	未知类型

可以通过 `IDiaEnumFrameData` 来枚举帧数据表中的各个帧对象, 然后使用 `IDiaFrameData` 接口获取它的属性。

25.9 本章总结

深入理解调试符号对于理解调试器的工作原理和提高调试能力有着重要意义, 本章以较大的篇幅详细介绍了调试符号的存储格式, 调试符号文件的种类、产生过程和内容布局, 并深入分析了常用的调试符号。

参考文献

1. C++ Name Mangling/Demangling. <http://www.kegel.com/mangle.html>
2. Description of the .PDB files and of the .DBG files (KB121366). Microsoft Corporation
3. How To Call 16-bit Code from 32-bit Code Under Windows 95, Windows 98, and Windows Me (KB155763). Microsoft Corporation
4. Shaun Miller. Generating and Deploying Debug Symbols with Microsoft Visual C++ 6.0. Microsoft Corporation
5. Microsoft Portable Executable and Common Object File Format Specification. Microsoft Corporation
6. Product Changes: Visual C++ .NET 2002. Microsoft Corporation
7. Breaking Changes in the Visual C++ 2005 Compiler. Microsoft Corporation
8. Sven B. Schreiber. Undocumented Windows 2000 Secrets: The Programmers Cookbook. Addison Wesley, 2001

第 5 篇

可调试性



前面 3 篇我们分别探讨了 CPU、操作系统和编译器对调试的支持，它们分别是计算机系统的硬件核心、软件核心和生产软件的最重要工具。可以说，来自这三方面的支持是软件调试技术的三大根基。在对这三个方面的调试支持有了比较深的理解后，本篇和下一篇将讨论影响软件调试效率的另外两个主要因素：被调试软件的可调试性和调试工具。

概言之，软件的可调试性（Debuggability）就是指软件容易被调试的程度，或者说当这个软件发生故障时，调试人员可以多方便或多快地寻找到问题的根源。如果上升到成本和费用的层次，那么可调试性就是调试这个软件所需成本的倒数，调试代价越高，那么可调试性越差，代价越低，可调试性就越好。在第 26 章中我们将详细讨论可调试性的内涵、外延和衡量标准。

如果把软件故障比喻成灾害（不为过吧），则提高软件的可调试性就是增强软件抵

御灾害的能力。在如何应对灾害方面，古人留给我们一个非常好的成语：未雨绸缪。它来源于我国最早的诗歌总集《诗经》中的一首著名的寓言诗《鵲巢（chī xiāo）》，原诗如下：

鵲巢

鵲巢鵲巢，既取我子，无毁我室。恩斯勤斯，鬻（yù）子之闵斯。

迨天之未阴雨，彻彼桑土，绸缪（móu）牖（yǒu）户。今女下民，或敢侮予？

予手拮据，予所捋荼。予所蓄租，予口卒瘞，曰予未有室家。

予羽谯谯，予尾翛翛，予室翬翬。风雨所漂摇，予维音哓哓！

这首诗以一只被猫头鹰（鵲）夺去幼鸟的母鸟的口气叙述了它重建鸟巢时的所想和所感：“猫头鹰夺取了我的幼鸟，再不能毁坏我的鸟巢。我辛苦养育（鬻）儿女，已经病（闵）了。趁着天还未阴雨，我啄取桑根，缚紧（绸缪）巢的缝隙（牖本指窗，户指门）……”

未雨绸缪的道理对软件开发也是适用的，它告诉我们应该在设计软件时就为调试这个软件做好准备，否则等故障出现了，就会措手不及，甚至为时晚矣！这种在设计软件时就考虑软件调试的思想经常被称为 Design For Debug (DFD) 或 Design Better, Debug Faster。

对于较大型的软件，要做到 DFD 并不单单是某个角色（比如程序员）的事情，它需要整个软件团队所有成员的共同努力。这是提高软件可调试性的另一个重要原则，那就是要把提高可调试性纳入到软件工程的每个环节当中，使其成为所有团队成员的目标。我们将在第 27 章详细讨论如何在软件工程中贯穿可调试性设计，提高整个软件项目的可调试性。

可调试性概览

本章将先介绍软件可调试性的概念（26.1 节）和意义（26.2 节），然后讨论实现可调试性的基本原则（26.3 节）。第 26.4 节将从反面讨论不可调试代码对可调试性的危害，第 26.5 节将分析 Windows 系统中所包含的可调试设计，第 26.6 节将探讨在实现可调试性时应该注意的问题。

26.1 简介

在计算机硬件领域，人们很早就开始重视系统的可调试性。以著名的 UNIVAC 计算机为例，系统内部有专门的错误检测电路，控制面板上有多个提示错误的指示灯，操作手册有对这些设施的详细介绍。通过这些设施，人们可以很方便地了解内部电路或元器件的状态，如果出现故障，也可以比较迅速地找到原因并排除故障。人们把这些支持检修和调试的设计统称为 Design For Test 或 Design For Testability，简称 DFT。这里的测试一词（Test）显然包含了调试（Debug）的含义，于是逐渐地有人开始使用 Design For Debug 或 Design For Debuggability（DFD）来称呼可调试设计。

随着大规模集成电路的出现，人们开始更加重视可调试性。很多芯片设计厂商开始在芯片中加入支持调试的机制，并着手建立行业标准，以便让多个芯片组成的系统也具有很好的可调试性。1990 年，多家厂商联合制定了 JTAG 标准得到 IEEE 的批准，并从此得到业界的广泛接受（详见 7.1.2 节）。

2005 年 6 月 13 日，Design-for-Debug Consortium（DFD 联盟）(<http://designfordebug.org>) 成立，其宗旨就是要解决集成芯片领域的调试问题（silicon debug）。DFD 联盟的成员包括 Corelis, Inc.、DAFCA, Inc.、First Silicon Solutions (FS2)、Intellitech Corporation、JTAG Technologies、Fidel Muradali (consultant) 和 Novas Software, Inc. 等。

从以上介绍可以看到，在电子、电路和芯片设计生产等领域，可调试设计已经发展多年，并且得到了广泛的重视，应用得也比较好。在今天的很多集成芯片中，都可以找到调试支持，比如 JTAG 扫描和 BIST（Built-In Self Test）。

与硬件领域相比，软件方面的可调试性还没有得到足够的重视。从行业标准角度

来看，目前尚没有专门针对提高软件可调试性的标准，有关的两个标准是：

- DMTF(Distributed Management Task Force)组织发起的 Common Diagnostic Model，简称 CDM。CDM 是对 DMTF 的 CIM 模型（WMI 的基础）的扩展，旨在指导软件实现标准的诊断支持，以便可以通过统一的方式发现和执行诊断功能，萃取诊断信息。
- DMTF 的 Web-Based Enterprise Management 标准，简称 WBEM，这个标准不是专门针对调试设计的，但是其中的分布式管理方法有利于收集软件的执行状态，提高软件的可调试性。

从工具角度来看，软件领域中还没有像硬件领域中的如示波器和分析仪那样成熟的工具来测量软件。尽管这方面的努力已经持续了很久：

- Program Instrumentation，即通过向程序中加入测量性代码（Instrumented Code）来收集软件的执行路径和状态信息，以实现观察、记录和寻找错误（调试）等目标。插入测量代码的方法有编译期插入和在执行期动态插入等多种方法。
- Software Profiling，即先通过工具软件为被分析软件产生某一方面的 Profile（描述），统计其在某个时间段内的活动资料，比如内存分配和释放及执行轨迹等，然后使用这些资料来发现内存使用方面的问题或寻找运行为调优（Tuning）提供信息。收集 Profile 时可以使用上面介绍的加入测量代码，也可以使用 CPU 提供的监视功能，比如第 7 章介绍的分支监视和记录机制。

从工程实践的角度来看，目前最有效的还是在设计软件中规划出支持调试的各种机制，并将其实现在软件代码中。这也是本篇重点讨论的方向——在软件开发过程中考虑并实现软件的可调试性。

26.2 Showstopper 和未雨绸缪

在戏剧和表演方面，人们使用 showstopper 一词来形容令人拍手叫绝的精彩演出，它被观众的掌声和喝彩声打断，不得不停下来等人们安静后才能再继续。

在日常生活中，showstopper 也是一个很好的词，人们用它来形容超乎寻常的美丽和迷人……

但在这个词被引入到计算机特别是软件领域后，它的含义发生了根本性的变化，它代表的是最严重的问题（Bug）！这样的问题会使整个项目停滞不前，这样的问题解决不了，产品就不可能发布……

26.2.1 NT 3.1 的故事

NT 3.1 是 Windows NT 系列操作系统的第一个版本，Windows 2000、XP、Server

2003 和 Vista 都来源于它。可以说，NT 3.1 的很多经典设计还一直保留在今天的 Windows 系统中，也正是这些经典设计为 NT 系列操作系统的成功打下了坚实基础。

如果从 1988 年 10 月 31 日 NT 内核之父 David Cutler 加入微软、11 月正式成立 NT 开发团队算起，到 1993 年 7 月 26 日 NT 3.1 发布，NT 3.1 的开发时间经历了 4 年零 9 个月。在这 4 年多时间里，NT 3.1 开发团队从最初的 6 人增加到结束时的 200 人左右。

在我的案边，有一本书生动翔实地记录了 NT 3.1 的开发过程，作者是 G. Pascal Zachary，书名是 *Showstopper*，这个书名以红色字体大大的印在封面上，格外显眼（见图 26-1）。



图 26-1 记录 NT 3.1 开发过程的经典之作——*Showstopper*

除了书名，这本书中第 10 章的标题也叫 *Showstopper*，这一章记述了 NT 3.1 发布前的约半年时间里 NT 团队与 *Showstopper* 战斗的历程，下面这些情节特别值得回味。

1993 年 2 月 28 日，计划发布 Beta 2 的日子，这一天还有 45 个 *Showstopper* 级别的问题困扰着整个团队，如此多的 *Showstopper* 是不符合发布标准的。

1993 年 3 月 8 日，NT 3.1 的 Beta 2 发布，并将最终版本的发布时间定在 5 月 10 日。但在接下来的几周内，*Showstopper* 和 1 号优先级的问题迅猛出现，到 4 月 19 日，已达到令人不安的 448 个。

1993 年 4 月 23 日，因为还有 361 个严重问题和近 2000 个其他问题，David Cutler 不得不取消了 5 月 10 日的发布计划，通过电子邮件告诉大家最终版本的发布日期推迟到 6 月 7 日。

接下来的一两个月，很多人经历了不眠之夜，与自己负责的 Bug 特别是 *Showstopper* 作斗争。能成功将自己拥有的 Bug 数降为 0 的人可以穿上“Zero Bug”衬衫。

1993 年 7 月 9 日，*Showstopper* 的数量终于降低到十几个。7 月 16 日，NT 的第 509 个 Build 被投入到最后的紧张测试，这是 1993 年的第 170 个 Build。这样的 Build 编号一直延续到今天，Windows XP 的发布版本使用的是 2600，Windows Vista 的发布版本使用的是 6000。

1993 年 7 月 23 日（星期五），David Cutler 召集了 NT 3.1 开发历史上的最后一次早 9 点会议。大家都意识到离项目结束已经很近，对其充满期待。但是负责测试工作的 Moshe Dunnie 描述了一个与 Pagemaker 5 有关的 Showstopper，为 NT 的最后发布又添加了未知性。

为了这个最后的 Showstopper，Dunnie 从 7 月 23 日早晨一直工作到 7 月 24 日夜里 10 点。Dunnie 首先怀疑是字体和打印方面的问题，并请这方面的工程师来进行调试，但是到中午时，这个怀疑被排除了。于是 Dunnie 把希望转到图形（Graphics）方面，几个工程师开始一行行地跟踪图形方面的代码，艰苦的调试工作持续到夜里 10 点之后，终于发现了一个问题，并在 10 分钟内修正了相关的代码。但是问题并没有完全解决，当 Pagemaker 打印时，还是会占用非常多的内存并且极其缓慢。这时已经是第二天（星期六）的凌晨 2 点。Dunnie 找到了 Pagemaker 的设计者一同解决问题。第二天上午，终于发现了 Pagemaker 程序中的问题。考虑到最终用户并不知道是 NT 的问题还是应用软件的问题，大家决定在图形代码中加入一个标志来兼容 Pagemaker 程序。接下来是谨慎的修改和严格的测试，直到确认测试通过后，大家才走出了办公室，这时已经是第二天的夜里 10 点。

又经历了 41 个小时的不停测试后，NT 3.1 发布给工厂制作拷贝，当时的时间是 1993 年 7 月 26 日下午 2：30。

从上面的叙述可以看出，在 NT 3.1 开发的最后约半年时间内，解决 Showstopper 已经成为整个开发团队的 1 号任务，也是决定产品能否发布的最关键指标。

Showstopper 除了对项目发布日期的影响之外，还会大量浪费团队的资源，增加整个项目的成本。而且，项目的参与人员越多，影响也越大，因为一个 Showstopper 可能导致团队中大多数人都陷入等待状态，直到这个 Showstopper 被铲除。

26.2.2 未雨绸缪

在软件开发领域，像开发 NT 3.1 那样与 Showstopper 斗争的故事应该有很多很多，而且某些可能更精彩和激烈。或许可以说，每个夜晚都有疲倦的眼神在跟踪冗长的代码，寻找问题的根源。

与 NT 3.1 的故事类似，很多软件项目延期与存在大量 Showstopper 直接相关。而解决 Showstopper 的关键是寻找问题的根源。很多问题如果找到了根源，那么解决起来通常非常简单。NT 3.1 的最后一个 Showstopper 也证明了这一点，寻找问题根源用去了十几个小时，修正代码只花了 10 分钟。

通常一个软件的 Beta 版本已经包含了所有功能的实现。那么，从此之后最核心的问题就是发现和修正错误，甚至有人把从 Beta 开始到软件发布这一段时间叫做调试阶段。NT 3.1 的第一个 Beta 版本是在 1992 年 10 月 12 日发布的，到次年 7 月发布最终

版本，又经历了 9 个多月的时间，这相当于全部开发时间的 $9/57 = 16\%$ 。Windows Vista 的 Beta 1 在 2005 年 7 月 27 日发布，正式版本（Volume Licence 客户）在 2006 年 12 月发布，其间经历了大约 17 个月，约占总开发时间的 $17/67 = 25\%$ 。可见，花在 Beta 阶段的时间占整个开发周期的比重是比较高的，而且这一阶段也往往是一个项目最紧张的时候。紧张的原因有很多，时间压力当然是一个方面，另一方面就是很多问题难以解决。因为 Beta 阶段主要是定位和修正错误，所以提高调试效率是这一阶段成功的关键。

高效调试是一项系统工程，除了系统提供好的调试支持外，被调试软件的可调试性也是至关重要的。要实现好的可调试性，应该从软件的设计阶段就开始为软件调试做准备，然后把它贯彻到整个项目的实施过程中。这样就可以在相对比较宽松的项目前期为紧张的调试阶段打下比较好的基础。相反，如果平时不做准备，那么问题出现了就要花更多的时间，并且所需的时间变得难以估算。另外，因为很多问题是在临近产品发布才出现的，所以时间非常紧迫，调试时的压力也很大。这与未雨绸缪的经验恰好吻合，在下雨之前要把基础设施建好，不要等风雨来临时叫苦不迭。

26.3 基本原则

软件调试是一项难度较大、复杂度较高的脑力劳动，很多时候为了解决一个问题要付出数小时乃至数天的探索和努力。因此，提高软件可调试性的宗旨就是要降低软件调试的难度，使软件易于被调试。软件调试中难度最大的部分通常是寻找导致问题的根源（Root Cause）。那么，降低软件调试的难度也就是要让被调试软件可以更容易地被诊断和分析，让其中的问题更容易被发现。基于这一思想，我们归纳出了以下几个原则以提高软件的可调试性。

26.3.1 最短距离原则

简要描述：使错误检查代码距离失败操作的距离最短

这一原则的目标是及时检测到异常情况。换言之，哪里可能有失败，哪里就做检查。举例来说，某个类的 Init 方法调用 Load 方法来从配置文件中读取设置并初始化成员变量。因为配置文件丢失，Load 方法读取文件失败，但它没有及时检查到失败情况，并“成功”返回，而后 Init 方法在使用没有正确初始化的成员变量时而导致错误。对于这样的情况，错误的第一现场被错过了，这会增加调试的难度。

遵循最短距离原则要求程序员编写足够多的错误检查代码。如果只有一两个错误检查，那么是很难做到覆盖程序中所有错误情况的。有些程序员不喜欢编写错误检查代码，很少做错误检查，这是应该纠正的。

26.3.2 最小范围原则

简要描述：使错误报告或调试信息所能定位到的范围尽可能小

换句话说，就是让调试人员可以利用调试信息精确地定位到某个代码位置，或者某个条件，以加快发现问题根源的速度。如果调试信息非常空泛，或者模棱两可，那么，这样的信息所提供的帮助便可能很有限，甚至产生误导作用。比如有些程序员输出调试信息时经常只包含一个“error”或“abnormal”，这样的信息聊胜于无。

最小范围原则的另一个例子是 BIOS 程序所使用的 Post Code。BIOS 代码是 CPU 复位后最先执行的代码，此时还不能通过日志文件或窗口等方式来报告错误。因此，BIOS 软件使用的典型方式是向 0x80 端口输出一个称为 Post Code 的整数。每个 Post Code 值代表某一个代码块，或者一类错误。如果启动失败，那么可以从最后一个 Post Code 值来推测失败原因和执行位置。使用供调试用的 Post Code 接收卡（插在主板上）可以接收到 Post Code。某些 BIOS 也会将 Post Code 打印在屏幕上，以方便调试。Post Code 的取值应该是精心编排的，为不同的代码块分配不同的代码范围，每个代码具有明确的含义，这样便有利于通过 Post Code 值反向追溯到相关的代码。类似的，当我们编写驱动程序和应用软件时，也应该合理地规划返回值和错误代码，尽可能使每个错误代码有明确的范围和含义，以便使用它可以比较精确地定位到导致错误的代码。

对于使用文字（字符串）来报告错误和调试信息的情况，应该力争在文字中提供尽可能多而且精确的信息，使不同地方产生的错误描述能够相互区分，这样可以很容易缩小分析范围，提高调试效率。如果很多个函数产生的错误信息都是一样的，那么依靠这样的错误信息仍然难以判断出是哪里出了问题。从这一角度来说，本原则也可以称为信息唯一性原则。

26.3.3 立刻终止原则

简要描述：当检测到严重的错误时，使程序立刻终止并报告第一现场的信息

这一原则的一个典型应用就是操作系统的错误检查（Bug Check）机制，比如 Windows 操作系统的蓝屏机制（Blue Screen Of Death，简称 BSOD）。蓝屏机制可以让系统在遇到危险情况时以可控的方式停止，防止继续运行可能造成的更大损失。如我们在第 13 章所介绍的，一旦内核中的代码发起蓝屏，那么系统便立刻停止运行用户态代码和一切例行工作（文件服务等），只以单线程方式做错误记录和报告。另外，使整个系统终止在发现错误的第一现场有利于分析错误发生的原因。因为如果让系统继续执行很多任务，那么执行轨迹就会偏离错误的第一现场。如果执行其他任务又导致错误，原始的错误情况就会被掩盖起来，为调试设置障碍。因此，这一原则也可以称为

报告第一现场原则。

这个原则的另一层含义是当程序中遇到错误时，应该“让错误立刻跳出来”，而不要使其隐匿起来。以蓝屏的方式终止是使错误跳出来的一种方式。但这种方式的代价是比较大的，系统中运行的所有程序都会戛然而止，没有保存的文档可能会丢失。因此有人对蓝屏机制发出质疑，但是笔者认为应该从以下两方面来看待这个问题：

- 触发蓝屏的原因主要是发生在内核空间的错误，有时是硬件方面的无法恢复错误，有时是某些内核模块（设备驱动程序）中的编码错误。对于前者，立刻终止是必要的；对于后者，因为内核模块是操作系统的信任代码，所以对其中的错误实施严厉的制裁也是可以理解的，这不仅有利于提高内核模块的质量，而且有利于抵御侵入到内核空间的恶意软件的攻击。
- 作为一个通用的操作系统，应该有一套高效且通用的错误检测和报告规则，这套机制的主要目标是及时检测到错误，精确报告错误的第一现场，以便可以准确地定位到导致错误的软硬件模块，然后报告给这些部件的开发者。从这个角度来衡量，蓝屏机制是合理而且有效的。

对于应用软件，应该根据软件的特征制定错误报告策略，根据错误的严重级别决定应该继续执行还是停止运行。

26.3.4 可追溯原则

简要描述：使代码的执行轨迹和数据的变化过程可以追溯

所谓可以追溯（Tracable），对于代码，就是可以查找出当前线程是如何运行到这个代码位置的。对于数据（变量），就是可以知道它的值是经历了什么样的变化过程而成为当前取值的。

很多时候，尽管我们知道了错误发生的位置，如某个模块的某个函数，但是仍然无法理解它为什么会出错。因为这个函数本身可能根本没有错误，发生错误是因为其他函数不应该在当前情况下调用它，或者传递给它的参数有错误。举例来说，很多内核函数只能在特定的中断级别（IRQL）下执行，如果某个驱动程序在不满足这个条件的情况下调用这些函数，就会发生错误。但这时只知道错误的发生位置是不够的，还必须有信息可以追溯函数的调用过程，寻找发起错误调用的那个函数和它所属的驱动程序。

追溯代码执行轨迹的最有效技术就是利用栈中的栈帧信息来生成栈回溯序列，即所谓的 Stack Backtrace。根据我们在第 22 章的介绍，当 A 函数调用 B 函数时，函数调用指令（如 call）会自动将函数 B 执行后的返回地址（函数 A 中 call 指令之后的那条指令的地址）压入到栈中。依此类推，当函数 B 调用 C 时，栈中也会记录下从函数 C 返回到函数 B 的地址。根据这样的信息，我们就可以产生调用函数 C 的过程（A→B

→C)，也就是程序执行到函数 C 的轨迹。几乎所有调试器都提供了产生栈回溯信息的功能，比如 WinDBG 的 k 系列命令、GDB 调试器的 bt (Backtrace 的缩写) 命令和 VS 调试器的函数调用 (Call Stack) 窗口。

因为只有正确地找到每个函数的栈帧，才能据此找到它的返回地址，所以对于使用了帧指针省略 (FPO) 的函数，必须依靠调试符号中记录的 FPO 信息才能找到其栈帧。如果没有调试符号，那么回溯序列中就会缺少调用这个函数的记录。因为这个因素，禁止编译器对函数做 FPO 优化有利于调试。在代码中通过编译器指令 (directives) 可以禁止使用 FPO，举例来说，如果在源程序文件中加入 #pragma optimize("y", off)，那么这个指令后的函数就不会被 FPO 优化，直到再开启 FPO 为止。Windows Vista 的大多数模块都不再使用 FPO。

迄今为止，还没有非常方便而且开销又低的方法来实现数据的可追溯性。因为要给变量赋一个新的值，便会覆盖掉它以前的值。要想记住旧的值，必须先保存起来，这必然需要额外的时间和空间开销。以下是几种可能的解决方案。

第一，通过日志 (log) 的方式将变量的每个值以文件或其他方式记录下来。

第二，如果允许使用数据库，那么可以利用数据库本身的功能或编写脚本记录一个字段的每次取值。

第三，编写专门的类，为要追溯的数据定义用于记录其历史取值的环形缓冲区，重载赋值运算符。当每次赋值时，先将当前值保存在缓冲区中。

本书下一章将进一步讨论实现可追溯性的方法，并给出一些演示性的代码。

26.3.5 可控制原则

简要描述：通过简单的方式就可以控制程序的执行轨迹

软件功能的可控性 (Controllability) 和灵活性 (Flexibility) 是软件智能的重要体现，对于软件调试也有重要意义。就调试而言，可控性意味着调试人员可以轻易地调整软件的执行路线，使其沿着需要跟踪和分析的路线执行。很多时候，测试人员报告软件错误时会给出重现这个问题的一系列步骤。为了发现问题的根源，调试时通常也需要沿着这些步骤来进行跟踪和分析。但在很多情况下，调试环境与用户环境可能有较大的差异。比如某个问题只发生在 1GB 内存的情况下，而调试环境为 2GB 内存。这时一种方法是更换内存或寻找 1GB 内存的系统，但如果被调试的系统（操作系统）支持通过配置选项来指定只使用 1GB 的内存，就方便得多。Windows 操作系统启动配置文件中的 /MAXMEM 选项恰好提供了这样的功能。

与可控制原则有关的一个原则是可重复原则。

26.3.6 可重复原则

简要描述：使程序的行为可以被简单地重复

这一原则的初衷是可以比较简单地重复执行程序的某个部分或整体。因为当调试时，很多时候我们要反复跟踪和观察某段代码才能发现其中的问题。如果每次重新执行都需要大量繁琐的操作，那么必然会影响调试的效率。举例来说，如果要重新执行某个函数，就需要重新启动一次电脑，或者要复位很多其他关联的程序，那么这就是有悖于可重复（Repeatable）原则的。

对于某些与硬件协同工作的软件，重复某个操作的成本可能很高。以用于医疗等用途的图像采集和分析软件为例，让硬件反复拍摄照片是有明显开销的。这时，可以考虑使用模拟程序来伪装硬件，这样程序员在调试时就不必担心反复运行的次数。当然，模拟器并不能完全代替真实的硬件，在模拟器上运行没有问题后还是应该测试与真实硬件一起工作的情况。

可重复原则的一个隐含要求是每次重复执行时，程序的执行行为应该是有规律的，这个规律越简单就越有利于调试，如果这个规律比较复杂，那么它应该是调试人员所能理解并可控制的。举例来说，某些软件会记录是否是第一次运行，如果是，就做很多初始化操作，如果不是，便跳过初始化过程。依据本原则，调试人员应该总是可以通过简单的操作就模拟第一次执行的情况，以便反复跟踪这种情况。如果执行过一次，一定要重新安装这个软件（甚至整个系统）才能再次跟踪初始化过程，那么就会影响调试的效率。

26.3.7 可观察原则

简要描述：使软件的特征和内部状态可以被方便地观察

这一原则的目的是提高软件的可观察性（Observability），让调试人员可以方便地观察到程序的静态特征和动态特征。静态特征包括文件（映像文件、源文件、符号文件和配置文件等）信息、函数信息等。动态特征是指处于运行状态的软件在某一时刻的属性，包括程序的执行位置、变量取值、内存和资源使用情况等。

可观察性的好与坏通常是相对的，根据达到观察目的所需要的“成本”，可以初步分成如下一些级别。

1. 不需要任何额外工具，通过软件自身提供的功能就可以观察到。
2. 借助软件运行环境中的通用工具可以观察到，比如操作系统的文件浏览工具和文件显示工具。
3. 借助通用的调试器和其他通用工具可以观察到。

4. 购买和安装专门的软硬件工具才可以观察到。
5. 只有安装被调试软件的特殊版本才可以观察到。

以上几点基本是按可观察性从高到低的顺序来编排的。高可观察性有利于发现系统状态和可能存在的问题，便于调试。尤其是当软件故障发生时那一刻的动态特征对于调试特别有意义。但是以所有用户都可以访问的方式来显示过多的内部信息可能存在泄漏技术和商业秘密等问题。因此，比较好的做法是根据用户的身份来决定哪些信息对其是可见的。

26.3.8 可辨识原则

简要描述：可以简单地辨识出每个模块乃至类或函数的版本

版本错误是滋扰软件团队的一个古老话题，某些时候，花了很大功夫得出的唯一结论就是使用的版本不对。为了防止这样令人哭笑不得的事情发生，设计和编码时就应该重视版本问题，提高每个软件模块的可辨识性（Identifiability）。比如，有固定的版本记录机制，并及时更新其中的版本信息，通过一种简单的方式就可以访问到这个信息等。

本节介绍了提高软件可调试性的一些基本原则，这些原则从不同的角度来降低软件调试的复杂度。但需要声明的是，这些原则并不是用之四海皆准的灵丹妙药，应该根据每个软件的特征和需求制定具体的方案。

26.4 不可调试代码

我们把调试器无法跟踪或无法对其设置断点的代码称为不可调试代码。当然，这种不可调试性是相对调试器而言的。比如使用软件调试器不可调试的代码可能是可以被硬件调试器所调试的。考虑到软件调试器是最常用的调试工具，所以本节我们将先介绍几种不可被软件调试器（例如 WinDBG）所调试的典型实例，然后讨论如何降低它们对调试的影响。

26.4.1 系统的异常分发函数

通常，操作系统的异常分发函数是不可以被调试的，如果对其强行设置断点或单步跟踪，那么通常会因为递归而导致被调试系统崩溃。举例来说，当进行内核调试时，如果对 Windows 内核的 `KiDispatchException` 函数设置断点，那么一旦该断点被触发，被调试系统就会自动重启。这是因为断点事件本身也会被当作异常来分发，而当分发过程执行到 `KiDispatchException` 函数时会再次触发断点异常，这样的死循环很快就导致 CPU 复位了。

因为硬件中断需要及时确认（Acknowledge），所以通常最好也不要在中断处理例

程的入口处设置断点，以防止系统无法及时确认中断，而导致硬件反复发送中断请求，即所谓的中断风暴（Interrupt Storm）。

26.4.2 提供调试功能的系统函数

提供调试功能的很多系统函数是不可以被调试的，因为如果这些函数中的断点被触发后很可能会导致死循环。比如在内核调试时，负责与调试器通信的 `nt!KdSendPacket` 和 `nt!KdReceivePacket` 函数都不可以被设置断点和单步跟踪，因为发送断点事件会再次触发断点。

类似的，某些注册在调试事件循环中的函数也是不可以被调试的。在这些函数中使用调试有关的 API 也需要慎重，以防止导致递归调用。

举例来说，在向量化异常处理函数（VEH）中不可以一开始就调用 `OutputDebugString` 函数。第 10.7 节我们介绍过 `OutputDebugString` 函数的工作原理，简单来说，它是靠 `RaiseException` 来产生一个特殊的异常而工作的。因此，如果在 VEH 处理函数中没采取任何措施就调用 `OutputDebugString` 函数，那么 `OutputDebugString` 函数产生的异常会触发 VEH 处理函数再次被调用，如此循环不断，直到栈溢出而程序崩溃。因为系统是在寻找基于帧的异常处理器之前调用 VEH 处理函数的，所以应用程序错误对话框也不会弹出来，从表面上看程序只是突然消失掉。

解决这个问题的办法是在 VEH 处理函数中将 `OutputDebugString` 函数所产生的异常排除掉，然后调用 `OutputDebugString` 函数就没有问题了。

```
LONG WINAPI MyVectoredHandler( struct _EXCEPTION_POINTERS *ExceptionInfo )
{
    // 这里调用 OutputDebugString 会导致死循环
    if(ExceptionInfo->ExceptionRecord->ExceptionCode==0x40010006L)
        return EXCEPTION_CONTINUE_SEARCH;
    // 现在可以调用 OutputDebugString 了
    OutputDebugString(_T("MyVectoredHandler is invoked"));
    // ...
}
```

这个问题是笔者在一个实际软件项目中遇到的，在产品即将发布的最后阶段，项目中的一个主要程序突然出现问题，运行一段时间后进程悄无声息地消失，没有错误窗口，没有征兆，经历了近一个小时的追查后终于发现是 VEH 函数中新加入的 `OutputDebugString` 调用惹了祸，增加上面代码中的判断语句后问题就解决了。

26.4.3 对调试器敏感的函数

某些函数会检测当前是否在被调试，如果不在被调试，会执行一种路线，如果在被调试，会执行另一种路线。这样一来，前一种路线便成为不可调试代码。比如位于 NTDLL 中的 `UnhandledExceptionFilter` 函数，如果当前程序不在被调试，它会启

动 WER 机制报告应用程序错误；如果当前程序在被调试那么它会简单的返回 EXCEPTION_CONTINUE_SEARCH，引发第二轮异常分发和处理。这样一来，启动 WER 并报告应用程序错误的代码就变得不可调试。当然不可调试永远是相对的，第 12.6.4 节为大家介绍了一种方法来调试产生错误对话框的过程。

26.4.4 反跟踪和调试的程序

出于各种考虑，某些程序会故意阻止被跟踪或调试，当检测到被调试时，它们会通过进入死循环等方式抵抗跟踪；清除断点寄存器破坏断点工作；插入所谓的花指令干扰反汇编程序进行反汇编等。被这些逻辑所保护和遮挡的代码是难以调试的。

26.4.5 时间敏感的代码

当软件在调试器中运行时，它的运行速度通常会变慢，如果进行单步跟踪和交互式调试，那么被调试软件可能长时间停留在一个位置。为了支持调试，应该尽可能地避免编写时间敏感的代码，保证软件在被调试时仍以原来的逻辑运行。

26.4.6 应对措施

不可调试代码的存在会为调试增添难度。而且没有通用的方法来解决，以下是一些可能的一些方案。

第一，使用不同的调试器，特别是硬件调试器，比如我们在第 8 章介绍的 ITP/XDP 调试器。

第二，动态修改检测调试器是否存在的检测结果（寄存器），调试 Unhandled-ExceptionFilter 函数的方法就是这样做的。

第三，修改程序指针寄存器（EIP）强制跳转到要调试的程序路径。比如在 WinDBG 中使用 r 命令就可以修改 EIP 寄存器的值。不过这样做有较大的风险，容易破坏栈平衡，使程序异常终止。

第四，使用调试器的汇编功能动态修改阻碍调试的代码。举例来说，如果被调试的路径上有一条断点指令（INT 3）防止我们向前跟踪，那么可以使用 WinDBG 的 a 命令将其替换为 nop 指令。操作步骤是执行 a<地址>，然后在 WinDBG 的交互式编辑提示框中输入 nop，按回车后 WinDBG 便把 nop 指令编译到指定的地址，然后直接按回车结束编辑。

26.5 可调试性例析

Windows 是一个庞大而且复杂的操作系统，Vista 的源代码行数约为 5 千万行（50

million), NT 3.1 的也有 560 万 (5.6 million) 行。Vista 安装后在磁盘上所占的空间约为 6GB, 其中内核态核心模块 NTOSKRNL.EXE 的大小为 3.4MB, NTDLL.DLL 为 1.1MB。表 26-1 列出了 Windows 几个主要版本的 NTOSKRNL.EXE 和 NTDLL.DLL 文件的典型大小。

表 26-1 Windows 的内核和 NTDLL 文件的大小 (字节)

	NT3.51	Win2K	XP SP1	XP SP2	Vista RTM
NTOSKRNL.EXE	804,864	1,640,976	2,042,240	2,180,096	3,467,880
NTDLL.DLL	307,088	481,040	668,672	708,096	1,162,656

从运行态来看, 在一个典型的 Windows 系统中, 大多数时候都有几十个进程 (数百个线程) 在运行。以笔者写作此内容时所使用的 Windows XP SP2 系统为例, 系统中共有 82 个进程, 783 个线程在工作。对于 Windows 这样复杂的系统, 可调试性对其是极其重要的。事实上, 好的可调试性是 Windows 系统成功的一个关键技术因素。如果没有这个因素, 这个系统也许就会因为太多的 Showstopper 而永远不能发布。本节我们将从 Windows 操作系统中选取一些体现可调试性的特征进行分析, 以学习其中所蕴含的设计思想, 并加深大家对软件可调试性的理解。

26.5.1 Sanity Check 和 BSOD

在很多 Windows 内核函数中, 存在类似如下的代码:

```
if (...)  
    BugCheckEx (...);
```

这样的代码通常被称为 Sanity Check, 即为了保证系统健全性而作的额外检查。如果检查失败, 则以蓝屏 (BSOD) 的形式报告出来。

Sanity Check 与断言 (ASSERT) 不同, 断言只存在于 Checked 版本中, 而 Sanity Check 既存在于 Checked 版本中, 又存在于 Free 版本中。事实上, Checked 和 Free 这两种称呼就是在开发 Windows NT 时, 为了将包含 Sanity Check 的版本与不包含 Sanity Check 的 Release 版本区分开来而引入的。在此之前, 通常只使用 Debug 和 Release 两种定义方式。表 26-2 列出了这 4 种版本定义方式的主要差异。

表 26-2 构建方式比较

	Debug	Checked	Free	Release
编译器优化 (Compiler Optimization)	OFF	ON	ON	ON
调试追踪 (Debug Traces)	ON	ON	OFF	OFF
断言 (Assertions)	ON	ON	OFF	OFF
Sanity Checks	ON	ON	ON	OFF

因为开发 NT 3.1 时, 测试团队主要测试的是 Checked 版本和 Free 版本, 并没有对 Debug 和 Release 版本做很多测试, 所以 NT 3.1 发布时没有使用 Release 版本。也就是说, Sanity Check 在正式发布的产品版本中依然存在。这种做法一直延续到今天, 而且

这种定义方式也被应用到驱动程序开发中。从可调试性的角度来看，Sanity Check 和蓝屏机制有利于及时发现错误和异常情况，并让错误“跳出来”。

26.5.2 可控制性

Windows 操作系统中几乎随处都可以看到高灵活性和可控性的设计。比如，Windows 的启动配置文件（BOOT.INI）支持 40 多个选项来定义系统的工作参数。另外，Windows 的很多行为都可以通过修改注册表中的键值来控制。可配置能力不仅提高了 Windows 系统的灵活性，而且有利于调试和维护。以下是使用配置选项来帮助调试的几个例子：

- /BREAK，这个选项会告诉 HAL 在初始化时等待内核调试会话建立，以便可以使更多的初始化代码可以被调试，详见 18.7.2 节。
- /KERNEL=/HAL=，可以利用这两个选项来指定要使用的内核文件和 HAL 文件。比如可以使用这种方法来将内核和 HAL 文件切换为 Checked 版本。
- /MAXMEM=，指定 Windows 要使用的内存，利用这个选项可以模拟只有在小内存系统才出现的问题。
- /DEBU 和/DEBUGPORT=，启动和配置内核调试引擎。
- /CRASHDEBUG，与/DEBUG 在启动期间就启动内核调试不同，这个选项告诉 Windows，当系统发生崩溃（BSOD）时再启动内核调试引擎，这与应用程序中的 JIT 调试很类似。
- /NOPAE，强制加载不包含 PAE（Physical Address Extension）支持的内核文件。这对于在支持 PAE 的硬件上调试非 PAE 情况下发生的问题是很有帮助的。
- /NUMPROC=，指定要使用的 CPU 数目，使用这个选项可以在多 CPU 系统上调试单 CPU 情况下的问题。
- /YEAR=，强制系统使用指定的年份，忽略计算机系统的实际年份，选项是为了帮助调试“2000 年问题”而设计的。

使用注册表来帮助调试的例子也有很多，比如可以在 Image File Execution Options 下为一个程序设置执行选项，让系统加载这个程序时先启动调试器。

26.5.3 公开的符号文件

在第 25 章我们深入讨论了调试符号对于软件调试的意义，有了调试符号可以大大降低跟踪执行的难度，加快发现问题根源的速度。微软的调试符号服务器为 Windows 操作系统的几乎所有程序文件提供了符号文件，并且包含了公开发布的大多数版本。

从微软的网站也可以根据操作系统版本下载其对应的符号文件包。

公开的符号文件不仅为调试和学习 Windows 操作系统提供了帮助，而且也为开发和调试 Windows 驱动程序和应用程序提供了支持。

26.5.4 WER

Windows 错误报告（Windows Error Reporting，简称 WER）机制可以自动收集应用程序或系统崩溃的信息，生成报告，并在征求用户同意后发送到用于错误分析的服务器（详见第 14 章）。自动报告是一种有效的辅助调试手段，有利于降低调试成本，尤其对于产品期调试有着极高的价值。

26.5.5 ETW 和日志

Windows 的 Event Trace for Windows（简称 ETW）机制可以高效地记录操作系统、驱动程序，或者应用程序的事件（详见第 16 章）。使用 ETW 可以有效地提高软件的可追溯性。

Windows 操作系统主要有两种日志机制，一种是基于日志服务的，调用 ReportEvent API 来写日志记录。另一种是 Windows Vista 引入的 Common Log File System（CLFS）。CLFS 的核心功能是由一个名为 CLFS.SYS 的内核模块所提供的。CLFS.SYS 输出了一系列以 Clfs 开头的函数和结构，如 ClfsCreateLogFile 等，内核模式的驱动程序可以直接调用这些函数。用户态的程序可以调用 Clfsw32.dll 所输出的用户态 API。我们在第 15 章详细介绍了这两种日志机制。

26.5.6 性能计数器

Windows 操作系统内建了性能监视（Performance Monitor）机制，通过性能计数器（Performance Counters）来记录软件的内部状态。Windows 预定义了大量反映操作系统内核和系统对象状态的计数器，软件开发商也可以定义并登记其他计数器。

使用 perfmon 工具可以以图形化的方式观察性能计数器的值。Windows XP 引入了一个名为 typeperf 的命令行工具来观察性能计数器。例如，以下是使用 typeperf 命令显示可用内存数量时的执行结果：

```
C:\> typeperf "Memory\Available Bytes" -si 00:05 //每 5 秒钟更新一次
" (PDH-CSV 4.0) ", "\AdvDbg002\Memory\Available Bytes"
"05/07/2007 17:08:36.375", "1213886464.000000"
"05/07/2007 17:08:41.375", "1211101184.000000"
...
```

性能计数器为系统管理员和计算机用户了解系统运行情况提供了一种简单而有效

的方式，对于调试系统中与性能有关的软硬件问题有着重要作用。

26.5.7 内建的内核调试引擎

Windows 内核调试引擎内建在每个 Windows 系统的内核之中，主要功能包含在 NTOSKRNL.EXE 中（详见第 18 章）。这意味着，内核调试支持始终存在于 Windows 系统中，要对一个发生故障的系统进行内核调试时不需要重新安装特别的版本或其他文件，这为调试 Windows 内核和内核态的其他程序提供了很大的便利。

26.5.8 手动触发崩溃

Windows 还有一个不太被注意的调试支持，即在注册表中设置了 CrashOnCtrlScroll = 1 并重启后（详见 13.2.4 节），按住 PS2 键盘右侧的 Ctrl 键后再按 ScrollLock 键，系统会产生一个特别的蓝屏崩溃，其停止码为 MANUALLY_INITIATED_CRASH (0xE2)。因为蓝屏可以触发崩溃时才被激活的内核调试（/CRASHDEBUG）和内核转储，所以这个支持对调试某些随机的系统僵死很有用，比如突然没有响应或在开机关机过程中发生的无限等待。

本节介绍了 Windows 操作系统内建的一些调试支持，类似的例子还有很多。通过这些例子，我们可以认识到支持可调试性的意义，以及带来的好处，同时也应该思考如何在自己的软件产品中加入类似的机制。

26.6 与安全、性能和商业秘密的关系

任何事物都有两面性，我们不得不承认实现可调试性本身也是有代价的。特别应该注意以下几个方面的影响：安全、性能和商业秘密。

26.6.1 可调试性与安全

高可调试性追求对软件的全方位掌控，可以了解其状态的任何细节，并控制它的行为。这对调试来说是有利的，但是如果这些功能或机制被恶意软件或入侵的黑客所使用，那么导致的后果可能很严重。这就好比武器被别人盗用了，武器越强大，导致的危害可能越大。

考虑到这一点，当设计可调试性机制时，应该配以必要的安全防范措施。例如，可以设计登录和验证机制，根据用户的角色决定他可以使用的功能。这就好像网站的管理和维护功能只对网站的管理员开放。

26.6.2 可调试性与商业秘密

实现可调试性时也要注意防止泄漏商业和技术秘密。如果日志或调试信息中包含重要的算法和资料，那么在存储和输出信息前，应该先将信息加密，或者利用 ETW 技术使用二进制格式来输出日志信息，并控制好格式文件 (TMF)。但是这种保密只是增加阅读的难度，攻击者还是可能分析出有效的数据。

因为符号文件包含了软件的很多细节，所以应该注意合理保护 PDB 文件，尤其是包含全部调试信息的私有符号文件。对于合作伙伴或客户通常只提供剥离私有符号后的公开符号文件。使用/PDBSTRIPPED 链接选项可以产生公开符号文件（详见第 25 章）。

26.6.3 可调试性与性能

可调试性对性能的影响主要体现在两个方面，从空间角度来看，在程序中支持可调试性必然会增加可执行文件的大小，生成日志等信息会占用一定的磁盘空间。从时间来看，用于提高可调试性的代码可能会占用少量的 CPU 时间。因此当设计可调试性机制时，应该注意以下两点。

第一，将调试机制设计成可开关的，最好是可以动态开关的。这样当不需要调试时，调试机制的影响非常小；当需要调试时，又可以立刻开启。

第二，防止调试机制被错用和滥用，调试机制的目的是辅助调试，不应该用于其他目的。应该避免过量使用调试机制，否则不仅会影响性能，而且对调试本身也可能产生副作用。举例来说，如果频繁打印大量的重复信息，会使调试者眼花缭乱，难以找到真正有用的信息。

相对于提高可调试性所带来的好处，它的副作用还是可以接受的，而且只要处理得当，可以把这种影响降得很低。

26.7 本章总结

从调试和维护软件所付出的代价来看，人们对软件可调试性的重视还很不够。如果像每个建筑中都必须配备消防设施那样在每个软件中都配备必要的调试设施，那么花在软件调试上的时间和投入都会大幅下降。

本章比较详细地讨论了软件可调试性的内涵、基本原则和重视可调试性的意义，并分析了 Windows 操作系统中体现可调试性的设计。在最后一节我们探讨了可调试性与安全、性能和商业秘密的关系。下一章我们将进一步讨论如何在软件工程中实现可调试性。

参考文献

1. H&E. Vranken', M.P.J. Stevens, M.T.M. Segers. Design-For-Debug in Hardware/Software Co-Design
2. G. Pascal Zachary. Showstopper: The Breakneck Race to Create Windows NT and the Next Generation at Microsoft. The Free Press, 1994
3. Larry Osterman. Where do "checked" and "free" come from?
<http://blogs.msdn.com/larryosterman>

可调试性的实现

上一章我们讨论了增强软件可调试性的意义、目标和基本原则。本章将在上一章的基础上讨论实现可调试性的基本方法。我们首先介绍软件团队中的各种角色在提高可调试性方面应该承担的职责（27.1 节），然后介绍如何在架构设计阶段就为提高可调试性做好各种规划和准备（27.2 节）。后面 4 节将分别讨论实现可追溯性（27.3~27.4 节）、可观测性（27.5 节）和自动诊断与报告（27.6 节）的典型方法。

27.1 角色和职责

因为调试的效率直接影响项目的进度，无法解决的调试问题可能导致整个项目陷入停滞，所以实现可调试性应该是软件团队中所有人的共同目标。大家都应该对提高可调试性给与足够的重视和支持，就像重视安全、质量和可靠性等一样。

27.1.1 架构师

软件架构师是规划和缔造软件系统的核心角色，他们负责规划软件系统的整体框架、模块布局、基础设施和基本的工作方式，参与制定开发策略、方针（Policy）和计划，并指导开发过程。软件架构师在软件团队中的地位好比是建筑团队中的总设计师。作为一个好的架构师，应该充分意识到提高可调试性的重要意义，承担起如下职责。

1. 在架构设计中规划统一的支持可调试性的策略、机制和设施，包括检查、记录和报告错误的方法；输出调试信息和记录日志的方式；专门用来辅助调试的模块（如模拟器等）；简化调试的设施等。
2. 设计必要的技术手段，提醒或强制程序员在编码时实现可调试性。
3. 制定提高可调试性的指导意见和纪律，并写入到软件项目的开发方针中去，以纪律的方式强制这些策略的执行，并检查和监督执行情况。
4. 通过培训或其他沟通方式让团队的成员理解可调试性的意义和实现方法。

5. 参与调试重大的软件问题，验证调试机制的有效性，并向团队证明这些机制所带来的好处。

下一节将更详细地讨论如何在架构设计中规划和设计支持调试的基础设施。

27.1.2 程序员

程序员是建造软件大厦的主力军。他们在搭建这个大厦的同时还负责调试这个大厦中存在的问题。在产品发布之前，程序员要负责调试与自己所编写代码有关的问题。在产品发布后，支持和维护工程师会承担大部分调试工作，但如果支持工程师无法解决，那么通常还是需要程序员来解决。根据粗略的统计，大多数程序员花一半以上的时间在调试上，当项目进入 Beta 阶段和邻近产品发布时这个比例通常会更高。某些团队的测试工程师会承担一部分调试职能（稍后更多讨论），但是他们通常只负责初步的分析，将错误定位到模块一级，然后通常还是分派给开发人员（程序员）。概而言之，程序员同时是编写代码和调试代码的主要力量。因此，提高可调试性，程序员是主要的执行者，也是主要的受益者。他们应该承担的职责包括。

1. 提高对错误处理的重视，认真编写错误检查和错误处理代码。不要因为错误情况是小概率事件就草草编写一些代码，要知道很多造成重大损失的大问题都是由于编写代码时的小疏忽所埋下的。
2. 认真执行架构师所制定的提高可调试性的各项策略和方针。当编写代码时，合理应用各种提高可调试性的原则，努力提高代码的可调试性。如果发现有不合理的地方，应该及时提出，而不是消极放弃。
3. 熟练使用各种调试工具，善于使用调试方法来充分理解程序的执行流程，发现并纠正其中潜在的错误。
4. 正确对待分配给自己的软件问题（Bug），不推诿，不敷衍，积极使用调试工具和提高可调试性的机制来定位问题根源，及时更新问题记录。
5. 检查日志文件和其他调试机制所生成的信息，检查是否存在不正常情况，并根据日志信息审查代码中可能存在的问题。

Robert M. Metzger在他的*Debugging by Thinking: A Multidisciplinary Approach*一书中说，导致现在的软件有如此多缺陷的原因有很多，其中之一是很多程序员并不擅长调试。提高软件的可调试性有利于降低程序员调试软件的难度，培养程序员的调试兴趣。

27.1.3 测试人员

在软件团队中，测试人员与开发人员之间经常发生争吵或相互埋怨。测试人员会抱怨软件中存在的问题太多；一个以前修复了的问题为什么又再次出现。开发人员会抱怨

测试人员的问题报告含糊不清，难以理解，或者报告的问题根本无法再现。提高软件的可调试性尽管不能彻底解决这些矛盾，但是至少会从如下几个方面有所帮助。

- 高可调试性有利于程序员深刻理解代码的执行过程，提高对代码的控制力，从根本上提高代码的质量，降低代码的问题率（每千行源代码的 Bug 数）。另外，利用日志等调试机制，程序员可以在开发阶段或单元测试阶段发现和解决更多问题，这样发布给测试人员的软件质量就会明显提高。因此高可调试性有利于减少测试人员所发现的问题（Bug）数，使他们集中测试程序员难以测试的情况。
- 测试人员可以利用调试机制来辅助测试并发现和描述问题。这样发现的问题通常更容易被开发人员所理解和解决。
- 调试机制可以帮助测试人员理解软件的工作机理和内部状态，指导测试方法，尽早发现问题，特别是深层次的问题。
- 从事白盒测试的测试人员可以利用调试机制审核代码和算法，当编写测试脚本时，也可以使用调试机制所提供的设施来检测测试案例的执行结果。

综上，提高软件的可调试性会给测试工作和测试人员带来直接的好处。测试人员应该积极支持为提高可调试性所做的各种努力，并承担起如下职责：

1. 理解提高软件可调试性的重要意义，支持开发人员实现可调试性，为他们提供测试支持。
2. 充分利用可调试机制帮助测试工作，提高测试效率。这有利于进一步发挥可调试机制的价值，进一步推动并提高软件的可调试性。

从根本上来说，测试的目的是发现软件问题，保证软件质量，按期完成开发计划。如果有很多 Showstopper 难以解决，测试人员与开发人员都承受着很大的压力，从这个意义上来说，他们应该共同为提高可调试性努力。

27.1.4 产品维护和技术支持工程师

在软件产品发布后，产品维护和技术支持工程师（以下简称支持工程师）成为调试各种产品期问题的主要力量。在客户报告一个问题后，支持工程师需要理解客户的描述，思考是用户使用的问题还是产品本身的问题。如果可能是产品本身的问题，那么需要在自己的系统中重现这个问题，然后利用各种调试手段定位问题的根源。这种情况下可能遇到的一个棘手问题就是无法重现客户报告的问题。要解决这个问题，可能不得不跑到客户那里去。但另一种更有效的办法就是通过软件的调试支持，让软件自动收集各种环境信息和错误信息，并生成报告，然后，支持人员可以利用这些报告来定位问题的根源。除了错误报告，日志文件是支持工程师经常使用的另一主要途径，很多支持工程师使用日志来寻找导致错误的原因和线索。总之，提高软件的可调试性对于产品期调试和技术支持有着重要意义。技术支持工程师应该积极支持并推动软件产品的可调试性，利用支持调试的设施解决问题，并将改善调试设施的意见反馈给架构师和开发人员。

27.1.5 管理者

像 David Cutler 这样的代码勇士 (Code Warrior) 和软件天才是不喜欢管理者干涉软件项目的。1975 年, 他在 DEC 带领一个十几个人的团队开始开发 VMS (Virtual Memory System, 后来改名为 OpenVMS) 操作系统。1977 年 VMS 操作系统随着 VAX 计算机的发布而同时发布, 这是第一台商业化的 32 位计算机系统, 也是计算机历史上硬件与操作系统一起从头开发并一起发布的很少组合之一。VMS 非常成功, 这个成功使 DEC 公司开始重视 Cutler 所带领的这个操作系统项目, 很多管理者开始频繁介入到项目中, 这使 Cutler 发怒了: “无论你要做什么, 所有的 Tom、Dick 和 Harry 都会跑过来挑三拣四, 挡住项目的去路。”不久, Cutler 离开了 VMS 团队, 并声称要离开 DEC, 这让他的老板 Gordon Bell 说出了那句经典的话: “带上你想要的任何人, 去任何你想去的地方, 做任何你想做的事。DEC 都会为你付钱, 告诉我你需要多少钱, 我们会为你拨款。”这样好的老板感动了 Cutler, 在新创建的西雅图实验室, 他开始了新的开发。1983 年 Gordon Bell 离开 DEC, 1988 年 Cutler 的 Prism 项目被取消, 他因此离开了 DEC。

在开发 NT 的长达近 5 年的时间里, 尽管有多次延期, 难以避免的意见分歧, 但是 Cutler 始终得到了管理者的强有力支持。NT 项目给了他成就理想的机遇, 他可以按照自己的想法设定目标, 并有充分的自由使其成为现实, 没有管理层来管闲事。但这样的和谐在今天的软件开发中越来越少了。很多管理者脱离技术和项目的实际状况, 武断地干预开发计划, 压缩项目时间。

实现可调试性会需要一定的开发资源, 但是正如我们前面所讨论的, 它会带来很多好处: 提高程序员的调试和开发效率; 加快解决软件问题的速度; 使整个项目的可控性提高; 辅助技术支持降低产品维护成本, 等等。从这个意义上来说, 提高可调试性有利于保证项目的如期完成并节约成本。所以, 管理者应该充分支持为提高软件可调试性所做的努力, 为其分配足够的资源。

本节我们讨论了软件团队中的几个关键角色在提高实现软件可调试性方面应该承担的职责。这些内容完全是笔者根据个人经验的归纳, 希望读者能从中得到启发。对于具体的软件项目, 应该根据实际情况为每个角色定义更明确的职责。

27.2 可调试架构

架构是构建软件的蓝图, 它决定了软件基本结构和工作方式, 包括包含哪些模块, 各模块间如何通信等。因为调试支持涉及到软件的总体结构, 对整个软件的质量有着直接的影响。所以架构师在设计软件结构时应该重视软件的可调试性, 规划好支持调试的各种设施。

27.2.1 日志

日志对提高软件的可观察性和可追溯性都大有好处。好的日志反映了软件的内部状态，特别是软件运行时遇到的异常情况，是调试软件问题的宝贵资源。

日志大多是以文本文件方式记录的，但也可以记录在数据库中，或者以二进制文件方式记录。记录日志并不复杂，但是方法很多，灵活性很大。对于一个软件产品，应该使用一套统一的日志机制。

首先，日志信息应该集中存储在一个地方，这样不仅有利于节约存储空间，而且便于检索和维护。试想，如果一个软件的日志存储在很多地方，需要到很多地方去寻找，那么当调试与很多软件有关的系统问题时，调试人员可能根本找不到需要的日志文件。

第二，选择并定义一种方法来记录日志，可以使用操作系统的 API（如 Windows 的 Event Log 或 CLFS），也可以自己写文件。但无论使用哪种方法，都最好将其封装为简单的类或函数，将存储日志的细节隐含起来，使程序员只要通过一个简单的函数调用就可以添加日志记录，比如：

```
void Log(LPCTSTR lpszModule, UINT nLogType, LPCTSTR lpszMessage);
```

封装好的类或函数应该以公共模块的方式发布给所有开发人员。使用统一的方法来添加日志记录，既有利于实现日志的集中存储，又简化了程序员写日志所需的工作量，防止他们对写日志产生厌烦甚至抵触情绪。

27.2.2 输出调试信息

输出调试信息是另一种常用的调试手段，与写日志相比，它具有更加简单快捷的优点，通常需要的时间和空间开销也更小。输出调试信息的常见方式有如下几种。

1. 使用类似 `print` 这样的函数直接打印到某个输出设备。比如在 DOS 和 Windows 的控制台程序中使用 `print` 语句，可以将信息直接打印到屏幕或控制台窗口。这种方法也被使用在某些嵌入式和移动设备（手机）的开发中。
2. 使用操作系统的 API 将调试信息输出到调试器或者专门的工具，典型的例子是 Windows 的 `OutputDebugString` 函数。因为 `OutputDebugString` 是通过 `RaiseException` API 产生的一个特殊的异常来发布调试信息，所以，过于频繁的调用这个函数会对软件的性能产生影响。
3. 使用编译器提供的宏来打印调试信息，比如 MFC 类库提供的 `TRACE` 宏，`n` 可以为 0、1、2、3，代表格式字符串中包含的参数个数，这几个宏实际上都是调用 `AfxTrace` 函数。`AfxTrace` 函数将要输出的信息格式化为一个大的字符串后送给一个名为 `afxDump` 的全局变量。`afxDump` 是 `CDumpContext` 类的一个全局实例。

最终, `afxDump` 通过 `OutputDebugString` 将调试信息输出给调试器。类似的, ATL 类库提供了 `ATLTRACE2` 宏。TRACE 宏的最重要特征就是只在调试版本中有效, 当编译发布版本时, 它们会被自动替换为空 (编译为`(void) 0`)。

合理地使用调试信息输出有利于提高软件的可调试性, 但应该注意以下几点。

1. 因为可能有很多个进程和线程都向调试器输出调试信息, 使得很多信息混杂在一起, 难以辨认, 所以输出调试信息时应该附加必要的上下文信息 (线程、函数名等), 以提高信息的价值。
2. 合理安排输出信息的代码位置, 认真选择要输出的内容 (变量值、位置等), 不要输出含糊不清的信息。要适当控制输出信息的数量, 如果输出的信息太多, 有时反而适得其反。例如, 在手机等嵌入式设备开发中, 输出的信息通常被显示在很小的屏幕上, 新的信息会将旧的信息覆盖掉, 所以, 如果输出的信息过于频繁, 那么有用的信息很可能被后来的没什么价值的信息所冲掉。
3. 因为调试信息输出通常是不被保存的, 而且 `TRACE` 宏在发布版本中是被自动移除的, 所以不能因为输出调试信息而忽视了记录日志。

在软件的架构设计阶段, 应该根据软件产品的实际情况选择合适的方法, 并将决定写入到项目的开发规范中。

27.2.3 转储 (Dump)

所谓转储就是将内存中的软件状态输出到文件或者其他设备 (如屏幕等) 上。常见的转储有。

- 对象转储, 对某个内存对象的状态 (属性值) 进行转储。
- 应用程序转储, 对应用程序用户空间中的关键状态信息转储, 包括每个线程的栈、进程的环境信息、进程和线程的状态等。使用 Windows 的 `MiniDumpWriteDump` API 可以很方便地将一个进程的当前状态转储到一个文件中。当应用程序发生严重错误时, 系统会自动为其产生转储文件, 但是也可以在其他时候产生转储, 产生转储并不意味着程序就要终止 (参见 12.9.3 节)。
- 系统转储, 即对整个系统的状态进行转储, 比如发生蓝屏时所产生的转储 (参见 13.3 节)。

可以把转储看作是对软件的拍照, 它记录了被转储对象在转储那一瞬间的真实情况。完全的系统转储包含了内存中的所有数据, 可以为调试提供丰富的信息。转储的另一个有用特征就是它可以将某一瞬间的状态永远保存下来, 然后发送和传递到任何地方, 这对于产品期调试和那些无法亲临现场进行调试的情况非常有价值。另外, 转储操作非常适合软件来自动生成, 因此, 在设计崩溃处理或自动错误报告功能时可以将其作为收集错误现场的一种方法。

MFC 的基类 CObject 定义了 Dump 方法用于实现对象转储，该方法的默认实现如下：

```
void CObject::Dump(CDumpContext& dc) const
{
    dc << "a " << GetRuntimeClass()->m_lpszClassName <<
        " at " << (void*)this << "\n";
    UNUSED(dc); // unused in release build
}
```

派生类通常会重新实现这个方法，输出更多的状态信息。例如以下是 CDialog 类的 Dump 方法所输出的信息：

a CDialo	g at \$12FE1C	//对象的类名和内存位置
m_hWnd = 0x7200AC (perma	nent window)	//窗口句柄值
caption = "D4D Testing"		//窗口标题
class name = "#32770"		//窗口类的名称
rect = (L 221, T 142, R 803, B 597)		//窗口的坐标
parent CWnd* = \$0		//父窗口对象的地址
style = \$94C800C4		//窗口风格
m_lpszTemplateName = 102		//窗口资源的模板 ID
m_hDialogTemplate = 0x0		//对话框资源的句柄
m_lpDialogTemplate = 0x0		//对话框资源的地址
m_pParentWnd = \$0		//父窗口对象的地址
m_nIDHelp = 0x66		//帮助信息的 ID

其中第 1 行是 CObject 类输出的，第 3~8 行是 CWnd 类输出的，最后 5 行是 CDialog 类输出的。

27.2.4 基类

当设计软件架构时，可以通过定义统一的基类来传达设计理念和强化设计规范。比如设计一些公共方法，以方便派生类的实现，也可以设计一个纯虚的方法来要求每个需要实例化的派生类都必须实现这个方法。

```
class D4D_API CD4dObject //D4D 代表 Design for Debug
{
public:
    CD4dObject(); //构造函数
    virtual ~CD4dObject(); //析构函数

    void Log(LPCTSTR szModule, LPCTSTR szFunction, //日志方法
             UINT uLogLevel, LPCTSTR szMsgFormat, ...);
    void Msg(UINT nResID, ...); //消息提示

    virtual DWORD UnitTest (DWORD dwParaFlags) = 0; //单元测试
    virtual DWORD Dump(HANDLE hFile); //对象转储
};
```

以上是一个示意性的基类定义，其中的 Log 方法用来提供日志功能，Msg 方法用于输出用户可见的消息，UnitTest 方法用来支持单元测试，它是纯虚的，因此要求非虚派生类必须实现它，Dump 方法用于支持对象转储。

27.2.5 调试模型

当设计软件架构时，应该考虑如何来调试这个软件，包括开发期调试的方法和产品期调试方法。如下问题特别值得注意。

1. 对于不能独立运行的库模块，如 DLL 和静态库，应该设计一个简单的可执行程序（EXE）专门供调试使用，我们把这样的程序称为靶子程序。利用靶子程序，开发者可以测试和调试不能直接运行的库模块，不必等待项目中真正使用这个模块的程序开发后才能调试；另外，出现 Bug 时也容易定位和排除。当然集成测试阶段，需要与真实的模块工作在一起。
2. 对于被系统或者其他模块自动启动的程序，最好要设计一个特别的命令行开关，使其支持手动启动，因为调试时手动启动更方便。为了启动被调试程序而执行一大堆操作必然会影响调试的效率。
3. 对于依赖于小概率事件（比如系统崩溃）触发才开始工作的模块，应该设计一个工具软件，使用这个工具可以方便地触发这个事件并开始调试。
4. 对于需要硬件配合才能调试的程序，如果这个硬件比较昂贵或稀缺，那么需要很多人共享一台，或者这个硬件开启成本较高，那么最好编写一个模拟器程序，这个模拟器程序可以模拟硬件的行为，输出真实硬件所产生的数据。利用模拟器，程序员不仅可以在没有硬件设备的情况下进行调试，而且可以利用模拟器来模拟硬件设备不支持的功能以辅助调试，比如可以让模拟器停止在某个状态或者慢速工作以配合调试。
5. 如果软件的某些功能难以通过普通的手工测试来发现问题，那么应该设计专门的测试工具，这些工具有利于测试，对调试也是有帮助的。

综上所述，应该为每个模块设计一种简便的调试方式，使程序员可以方便地调试他所负责的模块，这有利于提高程序员进行调试的积极性，使用调试方法解决问题和提高代码质量。

架构设计是软件开发中关键而且复杂的任务，本节介绍的内容仅供架构师们参考。

27.3 通过栈回溯实现可追溯性

在第 22 章我们介绍了栈的基本原理和栈上数据的布局。在基于栈的计算机系统（今天的大多数计算机系统）中，栈中详细记录了线程的执行过程，包括函数的参数、局部变量和返回地址等信息，这些信息反映了线程在函数一级的执行路线，这对于追溯代码执行轨迹和追踪问题根源有着重要意义。因为越晚被调用的函数离栈顶越近，所以通过栈信息生成函数调用记录时，是从栈顶开始向栈底追溯，因此这个过程被称为栈回溯（Stack Backtrace）。本节我们将讨论如何利用栈回溯来实现代码的可追溯性。

27.3.1 栈回溯的基本原理

对于基于栈的计算机系统，栈是进行函数调用的必须设施，因为函数调用指令（如 call）需要将函数返回地址压入到栈中，而函数返回指令（如 ret）就是通过这个地址知道要返回到哪里。除了函数返回地址，如果一个函数使用的调用规范需要通过栈来传递参数，那么栈上还会有调用这个函数的参数。栈也是分配局部变量的主要场所。这样一来，对于一个工作中的线程，每个尚未返回的函数在栈上后会有一个数据块，这个数据块至少包含它的返回地址，还可能包含参数和局部变量，这个数据块即所谓的栈帧（Stack Frame）。每个尚未返回的函数都拥有一个栈帧，按照函数调用的先后顺序，从栈底向栈顶依次排列。

因为每个函数需要在栈上存储信息的数量不固定，所以每个栈帧的长度是不固定的，这就使得定位每个函数的栈帧起止位置有时可能很困难。为了记录各个栈帧的位置，x86 CPU 配备了一个专门的寄存器，即 EBP（Extended Base Pointer）。通常 EBP 寄存器的值就是当前函数栈帧的基准地址。所谓基准地址，是指用来标识这个栈帧的一个参考地址。有了这个基准地址，就可以使用它来索引参数和局部变量，比如使用 EBP + 4 来索引函数返回值，EBP + 8 来索引放在栈上的第一个参数，EBP - XXX 来索引局部变量。

通过 EBP 寄存器通常可以知道当前函数的栈帧，那么如何找到上一个函数的栈帧呢？简单的回答是，在当前栈帧的基准地址处记录了上一个栈帧地址的值。以第 22 章中的图 22-3 所示的情况为例，0x0012ff74 是 FuncC 函数的栈帧基准地址，这个地址的内容是 0x0012ff80，那么地址 0x0012ff80 就是上一个函数（Main 函数）的栈帧基准地址。依此类推，可以逐级找到前一个函数的栈帧。

影响栈回溯的一个因素就是帧指针省略，即通常所说的 FPO。因为在栈上保存帧指针至少需要执行一条压入操作（push ebp），所以作为一项优化措施，编译器在编译某些短小的函数时，可能不更新 EBP 寄存器，不为这个函数建立独立的栈帧。对于这样被 FPO 优化的函数，栈上尽管还有它的返回地址信息和可能的局部变量等数据，但是由于它的栈帧基准地址没有被保存到栈上，也没有 EBP 寄存器指向它，所以就给栈回溯带来了困难，这是就需要符号文件中的 FPO 数据的帮忙，否则关于这个函数的调用就会被跳过。不过，因为现代 CPU 的强大性已经大大淡化了 FPO 优化的意义，所以很多新的软件都不再启用这种优化方法，这使得因为 FPO 带来的栈回溯问题会慢慢减少。

了解了上面的知识后，可以归纳出栈回溯的基本算法。

1. 取得标识线程状态的上下文结构（CONTEXT）。当有异常发生时，系统会创建这样的结构记录发生异常时的状态。使用 RtlCaptureContext 和 GetThreadContext API 可以在没有发生异常时取得线程的 CONTEXT 结构。
2. 通过 CONTEXT 结构或直接访问寄存器，取得程序指针寄存器（EIP）的值，通过它可以知道线程的当前执行位置，然后搜索这个位置附近的符号（SymFromAddr），

可以知道所在函数的函数名。

3. 通过 CONTEXT 结构或者直接访问寄存器取得当前栈帧的基准地址，在 x86 系统中，如果没有使用 FPO，那么 EBP 寄存器的值就是栈帧基地址。
4. 栈帧基地值向上偏移一个指针宽度（32 位系统，即 4 字节）的位置是函数的返回地址。紧接着便是放在栈上的参数，具体个数因为函数原型和调用规范而不同。
5. 搜索函数返回地址的邻近符号，可以找到父函数的函数名对应的源文件名等信息。
6. 当前栈帧基准地址处保存的是前一个栈帧的值，取出这个值便得到上一个栈帧的基地址，回到第 4 步循环，直到取得的栈帧基地址等于 0。

根据上面的算法，可以自己编写代码来实现栈回溯，也可以借助 Windows 的 API，下面分别介绍使用 DBGHELP 函数和 RTL 函数的方法。

27.3.2 利用 DBGHELP 函数回溯栈

DbgHelp 系列函数是 Windows 平台中用来辅助调试和错误处理的一个函数库，其主要实现位于 DbgHelp.DLL 文件中，因此通常被称为 DbgHelp 函数库。DbgHelp 为实现栈回溯提供了如下支持。

- StackWalk64 和 StackWalk 函数，用于定位栈帧和填充栈帧信息，包括函数返回值、参数等。
- 调试符号支持，包括初始化调试符号引擎，加载符号文件，设置符号文件搜索路径，寻找符号等。
- 模块和映像文件，包括枚举进程中的所有模块，查询某块的信息等。

为了演示如何使用 DbgHelp 函数库来回溯栈，我们编写了一个名为 CCallTracer 的 C++ 类，完整的代码位于 code\chap27\D4D 目录中，清单 27-1 给出了 CCallTracer 类的 WalkStack 方法的源代码。

清单 27-1 WalkStack 函数

```

1  HRESULT CCallTracer::WalkStack(PFN_SHOWFRAME pfnShowFrame,
2                                PVOID pParam, int nMaxFrames)
3  {
4      HRESULT hr=S_OK;
5      STACKFRAME64 frame;           // 描述栈帧信息的标准结构
6      int nCount=0;
7      TCHAR szPath[MAX_PATH];
8      DWORD dwTimeMS;
9
10     dwTimeMS=GetTickCount();       // 记录开始时间
11
12     RtlCaptureContext(&m_Context); // 获取当前的上下文
13
14     memset(&frame, 0x0, sizeof(frame));
15     // 初始化起始栈帧
16     frame.AddrPC.Offset    = m_Context.Eip;

```

```

17     frame.AddrPC.Mode      = AddrModeFlat;
18     frame.AddrFrame.Offset = m_Context.Ebp;
19     frame.AddrFrame.Mode   = AddrModeFlat;
20     frame.AddrStack.Offset = m_Context.Esp;
21     frame.AddrStack.Mode   = AddrModeFlat;
22
23     while (nCount < nMaxFrames)
24     {
25         nCount++;
26         if (!StackWalk64(IMAGE_FILE_MACHINE_I386,
27                           GetCurrentProcess(), GetCurrentThread(),
28                           &frame, &m_Context,
29                           NULL,
30                           SymFunctionTableAccess64,
31                           SymGetModuleBase64, NULL))
32         {
33             hr = E_FAIL; // 发生错误, StackWalk64 函数通常不设置 LastError
34             break;
35         }
36         ShowFrame(&frame, pfnShowFrame, pParam);
37         if (frame.AddrFrame.Offset == 0 || frame.AddrReturn.Offset == 0)
38         {
39             // 已经到最末一个栈帧, 遍历结束
40             break;
41         }
42     }
43
44     // 显示归纳信息
45     _stprintf(szPath, _T("Total Frames: %d; Spend %d MS"),
46                nCount, GetTickCount() - dwTimeMS);
47     pfnShowFrame(szPath, pParam);
48
49     // 显示符号搜索路径
50     SymGetSearchPath(GetCurrentProcess(), szPath, MAX_PATH);
51     pfnShowFrame(szPath, pParam);
52
53     return hr;
54 }
```

其中第 12 行是调用 RtlCaptureContext API 来取得当前线程的上下文信息，即 CONTEXT 结构。尽管函数名中包含 64 字样，但是 StackWalk64 函数也可以用在 32 位的系统中。需要说明的一点是，在 RtlCaptureContext 返回的 CONTEXT 结构中，其程序指针和栈栈帧等寄存器的值对应的都是上一级函数的，即调用 WalkStack 函数的那个函数。

第 14~21 行是初始化 frame 变量，它是一个 STACKFRAME64 结构（见清单 27-2）。

清单 27-2 描述栈帧的 STACKFRAME64 结构

```

typedef struct _tagSTACKFRAME64 {
    ADDRESS64 AddrPC;           //程序指针, 即当前执行位置
    ADDRESS64 AddrReturn;        //返回地址
    ADDRESS64 AddrFrame;         //栈帧地址
    ADDRESS64 AddrStack;         //栈指针值, 相当于 ESP 寄存器的值
    ADDRESS64 AddrBStore;        //安腾架构使用的 Backing Store 地址
    PVOID FuncTableEntry;        //指向描述 FPO 的 FPO_DATA 结构或 NULL
    DWORD64 Params[4];          //函数的参数
    BOOL Far;                   //是否为远调用
}
```

```

    BOOL Virtual;           //是否为虚拟栈帧
    DWORD64 Reserved[3];   //保留
    KDHELP64 KdHelp;        //用来协助遍历内核态栈
} STACKFRAME64, *LPSTACKFRAME64;

```

其中，前 4 个成员分别用来描述程序计数器（Program Counter，即程序指针）、函数返回地址、栈帧基准地址、栈指针和安腾 CPU 所使用的 Backing Store 的地址值，它们都是 ADDRESS64 结构：

```

typedef struct _tagADDRESS64 {
    DWORD64 Offset;          //地址的偏移部分
    WORD Segment;            //段
    ADDRESS_MODE Mode;       //寻址模式
} ADDRESS64, *LPADDRESS64;

```

其中 ADDRESS_MODE 代表寻址方式，可以为 AddrMode1616(0)、AddrMode1632(1)、AddrModeReal(2) 和 AddrModeFlat(3) 4 个常量之一。

STACKFRAME64 结构的 FuncTableEntry 字段指向的是 FPO 数据（如果有），Params 数组是使用栈传递的前 4 个参数，应该根据函数的原型来判断与实际参数的对应关系。如果栈帧对应的是一个 WOW 技术中的长调用，那么 Far 字段的值为真，WOW（Windows 32 On Windows 64 或 Windows 16 On Windows 32）是在高位宽的 Windows 系统中运行低位宽的应用程序所使用的机制，比如在 64 位的 Windows 系统中执行 32 位的应用程序。如果是虚拟的栈帧，那么 Virtual 字段为真。KdHelp 字段供内核调试器产生内核态栈回溯时使用。

第 23~42 行是一个 while 循环，每次处理一个栈帧。第 26~31 行是调用 StackWalk64 API，将初始化了的 frame 结构和 context 结构传递给这个函数。StackWalk64 的后 4 个参数可以指定 4 个函数地址，目的是为 WalkStack64 函数分别提供以下 4 种帮助：读内存、访问函数表、取模块的基址和翻译地址，当 WalkStack64 需要某种帮助时，那么会调用相应的函数（如果不为空）。

第 36 行调用 ShowFrame 方法来显示一个栈帧的信息。第 37 行是循环的正常出口，即回溯到最后一个栈帧时，那么它的帧指针的值为 0，这时这个栈帧的函数返回地址也为空，因为每个线程的第一个函数的栈帧不是因为函数调用而开始执行的。

第 45~47 行用来显示统计信息。其中 pfnShowFrame 是参数中指定的一个函数指针，WalkStack 方法通过这个函数把信息汇报给调用者。第 50 和 51 行显示符号文件的搜索路径。在 CCallTracer 类的构造函数中它会调用 SymInitialize 函数来初始化符号引擎，即：

```

SymSetOptions(dwOptions | SYMOPT_LOAD_LINES
             | SYMOPT_DEFERRED_LOADS
             | SYMOPT_OMAP_FIND_NEAREST);
bRet = SymInitialize(GetCurrentProcess(), NULL, TRUE);

```

因为我们在符号搜索路径参数中指定的是 NULL，所以 DbgHelp 会使用当前路径以及环境变量_NT_SYMBOL_PATH 和 _NT_ALTERNATE_SYMBOL_PATH 的内容作为搜索路

径。在笔者的系统中，51 行显示的内容是：

```
.;SRV*d:\symbols*http://msdl.microsoft.com/download/symbols
```

为了测试 CCallTracer 类，我们编写了一个 MFC 对话框程序（D4dTest.EXE），当用户点击界面上的 Stack Trace 按钮时，前面介绍的 WalkStack 方法会被调用，清单 27-3 摘录了部分执行结果。

```
void CD4dTestDlg::OnStacktrace()
{
    CCallTracer cs;
    cs.WalkStack(ShowStackFrame, this, 1000);
}
```

清单 27-3 CCallTracer 类显示的栈回溯信息

```
1 -----
2 Child EBP: 0x0012f648, Return Address: 0x5f43749c
3 Module!Function: D4dTest!CD4dTestDlg::OnStacktrace
4 Parameters: (0x0012f8e8,0x00000000,0x00144728,0x00000000)
5 C:\dig\dbg\author\code\chap27\d4d\Debug\CD4dTestDlg.cpp
6 c:\dig\dbg\author\code\chap27\d4d\Debug\CD4dTest.exe
7 C:\dig\dbg\author\code\chap27\d4d\Debug\CD4dTest.pdb
8 Far (WOW): 0; Virtual Frame: 1
9 -----
10 【省略关于中间 35 个栈帧的很多行】
11 -----
12 Child EBP: 0x0012ffff0, Return Address: 0x00000000
13 Module!Function: kernel32!BaseProcessStart
14 Parameters: (0x00402740,0x00000000,0x00000000,0x00000000)
15 C:\WINDOWS\system32\kernel32.dll
16 d:\symbols\kernel32.pdb\b62A5E0D6EC649ACB3ED74E9CE5701832\kernel32.pdb
17 FPO: Para dwords: 1; Regs: 0; Frame Type: 3
18 Far (WOW): 0; Virtual Frame: 1
19 Total Frames: 37; Spend 20297 MS
20 .;SRV*d:\symbols*http://msdl.microsoft.com/download/symbols
```

为了节约篇幅，清单 27-3 只保留了第一个栈帧和最后一个栈帧的信息。第 12 行是被追溯的最后一个栈帧，其返回地址为 0。最后一个栈帧的基址为 0012ffff0，使用调试器可以看到这个地址的值为 0。使用这个特征或者返回地址等于 0 可以判断是否到了最后一个栈帧（清单 27-2 的第 37 行）。

上面的例子是为当前进程的当前线程产生回溯，事实上，也可以为另一个进程中的某个线程来产生栈回溯，比如调试器显示被调试程序中的函数调用序列（Calling Stack）就属于这种情况。

关于使用 DbgHelp 函数来进行栈回溯，还有以下几点值得注意。首先是 DbgHelp 库的版本，Windows XP 预装了一个较老版本的 DbgHelp.DLL，使用这个版本有很多问题，比如它会使用 DLL 的输出信息作为符号的来源，而且调用 SymGetModuleInfo64 这样的函数会失败。这时，得到的栈帧信息可能残缺不全或有错误。比如以下是使用老版本的 DbgHelp.DLL 时，D4dTest.EXE 程序得到的最后一个栈帧信息：

```
Child EBP: 0x0012ffff0, Return Address: 0x00000000
Module!Function: Unknown!RegisterWaitForInputIdle
```

```
Parameters: (0x00402770, 0x00000000, 0x00000000, 0x00000000)
```

可见没有找到合适的模块信息，解决的办法是将新版本的 DbgHelp.DLL 和 EXE 文件放在同一个目录下。WinDBG 工具包中包含的 DbgHelp.DLL 是比较新的。

使用了新版本的 DbgHelp.DLL 后，大多数栈帧的信息都没问题了，但是个别栈帧还有问题，比如，最后一个栈帧：

```
Child EBP: 0x0012fff0, Return Address: 0x00000000
Module!Function: kernel32!RegisterWaitForInputIdle
Parameters: (0x00402770, 0x00000000, 0x00000000, 0x00000000)
C:\WINDOWS\system32\kernel32.dll
SymType:-exported-;PdbUnmthd:0,DbgUnmthd:0,LineNos:0,GlblSym: 0,TypeInfo:0
Far (WOW): 0; Virtual Frame: 1
```

上面的函数名显然有错误，符号的类型为“输出”，可见没有找到合适的 PDB 文件。符号搜索路径中不是指定了 SRV 格式的本地符号库和符号服务器么？为什么还没有找到 kernel32.dll 的 PDB 文件呢？原因是 DbgHelp.DLL 没有找到 symsrv.dll。将这个文件也复制到 D4dTTest.EXE 文件所在目录就没有这个问题了，显示的信息即如清单 27-3 所示，从第 16 行关于 kernel32.pdb 的全路径中可以看出，symsrv.dll 在本地符号库中找到了合适的符号。但是，一旦使用了 symsrv.dll，它就会检索符号库并可能通过网络链接远程的服务器，这通常要花费较多时间。因此本小节介绍的方法适合处理程序崩溃或者个别的情况，不适合在程序的执行过程中频繁记录某一事件的踪迹信息，接下来我们将介绍一种负荷很小的快速记录方法。

27.3.3 利用 RTL 函数回溯栈

在第 23 章（23.7 节）介绍 Win32 堆的调试支持时，我们介绍了用户态栈回溯（User-Mode Stack Trace），简称 UST。一旦启用了 UST 机制后，当内存分配函数再被调用时，堆管理器会将函数调用信息（栈回溯信息）保存到一个被称为 UST 数据库的内存区中。然后使用 UMDH 或 DH 就可以得到栈回溯记录，例如：

```
00009DD0 bytes in 0x1 allocations (@ 0x00009D70 + 0x00000018) by: BackTrace00803
    7C96D6DC : ntdll!RtlDebugAllocateHeap+000000E1
    7C949D18 : ntdll!RtlAllocateHeapSlowly+00000044
    7C91B298 : ntdll!RtlAllocateHeap+00000E64
    1020DE9C : MSVCRTD!_heap_alloc_base+0000013C
...
```

UST 的最大特征是直接记录函数的返回地址，而不是它的符号。将函数地址转换为符号的工作留给 UMDH 这样的工具来做，这样便大大节约了查找和记录符号所需的时间和空间。UST 机制主要是由 NTDLL.DLL 中的以下函数实现的。

- RtlInitializeStackTraceDataBase，负责初始化 UST 数据库。
- RtlLogStackBackTrace，负责发起记录栈回溯信息，它会调用 RtlCaptureStackBackTrace 收集栈信息，然后将其写到由全局变量 RtlpStackTraceDataBase 所标识的 UST 数据库中。

- RtlCaptureStackBackTrace 负责调用 RtlWalkFrameChain 来执行真正的信息采集工作。

尽管以上函数没有公开文档化，但因为 NTDLL.DLL 输出了以上所有函数，因此还是可以调用它们。利用这些函数，应用程序也可以使用 UST 机制来记录重要操作的函数调用记录。为了演示其用法，我们在 D4D 程序中设计了一个 CFastTracer 类，完整代码位于 code\chap27\D4D 目录中。

RtlWalkFrameChain 用于获取栈帧中的函数返回地址，它的原型为：

```
ULONG RtlWalkFrameChain (PVOID *pReturnAddresses, DWORD dwCount, DWORD dwFlags);
```

其中，参数 pReturnAddresses 指向一个指针数组，用来存放每个栈帧中的函数返回地址，第二个参数是这个数组的大小，第三个参数用于指定标志值，可以为 0。根据函数原型可以定义一个函数指针类型：

```
typedef ULONG (WINAPI *PFN_RTLWALKFRAMECHAIN) (PVOID *pReturnAddresses,
    DWORD dwCount, DWORD dwFlags);
```

然后使用 GetProcAddress API 取得 RtlWalkFrameChain 函数的地址。

```
hNtDll=LoadLibrary ("NTDLL.DLL");
m_pfnWalkFrameChain=(PFN_RTLWALKFRAMECHAIN)
    GetProcAddress(hNtDll,"RtlWalkFrameChain");
```

接下来就可以通过这个函数指针来调用 RtlWalkFrameChain 函数了，在 D4dTest 程序中调用 CFastTracer 的 GetFrameChain 方法得到的结果如下：

```
Return Address[0]: 0x1000326f
Return Address[1]: 0x004024a2
...
Return Address[37]: 0x7c816fd7
```

使用 DbgHelp 函数加载了符号后，便可以得到这些地址所属的函数和模块名称。

得到了函数返回地址信息后，接下来要解决的问题是如何记录这些信息。根据具体情况，可以记录在应用程序自己维护的文件中，也可以复用 UST 数据库。如果使用 UST 数据库，那么应该先调用 RtlInitializeStackTraceDataBase 函数初始化 UST 数据库。当每次需要添加记录时，可以调用 RtlLogStackBackTrace 函数，这个函数会将当时的函数调用记录记在 UST 数据库中。使用 UMDH 这样的工具便可以从 UST 数据库中读取记录。

27.4 数据的可追溯性

在调试时，我们经常诧异某个变量的值怎么变成这个样子，想知道是哪个函数在何时将其修改成出乎预料的值。有时我们也希望知道一个变量取值的变化过程，它曾经取过哪些值，或者在过去的某个时间，它的取值是什么。要解决这些问题，就要提高数据的可追溯性，也就是记录数据的修改经过和变化过程，使其可以查询和追溯。

因为函数调用时栈上记录了被调用函数的返回地址，这为实现代码的可追溯性提供了一个很好的基础。但对于数据的可追溯性，目前的计算机架构所提供的支持还很有限。CPU 的数据断点功能可以算是其中一个。第 4 章介绍过，在软件调试时，可以对感兴趣的数据设置硬件断点，此后，当这个数据再被访问时，CPU 便会发出异常而中断到调试器。那么能否在非调试情况下利用 CPU 的数据断点功能来监视变量呢？答案是肯定的。下面我们就介绍这种依赖于 CPU 的数据断点功能来监视数据并记录其访问经过的方法。

27.4.1 基于数据断点的方法

简单来说，这种方法的原理就是将要监视的变量的地址以断点的形式设置到 CPU 的调试寄存器中，这样，当这个变量被访问时，那么 CPU 便会报告异常，而后应该在程序中捕捉这个异常并做必要的分析记录。记录可以包含被访问的变量名称、访问时间、访问代码的地址（即触发断点的代码地址）等简要信息，还可以根据异常中的上下文结构进行栈回溯从而得到访问这个变量的函数调用过程，最后把得到的信息记录下来。

以上过程的一个难点就是如何捕捉异常。如果程序在被调试，那么数据断点导致的异常会先发给调试器，调试器会处理这个异常，因此应用程序自己的代码是察觉不到这个异常的。当没有调试器时，数据断点异常会发给应用程序，如果没有得到处理，那么就会导致应用程序崩溃而结束。那么应用程序应该如何处理数据断点异常呢？使用结构化异常处理器或者 C++ 的异常处理器显然有很多问题，因为不知道什么代码会访问被监视的变量而触发异常，连哪个线程都不确定，所以难以选择这些异常处理器的设置位置。幸运的是，可以通过 Windows XP 引入的向量化异常处理（VEH）机制来解决这个问题。因为一旦注册了 VEH 处理器，那么进程中所有线程导致的异常都会发给 VEH 处理器，VEH 处理器不处理后才交给结构化异常处理器或者 C++ 异常处理器。这样，只要我们注册了一个 VEH 处理器，当它被调用后，先判断是否是数据断点异常，如果不是，那么便返回 EXCEPTION_CONTINUE_SEARCH 交给 SHE 去处理，如果是，那么说明有人访问了被监视的变量。清单 27-4 给出了实现这一逻辑的 VEH 处理器的简单代码。

清单 27-4 接收数据断点异常的 VEH 处理器

```
LONG WINAPI DataTracerVectoredHandler( struct _EXCEPTION_POINTERS
*ExceptionInfo )
{
    if(ExceptionInfo->ExceptionRecord->ExceptionCode==0x40010006L)
        return EXCEPTION_CONTINUE_SEARCH;           //参见 26.4.2

    if(ExceptionInfo->ExceptionRecord->ExceptionCode
       ==STATUS_SINGLE_STEP                  //0x80000004L
       && g_pDataTracer!=NULL)
    {
        g_pDataTracer->HandleEvent(ExceptionInfo);
        return EXCEPTION_CONTINUE_EXECUTION;      //继续执行触发断点的代码
    }
    return EXCEPTION_CONTINUE_SEARCH;
}
```

数据断点的异常代码与单步执行是一样的，即 0x80000004L。如果希望严格判断

是否是数据断点，那么应该判断上下文结构中的 DR6 寄存器，即 ExceptionInfo->ContextRecord->Dr6。清单中，pDataTracer 是 CDataTracer 类的实例，这个类封装了设置断点和处理断点事件等功能，清单 27-5 给出了它的定义。

清单 27-5 演示数据追溯功能的 CDataTracer 类

```

class D4D_API CDataTracer
{
public:
    HRESULT HandleEvent(struct _EXCEPTION_POINTERS * ExceptionInfo);
    ULONG GetDR7(int nDbgRegNo, int nLen, BOOL bReadWrite);
    HRESULT StartTrace();                                //启动监视功能
    BOOL IsVarExisted(ULONG ulVarAddress);              //判断指定的变量是否在被监视
    HRESULT RemoveVar(ULONG ulAddress);                  //移除一个变量
    HRESULT AddVar(ULONG ulVarAddress,int nLen, int nReadWrite); //增加要监视变量
    CDataTracer();
    virtual ~CDataTracer();
    void ShowString(LPCTSTR szMsg);
    HRESULT ClearAllDR();                               //清除所有调试寄存器，停止监视
    void SetListener(HWND hListBox);
protected:
    HRESULT RegVeh();                                 //注册 VEH 处理器
    HRESULT UnRegVeh();                               //注销 VEH 处理器
    ULONG m_VarAddress[DBG_REG_COUNT];                //记录被监视的变量地址
    ULONG m_VarLength[DBG_REG_COUNT];                 //记录被监视的长度
    ULONG m_VarReadWrite[DBG_REG_COUNT];              //记录监视的访问方式
    PVOID m_pVehHandler;                            //VEH 处理器句柄
    HWND m_hListBox;                                //接收提示信息的列表框句柄
};

```

其中 m_VarAddress 用来记录要监视的变量地址，m_VarLength 用来记录要监视变量的长度，可以为 0(1 字节)、1(2 字节)、3(4 字节)3 个值之一，m_VarReadWrite 用来记录触发断点的访问方式，可以等于 1(只有写时触发)或者 3(读写都触发)。

成员 m_hListBox 用来记录接收访问记录，出于演示目的，我们只是将访问记录输出到一个列表框中。SetListener 方法用来设置 m_hListBox 的值。

AddVar 方法用来添加要监视的变量，x86 架构支持最多监视 4 个变量。StartTrace 方法用来启动监视，也就是将记录在成员变量 m_VarXXX 中的变量信息设置到 CPU 的调试器中，其源代码如清单 27-6 所示。

清单 27-6 设置数据断点的源代码

```

1  HRESULT CDataTracer::StartTrace()
2  {
3      CONTEXT ctxt;
4      HANDLE hThread=GetCurrentThread();
5
6      ctxt.ContextFlags=CONTEXT_DEBUG_REGISTERS;//|CONTEXT_FULL;
7      if(!GetThreadContext(hThread,&ctxt))
8      {
9          OutputDebugString("Failed to get thread context.\n");
10         return E_FAIL;
11     }
12     ctxt.Dr0=m_VarAddress[0];
13     ctxt.Dr1=m_VarAddress[1];

```

```

14     ctxt.Dr2=m_VarAddress[2];
15     ctxt.Dr3=m_VarAddress[3];
16
17     ctxt.Dr7=0;
18     if(m_VarAddress[0]!=0)
19         ctxt.Dr7|=GetDR7(0,m_VarLength[0],m_VarReadWrite[0]);
20     if(m_VarAddress[1]!=0)
21         ctxt.Dr7|=GetDR7(0,m_VarLength[1],m_VarReadWrite[1]);
22     if(m_VarAddress[2]!=0)
23         ctxt.Dr7|=GetDR7(0,m_VarLength[2],m_VarReadWrite[2]);
24     if(m_VarAddress[3]!=0)
25         ctxt.Dr7|=GetDR7(0,m_VarLength[3],m_VarReadWrite[3]);
26
27     if(!SetThreadContext(hThread,&ctxt))
28     {
29         OutputDebugString("Failed to set thread context.\n");
30         return E_FAIL;
31     }
32
33     if(m_pVehHandler==NULL && RegVeh()!=S_OK)
34         return E_FAIL;
35
36     return S_OK;
37 }
38 ULONG CDataTracer::GetDR7(int nDbgRegNo, int nLen, BOOL bReadWrite)
39 {
40     ULONG ulDR7=0;
41
42     ulDR7|= (BIT_LOCAL_ENABLE<<(nDbgRegNo*2));
43     // bit 0, 2, 4, 6 are for local breakpoint enable
44     //
45
46     // read write bits
47     if(bReadWrite)
48         ulDR7|=BIT_RW_RW<<(16+nDbgRegNo*4);
49     else
50         ulDR7|=BIT_RW_WO<<(16+nDbgRegNo*4);
51
52     ulDR7|=nLen<<(16+nDbgRegNo*4+2);
53
54     return ulDR7;
55 }

```

其中第 7 行是通过 GetThreadContext API 取得线程的上下文结构 (CONTEXT)，第 12~25 行是将 CONTEXT 结构中的调试寄存器值设置好，然后第 27 行通过 SetThreadContext API 将 CONTEXT 结构设置到硬件中。GetDR7 方法用来计算某个断点所需的 DR7 寄存器值，因为多个断点共用 DR7 寄存器，所以多个断点的设置被通过或操作集成在一起。关于 DR7 寄存器的细节，我们在第 4 章做过详细的介绍。

清单 27-7 处理数据断点事件的源代码

```

1  HRESULT CDataTracer::HandleEvent(_EXCEPTION_POINTERS *ExceptionInfo)
2  {
3      ULONG ulDR6;
4      TCHAR szMsg[MAX_PATH]=_T("CDataTracer::HandleEvent");
5
6      // check dr6 to see which break point was triggered.
7      ulDR6=ExceptionInfo->ContextRecord->Dr6;
8
9      for(int i=0;i<DBG_REG_COUNT;i++)

```

```

10     {
11         if( ( ulDR6&(1<<i) ) != 0 ) // bit i was set
12             _stprintf(szMsg, _T("Data at 0x%08x was accessed by code at 0x%08x."),
13             m_VarAddress[i], ExceptionInfo->ExceptionRecord->ExceptionAddress);
14     }
15     ShowString(szMsg);
16     CCallTracer ct;
17     ct.SetOptions(CALLTRACE_OPT_INFO_LEAN);
18     ct.WalkStack(ShowStackFrame, this, 1000, ExceptionInfo->ContextRecord);
19
20     return S_OK;
21 }

```

清单 27-7 列出了用来处理数据断点事件的 HandleEvent 函数的源代码。其中第 7 行先取出记录断点信息的 DR6 寄存器，因为单步执行和所有数据断点触发的都是一个异常（1 号），所以只有通过 DR6 寄存器才能判断出发生的到底是哪个断点（详见第 4 章）。第 9~14 行的 for 循环依次判断 DR6 的低 4 位，如果某一位为 1，那么则说明对应的断点被触发了。第 16~18 行是使用我们上一节介绍的 CCallTracer 类来显示栈回溯信息，也就是访问被监视变量的过程。

为了验证 CDataTracer 类的有效性，我们在 D4dTTest 程序中用它来监视 CD4dTTestDlg 类的 m_nInteger 成员。在 OnInitDialog 方法中，我们加入了如下代码：

```

if(g_pDataTracer==NULL)
{
    g_pDataTracer=new CDataTracer();
    g_pDataTracer->SetListener(m_ListInfo.m_hWnd);
    g_pDataTracer->AddVar((ULONG)&m_nInteger, 0, TRUE);
    g_pDataTracer->StartTrace();
}

```

因为 m_nInteger 是使用 MFC 的 DDX（Dialog Data Exchange）机制与界面上的编辑框绑定的，所以当点击界面上的 Assign 按钮（图 27-1），便会触发我们的监视机制，看到列表框中输出 nInteger 变量被访问的经过。

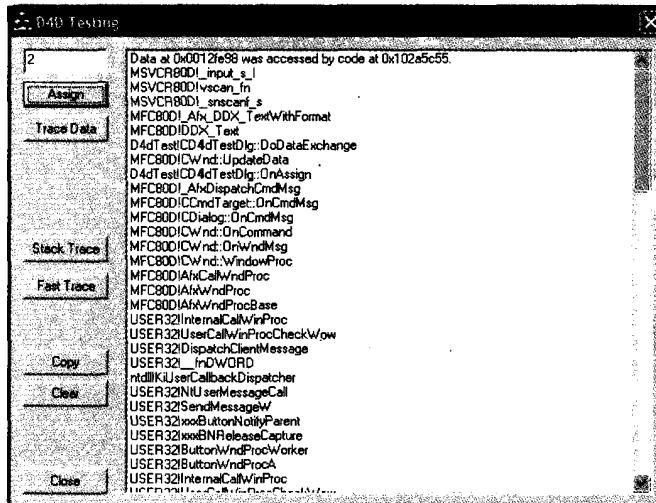


图 27-1 显示访问变量过程的 D4dTTest 程序

列表框中的信息告诉我们，在点击界面上的 Assign 按钮后，`m_nInteger` 变量被访问了两次（图中只显示出第一次），一次是被 `_AfxSimpleScanf` 函数访问，另一次是被 `CD4dTestDlg::OnAssign` 函数访问。输出的信息中包含了每次访问的详细过程，这证明了本方法的可行性。在实际使用时，可以考虑使用类似 `CFastTrace` 类的方法只记录函数的返回地址，这样可以大大提高速度。

需要说明的是，`CDataTracer` 类还只是为了满足演示目的而设计的，如果要用到实际的软件项目中，还需要做一些增强和完善，比如在每次收到数据断点事件时，最好显示出变量的当前值，这需要暂时禁止数据断点，否则，如果设置的访问方式是读写都触发，那么读取变量时又会触发断点导致死循环。

使用数据断点方法的一个不足是可以监视的变量数量非常有限，这是硬件平台所决定的。下面将介绍的基于对象封装技术的方法没有这个限制。

27.4.2 使用对象封装技术来追踪数据变化

通过对对象封装技术来追踪变量访问过程的基本思想是将要监视的数据封装在一个类中，然后通过运算符重载截获对变量的访问，并进行记录。记录的具体方法可以使用环形的缓冲区来循环记录变量的历史值。也可以维护一块专门的内存区。对于每次变量被访问时的栈回溯信息可以记录在 UST 数据库中。

为了演示这种方法，我们编写了一个用于追踪整形变量的 `CD4dInteger` 类，清单 27-8 给出了这个类的定义，成员 `m_pTracker` 用来指向保存历史值的环形缓冲区，`m_nTrackDepth` 代表这个缓冲区的长度。

清单 27-8 具有可追溯性的整数类型

```
class D4D_API CD4dInteger : public CD4dObject
{
protected:
    long m_nTrackerIndex;           //追踪数组的可用位置
    long * m_pTracker;             //记录变量历史值的追踪数组
    long m_nTrackDepth;            //追踪数组的长度
    long m_nCurValue;              //变量的当前值
public:
    long CurValue(){return m_nCurValue;};
    long GetTrace(int nBackStep);      //读取历史值
    long GetTrackDepth(){return m_nTrackDepth;};
    CD4dInteger& operator =(long nValue); //重载赋值运算符
    CD4dInteger(int nTrackDepth=1024);
    virtual ~CD4dInteger();
    virtual DWORD UnitTest (DWORD dwParaFlags);
    virtual DWORD Dump(HANDLE hFile);
};
```

因为重载了赋值运算符，所以当为 `CD4dInteger` 的实例赋值时，就会触发它的赋值运算符方法，即：

```
CD4dInteger& CD4dInteger::operator =(long nValue)
```

```

{
    int nIndex = InterlockedIncrement (&m_nTrackerIndex);
    nIndex %= m_nTrackDepth;

    this->m_nCurValue=nValue;
    this->m_pTracker[nIndex] = nValue;

    return *this;
}

```

这个方法除了更新当前值外，还在环形缓冲区找一个新的位置将当前值保存起来。这样环形缓冲区便记录下了这个数据的变化过程。如果要记录每次更新数据时的函数调用过程，那么只要在这个方法中加入记录栈回溯信息的代码。

27.5 可观察性的实现

当我们在调试软件时，经常有这样的疑问：在某一时刻，比如当程序发生错误或崩溃时，CPU 在执行哪个函数或函数的哪一部分？此时循环 L 已经执行了多少次？变量 A 的值是什么？这个时候进程中共有多少个线程？有多少个模块？动态链接库 M 是否被加载了？如果是，被加载的版本是多少？如此等等。

能否迅速找到这些问题的答案对调试效率有着直接的影响。很多时候，就是因为无法回答上面的某一个问题，使调试工作陷入僵局。然后，可能要花费数小时乃至数天的时间来修改软件，向软件中增加输出状态信息的代码，然后重新编译安装执行，再寻找答案。这种方法有时候被形象的称为“代码注入”，即向程序中注入用来打印软件状态或者其他辅助观察的代码。

因为每次注入代码都需要重新编译程序，因此这种方法的效率是比较低的。为了提高效率，在设计软件时就应该考虑如何使软件的各种特征可以被调试人员简便地观察到，即提高软件的可观察性。

27.5.1 状态查询

为了便于观察软件的内部运行状态，设计软件时应该考虑如何查询软件的内部状态，这对软件维护和调试乃至最终用户都是有帮助的。

提供状态查询的方式可以根据软件的具体特征而灵活设计，对于网站或网络服务（Web Service），可以通过网页的形式来提供。如果是简单的客户端软件，可以采用对话框的形式。

大多数设计完善的软件系统都会提供专门的工具供用户查询系统的状态，以 Windows 操作系统为例，使用任务管理器可以查询系统中运行的进程、线程和内存使用情况等信息；使用 driverquery 命令可以查询系统中加载的驱动程序；使用 netstat 命令可以查询系统的网络连接情况；使用设备管理器可以查询系统中各种硬件和设备驱动程序的工作情况，等等。

越大型的软件系统，状态查询功能越显得重要，因此在架构设计阶段就应该考虑如何支持状态查询功能，设计统一的接口和附属工具。对于中小型的软件可以考虑配备简单状态查询功能，比如一个对话框，里面包含软件的重要运行指标。

如果从设计阶段就将状态查询功能考虑进来，那么所需花费的开发投入通常并不大，但如果等到发生了问题再考虑如何增加这些功能，那么不仅要花费更多的精力，而且效果也很难做到“天衣无缝”。

除了设计专用的接口和查询方式外，也可以使用操作系统或者工业标准定义的标准方式来支持状态查询，比如后面介绍的 WMI 方式和性能计数器方式，使用标准方式的好处是可以被通用的工具所访问。

27.5.2 WMI

Windows 是个庞大的系统，如何了结系统中各个部件的运行状况并对它们进行管理和维护是个重要而复杂的问题。如果系统的每个部件都提供一个管理程序，那么不仅会导致很多的重复开发工作，而且也会影响系统的简洁性和性能。更好的做法是操纵系统实现并提供一套统一的机制和框架，其他部件只需按照一定的规范实现与自身逻辑密切相关的部分，WMI（Windows Management Instrumentation）便是针对这一目标设计的。

WMI 提供了一套标准化的机制来管理本地及远程的 Windows 系统，包括操作系统自身的各个部件及系统中运行的各种应用软件，只要它们提供了 WMI 支持。WMI 最早出现在 NT4 的 SP4 中，并成为其后所有 Windows 操作系统的必不可少的一部分。在今天的 Windows 系统中，很容易就可以看到 WMI 的身影，比如计算机管理控制台（Computer Management）、事件查看器、服务管理器（Services Console）等，事实上，这些工具都是使用 MMC（Microsoft Management Console）程序来提供用户接口。

从架构角度来看，整个 WMI 系统由以下几个部分组成。

- 受管对象（Managed Objects）：即要管理的目标对象，WMI 系统的价值就是获得这些对象的信息或配置它们的行为。
- WMI 提供器（WMI Providers）：按照 WMI 标准编写的软件组件，它代表受管对象与 WMI 管理器交互，向其提供数据或执行其下达的操作。WMI 提供器隐藏了不同受管对象的差异，使 WMI 管理器可以以统一的方式查询和管理受管对象。
- WMI 基础构件（WMI Infrastructure）：包括存储对象信息的数据库和实现 WMI 核心功能的对象管理器。因为 WMI 使用 CIM（Common Information Model）标准来描述和管理受管对象。因此，WMI 的数据库和对象管理器分别被命名为 CIM 数据仓库（CIM Repository）和 CIM 对象管理器（CIM Object Manager，简称 CIMOM）。
- WMI 编程接口（API）：WMI 提供了几种形式的 API 接口，以方便不同类型的 WMI 应用使用 WMI 功能，比如供 C/C++ 程序调用的函数形式（DLL，Lib 和头文

件), 供 VB 和脚本语言调用 ActiveX 控件形式和通过 ODBC 访问的数据库形式 (ODBC Adaptor)。

- WMI 应用程序 (WMI Applications): 即通过 WMI API 使用 WMI 服务的各种工具和应用程序。比如 Windows 中的 MMC 程序, 以及各种实用 WMI 的 Windows 脚本。因为从数据流向角度看, WMI 应用程序是消耗 WMI 提供器所提供的信息的, 所以有时又被称为 WMI 消耗器 (WMI Consumer)。

在对 WMI 有了基本认识后, 下面我们介绍如何在驱动程序中通过 WMI 机制提供状态信息。图 27-2 显示了 WDM 模型中用来支持 WMI 机制的主要部件及它们在 WMI 架构中的位置。其中用户态的 WDM 提供器负责将来自 WMI 应用程序的请求转发给 WDM 的内核函数, 这些函数在 DDK 文档中被称为 WDM 的 WMI 扩展。

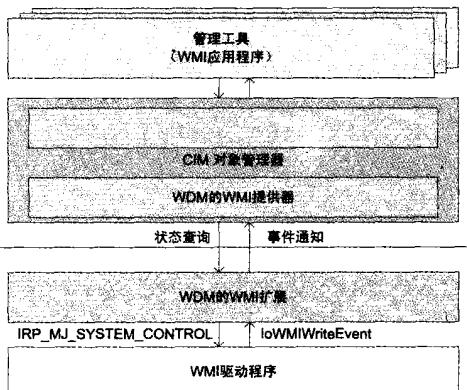


图 27-2 WDM 中支持 WMI 的软件架构

WMI 扩展会将来自用户态的请求以 IRP(I/O request packet)的形式发给驱动程序。所有 WMI 请求的主 IRP 号都是 IRP_MJ_SYSTEM_CONTROL, 子号码 (minor code) 可能为如下一些值。

- IRP_MN_REGINFO 或 IRP_MN_REGINFO_EX: 查询或者更新驱动程序的注册信息。在驱动程序调用 IoWMIRegistrationControl 函数后, 系统便会向其发送这个 IRP, 以查询注册信息, 包括数据块格式等 (见下文)。
- IRP_MN_QUERY_ALL_DATA 和 IRP_MN_QUERY_SINGLE_INSTANCE: 查询一个数据块的所有实例或单个实例。
- IRP_MN_CHANGE_SINGLE_ITEM 和 IRP_MN_CHANGE_SINGLE_INSTANCE: 让驱动程序修改数据块的一个或多个条目 (item)。
- IRP_MN_ENABLE_COLLECTION 和 IRP_MN_DISABLE_COLLECTION: 通知驱动程序开始累积或停止累积难以收集的数据。
- IRP_MN_ENABLE_EVENTS 和 IRP_MN_DISABLE_EVENTS: 启用或禁止事件。
- IRP_MN_EXECUTE_METHOD: 执行受管对象的方法。

WMI 使用一种名为 MOF (Managed Object Format) 的语言来描述受管对象。MOF 是基于 IDL (Interface Definition Language) 的，熟悉 COM 编程的读者知道 IDL 是描述 COM 接口的一种主要方法。MOF 有它独有的语法，使用 DMTF 提供的 DTD (Document Type Definition) 可将 MOF 文件转化为 XML 文件。驱动程序可以将编译好的 MOF 以资源方式放在驱动程序文件中，或者将其放在其他文件中，然后在注册表中通过 MofImagePath 键值给出其路径，也可以直接将 MOF 数据包含在代码中，然后在收到 IRP_MN_QUERY_ALL_DATA 或 IRP_MN_QUERY_SINGLE_INSTANCE 时将格式化后的数据包返回给 WMI。

为了更方便在 WDM 驱动程序中支持 WMI，DDK 还提供了一套 WMI 库，只要在驱动程序中包含头文件 `wmilib.h`，那么就可以使用其中的函数，例如 `WmiFireEvent` 和 `WmiCompleteRequest` 等。Windows SDK 和 DDK 中都包含了演示 WMI 的实例，SDK 中的示例程序位于 `Samples\SysMgmt\WMI` 目录下，DDK 的示例程序位于 `src\wdm\wmi` 目录下。

27.5.3 性能计数器

图 27-3 所示的是 Windows 的性能监视器 (Performance Monitor) 程序的窗口，只要在开始菜单中选择运行 (Run) 然后输入 `perfmon` 就可以将其调出来。图中目前显示了 5 个性能计数器 (Performance Counter)，点击曲线上方的加号可以选择加入其他性能计数器。在典型的 Windows 系统中，通常有上千个性能计数器，分别用来观察内存、CPU、网络服务、.NET CLR、SQL Server、Outlook、ASP.Net、Terminal Service 等部件的内部状态。

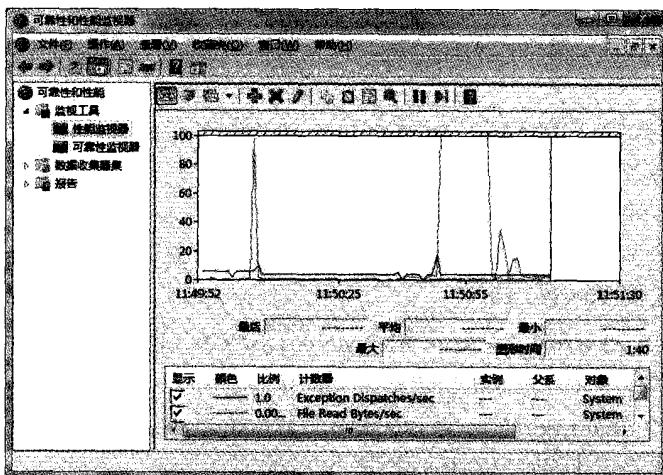


图 27-3 Windows 的性能监视器 (Performance Monitor) 程序

Windows 的性能监视机制是可以扩展的，图 27-4 画出了其架构示意图。最上面是

查询性能数据的应用程序，比如 PerfMon，最下面是提供性能数据的 DLL 模块。Windows 的 system32 目录已经预装了一些性能数据提供模块，比如 perfos.dll（操作系统）、perfdisk.dll（磁盘）、perfnet.dll（网络）、perfproc.dll（进程）、perfts.dll（终端服务）等。应用软件也可以安装和注册新的性能数据提供模块（稍后讨论）。

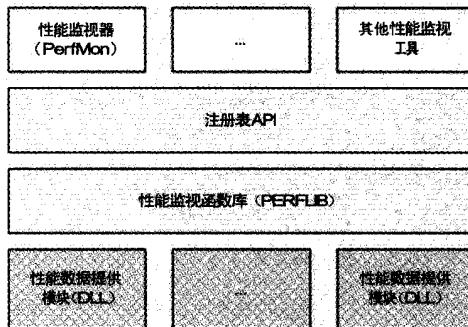


图 27-4 Windows 性能监视机制的架构示意图

每个性能模块 (DLL) 都至少输出以下 3 个方法：打开 (Open)、收集数据 (Collect)、和关闭 (Close)。具体的方法名可以自由定义，注册时登记在注册表中。图 27-5 显示了 PerfGen 模块的注册信息，右侧的 Open、Collect 和 Close 3 个键值指定的是 PerfGen.dll (Library 键值) 输出的 3 个函数。

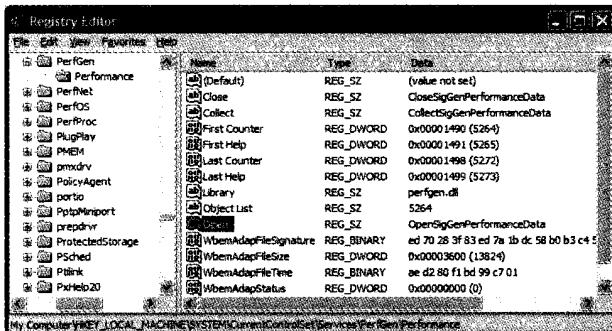


图 27-5 性能监视数据提供模块的注册信息

性能监视程序访问性能数据提供模块的基本方式是通过注册表 API。首先调用 RegOpenKey 或 RegQueryValueEx API 并将 HKEY_PERFORMANCE_DATA 作为第一个参数。当注册表 API 发现要操作的根键是 HKEY_PERFORMANCE_DATA 时，会将其转给所谓的性能监视函数库，即 PerfLib。比如以下调用第一次执行时，PerfLib 会寻找 ID 号为 234 的性能数据提供模块，在笔者的机器上它对应的是 PhysicalDisk 性能对象：

```
RegQueryValueEx( HKEY_PERFORMANCE_DATA,
    "234", NULL, NULL, (LPBYTE) PerfData, &BufferSize )
```

PerfLib 收到调用后，会枚举注册表中注册的所有服务，即枚举以下表键：
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services

对于每个服务子键，PerfLib 会试图打开它的 Performance 子键，如果这个服务存在 Performance 子键，那么会进一步查询 Performance 子键下的 Object List 键值（参见图 27-5 中 PerfGen 的注册表选项）。Object List 键值标识了这个性能提供模块所支持的性能对象 ID，如果查询到的 Object List 键值中包含所寻找的 ID，那么 PerfLib 会根据 Library 键值中所指定的 DLL 文件名加载这个模块，然后根据 Open 键值中指定的函数名取得这个函数的地址并调用这个函数。因此，上面的 RegQueryValueEx 调用会导致 PerfLib 加载 perfdisk.dll，并调用它的 Open 方法，其函数调用序列如清单 27-9 所示。

清单 27-9 性能监视程序访问性能数据提供模块的过程

ChildEBP	RetAddr
0012f168	77e42180 perfdisk!OpenDiskObject
0012f858	77e40e5c ADVAPI32!OpenExtObjectLibrary+0x58f
0012f9cc	77e09c8e ADVAPI32!QueryExtensibleData+0x3d8
0012fda4	77df4406 ADVAPI32!PerfRegQueryValue+0x513
0012fe94	77dd7930 ADVAPI32!LocalBaseRegQueryValue+0x306
0012feec	00401500 ADVAPI32!RegQueryValueExA+0xde
0012ff80	00403b39 PerfView!main+0x70 [c:\...\chap27\perfview\perfview.cpp @ 212]
0012ffc0	7c816fd7 PerfView!mainCRTStartup+0xe9 [crt0.c @ 206]
0012fff0	00000000 kernel32!BaseProcessStart+0x23

性能数据提供模块的 Open 方法应该返回一个 `PERF_DATA_BLOCK` 数据结构，即：

```
typedef struct _PERF_DATA_BLOCK {
    WCHAR Signature[4];           // 结构签名，固定为“PERF”
    DWORD LittleEndian;          // 字节排列顺序
    DWORD Version;               // 版本号
    DWORD Revision;              // 校订版本号
    DWORD TotalByteLength;        // 性能数据的总长度、字节数
    DWORD HeaderLength;           // 本结构的长度
    DWORD NumObjectType;          // 被监视的对象类型个数
    DWORD DefaultObject;          // 要显示的默认对象序号
    SYSTEMTIME SystemTime;         // UTC 格式的系统时间
    LARGE_INTEGER PerfTime;        // 性能计数器的取值
    LARGE_INTEGER PerfFreq;        // 性能计数器的频率，即每秒钟的计数器变化量
    LARGE_INTEGER PerfTime100nSec;   // 以 100 纳秒为单位的计数器取值
    DWORD SystemNameLength;        // 字节为单位的系统名称长度
    DWORD SystemNameOffset;        // 系统名称的偏移，相对于本结构的起始地址
} PERF_DATA_BLOCK;
```

紧邻 `PERF_DATA_BLOCK` 数据结构的应该是一个或多个 `PERF_COUNTER_DEFINITION` 结构，每个结构对应一个性能计数器。

当第二次调用 `RegQueryValueEx` 时，PerfLib 会认为是在查询性能数据，因此会调用性能提供模块的 `Collect` 方法。当查询完成后，性能监视程序应该调用 `RegCloseKey` API 结束查询。

Windows SDK 中给出了一个性能数据提供模块的例子，其名称为 `PerfGen`，路径为：`<SDK 根目录>\Samples\WinBase\WinNT\PerfTool\PerfDllS\PerfGen`

其实性能数据提供模块就是一个输出了前面提到的 `Open`、`Collect` 和 `Close` 方法的 DLL。注册一个性能数据提供模块通常需要两个步骤。第一步是使用一个 `.reg` 文件

向注册表的 HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services 键值下注册服务，其结果就是在注册表中建立图 27-5 所示的键值。第二步是使用一个 INI 文件，然后利用命令行工具 lodctr 来向图 27-6 所示的 Counter 和 Help 键值中增加性能对象。

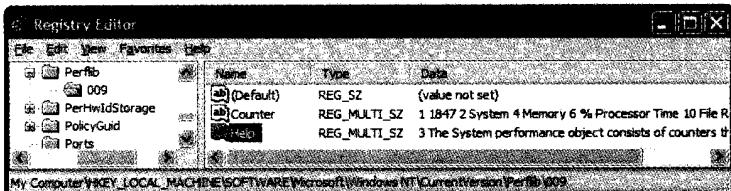


图 27-6 PerfLib 在注册表中的英语语言键值

Counter 键值的每个字符串对应一个性能计数器的 ID 或名称，Help 键值存储了每个性能对象的说明文字的 ID 和内容。计数器 ID 和说明 ID 是相邻的，前者为偶数，后者为相邻的奇数。计数器 ID 是从 2 开始的，ID 1 用来表示 Base Index。

为了简化性能数据提供模块的实现过程，ATL 类库提供了一系列类，如 CPerfObject、CPerfMon 等。MSDN 中提供了如下几个实例程序来演示这些类的用法：PerformancePersist、PerformanceCounter、PerformanceScribble。

使用 unlodctr 命令可以删除一个计数器模块。例如 unlodctr perfgen 命令可以删除 perfgen 模块注册的性能计数器。

当 PerfLib 在加载某个性能数据提供模块时遇到问题，那么它会向系统日志中加入错误信息，例如，当加载 PERFGEN.DLL 失败时，系统会产生以下日志：

```
Event Type: Warning
Event Source: WinMgmt
Event Category: None
Event ID: 37...
Description: WMI ADAP was unable to load the perfgen.dll performance library due
to an unknown problem within the library: 0x0
```

在软件中通过性能计数器来提高可观察性的好处是可以被所有性能监视工具（包括 PerfMon）所访问到，而且可以复用性能监视工具的图形化显示功能。为了简化性能监视工具的编写，Windows 提供了一个 PDH 模块（PDH.DLL），PDH 的全称是 Performance Data Helper。

27.5.4 转储 (Dump)

本章的第 27.2 节我们简要地介绍过转储。可以认为转储是给被转储对象拍摄一个快照，将被转储对象在转储发生那一时刻的特征永久定格在那里，然后可以慢慢分析。另外，因为转储结果通常直接来自内存数据，所以转储结果具有信息量大、准确度高等优点。第 27.2 节中我们介绍了对象转储，本节将介绍进程转储。所谓进程转储，就是把一个进程在某一时刻的状态存储到文件中。转储的内容通常包括进程的基本信息，进程中各个线程的信息，每个线程的寄存器值和栈数据，进程所打开的句柄，进程的数据段内

容等。

进程转储通常用在进程发生严重错误时，比如 Windows 的 WER 机制会在应用程序出现未处理异常时调用 Dr. Watson 自动产生转储（参见第 12 章）。但事实上，当应用程序正常运行时，也可以进行转储，而且这种转储不会影响应用程序继续运行。这意味着，从技术角度来讲，完全可以通过热键或菜单项来触发一个进程让其对自身进行转储。但这样做应该要考虑以下几个问题。

- 转储的过程要占用 CPU 时间和系统资源（磁盘访问），转储类型中定义的信息种类越多，转储所花的开销也越大。
- 为了防止普通用户意外使用转储功能，最好定义一种机制，需要先做一个准备动作（比如登录），然后才启动触发进程转储的热键或者开启有关的菜单项。
- 转储中包含了应用程序的内存数据，其中可能包括用户的工作数据。具体说，财务报表程序的转储中可能包含使用者的财务数据。因此，应该注意转储文件的安全性和保密性。

使用 WinDBG 或 Visual Studio 2005 都可以打开并分析进程转储文件，我们在第 12.9 节介绍过用户态转储文件的文件格式、产生方法和分析方法。

27.5.5 打印或者输出调试信息

使用 `print` 这样的函数或 `OutputDebugString` API 输出调试信息对提高软件的可观察性也是有帮助的。因为这些信息不仅可以提供变量取值等状态信息，还可以提供代码执行位置这样的位置信息。但使用这种方法应该注意以下几点。

1. 努力提高输出信息的信息量，使其包含必要的上下文信息（线程 ID、模块名等）和具体的特征和状态，切忌不要频繁输出 `Error happened` 这样的模糊信息。
2. 注意信息的言简意赅，既易于理解，又不啰嗦累赘。累赘的信息不仅会对软件的大小和运行速度造成影响，而且可能干扰调试者的注意力。
3. 最好制定一种可以动态开启或者关闭的机制，在不需要输出信息的时候不要输出大量信息，以免影响性能或者干扰调试。

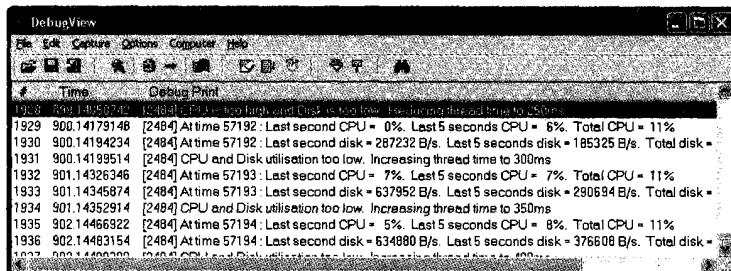


图 27-7 过多的调试信息输出

关于问题 2 的一个反面例子就是图 27-7 所示的某个病毒扫描程序所输出的调试信息。根据图中的时间信息可以判断出这个程序输出的信息非常频繁，如此频繁的信息输出会明显地使系统变慢，而且会干扰我们调试其他软件。

一个好的例子是当 WinDBG 与被调试系统成功建立内核调试会话时 WinDBG 输出的描述信息：

```
Connected to Windows XP 2600 x86 compatible target, ptr64 FALSE
Kernel Debugger connection established. (Initial Breakpoint requested)
Symbol search path is:
SRV*d:\symbols*http://msdl.microsoft.com/download/symbols
Executable search path is:
Windows XP Kernel Version 2600 MP (1 procs) Free x86 compatible
Built by: 2600.xpsp_sp2_gdr.050301-1519
Kernel base = 0x804d8000 PsLoadedModuleList = 0x805634a0
System Uptime: not available
```

第 1 行不仅陈述了连接的事实，还说明了目标系统的基本信息，Windows XP，Build 号为 2600，CPU 架构为 x86，32 位系统（非 64 位）。第 2 行表示与目标系统的调试引擎握手成功，并请求了初始断点。第 3 到 5 行打印出了当前的符号搜索路径和可执行文件搜索路径。第 6 行打印出了目标系统的详细版本信息，MP 代表启动的是支持多处理器的内核文件，括号中 1 procs 表示目前有一个 CPU 在工作，Free 代表是发行版本（非 Checked）。第 7 行给出了内核文件的构建信息。第 8 行是内核模块的加载地址和用来记录内核模块链表表头的全局变量 PsLoadedModuleList 的取值。

之所以说以上信息较好，是因为它很好地回答了调试时经常要用到的基本信息，比如目标系统的软硬件情况（OS 版本、CPU 数量），本地路径的设置情况和内核模块链表地址等。

27.5.6 日志

与打印到屏幕上或输出到调试窗口的调试信息相比，日志具有更好的可靠性和持久性。而且因为日志通常是以简单的文本文件形式存储的，所以它具有非常好的可读性和易传递性。对于需要长时间运行的系统服务类软件，比如数据库系统或网络服务等，日志是了解其运行状态和进行调试的重要资源。重大的事件或者异常情况通常应该写入日志，以下是一些典型的例子。

- 进程或系统的启动和终止，在启动时通常会将当前的版本等重要信息一并写到日志中。
- 重要工作线程的创建和退出，特别是非正常退出。
- 模块加载和卸载。
- 异常情况或检测到错误时。

日志记录也应该注意简明扼要，而且记录的数量不易过多，因为日志通常是要保

存较长时间的，记录的信息太多，不易于保存和查阅。第 15 章介绍过 Windows 系统提供的日志功能。

本节介绍了提高软件观察性的一些常用方法，这当然不是全部，比如使用编译器的函数进出挂钩（/Gh 编译器选项）功能有利于观察程序的执行位置。因为在设计软件和编写代码时无法完全预料错误会发生在哪个位置和调试时希望观察什么样的信息。这就要求我们始终把软件的可观察性记在心中，带着未雨绸缪的思想写好每一段代码，这样整个软件的可观察性和可调试性自然就会得到提高。

27.6 自检和自动报告

本节我们将简要地讨论用来提高软件可调试性的另两种机制：自检（自我诊断）和自动报告，我们先从集成电路领域的 BIST 说起。

27.6.1 BIST

BIST（Built-In Self-Test）是集成电路（IC）领域的一个术语，意思是指构建在芯片内部的自我测试功能，或者说自检。很多较大规模的集成芯片，比如 CPU、芯片组等都包含 BIST 机制。BIST 检查通常在芯片复位时自动运行，但也可以根据需要被调用和运行，比如通过 TAP 接口可以启动英特尔 IA-32 CPU 的 BIST 测试，测试结束时 EAX 寄存器中存放着测试的结果，0 代表测试通过。

BIST 通常是对芯片而言的，对于计算机系统，通常也会实现自检机制。事实在每次启动一个计算机系统时，CPU 复位后首先执行的就是所谓的 POST 代码（位于 BIOS 中），POST 的含义就是上电自检（Power On Self Test）。

自检可以防止系统在存在问题时继续工作而导致错误的计算结果和工作输出，以免导致更严重的问题。

27.6.2 软件自检

与硬件领域的 BIST 类似，在软件中实现自检功能也是非常有意义的。软件自检与单元测试不同，首先，单元测试的目的是在开发软件的过程中用来辅助测试的，而软件自检对于进入产品期的软件也是有意义的。另外，单元测试主要关注某个模块（单元）的工作情况，而软件自检更关注整个系统的完整性。

软件自检的内容应该根据每个软件的实际情况来定义，通常包括以下内容。

- 组成模块的完备性（不缺少任何模块）和完整性（没有哪个模块残缺或者被篡改）。
- 各个模块的版本，版本间的依赖关系是否满足运行要求。

- 系统运行所依赖的软硬件条件是否满足，比如依赖的硬件或者其他基础软件是否存在并工作。
- 模块间的通信机制是否畅通。

启动软件自检的方式可以在软件开始运行时自动执行，也可以提供一个专门的工具程序或者用户界面。比如用来诊断和检查 DirectX 的自检工具 DXDIAG 使用的就是后一种方式。执行 DXDIAG 后，它会启动一个窗口界面，包含很多个页面（Tab），分别用来提供不同方面的检查功能（参见图 27-8）。

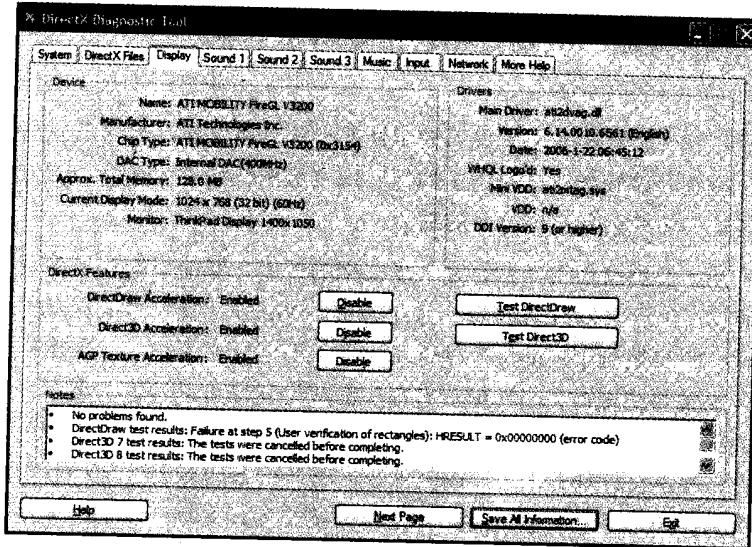


图 27-8 DirectX 的自检工具

点击 DXDIAG 程序中的测试按钮就可以测试当前系统中的有关模块和功能，并将测试结果显示在列表中。

27.6.3 自动报告

与自检有关的一个功能是自动产生和发送报告。自动产生报告就是让软件自己收集自己的状态信息和故障信息，并把这些信息写在文件中。可以把转储文件看作是一种二进制的报告。为了便于阅读，很多软件可以产生文本文件形式的报告，或者同时使用文本文件和二进制文件。

例如前面介绍的 DirectX 自检工具就可以将收集的信息和检查的结果保存在一个文本文件中（点击图 27-8 中的 Save All Information 按钮）。

Windows 中的 Msinfo32 工具可以将系统的软硬件信息保存到文件中。例如执行以下命令，Msinfo32 会以静默的方式将系统信息写入到 sysinfo.txt 文件中：

```
Msinfo32 /report c:\sysinfo.txt
```

通过 `winmsd/?` 命令可以得到 `Msinfo32` 的一个简单帮助。当测试和调试时，很多时候我们需要比较多个系统的配置信息。这时，使用 `DXDIAG` 或 `Msinfo32` 产生的信息报告是一种便捷有效的方式。

`Windows` 的 WER (Windows Error Reporting) 机制是自动发送报告的一个典型实例（详见第 14 章）。自动发送报告功能的一个重要问题就是要注意不能侵犯用户的隐私信息，在报告中不应该包含用户的标识信息，比如用户名地址等，另外应该征得用户同意后才能将报告发送到自己的服务器。

27.7 本章总结

本章讨论了在软件工程中实现软件可调试性的一些具体问题，包括角色分工、架构方面的考虑，特别是比较详细地讨论了如何实现可追溯性和可观察性。因为篇幅限制，我们无法讨论太多编码和实现方面的细节。

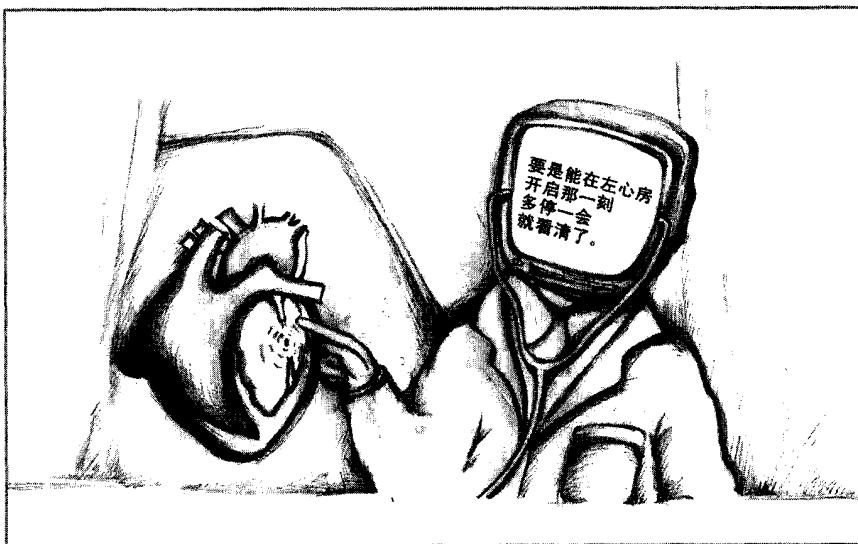
最后想说的是，提高软件可调试性是一项长期的投资，即使短期看不到明显的效益，也不应该放弃。仍然以 `Windows` 操作系统为例，`Windows` 操作系统流行的一个重要原因是无比丰富且源源不断的应用软件可以在上面运行。否则，一个操作系统本身的技术再好，安装和配置再灵活，但只有很少的应用软件可以运行，那么它也很难流行起来。因为人们购买一台计算机（硬件和操作系统）的目的，主要是在上面安装和使用应用软件，如办公软件、工程绘图软件等。而 `Windows` 操作系统中有如此丰富应用软件的一个重要原因，就是它对软件开发和调试的良好支持。`Windows` 不仅自身有很好的可调试性，而且它为其上运行的其他软件实现可调试性提供了强大的支持。

参考文献

1. G. Pascal Zachary. *Showstopper: The Breakneck Race to Create Windows NT and the Next Generation at Microsoft*. The Free Press, 1994
2. Robert M. Metzger. *Debugging by Thinking: A Multidisciplinary Approach*. Elsevier Digital Press, 2003

第6篇

调试器



调试器（Debugger）是解决复杂软件问题的最重要工具。如果把软件调试技术比作一门武功，那么调试器就是应用这门武功时所必不可少的武器。

简单来说，调试器就是用来提供调试功能的软件或硬件工具。到底提供哪些功能才能称得上是调试器没有明确的标准呢？我们通常认为一个调试器至少应该具有如下两项功能。

1. 可以控制被调试程序的执行，包括将其中断到调试器，单步跟踪执行，恢复运行，设置断点等。
2. 可以访问被调试程序的代码和数据，包括读写数据，观察和反汇编成代码，读写寄存器等。

以上两项功能是相辅相成的，前者让被调试程序根据调试人员的要求运行或停止，后者对其进行分析。二者结合起来，就可以将被调试程序中断在几乎任意的时

间或空间位置，然后对其进行观察、分析和修改，待分析结束后再让被调试程序继续运行，这种调试方式被称为交互式调试（*Interactively Debugging*），这是区别调试器和其他普通工具的最重要标准。这种将被调试对象静止下来，然后无限期地“慢慢”分析其状态，而后再像什么也发生过似的让其继续运行的交互式诊断方法是软件调试所特有的，在硬件和传统的医学或其他领域都是很难施行的。例如，医生诊断心脏疾病时是不可能将病人的心脏停止在某个位置来观察的。

根据被调试程序的工作模式，可以把调试器分为用户态调试器和内核态调试器，前者用于调试用户态下的各种程序，如应用程序、系统服务，或者用户态的 DLL 模块；后者用于调试工作在内核态的程序，如驱动程序和操作系统的内核模块。在第 28 章我们将讨论调试器的通用模型、一般原理、基本任务和常见调试器的分类。

WinDBG 是 Windows 平台中使用非常广泛的著名调试器，它既可以用作用户态调试器，也可以用作内核态调试器，是调试 Windows 操作系统下各种软件的一个强有力的工具。本篇的第 29 章和第 30 章将分别讨论 WinDBG 调试器的工作原理和使用方法。

调试器的设计初衷是为了辅助软件开发人员定位和去除软件故障，但因为调试器所具有的对软件的强大控制力和观察力，其应用早已延伸到了很多其他领域，比如逆向工程、计算机安全等。可以说，调试器是几乎所有软件高手的必备工具，他们都非常擅于使用调试器。因此，每个软件工程师都应该把学习和熟练使用调试器当作一门必修课。

调试器概览

调试器是软件工程师战斗的武器，只有全面而深刻地理解它，才能充分发挥它的威力。尽管自计算机程序（Program）出现那天起，人们就开始面临调试这项繁重任务，但调试器却是在很多年后才逐渐形成的。与很多其他软件技术一样，调试器也是经历了很多人的努力才逐步发展成今天这个样子。为了更好地理解决调试器的发展过程，本章的前 3 节将分别介绍大型机（28.1 节）、小型机（28.2 节）和个人计算机（28.3 节）上的著名调试器。接下来的四节将分别介绍调试器的典型功能（28.4 节）、分类标准（28.5 节）、实现模型（28.6 节）和经典架构（28.7 节）。第 28.8 节将介绍一个公开的调试器标准——HPD。

28.1 TX-0 计算机和 FLIT 调试器

TX-0 是 20 世纪 50 年代由林肯实验室（Lincoln Laboratory）设计的晶体管计算机，其全称为 Transistorized Experimental Computer Zero，它是世界上最早使用晶体管替代电子管的可编程（Programmable）通用（General-purpose）计算机，从 1955 年开始制造，于 1956 年投入使用。

林肯实验室构建 TX-0 计算机的主要目的是测试准备为 TX-2 计算机使用的大型磁芯存储器（“large” magnetic core memory）。在这个测试任务完成后，TX-0 被移交给麻省理工学院（MIT）的 RLE（MIT Research Laboratory of Electronics）实验室，安装在 MIT 的 26 号楼，这时是 1959 年。

当 TX-0 移交给 RLE 实验室时，它上面只有两个软件工具，一个是简单的汇编程序，另一个叫 UT-3（Utility Tape 3）。UT-3 是一个简单的帮助调试的工具，通过它，用户可以向指定的内存地址输入数据或指定一个地址范围，让系统打印出来。从调试器的角度来看，UT-3 提供了观察和修改内存的功能，但根据本篇序言中的标准，UT-3 还不能算作真正的调试器。

为了弥补 TX-0 的软件不足，RLE 实验室开始为其编写各种软件工具，其中最著名的就是 Jack B. Dennis 所编写的 MACRO 汇编程序及 Dennis 与 Thomas G. Stockham

(1933-10-22—2004-1-6) 共同编写的 FLIT 程序。

MACRO 汇编器所引入的宏概念今天仍然被广泛使用着，利用宏，程序员可以把一系列指令定义在一起，大大节约了录入程序所需的时间。MACRO 的另一个重要特征就是不仅可以产生程序代码，而且可以生成供调试和分析使用的调试符号(Symbol)。

FLIT 的全称是 FLexewriter Interrogation Tape，这个名字来源于当时的一种杀虫喷雾剂的名字。表 28-1 列出了 FLIT 程序所支持的主要命令。

表 28-1 FLIT 的主要命令（伪指令）

伪指令	功能
clear a,b	清除从地址 a 到地址 b 的内存区域
print a,b	打印从地址 a 到地址 b 的内存区域
word w,a,b	在从地址 a 到地址 b 的内存范围内搜索 w
address l,a,b	在从地址 a 到地址 b 的内存范围内搜索地址 l
surprise	比较内存和磁带
table	读磁带上的符号表
read	从磁带读取程序
start	开始执行程序
start l	开始从位置 l 执行程序
break bp1,bp2,bp3,bp4	设置断点
break	删除所有断点
proceed	因为断点中断后使用这个命令继续执行
输入地址或包含符号的表达式	显示指定地址或表达式的值，继续输入新的值便修改这个内容（回车或 Tab 确认）

从这些命令可以看出 FLIT 比较全面地提供了各种调试功能，包括设置断点，观察和编辑内存，控制程序执行，读取符号等。特别值得一提的是，FLIT 支持符号化调试（Symbolic Debugging），调试人员可以通过符号名来指定要观察的变量，也可以在表达式中包含符号。另外，FLIT 程序提供的断点功能和今天的断点功能已经非常类似。通过断点命令，用户可以使用 break 命令设置和清除断点。当设置断点时，FLIT 调试器先将指定位置的指令保存起来，然后向这个位置写入跳转指令，当程序执行到这个位置时，便会跳转到 FLIT 调试器的断点处理代码。断点处理代码先将程序状态保存起来，然后将控制权交给用户，用户可以使用其他调试器命令观察和分析被调试程序，观察结束后，再发出命令让程序继续执行。

FLIT 将各种调试功能集中在一个单独的程序中，同时实现了控制被调试程序执行和观察分析被调试程序的功能，因此 FLIP 已经符合了调试器的标准，是真正意义上的调试器。因为早期的计算机大多在硬件上配备一个开关来切换到单步执行模式，因此在调试器中没有必要再设计单步跟踪功能。

从内存布局来看，FLIT 加载在内存空间中的固定位置与调试程序在同一个空间中。这样做有个不足，就是当修改和向被调试程序添加代码时，可能覆盖和破坏调试器的内容。为了弥补这个不足，FLIT 有个缩减版本名为 MicroFLIP，它占用的内存空间更小。

FLIT 的完成时间大约为 1960 年，经过很多搜索之后，这是笔者发现的最早的软件调试器。FLIT 的出现不仅给当时开发其他软件提供了强有力的工具，而且它对软件调试技术的贡献也是具有里程碑意义的，其后的很多调试器都是基于或借鉴了 FLIT 的实现方法。包括下一节要介绍的 DDT 调试器。

28.2 小型机和 DDT 调试器

在现代计算机从大型机向小型化发展的历史上，DEC 公司（Digital Equipment Corporation）的 PDP 系列小型机起到了重要的作用。PDP 是 Programmed Data Processor（可编程数据处理机）的缩写，这个名字中故意没有带 Computer 字样，因为当时在人们脑海中计算机都是要占满一个房间的甚至一个楼面的庞然大物。本节将介绍 PDP 系列小型机上使用的主要调试器。

28.2.1 PDP-1

PDP 系列的第一代产品 PDP-1 于 1960 年发布，共生产了大约 50 台。尽管数量不大，但是它的影响是巨大的，它的体积和今天的服务器机柜差不多，价格约 12 万美元，与今天的计算机价格相比，这个价格很惊人，但在当时，这只是大型机价格的一个零头。最重要的是，PDP 配备了 CRT 显示器，让用户可以直接与机器对话。用户可以坐在计算机前编写和调试程序，执行命令并观察机器处理数据的全过程。体积小，价格低，友好的人机交互方式——PDP 的这些特征代表了计算机设计理念的一个根本变革，预示了计算机从大型化向小型化发展的大趋势。

因为 DEC 公司的创始人哈兰·安德森（Harlan Anderson）和肯·奥尔森（Ken Olsen）就来自设计 TX-0 的林肯实验室，所以可以说 PDP-1 在很多方面都基于或受到了 TX-0 的启发，包括指令集和与用户交互的理念。但与 TX-0 的实验目的不同，PDP-1 一开始就是为商业化目的而设计的。

1961 年夏季，DEC 公司向林肯实验室赠送了一台 PDP-1。尽管这台 PDP-1 没有为 DEC 公司带来直接的收入，但是它为 PDP 系列计算机的发展带来了更有价值的两个收益，一是让很多天才开始为 PDP 编写软件，二是吸引了很多 MIT 的人到 DEC 工作。

在 PDP-1 到达 MIT 不久，很多著名的软件纷纷诞生了，包括第一个交互式的计算机视频游戏（Interactive Computer Video Game）星球大战（Spacewar），第一个分时操作系统，还有后来被所有 PDP 计算机采用的 DDT 调试器。

DDT 是 MIT 的学生们为 PDP-1 编写的众多软件中的一个，它的作者是 Alan Kotok，时间是在 1961 年夏季。DDT 的最初含义是 DEC Debugging Tape，即 DEC 调试器磁带，因为当时 DDT 是存储在磁带上的，在磁带被其他存储介质（比如磁盘）取代后，DDT 被改称为 Dynamic Debugging Tool。因为使用 DDT 大大提高了调试效率，因此很快流

行起来，并成为之后 PDP 小型机上必不可少的一个配套软件。当 1962 年 PDP-4 推出时（PDP-2 一直没有开发，PDP-3 也没有市场化），DDT 已经被包含在系统的软件列表中。图 28-1 是 PDP-4 的产品手册里对 DDT（DDT-4）的介绍。

Symbolic Assembly and Debugging System

ASSEMBLY PROGRAM

A one-pass assembler that allows mnemonic symbols to be used for addresses and instructions. Constant and variable storage registers are automatically assigned. This assembler will produce relocatable or absolute binary output, as desired by the user.

RELOCATING LOADER

Performs relocation and linking of binary programs that have been assembled separately.

DDT-4 (DEC DEBUGGING TAPE-4)

Debugging may be done at run time using the teleprinter. Break points may be inserted into a program at arbitrary points so that the state and operation of a program may be observed. The source program symbols may be used for communication.

EDMUND THE EDITOR

Allows the editing of symbolic tapes.

图 28-1 PDP-4 产品手册中关于 DDT 的介绍

随着 PDP 计算机的流行，DDT 逐渐成为 20 世纪 60 和 70 年代最著名的工具软件之一，被越来越多的人使用。为了便于区分不同版本的 DDT 工具，人们把与 PDP-4 一起发行的 DDT 称为 DDT-4，把 Alan Kotok 最初为 PDP-1 设计的 DDT 称为 DDT-1。

DDT 是从 Micro FLIP 改编过来的，因此它的最初功能和工作方式都与 FLIP 调试器非常类似。表 28-2 列出了 DDT-1 的主要命令。这些命令是从 Alan Kotok 所写的 DDT 文档中（参考文献 2）摘录的，这份文档的修改日期为 1964 年 8 月 13 日，它是 PDP-1 程序库的一部分。当时的程序库和今天 SDK 的作用很类似。

表 28-2 DDT-1 的主要命令

命令	功能
断点 (Breakpoints)	
B	在指定位置插入断点，或者删除断点 (未指定地址)
P	从断点继续
G	跳到 (go to) 参数指定的位置
程序分析 (Program Examination)	
/	打开指定的寄存器
回车	将新的信息放入打开的寄存器并将其关闭
Backspace	除了具有与回车同样的功能外，同时打开相邻的下一个寄存器
Tab	除了具有与回车同样的功能外，同时打开指定的寄存器，打出它的内容
模式控制 (Mode Control)	
S	将 DDT 的字 (Words) 输出模式设置为符号方式
C	将 DDT 的字 (Words) 输出模式设置为 n 进制常数
R	将 DDT 的位置输出模式设置为相对于符号
O	将 DDT 的位置输出模式设置为 n 进制数
U	十进制模式输出
H	八进制模式输出

续表

命令	功能
字搜索 (Word Searches)	
W	在指定内存范围内搜索与指定表达式的值相等的内存字
N	与 W 类似, 但搜索与指定表达式不相等的内存字
E	与 W 类似, 但搜索有效地址等于指定表达式的内存字
存储 (Storage)	
K	将符号表复位成初始的列表
Z	将不是 DDT 使用的内存全部清 0
加载磁带 (Loading Tapes)	
Y	从磁带加载二进制程序 (binary) 到内存
T	从磁带读取 Macro 编译器产生的符号表
打孔 (Punching)	
L	让 DDT 进入等待输入标题模式, 把用户键入的字符以可读的方式打孔到纸带上
D	将参数所指定地址范围内的数据打孔到纸带上
J	将跳转块 (jump block) 打孔到纸带上
其他	
X	执行指定的指令

DDT-1 的断点功能是使用一条特殊的跳转命令 jda 来实现的, 我们在第 1 章的 1.3.2 节介绍了这种方法的工作原理。另外, DDT-1 只支持一个断点, 设置一个新的断点会自动将前一个删除掉 (参考文献 2)。

28.2.2 TOPS-10 操作系统和 DDT-10

1967 年推出的 PDP-10 配备了名为 TOPS-10 的操作系统。TOPS-10 是 DEC 公司基于 PDP-6 的监视程序 (PDP-6 Monitor) 而设计的分时操作系统。TOPS-10 有很多版本, 从 1964 年的 1.4 到 1980 年的 7.01。1976 年 DEC 公司推出的 TOPS-20 是基于 TOPS-10 和另一个早期的操作系统 TENEX 而设计的。

在 TOPS-10 中, 已经包含了很多调试支持, 如内核调试和用户态调试。TOPS-10 系统自带的标准调试工具为 DDT-10。从关于 TOPS-10 操作系统的手册中可以看出, DDT-10 是一个与操作系统和开发工具有机结合的工具集。其关系好比是 NT 操作系统与 WinDBG。也可能是, NT 内核的设计者直接或间接地受到了 TOPS-10 和 DDT 的影响。

DDT-10 的使用手册 (参考文献 3) 是 TOPS-10 文档 (称为 Notebook) 的一个部分。笔者找到的一个版本是 1986 年 4 月更新过的, 其长度有 130 页。

DDT-10 包括了如下一些模块。

DDT.REL: 这是一个可以重定位的 DDT 模块, 它类似于今天的静态库, 可以被链接到被调试程序中, 其方法是使用 TOPS-10 的 DEBUG 命令:

```
DEBUG <被调试程序文件>
```

DEBUG 命令会使用 **LINK** 找到被调试程序的符号文件然后将 **DDT.REL** 和被调试的程序融合在一起。待链接结束后，链接好的程序会被启动，**DDT** 开始运行，用户可以输入各种调试命令，包括设置断点，让被调试程序开始运行，等等。这是使用 **DDT** 调试应用程序的最常用方法。从用户角度来看，这个过程已经与今天的从调试器中启动被调试程序并开始调试会话非常接近。

DDT.EXE：这是一个可以独立运行的调试工具，使用 **TOPS-10** 的运行命令（**R DDT**）便可以启动它。启动后可以执行硬件指令和 **TOPS-10** 的 **UUO**（**Unimplemented User Operation**），**UUO** 是调用 **TOPS-10** 系统服务的一种方式，类似于今天的从用户态呼叫内核态的系统服务。

VMDDT.EXE：这是用来调试已经加载到内存中的应用程序的 **DDT** 模块，其工作方式类似于今天的附加（**Attach**）到被调试进程。但是其功能还没有今天的调试器那么强大，只有当被调试程序处于以下非运行状态时，才能将 **VMDDT** “附加”到被调试程序上：应用程序刚刚加载；应用程序结束；应用程序崩溃；应用程序被 **Ctrl+C** 终止。在以上情况下，输入 **DDT** 命令，那么 **DDT** 会检查用来管理进程的 **JOBDAT** 结构的 **JBDDT** 字段，如果这个字段为 0，就说明还没有调试器加载，于是 **DDT** 会加载 **VMDDT.EXE**。内存页 700-777 是保留给 **VMDDT.EXE** 使用的。从这一点我们可以看出当时的操作系统对软件调试已经提供了特殊的支持。

FILDDT.EXE：用来分析和修改程序文件、崩溃转储和磁盘上的数据。

EDDT.EXE：用于调试和修补操作系统的内核（当时称为 **Monitor**）。名称中的 **E** 代表 **Executive**，即管理模式（内核模式）。**EDDT** 有两种工作方式，一种类似于今天的使用 **WinDBG** 调试 **NT** 内核，必须在启动内核时就调用 **EDDT**。

BOOT><内核文件名称> /E

/E 开关会导致内核和调试模块被同时加载，并且使内核工作在调试模式下。当 **EDDT** 初始化结束后，会显示调试提示符，输入 **G** 命令可以让系统继续加载。在系统的分时功能（多任务）初始化后，可以在终端上（**TTY**）键入 **Ctrl+D** 来触发系统内核的陷阱（中断）机制，调用内核的一个内部过程跳转到 **DDT** 的代码，也就是使系统中断到调试器。中断到调试器后，用户可以设置断点、单步执行或进行其他各种调试操作。

EDDT 的另一种工作方式是在用户态执行来修补和分析内核文件，其方法是先使用 **TOPS-10** 的 **GET** 命令将内核的 **.EXE** 文件加载到内存，然后通过 **DDT** 命令启动 **EDDT.EXE**。启动后，用户可以使用 **DDT** 的各种命令来分析和修改内核文件，也可以设置断点，如果退出时还有断点未删除，则这些断点在下次启动这个内核文件时还会起作用。

COBDDT：这是用来调试 **COBOL** 程序的 **DDT** 变体。

FORDDT：这是用来调试 **FORTRAN** 程序的 **DDT** 变体。

从上面的介绍可以看出，**DDT-10** 已经是一个功能非常齐全的调试工具包。除了上

面介绍的功能，值得注意的 DDT-10 功能还有。

- 使用单步执行命令来进行单步跟踪，此前，单步执行是由硬件开关来支持的。
- 支持条件断点。
- 执行子过程或某一范围内的指令。
- 动态插入补丁，即在调试过程中动态地插入一段代码，并让程序跳转过去执行。

支持 PDP-11 和 TOPS-20 的 DDT 被称为 DDT-11。DDT-11 加入了远程调试功能并加入了转储文件有关的功能。感兴趣的读者可以阅读参考文献 4 了解更详细的内容。

28.3 个人计算机和它的调试器

随着集成芯片的发明和微处理器（Micro-CPU）的诞生，从 20 世纪 70 年代末起，个人计算机（PC）开始蓬勃发展。与小型机的情况类似，PC 的发展也离不开调试技术的参与，或者说某一领域在快速发展之前，调试技术作为其他技术发展的基础必然会先发展起来。

28.3.1 8086 Monitor

DOS 是图形化操作系统出现前个人电脑上的主要操作系统，80 年代的大多数 PC 使用的都是 DOS 操作系统。很多使用过 DOS 的人都知道 DOS 系统自带了一个调试程序，叫 Debug。Debug 程序完全实现在一个文件中，即 Debug.exe。在 DOS 提示符下输入 Debug 便可以启动这个程序，然后可以加载被调试程序，执行汇编反汇编，访问 I/O 端口等。

DOS 和 Debug 程序的最初设计者是同一个人，他就是被称为 DOS 之父的 Tim Paterson。1980 年 4 月到 7 月，Paterson 在 CP/M 的基础上设计了针对 8086 CPU 的简单操作系统，称为 QDOS（Quick and Dirty Operating System）。1981 年微软从 Paterson 所在的 Seattle Computer Products（SCP）公司购买了 QDOS，将其改名为 86-DOS 并提供给 IBM，随 IBM PC 一起销售，称为 PC DOS 1.0。从 DOS 4.0 开始，开始改称为 MS DOS。

在设计 QDOS 之前，Tim Paterson 设计了一个名为 8086 Monitor 的调试器，这个调试器后来被集成到 DOS 中，即 Debug 程序。在 Paterson 创立的 Paterson Technology 公司的网站上可以找到 8086 Monitor 的 1.4 版本的手册和源代码清单，其链接为：
<http://www.patersontech.com/Dos/Manuals.aspx>

8086 Monitor 是与名为 CPU Support Card 的硬件一起工作的，其功能是调试 8086 CPU 卡上的软硬件。8086 CPU 卡是 SCP 公司当时的主要产品。8086 Monitor 是从 1979 年初开始开发的，第一版的完成时间是在 1979 年 11 月前，1980 年初做过改进。

8086 Monitor 提供了断点，单步跟踪，观察和修改内存，观察和修改寄存器，从

磁盘加载程序，读写 I/O 端口等功能，是个人计算机发展初期被广泛使用的著名调试器。本书第 4 章比较详细地介绍过 8086 Monitor 的主要功能和实现细节。

28.3.2 SYMDEB

8086 Monitor 和 DEBUG 都是汇编级别的调试器，不支持符号化的调试，也就是只能通过内存地址来观察变量和函数。SYMDEB 是个人电脑上较早支持符号化调试的调试器。它最初是随微软的宏汇编编译器 MASM 一起发布的，后来也被包含在 16 位 Windows 的 SDK 中。SYMDEB 的全程是 Microsoft (R) Symbolic Debug Utility。

在使用 SYMDEB 进行符号化调试前，应该先产生符号文件。这需要两个步骤，第一步是在链接时指定/MAP 开关生成 MAP 文件。第二步是通过 MAPSYM.EXE 从 MAP 文件产生 SYM 文件。

有了 SYM 文件后，便可以使用以下语法开始符号化调试了：

SYMDEB [参数] [SYM 文件名] [被调试程序名] [被调试程序的参数]

借助 SYM 文件，使用 SYMDEB 进行调试时，便可以通过变量名来观察变量，通过函数名设置断点，这比根本不支持符号的 DEBUG 程序进步了很多。

除了支持 DEBUG 定义的几乎所有命令外，SYMDEB 还支持表 28-3 所列出的命令。

表 28-3 SYMDEB 支持的新命令

命令	功能
BP	设置断点
BC, BD, BE, BL	分别为清除断点，禁止断点，启用断点和列出所有断点
C	比较内存
K	栈回溯
V	观察源代码
X	检查符号
XO	打开新的符号文件
Z	通过符号来设置变量的取值
?	评估表达式
!	执行 Shell 命令
.	显示当前的源文件行
*	注释

上表中的很多命令一直被沿用到今天，比如 BP、BC、BD、BL、C、K 和*命令的写法和含义与 WinDBG 中的完全一样。

28.3.3 CodeView 调试器

尽管 SYMDEB 已经具有了基本的符号支持，可以使用符号来访问变量和设置断点，但是还不支持真正的源代码级调试，比如不能显示数据结构，不能在源代码级跟

踪。因为 SYMDEB 使用的符号文件所包含的符号信息还不足以支持这些功能。在一
这一背景下，大约在 1985 年，CodeView 调试器诞生了，它的主要设计者是 Dave Norris
和 Mike O'Leary。CodeView 最先是与 4.0 版本的微软 C 编译器（MSC 4.0）一起发布
的。我们在 25 章介绍的编译选项/Zi 也是这个时候引入的，字符 i 来源于 CodeView 项
目的代号 Island。当编译程序时使用了/Zi 选项，那么编译器就会产生调试符号，链接
时这些符号会被连接到可执行文件中。有了符号信息后，便可以使用 CodeView 调试
器进行真正的源代码级调试了。

在随 MSC 4.0 一起发布后，CodeView 被加入到 MASM、Basic 编译器和之后的 MSC 编译器中。当 Visual C++ 1.0 发布时，CodeView 调试器被集成到了 IDE 环境中，这种做法一直被延续到后来的 Visual Studio 集成开发环境。CodeView 调试器有用于调试 DOS 程序的 DOS 版本和调试 16 位 Windows 程序的版本（简称为 CVW）。

28.3.4 Turbo Debugger

Turbo Debugger(简称 TD)是 Borland 公司的著名调试器。它的第一个版本 TD 1.0 于 1989 年发布。TD 除了支持 Borland 公司的调试符号外，还支持 CodeView 格式的调试符号，因此使用 TD 既可以调试使用 Borland 编译器开发的程序，也可以调试使用微软编译器开发的程序。加上 TD 功能强大而且稳定，所以 TD 很快成为当时最流行的调试器。Borland 公司的汇编编译器 TASM 和 C/C++ 编译器中都包含了 TD。

TASM 5 和 Borland C/C++ 5.02 包含了 5.0 版本的 TD 调试器。这一调试器在今天依然是很多人调试 DOS 程序的首选工具。TD 5.0 在 Windows XP 的 DOS 窗口中可以很好地工作，图 28-2 显示了使用它调试一个使用 C 语言编写的 Hello World 程序时的情景。



图 28-2 Turbo Debugger 调试器

TD 的显示模式是彩色的文本模式 (Text Mode)，因此它的所有窗口边框和控制按钮都是使用扩展 ASCII 码来绘制的。可以说，TD 将文本方式的窗口界面发挥到了非常高的水平。在图 28-2 中，我们打开了 5 个窗口，分别是显示 C 语言源程序的模块窗口 (1 号)，包含汇编指令和寄存器信息的 CPU 窗口 (4 号)，观察变量的 Watch 窗口 (2 号)，观察

内存的 Dump 窗口（5 号）和显示跟踪执行经过的 Execution History 窗口（3 号）。

TD 调试器的功能已经非常完善，包含了今天主流调试器支持的大多数功能，例如，设置软硬件断点，观察和修改内存与寄存器，以及跟踪执行等。此外，TD 还包含了今天的调试器不再支持的一些功能。我们不妨选取其中的两个功能作为例子。一个是执行状态回退，使用这一功能前需要使用 TD 的配置程序 TDINST 将 Full trace history 选项打开，然后再跟踪程序时，通过 Run 菜单的 Back trace 命令或 Alt+F4 热键便可以回到执行历史窗口中所显示的前一个状态。使用这一功能就好像让被调试程序反方向执行一样，因此又被称为反向执行或反向跟踪。事实上，这一功能是将跟踪过的执行上下文保存起来，当反向跟踪时，将保存的上下文恢复出来。TD 的另一个值得称道的功能就是它的动画执行功能。当单步跟踪程序时，选择 Run 菜单的 Animated 命令，然后指定一个“动画放映速度”便可以让调试器不断地重复单步跟踪执行动作，像放动画一样单步执行程序，直到按任意键停止。利用这个功能，调试者就不必频繁地按跟踪键了，只要像看电影一样地观察程序的“慢动作”执行过程，当执行到感兴趣的位置时再让它停下来。

与 CodeView 类似，TD 也有 DOS 版本和 Windows 版本之分，Windows 版本通常被称为 TDW（调试 16 位的 Windows 程序）或 TDW32（调试 32 位的 Windows 程序）。

28.3.5 SoftICE

DOS 时代出现的著名调试器还有 SoftICE。SoftICE 的最初版本是由 Frank Grossman 和 Jim Moskun 在 1987 年时设计的，Frank 和 Jim 也是经营 SoftICE 的 NuMega 公司的两个创始人。与前面介绍的几个工作在 DOS 上的调试器不同，SoftICE 使用了根本不同的设计思路。我们知道 DOS 是运行在实模式的一个单任务操作系统中的，因此普通的 DOS 调试器和被调试程序是运行在一个地址空间中的，调试器先运行，然后把被调试程序加载在同一地址空间中的较低位置。SoftICE 没有沿用这种做法，它利用了 80386 CPU 引入的虚拟 8086 模式功能，让自己运行在保护模式下，让被调试程序和 DOS 系统运行在虚拟 8086 模式中。这一设计使 SoftICE 具有了两个普通 DOS 调试器无法得到的两个好处。第一是 SoftICE 可以完全控制 DOS 系统和被调试程序，就好像今天的虚拟机管理器可以完全控制虚拟机一样。第二个好处是，利用保护模式的分页机制，SoftICE 可以运行在不同的地址空间中，不必与被调试程序共享有限的 640K 地址空间。基于这两大优势，加上 SoftICE 设计合理，灵巧强大，所以它很快得到了众多软件高手的认可，成为很多人眼里的最好调试器。

在 Windows 开始流行后，SoftICE 推出了 Windows 版本，可以在一台机器上进行内核调试，成为调试 Windows 驱动程序的首选工具之一。SoftICE 的强大功能和领先优势保持了很多年。但是出于种种原因，2006 年 4 月，SoftICE 停止开发，这一著名的调试器从此停顿不前了。

本节介绍了个人电脑上有代表性的调试器，包括 8086 Monitor、SYMDEB、

CodeView、Turbo Debugger 和 SoftICE。第 29 章和第 30 章将介绍 WinDBG 调试器。

28.4 调试器的功能

本节将浏览调试器的典型功能，并简要说明这些功能的工作原理。

28.4.1 建立和终止调试会话

调试器在调试一个程序前，必须先与其建立起调试与被调试的关系，即建立调试会话（Session）。对于用户态调试，建立调试会话的方式通常有两种，一种是在调试器中启动被调试程序，另一种是在被调试程序加载后，再将调试器附加（Attach）到被调试程序。对于内核态调试，建立调试会话的过程就是调试引擎介入到被调试系统的过程，这通常是在系统启动过程中发生的。

终止调试会话的方式有很多种，可以直接退出调试器，这通常也会导致被调试程序终止；或者将调试器分离（Detach），让被调试程序继续运行；当然也可以直接退出被调试程序。

28.4.2 控制被调试程序执行

能够控制被调试程序执行是调试器区别于其他软件工具的重要特征。这一功能包括如下几个子功能。

1. 将被调试程序中断到调试器（Break into Debugger），典型的做法是在被调试程序中触发一个调试事件使其中断到调试器。很多调试器都有 Break 命令来支持这一功能。Windows 操作系统支持使用 F12 热键将被调试程序中断到调试器（详见 10.6 节）。
2. 让被调试程序以受控的方式执行，比如单步执行，单步执行到指定位置，从指定的位置开始执行等。
3. 对线程执行挂起（Suspend）/恢复（Resume）和冻结（Freeze）/解冻（Unfreeze）操作，比如 WinDBG 的~m、~n、~f 和~u 命令可以对一个或多个线程分别执行挂起、恢复冻结和解冻操作。
4. 恢复被调试程序运行。
5. 终止被调试进程。
6. 重新启动（Restart）被调试程序和调试会话，比如 WinDBG 的.restart 命令可以重新运行被调试程序，内核调试中的.reboot 命令可以让目标系统重新启动。

直接修改寄存器和内存也可以达到控制程序执行的目的，比如设置程序指针寄存器（EIP）的值和修改内存中的指令，但是我们把这些功能归类到访问内存和寄存器中。

28.4.3 访问内存

内存是软件工作的舞台，软件的代码必须先被读入到内存后才能被 CPU 执行；除了少数分配在寄存器中的局部变量外，软件的大多数变量也都是分配在内存中的。因此，观察和操作被调试程序的内存对调试器来说是非常重要的。访问内存的功能又可以细分为。

1. 以不同格式显示内存中的数据，比如按字节、字、双字、字符串或结构类型 (Type) 来显示某一地址的数据。WinDBG 的 d 系列命令用来实现这一功能。
2. 编辑指定地址的内存数据，比如 WinDBG 的 e 系列命令。
3. 通过变量名来观察和修改它的取值，变量包括局部变量、全局变量、成员和静态变量等。
4. 移动 (Move) 某一范围的内存数据到另一位置。
5. 填充 (Fill) 某一内存区域。
6. 比较指定范围的内存数据。
7. 在内存中搜索某一个数据模式。
8. 观察用来组织内存的数据结构，比如堆 (Heap) 和栈 (Stack)。WinDBG 的 !heap 和 !teb 扩展命令分别可以显示堆和栈的信息。

好的调试器应该允许用户以多种方式来指定要访问的内存地址，包括物理地址、虚拟地址、变量名称和表达式。

28.4.4 访问寄存器

访问寄存器功能用来观察或修改 CPU 寄存器。这里要说明的一点是，因为一个 CPU (多核中的一个核) 只有一套寄存器，所以在调试器中观察到的寄存器都是上下文结构 (CONTEXT) 中的寄存器。当调试事件发生时，系统会将当时的寄存器状态保存在一个执行上下文结构中，调试器中操作的都是这个 CONTEXT 结构中的值。当被调试程序被恢复执行时，系统会将 CONTEXT 结构中的寄存器值更新到 CPU 的物理寄存器中。出于这个原因，对寄存器的修改是在恢复程序运行时才生效的，而对于内存的修改是执行命令后便写到内存中的。

因为上下文结构中不包含 MSR 寄存器，所以对 MSR 寄存器的访问是直接针对 CPU 中的物理寄存器的，比如 WinDBG 的 rdmsr 和 wrmsr 分别用来读写 MSR 寄存器。

包括 VC6 在内的很多调试器只提供观察寄存器的功能，不提供修改功能。但是 VC8 和 WinDBG 都有修改寄存器的功能（使用 r 命令）。特别是，修改程序指针寄存器（如 r eip=0xxx）的值可以起到跳转执行的目的，但使用时应该慎重，因为这种跳转可能导致栈失去平衡或变量未初始化等各种问题。

28.4.5 断点 (Breakpoints)

设置和管理断点是调试器的最重要功能之一。设置断点的目的是让被调试程序执行到某一空间或时间点时将其中断到调试器中，然后对其进行分析。根据断点的中断条件可以将其分为如下几种。

1. 指令断点：当程序执行指定内存地址的指令时中断到调试器，又叫代码断点。
2. 数据断点：当程序访问指定内存地址的数据时中断到调试器。
3. I/O 断点：当程序访问指定 I/O 地址的端口时中断到调试器。

代码断点的设置方法通常是使用特殊的指令将要中断位置的指令动态替换掉。例如，像 FLIT 这样的早期调试器大多是使用跳转指令来替换原来的指令，这样当执行到断点位置时，CPU 便会跳转到用于调试的代码。对于 x86 CPU，INT 3 指令（机器码为 0xCC）专门是为实现代码断点而设计的，因此运行在 x86 CPU 上的调试器通常都使用 INT 3 指令来设置代码断点。

设置代码断点的另一种方法就是使用 CPU 的硬件支持。最常见的就是使用 CPU 的调试寄存器。比如 IA32 CPU 定义了 8 个调试寄存器，DR0~DR7，对于一个调试会话可以最多同时设置 4 个断点。这 4 个断点的地址可以为代码地址、数据地址或 I/O 地址，分别对应于代码断点、数据断点和 I/O 断点。

使用前一种替换程序指令的方法设置的断点叫软件断点，后一种利用 CPU 的硬件支持而设置的断点叫硬件断点。通过硬件方法可以设置以上 3 种断点中的任一种，但是通过替换指令只可以设置代码断点。

在断点被触发后（又称为命中，Hit），操作系统的调试子系统或调试器的接收例程会保存程序的状态，并利用操作系统的进程/线程管理功能将被调试程序挂起，然后将断点事件发送给调试器的断点处理模块。断点处理模块得到通知后，会根据断点的属性和调试器的设置决定是否需要通知用户并与用户交互，如果需要，则通过人机交互接口把控制权交给用户。在用户分析结束后再发出命令恢复继续执行。

对于 IA32 CPU，数据断点和 I/O 断点触发的都是陷阱类（Trap）异常，CPU 报告这类异常时，触发异常的那条指令已经执行完毕，程序指针已经指向要执行的下一条指令。这意味着当数据断点被报告到调试器时，访问数据的那条语句已经执行完毕，因此调试器的执行位置会显示在下一行。与此不同的是，通过调试寄存器设置的代码断点触发的异常属于错误类（Fault）异常，这类异常报告时，程序指针指向的仍然是触发异常的那条指令。所以当调试器收到这类异常时，执行位置会显示在断点地址所对应的那一行。这时如果直接继续执行，那么还会触发异常。为了防止死循环，调试器应该在恢复执行前设置标志寄存器（EFALGS）的 RF（Resume Flag）位，在 CPU 看到 RF 位被置起后，会忽略指令断点。对于使用 INT3 指令设置的代码断点，因为 INT 3 指令触发的是陷阱类异常，所以当调试器收到调试事件时，程序指针指向的是 INT 3 指令的下一条指令。Windows 系统在分发 INT 3 导致的断点异常时会将程序指针递减

1，所以，用户在调试器看到的执行位置依然是在断点地址所对应的那一行。综上所述，指令断点命中时，执行位置在当前行，当前行尚未执行；数据和 I/O 断点命中时，执行位置在下一行（汇编级），触发断点那一指令已经执行完毕。

根据调试器收到断点事件后是否中断给用户开始交互式调试，可以把断点分成条件断点（Conditional Breakpoint）和无条件断点（Unconditional Breakpoint）两种。对于支持条件断点的调试器来说，在它收到断点事件后，它会评估这个断点所附带的条件是否成立，如果成立，则启动用户界面开始交互式调试，否则，直接让被调试程序继续运行。

图 28-3 显示了使用 VC8 集成调试器调试 C++ 程序时设置的两个断点，第一个是无条件的代码断点，第二个是有条件的 (`argv[0] != 0`) 的数据访问断点。其中的地址 `0x12FF70` 是 `main` 函数的参数 `argc` 的地址。值得说明的是，包括 VC8 在内的很多调试器对于数据断点只支持改写数据时触发断点。事实上，CPU 一级是支持读数据时也触发断点的。使用 WinDBG 可以指定数据断点的读写触发条件，如果使用 `ba r<长度><地址>`，则读写都触发，如果使用 `ba w<长度><地址>`，那么便只有向这个地址执行写操作时才触发。

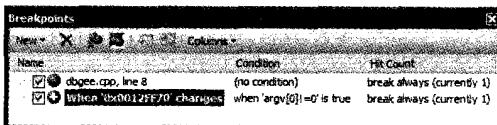


图 28-3 断点示例（VC8）

很多调试器都提供有运行到光标位置（Run to Cursor）这样的功能。这一功能也是通过插入断点来实现的。

断点的另一个衍生功能叫追踪点（Tracepoint）。其实现原理仍然是基于断点，只是当命中后打印一些用于追踪的调试信息或者执行某个脚本，然后让程序继续执行。图 28-4 显示了 VC8 的插入追踪点对话框。通过这个对话框可以定义这个追踪点命中后要打印的消息、执行的宏，以及是否要开始用户交互。

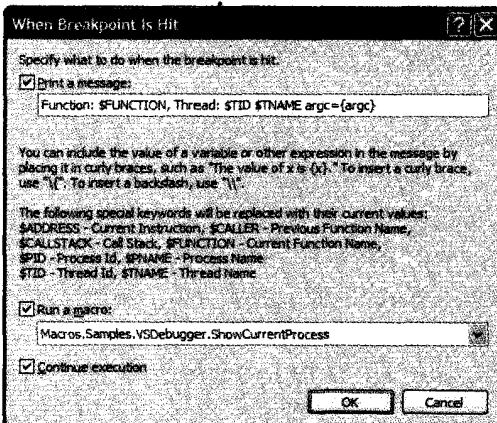


图 28-4 VC8 的插入追踪点（tracepoint）对话框

在这个追踪点所在位置的代码被执行后，在 VC8 的输出窗口会有如下消息：

```
Function: wmain(int, wchar_t * *), Thread: 0x82C wmainCRTStartup argc=1
0516: c:\dig\dbg\author\code\chap28\dbgee\debug\dbgee.exe
```

其中第二行是图 28-4 中指定的 ShowCurrentProcess 宏所打印出的当前进程 ID 和被调试程序全路径。

断点的有关调试功能还有为断点设置适用条件（比如指定适用线程），对断点命中次数进行计数，暂时禁止和启用断点，显示断点列表，断点命中后自动执行其他命令或脚本等。

28.4.6 跟踪执行 (Tracing)

跟踪执行被调试程序是调试器的另一常见功能，其作用是让被调试程序按照一种受控的方式来执行，典型的跟踪方式有以下几种。

1. 每次执行一条汇编指令，称为汇编语言一级的单步跟踪。设置 IA32 CPU 标志寄存器的 TF (Trap Flag，即陷阱标志位) 位，便可以让 CPU 每执行完一条指令后便产生一个调试异常 (INT 1) 而中断到调试器。早期的很多计算机，比如 PDP-1，在控制面板上有一个 Single Step 按钮，按下这个按钮，那么计算机每执行一条指令后便会停下来。
2. 每次执行源代码（比汇编语言更高级的程序语言，如 C/C++）的一条语句，又称为源代码级的单步跟踪。通常高级语言的单步执行是通过很多次指令一级的单步执行而实现的。调试器在收到单步执行事件后，会根据调试符号中的源代码行信息计算出程序指针值所对应的源代码行，如果源代码行与被单步执行的源代码行一致，那么便重新设置陷阱标志并让程序继续执行，否则便说明当前源代码行已经执行完毕而中断给用户。
3. 每次执行一个程序分支，又称为分支到分支单步跟踪。IA32 CPU 的 DbgCtl MSR 寄存器的 BTF (Branch Trap Flag) 标志用于启用分支到分支单步跟踪。
4. 每次执行一项任务（线程），即当一项任务（线程）被调度执行时中断到调试器。IA32 架构所定义的任务状态段 (TSS) 中的 T 标志为实现这一功能提供了硬件一级的支持，但是目前的大多数调试器都还没有提供这项功能。
5. 执行到上一级函数，又称为 Step Out，即从当前函数一直执行到返回调用这个函数的函数。

很多调试器允许设置单步执行的次数，这也是通过反复多次的指令一级的单步执行来实现的，如 WinDBG 的 t 命令。类似的，还有执行到指定的地址，如 WinDBG 的 ta 命令。

当单步执行的代码行存在函数调用时，根据是否跟踪进入要调用的函数，单步执行又分为 Step Into 和 Step Over 两种。

28.4.7 观察栈和栈回溯

栈是程序执行的重要数据结构，是函数调用和分配局部变量所必不可少的特殊内存区。调试器通常提供如下几种观察栈的功能。

1. 显示栈的基本信息，包括栈的地址、限制、当前栈顶地址（ESP）等。
2. 显示栈帧（Stack Frame）的信息，包括栈帧的基准地址（EBP）、栈帧中的函数返回地址、参数和局部变量。显示局部变量的名称和取值，需要调试符号的支持，如果没有调试符号，那么只能以原始数据的形式来显示栈中的变量。
3. 栈回溯（Stack Backtrace），又叫打印函数调用序列（Calling Stack），即显示记录在栈中的函数调用过程。通过栈回溯可以了解程序的执行过程，因此对软件调试有着非常重要的意义。要显示正确的栈回溯信息也需要调试符号的帮助。

28.4.8 汇编和反汇编

汇编（Assemble）功能是将一段汇编程序编译为机器码并写到指定的内存地址。比如 WinDBG 的 a 命令。反汇编是将某一地址范围的机器指令翻译为可读的汇编程序代码。WinDBG 的 u 和 uf 命令都是用来反汇编的。关于反汇编有一点需要注意的是，对于 x86 这样指令长度不固定的程序，一定要保证反汇编的起始地址正好是一条指令的开始，否则就会导致指令错位，得到错误的反汇编结果。

28.4.9 源代码级调试

源代码级（Source Level）调试是指在源代码层次上调试程序，包括：

1. 使用源程序中定义的数据结构来显示和修改数据。
2. 在源程序文件上设置断点，跟踪执行源程序和在源程序中显示执行位置。

通常，一种调试器只支持一种编程语言的源代码级调试，比如 VC 调试器支持 C/C++ 语言的源代码级调试。但是理论上也可以在一个调试器中同时支持几种源代码级调试，比如 VS8 调试器支持多种.NET 语言的源代码级调试，WinDBG 支持 C/C++ 和汇编语言的源代码级调试。

28.4.10 EnC

EnC 是 Edit and Continue 的缩写，即在调试的过程中编辑源代码然后自动编译并继续调试。支持 EnC 的调试器通常是 VC 这样同时具有编译和调试功能的 IDE 工具。因为 EnC 功能需要对编辑后的代码进行编译，增量链接和更新符号文件。EnC 功能并不代表可以对代码进行大量的修改，通常只有局部的少量修改才可以成功应用 EnC 功能。像 WinDBG 这样不带编译能力的调试器是不支持 EnC 功能的。

28.4.11 文件管理

包括对模块文件、符号文件和源程序文件的管理，可以细分为。

1. 维护和显示程序模块的信息，包括模块的文件名、全路径、模块在内存中的起至地址、时间戳，以及它的符号文件的类型，它的符号文件的全路径等。
2. 寻找和加载模块的符号文件。
3. 提供多种方式查找符号，包括根据地址查找对应的符号和源代码行，搜索和检查符号等。
4. 设置文件的搜索路径。
5. 从符号服务器寻找模块或符号文件并下载。

因为很多功能都是基于调试符号的，所以调试符号对于调试有着特殊的重要意义。使用符号服务器来维护符号文件是一种有效的方法。例如，微软的调试符号服务器 (<http://msdl.microsoft.com/download/symbols>) 提供了大多数 Windows 系统文件的调试符号。

28.4.12 接收和显示调试信息

捕获并显示被调试程序输出的调试信息也是调试器的一个常见功能。比如，VC 调试器会显示程序中通过 TRACE 宏或 OutputDebugString API 输出的信息。当内核调试时，调试器会显示使用 DbgPrint/DbgPrintEx 输出的调试信息。

28.4.13 转储 (Dump)

这一功能包括产生和调试转储文件。比如当使用 WinDBG 调试时，可以使用.dump 命令产生应用程序转储（用户态调试）和系统转储（内核态调试）。因为程序或系统崩溃时往往会触发转储机制，所以也可以通过强制崩溃来触发转储。

调试转储文件经常被看作一种特殊类型的调试目标（Target）。打开一个转储文件，便与这个目标（应用程序或内核）建立了调试会话，然后可以像调试活动的目标那样执行各种调试功能，包括观察寄存器，观察内存、回溯栈，显示异常信息、模块信息等，但是某些命令是不能用于调试转储文件的，比如跟踪执行，恢复程序运行等。调试器的用户手册应该说明每个调试命令的适用条件，比如 WinDBG 的帮助文件中会给每个命令附带一个表格（参见表 28-4），说明这个命令所适用的调试模式、目标类型和平台（操作系统和处理器架构）。

表 28-4 描述调试命令的适用条件

Modes	user mode, kernel mode
Targets	live debugging only
Platforms	All

本节介绍了调试器的典型功能，这当然不能涵盖所有调试器的所有功能，我们的

目的是让大家了解具有代表性和通用性的调试器功能。

28.5 分类标准

根据第 28.1 节的介绍，第一个软件调试器出现在 1960 年，当时的软件还都比较简单，数量也比较少。在这之后的 40 多年里，随着计算机和软件迅猛发展，调试技术也在不断发展，各种各样的调试器也层出不穷。今天，我们已经没有办法说出迄今为止到底存在多少种调试器了，因为每当一种新的软件技术或一种新的操作系统诞生时，通常都会诞生一系列与其配套的调试器。举例来说，.NET 技术的出现和发展引入了一批支持托管代码调试的调试器，而且衍生出了混合调试（Interop Debugging）这样新的调试模式。为了便于认识和理解庞大的调试器家族，我们归纳出了一些分类标准，用来对调试器进行分类。

28.5.1 特权级别

按照被调试程序所处的特权级别（Privilege），可以把调试器分为用户态调试器（User-Mode Debugger）和内核态调试器（Kernel-Mode Debugger）两种，内核态调试器经常被简称为内核调试器（Kernel Debugger）。用户态调试器用来调试在用户态执行的各种程序，包括应用程序、系统服务和工作在用户态的驱动程序。内核调试器用来调试运行在内核模式的各种代码，主要包括操作系统内核和工作在内核态的驱动程序。举例来说，VC IDE 所集成的调试器是用户态调试器。WinDBG 工具包中的 KD.EXE 是一个内核调试器。WinDBG.exe 既可以调试用户态程序也可以调试内核，所以它既属于用户态调试器，又属于内核调试器。

28.5.2 操作系统

大多数调试器都只适用于某一种或一类操作系统，所以，可以根据调试器所适用的操作系统对其进行分类。例如 WinDBG 是 Windows 操作系统上的调试器，GDB（GNU Project Debugger）是 Linux 操作系统上的调试器，Dbx 是 Unix 操作系统（Solaris、AIX、IRIX 和 BSD Unix）上的调试器。

28.5.3 执行方式

可以根据被调试的程序是解释执行（Interpreted）还是编译后执行（Compiled）将调试器分成脚本代码调试器（Script Debugger）和编译代码调试器。

使用 Java 和.NET 语言编写的程序尽管是以字节码方式存储的，但是当 CPU 执行时还是会被即时（Just-In-Time）编译为机器码，所以调试 Java 和.NET 程序的调试器不属于脚本调试器。事实上 WinDBG 这样的调试器就可以调试.NET 程序，只要将其

看作普通的本地程序，但如果要支持源代码级的调试或者显示.NET技术的特有数据结构，那么就必须对WinDBG进行扩展。因为.NET程序主要是以托管代码（Managed Code）为主的，所以通常把调试.NET程序的调试器称为托管调试器（Managed Debugger）。从程序执行方式来看，托管调试器是编译代码调试器中的一个子类。

28.5.4 处理器架构

因为软件总是与硬件平台的处理器架构（Processor Architecture）相关的，所以，可以根据调试器和它所能调试的程序所运行的平台架构来对调试器进行分类。比如WinDBG目前支持的CPU架构主要有x86、x64和安腾（Itanium）3种。

28.5.5 编程语言

对于支持源代码级调试的调试器，可以按照它所支持的开发语言来对调试器进行分类。比如WinDBG支持C/C++和汇编语言的源代码级调试。JDB是调试Java程序的一个调试器。某些调试器只支持汇编代码一级的调试，不支持源代码级调试，比如DOS下的DEBUG调试器。

本节介绍了对调试器进行分类的典型标准。因为每个分类标准代表了调试器在某一方面的关键特征，所以使用以上标准来衡量和描述调试器，有助于理解调试器的主要功能和特点，这也是我们介绍这些标准的主要目的。

28.6 实现模型

如何设计和实现一个调试器呢？这不是一个简单的问题。调试器的目标是调试被调试程序。要做到这一点就要与被调试程序建立起联系。那么应该如何建立联系？这个联系应该多密切呢？一般来说，关系越密切越有利于了解被调试程序的信息和更好地实现各种控制功能。但是这样做可能因为调试器的介入而影响被调试程序的行为，使得某些错误无法在调试时再现，这便是所谓的海森伯效应。

28.6.1 海森伯效应

海森伯效应（Heisenberg Effect）来源于德国著名物理学家沃纳·海森伯（Werner Heisenberg）的不确定原理（Uncertainty Principle）。这个原理指出不可能同时精确地测量出粒子的动量和位置，因为测量仪器会对被测量对象产生干扰，测量其动量就会改变其位置，反之亦然。不确定原理也被称为测不准原理，即测量的过程会影响被测试的对象。换句话说，因为海森伯效应的存在，测量的过程会影响被测量对象使测量结果不准确。

在软件领域，人们把调试时无法复现或行为发生改变的错误（Bug）称为海森伯

错误 (Heisen Bug)，把可以稳定复现的错误称为波尔错误 (Bohr Bug)。因为在调试环境下无法稳定重现，所以调试海森伯错误通常更加难以解决。

为了降低调试过程对软件错误所造成的影响，设计调试器的一个最根本原则就是使海森伯效应最低。换句话说，就是要使调试器对被调试对象的影响尽可能的小，或者说二者的关系最好是互不影响，互不干涉。但是为了实现调试功能，调试器又必须与被调试程序建立起比较密切的联系。也就是说，强大的调试功能要求调试器对被调试程序有较高的可控性，要求二者建立密切的关系。而海森伯效应又要求调试器和被调试程序的关系不能太紧密。看来这两者之间存在着一定的矛盾，如何平衡这个矛盾是设计调试器的一个关键问题。

下面将分别介绍用户态调试和内核调试所使用的典型模型，分析它们各自的优缺点。

28.6.2 进程内调试模型

所谓进程内调试模型 (In-process Debugging Model)，就是指调试器与被调试程序工作在同一个进程的地址空间中（见图 28-5）。

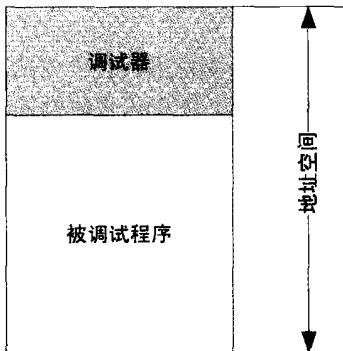


图 28-5 进程内调试模型的示意图

因为与被调试程序在同一个空间中，所以进程内模型有如下优势。

1. 可以非常直接地访问被调试程序的代码和数据，速度快，实现也简单。
2. 调试器可以复用被调试程序的某些资源和数据结构，从总体上来看可以节约资源。
3. 在调试器代码和被调试程序的代码之间可以方便地跳转或相互调用。

但是因为在同一空间中，进程内模型也存在如下不足。

1. 调试器的存在和工作可能影响被调试程序的行为，导致较大的海森伯效应。
2. 调试器的代码和数据容易遭到破坏。
3. 因为在同一进程中，所以不利于独立控制被调试程序，例如，挂起进程时，调试器线程也会被挂起。

在某些简单的计算机系统中，要么没有操作系统，要么操作系统是 DOS 这样的单

任务操作系统，这些系统中的调试器通常是和被调试程序工作在同一个空间中的。比如早期的 FLIT 调试器和 DDT 调试器，它们都是工作在内存空间中的一个固定位置，被调试程序工作在同一空间的其他部分。DOS 下的 DEBUG 调试器也属于这种情况。调试时，DEBUG 需要先运行并将其自己加载到内存中，然后再将被调试程序加载到内存中。某些嵌入式系统的调试器使用的也是类似的方法。尽管在这样的简单系统中不存在严格意义上的进程，但是从地址空间的角度来看，仍然可以将这些调试器所使用的模型称为进程内模型。

在 Windows 这样的操作系统中，完全使用进程内模型的调试器是不存在的。因为当操作系统发送调试事件时会先将被调试进程挂起，所以，如果调试器也在同一个进程中，那么调试器的代码也会被挂起，根本无法接收和处理调试事件了。稍后我们会介绍.NET 调试器使用的 CLR 调试模型，这个模型部分采用了进程内模型。

28.6.3 进程外调试模型

顾名思义，所谓进程外调试模型（Out-of-process Debugging Model）就是指调试器和被调试程序分别工作在各自的进程空间中（见图 28-6）。

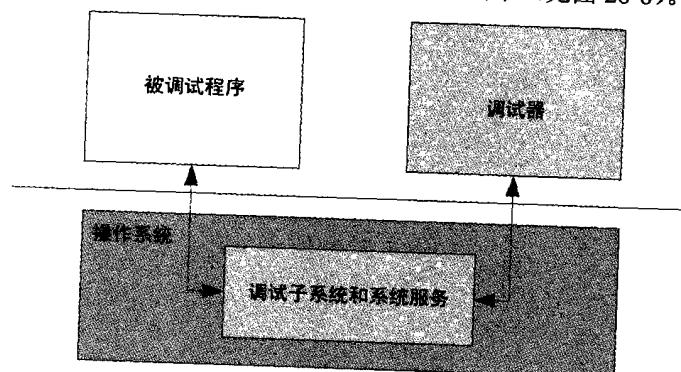


图 28-6 进程外调试模型的示意图

在进程外模型中，调试器与被调试程序分别在各自的进程空间中工作，调试器借助操作系统的 API 和调试子系统与被调试程序进行通信。

因为工作在两个独立的进程中，进程外模型有效地克服了进程内的模型的不足，具有如下优势。

1. 调试器进程使用自己的进程空间来工作，不占用被调试程序的地址空间，也不会直接访问它的数据和代码，因此导致的海森伯效应很低。
2. 被调试程序很难触及到调试器的代码和数据，因此调试器的代码和数据不容易被被调试程序所破坏。
3. 控制被调试进程时不会影响调试器进程。

但具有以上优点的同时，进程外模型的特征也决定了它的不足。

1. 二者在不同的地址空间中，调试器无法直接访问被调试程序的代码和数据，必须借助操作系统的 API 间接读取。
2. 调试事件是通过操作系统调试子系统的转发而送给调试器的，调试器处理后再把处理结果返回给调试子系统。这个过程通常要经历内核态和用户态之间的多次转换，所以速度较慢。
3. 调试器和被调试程序之间很难进行代码共享和函数调用。

Windows 操作系统使用进程外模型来实现对普通应用程序的调试。

28.6.4 混合调试模型

所谓混合调试模型（Mixed Debugging Model）就是将进程内调试模型和进程外调试模型放在一个调试方案中混合使用，图 28-7 画出了混合调试模型的示意图，除了有一个专门的调试器进程外，在被调试程序的进程中也存在一部分调试器代码，通常是一个线程。调试器进程和被调试进程内的调试器代码相互配合共同完成调试任务。为了便于描述，通常将位于被调试进程中的那部分称为调试器左端（Left Side），简称 LS，将独立的调试器进程称为右端（Right Side），简称 RS。

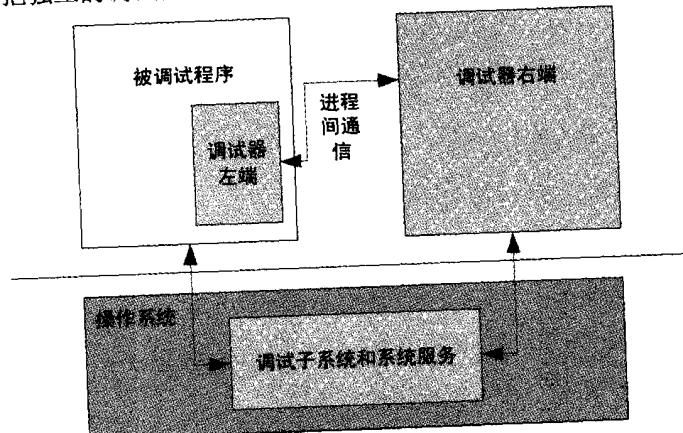


图 28-7 混合调试模型的示意图

混合调试模型同时使用了进程内模型和进程外模型，理论上来说，这引入了两个模型的优点，也同时引入了缺点。同时带来的一个新问题就是整个模型比较复杂，工作起来效率不高。

.NET 的 CLR 调试模型采用的是混合调试模型。在每个.NET 进程中，CLR 都会创建一个调试辅助线程，即 LS。LS 的作用主要是处理托管代码有关的各种调试问题。LS 与 RS 一起工作，实现对本地代码和托管代码的混合调试。

28.6.5 内核调试模型

前面我们介绍了用户态调试的3种模型，下面我们再看一下内核调试的典型模型。内核调试的目标是操作系统内核和内核模式下的其他代码。这意味着要对内核模式下的代码设置断点和使系统内核中断到调试器中。因为内核负责着整个系统的线程调度，所以内核中断意味着整个系统将被停下来，那么负责调试的代码还如何正常运行呢？这便是内核调试要解决的一个根本问题。

解决这个问题通常有3种做法。第一种是使用ITP（参见第7章）这样的硬件调试器进行调试，在CPU一级实现各种调试功能，这样做不需要对操作系统的内核加入任何调试支持（低海森伯效应），而且几乎可以调试操作系统从启动到关闭的任何代码，但这种方式存在如下局限。

1. 需要比较昂贵的硬件，而且要求主板上有调试接头或有必要的转接头，另外，设置调试环境比较麻烦，时间较长。
2. 适合做汇编指令一级的分析，难以观察进程、线程等操作系统一级的数据结构和数据对象。

第二种是所谓的双机内核调试。顾名思义，这一种方式需要两台计算机（参见图28-8），一台运行调试器，被称为主机（Host）；另一台运行被调试的系统，称为目标机（Target）。

主机与目标机之间通过某种电缆进行通信，常见的有以下几种方式。

1. Zero-Modem，即使用收发信号线对接的RS-232串行通信电缆分别插在主机和目标机的串行口上。
2. 1394线缆，又称为火线（Firewire）。
3. 支持主机到主机（Host to Host）通信的USB 2.0电缆。因为计算机上的USB端口大都是上行口（Upstream Port），所以这样的电缆中间会有一个负责中转的芯片。

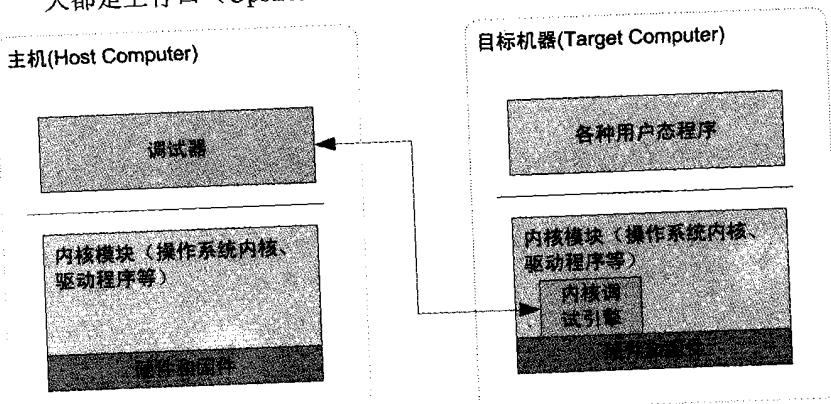


图 28-8 双机内核调试模型

随着虚拟机的流行，也可以在一台机器（主机）上调试虚拟机（目标机）中的系统。这时可以使用管道虚拟出的串口来提供主机和目标机之间的通信渠道，但其实质与两台机器上是一致的。

主机主要是用来运行调试器的，因此它的操作系统可以和目标机中的不同。目标系统中的内核调试引擎（Kernel Debug Engine）负责与调试器进行通信，报告调试事件并执行调试器所下达的命令，比如读写内存，收集信息，设置断点等。内核调试引擎通常是与操作系统内核紧密结合在一起的，比如 Windows 的内核调试引擎就在内核文件（NTOSKRNL.EXE）中。

第三种是所谓的单机内核调试，也就是在同一个系统中进行内核调试。图 28-9 画出了进行单机内核调试的调试模型，其中带有网格线的模块是调试器模块，它们分别为。

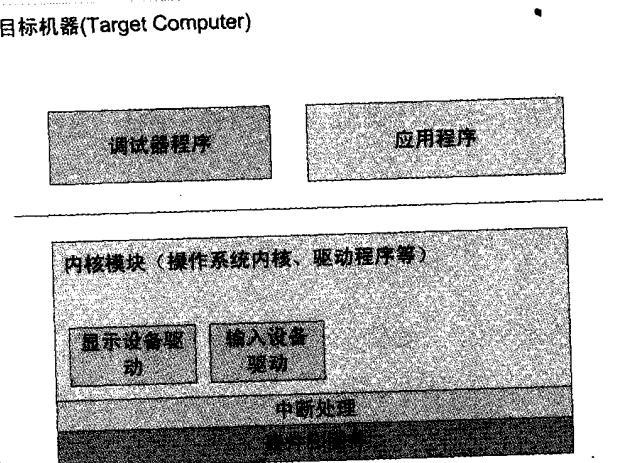


图 28-9 单机内核调试模型

中断处理函数：接收并处理 CPU 的中断和异常，特别是调试有关的异常（如 INT 1、INT 3 等等）以及与输入输出有关的中断。值得说明的是，这些处理函数是先于被调试系统的内核得到处理权的。因为只有这样，才能在被调试系统的内核被中断时，调试器依然工作。

显示设备驱动和输入设备驱动：为了能在被调试系统被中断的时候，调试器依然能够与用户进行交互——接收输入，显示输出，所以调试器通常要在系统中安装自己的输入输出驱动程序，包括鼠标、键盘和显卡。因为这 3 类设备的种类非常多，如鼠标和键盘有串行口、PS/2 和 USB 3 种接口，显卡有 PCI、AGP、PCI Express 等接口，每种接口又有无数种品牌，所以实现单机内核调试器的一个棘手问题就是要兼容很多种硬件。

从图 28-9 可以看到，在单机内核调试模型中，调试器的中断处理函数是位于操作系统的内核之下的，因此可以把操作系统和调试器看作是运行在中断处理函数之上的。

两个“系统”，当调试器没有激活时，中断处理函数会将中断和异常转发给操作系统，这样操作系统才可以正常工作。当有调试事件发生（比如断点命中）或者通过热键将调试器被激活时，中断处理函数会将操作系统冻结，让调试器活动。

使用单机内核调试模型的最著名调试器是 SoftICE，SYSER 调试器也使用了这一模型。Windows 系统（从 Windows XP 开始）和 WinDBG 调试器的本机内核调试功能只支持有限的观察功能，不可以设置断点和跟踪，因此不属于严格意义上的单机内核调试。

本节我们简要介绍了用户态调试和内核调试的典型模型。下一节我们将深入到调试器软件内部，介绍用于实现调试器的典型架构。

28.7 经典架构

很多流行的调试器使用了 CodeView 调试器采用的软件架构，包括 WinDBG 调试器和从 Visual C++ 1.0 开始直到 VC6 的集成开发环境（IDE）中的调试器（我们称其为 VsDebugger）。很多其他调试器虽然没有直接采用这个架构，但也是借鉴了它的思想，因此这个架构对调试器的设计和实现具有深远的影响，我们把这个架构称为调试器的经典架构。本节我们将先介绍构成调试器经典架构的基本单元，然后介绍如何应用这一架构来实现远程调试和多语言调试。

28.7.1 基本单元

如图 28-10 所示，使用经典架构的调试器首先将整个软件划分成外壳（Shell）和调试器内核两大部分，前者负责与用户交互（UI），后者用来实现调试器的内部逻辑和各种调试功能，又分为如下几个功能单元（Function Unit）。

1. EE：全称为 Expression Evaluator，即表达式评估器，用来解析和评估表达式，例如对断点命令中的地址参数进行解析，然后转化为调试器内部所使用的地址格式。
2. SH：全称为 Symbol Handler，即符号处理器，用来管理调试符号和程序模块，包括维护模块信息、加载符号文件、搜索符号等。
3. EM：全称为 Execution Model，即执行模型，是对被调试程序的封装，是调试器内核与调试器外壳之间交流的主要接口，通过 EM 模块外壳向内核下达，内核向外壳发送各种通知。
4. DM：全称为 Debuggee Module，即被调试程序模块，用来真正访问和控制被调试程序，比如设置断点，单步执行，读写内存等。可以认为，DM 是被调试程序在调试器中的代表。
5. TL：全称为 Transport Layer，即传输层，用来实现 EM 与 DM 之间的通信。TL 层的存在，使得 EM 和 DM 之间不再是紧密的绑定关系，通过不同的 TL，EM 和 DM 可以在不同的进程中，或者不同的机器上，这为实现远程调试奠定了基础。

从模块间通信的角度来看，调试器外壳与 SH、EE 和 EM 之间存在直接的通信。例如，使用 SH 加载符号文件，将用户下达的命令先使用 EE 来评估和转换，然后交给 EM 去执行。

在图 28-10 中，我们为每种功能单元只画出了一个实例。对于支持多种程序语言和多种处理器架构的调试器来说，它会设计很多个 EE、SH、EM 和 DM，我们稍后再详细介绍。

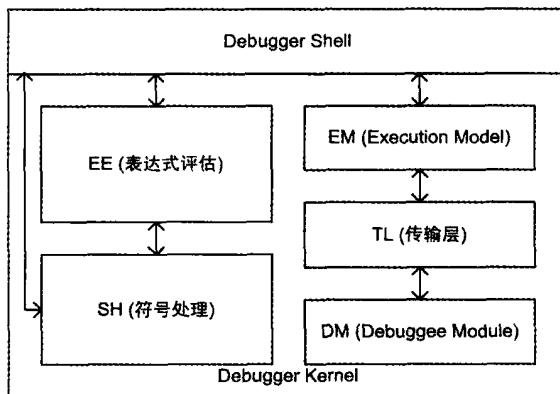


图 28-10 经典调试器架构的最简形式

28.7.2 远程调试

因为 EM 和 DM 之间是使用传输层来通信的，所以通过可以跨机器的传输层就可以将不同机器上的 EM 和 DM 联系起来，实现远程调试。如图 28-11 所示，左侧是目标机器（Target），右侧是宿主机器（Host）。传输层负责 EM 和 DM 之间的通信，这种通信可以是以太网络，也可以是使用 RS232 协议的串行通信。因为 TL 封装了通信的细节，所以宿主机器上的调试器外壳可以不必在乎 DM 在本机还是另一台机器上。目标机器上的远程外壳（Remote Shell）进程的作用是加载合适的传输层并根据 EM 的指示加载合适的 DM，同时为 TL 和 DM 提供执行环境。

对被调试程序来说，远程外壳就是它的调试器，因此调试事件会被发给其中的 DM，DM 通过传输层发送给机器上的 EM 和真正的调试器外壳。对调试人员来说，他们可以像调试本地的程序一样调试远程的程序，不必在乎被调试程序是否运行在本机上。

VC 调试器和重构前的 WinDBG 的远程调试使用的都是图 28-11 所示的模型。比如，在使用 VC5 或者 VC6 调试器的远程调试功能前，需要将如下文件复制到被调试程序所在的目标机器上。

1. vcmon.exe：远程外壳。
2. Tln0t.dll：使用 TCP/IP 网络协议的传输层。
3. Dm.dll：DM 模块。

4. Msvc60.dll: Visual C++运行时 DLL。
5. Msdis110.dll: 反汇编模块。
6. Msvcrt.dll: VC 运行库。

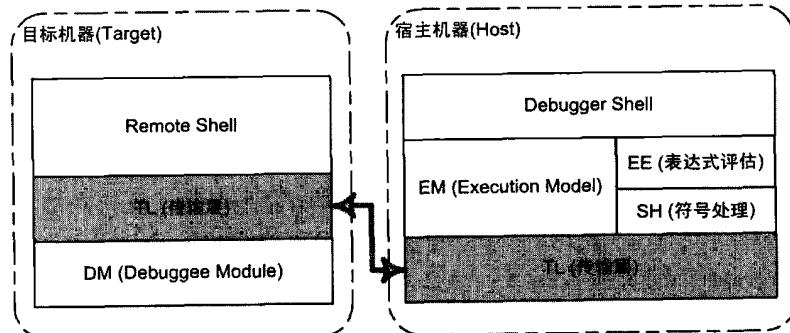


图 28-11 经典调试器架构的远程调试模型

因为使用的传输层支持通过互联网络（TCP/IP 协议）来通信，所以使用上述方法可以调试 TCP/IP 可以抵达的任何机器上的程序。

28.7.3 多语言和多处理器架构调试

在实际应用中，很多软件系统是分布式的，多个进程运行在多台计算机上。为了支持调试这样的系统，就要求调试器能够同时调试多个进程，支持多种代码语言，支持多种平台架构。图 28-12 画出了经典调试架构一般模型，通过这个模型可以实现支持多语言和多处理器的调试器。比较图 28-10 和图 28-12，后者只是对前者的一个简单扩展。在图 28-12 中，每个单元有多个实例，分别用来完成不同特征的任务。比如多个表达式评估器分别用来评估不同语言和语法的表达式；多个符号处理器处理不同格式的符号；多个执行模型模块处理不同执行架构的执行模型；多个 DM 模块来代表多个不同的被调试程序。

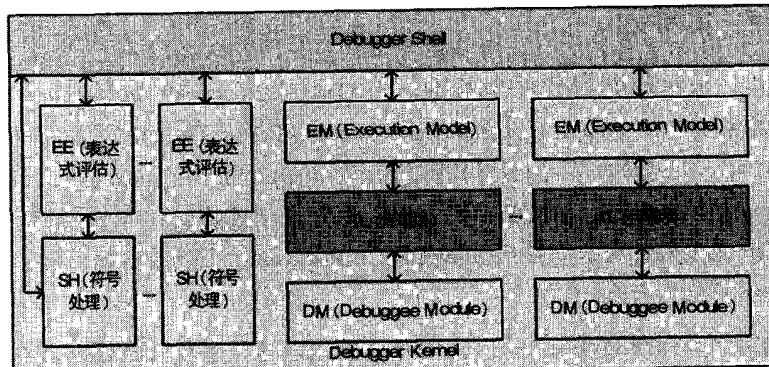


图 28-12 支持多进程和多种语言的经典调试器架构示意图

在图 28-12 所示的调试器中，当创建调试会话时，调试器会根据被调试程序的特征选择合适的执行模型（EM）。当有调试事件发生时，调试器可以根据调试事件中的线程 ID 和进程 ID 寻找到合适的 EM。因为每个角色都有多个实例，调试器内核通常使用链表来管理这些实例。

28.8 HPD 标准

尽管调试器已经成为软件工程中的最重要工具之一，但是迄今为止关于调试器的功能和用法还没有一种非常流行的书面标准。不同调试器所使用的用户界面、热键和所支持的命令大多是各不相同的。举例来说，微软公司的调试器和 Borland 公司的调试器使用了两种完全不同的热键定义（表 28-5）。

即使是同一个公司的调试器产品，不同版本间也可能存在较大不同。例如，VC6、VC7 和 VC8 调试器用来设置调试异常处理方法的对话框就有明显的差异（参见 30.9 节）。

表 28-5 两种风格的调试器热键

功能	Microsoft 调试器的热键	Borland 调试器的热键
运行（Run）	F5	F9
单步进入（Step Into）	F11	F7
单步跳过（Step Over）	F10	F8
单步跳出（Step Out）	Shift+F11	Alt+F8
运行到光标	Ctrl+F10	F4
重新开始（Restart）	Ctrl+Shift+F5	Ctrl+F2
切换断点	F9	F2

以上差异不仅给用户使用带来了不便，而且不利于调试技术的发展和交流，缺乏统一的标准是导致以上问题的关键。下面我们介绍一个名为 HPD 的调试器标准。

28.8.1 HPD 标准简介

HPD 标准的全称是高性能调试器标准，即 High Performance Debugger (HPD) Standard，它是由一个名为 High Performance Debugging Forum (HPDF) 的组织制定的。HPDF 于 1997 年 3 月成立，其目的是为高性能计算 (High-Performance Computing) 系统制定调试工具有关的标准。HPDF 组织的成员既有大学等研究机构，也有调试器开发厂商。HPDF 的主要赞助单位为 Parallel Tools Consortium，简称 Ptools，其官方网站为：<http://www.ptools.org/hpdf>。

HPDF 的第一个标准 HPD 版本 1 于 1998 年推出。这一标准所针对的调试目标主要是对性能有较高要求的并行程序，包括多线程、多进程以及可能在不同架构的计算

机系统上运行的软件。HPD 标准主要考虑的语言是 C、C++ 和 Fortran (F77 和 F90)。HPD 标准共分如下 4 个部分。

1. 高层概览 (High-Level Overview)，介绍 HPD 调试器的概念模型，调试器与用户程序的关系，并行性对调试器行为的影响，控制程序执行，状态模型（机器状态、程序状态和调试器状态），符号、名称和表达式的处理方法。
2. 命令描述，将调试命令分为如下 8 类逐一做了介绍：调试器接口（即设置和控制调试环境的命令）；进程和线程集合；调试会话的建立和终止；观察程序信息；显示和操纵数据；控制目标程序的执行；动作点 (Actionpoints)。
3. HPD 用户指南。
4. 命令语法归纳。

下面我们对 HPD 标准的主要内容分别进行说明。

28.8.2 动作点

HPD 标准把各种类型的断点和追踪点统称为动作点 (Actionpoints)。通过设置动作点，用户可以定义让目标程序中断到调试器中的条件和行为。HPD 标准共定义了 3 类动作点，分别是。

1. 断点 (Breakpoints)，即当程序执行到某个位置时中断下来，也就是通常意义上的代码断点。
2. 观察点 (Watchpoints)，当变量的值被改变时中断到调试器中，也就是数据断点。
3. 屏障 (Barriers)，当一个进程执行到这一点时，必须等待其他进程也执行到这一点才可以继续执行，用于多个进程同步。

每个动作点都有一个与其关联的触发集合 (Trigger Set) 和停止集合 (Stop Set)。前者定义了这个动作点所适用的线程，后者定义了这个动作点被触发后哪些线程应该被停下来。屏障的触发集合和停止集合必须是被调试进程中的所有线程。

28.8.3 进程和线程的表示和命名

HPD 标准针对的主要调试目标是多线程和多进程的并行计算软件，因此它定义了丰富的机制来支持多线程和多进程调试。首先，可以使用进程号和线程号来表示进程和线程，并且可以使用 * 通配符和范围符号 “—”。举例来说：

- 0.1——进程 0 的 1 号线程。
- 1.0——进程 1 的 0 号线程。
- 0.*——进程 0 中的所有线程。

- `0.*:4.*`——进程 0 到进程 4 之间的所有线程。
- `1.2:1.5`——进程 1 的 2 到 5 线程，也就是 1.2、1.3、1.4、1.5。
- `1.2:2.3`——从 1.3 到 2.5 范围内的所有线程，即进程 1 中除了线程 0 和 1 的所有线程再加上进程 2 的线程 0、1、2、3。

另外，为了更方便地引用一类线程，HPD 定义了 6 个特别的线程集合，并为它们分别定义了简短的名称。

- `all`: 与目标程序关联的所有线程。
- `running`: 所有目前在运行的线程，也就是所有处于运行状态的线程。
- `stopped`: 所有目前不在运行的线程，也就是处于 `stopped/runnable` 或者 `stopped/held` 状态的线程。
- `Runnable`: 所有能够运行的线程，也就是处于 `stopped/runnable` 状态的线程。
- `held`: 所有直到某些事件发生才能执行的线程，也就是处于 `stopped/held` 状态的线程。
- `exec (executable)`: 所有与一个特定的可执行文件相关联的线程。

有了以上定义，在调试时就可以使用以上名称来引用这类线程，不再需要先列出所有线程，再一个个地按线程 ID 来进行操作。

28.8.4 命令

HPD 标准共定义了 45 条命令（表 28-6），其中有些是要求一定要实现的，有些是可以选择实现的。

表 28-6 HPD 标准定义的命令

命令	必须实现	功能
调试器接口 (General Debugger Interface)		
#	否	用于开始注释行
alias	是	创建或观察用户定义的命令
unalias	是	删除用户定义的命令
history	是	观察本次调试会话所输入命令的历史记录
set	是	观察或设置调试器的状态变量
unset	是	将调试器的状态变量恢复成默认值
log	是	开始或停止记录 (log) 调试器的输入和输出
input	是	读取并执行保存在文件中的调试命令
info	是	显示调试器的环境信息
help	是	显示帮助信息
进程和线程集合 (Process/Thread Sets)		
focus	是	改变当前的进程或线程集合
defset	是	定义一个进程或线程集合，并为其指定一个名称

续表

命令	必须实现	功能
unset	是	取消 defset 命令定义的进程和线程集合
viewset	是	列出一个进程或线程集合所包含的成员
whichsets	是	列出一个进程或线程所属于的所有集合
调试器 (调试会话) 的初始化和终止 (Debugger Initialization/Termination)		
load	是	加载被调试程序的调试信息并准备执行
run	是	开始或重新开始执行目标程序
attach	是	附加进程，将一个 (或一些) 已经执行的进程加入调试会话
detach	是	分离进程，使目标进程脱离调试会话，并让其继续运行
kill	是	终止目标进程
core	是	加载进程的 core 文件映像 (DUMP)，以供分析
status	是	显示当前进程和线程的状态
quit 和 exit	是	终止调试会话
程序信息 (Program Information)		
list	是	显示源代码行
where	是	显示当前的执行位置和调用栈
up	是	在调用栈中向上移动一个或多个层次
down	是	在调用栈中向下移动一个或多个层次
what	否	判断目标程序中的一个符号名 (变量、过程等) 的指向和含义
显示和操纵数据 (Data Display and Manipulation)		
print	是	评估并显示变量或表达式的值
assign	是	改变变量的值
执行控制 (Execution Control)		
go	是	恢复目标程序的执行
step	是	单步执行，包括执行一条或多条语句 (repeat)、单步进入 (step into)、单步越过 (step over)、单步跳出 (step -finish)。
halt	是	将进程挂起 (suspend)
wait	是	阻塞命令输入直到进程停止
cont	否	恢复程序执行并阻塞命令输入 (直到有进程中断到调试器)
动作点 (Actionpoints)		
break	是	定义断点
barrier	是	定义屏障点
watch	是	定义观察点
actions	是	显示动作点列表
delete	否	删除动作点
disable	否	临时禁止动作点
enable	否	重新启用动作点
export	否	将动作点设置输出并保存起来供以后使用

HPD 标准推出后，已经被一些调试器所采用，特别是 UNIX 和 Linux 操作系统下的调试器，但是其影响力还是比较有限的。下一节将介绍 Java 平台的调试器标准，JPDA。

28.9 本章总结

本章的前 3 节介绍了调试器由简单到复杂的发展历史和有影响的著名调试器，然后介绍了调试器的一般功能和分类方法（第 28.4、28.5 节）。第 28.6、28.7 节分别介绍了调试器的实现模型和经典架构。第 28.8 节介绍了用于设计高性能调试器的 HPD 标准。从下一章开始我们将集中介绍广泛用于 Windows 操作系统的 WinDBG 调试器。

参考文献

1. FLIT—Flexowriter Interrogation Tape: A Symbolic Utility Program for TX-0
2. Alan Kotok. DEC Debugging Tape. Digital Equipment Corporation
3. TOPS-10 DDT Manual. Digital Equipment Corporation
4. TOPS-10/TOPS-20 DDT11 Manual. Digital Equipment Corporation
5. Joan M. Francioni and Cherri M. Pancake. High Performance Debugging Standards Effort

WinDBG 及其实现

本章将介绍 Windows 平台中的著名调试器 WinDBG。WinDBG 是专门针对 Windows NT 系列操作系统而设计的调试器。WinDBG 的最初版本是微软公司在开发最初 Windows NT 操作系统（NT 3.1）期间推出的，它是当时 NT 团队内部开发和调试 NT 操作系统的最主要工具。WinDBG 与 NT 系列操作系统有着密不可分的联系。

在 1993 年 NT 3.1 发布时，WinDBG 作为 NT 3.1 的附属工具开始对外发布。随后，NT 操作系统的每次升级，WinDBG 也会随之升级。很长一段时间里，WinDBG 的版本号与 NT 操作系统的版本号是一致的，比如为 NT 3.51 设计的 WinDBG 的版本号就是 WinDBG 3.51。这种状况一直持续到 Windows 2000 时代。

在 2000 年时，WinDBG 的代码经历了一次非常大的重构，软件架构进行了重大调整，很多模块都进行了重新设计，大量源代码重写，WinDBG 的版本号也复位到从 1.0 开始重新算起。重构前的 WinDBG 是完全使用 C 语言开发的，重构后使用了 C++ 语言，因此我们以这次重构为界，将 WinDBG 的历史分为两个阶段，前一阶段称为 C 阶段，后一阶段称为 P（Plus）阶段。

我们将先介绍 C 阶段的 WinDBG（29.1 节和 29.2 节），然后介绍重构的经过和变化（29.3 节）。从第 29.4 节开始我们将集中分析重构后的 WinDBG，先介绍基本架构（29.4 节），然后分为调试目标（29.5 节）、调试会话（29.6 节）、命令处理（29.7 节）三个部分介绍 WinDBG 的内部设计。

29.1 WinDBG 溯源

Windows NT 的第一个版本 NT 3.1 是从 1989 年开始正式编码，1993 年 7 月正式发布的。在开发 NT 的过程中，NT 团队一开始就意识到了调试这个复杂系统的重要性，因此在设计系统时就把调试功能看作是系统中必不可少的一个部分，而且优先实现这个部分。

29.1.1 KD 和 NTSD 诞生

大约在 1990 年年末，用于内核态调试的内核调试引擎 KD 和用于用户态调试的调试子系统初步完成。在同一时间，与以上部件配合的调试器也开始工作，这个调试器的名字叫 NTSD，全称为 Symbolic Debugger for NT，即用于 NT 系统的符号调试器。从此，NT 系统和 NTSD 调试器一起走上了成长之路，或者说 NT 是在 NTSD 调试器的帮助下一点点成长起来的。

通过 NTSD 项目可以构建出以下几个调试器程序，分别用来满足不同的调试需求。

- NTSD.exe：用户态调试器。
- i386kd.exe：内核调试器，用于调试运行在 x86 架构上的 NT 系统。
- alphakd.exe：内核调试器，用于调试运行在 Alpha 处理器上的 NT 系统。
- mipskd.exe：内核调试器，用于调试运行在 MIPS 处理器上的 NT 系统。
- ppckd.exe：内核调试器，用于调试运行在 PowerPC 处理器上的 NT 系统。

其中，KD 的含义是 Kernel Debugger。值得说明的一点是，因为在内核调试时，并不需要主机与被调试系统使用同样的处理器。举例来说，完全可以在 x86 系统上调试运行在 MIPS 架构上的 NT 系统。所以，以上内核调试器中的架构是指被调试系统的架构，而不是这个程序文件本身的 CPU 架构。事实上，以上每个程序又分为四种发行版本，分别用于 NT 所支持的四种 CPU 架构，参见图 29-1。为了行文方便，我们把四种 KD 调试器泛称为 KD 调试器或 KD。

KD 调试器与 NTSD 的很多命令是一样的，如设置断点，访问寄存器和内存，观察栈等。因此它们的很多源程序文件都是共享的，这也是它们共享一个项目的原因。

29.1.2 WinDBG 诞生

NTSD 和 KD 调试器都是以命令行方式工作的，没有充分发挥出 Windows 这个图形化操作系统的易用性特征。于是大约从 1992 年 4 月份开始，一个 GUI 版本的调试器开始开发了，它的名字就叫 WinDBG，意思是 Windows Debugger——窗口风格的调试器。

从时间上来看，WinDBG 是在 NTSD 和 KD 之后开发的，但是它并没有直接复用 NTSD 和 KD 的代码。其中的一个原因是 NTSD 和 KD 基本上是以单独的 EXE 方式组织的，它们并没有把公共的部分独立成易于被 WinDBG 共享的 DLL。尽管没有共享代码，WinDBG 是努力兼容 NTSD 和 KD 的命令和工作方式的。从用户的角度来看，用户可以使用同一套命令并得到基本一致的结果。

与 NTSD 和 KD 使用两个相对独立的 EXE 分别做用户态调试和内核态调试不同，

WinDBG.EXE 既可以作为用户态调试器，也可以做内核态调试器来使用。从架构上来讲，WinDBG 将这个调试器划分成外壳（Shell）、传输层（Transport Layer）等多个层次，并将相对独立的功能封装在动态链接库（DLL）中，使整个软件更容易扩展和维护，并且可以方便地支持远程调试，我们将在下一节详细讨论其中的细节。

29.1.3 发行方式

下面我们简要介绍以上调试工具的发行方式。首先，NTSD.EXE 是作为 NT 操作系统的一个模块随 NT 操作系统一起发行的，对于 x86 结构，它位于安装盘的 i386 目录下，安装之后，位于 system32 目录下，典型的路径为：c:\winnt\system32\ntsd.exe。因为这个路径是系统寻找可执行文件的默认路径之一，所以在一个安装好的 NT 系统中，不论当前在任何位置（目录），只要输入 ntsd 就可以启动它，这种状况一直持续到 Windows XP 时代。Windows Vista 的 system32 目录中不再包含 NTSD。

NTSD 的另一种发行方式就是与 KD 一起作为 NT 系统的支持工具而发行，它们位于安装光盘的\Support\Debug 目录中。在这个目录下首先以 CPU 架构命名了 4 个子目录，分别为 ALPHA、I386、MIPS 和 PPC（见图 29-1），每个子目录中存放的是适合在这种 CPU 架构上运行的调试器。对于 KD 调试器，又分为 4 个模块文件，分别用来调试不同的目标系统。

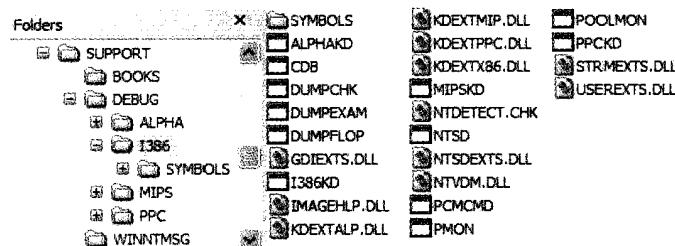


图 29-1 位于 NT 操作系统安装光盘上的 KD 调试器

KD 调试器的第二种发行方式就是作为调试工具包含在 DDK（设备驱动程序开发包）中。图 29-2 显示了 NT 3.51 DDK 目录树的一部分。从图中可以看出，在 DDK 的 BIN 目录下有 ALPHA、I386、MIPS、PPC 4 个子目录，分别对应于当时 NT 所支持的 4 种 CPU 架构。在每个目录下分别存放了在该架构下开发驱动程序所需的工具。在右侧的文件列表中，我们可以看到经典的 BUILD 工具、内核 Profiling 工具（Kernel Profiler）KERNPROF、磁盘监视工具 DiskMon 和 3 个 KD 调试器，包括 AlphaKD、I386KD 和 MIPSKD。这些 KD 调试器的版本号都是 3.51.1029.1，与一同发行的 NT 操作系统的版本号是一致的。

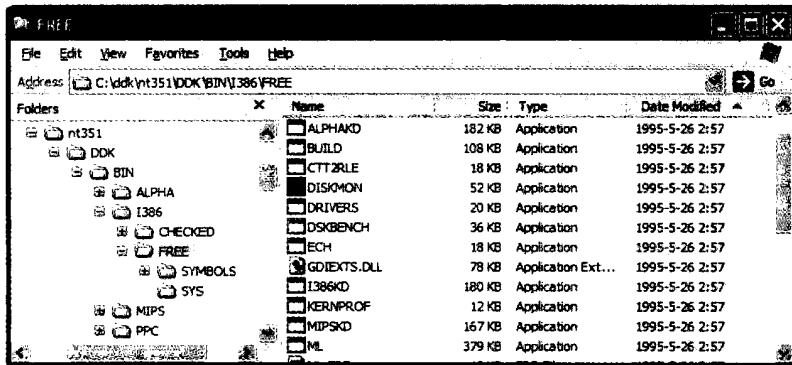


图 29-2 NT 3.51 DDK 的文件布局

在上面所示的目录中，我们还可以看到三个调试器扩展命令模块，分别为 Gdiexts.dll、Strmexts.dll 和 Userexts.dll，它们分别用来支持调试 GDI、流和窗口系统（USER）。

NT 3.51 DDK 的发行时间是 1995 年 5 月，NT 4.0 DDK 的发行时间是 1996 年 7 月。WinDBG 的开发时间是 1992 年。因此当以上的 DDK 发行时，WinDBG 的早期版本已经开发完成了。但也许当时的软件高手们还都习惯于使用命令行窗口，甚至有些藐视 WinDBG 这样的窗口程序。所以在以上两个 DDK 目录下，我们找不到 WinDBG.exe 和它的附属文件。但是在笔者从 MSDN 订阅 (MSDN Subscriber) 网站上下载的 NT 3.51 DDK 包中，与 DDK 目录并列的还有一个 HCT 目录。在这个目录中包含了 WinDBG (见图 29-3)。HCT 是 Hardware Compatibility Test，意思是硬件兼容性测试，它是微软提供给硬件和软件（驱动程序）开发商的一个测试工具包，用来测试硬件设备和驱动程序的兼容性，通过 HCT 测试是得到 WHQL 徽标的必要条件。

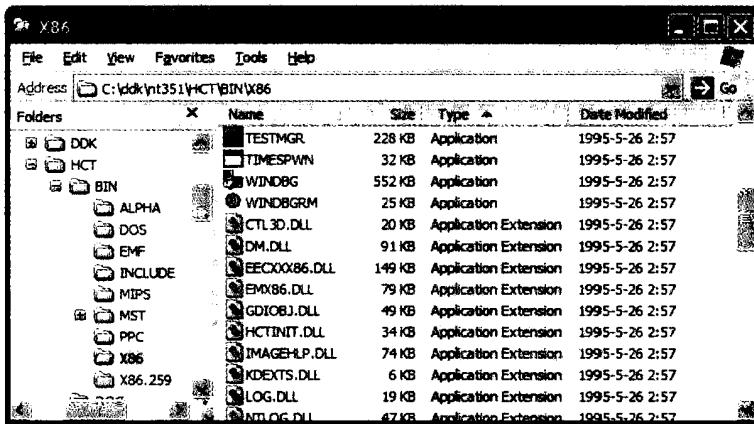


图 29-3 随 HCT 工具一起发行的 WinDBG

与 DDK 的 BIN 目录类似，在 HCT 的 BIN 目录下，也按处理器架构有四个子

目录，这四个子目录中的文件内容几乎一样，只是适用于不同的 CPU 架构。图 29-3 显示了用于 x86 架构的部分文件，其中可以看到管理 HCT 测试的主程序 TESTMGR.exe，WinDBG 的主程序 WinDBG.exe，还有 WinDBG 的几个重要模块 DM.DLL（Debuggee Module）、EECXXX86.DLL（表达式评估）、EMX86.DLL（执行模型），WinDBG 的内核调试扩展命令模块 KDEXTS.DLL。图中没有显示出来的 WinDBG 模块还有它的几个传输层 DLL，分别是 Tlpipe.dll（命名管道传输层）、Tlser.dll（串行口传输层）、Tlser32.dll（串行口传输层）、Tlloc.dll（本地传输层），以上文件的版本号为 3.51.1035.1。

在 2000 年 8 月的 Windows 2000 DDK 中，笔者高兴地看到了 WinDBGexe 和它的支持模块，这个 WinDBG 的版本是 5.0.2195.1163。这个版本的 WinDBG 仍然可以在笔者的 Windows XP SP2 系统上较好地运行，而且从菜单结构和功能上看，与今天的 WinDBG 已经很接近（图 29-4）。虽然笔者不能肯定地说这是 C 阶段 WinDBG 的最后一个版本，但是这时重构后的 WinDBG 已经开始公开测试了。所以可以确定地说，即使有 C 阶段的更高版本，那也不会有显著的功能变化了，因此我们后面对 C 阶段 WinDBG 的架构分析和更多的介绍都是基于这个版本的。

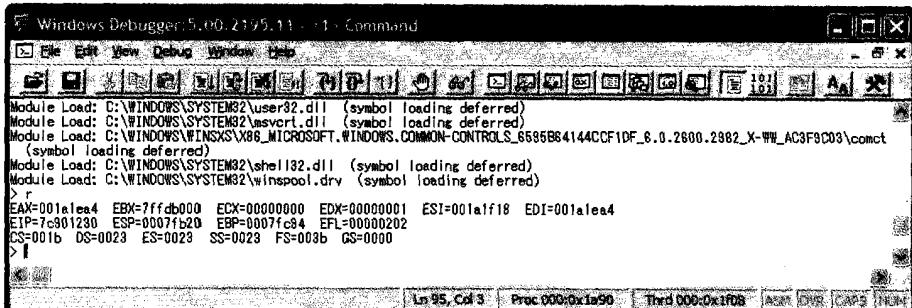


图 29-4 WinDBG (C 阶段) 的典型界面

根据 DDK 中与 WinDBGexe 同目录的 debuggers.txt 的说明，在 Windows 2000 时代，WinDBG 有以下三种发行方式：Platform SDK、Windows 2000 DDK、Windows 2000 Customer Support Diagnostics CD。

在 NT 3.51 和 NT 4 DDK 的 DDK\SRC\KRNLDDBG 子目录下包含了两个与内核调试关系密切的源程序项目，一个是 Kdapis，内部包含了内核调试 API (DbgKdApi) 的头文件 (Dbgkapi.h 和 WinDBGkd.h) 和源代码 (Dbgkapi.c)，其中的 Packet.c 包含了与内核调试引擎进行通信的细节。这个项目中的 givit.c 演示了如何使用内核调试 API 来建立一个简单的内核调试器，其中的 main 函数中包含了基本的调试循环，包括建立连接，等待调试事件，处理和继续等。

另一个项目叫 Kdexts，它演示了如何编写内核调试扩展模块。其中的源代码包含

了 igrep（搜索指令流）、str（输出 ANSI 字符串）、ustr（输出 Unicode 字符串）和 obja（显示内核对象属性）4 个扩展命令的实现。

在 Windows 2000 的 DDK 中，以上两个例子被删除了，但是在 INC 目录里保留了 WinDBGkd.h。在 XP 的 DDK 中，这个头文件也被移除了，此后的 DDK 没有再恢复这些内容。

29.1.4 版本历史

表 29-1 列出了 C 阶段的 WinDBG.exe 和 NTSD.exe 的重要版本，包括文件日期和简单说明。

表 29-1 C 阶段的 WinDBG.exe、I386KD.exe 和 NTSD.exe 的重要版本

文件	典型版本	文件日期	说明
I386KD.exe	3.51.1029.1	05/26/1995	NT 3.51 DDK 包含的版本
WinDBG.exe	3.51.1035.1	05/26/1995	NT 3.51 HCT 工具中包含的版本
I386KD.exe	4.0.1381.1	07/26/1996	NT 4.0 DDK 包含的版本
I386KD.exe	5.0.2184.1	07/19/2000	Windows 2000 DDK 包含的版本
WinDBG.exe	5.00.1867.1	不详	1999 年 10 月 1 日发布的 OEM Support Tools Version 2 SR2 中提到的版本
WinDBG.exe	5.00.2184.1	不详	2000 年 6 月 23 日发布的 OEM Support Tools Phase 3 Service Release 2 中提到的版本
WinDBG.exe	5.00.2195.1163	07/19/2000	Windows 2000 DDK 中的版本

以 5.0 版本的 WinDBG 为例，C 阶段的 WinDBG 调试器已经具有目前版本所支持的大多数功能，包括用户态调试，活动内核调试，分析系统崩溃转储（Dump），JIT 调试，远程调试和通过扩展模块支持扩展命令等。

29.2 C 阶段的架构

C 阶段（重构前）的 WinDBG 采用的是上一章介绍的经典调试器架构。本节我们将简要地介绍用于实现这一架构的各个模块，并分析远程调试功能的实现。为了行文简洁，本节下文中的 WinDBG 就是指 C 阶段的 WinDBG。

29.2.1 功能模块

图 29-5 画出了 WinDBG 的主要模块和它们在经典架构中所承担的角色。其中的文件名称来自 NT 3.51 的 HCT 包中的 WinDBG 3.51，其后版本的文件名略有变化，比如 5.0 版本的 WinDBG（对应于 Windows 2000）中的 EE 模块的名字叫 Eecxx.dll。

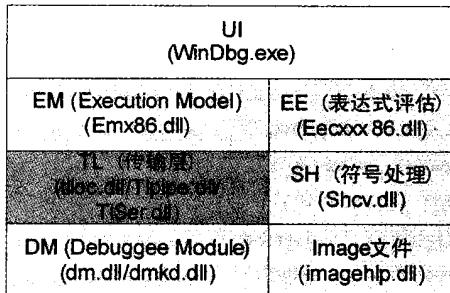


图 29-5 WinDBG (C 阶段) 的架构

下面分别介绍一个每个模块的用途。

WinDBG.EXE: 主程序文件，用于显示用户界面和与用户进行交互。

Dm.DLL: DM 模块，封装了访问和管理被调试程序的一系列函数，包括读写内存，设置和读取线程上下文等。这个 DLL 用于用户态调试，还有一个 DMKD.DLL 用于内核态调试。

Eecxxx86. DLL: EeCxx 是 Expression Evaluator of C Plus Plus 的缩写，因此这个模块用于评估和处理使用 C++语法的各种表达式（x86 平台）。

Emx86. DLL: 执行模型模块，这个模块封装用来启动调试会话和控制被调试程序执行的各种函数，比如附加到进程、链接被调试系统、冻结线程、恢复执行、设置删除断点和观察点、单步执行等。

Tlloc. DLL: 传输层模块，Tlloc 是 Transport Layer Local 的缩写，负责与本地的 DM 模块通信，发送请求和收发通信包。类似的模块还有 Tlpipe.dll（使用命名管道来进行通信）、Tlser.dll（使用串行口通信）、Tlser32.dll（串行口通信）。

Shcv. DLL: 符号处理器（Symbol Handler）模块，CV 是 CodeView 的缩写，因此这个模块用来处理 CodeView 格式的调试符号。

以上模块都是动态加载的，WinDBGexe 会根据命令行参数和调试类型加载合适的传输层 DLL，然后加载合适的 EM 模块。EM 加载后会调用 TL 的服务函数让其加载指定的 DM 模块。这样，对于本地调试，本地的 TL 会将 DM 模块加载到当前进程中。对于远程调试，TL 的远程部分会将 DM 模块加载到目标机器上的远程进程中。

29.2.2 远程调试

图 29-6 画出了 WinDBG 的远程调试示意图，左侧是目标机器，右侧是主机。WinDBG 提供了两个可以支持远程调试的传输层 DLL，一个是 TlSer.dll，需要通过一根 Null-Modem 线缆分别连接在主机和目标机的串行口（COM 口）上；另一个是 TlPipe.dll，使用命名管道来通过网络进行通信。

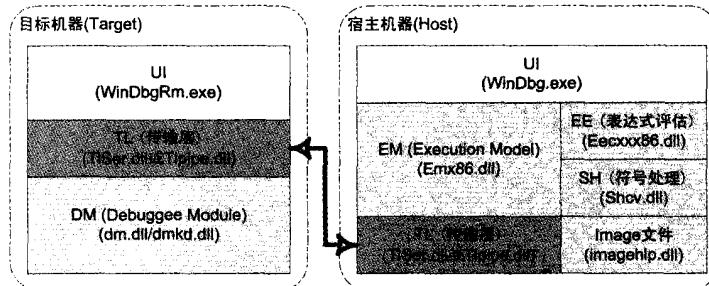


图 29-6 WinDBG (C 阶段) 的远程调试模型

在目标机器上启动 WinDBGRm 后，其初始界面如图 29-7 所示，其中的提示信息告诉我们默认的传输层 DLL (TlPipe.dll) 已经加载。观察此时的 WinDBGRm 进程，DM 模块还没有加载。



图 29-7 在目标机器上运行的 WinDBGRm 程序

接下来在主机上启动 WinDBG，在 View 菜单选择 Options 打开选项对话框，将传输层改为 Pipes，如图 29-8 所示。

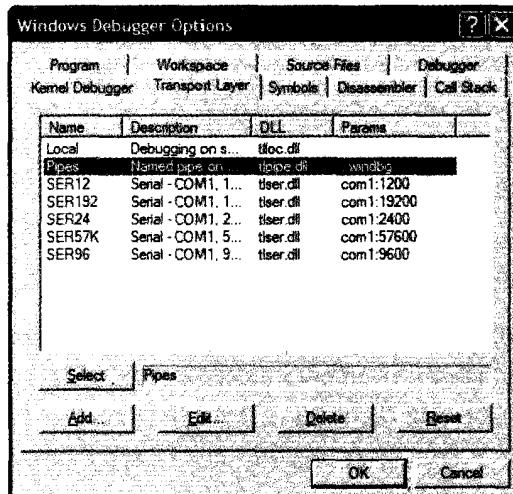


图 29-8 WinDBG 的选择传输层界面

然后再选择 Debug 菜单的 Attach to a Process，或者使用.attach 命令就可以将调试器附加到一个被调试程序了。值得注意的是，这时无论是 Attach to a Process 对话框所显示的进程，还是.attach 命令的参数中指定的进程，都是目标机器中的进程。

成功建立调试会话后，WinDBGRm 的显示会变为图 29-9 所示的样子。



图 29-9 与主机上的 WinDBG 成功建立连接后的 WinDBG Rm 程序

此时再观察 WinDBG Rm 进程，它已经加载了 DM 模块，使用的是同一目录下的 DM.DLL。这时就可以在主机的 WinDBG 上调试目标机器上的程序了。

因为 C 阶段的 WinDBG 已经不再具有广泛的使用价值，所以本节的目的主要是为了帮助大家进一步理解调试器的经典结构和 WinDBG 的发展历史。

29.3 重构

C 阶段的 WinDBG 从 1992 年开始设计，到 2000 年的 5.0 版本，经历了大约 8 年时间。8 年对于一个通用软件来说不算一个很短的时间。这 8 年中，无论是整个软件工业，还是微软的软件产品线都有了很大发展，也出现了很多新的软件技术和开发方式，例如 COM 技术以及 2000 年开始推出的.NET 技术。作为 Windows 平台中的最重要调试器，WinDBG 也是微软内部使用的主要调试工具，他们在开发新产品的时候，经常会希望 WinDBG 能够加入新的调试功能来支持新产品或者技术。加入新支持的一种最常用方法就是编写扩展模块，但是这不能解决所有问题，有时还是需要对 WinDBG 的核心模块进行修改才能支持新的功能。

因为旧的 WinDBG 是完全使用 C 语言编写的，所以在上面不断加入新的功能是比较困难的。于是大约在 1999 年或者 2000 年上半年时，WinDBG 团队很自然地想到了使用 C++ 和新的技术来重写 WinDBG。

29.3.1 版本历史

重构后的 WinDBG 最先是在 2000 年 4 月的 WinHec 会议上提供给参会者预览的。2000 年 6 月 12 日 WinDBG 团队在 OSR 的 WinDBG 邮件组中发布了一条消息，题目为 New test release of WinDBG，鼓励大家下载新的 WinDBG 测试版本，这是新版的 WinDBG 第一次完全公开。

非常幸运的是，作者在写作此内容时，从一张 Windows 2000 DDK 光盘上发现了 WinDBG Beta 1 版本，它位于光盘根目录下的 Debuggers 子目录中。这也许是重构后的 WinDBG 第一次随 DDK 一起发布。在这张光盘上仍然包含了重构前的 WinDBG，因此它是同时包含两种 WinDBG 的 DDK。

WinDBG Beta 1 的安装文件时间和安装之后的发布说明文件（relnotes.txt）中的时间都是 2000 年 7 月。另一点值得注意的是，Beta 1 中的 WinDBG.exe 和 DbgEng.dll 的

版本号分别为 5.1.2250.3 和 5.1.2250.5。这说明 Beta 1 时，重构后（P 阶段）的 WinDBG 还在延续以前的版本号。

在 OSR 的 WinDBG 邮件组中，有一条 2000 年 9 月 26 日发布的消息，题目为 StackTrace Failed from new WinDBG，其中提到的 WinDBG 的版本号为 1.0.0006.0。笔者认为这是 WinDBG 重构后的第一个正式发布版本。

表 29-2 列出了 P 阶段的 WinDBG 的主要版本，包括每个版本的文件时间和简要说明。

表 29-2 P 阶段的 WinDBG 主要版本

版本号	时间	说明
不详	2000 年 4 月	WinHec 会议的预览版本
不详	06/12/2000*	第一个公开测试版本
5.1.2250.3**	07/11/2000**	Beta 1 版本
1.0.0006.0	09/21/2000	重构化的第一个正式版本
2.0.23.0	02/08/2001	WinDBG 2.0 的流行版本，引入 dbgsrv.exe 和 1394 传输支持（内核和远程调试）
3.0.10.0	05/08/2001	3.0 的 Beta 版本，曾经包含在 2001 年 6 月的 Platform SDK 中，引入分支单步（tb）
3.0.20.0	07/26/2001	3.0 的正式版本，加入对 Windows XP 的增强，增加.kdfiles 命令
4.0.11.0	10/19/2001	4.0 的 Beta 版本
4.0.18.0	12/17/2001	4.0 的正式版本，引入新的 1394 调试协议
6.0.17.0	06/04/2002	引入 symchk 工具，pc、tc 命令，支持同时加载多个转储文件
6.2.13.1	11/27/2003	引入.send_file、.record_branches、.ignore_missing_pages、.quit_lock、.ttime 和.fpo 命令
6.3.17	07/20/2004	引入 Windows Vista 和 Server 2008 支持和更多的流程控制命令
6.4.7.2	01/21/2005	Windows Server 2003 SP1 DDK 中包含了这个版本，引入 SymProxy 工具、内核流扩展（KS.DLL）以及 gc、.event_code、.fnret 命令
6.5.3.8	08/10/2005	引入 USB2.0 支持、DBH 工具和 EngExtCpp 扩展模块扩展
6.6.7.5	07/18/2006	界面增强，内核调试连接对话框增加 USB2.0 和 VPC 支持，彩色显示源代码
6.7.5.0	04/26/2007	将关于扩展模块的帮助内容（debugext.chm）合并到 WinDBG 帮助文件中；开始被纳入到 WDK 中
6.7.5.1	06/20/2007	引入 Convertstore 工具，默认安装 SDK

* 此时间为 WinDBG 团队发布在 OSR 邮件组的邮件时间 <http://www.osronline.com/showThread.cfm?link=2666>

** 分别是 WinDBG.exe 的版本号和文件时间。

值得说明的是，因为 WinDBG 的 Beta 版本使用的版本号是 5.1.2250.3，为了避免混淆，重构后的正式版本没有再使用主版本号 5，从版本 4 直接跳到了版本 6。

29.3.2 界面变化

为了让以前的用户不感觉到陌生，重构后的 WinDBG 在用户界面方面保持了原来的风格，菜单布局及快捷键等都没有大的变化，但是以下几个方面有了变化。

首先是命令输入位置改变了，从图 29-3 可以看出，重构前的命令提示符就在命令窗口的用户区内，因此是浮动的，与控制台窗口的工作方式类似。重构之后，命令提示符改为命令窗口下方的一个固定横条，即今天的样子（见图 29-10）。

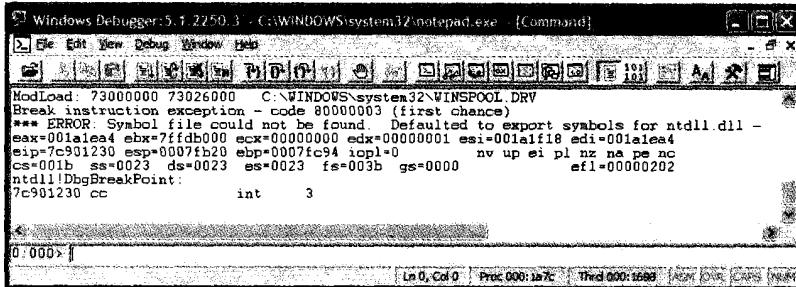


图 29-10 重构后的 WinDBG 界面

另一个变化是源代码的显示颜色，重构前的源代码窗口会按照语法以彩色的方式显示源程序。重构后去掉了这个特征，所有代码都是以黑白方式显示的。但是从 6.6.7.5 版本开始又重新加入了彩色显示。

笔者注意到的第三个界面变化是选项对话框，在重构前的 5.00.2195.1163 版本中，选项对话框包含很多个页面（见图 29-11），分别用来设置调试符号、反汇编、函数调用栈、传输层、内核调试、工作空间、源程序文件、程序启动参数等选项。

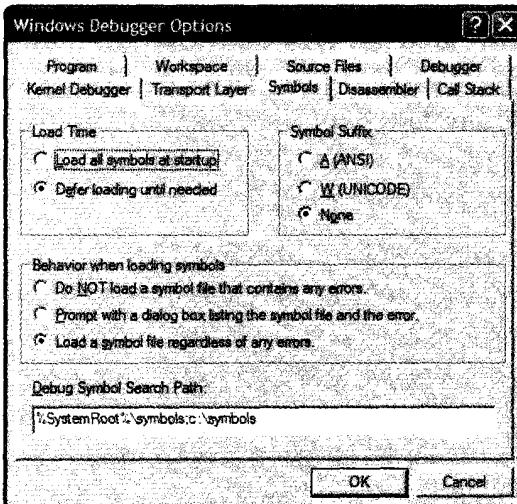


图 29-11 重构前的 WinDBG 的选项对话框

重构后的选项对话框比以前简单了很多（图 29-12），很多功能改成通过命令来设置，符号、映像文件和源文件的路径设置被移到文件菜单中。

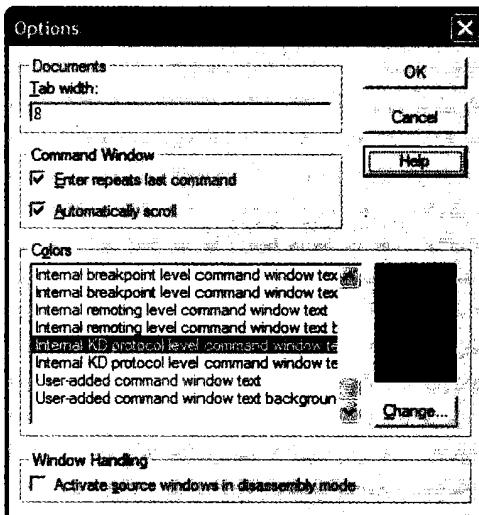


图 29-12 重构后的 WinDBG (2.0.0023.0 版本) 选项对话框

调试事件设置对话框在重构后也有所变化，重构前仅限于设置异常，启动对话框的菜单项为 Exceptions。重构后，除了设置异常事件外，还加入模块加载、线程启动和退出等调试事件，相应的启动菜单项的名称改为 Event Filters，但位置仍位于 Debug 菜单下。

29.3.3 模块变化

重构后的一个最大变化就是将公共的调试功能集中到一个动态链接库 (DLL) 中，EXE 模块中只保留 UI 和各自的特有功能。这个公共的 DLL 模块被称为调试器引擎 (Debugger Engine)，它的文件名为 DBGENGDLL。在调试器引擎之上，是不同形式的调试器接口程序，它们仍然保持着重构前的文件名，即 WinDBG.exe、NTSD.exe、CDB.exe 和 KD.exe。图 29-13 画出了这些模块的相互关系。

将公共代码统一到调试器引擎模块的好处除了更好地复用代码之外，从用户角度来看，使用不同调试器执行同一个命令可以得到更一致的结果。当然这样做也存在明显的缺点，从软件架构和面向对象的角度来讲，把很多不同的职能都集中到一个模块中是不合理的，缺少必要的模块划分，不利于开发和调试。这样做也导致编译好的调试器引擎文件比较大，比如 6.7.5.0 版本的调试器引擎文件的大小为 3,147,608 字节，2.0.23.0 版本的大小为 1,156,608 字节。因为所有调试器都依赖并加载这个文件，所以对于 KD 这样的内核调试调试器和 NTSD 与 CDB 这样的用户态调试器来说，调试器引擎中的某些代码和数据是冗余的，存在着空间和时间上的浪费。

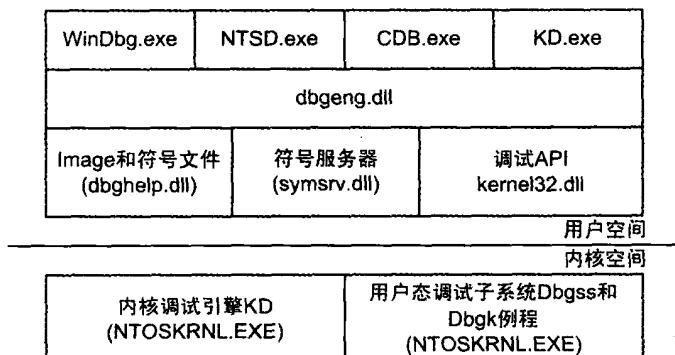


图 29-13 构建在共同调试器引擎上的多个调试器

29.3.4 发布方式和 NTSD 问题

WinDBG 重构后，以 WinDBG 为核心的一系列工具被集中到一个软件包，命名为 Microsoft Debugging Tools for Windows，我们将其简称为 WinDBG 工具包。

尽管 Windows 平台 SDK、DDK 和 WDK 都包含了 WinDBG 工具包，但是用户获取 WinDBG 工具包的最主要方式还是从微软网站自由下载，因为这样获得的版本最新。

与 Windows XP 系统一起发布的 NTSD 是重构后的 NTSD 第一次与操作系统一起发行。在安装好的 Windows XP 系统中，system32 目录下有 NTSD.exe 和与其对应的 DBGENG.DLL。为了防止与 Windows 系统中预装的 NTSD 不一致，在 WinDBG 工具包的早期版本（Beta、版本 1 和版本 2）没有包含 NTSD.exe。但 WinDBG 工具包的版本 3 开始包含 NTSD.exe，这导致的一个问题就是工具包中的 NTSD 与系统中的 NTSD 不一致。

也许是因为新的 NTSD 要依赖于庞大的调试器引擎模块（DbgEng.dll），也许是因为无论如何也赶不上网站版本的更新速度，也许有其他的原因，重构后的 NTSD 自从加入到 WindowsXP 系统后，就没有被更新过，在其后的几次操作系统更新中（SP1，SP2），系统自带的 NTSD 都没有更新，其版本号定格为 5.1.2600.0，名称也始终为 Symbolic Debugger for Windows 2000。举例来说，笔者比较了 2001 年 8 月 23 日的 Windows XP Professional 试用版本中的 NTSD.exe 和笔者目前机器上升级到 Windows XP SP2 后的 System32 目录下的 NTSD.exe，这两个文件的版本、文件大小和内容完全一样。在 Windows Vista 的系统目录中不再包含 NTSD。

29.3.5 文件

除了几种形式的调试器外，WinDBG 工具包中还有很多有价值的工具和文档。为

了帮助大家全面了解这些资源，表 29-3 列出了 6.8.4.0 版本的 WinDBG 工具包所包含的所有文件和一级子目录，并简要描述了每个文件和目录的用途。

表 29-3 WinDBG 工具包的文件列表

文件名	文件大小	描述
adplus.vbs	204,793	用于自动产生转储文件的 VB 脚本
agestore.exe	36,288	用于删除下游符号库或者源文件库中的过时文件
breakin.exe	19,904	将指定进程中断到调试器
cdb.exe	291,776	控制台界面的调试器
convertstore.exe	31,680	转化符号库
dbengprx.exe	112,576	用于远程调试的转发器 (repeater)，小的代理服务器
dbgeng.dll	3,150,272	调试器引擎，包含了大多数核心调试功能
dbghelp.dll	1,040,320	操作系统的调试辅助库
dbgrpc.exe	38,336	用于调试 RPC 的辅助工具
dbgsrv.exe	34,752	用于远程用户态调试的进程服务器 (Process Server)
dbh.exe	151,488	调试辅助库的外壳工具，用于调用库中的 API
debugger.chi	298,054	帮助文件的辅助文件
debugger.chm	4,071,064	帮助文件
decem.dll	419,776	IA64 反汇编模块
dml.doc	55,296	介绍 DML (Debugger Markup Language) 语言用法的文档
dumpchk.exe	20,928	检查内存转储文件
dumpexam.exe	19,904	用于分析转储文件，已经过时
gflags.exe	122,816	修改 PE 程序的全局标志 (Global Flags)
i386kd.exe	290,240	用于调试 x86 系统的内核调试器 (控制台界面)
ia64kd.exe	290,240	用于调试 IA 64 系统的内核调试器 (控制台界面)
kd.exe	303,552	控制台界面的内核调试器
kdbgctrl.exe	36,288	动态改变内核调试选项
kdsrv.exe	154,560	用于远程内核调试的服务器
kernel_debugging_tutorial.doc	1,280,512	内核调试教程
kill.exe	33,728	用于杀死应用程序的小工具
license.txt	9,562	使用许可
list.exe	63,936	工作在控制台窗口的文件显示工具
logger.exe	72,128	监视 API 调用的工具
logviewer.exe	161,216	阅读 logger.exe 产生的记录文件 (.lgv)
ntsd.exe	292,288	NT 系统中最早的用户态调试器，运行在控制台窗口
pdbcopy.exe	26,560	复制调试符号
redist.txt	64	允许用户放入到自己软件中发布的文件列表
relnotes.txt	13,128	发布说明，里面包含当前版本的改动
remote.exe	57,280	用于远程调试

续表

文件名	文件大小	描述
rtlist.exe	27,584	列出远程系统的任务列表 (Remote Task List Viewer)
srcsrv.dll	95,168	访问源文件服务器的 DLL 模块
symbolcheck.dll	33,216	供 symchk.exe 工具使用的 DLL 模块
symchk.exe	81,344	检查指定的模块 (EXE 或 DLL) 是否有配套的符号文件
symsrv.dll	125,376	访问符号服务器的 DLL 模块
symsrv.yes	1	标记用途
symstore.exe	142,784	管理符号库, 增加、删除和查询符号文件
tlist.exe	40,384	显示进程列表
tools.doc	46,592	介绍 PDBCopy 和 DBH 用法的文档
umdh.exe	79,808	堆转储工具, 参见 23 章
WinDBG.exe	522,176	图形接口 (GUI) 的调试器
1394	<DIR>	以 1394 方式进行内核调试时主机端使用的驱动程序
clr10	<DIR>	用于调试托管程序的 SOS 扩展命令模块
nt4chk	<DIR>	调试 NT4 Check 版本目标系统时的扩展命令模块
nt4fre	<DIR>	调试 NT4 Free 版本目标系统时的扩展命令模块
sdk	<DIR>	编写 WinDBG 扩展命令的 SDK
sympoxy	<DIR>	构建符号服务器所需的文件和文档
Themes	<DIR>	定义窗口布局的示例文件和说明文档
triage	<DIR>	包含了 PoolTag.txt 和 Traige.ini, 前者列出了内核池分配标记所对应的模块名称, 供!poolused 命令使用, 后者用来定义模块或者函数的负责人, 供!owner 和!analyze 命令使用
usb	<DIR>	使用 USB 2.0 进行内核调试时, 主机端需使用的驱动程序
w2kchk	<DIR>	调试 Windows 2000 Check 版本目标系统时的扩展命令模块
w2kfre	<DIR>	调试 Windows 2000 Free 版本目标系统时的扩展命令模块
winext	<DIR>	公共的扩展命令模块
winxp	<DIR>	调试 Windows XP 目标系统时的扩展命令模块

本节介绍了重构后的 WinDBG 调试器的概况, 从下一节开始我们将从不同角度对其作深入介绍。

29.4 调试器引擎的架构

重构后的 WinDBG 将大多数调试功能都集中到一个调试器引擎模块(DbgEng.DLL)中。除了 WinDBG 系列调试器都是建立在这个公共的调试器引擎之上外, 用户也可以在自己的软件中调用这个引擎中的功能, 也可以基于它开发一个新的调试器。WinDBG 的帮助文件包含了对调试器引擎的简单介绍, 包括它的角色和它的对外接口, 但是并没有介绍它的内部设计。

29.4.1 概览

图 29-14 画出了 WinDBG 调试器引擎的架构示意图，其中画出调试器引擎的主要接口和内部类。最上方是使用调试器引擎的应用程序（Application），可以是 WinDBG.exe、CDB.exe、NTSD.exe、KD.exe 这些 WinDBG 工具包的调试器，也可以是用户自己开发的应用程序。

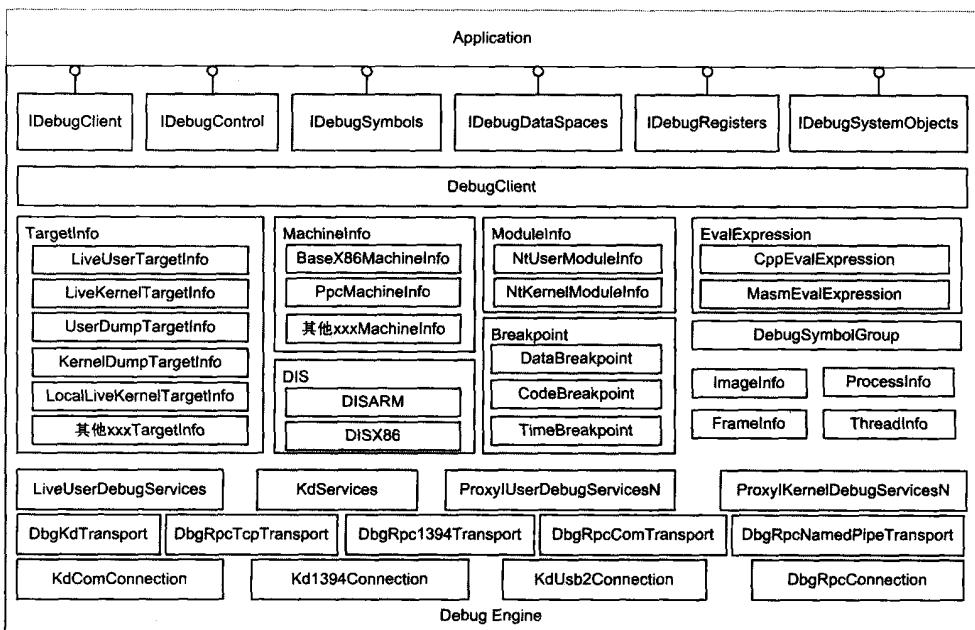


图 29-14 调试器引擎的架构

调试器引擎内部包含了一百多个 C++ 类和接口，整个模块输出的符号（类方法、变量等）超过 1 万个。为了便于理解和分析，我们把调试器引擎的分成如下六个逻辑层。

公开接口层：这一层定义了调试器引擎对外的 6 个公开接口，是调试器引擎与应用程序交互的渠道。

DebugClient 类：这是调试器引擎内部的一个主要类，它包装了几乎所有的调试功能。从层次上来说，这个类为公开接口层调用下面的调试服务提供了一种简洁的方式，隐藏了下层的复杂性。除了为公开的接口提供来自本地的服务外，使用 DebugClient 类的另一种情况就是转发器程序（Repeater）通过 ProxyIDebugxxx 系列类调用它。ProxyIDebugxxx 系列类的接口定义与公开接口层的各个接口定义是完全一样的。WinDBG 工具包包含了一个转发器程序，文件名是为 dbengprx.exe。

中间层：这一层将与被调试程序有关的信息和调试任务封装为很多个 C++ 类，例

如 `EvalExpression` 类用来评估表达式，它的两个子类 `MasmEvalExpression` 和 `CppClassEvalExpression` 分别用来解析和评估宏汇编和 C/C++ 表达式；`TargetInfo` 类是对调试目标（被调试程序）的封装，它有很多个子类分别代表不同类型的调试目标；`MachineInfo` 类用来包装 CPU 架构有关的信息，它的每个子类代表一种指令架构，`TargetInfo` 和 `MachineInfo` 类加起来相当于经典调试器模型中的 EM（Execute Model）；`ModuleInfo` 和 `ImageInfo` 类封装了模块、执行映像、以及与之配套的符号文件有关的属性和方法；`BreakPoint` 类用来记录断点有关的信息。

服务（Services）层：这一层提供对调试目标的访问和控制，类似于经典模型中的 DM（Debuggee Module）。例如，`LiveUserDebugServices` 类提供了访问和操纵用户态活动目标的很多方法，负责执行读写目标内存、设置断点、接收调试事件等任务；`KdServices` 用于内核调试；`ProxyIUserDebugServicesN` 用在远程调试的情况下，它通过其下的传输层与远程的另一个调试器引擎实例通信。

传输层：在内核调试或者远程调试的情况下负责传输信息，将信息组织成适合传输的数据包后交给连接层进行发送，同时也负责接收来自连接层的数据。

连接层：负责建立和维护数据连接，以及实际的数据收发工作。

下面我们将对以上各层分别作进一步的介绍。

29.4.2 对外接口

调试器引擎对外公开了如下 6 个接口。

1. **IDebugClient**: 负责启动和结束调试会话，设置和管理各种回调对象（callbacks），处理转储文件。
2. **IDebugControl**: 控制调试目标（如断点）、反汇编、评估表达式、执行命令和设置调试器引擎的选项。
3. **IDebugDataSpaces**: 访问调试目标的数据，如读写内存（虚拟地址、物理地址）、IO 空间、MSR 寄存器、总线数据等。
4. **IDebugSystemObjects**: 用于访问调试目标的系统对象，例如进程、线程、PEB（进程环境块）和 TEB（线程环境块）结构等。
5. **IDebugSymbols**: 读取和设置符号文件、源文件和模块文件的搜索路径，以及各种方式搜索符号，设置符号选项。
6. **IDebugRegisters**: 读取和修改调试目标的寄存器。
7. **IDebugAdvanced**: 读取和设置线程的上下文结构（Context），获取源文件信息，通过 Request 方法直接向调试器引擎发送请求。

随着 WinDBG 的功能丰富、发展与变化，上述接口也在变化，因此每个接口都有几个版本，以接口名中的数字来表示，例如 IDebugClient5, IDebugControl5 等。WinDBG SDK 的 dbgeng.h 中定义了以上所有接口的 IID 以及每个接口的声明和有关的常量。

值得说明的是，尽管以上接口定义借鉴了 COM (Component Object Model) 技术的概念和做法，而且 WinDBG SDK 中也把这些接口称为 COM 接口，但事实上，这些接口并不是严格意义上的 COM 接口，它们也不是使用 COM 技术来实现的。首先，使用这些接口前不需要像使用 COM 接口前那样调用 CoInitialize 或 OleInitialize 来初始化 COM 库，另外，创建这些接口的方法也不是调用 CoCreateInstance 函数，而是调用调试器引擎自己公开的 DebugConnect 或 DebugCreate 方法。

比如，以下代码可以创建一个具有 IDebugClient 接口的对象，并得到它的指针：

```
IDebugClient *DebugClient;
HRESULT hr = DebugCreate(__uuidof(IDebugClient), (void **)&DebugClient);
```

有了 IDebugClient 接口后，便可以使用它来得到其他接口：

```
hr = DebugClient->QueryInterface(__uuidof(IDebugControl), (void **)&DebugControl);
```

如果是与工作在调试服务进程 (Server Process) 中的远程调试器引擎建立连接，那么可以使用 DebugConnect 方法：

```
IDebugClient *DebugClient;
HRESULT hr = DebugConnect(szConnect, __uuidof(IDebugClient), (void**)&g_Client);
```

参数 szConnect 用来指定连接参数，其内容为下面这样的字符串：

```
"-remote tcp:server=\\server_name, port=1225".
```

29.4.3 DebugClient 类

DebugClient 类有 500 多个方法，可以说是调试器引擎中最大的一个类。这个类封装了调试器引擎提供的几乎所有功能。上面说的 5 个公开接口所定义的方法实际上都是指向 DebugClient 方法的指针。以 IDebugAdvanced3 接口为例，它的方法表定义如下：

```
.text:02008574 ; const DebugClient::`vftable'{for `IDebugAdvanced3'}
.text:02008574 dd offset ?QueryInterface@DebugClient@@UAGJABU_GUID@@PAPAX@Z
.text:02008578 dd offset ?AddRef@DebugClient@@UAGKXZ
.text:0200857C dd offset ?Release@DebugClient@@UAGKXZ
.text:02008580 dd offset ?GetThreadContext@DebugClient@@UAGJPAXK@Z
.text:02008584 dd offset ?SetThreadContext@DebugClient@@UAGJPAXK@Z
.text:02008588 dd offset ?Request@DebugClient@@UAGJKPAXK0KPAK@Z
```

观察 dbgeng.h 文件中对 IDebugAdvanced3 接口的定义，其前 6 个方法依次分别是 QueryInterface、AddRef、Release、GetThreadContext、SetThreadContext 和 Request，看来它们都指向了 DebugClient 类中的同名方法。

事实上 DebugClient 类的开头便是一个数组，存放了 7 个公开接口类的方法表起始地址。当调用 DebugClient 的 QueryInterface 方法创建一个公开接口时，DebugClient 类便根据参数中的 UUID 返回对应接口的方法表地址。例如，以下是通过 DD 命令观察到的 DebugClient 对象的前 8 个 DWORD：

```
0:001> dd 007b4c80
```

```
007b4c80 02008574 020083f8 02008150 020080a0
007b4c90 02008028 02007e30 02007d70 02007d64
```

其中 02008574 便是 IDebugAdvanced3 接口的方法表，当创建 IDebugAdvanced3 接口时，用户得到的对象指针便是 007b4c80。类似的，020083f8 是 IDebugClient5 接口的方法表起始地址，02008150 对应的是 IDebugControl4 接口，020080a0 对应 IDebugDataSpaces4 接口，02008028 对应 IDebugRegisters2 接口，02007e30 对应 IDebugSymbols3 接口，02007d70 对应的是 IDebugSystemObjects4 接口，02007d64 对应的是一个未公开的 DbgRpcClientObject 接口。

WinDBG 使用全局变量 g_UiAdv、g_UiClient、g_UiControl 和 g_UiSymbols 来记录当前进程创建的 IDebugAdvanced、IDebugClient、IDebugControl 和 IDebugSymbols 接口对象，观察这些变量的值，可以看到它们的指针值就是上面的方法表起始地址。

```
0:001> dd WinDBG!g_UiAdv 11
01065e70 007b4c80
```

调试器通常在初始化阶段便创建 IDebugClient 接口的实例，比如 WinDBG 一运行便在 CreateUiInterfaces 方法中调用 DebugCreate 方法。DebugCreate 方法首先会调用全局函数 dbgeng!OneTimeInitialization 执行一次性的初始化工作，如果这个函数是第一次被调用，那么它会返回 0，根据这个返回值，DebugCreate 会创建第一个 DebugClient 类的实例。之后如果是同一个线程创建 7 个公开接口中的某一个，那么 DebugCreate 只是增加这个实例的引用计数（AddRef）。DbgEng 模块中的 g_NumRawClients 全局变量记录了已经创建的 DebugClient 类的实例个数。调用 DebugClient 的 CreateClient 方法可以创建新的 DebugClient 对象，并返回一个 IDebugClient 指针。全局指针 dbgeng!g_RawClients 用来记录已经创建的所有 DebugClient 对象。

29.4.4 中间层

中间层是实现调试器工作逻辑和各种调试功能的核心部分，也是整个调试器引擎中最复杂的一个部分。这个部分有几十个 C++类和一些全局函数组成。按照类的描述对象，可以把中间层的各个类分成如下几个部分。

调试目标：用来描述调试目标，从基类 TargetInfo 派生出一系列子类，分别代表不同的目标类型。我们将在下一节详细讨论。

CPU 架构：用来描述被调试程序所对应的 CPU 架构，其基类为 MachineInfo，派生类有：Mips64MachineInfo、MipsMachineInfo、ShMachineInfo、PpcMachineInfo、Mips32MachineInfo、Ia64MachineInfo、EbcMachineInfo、BaseX86MachineInfo、ArmMachineInfo、Amd64MachineInfo，分别对应于不同的 CPU 架构。以下类用于描述在 64 位系统中执行的 32 位代码：X86OnIa64MachineInfo、X86OnAmd64MachineInfo。

反汇编: 用于反汇编不同 CPU 架构的程序, 基类名为 DIS (Disassemble), 派生类有: DISARM、DISX86、DISPPC、DISSHCOMPACT、DISTHUMB、DISMIPS、DISIA64、DISCEE。

进程和线程: 类名分别为 ProcessInfo 和 ThreadInfo。

模块和映像: 前者用来描述各种模块, 基类为 ModuleInfo, 派生类有 NtKernel-ModuleInfo、NtKernelUnloadedModuleInfo、NtTargetUserModuleInfo、NtUserModuleInfo、CeDumpModuleInfo、IDNAModuleInfo、ImageFileModuleInfo, 后者用来描述可执行的程序映像, 有 ImageInfo 和 UnloadedImageInfo 两个类。

断点: 用来描述不同类型的断点, 包括代码断点 (CodeBreakpoint)、数据断点 (DataBreakpoint、Ia64DataBreakpoint), 这些类的基类为 Breakpoint。

表达式评估: 基类为 EvalExpression, 两个派生类为 MasmEvalExpression、CppEvalExpression, 分别用于 MASM (宏汇编) 和 C/C++ 表达式。

调试器引擎使用链表来管理某一个类的对象, 比如全局变量 g_Targets 用来存放调试目标, 它是一个单向链表结构 (DbsSingleList)。类似的, DbsDoubleList 是调试器引擎使用的双向链表结构。

29.4.5 服务层

服务层用来执行与调试目标密切相关的各种操作, 比如接收调试事件、读写内存和控制线程执行等。根据调试的场景不同, 调试器引擎设计了几个调试服务类, LiveUserDebugServices 类用在调试本地用户态进程, ProxyIUserDebugServicesN 用于调试远程的用户态程序, ProxyIUserDebugServicesN 用在远程内核调试的情况。这些类都派生自一个共同的基类 UserDebugServices。

通过 LiveUserTargetInfo 类的 SetServices 方法可以设置调试目标使用的 UserDebugServices 对象。

```
long __thiscall SetServices(struct IUserDebugServices *, int)
```

这样当进行本地调试时调试器可以将 LiveUserDebugServices 实例设置给 LiveUserTargetInfo 对象, 远程调试时可以将 ProxyIUserDebugServicesN 的实例设置给它。这样, 中间层便可以不关心调试目标是在本地还是远程。

29.4.6 传输和连接层

为了满足不同的数据传输方式, 调试器引擎定义了一系列 DbgKdTransport 类用在使用本机的端口 (COM、1394 或 USB2) 进行内核调试的情况, DbgRpcTransport 是个基类, 用于与另一台机器上的调试器通信进行远程调试, 根据远程调试所使用的

传输方式,它有多个子类:DbgRpcTcpTransport(TCP)、DbgRpcNamedPipeTransport(命名管道)、DbgRpc1394Transport(1394)、DbgRpcComTransport(串行通信)、DbgRpcSectionTransport、DbgRpcSecureChannelTransport。

传输层之下是连接层,负责具体的数据收发工作。例如,KdConnection类用于内核调试,它有三个派生类KdUsb2Connection、KdComConnection和Kd1394Connection与内核调试的三种连接方式一一对应。DbgRpcConnection用于远程调试。

本节简要介绍了调试器引擎的架构和它内部的主要类,因为微软的文档没有公开这些类的细节,所以以上介绍可能有不确切的地方。介绍这一内容的目的是为了帮助大家更好地理解WinDBG的工作原理和使用方法。

29.5 调试目标

简单来说,调试目标(Target)就是包含被调试代码的程序实体,有时也被称为被调试对象(Debuggee)。WinDBG是一个复合型的调试器,使用它可以调试多种类型的调试目标。首先根据调试目标所对应的运行模式,可以将其分为用户态目标和内核态目标两类。其次,根据调试目标是否处于运行状态,可以将其分为活动目标(Live Target)和转储文件(Dump File)两类。最后,根据调试目标与调试器的位置关系,可以把调试目标分为本地目标和远程目标。

WinDBG的调试器引擎使用TargetInfo类来描述调试目标的公共属性和行为,并从这个类派生出一系列子类来描述不同类型的目标,图29-15画出了这些类和它们之间的继承关系。

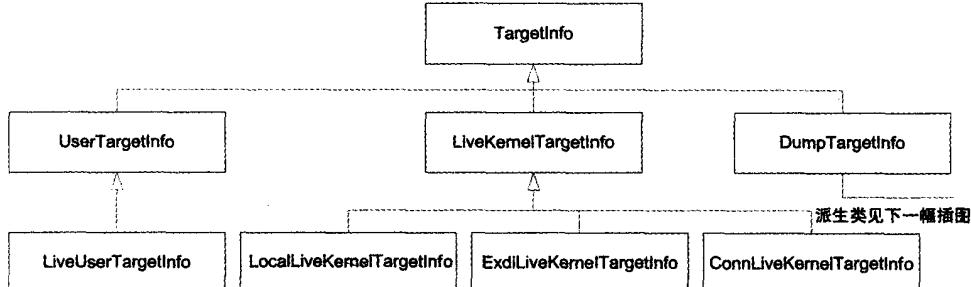


图 29-15 TargetInfo 类和它的派生类

TargetInfo类有3个子类,分别代表了用户态调试目标(UserTargetInfo)、内核态调试目标(LiveKernelTargetInfo)和转储文件目标(DumpTargetInfo)。下面我们先来看TargetInfo类,然后再分别介绍它的3个子类。

29.5.1 TargetInfo 类

TargetInfo 类是调试器引擎中描述调试目标的基类，它把调试不同类型调试目标的公共属性和操作提炼在这个统一的类中，这样做带来的好处就是接口层可以利用 C++ 的多态特征使用统一的指针类型和方法来处理不同类型的调试目标。举例来说，因为所有调试目标都派生自 TargetInfo 类，所以调试器就可以使用 TargetInfo 指针来记录所有调试目标，调试器引擎中的 g_Targets 全局变量就是用作这个目的，它是一个单向链表：

```
class DbsSingleList<class TargetInfo, 16> g_Targets;
```

全局变量 g_NumberTargets 记录了当前的调试目标个数：

```
unsigned long g_NumberTargets;
```

TargetInfo 类中定义了 200 多个方法，囊括了访问和操纵调试目标的所有操作，为了便于理解，我们将这些方法分为以下 14 类。

启动和终止调试会话：包括建立用户态调试会话的 StartAttachProcess 和 StartCreateProcess，复位调试会话的 DebuggeeReset，分离被调试进程的 DetachAllProcesses、CanDetachProcess 和 DetachProcess。

调试事件接收和处理：包括等待调试事件的 WaitForEvent，释放上一个调试事件的 ReleaseLastEvent，初始化用于接收调试事件的资源的 InitializeEventFirstWait，实现 WT 命令(详见 30.10.8)的 ProcessWatchTraceEvent，读取异常信息的 ReadExceptionRecord 和 GetTargetException，读取调试事件描述的 GetEventIndexDescription。

进程和线程控制：包括恢复和挂起线程的 ResumeThreads 和 SuspendThreads，终止进程的 TerminateAllProcesses 和 TerminateProcess，放弃被调试进程(.abandon 命令)的 AbandonProcess，请求中断调试目标的 RequestBreakIn，准备恢复调试目标的 PrepareForExecution。

读取进程和线程信息：包括查询和读取隐含进程信息的 GetImplicitProcessData、GetRawImplicitProcess、ReadImplicitProcessInfoPointer、GetImplicitProcessData-ParentCID 和 GetImplicitProcessDataPeb，读取进程属性的 GetProcessCookie、GetProcessInfoDataOffset、GetProcessInfoPeb、GetProcessTimes 和 GetProcessUpTimeN，查询和读取线程信息的 GetThreadBasicInfo、GetThreadIdByProcessor、GetThreadInfoDataOffset、GetThreadInfoTeb 和 GetThreadStackBounds。

断点管理：包括插入断点的 InsertCodeBreakpoint、InsertDataBreakpoint、InsertTargetCountBreakpoint 和 InsertTimeBreakpoint，删除断点的 RemoveAllData-Breakpoints、RemoveAllTargetBreakpoints、RemoveCodeBreakpoint、RemoveDataBreakpoint、RemoveTargetCountBreakpoint 和 RemoveTimeBreakpoint，判断断点命中的 IsDataBreakpointHit 和 IsTimeBreakpointHit，获取和释放同步对象的 BeginInserting-

Breakpoints、BeginRemovingBreakpoints、EndInsertingBreakpoints 和 EndRemovingBreakpoints。

访问调试目标: 包括读写 PCI 空间的 ReadBusData 和 WriteBusData, 读写 IO 空间的 ReadIo 和 WriteIo, 读写 MSR 寄存器的 ReadMsr 和 WriteMsr, 读写物理内存的 ReadPhysical、ReadPhysicalUncached、ReadAllPhysical、WritePhysical、WritePhysicalUncached, 读写虚拟内存的 ReadVirtual、ReadAllVirtual、ReadAndSwapAllVirtual、ReadVirtualUncached、WriteVirtual, WriteVirtualUncached、WriteAllVirtual、WriteAndSwapAllVirtual, 读写指针的 ReadPointer 和 WritePointer, 访问句柄数据的 ReadHandleData, 读取链表节点的 ReadListEntry, 读取字符串的 ReadMultiByteString、ReadMultiByteStringToUnicode、ReadUnicodeString、ReadUnicodeStringStruct 和 ReadUnicodeStringToMultiByte, 读写目标文件的 ReadTargetFile 和 WriteTargetFile。

读取和设置上下文: 包括读取上下文结构的 GetContext、GetTargetContext 和 GetContextFromThreadStack, 设置上下文结构的 SetContext 和 SetTargetContext, 以冲转 (flush) 上下文结构的 FlushRegContext 和 FlushRegContextToBase, 以及 InvalidateTargetContext 和 ChangeRegContext。

访问处理器 (CPU) 信息: 包括查询处理器功能的 GetGenericProcessorFeatures 和 GetSpecificProcessorFeatures, 读取处理器 ID 的 GetProcessorId, 读取电力信息的 GetProcessorPowerInfo 和 ReadPrcbProcessorPowerInfo, 读取描述处理器的系统数据的 GetProcessorSystemDataOffset, 读取特定寄存器的 GetTargetSpecialRegisters, 以及用于切换隐含处理器的 SwitchProcessors。

搜索进程和线程: 包括搜索进程的 FindProcessByDataOffset、FindProcessByHandle、FindProcessByPeb 和 FindProcessBySystemId, 搜索线程的 FindThreadByDataOffset、FindThreadByHandle 和 FindThreadBySystemId。

栈帧: 包括获取函数返回地址的 GetCallerAddress, 读取栈帧的 GetTargetStackFrames。

模块、映像和符号管理: 包括从映像文件中读取信息的 ReadImageNameString、ReadImageNtHeaders、ReadImageVersionInfo 和 GetImageVersionInformation, 取模块信息的 GetModuleInfo、GetUnloadedModuleInfo 和 GetUnloadedModuleMemoryInfo, 查询和清除系统符号的 QuerySystemSymbols 和 ClearSystemSymbols, 以及重新加载模块的 Reload。

版本和产品信息: 包括读取内核调试器引擎版本信息的 GetTargetKdVersion, 通过 dprintf 向命令窗口输出系统版本信息的 OutputVersion, 读取版本信息的 GetBuildAndPlatformFromWin32Version、GetProductInfo 和 ReadSharedUserProductInfo,

转换版本信息的 NtBuildToSystemVersion、Win9xBuildToSystemVersion 和 WinCeBuildToSystemVersion，读取时间信息的 GetCurrentSystemUpTimeN 和 GetCurrentTimeDateN。

内核调试特有的操作：包括复位目标系统的 Reboot，触发目标系统蓝屏的 Crash，读取进程信息的 KdGetProcessInfoDataOffset 和 KdGetProcessInfoPeb，读取线程信息的 KdGetThreadInfoDataOffset 和 KdGetThreadInfoTeb，读取内核调试数据块 ReadKdDataBlock，读写内核调试控制区的 ReadControl 和 WriteControl，读取页交换文件的 ReadPageFile，读页目录表地址的 ReadDirectoryTableBase。

访问蓝屏信息和转储文件：包括读取蓝屏停止代码和参数的 ReadBugCheckData，从转储文件读取自定义数据块的 ReadTagged。

需要说明的是，为了节约篇幅，我们并没有列出 TargetInfo 类的所有方法。另外，以上方法虽然都是定义在 TargetInfo 类中的，但是大多数方法都只是简单地返回一个错误代码 E_UNEXPECTED (0x8000FFFF)，真正的功能是留给派生类去实现的。

29.5.2 用户态目标

UserTargetInfo 类是描述用户态调试目标的基类，它派生自 TargetInfo 类。在目前的版本中，除了构造和析构函数外，UserTargetInfo 类只定义了两个方法，InitializeWatchTrace，ProcessWatchTraceEvent，它们都是用来实现 WT 命令的，我们将在 30.10 节对其做详细介绍。UserTargetInfo 类的派生类 LiveUserTargetInfo 才是真正用来描述用户态活动的目标的关键类。从调试符号可以看出，LiveUserTargetInfo 有 70 多个方法，其中绝大部分来自于 TargetInfo 类，也就是说，LiveUserTargetInfo 类是 TargetInfo 类针对户态活动目标的具体实现。

29.5.3 内核态目标

LiveKernelTargetInfo 类是描述内核态活动目标的基类，它也是派生自 TargetInfo 类。在目前版本的调试器引擎中，LiveKernelTargetInfo 类有三个派生类，分别如下。

- ConnLiveKernelTargetInfo：用来描述双机内核调试中的调试目标。
- LocalLiveKernelTargetInfo：用来描述本地内核调试中的调试目标。
- ExdiLiveKernelTargetInfo：用来描述通过 eXDI 驱动程序和硬件调试工具进行调试时的调试目标。

除了从基类继承的方法外，LiveKernelTargetInfo 类还定义了 InitFromKdVersion 方法，用于初始化内核调试会话。

ConnLiveKernelTargetInfo 是实现双机内核调试逻辑的一个主要类，包括读写

内存，等待内核调试事件，处理断点，控制目标系统（Crash、Reboot）等。

29.5.4 转储文件目标

转储文件（Dump File）是将系统（内核转储）或者进程（用户态转储）的状态永久凝固在文件中，好像是给软件拍摄的照片（Snapshot）。分析转储文件的最好工具就是调试器。为了复用各种调试功能，调试器把转储文件看作是普通调试目标的一个特例。当我们通过调试器分析转储文件时，就好像把时光倒流回产生转储文件的那一刻，我们可以观察变量，显示栈回溯、分析堆等等。但因为转储文件只包含转储那一刻的状态和特征，所以调试转储文件时不能执行任何需要恢复目标运行的命令，包括单步执行、继续运行等，也就是不可以把调试目标切换到转储那一时刻之外的其他状态。

WinDBG 的调试器引擎使用 `DumpTargetInfo` 类来描述转储文件目标，它派生自 `TargetInfo` 类，与 `LiveKernelTargetInfo` 和 `UserTargetInfo` 类并列，一起代表了 3 种主要的调试目标。

根据转储文件所拍摄的对象和文件中包含信息的多少，转储文件有很多种，相应的，调试器引擎中从 `DumpTargetInfo` 类派生出了很多个子类来分别描述不同种类的转储文件目标，图 29-16 画出了这些类和它们之间的关系。

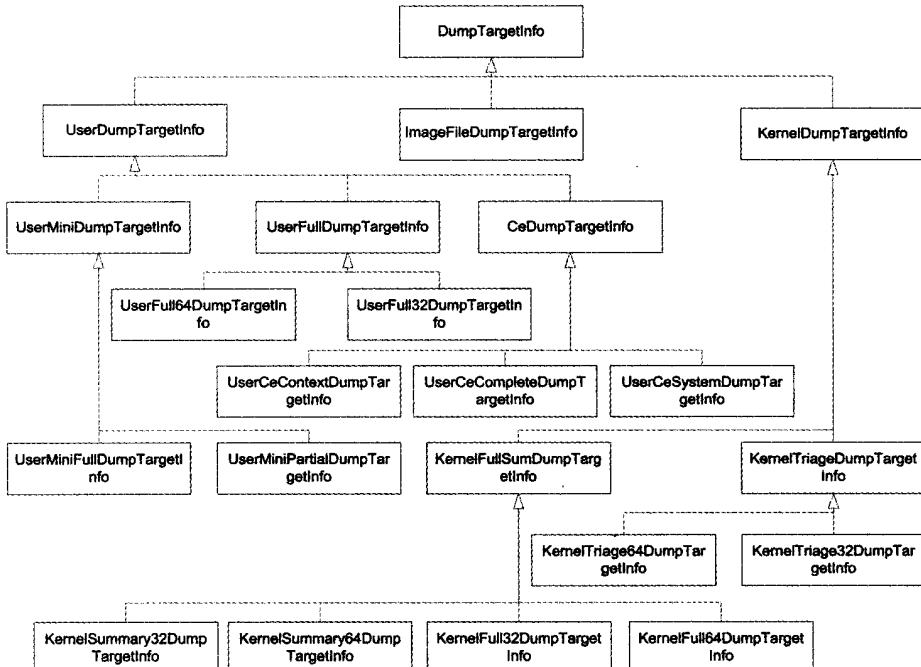


图 29-16 描述转储文件目标的各个类

本节我们简要介绍了 WinDBG 调试器引擎所支持的各种调试目标，以及描述它们

的各个 C++类，后面各节将结合具体的调试任务来介绍这些类的用法。

29.6 调试会话

调试器与被调试目标之间的通信对话通常被简称为调试会话（Debug Session）。调试会话的建立和结束标志着调试的真正开始和结束。

29.6.1 建立调试会话

清单 29-1 所示的函数调用序列显示了 WinDBG 调试器与一个已经运行的应用程序建立调试会话的执行过程。

清单 29-1 建立调试会话的执行过程

```

00 00e0fc94 0229a146 ntdll!NtDebugActiveProcess    // 调试活动进程的系统服务
01 00e0fcbb 0229a2b0 dbgeng!LiveUserDebugServices::CreateDebugActiveProcess...
02 00e0fcce 020f3a31 dbgeng!LiveUserDebugServices::AttachProcess+0xb0
03 00e0fcfc 020c1344 dbgeng!LiveUserTargetInfo::StartAttachProcess+0xd1
04 00e0fd40 0102a385 dbgeng!DebugClient::CreateProcessAndAttach2Wide+0x104
05 00e0ffa4 0102a9bb WinDBG!StartSession+0x445      // 开始调试会话
06 00e0ffb4 7c80b6a3 WinDBG!EngineLoop+0x1b        // 调试会话循环
07 00e0ffec 00000000 kernel32!BaseThreadStart+0x37

```

首先，以上过程发生在 WinDBG 的工作线程中，其线程号通常为 1 或者更大。因为 0 号线程是初始线程，通常用来处理用户交互（UI）。WinDBG 总是用一个专门的线程来启动和管理调试会话，我们把这个线程称为调试会话线程，有时也称为调试器工作线程。

从清单 29-1 可以看出，最下面是线程启动函数 `BaseThreadStart`，而后是调试会话线程的入口函数 `EngineLoop`。`EngineLoop` 函数是调试会话的主函数，负责初始化调试器引擎接口，启动调试会话和管理调试事件循环。

事实上，`EngineLoop` 函数会先调用 `InitializeEngineInterfaces` 来创建调试器引擎的主要接口，如 `IDebugClient`, `IDebugControl` 等，而后再调用 `StartSession` 开始创建调试会话。

`StartSession` 函数没有参数，它通过以下全局变量来判断要与什么样的调试目标建立调试关系。

- `g_NumDumpFiles` 和 `g_DumpFiles`: 通过这两个变量了解是否要调试转储文件目标。
- `g_DebugCommandLine`: 通过这个全局变量了解命令行中包含的参数。
- `g_PidToDebug` 和 `g_AttachProcessFlags`: 通过这两个变量了解是否附加到一个用户态进程和附加标志。

- `g_ProcNameToDebug`、`g_CreateProcessStartDir`、`g_DebugCreateOptions`: 通过这个变量了解是否创建并调试指定名称的进程（程序）。
- `g_AttachKernelFlags` 和 `g_KernelConnectOptions`: 通过这个变量了解是否要调试内核目标。

对于上面所示的栈回溯，我们是把 WinDBG 调试器附加到一个记事本进程，所以当我们观察 `g_PidToDebug` 变量时，可以看到它的值为 0x1430，这正是记事本进程的 ID。事实上，当我们通过 Attach Process 对话框选定记事本程序后，UI 线程将选定进程的进程 ID 保存到 `g_PidToDebug` 变量中，而后启动调试会话线程。

因为是要调试活动用户目标，所以接下来，`StartSession` 函数通过保存在 `g_DbClient` 变量中 `IDebugClient` 实例指针来调用它的 `CreateProcessAndAttach2Wide` 方法。

```
HRESULT IDebugClient5::CreateProcessAndAttach2Wide(
    IN ULONG64 Server,                                //用于远程调试
    IN OPTIONAL PWSTR CommandLine,                    //调试新创建进程时的命令行
    IN PVOID OptionsBuffer,                          //DEBUG_CREATE_PROCESS_OPTIONS 结构
    IN ULONG OptionsBufferSize,                     //OptionsBuffer 的结构大小
    IN OPTIONAL PCWSTR InitialDirectory,            //初始目录
    IN OPTIONAL PCWSTR Environment,                 //新进程的环境信息
    IN ULONG ProcessId,                            //附加到已经运行进程时的进程 ID
    IN ULONG AttachFlags,                          //附加标志
);
```

其中 `Server` 参数用于远程调试的情况，即通过一个进程服务器（Process Serer）来调试另一台机器上的程序，对于本地调试，这个参数为 0。`CommandLine`、`OptionsBuffer`、`OptionsBufferSize`、`Environment` 和 `InitialDirectory` 参数都是用于创建并调试一个新进程的情况，分别用来指定新进程的命令行、创建选项、环境变量和初始目录。`ProcessId` 参数用来指定要调试的已经存在的进程的 ID，对于我们的例子，这个参数的值为 0x1430。`AttachFlags` 参数用于指定附加选项，其值为 `DEBUG_ATTACH_XXX` 标志所定义的常量组合（详见 WinDBG 帮助文件）。

`CreateProcessAndAttach2Wide` 函数在对参数做基本检查后，会调用 `DebugClient` 类的 `UserInitialize` 函数来初始化用户态调试所需的基本设施，最主要的是会创建一个 `LiveUserTargetInfo` 实例和一个 `LiveUserDebugServices` 实例，并通过 `LiveUserTargetInfo` 类的 `SetServices` 方法将二者关联起来。`UserInitialize` 函数成功返回后，`CreateProcessAndAttach2Wide` 会调用新创建的 `LiveUserTargetInfo` 实例的 `StartAttachProcess` 方法并将进程 ID 作为参数传递给它。接下来 `LiveUserTargetInfo` 实例调用自己的调试服务对象（`LiveUserDebugServices`）的 `AttachProcess` 方法，后者再调用自己的 `CreateDebugActiveProcess` 方法，最后再调用操作系统的调试服务 `NtDebugActiveProcess`。

如果 StartAttachProcess 方法成功返回，那么说明调试会话已经成功建立，CreateProcessAndAttach2Wide 函数会将刚才创建的 LiveUserTargetInfo 类实例保存到全局变量 g_Target 中，以便以后可以方便地引用它。

29.6.2 调试循环

在第 9 章介绍用户态调试模型时，我们介绍过调试器的基本循环，即等待调试事件，处理调试事件和继续调试事件，然后再等待调试事件的循环过程。WinDBG 的 EngineLoop 函数既是它的调试会话线程的入口，同时也是实现基本调试循环的地方。图 29-17 画出了 EngineLoop 函数的基本流程图，其中间部分就是调试循环。

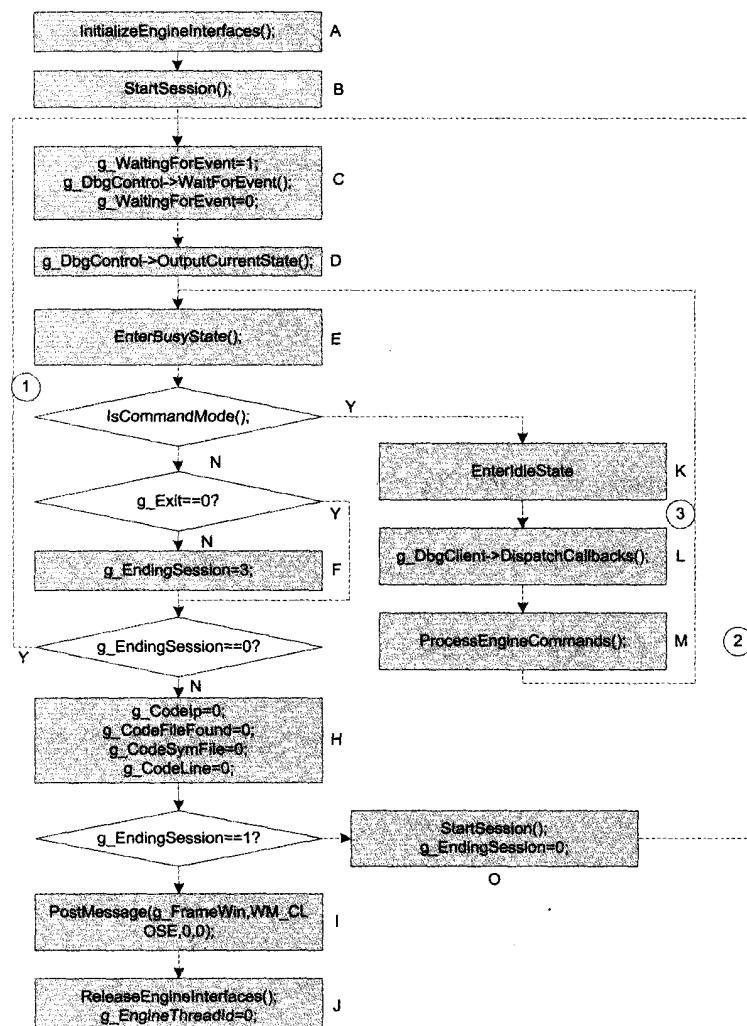


图 29-17 调试会话循环 (EngineLoop) 流程图

图 29-17 中共有三个循环，我们分别用数字 1~3 标出。1 号和 2 号循环的起点都是矩形 C 所标示的步骤，即先将全局变量 `g_WaitingForEvent` 设置为 1，然后调用全局变量 `g_DbControl` 的 `WaitForEvent()` 方法来等待调试事件。尽管这个函数的名字叫等待事件（Wait For Event），但事实上它发起的动作既包含等待调试事件，还有对调试事件的处理，我们稍后再讨论其细节。`g_DbControl->WaitForEvent()` 返回后，`g_WaitingForEvent` 被设置为 0，`EngineLoop` 函数调用 `g_DbControl` 的 `OutputCurrentState()` 方法输出状态信息。而后的菱形框用来判断是否需要让调试器进入命令模式。如果 `IsCommandMode` 方法返回 1，那么调试器便进入 3 号循环，先调用 `EnterIdleState` 切换到空闲状态，然后调用 `g_DbClient` 的 `DispatchCallbacks()` 方法处理注册的回调对象，其中就包括调试器 UI 注册的用户输入回调，让用户输入命令。`ProcessEngineCommands` 用来执行处理各种命令，我们下一节再详细介绍，本节将集中介绍 1 号和 2 号循环。

29.6.3 等待和处理调试事件

在成功调用 `StartSession` 之后，`EngineLoop` 函数调用 `DebugClient` 的 `WaitForEvent` 方法开始进入调试事件循环。而后，`DebugClient` 的 `WaitForEvent` 调用调试器引擎的一个全局函数 `RawWaitForEvent` 函数。

```
long RawWaitForEvent(class DebugClient *, unsigned long, unsigned long);
```

`RawWaitForEvent` 是一个比较复杂的函数，其内部包含的处理主要有。

1. 调用 `PrepareForWait` 函数。
2. 调用 `WaitInitialize` 方法。
3. 判断全局变量 `g_CmdState`，如果不等 `0x102`，则调用 `ProcessDeferredWork` 函数。
4. 如果 `g_EventTarget` 不为空，则调用它的 `ReleaseLastEvent` 方法，然后调用 `DiscardLastEvent` 函数将其抛弃。
5. 调用 `WaitForAnyTarget` 函数。
6. 成功等待调试事件后将对应的 `TargetInfo` 指针赋给 `g_EventTarget` 变量。
7. 调用 `g_EventTarget` 的 `ProcessDebugEvent` 方法处理调试事件。
8. 调用 `g_EventTarget` 的 `ReleaseLastEvent` 方法释放调试事件。
9. 将全局变量 `dbgeng!g_EventTarget` 设置为空，然后跳回到第 3 步继续循环。

下面我们从 `WaitForAnyTarget` 函数说起，因为 `g_Targets` 全局变量记录了目前的所有调试目标，所以 `WaitForAnyTarget` 函数的做法是从 `g_Targets` 所标识的链表中依次取出每个调试目标对象，然后调用它的 `WaitForEvent` 方法。

LiveUserTargetInfo 类的 WaitForEvent 方法会调用它的调试服务对象的 WaitForEvent 方法，后者调用 WaitForDebugEvent API 真正向调试子系统查询调试事件，其函数调用序列如清单 29-2 所示。

清单 29-2 等待调试事件

```

00 00dfffd08 0229c30a ntdll!ZwWaitForDebugEvent      //等待调试事件的系统服务
01 00dfffd00 021361f3 dbgeng!LiveUserDebugServices::WaitForEvent+0x10a
02 00dfffd10 020ceacf dbgeng!LiveUserTargetInfo::WaitForEvent+0x3b3
03 00dfffd34 020cee9e dbgeng!WaitForAnyTarget+0x5f    //轮番等待每个调试目标
04 00dfffd80 020cf110 dbgeng!RawWaitForEvent+0x2ae   //等待调试
05 00dfffd98 0102aadf dbgeng!DebugClient::WaitForEvent+0xb0
06 00dfffb4 7c80b6a3 WinDBG!EngineLoop+0x13f        //调试会话循环
07 00dfffec 00000000 kernel132!BaseThreadStart+0x37

```

调试用户态程序时，调试器收到的第一个调试事件通常是进程创建事件，即 CREATE_PROCESS_DEBUG_EVENT。LiveUserTargetInfo 的 WaitForEvent 收到调试事件后，会将这个调试事件所对应的进程 ID 和线程 ID 分别保存在 g_EventProcessSysId 和 g_EventThreadSysId 中，并将当前的 TargetInfo 对象指针赋给全局变量 dbgeng!g_EventTarget，而后调用本类中的 ProcessDebugEvent 方法。

ProcessDebugEvent 方法中包含一个 switch...case 结构，针对不同的事件做不同的处理，对于进程/线程创建和退出以及模块加载和卸载，ProcessDebugEvent 方法会把处理权交给对应的 NotifyXXXEvent 函数，即如下六个函数：

dbgeng!NotifyCreateProcessEvent	//处理进程创建事件
dbgeng!NotifyExitProcessEvent	//处理进程退出事件
dbgeng!NotifyCreateThreadEvent	//处理线程创建事件
dbgeng!NotifyExitThreadEvent	//处理线程退出事件
dbgeng!NotifyLoadModuleEvent	//处理模块加载事件
dbgeng!NotifyUnloadModuleEvent	//处理模块卸载事件

NotifyXXXEvent 函数的操作主要有如下两种。

- 信息更新，比如对于创建进程事件，会创建一个 ProcessInfo 对象，记录下这个进程的信息。类似的，对于线程创建和模块加载事件，会创建 ThreadInfo 和 ModuleInfo 对象。
- 调用注册的 IDebugEventCallbacks 回调对象。

在处理进程创建事件之前，LiveUserTargetInfo 的 WaitForEvent 会调用 NotifyDebuggeeActivation 函数来做一些初始化工作，包括：

1. 调用 dbgeng!TargetInfo::AddSpecificExtensions，加载与调试会话类型相匹配的扩展命令模块。
2. 调用 dbgeng!NotifySessionStatus，通知调试会话状态变化。
3. 调用 dbgeng!NotifyChangeDebuggeeState，通知被调试程序状态变化。

`WaitForEvent` 函数会保证对于一个调试会话，只调用 `NotifyDebuggeeActivation` 一次。

`DebugClient` 类的 `WaitForEvent` 方法返回后，`EngineLoop` 函数会通过 `g_DebugControl` 指针调用 `DebugClient` 类的 `OutputCurrentState()` 方法来更新当前状态。`OutputCurrentState()` 方法采取的动作主要如下。

1. 调用 `DebugClient` 类的 `PushOutCtl()` 方法。
2. 调用 `DebugClient` 类的 `OutCurInfo()` 方法和 `PopOutCtl()` 函数输出机器信息（寄存器）。
3. 调用 `FlushCallbacks()` 函数。

对于某些调试事件，调试器会进入到所谓的命令模式，让用户可以输入命令来观察和分析调试目标，即所谓的交互式调试（Interactive Debugging），我们将在下一节讨论。

29.6.4 继续调试事件

在 `LiveUserTargetInfo` 类的 `ProcessDebugEvent` 方法处理完一个调试事件返回到 `WaitForEvent` 方法后，`WaitForEvent` 方法也很快返回到 `WaitForAnyTarget` 函数，后者又返回到 `RawWaitForEvent` 函数。接下来 `RawWaitForEvent` 函数准备恢复调试目标继续执行，它会依次采取如下动作。

首先调用 `PrepareForExecution`，这个函数会调用另一个全局函数 `InsertBreakpoints` 来落实断点，对于软件断点，需要将 INT 3 指令插入到断点位置，即：

```
0194f58c 020b14e4 dbgeng!BaseX86MachineInfo::InsertBreakpointInstruction
0194f5dc 020aaaee dbgeng!LiveUserTargetInfo::InsertCodeBreakpoint+0x64
0194f610 020acd6e dbgeng!CodeBreakpoint::Insert+0xae //代码断点的插入方法
0194fea4 02132056 dbgeng!InsertBreakpoints+0x5be //插入断点
0194ff28 02130fe8 dbgeng!PrepareForExecution+0x5d6 //为恢复执行做准备
```

`PrepareForExecution` 函数在退出前会将全局变量 `dbgeng!g_Process`、`dbgeng!g_Thread`、`dbgeng!g_EventProcess`、`dbgeng!g_EventThread` 和 `dbgeng!g_EventMachine` 都赋值为空。

在 `PrepareForExecution` 函数返回后，`RawWaitForEvent` 函数会调用 `EventStatusToContinue` 函数来决定恢复执行的状态参数，也就是调用 `ContinueDebugEvent` API 所需的 `dwContinueStatus` 参数。

在这些动作完成后，`RawWaitForEvent` 函数调用由全局变量 `g_EventTarget` 所标识的调试目标对象的 `ReleaseLastEvent` 方法。`ReleaseLastEvent` 方法再调用它的调试服务对象的 `ContinueEvent` 方法。最后 `ContinueEvent` 方法调用系统的 `ContinueDebugEvent` API 真正恢复调试目标继续执行，其函数调用过程如清单 29-3。

清单 29-3 让调试目标继续执行

```

00 00ffff08 0229c45d ntdll!NtDebugContinue           //继续调试事件的系统服务
01 00ffff2c 02135ded dbgeng!LiveUserDebugServices::ContinueEvent+0x6d
02 00ffff44 020cee02 dbgeng!LiveUserTargetInfo::ReleaseLastEvent+0x3d
03 00ffff80 020cf110 dbgeng!RawWaitForEvent+0x212    //等待调试事件
04 00ffff98 0102aadf dbgeng!DebugClient::WaitForEvent+0xb0
05 00ffffb4 7c80b6a3 WinDBG!EngineLoop+0x13f        //调试会话循环
06 00ffffec 00000000 kernel32!BaseThreadStart+0x37

```

在 ReleaseLastEvent 方法返回后, RawWaitForEvent 函数会调用另一个全局函数 dbgeng!DiscardLastEvent。这个函数会执行下操作。

- 调用 dbgeng!DiscardLastEventInfo。
- 将全局变量 dbgeng!g_EventProcessSysId、dbgeng!g_EventThreadSysId、
dbgeng!g_TargetEventPc、dbgeng!g_TargetEventPc 都赋值为 0。
- 调用 dbgeng!InvalidateAllMemoryCaches 函数, 将缓存的内存数据设置为无效。为了提高速度, WinDBG 会把从调试目标读到数据做缓存, 这样如果连续执行同一个命令, 那么就不需要真正从调试目标读两次。因为一旦恢复程序继续执行, 那么这些缓存数据便过时了, 所以在恢复执行前应该把它们设置为无效状态。

接下来, RawWaitForEvent 函数将全局变量 dbgeng!g_EventTarget 设置为空。这个全局变量 g_EventTarget 用来标识当前调试事件所来自的调试目标, 对于只有一个调试目标的情况, 它要么为空(不在处理器调试事件的状态), 要么等于 g_Target。

而后, RawWaitForEvent 函数会将标志调试器引擎状态的 g_EngStatus 变量的 0x2000 位清除; 然后判断 g_EngStatus 的 0x4000 位是否被设置, 如果没有, 那么则跳回到上面调用 WaitForAnyTarget 开始新一轮等待, 如果这个位被设置, 那么则退出循环。

29.6.5 结束调试会话

全局变量 g_EndingSession 和 g_Exit 一起来控制调试循环和调试工作线程的结束。g_EndingSession 可以有以下四种值。

0: 代表不需要终止调试会话。

1: 代表重新开始调试会话的情况, 即.restart 命令或在 Debug 菜单中选择 Restart。调试循环看到这个值后会重新调用 StartSession, 将 g_EndingSession 复位为 0, 然后返回到步骤 C 等待调试事件, 参见图 29-8 中的步骤 O 和 3 号循环。

2: 当在调试器选择分离调试目标、停止调试, 或者当调试器接收到重新开始调试会话 (.restart) 命令后, 如果调用 StartSession 失败, 那么 g_EndingSession 会被设置为 2。

3: 当 g_Exit 被设置为非 0 时, g_EndingSession 会被设置为 3, 参见图 29-8 中的步骤 F。

g_Exit 可以有两种值, 0 或者 1。0 是默认值, 代表让调试循环正常运行; 1 代表要退出调试器。例如, 当我们直接关闭 WinDBG 程序 (Alt+F4), 那么 WinDBG 的 TerminateApplication 函数会将 g_Exit 赋值为 1, 通知调试会话线程退出。

图 29-17 中的步骤 I 用于关闭 WinDBG 的框架窗口, 步骤 J 用于释放 InitializeEngineInterfaces() 函数 (步骤 A) 所创建的调试器引擎接口。

调试器的应用层可以调用 DebugClient 类的 EndSession 方法来结束调试会话。比如当我们从 Debug 菜单选择 Detach Debuggee 时, WinDBG 的 ProcessCommand 函数会将全局变量 g_EndingSession 设置为 2, g_ExecStatus 设置为 7。而后 ProcessCommand 函数调用 EndSession 方法, 后者再调用 SeparateAllProcesses 函数, 其执行过程如清单 29-4 所示。

清单 29-4 分离调试目标的过程

```

00 017ded94 0229a390 ntdll!NtRemoveProcessDebug           //系统调用
01 017dedb8 020f3fb2 dbgeng!LiveUserDebugServices::DetachProcess+0x90
02 017dedd4 020f45e6 dbgeng!LiveUserTargetInfo::DetachProcess+0xf2
03 017deeb4 02195cd9 dbgeng!LiveUserTargetInfo::DetachAllProcesses+0xf6
04 017deed4 020c254b dbgeng!SeparateAllProcesses+0xe9      //分离所有被调试进程
05 017deeeec 0102861d dbgeng!DebugClient::EndSession+0x12b //结束调试会话
06 017def8c 01028a43 WinDBG!ProcessCommand+0x20d
07 017dfffa0 0102ad06 WinDBG!ProcessEngineCommands+0xa3
08 017dfffb4 7c80b6a3 WinDBG!EngineLoop+0x366             //调试会话循环
09 017dfffec 00000000 kernel32!BaseThreadStart+0x37

```

分离目标进程后, EndSession 方法再调用 DiscardSession 方法, 后者又调用 DiscardTargets。DiscardTargets 采取的动作主要有: (1) 调用 dbgeng!RemoveAllBreakpoints 函数清除所有断点。(2) 将 g_EngStatus 在内的很多全局变量复位为 0。3) 调用 dbgeng!DiscardedTargets 函数, 这个函数又调用 dbgeng!NotifyChangeEngineState 通知调试器引擎状态变化, dbgeng!NotifySessionStatus 通知会话状态变化, 调用 dbgeng!ExtensionInfo::NotifyAll 通知扩展命令模块。

在 EngineLoop 函数退出前, 它会将 g_EndingSession 设置为 0, 这个函数的返回后, 调试会话线程也就自然退出了。

29.7 接收和处理命令

WinDBG 是一个典型的交互式调试器, 允许用户将调试目标中断到调试器, 然后输入各种命令观察和分析调试目标的数据和代码, 待分析结束后可以恢复其继续执行。本节将介绍 WinDBG 接收和处理用户命令的方法。

29.7.1 调试器的两种工作状态

WinDBG 将调试会话的工作状态分为两种，一种叫空闲状态（Idle State），另一种叫繁忙状态（Busy State）。前者是指调试目标被中断到调试器中接受用户分析时的状态，后者是指调试目标恢复运行时的状态。只有在空闲状态时，调试器才允许用户输入各种诊断和观察命令，这时调试目标被中断到调试器中，是处于冻结状态的。与此相反，当调试会话处于繁忙状态时，大多数命令都是不允许的，但中断调试目标的命令（Break）除外，以便可以将调试目标中断并过渡到空闲状态。因为大多数调试命令只有在空闲状态时才能执行，所以空闲状态有时也被称为命令状态（Command State）。

29.7.2 进入命令状态

观察图 29-17 所示的调试会话循环，对于每个调试事件，DebugClient 类的 WaitForEvent 方法返回后，EngineLoop 函数会调用 DebugClient 类的 OutputCurrentState() 方法来更新调试目标的状态，包括数据观察窗口中的数据。然后 EngineLoop 调用 EnterBusyMode 将命令调试符显示为 *BUSY* 字样。再之后 EngineLoop 调用 IsCommandMode 函数来判断是否需要进入空闲模式。IsCommandMode 函数的伪代码如清单 29-5 所示。

清单 29-5 IsCommandMode 函数的伪代码

```
int IsCommandMode()
{
    if(g_Exit!=0 || g_EndingSession!=0)
        return 0;
    if(g_RemoteClient!=0 || g_ExecStatus==DEBUG_STATUS_BREAK)
        return 1;

    return 0;
}
```

其中 g_ExecStatus 是 WinDBG 定义的一个全局变量，其值是表 29-4 列出的 DEBUG_STATUS_XXX 系列常量。这些常量有以下两种用途。

- 供调试器向调试器引擎发出指示。比如，当调试器处理好一个调试事件后，它可以用这些常量来指示恢复调试目标的方式，也就是继续（Continue）调试事件的参数。继续调试事件的细节我们稍后讨论。
- 供调试器引擎报告调试目标或者调试会话的状态，比如调试事件等待函数的返回值和 IDebugControl 接口的 GetExecutionStatus 方法的返回值中都使用了这些常量。

在 WinDBG 的文档中，以上两种用途分别被简称为指示用途和报告用途，表 29-4 的第 3、4 两列分别描述了 DEBUG_STATUS_XXX 系列常量在这两种用途时的含义。

表 29-4 DEBUG_STATUS_XXX 系列常量

常量	取值	指示时的含义	报告时的含义
DEBUG_STATUS_NO_CHANGE	0	N/A	会话状态没有变化
DEBUG_STATUS_GO	1	让目标恢复正常执行	目标正常执行
DEBUG_STATUS_GO_HANDLED	2	让目标执行, 告诉系统事件(异常)已经处理	N/A
DEBUG_STATUS_GO_NOT_HANDLED	3	让目标执行, 告诉系统事件(异常)没有处理	N/A
DEBUG_STATUS_STEP_OVER	4	让目标单步执行, 跨越函数调用	目标在以 Step Over 方式单步执行
DEBUG_STATUS_STEP_INTO	5	让目标单步执行, 进入函数调用	目标在以 Step Into 方式单步执行
DEBUG_STATUS_BREAK	6	挂起调试目标	目标被中断
DEBUG_STATUS_NO_DEBUGGEE	7	N/A	没有活动的调试会话
DEBUG_STATUS_STEP_BRANCH	8	让目标继续执行, 直到遇到下一个分支指令	目标在以单步执行到下一分支方式单步执行
DEBUG_STATUS_IGNORE_EVENT	9	忽略上一调试事件, 让目标继续执行	N/A
DEBUG_STATUS_RESTART_REQUESTED	10	重新启动调试目标	目标在重新启动

在调试器引擎模块中, 全局变量 `g_EngStatus` 用来记录调试器引擎的状态, `g_CmdState` 用来记录命令状态。根据笔者观察, 当 WinDBG 处于空闲状态(命令状态)时, `g_CmdState` 的值为 0x101, 当调试会话处于繁忙状态时, `g_CmdState` 的值为 1。

下面我们继续讨论进入到命令状态的过程。当 `IsCommandMode` 函数返回 1 后, `EngineLoop` 函数便调用 `EnterIdleState` 进入命令状态, 这个函数执行的主要操作有。

1. 通过 `g_DbControl` 指针调用 `DebugClient` 类的 `GetPromptTextWide` 方法, 目的是取得命令提示文字, 如"0:001>"。
2. 调用 `UpdateBufferWindows` 函数更新缓冲区窗口, 如变量观察窗口和寄存器窗口等。
3. 向由全局变量 `g_FrameWin` 所标识的调试器窗口发送 0x405 号消息, 启动调试器的用户交互逻辑。

接下来, `EngineLoop` 函数通过 `g_DbControl` 指针调用 `DebugClient` 类的 `DispatchCallbacks` 方法, 其原型如下:

```
HRESULT IDebugClient::DispatchCallbacks( IN ULONG Timeout);
```

`DispatchCallbacks` 函数内部会调用操作系统的同步对象等待函数(`WaitForSingleObjectEx`)来等待一个信号量, 使调试会话线程处于等待状态。此时 UI 线程是在工作的, 因此用户可以通过命令窗口输入命令。当用户输入一条命令并

回车后，UI 线程会调用 DebugClient 类的 ExitDispatch 方法，释放信号量，使 DispatchCallbacks 中的等待函数返回。清单 29-6 显示了 UI 线程中结束一条命令输入后调用 ExitDispatch 函数的过程。

清单 29-6 用户输入一条命令之后的 UI 线程

```

0:000> kn 30
# ChildEBP RetAddr
00 0006d15c 020c2898 kernel32!ReleaseSemaphore      //释放信号量
01 0006d1b4 0102af20 dbgeng!DebugClient::ExitDispatch+0xd8 //调试器引擎接口函数
02 0006d1c4 010279d5 WinDBG!UpdateEngine+0x30      //与调试器引擎交互的函数
03 0006d1cc 01027acf WinDBG!FinishCommand+0x15
04 0006d1f0 010147d4 WinDBG!AddStringCommand+0xef //添加字符串命令
05 0006d208 0103ca68 WinDBG!CmdExecuteCmd+0xa4 //执行命令
06 0006ddaa 7e418724 WinDBG!FrameWndProc+0x1338 //窗口函数
【省略关于消息分发的多个栈帧】
11 0006ffff 00000000 kernel32!BaseProcessStart+0x23 //进程的启动函数

```

值得注意的是，上面显示的是 UI 线程（0 号线程）中发生的事情。在用户通过热键、菜单或命令窗口输入命令后，UI 线程调用 CmdExecuteCmd 函数，然后通过 UpdateEngine 函数调用 ExitDispatch，最后调用 ReleaseSemaphore 释放信号，唤醒调试会话线程让其开始处理命令。

29.7.3 执行命令

当调试会话线程被唤醒后，它先从 DispatchCallbacks 函数返回到 EngineLoop 函数，而后 EngineLoop 调用 ProcessEngineCommands 来处理命令。清单 29-7 显示了调试会话线程执行 kn 命令的过程。

清单 29-7 调试会话线程执行 kn 命令的过程

```

0:001> kn
# ChildEBP RetAddr
00 00f1db80 021f8045 dbghelp!StackWalk64+0x18      //DbgHelp 库的栈回溯函数
01 00f1e7ec 021f8609 dbgeng!TargetInfo::GetTargetStackFrames+0x645 //调试目标
02 00f1e880 0218651f dbgeng!DoStackTrace+0x1c9 //栈回溯功能的入口
03 00f1e8fc 02187b80 dbgeng!WrapParseStackCmd+0x15f //解析栈回溯命令
04 00f1e9d8 021889a9 dbgeng!ProcessCommands+0xab0
05 00f1ea1c 020cbc9 dbgeng!ProcessCommandsAndCatch+0x49
06 00f1eeb4 020cc12a dbgeng!Execute+0x2b9 //ANSI 版本
07 00f1eee4 01028553 dbgeng!DebugClient::ExecuteWide+0x6a //宽字符的命令执行方法
08 00f1ef8c 01028a43 WinDBG!ProcessCommand+0x143 //分发命令，见下文
09 00f1ffa0 0102ad06 WinDBG!ProcessEngineCommands+0xa3 //命令处理的顶层函数
0a 00f1ffb4 7c80b6a3 WinDBG!EngineLoop+0x366 //调试会话循环
0b 00f1ffec 00000000 kernel32!BaseThreadStart+0x37 //调试会话线程

```

其中的 Execute 方法是调试器向调试器引擎提交命令的主要接口，WinDBG SDK 详细描述了这个方法，其函数原型如下：

```

IDebugControl::Execute(
    IN ULONG OutputControl, //输出命令结果的方式，其值为 DEBUG_OUTCTL_XXX 常量

```

```

IN PCSTR Command,           //命令字符串
IN ULONG Flags             //执行命令的标志选项
);

```

ProcessCommands 函数是分发命令的主要场所，它先把命令解析为一个个元素，然后再把不同的命令分发给负责该命令的工作函数。在 ProcessCommands 函数中有很多个 switch case 这样的结构，从 IDA 工具为其绘制的代码结构图（图 29-18）中也可以看出这一特征。

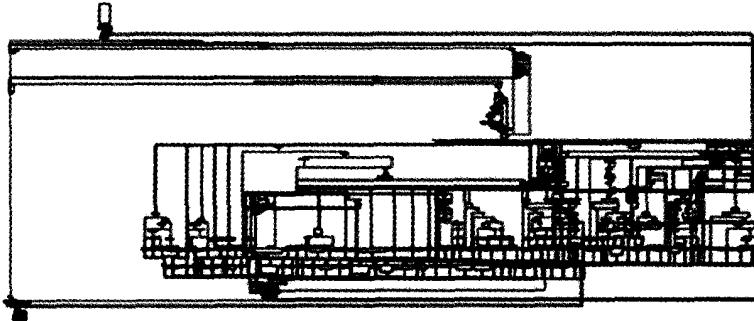


图 29-18 ProcessCommands 函数的代码结构

举例来说，ProcessCommands 函数将包括 kv 命令在内的 k 系列命令分发给 WrapParseStackCmd 函数，将 u 命令分发给 ParseUnassemble。类似的，ParseGoCmd 是处理 go 系列命令的入口，ParseRegCmd 是处理寄存器类命令的入口，ParseBpCmd 是断点命令的入口，ParseThreadCmds 是线程类命令（~等）的入口，ParseBangCmd 是所有扩展命令的入口，DotCommand 是所有元命令（Meta-Commands）的入口，ParseEnterCommand 是数据编辑类命令的入口，ParseStepTrace 是单步跟踪类命令的入口等等。

29.7.4 结束命令状态

结束调试会话或者恢复目标执行都会导致调试器脱离命令状态。上一节我们已经介绍过结束调试会话的情况，现在我们看恢复目标执行的情况。执行 g 命令的 SetExecGo 函数会调用 NotifyChangeEngineState 来通知调试器引擎状态将发生变化。这时调试目标还没有真正恢复执行。

当 ProcessEngineCommands 函数返回后，一次命令循环结束，EngineLoop 函数又跳回执行 EnterBusyMode，显示*BUSY*提示符，而后再调用 IsCommandMode 函数时，这时因为 g_ExecStatus 不再等于 DEBUG_STATUS_BREAK，而是 DEBUG_STATUS_GO (1)，因此 IsCommandMode 返回 0，于是整个调试会话循环沿着路线 1 回到步骤 C 去等待下一个调试事件。也就是调用 RawWaitForEvent 函数。RawWaitForEvent 会调用 PrepareForWait 做等待下一个调试事件的准备工作。之后

设置执行状态的 `SetExecutionStatus` 函数会被调用。准备函数返回后，`RawWaitForEvent` 会根据全局变量 `g_EventTarget` 来判断上一个调试事件是否为空，如果不为空会调用 `ReleaseLastEvent` 来释放。`ReleaseLastEvent` 会调用调试服务对象的 `ContinueEvent` 方法，后者会调用 `NtDebugContinue` 真正恢复目标执行，上一节的清单 29-3 显示了这一过程。`ReleaseLastEvent` 方法返回后，`RawWaitForEvent` 会调用 `WaitForAnyTarget`，后者再通过调试目标和调试服务对象调用操作系统的等待事件函数，也就是清单 29-2 所示的执行过程。

29.8 本章总结

本章比较详细地介绍了 WinDBG 调试器的概况、发展历史、模块组成、软件架构和主要功能的实现方法。为了不流于空泛，我们在介绍中较多地引用了 WinDBG 的内部类和函数。在将来的版本中，某些类和函数的名字可能会变化。不过，我们的目的主要是让大家更深入地理解调试器的设计原理，以便可以更好地使用它，因此将来变化并不会影响这个目标。

参考文献

1. How to set up remote debugging quickly by using Visual C++ 5.0 or Visual C++ 6.0.
Microsoft Corporation
2. Debugging Tools for Windows 帮助手册. Microsoft Corporation

WinDBG 用法详解

WinDBG 是个非常强大的调试器，它设计了极其丰富的功能来支持各种调试任务，包括用户态调试、内核态调试、转储文件调试、远程调试等等。而且，WinDBG 调试器具有非常好的灵活性和可扩展性，提供了丰富的选项允许用户定制现有的调试功能，包括改变调试事件的处理方式。如果现有的功能和选项不能满足要求，那么用户可以编写命令程序和扩展命令来定制和补充 WinDBG 的功能。

尽管 WinDBG 是个典型的窗口程序，但是它的大多数调试功能还是以手工输入命令的方式来工作的。目前版本的 WinDBG 共提供了 130 多条标准命令，140 多条元命令(Meta-commands)和难以计数的扩展命令，学习和灵活使用这些命令是学习 WinDBG 的关键，也是难点。

上一章我们从设计的角度分析了 WinDBG，本章将从使用角度进一步介绍 WinDBG。我们先介绍工作空间的概念和用法(30.1 节)，然后介绍命令的分类(30.2 节)、用户界面(30.3 节)以及如何输入和执行命令(30.4 节)。第 30.5 节介绍常用的调试模式和建立调试会话的方法。第 30.6 节介绍终止调试会话的各种方法。第 30.7 节介绍上下文的概念和在调试时如何切换和控制上下文。第 30.8 节介绍调试符号。第 30.9 节讨论如何定制调试事件的处理方式。从第 30.10 节到第 30.17 节将介绍如何在 WinDBG 中完成典型的调试操作，包括控制调试目标(30.10 节)、单步执行(30.11 节)、设置断点(30.12 节)、控制进程和线程(30.13 节)、观察栈(30.14 节)、观察和修改数据(30.15 节)、遍历链表(30.16 节)和调用函数(30.17 节)。第 30.18 节介绍编写命令程序的方法。

30.1 工作空间

WinDBG 使用工作空间(Workspace)来描述和存储调试项目的属性、参数以及调试器设置等信息，其功能相当于集成开发环境(IDE)中的项目文件。

WinDBG 定义了两种工作空间，一种称为默认的工作空间(Default Workspace)，

另一种称为命名的工作空间（Named Workspace）。当没有明确使用某个命名的工作空间时，WinDBG 总是使用默认的工作空间，因此默认的工作空间也叫隐含的（implicit）工作空间，命名的工作空间也叫显式的（explicit）工作空间。

WinDBG 安装时就预先创建了一系列默认的工作空间，分别是。

- 基础工作空间（Base Workspace），当调试会话尚未建立，WinDBG 处于赋闲（dormant）状态时，它会使用基础工作空间作为默认工作空间。
- 默认的内核态工作空间（Default Kernel-mode Workspace），当在 WinDBG 中开始内核调试，但是尚未与调试目标建立连接时，WinDBG 会使用这个工作空间作为默认工作空间。
- 默认的远程调试工作空间（Remote Default Workspace），当通过调试服务器（DbgSrv 或 KdSrv）进行远程调试时，WinDBG 会使用这个工作空间作为默认工作空间。
- 特定处理器的工作空间（Processor-specific Workspace），在进行内核调试时，当 WinDBG 与调试目标建立起联系，并知道对方的处理器类型后，WinDBG 会使用与目标系统中的处理器类型相配套的工作空间作为默认工作空间。典型的处理器类型有 x86、AMD64、Itanium 等。
- 默认的用户态工作空间（Default User-mode Workspace），当使用 WinDBG 调试一个已经运行的进程时，它会使用这个工作空间作为默认的工作空间。

此外，当在 WinDBG 中打开一个应用程序并开始调试时，WinDBG 会根据可执行文件的路径和文件名为其建立一个默认的工作空间，如果这个工作空间已经存在，那么它就使用已经存在的。类似的，当使用 WinDBG 分析转储文件（dump file）时，WinDBG 会根据转储文件的全路径建立和维护默认的工作空间。

在 WinDBG 的文件菜单中选择 Save workspace as... 命令，将当前工作空间另存为一个特定的名字，那么便创建了一个命名的工作空间。在 Save Workspace As 对话框的标题中包含了 WinDBG 当前所使用工作空间的名字。当 WinDBG 切换到一个新的工作空间或者退出时，如果工作空间的内容变化，那么 WinDBG 会提示是否要保存工作空间，提示对话框（图 30-1）的标题中也包含了工作空间的名字。

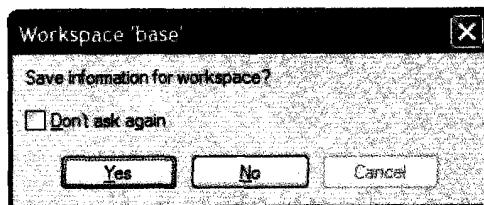


图 30-1 切换或者关闭调试会话时 WinDBG 提示是否要保存工作空间

WinDBG 的工作空间中保存了如下几类信息。

- 调试会话状态：包括断点，打开的源文件，用户定义的别名（alias）等。
- 调试器设置：包括符号文件路径，可执行映像文件路径，源文件路径，用 l+/-命令设置的源文件选项，日志文件设置，通过启动内核调试对话框设置内核调试连接设置，最近一次打开文件对话框所使用的路径和输出设置等。
- WinDBG 图形界面信息：包括 WinDBG 窗口的标题，是否自动打开反汇编窗口，默认字体，WinDBG 窗口在桌面的位置，打开的 WinDBG 子窗口，每个打开窗口的详细信息，包括位置、浮动状态等，命令窗口的设置，是否显示状态条和工具条，寄存器窗口的定制信息，源文件窗口的光标位置，变量观察窗口的变量信息等。

WinDBG 默认使用注册表来保存工作空间设置，其路径为：

HKEY_CURRENT_USER\Software\Microsoft\WinDBG\Workspaces

在这个键下通常有 4 个子键 User、Kernel、Dump 和 Explicit（参见图 30-2），前 3 个子键分别用来保存用户态调试、内核态调试、调试转储文件时使用的默认工作空间，Explicit 用来保存命名的工作空间。

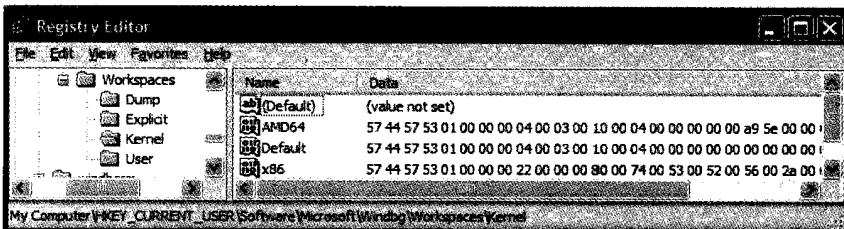


图 30-2 WinDBG 在注册表中保存的工作空间信息

在以上 4 个子键下的每个键值对应于一个工作空间，键值名是工作空间的名称，键值值就是这个工作空间的二进制数据。

WinDBG 支持使用文件来保存工作空间。使用 File 菜单的 Save Workspace to File 功能可以将当前的工作空间保存为一个.WEW 文件。这个文件是二进制的，其内容与注册表中的数据是一样的。

启动 WinDBG 时可以通过-W 开关指定要使用的工作空间名称，也可以通过 File 菜单来打开一个工作空间以显式地加载这个工作空间的设置。

值得说明的是，WinDBG 是以累积方式来应用工作空间中的大多数设置的。当 WinDBG 启动时，它会应用默认的基础设置，当加载新的工作空间时，WinDBG 只是加载这个新工作空间中的特别内容。

通过 WinDBG 的 File 菜单的 Delete Workspaces 命令可以删除工作空间。一种更快的方法就是使用注册表编辑器（regedit）直接删除保存在注册表中的键值。如果要删掉全部工作空间，那么就把 Workspaces 子键全部删除，这样做不会影响 WinDBG 正常运行，下次使用时它会自动创建默认工作空间所需的键值。

WinDBG 程序目录中的 Themes 子目录中包含了 4 种经过定制的工作空间设置，称为主题（Theme）。每个主题配备了一个.reg 文件和一个.WEW 文件，将.reg 文件导入到注册表或者使用 WinDBG 打开.WEW 文件（Open Workspace in File）便可以应用对应的主题。

30.2 命令概览

WinDBG 的大多数功能是以命令方式工作的，本节将介绍 WinDBG 的 3 类命令：标准命令、元命令和扩展命令。

30.2.1 标准命令

标准命令用来提供适用于所有调试目标的基本调试功能。所有基本命令都是实现 在 WinDBG 调试器内部的，执行这些命令时不需要加载任何扩展模块。大多数标准命令是一两个字符或者符号，只有 version 等少数命令除外。标准命令的第一个字符是不分大小写的，第二个字符可能区分大小写。迄今为止，WinDBG 调试器共实现了 130 多条标准命令，分为 60 多个系列。为了便于记忆，可以根据功能将标准命令归纳为如下 18 个子类。

- 控制调试目标执行，包括恢复运行的 g 系列命令、跟踪执行的 t 系列命令（trace into）、单步执行的 p 系列命令（step over）和追踪监视的 wt 命令。
- 观察和修改通用寄存器的 r 命令，读写 MSR 寄存器的 rdmsr 和 wrmsr，设置寄存器显示掩码的 rm 命令。
- 读写 IO 端口的 ib/iw/id 和 ob/ow/od 命令。
- 观察、编辑和搜索内存数据的 d 系列命令、e 系列命令和 s 命令。
- 观察栈的 k 系列命令。
- 设置和维护断点的 bp（软件断点）、ba（硬件断点）和管理断点的 bl（列出所有断点）、bc/bd/be（清除、禁止和重新启用断点）命令。
- 显示和控制线程的~命令。
- 显示进程的!命令。
- 评估表达式的?命令和评估 C++ 表达式的??命令。
- 用于汇编的 a 命令和用于反汇编的 u 命令。
- 显示段选择子的 dg 命令。
- 执行命令文件的\$命令。
- 设置调试事件处理方式的 sx 系列命令，启用与禁止静默模式的 sq 命令，设置内核选项的 so 命令，设置符号后缀的 ss 命令。

- 显示调试器和调试目标版本的 `version` 命令，显示调试目标所在系统信息的 `vertarget` 命令。
- 检查符号的 `x` 命令。
- 控制和显示源程序的 `ls` 系列命令。
- 加载调试符号的 `ld` 命令，搜索相邻符号的 `ln` 命令和显示模块列表的 `lm` 命令。
- 结束调试会话的 `q` 命令，包括用于远程调试的 `qq` 命令，结束调试会话并分离调试目标的 `qd` 命令。

在命令编辑框中输入一个问号 (?) 可以显示出主要的标准命令和每个命令的简单介绍。附录 B 按字母顺序列出了所有标准命令。

30.2.2 元命令

元命令 (Meta-Command) 用来提供标准命令没有提供的常用调试功能，与标准命令一样，元命令也是内建在调试器引擎或者 WinDBG 程序文件中的。所有元命令都以一个点 (.) 开始，所以元命令也被称为点命令 (Dot Command)。

按照功能，可以把元命令分成如下几类。

- 显示和设置调试会话和调试器选项，比如用于符号选项的 `.symopt`，用于符号路径的 `.sympath` 和 `.symfix`，用于源程序文件的 `.srcpath`、`.srcnoise` 和 `.srcfix`，用于扩展命令模块路径的 `.extpath`，用于匹配扩展命令的 `.extmatch`，用于可执行文件的 `.exepath`，设置反汇编选项的 `.asm`，控制表达式评估器的 `.expr` 命令。
- 控制调试会话或者调试目标，如重新开始调试会话的 `.restart`，放弃用户态调试目标 (进程) 的 `.abandon`，创建新进程的 `.create` 命令和附加到存在进程的 `.attach` 命令，打开转储文件的 `.opendump`，分离调试目标的 `.detach`，用于杀掉进程的 `.kill` 命令。
- 管理扩展命令模块，包括加载模块的 `.load` 命令，卸载用的 `.unload` 命令和 `.unloadall` 命令，显示已经加载模块的 `.chain` 命令等。
- 管理调试器日志文件，如 `.logfile` (显示信息)、`.logopen` (打开)、`.logappend` (追加) 和 `.logclose` (关闭)。
- 远程调试，如用于启动 `remote.exe` 服务的 `.remote` 命令，启动调试引擎服务器的 `.server` 命令，列出可用服务器的 `.servers` 命令，用于向远程服务器发送文件的 `.send_file`，用于结束远程进程服务器的 `.endpsrv`，用于结束引擎服务器的 `.endsrv` 命令。
- 控制调试器，如让调试器睡眠一段时间的 `.sleep` 命令，唤醒处于睡眠状态的调试器的 `.wake` 命令，启动另一个调试器来调试当前调试器的 `.dbgdbg` 命令。
- 编写命令程序，包括一系列类似 C 语言关键字的命令，如 `.if`、`.else`、`.elsif`、`.foreach`、`.do`、`.while`、`.continue`、`.catch`、`.break`、`.continue`、`.leave`、`.printf`、`.block` 等，我们将在本章的第 18 节介绍命令程序的编写方法。

- 显示或者转储调试目标数据，如产生转储文件的.dump 命令，将原始内存数据写到文件的.writemem 命令，显示调试会话时间的.time 命令，显示线程时间的.ttime 命令，显示任务列表的.tlist (task list) 命令，以不同格式显示数字的.formats 命令。输入.help 可以列出所有元命令和每个命令的简单说明。

30.2.3 扩展命令

扩展命令 (Extension Command) 用于实现针对特定调试目标的调试功能。与标准命令和元命令是内建在 WinDBG 程序文件中不同，扩展命令是实现在动态加载的扩展模块 (DLL) 中的。

利用 WinDBG 的 SDK，用户可以自己编写扩展模块和扩展命令。WinDBG 程序包中包含了常用的扩展命令模块，存放在以下几个子目录中。

- NT4CHK：调试目标为 Windows NT 4.0 Checked 版本时的扩展命令模块。
- NT4FRE：调试目标为 Windows NT 4.0 Free 版本时的扩展命令模块。
- W2KCHK：调试目标为 Windows 2000 Checked 版本时的扩展命令模块。
- W2KFRE：调试目标为 Windows 2000 Free 版本时的扩展命令模块。
- WINXP：调试目标为 Windows XP 或者更高版本时的扩展命令模块。
- WINEXT：适用于所有 Windows 版本的扩展命令模块。

表 30-1 列出了 WINEXT 和 WINXP 目录中的所有扩展命令模块，第一列是文件名称，第二列是路径，第三列是简单描述。

表 30-1 WinDBG 工具包中的扩展命令模块

扩展模块	路径	描述
ext.dll	WINEXT	适用于各种调试目标的常用扩展命令
kext.dll	WINEXT	内核态调试时的常用扩展命令
uext.dll	WINEXT	用户态调试时的常用扩展命令
logexts.dll	WINEXT	用于监视和记录 API 调用 (Windows API Logging Extensions)
sos.dll	WINEXT	用于调试托管代码和 .Net 程序
ks.dll	WINEXT	用于调试内核流 (Kernel Stream)
wdfkd.dll	WINEXT	调试使用 WDF (Windows Driver Foundation) 编写的驱动程序
acpkd.dll	WINXP	用于 ACPI 调试，追踪调用 ASL 程序的过程，显示 ACPI 对象
exts.dll	WINXP	关于堆 (!heap)、进程/线程结构 (!teb!/peb)、安全信息 (!token、!sid、!acl) 和应用程序验证 (!avrf) 等的扩展命令
kdexts.dll	WINXP	包含了大量用于内核调试的扩展命令
fltkd.dll	WINXP	用于调试文件系统的过滤驱动程序 (FsFilter)
minipkd.dll	WINXP	用于调试 AIC78xx 小端口 (miniport) 驱动程序
ndiskd.dll	WINXP	用于调试网络有关驱动程序
ntsdexts.dll	WINXP	实现了!handle、!locks、!dp、!dreg (显示注册表) 等命令
rpcexts.dll	WINXP	用于 RPC 调试
scsikd.dll	WINXP	用于调试 SCSI 有关的驱动程序

续表

扩展模块	路径	描述
traceprt.dll	WINXP	用于格式化 ETW 信息
vdmexts.dll	WINXP	调试运行在 VDM 中的 DOS 程序和 WOW 程序
wow64exts.dll	WINXP	调试运行在 64 位 Windows 系统中的 32 位程序
wmitrace.dll	WINXP	显示 WMI 追踪有关的数据结构、缓冲区和日志文件

执行扩展命令时，应该以叹号（!）开始，叹号在英文中被称为 bang，因此扩展命令也被称为 Bang Command。执行扩展命令的完整格式是：

`![扩展模块名].<扩展命令名> [参数]`

其中扩展模块名可以省略，如果省略，WinDBG 会自动在已经加载的扩展模块中搜索指定的命令。

因为扩展命令是实现在动态加载的扩展模块中的，所以执行时需要加载对应的扩展模块。当调试目标被激活（Debuggee Activation）时，WinDBG 会根据调试目标的类型和当前的工作空间自动加载命令空间中指定的扩展模块。用户也可以使用以下方法手动加载扩展模块。

- 使用.load 命令加上扩展模块的名称或者完整路径来加载它。如果没有指定路径，那么 WinDBG 会在扩展模块搜索路径（EXTPATH）中寻找这个文件。
- 使用.loadby 命令加上扩展模块的名称和一个已经加载的程序模块的名称。这时 WinDBG 会在指定的程序模块文件所在目录中寻找和加载扩展命令模块。例如，在调试托管程序时，可以使用.loadby sos mscorewks 命令让 WinDBG 在 mscorewks 模块所在的目录中加载 SOS 扩展模块，这样可以确保加载正确版本的 sos 模块。

当使用“!扩展模块名.扩展命令名”的方式执行扩展命令时，如果指定的扩展模块还没有加载，那么 WinDBG 会自动搜索和加载这个模块。

使用.chain 命令可以列出当前加载的所有扩展模块，使用.unload 和.unloadall 命令可以卸载指定的或者全部扩展模块。大多数扩展模块都支持 help 命令来显示这个模块的基本信息和所包含的命令，例如执行!ext.help 可以显示 ext 模块中的所有扩展命令。

30.3 用户界面

本节将先介绍 WinDBG 的窗口结构和各种子窗口的用途，然后重点介绍命令窗口和各种命令提示符的含义。

30.3.1 窗口概览

WinDBG 是个典型的窗口程序（图 30-3），最外层是框架窗口（Frame Window），框架窗口的用户区上边是菜单和工具条，下边是状态条，中部的用户区可以摆放各种

工作窗口。

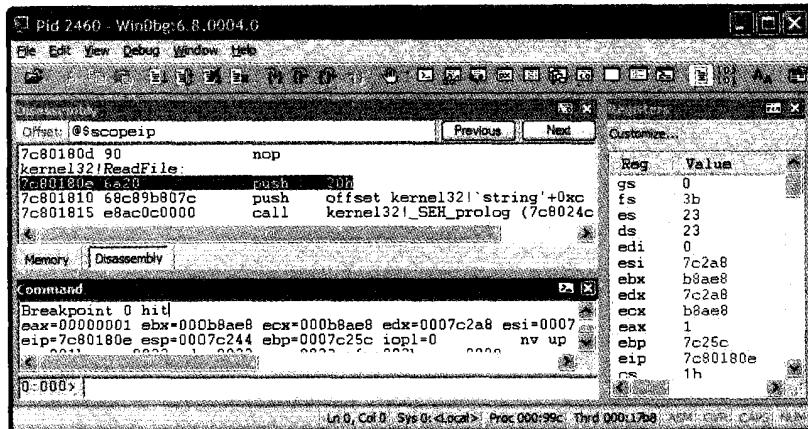


图 30-3 WinDBG 的基本用户界面

图 30-3 中打开了四个常用的工作窗口，分别是反汇编窗口、内存窗口、命令窗口和寄存器窗口，其中内存窗口与反汇编窗口共享一个窗口区域。通过 View 菜单或者热键还可以打开其他子窗口，表 30-2 列出了目前版本的 WinDBG 所支持的所有工作窗口。

表 30-2 WinDBG 的工作窗口

名称	热键	用途
Command	Alt+1	输入命令，显示命令结果和调试信息
Watch	Alt+2	观察指定的全局变量、局部变量和寄存器信息
Locals	Alt+3	自动显示当前函数的所有局部变量
Registers	Alt+4	观察和修改寄存器的值
Memory	Alt+5	观察和修改内存数据
Call Stack	Alt+6	显示函数调用序列
Disassembly	Alt+7	反汇编
Scratch Pad	Alt+8	白板，可以用来做调试笔记等
Processes and Threads	Alt+9	显示被调试的进程和线程
Command Browser	Alt+N	执行和浏览命令

使用 Spy++ 工具可以观察 WinDBG 的窗口结构，例如，图 30-4 所示的窗口树描述了图 30-3 中的所有窗口。

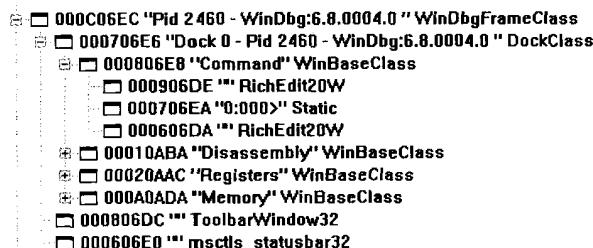


图 30-4 WinDBG 程序的窗口结构

从图中可以看到，最顶层是窗口类 WinDBGFrameClass 的一个实例，它有三个子窗口，分别是工具条、状态条和一个 DockClass 类型的子窗口 Dock0。Dock0 又有 5 个子窗口，即我们打开的 5 个工作窗口。

WinDBG 支持两种方式来摆放工作窗口，一种是浮动方式（floating），另一种是码放方式（dock）。对于浮动方式，WinDBG 的 Window 菜单提供了水平平铺、垂直平铺和层叠（Cascade）命令来自调整窗口位置。对于码放方式，所有工作窗口填充在图 30-4 中的 DockClass 窗口中，用户可以使用鼠标来调整子窗口的大小和位置。图 30-3 所示的情况使用的是码放方式。

可以把窗口布局保存到工作空间中，这样下次再打开这个工作空间时，WinDBG 会自动打开上次使用的子窗口并恢复到保存时的状态。

30.3.2 命令窗口和命令提示符

命令窗口是用户与 WinDBG 交互的主要接口。图 30-3 中左下方的子窗口就是命令窗口，它由上下两个部分组成，上面是信息显示区，下面是命令横条。命令横条又分为左右两个部分，左边是命令提示符，右边是命令编辑框。

信息显示区是 WinDBG 输出各种调试信息的主要场所，包括命令的执行结果、调试事件、错误信息和调试引擎的提示信息等。

WinDBG 的命令提示符由文字和大于号或星号两部分组成。对于不同类型的调试目标和调试会话状态，命令提示符的内容也会不同，下面将详细介绍。

首先，当 WinDBG 启动后尚未与任何调试目标建立调试对话处于待用状态时，它的提示符区域不显示任何内容，命令编辑框显示尚未与调试目标建立连接。我们将这种提示符称为空白提示符。当进行内核调试等待与调试目标建立连接时，WinDBG 显示的也是空白提示符（图 30-5）。

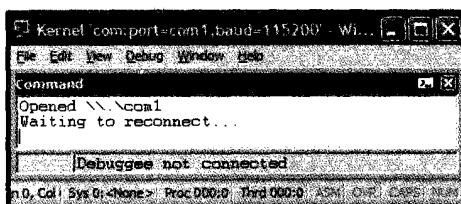


图 30-5 等待与调试目标建立连接时的空白提示符

第二，当 WinDBG 处于命令模式等待用户输入命令时，它的提示符是描述当前调试目标的简短文字加一个大于号，即图 30-6 中的样子。考虑到 WinDBG 支持同时调试多个系统中的多个调试目标，因此对于用户态目标，命令提示符的完整格式是：

```
[|||system_index:]process_index:thread_index>
```

对于双机内核调试时的内核态目标或者内核转储文件目标，命令提示符的完整形式是：

[||| system index:] [processor index:] kd>

对于本地内核态调试，命令提示符的完整形式是：

```
[|||system_index:] [processor_index:] lkd>
```

其中，`system_index` 代表系统序号，同一个 Windows 系统中的多个用户态目标属于一个系统，每个内核目标单独属于一个系统；`process_index` 代表进程序号，`processor_index` 代表处理器序号；`thread_index` 代表线程序号。所有序号都是从 0 开始全局编排的。当调试目标既有内核态目标，又有用户态目标时，处理器序号与线程序号同等编排，每个内核态目标也被分配一个进程序号。

下面通过一个实验来说明。启动 WinDBG 和一个计算器程序（Calc.exe），并将 WinDBG 附加到计算器进程，然后使用 .opendump 命令打开一个内核转储文件，需要输入 g 命令执行一次才真正开始与调试目标建立调试会话。最后再使用 .create notepad.exe 调试一个新创建的进程（也需要恢复执行一次）。此时，打开进程和线程工作窗口，其状态如图 30-6 所示。

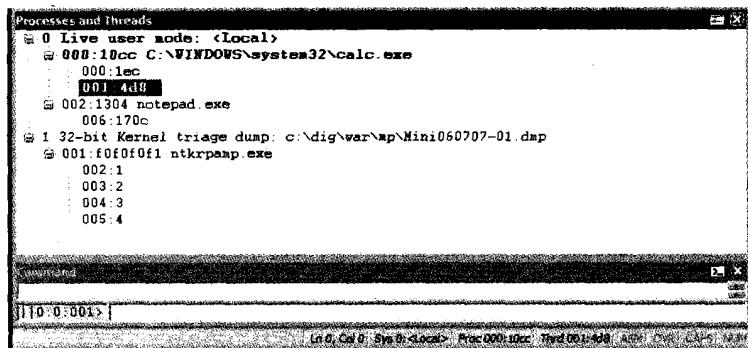


图 30-6 使用 WinDBG 调试多个目标

首先看系统序号的分配，0号代表的是用户态调试目标 Calc.exe 所在的本地系统，1号代表的是内核转储文件所对应的系统。

再看进程号，0 号进程是 Calc 程序，0x10cc 是它的进程 ID。2 号进程是 notepad.exe，0x1304 是它的进程 ID。1 号“进程号”分给了内核转储文件，0xf0f0f0f1 是个虚拟的进程 ID，如果再打开一个内核转储文件，那么它的虚拟进程号是 f0 f0 f0 f2。因为我们的操作顺序是在创建 notepad 之前打开转储文件，所以转储文件的进程号为 1，记事本程序的进程号为 2。

最后再看线程号和处理器号，0 和 1 分给了 Calc 进程的两个线程。2~5 分给了转储文件的四个处理器，6 号分给了记事本程序的唯一线程。

在 WinDBG 的状态条上，也显示了当前的系统号 (Sys 0:<Local>)、进程号 (Proc 000:10cc) 和线程号 (001:4d8)。如果切换到转储文件，那么状态条的显示分别为 Sys 1:c:\dig\...、Proc 001:0 (进程 ID 显示为 0) 和 Thrd 002:0 (0 代表 0 号 CPU)。

图 30-7 中的命令提示符是 ||0:0:001>，其中第 1 个 0 代表当前的系统号，第 2 个 0 代表当前的进程号，001 代表当前的线程号。使用 ||<system_index> s 命令可以切换当前系统，比如执行 ||1 s 便可以把当前系统切换到 1 号。使用 |<process_index> s 可以切换当前进程。因为进程号是全局编排的，所以可以直接从一个系统的某个进程切换到另一个系统的某个进程。使用 ~<thread_index> s 可以切换当前线程，只能在当前系统的范围内切换线程。举例来说，如果当前系统是 0 号，那么执行 ~6 s 便切换到 Notepad 进程的线程，如果执行 ~0 s 便切换回计算器进程的 0 号线程。但是如果目前系统是转储文件，那么执行 ~0 s 或者 ~6 s 会得到错误：

```
||1:0: kd> ~0 s
^ Extra character error in '~0 s'
||1:0: kd> ~6 s
6 is not a valid processor number
^ Extra character error in '~6 s'
```

第三种提示符是 BUSY 提示符，它的内容固定为 BUSY 单词前后各加一个星号，即 *BUSY*。BUSY 提示符的含义是调试会话或调试引擎处于繁忙状态。当调试目标处于运行状态时，WinDBG 会使用 BUSY 提示符，并在命令编辑框中显示调试目标正在运行，也就是图 30-7 左侧所示的样子。

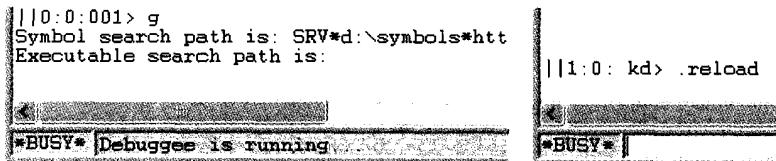


图 30-7 调试目标在运行（左）和执行命令（右）时的 BUSY 提示符

当 WinDBG 开始执行命令时，它也会显示 BUSY 提示符，当命令执行结束后又切换回其他提示符。对于执行时间非常短的命令，我们看不到这种切换，但对于重新加载模块这样需要时间较长的命令，有时可以看到 BUSY 提示符，如图 30-7 右侧所示。

第四，当使用 .abandon 命令放弃所有调试目标后，命令提示符会变为 No Target 加大于号，即图 30-8 所示的样子。

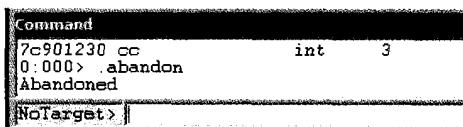


图 30-8 放弃所有调试目标后的命令提示符

第五种提示符是征询用户意见的输入提示符，它由 Input 单词加大于号组成。例如，

图 30-9 显示了调试目标遇到断言失败时，内核调试引擎通过 WinDBG 调试器征询处理意见时的状态。此时用户可以输入字符 b、g、p 或者 t，分别代表中断到调试器，继续执行，终止进程和终止线程。

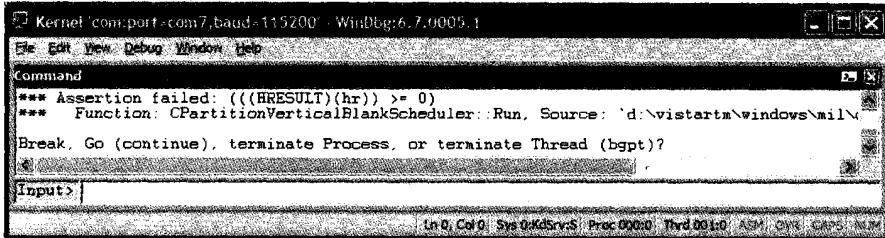


图 30-9 征询用户意见的输入提示符

某些调试命令也会使用输入提示符，例如，实现汇编功能的 a 命令，当使用 e 命令编辑变量时，如果没有指定变量的取值，那么 WinDBG 就会使用输入提示符让用户输入，第 30.14 节将介绍这种情况。

30.4 输入和执行命令

上一节介绍了 WinDBG 的命令提示符，本节将继续介绍输入和执行命令的一些常识和技巧。包括表达式、伪变量、重复执行、条件执行等。

30.4.1 要点

首先，应该在 WinDBG 处于等待命令状态时输入命令。如果提示符显示为 *BUSY*，即使命令编辑框可以输入命令（图 30-7 右），但是这个命令也不会被马上执行，要等 WinDBG 恢复到空闲状态才能执行。此外，还应该记住以下这些要点。

- 可以在同一行输入多条命令，用分号（;）作为分隔符。
- 直接按回车键可以重复上一条命令。比如在使用 u 命令反汇编时，第一次输入 u 命令后，每次再按回车，WinDBG 便继续反汇编接下来的代码。
- 按上下方向键可以浏览和选择以前输入过的命令。
- 大多数命令是不区分大小写的，但是某些命令的选项是分大小写的，以显示栈回溯的 kPL 命令为例，k 不分大小写，P 和 L 是分大小写的。
- 输入元命令时应该以点（.）开始，输入扩展命令时应该以叹号（!）开始。
- 可以使用 Ctrl+Break 来终止一个长时间未完成的命令。如果使用 KD 或者 CDB，那么用 Ctrl+C。
- 按 Ctrl+Alt+V 热键可以启用 WinDBG 的详细输出（Verbose Output）模式，观察到更多的调试信息，再按一次恢复到本来的模式。

- 当进行内核调试时，可以按 Ctrl+Alt+D 热键，让 WinDBG 显示与内核调试引擎之间的数据通信，再按一次可以停止显示。
- WinDBG 的帮助文件是学习和使用 WinDBG 的好帮手，按 F1 热键或者输入.hh 加上希望了解的命令，随时可以打开帮助文件并转到相关的主题。

另外，WinDBG 默认使用十六进制数字，使用命令 n 可以改变默认的数制，也可以在数字前加 0x、0n 和 0y 来显式指定十六进制、十进制和二进制。对于十六进制数，也可以在末尾加字符 h 来表示。对于 64 位长的数字，可以在第 31 位前加`符号，也可以不加，例如 FFFFFFFF`80000000。

30.4.2 表达式

在 4.0 版本之前，WinDBG 只支持使用宏汇编（MASM）语法编写的表达式，4.0 版本引入了对 C++ 表达式的支持，但默认使用的仍是 MASM 表达式。执行.expr 命令可以观察和设置默认使用的表达式语法。使用 @@masm(...) 或者 @@c++(...) 可以显式指定括号中的表达式所使用的语法规则。例如下面的第一条命令显示当前使用的是 MASM 表达式，第二条命令显式声明使用 C++ 表达式：

```
0:000> .expr
Current expression evaluator: MASM - Microsoft Assembler expressions
0:000> ? @@c++(reinterpret_cast <int>(0xfffffffEE))
Evaluate expression: -18 = ffffffEE
```

因为 ?? 命令专门用来评估 C++ 表达式，所以第 2 条命令也可以写为 ?? const_cast<unsigned int>(0xfffffffEE)。

在 MASM 表达式中，除了可以使用加减乘除 (+、-、*、/)、移位 (>>、<<、>>>)、求余 (% 或 mod)、比较 (= 或 ==、>、<、>=、<=、!=)、按位与 (& 或 and)、按位异或 (^ 或 xor)、按位或 (| 或 or)、正负号等运算符外，还可以使用如下特殊的运算符。

- 使用 hi 或者 low 分别得到一个 32 位数的高 16 位或低 16 位。
 - 使用 by 或者 wo 从指定地址分别取得低位的一个字节(BYTE)和一个字(WORD)。
 - 使用 dwo 或者 qwo 从指定地址分别得到一个 DWORD 或者 QWORD (四字)。
 - 使用 poi 从指定地址得到指针长度的数据，例如在 32 位系统中，poi(ebp+8) 可以返回栈帧基址+8 处的 DWORD 值，通常也就是放在栈上的第一个参数的值。
- 为了支持复杂的调试命令，WinDBG 还定义了一些特殊的类似函数的运算符，它们都以 \$ 字符开头。
- \$iment (Address)：返回参数 Address 所代表模块的入口地址，Address 应该为模块的基址。例如，对于 VC 编译器产生的普通 EXE 模块，其基地址为 400000，\$iment (400000) 的返回值就是编译器插入的入口函数 (_WinMainCRTStartup) 地址。
 - \$scmp (“string1”, “string2”)：比较参数指定的两个字符串，与 strcmp 类似。

- `$sicmp ("string1", "string2")`: 比较参数指定的两个字符串, 忽略大小写, 与 `strcmp` 类似。
- `$spat ("string1", "pattern")`: 判断参数 1 指定的字符串是否符合参数 2 指定的模式。模式字符串中可以包含?、*、#等特殊符号, WinDBG 帮助文件中 String Wildcard Syntax 一节包含了详细的说明。
- `$vvalid (Address, Length)`: 判断指定区域是不是有效的内存区, 有效返回 1, 否则返回 0。
- `$fnsucc (FnAddress, RetVal, Flag)`: 根据函数的返回值评估函数是否成功。

另外, 在 MASM 表达式中, 可以使用如下格式来指定源文件的行:

```
'[[Module!]Filename][:LineNumber]'
```

其中, 两端是重音符号, 并不是单引号, 行号应该总是为十进制数, 模块名和源文件名可以省略, 如果省略 WinDBG 会使用当前程序指针所对应的源文件。比如, 可以使用 `bp `d4dtest!d4dtestdlg.cpp:196`` 在 `d4dtestdlg.cpp` 文件的 196 行设置一个断点。

在 C++ 表达式中, 可以使用 C 和 C++ 语言定义的各种运算符, 包括取地址 (&)、引用指针 (*)、索引结构中的字段 (->或.)、指定类名 (::)、类型转换 (`dynamic_cast`、`static_cast`、`const_cast`、`reinterpret_cast`) 等各种运算符, 比如执行 `?? &(this->m_Button1)` 可以显示出 `m_Button1` 对象的所有成员。此外, 在 C++ 表达式中还可以使用如下以 # 号开始的宏。

- `##CONTAINING_RECORD(Address, Type, Field)`: 根据 Field 字段的地址 (Address), 返回这个字段所属的 Type 结构的基址。
- `#FIELD_OFFSET(Type, Field)`: 返回 Field 字段在 Type 结构中的字节偏移。
- `#RTL_CONTAINS_FIELD(Struct, Size, Field)`: 判断在 Size 指定的长度范围内是否包含了 Field 字段。
- `#RTL_FIELD_SIZE(Type, Field)`: 返回结构 (Type) 中指定字段 (Field) 的长度。
- `#RTL_NUMBER_OF(Array)`: 返回数组的元素个数。
- `#RTL_SIZEOF_THROUGH_FIELD(Type, Field)`: 返回截止到指定字段 (Field) 的结构 (Type) 长度, 包含这个字段。

我们将在后面的内容中演示以上部分运算符和宏的用法, 接下来看一下在命令中填加注释的方法。WinDBG 支持两种方法在命令中加入注释文字。一种是使用 * 命令, 另一种是使用 \$\$ 命令。因为二者都是特别的命令, 所以使用前应该在前一条命令后加上分号作为分隔。二者的差异是, * 之后的所有内容都会被当作注释, 而 \$\$ 后的注释可以用分号结束, 然后后面再写其他命令。

例如, 在命令编辑框中输入 `r eax; $$ address of var_a; r ebx; * var_b; r ecx <will not be executed>`, 那么命令信息区显示的执行结果为:

```
110:2:006> r eax; $$ address of var_a; r ebx; * var_b; r ecx <will not be executed>
eax=001a1ea4
ebx=7ffd000
```

可见，\$\$之后的rebx仍被当作命令执行，而*之后的reecx被当作注释文字。

因为命令的执行结果可以被写入到记录文件中，所以为某些命令加上注释相当于做调试笔记，这是一个好的调试习惯。注释命令的另一个用途就是用在命令程序中，我们将在30.18节介绍。

30.4.3 伪寄存器

为了可以方便地引用被调试程序中的数据和寄存器，WinDBG 定义了一系列伪寄存器（Pseudo-Register）。在命令编辑框和命令文件中都可以使用伪寄存器，解析命令的过程中，WinDBG 的调试器引擎会自动将伪寄存器替换（展开）为合适的值。表 30-3 列出了当前版本的 WinDBG 所定义的伪寄存器。

表 30-3 WinDBG 的伪寄存器

伪寄存器	值的含义
\$ea	上一条指令中的有效地址 (effective address)
\$ea2	上一条指令中的第二个有效地址
\$exp	表达式评估器所评估的上一条表达式
\$ra	当前函数的返回地址 (return address)。例如，可以使用 g @\$ra 返回到上一级函数，与 gu (go up) 具有同样的效果
\$ip	指令指针寄存器。x86 中即 EIP，x64 即 eip
\$eventip	当前调试事件发生时的指令指针
\$previp	上一事件的指令指针
\$relip	与当前事件关联的指令指针，例如按分支跟踪时的分支源地址
\$scopeip	当前上下文 (scope) 的指令指针
\$exentry	当前进程的入口地址
\$retreg	首要的函数返回值寄存器。x86 架构使用的是 EAX，x64 是 RAX，安腾是 ret0
\$retreg64	64 位格式的首要函数返回值寄存器，x86 中是 edx:eax 寄存器对
\$csp	帧指针，x86 中即 ESP 寄存器，x64 是 RSP，安腾为 BSP
\$p	上一个内存显示命令 (d*) 所打印的第一个值
\$proc	当前进程的 EPROCESS 结构的地址
\$thread	当前线程的ETHREAD 结构的地址
\$peb	当前进程的进程环境块 (PEB) 的地址
\$teb	当前线程的线程环境块 (TEB) 的地址
\$tpid	拥有当前线程的进程的进程 ID (PID)
\$tid	当前线程的线程 ID
\$bp	x 号断点的地址
\$frame	当前栈帧的序号
\$dbgttime	当前时间，使用formats 命令可以将其显示为字符串值
\$callret	使用.call 命令调用的上一个函数的返回值，或者使用.fnret 命令设置的返回值
\$ptrsize	调试目标所在系统的指针类型宽度
\$pagesize	调试目标所在系统的内存页字节数

可以直接用上表中的名称来使用伪寄存器，但是更快速的方法是在\$前加上一个@符号。这样，WinDBG 就知道@后面是一个伪寄存器，不需要搜索其他符号。以下是

使用伪寄存器的两个例子，我们将在后面给出更多的例子：

```
!l0:0:001> ln @$exentry
(01012475) calc!WinMainCRTStartup | (0101263c) calc!__CxxFrameHandler
Exact matches:
calc!WinMainCRTStartup = <no type information>
!l0:0:001> ? @$pagesize
Evaluate expression: 4096 = 00001000
```

除了表 30-3 列出的伪寄存器，WinDBG 还为用户准备了 20 个伪寄存器，称为用户定义的伪寄存器（User-Defined Pseudo-Registers），它们的名称是\$0~\$t19，用户可以使用这些寄存器来保存任意的整数值，它们的初始值是 0，可以使用 r 命令来设置新的取值。

30.4.4 别名

WinDBG 支持定义和使用三类别名（Alias）。第一类是所谓的用户命名别名（User-Named Alias），即别名的名称和实体都是用户指定的。第二类是固定名称别名（Fixed-Name Alias），其名称固定为\$0~\$u9。第三类是 WinDBG 自动定义的别名（Automatic Aliases）。表 30-4 列出了目前版本的 WinDBG 所包含的自动定义别名。

表 30-4 WinDBG 的自动定义别名

别名名称	含义
\$ntnsym	NT 内核或者 NT DLL 的符号名，内核态调试时值为 nt，用户态时为 ntdll
\$ntwsym	在 64 位系统上调试 32 位目标时的 NT 系统 DLL 符号名，可能为 ntdll32 或 ntdll
\$ntsym	与当前调试目标的机器模式匹配的 NT 模块名称
\$CurrentDumpFile	转储文件名称
\$CurrentDumpPath	转储文件路径
\$CurrentDumpArchiveFile	最近加载的 CAB 文件名称
\$CurrentDumpArchivePath	最近加载的 CAB 文件路径

可以使用 echo 命令来显示某个别名的取值，比如：

```
!l1:0: kd> .echo $ntnsym * 在内核态调试会话中
nt
!l0:2:006> .echo $ntnsym * 在用户态调试会话中
ntdll
```

可以使用 as 命令来定义或者修改用户命名别名，其基本语法如下：

as 别名名称 别名实体

比如，以下例子为内部命令 version 定义了一个别名 v：

```
!l1:0: kd> as v version
```

接下来便可以使用 v 来执行 version 命令：

```
!l1:0: kd> v
Windows Server 2003 Kernel Version 3790 (Service Pack 2) MP (4 procs) Free x86
compatible..
```

可以使用如下命令格式来修改固定别名所代表的实体:

```
r $.u<0~9>=<别名实体>
```

例如,以下第一条命令将 u9 定义为 nt!KiServiceTable, 第二条命令显示它的内容,第三条是在 dd 命令中使用这个别名:

```
||1:0: kd> r $.u9=nt!KiServiceTable
||1:0: kd> .echo $u9
nt!KiServiceTable
||1:0: kd> dd $u9
80834190 8092023a 8096b71e 8096f9be 8096b750 .....
```

下面介绍一下别名的替换规则。首先,如果用户别名与命令的其他部分是明确分隔开的,那么可以直接使用这个别名名称,就像上面用 v 来执行 version 命令那样。但如果用户别名是和命令的其他部分是连续的,那么必须使用 \${用户别名} 将用户别名包围起来,或者使用空格将别名与其他部分分隔开来。举例来说,以下命令为符号 nt!KiServiceTable 定义了一个别名 SST:

```
||1:0: kd> as SST nt!KiServiceTable
```

接下来,我们可以在命令中使用这个别名,例如:

```
||1:0: kd> dd SST 14
80834190 8092023a 8096b71e 8096f9be 8096b750
||1:0: kd> dd SST +8 14
80834198 8096f9be 8096b750 8096f9f8 8096b786
||1:0: kd> dd ${SST}+8 14
80834198 8096f9be 8096b750 8096f9f8 8096b786
```

但像下面这样便会出错:

```
||1:0: kd> dd SST+8 14
Couldn't resolve error at 'SST+8'
```

因为固定别名的长度是确定的,所以使用固定别名时,可以直接使用 \$u0, 不需要大括号,如:

```
||1:0: kd> dd $u9+8 14
80834198 8096f9be 8096b750 8096f9f8 8096b786
```

使用 al 命令可以列出目前定义的所有用户命名别名。使用 ad 可以删除指定的或者全部 (ad *) 用户别名。

30.4.5 循环和条件执行

可以使用 z 命令来循环执行一或多个命令,例如:

```
||0:0:001> r ecx=2 * 将 ecx 设置为 2, 防止循环太多次
||0:0:001> r ecx=ecx-1; r ecx; z(ecx); recx=ecx+1 *递减 ecx 直到为 0, 然后再递增一次
ecx=00000001
redo [1] r ecx=ecx-1; r ecx; z(ecx); recx=ecx+1
ecx=00000000
```

上面的 redo 行便是循环执行的提示, [1]代表循环次数。命令执行结束后再观察

ecx，其值为 1。概言之，z 命令会循环执行它前面的命令，然后测试自己的条件。循环结束后，WinDBG 会继续执行 z 命令后的命令。

循环执行的另一种常用方法是使用!for_each_XXX 扩展命令，比如!for_each_frame 命令可以对每个栈帧执行一个操作，!for_each_local 是对每个局部变量。例如以下命令会打印出每个栈帧的每个局部变量：

```
!for_each_frame !for_each_local dt @#Local
```

命令 j 可以判断一个条件，然后选择执行后面的命令，类似于 C 语言中的 if...else...，其格式为：

```
j <条件表达式> [Command1] ; [Command2]
```

如果条件成立，那么便执行 Command1，否则便执行 Command2。如果要执行一组命令，那么可以使用单引号，即：

```
j Expression ['Command1'] ; ['Command2']
```

例如：

```
0:001> r ecx; j (ecx<2) 'r ecx';'r eax'  
ecx=00000002  
eax=7fffdc000
```

上面的命令是先显示寄存器 ecx 的值，等于 2。而后执行 j 命令，它判断 ecx 是否小于 2，因为不成立，所以便执行后半（else）部分，显示 eax 寄存器的值。因为只有一个命令，所以上面的单引号可以省略，但为清晰考虑，建议大家总是使用引号来包围每组命令。以下是一个更复杂一点的例子：

```
bp `my.cpp:122` "j (poi(MyVar)>5) '.echo MyVar Too Big'; '.echo MyVar Acceptable;  
gc" "
```

上面命令的含义是在 my.cpp 的 122 行设置一个断点，当这个断点命中时，WinDBG 自动执行双引号包围的命令，即 j 命令。J 命令判断变量 MyVar 的值是否大于 5，如果是，则执行第一对单引号包围的命令（显示'MyVar Too Big'，并中断到调试器），如果不是，则执行第二对单引号包围的命令（显示'MyVar Acceptable'，然后立刻恢复目标继续执行）。

条件执行的另一种方法是使用元命令中的.if、.else 和 .elseif。比如上面的那个简单例子可以表示为：

```
r ecx; .if (ecx>2) {r ecx} .else {r eax}
```

每对大括号中可以包含多个以分号分隔的命令。

30.4.6 进程和线程限定符

在很多命令前可以加上进程和线程限定符，用来指定这些命令所适用的进程和线程，表 30-5 列出了两种限定符的典型用法和含义。

表 30-5 进程和线程限定符

进程限定符	含义	线程限定符	含义
!.	当前进程	~.	当前线程
!#	导致当前调试事件的进程	~#	导致当前调试事件的线程
!*	当前进程的所有进程	~*	当前进程的所有线程
!Number	序号为 Number 的进程	~Number	序号为 Number 的线程
!-[PID]	进程 ID 等于 PID 的进程	~~[TID]	线程 ID 等于 TID 的线程

例如，可以使用以下命令来显示 0 号线程的寄存器和栈回溯，尽管当前线程是 1 号线程：

```
0:001> ~0r; ~0k;
```

以上两条命令可以简写为如下形式：

```
0:001>~0e r; k;
```

注意这里，如果没有 e，那么 k 命令便是显示当前线程（1 号）的栈回溯。利用~* 可以对当前进程的所有线程执行一系列命令，比如以下命令对每个线程分别执行 r 和 k 命令：0:001> ~*e r; k;

30.4.7 记录到文件

WinDBG 可以把输入的命令和命令的执行结果记录在一个文本文件中，称为 Log File（日志文件）。可以使用 Edit 菜单的 Open/Close Log File 功能来启用和关闭日志文件，也可以使用.logopen、.logclose、.logfile 三个元命令来打开、关闭和显示日志文件。

本节介绍了使用 WinDBG 命令的基本要领和如何设计比较复杂的命令。我们将在第 30.18 节介绍 WinDBG 命令程序时讨论如何编写更复杂的命令。

30.5 建立调试会话

建立调试会话是开始调试的必须步骤，在调试会话建立以前，除了用于建立调试会话的命令和少数配置命令可以执行外，其他大多数命令都是被禁止的，只有建立调试会话后，WinDBG 才允许执行这些命令。本节我们将讨论使用 WinDBG 调试器建立调试会话的典型方法。

30.5.1 附加到已经运行的进程

有以下几种方法可以把调试器附加到已经运行的进程。

- 使用 WinDBG 的 File 菜单中的 Attach to a Process 命令，或者按 F6 热键，然后在进程列表中选择要附加的进程。
- 将 WinDBG 设置为 JIT 调试器（执行 WinDBG.exe -I），这样当应用程序崩溃时，

在应用程序错误对话框中选择 Debug 系统便会启动 WinDBG 并将其附加到这个进程。详细说明请参见第 12 章关于应用程序错误和 JIT 调试的讨论。

- 启动 WinDBG 时通过-p 开关指定要附加的进程 ID，让 WinDBG 启动后便附加到这个进程。
- 启动 WinDBG 时通过-pn 开关指定要附加进程的程序名，让 WinDBG 启动后搜索程序名对应的进程 ID 并附加到这个进程，例如 c:\windbg\windbg.exe -pn notepad.exe。如果系统中有多个同样程序名的进程，那么 WinDBG 会提示错误。
- 在当前的调试会话中执行.attach 命令。使用这种方法需要有一个调试会话，然后才能输入这个命令，因此这种方法常用于同时调试多个目标的情况。
- 对于使用.abandon 命令抛弃的被调试进程，可以使用 windbg.exe -pe -p PID 的方式重新附加到这个进程。

当调试自动启动的系统服务或者以其他方式自动启动的程序时，可以使用上面介绍的方法来建立调试会话。如果要调试的程序可以在 WinDBG 中启动，那么可以使用下面介绍的创建并调试新进程的方法。

30.5.2 创建并调试新的进程

与将调试器附加到已经运行的进程类似，创建并调试一个新的进程也有很多种方法。

- 使用 WinDBG 的 File 菜单中的 Open Executable 命令，或者按 Ctrl+E 热键，然后使用打开可执行文件对话框选择要调试的程序文件，根据需要可以同时指定程序的命令行参数和启动目录。
- 启动 WinDBG 时将要调试的程序文件作为命令行参数传递给 WinDBG。
- 在注册表的 HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\Current-Version\Image File Execution 键下，创建一个以要调试程序文件名（不包括路径）命名的子键，然后在这个子键下建立一个名为 Debugger 的 REG_SZ 类型的键值，取值为 WinDBG 程序的完整路径，比如 c:\windbg\windbg.exe。有了这个设置后，再运行要调试的程序时，操作系统就会先启动 WinDBG，并把要执行的程序名和路径传递给它，我们在第 10 章（10.3.4 节）详细介绍了这种用法。
- 在当前的调试会话中使用.create 命令，使用这种方法需要有一个调试会话，然后才能输入这个命令，因此这种方法常用于同时调试多个目标的情况。

无论使用哪种方法，被调试程序都是作为 WinDBG 的子进程而被创建的，WinDBG 在创建这个子进程时会指定要与这个程序建立调试关系。

30.5.3 非入侵式调试

非入侵式调试是调试用户态进程的一种特殊方式。使用这种方式，WinDBG 与目标进程没有真正建立调试与被调试的关系，不能接收到任何调试事件，因此只可以使用行观目标进程的各种命令，不可以使用控制调试目标执行的各种命令，包括单步跟踪、继续执行等。非入侵式调试的好处是减小调试器对目标进程的干预，最大程度地减少海森伯效应。

非入侵式调试只适用于附加到已经运行进程的情况。当使用图形界面方式附加到一个进程时，只要选中对话框中的 Noninvasive 复选框。对于使用命令行的情况，只要加上-pv 开关。如果使用.attach 命令，那么只要加上-v 开关。JIT 调试不支持这种方式。

因为没有建立真正的调试关系，所以使用一个 WinDBG 以非入侵方式调试一个进程时，不会影响其他调试器再附加到这个进程进行普通方式的调试。

因为 Windows NT 和 Windows 2000 不支持调试器与调试目标分离（Detach），一旦建立调试关系，那么终止调试会话就会终止被调试进程。当调试这些系统中的系统服务时，以非入侵方式调试有时很有用，分析结束后只要执行分离命令便可以恢复调试目标继续运行，不然的话，就需要重新启动被调试的服务。

30.5.4 调试内核目标

与用户态调试相比，建立双机内核调试要复杂一些。通常包括以下几个步骤：

第一步，选择两台系统之间的通信方式，目前 WinDBG 支持串行口、1394、USB2（USB 2.0）三种方式。串行口方式需要有一根 Null-Modem 电缆，并且要求主机和目标系统都具备串行口。对于主机端，可以使用“USB 到串口”转接头提供的串口。对于目标系统，必须具有真正的串口设备和接口。尽管串口方式的通信速度不如 1394 和 USB2，但是串口方式的优点是兼容性好，稳定可靠。1394 又称为火线，使用这种方式要求两台系统都具有 1394 端口。1394 的通信速度比串口要高得多，但是这种方法的缺点是不够稳定，有些时候难以建立调试连接。USB2 方式是一种比较新的方法，它要求目标系统是 Windows Vista 或者更高的版本。因为 USB 通信具有方向性，个人电脑系统上的 USB 端口都是所谓的上游（upstream）接头，所以 USB2 方式需要有一根特殊的 USB2.0 主机到主机（Host to Host）电缆。另外，并不是每个 USB2.0 端口都可以支持内核调试。我们在第 18.2 节详细介绍过以上各种连接方式。

第二步，启用目标系统的内核调试引擎。虽然内核调试引擎内建在 Windows 系统中，但默认是禁止的，在调试前需要先启用它。如果目标系统是 Vista 之前的版本（Windows 2000、XP 或 NT），那么应该修改 Boot.INI。如果目标系统是 Windows Vista，那么应该使用 BCDEdit 工具来修改启动选项。我们在第 18.3 节详细介绍过具体的修改方法。

第三步，使用以下两种方法之一在主机上启动 WinDBG 和内核调试会话。

- 不带命令行参数直接启动 WinDBG，然后在其 File 菜单选择 Kernel Debug，或者按 Ctrl+K。然后在图 18-9 所示的内核调试对话框中选择与通信电缆和目标机器一致的类型和参数。
- 使用 -k 开关和命令行参数来启动 WinDBG。例如，windbg -k com:port=Com1, baud=115200

无论使用上面何种方法，WinDBG 都会显示‘Waiting to reconnect...’并进入等待状态，等待来自目标系统的调试数据，并以一定时间间隔（10 秒钟）发送复位数据包（PACKET_TYPE_KD_RESET）。当目标系统在启动早期初始化内核调试引擎时会向主机上的调试器发送信息，WinDBG 收到后便会开始与其对话以建立调试连接。如果 WinDBG 错过了调试引擎主动发送的数据，那么可以在调试器上按 Ctrl+Break 来触发 WinDBG 调试器主动向目标系统发送信息，如果发送成功，二者也会开始通信并建立起调试连接，详见第 18.7 节。

30.5.5 本地内核调试

有以下三种方式启动本地内核调试。

- 运行 WinDBG，接着选择 File 菜单中的 Kernel Debug，然后在图 18-9 所示的对话框中选择 Local。
- 使用 -kl 作为命令行参数来启动 WinDBG。
- 在当前调试会话中执行.attach -k 命令，这种方式需要先有一个调试会话，适用于调试多个目标的情况。

对于 Windows Vista，需要以调试选项启动当前系统才能进行本地内核调试，Windows XP 没有这个要求。Windows 2000 或者更早的 Windows 不支持本地内核调试。我们在第 18.8 节详细讨论过本地内核调试。

30.5.6 调试转储文件

WinDBG 支持以下三种方式来打开一个转储文件。

- 运行 WinDBG，接着选择 File 菜单中的 Open Crash Dump，然后选择要打开的转储文件。
- 使用命令行方式，通过 WinDBG 的-z 开关来指定要打开的转储文件。
- 在当前调试会话中执行.opendump 命令，这种方式需要先有一个调试会话，适用于调试多个目标的情况。

我们在第 12.9 节和第 13.5 节分别详细介绍过调试用户态转储文件和系统转储文件的方法。

30.5.7 远程调试

WinDBG 工具包支持多种远程调试方式。一种方法是在远程的计算机上运行 DbgSrv（远程用户态调试）或者 KdSrv（远程内核态调试）作为服务器，然后将本地的 WinDBG 连接到远程的服务器。另一种方法是在服务器端和客户端都使用 WinDBG。本节将介绍后一种方法。

首先服务端（被调试程序运行的系统）和客户端（调试者进行调试的系统）都应安装相同版本的 WinDBG 工具包，而且主机和客户机之间应该有网络连接，局域网或者互联网都可以。

然后，在服务端使用以下两种方式之一启动服务器。

- 命令行方式，使用-server 开关来指定连接方式，例如以下命令创建了一个使用命名管道（名称为 advdbg）方式通信的服务器：windbg -server npipe:pipe=advdbg。WinDBG 启动后再使用前面介绍的方法与被调试机器建立内核调试连接。
- 启动 WinDBG 并建立内核调试会话，与调试目标建立连接后再执行.server 命令。比如执行.server npipe:pipe=advdbg，那么 WinDBG 会显示：

```
0:001> .server npipe:pipe=advdbg
Server started. Client can connect with any of these command lines
0: <debugger> -remote npipe:Pipe=advdbg,Server=ADVDBGPC
```

而后，在客户端使用以下两种方式之一与服务端建立连接。

- 命令行方式，使用-remote 开关来指定连接方式，例如可以使用以下命令与上面创建的服务器建立连接：WinDBG -remote npipe:server=ADVDBGPC,pipe=advdbg
- 直接启动 WinDBG，然后选择 File 菜单的 Connect to Remote Session (Ctrl+R)，而后在“连接到远程调试会话 (Connect to Remote Debugger Session)”对话框中直接输入连接字符串 (npipe:Pipe=advdbg,Server=ADVDBGPC) 或者点击 Browse 按钮，随后输入服务器机器名，然后从 WinDBG 搜索到的服务器中选择要连接的目标。

成功连接后，服务器端的命令信息区会提示有客户连接到这个调试会话：

```
ADVDBGPC\raymond (npipe advdbg) connected at Thu Jul 05 18:24:01 2007
```

因为作者是使用同一台机器同时作为客户端和服务端，所以客户端的机器名与服务器端的一样。之后可以在客户端或服务端的 WinDBG 中执行各种调试命令，执行结果会同时显示在两个调试器中。如果使用 TCP 端口方式建立连接，那么服务器端可以执行类似这样的命令：.server tcp:port=2002,password=2008，其中的端口号和密码可以根据需要改变。

30.6 终止调试会话

WinDBG 提供了多种方式来终止调试会话，本节将分别作简单介绍。

30.6.1 停止调试

当WinDBG处于命令模式时，在WinDBG的Debug下拉菜单中选择Stop Debugging可以终止当前的调试会话，使调试器恢复到赋闲（Dormant）状态。如果是在调试活动的用户态目标，那么这一操作也会导致调试目标被终止。如果是在调试活动的内核目标，那么目标系统会保持被中断到调试器的状态，可以重新与其建立连接。也可以使用标准命令q来停止调试。

30.6.2 分离调试目标

可以使用分离调试目标功能来结束调试器与调试目标的调试关系。操作方式有两种，一种是使用Debug菜单中的Detach Debuggee命令，另一种是输入.detach命令。

当调试用户态的活动目标时，这一命令会保持目标进程继续运行，这与选择Stop Debugging不同。但因为这一功能依赖于Windows XP才引入的操作系统支持（参见DebugSetProcessKillOnExit API），所以要求目标系统为Windows XP或者更高版本。当只调试一个目标时，执行这个命令后WinDBG就会进入到赋闲状态，如果调试多个目标，那么可以继续调试其他目标。

当调试单一的内核目标时，执行分离操作和上面介绍的Stop Debugging操作具有同样的效果。

30.6.3 抛弃被调试进程

使用分离调试目标命令时，系统会修改目标进程的进程属性，使其脱离被调试状态成为一个普通的进程。如果想让其保持被调试状态，那么可以使用.abandon命令来抛弃当前的被调试进程，例如：

```
0:000> .abandon
Abandoned
```

如果只有一个调试目标，那么执行这个命令后，调试器便会恢复到无调试目标状态，命令提示符变为No Target。如果有多个调试目标，那么可以继续调试其他目标。

被调试进程被抛弃后仍处于挂起状态。简单来说，这个命令只是在调试器中执行注销操作，但是并没有把被调试进程恢复到调试前的状态。这种情况下，可以使用另一个调试器附加到被调试进程，但需要在启动调试器的命令行中指定-pe开关，比如：

```
WinDBG -pe -p 2272
```

其中2272是处于被抛弃状态的进程的进程ID。如果没有-pe开关，那么调试器会附加失败，并报告DebugPort不为空。有了-pe开关后，WinDBG会知道这是重新附加，不会报告错误，调试器引擎会产生一个合成的异常，触发调试器进入命令模式，使调试可以继续。与第一次附加到一个进程不同，重新附加不会收到调试子系统所发送的关于现有模块和线程的杜撰调试事件。

30.6.4 杀死被调试进程

在内核调试时，可以使用.kill 命令杀死指定的进程，在用户态调试时，可以使用这个命令杀死当前的被调试进程，例如：

```
0:000> .kill
Terminated. Exit thread and process events will occur.
```

事实上，这个命令会调用操作系统的 TerminateProcess API 来终止当前线程，其执行过程如清单 30-1 如示。

清单 30-1 使用.kill 命令分离调试目标的执行过程

```
0:001> kn
# ChildEBP RetAddr
00 00f0e6c0 0229aa44 kernel32!TerminateProcess      //终止进程的 API
01 00f0e6d4 020f3e05 dbgeng!LiveUserDebugServices::TerminateProcess+0x14
02 00f0e6f8 0219250e dbgeng!LiveUserTargetInfo::TerminateProcess+0xc5
03 00f0e77c 020f8799 dbgeng!ProcessInfo::Separate+0x2ee //进程对象的分离方法
04 00f0e8e4 0210577f dbgeng!ParseSeparateCurrentProcess+0x2e9
05 00f0e8f8 0218758e dbgeng!DotCommand+0x3f          //元命令处理函数
06 00f0e9d8 021889a9 dbgeng!ProcessCommands+0x4be    //命令分发函数
.....                                                 //以下栈帧省略
```

执行.kill 命令后，仍然可以观察调试目标的数据。当再执行 g 命令时，WinDBG 会收到线程退出事件。如果只是在调试一个进程，那么 WinDBG 会结束当前调试会话恢复到赋闲状态。如果同时调试多个进程，那么调试会话不会终止，还可以继续调试其他目标。

30.6.5 调试器终止或僵死

如果直接关闭调试器程序，那么它所建立的调试会话也会终止，调试会话中如果包含活动的调试目标进程，那么这些进程也会随之终止。

如果调试器因为某种原因僵死，但是调试任务尚未完成，此时可以使用上面介绍的-pe 开关启动一个新的调试器附加到被调试的进程，然后再终止僵死的调试器实例。

30.6.6 重新开始调试

Debug 菜单中的 Restart (Ctrl+Shift+F5 热键) 菜单项和.restart 命令用来重新开始当前的调试会话。如果调试目标是调试器创建的用户态进程，那么执行这个命令后目标进程会被关闭并重新运行。如果调试目标是调试器附加到已经运行的进程，那么执行这个命令时 WinDBG 会提示如下信息：

```
0:000> .restart
Process attaches cannot be restarted. If you want to
restart the process, use !peb to get what command line
to use and other initialization information.
```

当进行内核态调试时，Restart 命令相当于重新启动调试器然后再建立调试连接，如果要让目标系统重新启动，那么可以使用.reboot 命令。

30.7 理解上下文

Windows 是个典型的多任务操作系统，在一个系统中可以有多个登录会话（Logon Session），每个会话中可以运行多个进程，每个进程又可以包含很多个线程。在调试这样的系统时，大多数命令操作或者执行结果都是基于一定上下文的（Context）。根据 Windows 操作系统的特征，WinDBG 定义了如下几种上下文：会话上下文、进程上下文、寄存器上下文和局部（变量）上下文。本节将分别介绍每种上下文的含义和切换方法。

30.7.1 登录会话上下文

Windows 支持同时有多个登录会话，每个会话有自己的输入输出设备和桌面。在典型的 Windows XP 系统中通常只有一个会话，当从另一台机器使用远程桌面功能登录到这个系统后，系统中便有了两个会话。Windows Vista 引入了会话隔离（Session Isolation）技术让所有系统服务运行在会话 0 以增强系统服务的安全性，所以典型的 Vista 系统中至少有两个会话。

所谓登录会话上下文（Login Session Context）就是当前操作或者陈述所基于的登录会话语境。例如，对于会话 A 的所有进程来说，会话 A 的状态和属性便是它们的会话上下文。使用!session 扩展命令可以显示或者切换登录会话上下文。在内核调试时，可以使用!session 命令观察和设置会话信息，例如：

```
0: kd> !session
Sessions on machine: 2                                [系统中共有两个会话]
Valid Sessions: 0 1                                     [有效的会话 ID 是 0 和 1]
Current Session 0                                      [目前的会话 ID 是 0]
0: kd> !session -s 1                                    [使用-s可以设置当前的会话上下文]
Sessions on machine: 2                                [系统中共有两个会话]
Implicit process is now 848178d8                      [同时把默认的进程切换为 848178d8]
WARNING: .cache forcedecodeuser is not enabled [见下文]
Using session 1                                         [使用 1 号会话作为会话上下文]
```

改变会话后，默认进程也随之改变成新会话中的进程，因此以前缓存的用户空间数据不再有效。为了避免用户观察到错误的数据，可以使用.cache 命令在缓存选项中加入 forcedecodeuser 或者 forcedecodeptes 选项禁止缓存功能，让调试器每次都重新读取内存数据。上面命令结果中的警告信息告诉我们，目前还没有启用这两个缓存选项。

每个进程的 EPROCESS 结构的 Session 字段记录着这个进程所属的会话。使用!sprocess 扩展命令可以列出指定会话中的所有进程。每个会话都会包含 Windows 子系统服务器进程（CSRSS）。另外，会话管理器进程本身不属于任何一个会话。

目前，会话上下文只有在内核调试时才有意义，所以!session 和!sprocess 命令也只有在调试内核目标时才能使用。

30.7.2 进程上下文

所谓进程上下文就是指当前操作或者陈述所基于的进程语境。我们知道，Windows 系统中的内核空间是共享的，但用户空间是独立的。例如，在典型的 32 位 Windows 系统中，每个进程的进程空间是 4GB，高 2GB 是内核空间，低 2GB 是用户空间，在同一个系统中，所有进程的高 2GB 内存空间都是相同的，但是低 2GB 空间是各自独立的。

在内核调试时，如果要观察内核空间的数据，那么可以不必关心当前进程是哪一个，但如果要观察用户空间的数据，那么就必须注意当前进程是不是要观察的进程。因为同一个用户态地址在不同进程中的含义是不同的。当调试目标中断调试器中后，WinDBG 会根据调试事件的内容将相关的进程设置为默认进程。如果要观察其他进程的用户空间，那么必须先将进程上下文切换到那一个进程。WinDBG 的.process 命令用来观察和设置默认进程，例如，以下命令将进程 83f7fc78 设置为默认进程：

```
1: kd> .process 83f7fc78
Implicit process is now 83f7fc78
```

其中 83f7fc78 是进程的 EPROCESS 结构的地址。使用!process 0 0 命令可以列出系统中的所有进程的基本信息，其中包含 EPROCESS 结构的地址。

一个有关的命令是.context，它可以设置或者显示用来翻译用户态地址的页目录基址（Base of Page Directory）。例如以下命令显示当前使用的页目录基址（物理地址）：

```
kd> .context
User-mode page directory base is a675000
```

页目录基址是进程的一个重要属性，因此使用.process 设置进程上下文时，它会自动设置页目录基址。

对于 x86 系统，cr3 寄存器用来存放页目录基址，每个进程的用户空间都是基于一个页目录基址的，因此.context 命令和.process 命令的效果几乎是一样的。对于安腾系统，一个进程可能使用多个页目录基址，这时使用.process 命令切换更高效。

当调试用户态目标时，所有虚拟地址都是相对于当前进程的，不需要切换进程上下文，因此.process 和.context 命令都只能用在内核态调试会话中。当在一个调试会话中调试多个用户态目标时，应该使用!<进程号> s 命令来切换当前进程。

30.7.3 寄存器上下文

所谓寄存器上下文（Register Context）就是寄存器取值所基于的语境。因为一个

CPU只有一套寄存器，所以当它轮番执行系统中的多个任务（线程）时，CPU寄存器中存放的是当前正在执行线程的寄存器值。对于没有执行的线程，它的寄存器值被保存在内存中，当CPU要执行这个任务时，这些寄存器值被从内存加载到物理寄存器中。

当我们在调试器中观察一个线程的寄存器（不包括MSR寄存器）时，这个线程是处于挂起状态的，所以我们看到的寄存器都是保存在内存中的寄存器值，而不是此时物理寄存器的值。当我们修改寄存器时，也是修改保存在内存中的寄存器值。

系统在以下几种情况下会将CPU的寄存器值保存到当前线程的上下文记录（Context Record）中。

- 当系统做线程切换时，系统会将要挂起线程的寄存器取值保存起来，这个上下文常被称为线程上下文。
- 当发生中断或者异常时，系统会将当时的寄存器取值保存起来，这个上下文常被称为异常上下文。

使用.thread命令可以显示或者设置寄存器上下文所针对的线程，例如以下命令显示当前的隐含线程：

```
1: kd> .thread
Implicit thread is now 83f81950
```

使用!process <所属进程的EPROCESS结构地址> f可以列出一个进程的所有线程，包括每个线程的ETHREAD结构，把ETHREAD结构的地址作为.thread命令的参数，便可以将这个线程的上下文设置为新的线程上下文：

```
1: kd> .thread 84018d78
Implicit thread is now 84018d78
```

这时再使用观察寄存器和栈命令，WinDBG会提示命令结果是针对上次设置的上下文的，例如：

```
1: kd> r
Last set context:
eax=00000000 ebx=00000000 ecx=00000000 edx=00000000 esi=00000000 edi=00000000
...
1: kd> kv
*** Stack trace for last set context - .thread/.cxr resets it
ChildEBP RetAddr  Args To Child
9ce23be0 818ac9cf 84018d78 818f4820 84018e00 nt!KiSwapContext+0x26 ...
```

输入.cxr或者输入不带参数的.thread命令，可以将线程上下文恢复成以前的情况。

当调试用户态的转储文件时，可以使用.ecxr命令将转储文件中保存的异常上下文设置为寄存器上下文。

30.7.4 局部（变量）上下文

所谓局部上下文（Local Context），就是指局部变量所基于的语境。局部变量是指定义在函数内部的变量，这些变量的含义与当前的执行位置密切相关。在调试时，调

试器默认显示的是当前函数（程序指针）所对应的局部上下文。因为当前函数和局部变量都是与栈帧密切相关的，所以 WinDBG 调试器通常使用栈帧号来代表局部上下文。

使用不带任何参数的.frame 命令可观察当前的局部上下文，例如：

```
0:000> .frame
00 0012fdb4 7e418724 UefWIn32!WndProc+0xe1 [C:\...\UefWin32.cpp @ 151]
```

这说明当前栈帧对应的函数是 WndProc，此时使用 dv 命令可以显示这个函数的参数和局部变量：

```
0:000> dv
        rt = struct tagRECT
        hWnd = 0x002d0500 ...
```

使用.frame 加上栈帧号可以将局部上下文切换到指定的栈帧，例如：

```
0:000> .frame 5
05 0012ff30 004018b3 UefWIn32!WinMain+0xf3 [C:\...\UefWin32.cpp @ 48]
```

此时可以使用 dv 命令显示 WinMain 函数的参数和局部变量：

```
0:000> dv
        hInstance = 0x00400000
        hPrevInstance = 0x00000000 ...
```

值得说明的是，因为 VC 编译器默认将类型符号放在 VCx0.PDB 文件中，而 WinDBG 不会自动加载这样的符号文件，所以在显示局部变量时，会显示很多 no type information 错误。解决的方法是将符号格式设置为 C7 Compatable (Settings>C++ > General > Debug Info)，上面的实验结果就是使用这种格式显示的。另一种解决方法是在链接选项中指定/PDBTYPE:CON 选项，参见第 25.7.2 节。

最后要说明的是，改变大范围的上下文必然会影响小范围的上下文。例如，线程上下文切换后，那么局部变量上下文也一定会变化。

30.8 调试符号

在第 25 章中，我们详细地介绍了调试符号的概念、种类、产生过程和存储方式。本节将讨论如何在 WinDBG 调试器中使用调试符号，包括加载调试符号，设置调试符号选项以及解决有关的问题。

30.8.1 重要意义

调试符号 (Debug Symbols) 是调试器工作的重要依据，保证调试符号的准确对于调试器的正常工作非常重要。如果缺少调试符号或调试符号不匹配，那么调试器就可能显示出错误的结果。为了让大家对这一点有深刻的认识，我们先来看一个简单的例子。使用 WinDBG (尚未设置符号路径) 附加到一个记事本进程上，然后使用~0 s 切换到 0 号线程，再输入 k 命令显示栈回溯信息，其结果如清单 30-2 所示。

清单30-2 没有调试符号时显示的栈回溯

```
0:000> k
*** ERROR: Module load completed but symbols could not be loaded for C:\...notepad.exe
ChildEBP RetAddr
WARNING: Stack unwind information not available. Following frames may be wrong.
0007fed8 01002a1b ntdll!KiFastSystemCallRet
0007ff1c 01007511 notepad+0x2a1b
*** ERROR: Symbol file could not be found. Defaulted to export symbols for
C:\...\kernel32.dll -
0007ffc0 7c816fd7 notepad+0x7511
0007fff0 00000000 kernel32!RegisterWaitForInputIdle+0x49
```

在上面的结果中，WinDBG报告了两个错误和一条警告，都与符号有关。第一个错误是告诉我们未能为 notepad.exe 加载符号。接下来的警告告诉我们因为缺少符号文件提供栈展开信息，所以其下各帧的信息可能是错误的。这个警告决不是空穴来风，看到了这样的警告，确实需要提高警惕，开始以怀疑的眼光观察其后的内容。例如最下面一行显示的函数名是 kernel32.dll 中的 RegisterWaitForInputIdle 函数。栈回溯结果的最下面一个栈帧对应的是当前线程的起始函数，这个函数名怎么会是线程的起始函数呢？键入.symfix c:\symbols 命令设置符号文件的搜索路径（稍后将详细介绍这条命令），然后输入.reload 加载符号，再次输入 k 命令的结果如清单 30-3 所示。

清单30-3 有调试符号时显示的栈回溯

```
0:000> k
ChildEBP RetAddr
0007feb8 7e4191ae ntdll!KiFastSystemCallRet
0007fed8 01002a1b USER32!NtUserGetMessage+0xc
0007ff1c 01007511 notepad!WinMain+0xe5
0007ffc0 7c816fd7 notepad!WinMainCRTStartup+0x174
0007fff0 00000000 kernel32!BaseProcessStart+0x23
```

这次的结果中没有任何错误和警告，是正确的结果。与这个结果相比，清单 30-2 中的显示不仅少了一个栈帧，而且显示出的四个栈帧中有三个都是不准确的，可见调试符号的重要性。

30.8.2 符号搜索路径

大多数调试任务都涉及到多个模块，因此需要加载很多个符号文件，而且这些符号文件很可能不在同一个位置上。为了方便调试，WinDBG 允许用户指定一个目录列表，当需要加载符号文件时，WinDBG 会从这些目录中搜索合适的符号文件。这个目录列表被称为符号搜索路径，简称符号路径（Symbol Path）。在符号路径中可以指定两类位置，一类是普通的磁盘目录或者网络共享目录的完整路径，另一类是符号服务器，多个位置之间使用分号分隔。例如，以下是一个典型的符号路径：

```
SRV*d:\symbols*http://msdl.microsoft.com/download/symbols;c:\work\debug;
```

第一个分号后面定义的是一个本地目录，前面定义的是符号服务器，我们稍后再详细介绍。可以有几种方法来设置符号路径。

- 设置环境变量 _NT_SYMBOL_PATH 和 _NT_ALT_SYMBOL_PATH。
- 启动调试器 (WinDBG) 时，在命令行参数中通过 -y 开关来定义。
- 使用 .sympath 命令来增加、修改或者显示符号路径。如执行 sympath + c:\folder2 便将 c:\folder2 目录加入到符号搜索路径中。
- 使用 .symfix 命令来自动设置符号服务器（详见下文）。
- 使用 WinDBG 的 GUI，通过 File>Symbol File Path 菜单打开 Symbol Search Path 对话框，然后通过图形界面进行设置。

执行不带任何参数的 .sympath 命令可以显示当前的符号路径。

30.8.3 符号服务器

无论是用户态调试，还是内核态调试，通常都涉及到很多个模块，而且不同的模块可能属于不同的开发部门或者公司，一个模块通常还会有很多个不同的版本，所以在调试时要为每个模块都找到正确的符号文件并不是一件简单的事。

解决以上问题的一个有效方法是使用符号服务器 (Symbol Server)。简单来说，符号服务器就是用来存储调试符号文件的一个大文件库，调试器可以从这个文件库中读取指定特征（名称、版本等）的符号文件。图 30-10 画出了符号服务器的示意图，图中左侧是使用 WinDBG 调试器的工作机，右侧是符号服务器。

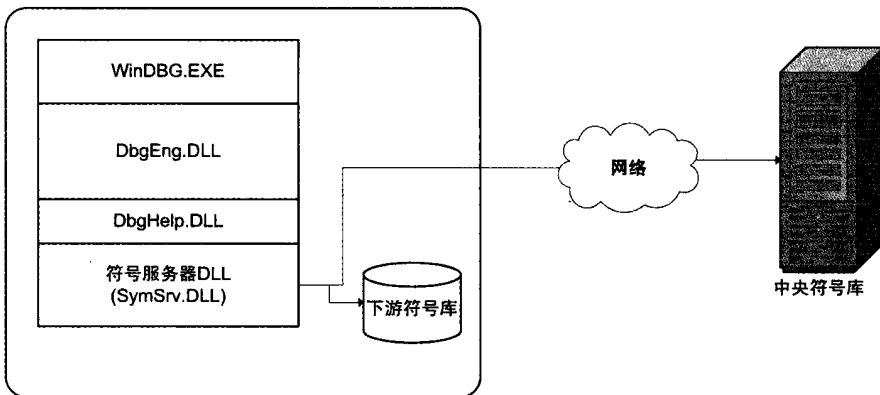


图 30-10 符号服务器架构示意图

在工作机一端，我们画出了 WinDBG 进程中与访问符号有关的各个模块。其中，DbgHelp.DLL 是 Windows 操作系统的调试辅助库模块，WinDBG 通过它读取和解析调试符号；符号服务器 DLL 是符号服务器的本地模块，负责从符号服务器查找、下载和管理符号文件。

为了避免重复下载以前下载过的符号文件，符号服务器 DLL 会将下载好的文件保存在本地的一个文件夹中，这个文件夹使用与符号服务器上相似的方式来组织符号文

件，称为下游符号库（Downstream Store），以便于符号服务器上的中央符号仓库（Centralized Store）相区分。当符号服务器 DLL 接收到 DbgHelp 的请求需要某个符号文件时，符号服务器 DLL 会先在下游仓库中寻找，如果寻找不到才到远程的中央符号仓库去寻找。

DbgHelp 通过所谓的符号服务器 API（Symbol Server API）来调用符号服务器 DLL。符号服务器 DLL 输出这些 API 供 DbgHelp 来调用。WinDBG 开发工具包中的 DbgHelp 帮助文件（sdk\help\dbghelp.chm）详细描述了符号服务器 API 的函数原型和功能。根据符号服务器 API 的定义，用户也可以编写符号服务器 DLL 来实现自己的符号服务器，只要这个 DLL 正确地实现和输出符号服务器 API 所定义的函数。WinDBG 工具包中包含了一个符号服务器 DLL，名为 SymSrv.DLL。

可以通过以下格式来向符号搜索路径中加入符号服务器：

```
symsrv*ServerDLL*[DownstreamStore*]ServerPath
```

其中 *ServerDLL* 是符号服务器 DLL 的文件名称，*DownstreamStore* 是下游符号库的位置，*ServerPath* 是符号服务器的 URL 或共享路径，例如以下是两个有效的定义：

```
symsrv*symsrv.dll*\mybuilds\mysymbols  
symsrv*symsrv.dll*\localsrv\mycache*http://www.somecompany.com/manysymbols
```

因为大多用户都是使用 WinDBG 工具包中的 SymSrv.DLL 作为符号服务器 DLL，所以可以使用以下简化形式：

```
srv*[DownstreamStore*]ServerPath
```

其中的 *srv* 相当于 *symsrv*SymSrv.DLL**。

30.8.4 加载符号文件

下面通过几个例子来说明符号文件的加载过程。清单 30-4 列出了在 WinDBG 调试器中使用 *!d kernel32* 命令从符号服务器加载符号文件的过程。也就是调试器工作线程通过 SymSrv 模块向符号服务器请求符号文件的过程。

清单 30-4 从符号服务器获取符号文件的执行过程（节选）

```
0:001> !d kernel32
# ChildEBP RetAddr
00 00f1b158 01d1182a WININET!HttpSendRequestW      //向符号服务器发送请求
01 00f1b180 01d11528 symsrv!StoreWinInet::request+0x2a
05 00f1b4f8 01d06277 symsrv!cascade+0x87
06 00f1ba48 01d06087 symsrv!SymbolServerByIndexW+0x127 //根据索引串找符号文件
07 00f1bc78 0302dfee symsrv!SymbolServerW+0x77      //符号服务器的接口函数
08 00f1c0b8 03018e7d dbghelp!symsrvGetFile+0x12e    //调用符号服务器模块
09 00f1cd00 03019ee7 dbghelp!diaLocatePdb+0x33d    //寻找 PDB 文件
0a 00f1d01c 030415fe dbghelp!diaGetPdb+0x207
0e 00f1dbd4 0303815a dbghelp!SymLoadModuleEx+0x7d   //模块加载函数
0f 00f1dc00 02185a18 dbghelp!SymLoadModule64+0x2a   //调试辅助库的模块加载函数
10 00f1e900 02187ca8 dbgeng!ParseLoadModules+0x188 //解析模块加载命令
11 00f1e9d8 021889a9 dbgeng!ProcessCommands+0xbdb8 //分发命令
```

```

14 00f1eee4 01028553 dbgeng!DebugClient::ExecuteWide+0x6a //调试引擎的接口函数
15 00f1ef8c 01028a43 WinDBG!ProcessCommand+0x143 //处理命令
17 00f1fffb4 7c80b6a3 WinDBG!EngineLoop+0x366 //调试会话工作循环
18 00f1ffec 00000000 kernel32!BaseThreadStart+0x37 //调试会话工作线程

```

其中，从栈帧#16 到栈帧#10 是分发命令的过程，栈帧#0f 是调用 DbgHelp 库的模块加载函数，而后调用 diaGetPdb 发起读取 PDB 文件的过程，栈帧#09 中的 diaLocatePdb 函数是搜索 PDB 文件的一个主要函数，当它在符号搜索路径中指定的普通位置中找不到符号文件时，它会调用 symsrvGetFile 函数试图从符号服务器下载文件。接下来 symsrvGetFile 函数加载符号搜索路径中定义的符号服务器 DLL，并调用它的 SymbolServer 接口函数。

SymbolServer 函数是符号服务器 API 中的一个重要函数，它的作用就是向符号服务器请求指定的符号文件，并返回访问这个文件的完整路径。SymbolServer 函数的典型实现是，如果所需要的符号文件已经在下游库中，那么便返回它的全路径，不然的话便向远程查询；如果在远程查找到，那么便将其下载到下游库，然后返回符号文件在下游库的全路径，它的函数原型如下：

```
BOOL CALLBACK SymbolServer(LPCSTR params, LPCSTR filename,
    PVOID id, DWORD two, DWORD three, LPSTR path);
```

其中，第 1 个参数 params 是符号服务器的设置信息，例如“d:\symbols*http://msdl.microsoft.com/download/symbols”，第 2 个参数 filename 是符号文件名，如 kernel32.pdb，最后一个参数 path 用来返回符号文件的完整路径。第 3~5 三个参数用来定义符号文件的版本特征，它们的用法因 filename 参数中的文件类型而定，如表 30-6 所示。

表 30-6 SymbolServer 函数用来指定文件版本的参数

模块后缀	参数 id	参数 two	参数 three
.dbg	PE 文件头中定义的映像时间戳 (TimeStamp)	PE 文件头中定义的映像文件大小 (SizeOfImage)	没有使用，为 0
PE 文件 (.exe/.dll)	同上	同上	同上
.pdb	PDB 签名	PDB 年龄 (Age)	没有使用，为 0

SymbolServer 首先根据参数中指定的特征调用 SymbolServerGetStringW 函数生成一个索引串。比如，以下是在某一版本的 kernel32.dll 模块生成的符号索引串：

```
0:001> du 00f1ba60
00f1ba60 "006D2240474D414087FF801C64935DDD"
00f1baa0 "2"
```

其中，006D2240474D414087FF801C64935DDD 是 GUID 签名，即{006D2240-474D-4140-87FF-801C64935DDD}，2 是 PDB 文件的年龄 (Age)。

接下来，SymbolServer 函数调用 SymbolServerByIndexW 函数取符合指定索引串的符号文件。后者会调用一个名为 cascade 的函数，cascade 函数先使用 StoreUNC 类在下游库中查找指定的符号文件是否存在，如果存在便返回完整的路径。

如果在下游库中没有找到匹配的符号文件，那么 cascade 便使用 StoreWinInet 类来

搜索远程的中央符号库，即栈帧#04 到栈帧#00 所示的情况。

以下是 SymbolServer 函数返回时 path 参数中所存放的内容：“c:\dstore\kernel32.pdb\006D2240474D414087FF801C64935DDD2\kernel32.pdb”，其中 006D2240474D414087FF801C64935DDD2 就是索引串，c:\dstore 是下游符号库的根目录。

使用.reload 命令可以重新加载所有或者指定模块的符号文件，以下是主要的执行步骤：

```
00f1d488 030380b5 dbghelp!LoadModule+0x501           //DbgHelp 库的内部函数
00f1d4f0 02190a29 dbghelp!SymLoadModuleExW+0x65       //DbgHelp 库的加载模块函数
00f1e440 022215ed dbgeng!ProcessInfo::AddImage+0xb9  //增加模块对象
00f1e8d0 02102822 dbgeng!TargetInfo::Reload+0x1cbd    //调试目标的基类方法
00f1e8e4 0210577f dbgeng!DotReload+0x22               //分发给 Reload 命令
00f1e8f8 0218758e dbgeng!DotCommand+0x3f             //元命令的入口
00f1e9d8 021889a9 dbgeng!ProcessCommands+0x4be        //调试器引擎的命令分发函数
```

因为是元命令，所以 ProcessCommands 先分发给 DotCommand 函数，后者再调用 DotReload，DotReload 交给 TargetInfo 类的 Reload 方法。Reload 方法枚举进程中的各个模块，对于每个模块调用 ProcessInfo 类的 AddImage 方法将其加入到进程信息中。AddImage 方法会调用调试辅助库（dbghelp）的 SymLoadModuleExW 方法来加载这个模块的信息，包括符号文件，接下来的过程与清单 30-4 所示的情况非常类似。

除了使用 ld 和.reload 命令直接加载符号文件，某些使用符号的命令也可以触发调试器来加载符号，比如栈回溯命令（k*）和反汇编命令等。

值得说明的是，因为 WinDBG 默认使用所谓的懒惰式符号加载策略（lazy symbol loading），所以当它接收到模块加载事件时，它并不会立刻为这个模块加载符号文件。因此，当我们观察模块列表时会看到很多模块的符号状态都是 deferred，即推迟加载。

30.8.5 观察模块信息

可以使用以下方法之一来观察模块信息，包括加载符号文件的情况：

- 使用 lm 命令。
- 使用!lmi 扩展命令。
- 使用 WinDBG 图形界面的模块列表对话框（Debug>Modules）。

我们先介绍 lm 命令。如果不指定任何参数，那么 lm 命令显示一个简单的列表：

```
0:001> lm
start   end     module name
01000000 01093000  WinDBG      (pdb symbols)          d:\...\WinDBG.pdb
01400000 015c6000  ext        (deferred)           ..
```

其中 start 列和 end 列分别是该模块在进程空间中的起始地址和终止地址，module name 列是模块名称，接下来的一类是加载符号文件的状态，第 4 列是符号文件的完整

名称（如果已经加载符号文件）或者空白。表 30-7 列出了符号状态列中可能出现的状态信息和它们的含义。

表 30-7 符号文件加载状态

缩写	含义
deferred	模块已经加载，但是调试器还没有试图为其加载符号，会在需要时尝试
#	符号文件和执行映像文件存在不匹配，比如时间戳或校验和不一致
T	时间戳缺失、不可访问或者等于 0
C	校验和缺失、不可访问或者等于 0
DIA	符号文件是通过 DIA (Debug Interface Access) 方式加载的
Export	没有发现符号文件，使用映像文件的输出信息（如 DLL 的 Export）作为符号
M	符号文件和执行映像文件存在不匹配，但是仍然加载了这样的符号文件
PERF	执行文件包含性能优化代码，对地址进行简单加减运算可能产生错误结果
Stripped	调试信息是从映像文件中抽取出来的
PDB	符号信息是 PDB 格式
COFF	符号信息是 COFF 格式 (Common Object File Format)

PDB 文件又分为私有 PDB 文件和公共 PDB 文件，后者是在前者的基础上剥离私有信息后产生的（详见 25.2.3 节）。

如果要为每个模块显示更丰富的信息，那么可以使用 v 选项，例如：

```
0:001> lm v
start end module name
01000000 01093000 WinDBG (pdb symbols) d:\....\WinDBG.pdb
Loaded symbol image file: C:\WinDBG\WinDBG.exe
Image path: C:\WinDBG\WinDBG.exe
Image name: WinDBG.exe
Timestamp: Thu Mar 29 21:09:08 2007 (460C00C4)
CheckSum: 0008852B...
```

如果想控制要显示的模块，那么可以使用如下方法。

- 使用 m 开关来指定对模块名的过滤模式，比如 lm m k* 显示模块名以 k 开头的模块。
- 使用 M 开关来指定对模块路径的过滤模式（参见下文关于 x 命令的说明）。
- 使用 o 开关只显示加载的模块（排除已经卸载的模块）。
- 使用 l 开关只显示已经加载符号的模块。
- 使用 e 开关只显示有符号问题的模块。

也可以使用 !lmi 扩展命令来观察模块的信息，但是这个命令每次只能观察一个模块，清单 30-5 给出了针对 WinDBG 模块的执行结果。

清单 30-5 使用 !lmi 命令观察模块信息

```
0:001> !lmi WinDBG //参数也可以是模块的基址
Loaded Module Info: [WinDBG]
Module: WinDBG //模块名称
Base Address: 01000000 //模块在内存中的基址
Image Name: C:\WinDBG\WinDBG.exe //映像文件的全路径
```

```

Machine Type: 332 (I386)          //模块所针对的CPU架构
Time Stamp: 460c00c4 Thu Mar 29 21:09:08 2007 //时间戳
Size: 93000                      //文件大小,字节
CheckSum: 8852b                  //校验和
Characteristics: 102
Debug Data Dirs: Type Size      VA Pointer //调试数据目录,详见25.4.3节
    CODEVIEW 23,c348,b748 RSDS - GUID: {CDA70185-4AB9-4F6F-8B60-FDC14F75FB31}
        Age: 1, Pdb: WinDBG.pdb
Image Type: FILE    - Image read successfully from debugger.
    C:\WinDBG\WinDBG.exe
Symbol Type: PDB    - Symbols loaded successfully from symbol server.
    d:\symbols\WinDBG.pdb\CDA701854AB94F6F8B60FDC14F75FB31\WinDBG.pdb
Load Report: public symbols , not source indexed

```

30.8.6 检查符号

可以用标准命令 `x` (或 `X`, 不分大小写) 来检查调试符号, 其命令格式如下:

`x [选项] 模块名!符号名`

其中的模块名和符号名都可以包含通配符, *代表0或任意多个字符, ?代表任一个单一字符, #代表它前面的字符可以出现任意次, 比如 `lo#p` 表示所有以 l 开始 p 结束, 中间有任意多个 o 的所有符号, 比如 `lop`, `loop`, `looop`, ...。如果中间允许多个字符重复, 那么可以使用方括号, 例如用 `m[ai]#n` 可以通配 `man`、`min`、`maan`、`main`、`maiaiain` 等。

举例来说, 使用 `x ntdll!dbg*` 可以列出 `ntdll` 模块的所有以 `dbg` 开头的符号, 即:

```

0:000> x ntdll!dbg*
7c95081a ntdll!DbgUiDebugActiveProcess = <no type information>
...

```

第一列是这个符号的地址, 如果符号是函数, 那么便是这个函数的入口地址, 如果符号是变量, 那么便是这个变量的起始地址。等号后面用来显示符号的类型或取值, 这需要私有符号文件中的类型信息。因为我们没有 `NTDLL` 的私有符号信息, 所以 `WinDBG` 显示 `<no type information>`。

打开调试版本的 `dbgee` 小程序, 然后执行 `x dbgee!arg*` 命令, 得到的结果如下:

```

0:000> x dbgee!arg*
0041718c dbgee!argret = 0
00417184 dbgee!argv = 0x003a2e90
0041717c dbgee!argc = 3

```

可见, 等号后面出现了每个变量的取值, `argc` 是命令行参数的个数, `argv` 是参数数组的指针。

类似的, 模块名中也可以使用通配符, 比如 `x *!_crtheap` 会检查所有模块, 看其是否有 `_crtheap` 符号, 如果有便显示出来:

```

0:000> x *!_crtheap
103130d0 MSVCR80D!_crtheap = <no type information>
77c62418 msrvct!_crtheap = <no type information>

```

下面我们看一下 x 命令的选项，根据选项的功能可以分为如下几类。

- 控制显示结果的排列顺序，例如/a 和/A 分别代表按地址的升序和降序，/n 和/N 分别代表按名称的升序和降序，/z 和/Z 分别代表按符号大小（size）的升序和降序。
- 显示符号的数据类型，即/t。
- 显示符号的符号类型和大小 (/v)，其中符号类型分为 local（局部）、global（全局）、parameter（参数）、function（函数）或者 unknown（未知）。
- 按符号大小设置过滤条件，其格式为/s <符号大小>。对于函数类符号，其大小是这个函数在内存中的大小（字节数），对于其他符号，是这个符号的数据类型的大小。
- 控制显示格式，/p 可以省去函数名与括号之间的空格，/q 参数可以启用所谓的引号格式来显示符号名。

下面给出几个例子来说明以上选项。首先我们看/v 选项，在前面的 x dbgee!arg* 命令中加入/v：

```
0:000> x /v dbgee!arg*
prv global 0041718c    4 dbgee!argret = 0
prv global 00417184    4 dbgee!argv = 0x003a2e90
prv global 0041717c    4 dbgee!argc = 3
```

现在的显示多了三列，最左边的 prv 代表这个符号属于私有（private）符号信息，如果是公共符号，那么显示为 pub（public）。第二列是符号类型，global 代表全局变量，接下来是这个符号在调试目标中的地址，第 4 列是符号的大小，这里列出的几个符号的大小都是 4 个字节。如果再增加/t 选项，那么显示结果变为：

```
0:000> x /v /t dbgee!arg*
prv global 0041718c    4 int dbgee!argret = 0
prv global 00417184    4 unsigned short ** dbgee!argv = 0x003a2e90
prv global 0041717c    4 int dbgee!argc = 3
```

可见，在符号大小后面多了数据类型，argret 和 argc 的类型都是 int（整数），argv 是 unsigned short ** 即 wchar_t **，也就是字符串指针数组。以下是使用/v 开关来观察函数符号：

```
0:000> x /v dbgee!wmain
prv func 00411790 51 dbgee!wmain (int, wchar_t **)
```

注意，在函数名 wmain 与左括号之间有一个空格，如果不需要这个空格，那么可以指定/p 开关，即：

```
0:000> x /v /p dbgee!wmain
prv func 00411790 51 dbgee!wmain(int, wchar_t **)
```

可见空格被删除了，这主要是为了复制整个函数声明时会更方便些。以下是使用更多选项的例子：

```
0:000> x /v /q /t /N dbgee!*main*
prv func 00411520 f <function> @!"dbgee!wmainCRTStartup" ()
prv func 00411790 51 <function> @!"dbgee!wmain" ()
```

```

prv global 00417194 4 int @!"dbgee!mainret" = 0
pub global 00418288 0 <NoType> @!"dbgee!__imp____wgetmainargs" = <no type info...>
pub global 00411c92 0 <NoType> @!"dbgee!__wgetmainargs" = <no type information>
prv func 00411540 244 <function> @!"dbgee!__tmainCRTStartup" ()
prv global 00417020 4 unsigned int @!"dbgee!__native_dllmain_reason" = 0xffffffff

```

因为使用了/q 参数，所以以上符号名是以@!"模块名!符号名”的格式显示的。另一点值得注意的是，因为公开的符号信息不包括类型信息，所以其类型部分显示为<NoType>，符号大小也显示为 0。

30.8.7 搜索符号

标准命令 ln (List Nearest Symbols) 用来搜索距离指定地址最近的符号，比如：

```

1kd> ln 8053ca11
(8053ca11) nt!KiSystemService    | (8053ca85)  nt!KiFastCallEntry2
Exact matches:
nt!KiSystemService = <no type information>

```

上面的结果显示了地址 8053ca11 附近的两个符号，其中 KiSystemService 与指定的地址精确匹配。

30.8.8 设置符号选项

元命令.symopt 用来显示和修改符号选项，其命令格式为：

```
.symopt [+/- 选项标志]
```

WinDBG 使用一个 32 位的 DWORD 来记录符号选项，每个二进制位代表一个选项。使用+可以设置指定的标志位，使用-可以移除指定的标志位，不带任何参数便显示当前的设置。表 30-8 列出了目前定义的所有标志位。

表 30-8 符号选项的各个标志位

标志位	标志位名称	描述	当前状态
0x1	SYMOPT_CASE_INSENSITIVE	不分大小写	On
0x2	SYMOPT_UNDNAME	显示未装饰的符号名	On
0x4	SYMOPT_DEFERRED_LOADS	延迟加载符号	On
0x8	SYMOPT_NO_CPP	关闭 C++ 翻译*	Off
0x10	SYMOPT_LOAD_LINES	加载源代码行信息	**
0x20	SYMOPT_OMAP_FIND_NEAREST	允许为优化过的代码使用最近的符号	On
0x40	SYMOPT_LOAD_ANYTHING	降低匹配符号的挑剔度	Off
0x80	SYMOPT_IGNORE_CVREC	忽略映像文件的 CV 记录	Off
0x100	SYMOPT_NO_UNQUALIFIED_LOADS	禁止符号处理器自动加载模块	Off
0x200	SYMOPT_FAIL_CRITICAL_ERRORS	显示关键错误	On
0x400	SYMOPT_EXACT_SYMBOLS	严格评估所有符号文件	Off

续表

标志位	常量	含义	默认值
0x800	SYMOPT_ALLOW_ABSOLUTE_SYMBOLS	允许位于内存绝对地址的符号	Off
0x1000	SYMOPT_IGNORE_NT_SYMPATH	忽略环境变量中的符号和映像路径	Off
0x2000	SYMOPT_INCLUDE_32BIT_MODULES	对于安腾处理器系统，强制列举 32 位模块	Off
0x4000	SYMOPT_PUBLICS_ONLY	忽略全局、局部和作用域相关的符号	Off
0x8000	SYMOPT_NO_PUBLICS	不搜索公共符号表	Off
0x10000	SYMOPT_AUTO_PUBLICS	其他方法失败时才使用 PDB 文件中的公共符号	On
0x20000	SYMOPT_NO_IMAGE_SEARCH	不搜索映像文件	On
0x40000	SYMOPT_SECURE	(内核调试) Secure Mode	Off
0x80000	SYMOPT_NO_PROMPTS	(远程调试) 不显示代理服务器的认证对话框	***
0x80000000	SYMOPT_DEBUG	显示符号加载过程	Off

* 使用 C++ 翻译时，类成员的 _ 或被替换为 ::。

** 在 KD 和 CDB 中默认为 Off，在 WinDBG 中默认为 On。进行源代码级调试时，必须设置这个选项。

*** 在 KD 和 CDB 中默认为 On，在 WinDBG 中默认为 Off。

因为记忆和使用十六进制的标志位比较困难，所以 WinDBG 提供了扩展命令 !sym 来设置常用的选项。比如 !sym noisy 相当于 .symopt+0x80000000，即开启所谓的“吵杂”式符号加载，显示加载符号的过程信息，!sym quiet 相当于 .symopt-0x80000000，用来关闭吵杂模式。

30.8.9 加载不严格匹配的符号文件

在实际工作中，有时要调试的程序只是做了简单的重新构建（rebuild），代码仅有微小的变化或者根本没有变化。这时，如果调试环境中只有旧的符号文件，那么调试器默认仍会因为符号文件和映像文件不匹配而拒绝加载符号文件。

一种解决方法是使用 .reload /i 命令来加载不完全匹配的符号文件。为了便于发现问题，最好先使用 !sym noisy 命令开启加载符号的“吵杂”模式，清单 30-6 给出了启动吵杂模式后重新加载 dbgee.exe 内核模块的执行结果。

清单 30-6 强制加载不严格匹配的符号文件

```
0:000> .reload /i dbgee.exe
SYMSRV: d:\symbols\dbgee.pdb\75DC...15565162\dbgee.pdb not found
SYMSRV: http://msdl.microsoft.com/.../dbgee.pdb/75DC...15565162/dbgee.pdb not found
DBGHELP: C:\dig\dbg\author\code\chap28\dbgee\Debug\dbgee.pdb - mismatched pdb
```

```

DBGHELP: c:\dig\dbg\author\code\chap28\dbgee\Debug\dbgee.pdb - mismatched pdb
DBGHELP: Loaded mismatched pdb for C:\dig\dbg...\Debug\dbgee.exe
*** WARNING: Unable to verify checksum for dbgee.exe
DBGENG: dbgee.exe has mismatched symbols - type ".hh dbgerr003" for details
DBGHELP: dbgee - private symbols & lines
C:\dig\dbg\author\code\chap28\dbgee\Debug\dbgee.pdb - unmatched

```

以上信息说明，WinDBG 在本地符号库（第 2 行）和符号服务器（第 3 行）都没有找到精确匹配的符号文件，然后从 debug 目录加载了不完全匹配的符号文件（第 4、5 行）。执行 lm 命令显示模块列表，可以看到 dbgee 模块的符号状态栏中包含字符 M，表示符号文件和执行映像文件存在不匹配：

```

00400000 0041a000  dbgee      M (private pdb symbols)
C:\... \dbgee\Debug\dbgee.pdb

```

除了使用带有 /i 开关的 reload 命令，也可以通过设置符号选项 SYMOPT_LOAD_ANYTHING (0x40) 来让调试器加载不严格匹配的符号文件。

```

0:000> .symopt+0x40
Symbol options are 0x30277:
0x00000001 - SYMOPT_CASE_INSENSITIVE
0x00000002 - SYMOPT_UNDNAME
0x00000004 - SYMOPT_DEFERRED_LOADS
0x00000010 - SYMOPT_LOAD_LINES
0x00000020 - SYMOPT_OMAP_FIND_NEAREST
0x00000040 - SYMOPT_LOAD_ANYTHING
0x00000200 - SYMOPT_FAIL_CRITICAL_ERRORS
0x00010000 - SYMOPT_AUTO_PUBLICS
0x00020000 - SYMOPT_NO_IMAGE_SEARCH

```

本节使用比较大的篇幅详细介绍了调试符号有关的 WinDBG 命令，熟练使用这些命令对于调试非常重要，希望读者能够认真体会并在实际调试中灵活应用。

30.9 事件处理

正如我们在第 9 章所介绍的，Windows 的调试模型是事件驱动的。整个调试过程就是围绕调试事件的产生、发送、接收和处理为线索而展开的。调试目标是调试事件的发生源，调试器负责接收和处理调试事件，调试子系统负责将调试事件发送给调试器并为调试器提供服务。第 9 章已经详细介绍了调试事件，本节将先做简单回顾，然后从调试器的角度来介绍与调试事件有关的问题。

30.9.1 调试事件与异常的关系

简单说来，异常是调试事件的一种。Windows 定义了 9 类调试事件，分别用以下 9 个常量来表示：EXCEPTION_DEBUG_EVENT (1)、CREATE_THREAD_DEBUG_EVENT (2)、CREATE_PROCESS_DEBUG_EVENT (3)、EXIT_THREAD_DEBUG_EVENT (4)、EXIT_PROCESS_DEBUG_EVENT (5)、LOAD_DLL_DEBUG_EVENT (6)、UNLOAD_DLL_DEBUG_EVENT(7)、OUTPUT_DEBUG_STRING_EVENT(8)、

RIP_EVENT (9)，其中 EXCEPTION_DEBUG_EVENT (1) 便是异常事件的代码。

因为有很多种异常，所以异常事件又根据异常代码分为很多个子类。其他事件都比较单纯，不再包含子类。常见的异常子类有。

- Win32 异常，这是 Windows 操作系统所定义的异常，包括 CPU 产生的异常和系统内核代码定义的异常，典型的有非法访问、除零等。这类异常的异常代码定义在 ntstatus.h 中。
- Visual C++ 异常，这是 Visual C++ 编译器的 throw 关键字所抛出的异常，throw 关键字调用 RaiseException API 产生异常，所有这类异常的异常代码都是 0xe06d7363 (.msc)。
- 托管异常，这是 .Net 程序使用托管方法抛出的异常，所有这类异常的异常代码都是 0xe0636f6d (.com)。
- 其他异常，包括用户程序直接调用 RaiseException API 抛出的异常，以及其他 C++ 编译器抛出的异常等。

除了以上 9 类调试事件，为了复用事件处理机制，调试器定义了某些专门供调试使用的事件，比如 WinDBG 定义了用于将调试器从睡眠状态唤醒的 Wake Debugger 事件，我们把这类事件通称为调试器事件。

30.9.2 两轮机会

我们在第 10 章介绍异常管理时，曾经详细讨论过 Windows 操作系统分发和处理异常的过程，其中最重要的一点就是，对于每个异常 Windows 系统会最多给予两轮处理机会，对于每一轮机会 Windows 都会试图先分发给调试器，然后再寻找异常处理器 (VEH、SEH 等)。这样看来，对于每个异常，调试器最多可能收到两次处理机会，每次处理后调试器都应该向系统返回一个结果，说明它是否处理了这个异常。

对于第一轮异常处理机会，调试器通常是返回没有处理异常，然后让系统继续分发，交给程序中的异常处理器来处理。对于第二轮机会，如果调试器不处理，那么系统便会采取终极措施：如果异常发生在应用程序中，那么系统会启动应用程序错误报告过程（参见 12 章）并终止应用程序；如果发生在内核代码中，那么便启用蓝屏机制停止整个系统。所以对于第二轮处理机会，调试器通常是返回已经处理，让系统恢复程序执行，这通常会再次导致异常，又重新分发异常，如此循环。值得说明的是，对于断点异常和调试异常，调试器是在第一轮就返回已经处理的。

WinDBG 把异常和其他调试事件放在一起管理，但是必须清楚的是，只有异常事件可能有两轮处理机会，异常以外的其他调试事件（比如进程创建）都只有一轮处理机会。

30.9.3 定制事件处理方式

大多数调试器都允许用户来定制处理调试事件的方式，WinDBG也如此。因为异常事件最多有两轮处理机会，而且对于每一轮机会都需要决定如下两个问题：

- 当收到事件通知后是否中断给用户（进入到命令模式）。
- 返回给系统的处理结果，是返回已经处理（handled），还是没有处理（not handled），即所谓的处理状态（handling status），有时也称为继续状态（continue status）。

所以，对于每种异常事件存在以下四个选项：

- 第一轮机会是否中断给用户；
- 第二轮机会是否中断给用户；
- 第一轮机会的处理结果；
- 第二轮机会的处理结果。

前两个选项通常被称为中断选项，后两个被称为继续选项。为了允许用户设置这些选项，不同调试器提供了不同形式的界面。图 30-11 是 VC6 调试器的设置界面，对于每种异常，用户可以设置两轮机会都中断给用户（stop always），也可以只在第二轮时中断给用户（stop if not handled）。看来，VC6 调试器的异常设置界面只允许用户配置中断选项，不允许配置继续选项。

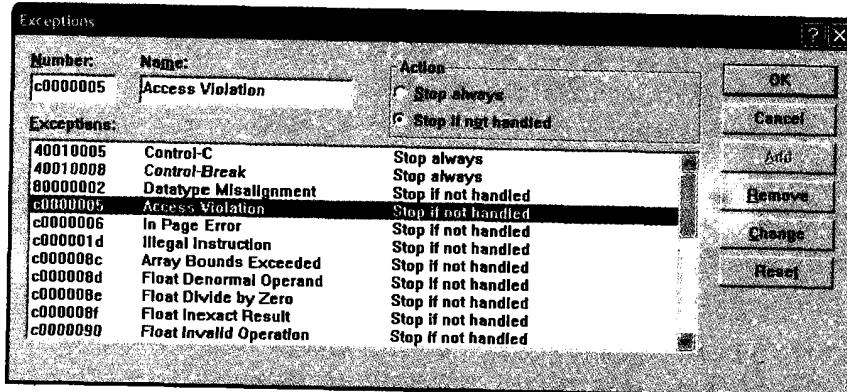


图 30-11 VC6 调试器的异常处理选项对话框

图 30-12 所示的对话框是 Visual Studio .Net 2002 和 2003 (VS7) 的集成调试器所提供的设置界面，虽然界面上看起来与 VC6 的差异很大，但本质上变化不大，只不过改变了设置中断选项的方式，上面一组单选按钮用来设置第一轮的中断选项，下面一组用来设置第二轮的中断选项。

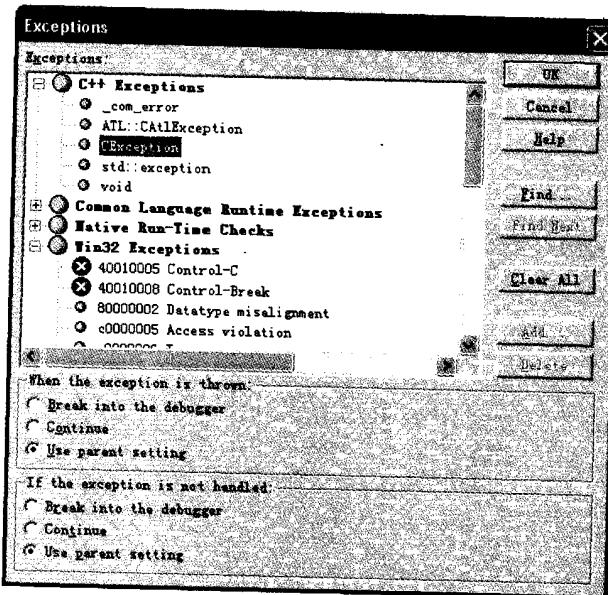


图 30-12 VS7 的异常处理选项对话框

Visual Studio 2005 (VS8) 集成调试器的配置对话框 (图 30-13) 外观上又有了很大的变化, 对于每种异常, 它提供了两个复选框, 分别称为 Thrown 和 User-unhandled, 前者的含义是对于第一轮机会是否中断给用户, 后者的含义是对于第二轮机会是否中断给用户。

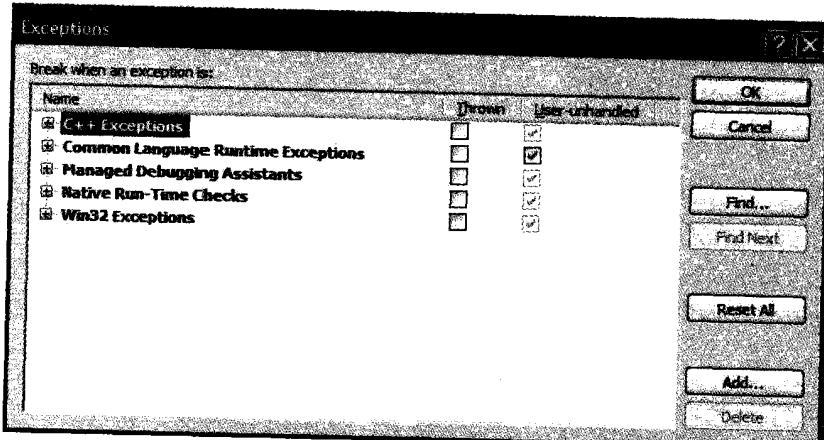


图 30-13 Visual Studio 2005 的异常处理选项对话框

图 30-14 是 WinDBG 的调试事件配置对话框。WinDBG 从 2.0 版本开始便一直使用这个对话框界面, 保持了很好的稳定性, 不像 VS 调试器那样几乎每个版本都各不一样。

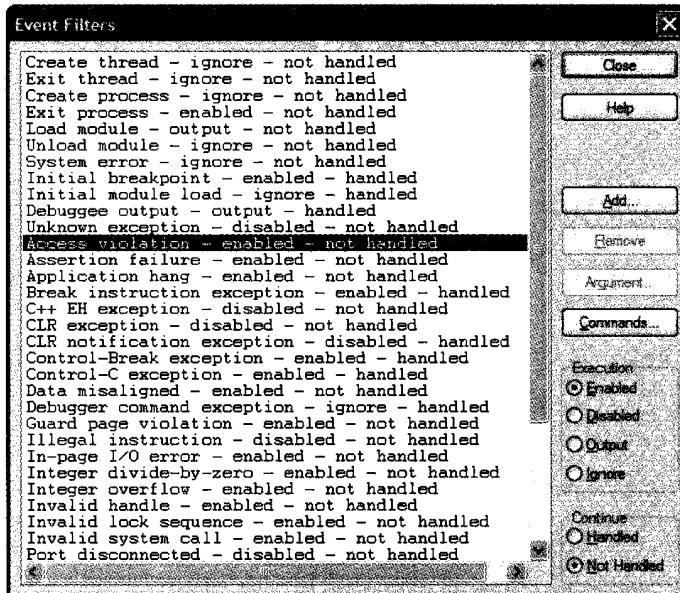


图 30-14 WinDBG 的异常处理选项对话框

首先，右下角 Execution 组的四个单选按钮用来配置中断选项，它们的含义如下。

- **Enabled:** 收到该事件后便中断给用户。对于异常事件，意味着两轮机会都中断给用户；对于其他调试事件，意味着收到时便中断。
- **Disabled:** 对于异常事件，第二轮机会时中断给用户，第一轮不中断。对于其他调试事件，不中断到命令模式。
- **Output:** 输出信息通知用户。
- **Ignore:** 忽略这个事件。

Continue 组的两个单选按钮用于配置返回给系统的异常事件处理状态，只适用于异常类事件，而且它是针对第一轮处理机会的。如果选择 Handled，那么便返回已经处理异常，否则返回没有处理异常。对于大多数异常，默认是返回没有处理异常。对于第二轮异常，WinDBG 默认返回已经处理，但是调试时，可以使用 gn 命令强制返回没有处理。

由此可见，WinDBG 既允许配置中断选项，也允许配置继续选项，比 VS 调试器提供了更大的灵活性。另外，WinDBG 允许为每个事件定义关联命令，点击对话框中的 Commands 按钮便弹出一个对话框，允许用户输入一系列命令，对于异常事件，可以为每一轮机会输入一组命令。

对于大多数调试事件，可以使用 Arguments 按钮指定一个参数，并设置满足这个参数条件时的配置选项。举例来说，在图 30-14 的列表中选择加载模块事件（Load module），然后点击 Arguments 按钮，在弹出的对话框中输入 kernel32.dll 后关闭，然后

在 Exception 组中选择 Enabled, 这样当被调试程序再加载 kernel32.dll 时便会中断到命令模式, 当加载其他模块时不会中断, 因为加载模块事件的默认处理方式是忽略 (Ignore)。

除了使用图形界面, 也可以使用命令来配置 WinDBG 的异常处理选项, 其语法为:

```
sx{e|d|i|n} [-c "Cmd1"] [-c2 "Cmd2"] [-h] {Exception|Event|*}
```

其中的 e|d|i|n 对应于图形界面的 Enabled、Disabled、Output 和 Ignore, -c 和 -c2 分别用来定义第一轮机会和第二轮机会的关联命令, Exception|Event 用来指定要设置的调试事件 (异常或者其他事件)。WinDBG 为常用的调试事件定义了一个简单的代码, 比如非法访问异常的代码是 av, 除零异常的代码是 dz, 线程退出的代码是 et 等等, 详见 WinDBG 帮助文件中关于 Controlling Exceptions and Events 的介绍。

如果指定了-h 开关, 那么这条命令就是用来设置处理状态 (handling status), 而不是中断状态。此时, sxe 命令设置的处理状态是 Handled, 其他三条命令都是设置为 Not Handled。

使用 sxr 命令可以将所有事件处理选项恢复为默认值, 直接输入 sx 命令可以列出各个事件的代码和目前的设置状态。

30.9.4 GH 和 GN 命令

当因为发生异常而中断到调试器中时, 如果使用 g 命令恢复调试目标执行, 那么, 调试器将使用上面介绍的配置来决定返回给系统的处理状态。如果调试人员希望返回与设置不同的状态, 那么可以使用 GH 或者 GN 命令。GH 用来强制返回已经处理 (Handled), GN 用来强制返回没有处理 (Not Handled), 不论关于该异常的设置如何。

30.9.5 实验

下面通过一个实验来加深大家的理解, 我们使用第 28 章曾经使用过的 dbgee 小程序作为调试目标, 它的主要代码如下:

```
1 int _tmain(int argc, _TCHAR* argv[])
2 {
3     if(argc==1)
4     {
5         *(int *)0=1;
6         printf("test\n");
7     }
8     return 0;
9 }
```

启动 WinDBG 然后通过 Open Executables 打开 dbgee.exe。WinDBG 成功创建进程和创建调试会话后, 会因为初始断点和中断给用户。命令窗口显示如下信息:

```
(9fc.db0): Break instruction exception - code 80000003 (first chance)
```

其中 80000003 是断点异常的异常代码，括号中的 first chance 代表这是第一轮处理机会。这说明对于断点异常，WinDBG 收到第一轮通知时，便中断给用户。观察图 30-14 所示的对话框，可以看到这个异常（Break instruction exception）的处理选项是 enabled – handled，即第一轮机会便中断，并返回已经处理这个异常。接下来依次执行如下步骤：

执行 sxe av 命令设置对于非法访问异常（av）第一轮便中断。

执行 sxd -h av 命令设置对于非法访问异常的处理状态是不处理。

执行 sx 命令，确保关于访问异常的设置是如下内容：

```
av - Access violation - break - not handled
```

输入 g 命令让调试目标执行。

因为源代码第 5 行故意设计了一个空指针访问，所以程序执行到这里时会导致一个非法访问异常。调试器收到这个调试事件后会检查异常设置，发现这个异常的设置是 Enable（第一轮便中断给用户）便进入命令模式，并显示如下内容：

```
(1574.14f8): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.

...
dbgee!wmain+0x24:
004113b4 c7050000000001000000 mov dword ptr ds:[0],1  ds:0023:00000000=???????
```

第一行的 c0000005 是非法访问异常的异常代码，(first chance) 代表这是第一轮处理机会。第 2、3 行提示我们系统还没有执行程序中的异常处理器，对于某些程序来说，异常可能是故意抛出的，可能属于期望的情况。

接下来，执行 gh 命令强制让调试器返回已经处理了这个异常。系统收到这个回复后会停止分发异常（因为调试器声称已经处理了异常），恢复调试目标继续执行，但由于异常条件仍在，所以还会产生异常，于是再次分发，WinDBG 再次中断到命令模式，并显示上面的信息。

而后执行 g 命令。因为对这个异常的设置是 Not Handled，所以调试器执行 g 命令时会向系统返回没有处理这轮异常，所以系统会继续分发这个异常，寻找程序中的异常处理器（VEH、SEH 等）。因为上面的代码没有任何异常处理器，所以系统会执行默认的异常处理器（参见第 12 章），执行系统的 UnhandledExceptionFilter 函数（位于 kernel32.dll 中）。UnhandledExceptionFilter 函数会判断当前程序是否在被调试，如果不在被调试，那么便启动应用程序错误对话框，通知用户终止程序。如果在被调试，那么 UnhandledExceptionFilter 会返回 EXCEPTION_CONTINUE_SEARCH，这会导致系统继续分发这个异常，即进入异常的第二轮分发。对于第二轮机会，系统仍然是先分给调试器，WinDBG 收到通知后会中断到命令模式，并显示如下信息：

```
(1574.14f8): Access violation - code c0000005 (!!! second chance !!!)
...
dbgee!wmain+0x24:
004113b4 c7050000000001000000 mov dword ptr ds:[0],1  ds:0023:00000000=???????
```

输入 g 命令让目标继续执行, WinDBG 会使用默认的处理选项(已经处理)返回给系统。这会导致系统恢复执行目标, 重新产生异常, WinDBG 又得到第一轮机会, 输入 g 后, WinDBG 又得到第二轮处理机会。如果在得到第二轮初处理机会时执行 gn 命令强制调试器告诉系统没有处理第二轮异常, 那么调试目标会突然消失, 因为系统将其强制终止了。

本节介绍了调试事件的有关内容, 这些内容是本书关于异常这一主题的最后一部分。理解这部分内容需要前面的基础, 建议读者在阅读本节时遇到不清楚的内容便返回到前面的章节, 复习一下前面的内容。

30.10 控制调试目标

有些软件问题通过观察症状然后审查代码就可以发现根源并找到解决方案, 但更多的问题是需要跟踪程序的执行过程才能摸清来龙去脉发现症结所在的。让被调试程序以可控的方式运行是设计调试器的最基本目标, 也是调试器的威力所在。所谓交互式调试, 其主要内涵就是可以与被调试程序互动, 可以让其停下来接受观察, 观察好了可以让其继续运行一段时间, 然后再停下来观察……控制调试目标(被调试程序)是调试器的一个核心任务, 其宗旨就是使调试目标始终处于调试器的控制之下, 让调试人员可以随心所欲地控制程序的执行状态。WinDBG 提供了强大的机制和丰富的命令来控制调试目标, 本节开始的四节(第 30.10 节~第 30.13 节)将分类介绍这些命令和有关的使用技巧。

30.10.1 初始断点

当调试一个新创建的进程(用户态目标)时, 为了让调试人员可以尽早地分析调试目标, Windows 操作系统的进程加载器加入了特别的调试支持, 在完成最基本的用户态初始化工作后, 系统的模块加载函数就会主动执行断点指令, 触发断点, 让调试目标中断到调试器中。这个断点被称为初始断点(Initial Breakpoint)。

在我们使用 WinDBG 打开一个程序文件(Open Executable)后, 很快 WinDBG 便会显示收到断点事件, 这个断点便是位于 NTDLL 中的 LdrpInitializeProcess 函数调用 DbgBreakPoint 而触发的初始断点。我们知道, 当创建一个新进程时, 很多早期的创建工作(创建进程对象、进程空间、建立初始线程、通知子系统等)都是在父进程的环境下完成的。初始线程真正在新进程环境下执行是从内核态的 KiThreadStartup 开始的。KiThreadStartup 将线程的 IRQL(中断级别)降到 APC 级别后调用 PspUserThreadStartup 来为线程在用户态执行作准备。因为 PspUserThreadStartup 仍然是在内核态的, 为了可以执行用户态的加载工作, 它初始化了一个对用户态代码的异步过程调用(APC), 并插入到 APC 队列中, 这个 APC 便是调用 NTDLL.DLL 中的 LdrpInitialize 函数。因此

可以说，`LdrpInitialize` 函数是一个新进程的初始线程开始在用户态执行的最早代码。`LdrpInitialize` 在初始化加载器和读取执行选项后，调用 `LdrpInitializeProcess` 函数。`LdrpInitializeProcess` 函数的一个主要任务就是加载 EXE 文件所依赖的动态链接库。在加载每个 DLL 后，`LdrpInitializeProcess` 检查当前进程是否在被调试（PEB 的 `BeingDebugged` 字段），如果是，则调用 `DbgBreakPoint` 通知调试器。注意此时尚未调用每个 DLL 的 `DllMain` 函数。当 `LdrpInitialize` 执行完毕后，`KiUserApcDispatcher` 调用 `ZwContinue` 返回到内核态的 `PspUserThreadStartup` 函数中。接下来，`PspUserThreadStartup` 函数把线程的 IRQL 降低到 0 (PASSIVE)。而后，系统开始执行已经放在线程上下文中的进程启动函数 `BaseProcessStart`，后者调用程序的入口函数使应用程序开始运行。

当将 WinDBG 附加到一个已经运行的进程时，WinDBG 默认也会通过在目标进程创建一个远程线程来触发一个初始断点，这个断点发生在新创建的线程上下文中，其栈调用通常为：

```
0:001> kn
# ChildEBP RetAddr
00 00cdfffc8 7c9507a8 ntdll!DbgBreakPoint
01 00cdfff4 00000000 ntdll!DbgUiRemoteBreakin+0x2d
```

值得注意的是，这个线程并不是目标进程的本来线程，它是调试器创建的。当我们恢复目标执行时，这个线程也会立刻退出。

在 WinDBG 的命令行中加入-g 开关，可以让其忽略或者不发起初始断点，也就是对于调试新进程的时候，当接收到初始断点事件时，不中断给用户；当附加到已经创建的进程时，不再发起远程线程来触发断点。

另外，要说明的是，初始断点并不是调试器可以得到的最早控制机会，进程创建事件和 EXE 模块的加载事件都比初始断点的时间还要早。但对于跟踪和分析程序的入口函数或者 DLL 的入口函数，初始断点的中断时机是足够早的了。

30.10.2 俘获调试目标

初始断点为我们分析被调试程序提供了一个初始机会，通常设置了断点或者做基本的准备工作后，我们便恢复目标继续执行。如果希望把运行的调试目标再次中断到调试器中，那么可以使用如下方法。

- 在调试器界面中选择中断命令（`Debug > Break`）或者使用 `Ctrl+Break` 热键。
- 对于有窗口界面的程序，将被调试程序窗口切换到前台，然后按 `F12` 热键（参见 10.6.5）。
- 如果已经设置了断点，或者在代码中加入了触发异常的代码，那么可以执行相应的操作，让程序触发断点或者异常，使其中断到调试器。

因为第一种方法使用最多，所以我们介绍一下它的工作细节。清单 30-7 显示了 WinDBG 的 UI 线程收到 Break 命令后的工作过程。

清单 30-7 调试器 UI 线程处理 Break 命令的过程

```
0:000> kn
# ChildEBP RetAddr
00 0006ce54 7c93401e ntdll!ZwCreateThread //创建远程线程
01 0006d1bc 7c9507ff ntdll!RtlCreateUserThread+0xdc
02 0006d1fc 7c85a383 ntdll!DbgUiIssueRemoteBreakin+0x26 //发起远程中断动作
03 0006d208 0229c11b kernel32!DebugBreakProcess+0xd //Windows 的调试 API
04 0006d230 02225a4c dbgeng!LiveUserDebugServices::RequestBreakIn+0x1b //服务层
05 0006d24c 020c7126 dbgeng!LiveUserTargetInfo::RequestBreakIn+0x5c //目标层
06 0006d258 0103cb21 dbgeng!DebugClient::SetInterrupt+0xa6 //调试引擎的接口函数
07 0006ddf4 7e418724 WinDBG!FrameWndProc+0x13f1 //窗口过程, 以下省略
```

依照函数调用的先后顺序（从下至上），栈帧#7 是窗口的过程函数，它收到 Break 命令后通过全局变量 g_DbClient 调用 SetInterrupt 方法，这个方法的用途就是让调试器进入命令模式，其函数原型如下：

```
HRESULT IDebugControl::SetInterrupt( IN ULONG Flags );
```

其中 Flags 参数可以包含如下标志：

DEBUG_INTERRUPT_PASSIVE (1): 向调试引擎注册用户希望中断到命令模式，但是不强制。其函数内部是将 dbgeng!g_UserInterruptCount 加 1，将 dbgeng!g_EngStatus 设置为 0x1005。

DEBUG_INTERRUPT_EXIT (2): 设置 dbgeng!g_EngStatus 的 0x800 位，让调试器引擎取消等待调试事件，强制返回。通常这会导致调试器没有中断调试目标就进入到命令模式，因为没有合适的进程和线程上下文，所以命令提示符会包含多个问号（图 30-15）。使用这种方法中断后，大多数控制调试目标执行的命令都无法执行。

DEBUG_INTERRUPT_ACTIVE (0): 判断全局变量 dbgeng!g_CmdState，如果当前调试器没有处于命令模式，那么要求调试目标中断到调试器以进入命令模式，如果调试器已经在命令模式，那么只是简单地递增 dbgeng!g_UserInterruptCount 变量。

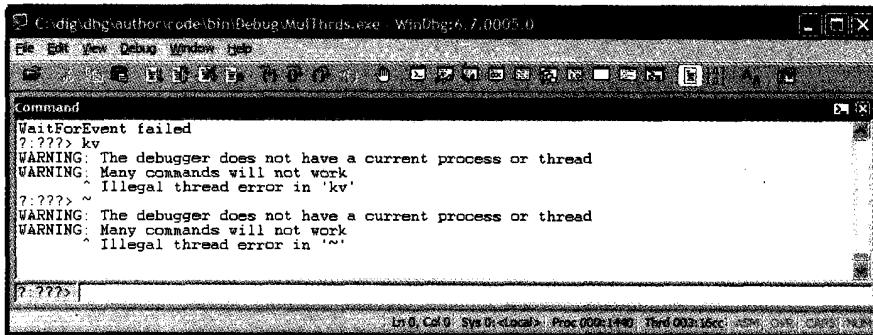


图 30-15 调试器被强制进入命令模式

在清单 30-7 中，UI 线程是使用参数 0 来调用 SetInterrupt 方法的，也就是要求调试目标中断到调试器中。栈帧 #5 和栈帧 #4 分别是调用调试目标类（LiveUserTargetInfo）和调试服务类（LiveUserDebugServices）的 RequestBreakIn 方法，将中断请求层层下传。栈帧 #3 是调用操作系统的调试 API，DebugBreakProcess。栈帧 #2~栈帧 #0 显示了 DebugBreakProcess API 的内部工作过程，栈帧 #2 是调用 NTDLL 中的 DbgUiIssueRemoteBreakin，栈帧 #1 和栈帧 #0 是调用线程创建函数在调试目标进程中创建远程线程。这个远程线程的线程函数是 NTDLL 中的 DbgUiRemoteBreakin 函数，它内部的代码很少，只是简单地调用 DbgBreakPoint API 来执行断点指令，产生一个断点异常。当目标进程中断到调试器中时，当前线程就是调试器创建的这个远程线程。

上面讲的是正常情况，也就是创建一个远程线程，这个远程线程一执行就产生一个断点异常，触发调试目标中断到调试器中。但是以上过程有时可能失败，比如远程线程创建后还没有执行断点指令就被挂起了，这时 WinDBG 就收不到断点事件了。对于这种情况，WinDBG 等待一段时间后，会显示如下提示信息：

```
Break-in sent, waiting 30 seconds...
```

再等待 30 秒后，WinDBG 会“人工合成”（Synthesize）一个代码为 0x80000007 的异常事件，这个事件的调试引擎函数名为 SynthesizeWakeEvent，调用过程如下：

```
00 015dff10 020ceacf dbgeng!SynthesizeWakeEvent+0x4b
01 015dff34 020cee9e dbgeng!WaitForAnyTarget+0x5f
02 015dff80 020cf110 dbgeng!RawWaitForEvent+0x2ae
03 015dff98 0102aadf dbgeng!DebugClient::WaitForEvent+0xb0
04 015dffb4 7c80b6a3 WinDBG!EngineLoop+0x13f
05 015dffec 00000000 kernel32!BaseThreadStart+0x37
```

这个合成事件触发调试器的事件等待函数返回，并开始处理这个事件，这会引发 dbgeng!SuspendExecution 函数被调用，这个函数会依次挂起调试目标中的所有线程，使调试目标被中断，最后调试器进入到命令模式中并显示如下信息：

```
(1440.1554): Wake debugger - code 80000007 (first chance)
```

也就是说，WinDBG 会先使用远程线程来中断调试目标，如果超时，那么它会使用挂起方式来将被调试进程强制俘获到调试器中。

30.10.3 继续运行

WinDBG 提供了很多命令来让调试目标恢复继续运行，最常用的就是 g (go) 命令。热键 F5 和 Debug 菜单的 Go 菜单项对应的就是 g 命令，G 命令的一般形式为：

```
g[a] [= StartAddress] [BreakAddress ... [; BreakCommands]]
```

其中 StartAddress 用来指定恢复执行的起始地址，默认为当前位置。BreakAddress 用来指定一个断点地址，BreakCommands 用来指定断点命中后所执行的命令。开关 a 只有在使用 BreakAddress 设置断点时才有用，使用 a 将断点设置为硬件断点，如果没有 a，则设置为软件断点。

如果指定了断点地址 (BreakAddress)，那么 WinDBG 会设置一个隐藏的断点，然后恢复目标执行，当执行到这个断点时，WinDBG 中断并自动删除这个断点。当使用汇编或者源代码窗口时，可以使用“运行到光标处”(Run to Cursor) 命令 (Ctrl+F10 或者菜单 Debug > Run to Cursor) 来执行到光标所在位置。这其实也就是使用 g 命令加断点地址来实现的。当发出这个命令时，命令信息区会显示出对应的 g 命令，如 0:000>g 0x`4113c0。

对于因为异常而中断到调试器的情况，当恢复目标运行时调试器需要回复系统是否处理了这个异常，如果是使用 g 命令，那么 WinDBG 会使用关于所发生异常的配置信息来决定回复内容。为了提高灵活性，用户可以使用 gn 或者 gh 命令来指定要回复给系统的异常处理决定，如果要回复已经处理，那么就用 gh (Go with Exception Handled) 命令；否则就使用 gn (Go with Exception Not Handled) 命令。这两条命令的语法和其他特征与 g 命令完全一样。

除了以上介绍的 g、gh 和 gn 命令，gu (go up) 命令用来执行到上一级函数，即执行完当前函数，返回到上一级函数，另外 gc 命令用在使用条件断点的情况下，我们将在 30.12.3 节介绍，Gu 和 gc 命令都没有参数。

30.11 单步执行

单步执行是历史最悠久的调试机制之一，可以追溯到大型机时代，例如 UNIVAC I 上的 IOS 开关就是用来切换到单步执行模式，让处理器每次执行一条指令。通过单步执行可以更好地理解程序的执行流程和数据的变化情况。在软件日益复杂和庞大的今天，单步跟踪仍十分重要。但是跟踪整个程序通常都是不现实的，可行的方法是只跟踪最关心的代码，让不关心的部分全速执行。WinDBG 调试器为实现这一目标提供了丰富的命令，让调试人员可以快速跳过不关心的部分，抵达要跟踪的位置。

30.11.1 概览

首先，根据当前是否处于源代码模式 (Source Mode)，单步跟踪分为源代码级的单步和汇编指令一级的单步。选中 Debug 菜单的 Source Code 菜单项或者执行 l+t 命令可以进入源代码模式，反选 Source Code 菜单项或者执行 l-t 命令可以退出源代码模式，进入汇编模式。在汇编模式，每次单步执行执行一条汇编指令。在源代码模式时，每次单步执行源代码的一行。

如果当前的指令或者代码行包含函数调用，那么有两种选择，一种是跟踪进入要调用的函数，另一种是忽略要调用函数的执行过程，让其执行完毕后再停下来。前一种方式通常称为单步进入 (Step Into)，后一种方式称为单步越过 (Step Over)。在 WinDBG 中，使用单步 (Step) 一词来指代前一种方式，对应的命令是 p；使用跟

踪 (Trace) 一词指代后一种方式，对应的命令是 t。如果当前行不包含函数调用，那么 p 命令和 t 命令的作用是一样的。

从实现的角度来讲，汇编一级的单步执行 (p)、跟踪进入 (Trace Into) 和针对非函数调用执令执行 t 命令都是依赖 CPU 的陷阱机制来实现的，对于 x86 CPU，也就是设置标志寄存器的 T 标志。举例来说，当我们在 WinDBG 中发出 t 命令后，工作线程会解析此命令 (ParseStepTrace)，然后调用 SetExecStepTrace 和 SetNextStepTraceState 函数，这两个函数会将用户命令转化并设置到内部对象、变量和线程的上下文结构中。在设置完毕后，调试引擎便报告此命令执行完毕，于是函数层层返回，直到 ProcessEngineCommands 返回到 EngineLoop。EngineLoop 继续执行，调用 DebugClient::WaitForEvent 等待下一个调试事件，在等待前先要恢复目标运行，在恢复目标执行时 (ResumeExecution)，调试引擎会将线程上下文通过 SetThreadContext API 设置给系统，清单 30-8 所示的栈回溯记录了调试会话线程执行以上函数的过程。

清单 30-8 调试会话线程设置线程上下文的过程

```
0:004> kn
# ChildEBP RetAddr
00 0144f350 0229ba85 kernel32!SetThreadContext //系统 API
01 0144f368 020f2de9 dbgeng!LiveUserDebugServices::SetContext+0x45 //服务层
02 0144f38c 020f0bf8 dbgeng!LiveUserTargetInfo::SetTargetContext+0x59
03 0144fe24 021244fe dbgeng!TargetInfo::SetContext+0xb8
04 0144fe40 0217257c dbgeng!MachineInfo::UdSetContext+0x3e
05 0144fe54 0221bbfa dbgeng!MachineInfo::SetContext+0x15c //架构信息类
06 0144fe6c 0221c82d dbgeng!TargetInfo::ChangeRegContext+0xca
07 0144fe84 021311f8 dbgeng!TargetInfo::PrepareForExecution+0x1d //目标层
08 0144fe94 0213148f dbgeng!PrepareOrFlushPerExecution+0x38
09 0144fea4 02132098 dbgeng!ResumeExecution+0x2f //恢复执行
0a 0144ff28 02130fe8 dbgeng!PrepareForExecution+0x618 //准备执行
0b 0144ff3c 020cec09 dbgeng!PrepareForWait+0x28 //准备等待
0c 0144ff80 020cf110 dbgeng!RawWaitForEvent+0x19 //调试引擎的内部函数
0d 0144ff98 0102aadf DebugClient::WaitForEvent+0xb0 //等待调试事件
0e 0144fffb 7c80b6a3 WinDBG!EngineLoop+0x13f //调试会话循环
0f 0144ffec 00000000 kernel32!BaseThreadStart+0x37 //系统的线程启动函数
```

传递给 SetThreadContext 函数的参数是一个 CONTEXT 结构，使用 dt 命令可以观察：

```
0:004> dt _CONTEXT 01473e00
+0x0c0 EFlags : 0x302
```

其中 EFlags 就是标志寄存器字段，将它的值翻译为二进制：

```
0:004> .formats 0x302
Binary: 00000000 00000000 00000011 00000010
```

位 1 是保留位，永远为 1，位 8 就是跟踪标志 (Trap Flag)，1 代表单步执行。位 9 的另一个 1 的含义是启用中断 (Interrupt Enable Flag)，即 IF 位。

可见，对于 t 命令，调试器是通过设置标志寄存器来实现的。如果当前指令是 CALL 指令，而且执行 p 命令，那么 WinDBG 需要一步执行好要调用的函数，即单步越过，这时尽管命令的执行过程仍然与清单 30-8 所示的基本一致，但是设置的线程上下文中的 EFlags 值为：

```
+0x0c0 EFlags      : 0x246
```

也就是 TF 标志没有设置。如果观察调试引擎的 ProcessDebugEvent 函数收到的调试事件，我们会发现异常代码是 80000003，也就是断点异常，不是 t 命令所触发的单步异常（异常代码是 80000004）。事实上，针对 CALL 指令的单步越过命令是通过在 CALL 指令的下一条指令处设置一个软件断点来实现的。

源代码一级的单步执行是通过多次设置陷阱标志，也就是多次汇编一级的单步执行而实现的。因为篇幅关系，不再详细讨论，感兴趣的读者可以使用上面的方法自己来探索。

除了基本功能外，可以通过向 p 和 t 命令附带参数来使用它们的附加功能，这两个命令的完整语法为：

```
p|t [r] [= StartAddress] [Count] ["Command"]
```

其中 r 的用处是禁止自动显示寄存器内容，默认情况下每次单步执行后，WinDBG 会自动显示各个寄存器的值，如果不想显示，则使用 r 开关，r 与命令之间的空格可有可无。

默认情况下，调试器总是让目标程序从当前位置开始单步执行，但是也可以通过等号 (=) 来指定一个新的起始地址，让程序从这个地址开始执行。需要注意的是，如果指定的地址跳过了调整栈的代码，那么栈就会失去平衡，目标程序很快会出现严重错误，所以使用这个功能时应该特别慎重。

可选的参数 [Count] 用来指定要单步执行的次数。如果 Count 大于 1，那么执行好一次单步并更新显示后，WinDBG 会再发送一次单步命令，直到达到指定的次数。例如：t 2 会单步执行两次，每次执行一条指令（汇编模式）或者源代码的一行（源代码模式）。

["Command"] 参数用来指定每次单步执行后要执行的命令。例如：p “kb”会在单步执行后自动执行 kb 命令。

30.11.2 单步执行到指定地址

WinDBG 提供了 pa 和 ta 命令用来执行到指定的代码地址，其命令格式为：

```
pa|ta [r] [= StartAddress] StopAddress
```

其中 pa 是 Step to Address 的缩写，即单步执行到 StopAddress 参数所代表地址处的指令，如果中间有函数调用，那么不进入所调用的函数。在执行过程中，WinDBG 会显示程序执行的每一步，其效果相当于反复执行 p 命令。Ta 命令与 pa 非常相似，只不过遇到函数调用时会进入到函数中，而不是越过，这和 t 命令与 p 命令的差异是一样的。

因为伪寄存器 \$ra 总是代表当前函数的返回地址（return address），因此可以使用 pa 或者 ta 命令加上 @\$ra 来“步出”当前函数，也就是从当前位置反复单步直到返回到上一级函数，其效果相当于 gu 命令（执行到上一层函数）。

如果在到达目标地址前遇到断点，那么 WinDBG 会报告断点，这个命令也就从此被中断。如果在到达目标地址前程序发生异常，那么 pa 或者 ta 命令也可能被中断。举例来说，假设当前的程序指针（EIP）等于 004113b4，相关的指令如下：

```
004113b4 c7050000000001000000 mov dword ptr ds:[0],1 ds:0023:00000000=????????  
004113be 8bf4          mov     esi,esp  
004113c0 683c564100    push    offset dbgee+0x1563c (0041563c)
```

如果这时发出 par 004113c0 命令，那么这个命令会因为第 1 条指令导致异常而中断，并不会单步执行到第 3 条指令。

30.11.3 单步执行到下一个函数调用

pc 和 tc 命令用来单步执行到下一个函数调用指令（CALL），其格式为：

```
pc|tc [r] [= StartAddress] [Count]
```

首先，它们都是让调试目标从当前地址或者 StartAddress 指定的地址恢复执行，直到遇到函数调用指令时停下来，Count 参数用来指定遇到的函数调用指令个数，默认为 1。

这两个命令的差异依然与 p 指令的和 t 指令的差异一样，对于 CALL 指令使用 tc 时会单步进入所调用的函数，使用 pc 命令会一次执行完 CALL 指令。如果当前指令不是函数调用指令，而且 COUNT 参数为 1，那么 pc 与 tc 是等价的。

从实现角度来看，WinDBG 依然是反复单步执行，每次收到调试事件后，判断下一条指令是否是函数调用指令，如果不是，就重新设置单步标志，然后继续执行，如果是，那么就停下来，命令执行完毕。当处理 CALL 指令时（当前指令是 CALL 或者 Count 参数大于 1 时中途遇到 CALL），pc 命令需要使用在下一条指令设置断点的方法。

30.11.4 单步执行到下一分支

在第 2 篇介绍 CPU 的调试支持时，我们介绍了 CPU 的分支记录和监视功能。利用这一功能可以实现分支到分支的单步执行，即一次执行到下一条分支指令。WinDBG 的 tb 命令就是利用这一机制而实现的。

因为使用了 CPU 的硬件支持，所以 tb 命令与 tc 和 pc 这样的反复多次单步执行不同，它设置好标志寄存器和 MSR 寄存器后，便让目标程序恢复运行，然后当 CPU 执行到分支指令时，报告异常停下来。从这个意义上来说，tb 命令要比 tc 和 pc 更高效。Tb 命令的语法与 tc 和 pc 一样：

```
tb [r] [= StartAddress] [Count]
```

对于安腾系统和 x64 系统，tb 命令既可以用在内核调试，也可以用在用户态调试。但是在 x86 平台上这个命令只能用在内核调试中，为了克服这一局限，可以使用 ph 和 th 命令：

```
ph|th [r] [= StartAddress] [Count]
```

这两个命令分别用来单步执行或者追踪到下一分支指令，处理 CALL 指令的方式不同是它们的唯一差异。

30.11.5 追踪并监视

如果我们想了解一个函数的执行路径和它调用了哪些其他函数，以及每个函数包含了多少条指令，但我们又不想一步步地跟踪执行，那么可以使用 wt 命令让它帮我们跟踪执行并生成一份清单 30-9 那样的报告。

清单 30-9 wt 命令产生的追踪报告

```

1      0:000> wt
2      Tracing wtee!main to return address 00401100
3          6    0 [ 0] wtee!main
4          2    0 [ 1] wtee!GetRandom
5          5    0 [ 2] kernel32!GetTickCount
6          4    5 [ 1] wtee!GetRandom
7          12   0 [ 2] kernel32!GetVersion
8          11   17 [ 1] wtee!GetRandom
9          8    28 [ 0] wtee!main
10
11     36 instructions were executed in 35 events (0 from other threads)
12
13     Function Name                      Invocations MinInst MaxInst AvgInst
14     kernel32!GetTickCount               1           5       5       5
15     kernel32!GetVersion                1           12      12      12
16     wtee!GetRandom                   1           11      11      11
17     wtee!main                         1           8       8       8
18
19     0 system calls were executed
20
21     eax=823e9fa0 ebx=7ffd3000 ecx=00000064 edx=00000a28 esi=01caf764 edi=01caf6f2
22     eip=00401100 esp=0012ff88 ebp=0012ffc0 iopl=0         nv up ei pl nz na pe nc
23     cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000             efl=00000206
24     wtee!mainCRTStartup+0xb4:
25     00401100 83c40c      add    esp, 0Ch

```

上面的结果是在本章的示例程序 wtee 的 main 函数入口处执行 wt 命令产生的。如果不是在函数开始处执行 wt 命令，那么它的效果相当于 p 命令，只是单步执行一次。

可以把 wt 命令的结果分成 6 个部分。第一部分是标题（第 2 行），显示了所追踪的函数名 wtee!main 和追踪的结束地址，即函数的返回地址（00401100）。

第二部分是详细的执行情况表，分成若干行，每一行用来描述一段执行路线，每次函数变换会重新开始一行。比如，第 3 行描述的是从 main 函数的入口到调用 GetRandom 方法这一段，第 4 行描述的是从 GetRandom 方法的入口到调用 GetTickCount API 这一段，第 5 行描述的是在 GetTickCount API 执行的过程，第 6 行描述的是从 GetTickCount 函数返回后到调用 GetVersion API 的部分，依此类推。每一行包含如下 4 列。

第 1 列为本行所描述函数已经执行的指令数。对于不调用其他函数的函数，本列

的数字就是这一次执行这个函数一共执行的指令数。以第5行为例，数字5代表执行GetTickCount函数一共执行了5条指令，反汇编GetTickCount可以发现它确实是共有5条指令。对于内部调用其他函数的函数，那么这一列的值是这个函数已经执行的总指令数。以GetRandom函数为例，它调用了两个其他函数，报告中共有三行是关于这个函数的，即4、6、8行。第4行的数字2代表从GetRandom函数入口到调用GetTickCount函数这一段共有2条指令，包括CALL指令，即清单30-10中的2~3行中的指令。第6行的4代表调用GetVersion时GetRandom函数一共执行了4条指令，包括第4行记录的2条指令。类似的，第8行的数字11代表已经执行了11条指令，因为这是关于这个函数的最后一行，所以这也是本次执行GetRandom函数一共执行的指令数，观察清单30-10，这恰好就是这个函数的总指令数，因为这个函数中没有任何条件跳转。

清单30-10 GetRandom函数的反汇编

1	wtee!GetRandom:		
2	00401000 56	push	esi
3	00401001 ff1504604000	call	dword ptr [wtee!_imp__GetTickCount (00406004)]
4	00401007 8bf0	mov	esi,eax
5	00401009 ff1500604000	call	dword ptr [wtee!_imp__GetVersion (00406000)]
6	0040100f 0fafc6	imul	eax,esi
7	00401012 8b4c2408	mov	ecx,dword ptr [esp+8]
8	00401016 5e	pop	esi
9	00401017 0fafc1	imul	eax,ecx
10	0040101a 0fafc1	imul	eax,ecx
11	0040101d d1e0	shl	eax,1
12	0040101f c3	ret	

第2列用来显示本行所描述函数调用其他函数所执行的总指令数。以第6行为例，5代表GetRandom函数调用其他函数执行了5条指令，也就是调用GetTickCount所执行的5条指令，第8行的17代表GetRandom函数调用其他函数执行了17条指令，因为又调用了GetVersion执行了12条指令。

第3列用来表示函数调用深度，被追踪的函数的深度为0，如第3行和最后一行，每进入一个函数深度加1，每返回一次，深度减1。

第4列为函数名称，名称前的缩进长度与调用深度是成比例的。

下面我们继续看wt命令结果的第三部分，即第11行，它是对被追踪函数的简单归纳，共执行了36条指令，共处理了35次调试事件。

Wt结果的第四部分（13~17行）是按函数统计的指令表格，每一行是执行过的一个函数。表格的前两列分别是函数名称和调用次数（Invocations），后三列是这个函数每次执行时的最少指令数（MinInst）、最多指令数（MaxInst）和平均指令数（AvgInst）。因为本例中，所有函数都只执行一次，所以后三列的数字是相同的。

Wt结果的第五部分（19行）是调用系统服务的情况，本例中没有调用任何系统服务，所以显示为0次。

第六部分（21~25 行）是追踪执行完成后的寄存器状态和当前程序指针位置，显示的是函数返回到上一级函数后即将执行的下一条指令。

如果使用 `wt` 命令追踪复杂的函数或者位于顶层的函数，那么可能需要较长的时间，为了提高效率，可以通过命令选项来限制追踪的范围，比如使用`-l` 选项来指定追踪的深度，超过这一深度的函数调用可以一次执行。类似的，可以使用`-m` 开关来指定追踪的模块，使用`-i` 开关指定忽略的模块。如果在追踪的过程中遇到断点或者发生其他调试事件，那么 `wt` 命令会被中断而停止。

30.11.6 程序指针飞跃

在前面介绍的单步命令和 `g` 命令中，都可以指定起始执行地址，如果指定了起始地址，那么 WinDBG 便会把这个地址设置到线程上下文中的程序指针寄存器中。这样一来，当目标程序恢复执行时，系统就会把这个地址放入到真正的程序指针寄存器（EIP）中，于是 CPU 也就从这个地址开始执行了。这意味着，程序一下子“飞跃”到这个地址。这个飞跃不是目标程序的代码所定义的，而是我们通过调试器来操纵的，我们把这种特殊的跳转称为“程序指针飞跃”。

在某些情况下程序指针飞跃有利于软件调试。比如在调试时，如果我们不想执行某个函数调用，那么就可以通过程序指针飞跃来“飞过”这个函数。或者有时我们想跳过一条导致异常的指令，那么也可以从其下一条地址恢复执行而绕过它。但需要注意的是，这种飞跃是很危险的，如果所跨跃的代码包含栈操作，那么很容易导致栈不平衡，从而使目标程序无法继续运行。

除了通过在恢复程序执行的这些命令中指定起始地址外，也可以使用寄存器命令（r）直接修改程序指针寄存器来实现程序指针飞跃。

30.11.7 归纳

本节和上一节比较详细地介绍了用来控制目标程序执行的 WinDBG 命令，表 30-9 将这些命令归纳在了一起。

表 30-9 控制目标程序执行的命令一览

命令	含义	说明
p	Step	单步，如果遇到函数调用则一次执行完函数调用
t	Trace	追踪，如果遇到函数调用则进入被调用函数
pa	Step to Address	单步到指定地址，不进入子函数
ta	Trace to Address	追踪到指定地址，进入子函数
pc	Step to next Call	单步执行到下一个函数调用（CALL 指令）
tc	Trace to next Call	追踪执行到下一个函数调用（CALL 指令）
tb	Trace to next branch	追踪执行到下一条分支指令，只适用于内核调试
pt	Step to next return	单步执行到下一条函数返回指令

续表

命令	含义	说明
tt	Trace to next return	追踪执行到下一条函数返回指令
ph	Step to next branch	单步执行到下一条分支指令
th	Trace to next branch	追踪执行到下一条分支指令
wt	Trace and watch data	自动追踪函数执行过程
g	Go	恢复运行
gh	Go Handled	恢复运行, 告诉系统已经处理异常
gn	Go Not handled	恢复运行, 告诉系统没有处理异常
gu	Go Up	执行到本函数返回

以上命令除了 tb 命令在 x86 系统中只能在内核调试时使用外, 其他命令都是既可以在用户态调试, 又可以在内核态调试, 但是调试目标不能是转储文件, 必须是活动目标。

除了以上标准命令, WinDBG 还设计了一些元命令和扩展命令来辅助以上命令的使用, 比如在连续单步跟踪时, 如果觉得界面更新太频繁, 影响速度并且叫人眼花缭乱, 那么可以使用 .suspend_ui 命令暂时停止刷新信息窗口。

30.12 使用断点

断点是软件调试中最重要的技术之一。设置断点看似简单, 但是要把断点功能运用得恰到好处不容易, 必须认真学习不同类型断点的特征, 并仔细琢磨目标程序的执行流程, 然后才能在合理的位置设下断点, 使其在合适的时机命中。如果断点设置得不好, 那么可能频繁命中, 反复中断到调试器中, 浪费大量时间, 也可能根本不命中, 空等一场。有人把设置断点称为“埋”断点, 就像埋地雷一样, 我觉得非常形象生动。埋地雷要讲究位置和埋设时间, 以便刚好被敌人踩上(命中), 埋断点也一样, 要设置一个好的断点既要考虑位置, 又要考虑命中的时机。为了支持以上目标, WinDBG 提供了多种断点命令, 以满足不同的需要, 本节将分别进行介绍。

30.12.1 软件断点

我们在前面很多章节中提到了软件断点, 简单来说, 软件断点就是通过将指定位置的指令替换为断点指令(INT 3)而设置的断点。WinDBG 设计了 3 条命令来设置软件断点, 分别是 bp、bu 和 bm。其中 bp 是基本的而且是最常用的, 其命令格式如下:

```
bp [ID] [Options] [Address [Passes]] ["CommandString"]
```

其中 ID 用来指定断点编号, 如果不指定, 那么 WinDBG 会为其自动选择一个编号。Options 用来指定选项, 我们稍后再讨论。Address 用来指定断点的地址, 如果不指定, 那么默认使用当前程序指针所代表的地址。Passes 用来指定因为这个断点而中断到命令模式所需的穿越(命中)次数, 其默认值为 1, 也就是命中一次就中断到命令模式, 如果这个值大于 1, 那么当这个断点命中时, WinDBG 会把穿越计数递减 1, 然

后判断其值是否等于 0，如果大于 0，便直接让程序恢复执行，直到等于 0 时才进入命令模式中断给用户看。“CommandString”用来指定一组命令，当断点中断时，WinDBG 会自动执行这组命令，应该使用双引号将命令包围起来，多个命令用分号分隔。

例如，以下 bp 命令可以在 printf 函数的入口偏移 3 的地址处设置一个断点，当 CPU 第二次“穿越”这个位置时中断给用户，并自动执行 kv 和 da poi(ebp+8) 命令。

```
bp MSVCR80D!printf+3 2 "kv;da poi(ebp+8)"
```

其中 kv 命令用来显示函数调用序列，da poi(ebp+8) 用来显示 printf 的第一个参数所指定的字符串，之所以要对入口偏移 3 的位置设置断点，而不是入口，是因为要等入口处的栈帧建立代码执行好后，ebp+8 才能指向第一个参数，以下是 printf 函数入口处的前两条指令：

```
MSVCR80D!printf:  
1022e3f0 55          push    ebp  
1022e3f1 8bec        mov     ebp,esp
```

bu 命令用来设置一个延迟的以后再落实的断点，用于对尚未加载模块中的代码设置断点。当指定的模块被加载时，WinDBG 会真正落实（解决）这个断点。所以 bu 命令对于调试动态加载模块的入口函数或初始化代码特别有用。例如，当调试即插即用设备的驱动程序时，因为驱动程序是由操作系统的 I/O 管理器动态加载的，当我们发现它加载时，它的入口函数（DriverEntry）和初始化代码已经执行完了。对于这种情况，就可以使用 bu 命令在这个驱动加载前就对它的入口函数设置一个断点，即 bu MyDriver!DriverEntry。

bm 命令用来设置一批断点，相当于执行很多次 bp 或 bu 命令。比如以下命令对于 msrvcr80d 模块中的所有 print 开头的函数设置断点：

```
0:000> bm msrvcr80d!print*  
2: 1022e3f0 @!"MSVCR80D!printf"  
3: 1022e590 @!"MSVCR80D!printf_s"
```

因为在数据区设置软件断点会导致数据意外变化，所以 bm 命令在设置断点时会判断符号的类型，只对函数类型的符号设置断点。出于这个原因，bm 命令要求目标模块的调试符号有类型信息，这通常需要私有符号文件。如果对只有公共符号文件的模块使用 bm 命令，那么它会显示下面这样的错误信息：

```
0:000> bm ntdll!DbgPrint*  
No matching code symbols found, no breakpoints set.  
If you are using public symbols, switch to full or export symbols.
```

解决这个问题的一种简单方法是使用/a 开关，这个开关强制 bm 命令针对所有匹配的符号设置断点，不管这个符号对应的是数据还是代码。但这样做是有风险的，建议只有在确信所有符号都是函数时才使用。另一种更可靠的解决方式是按最后一句提示的建议使用完全的符号或 DLL 输出符号。DLL 输出符号尽管包含的符号较少，但是可以判断出是代码还是数据。

bm 和 **bu** 命令的格式与 **bp** 类似:

```
bu [ID] [Options] [Address [Passes]] ["CommandString"]
bm [Options] SymbolPattern [Passes] ["CommandString"]
```

其中 Options 可以为以下内容。

- **/1:** 如果指定此选项, 那么这个断点命中一次后会被自动从断点列表中删除, 这种断点被称为一次命中断点。
- **/p:** 这个开关只能用在内核调试中, /p 后跟一个 EPROCESS 结构的地址, 作用是只有断点事件发生在指定进程时才中断到命令模式, 也就是增加了一个过滤条件。
- **/t:** 与 /p 开关类似, 只能用在内核调试中, 用来指定一个ETHREAD 结构, 作用是只当断点事件发生在指定的线程时, 才中断到命令模式。
- **/c 和/C:** 这两个开关后面可以带一个数字, 用来指定中断给用户的最大函数调用深度和最小函数调用深度。举例来说, 使用命令 `bp /c5 msvcr80d!printf` 设置的断点, 只有当函数调用深度浅于 5 时才中断给用户。

以上命令选项对于下面介绍的硬件断点命令也是适用的。

30.12.2 硬件断点

硬件断点就是通过 CPU 的硬件寄存器设置的断点。硬件断点具有数量限制, 但是可以实现软件断点不具有的功能, 比如监视数据访问和 I/O 访问等(详见第 4 章)。

WinDBG 的 **ba** 命令用来设置硬件断点, 其格式如下:

```
ba [ID] Access Size [Options] [Address [Passes]] ["CommandString"]
```

其中 ID 用来指定断点的序号, 与 **bp** 命令一样, 如果指定序号, 那么 WinDBG 会自动编排。Access 用来指定触发断点的访问方式, 可以为以下几个字母之一。

- **e:** 当从指定地址读取和执行指令时, 触发断点, 这种断点又称为访问代码硬件断点。从效果上来看, 这种断点与软件断点的效果是类似的, 都是执行代码时触发断点, 但是硬件断点的好处是不需要做指令替换和恢复。
- **r:** 当从指定地址读取和写入数据时, 触发断点。
- **w:** 当向指定地址写数据时触发断点。通过 r 和 w 选项设置的断点又称为访问数据断点。
- **i:** 当向指定的地址执行输入输出访问(I/O)时触发断点, 对于 x86 架构, IN 和 OUT 指令用于读写 IO 端口, 这种断点又称为访问 I/O 断点。

Size 参数用来指定访问的长度, 对于访问代码硬件断点, 它的值应该为 1。对于其他硬件断点, 允许的长度值因平台的不同而不同, 对于 x86 系统, 可以为 1、2、4 三种值, 分别代表对指定地址的 1 字节访问、字访问和双字访问。对于 x64 系统可以为 1、2、4、8(四字访问)四种值。对于安腾系统, 可以为 1 到 0x80000000 间的任

何 2 的次方值。因为 CPU 是根据实际访问是否包含断点定义区域来判断是否命中的，所以当实际访问长度大于断点定义的访问长度时，断点也会命中，举例来说，当使用以下命令设置一个断点时：

```
ba r1 0041717c
```

对内存地址 0041717c 的 1 字节访问、字访问、双字访问（读写）都会触发这个断点。

Address 参数用来指定断点的地址。需要注意的是，地址值一定是按照 Size 参数的值做内存对齐的。比如，如果 Size 是 4，那么地址值应该是按 4 字节做内存对齐的，也就是它的值应该是 4 的整数倍。

Passes 参数和 CommandString 参数的用法与软件断点一样，不再重复介绍。

硬件断点是设置在 CPU 的调试寄存器中的，在 x86 CPU 中就是 DR0~DR7。比如当我们设置上面的断点并让调试目标恢复执行，然后再将其中断到调试器时，可以看到 dr0 和 dr7 的寄存器内容为：

```
0:000> r dr0,dr6,dr7
dr0=0041717c dr6=fffff0ff0 dr7=00030501
```

dr0 就是断点的地址值，dr6 是断点的状态寄存器，dr7 是断点的控制寄存器，详见第 4 章关于这些寄存器的介绍。

因为硬件断点要占用 CPU 的调试寄存器，所以硬件断点的数量是很有限的。但是，因为在恢复目标执行时才会把断点落实到上下文的寄存器中，所以在发出 ba 命令时，即使超出了硬件断点的数量限制，WinDBG 也不会报告错误，只有当恢复目标执行时，它才报告错误：

```
0:000> g
Too many data breakpoints for thread 0
bp8 at 00417180 failed
WaitForEvent failed
```

上面信息的含义是数据断点个数太多，WaitForEvent 调用失败，失败后 WinDBG 会返回到命令模式。

最后要说明的一点是，当初始断点命中时，尚不能设置硬件断点，如果设置，那么会得到如下错误：

```
0:000> ba r1 kernel32!BasepCurrentTopLevelFilter
^ Unable to set breakpoint error
The system resets thread contexts after the process breakpoint so hardware
breakpoints cannot be set. Go to the executable's entry point and set it then.
```

提示信息告诉我们，初始断点之后系统会重新设置线程上下文，因此还不能设置硬件断点，建议执行到程序的入口后再设置。

30.12.3 条件断点

在调试时，如果要分析的代码或变量被多次执行和访问，那么对它设置的断点就

会反复命中，而我们可能只关心特定条件时的命中，不关心其他情况。为了避免断点反复命中而浪费时间，可以使用条件断点。当断点发生时，调试器会检查断点条件，对于不关心的情况，立刻恢复目标执行，当关心的情况发生时中断给用户。

根据前面的介绍，软件断点命令和硬件断点命令都支持指定一组关联命令，当断点命中时，WinDBG会自动执行这组命令。因此我们可以在这组命令中判断是否需要中断到命令模式，如果不需要，便立刻恢复执行。那么如何判断中断条件呢？答案是可以使用 `j` 命令或 `.if...:else` 命令，即：

```
bp|bu|bm|ba Address "j (Condition) 'OptionalCommands'; 'gc' "
bp|bu|bm|ba Address ".if (Condition) {OptionalCommands} .else {gc}"
```

其中 `Condition` 用来定义希望中断的情况，`OptionalCommands` 用来定义关心的情况发生后中断到命令模式时顺便执行的命令。举例来说，以下是使用 `j` 命令设置的条件断点命令：

```
bp dbgee!wmain "j (poi(argc)>1) 'dd argc 11;du poi(poi(argv)+4)'; 'gc'"
```

这个命令对 `dbgee` 程序的 `wmain` 函数设置一个条件断点，只有当命令行参数的个数 (`argc`) 大于 1 时，才中断给用户，中断时执行两条命令，一条是 `dd argc 11`，用来显示 `argc` 参数的值，另一条是 `du poi(poi(argv)+4)`，用来显示第一条命令行参数（即 `argv[1]`，`argv[0]` 是程序文件）的字符串内容。

事实上，无论后面是否有条件命令，对于上面的 `bp` 命令，WinDBG 都是在 `dbgee!wmain` 函数的入口处设置一个软件断点。当 CPU 执行到这个位置时，也总是触发断点事件，并报告给调试器。调试器收到断点事件后，会在内部维护的断点队列中找到这个断点，然后执行与这个断点关联的命令串，也就是双引号中的内容。于是 WinDBG 执行 `j` 命令，判断小括号中的条件，如果条件不满足，就执行分号后的 `gc` 命令，直接恢复调试目标执行，如果条件满足，就执行单引号中的命令，然后进入命令模式中断给用户。

如果使用 `.if` 命令，那么上面的命令可以写成：

```
bp dbgee!wmain ".if (poi(argc)>1) {dd argc 11;du poi(poi(argv)+4)} .else {gc}"
```

其中 `poi` 是 MASM 表达式支持的特殊运算符。在 MASM 语法中，`argc` 代表一个地址，要取它的值就需要使用 `poi` 操作符，`poi` 的含义是从指定地址取指针长度的数据 (Pointer-sized data)，类似的还有 `by`、`wo`、`dwo`、`qwo`，分别表示从指定地址取一个字节 (Byte)、一个字 (Word)、一个双字 (Double-word) 和一个四字 (Quad-word)。本章第 30.4 节介绍了 MASM 表达式，关于 MASM 表达式的更多内容，请读者参考 WinDBG 帮助文件中 MASM Numbers and Operators 的介绍。

下面我们再给出一个针对函数参数设置条件断点的例子。比如，我们想了解 IO 管理器的 `IoGetDeviceProperty` 函数的工作细节，这个函数的原型如下：

```
NTSTATUS IoGetDeviceProperty( IN PDEVICE_OBJECT DeviceObject,
    IN DEVICE_REGISTRY_PROPERTY DeviceProperty,
```

```
IN ULONG BufferLength, OUT PVOID PropertyBuffer,
OUT PULONG ResultLength );
```

其中第二个参数是个枚举型的常量，用来指定要查询的设备属性，比如 DevicePropertyBusNumber(14) 用来查询设备的总线号。如果直接对 IoGetDeviceProperty 函数设置断点，那么它会频繁命中，而我们实际上只关心第二个参数等于某个值，比如 14 的情况。这时便可以这样设置断点：

```
0: kd> bp nt!IoGetDeviceProperty+0x5 ".if poi(@ebp+0xc) = 0xe {} .else {.echo Entered IoGetDeviceProperty with ;dd (@ebp+0xc) 11; gc}"
```

首先，为什么将第一个参数（断点地址）设置为 nt!IoGetDeviceProperty+0x5，而不是 nt!IoGetDeviceProperty 呢？这是为了让建立栈帧的代码执行好后再中断，以便后面的条件表达式可以引用到栈帧中的参数。其中 poi(@ebp+0xc) 用来表示第二个参数的取值。恢复调试目标运行后，当这个函数被调用但第二个参数不等于 14 时，WinDBG 会执行 else 块中的命令打印出函数每次被调用的信息：

```
Entered IoGetDeviceProperty with
f7ca1a00 00000007
```

因此，我们可以利用这样的条件断点来实现 VS 调试器中的追踪点功能。第二个参数等于 14 时，WinDBG 会中断到命令模式，让我们做进一步的分析。

30.12.4 地址表达方法

可以使用以下 3 种方法来指定断点命令中的地址参数。

- 使用模块名加函数符号的方式，比如 bp dbgee!wmain 代表对 dbgee 模块中的 wmain 函数设置断点，也可以在符号后增加一个地址偏移，比如 bp dbgee!wmain+3。
- 直接使用内存地址，比如 bp 00411390。
- 如果是使用完全的调试符号，调试符号中包含源代码行信息，那么可以使用如下形式：`[[Module!]Filename][:LineNumber]`，其中 Module 为模块名，Filename 为源程序文件名，LineNumber 为行号。整个表达式应该使用两个重音符号 (`) 包围起来，注意是重音符号，而不是单引号 ('')。比如以下命令 `bp `dbgee!dbgee.cpp:16` 是对 dbgee.cpp 的 16 行设置断点，其中 dbgee! 可以省略。
- 对于 C++ 的类方法，也可以使用类名双冒号 (::) 或双下画线 (__) 来连接类名和方法名，比如：bp MyClass__MyMethod，bp MyClass::MyMethod 或 bp @@(MyClass::MyMethod)。

如果使用前两种方法设置软件断点，那么应该确保断点地址指向的是指令的起始处，而不是一条指令的中部。如果指向一条指令的中部，那么当落实这个断点时，调试器就会把这条指令的中间字节替换为断点指令。这样，当 CPU 执行到这个位置时，CPU 会认为这里是一条多字节指令，把原来的指令和断点指令放在一起解码，这会导致难以预知的结果。

30.12.5 设置针对线程的断点

对于多线程程序，如果有多个线程都会调用某个函数，那么，有时我们可能只希望在某个线程调用这个函数时才中断到调试器。为了满足这一需要，WinDBG 的软件断点设置命令支持在命令前增加线程限定符，即~加线程号。例如，以下命令对 MSVCR80D!printf 设置一个断点，这个断点是线程相关的，只有当 0 号线程执行到这个函数时才会中断给用户：

```
~0 bp MSVCR80D!printf
```

以上方法适用于用户态调试的情况，对于内核态调试可以使用我们前面介绍的/p 和/t 选项来指定断点的进程上下文和线程上下文。

30.12.6 管理断点

使用 bl 命令可以列出当前已经设置的所有断点，清单 30-11 给出了使用 bl 命令观察到的断点列表。

清单 30-11 使用 bl 命令显示断点列表

序号	状态	地址	穿越次数	线程号	描述
0	e	[c:\...\dbgee.cpp @ 16]	0001 (0001)	0:*****	dbgee!wmain+0x73
1	d	[c:\...\dbgee.cpp @ 8]	0001 (0001)	0:*****	dbgee!wmain "kv" Call stack shallower than: 00000005 // 中断条件为栈帧深度小于 5
2	e	7c843ac r 1 0001 (0001)	0:*****	kernel32!BasepCurrentTopLevelFilter	
3	e	[f:\rtm\...\printf.c @ 49]	0002 (0002)	0:~000	MSVCR80D!printf
4	e	7c91eb28 0001 (0001)	0:*****	ntdll!DbgPrintEx+0x3	"kv"

在以上命令结果中，第 1 列是断点的序号。第 2 列是断点的状态，e 代表启用 (enable)、d 代表暂时禁止使用 (disable)，对于使用 bu 命令设置的断点，e 或 d 后可能跟有字母 u，代表断点尚未落实 (unresolved)。第 3 列是断点的地址，跟设置断点时指定地址一样有多种表示方法，可以为源文件加行号 (断点 0、1、3)，或者内存地址 (断点 2、4)。对于数据断点 (断点 2)，地址后跟有访问方式 (r) 和访问长度 (1)，我们把它们连同地址看作一列。第 4 列和第 5 列都与穿越断点的次数有关，第 4 列用来指示还要穿越 (pass) 这个断点多少次才会中断到命令模式。第 5 列是穿越计数的初始值，也就是设置断点时 Passes 参数所指定的值，默认为 1。第 6 列是断点所关联的进程和线程，冒号前是进程号，冒号后是线程号，***** 代表这个断点是针对所有线程的。第 7 列是断点地址的符号表示。

如果断点有关联的命令，会被显示在第 7 列之后，例如断点 1 的 kv 命令。断点 1 下面的信息是因为这个断点使用了/c5 选项。

命令 bc、bd 和 be 分别用来删除、禁止、启用断点，它们的格式都是：

```
bc | bd | be 断点号
```

其中断点号可以使用*来通配所有断点，使用-来表示一个范围，或者使用逗号来指定

多个断点号，例如以下命令都是有效的：

```
bd 0-2,4 //禁止 0、1、2、和 4 号断点。  
be * //启用所有断点。
```

可以使用 br 命令对改变某个断点的编号。例如当 3 号断点删除后，可以使用 br 4 3 将 4 号断点的编号改为 3 号。

30.13 控制进程和线程

很多软件是由多个进程所构成的一个系统，每个进程中可能还包含着多个线程。随着多核处理器的出现和迅速发展，越来越多的软件开始考虑如何通过并行化提高软件的执行速度。并行化的一个基本方式就是多线程，也就是将本来在一个线程中串行执行的任务分解成多个任务放到多个线程中并行执行。就像编写多线程程序比编写单线程程序复杂一样，调试多线程程序通常也要比调试单线程程序难度更大。本节将先介绍调试多线程程序和同时调试多个进程所需掌握的基本知识和要领。

30.13.1 MulThrds 程序

为了便于说明后面要介绍的内容，我们特意编写一个可以动态创建线程的小程序，名为 MulThrds（Multi-Threads 之意）。它的界面如图 30-16 所示。

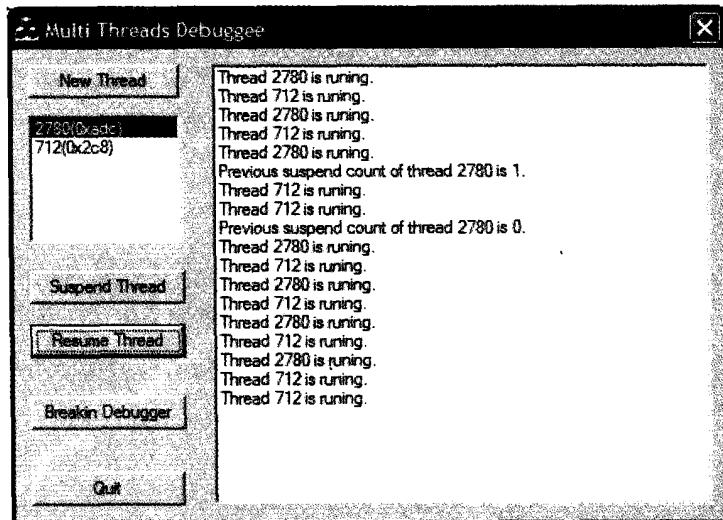


图 30-16 MulThrds 程序的界面

起初运行时，MulThrds 是单线程的，只有用户界面（UI）线程。每次点击 New Thread 按钮时，UI 线程会创建一个新的工作线程，工作线程的任务就是每秒钟向右侧的列表框中加一条消息，显示自己的 ID 和说明自己在运行。每个工作线程的 ID 会被放入到

New Thread 按钮下面的列表框中。在这个列表框中选中一个线程 ID 后，再点击下面的 Suspend Thread 按钮，会触发 UI 线程调用 SuspendThread API 来增加选中线程的挂起计数(Suspend Count)，点击下面的 Resume Thread 按钮，会触发调用 ResumeThread API 来减少挂起计数。挂起计数是线程的一个基本属性，当一个线程的挂起计数大于 0 时，那么这个线程处于被挂起状态(Suspended)，不会被执行。只有当挂起计数降低到 0 时，它才会被执行。因此当选中一个运行着线程并点击挂起按钮后，对应的线程会被挂起而停止运行，当再点击恢复按钮后，它又会恢复执行。

Breakin Debugger 按钮会触发 UI 线程调用 DebugBreak API，因此当这个程序处于被调试状态时按这个按钮会将其中断到调试器中。如果这个程序没有处于被调试状态而点击 Breakin Debugger 按钮，那么它会因为发生未处理异常而被系统关闭。

30.13.2 控制线程执行

通常，当被调试程序被中断到调试器中时，它的所有线程都是被挂起的；当恢复执行时，所有线程都被恢复运行。但是在调试时，可以根据调试任务的需要而保持某些线程仍处于停止运行状态。为了实现这一目标，WinDBG 提供了多种方法，我们介绍以下三种。

第一种方法是通过增加线程的挂起计数来禁止线程被恢复运行。当调试目标中断到调试器中时，WinDBG 会对所有线程依次调用 SuspendThread API，当恢复程序执行时，再对所有线程调用 ResumeThread API。因此当我们在调试器中观察线程时，每个线程的挂起计数通常是 1。以 MulThrds 程序为例，点击两次 New Thread 按钮后，点击 Breakin Debugger 使其中断到调试器中，使用~命令列出所有线程：

```
0:000> ~
. 0  Id: 1440.1554 Suspend: 1 Teb: 7ffde000 Unfrozen
  1  Id: 1440.2c8 Suspend: 1 Teb: 7ffdd000 Unfrozen
  2  Id: 1440.adc Suspend: 1 Teb: 7ffdc000 Unfrozen
```

第一列是线程序号，Id：后面是进程 ID 和线程 ID，之后便是挂起计数，而后是线程环境块(Teb) 的地址。最后一列是线程的冻结状态，我们稍后再详细介绍。

此时可以使用~Thread n 命令来增加 Thread 线程的挂起计数。比如输入如下命令可以增加 1 号线程的挂起计数：

```
0:000> ~1 n
```

此时再观察线程状态，可以看到 1 号线程的挂起计数变成 2 了：

```
0:000> ~1
  1  Id: 1440.2c8 Suspend: 2 Teb: 7ffdd000 Unfrozen
```

此时输入 g 命令恢复目标程序执行，恢复执行后，我们可以看到只有一个工作线程在活动了，另一个工作线程没有恢复执行。

与~n 命令相对，~m 命令用来减少线程的挂起计数。比如~1 m 可以将 1 号线程的

挂起计数减 1。当线程的挂起计数降低到 -1 后，不可再降低。从实现角度来看，`-n` 命令就是调用 `SuspendThread`，而`-m` 命令就是调用 `ResumeThread API`。由此可见，我们可以通过`-n` 和`-m` 命令来改变线程的挂起计数从而控制被调试的线程是否运行。

控制线程执行的第二种方法是使用`-f` 和`-u` 命令，前者用来冻结（`Freeze`）一个线程，后者用来解冻（`Unfreeze`）。当一个线程处于冻结状态时，恢复目标执行时这个线程不会恢复执行。

例如，使用以下命令可以冻结 1 号线程：

```
0:000> ~1 f
```

观察线程状态，可以看到其冻结状态为 `Frozen`（被冻结）。

```
0:000> ~1
1 Id: 1440.2c8 Suspend: 1 Teb: 7ffdd000 Frozen
```

输入`g` 命令恢复目标执行，WinDBG 会提示有一个线程在冻结状态。

```
0:000> g
System 0: 1 of 3 threads are frozen
```

程序恢复执行后，从 UI 上看，也可以发现 1 号线程没有恢复执行。

从实现角度来看，与`-n` 和`-m` 命令是调用操作系统的 API 来改变线程的系统属性（挂起计数）不同，`-f` 和`-u` 命令完全是调试器内部维护的一个线程属性。当执行`-f` 和`-u` 命令时，调试器引擎内部调用 `ThreadInfo` 类的 `ChangeFreeze` 方法，修改线程的属性信息。通常，当恢复程序执行时，调试器会对所有线程依次调用 `ResumeThread API`。而一旦某个线程被设置为冻结状态，那么 WinDBG 便不再对其调用 `ResumeThread API`，因此这个线程也就不会恢复执行。从操作系统的角度来看这个线程是处于被挂起状态。值得提醒的是，如果使用`-m` 命令将一个冻结线程的挂起计数先降为 0，然后恢复目标程序执行，那么这个线程是会恢复运行的。

控制线程执行的第三种方法就是在恢复执行的命令前通过线程限定符和线程号码只恢复执行指定的线程。比如，不管目标程序有多少个线程，命令`-0 g` 都只是恢复 0 号线程执行，例如：

```
0:000> ~0 g
System 0: 2 of 3 threads are frozen
```

从实现角度来讲，WinDBG 在执行`-0 g` 命令时就是只对 0 号线程调用 `ResumeThread API`，对其他线程不调用，使它们处于挂起状态。

值得说明的是，当这样只恢复一个线程执行后，如果试图使用 `Ctrl+Break`（或者菜单 `Debug>Break`）来俘获调试目标时，默认使用的远程中断线程方法会失败，超时后 WinDBG 会使用挂起方法。具体来说，按 `Ctrl+Break` 后 WinDBG 显示如下信息：

```
System 0: 2 of 4 threads were frozen
Create thread 3:16fc 【要看到此行信息需要将线程创建事件的处理方式改为 Output】
System 0: 3 of 4 threads are frozen
Break-in sent, waiting 30 seconds...
```

我们知道，WinDBG 执行 Ctrl+Break 的方法是在目标进程中创建一个远程线程（对于 XP 或更高版本的 Windows 使用 DebugBreakProcess API），然后让这个线程执行 DebugBreak 函数而中断到调试器中。上面的第 1、2 行信息就是 WinDBG 收到新线程创建事件时所输出的信息。在 MulThrds 进程中本来有三个线程，我们只恢复 0 号执行，所以本来是 3 个中的 2 个被冻结，这里新创建了一个，因此第 1 行信息显示 4 个线程中的 2 个被冻结。第 3 行信息是当 WinDBG 处理好新线程创建事件让目标继续执行时而显示的，因为当前的调试器状态是只恢复 0 号线程执行，因此这里不会对新创建的线程调用 ResumeThread，所以它也被冻结了，变成 4 个线程中的 3 个被冻结，也就是说新创建的远程中断线程一创建就被冻结了。第 4 行信息是 Ctrl+Break 命令的输出，WinDBG 等待超时后会强行挂起目标进程而进入命令模式。

类似的，也可以在单步跟踪命令前加上线程限定符，这样保证只有指定的线程会被单步执行，其他线程不会执行，例如以下命令是让 0 号线程单步执行一次：

```
0:000> ~0 t
System 0: 2 of 3 threads are frozen
System 0: 2 of 3 threads were frozen
```

第 2 行信息是当恢复目标执行时调试引擎给出的提示，告诉我们目标进程共有 3 个线程，其中两个是被冻结的。也就是说，调试器不会对这两个线程调用 ResumeThread，因此它们不会恢复执行。其中的 System 0 是调试目标所在系统的编号，当同时调试多个系统上多个进程时这个信息很有用。第 3 行信息是单步执行后，调试器收到单步事件后准备进入命令时打印出的，告诉我们调试目标在上次执行时 3 个线程中的 2 个是被冻结着的。注意这两条提示信息很类似，但后一句是使用过去时态的。

30.13.3 多进程调试

WinDBG 支持使用一个调试器来调试多个进程，这些进程可以在一个系统（操作系统）上，也可以在多个系统上。我们先介绍同时调试与调试器处于同一个系统中的多个进程的情况。

我们继续使用上面调试 MulThrds 程序的调试会话，先运行一个记事本程序（进程 ID 为 2788），然后通过.attach 命令把它加入到调试会话中：

```
0:003> .attach 0n2788
Attach will occur on next execution
```

提示信息告诉我们，调试器已经做了必要的登记，但是真正的附加动作需要等下次恢复调试目标时发生。事实上，此时记事本程序已经被挂起了。执行任意一个恢复目标执行的命令，比如 g，让调试目标恢复执行，这时 WinDBG 会提示有悬而未决的附加操作：

```
*** wait with pending attach
```

很快，WinDBG 会进入命令模式，新附加的进程中断到调试器中。WinDBG 的命

令提示符为 1:004 的样子，表示当前上下文是 1 号进程的 4 号线程。使用~命令可以列出当前进程的所有线程：

```
1:004> ~
 3  Id: ae4.8d4 Suspend: 1 Teb: 7ffd000 Unfrozen
 . 4  Id: ae4.c30 Suspend: 1 Teb: 7ffde000 Unfrozen
```

因为线程号是全局编排的，0~2 号分给了 MulThrds 进程的三个线程，所以记事本的两个线程的编号分别是 3 和 4，其中一个是 UI 线程，另一个是调试器创建的远程中断线程。

至此，当前调试器已经和两个进程建立了调试关系。我们可以使用前面介绍的调试命令来分析和控制这两个进程，比如可以单独让 3 号线程恢复执行：

```
1:003> ~3 g
System 0: 4 of 5 threads are frozen
```

这时，只有 Notepad 的 UI 线程是恢复执行的，Notepad 进程中的远程中断线程和 MulThrds 进程的所有线程都是被挂起的。

也可以利用我们前面介绍的线程控制命令来控制哪些线程执行，哪些不执行，这在调试存在相互协作的分布式应用时可能非常有价值。

利用!<进程号> s 命令可以切换当前进程，使用~<线程号> s 可以切换当前线程。如果省略进程号和线程号，那么可以观察当前进程和线程，例如：

```
0:001> !
. 0  id: c10  attach  name: C:\WINDOWS\system32\notepad.exe
```

如果要调试位于多个系统中的进程，那么可以在.attach 命令中通过-premote 开关来指定另一个系统的位置和通信方式。类似的，.create 命令也支持-premote 开关。-premote 参数的写法与建立远程调试相同，请参阅 30.5.7 节。

30.14 观察栈

今天我们所使用的计算机系统都是基于栈架构的，栈是进行函数调用的基础。栈中记录了软件运行的丰富信息，观察和分析栈是软件调试的一种重要手段。第 22 章详细讨论过栈的布局，栈帧的建立和变量分配等内容，本节将介绍如何在 WinDBG 调试器中观察和分析栈。

30.14.1 显示栈回溯

因为函数调用指令（CALL）会将函数的返回地址记录在栈上，因此通过从栈顶向下遍历每个栈帧来追溯函数调用过程，这个过程被称为栈回溯（Stack Backtrace）。WinDBG 的 k 系列命令就是用来帮助我们进行栈回溯的。这一系列命令都是以字符 k 开头的，它们的功能类似，显示格式有所不同。下面我们以调试 dbgee 程序为例逐步介绍各个命令。

先来看基本的 k 命令，清单 30-12 给出了当设置在 main 函数入口处的断点命中时执行 k 命令后的结果。

清单 30-12 使用 k 命令进行栈回溯

```
0:000> k
ChildEBP RetAddr
0012ff68 00411ad6 dbgee!wmain [c:\dig...\code\chap28\dbgee\dbgee.cpp @ 8]
0012ffb8 0041191d dbgee!__tmainCRTStartup+0x1a6 [f:\...\crtexe.c @ 583]
0012ffc0 7c816ff7 dbgee!wmainCRTStartup+0xd [f:\...\crtexe.c @ 403]
0012fff0 00000000 kernel32!BaseProcessStart+0x23
```

清单中的每一行描述当前线程的用户态栈上的一个栈帧。最上面一行描述的是程序指针所对应的函数，也就是当这个线程中断到调试器时正在执行的函数。每个函数下面的一行是调用这个函数的上一级函数，有时称为父函数。总体看来，函数的调用顺序是由下至上的，下面的函数调用上面的。最下面一行是栈中的第一个栈帧，对应的是当前线程的启动函数 BaseProcessStart，倒数第 2 行是 wmainCRTStartup 函数，即编译器插入的程序入口函数。

横向来看，第 1 列是栈帧的基地址，因为 x86 架构中通常使用 EBP 寄存器来记录栈帧的基地址，所以这一行的标题叫 ChildEBP，意思是子栈帧（子函数）的基址寄存器（EBP）值。第 2 列是函数的返回地址，这个地址是父函数中的指令地址，通常就是调用本行函数的那条 CALL 指令的下一条指令的地址。第 3 列是函数名以及执行位置，其中执行位置表示的是程序指针指向的位置（对于正在执行的函数）或者返回到这个函数时将执行的指令地址（对于父函数）。例如第一行的 dbgee!wmain 表示当前的程序指针指向的是这个函数的第一条指令。第 2 行的 dbgee!__tmainCRTStartup+0x1a6 表示的是 wmain 返回后将执行的指令地址，这个值其实是通过寻找距离第 1 行的返回地址（00411ad6）最近的符号而得到的，即：

```
0:000> ln 00411ad6
f:\rtm\vctools\crt_bld\self_x86\crt\src\crtexe.c(583)+0x19
(00411800)  dbgee!__tmainCRTStartup+0x1a6 | (00411ae0)  dbgee!NtCurrentTeb
```

第 3 列之后是源文件信息（如果有）。如果不想要看到这个信息可以在 k 命令后加上 L（需要大写）选项。

K 命令显示了函数名信息，但是没有显示每个函数的参数，命令 kb 可以显示放在栈上的前三个参数。例如以下是在与清单 30-13 同样的条件下执行 kb L 命令的结果（使用 L 开关是为了屏蔽掉与上文相同的源文件信息，以节约篇幅）。

清单 30-13 使用 kb 命令进行栈回溯

```
0:000> kb L
ChildEBP RetAddr Args to Child
0012ff68 00411ad6 00000002 003a2e90 003a5c20 dbgee!wmain
0012ffb8 0041191d 0012fff0 7c816ff7 0175f6f2 dbgee!__tmainCRTStartup+0x1a6
0012ffc0 7c816ff7 0175f6f2 0175f77a 7ffd8000 dbgee!wmainCRTStartup+0xd
0012fff0 00000000 0041107d 00000000 78746341 kernel32!BaseProcessStart+0x23
```

显而易见，前两列以及最后一列的内容与 k 命令的结果是一样的。所不同的是中间三列，这三列被称为是子函数的参数（Args to Child）。不管函数的实际参数个数是多少，这里总是显示三个，第一个是位于 EBP+8 处的，第二个是位于 EBP+C 处，第三个是 EBP+0x10 处的，如果要观察第四个参数，那么可以使用 dd EBP+0x14，依此类推。尽管通常说这三列是函数的前三个参数，事实上这是不准确的，严格来说，这只是放在栈上的前三个参数。对于使用快速调用协议（FASTCALL）的函数来说，某些参数是用寄存器来传递的，因此栈上的前三个参数很可能并不是真正的前三个参数。换句话来说，调试器只是把栈帧上用来传递参数的三个内存位置的值显示出来，至于它们到底对应的是哪个参数，则应该参考函数的原型。如果符号文件中包含私有符号信息，那么 kp 命令会根据符号文件中的函数原型信息来帮助我们自动做这件事（见清单 30-14）。

清单 30-14 使用 kp 命令进行栈回溯

```
0:000> kp L
ChildEBP RetAddr
0012ff68 00411ad6 dbgee!wmain(int argc = 2, wchar_t ** argv = 0x003a2e90)
0012ffb8 0041191d dbgee!__tmainCRTStartup(void)+0x1a6
0012ffc0 7c816ff7 dbgee!wmainCRTStartup(void)+0xd
0012fff0 00000000 kernel32!BaseProcessStart+0x23
```

kp 命令把参数和参数值都以函数原型的格式显示出来，显然更易于理解。但是仅在有完全的调试符号（私有符号）时才能做到这一点。对于没有私有符号的函数，kp 命令无法显示它的参数，比如清单 30-13 的最后一行。如果希望每个参数占一行，那么可以使用 kP（P 大写）命令。

kv 命令可以在 kb 命令的基础上增加显示 FPO（栈指针省略）信息和调用协议（清单 30-15）。

清单 30-15 使用 kv 命令进行栈回溯

```
0:000> kv L
ChildEBP RetAddr Args to Child
0012ff68 00411ad6 ... dbgee!wmain+0x3 (FPO: [Non-Fpo]) (CONV: cdecl)
0012ffb8 0041191d ... dbgee!__tmainCRTStartup+0x1a6 (FPO: [Non-Fpo]) (CONV: cdecl)
0012ffc0 7c816ff7 ... dbgee!wmainCRTStartup+0xd (FPO: [Non-Fpo]) (CONV: cdecl)
0012fff0 00000000 ... kernel32!BaseProcessStart+0x23 (FPO: [Non-Fpo])
```

Kv 命令的前六列与 kb 命令都是一样的，为了节约空间，我们省略了关于参数的三列。第 7 列和第 8 列分别是 FPO 信息和调用协议，前者以 FPO 开始，后者以 CONV 开始。因为上面的各个函数都没有使用帧指针省略，所以显示的都是（FPO: [Non-Fpo]），意思是帧中没有 FPO 数据。

除了以上命令，还有 kn 命令，它会在每行前显示栈帧的序号。另外可以在所有 k 命令中指定 f 选项，有了这个选项后，WinDBG 会显示每两个相邻栈帧的内存距离（见清单 30-16），即栈帧地址的差值，这可以帮助我们观察栈的使用情况，对观察空间较少的内核态帧特别有用，经验丰富的工程师可以通过这些数据推测栈的健康状况。

判断是否发生栈溢出。

清单 30-16 在 k 命令中使用 f 和 L 选项

```
0:000> kn f L
# Memory ChildEBP RetAddr
00          0012ff68 00411ad6 dbgee!wmain+0x3
01          50 0012ffb8 0041191d dbgee!__tmainCRTStartup+0x1a6
02          8 0012ffc0 7c816f7 dbgee!wmainCRTStartup+0xd
03         30 0012fff0 00000000 kernel32!BaseProcessStart+0x23
```

其中，第 1 列是栈帧序号，当前正在执行函数的栈帧为 0 号，依次类推。第 2 列是相邻栈帧的基址差值，这个数值越大，说明对应的函数使用的栈空间越多。

30.14.2 观察栈变量

大多数局部变量是分配在栈上的（详见第 22 章）。观察函数的栈帧就可以看到这个函数在栈上的局部变量。可以使用 dd 命令加上栈帧地址来观察栈帧的原始内存，但这样观察到的结果难以理解，WinDBG 的 dv 命令可以帮助我们以更友好的方式显示栈上的局部变量。

仍然以前面的 dbgee 小程序为例，当位于 main 函数入口的断点命中时，执行 dv 命令，其结果如清单 30-17 所示。

清单 30-17 使用 dv 命令观察栈上的局部变量

```
0:000> dv /i/t/V
prv param 0012ff70 @ebp+0x08 int argc = 1
prv param 0012ff74 @ebp+0x0c wchar_t ** argv = 0x003a2e90
prv local 0012fd4c @ebp-0x21c wchar_t [260] szBuffer = wchar_t [260] "????"
prv local 0012ff5c @ebp-0x0c int nRet = 0
```

其中第 1 列是符号类型，prv 是 private 的缩写，表示这个信息是利用私有符号产生的。第 2 列是变量的类型，param 表示函数参数，local 表示局部变量。第 3 列是变量在内存中的起始地址，这个地址应该在当前函数的栈帧范围内。第 4 列是使用栈帧基址址（EBP）表示的变量起始地址，因为栈是向低地址方向生长的，所以参数位于 EBP 的正偏移方向，局部变量位于 EBP 的负偏移方向。对于上面的数据，当前函数的栈帧基址是 0012ff68（清单 30-15），即 EBP = 0012ff68，第一个参数的位置是 EBP+8，即 0012ff70，第二个参数是 EBP+0xC，即 0012ff74。第 5 列是变量类型，第 6 列是变量名称，而后是等号和变量取值。值得说明的是，因为现在还在 main 函数的入口处，初始化局部变量的代码尚未运行，所以 szBuffer 变量的值还是随机的。

如果要观察父函数的局部变量，那么可以使用.frame 命令加上父函数的帧号将局部上下文切换到那个栈帧，然后再使用 dv 命令。例如，从清单 30-15 可以看到 __tmainCRTStartup 函数的帧号为 1，所以可以先切换到 1 号栈帧，然后再用 dv 命令列出 __tmainCRTStartup 函数的栈变量，即：

```
0:000> .frame 1
01 0012ff8 0041191d dbgee!__tmainCRTStartup+0x1a6 [f:\rtm\...\crtexe.c @ 583]
0:000> dv /i /t /v
prv local 0012ff94 @ebp-0x24 void * lock_free = 0x00000000
prv local 0012ff98 @ebp-0x20 void * fiberid = 0x00130000
prv local 0012ff9c @ebp-0x1c int nested = 0
```

使用 `dv` 命令显示局部变量需要有私有符号信息，对于没有私有符号的模块，`dv` 命令是无法工作的。比如栈帧 3 是 `kernel32` 的 `BaseProcessStart` 函数，我们没有这个函数的私有符号，因此当把局部上下文切换到这个函数再执行 `dv` 命令时会得到错误信息，即：

```
0:000> .frame 03
03 0012ffff 00000000 kernel32!BaseProcessStart+0x23
0:000> dv /i /t /v
Unable to enumerate locals, HRESULT 0x80004005
Private symbols (symbols.pri) are required for locals.
Type ".hh dbgerr005" for details.
```

可见 `dv` 命令失败了，原因是缺少 `kernel32` 模块的私有调试符号。对于这种情况只能用手工方法来观察局部变量。这通常有两种方法，第一种是直接使用内存观察窗口或者内存显示命令（下一节介绍）浏览当前栈帧的内存区域，根据 `EBP` 的值和栈帧布局知识来寻找局部变量。对于字符串类型的变量，有时候可以根据变量内容找到它的起始位置。

第二种方法是根据汇编指令中对局部变量的引用来得到局部变量的地址，然后再观察这个地址的内容。对于没有使用 FPO 的函数，局部变量大都是通过 `EBP-XXX` 的方式来引用的。例如以下就是 `main` 函数中使用 `nRet` 局部变量的汇编语句：

```
004113c8 c745f400000000 mov dword ptr [ebp-0Ch], 0
```

根据这一行汇编，我们可以知道 `nRet` 变量的地址是 `EBP-0xC`。对于 VC7 或者更高版本的 VC 编译器，局部变量至少是从偏移-0xC 开始的，因为 `EBP-4` 是安全 Cookie 值，`EBP-8` 是保护安全 Cookie 的屏障字段（即 0xFFFFFFFF）。另外，在 32 位系统中，栈空间是以 4 字节为单位分配的，所以 0xC 是可能的最小局部变量偏移地址，这个局部变量可以是个整数或者比整数还短的类型。以下是 `main` 函数栈帧附近内存的原始内容：

```
0:000> dd ebp-10
0012ff58 cccccccc 00000000 cccccccc 04b69550
0012ff68 0012ffb8 00411ad6 00000001 003a2e90
```

其中，`EBP` (`0012ff68`) 处的数据 (`0012ffb8`) 是父函数的 `EBP` 值，`EBP+4` 是 `main` 函数的返回地址 (`00411ad6`)，`EBP-4` 处的值 (`04b69550`) 是安全 Cookie，`EBP-8` 是 Cookie 屏障，`EBP-0xC` 就是局部变量 `nRet`。

下面再介绍一种通过函数调用来推测变量类型的简易方法，以下是访问局部变量的一段典型代码：

```
0041146e 8d85e4fdffff lea eax,[ebp-21Ch]
00411474 50 push eax
```

```
00411475 ff15cc824100    call    dword ptr [dbgee!_imp__wprintf (004182cc)]
```

第一句是把变量的地址（指针）放入到 EAX 寄存器，第二句是将 EAX 作为参数压入到栈中，第三句是调用字符串打印函数。由此推断 `ebp-21Ch` 一定是个字符串类型的局部变量，因此可以使用 `du` 命令观察它的值：

```
0:000> du ebp-21Ch
0012fd4c  "Arg [0]: C:\dig\dbg\author\code\
0012fd8c  "chap28\dbgee\Debug\dbgee.exe"
```

这正是 `main` 函数中 `szBuffer` 变量的值。

可以使用扩展命令 `!for_each_local` 来枚举当前栈帧的所有局部变量，并对每个变量执行一系列命令，在命令中可以使用别名 `@#Local` 来指代相关的变量。例如，使用 `!for_each_local dt @#Local` 命令可以显示出每个局部变量的类型和取值。类似的，扩展命令 `!for_each_frame` 用来遍历所有栈帧，比如使用以下命令可以遍历当前线程的所有栈帧，并显示出每个栈帧的所有局部变量：

```
0:000> !for_each_frame dv
0012ff68 00411ad6 dbgee!wmain+0x2f [c:\...\dbgee.cpp @ 15]
argc = 2
argv = 0x003a2e90 .....
```

其效果相对于先使用 `.frame` 命令切换到每个栈帧，然后执行 `dv` 命令。

30.15 分析内存

内存是软件工作的舞台，程序的代码必须先加载到内存中才可以被 CPU 所执行，程序的数据（变量）也主要是分配在内存中的（极少数变量分配在寄存器中）。除了外界因素外，程序的行为就是由它的代码和数据所决定的，程序的内存状态决定了它的外在行为。很多时候，软件调试的目标就是为了搞清楚某个行为的内在根源（Root Cause），观察和分析内存是寻找根本原因的一种有效方式。内存空间是通过地址来标识和引用的，内存地址有多种，常用的有物理内存地址和虚拟内存地址，本节中的内存地址除非特别说明外都是指虚拟内存地址。

30.15.1 显示内存区域

WinDBG 的 `d` 系列命令用来显示指定地址的内存区域，这些命令的格式为：

```
d{a|b|c|d|D|f|p|q|u|w|W} [Options] [Range]
dy{b|d} [Options] [Range]
d [Options] [Range]
```

其中大括号中的字母用来指定数据的显示方式，是区分大小写的，`a` 表示 ASCII 码，`b` 表示字节和 ASCII 码，`c` 表示 DWORD 和 ASCII 码，`d` 表示 DWORD，`D` 表示双精度浮点数，`f` 表示单精度浮点数，`p` 表示按指针宽度显示，`q` 表示四字（8 字节），

u 表示 UNICODE 字符，**w** 表示字，**W** 表示字和 ASCII 码，**yb** 表示二进制和字节，**yd** 表示二进制和双字。Range 参数用来指定要显示的内存范围，可以有以下几种表示方法。

第一种方法是起始地址加空格加终止地址，比如 dd 0012fd9c 0012fda8 命令以双字格式显示从 0012fd9c 开始到 0012fda8 结束的 16 字节内存数据：

```
0:000> dd 0012fd9c 0012fda8
0012fd9c cccccccc cccccccc cccccccc cccccccc
```

第二种方法是起始地址加空格加 L（或者 1）和元素个数，比如上面的命令可以等价地写为：dd 0012fd9c L4。

第三种方式是终止地址加空格加 L（或者 1）加负号和对象个数。使用这种方式可以把上面的命令写为：dd 0012fdac L-4。

注意理解上面的对象个数（L 后的数字），它是“数据单元”的个数，而不是字节数。对于 dd 它的单位是 DWORD，对于 db，它的单位是字节，例如：

```
0:000> db 0012fd9c 14
0012fd9c cc cc cc cc
....
```

如果省略数据显示格式而直接执行 d 命令，那么它将采用最近使用过的数据显示格式。

30.15.2 显示字符串

对于以 0 结尾的简单字符串，可以使用 da 或者 du 命令来显它的内容，前者适用于单字节字符集的字符串，后者适用于 UNICODE 字符集的字符串。当遇到字符串末尾的 0 时，WinDBG 会自动停止显示，例如：

```
0:000> du 003a2e9c
003a2e9c "C:\dig\dbg\author\code\chap28\db"
003a2edc "gee\Debug\dbgee.exe"
```

如果使用 da 命令显示 UNICODE 字符集的字符串，如果字符串的内容是英文字母，那么通常只能显示第一个字符，因为第二个字节便是 0，例如：

```
0:000> da 003a2e9c
003a2e9c "C"
```

有些字符串是使用数据结构来表示的，常用的结构有 **UNICODE_STRING** 结构和 **STRING** 结构，它们的定义分别为：

```
0:000> dt _UNICODE_STRING
ntdll!_UNICODE_STRING
+0x000 Length      : Uint2B          //Buffer 缓冲区中字符串的字节数
+0x002 MaximumLength : Uint2B          //Buffer 缓冲区可以容纳的最多字节数
+0x004 Buffer       : Ptr32 Uint2B    //指向宽字符的字符串，可能不以零结束
0:000> dt _STRING
ntdll!_STRING
+0x000 Length      : Uint2B          //Buffer 缓冲区中字符串的字节数
```

```
+0x002 MaximumLength    : UInt2B           //Buffer缓冲区可以容纳的最多字节数
+0x004 Buffer          : Ptr32 Char        //指向单字符的字符串
```

对于使用这两种结构存储的字符串，可以分别使用 `ds` (`S` 大写) (`UNICODE_STRING` 结构) 和 `ds` 命令 (`STRING` 结构) 来显示，对于后者，也可以使用 `!str` 命令来显示。

30.15.3 显示数据类型

WinDBG 的 `dt` 命令用来显示数据类型和按照类型来显示内存中的数据。`Dt` 的含义是 Dump symbolic Type information。`Dt` 是个比较复杂的命令，下面按照它的功能逐步来介绍。

首先，可以使用 `dt` 来显示一个数据类型（数据结构），比如上文我们用 `dt` 命令显示了 `_UNICODE_STRING` 结构的定义，这种用法的一般格式是：

```
dt [模块名!] 类型名
```

其中模块名部分可以省略，如果省略，那么调试器会自动搜索所有符号文件。类型名是程序中定义的数据结构名称或者通过 `typedef` 定义的别名。类型名中可以包含通配符，比如以下命令会列出 `NTDLL` 模块中的所有类型：

```
0:000> dt ntdll!*  
ntdll!LIST_ENTRY64 .....
```

如果类型名是确定的类型，那么 `dt` 便会显示这个类型的定义，如果类型中还包含子类型，那么可以用 `-b` 开关来递归式显示所有子类型，也可以使用 `-r` 开关来指定显示深度，`-r0` 表示不显示子类型，`-r1` 表示显示 1 级子类型，依此类推，例如：

```
0:000> dt -r1 _TEB  
ntdll!_TEB  
+0x000 NtTib      : _NT_TIB  
+0x000 ExceptionList  : Ptr32 _EXCEPTION_REGISTRATION_RECORD  
+0x004 StackBase   : Ptr32 Void  
+0x008 StackLimit  : Ptr32 Void  
.....
```

如果不显示整个结构，而只显示某些字段，那么可以在类型名后使用 `-ny` 开关附加搜索选项，比如以下命令只显示 `TEB` 结构中以 `LastError` 开始的字段：

```
0:000> dt _TEB -ny LastError  
ntdll!_TEB  
+0x034 LastErrorValue : UInt4B
```

`Dt` 命令的第二种用法是在上一种方法的基础上增加内存地址，让 `dt` 按照类型显示指定地址的变量。例如，使用 `dt _PEB 7ffdd000` 命令可以把内存地址 `7ffdd000` 处的数据按照 `_PEB` 结构显示出来。这时仍可以使用前面介绍的 `-r` 和 `-y` 开关。

`Dt` 命令的第三种用法是显示类型的实例，包括全局变量、静态变量和函数，比如以下命令显示 `dbgee` 程序的 `g_szGlobal` 全局变量：

```
0:000> dt dbgee!g_szGlobal
[14] "A global var."
```

因为函数是一种特殊的类型实例，所以也可以使用它来枚举函数符号，这样使用 dt 命令实际上与 x 命令的功能很类似，比如：

```
0:000> dt dbgee!*wmain*
004113a0 dbgee!wmain
00411910 dbgee!wmainCRTStartup
```

如果参数中指定的是某个确定的函数，那么 dt 会显示它的参数和取值：

```
0:000> dt dbgee!wmain
wmain int (int argc = 2, wchar_t** argv = 003a2e90 )
```

dt 命令的第四种用法是使用-l 开关来显示链表中的所有元素，也就是遍历链表，我们将在下一节详细讨论这种用法。

30.15.4 搜索内存

一个 32 位 Windows 程序的进程空间是 4GB，用户空间有 2GB 或者 3GB，一个典型的 Windows 程序实际使用的内存空间通常在几百 KB 到几十 MB 之间。即使是几百 KB，手工在这么大的内存范围内寻找某个内容也是很困难的，这时可以让 WinDBG 的 s 命令来帮忙。S 命令有三种使用方法，我们按照从简单到复杂的顺序依次介绍。

第一种用法是在指定的内存范围内搜索任何 ASCII 字符或者 UNICODE 字符串，其格式如下：

```
s -[[Flags]]sa|su Range
```

其中 Range 用来指定内存范围，其写法与 d 命令的 Range 参数一样，sa 开关用于搜索 ASCII 字符串，su 用来搜索 UNICODE 字符串。[Flags] 用来指定搜索选项，比如可以用 l 加一个整数来指定字符串的最小长度，使用 s 将搜索结果保存起来，然后使用 r 再在保存的结果中搜索。

例如，以下命令搜索 nt!PsInitialSystemProcess 变量所指向地址开始的 512 个字节范围内任何长度不小于 5 的 ASCII 字符串：

```
1kd> s-[15]sa poi(nt!PsInitialSystemProcess) 1200
8a672764 "System"
```

以上结果表明在内存地址 8a672764 处找到了"System"字符串，事实上这正是系统进程的名字，也就是_EPROCESS 结构的 ImageFileName 字段。

第二种用法是在指定内存地址范围内搜索与指定对象相同类型的对象，这里的对象是指包含虚拟函数表的使用面向对象语言（如 C++）编写的类（Class）对象，其格式为：

```
s -[[Flags]]v Range Object
```

例如，在我们编写的 MfcHello 程序中，CMfcHelloDlg 类定义了五个 CButton 类的实例 m_Button1~m_Button5。通过观察我们知道 m_Button1 对象的地址是 0x12fe4c，

`CMfcHelloDlg`类实例的地址是`0x12fc30`,因此我们可以输入以下命令来搜索与`m_Button1`同类型的其他对象:

```
0:000> s-v 0x12fc30 11000 0x12fe4c
```

但是在6.7.5.1和6.8.4版本的WinDBG中,S命令的这种用法总是返回以下错误信息: Object '0x12fe4c' has no vtables

S命令的第三种用法是在指定范围内搜索某一内容模式,其语法格式为:

```
s [-[[Flags]]Type] Range Pattern
```

其中Type用来指定要搜索内容的数据类型(宽度),它决定了匹配搜索内容的方式,可以为字母b(字节)、w(字)、d(双字)、q(四字)、a(ASCII字符串)或者u(Unicode字符串)之一,如果不指定类型,那么默认类型为b,即按字节搜索指定的内容。Range参数用来指定搜索范围,Pattern参数用来指定要搜索的内容,可以用空格分隔依次要搜索的数值,比如:

```
0:000> s-w 0x400000 12a000 41 64 76 44 62 67  
0041b954 0041 0064 0076 0044 0062 0067 0000 0000 A.d.v.D.b.g.....
```

因为要搜索的内容可以表示为ASCII码,因此以上命令也可以等价表示为如下形式:

```
0:000> s-w 0x400000 12a000 'A' 'd' 'v' 'D' 'b' 'g'  
0041b954 0041 0064 0076 0044 0062 0067 0000 0000 A.d.v.D.b.g.....
```

或者按字符串搜索,需要用双引号来包围要搜索的内容,比如:

```
0:000> s-u 0x400000 12a000 "AdvDbg"  
0041b954 0041 0064 0076 0044 0062 0067 0000 0000 A.d.v.D.b.g.....
```

以下是搜索双字的一个例子:

```
0:000> s-d 12fe4c 120 782e35fc  
0012fe4c 782e35fc 00000001 00000000 .5.x.....
```

可以借助!for_each_module扩展命令在当前进程的所有模块中进行搜索,例如,以下命令会在每个模块中搜索字符串Debugger:

```
0:000> !for_each_module s-a @#Base @#End "Debugger"  
00420e4c 44 65 62 75 67 67 65 72-50 72 65 73 65 6e 74 00 DebuggerPresent.  
10304b1a 44 65 62 75 67 67 65 72-50 72 65 73 65 6e 74 00 DebuggerPresent.
```

其中@#Base和@#End是!for_each_module命令定义的别名,除了这两个外还有@#ModuleName(模块名称)、@#SymbolFileName(符号文件名称)、@#Size(模块大小)和@#SymbolType(符号文件类型)等。

30.15.5 修改内存

命令e用来修改指定内存地址或者区域的内容,我们仍然按用法来介绍。

第一种是按字符串方式编辑指定地址的内容,其一般格式为:

```
e{a|u|za|zu} Address "String"
```

其中Address是要修改内存的起始地址,za代表以0结尾的ASCII字符串,zu代表以0

结尾的 Unicode 字符串，`a` 和 `u` 分别代表不是以 0 结尾的 ASCII 和 Unicode 字符串。

仍然以 MfcHello 小程序为例，在 CMfcHelloDlg 类中我们定义了一个 TCHAR m_szBuffer[MAX_PATH] 成员，在构造函数中将其初始化为 "AdvDbg"，使用 CMfcHelloDlg 实例的地址作为开始地址搜索 "AdvDbg" 字符串，我们可以找到 m_szBuffer 成员的地址：

```
0:000> s-u 0x0012fa20 l200 "AdvDbg"
0012fc94 0041 0064 0076 0044 0062 0067 0000 cccc A.d.v.D.b.q....
```

从上面的命令结果知道从内存地址 0012fc94 开始存放着 Unicode 类型的字符串“AdvDbg”，它是以 0 结尾的，0 之后是初始化时填充的 CC，现在可以输入如下命令修改这个字符串的内容：

0:000> ezu 12fc94 "DbgAdv"

使用内存观察命令观察，可以看到修改成功了：

```
0:000> dw 12fc94
0012fc94 0044 0062 0067 0041 0064 0076 0000 cccc
0012fcfa4 cccc cccc cccc cccc cccc cccc cccc cccc....
```

将上面命令中的 `ezu` 换成 `eu` 得到的结果也是一样的，但是如果新的串比原来的串长，这时再使用 `eu` 命令就可能导致原来的字符串失去结尾的 0，而无法正常显示：

这时如果恢复程序运行，那么可能导致程序错误，只要使用 `ezu` 编辑一下，就恢复正常了：

```
0:000> ezu 12fc94 "Advanced Debugging"
0:000> du 12fc94
0012fc94  "Advanced Debugging"
```

也就是说，ezu 命令或保证在编辑好的串末尾加 0，而 eu 不会。类似的，ea 和 eza 的差异也是这样。

第二种用法是以数值方式编辑，其格式为：

`es[bl|d|P|f|p|q|w} Address [Values]`

其中，大括号中的字母用来表示要修改数据的类型，也决定了修改内存的方式。Address 用来指定要改内存的起始地址。Values 用来指定新的值，其表示方法与前面搜索内存中的方法相同。Values 参数的多少决定了要修改内存的长度，例如以下命令将 0x12fc94 开始的 5 个 WORD 都改为 0x41（字符 A）：

0:000> ew 12fc94 41 41 41 41 41 41

显示修改后的内存：

```
0:000> du 12fc94
0012fc94  "AAAAAced Debugging"
```

如果在命令中没有指定 Values 参数，那么 WinDBG 会以交互式的方式来让用户输入，命令提示符会改变为“Input>”，即图 30-17 所示的样子。

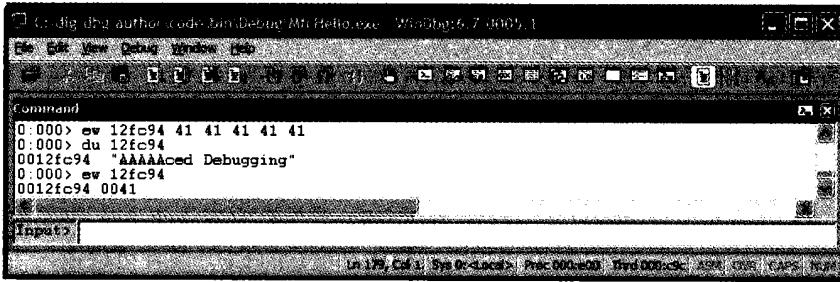


图 30-17 交互式的修改内存

根据命令中指定的编辑类型，WinDBG 会先显示出要编辑的内存地址（12fc94）和当前的取值（0041），然后等待用户输入。此时可以输入新的值，然后按回车提交输入值。如果想保留当前值，那么可以按空格然后再按回车。如果想停止输入，那么直接按回车键。

30.15.6 使用物理内存地址

前面讲的命令中，使用的都是虚拟地址，如果要显示和修改物理地址，那么需要使用扩展命令!d{blcld|plqlulw}和!e{bld}，其用法与前面介绍的很类似，但只有在内核调试时才可以使用这些命令。

30.15.7 观察内存属性

使用扩展命令!address 可以显示某一个内存地址（区域）的特征信息，它的基本格式是：

```
!address [Address]
```

其中 Address 是要观察的内存地址，比如，以下是关于 m_szBuffer 变量所在内存的信息：

```
0:000> !address 12fc94
00030000 : 00126000 - 0000a000
  Type      00020000 MEM_PRIVATE
  Protect   00000004 PAGE_READWRITE
  State     00001000 MEM_COMMIT
  Usage     RegionUsageStack
  Pid.Tid   e00.c9c
```

其中第 1 行的含义是，指定的内存地址属于一个从 00030000 开始的较大内存区(region)中以 00126000 开始的较小内存区，这个内存区的大小是 0000a000。

第 2 行是内存类型，可以包含以下标志位：MEM_IMAGE、MEM_MAPPED 或者 MEM_PRIVATE，分别代表从执行映像文件映射的内存、从其他文件映射的内存和私

有内存（不是从文件映射的，也不是与其他进程共享的）。

第3行是内存的页属性，可以包含以下标志位：PAGE_READONLY、PAGE_READWRITE、PAGE_READONLY、PAGE_EXECUTE、PAGE_GUARD等值，其含义请参考MSDN中关于VirtualAlloc API的说明。

第4行是内存的状态，可以为MEM_COMMIT、MEM_RESERVE和MEM_FREE之一，分别代表已经提交的内存、保留的内存和标志为释放的内存。

第5行是内存区的用途，可以为以下常量之一：RegionUsageIsVAD（虚拟地址描述符VAD）、RegionUsageFree（空闲）、RegionUsageImage（执行映像的映射）、RegionUsageStack（栈）、RegionUsageTeb（线程环境块）、RegionUsageHeap（堆）、RegionUsagePageHeap（页堆，参见23章）、RegionUsagePeb（进程环境块）、RegionUsageProcessParametrs（进程参数）、RegionUsageEnvironmentBlock（环境块）。

第6行的内容会因为用途的不同而有所不同，因为上面的内存地址属于栈，所以显示的是这个栈所属的进程ID和线程ID。

以下是另一个例子，它显示的是字符串常量“AdvDbg”所在地址的信息：

```
0:000> !address 41b954
00400000 : 0041b000 - 00004000
  Type      01000000 MEM_IMAGE
  Protect    00000002 PAGE_READONLY
  State     00001000 MEM_COMMIT
  Usage      RegionUsageImage
  FullPath  MfcHello.exe
```

因为常量是作为映像文件的一部分而映射到内存中的，所以可以看到它的用途是RegionUsageImage，最后一行的信息是映像文件的名称，内存页属性是只读的。

如果不指定地址参数，那么WinDBG会显示当前进程（或者内核空间——对于内核调试）中的所有内存区域和关于这些区域的统计信息。例如，清单30-18给出了针对MfcHello进程的显示结果。

清单30-18 使用!address命令观察进程的所有内存区域（节选）

```
0:000> !address
00000000 : 00000000 - 00010000 【空指针区】
  Type      00000000
  Protect    00000001 PAGE_NOACCESS 【不可访问，一旦触及便导致异常】
  State     00010000 MEM_FREE 【尽管属性为空闲，但是永远不会使用】
  Usage      RegionUsageFree
00010000 : 00010000 - 00002000 【环境变量区】
  Type      00020000 MEM_PRIVATE
  Protect    00000004 PAGE_READWRITE 【可读写】
  State     00001000 MEM_COMMIT
  Usage      RegionUsageEnvironmentBlock
... 【省略用于进程参数的1段和两段空闲区】
00030000 : 00030000 - 000f5000 【初始线程的栈内存区，又分为多个子区】
  Type      00020000 MEM_PRIVATE 【这个区是尚未提交的保留区】
  State     00002000 MEM_RESERVE
```

```

Usage      RegionUsageStack
Pid.Tid   e00.c9c
00125000 - 00001000 【这是保护页面使用的区域，大小为4KB】
Type      00020000 MEM_PRIVATE
Protect   00000104 PAGE_READWRITE | PAGE_GUARD 【具有保护属性】
State     00001000 MEM_COMMIT
Usage     RegionUsageStack
Pid.Tid   e00.c9c
00126000 - 0000a000 【已经提交的区域】
Type      00020000 MEM_PRIVATE
Protect   00000004 PAGE_READWRITE 【可读写】
State     00001000 MEM_COMMIT 【已经提交】
Usage     RegionUsageStack
Pid.Tid   e00.c9c
..... 【省略很多个用于堆、VAD和模块的区域】
----- Usage SUMMARY ----- 【根据用途统计的报表】
TotSize (    KB) Pct(Tots) Pct(Busy) Usage
e2b000 ( 14508) : 00.69% 47.54% : RegionUsageIsVAD
7e222000 ( 2066568) : 98.54% 00.00% : RegionUsageFree
c3e000 ( 12536) : 00.60% 41.07% : RegionUsageImage
100000 ( 1024) : 00.05% 03.36% : RegionUsageStack
1000 (     4) : 00.00% 00.01% : RegionUsageTeb
260000 ( 2432) : 00.12% 07.97% : RegionUsageHeap
0 (     0) : 00.00% 00.00% : RegionUsagePageHeap
1000 (     4) : 00.00% 00.01% : RegionUsagePeb
1000 (     4) : 00.00% 00.01% : RegionUsageProcessParameters
2000 (     8) : 00.00% 00.03% : RegionUsageEnvironmentBlock
Tot: 7fff0000 (2097088 KB) Busy: 01dce000 (30520 KB)
----- Type SUMMARY ----- 【根据类型统计的报表】
TotSize (    KB) Pct(Tots) Usage
7e222000 ( 2066568) : 98.54% : <free> 【仍有很大的可用空间】
c3e000 ( 12536) : 00.60% : MEM_IMAGE 【映像文件】
ca8000 ( 12960) : 00.62% : MEM_MAPPED 【内存映射】
4e8000 ( 5024) : 00.24% : MEM_PRIVATE 【栈、堆和环境信息等】
----- State SUMMARY ----- 【根据状态统计的报表】
TotSize (    KB) Pct(Tots) Usage
1170000 ( 17856) : 00.85% : MEM_COMMIT 【已经提交】
7e222000 ( 2066568) : 98.54% : MEM_FREE 【空闲】
c5e000 ( 12664) : 00.60% : MEM_RESERVE 【保留】
Largest free region: Base 10320000 - Size 27be0000 (651136 KB) 【最大空闲区】

```

在按用途统计的报表中，第1列是总字节数，第2列是十进制表示的KB字节数，第3列是占总内存的百分比，第4列是占繁忙状态的百分比，第5列是用途。在另两个报表中，去除了第4列。总内存大小是按用户空间的总长度2GB来计算的。

除了!address命令，还可以使用!vprot命令来显示一个内存地址的属性，它的显示结果与!address类似，例如：

```

0:001> !vprot 12fc94
BaseAddress: 0012f000 【区域的基地址】
AllocationBase: 00030000 【虚拟内存区的基地址】
AllocationProtect: 00000004 PAGE_READWRITE 【保护属性】
RegionSize: 00001000 【区域大小】
State: 00001000 MEM_COMMIT 【状态，已经提交】
Protect: 00000004 PAGE_READWRITE 【保护属性】
Type: 00020000 MEM_PRIVATE 【类型，私有】

```

也可以使用扩展命令!vadump 来显示当前进程的所有虚拟地址，其显示与不带参数的!address 命令类似，但没有统计报表。

除了以上介绍的命令，在内核调试中，还可以使用!pte 命令来显示指定虚拟地址所属的页表表项（PTE）和页目录表项（PDE），比如：

```
kd> !pte 801544f4
      VA 801544f4
PDE at 00000000C0602000    PTE at 00000000C0400AA0
contains 000000000741163  contains 0000000000154163
pfn 741 -G-DA--KWEV      pfn 154 -G-DA-KWEV
```

其中，第 3 行显示的分别是 PDE 和 PTE 的虚拟地址，第 4 行是 PDE 和 PTE 的内容，第 5 行是将第 4 行的内容按高 20 位和低 12 位分解为 PFN（Page Frame Numer）和页属性。PFN 用来转换物理地址，其规则是先将其（PFN）乘以 0x1000（页大小，也就是左移 12 位），然后再加上虚拟地址的页内偏移，即后 12 位。因此，以上虚拟地址的物理地址为：

```
154*0x1000+4F4 = 1544F4
```

分别使用!dd 命令和 dd 观察内存地址 1544F4 和 801544f4，可以看到它们的内容是一致的，说明以上转换结果是正确的。页属性中的 G 代表全局（Global），D 代表数据，A 代表访问过（Accessed），K 代表这是内核态拥有的内存页，W 代表可以写，E 代表可以执行，V 代表有效（Valid），即对应的内存页已经在物理内存中。关于虚拟内存和内存保护的更详细介绍，请参阅第 2 章。

30.16 遍历链表

链表是非常常用的一种数据结构，Windows 操作系统的很多重要数据都是以链表方式组织的。在任何一个用户态调试会话中执行 dt ntdll!*List* 可以看到很多种链表类型，如 LIST_ENTRY64、LIST_ENTRY32、_LIST_ENTRY 等等。在调试 Windows XP 的内核调试会话中执行 x nt!*List* 可以显示出 300 多个符号，有些是用来记录链表地址的全局变量，有些是处理链表的函数。

30.16.1 结构定义

Windows 中主要使用两种链表，一种是双向链表，另一种是单向链表。链表的每个节点都由两部分组成，一部分是起连接作用的 LIST_ENTRY 结构或者 SINGLE_LIST_ENTRY 结构，我们将它们统称为链接结构。另一部分是负载，也就是链表要管理的内容。清单 30-19 给出了两种链接结构的定义。

清单 30-19 链表结构的链接结构

```
typedef struct _LIST_ENTRY { //用于双向链表的链接结构
    struct _LIST_ENTRY *Flink; //指向前向节点
```

```

struct _LIST_ENTRY *Blink; //指向后向节点
} LIST_ENTRY, *PLIST_ENTRY, *RESTRICTED_POINTER PRLIST_ENTRY;
typedef struct _SINGLE_LIST_ENTRY { //用于单向链表的链接结构
    struct _SINGLE_LIST_ENTRY *Next; //指向下一个节点
} SINGLE_LIST_ENTRY, *PSINGLE_LIST_ENTRY;

```

30.16.2 双向链表示例

下面我们先来看一个双向链表的例子。第 10 章曾经介绍过，Windows 内核使用 EPROCESS 结构来记录每个 Windows 进程，并使用双向链表来把每个进程的 EPROCESS 结构串联在一起，全局变量 PsActiveProcessHead 记录着这个链表的起始地址。EPROCESS 结构中的 ActiveProcessLinks 字段便是起链接作用的 LIST_ENTRY 结构，使用 dt 命令可以观察到这个字段：

```

1kd> dt _EPROCESS -y ActiveProcess
nt!_EPROCESS
+0x088 ActiveProcessLinks : _LIST_ENTRY

```

值得注意的是，ActiveProcessLinks 字段在结构中的偏移是 0x88，也就是说它并不是位于结构的起始处，这意味着这个链表是通过每个元素中部的链接结构衔接在一起的，即图 30-18 所描绘的样子。

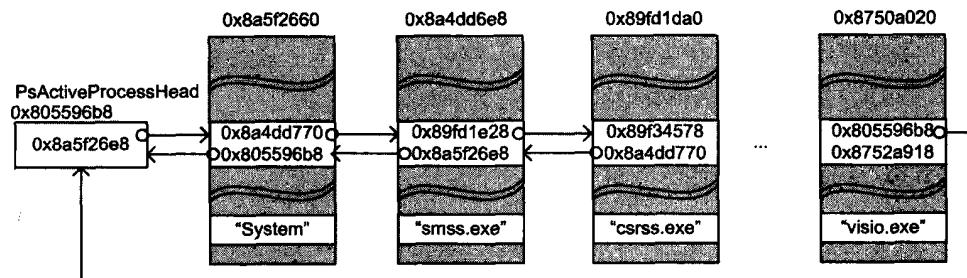


图 30-18 记录进程执行结构的双向链表

图中给出的地址值是作者实验的结果，给出这些地址的目的是更清楚地观察地址间的相互关系。其中，全局变量 PsActiveProcessHead 的地址是 0x805596b8，取值是 0x8a5f26e8，而 0x8a5f26e8 正是系统进程的 EPROCESS 结构的 ActiveProcessLinks 字段的地址，也就是说，PsActiveProcessHead 变量指向的就是系统进程的 EPROCESS 结构中的链接结构（_LIST_ENTRY）。地址 0x8a5f26e8 的值是 0x8a0475c8，指向的是下一个进程（smss.exe）的 EPROCESS 结构的 ActiveProcessLinks 字段，依此类推，直到最后一个进程的 ActiveProcessLinks.Flink 指向 PsActiveProcessHead，形成一个回路。

PsInitialSystemProcess 记录了系统进程的 EPROCESS 结构的地址，我们可以使用 dt -l 命令来列出从这个地址开始的所有节点，清单 30-20 给出了执行这个命令的部分结果。

清单 30-20 使用 dt 命令来遍历进程链表

```

lkd> dt nt!_EPROCESS -l ActiveProcessLinks.Flink -y Ima -yoI Uni
poi(PsInitialSystemProcess)
ActiveProcessLinks.Flink at 0x8a5f2660          //链表节点的起始地址
----- //开始显示链表节点的内容
UniqueProcessId : 0x00000004                      //进程 ID
ActiveProcessLinks : [ 0x8a4dd770 - 0x805596b8 ]   //链接结构_LIST_ENTRY
ImageFileName : [16] "System"                      //进程的映像文件名称
                                                //两个节点之间空一行
ActiveProcessLinks.Flink at 0x8a4dd6e8          //第 2 个链表节点的地址
----- //分隔线
UniqueProcessId : 0x000005f8                      //进程 ID
ActiveProcessLinks : [ 0x89fd1e28 - 0x8a5f26e8 ]   //链接结构_LIST_ENTRY
ImageFileName : [16] "smss.exe"                    //会话管理器进程
                                                //省略很多其他节点
.....

```

需要注意几点，第一，以上命令中 poi (PsInitialSystemProcess) 是作为地址参数传递给 dt 命令的，这个地址指向的是系统进程的_EPROCESS 结构，而不是 LIST_ENTRY 结构。因此，尽管 dt 命令的结果中提示 ActiveProcessLinks.Flink at 0x8a5f2660，但 at 后的地址并不是 LIST_ENTRY 结构的地址，而是链表节点的起始地址，也就是 EPROCESS 结构的地址。二者的差异就是 ActiveProcessLinks 字段在 EPROCESS 结构中的偏移，即 0x88。举例来说，PsActiveProcessHead 的值是 0x8a5f26e8，它指向的是 ActiveProcessLinks 字段，即 $0x8a5f26e8 = 0x8a5f2660 + 0x88$ 。为了避免这种额外的换算，大多数链表节点都把链接结构放在节点的开始处。

第二，-l 开关后的字段参数的作用是帮助 dt 命令寻找下一个 LIST_ENTRY 结构，它可以使用“子结构字段.字段”的形式，比如上面指定的是 ActiveProcessLinks.Flink。尽管 Flink 字段指向的是下一个类型结构中的 ActiveProcessLinks 字段地址，而不是 EPROCESS 结构的地址，但我们不用担心，dt 命令会帮我们自动将 ActiveProcessLinks 字段地址减去这个字段在结构中的偏移而得到 EPROCESS 结构的地址。观察上面的命令结果，第一个 EPROCESS 结构的地址是 0x8a5f2660，它的 ActiveProcessLinks.Flink 的内容是 0x8a4dd770。在显示下一个 EPROCESS 结构时，dt 命令提示它的地址为 0x8a4dd6e8，可见此时 dt 已经将 0x8a4dd770 做了调整，调整的幅度恰好是 ActiveProcessLinks 字段的偏移，即 $0x88 = 0x8a4dd770 - 0x8a4dd6e8$ 。

30.16.3 单向链表示例

下面我们再给出一个单向链表的例子，以 TEB 结构中的异常处理器登记链表为例，即：

```

0:000> dt -r2 _TEB
+0x000 NtTib           : _NT_TIB
+0x000 ExceptionList  : Ptr32 _EXCEPTION_REGISTRATION_RECORD
+0x000 Next             : Ptr32 _EXCEPTION_REGISTRATION_RECORD
+0x004 Handler          : Ptr32      _EXCEPTION_DISPOSITION

```

其中 `ExceptionList` 字段指向的是一个 `EXCEPTION_REGISTRATION_RECORD` 结构，用来记录异常处理器链表的起始地址（详见 24 章）。`EXCEPTION_REGISTRATION_RECORD` 结构的 `Next` 字段指向下一个 `EXCEPTION_REGISTRATION_RECORD` 结构，`Handler` 字段和其后的数据是链表的负载。使用~命令列出当前进程的所有线程：

```
0:000> ~
. 0 Id: 1700.164 Suspend: 1 Teb: 7ffdf000 Unfrozen
```

因为 `NtTib` 在 `TEB` 结构的开始处，而 `ExceptionList` 又是在 `NtTib` 结构的开始处，所以 `TEB` 结构的地址（7ffdf000）也就是 `ExceptionList` 字段的地址，于是可以使用 `dt _EXCEPTION_REGISTRATION_RECORD -1 Next poi(7ffdf000)` 命令显示出当前线程的所有异常处理器（清单 30-21）。

清单 30-21 使用 dt 命令来遍历异常登记链表

```
0:000> dt _EXCEPTION_REGISTRATION_RECORD -1 Next poi(7ffdf000)
dbgee!_EXCEPTION_REGISTRATION_RECORD
Next at 0x12fd0c                                     //链表的第一个节点的地址
-----                                         //开始显示节点的内容
+0x000 Next      : 0xffffffff _EXCEPTION_REGISTRATION_RECORD
+0x004 Handler   : 0x7c90ee18 _EXCEPTION_DISPOSITION ntdll!_except_handler3+0

Next at 0xffffffff                                    //下一个节点的地址
-----                                         //开始显示节点的内容
+0x000 Next      : ????                           //读取内存失败
+0x004 Handler   : ????                           //读取内存失败
Memory read error 00000003
```

因为 `dt` 命令使用 `NUL` 来判断链表结束，而异常登记链表使用 `0xFFFFFFFF` 来表示链表结束，所以上面的结果中的错误是因为 `dt` 命令试图读取位于 `0xFFFFFFFF` 处的节点而失败。

30.16.4 DI 命令

除了 `dt` 命令，也可以使用 `dl` 命令来遍历链表，其格式为：

```
dl[b] Address MaxCount Size
```

其中 `Address` 是链表的起始地址，它应该指向 `LIST_ENTRY` 或者 `SINGLE_LIST_ENTRY` 结构，`MaxCount` 用来指定要显示的最多节点数，`Size` 用来指定链表节点的结构长度，或者说希望显示的结构长度，它是以指针长度为单位的。仍以进程链表为例，使用如下 `dl` 命令可以显示出进程链表中的所有元素：

```
1kd> dl nt!PsActiveProcessHead 1000
805596b8 8a5f26e8 87c76d18 00000001 ac3bda6c
8a5f26e8 8a4dd770 805596b8 00000000 00000000
8a4dd770 89fd1e28 8a5f26e8 00000280 00001884
.....
```

以上结果中，每一行对应一个链表节点，即一个 `_EPROCESS` 结构，第 1 列是

`LIST_ENTRY` 结构的地址，也就是各个进程的 `EPROCESS` 结构的 `ActiveProcessLinks` 字段的地址。第 2 列是 `ActiveProcessLinks` 字段的 `Flink` 子字段的值，第 3 列是 `Blink` 子字段值，第 4 列和第 5 列是 `EPROCESS` 结构中 `ActiveProcessLinks` 字段后的内容，即 `QuotaUsage`（偏移+`0x090`）和 `QuotaPeak`（`+0x09c`）字段的值。因为我们没有指定结构长度参数，所以 `dl` 命令使用默认值显示 4 个指针长度的值，或者说它默认链表节点的结构长度为 4 个指针长度。可以使用`@@c++(sizeof(_EPROCESS)/sizeof(int *))`来指定 `EPROCESS` 结构的长度。

从上面结果我们还看到，因为 `dl` 命令是假定 `LIST_ENTRY` 结构是定义在链表表项结构的头部的，所以它在显示完 `LIST_ENTRY` 结构后，便显示与其相邻地址的其他数据。但这种假设有时是不成立的，比如 `EPROCESS` 结构就把 `LIST_ENTRY`（`ActiveProcessLinks` 字段）定义在中间。对于这种情况，我们需要把 `dl` 命令显示的 `ActiveProcessLinks` 字段地址减去它的偏移才能得到 `EPROCESS` 结构的地址。可见 `dl` 命令既简单功能又比较有限。

30.16.5 !list 命令

也可以使用扩展命令 `!list` 来遍历链表，显示链表的每个元素或者对其执行一组命令，`!list` 命令的格式为：

```
!list -t [Module!]Type.Field -x "Commands" [-a "Arguments"] [Options] StartAddress
!list "-t [Module!]Type.Field -x \"Commands\" [-a \"Arguments\"] [Options]
StartAddress"
```

例如，可以使用`!list -t _EPROCESS.ActiveProcessLinks.Flink -e -x \"$extret l4; dt _EPROCESS @$extret -y Image\" poi(nt!PsInitialSystemProcess)`命令来枚举系统内的所有进程，显示出每个进程映像文件名称。

命令中的`$extret` 是`!list` 命令所定义的伪寄存器，用来代表当前链表节点的地址，上面的命令可以分解为如下几个部分。

- 使用`-t` 开关指定的数据类型和它的链接字段。这里的数据类型是指链表所链接的那个数据结构，在本例中也就是 `EPROCESS` 结构，为了与它所包含的 `LIST_ENTRY` 结构相区别，我们将这个较大的数据结构称为**外层结构**（Outer Structure）。链接字段是供`!list` 命令寻找下一个元素的。外层结构和链接字段使用点相分隔，与编程语言中的表示很类似。这与 `dt` 命令将类型放在前面，然后使用`-l` 来指定链接字段不同。
- 使用`-x` 来指定对每个节点所执行的命令，比如我们在上面的例子中指定的命令是：
`dd @$extret l4; dt _EPROCESS @$extret -y Image`
- 使用`-e` 或者`-m` 来指定执行选项，`-e` 的含义是显示（echo）针对每个节点所执行的命令。`-m` 加一个数字用来限制最多显示的节点数。

总结一下，dt、dl、和!list 三个命令都可以遍历链表结构，dl 命令最简单，只需向其提供链表表头的地址。!list 命令最复杂，它可以对每个节点执行一组命令。!list 与 dt 命令的地址参数指定的都是外层结构的地址，而不是 LIST_ENTRY 结构的地址；而 dl 命令的地址参数必须是链接结构（LIST_ENTRY 或者 SINGLE_LIST_ENTRY）的地址。

30.17 调用目标程序的函数

本节将介绍如何使用 WinDBG 的.call 元命令来从调试器中调用被调试程序中的函数。我们通过一个实例看这个命令的用法，然后再讨论它的工作原理。

30.17.1 调用示例

使用 WinDBG 打开 dbgee 小程序（调试版本），当初始断点命中时，执行 bp wmain 对程序的主函数设置一个断点，当这个断点命中时，执行如下命令：

```
0:000> .call MSVCR80D!wprintf(argv[0])
```

因为 WinDBG 总是使用 C++ 表达式评估器来解析命令中的函数参数，所以这里我们使用了 C/C++ 的语法（argv[0]）来引用 dbgee 程序的命令行参数。发出以上命令后，WinDBG 提示如下信息：

```
Thread is set up for call, 'g' will execute.  
WARNING: This can have serious side-effects,  
including deadlocks and corruption of the debuggee.
```

第一行信息表示 WinDBG 已经为函数调用做好了准备，输入 g 命令将执行这个函数。因为是在目标程序中执行要调用的函数，所以需要目标程序恢复执行后这个函数才能执行。后两行信息警告我们，这样调用目标程序中的函数可能有严重的副作用，包括导致被调试程序死锁和崩溃。通常我们应该只调用比较单纯的函数，这个函数不依赖太多其他函数，它执行完某个操作就立刻返回。

输入 g 命令来恢复目标运行，但 WinDBG 很快便又进入命令模式，并显示：

```
.call returns:  
int 51
```

以上信息的含义是函数调用的返回值是整数 51（int 51）。同时，函数返回值还被放在伪寄存器\$callret 中，可以使用 r 命令来观察：

```
0:000> r $callret  
$callret=00000033
```

对于多线程程序，为了减小副作用，可以使用~.g 命令只恢复当前线程。

30.17.2 工作原理

简单来说，.call 命令采取的主要动作是：

在栈上插入一小段代码，这段代码被用作要调用函数的父函数，让被调用函数执行后返回到这段代码中。这段代码的内容就是中断指令，目的是让被调用函数返回后便立刻触发一个断点中断到调试器中。

在栈上建立一个新的栈帧模拟调用.call 参数中指定的函数，压入指定的参数和返回地址，返回地址的值就是上面那段代码的地址。

修改寄存器，使程序指针寄存器指向要调用函数的起始地址，以便当前线程恢复执行后便执行这个函数。

对于上面的例子，在执行.call 命令前的函数调用关系为：

```
00 0012ff68 00411ad6 dbgee!wmain [c:\dig\dbg\author\code\chap28\dbgee\dbgee.cpp @ 11]
01 0012ffb8 0041191d dbgee!__tmainCRTStartup+0x1a6 [f:\rtm\...\crtexe.c @ 583]
02 0012ffc0 7c816ff7 dbgee!wmainCRTStartup+0xd [f:\rtm\...\crtexe.c @ 403]
03 0012fff0 00000000 kernel32!BaseProcessStart+0x23
```

寄存器内容为：

```
eax=003a5c20 ebx=7ffd4000 ecx=003a2e90 edx=00000001 esi=0125f774 edi=0125f6f2
eip=004113a0 esp=0012ff6c ebp=0012ffb8 iopl=0 nv up ei pl nz na po nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000202
dbgee!wmain:
004113a0 55         push    ebp
```

记下以上内容后，执行.call 命令，然后再观察栈中的函数调用情况，变为：

```
00 0012ff5c 0012ff68 MSVCR80D!wprintf
[f:\rtm\vctools\crt_bld\self_x86\crt\src\wprintf.c @ 49]
WARNING: Frame IP not in any known module. Following frames may be wrong.
01 0012ffb8 0041191d 0x12ff68
02 0012ffc0 7c816ff7 dbgee!wmainCRTStartup+0xd [f:\rtm\...\crtexe.c @ 403]
03 0012fff0 00000000 kernel32!BaseProcessStart+0x23
```

可见，此时 WinDBG 已经模拟好了当前线程在执行 wprintf 函数的假象。第 3 行的警告信息是说 wprintf 函数的返回地址 0012ff68 不属于任何模块，这是因为这个返回地址是指向栈上那小段代码的，当然不属于任何模块，执行 u 命令可以看到它的内容：

```
0:000> u 0012ff68
0012ff68 cc          int     3
0012ff69 ebfd        jmp     0012ff68
0012ff6b cc          int     3
0012ff6c d6          ???    
0012ff6d 1a4100      sbb     al,byte ptr [ecx].....
```

从前面的寄存器信息我们知道执行.call 命令前的栈顶指针为 0012ff6c，所以那小段代码其实就是 0012ff68 到 0012ff6b 的三条指令（占四个字节）。

此时的寄存器内容为：

```
eax=003a5c20 ebx=7ffd4000 ecx=003a2e90 edx=00000001 esi=0125f774 edi=0125f6f2
eip=1029ed60 esp=0012ff60 ebp=0012ffb8 iopl=0 nv up ei pl nz na po nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000202
MSVCR80D!wprintf:
1029ed60 55         push    ebp
```

可见程序指针寄存器已经被修改为 1029ed60，即 wprintf 函数的地址，另外栈顶指

针寄存器 ESP 也变了，因为需要为 wprintf 函数压入返回地址和参数。使用 dd 显示栈上的原始数据：

```
0:000> dd 0012ff60
0012ff60 0012ff68 003a2e98 ccfdebcc 00411ad6
0012ff70 00000001 003a2e90 003a5c20 a87c6bd4
```

其中 0012ff68 是函数返回地址，即那小段代码的地址，003a2e98 是参数 argv[0]，第 3 个 DWORD (ccfdebcc) 就是那小段代码的机器码，00411ad6 是 main 函数的返回地址，再后面是 main 函数的参数。

通过以上分析我们便很容易理解.call 命令的工作原理了，简单来说，WinDBG 就是在当前线程的栈上模拟出函数调用现场，将参数、返回地址和寄存器等准备好，然后恢复目标执行，目标线程恢复运行后便执行这个要调用的函数。

30.17.3 限制条件和常见错误

首先，.call 命令只能在调试用户态活动目标时使用，不能用在内核态调试和调试转储文件的情况。其次，必须有被调用函数的私有符号，也就是包含类型信息的函数符号。如果对没有类型符号的函数执行.call 命令，那么 WinDBG 会提示错误：

```
0:000> .call dbgee!wprintf(argv[0])
          ^ Symbol not a function in '.call dbgee!wprintf(argv[0])'
```

另外，每个线程一次只能调用一个函数，只有当这次调用结束后，才能再调用另一个，否则便会得到如下错误：

```
0:000> .call MSVCR80D!wprintf(argv[0])
          ^ Thread already has call in progress error in '.call
MSVCR80D!wprintf(argv[0])'
```

但可以使用.call /C 来清除当前线程中的函数调用。

30.18 命令程序

类似于 DOS 或 Windows 控制台中的批处理命令文件和脚本文件，WinDBG 也支持把一系列调试器命令放在一个文件中，然后以文件的形式提交给调试器来执行，这样的文件被称为调试器命令程序（Debugger Command Program），简称命令程序。在命令程序中，除了可以使用 WinDBG 的标准命令、元命令和扩展命令外，还可以使用专门用来控制执行流程的流程控制符号，使用别名和各种伪寄存器来充当变量，下面我们分别来介绍。

30.18.1 流程控制符号

模仿 C/C++语言中的流程控制关键字，WinDBG 定义了一系列元命令和扩展命令

来实现流程控制，统称为流程控制符号（Control Flow Token），列举如下。

- 用作分支和判断的.if、.else 和}elseif。
- 用作循环的.do、.while、!for_each_module、!for_each_frame 和!for_each_local。以及用在循环体中的.break 和.continue。
- 捕捉异常的.catch 和从.catch 块中退出的.leave。
- 定义代码块的.block。因为大括号（{}）已经在别名和很多命令中有用途，所以不可以单独使用大括号来定义代码块。

以上符号的用法大多与 C/C++语言中的同名关键字类似，我们不做详细说明，稍后我们会通过几个例子来演示它们的用法。

30.18.2 变量

编写程序总离不开变量，在 WinDBG 命令程序中可以使用如下几种变量。

- 自动的伪寄存器，即 WinDBG 调试器内部已经定义好的模拟寄存器，比如\$peb、\$ip 等。这些寄存器不需要定义和初始化，可以直接使用，WinDBG 会自动将它们替换为合适的值。
- 用户赋值的伪寄存器，共有 20 个，\$t0~\$t19。这些伪寄存器的默认类型是整数。可以使用 r 命令为其赋值，但也可以使用 r?命令让其自动获取所赋参数的类型。比如 r \$t1 = 7 是将\$t1 赋值为整数 7；r? \$t2 = &@\$peb->Ldr 是让\$t2 获取 Ldr 字段的类型_PEB_LDR_DATA，&符号用来取地址，含义与 C++中相同，如果有&符号，那么\$t2 的值为 Ldr 字段的内存地址。举例来说，如果 PEB 结构的地址为 7ffd000，那么\$t2 的值为 7ffd00c，因为 Ldr 字段的偏移为 0xC，如果不带&符号，那么\$t2 的值就是 Ldr 字段的内容。
- 用户定义的别名，可以通过 as 命令来定义别名，然后使用，不用时使用 ad 命令删除。
- 自动别名，如\$ntsym、\$CurrentDumpFile 等。
- 固定名称的别名，共有 10 个，分别为\$u0~\$u9。定义固定名称别名的等价量时应该在 u 前加一个点（.），如 r \$.u5="dd esp; g"。

如果使用的表达式评估器为 MASM，那么引用伪寄存器的方法有两种，一种是在 \$符号前加@符号，另一种是不加@符号，但如果使用 C++表达式评估器，那么一定要加@符号。引用（解释）别名的典型方式是使用\${ }。

30.18.3 命令程序示例

下面通过一个例子来说明命令程序的编写方法。清单 30-22 列出了一个典型的命令程序，它可以按照加载顺序列出当前进程中的所有模块。

清单 30-22 命令程序示例

```

1   $$ Get module list LIST_ENTRY in $t0.
2   r? $t0 = &@$peb->Ldr->InLoadOrderModuleList
3
4   $$ Iterate over all modules in list.
5   .for (r? $t1 = *(ntdll!_LDR_DATA_TABLE_ENTRY**)@$t0;
6       (@$t1 != 0) & (@$t1 != @$t0);
7       r? $t1 =
8       (ntdll!_LDR_DATA_TABLE_ENTRY*)@$t1->InLoadOrderLinks.Flink)
9   {
10      $$ Get base address in $Base.
11      as /x ${/v:$Base} @@c++(@$t1->DllBase)
12
13      $$ Get full name into $Mod.
14      as /msu ${/v:$Mod} @@c++(&@$t1->FullDllName)
15
16      .block
17      {
18          .echo ${$Mod} at ${$Base}
19      }
20
21      ad ${/v:$Base}
22      ad ${/v:$Mod}
}

```

在以上清单中，1、4、9、12 行都是注释行，尽管*也可以开始一个注释行，但是在命令程序中通常需要使用\$\$，其原因是命令程序执行时会将自动合并为一行，换行符被替换为分号，而*注释符是注释整行，\$\$是注释到分号为止。

第 2 行是把当前进程_PEB 结构(使用伪寄存器\$peb 代表)的 Ldr 子结构的 InLoadOrderModuleList 字段的地址赋给\$t0 伪寄存器，并使之自动获得 InLoadOrderModuleList 字段的类型_LIST_ENTRY。

第 5~7 行是开始一个 for 循环，与 C++ 中的 for 语句类似，第 5 行是给循环变量(伪寄存器\$t1)赋初值，也就是让\$t1 等于\$t0 所代表地址处的值，并且把这个内容转换为 ntdll!_LDR_DATA_TABLE_ENTRY* 结构。第 6 行是循环条件，即 Flink 的值(\$t1)不为空，并且 Flink 不等于\$t0 所代表的起始节点地址，这是遍历链表的典型判断方法。第 7 行是更新循环变量，为下一轮循环做准备。

第 8~22 行是循环体，也就是显示\$t1 所指向的_LDR_DATA_TABLE_ENTRY 结构的内容。

第 10 行是定义一个用户命名的别名\$Base，用其表示模块的基地址。其中的@@c++ 用来强制使用 C++ 表达式评估器，/x 是使这个别名取后面表达式的 64 位值，/v 用来阻止别名替换，不管其是否已经定义，因为这句尚是在定义别名阶段，省略亦可，加上更稳妥。

第 13 行是定义另一个别名 Mod，使其值等于 FullDllName 字段的地址。其中的/msu 用来使别名 Mod 等价于后面地址处的 UNICODE_STRING，因为它要求后面是一个地址，所以小括号中的取地址符号&不能省略，否则便会产生如下错误：

```
0:000> as /msu ${/v:$Mod} @@c++(@$t1->FullDllName)
Type conflict error at '@@c++(@$t1->FullDllName)'
```

第 15~18 行定义了一个块，尽管其中只有一行命令，这个块定义仍是必要的，它起到的作用是强制评估块中的所有别名。第 17 行的是用.echo 命令显示别名\$Mod 和 \$Base 的值。第 20 和 21 行是删除别名定义，然后开始下一轮循环。

30.18.4 执行命令程序

将清单 30-21 所示的内容保存为一个文件，然后便可以在 WinDBG 中通过如下命令来执行它：

```
$><c:\dig\dbg\author\code\chap30\lm.dbg
以下是在调试 dbgee 程序的调试会话中的执行结果:
C:\dig\dbg\author\code\chap28\dbgee\Debug\dbgee.exe at 0x400000
C:\WINDOWS\system32\ntdll.dll at 0x7c900000
C:\WINDOWS\system32\kernel32.dll at 0x7c800000
C:\WINDOWS\WinSxS\x86_Microsoft.VC80...-ww_f75eb16c\MSVCR80D.dll at 0x10200000
C:\WINDOWS\system32\msvcrt.dll at 0x77c10000
```

\$><的含义是让 WinDBG 读取后面的文件，并将其中的内容浓缩成一个单一的命令，然后执行，浓缩时，WinDBG 会自动把换行符替换为分号。

如果不希望 WinDBG 进行浓缩处理那么可以将\$><换为\$>，这时 WinDBG 每次从文件中读取一行，然后执行。对于我们上面的 lm.dbg 文件，这样执行到第 5 行时由于这一行并没有包含完整的.for 命令，便会出错。

也可以使用\$\$><或者\$\$<来执行一个命令文件，它们与\$><和\$<的差异就是允许在文件名前有空格，并允许使用双引号包围文件名。

如果要为命令文件指定参数，那么可以使用如下形式：

```
$$>a< Filename arg1 arg2 arg3 ... argn
```

另外，在命令程序中可以像使用别名那样使用参数，比如\${\$arg1}。

30.19 本章总结

本章比较详细地介绍了 WinDBG 调试器的用法，覆盖了常用的功能和命令。WinDBG 是个多用途的调试器，因为篇幅有限，我们没有按照调试目标分别介绍 WinDBG 的每一种用途，而是集中精力介绍了普遍适用于大多数调试任务的一般知识和要领。

WinDBG 的帮助文件是学习 WinDBG 的一个宝贵资源，它详细介绍了 WinDBG 的所有功能和命令。本章的目的是帮助大家更好地使用帮助文件，弥补帮助文件的不足，而不是替代它。首先，本章对数百万字的帮助信息进行了归纳和浓缩，使读者可以在较短的时间内了解到 WinDBG 的全貌。另外，我们选取了帮助文件中介绍较少或

较难理解的内容，比如遍历链表（第 30.16 节）、事件处理（第 30.9 节）和调试上下文（第 30.7 节）等。建议大家在阅读本章时和日常调试时都经常打开帮助文件，慢慢加深对每个命令和功能的理解。

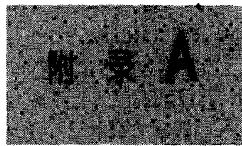
除了 WinDBG 的帮助文件，WinDBG 工具包中还包含了以下几个文档。

- `kernel_debugging_tutorial.doc`: 位于 WinDBG 的根目录中，详细介绍了如何使用 WinDBG 进行内核调试。
- `symhttp.doc`: 位于 `symproxy` 子目录中，介绍了如何建立符号服务器。
- `srcsrv.doc`: 位于 `sdk\srcsrv` 子目录中，详细介绍了源文件服务器的概况和如何建立和配置源文件服务器。
- `dml.doc`: 位于 WinDBG 的根目录中，介绍了 DML (Debugger Markup Language) 的用途和编写方法。DML 是一种标记语言，用于标记 WinDBG 或扩展命令的信息输出。
- `themes.doc`: 位于 `themes` 目录中，介绍了主题 (theme) 的概念（一个主题代表一套特定风格的界面布局和工作空间配置）和如何加载及使用该目录中的四套主题配置。
- `tools.doc`: 位于 WinDBG 的根目录中，介绍了 PDBCopy 和 dbh (DbgHelp Shell) 两个 WinDBG 附带的命令行工具的用法，前者主要用来复制调试符号，后者用来调用调试辅助库 (`DbgHelp.dll`) 的各种功能，包括处理模块和调试符号。
- `pooltag.txt`: 位于 `triage` 目录中，包含了 Windows 内核模块和驱动程序所使用的内存分配标记 (Pool Tag)。在启用了 Windows 操作系统的内存池标记 (Pool Tagging) 功能后 (Windows Server 2003 开始永久启用，之前的 Windows 需要用 GFlags 来启用)，系统会为每个内存块维护一个分配标记来标识它的使用者。用于显示内存池使用情况的扩展命令 `!poolused` 就是使用这个文件来查找每个分配标记所对应的模块。

本书的网站上列出了关于 WinDBG 的更多文档和资源，在此不再一一列举。

参考文献

1. Debugging Tools for Windows 帮助手册. Microsoft Corporation
2. WinDBG SDK 中的 Debug Help Library 文档 (`\sdk\help\dbghelp.chm`). Microsoft Corporation



示例程序列表

程序名称	用途	章节
Err2Fail.exe	演示在特定条件下才表现出来的错误	1.6.1
AccKernel.exe	从用户空间访问内核空间	2.5.2
AcsVio.exe	写代码段导致非法访问异常	2.5
ProtSeg.exe	应用程序代码不可以直接修改段寄存器	2.5
Fault.exe	使用结构化异常处理器处理除零异常后恢复程序运行	3.3.3
B2BStep.exe	分支到分支单步执行	4.3.4
Hlnt3.exe	在代码中插入断点指令	4.1.1
DataBP.exe	手工设置数据断点	4.2.8
TryInt1.exe	在用户态代码中插入 INT 1 指令会违反保护规则	4.3.1
CpuWhere.exe	使用 CPU 的调试存储机制记录 CPU 的执行路线	5.4
Bts.sys	支持 CpuWhere 的驱动程序	5.4
LBR.dll	使用分支记录功能的 WinDBG 扩展模块	5.2.3
McaViewer.exe	读取 MCA 寄存器	6.3.2
Breakout.exe	试验应用程序自己调用 DbgUiRemoteBreakin 的效果	10.6.4
DebString	用于验证 OutputDebugString API 的工作原理	10.7
EvtFilter.exe	用于试验 VC 调试器的异常处理选项	10.5.5
HungWnd.exe	用于观察被中断到调试器后的程序窗口	10.6.9
MiniDbgee.exe	用作调试目标的简单 Win32 程序	10.4.2
TinyDbge.exe	用作调试目标的简单控制台程序	10.4.2
TinyDbgr.exe	使用调试 API 编写的简单调试器	10.4.2
SEH_Excp.exe	探索 SEH 的异常处理	11.4.3
SEH_Trmt.exe	探索 SEH 的终结处理	11.4.2
SEH_Mix.exe	嵌套使用 SEH 的异常处理和终结处理	11.4.6
VEH.exe	演示向量化异常处理器的用法	11.5.3
JitDbgr.exe	一个简单的 JIT 调试器	12.5.3
UdmpView.exe	读取和解析用户态转储文件	12.9.4
UEF.exe	触发未处理异常的控制台程序	12.1
UefWin32.exe	触发未处理异常的窗口程序	12.1
UefSndThrd.exe	在第二个线程中触发未处理异常的控制台程序	12.1

续表

程序名称	用途	正文
UefSrvc.exe	触发未处理异常的系统服务程序	12.1
UefCSharp.exe	触发未处理异常的.NET 程序	12.1
UefSilent.exe	不显示应用程序错误对话框	12.4
ErrorMode.exe	观察 SetErrorMode API 的效果	13.6.1
HiCLFS.exe	使用 CLFS API 创建日志文件和读写日志记录	15.6
Crimson.exe	演示 Crimson API 的用法	16.9
ETW.exe	演示使用编程方法控制 NT Kernel Logger	16.7.2
RawLog.exe	不使用清单文件而直接输出日志信息	16.9
KdTalker.exe	与内核调试引擎的对话程序	18.5.7
Verifiee.exe	探索程序验证器的分析目标	19.4.1
AllcStk.exe	演示栈的创建过程和栈溢出	22.2.3
BoAttack.exe	缓冲区溢出攻击的基本原理	22.10.2
BufOvr.exe	存在缓冲区溢出错误的小程序	22.10.1
CallConv.exe	包含各种函数调用协议的小程序	22.7
CallCV64.exe	演示 64 位系统下的函数调用协议	22.7.6
CheckESP.exe	不遵守栈平衡原则的小程序	22.6
HiStack.exe	用于观察栈的小程序	22.3.3
LocalVar.exe	用于观察局部变量的小程序	22.4.1
SecChk.exe	演示编译器的安全检查功能	22.11
StackChk.exe	演示栈检查函数的工作原理	22.8.3
StackOver.exe	通过死循环导致栈溢出的小程序	22.8.2
StkUFlow.exe	存在栈下溢错误的小程序	22.9
MemLeak.exe	使用 CRT 的调试支持自动转储内存泄漏	23.15
FreCheck.exe	用于分析释放堆块时触发的堆检查	23.8.3
HeapHFC	演示 Win32 堆的释放检查 (HFC) 机制	23.6.2
HeapMfc	演示内存泄漏的 MFC 程序	23.7
HeapOver	演示发生在堆上的缓冲区溢出	23.8
HiHeap.exe	用来分析基本内存分配和释放操作的控制台程序	23.3
SBHeap.exe	使用 CRT 的小堆块堆	23.11.2
Interop.exe	用于分析在同一个程序中使用两种异常处理机制	24.7
SehComp.exe	用于分析 SEH 异常处理编译方法的调试目标	24.5.2
SehRaw.exe	手工注册异常处理器	24.4.1
VC8Win32.exe	用于分析异常处理有关的安全问题	24.6
HiWorld.exe	VC2005 产生的典型 Windows 程序	25.4.1
PdbFairy.exe	直接读取 PDB 文件的小程序	25.6.5
Sig2Time.exe	将 PDB 文件中的时间戳转换为时间	25.8
SymOption.exe	试验不同的符号文件选项	25.2
SymView.exe	符号文件观察器	25.6.8
D4D.dll	演示可调试设计的 DLL 模块	27.4
D4dTest.exe	使用 D4D.dll 的测试程序	27.4
PerfView.exe	演示性能监视程序的工作原理	27.5.3
MulThrds.exe	用于演示线程控制命令的调试目标	30.13.1

附录 B

WinDBG 标准命令列表

	命令	功能	正文
A	a	汇编	
	ad, aS/as, al	删除、定义和列出别名	30.4.3
	ah	控制断言处理方式	
B	ba	设置硬件断点	30.12.2
	bp, bu, bm	设置软件断点	30.12.1
	bl, be, bd, bc, br	管理断点	30.12.6
C	c	比较内存	
D	da, db, dc, dd, dD, df, dq, du, dw, dW, dyb, dyd	显示内存, d 后的字符用来指示显示格式, a 代表 ASCII 码, b 代表字节, c 代表 ASCII 和双字, d 代表双字, D 代表双精度浮点, f 代表单精度浮点, w 代表字, W 代表字和 ASCII 码, y 代表二进制	30.15.1
	dda, ddp, ddu, dpa, dpu, dpp, dqa, dqp, dqu	显示被引用的内存	
	dds, dps, dqs	显示内存和符号	
	dg	显示段选择子	2.6.4
	dl	显示链表	30.16.4
	dv	显示局部变量	30.14.2
	ds, dS	显示 STRING、ANSI_STRING 或者 UNICODE_STRING 类型的结构	30.15.2
	dt	显示数据类型	30.15.3
E	e, ea, eb, ed, eD, ef, ep, eq, eu, ew, eza, ezu	编辑内存	30.15.5
F	f, fp	填充内存区, fp 用来填充物理内存	
G	g, gc, gh, gn, gu	恢复执行	30.10.3
I	ib, iw, id	读 IO 端口	
J	j	根据指定条件选择执行一组命令, Execute If - Else	30.4.4
K	k, kb, kd, kp, kP, kv	显示栈回溯	30.14.1
L	l+, l-	设置源文件选项	
	ld	加载调试符号	30.8.4
	ls, lsa	显示源代码	

续表

命令	功能	正文
lm	显示已经加载的模块	30.8.5
ln	显示相邻的符号	30.8.7
lsf, lsc	加载和显示源文件	
M m	移动内存	
N n	设置数字基数	30.4.1
O ob/ow/od	写 IO 端口	
P p, pa, pc, pt	单步执行	30.11
Q q, qq, qd	退出调试会话	30.6
R r	读写寄存器	30.7.3
rdmsr	读 MSR 寄存器	
rm	设置寄存器显示掩码	
S s	搜索内存	30.15.4
so	设置内核调试选项	
sq	设置静默模式	
ss	设置符号后缀	
sx, sxd, sxe, sxi, sxn, sxr	设置调试事件处理方式	30.9.3
T t, ta, tb, tc, tct, th, tt	追踪执行	30.11
U u, ub, uf, ur, ux	反汇编	
V version	显示调试器和扩展模块的版本信息	
vertarget	调试目标所在系统的版本信息	
W wrmsr	写 MSR 寄存器	
wt	追踪执行	30.11.5
X x	显示调试符号	30.8.6
Z Z	循环执行, 即 Execute While	30.4.4
	显示调试目标所在的系统信息	
I I, <n> s	显示和切换被调试进程, <n>是进程编号	
#, ., <n>	放在命令前限定这个命令所针对的进程	30.4.5
~ ~, ~#s	显示和切换被调试线程, <n>是线程编号	
~<n>f, ~<n>u, ~<n>n, ~<n>m	控制线程, <n>是线程编号	30.13.2
~<n>	放在命令前限定这个命令所针对的线程	30.4.5
?	显示标准命令列表	30.2.1
? <MASM 表达式>	评估使用 MASM 语法的表达式	30.4.1
??	评估 C++ 表达式	30.4.1
\$	执行命令程序	30.4.1
\$ \$	注释	30.4.1
!	所有扩展命令的起始符号	30.2.3
.	所有元命令的起始符号	30.2.2



/noumex, 287, 476, 482
8086 Monitor, 102~105, 841, 842, 844
ARM, 29
ASSERT, 571~574, 578
BCD, 476
BeingDebugged, 173, 209, 231~233, 236, 237, 241, 253, 269, 952
 观察, 232
 设置, 204
BIOS, 36, 63, 73, 83, 86, 138, 147, 161, 449
BOOT.INI, 475
Bootvis, 18, 439
BSOD, 145, 287, 359, 366, 795
Bug
 ~的生命周期, 22
 第一个~, 4
CALL 指令, 113, 590
CIM, 429, 822
CISC, 29
CodeView, 842, 544, 743
COFF, 742
CONTEXT, 16, 92, 283, 306
 Context Switch, 177, 582
Cookie, 547, 636
CreateProcess, 234
Crimson, 440, 443
CRT 堆, 688
 ~的三种模式, 688
 ~的分配函数, 651
 ~的调试功能, 698
 ~的调试堆块, 692
 分配堆块, 645
 创建~, 645
 泄漏转储, 704
 堆块转储, 693, 700
CSRSS, 215, 214
 GPF, 51
 子系统服务器, 213
 会话, 175, 174
 创建~, 189
 杜撰消息, 210
 服务, 214, 216, 217, 241, 257
 终止, 266
 调试子系统, 210, 195
 调试支持, 221
调试端口, 200, 232, 234, 241,
 ~的职责, 194
 硬错误, 359
 错误端口, 362
DbgSsReserved, 204, 213, 221, 222, 229, 230, 235, 240, 267, 268, 270
DBG 文件, 762
DBWIN, 261
DDI, 368
DDT, 837
debug
 定义, 3
DebugActiveProcess, 240
Debuggee, 194
DebugPort, 210, 231, 232, 236, 241, 266, 267, 269, 270, 287, 928
 EPROCESS, 169, 170, 171
 发送消息, 200
 创建进程, 235
 观察~, 232
 设置调试对象, 204
 进程的~, 78
 采集调试消息, 196
 被调试进程, 231
DFD, 783
Dr. Watson, 309
 ~的日志文件, 309
DR6, 44, 84, 88, 89, 91, 92, 96, 98,
 DWARF, 745
 dwwin, 331
 EBP, 40
 EnC, 850
 ERST, 457
 ETW, 18, 797
 ~的架构, 422
 FLIT, 835, 847, 835
 FPO, 604
 FPU, 33
 FS:[0]链条, 288
 GLS, 436
 GPF, 328
 GPR, 40
 Hard Error, 360
 IA-32e 模式, 37, 39, 40, 42, 51, 52, 114, 117
 ICE, 148
IDLE 进程, 187
intrinsic, 292, 778
ITP, 158
ITP700, 159
JIT 调试, 309
 /CRASHDEBUG, 370, 476
 /GS, 570
 VS, 546, 547
WinDBG, 872, 923
启动~, 234
举例, 339
配置~, 334
等待~, 346
编写~, 338
JTAG, 34, 147~162
KD, 868, 103
KiDispatchException, 200, 205, 248, 260, 261, 283~291, 496
Loader, 199, 527
LSASS, 189
MCA, 34, 42, 133~145, 162
MOF, 429
MSR, 35
MSVC, 544
MTRR, 36
NKL, 432
NMI, 65, 69, 72, 137, 278, 279
Noninvasive, 925
NTDLL, 190, 198
NTLDR, 186, 274, 464
NTOSKRNL, 198, 290
NTSD, 868
PDB, 739
 ~文件, 744
 数据表, 756
PDE, 56, 987
 地址翻译, 57
 格式, 56
PEB, 173
PE 文件, 753
PE 格式, 542
PFN, 175
POSIX 子系统, 190
PTE, 57
RET 指令, 590
RISC, 29

- SAL, 541
 SCI, 457
 SEH
 SAFESEH, 718, 735
 异常处理, 294
 异常处理块, 299
 过滤表达式, 295
 终结处理, 291
 编译, 311
 SMI, 39
 SMSS, 189, 210-212, 214, 218, 219, 220, 222-223, 235, 381
 调试子系统, 195
 SoftICE, 148, 461, 844, 845, 859
 Stop Code, 368
 SYMDEB, 842
 Syser, 461
 TAP, 149
 TEB, 54
 TIB, 287
 TryLevel, 728
 TSC, 34
 TSS, 276
 Turbo Debugger, 843
 UMDH, 668
 UnhandledExceptionFilter, 328
 USB, 470
 UST, 662
 UST 数据库, 666
 VEH, 302
 示例, 304
 调用~, 303
 登记和注销, 302
 WaitForDebugEvent, 204, 218, 221, 245, 896
 VC, 544
 WinDBG, 228
 工作过程, 458
 示例, 241
 同步, 196
 调用, 230
 基本框架, 229
 WER, 359
 ~ 1.0, 392
 ~ 2.0, 328
 WHEA, 145, 185, 445
 WinDBG, 867
 ~诞生, 868
 ~的用户界面, 911
 ~的版本历史, 872
 ~的标准命令, 908
 重构, 875
 ~的调试会话循环, 894
 调试器引擎, 881
 溯源, 867
 输入和执行命令, 916
 Windows, 165
- WPP, 430
 XDP, 158, 160
 XDP-SSA, 160
 上下文, 930
 工作空间, 905
 门描述符, 275
 不可调试代码, 792
 中断, 65
 处理, 73
 定义, 1
 内存泄漏, 513
 定位, 4, 278
 检查点, 700
 解决方法, 391
 默认堆, 646
 内存管理器, 55
 内核态调试, 845, 6
 WinDBG 及其实现, 867
 内核空间, 185
 内核调试, 14
 与内核交互, 492
 本地~, 7
 协议, 25
 内核模式, 176
 分支监视, 107
 BTS, 113
 LBR, 108
 历史, 122
 概览, 107
 分配挂钩, 699
 反汇编, 19
 日志, 17
 CLFS, 18
 ELF, 405
 定义, 3
 概述, 478
 火线, 857
 代码注入, 821
 用户空间, 189
 用户模式, 176
 会话管理器. *See* SMSS, 189
 全局标志, 217, 527, 662
 华生医生. *See* Dr. Watson, 343
 名称修饰, 739
 向量化异常处理. *See* VEH, 302
 寻址方式, 31
 异步阻停, 252
 异常, 65
 CPU~, 65
 CPU~分类, 67
 CPU~列表, 69
 分发过程, 284
 处理~, 73
 未处理~, 309
 优先级, 72
 登记 CPU~, 283
 登记软件~, 283
- 异常处理, 276
 基于表的~, 737
 ~的两轮机会, 945
 异常处理器, 311
 内嵌~, 281
 寻找~, 288
 登记~, 302
 默认的~, 288
 扩展命令, 910
 池管理器, 645
 初始断点, 951
 启动过程, 147
 局部变量, 86
 快速系统调用, 179
 运行库, 311
 ~的初始化, 311
 C/C++~, 545
 链接~, 542
 运行期检查, 569
 进程, 167
 进程空间, 167
 远程调试, 14, 873, 927
 DDT-11, 841
 MSC 7.0, 544
 remote, 880
 VC6, 545
 WinDBG, 867, 868, 869, 872, 873, 876, 880, 893
 ~的工作空间, 906
 ~的方法, 927
 产品期调试, 14
 传输层, 883, 887
 连接, 927
 连接层, 886
 参数, 973
 ~的命令, 909
 ~的实现, 872
 服务层, 886
 退出~, 906
 ~的选项, 943
 事件追踪, 18
 目标, 12
 单步执行, 75
 ~标志, 95
 分支到分支~, 98
 高级语言的~, 97
 实模式, 38, 45, 53, 73, 74, 77, 78, 100, 101, 102, 103, 105, 844
 内存, 33
 启动过程, 478
 性能监视, 123
 泄漏, 668
 环境子系统, 190
 转储文件, 351
 用户态~, 351
 系统~, 371
 调试~, 926

- 转储挂钩, 702
 软件调试
 ~的分类, 12
 ~的历史, 8
 ~的定义, 3
 ~的特征, 6
 ~的基本过程, 5
 顶层过滤函数, 340
 保护模式, 38, 40, 46
 INT 3, 75
 INT n, 96
 LIDT, 273
 SoftICE, 844
 Windows, 165
 中断处理, 858
 分页机制, 55
 引入, 33
 异常, 66
 启用, 43
 启动过程, 478
 段, 50, 55
 段寄存器, 45, 53
 调试, 75
 断点, 75
 前端总线, 63, 107, 108, 109, 116, 133, 135
 复合文件, 764
 帧指针, 602
 指令, 29
 特权~, 49
 标准标注语言. *See SAL*, 555
 栈, 581
 ~的定义, 582
 ~溢出, 617
 ~下溢, 623
 内核态~, 582
 分配检查, 620
 用户态~, 582
 创建过程, 585
 观察~, 973
 ~增长, 616
 ~回溯, 973
 ~帧, 595
 ~指针检查, 606
 段选择子, 45, 48, 49, 52-55, 61, 70, 275, 277-279, 590, 908
 段描述符, 48, 50-55, 61, 62, 179, 275, 277, 278, 594
 结构化异常处理. *See she*
 逆向工程, 25
 逆向调用, 182, 183, 190, 280, 583
 海森伯效应, 161, 194, 853, 854, 855, 857, 925
 被调试进程, 231
 调用门描述符, 594
 调用协定, 609
 调试
 用户态~, 193
 非入侵式~, 925
 调试 API, 224
 调试子系统, 195
 调试对象, 204
 调试会话, 234
 建立~, 892
 终止~, 266
 终止~, 927
 结束~, 898
 调试异常, 88
 调试事件, 195
 ~处理方式, 249
 ~循环, 246
 WinDBG, 891
 处理~, 243, 944
 回复~, 248
 第一批~, 236
 调试消息
 发送~, 200
 传递~, 204
 杜撰的~, 206
 采集~, 196
 调试热键, 257
 调试寄存器, 84
 BPM, 160
 VC6, 95
 引入, 33, 83
 用途, 18
 示例, 89
 观察, 91, 92
 设置, 90, 92, 93, 94, 204
 访问, 49, 92
 应用, 110, 120, 144, 154, 161, 328
 其他寄存器, 45
 保护, 46, 47
 硬件断点, 964, 965
 调试符号, 739, 933
 ~服务器, 935
 ~表, 750, 775
 bm 命令, 963
 CodeView, 743, 843, 873
 COFF 格式的~, 742
 DbgHelp, 810, 935
 DPH, 662
 EnC, 850
 FPO, 604
 HEAP_ENTRY 的~, 658
 HEAP 结构的~, 656
 kp 命令, 975
 MACRO, 835
 PDB 格式的~, 744
 SymTagEnum, 776
 TD, 843
 VC8, 546
 VS7, 546
 公共~, 744
 ~的分类, 12
 反汇编, 19
 加载~, 500
 发布版本的~, 552
 目标文件中的~, 745
 设置~, 877
 设置~路径, 338
 私有~, 977
 使用~, 933
 复制~, 880, 998
 显示数据, 980
 栈回溯, 18, 375, 850
 缺少~, 933
 检查, 940
 编译选项, 771
 意义, 851, 933
 管理, 859
 调试器
 ~进程退出, 267
 中断到~, 251
 内核~, 259
 ~的分类, 852
 用户态~, 878
 自动启动~, 238
 ~的实现模型, 853
 实模式~, 100
 ~的经典架构, 859
 ~的工作线程, 228
 陷阱帧, 376
 陷阱标志, 95
 验证器
 应用程序, 514
 驱动程序~, 514
 驱动程序静态~, 554
 堆, 643
 CRT~, 688
 LFH, 661
 内部结构, 654
 页~, 677
 私有~, 647
 准页~, 683
 ~尾检查, 674
 销毁~, 648
 寄存器, 40
 ~上下文, 931
 64 位~, 46
 MSR~, 42
 TAP~, 151
 伪~, 919
 ~的定义, 40
 标志~, 41
 调试控制~, 116, 85
 控制~, 42
 探测模式, 158
 断言, 571
 断点, 15

- I/O~, 15, 96
- 内存分配~, 698
- 初始~, 237, 251, 505
- 条件~, 965
- 条件转移~, 11
- 使用~, 962
- 软件~, 75, 962
- 指令~, 87
- 硬件~, 83, 94, 964
- 断点异常, 252
 - 8086 Monitor, 102, 104
 - CRT, 560
 - DPH, 677
 - INT 2D, 498
 - INT 3, 75
 - 分发~, 205, 284, 847
 - 分类, 67
 - 代码, 82, 282
 - 处理~, 78, 243, 945
 - 产生~, 67, 77, 224
 - 对~的优待, 80
 - 异常消息, 200
- 应用, 251, 254
- 远程中断, 253
- 轮询, 495
- 俘获调试目标, 952
- 调试服务, 497
- 调试热键, 257
- 调试器, 82, 248
- 堆检查, 643
- 溢出检测, 672
- 触发, 252
- 断点指令, 75, 83
 - 历史, 8
 - 符号表, 775
 - 虚拟 8086 模式, 38
 - 描述符表, 45, 52, 53, 73, 179, 225, 273, 756
 - 程序指针飞跃, 961
 - 缓冲区溢出, 624
 - /GS, 570
 - Cookie, 566
 - SAL, 541
 - SEH, 290
- 示例, 555
- 示意图, 670
- ~的危害, 640
- ~攻击, 591
- 栈上的~, 588
- ~的症状, 664
- 堆上的~, 643
- 堆尾检查, 662
- ~的检查, 646
- 检测~, 670
- 编码, 24
- 编译期检查, 551
- 链表, 987
- 蓝屏. *See* BSOD
 - ~后自动重启, 387
 - 手工触发~, 371
- 解除提交, 653
- ~阈值, 653
- 输出调试信息, 17
 - Windows API, 80
- 管道, 472