Broadview
www.broadview.com.cn

调软试件

Software Debugging

张银奎 著

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

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。

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　筠
责任编辑：陈元玉
印　　刷：
装　　订：
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：6 000 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zlts@phei.com.cn，盗版侵权举报请发邮件至dbqq@phei.com.cn。
服务热线：（010）88258888。

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

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

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要性，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

程序，对于程序员的职业发展也是非常有帮助的。之所以说写给“所有”程序员是因为本书主要讨论的是一般原理和方法，没有限定某种编程语言和某个编程环境，也没有局限于某个特定的编程领域。

第二，本书是写给从事测试、验证、系统集成、客户支持、产品销售等工作的软件工程师或 IT 工程师的。他们的职责不是编写代码，因此软件缺陷与他们不直接相关，但是他们也经常受累于软件缺陷。他们不负责解决问题，但他们需要知道找谁来解决。因此，他们需要把错误定位到某个模块，或者至少定位到某个软件。本书介绍的工具和方法对于实现这个目标是非常有益的。另外，他们也可以从关于软件可调试性的内容中得到启发。本书关于 CPU、操作系统和编译器的内容对于提高他们的综合能力，巩固软硬件知识也是有益的。

第三，本书适合从事反病毒、网络安全、版权保护等工作的技术人员阅读。他们经常面对各种怪异的代码，需要在没有代码和文档的情况下做跟踪和分析。这是计算机领域中最富挑战性的工作之一。关于调试方法和 WinDBG 的内容有利于提高他们的效率。很多恶意软件故意加入了阻止调试和跟踪的机制，本书介绍的软件调试原理有助于理解这些机制。

第四，本书也适合计算机、软件、自动控制、电子学等专业的研究生或高年级本科生来研读。他们已经学习了程序设计、操作系统、计算机原理等课程，阅读本书可以帮助他们把这些知识联系起来，并深入到一个新的层次。学会使用调试器来跟踪和分析软件，可以让他们在指令一级领悟计算机软硬件的工作方式，深入核心，掌握本质，把学到的书本知识与计算机系统的实际情况结合起来；同时，可以提高他们的自学能力，使他们养成乐于专研和探索的良好习惯。软件调试是从事计算机软硬件开发等工作的一项基本功，在学校里就掌握了这门技术，对于以后快速适应工作岗位是大有好处的。

第五，本书是写给勇于挑战软件问题的硬件工程师和计算机用户的。他们是软件缺陷的受害者。除了要忍受软件缺陷带来的不便之外，有时软件设计者还可能将责任推卸给他们，推诿是硬件问题或使用不当。使用本书的工具和方法，他们可以找到充足的证据来证明这是软件的问题。本书的大多数内容不需要很深厚的软件背景，有基本的计算机知识就可以读懂。

最后，或许还有不属于上面 5 种类型的读者也可能会阅读本书。比如，软件公司或软件团队的管理者、软件方面的咨询师和培训师、大学和研究机构的研究人员、非计算机专业的学生、自由职业者、编程爱好者、黑客等等。

前面说过，本书的大多数内容不需要深厚的软件开发背景，但如果读者具备以下基础，将更容易读懂和领会本书的内容：

- 曾经亲自参与编写程序，包括输入代码、编译，然后执行。

- 使用过某一种类型的调试器，用过断点、跟踪、观察变量等基本调试功能。
- 参加过某个软件开发项目，对软件工程有基本的了解。认同软件的复杂性，即开发一个软件产品与写一个 HelloWorld 程序根本不是一回事，

尽管本书给出了一些汇编代码和 C/C++代码，但是其目的只是在代码层次直截了当地阐述问题。本书的目标不是讨论编程语言和编程技巧，也不要求读者已经具备丰富的编程经验。

本书的主要内容

本书共有 30 章，分为以下 6 篇。

第 1 篇：绪论（第 1 章）

作为全书的开篇，这一部分介绍了软件调试的概念、基本过程、分类和简要历史，并综述了本书后面将详细介绍的主要调试技术。

第 2 篇：CPU 的调试支持（第 2～7 章）

CPU 是计算机系统的硬件核心。这一部分以 IA-32 CPU 为例，系统描述了 CPU 的调试支持，包括如何支持软件断点、硬件断点和单步调试（第 4 章），如何支持硬件调试器（第 7 章），记录分支、中断、异常和支持性能分析的方法（第 5 章），以及支持硬件可调试性的错误检查和报告机制——MCA（机器检查架构）（第 6 章）。为了帮助读者理解这些内容，以及本书后面的章节，第 2 章介绍了关于 CPU 的一些基础知识，包括指令集、寄存器和保护模式，第 3 章深入介绍了与软件调试关系密切的中断和异常机制。

第 3 篇：操作系统的调试支持（第 8～19 章）

操作系统（OS）是计算机系统的管理者和软件核心，也是应用软件运行的基础。第 8 章介绍了 Windows 操作系统的基本知识，包括架构、关键模块和系统进程等。然后以 Windows 操作系统为例，描述了操作系统的调试支持，包括如何支持应用程序调试（第 9 章和第 10 章），如何支持调试系统内核和驱动程序（第 18 章），以及支持可调试性的错误提示机制（第 13 章）、错误报告机制——WER（第 14 章）、错误记录机制（第 15 章）、事件追踪机制——ETW（第 16 章）、硬件错误处理机制——WHEA（第 17 章）。第 19 章介绍了提高测试和调试效率的程序验证（Verifier）机制和有关工具。第 11 章介绍了中断和异常的分发与管理。第 12 章介绍了未处理异常和 JIT 调试。

第 4 篇：编译器的调试支持（第 20～25 章）

编译器是软件生产的主要工具，它帮助我们将程序语言翻译为可以被 CPU 所理解的机器码。支持软件调试始终是编译器的一个设计目标。在编译过程中，编译器会帮我们检查程序中的静态错误（编译期检查）（第 20 章）。为了帮助发现只有在运行时才体现出来的问题，编译器可以在程序中插入代码并报告运行时的可疑情况（运行期检

查）（第 21 章）。很多软件缺陷是与局部变量、缓冲区和内存使用有关的，对此，编译器设计了很多种检查和保护栈（第 22 章）及堆（第 23 章）的机制。编译器对软件调试的另一个重大支持就是调试符号。调试符号是软件调试时的灯塔，是观察数据结构和进行源代码级调试所必需的。第 25 章详细介绍了调试符号的产生过程、种类、文件格式和用法。第 24 章介绍了异常处理代码是如何编译的。在介绍以上内容时，本篇还覆盖了有关函数调用规范，栈的布局，以及堆的内部结构等与软件调试密切相关的基础内容。

第 5 篇：可调试性（第 26～27 章）

提高软件调试效率是一项系统工程，除了 CPU、操作系统和编译器所提供的调试支持外，被调试软件本身的可调试性也是至关重要的。这一篇，我们先介绍了提高软件可调试性的意义、基本原则、实例和需要注意的问题（第 26 章）。然后讨论了如何在软件开发实践中实现可调试性（第 27 章），包括软件团队中各个角色应该承担的职责，实现可追溯性、可观察性和自动报告的方法。

第 6 篇：调试器（第 28～30 章）

调试器（Debugger）是软件调试的核心工具。借助调试器，我们可以将软件冻结（中断）在我们指定的位置，然后观察它的内部状态、了解它的运行轨迹和即将执行的操作。根据需要，我们可以分析它的任一条指令，查看它使用的任一个内存单元。分析后，我们可以让它从原来的地方恢复执行，也可以让它“飞”到一个新的地方继续执行，或者干脆将其终止。这一部分分为 3 章。第 28 章介绍了调试器的历史、主要功能、分类方法、实现模型、架构和一个公开标准——HPD（High Performance Debugger）。第 29 章分析了 WinDBG 调试器的架构和主要功能的实现原理。第 30 章分为 18 个主题，系统介绍了 WinDBG 调试器的使用方法。

除了以上 6 篇，附录 A 列出了与本书配套的工具和源程序，附录 B 列出了 WinDBG 的标准命令。

本书的三条线索

本书的内容是根据以下三条线索来组织的。

第一条线索是软件调试的“生态”系统（ecosystem）。我们介绍了这个系统中的所有“成员”，描述了每个成员的职责以及成员间的协作方式。CPU（第 2 篇）为关键的调试功能提供了硬件级的支持；操作系统（第 3 篇）把 CPU 的支持进行必要的封装，并构建一整套软件调试所需的基础设施，然后以 API 的形式提供给调试器和应用软件；编译器（第 4 篇）负责产生更易于调试的调试版本，以及包含调试信息的调试符号文件；被调试软件（第 5 篇）则应该努力提高自身的可调试性；调试器（第 6 篇）负责把所有“成员”的努力化为成果，以简单方便的形式呈现给用户（调试者），让他们利

用强大的调试功能随心所欲地探索软件世界。本书 6 篇的标题就是按照这个线索而设计的，所以这条线索是“明线”。

第二条线索是异常（Exception）。异常是计算机系统中的一个重要概念，出现在 CPU、操作系统、编程语言、编译器、调试器等多个领域，本书逐一对其做了解析。第 3 章从 CPU 一级介绍了异常的作用、分类，以及与中断的关系；第 11 章介绍了 Windows 操作系统管理异常的方法，包括分发异常的详细规则，以及 Windows 的结构化异常处理（SEH）机制；第 12 章介绍了操作系统处置未处理异常的策略和过程；第 24 章从编程语言和编译器的角度探讨了异常，介绍了编译器编译异常处理代码的方法。第 30 章（30.9 节）介绍了调试器“眼”中的异常、调试器处理异常的方法和有关调试命令。

第三条线索是调试器。调试器是解决软件问题最有力的工具。第 1 章介绍了单纯依赖硬件的调试方法。第 4 章分析了 DOS 下的 Debug 调试器的实现方法。第 7 章介绍硬件仿真和基于 JTAG 标准的硬件调试器。第 9 章和第 10 章介绍了 Windows 操作系统下用户态调试器的结构和工作原理，演示了如何使用 Windows 的调试 API 来实现一个简单的调试器。第 18 章介绍了 Windows 内核调试器的工作原理和实现方法。第 28~30 章对调试器做了归纳和更全面的介绍。另外，全书很多地方都使用了调试器输出的结果，穿插了使用调试器解决软件问题的方法。

本书的阅读方法

本书的厚度决定了不适合一口气将它看完。以下是笔者给您的建议。

第一，下载并安装 WinDBG 调试器。如果您还不了解它的基本用法，那么请先浏览第 30 章，学会它的基本用法，能读懂栈回溯结果。有了这个工具后，您就可以跟着做本书所描述的试验，自己在系统中探索书中提到的内容。

第二，选择前面提到的三条线索中的一条来阅读。如果您有充裕的时间，那么可以按第一条线索来阅读。如果您想深入了解异常，那么可以按第二条线索来阅读。如果您有难题等待解决，希望快速了解基本的调试方法，那么您可以选择第三条线索，从第 30 章开始阅读。

第三，先阅读每一篇开始处的简介，了解各篇的概况，浏览主要章节，建立一个初步的印象。当需要时，再仔细查阅感兴趣的细节。

以上意见中，第一条是希望您一定遵循的，其他谨供参考。

本书的写作方法

这是一本来自实践的书，它的大多数内容是依靠软件调试技术探索得到的。在笔者使用的系统中，在一个名为 Toolbox 的文件夹下保存了 100 多个不同功能的工具软

件。当然，使用最多的还是调试器。书中给出的大多数栈回溯结果是使用 WinDBG 调试器产生的。

写作本书的一个基本原则是从有代表性的实例出发，然后从这个实例推广到其他情况和一般规律。例如，在 CPU 方面笔者选择的是 IA-32 CPU；在操作系统方面选择 NT 系列的 Windows 操作系统；在编译器方面是 Visual Studio 系列；在调试器方面选择的是 Visual Studio 调试器和 WinDBG。

示例、工具和代码

本书的示例、工具和代码可通过以下链接免费下载：*http://advdbg.org/books/swdbg/*。

尽管笔者和编辑已经尽了最大努力，但是本书中仍然可能存在这样那样的错误，读者可以通过上面的链接反馈给我们。

关于封面

人们遇到百思不得其解或难以解释清楚的问题时可能不由自主地说：“见鬼了”。在软件开发和调试中也时常有这样的情况。钟馗是传说中的捉鬼能手，因此我们选取他作为本书的封面人物，希望本书能够帮助读者轻松化解“见鬼了”这样的复杂问题。

免责声明

本书的内容完全是作者本人的观点，不代表任何公司和单位。您可以自由地使用本书的示例代码和附属工具，但是作者不对因为使用本书内容和附带资料而导致的任何直接和间接后果承担任何责任。

感谢

首先感谢 Jack B. Dennis 教授，他向我讲述了大型机时代的编程环境和调试方法，以及他和 Thomas G. Stockman 为 TX-0 计算机编写 FLIT 调试器的经过，并专门为本书撰写了短文。FLIT 调试器是作者追溯到的最早的调试器程序。

感谢 Windows 领域的著名专家 David Solomon 先生，他回答了笔者的很多提问，并为本书题写了序言。

感谢 *Showstopper* 一书的作者 G. Pascal Zachary 先生，他允许我引用他书中的内容和该书的照片。

感谢 CPU 和计算机硬件方面的权威 Tom Shanley 先生，他在计算机领域的著作有十几本。感谢他回答了我的很多提问并允许我在本书中使用他绘制的关于 CPU 寄存器的大幅插图（因篇幅所限最终没有使用）。

探索 Windows 调试子系统让我感受到了软件之美，创造这种美的一个主要人物是 Mark Lucovsky 先生，感谢他在邮件中给予我的鼓励。

感谢 DOS 之父 Tim Paterson 先生，他向我介绍了他编写 8086 Monitor 的经过。DOS 系统中的 Debug 调试器来源于 8086 Monitor。

感谢 Syser 调试器的作者吴岩峰先生，我们多次讨论了如何在单机上实现内核调试的技术细节，他始终关心着本书的进度。

感谢我的老板和同事多年来给予我的帮助和支持，他们是：Kenny, Michael, Feng, Adam, Jim, Neal, Harold, Calvin, Cui Yi, Keping, Eric, Yu, Wei, Min, Fred, Rick, Shirley, Vivian, Luke, Caleb, Christina, Starry（请原谅，我无法列出所有名字）！

感谢我的好朋友刘伟力，我们一起加班解决了一个大 Bug 后，他感慨地说：“断点真神奇”，这句话让我产生了写作本书的念头。

感谢曾华军（我们一起翻译了《机器学习》）和李妍，他们帮助我翻译了 Dennis 和 David 所写的序言。

感谢以下朋友阅读了本书的草稿，提出了很多宝贵的意见：王毅鹏、王宇、施佳、夏桅、周祥、李晓宁、侯伟、吴巍。

感谢本书的编辑周筠和陈元玉，感谢他们给我的一贯支持，以及编辑本书所花费的大量时间！感谢责任美编胡文佳，她的精心设计让本书的封面如此美丽！

感谢我的家人，在写作本书漫长而且看似没有尽头的日子里，她们承担了繁重的家务，让我有时间完成本书。

最后，感谢您在茫茫书海中选择了本书，并衷心祝愿您能从中受益！

张银奎（Raymond Zhang）
2008 年 4 月于上海

目 录

样章 目 录

CHAPTER

第 4 章

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。

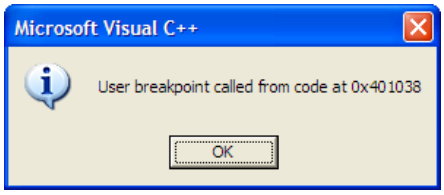

图 4-1 CPU 遇到 INT 3 指令时会把执行权移交给调试器

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。

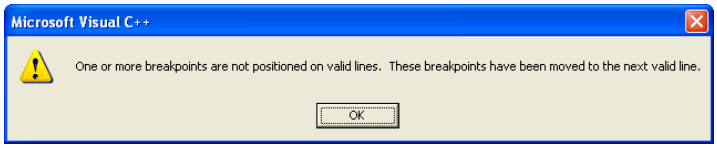

图 4-2 VC6 在开始调试时才真正设置断点，会对无法落实的断点发出提示

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  0000FFFFH *)
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。但是如果不替换，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。

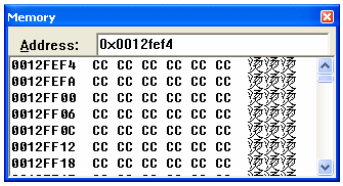

图 4-3 调试版本的运行库会用 INT 3 指令来填充新分配的内存区域

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。这也可以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。

```
lkd> u nt!DbgBreakPointWithStatus
804df8d1 8b442404        mov     eax,[esp+0x4]
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。

4.4 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“=号”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 指令指向前的值，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.

CHAPTER

第 10 章

用户态调试过程

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 debuggee 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，因为可以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 DbgUiIssue-RemoteBreakin 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             efl=00000246
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是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           efl=00000286
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
```

10.9 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*

CHAPTER

第 18 章

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。

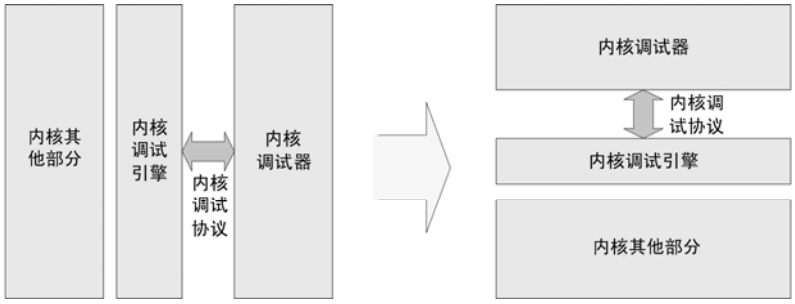

图 18-1 内核调试引擎在内核调试中的角色示意图

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如下几个部分。

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。

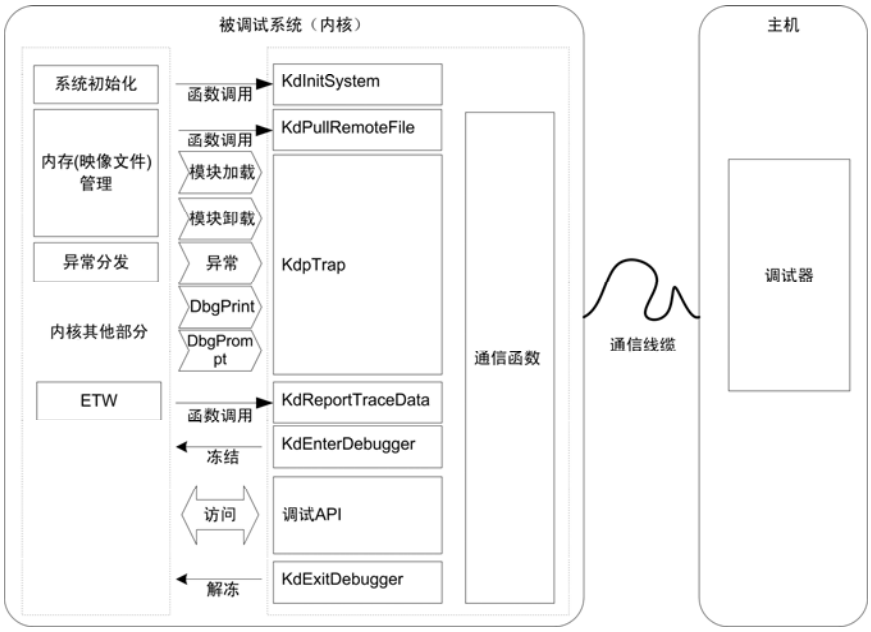

图 18-2 内核调试引擎示意图

18.1.4 模块文件

在 WindowsXP 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。

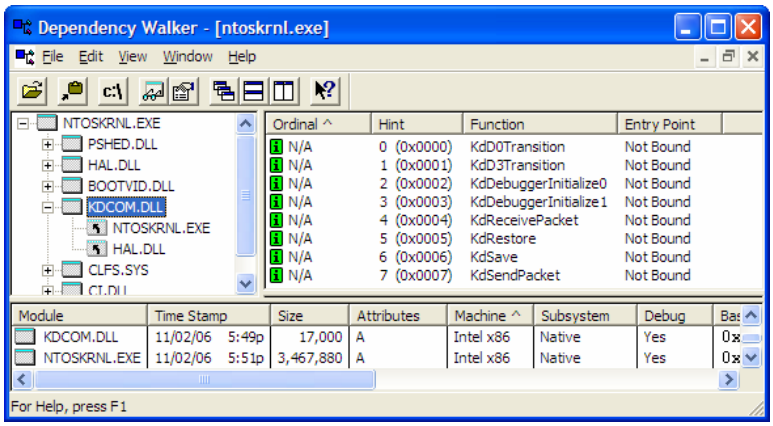

图 18-3 KDCOM 与 NTOSKRNL 之间的相互依赖关系

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。

18.4 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。

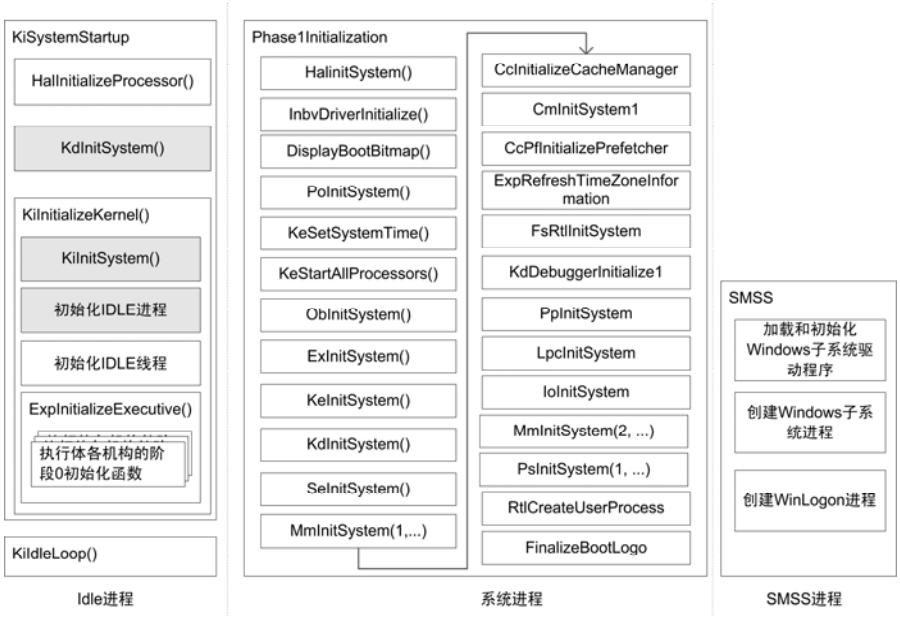

图 18-10 Windows 启动过程概览

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地址、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。

18.9 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 1934 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*

CHAPTER

第 23 章

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、ExAllocatePoolWithQuota、ExFreePool、ExFreePool-WithTag 等。

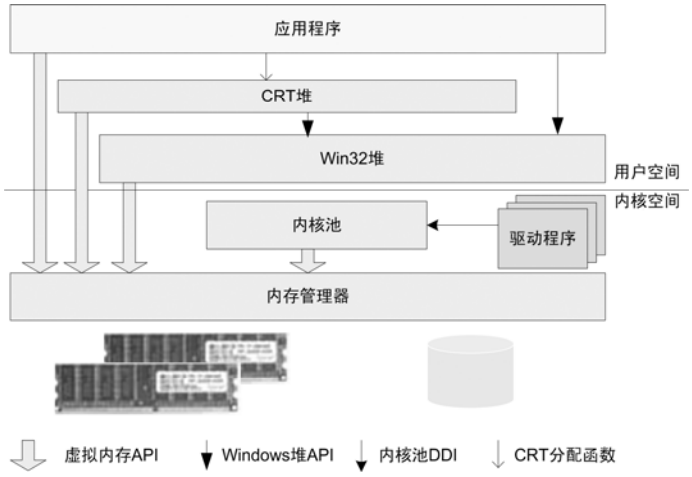

图 23-1 多级的内存分配体系

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 NTORKRNL.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!RtlpExtendHeap 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。

23.9 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（Fense 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 0xEEEEEEEE，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

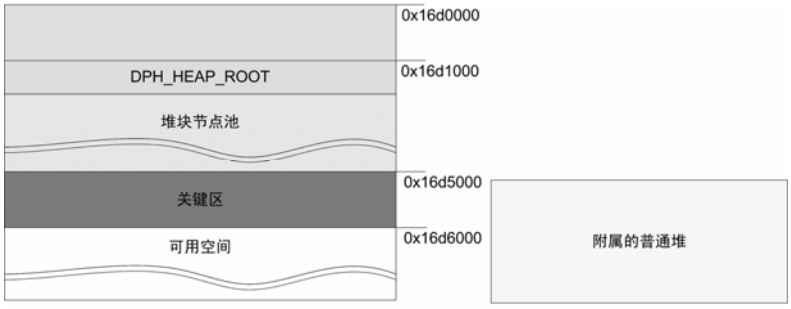

图 23-5 页堆的布局

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 0xeeffeeff 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（Fense 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及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。

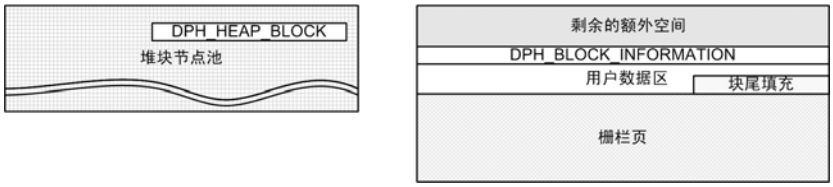

图 23-6 页堆堆块的数据布局

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基地址（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       : 0xabcdbbbb     //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	ABCDBBBB	ABCDBBBA
头结构的 End Magic	DCBABBBB	DCBABBBA
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  abcdbbbb 016d1000 00000009 00001000
016d6fe0  00000012 00000000 00346a60 dcbabbbb
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 l4
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。

23.16 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	检测堆缓冲区	调试/发布	gflags /i +htc

	式字段，详见第 23.8.3 节	溢出		
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

字节模式	堆管理器	用途	长度
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

CHAPTER

第 30 章

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。

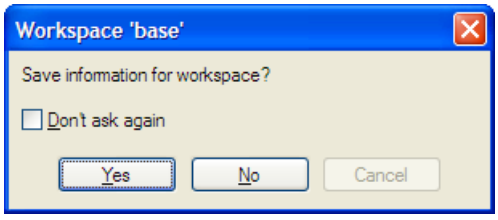

图 30-1 切换或者关闭调试会话时 WinDBG 提示是否要保存工作空间

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+/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 个子健 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。

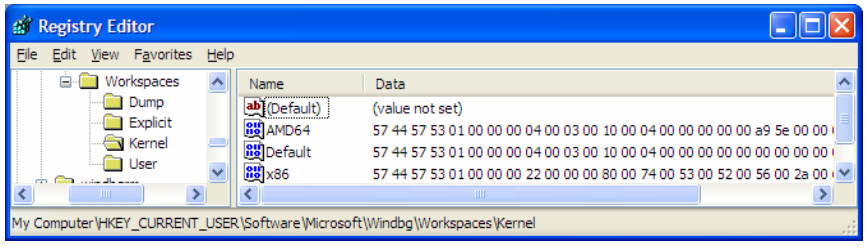

图 30-2 WinDBG 在注册表中保存的工作空间信息

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。一种更快速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
acpikd.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 mscorwks 命令让 WinDBG 在 mscorwks 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。

30.4 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>(0xfffffffe))
Evaluate expression: -18 = fffffffe
```

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>(0xfffffffe)。

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）可以返回栈帧基地址+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。

- $iment（Address）：返回参数 Address 所代表模块的入口地址，Address 应该为模块的基地址。例如，对于 VC 编译器产生的普通 EXE 模块，其基地址为 400000，$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 stricmp 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 结构的基地址。
- #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：

```
||0:2:006>  r eax; $$ address of var_a; r ebx; * var_b; r ecx <will not be executed>
eax=001a1ea4
ebx=7ffdf000
```

可见，$$ 之后的 r ebx 仍被当作命令执行，而*之后的 r ecx 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
$ra	当前函数的返回地址（retrun address）。例如，可以使用 g @$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
$bpx	x 号断点的地址
$frame	当前栈帧的序号
$dbgtime	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：

```
||0:0:001> ln  @$exentry
(01012475)  calc!WinMainCRTStartup  | (0101263c)  calc!__CxxFrameHandler
Exact matches:
   calc!WinMainCRTStartup = <no type information>
||0:0:001> ? @$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$t0~$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$u0~$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：

```
||1:0: kd> .echo $ntnsym * 在内核态调试会话中
nt
||0:2:006> .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：

```
||1:0: kd> as v version
```

接下来便可以使用 v 来执行 version 命令：

```
||1:0: 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 l4
80834190  8092023a 8096b71e 8096f9be 8096b750
||1:0: kd> dd SST +8 l4
80834198  8096f9be 8096b750 8096f9f8 8096b786
||1:0: kd> dd ${SST}+8 l4
80834198  8096f9be 8096b750 8096f9f8 8096b786
```

但像下面这样便会出错：

```
||1:0: kd> dd SST+8 l4
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 l4
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=7ffdc000
```

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），如果否，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。

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。

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。

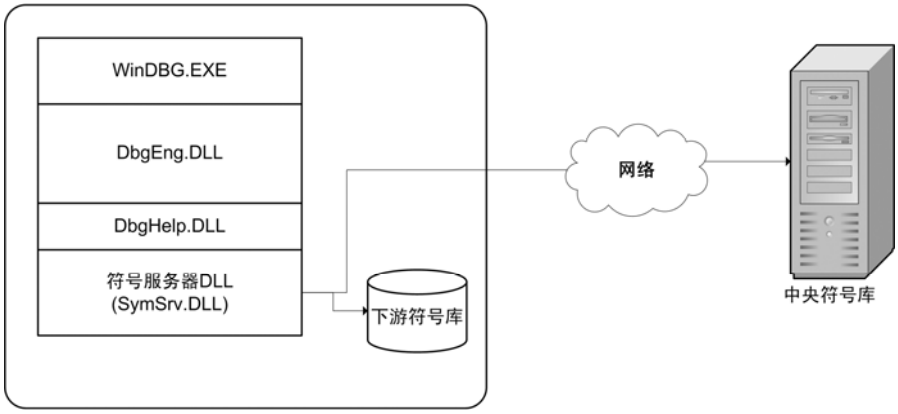

图 30-10 符号服务器架构示意图

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 ld 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> kn 30
 # 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 00f1cda0 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+0xbd8    //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 00f1ffb4 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（TimeDateStamp）	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 SymbolServerGetIndexStringW 函数生成一个索引串。比如，以下是为某一版本的 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+0xbf9   //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地址
Loaded Module Info: [WinDBG]
        Module: WinDBG                      //模块名称
   Base Address: 01000000                   //模块在内存中的基地址
     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\CDA701854AB94F6F8B60FDC14F75FB311\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、maiain 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 msvcrt!_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。

3.8.7 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：

```
lkd> 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	常量	含义	默认值
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相近的符号	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来管理，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。

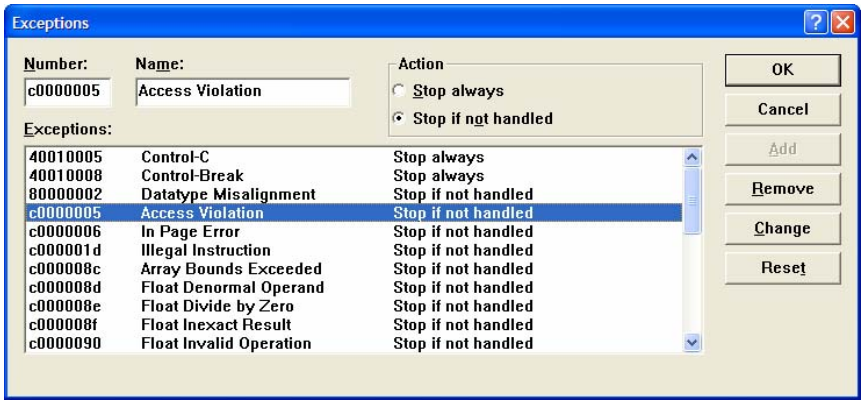

图 30-11 VC6 调试器的异常处理选项对话框

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。

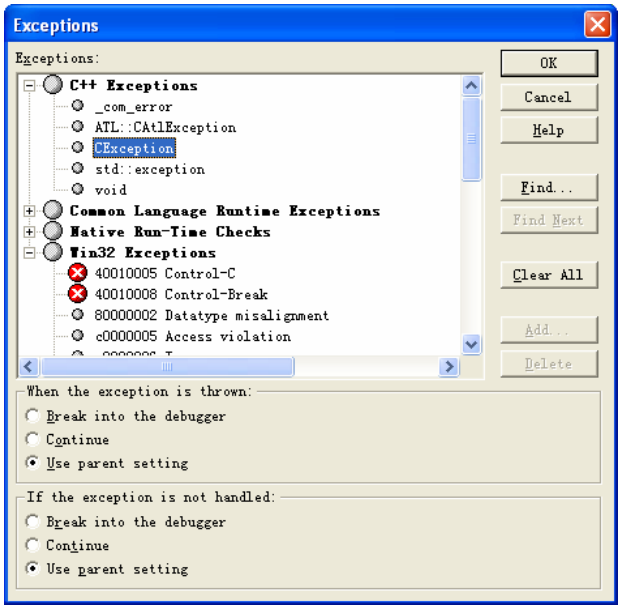

图 30-12 VS7 的异常处理选项对话框

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。

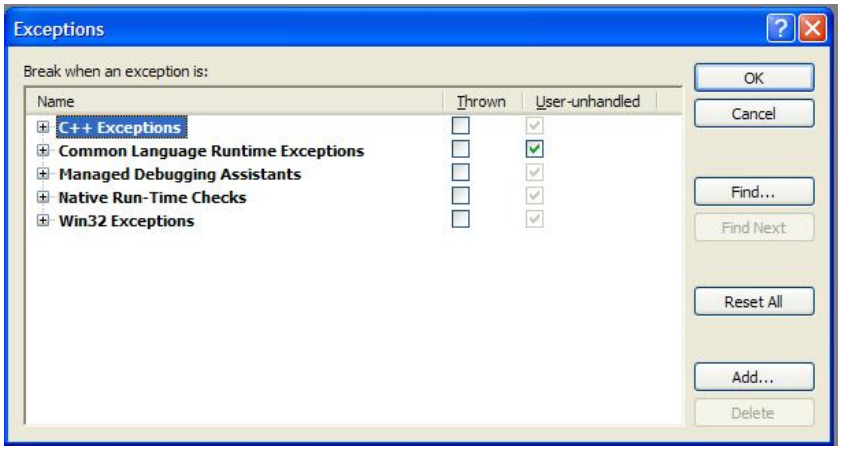

图 30-13 Visual Studio 2005 的异常处理选项对话框

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。

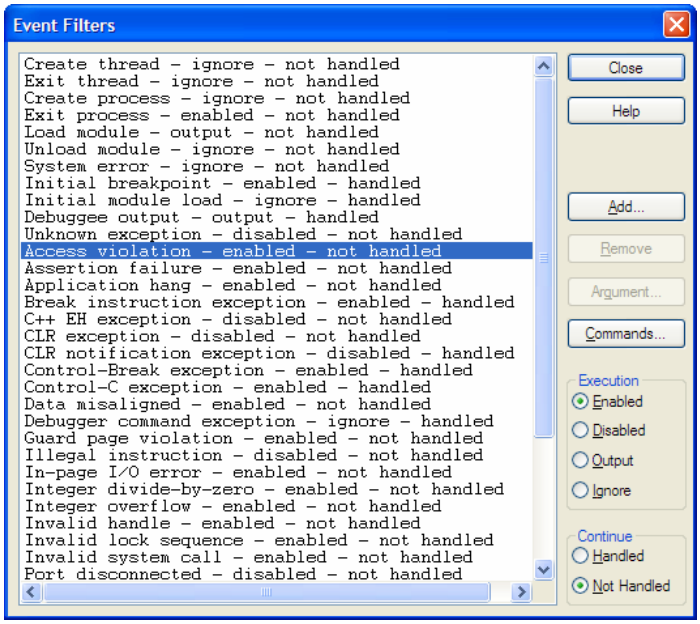

图 30-14 WinDBG 的异常处理选项对话框

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 WinBDG 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 c705000000001000000 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 c705000000001000000 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。

30.18 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 7ffdf000，那么$t2 的值为 7ffdf00c，因为 Ldr 字段的偏移为 0xC，如果不带&符号，那么$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$$，其原因是命令程序执行时会变自动合并为一行，换行符被替换为分号，而*注释符是注释整行，$$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$peb 代表）的 Ldr 子结构的 InLoad-OrderModuleList 字段的地址赋给$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

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	
|	|, |<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