开发人员专业技术丛书

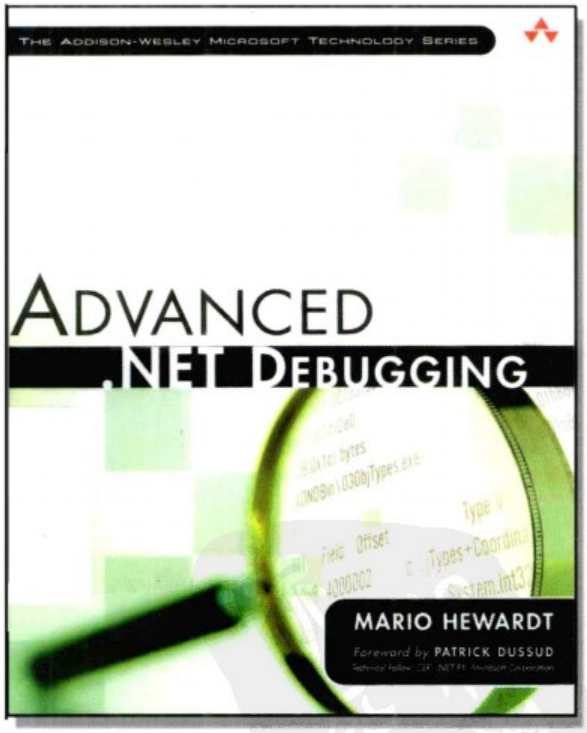
HZ BOOKS 华章科技

PEARSON

.NET高级调试

Advanced .NET Debugging

- 畅销书《Windows高级调试》姊妹篇
- .NET调试权威参考书
- 涉及.NET CLR 4.0最新调试功能



（美） Mario Hewardt 著

聂雪军 等译

机械工业出版社
China Machine Press

PEARSON Education

.NET高级调试

Advanced .NET Debugging

"对于任何一个.NET开发人员来说，本书都具有极高的参考价值。它包含了许多调试技巧以及CLR内部工作机制的细节，这对于设计软件架构的开发人员来说是非常有益的。"

—— Jeffrey Richter，Wintellect公司顾问、培训师和作者

"这是Mario推出的又一本好书。他之前著的《Windows高级调试》（与Daniel Pravat合著）对于非托管代码的调试来说是一本不可多得的参考书，而本书同样具有极高的质量，阐述清晰并且探讨深入，因此对于.NET调试来说同样具有帮助作用。"

—— Mark Russinovich，微软公司技术顾问

这是分析.NET应用程序问题方面的一本全面且实用的参考书。

本书首次专门且系统地介绍了如何分析当前最复杂和最具挑战性的.NET应用程序问题。这是一本介绍如何通过非托管调试器（包括WinDBG、NTSD和CDB等）来调试.NET应用程序的书籍。作者详细阐述了如何借助这些工具找出问题的真实原因——这比使用其他任何调试器都将节省大量的调试时间。

作者首先介绍了在使用.NET非托管调试器时的一些关键概念。接下来介绍了许多巧妙的调试技术，并且通过真实的示例来展示各种常见的C#编程错误。

读者在本书中可以学到：

- 使用事后调试技术，包括PowerDBG以及其他"强大的调试工具"。
- 理解在.NET CLR 4.0中包含的新调试功能以及与之前版本的差异。
- 掌握对Windows调试工具集、SOS、SOSEX、CLR分析器以及其他调试工具的使用。
- 深入理解CLR内部工作机制，例如分析线程特定的数据、托管堆和垃圾收集器、互用层以及.NET异常等。
- 解决一些复杂的同步问题、托管堆问题、互用性问题等。
- 如何生成和分析崩溃转储。

配套网站（advanceddotnetdebugging.com）上包含了所有的示例代码、程序等。

作者简介

Mario Hewardt 是微软公司的一位资深开发经理，在Windows系统级开发领域拥有十余年的开发经验。他目前领导开发团队负责Microsoft在线IT管理解决方案的开发。Hewardt是《Windows高级调试》(机械工业出版社2009年5月出版)的作者之一。

客服热线：(010) 88378991, 88361066
购书热线：(010) 68326294, 88379649, 68995259
投稿热线：(010) 88379604
读者信箱：hzjsj@hzbook.com

华章网站 http://www.hzbook.com

网上购书：www.china-pub.com

封面设计 · 锡 彬

PEARSON

www.pearsonhighered.com

上架指导：计算机/程序设计

ISBN 978-7-111-32085-2

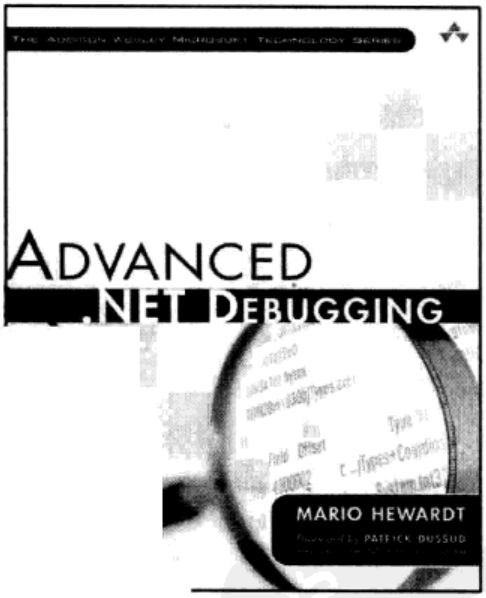
9 787111 320852

定价：59.00元

开发人员专业技术丛书

.NET高级调试

Advanced .NET Debugging



Mario Hewardt 著

聂雪军 等译

机械工业出版社
China Machine Press

这是一本介绍如何通过非托管调试器（包括 WinDBG、NTSD 和 CDB 等）来调试 .NET 应用程序的书籍。本书内容主要包括：调试工具简介、CLR 基础、基本调试任务、程序集加载器、托管堆与垃圾收集、同步、互用性以及一些高级主题，如事后调试、一些功能强大的调试工具和 .NET 4.0 中的新功能等。

本书内容翔实、条理清晰，适合软件开发人员、软件测试人员、质量保证人员和产品技术支持人员等参考。

本书版权登记号：图字：01-2009-7722

图书在版编目（CIP）数据

.NET 高级调试/（美）赫瓦特（Hewardt, M.）著；聂雪军等译. —北京：机械工业出版社，2010.11

（开发人员专业技术丛书）

书名原文：Advanced .NET Debugging

ISBN 978-7-111-32085-2

Ⅰ.N…　Ⅱ.①赫…　②聂…　Ⅲ.计算机网络－程序设计　Ⅳ.TP393

中国版本图书馆 CIP 数据核字（2010）第 192357 号

机械工业出版社（北京市西城区百万庄大街 22 号　邮政编码　100037）

责任编辑：秦　健

北京市荣盛彩色印刷有限公司印刷

2011 年 1 月第 1 版第 1 次印刷

186mm × 240mm · 25 印张

标准书号：ISBN 978-7-111-32085-2

定价：59.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

客服热线：（010）88378991；88361066

购书热线：（010）68326294；88379649；68995259

投稿热线：（010）88379604

读者信箱：hzjsj@hzbook.com

对本书的赞誉

“虽然.NET 环境为开发人员提供了一组强大的工具来编写软件，但是对开发过程中出现的各种问题进行调试仍然是一项困难的任务。本书介绍了 CLR 的内部工作原理，这对于分析.NET 程序中出现的各种类型的错误是非常有帮助的。此外，本书还详细介绍了如何解决一些常见的复杂问题。对于所有.NET 开发人员来说，本书都值得一读。”

——Lee Culver，微软公司 CLR 快速响应团队

“你是否碰到过.NET 程序失去响应？或者周期性地出现极高的 CPU 使用率？或者程序崩溃？当程序出现问题时，你需要使用正确的知识和工具，并深入程序内部进行分析。本书包含了丰富的知识，可以帮助开发人员迅速找出各种软件问题。欢迎进入调试领域！”

——Roberto A Farah，微软公司高级专案领域工程师

“对于.NET 开发人员来说，无论是新手还是高手，本书都具有极高的参考价值。本书包含了许多调试技巧以及 CLR 内部工作机制的细节，这对于设计软件架构的开发人员来说非常有用。我个人强烈推荐 Mario 的这本书。”

——Jeffrey Richter，Wintellect 公司顾问，培训师以及作者

“这是 Mario 推出的又一本好书。他之前编写的《Windows 高级调试》（与 Daniel Pravat 合著）对于非托管代码的调试来说是一本不可多得的参考书，而本书同样有着极高的质量，清晰的阐述以及深入的探讨，因此对于.NET 调试来说同样很有帮助。本书详细介绍了如何观察托管堆的使用情况、理解垃圾收集器的行为以及如何跟踪同步错误等，这些内容将使你在查找和修复托管代码中的错误时更加高效。”

——Mark Russinovich，微软公司技术顾问

“本书深入介绍了一些调试工具，例如 SOS，这是在其他的书中还不曾见过的。这些内容对于理解和调试托管程序来说无疑是非常有帮助的。”

——Maoni Stephens，微软公司 GC 开发工程师

“对于想深入了解.NET 通用语言运行时（.NET Common Language Runtime）内部工作原理的人来说，本书绝对值得一读。书中清晰地阐述了.NET 系统的层次结构以及程序集的加载和组织。对于需要调试同步和内存破坏等复杂问题的开发人员来说，我建议阅读这本书。此外，本书还详细介绍了事后调试技术。”

——Pat Styles，微软公司员工

译 者 序

本书是 Mario Hewardt 继《Windows 高级调试》之后推出的又一部力作。.NET 框架为开发人员隐藏了底层系统的复杂性，例如自动化内存管理机制使得开发人员无须关心内存的释放与回收，从而极大地提升软件开发效率。然而，这种抽象也使得一些问题调试起来更为困难，因此需要了解的底层技术细节也变得更多。本书以非托管调试器（包括 WinDBG、NTSD 和 CDB）为基础，详细介绍了.NET 框架中关键组件的运作原理及其与.NET 应用程序中一些常见问题的关联，并通过丰富的示例来阐述在调试不同问题时所应采取的策略、步骤以及工具。

作为《Windows 高级调试》的姊妹篇，本书的目的同样是为了将一些有价值的调试思路和调试工具推荐给软件开发人员，以提高开发人员的调试效率。本书的特点主要有：

- 深入介绍了.NET 中关键组件的内部运作原理，包括 CLR 垃圾收集器、程序集加载器、应用程序域以及类型元数据等。在调试.NET 应用程序时，对.NET 内部运行机制了解得越清楚，就越能控制程序的运行状态，调试效率也就越高，这与调试 Windows 系统应用程序或者其他任何类型程序的道理是相通的。
- 通过丰富的示例来讲解调试过程。本书分析了.NET 应用程序在程序集加载、托管堆、同步以及互用性等方面的一些常见问题。对于每类问题，作者首先给出了问题的外在表现，然后介绍如何做出合理假设以及采用相应的调试策略来进行验证，并最终找出问题的根源。对调试过程中的每个步骤，作者都给出了非常清晰的讲解和分析，使得读者更容易掌握书中内容。
- 重点介绍了两个调试器扩展 SOS 和 SOSEX。在本书各章的内容中穿插对 SOS 与 SOSEX 中各个命令的介绍，包括命令的语法格式、应用场合以及使用技巧等。这些内容在其他书籍中是很少出现的。

此外，本书还给出了一些高级调试主题，包括崩溃转储文件的生成、事后调试，并介绍了一些辅助调试工具，例如 PowerDbg 和 CLR 分析器等。这些内容都为开发人员调试复杂问题提供了必要的基础知识。在最后一章还介绍了在.NET 4.0 中引入的一些新功能。

本书保持了《Windows 高级调试》的行文风格与内容组织方式，围绕核心的调试思路，采用了由浅入深、由表及里的讲解方式，并辅以丰富的示例和详细的分析，使读者逐步掌握.NET应用程序的调试技巧。本书的技术性较强，需要读者对.NET 框架的底层架构和组件有一定的了解，并且具备一定的 C#编程基础。作者 Mario Hewardt 是微软公司的资深工程师，他在调试领域已经工作了十余年。本书汇聚了作者多年来的调试实践经验，对于.NET 开发

人员来说是一本不可多得的参考手册。

参与本书翻译工作的还有李杨、吴汉平、徐光景、童胜汉、陈军、胡凯、刘红、张玮、陈红、李斌、李勇涛、王海涛、周云波、彭敏才和张世锋等。由于译者的水平和时间有限，翻译中的疏漏和错误在所难免，还望读者和同行不吝指正。

聂雪军

2010 年 8 月于武汉

序

去年，我们在微软公司庆祝了 CLR 发布十周年。CLR 的目的是通过提供一种安全的和稳定的环境来提高开发人员的生产效率。目前，CLR 在各种环境中都得到了广泛应用，例如，在性能和可伸缩性上有着极高要求的大型服务器程序，以及日常使用的桌面程序等。随着 CLR 的日益普及，基于 CLR 来开发软件的人们同样面临着越来越多的挑战，因为他们的产品必须能够在不同的机器配置和网络环境中运行；此外，随着硬件的高速发展，人们正在构建的软件功能越来越强，同时复杂性也越来越高。所有这些情形都意味着，当程序没有按照预期方式运行时，你就需要负责分析和修复程序中的问题，因此了解一些调试知识和工具就显得尤为重要。

为了提高工作效率，CLR 为开发人员实现了许多基础的辅助机制，从而使开发人员能将主要精力放在关键逻辑上。事实上，人们无需花太多的时间来理解完整的 CLR 内部细节，而只需知道一些有助于分析问题的重要概念，这一点非常重要。然而，要想知道哪些概念是重要的却并不容易。许多人都是通过反复摸索之后才掌握这些知识，而这需要长时间的积累过程并且有时候可能得不到准确的答案。

本书对运行时的阐述恰到好处，它能帮助你理解在分析问题时遵循的思考过程以及在解决问题时采用的各种技术，此外书中还给出了从调试实际应用程序中提炼出的许多实用技术。因此，如果你希望提高调试 CLR 应用程序的速度，那么应该仔细阅读本书。本书涵盖了托管程序调试的许多方面——特别是对于一些难以诊断的领域，例如线程同步问题，本书给出了深入而细致的讲解。此外，本书在说明调试技术时使用了大量的示例，使得读者更容易掌握这些技术。

在本书中重点讲解的调试工具之一就是 SOS 调试器扩展，这个工具是由 CLR 小组开发和维护的。每当发布新版本的 CLR 时，都会对 SOS 进行升级，使 SOS 包含更多的新功能。对于分析托管进程中的问题来说，SOS 是一种功能强大的工具。它提供的大部分功能都是无法从其他调试工具中获得的。例如，SOS 可以找出引用托管堆中某个对象的根对象，这是托管程序开发人员经常遇到的问题之一。在熟悉了这个工具的使用后，你将可以进一步理解程序的工作流程。我还从未见过有其他的书比这本书更详细地介绍 SOS。

当掌握本书介绍的知识后，在分析问题时可以付出更少的时间和精力。我希望读者在阅读这本书时获得的乐趣与我在审阅本书手稿时获得的乐趣是一样的。

Patrick Dussud

微软公司技术顾问和.NET CLR 首席架构师

前　言

自从2007年底写完《Windows高级调试》（由机械工业出版社于2009年5月出版——译者注）一书后，许多读者都来信希望我再写一本关于.NET调试的书籍。最初在《Windows高级调试》中确实包含了一章内容专门介绍.NET调试，但最终还是去掉了这章，因为我认为，仅有一章的篇幅无法涵盖.NET调试的所有内容，这给读者带来的只能是更多的疑问而不是启迪。.NET已经成为众多开发人员的首选平台。根据统计数据表明，使用C#的开发人员与使用C++的开发人员在数量上基本相当。要想在.NET开发中取得成功，那么就需要知道如何正确地应对在开发中遇到的各种问题和挑战。

为什么在一本介绍.NET调试的书中使用了WinDBG、NTSD以及CDB等这些调试器？显然还有其他的调试器可以使用（有些调试器甚至有着更高的用户友好性）。初看上去，使用非托管调试器似乎有些棘手，但如果将它们的功能完全发挥出来，那么在分析一些复杂的问题时将节约大量的时间。部分原因是当使用非托管调试器时，更容易收集CLR本身（即.NET运行时）的关键内部信息，并可以根据这些信息来分析问题的原因。这些信息包括垃圾收集器、互用性层、同步原语等。这种信息不仅会在许多调试任务中起到关键作用，而且还是一种很好的学习资料，因为它给出了对运行时架构的详细描述。最后，在某些情况下（随着目前“互联”解决方案越来越多），需要使用一种零痕迹（ZEOR foot print）调试器。这种“更友好的”调试器通常需要在本地进行安装，在这个过程中将把所需的二进制文件复制到目标机器上，并在系统的不同位置保存配置信息等。如果在某台机器上不允许修改配置信息（例如在用户机器或者数据中心的机器上），那么唯一可行的方法就是使用非托管调试器，因为这种调试器不需要修改配置信息。

本书弥补了调试领域中的这种缺陷，并重点介绍了如何在CLR环境中发挥非托管调试器的功能。本书采用了与实践紧密相结合的讲解方式，使用真实的调试任务示例，以确保读者不仅可以学习书中介绍的内容，而且还能获得实际的体验。我希望读者在阅读本书时获得的乐趣与我在写这本书时获得的乐趣一样。

本书的目标读者

如果在分析问题时知道该采用哪些工具，并且知道如何对不同的问题进行分类，那么通常可以开发出高质量的软件产品，并降低在技术支持上的开销。然而，尽管我们已经竭尽全力消除产品中的错误，但在客户机器上仍然会冒出一些意料之外的问题，只有知道如何在这种情况下分析这些问题，才能有效地避免给客户造成麻烦或者停机。所有的软件工程师都有必要了解在各种不同的环境中应该使用哪些工具来分析问题。如果掌握了调试的技巧，那么

产品质量将得到极大提升，而且公司的形象也会由于软件的高质量和高可靠性而得到提升。

软件开发工程师

我曾看到许多开发人员努力解决各种复杂的错误，并且最终花费数天（有时候甚至是数星期）时间来缩小问题的范围以及找到问题的根源。在大多数时候，我都会询问开发人员使用了哪些工具。得到的答案通常大同小异：反复进行代码审查和跟踪，直至最终找出问题。代码审查和有针对性的跟踪对于分析错误来说固然重要，但它们通常也是非常耗时的。面对现实吧！假使我们能够获得在分析代码问题时需要的所有信息，那么将不需要使用调试器。然而，实际情况是，在某些情况中仅凭代码跟踪不足以解决问题，而是需要将调试器附载到出现问题的进程上。在许多时候，当我告诉开发人员某个工具可以极大地降低某种问题的分析时间后，开发人员会吃惊地发现居然有这种工具存在。

本书的目标读者之一就是那些在.NET 平台上编写代码以及分析复杂问题的开发人员。能够深入地理解哪些工具可以帮助开发人员找出一些复杂且耗时的问题，对于产品的成功来说非常重要。在开发过程中，知道使用哪些工具以及启动哪些配置，对于获得成功来说是非常关键的。

质量保证工程师

质量保证（Quality Assurance，QA）的目的是找出开发人员在代码中出现的问题。详细的测试计划以及完全自动化的测试流程都能使质量保证工程师们以高效的方式来测试各个组件。正如了解调试工具和配置信息对开发人员非常重要，这对于质量保证工程师也同样重要。在测试过程中，他们可能会遇到各种问题，如果在测试过程中能使用正确的工具，那么将帮助他们以及开发人员在解决问题上节约大量的时间。如果在进行测试时没有使用正确的工具，那么在重新启动测试流程（并且打开这个工具）时可能最终发现这不是一个可重现的问题。在本书中介绍的调试器和工具能够提高质量保证工程师的工作效率，并且帮助整个产品团队更快地获得结果。

产品支持工程师

产品支持工程师面临的情况与软件开发工程师和质量保证工程师遇到的情况非常相似。关键的区别在于他们所处的环境不同。他们不仅要解决客户的问题，而且经常必须面对来自多个地方的代码（即不仅仅是产品的代码）。此外，产品支持工程师通常只能使用进程的静态快照，而无法进行实时调试，这就使得找到问题的根源更加困难。在这些情况下，如果知道如何使用调试器以及相应的工具，那么就不需要多次往返于公司与客户之间（这不仅会带来很高的成本开销，而且很容易令人沮丧），并且能够立即解决问题。

运行工程师

随着越来越多的软件移动到云中（一种基于服务的提供方式），越来越多的代码是在专门的

数据中心运行，而不是在客户的机器上运行。负责这些服务正常运行的工程师们就被称为运行工程师。运行工程师面临的关键挑战之一是，要解决所有使服务无法最优化运行的问题。通常，这意味着要尽快地解决问题。如果运行团队无法解决某个问题，那么产品团队会参与进来，这将是一个耗时的过程，因为产品团队可能要反复地指导运行团队分析这个问题。通过使用正确的工具，运行团队可以解决遇到的大部分问题，而无需将这些问题提交给产品团队，因此节约了双方的时间；并且最重要的是，客户在使用服务时遇到的停机时间也将更短。

预备知识

虽然本书介绍的是如何使用非托管调试器，但重点在于介绍如何调试.NET 代码，而不是非托管调试的一些基本步骤。第 3 章将会简要介绍一些主题，例如如何将调试附载到目标进程，设置符号路径以及设置断点等，但并没有给出详细介绍。关于非托管调试器以及非托管代码调试的进一步介绍可以参考我之前写的一本书：Mario Hewardt 和 Daniel Pravat，《Advanced Windows Debugging》，Boston，MA：Addison-Wesley，2007。[⊖]

此外，还需要深入理解 C#，因为所有的示例代码都是采用 C#编写的。在学习 C#时可以使用下面这本书：Mark Michaelis，《Essential C# 2.0》，Boston，MA：Addison-Wesley，2006。

虽然读者需要预先具备 C#的知识，但并不需要深入了解 CLR。本书不仅介绍了如何调试.NET 程序，还对.NET 平台中大量的核心代码给出了详细解释，这些信息是获得调试成功的重要基础。

本书的组织结构

本书分为三个部分，下面介绍每一部分包含的内容。

第一部分　简介

这一部分包含的内容主要是介绍如何使用非托管调试器来调试.NET 的基础知识。介绍的主题包括：需要使用的工具、对 MSIL 的介绍、基本的调试任务等，我对这些主题都进行了全面的分析和阐述。如果你是第一次了解这些调试器，那么建议你按照先后顺序阅读这些内容。

第 1 章　调试工具简介

这一章对本书中使用的工具给出了简要介绍，包括应用场合、下载位置以及安装方法等。所介绍的工具包括：Windows 调试工具集、SOS、SOSEX、CLR 分析器等。

第 2 章　CLR 基础

这一章介绍了.NET 运行时的核心基础。首先介绍了主要的运行时组件，然后是一些相关主题，包括程序集的加载、运行时元数据等。其中使用非托管调试器和工具来说明运行时

⊖ 中文版为《Windows 高级调试》，由机械工业出版社于 2009 年 5 月出版。——译者注

的内部工作机制。

第 3 章　基本调试任务

这一章介绍了在使用非托管调试器调试.NET 程序时的一些最常见调试任务，此外还介绍了一些其他主题，例如查看线程数据、垃圾收集器堆、.NET 异常以及事后调试的基本概念等。

第二部分　调试实践

第二部分是本书的核心内容，介绍了主要的 CLR 组件以及如何分析与这些组件相关的常见问题。在这部分每一章的开始，首先给出组件的概述，并通过调试器来说明关键概念。之后给出使用这个组件时会出现的常见编程错误。接下来详细介绍了在解决这些错误时该采取的思路，并给出了直观的调试会话。第二部分中的各章可以按照任意顺序来阅读，因为每一章都是针对某个特定组件的问题进行讨论。

第 4 章　程序集加载器

.NET 程序既可以是简单的命令行程序，也可以是复杂的多进程/多机器服务器程序，其中包含了大量程序集的相互协作。要想高效地调试.NET 程序中的问题，你必须仔细地理解各个.NET 程序集之间的依赖性。这一章介绍了 CLR 程序集加载器的工作原理，以及在使用这个组件时最常见的问题。

第 5 章　托管堆与垃圾收集

虽然.NET 开发人员可以坐享自动内存管理机制带来的好处，但仍然需要小心地避免一些高开销的错误。CLR 垃圾收集器是一个自动的内存管理器，它使开发人员可以更少关注内存管理，而更多关注程序逻辑。尽管 CLR 负责为开发人员管理内存，但仍然要小心地避免一些可能对程序造成严重破坏的陷阱。在这一章中，我们将看到垃圾收集器的工作机制，学会观察垃圾收集器的内部信息以及与自动垃圾收集操作相关的一些常见编程错误。

第 6 章　同步

多线程的环境包含了大量的灵活性和高效因素。然而，伴随这种灵活性的是线程管理中的复杂性。为了避免在程序中出现高开销错误，我们必须小心地确保线程以一种相互协作的方式来执行工作。这一章介绍了.NET 中的同步原语，并介绍了如何使用调试器和工具来分析常见的线程同步问题。此外还介绍了死锁以及线程池等问题。

第 7 章　互用性

.NET 非常依赖底层的 Windows 组件。为了调用非托管的 Windows 组件，CLR 公开了两种主要的互用性方法：

- 平台调用
- COM 互用性

由于.NET 和 Win32 编程模型非常不同，因此一些编程习惯经常会导致难以跟踪的问题。在这一章中，我们将看到一些在使用互用性时常见的错误，以及如何使用调试器和工具来分

析这些错误。

第三部分　高级主题

这一部分介绍的主题包括：事后调试、一些功能强大的调试工具，以及.NET 4.0 中的新功能。

第 8 章　事后调试

通常，我们可能无法获得对故障机器的完全访问权限，因此无法对客户产品机器上出现的错误进行实时调试。本章介绍了如何在不需要访问物理机器的情况下对问题进行调试。讨论的主题包括：崩溃转储的基本知识，如何生成以及分析崩溃转储文件等。

第 9 章　一些功能强大的调试工具

在.NET 调试过程中，除了一些“标准的“工具外，还有其他一些功能强大的调试工具。这一章为读者介绍了这些强大的工具，例如 PowerDBG（通过 PowerShell 调试）以及其他工具。

第 10 章　CLR 4.0

随着 CLR 4.0 的即将发布，这一章将简要介绍 CLR 4.0 中的新增功能。此外，本章依次介绍了之前内容中每个主题在 CLR 4.0 中的不同之处。

排版约定

在本书的示例中给出了大量的调试输出。调试输出是指用户在执行某个动作后调试器显示的结果。通常，这种调试输出包含了以简洁形式给出的信息。为了引用这些数据并使读者更容易阅读，在调试输出中采用了粗斜体。此外，在调试输出中的粗体内容表示需要键入的命令。在接下来的示例中说明了这些字体。

```
0:000> ~*kb
.  0  Id: 924.a18 Suspend: 1 Teb: 7ffdf000 Unfrozen
ChildEBP RetAddr  Args to Child
0007fb1c 7c93edc0 7ffdf000 7ffd4000 00000000 ntdll!DbgBreakPoint
0007fc94 7c921639 0007fd30 7c900000 0007fce0 ntdll!LdrpInitializeProcess+0xffa
0007fd1c 7c90eac7 0007fd30 7c900000 00000000 ntdll!_LdrpInitialize+0x183
00000000 00000000 00000000 00000000 00000000 ntdll!KiUserApcDispatcher+0x7
0:000> dd 0007fd30
0007fd30  00010017 00000000 00000000 00000000
0007fd40  00000000 00000000 00000000 ffffffff
0007fd50  ffffffff f735533e f7368528 ffffffff
0007fd60  f73754c8 804eddf9 8674f020 85252550
0007fd70  86770f38 f73f4459 b2f3fad0 804eddf9
0007fd80  b30dccd1 852526bc b30e81c1 855be944
0007fd90  85252560 85668400 85116538 852526bc
0007fda0  852526bc 00000000 00000000 00000000
```

在这个示例中，需要在调试会话中键入 ~ * **kb**。在键入这个命令后，将输出几行信息，其中最关键的就是 **0007fd30**。接下来应该键入 **dd 0007fd30** 命令，这个命令将显示之前高亮

数值 **0007fd30** 的更多信息。在本书中使用的所有工具都是从安装文件夹中启动的。例如，如果 Windows 调试器安装到文件夹 C:\Program Files\Debugging Tools for Windows 中，那么启动 windbg.exe 的命令行如下所示：

```
C:\>windbg
```

所需工具

在本书中使用的所有工具都可以免费下载。在第 1 章中给出了在本书中使用的工具及其下载地址。

示例代码

要说明如何调试存在问题的代码，最有效的方式就是使用实际的示例。然而，在一本书中包含完整的示例代码是不现实的，因为这将使我们无法以简洁的方式说明问题。因此，对本书中的示例代码都进行了简化（但并不影响完整性）。所有的示例代码都是基于 C#和. NET 2.0 编写的。每个示例都可以从本书的 Web 站点下载，地址为 http://www.advanceddotnetdebugging.com。每个示例都包含了一个 MSBuild 项目文件。MSBuild 是一个功能完备的命令行构建环境，与. NET SDK 2.0 一起发行，并且与 Microsoft Visual Studio 是兼容的。所有的调试会话都是基于 32 位版本的. NET 框架。

技术支持

虽然我努力避免在本书中出现错误，但还是不可避免地存在一些疏漏。你可以将发现的错误提交到本书的网站 http://www.advanceddotnetdebugging.com 或者可以直接给我写信，电子邮件地址为 marioh@advanceddotnetdebugging.com。在这个网址上还包括一个勘误表以及相应的错误和修正。

小结

在当前复杂的软件解决方案中，无论是独立的命令行程序还是高度互连并且在全世界范围内通信的系统，都会存在代码问题。要确保在这些产品中不出现错误似乎是一项很艰难的任务，但只要有正确的工具以及知道如何使用这些工具，那么软件工程师的工作就会变得很轻松。这些强大的工具和正确的思路不仅有助于使分析问题的过程变得更高效，而且还为企业节省了大量的财力和减少了可能造成的客户流失。本书的目的是为了使软件工程师们获得. NET调试的知识和经验，从而避免一些严重的陷阱，并且使问题的分析过程变得更为高效和成功。

欢迎阅读。

致谢

基于之前编写《Windows 高级调试》一书的经验，我很清楚地认识到当一边在微软全职工作，一边在业余时间写书时需要付出多大的努力。在《Windows 高级调试》一书获得成功后，读者不断要求我再写一本关于.NET 和 CLR 调试的书，因此我决定再次迎接挑战。虽然与第一本书相比，这次的任务要更容易一些，但仍然付出了很大的努力——不仅仅是我自己，而且还包括本书中从构思到出版过程中参与的许多人。

首先，我要感谢我的家人。我的妻子 Pia 不仅毫无怨言地容忍我经常说“这个周末我要写作”，而且在 2008 年底为我带来了漂亮可爱的女儿 Gemma。如果没有 Pia 和 Gemma 的耐心、支持和鼓励，本书是无法完成的。

感谢 Addison-Wesley 团队，他们对一位在业余时间写作的作者再次给予了难以置信的宽容。我的团队成员经常不得不请求修改时间表，提供高质量的编辑过程并且加速出版计划，以便使读者能尽快看到这本书。特别要感谢 Joan Murray，感谢他使得整个过程极为顺畅，感谢 Chris Zahn 仔细审阅和纠正我的英语表述，感谢 Olivia Basegio 在 Joan 度假时负责接管编辑工作。此外，特别感谢 Curt Johnson 为本书所做的市场推广工作。

当然，无论作者如何小心，总会有些技术细节不能准确把握。因此，如果有一群优秀的工程师来审阅本书的内容，那么对于一本书的成功无疑是非常重要的。在本书的编写过程中，我很荣幸与一些优秀的工程师们共事（大多数都在.NET 部门工作），他们提出了极有见地的反馈和建议，并回答了许多常见的问题。真诚地感谢 Mark Russinovich、Maoni Stephens、Roberto Farah、Tess Ferrandez、Lee Culver、Pat Styles、Eric Eilebrecht、Steve Johnson 以及 Jon Langdon。

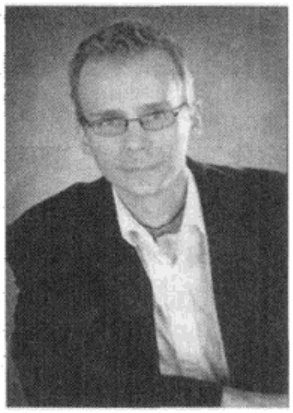
感谢 Patrick Dussud，他不仅非常细致地审阅了本书的手稿，而且还为本书作序。如果没有他的贡献，本书不会是大家现在看到的这样。

同样感谢 Easy Web Launch（www.easyweblaunch.com）的 Alexandra H. Anderson，感谢他为本书提供了网页。

最后，我想要感谢这么多年来为我提供反馈意见的所有读者。你们的支持使本书得以完成。

关于作者

Mario Hewardt 是《Windows 高级调试》的作者之一，他是微软公司的资深开发经理。他拥有 11 年的工作经验，从 Windows 98 一直到 Windows Vista。在过去的几年中，Mario 主要从事 SaaS 领域的工作，开发了 Asset Inventory Service，这个服务用于帮助用户跟踪他们的资产清单。他目前正在领导一个团队，为下一代 Microsoft 在线管理服务开发核心支撑平台。



©www. BrookeClark. com

目　录

第二部分 调试实践

第三部分 高级主题

第一部分 简 介

第1章 调试工具简介

我曾看到过许多开发人员使用错误的工具来分析问题，更有甚者，有些人连任何工具都不使用。他们采取的分析方法通常包括：输出更多的调试信息，或者做一些临时性的代码审查（code review）。这里的临时性是指，通过猜测来推断问题可能来自哪部分代码。例如，如果在系统的事务服务中出现了一个错误，那么他们可能会对整个事务引擎的代码进行审查，而问题实际上来自事务引擎之外的代码。有时候，开发人员会幸运地发现问题刚好处于他们正在审查的代码中。然而，在更多情况下并非如此，问题发生的位置和表现出来的位置往往相去甚远。通过使用一些功能强大的调试工具，开发人员可以极大地减少在分析问题时耗费的时间。

在分析.NET应用程序的问题时，有些工具可以使分析过程变得更为高效。其中，一部分工具侧重于分析某一类特定的问题，而其他工具则可以同时处理若干类问题。我们要知道每种工具在哪些情况下使用以及如何使用，这一点非常重要。本章将简要介绍在本书中使用的各种工具，包括每种工具的不同特性，适用于哪些类型的问题，从何处下载，如何安装，以及运行该工具的一些示例等。值得注意的是，这里介绍的所有工具都是免费下载的。

1.1 Windows 调试工具集

应用场合	一组调试器和调试工具
版本	6.8.4.0
下载地址	http：//www.microsoft.com/whdc/devtools/debugging/default.mspx

Windows 调试工具集是一个免费的软件包，它包含了一组功能强大的调试器和调试工具，用来帮助分析软件中的各种问题。软件包分为两个版本：32位版和64位版，具体使用哪个版本要取决于调试任务所在的架构。在本书中，所有的调试任务都将使用32位的版本。

在 Windows 调试工具集中共包括三种用户态调试器——NTSD、CDB 和 WinDbg，以及

一种内核态调试器（kd）。虽然这三种调试器彼此是相互独立的，但重要的是要知道它们都依赖于相同的内核调试器引擎。在这些调试器之间的主要差异是，WinDbg 有一个图形用户界面（GUI），这使得它在进行源代码级调试（Source Level Debugging）时用起来更为容易。而与之不同的是，NTSD 和 CDB 都只是基于控制台的调试器。本书给出的调试器会话截图都是通过 NTSD 来截取的。

在选择了合适的调试器（32 位或者 64 位）后，Windows 调试工具集的安装过程很简单，只需选择默认的安装选项就可以了。默认的安装路径是

```
%programfiles%\Debugging Tools for Windows
```

在编写本书时，Windows 调试工具集的最新版本为 6.8.4.0。因此，在读者拿到本书的时候，很可能会发布一个更新的版本。即便如此，调试器的输出信息并不会发生太大变动，而在本书中介绍的内容也同样适用。在调试器的下载站点上还保留了之前（2～3 个）版本调试器的下载地址。如果希望使用在本书中用到的版本，那么可以下载版本号为 6.8.4.0 的 Windows 调试工具集。

1.2 .NET 2.0 可再发行组件

应用场合	.NET 运行时（.NET Runtime）
版本	2.0
下载地址	http://www.microsoft.com/downloads/details.aspx?familyid=0856eacb-4362-4b0d-8edd-aab15c5e04f5&displaylang=en

.NET 2.0 可再发行组件包含了.NET 平台的核心功能。它包含了运行.NET 2.0 应用程序需要的所有基础运行时组件，包括所有的框架程序集（assembly）以及.NET 运行时的二进制文件。我们要注意区分.NET 框架程序集与运行时二进制文件之间的差异，因为在不同的.NET 发布版本中存在着许多容易混淆的地方。如果不考虑补丁包的话，.NET 目前发布的版本主要有：1.0、1.1、2.0、3.0 以及 3.5。表 1-1 给出了各个版本之间的主要差异。

表 1-1 .NET 当前的主要版本

.NET 版本	在哪些 Windows 版本包含	架 构	最高补丁包	运行时是否变动	框架是否变动
1.0	无	X86	3	初始版本	初始版本
1.1	Windows Server 2003	X86、x64、IA64	1	否①	是
2.0	Windows Server 2003 R2②	X86、x64、IA64	1	是	是
3.0	Windows Vista	X86、x64、IA64		否①	是（其中 WPF、WCF 和 WF 最为明显）
3.5	Windows Server 2008	X86、x64、IA64		否①	是

①对运行时做了一些细微改动来支持新的功能，但这些改动还不足以作为一个主要发行版本。

②随 Windows Server 2003 R2 一起发行，但在默认情况下并不安装。

如果在操作系统上没有安装. NET 2. 0 可再发行组件，那么可以直接安装，安装过程很简单。跳转到安装网址，然后选择默认的安装选项。

1. 3　. NET 2. 0 SDK

应用场合	辅助开发的库和工具
版本	2. 0
下载地址	http://www. microsoft. com/downloads/details. aspx? FamilyID = FE6F2099 - B7B4 - 4F47 - A244 - C96D69C35DEC&displaylang = en

虽然可再发行组件能够使. NET 应用程序在计算机上运行，但只有. NET SDK 才能使开发人员编写新的. NET 应用程序，因为它提供了所有必要的工具（编译器、汇编器、构建工具）以及库等。. NET SDK 并不提供基于图形界面的集成开发环境，而只是提供了一个基于命令行的环境。

在安装. NET 2. 0 SDK 之前，首先要确保已经安装了. NET 2. 0 可再发行组件。要安装. NET 2. 0 SDK，第一步就是跳转到下载网址并启动安装过程。在安装过程中会看到下面这些安装选项：

- 安装快速入门示例。快速入门示例是一些非常好的示例程序，它们可以帮助你快速地开始构建. NET 应用程序。这些示例都是一些小段代码，涵盖了. NET 的各个应用领域（包括 ADO、互用性以及网络等）。
- 工具与调试器。选择这个选项将安装一组调试工具和调试器。. NET 2. 0 SDK 中的调试器是 DbgClr。DbgClr 是一个源代码级的图形界面调试器，它的界面和操作类似于 Visual Studio。如果你熟悉 Visual Studio 调试器，那么用起 DbgClr 来就非常得心应手。
- 产品文档。强烈建议安装产品帮助文档，因为它包含了大量在开发. NET 应用程序时非常有用的信息。

安装过程的最后一步是选择安装位置（默认的位置为% programfiles \ Microsoft. net \ SDK）。

在. NET 2. 0 SDK 中还包含了一个构建工具，MSBUILD。这也是在 Visual Studio 中使用的构建工具，在使用这个工具时首先要编写基于 XML 格式的构建流程。本书的每个示例程序都有一个相应的 MSBUILD XML 文件（build. xml），可以用来简化示例程序的编译工作。虽然本书没有详细说明 MSBUILD 工具的使用方法，但知道如何通过它来构建书中的示例程序是非常必要的。在清单 1-1 中给出了一个 MSBUILD 项目文件的示例，这个示例表示将构建一个用 C#语言编写的. NET 命令行应用程序。

清单 1-1　用于构建某个.NET 命令行应用程序的 MSBUILD 项目文件

```
<Project DefaultTargets = "Compile"
    xmlns="http://schemas.microsoft.com/developer/msbuild/2003" >

    <!-- Set the application name as a property -->
    <PropertyGroup>
        <appname>Welcome</appname>
    </PropertyGroup>

    <!-- Specify the inputs by type and file name -->
    <ItemGroup>
        <CSFile Include = "simple.cs"/>
    </ItemGroup>
    <Target Name = "Compile">
        <!-- Run the Visual C# compilation using input files of type CS File -->
        <CSC
            Sources = "@(CSFile)"
            OutputAssembly = "$(appname).exe">
            <!-- Set the OutputAssembly attribute of the CSC task
            to the name of the executable file that is created -->
            <Output
                TaskParameter = "OutputAssembly"
                ItemName = "EXEFile" />
        </CSC>
        <!-- Log the file name of the output file -->
        <Message Text="The output file is @(EXEFile)"/>
    </Target>
</Project>
```

要构建一个 MSBUILD 项目，只需启动 MSBUILD 工具，然后将项目文件作为参数传递给它（与源代码位于同一个文件夹中）。

```
C:\MSBuild>c:\Windows\Microsoft.NET\Framework\v2.0.50727\MSBuild.exe build.xml
Microsoft (R) Build Engine Version 2.0.50727.1434
[Microsoft .NET Framework, Version 2.0.50727.1434]
Copyright (C) Microsoft Corporation 2005. All rights reserved.

Build started 4/7/2008 10:25:36 AM.

Project "C:\MSBuild\build.xml" (default targets):

Target Compile:
    C:\Windows\Microsoft.NET\Framework64\v2.0.50727\Csc.exe
/out:Welcome.exe simple.cs
    The output file is Welcome.exe

Build succeeded.
    0 Warning(s)
```

```
    0 Error(s)

Time Elapsed 00:00:00.86
```

1.4 SOS

应用场合	用于调试.NET 应用程序的扩展组件
版本	1.0, 1.1, 2.0
下载地址	包含在.NET SDK 中

SOS 是一个调试器扩展（debugger extension），用于调试.NET 应用程序。它提供了一组非常丰富的命令，这些命令使开发人员可以对 CLR 进行深入分析，并且有助于找出应用程序中各种复杂错误的原因。此外，还有一些命令可以用来查看终结队列（finalization queue）、托管堆（managed heap）、托管线程（managed thread），在托管代码中设置断点以及查看异常等。在本书中，我们将使用 SOS 调试器扩展来分析各种复杂的应用程序错误。

由于 SOS 能够提供 CLR 内部工作机制的抽象视图，因此在使用 SOS 进行调试时，必须使用正确版本的 SOS。每个版本的.NET 在发布时都带有相应的 SOS，可以在以下位置找到：

```
%windir%\microsoft.net\<architecture>\<version>\sos.dll
```

其中，“Architecture ”的值可以是 Framework（32 位）或者 Framework64（64 位），而“version”的值则表示所使用的.NET 框架的版本。

在使用 SOS 之前，必须首先通过 .load 命令将其加载到调试器中。清单 1-2 给出了在调试器下运行 notepad.exe 时的加载流程。

清单 1-2　加载 SOS 调试器扩展

```
0:000> .load c:\Windows\Microsoft.NET\Framework\v2.0.50727\SOS.dll
0:000> !help
-------------------------------------------------------------------------------
SOS is a debugger extension DLL designed to aid in the debugging of managed
programs. Functions are listed by category, then roughly in order of
importance. Shortcut names for popular functions are listed in parenthesis.
Type "!help <functionname>" for detailed info on that function.
Object Inspection                        Examining code and stacks
------------------------                 --------------------------
DumpObj (do)                             Threads
DumpArray (da)                           CLRStack
DumpStackObjects (dso)                   IP2MD
DumpHeap                                 U
DumpVC                                   DumpStack
GCRoot                                   EEStack
ObjSize                                  GCInfo
```

```
FinalizeQueue                          EHInfo
PrintException (pe)                    COMState
TraverseHeap                           BPMD

Examining CLR data structures          Diagnostic Utilities
------------------------               --------------------------
DumpDomain                             VerifyHeap
EEHeap                                 DumpLog
Name2EE                                FindAppDomain
SyncBlk                                SaveModule
DumpMT                                 GCHandles
DumpClass                              GCHandleLeaks
DumpMD                                 VMMap
Token2EE                               VMStat
EEVersion                              ProcInfo
DumpModule                             StopOnException (soe)
ThreadPool                             MinidumpMode
DumpAssembly
DumpMethodSig                          Other
DumpRuntimeTypes                       --------------------------
DumpSig                                FAQ
RCWCleanupList
DumpIL
0:000> !Threads
Failed to find runtime DLL (mscorwks.dll), 0x80004005
Extension commands need mscorwks.dll in order to have something to do.
0:000>
```

在清单 1-2 中，首先加载了 2.0 版本的 SOS 调试器扩展。在成功加载 SOS 后，执行扩展命令！help，这个命令将显示 SOS 中包含的一组命令。接下来，通过！threads 命令显示出进程中所有的托管线程。这个命令的执行结果是输出一个错误消息，表示没有找到 mscorwks.dll。当第一次加载. NET 应用程序时，CLR 将同时加载和初始化。负责实现运行时的库是 mscorwks. dll。如果在调试进程中没有找到这个库，那么 SOS 将返回一个错误信息，表示当前进程并非一个. NET 进程，或者当前还没有加载运行时，因此这些调试器扩展命令是不可用的。

如果不希望指定 SOS 的完整路径，那么可以使用调试命令 . loadby。. loadby 命令的语法如下所示：

```
.loadby <extension DLL> <module name>
```

“extension DLL”表示想要加载的调试器扩展名字（例如 SOS. dll），“module name”表示一个当前已加载的模块（例如 mscorwks. dll）。. loadby 命令将尝试从“module name”模块所在的路径中加载调试器扩展 DLL。例如，如果要从 mscorwks. dll 所在的目录中加载 SOS，可以使用以下命令：

```
.loadby SOS.dll mscorwks
```

最后需要注意的一点是，可以在 Visual Studio 2008 的即时窗口中输入 . load sos 命令来加载和使用 SOS。

这里的 SOS 是不是表示求救的意思

这里的 SOS 与通常理解的意思有所不同，它的含义并不是求救信号。当. NET 框架还在 1. 0 阶段时，Microsoft 团队使用了一个叫做“STRIKE”的调试器扩展来分析. NET 代码中的各种复杂问题。当. NET 框架日趋成熟时，这个调试器扩展的名字也就逐渐演变成了“Son of Strike”（SOS）。

1. 5　SOSEX

应用场合	用于调试. NET 应用程序的扩展组件
版本	1. 1
下载地址	http://www. stevestechspot. com/downloads/sosex_32. zip 或者 http://www. stevestechspot. com/downloads/sosex_64. zip

SOSEX 是另一个调试器扩展，可用于非托管代码（native code）和托管代码（managed code）的调试。它是由 Steve Johnson 开发的，可以免费下载。从名字很容易知道，SOSEX 表示“SOS Extended”。SOSEX 增加了一组功能强大的调试命令。这些命令包括死锁检测，基于代（generational）的垃圾收集（garbage collection）命令以及其他一些功能强大的断点命令。

SOSEX 的安装包是一个 ZIP 文件，只需将 ZIP 文件中压缩的文件释放到指定的位置即可完成安装。我通常将这些文件放到调试器的安装文件夹中，这样可以避免在加载 SOSEX 时指定完整路径。在加载 SOSEX 后，就可以开始使用它所支持的命令，如清单 1-3 所示。

清单 1-3　加载 SOSEX 调试器扩展

```
0:000> .load sosex.dll
0:000> !sosex.help
SOSEX - Copyright 2007-2008 by Steve Johnson - http://www.stevestechspot.com/
Quick Ref:
-------------------------------------------------------
dumpgen <decGenNum> [-stat] [-type <TYPE_NAME>]
gcgen   <hexObjectAddr>
refs    <hexObjectAddr>
bpsc    <strSourceFile> <decLineNum> [decColNum]
bpmo    <strTypeName> <strMethodName> <hexILOffset>
vars    [decFrameNum|-w]
date    <hexDateAddr>
isf     <strTypeName> <strFieldName>
dlk
```

```
Use !help <command> for more details about each command.
0:000>
```

在本书中，我们使用 SOSEX 来调试一些情况中的问题。

1.6 CLR 分析器

应用场合	对内存分配情况进行分析
版本	2.0
下载地址	http://www.microsoft.com/downloads/details.aspx? FamilyId = A362781C - 3870 - 43BE - 8926 - 862B40AA0CD0&displaylang = en

在调试.NET 应用程序中与内存相关的问题时，CLR 分析器是一种非常有用的工具。它提供的功能包括：

- 堆的统计信息（包括分配图）；
- 垃圾收集操作的统计信息；
- 垃圾收集器（Garbage Collector）句柄统计信息（包括分配图）；
- 垃圾收集中每代对象的总体大小；
- 性能分析的统计信息。

CLR 分析器的安装过程很简单。安装包是一个 ZIP 文件，只需将 ZIP 文件的内容释放到指定的文件夹（默认为 C:\CLRProfiler）即可完成安装。在 ZIP 文件释放出的所有文件中包括两个版本：x86 和 x64。要启动 CLR 分析器，只需运行 CLRProfiler.exe。在 CLR 分析器的安装过程中将对源代码进行一些修改和增强。

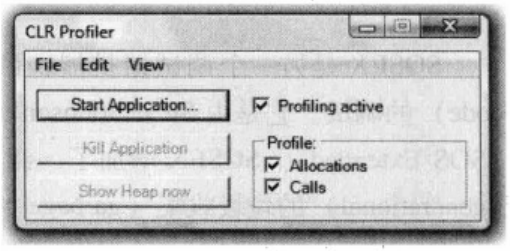


图 1-1 CLR 分析器启动时的对话框

在启动 CLR 分析器之后，会出现一个对话框，其中包含了一组选项，如图 1-1 所示。

点击“Start Application”按钮，将弹出一个对话框，其中可以选择需要分析的应用程序。在选择了一个应用程序以及相应的分析操作后，CLR 分析器将启动该程序并开始收集数据。CLR 分析器为收集到的数据提供了多种不同的统计视图。在图 1-2 中给出了当对 CLR 分析器本身进行分析时的视图。

从图 1-2 中可以看出，CLR 分析器提供了对指定应用程序中的内存分配历史记录以及层次结构关系的详细视图。在随后的章节中，我们将进一步看到其他的 CLR 分析器视图。

所收集的数据都将被输出到一个日志文件中，这个文件默认位置是% windir% \ Temp。日志文件名的格式为

```
Pipe_<pid>.log
```

其中 <pid> 是被分析进程的 ID。

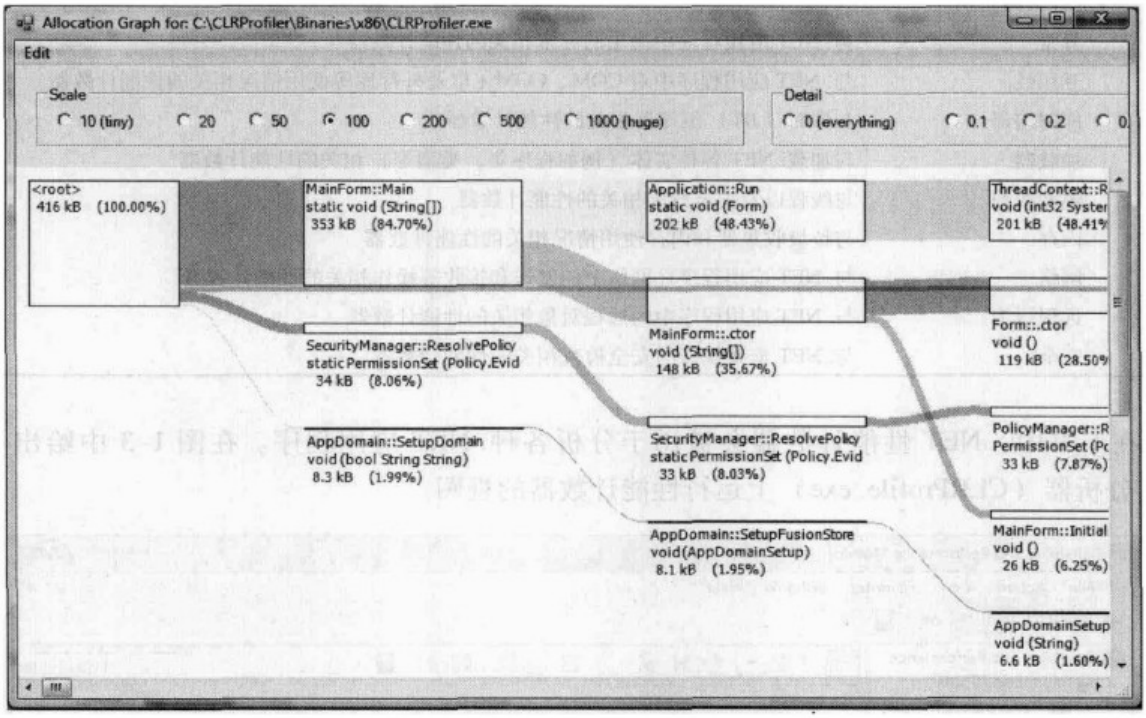


图 1-2　对 CLR 分析器本身进行分析时给出的视图

CLR 分析器还可以通过命令行来启动和控制。通过在运行 CLR 分析器时指定/? 开关，可以给出所有可用的选项。例如，如果要分析应用程序 hello. exe，并且将所有的分析数据都保存到日志文件 hello. log 中，那么可以使用以下命令

```
clrprofiler -o hello.log -p hello.exe
```

其中-o 开关指定的是日志文件名，而-p 开关指定的是将要被分析的可执行程序。

在随后的章节中，在介绍 CLR 分析器的功能时，我们既会使用命令行形式，也会使用图形界面形式。

1.7　性能计数器

在分析问题的过程中，性能计数器是一种非常重要的手段。在安装. NET 框架的同时会安装一组性能计数器。在分析. NET 应用程序的行为时，这些性能计数器将提供各种有用的信息。我们可以通过 Windows 性能监视器（Windows Performance Monitor）来查看这些性能计数器。表 1-2 列出了在. NET 运行时中包含的所有性能计数器类别。

表 1-2　.NET 2.0 性能计数器类别

性能计数器类别	描　述
异常	与.NET 应用程序中抛出的异常相关的性能计数器
互用性	与.NET 应用程序中对 COM、COM+以及外部库等使用情况相关的性能计数器
即时编译	与即时（JIT）编译器相关的性能计数器
加载器	与加载.NET 各种实体（例如程序集，类型等）相关的性能计数器
锁和线程	与线程以及锁定行为相关的性能计数器
内存	与垃圾收集器和内存使用情况相关的性能计数器
网络	与.NET 应用程序在网络上的发送和接收等操作相关的性能计数器
远程行为	与.NET 应用程序中的远程对象相关的性能计数器
安全	与.NET 框架执行的安全检查相关的性能计数器

在本书中,.NET 性能计数器会被用于分析各种.NET 应用程序。在图 1-3 中给出了在 CLR 分析器（CLRProfile.exe）上运行性能计数器的视图。

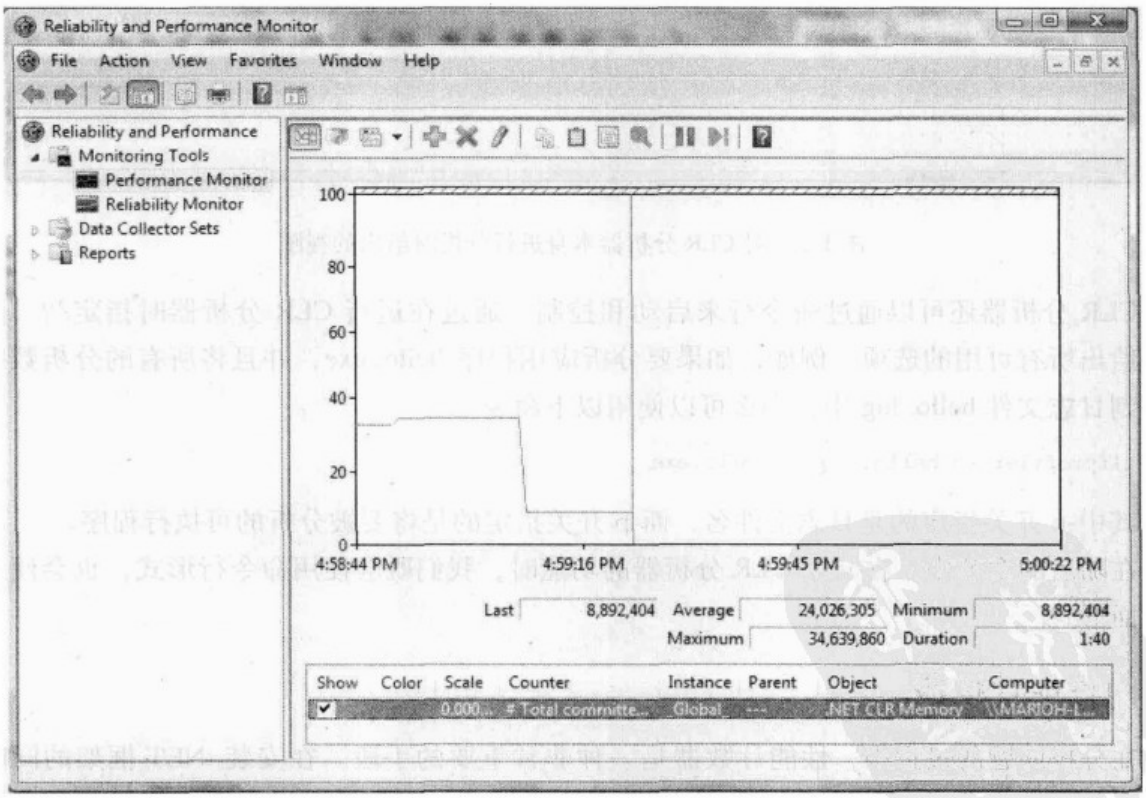


图 1-3　统计 CLR 分析器中所有已提交字节数的性能计数器

1.8　.NET 反编译器

应用场合	分析和反编译.NET 程序集
版本	5.1
下载地址	http：//www.aisto.com/roeder/dotnet/Download.aspx？File = Reflector

.NET 反编译器是一个查看.NET 程序集的工具，它包含了一个强大的反编译器，可以从 MSIL（Microsoft Intermediate Language，Microsoft 中间语言）中生成各种高级语言代码。所支持的高级语言包括 C#、Visual Basic、Delphi、托管 C++ 以及 Chrome。此外，它还包括一种基于插件 API 形式的可扩展模型。目前，有许多插件可供使用，包括代码审查插件以及代码度量（code metric）插件。图 1-4 给出的是通过.NET 反编译器对 Reflector.exe 本身进行分析的结果。

在安装这个工具时，需要在下载位置上输入你的姓名，公司以及 Email 地址。在输入这些信息后，就可以下载软件的安装包（ZIP 文件），然后将其释放到指定的位置。从安装文件夹中运行 reflector.exe 就可以启动.NET 反编译器。

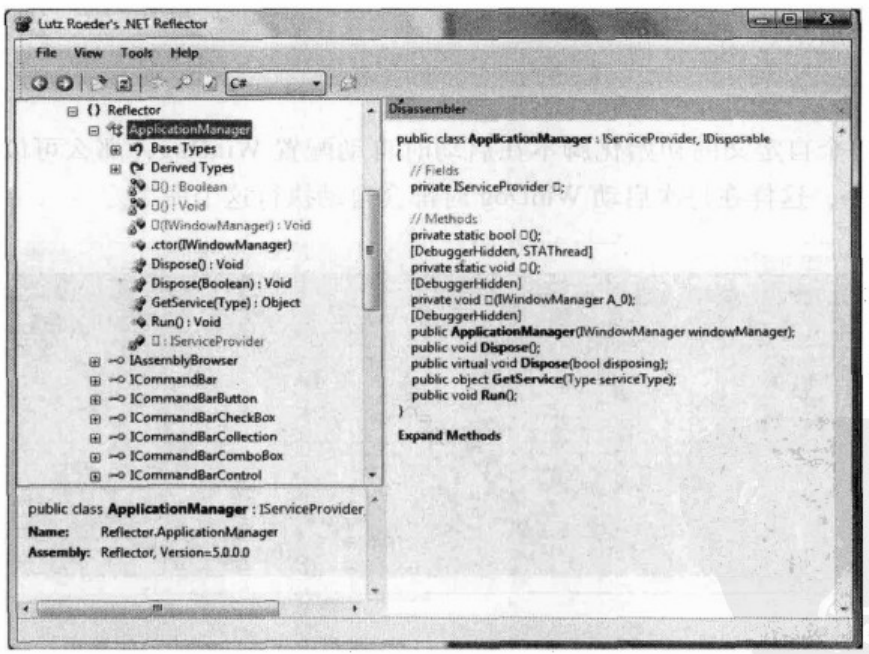


图 1-4　使用.NET 反编译器来分析它自身

1.9　PowerDbg

应用场合	调试器工具
版本	5.0
下载地址	http：//www.codeplex.com/powerdbg

PowerDbg 是由 Roberto Farah 开发的一个库，它使开发人员能够通过 Powershell（至少要求 1.0 版本）来控制非托管调试器。

当希望通过命令行来控制调试器的运行时，这将是一个非常有用的工具。PowerDbg 脚本以一种简洁的形式将信息返回给用户。PowerDbg 的强大之处在于，它易于扩展，并且能够对常用的命令（或者调试任务中的一组命令）进行调用和格式化。

要使用 PowerDbg，最简单的方式就是在 Powershell 配置文件中对其进行初始化。我使用的配置文件是：

```
%windir%\System32\WindowsPowerShell\v1.0\profile.ps1
```

只需把这行 PowerDbg 脚本复制到这个文件，然后重新打开 Powershell 命令窗口即可。现在，就可以使用 PowerDbg 命令了，如图 1-5 中所示。

PowerDbg 的工作原理是向运行中的 WinDbg 发送各种命令。任何命令在执行时，将首先打开一个日志文件，接下来再执行命令，最后关闭日志文件。PowerDbg 脚本通过这个日志文件来分析结果和产生输出信息。需要注意的是，为了使 PowerDbg 能够找到 WinDbg 实例，WinDbg 程序主窗口的名字必须为“WinDbg”。可以通过（WinDbg 中的）. wtitle 命令来修改窗口名字，如下所示：

```
.wtitle PowerDbg
```

如果通过一个自定义的初始化脚本在启动时自动配置 WinDbg，那么可以在脚本文件中增加 . wtitle 命令，这样在每次启动 WinDbg 时都会自动执行这个命令。

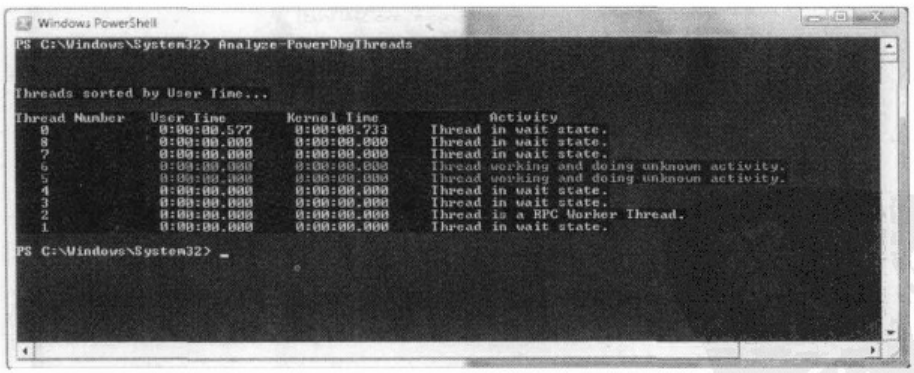


图 1-5　在 WinDbg（perfmon. exe）上运行 PowerDbg 的示例输出

1.10　托管调试助手

应用场合	用于一般性的 CLR 调试
版本	2.0
下载地址	CLR 的一部分

托管调试助手（MDA）并不是一个独立的工具，而是 CLR 中的一个组件，在运行和调试. NET 应用程序时将提供各种有用的信息。如果你熟悉在非托管代码中使用的应用程序验证器（application verifier），那么就会很快熟悉 MDA，二者的作用非常类似。通过对运行时进行监测，可以找出一些常见的编程错误并且能在发布应用程序之前修复它们。MDA 可以分为几类。在表 1-3 中列出了. NET 2. 0 中的各种类别。

表 1-3　CLR 2. 0 中的托管调试助手类别

MDA 类别	描　述
非托管互用性	与非托管互用性问题相关的调试助手
非托管互用性（COM）	与 COM 非托管互用性问题相关的调试助手
非托管互用性（P/Invoke）	与平台调用非托管互用性问题相关的调试助手
加载器	CLR 加载器相关的调试助手
线程	与线程问题相关的调试助手
BCL	与基础类库问题相关的调试助手
其他	其他调试问题

表 1-3 所列出的各种类别中，都包含了一组 MDA 来分析属于这个类别的各种问题。要使用 MDA，必须首先启用它们（在启动被调试进程之前）。启用 MDA 的方式是通过注册表。具体来说，你需要在下面的注册键中添加以下值：

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\.NETFramework\MDA="1"
```

注意，MDA 是一个字符串值。

在设置了注册表键值后，CLR 将需要使用 MDA。在使用它们之前，需要逐一为应用程序启用相应的 MDA。MDA 是通过配置文件来启用的，文件名必须符合以下规则：

```
<appname>.exe.dma.config
```

其中“appname”是需要启用 MDA 的应用程序名字。在配置文件中包含了需要启用的 MDA。我们通过一个示例来说明这个过程。这个示例程序位于以下位置：

源代码文件：C:\adnd\chapter1\MDASample

二进制代码：C:\adndbin\01mdasample. exe

程序的源代码（01mdasample. cs）非常简单，它只是启动一个工作线程（worker thread），这个线程在一个无限循环中自旋（spin），同时执行睡眠操作。在新线程启动之后，主线程将结束工作线程。代码中存在的主要问题是：一个线程将结束另一个线程。由于一些与资源相关的问题，这种方式可能会也可能不会（后者居多）正确地工作。问题是，我们如何使用 MDA 来分析这个潜在的问题？在前面已经讨论过，首先需要通过注册表来启用 MDA。在完成这个操作后，需要创建 MDA 的应用程序配置文件。清单 1-4 给出了示例程序（01MDASample. exe. mda. config）的 MDA 配置文件。

清单 1-4　示例 MDA 配置文件

```
<?xml version="1.0" encoding="UTF-8" ?>
<mdaConfig>
  <assistants>
    <asynchronousThreadAbort />
  </assistants>
</mdaConfig>
```

所有的 MDA 配置文件都必须包含 mdaConfig 标签，而这个标签又包含 assistants 元素。在 assistants 标签下是为这个应用程序启用的一组 MDA。请注意，assistant 标签需要按照字母表顺序进行排序，否则 MDA 将不会生效。在这里的应用程序中，将使用 asynchronousThreadAbort 。只有当这个 MDA 配置文件与应用程序放在一起时，MDA 才会生效。在清单 1-5 中给出了在调试器下运行 01MDASample. exe 的示例输出，其中 MDA 配置文件存放在同一目录下。

清单 1-5　在配置文件中指定了 aynchronousThreadAbort

```
...
...
...
0:000> g
ModLoad: 76800000 768bf000   C:\Windows\system32\ADVAPI32.dll
ModLoad: 77b60000 77c23000   C:\Windows\system32\RPCRT4.dll
ModLoad: 76ad0000 76b25000   C:\Windows\system32\SHLWAPI.dll
ModLoad: 762e0000 7632b000   C:\Windows\system32\GDI32.dll
ModLoad: 76e80000 76f1e000   C:\Windows\system32\USER32.dll
ModLoad: 76d50000 76dfa000   C:\Windows\system32\msvcrt.dll
ModLoad: 765b0000 765ce000   C:\Windows\system32\IMM32.DLL
ModLoad: 76730000 767f7000   C:\Windows\system32\MSCTF.dll
ModLoad: 765d0000 765d9000   C:\Windows\system32\LPK.DLL
ModLoad: 76e00000 76e7d000   C:\Windows\system32\USP10.dll
ModLoad: 75340000 754d4000   C:\Windows\WinSxS\x86_microsoft.windows.common-
controls_6595b64144ccf1df_6.0.6000.20533_none_4634c4a0218d65c1\comctl32.dll
ModLoad: 79e70000 7a3ff000   C:\Windows\Microsoft.NET\Framework\v2.0.50727
\mscorwks.dll
ModLoad: 755e0000 7567b000   C:\Windows\WinSxS\x86_microsoft.vc80
.crt_1fc8b3b9a1e18e3b_8.0.50727.762_none_10b2f55f9bffb8f8\MSVCR80.dll
ModLoad: 76f50000 77a1e000   C:\Windows\system32\shell32.dll
ModLoad: 76330000 76474000   C:\Windows\system32\ole32.dll
ModLoad: 790c0000 79bf6000   C:\Windows\assembly\NativeImages_v2.0
.50727_32\mscorlib\32e6f703c114f3a971cbe706586e3655\mscorlib.ni.dll
ModLoad: 79060000 790b6000   C:\Windows\Microsoft.NET\Framework
\v2.0.50727\mscorjit.dll
ModLoad: 60340000 60348000   C:\Windows\Microsoft.NET\Framework
\v2.0.50727\culture.dll
```

```
ModLoad: 60340000 60348000   C:\Windows\Microsoft.NET\Framework
\v2.0.50727\culture.dll
<mda:msg xmlns:mda="http://schemas.microsoft.com/CLR/2004/10/mda">
<!--
     User code running on thread 6332 has attempted to abort thread 8484. This
     may result in a corrupt state or resource leaks if the thread being aborted
     was in the middle of an operation that modifies global state or uses native
     resources. Aborting threads other than the currently running thread is
     strongly discouraged.
 -->
<mda:asynchronousThreadAbortMsg break="true">
  <callingThread osId="6332" managedId="1"/>
  <abortedThread osId="8484" managedId="3"/>
</mda:asynchronousThreadAbortMsg>
```

从清单 1-5 中我们可以看到在示例程序中执行了一个有问题的操作（结束一个线程），而调试器在这个操作发生时将停止程序的执行。除了停止执行外，MDA 还输出了一些更详细的信息，例如结束工作线程的线程以及工作线程本身的 ID。

在 CLR 中内置了许多有用的 MDA，在随后的章节中将介绍这些 MDA 在哪些调试情况下会给调试工作带来便利。

1.11　小结

如果要列举从历史中得到的一些教训，那么毫无疑问，其中之一就是意识到在软件中包含了极大的复杂性。为了减轻开发人员的工作压力，我们往往会引入新的抽象层，通过统一的框架及相应的运行时来屏蔽底层的各种不同机制，从而降低这种复杂性的程度。.NET 正是一种这样的抽象，它为开发人员提供了极大的帮助。尽管.NET 提供了这种便利，但理解底层的各种构件仍然非常重要。例如，虽然.NET 通过垃圾收集器实现了自动的内存管理功能，但我们遇到的最常见问题之一就是内存泄漏。对于成功的产品开发和技术支持来说，理解垃圾收集器的工作原理并且知道使用哪些工具来分析应用程序和垃圾收集器的行为都是极为关键的。本章简要介绍了在分析.NET 应用程序的问题时可以使用的一些最有用的工具。在随后章节中将详细介绍每种工具，这里只是先向读者介绍这些工具的一些基本概念（例如下载地址、安装过程以及应用场合等）。在本书后面，当我们分析实际问题时，将会充分发挥这些工具的强大功能。

第 2 章　CLR 基础

无论采取什么形式来分析问题，对被调试系统的底层了解得越多，就越有可能成功地找出问题根源。在.NET 领域，这个原则同样适用，即我们需要理解运行时本身的各种功能和行为。了解垃圾收集器的工作原理将使你在调试内存泄漏问题时更加高效。了解互用性的工作原理将使你在调试 COM 问题时更加高效，而了解同步机制的工作原理将使你在调试挂起问题时更为高效，诸如此类的观点不胜枚举。通过应用程序深入到运行时的内部，将极大地提高调试的成功概率。在传统调试方式中需要数星期才能解决的问题，现在或许在很短的时间内就可以解决。

在本章中将简要介绍.NET 运行时。我们会讨论一些在调试.NET 应用程序中非常有用的核心运行时组件及概念，并将通过一些调试器和工具来说明运行时的内部机制。但是，本章并不会对.NET 运行时进行详尽介绍，而只是侧重于介绍一些给开发人员造成问题的常见领域。

2.1　高层概览

从高层面上来看,.NET 是一个虚拟的运行时环境，它包含了一个虚拟执行引擎——通用语言运行时（Common Language Runtime，CLR）以及一组相关的框架库。基于.NET 编写的应用程序在编译时并不会转换为机器代码，而是转换为一种中间语言代码；在实际执行时，执行引擎将（根据不同的架构）把这种代码转换为机器代码。尽管这使得 CLR 看上去像一个解释器（解释中间语言），但在 CLR 和解释器之间存在一个本质差异，即 CLR 不会每次都对中间语言执行转换操作。CLR 每次将一部分中间语言代码转换为机器代码，然后在随后的执行中反复使用这段被转换的机器代码。虚拟执行引擎给开发人员带来了一些非常关键的好处，包括：

- 内存管理
- 安全管理
- 代码验证

为了更好地理解.NET 由哪些组件构成，图 2-1 给出了在.NET 中包含的各种不同组件。

.NET 的核心是一个 ECMA 标准，表示.NET 运行时的所有实现都要遵从这个标准。描述这个标准的文档通常称为通用语言基础架构（Common Language Infrastructure，CLI）。CLI 不仅为运行时本身定义了一组规则，还包括一组非常关键的通用类库。这组类库被称为基础类型库（Base Class Libraries，BCL）。在图 2-1 中，接下来的外面一层就是通用语言运行时

（CLR）。这个组件代表 Microsoft 对 CLI 的实现。当在机器上安装. NET 可再发行组件时，其中就包含了 CLR。在 CLR 之外是. NET 框架。. NET框架包含了开发人员在编写. NET 应用程序时需要使用的所有库。. NET 框架可以看成是 BCL 的一个超集，它包含了各种框架，例如 Windows 通信基础框架（Windows Communication Foundation，WCF）、Windows 表示基础框架（Windows Presentation Foundation，WPF）等。还有一些库虽然属于. NET 框架，但却不属于 BCL，这些库被认为不属于 ECMA 标准，如果在应用程序中使用了这些库，那么就可能无法在 CLR 之外的其他 CLI 上运行。在顶层是各种. NET 应用程序，它们需要在 CLR 环境内运行。本书的重点在于介绍 CLR 的工作原理及其对调试. NET 应用程序的重要性。

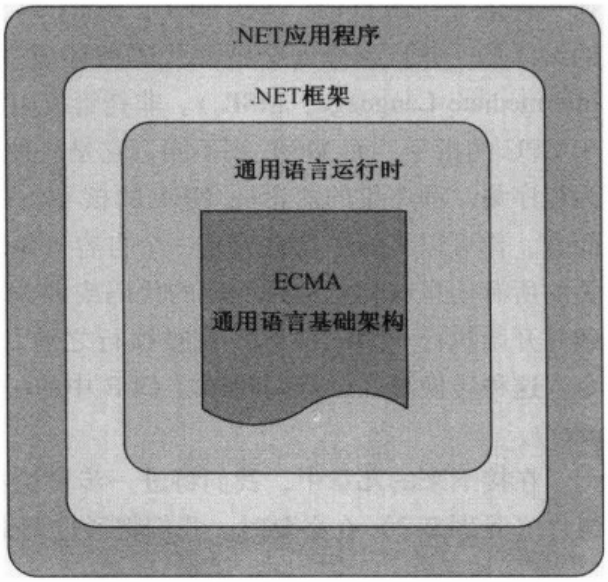


图 2-1　. NET 中包含的各种不同组件

有没有其他一些实现同样遵从 CLI

Microsoft 的 CLR 是否是唯一的 CLI 实现？不全对。由于 CLI 正得到越来越多的应用，因此有许多公司或机构都开发了自己的 CLI 实现。Mono 项目（由 Novell 赞助）就是其中的一个示例。Mono 的 CLI 实现是一个开源项目，它可以在 Windows、Linux、Solaris 以及 Mac OS X 等系统上运行。

此外，Microsoft 也发布了基于共享源代码的通用语言基础架构（Shared Source Common Language Infrastructure，2. 0），也被称为 Rotor 项目，它包括一个遵从 ECMA 标准的 CLI 实现。由于源代码是共享的，因此通过这个项目可以深入了解 CLI 实现的工作原理。

既然 CLR 需要负责. NET 应用程序执行的各个方面，那么通常的执行流程会怎样？在图 2-2 中给出了这种执行模型的示意图，首先从应用程序的源代码开始。

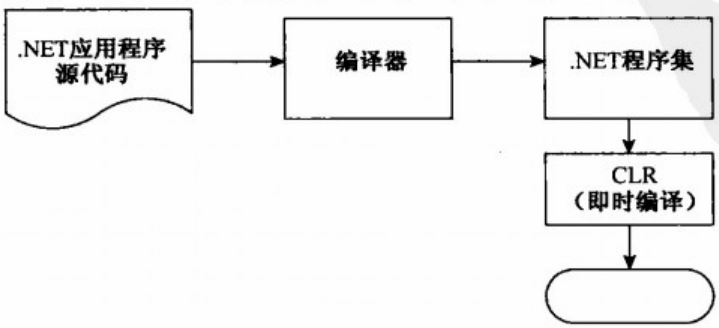


图 2-2　. NET 执行模型示意图

执行模型的起始点是.NET 应用程序的源代码。编写源代码的语言可以是.NET 支持的任何一种语言，例如 C#、VB.NET、托管C++ 以及其他一些语言等。然后，源代码被交给相应的编译器，并且编译器会把源代码编译成一种中间语言，即 Microsoft 中间语言（Microsoft Intermediate Language，MSIL）。非托管应用程序的源代码在被编译和链接之后将转换为特定于 CPU 的指令，而 MSIL 则不同，它是一种与平台无关的更高级的语言。编译的输出结果即为程序集。程序集的概念是.NET 的核心，在本章的稍后部分将给出更详细的介绍。就目前而言，你可以将程序集看成是一个自容（Self－Contained）的实体，它包含了与应用程序相关的所有信息（包括应用程序的代码或者 MSIL）。当运行.NET 程序集时，CLR 将被自动加载并开始执行 MSIL。MSIL 在被执行之前，首先被转换为与代码运行平台相对应的机器指令。这种转换是在运行期间通过 CLR 中的一个组件来完成的，也就是即时编译器（JIT Compiler）。

在接下来的几章中，我们将进一步讨论在.NET 应用程序的执行流程中涉及的各种不同组件（见图 2-2）在必要时，我们将通过调试器及相应的工具来说明相关概念。

2.2 CLR 和 Windows 加载器

在 Windows 上运行的应用程序可以通过多种不同的方式来启动。Windows 负责处理所有的相关工作，包括设置进程地址空间、加载可执行程序，以及指示处理器开始执行等。当处理器开始执行程序指令时，它将一直执行下去，直到进程退出。在本章的前面已经讨论过，在.NET 应用程序中并不包含机器指令。因此，在处理器开始执行前，首先需要将中间语言转换为机器指令。尽管存在着这种差异，.NET 应用程序仍然可以采取与非托管应用程序一样的启动方式。为什么这种方式可行？Windows 加载器是否知道.NET 应用程序的一些特定信息，因此能够自动地启动 CLR？答案在于 Windows 上一种由来已久的文件格式：可移植的可执行文件格式（Portable Executable，PE）。在图 2-3 中给出了 PE 映像文件的一般结构。

为了支持 PE 映像的执行，在 PE 的头部包含了一个域叫做 AddressOfEntryPoint。这个域表示 PE 文件的入口点（Entry point）位置。在.NET 程序集中，这个值指向.text 段中的一小段存根（stub）代码。下一个重要的域就是数据目录（data directory）。当.NET 编译器生成程序集时，它会在 PE 文件中增加一个数据目录项。具体来说，这个数据目录项的索引为 15，其中包含了 CLR 头的位置和大小。然后，根据这个位置在 PE 文件中找到位于.text 段中的 CLR 头。在 CLR 头中包含了一个结构 IMAGE_COR20_HEADER。在这个结构中包含了许多信息，例如托管代码应用程序的入口点，目标 CLR 的主版本号和从版本号，以及程序集的强名称签名（strong name signature）等。根据这个数据结构中包含的信息，Windows 可以知道要加载哪个版本的 CLR 以及关于程序集本身的一些最基本信息。在.text 段中还包含了程序集的元数据表，MSIL 以及非托管启动存根代码。非托管启动存根代码包含了由 Windows 加载器执行以启动 PE 文件执行的代码。

在接下来的章节中，我们将看到 Windows 加载器如何加载非托管映像和. NET 程序集。

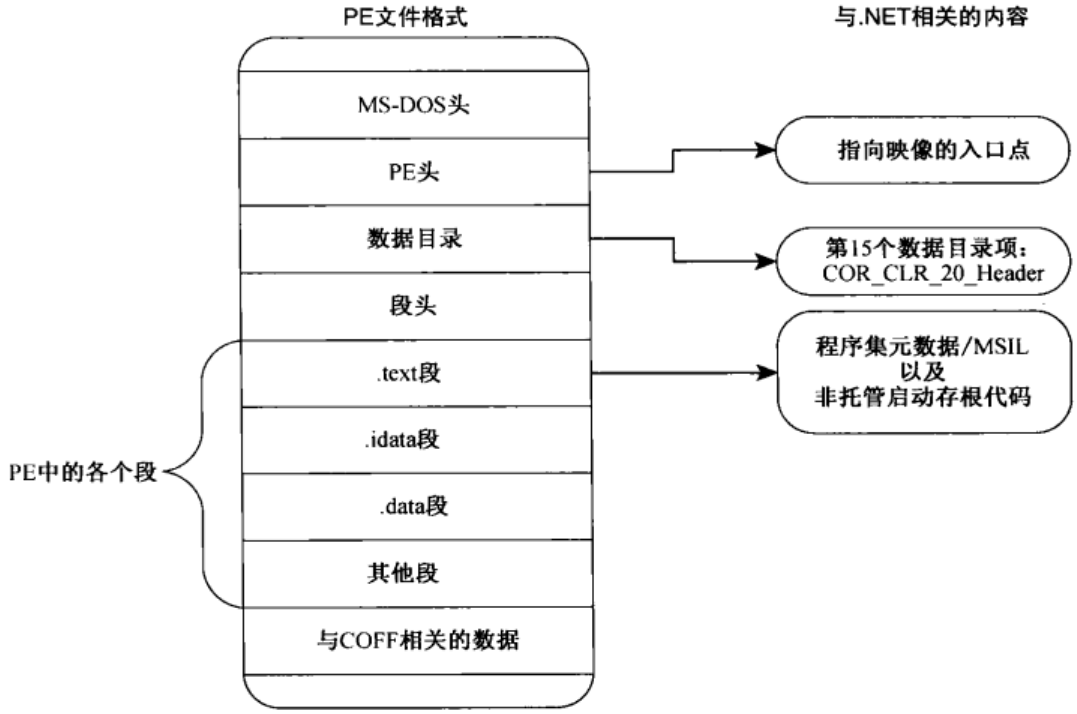


图 2-3　PE 文件格式概览

2.2.1　加载非托管映像

为了更好地理解. NET 程序集的加载过程，我们首先来看看 Windows 加载器是如何加载非托管的 PE 映像的。我们以 notepad. exe 为例（在 Windows Vista 企业版上运行）。注意，在分析 PE 文件时，需要首先理解两个重要的术语：

- 文件偏移（file offset）：指 PE 文件中任意位置的偏移。
- 相对虚地址（Relative Virtual Address，RVA）：仅当 PE 映像已经被加载后才需要使用这个值，它是在进程虚拟地址空间中的相对地址。例如，一个值为 0x200 的 RVA 表示当映像被加载到内存后离映像基地址 0x200 字节的位置。

用一个十六进制编辑器打开 notepad. exe，找到偏移 0x108 处，你将看到 AddressOfEntryPoint 域（记住这个域表示这段代码的起始执行位置）以及 RVA 值 0x31F8。接下来，在调试器下加载 notepad. exe，并且反汇编位于 < 映像基址 > + 0x31F8 处的代码，如清单 2-1 所示。

清单 2-1 找到 notepad.exe 的起始地址

```
Microsoft (R) Windows Debugger Version 6.8.0004.0 X86
Copyright (c) Microsoft Corporation. All rights reserved.

CommandLine: notepad.exe
Symbol search path is: *** Invalid ***
*****************************************************************************
* Symbol loading may be unreliable without a symbol search path.            *
* Use .symfix to have the debugger choose a symbol path.                    *
* After setting your symbol path, use .reload to refresh symbol locations. *
*****************************************************************************
Executable search path is:
ModLoad: 000b0000 000d8000   notepad.exe
ModLoad: 773c0000 774de000   ntdll.dll
ModLoad: 77180000 77258000   C:\Windows\system32\kernel32.dll
ModLoad: 770c0000 7717f000   C:\Windows\system32\ADVAPI32.dll
ModLoad: 76cc0000 76d83000   C:\Windows\system32\RPCRT4.dll
ModLoad: 75f20000 75f6b000   C:\Windows\system32\GDI32.dll
ModLoad: 77560000 775fe000   C:\Windows\system32\USER32.dll
ModLoad: 75c80000 75d2a000   C:\Windows\system32\msvcrt.dll
ModLoad: 75dc0000 75e34000   C:\Windows\system32\COMDLG32.dll
ModLoad: 77500000 77555000   C:\Windows\system32\SHLWAPI.dll
ModLoad: 74ce0000 74e74000   C:\Windows\WinSxS\x86_microsoft.windows.common-
controls_6595b64144ccf1df_6.0.6000.20533_none_4634c4a0218d65c1\COMCTL32.dll
ModLoad: 75f70000 76a3e000   C:\Windows\system32\SHELL32.dll
ModLoad: 6ffc0000 70001000   C:\Windows\system32\WINSPOOL.DRV
ModLoad: 76d90000 76ed4000   C:\Windows\system32\ole32.dll
ModLoad: 77260000 772ec000   C:\Windows\system32\OLEAUT32.dll
(3e28.3808): Break instruction exception - code 80000003 (first chance)
eax=00000000 ebx=00000000 ecx=0012f8c8 edx=77420f34 esi=fffffffe edi=77485d14
eip=77402ea8 esp=0012f8e0 ebp=0012f910 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for ntdll.dll -
ntdll!DbgBreakPoint:
77402ea8 cc              int     3
0:000> .symfix
No downstream store given, using c:\Program Files\Debugging Tools for Windows\sym
0:000> .reload
Reloading current modules
...............
0:000> u 000b0000+0x31F8
notepad!WinMainCRTStartup:
000b31f8 e8f7feffff      call    notepad!__security_init_cookie (000b30f4)
000b31fd 6a58            push    58h
000b31ff 6810330b00      push    offset notepad!WinSqmAddToStream+0x2a (000b3310)
000b3204 e80b090000      call    notepad!_SEH_prolog4 (000b3b14)
000b3209 33db            xor     ebx,ebx
000b320b 895de4          mov     dword ptr [ebp-1Ch],ebx
000b320e 895dfc          mov     dword ptr [ebp-4],ebx
```

```
000b3211 8d4598          lea     eax,[ebp-68h]
0:000>
```

从清单 2-1 中可以看到，这个 notepad. exe 实例被加载在地址 0x000b0000 处。当对位于 AddressOfEntryPoint（0x31F8）加上这个基地址后得到的地址处的代码进行反编译时，可以看到它对应于 WinMainCRTStartup 函数，这个函数也是应用程序的入口点。WinMainCRTStartup 是一个外层包装函数，它在调用 Notepad. exe 的 WinMain 函数之前执行一些 CRT 初始化工作。在 PE 文件中还包含大量其他信息，Windows 加载器在加载映像的过程中会用到这些信息，但从更高的层面上来看，这里说明的流程只是如何找到并执行 PE 映像文件的入口点。

接下来，我们将分析. NET 程序集的结构，并说明这种结构如何使 Windows 加载器支持. NET 程序集的执行。

2.2.2 加载. NET 程序集

分析. NET 应用程序启动过程的最佳方式就是观察一个简单的. NET 命令行程序。程序的源代码和程序集分别位于以下文件夹中：

- 源代码文件：C:\adnd\chapter1\MDASample
- 程序集文件：C:\adndbin\01mdasample. exe

如果运行上面的程序，它会成功执行，如清单 2-2 所示。

清单 2-2 执行 02simple. exe

```
C:\ADNDBin:>02Simple.exe
Welcome to Advanced .NET Debugging!
```

由于. NET 应用程序在执行时要预先加载 CLR，那么 Windows 如何知道加载并初始化 CLR？我们可能会做出一种假设：系统开发人员对 Windows 加载器进行了改动以识别. NET 程序集，并且当检测到. NET 程序集时自动启动 CLR。尽管这种假设只有部分是正确的，但在. NET 之前的 Windows 上执行. NET 应用程序时，必须首先打补丁。回答这个问题的关键在于：对 PE 格式进行了扩展。在前面已经提到过，PE 格式是 Windows 可执行程序的文件格式，用来管理 PE 文件中代码的执行。可执行程序包括 EXE、DLL、OBJ、SYS 等文件。为了支持. NET，在 PE 文件格式中增加了对程序集的支持，如图 2-3 所示。

现在，我们来分析当加载器遇到一个. NET 应用程序时将发生哪些动作。在这个示例中使用 02simple. exe，这个程序位于 C:\ADNDBin。需要注意的是，这个示例程序是在 Windows 2000 上运行。之所以要在一个旧版本的 Windows 上运行，是因为在 Windows 2000 之后的版本中增加了一些改动，而这些改动将影响 Windows 加载器加载. NET 程序集的方式（在本章的后面将进行介绍）。为了更好地说明这些概念，我使用了一个工具 dumpbin. exe，它能

够解析 PE 文件格式并且以简洁易读的形式转储出 PE 文件的内容。在 02simple. exe 上运行 dumpbin. exe 的结果被保存在一个文件中：

```
C:\ADNDBin\02simple.txt
```

第一部分值得注意的信息是在 Optional Header Values 段中，如下所示：

```
OPTIONAL HEADER VALUES
             10B magic # (PE32)
            8.00 linker version
             600 size of code
             600 size of initialized data
               0 size of uninitialized data
            246E entry point (0040246E)
            2000 base of code
```

上面的 Entry Point 域对应于 PE 文件中的 AddressOfEntryPoint 域，值为 0x00402464。要找出位置 0x00402464 所对应的代码，需要查看 PE 映像中的. text 段，具体来说就是如下面清单中的 RAW DATA 段：

```
00402430: 00 00 00 00 00 00 00 00 00 00 00 00 50 24 00 00  ............P$..
00402440: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00402450: 00 00 5F 43 6F 72 45 78 65 4D 61 69 6E 00 6D 73  .._CorExeMain.ms
00402460: 63 6F 72 65 65 2E 64 6C 6C 00 00 00 00 00 FF 25  coree.dll.....ÿ%
00402470: 00 20 40 00                                      . @.
```

上面信息中的粗体字节对应于 AddressOfEntryPoint ，这些字节对应的机器指令为：

```
JMP 402000
```

这里有一个问题：402000 表示什么意思？事实上，0x402000 指向的是 PE 映像文件中的另一部分，也就是 import 段，在这个段中列出的是 PE 文件依赖的所有模块。在加载时，系统将修正导入函数的实际地址，并执行正确的调用。要找到 0x402000 指向的内容，我们可以查看 PE 文件的导入段，可以发现以下内容：

```
mscoree.dll
            402000 Import Address Table
            40243C Import Name Table
                 0 time date stamp
                 0 Index of first forwarder reference

                 0 _CorExeMain
```

可以看到，0x402000 指向的是 mscoree. dll（Micorsoft 对象运行时执行引擎，Microsoft Object Runtime Execution Engine），这个库中包含了一个导出函数_CorExeMain。然后，前面的 JMP 指令可以转换为以下伪码：

```
JMP _CorExeMain
```

我们已经看到了，_CorExeMain 是 mscoree. dll 的一部分，这个函数也是在加载. NET 程序集时第一个被调用的函数。mscoree. dll（和_CorExeMain）的主要作用就是启动 CLR。mscoree. dll 在启动 CLR 时将执行一系列的工作：

1）通过查看 PE 文件中的元数据，找出. NET 程序集是基于哪个版本的 CLR 构建的。

2）找出操作系统中正确版本 CLR 的路径。

3）加载并初始化 CLR。

在 CLR 被初始化之后，在 PE 映像的 CLR 头中就可以找到程序集的入口点（Main()）。然后，JIT 开始编译并执行入口点。到目前为止，我们所谈到的 CLR 还只是一个逻辑组件，而并没有提到它的各项功能具体由哪些映像来实现。CLR 的大部分功能是由 mscorwks. dll 来实现的。而且，在任何一台机器上都可能有多个版本的 mscorwks. dll。例如，如果安装了. NET 1. 1 和. NET 2. 0，那么在机器上将存在以下 CLR DLL：

- C:\Windows\Microsoft. NET\Framework\v1. 1. 4322\mscorwks. dll
- C:\Windows\Microsoft. NET\Framework\v2. 0. 50727\mscorwks. dll

之所以要安装多个版本，是因为不同的. NET 应用程序可能需要不同版本的 CLR。基于 CLR 1. 0 编写的应用程序需要正确地加载 1. 0 版本的 CLR，即使有. NET 2. 0 也不行。这种机制本质上是实现. NET 中的平行执行模型（Side-By-Side Execution Model）。mscoree. dll 的作用是通过查看 PE 映像文件中的 CLR 头来找出程序集需要使用哪个版本的 CLR。具体来说，mscoree. dll 将查看 CLR 头中的 MajorRuntimeVersion 和 MinorRuntimeVersion 两个域，并且加载正确版本的 CLR。

到目前为止，我们已经介绍了. NET 程序集（后缀名为. EXE）的整个启动流程。正如非托管的 Windows 应用程序能够支持像动态库这种可执行形式，. NET 也有着同样的方式。就加载器而言，. NET 库与. NET 可执行程序之间的唯一差别就是，在 PE 映像中导入的函数不是_CorExeMain，而是_CorDllMain。

在加载 mscoree. dll 时有一个值得注意的问题，即为什么需要非托管的存根函数来调用_CorExeMain？因为 PE 映像文件包含了一个. NET 头，那么 Windows 加载器是否也可以将这个 PE 映像识别为一个. NET 程序集并且自动加载 mscoree. dll？是的，的确如此。在 Windows XP 以及之后的版本中对 Windows 加载器进行了升级，使其能够识别出一个. NET 程序集的 PE 映像，并且自动加载 CLR。

. NET 程序集的加载算法总结如下：

1）用户执行一个. NET 程序集。

2）Windows 加载器查看 AddressOfEntryPoint 域，并找到 PE 映像文件的. text 段。

3）位于 AddressOfEntryPoint 位置上的字节只是一个 JMP 指令，这个指令跳转到 mscoree. dll 中的一个导入函数。

4）将执行控制转移到 mscoree. dll 中的函数_CorExeMain 中，这个函数将启动 CLR 并且

把执行控制转移到程序集的入口点。

PE 文件格式是一种多用途的格式（从它可以很容易支持.NET 程序集的特性也说明了这一点），它包含了与被加载和执行的 PE 映像相关的大量信息。本节内容重点介绍了如何通过扩展 PE 文件格式来支持.NET 程序集的执行。接下来，我们将深入分析 CLR 中其他的关键内容，首先介绍应用程序域。

2.3 应用程序域

要提高代码的可靠性，就必须实现某种代码隔离层（或者说独立层）。隔离层要确保运行在某个隔离边层内的代码不会对其他隔离层中的代码产生负面影响。如果不能实现这种保证，那么一些不良行为的代码将很容易破坏其他的应用程序或者整个系统。换句话说，这种隔离层的作用就是提升系统的稳定性与可靠性。Windows 通过进程来实现这种隔离机制。所有的可执行代码、数据，以及其他资源都被包含在进程中，系统中其他进程通常不允许对它们进行访问（除非持有足够的权限）。同理,.NET 应用程序同样是被局限在进程内执行。但是,.NET 还进一步引入了另一种逻辑隔离层，称之为应用程序域。在图 2-4 中给出了进程与应用域之间的关系。

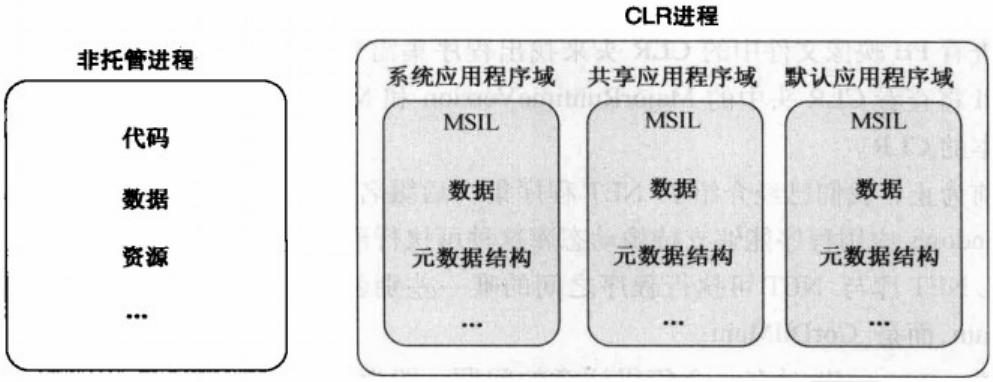


图 2-4 进程与应用程序域

在任何启动了 CLR 的 Windows 进程中都会定义一个或多个应用程序域，在这些域中包含了可执行代码、数据、元数据结构以及资源等。除了进程本身带有的保护机制外，应用程序域还进一步引入了以下保护机制：

- 在某个应用程序中的错误代码不会影响到同一进程中另一个应用程序域中运行的代码。
- 运行在某个应用程序域中的代码不能直接访问另一个应用程序域中的资源。
- 在每个应用程序域中都可以配置与代码特定的信息。例如，可以在每个应用程序域中配置不同的安全设置。

通常，应用程序域对于应用程序本身来说是透明的，大多数应用程序都不会显式地创建

任何应用程序域。只有那些需要在同一个进程中运行特定代码，并要求实现某种程度隔离性的应用程序才会创建新的应用程序域。为了确保运行的代码不会对系统的其他部分造成破坏，这些代码将被加载到自己的应用程序域中。例如，Internet 信息服务器（Internet Information Server，IIS）可以支持多个 ASP. NET 页面，我们可以将运行环境配置为：每个页面在同一进程的不同应用程序域中运行。对于没有显式创建应用程序域的应用程序来说，CLR（在加载的时候）将创建三个应用程序域：系统应用程序域，共享应用程序域以及默认应用程序域。换句话说，启动了 CLR 的进程在运行时至少拥有三个应用程序域（尽管程序本身并没有显式地创建任何应用程序域）。我们可以通过非托管调试器来查看当前进程中存在哪些应用程序域。首先在调试器下启动. NET 应用程序 02simple. exe，然后执行 SOS 调试器扩展支持的 dumpdomain 命令，如清单 2-3 所示。

清单 2-3 通过 dumpdomain 命令来显示 02simple. exe 中的所有应用程序域

```
...
...
...
0:001> .loadby sos.dll mscorwks
0:001> !dumpdomain
-----------------------------------------------
System Domain: 7a3bc8b8
LowFrequencyHeap: 7a3bc8dc
HighFrequencyHeap: 7a3bc934
StubHeap: 7a3bc98c
Stage: OPEN
Name: None
-----------------------------------------------
Shared Domain: 7a3bc560
LowFrequencyHeap: 7a3bc584
HighFrequencyHeap: 7a3bc5dc
StubHeap: 7a3bc634
Stage: OPEN
Name: None
Assembly: 004647d0
-----------------------------------------------
Domain 1: 0041fd40
LowFrequencyHeap: 0041fd64
HighFrequencyHeap: 0041fdbc
StubHeap: 0041fe14
Stage: OPEN
Name: 02Simple.exe
Assembly: 004647d0
[C:\Windows\assembly\GAC_32\mscorlib\2.0.0.0__b77a5c561934e089\mscorlib.dll]
ClassLoader: 00464868
SecurityDescriptor: 00458250
  Module Name
790c2000 C:\Windows\assembly\GAC_32\mscorlib\2.0.0.0__b77a5c561934e089\mscorlib.dll
```

```
Assembly: 0046d660 [C:\ADNDBin\02Simple.exe]
ClassLoader: 0046d6f8
SecurityDescriptor: 0046d560
  Module Name
003f2c3c C:\ADNDBin\02Simple.exe
```

从清单 2-3 中可以看到，在这个进程中有三个应用程序域：System、Shared、Domain 1。其中 Domain 1 是默认的应用程序域，它的名字就是映像本身的名字（02simple. exe）。在每个应用程序域的输出信息中包含以下内容：

- 指向应用程序域的指针。这个指针可以作为 dumpdomain 命令的输入参数，这样将只输出指定应用程序域的信息。例如，可以执行以下命令：

  ```
  !dumpdomain 7a3bc8b8
  ```

- 这个命令只会输出系统应用程序域的信息。
- LowFrequencyHeap，HighFrequencyHeap 以及 StubHeap。通常，每个应用程序域都有与之相关的 MSIL 代码。在 JIT 编译 MSIL 的过程中，JIT 编译器需要保存与编译过程相关的数据。例如，编译生成的机器代码和方法表等。因此，每个应用程序域都需要创建一定数量的堆来存储这些数据。在 LowFrequencyHeap 中包含的是一些较少被更新或者被访问的数据，而在 HighFrequencyHeap 中包含的则是被频繁访问的数据。最后一个堆是 StubHeap，在这个堆中包含的是 CLR 执行互用性调用（例如 COM 互用性或者平台调用）时需要的辅助数据。
- 在应用程序域中加载的所有程序集。从清单 2-3 中可以看出，02simple. exe 这个应用域加载了两个程序集：mscorlib. dll 和 02simple. exe。除了给出已加载. NET 程序集的版本外，还给出了底层程序集数据结构的地址。例如，02simple. exe 程序集的地址为 0x004647d0。在本章的稍后部分将用到这个程序集地址。

我们可以看到，非托管调试器有助于收集. NET 进程中与隔离边界（即应用程序域）相关的信息。最后需要讨论的是这三个应用程序域的作用。记住，即使. NET 应用程序没有显式地创建任何应用程序域，仍然会存在三个应用程序域：系统、共享以及默认。接下来将分别介绍这三个应用程序域的作用。

为什么不使用多个进程而使用多个应用程序域

这是在讨论应用程序域时经常提到的一个问题。答案是，构造和管理进程的开销是非常高的。如果使用应用程序域这种逻辑概念，那么将极大地降低在创建与销毁隔离层时需要的开销。

2.3.1 系统应用程序域

系统应用程序域实现的主要功能如下：

1）创建其他两个应用程序域（共享应用程序域和默认应用程序域）。

2）将 mscorlib. dll 加载到共享应用程序域中（在下面将进一步讨论）。

3）记录进程中所有其他的应用程序域，包括提供加载/卸载应用程序域等功能。

4）记录字符串池中的字符串常量，因此允许任意字符串在每个进程中都存在一个副本。

5）初始化特定类型的异常，例如内存耗尽异常，栈溢出异常以及执行引擎异常等。

2.3.2 共享应用程序域

在共享应用程序域中包含的是与应用程序域无关的代码。mscorlib. dll 将被加载到这个应用程序域中（由系统应用程序域负责加载），此外还包括在 System 命名空间中的一些基本类型（例如 String、enum、ValueType、Array 等）。在大多数情况下，非用户代码（non-user code）将被加载到共享应用程序域中，不过也有一些机制可以将用户代码（user code）加载到共享应用程序域中。启用了 CLR 的应用程序域可以通过加载器的优化属性来注入用户代码。

2.3.3 默认应用程序域

通常，. NET 程序在默认应用程序域中运行。位于默认应用程序域中的所有代码都只有在这个域中才是有效的。由于应用程序域实现了一种有逻辑并且可靠的边界，因此任何跨越应用程序域的访问操作都必须通过. NET 远程对象来进行。

在处理某些特定的问题时，如果能够理解应用程序域的本质，知道哪些应用程序域会被自动创建，以及应用程序域是可以被动态创建的，那么能给问题分析带来巨大的帮助。在第二部分调试实践中，我们将看到一些由于错误使用应用程序域而导致的问题。

到目前为止，我们已经介绍了. NET 应用程序的加载过程，隔离边界（也就是应用程序域），以及运行不同代码所在的位置，因此接下来就可以观察在程序集中包含的内容以及 CLR 与程序集之间的交互。

2.4 程序集简介

虽然我们多次提到了程序集，但并没有对程序集进行详细讨论。从高层面来看，程序集是. NET 应用程序的主要构件和部署单位，可以作为其他组件的一种自描述（Self-Describing）的逻辑容器。这里的自描述是指，在程序集中包含了能够唯一标识和描述程序集的所有必要信息。对程序集进行唯一标识和描述是非常重要的，它能够确保在加载和绑定程序集时不会产生冲突或者依赖于其他的配置数据。程序集的自包含性（Self-Contained）对于消除

DLL 地狱（Dll Hell，指由于 DLL 不同版本之间的冲突而导致的问题）起到了极大的作用。

共有两种类型的程序集：

- 共享程序集（shared assemblies）是指可以在不同. NET 应用程序中使用的程序集。例如，框架程序集就是一种共享程序集。由于共享程序集可以跨越不同的. NET 应用程序，因此在程序集上使用了强命名（strong name）。共享程序集必须完整地定义版本信息，以便使得 CLR 能够绑定到正确的版本。通常，共享程序集会被安装到全局程序集缓存（Global Assembly Cache，GAC）中。
- 私有程序集是指某个应用程序/组件的组成部分，通常不适合由其他应用程序/组件使用。私有程序集通常被部署在与应用程序安装目录相同的文件夹（或者子文件夹）中。由于这种程序集只会在有限范围内使用，因此对版本信息完整性的要求就相对松一些。

当加载私有程序集时，它通常只会局限于某个应用程序域中。根据之前对应用程序域的讨论，我们知道在一个. NET 应用程序中通常会包含三个应用程序域。除了系统应用程序域和共享应用程序域之外，程序集要么是被加载到默认应用程序域中，要么是被加载到显式创建的应用程序域中。当程序集被加载到某个应用程序域时，它将停留在这个应用程序域中，直到这个应用程序域被销毁。由于程序集都是局限在某个应用程序域中，那么对于任何一个应用程序域，我们如何找出其中加载了哪些程序集？在本章的前面，我们使用了 SOS 的 dumpdomain 命令来转储出某个进程中所有的应用程序域。在 dumpdomain 命令的输出中包含了每个应用程序域中加载的所有程序集。清单 2-3 给出了在 02simple. exe 上执行扩展命令 dumpdomain 时输出的信息。我们可以看到，在默认的应用程序域中包含了两个已加载的程序集：02simple. exe 和 mscorlib. dll。此外，程序集的名字同样也是它们的地址。当使用 SOS 的 dumpassembly 命令来获取每个程序集的进一步信息时，需要用到这些地址。我们可以使用扩展命令 dumpassembly，并将程序集的地址作为命令参数来获得更多的信息，如清单 2-4所示。

清单 2-4　dumpassembly 命令的使用示例

```
0:000> !dumpassembly 0034db98
Parent Domain: 002ffd38
Name: C:\ADNDBin\02Simple.exe
ClassLoader: 0034e4b8
SecurityDescriptor: 006d7fe8
  Module Name
000e2c3c C:\ADNDBin\02Simple.exe
```

除了名字之外，还显示了程序集的安全描述符，以及父应用程序域。通过将父应用程序域的地址作为参数传给 SOS 扩展命令 dumpdomain，可以输出加载程序集的应用程序域的相

关信息。

在前面已经提到了，程序集都是自描述的。为了进一步理解自描述程序集的含义，我们首先来看程序集清单。

2.5 程序集清单

既然程序集是.NET 应用程序的基础构件并且是完全自描述的，那么这些描述性信息被保存在什么位置？答案是，位于程序集的元数据段，这也被称为程序集的清单。通常，程序集清单位于程序集的 PE 文件中，但并非一定要在这个位置。例如，如果在程序集中包含了多个模块，那么这个程序集的清单就可以保存在一个单独的文件中（也就是在程序集的 PE 文件中只包含了清单），并且在这个文件中包含的是在加载和使用各个模块时需要用到的引用数据。在图 2-5 中给出了一个单文件程序集和一个多文件程序集。

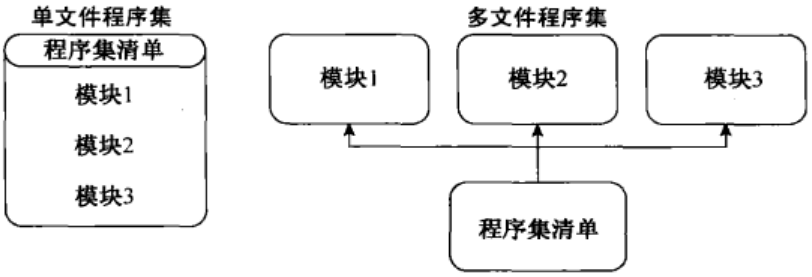


图 2-5 单文件程序集和多文件程序集的示例

在实际情况中，大多数程序集都是单文件程序集。接下来的一个问题是：在程序集清单中包含了哪些内容？在程序集清单中通常包含了以下信息：

- 需要依赖的非托管代码模块列表；
- 需要依赖的程序集列表；
- 程序集的版本；
- 程序集的公钥标记（Public Key Token）；
- 程序集的资源；
- 程序集标志，例如栈的预留空间、子系统等。

查看程序集清单最好的方式就是使用 ILDasm 工具。这个工具是随.NET 2.0 SDK 一起安装的，它可以显示丰富的程序集信息。要查看程序集的清单，从命令行启动 ildasm.exe，并将程序集的名字作为参数。我们通过示例程序 02simple.exe 来说明这个工具的用法。当 ILDasm 被启动时，双击 Manifest 段，会打开一个新窗口，其中包含了所有的清单数据。在图 2-6中给出了 02simple.exe 的清单数据。

在图 2-6 中给出了程序集 02simple.exe 的清单数据，其中第一部分数据指出这个程序集依赖于一个外部程序集 mscorlib.dll。如果这个程序集还引用了其他的程序集，那么它们也会

被列在这部分数据中。请注意，在每个外部程序集的引用信息中包含了被签名程序集的公钥标记，以及被引用程序集的版本。这个版本信息是非常重要的，CLR 加载器可以根据这个信息获取在构建程序集时使用的版本。清单中接下来的一部分数据是关于清单本身的信息。这些信息包括散列算法以及程序集的版本。在上面的示例中没有指定版本，因此版本号被设置为0:0:0:0。在最后一部分数据中包含了这个模块（02simple. exe）的一些信息。虽然02simple. exe 的程序集清单相对简单，但却很容易理解程序集自描述性的本质。在一些更为复杂的程序集中虽然包含了更多的数据，但它们都遵循上面相同的规则。

现在，我们已经了解了. NET 程序集的加载过程、应用程序域的概念，以及程序集和程序集清单的内容，接下来将讨论另一种自描述数据的形式，也就是在程序集中包含的类型元数据（type metadata）。

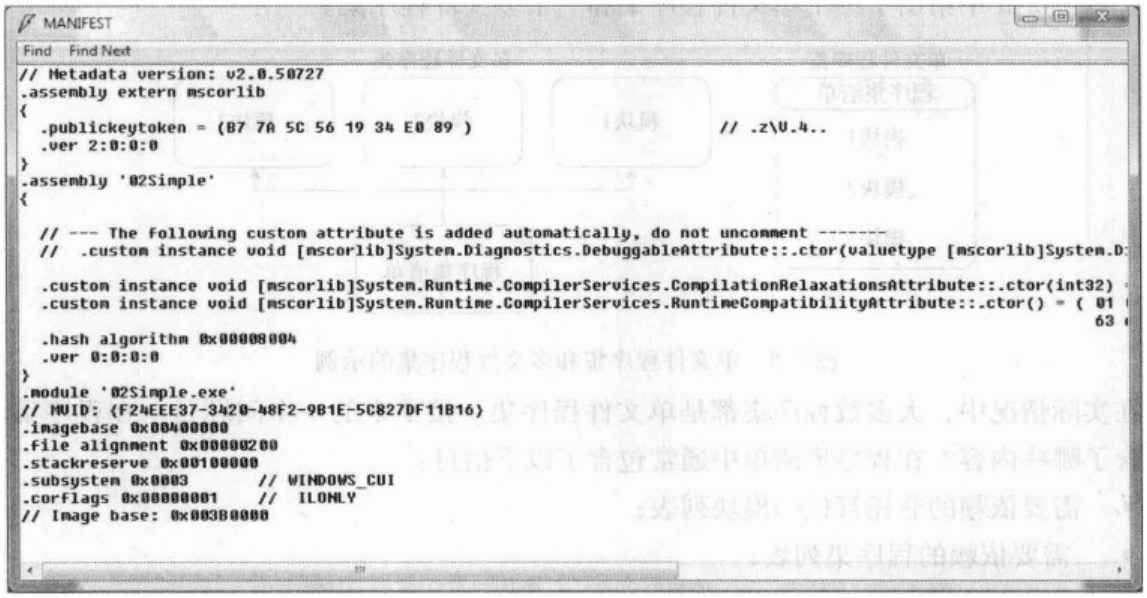


图 2-6　使用 ILDasm 来查看程序集的清单数据

2.6　类型元数据

类型是. NET 程序中的基本编程单元。在. NET 应用程序中，要么使用自定义的类型，要么使用现有的类型（例如在框架中提供的类型）。类、接口以及枚举等，这些都是可用的类型。我们可以进一步将类型分为两类：值类型（value type）和引用类型（reference type）。值类型是指保存在线程栈上的类型，包括枚举，结构以及简单类型（例如 int、bool、char）等。通常，值类型都是一些占据内存空间较小的类型。另一种类型叫做引用类型，它是在堆上分配的，并由垃圾收集器（Garbage Collector，GC）负责管理。在引用类型中也可以包含值类型，在这种情况下，值类型将同样位于堆上并且由垃圾收集器来管理。为什么要对这两

种类型进行区分？如果将所有的类型都作为引用类型，保存在堆上并且由 GC 来管理，那么岂不更简单？回答这个问题的关键在于要考虑程序的执行效率。将对象保存在堆上并且由 GC 来管理，这是一种开销很高的操作。对于一些很小并且作用范围有限的对象来说，保存在线程的栈上会更为高效。在图 2-7 中给出了值类型和引用类型的示例。

在图 2-7 中，左边是一个值类型，右边是一个引用类型。值类别表示一个局部变量 localVar，这个变量是在类的一个成员函数中声明的。在值类型中包含的是这个变量在栈上的地址。在上面的图示中，localVar 包含的是一个指针（0x0028f3c8），该指针指向栈中的一个位置，在这个位置上存储的是值类型的实例。可以使用调试器中转储命令（例如，命令 dd）来输出这个位置上的内容。图 2-7 中的引用类型包含的是一个指针（0x0150588c），该指针指向位于托管堆上的一个引用类型实例。托管堆由垃圾收集器负责控制（在第 5 章中将进一步讨论）。

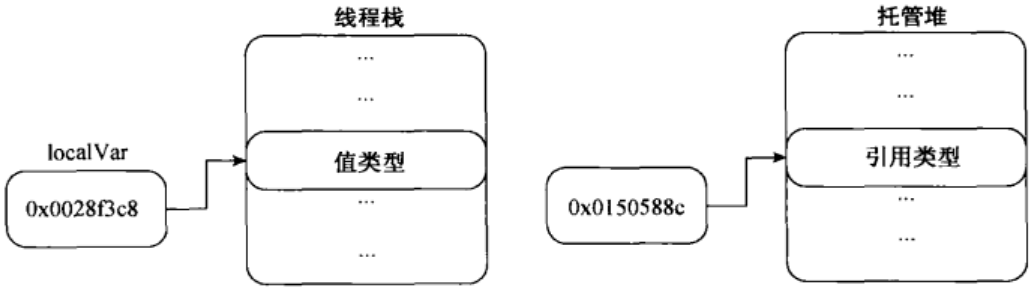


图 2-7 值类型和引用类型的示例

我们通过清单 2-5 中的示例代码来说明如何使用调试器查看值类型和引用类型。

清单 2-5 值类型和引用类型的示例

```
using System;
using System.Text;

namespace Advanced.NET.Debugging.Chapter2
{
    class TypeSample
    {
        TypeSample(int x, int y, int z)
        {
            coordinates.x = x;
            coordinates.y = y;
            coordinates.z = z;
        }

        private struct Coordinates
```

```
        {
            public int x;
            public int y;
            public int z;
        }

        private Coordinates coordinates;

        public void AddCoordinates()
        {
            int hashCode = GetHashCode();
            lock (this)
            {
                Coordinates tempCoord;
                tempCoord.x = coordinates.x + 100;
                tempCoord.y = coordinates.y + 50;
                tempCoord.z = coordinates.z + 100;

                System.Console.WriteLine("x={0}, y={1}, z={2}", tempCoord.x,
tempCoord.y, tempCoord.z);
            }
        }

        static void Main(string[] args)
        {
            TypeSample sample = new TypeSample(10,5,10);
            sample.AddCoordinates();
        }
    }
}
```

程序的源代码和程序集分别位于以下文件夹中：

- 源代码文件：C:\adnd\chapter2\TypeSample\02TypeSample. cs
- 程序集文件：C:\adndbin\01mdasample. exe

源代码本身很简单。在 Main 方法中声明了引用类型 TypeSample 的一个实例，并且调用方法 AddCoordinates。接着在函数 AddCoordinates 中声明了一个 Coordinates 类型的局部变量，然后执行数学运算并将值类型的内容转储到控制台上。此外，在 AddCoordinates 函数中还包含了一条 lock 语句，在本章的后面将通过这条语句来说明同步块的概念。

运行这个程序时，将输出以下结果：

```
C:\ADNDBin>02TypeSample.exe
x=110, y=55, z=110
```

现在，我们在调试器下运行 02TypeSample. exe，并且转储出局部变量 tempCoord 和 sample。在清单 2-6 中给出了调试器会话过程。

清单 2-6 通过调试器来转储值类型和引用类型

```
...
...
...
0:000> .symfix
No downstream store given, using c:\Program Files\Debugging Tools for Windows\sym
0:000> .reload
Reloading current modules
....
0:000> .load sosex.dll
0:000> !bpsc 02typesample.cs 34
0:000> g
ModLoad: 76800000 768bf000   C:\Windows\system32\ADVAPI32.dll
ModLoad: 77b60000 77c23000   C:\Windows\system32\RPCRT4.dll
ModLoad: 76ad0000 76b25000   C:\Windows\system32\SHLWAPI.dll
ModLoad: 762e0000 7632b000   C:\Windows\system32\GDI32.dll
ModLoad: 76e80000 76f1e000   C:\Windows\system32\USER32.dll
ModLoad: 76d50000 76dfa000   C:\Windows\system32\msvcrt.dll
ModLoad: 765b0000 765ce000   C:\Windows\system32\IMM32.DLL
ModLoad: 76730000 767f7000   C:\Windows\system32\MSCTF.dll
ModLoad: 765d0000 765d9000   C:\Windows\system32\LPK.DLL
ModLoad: 76e00000 76e7d000   C:\Windows\system32\USP10.dll
ModLoad: 75340000 754d4000   C:\Windows\WinSxS\x86_microsoft.windows.common-
Controls_6595b64144ccf1df_6.0.6000.20533_none_4634c4a0218d65c1\comctl32.dll
ModLoad: 79e70000 7a3ff000   C:\Windows\Microsoft.NET\Framework\v2.0.50727\mscorwks.dll
ModLoad: 755e0000 7567b000   C:\Windows\WinSxS\x86_microsoft.vc80.crt_1fc8b3b9a1e18e3b
_8.0.50727.762_none_10b2f55f9bffb8f8\MSVCR80.dll
ModLoad: 76f50000 77a1e000   C:\Windows\system32\shell32.dll
ModLoad: 76330000 76474000   C:\Windows\system32\ole32.dll
ModLoad: 790c0000 79bf6000   C:\Windows\assembly\NativeImages_v2.0.50727_32\mscorlib
\32e6f703c114f3a971cbe706586e3655\mscorlib.ni.dll
ModLoad: 79060000 790b6000   C:\Windows\Microsoft.NET\Framework\v2.0.50727\mscorjit.dll
(1fb0.1a64): CLR notification exception - code e0444143 (first chance)
Breakpoint 0 hit
eax=0000006e ebx=0027f1cc ecx=0133588c edx=0000000a esi=0133588c edi=0133588c
eip=00340152 esp=0027f178 ebp=0027f1b0 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
00340152 8b2d40303302    mov     ebp,dword ptr ds:[2333040h] ds:0023:02333040=013358a0
0:000> .loadby sos.dll mscorwks
0:000> !ClrStack -a
OS Thread Id: 0x1a64 (0)
ESP       EIP
0027f178 00340152 Advanced.NET.Debugging.Chapter2.TypeSample.AddCoordinates()
    PARAMETERS:
        this = 0x0133588c
    LOCALS:
        0x0027f178 = 0x0000006e
```

```
0027f198 003400b1
Advanced.NET.Debugging.Chapter2.TypeSample.Main(System.String[])
    PARAMETERS:
        args = 0x0133587c
    LOCALS:
        <CLR reg> = 0x0133588c

0027f3b8 79e7c74b [GCFrame: 0027f3b8]
0:000> dd 0x0027f178
0027f178  0000006e 00000037 0000006e 0027f1b0
0027f188  0027f1cc 0133588c 0133588c 003400b1
0027f198  0133587c 00545370 00000000 79e7c74b
0027f1a8  00000000 0027f1d8 0027f230 79e7c6cc
0027f1b8  0027f280 00000000 0027f250 00000000
0027f1c8  002ac030 0027f220 79f07fee 0027f3b8
0027f1d8  df49b45b 0027f404 0027f270 00000000
0027f1e8  0027f3b8 00545370 00000000 00000000
0:000> !dumpobj 0x0133588c
Name: Advanced.NET.Debugging.Chapter2.TypeSample
MethodTable: 002a30b0
EEClass: 002a1234
Size: 20(0x14) bytes
 (C:\ADNDBin\02TypeSample.exe)
Fields:
      MT    Field   Offset                 Type VT     Attr    Value Name
002a306c  4000001        4 ...ample+Coordinates  1 instance 01335890 coordinates
0:000>
```

在设置完调试符号后，我们第一个执行的就是加载调试器扩展 sosex. dll，这个扩展使我们可以在应用程序中通过扩展命令 bpsc 来设置断点。在扩展命令 bpsc 中可以指定想要设置断点的源代码文件的名字以及行号。当设置了断点后，通过命令 g(go)来恢复执行程序。当执行到断点处时，加载调试器扩展 sos. dll，这样就可以使用扩展命令来转储不同类型的对象（包括值类型对象和引用类型对象）。在转储对象之前，首先要找到这些对象的地址。可以使用扩展命命 Clrstack，这个命令将显示托管代码的调用栈，并且给出每个栈帧中的参数以及局部变量。从清单 2-6 中可以看到，在调用栈中包含两个栈帧：Main 和 AddCoordinates。在 AddCoordinates 帧中包含了一个局部变量，位于地址 0x0027f178 处。由于我们知道这个变量是一个值类型（类型为 Coordinates），因此可以使用命令 dd 来转储出地址 0x0027f178 处的内容。那么，如何知道某个局部变量指向的是一个值类型还是一个引用类型？在前面已经提到了，值类型参数被保存在栈上，因此值类型局部变量的地址应该落在与当前栈指针接近的地方。通过命令 r(registers)，我们可以知道当前栈指针的值。在命令 r 的输出结果中，寄存器 esp 的值就是当前栈指针的值。在清单 2-6 中，当转储这个值类型的局部变量时，可以看到以下数据：6e、37、62，这些数据分别对应 Coordinates 类型中的 x，y 和 z 坐标。接下来，我们观察 Main 栈帧，在这个栈帧中也包含了一个局部变量（除参数之外），位于地址 0x0133588c

处。由于这个局部变量是一个引用类型，因而我们无法直接使用调试器转储命令，而是应该使用另一个扩展命令 dumpobj。命令 dumpobj 的参数是引用类型的地址，它能把对象的内容转储出来。从清单 2-6 中，我们可以看到以下输出信息：

```
Name: Advanced.NET.Debugging.Chapter2.TypeSample
MethodTable: 002a30b0
EEClass: 002a1234
Size: 20(0x14) bytes
 (C:\ADNDBin\02TypeSample.exe)
Fields:
      MT    Field   Offset                 Type VT     Attr    Value Name
002a306c  4000001        4 ...ample+Coordinates  1 instance 01335890 coordinates
```

除了关于这个类型的一般性信息外（例如名字、大小等），还输出了这个类型中的各个域以及相应的偏移。在前面的输出中，我们可以看到在这个类型中包含了一个域，偏移是 4 个字节，类型为 Coordinates。而且，VT 列的值为 1，说明是一个值类型。在 value 列中给出了这个域所在的地址。要转储出引用类型对象中的各个域，可以再次使用 dumpobj 命令。补充说明一点：如果该类型是一个值类型，那么可以使用另一个扩展命令 dumpvc，如下所示：

```
0:000> !dumpvc 002a306c 01335890
Name: Advanced.NET.Debugging.Chapter2.TypeSample+Coordinates
MethodTable 002a306c
EEClass: 002a1298
Size: 20(0x14) bytes
 (C:\ADNDBin\02TypeSample.exe)
Fields:
      MT    Field   Offset                 Type VT     Attr    Value Name
79102290  4000002        0         System.Int32  1 instance       10 x
79102290  4000003        4         System.Int32  1 instance        5 y
79102290  4000004        8         System.Int32  1 instance       10 z
```

dumpvc 命令的参数可以是方法表的地址和值类型地址。

到目前为止，我们讨论了 CLR 中两种不同的基本类型，即值类型和引用类型。我们来快速浏览一下如何通过调试器来观察这些类型。请注意，我们并不需要彻底了解所有这些命令。在第 3 章基本调试任务中，我们会深入讲解各种命令的工作原理。

在前面已经提到了，.NET 的本质就在于它的自描述性。我们已经看到了程序集通过清单来描述它们自己，但到目前为止还没有讨论如何描述类型元数据。在每个程序集的 PE 文件中都存在许多流（Stream），其中包含了所有的类型元数据。表 2-1 给出了一些元数据流。

表 2-1 元数据流

流的名字	描 述	流的名字	描 述
#Blob	包含了二进制数据，例如方法表	#String	包含了 UTF-8 格式的字符串，包括类型和方法的名字
#US	包含了 UCS-2 格式的字符串	#~	包含了元数据表
#GUID	包含了在程序集中用到的所有 GUID		

在深入讨论类型元数据之前，我们首先要理解类型实例的内存布局。在图 2-8 中给出了托管堆上类型的内存布局。

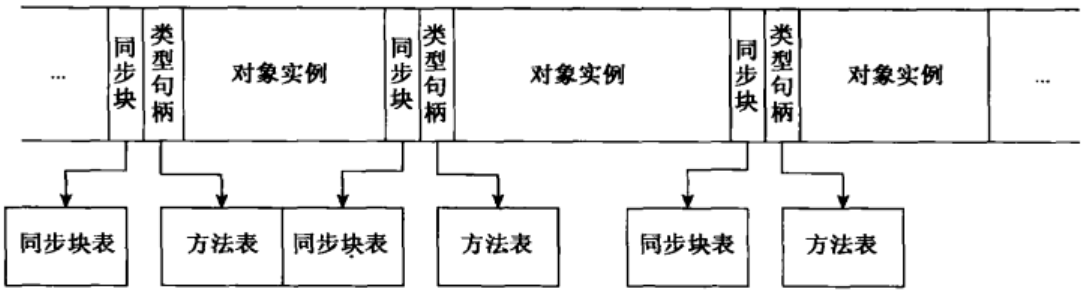


图 2-8 托管堆上对象的结构

在托管堆上的每个对象实例中都包含了以下辅助信息：

- 同步块（sync block）。同步块可以是一个位掩码，也可以是由 CLR 维持的同步块表中的索引，其中包含了关于对象本身的辅助信息。我们将在本章后面以及第 6 章中介绍同步块索引。
- 类型句柄（type handle）。类型句柄是 CLR 类型系统的基础单元。它可以用来对托管堆上的类型进行完整描述。稍后将更详细地介绍类型句柄。
- 对象实例。在同步块索引和类型句柄之后紧接着是实际的对象数据。

在接下来的章节中，我们将深入讨论类型句柄和同步块索引，并介绍 CLR 如何将类型信息与托管堆上的类型实例关联起来。在整个讨论过程中，我们都将沿用前面的示例程序 02TypeSample. exe。

2.6.1 同步块表

在托管堆上每个对象的前面都是一个同步块索引，它指向 CLR 中私有堆上的同步块表。在同步块表中包含的是指向各个同步块的指针，在同步块中包含了许多信息，例如对象的锁、互用性数据、应用程序域索引、对象的散列码（hash code）等。当然，在对象中也可能不包含任何同步块数据，此时的同步块索引值为 0。需要注意的是，在同步块中并不一定只包含简单的索引，也可以包含关于对象的其他辅助信息。在第 6 章中，我们将进一步介绍同步块中包含的各种信息。

在使用索引时需要注意，CLR 可以自由移动/增长同步块表，同时却不一定对所有包含同步块的对象头进行调整。

我们来看看同步块数据在调试器中的视图。首先在 Main 函数中设置一个断点，然后在 AddCoordinates 函数中设置另一个断点。之所以要设置两个断点，目的是为了说明一个对象在没有获取任何锁或者获取了一个锁这两种情况下的不同表现，在清单 2-7 中给出了在触发第一个断点时的调试会话。

清单 2-7 在调试器中观察对象的同步块数据

```
0:000> .symfix
No downstream store given, using c:\Program Files\Debugging Tools for Windows\sym
0:000> .reload
Reloading current modules
....
0:000> .load sosex.dll
0:000> !bpsc 02typesample.cs 34
0:000> !bpsc 02typesample.cs 41
0:000> g
ModLoad: 76680000 7673f000   C:\Windows\system32\ADVAPI32.dll
ModLoad: 763b0000 76473000   C:\Windows\system32\RPCRT4.dll
ModLoad: 76570000 765c5000   C:\Windows\system32\SHLWAPI.dll
ModLoad: 77b90000 77bdb000   C:\Windows\system32\GDI32.dll
ModLoad: 765e0000 7667e000   C:\Windows\system32\USER32.dll
ModLoad: 77870000 7791a000   C:\Windows\system32\msvcrt.dll
ModLoad: 76520000 7653e000   C:\Windows\system32\IMM32.DLL
ModLoad: 77710000 777d7000   C:\Windows\system32\MSCTF.dll
ModLoad: 765d0000 765d9000   C:\Windows\system32\LPK.DLL
ModLoad: 76330000 763ad000   C:\Windows\system32\USP10.dll
ModLoad: 75390000 75524000   C:\Windows\WinSxS\x86_microsoft.windows.common-
controls_6595b64144ccf1df 6.0.6000.20533 none_4634c4a0218d65c1\comctl32.dll
ModLoad: 79e70000 7a3ff000   C:\Windows\Microsoft.NET\Framework
\v2.0.50727\mscorwks.dll
ModLoad: 75630000 756cb000   C:\Windows\WinSxS\x86_microsoft
.vc80.crt_1fc8b3b9a1e18e3b_8.0.50727.762_none_10b2f55f9bffb8f8\MSVCR80.dll
ModLoad: 76b90000 7765e000   C:\Windows\system32\shell32.dll
ModLoad: 77920000 77a64000   C:\Windows\system32\ole32.dll
ModLoad: 790c0000 79bf6000   C:\Windows\assembly
\NativeImages_v2.0.50727_32\mscorlib\32e6f703c114f3a971cbe706586e3655\mscorlib.ni.dll
ModLoad: 79060000 790b6000   C:\Windows\Microsoft.NET\Framework
\v2.0.50727\mscorjit.dll
(e98.20fc): CLR notification exception - code e0444143 (first chance)
Breakpoint 0 hit
eax=0000000a ebx=002af46c ecx=0134588c edx=0000000a esi=0134588c edi=0134588c
eip=001900a7 esp=002af438 ebp=002af450 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
001900a7 8bcf            mov     ecx,edi
0:000> .loadby sos.dll mscorwks
0:000> !ClrStack -a
OS Thread Id: 0x20fc (0)
ESP       EIP
002af438 001900a7
Advanced.NET.Debugging.Chapter2.TypeSample.Main(System.String[])
    PARAMETERS:
        args = 0x0134587c
    LOCALS:
        <CLR reg> = 0x0134588c

002af65c 79e7c74b [GCFrame: 002af65c]
```

```
0:000> dd 0x0134588c-0x4
01345888  00000000 000d30b0 0000000a 00000005
01345898  0000000a 00000000 00000000 00000000
013458a8  00000000 00000000 00000000 00000000
013458b8  00000000 00000000 00000000 00000000
013458c8  00000000 00000000 00000000 00000000
013458d8  00000000 00000000 00000000 00000000
013458e8  00000000 00000000 00000000 00000000
013458f8  00000000 00000000 00000000 00000000
0:000>
```

当程序在断点 1 处停止执行时，使用扩展命令 ClrStack 来获得线程托管代码的调用栈。在每个栈帧中都显示了局部变量以及传递给该帧的参数。

在这里，只有一个栈帧（Main）和一个指向 TypeSample 类型实例的局部变量（0x0134588c）。这个指针指向的是对象实例的起始位置，因此要查看同步块索引，我们就必须将这个对象实例指针减去 4 个字节（DWORD）。从清单 2-7 中可以看出，在对位置 0x0134588c-0x4 进行转储时输出的内容为 0x0，这表示该对象并不包含相关的同步块索引。接下来，我们继续执行直到触发第 2 个断点，然后再次显示对象实例指针（减去 4 字节）：

```
…
…
…
0:000> g
(e98.20fc): CLR notification exception - code e0444143 (first chance)
Breakpoint 1 hit
eax=0000006e ebx=002af46c ecx=0134588c edx=00000001 esi=0134588c edi=0134588c
eip=00190177 esp=002af3fc ebp=002af430 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
00190177 8b1d40303402    mov     ebx,dword ptr ds:[2343040h]
ds:0023:02343040=013458a0
0:000> !ClrStack -a
OS Thread Id: 0x20fc (0)
ESP       EIP
002af3fc 00190177 Advanced.NET.Debugging.Chapter2.TypeSample.AddCoordinates()
    PARAMETERS:
        this = 0x0134588c
    LOCALS:
        0x002af404 = 0x0000006e
        0x002af3fc = 0x0134588c

002af438 001900b1 Advanced.NET.Debugging.Chapter2.TypeSample.Main(System.String[])
    PARAMETERS:
        args = 0x0134587c
    LOCALS:
        <CLR reg> = 0x0134588c

002af65c 79e7c74b [GCFrame: 002af65c]
0:000> dd 0x0134588c-0x4
01345888  00000001 000d30b0 0000000a 00000005
```

```
01345898  0000000a 80000000 790fd8c4 00000014
013458a8  00000013 003d0078 0030007b 002c007d
013458b8  00790020 007b003d 007d0031 0020002c
013458c8  003d007a 0032007b 0000007d 00000000
013458d8  00000000 00000000 00000000 00000000
013458e8  00000000 00000000 00000000 00000000
013458f8  00000000 00000000 00000000 00000000
```

注意，我们再次使用了命令 ClrStack 来获得对象实例指针。这么做是很关键的，因为在垃圾收集器的空闲期间，托管堆上的对象可能被移动到其他位置。在输出结果中，对象实例指针仍然是 0x0134588c，这表示对象没有被移动。接下来，我们再次转储出对象实例指针的内容（减去 4 个字节以得到同步块索引），这次我们看到在对象中包含了一个同步块索引 0x1。如果查看源代码，可以看到函数 AddCoordinates 在对象上获取了一个锁，因此 CLR 为这个对象创建了一个同步块索引。接下来的问题是，如何进一步地观察同步块表（即观察表中索引 1 处的内容）？由于同步块表位于 CLR 的私有内存中，因此无法直接访问同步块表。然而，在 SOS 中包含了一个扩展命令 syncblk，这个命令可以给出同步块表的一些信息。syncblk 命令的参数既可以是同步块的索引，也可以不带任何参数，在不带参数时将转储出同步块表中的所有元素。下面的清单给出了在示例程序（索引 1）上运行扩展命令 syncblk 的输出结果：

```
0:000> !syncblk 1
Index SyncBlock MonitorHeld Recursion Owning Thread Info  SyncBlock Owner
    1 0011fc74            1         1 001051b8  2288   0   0131588c
Advanced.NET.Debugging.Chapter2.TypeSample
-----------------------------
Total           1
CCW             0
RCW             0
ComClassFactory 0
Free            0
0:000>
```

从上面可以看出，索引为 1 的同步块指向的是一个已锁定的监视器（monitor），由线程 0x001051b8 持有。在第 6 章中，我们将进一步讨论与同步相关的问题以及如何通过同步块来跟踪这些问题。同步块命令主要输出与同步块的锁定方面相关的信息，而并不显示其他的信息（例如对象的散列码）。

你可能已经注意到，在示例程序的源代码中包含了一些陌生的语句。具体来说，在 AddCoordinates 函数中调用了 GetHashCode，此外还有一条 lock 语句。之所以要调用 GetHashCode，是为了强制创建一个同步块入口。当调用 lock 语句时，它将判断是否存在一个同步块与对象相关，如果存在的话，会把这个同步块作为同步数据。如果不存在同步块，那么 CLR 将初始化一个瘦锁（thin lock），而瘦锁保存的位置与同步块的位置是不同的。

到这里就结束了对同步块表的讨论。在第 6 章中，我们将进一步看到如何利用同步块信

息来分析同步问题。

在图 2-8 中接下来需要注意的域就是类型句柄域，下面将对其进行讨论。

2.6.2 类型句柄

引用类型的所有实例都被放在托管堆上，这个堆是由 GC 来控制。在所有的实例中都包含了一个类型句柄。简单地说，类型句柄指向的是某个类型的方法表。在方法表中包含了各种元数据，它们完整地描述了这个类型。在图 2-9 中说明了方法表的整体内存布局。

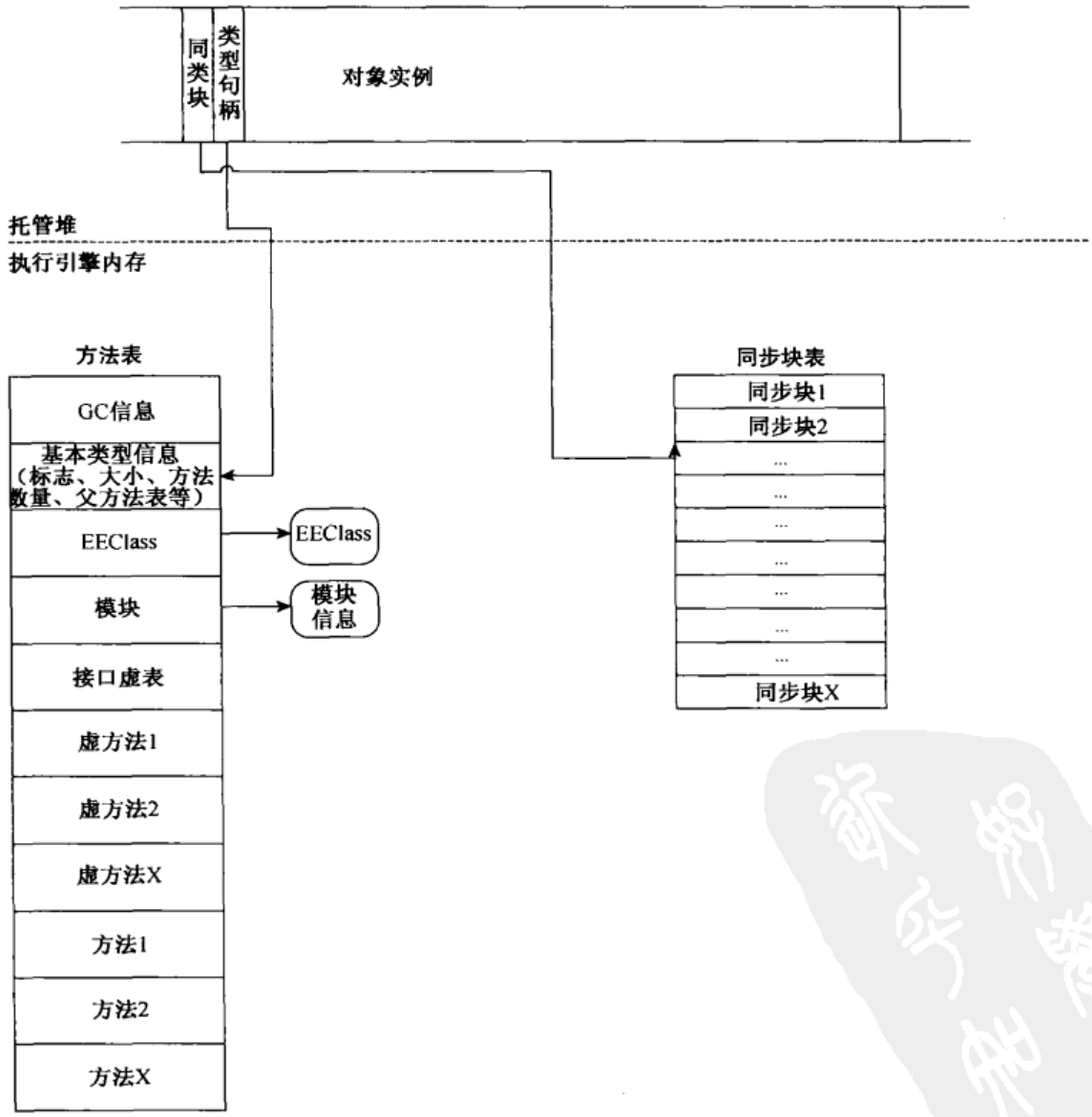


图 2-9 托管堆对象以及方法表

类型句柄是 CLR 类型系统中的粘合剂，它把对象实例及其所有的相关类型数据关联起来。对象实例的类型句柄存储在托管堆上，它是一个指针，指向类型的方法表。在方法表中包含了关于对象类型的大量信息，包括指向其他关键 CLR 数据结构（例如 EEClass）的指针。在类型句柄指向的第一类数据中包含了关于类型本身的一些信息。在表 2-2 中给出了这类数据中的各个域。

表 2-2 在方法表中杂项（Miscellaneous）部分的域

类型句柄偏移	成 员 名	描 述
+0	Flags	提供关于类型本身信息的位掩码，例如，0x00040000 表示该类型是一个类而不是数组
+4	Base Size	当类型实例被分配在托管堆上时的大小，单位为字节
+8	Flags2	额外的类型信息
+10	NumMethods	表示类型包含的方法总数
+12	NumVirtMethods	表示类型包含的虚方法总数
+14	NumInterfaces	表示类型实现的接口总数
+16	Parent	指向父方法表的指针

我们来看看调试器环境中的方法表。再次使用示例程序 02TypeSample. exe。在调试器下启动这个应用程序后，在第 41 行设置一个断点，然后继续执行直到触发断点。在清单 2-8 中给出了调试会话。

清单 2-8 在调试器中转储出方法表

```
0:000> .load sosex.dll
0:000> lbpsc 02typesample.cs 41
0:000> g
ModLoad: 76680000 7673f000   C:\Windows\system32\ADVAPI32.dll
ModLoad: 763b0000 76473000   C:\Windows\system32\RPCRT4.dll
ModLoad: 76570000 765c5000   C:\Windows\system32\SHLWAPI.dll
ModLoad: 77b90000 77bdb000   C:\Windows\system32\GDI32.dll
ModLoad: 765e0000 7667e000   C:\Windows\system32\USER32.dll
ModLoad: 77870000 7791a000   C:\Windows\system32\msvcrt.dll
ModLoad: 76520000 7653e000   C:\Windows\system32\IMM32.DLL
ModLoad: 77710000 777d7000   C:\Windows\system32\MSCTF.dll
ModLoad: 765d0000 765d9000   C:\Windows\system32\LPK.DLL
ModLoad: 76330000 763ad000   C:\Windows\system32\USP10.dll
ModLoad: 75390000 75524000   C:\Windows\WinSxS\x86_microsoft.windows.common-
controls_6595b64144ccf1df_6.0.6000.20533_none_4634c4a021
ModLoad: 79e70000 7a3ff000   C:\Windows\Microsoft.NET\Framework
\v2.0.50727\mscorwks.dll
ModLoad: 75630000 756cb000   C:\Windows\WinSxS\x86_microsoft.vc80
.crt_1fc8b3b9a1e18e3b_8.0.50727.762_none_10b2f55f9bffb8f8\MSVCR80.d
ModLoad: 76b90000 7765e000   C:\Windows\system32\shell32.dll
ModLoad: 77920000 77a64000   C:\Windows\system32\ole32.dll
ModLoad: 790c0000 79bf6000   C:\Windows\assembly
\NativeImages_v2.0.50727_32\mscorlib\32e6f703c114f3a971cbe706586e3655\
```

```
mscorlib.ni.dl
ModLoad: 79060000 790b6000   C:\Windows\Microsoft.NET\Framework
\v2.0.50727\mscorjit.dll
(2110.2740): CLR notification exception - code e0444143 (first chance)
Breakpoint 0 hit
eax=0000000a ebx=0025f42c ecx=0163588c edx=0000000a esi=0163588c edi=0163588c
eip=003400a7 esp=0025f3f8 ebp=0025f410 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
003400a7 8bcf            mov     ecx,edi
0:000> .loadby sos.dll mscorwks
0:000> !ClrStack -a
OS Thread Id: 0x2740 (0)
ESP       EIP
0025f3f8 003400a7
Advanced.NET.Debugging.Chapter2.TypeSample.Main(System.String[])
    PARAMETERS:
        args = 0x0163587c
    LOCALS:
        <CLR reg> = 0x0163588c

0025f618 79e7c74b [GCFrame: 0025f618]
0:000> dd 0x0163588c
0163588c  002930b0 0000000a 00000005 0000000a
0163589c  00000000 00000000 00000000 00000000
016358ac  00000000 00000000 00000000 00000000
016358bc  00000000 00000000 00000000 00000000
016358cc  00000000 00000000 00000000 00000000
016358dc  00000000 00000000 00000000 00000000
016358ec  00000000 00000000 00000000 00000000
016358fc  00000000 00000000 00000000 00000000
0:000> dd 002930b0
002930b0  00040000 00000014 00070402 00000004
002930c0  790fd0f0 00292c3c 002930f8 00291244
002930d0  00000000 00000000 79371278 7936b3b0
002930e0  7936b3d0 793624d0 003400c8 0029c015
002930f0  00340070 00000000 00000080 00000000
00293100  00000000 00000000 00000000 00000000
00293110  00000000 00000000 00000000 00000000
00293120  00000000 00000000 00000000 00000000
0:000>
```

在清单 2-8 中，我们使用了扩展命令 ClrStack 来找出栈帧 Main 上的局部变量。这个局部变量是一个指针，指向托管堆上的一个 TypeSample 类型的实例。接下来，我们转储出这个实例并且注意到第一个域的值为 0x002930b0，这个值对应于类型句柄。要了解关于类型句柄（指向方法表）的更多信息，我们可以转储出类型句柄的内容。第一个域（0x00040000）对应于标志域，表示该类型是一个类（而不是数组）。接下来的域表示实例的大小，在这里是 0x14（转换为十进制则是 20）个字节。从清单 2-5 的源代码中可以看到，在 TypeSample 类

中包含了一个结构，这个结构的大小为 4 个整数，即 16 个字节（4 * sizeof（DWORD））。剩下的 4 个字节是类型句柄的值。下一个需要注意的域是 Flags2（WORD），这个域的值为 0x0402。该位掩码告诉 CLR，这个类型不需要特殊的初始化逻辑，并且这个类也不包含任何安全属性。在接下来的 WORD 大小的域中包含的值是 0x0007，表示这个类共有 7 个方法。如果查看清单 2-5 中的源代码，我们可以发现这个类只有两个显式方法即 Main 和 AddCoordinates。那么其他五个方法是什么呢？首先要记得，即使我们没有显式地定义任何构造函数，C#编译器仍会自动生成一个默认的构造函数。其次要记得，所有的类都是从 Object 继承下来的。在 Object 类中定义了四个方法，所有的子类都将继承这四个方法，即 ToString、Equals、GetHashCode 和 Finalize。这四个继承而来的方法，再加上隐式的构造函数以及我们自己定义的两个，总共七个方法。下一个域（大小为 WORD）表示类中虚方法的数量。在本示例中，这个域的值为 0x0004，表示在类中定义了四个虚方法。这是因为在父类 Object 中定义了四个虚方法（在前面已经讨论过）。接下来的域表示类实现的接口数量，这里是 0x0000，表示没有实现任何接口。最后一个域是父类方法表，其中包含了一个指针，指向父对象的方法表。这里的指针值为 0x790fd0f0。你可以使用与前面相同的方法将父类方法表的内容转储出来并进行分析。

方法表的下一部分是一个指针，指向与类型相关的模块信息。从清单 2-8 中的转储信息中，我们可以看到这个值为 0x00292c3c。要获得模块的信息，可以使用扩展命令 DumpModule，并将这个指针值作为命令的参数：

```
0:000> !DumpModule 00292c3c
Name: C:\ADNDBin\02TypeSample.exe
Attributes: PEFile
Assembly: 0044d8b0
LoaderHeap: 00000000
TypeDefToMethodTableMap: 002900c0
TypeRefToMethodTableMap: 002900cc
MethodDefToDescMap: 002900fc
FieldDefToDescMap: 0029010c
MemberRefToDescMap: 00290120
FileReferencesMap: 00290148
AssemblyReferencesMap: 0029014c
MetaData start address: 00c8214c (1168 bytes)
```

在本章的后面，我们将进一步分析 Module 信息。

下一个需要注意的域是一个指针，指向一个 EEClass。我们将在本章的后面讨论 EEClass 数据结构。

接下来两个 DWORD 类型的域在调试.NET 应用程序时并不会用到，因此可以暂时忽略。前面这些域就构成了方法表的基本内容。在剩余的域中包含了类型的虚方法表。每个 DWORD 类型的域中都包含了一个指向方法自身的指针。在这里，我们通过方法域的数量知道共有 7 个方法。以下高亮字体显示的是虚方法表中的指针值：

```
002930d0  00000000 00000000 79371278 7936b3b0
002930e0  7936b3d0 793624d0 003400c8 0029c015
002930f0  00340070 00000000 00000080 00000000
```

可以使用扩展命令 U（Unassemble）对上述每个方法指针进行反汇编。例如，对地址 0x003400c8 上的方法指针反汇编，将得到以下结果：

```
0:000> !u 003400c8
Normal JIT generated code
Advanced.NET.Debugging.Chapter2.TypeSample..ctor(Int32, Int32, Int32)
Begin 003400c8, size 35
>>> 003400c8 57              push    edi
003400c9 56              push    esi
003400ca 8bf1            mov     esi,ecx
003400cc 8bfa            mov     edi,edx
003400ce 833d082e290000  cmp     dword ptr ds:[292E08h],0
003400d5 7405            je      003400dc
003400d7 e86b82de79      call    mscorwks!JIT_DbgIsJustMyCode
(7a128347)
003400dc 8bce            mov     ecx,esi
003400de e8d5230279      call    mscorlib_ni+0x2a24b8 (793624b8)
 (System.Object..ctor(), mdToken: 06000001)
003400e3 90              nop
003400e4 90              nop
003400e5 897e04          mov     dword ptr [esi+4],edi
003400e8 8b442410        mov     eax,dword ptr [esp+10h]
003400ec 894608          mov     dword ptr [esi+8],eax
003400ef 8b44240c        mov     eax,dword ptr [esp+0Ch]
003400f3 89460c          mov     dword ptr [esi+0Ch],eax
003400f6 90              nop
003400f7 90              nop
003400f8 5e              pop     esi
003400f9 5f              pop     edi
003400fa c20800          ret     8
0:000>
```

命令 U 不仅能对代码进行反汇编，还能提供一些额外的标注信息。例如，我们可以看到上述反汇编的代码正是 TypeSample 类的构造函数。请注意，在方法表中的一些方法指针可能会指向非托管的代码。例如，方法指针 0x0029c015 指向一个非托管代码段：

```
0:000> !u 0029c015
Unmanaged code
0029c015 b001            mov     al,1
0029c017 eb04            jmp     0029c01d
0029c019 b002            mov     al,2
0029c01b eb00            jmp     0029c01d
0029c01d 0fb6c0          movzx   eax,al
0029c020 c1e003          shl     eax,3
0029c023 0530302900      add     eax,293030h
0029c028 e917487600      jmp     00a00844
```

```
0029c02d 0000          add     byte ptr [eax],al
0029c02f 00e8          add     al,ch
```

出现这种情况的原因是，一些方法可能还没有被 JIT 编译器编译。事实上，你所看到的是一段启动编译过程的 JIT 存根代码，在这之后会把执行控制权转移到新编译生成的代码。

到这里就结束了对类型句柄和方法表的讨论。虽然我们之前是通过手动方式来观察方法表的内容，但实际上存在一个扩展命令 DumpMT，这个命令可以以一种简洁易读的方式转储出方法表中的信息。下面是执行扩展命令 DumpMT 的示例，该命令位于 0x002930b0 处的方法表。

```
0:000> !dumpmt 002930b0
EEClass: 00291244
Module: 00292c3c
Name: Advanced.NET.Debugging.Chapter2.TypeSample
mdToken: 02000002  (C:\ADNDBin\02TypeSample.exe)
BaseSize: 0x14
ComponentSize: 0x0
Number of IFaces in IFaceMap: 0
Slots in VTable: 7
```

尽管这个扩展命令使我们更容易观察方法表的内容，但知道如何通过手动方式来遍历方法表同样是非常重要的。

接下来，我们将介绍方法描述符。

2.6.3 方法描述符

在上一节中，我们介绍了方法表的概念以及它是如何描述某个类型的。在方法表中包含了虚方法表，里面包含了一些指向隐藏在类型方法背后的代码的指针。既然虚方法表中包含了指向代码的指针，那么这些方法本身如何自行描述？这个问题的答案就在于方法描述符。在方法描述符中包含了关于方法的详细信息，例如方法的文本表示、它所在的模块、标记，以及实现方法的代码地址。

要找到指定方法的方法描述符，可以使用扩展命令 dumpmt，同时使用开关-md。例如，如果在调试器下运行 02typesample. exe，并且将局部变量 sample 的方法表转储出来，那么可以看到以下内容：

```
0:000> !dumpmt -md 000e30b0
EEClass: 000e1244
Module: 000e2c3c
Name: Advanced.NET.Debugging.Chapter2.TypeSample
mdToken: 02000002  (C:\ADNDBin\02TypeSample.exe)
BaseSize: 0x14
ComponentSize: 0x0
Number of IFaces in IFaceMap: 0
Slots in VTable: 7
```

```
-----------------------------------------------
MethodDesc Table
   Entry MethodDesc      JIT Name
79371278   7914b928   PreJIT System.Object.ToString()
7936b3b0   7914b930   PreJIT System.Object.Equals(System.Object)
7936b3d0   7914b948   PreJIT System.Object.GetHashCode()
793624d0   7914b950   PreJIT System.Object.Finalize()
009200c8   000e3030      JIT Advanced.NET.Debugging.Chapter2.TypeSample..ctor(Int32,
Int32, Int32)

000ec015   000e3038     NONE
Advanced.NET.Debugging.Chapter2.TypeSample.AddCoordinates()
00920070   000e3040      JIT
Advanced.NET.Debugging.Chapter2.TypeSample.Main(System.String[])
0:000>
```

在输出信息的“MethodDesc Table”部分中列出了 TypeSample 类型的所有方法描述符。在 Entry 列中包含的是该方法的代码被加载到内存中的位置，而在 MethodDesc 列中给出了方法描述符的地址。在 JIT 列给出的是代码地址的状态，可以是以下值：

- PreJIT 表示位于 Entry 地址处的代码被 JIT 预编译了。
- JIT 表示这段代码已经被 JIT 编译过了。
- NONE 表示这段代码还没有被 JIT 编译过。

最后，在 Name 列中给出了方法名的文本表示。要进一步获得指定方法的信息，我们可以将 MethodDesc 列中的地址传递给扩展命令 dumpmd ，如下所示：

```
0:000> !dumpmd 000e3040
Method Name: Advanced.NET.Debugging.Chapter2.TypeSample.Main(System.String[])
Class: 000e1244
MethodTable: 000e30b0
mdToken: 06000003
Module: 000e2c3c
IsJitted: yes
m_CodeOrIL: 00920070
```

在输出信息中，最需要注意的部分是 IsJitted 和 m_CodeOrIL 这两个域，它们同样能告诉我们方法的状态（就 JIT 编译过程而言）。如果 IsJitted 设置为“yes”，那么就意味着这个方法已经被 JIT 编译了，并且在地址 m_CodeOrIL 中包含了实际代码。如果 IsJitted 设置为“no”，那么 m_CodeOrIL 域就被设置为 0xffffffff，如下所示：

```
0:000> !dumpmd 000e3038
Method Name: Advanced.NET.Debugging.Chapter2.TypeSample.AddCoordinates()
Class: 000e1244
MethodTable: 000e30b0
mdToken: 06000002
Module: 000e2c3c
IsJitted: no
m_CodeOrIL: ffffffff
```

2.6.4 模块

我们在前面已经解释过，程序集可以看成是一个逻辑容器，其中包含一个或者多个代码模块。模块则可以看成是包含了具体组件的实际代码或资源。当我们观察各种 CLR 数据结构时（例如方法表、方法描述符等），会发现在这些数据结构中通常包含一个指针指向定义它们的模块。例如，将 02TypeSample.exe 中函数 AddCoordinates 的方法描述符转储出来：

```
0:000> !dumpmd 000e3038
Method Name:
Advanced.NET.Debugging.Chapter2.TypeSample.AddCoordinates()
Class: 000e1244
MethodTable: 000e30b0
mdToken: 06000002
Module: 000e2c3c
IsJitted: no
m_CodeOrIL: ffffffff
```

这个类型所在模块的地址为 0x000e2c3c。要获得任意地址上模块的扩展信息，可以使用扩展命令 dumpmodule，如下所示：

```
0:000> !dumpmodule 000e2c3c
Name: C:\ADNDBin\02TypeSample.exe
Attributes: PEFile
Assembly: 002cd8b8
LoaderHeap: 00000000
TypeDefToMethodTableMap: 000e00c0
TypeRefToMethodTableMap: 000e00cc
MethodDefToDescMap: 000e00fc
FieldDefToDescMap: 000e010c
MemberRefToDescMap: 000e0120
FileReferencesMap: 000e0148
AssemblyReferencesMap: 000e014c
MetaData start address: 000c214c (1168 bytes)
```

除了名字、属性、模块所处的程序集地址以及加载器堆等域之外，还定义了一组映射（Map）。这些映射的作用只是将标记映射到底层的 CLR 数据结构。例如，如果要将一个方法定义标记映射到一个方法描述符，可以首先将 MethodDefToDescMap 地址处的数据转储出来。

```
0:000> dd 000e00fc
000e00fc  00000000 000e3030 000e3038 000e3040
000e010c  00000000 000e3014 000e3048 000e3054
000e011c  000e3060 00000000 00000000 00000000
000e012c  00000000 7914b920 00000000 00000000
000e013c  00000000 00000000 00000000 000e2c3c
000e014c  00000000 790c2000 000e0174 00000400
000e015c  00000002 00000000 00000000 00000000
000e016c  00000000 000e2c3c 00000000 00000000
```

在这里，我们可以看到位于位置 1，2 和 3 处的方法定义标记包含了指向方法描述符的地

址。我们可以使用扩展命令并将上述各个地址作为参数传递给命令，从而获得方法描述符的进一步信息：

```
0:000> !dumpmd 000e3030
Method Name: Advanced.NET.Debugging.Chapter2.TypeSample..ctor(Int32, Int32, Int32)
Class: 000e1244
MethodTable: 000e30b0
mdToken: 06000001
Module: 000e2c3c
IsJitted: yes
m_CodeOrIL: 009200c8
0:000> !dumpmd 000e3038
Method Name: Advanced.NET.Debugging.Chapter2.TypeSample.AddCoordinates()
Class: 000e1244
MethodTable: 000e30b0
mdToken: 06000002
Module: 000e2c3c
IsJitted: no
m_CodeOrIL: ffffffff
0:000> !dumpmd 000e3040
Method Name: Advanced.NET.Debugging.Chapter2.TypeSample.Main(System.String[])
Class: 000e1244
MethodTable: 000e30b0
mdToken: 06000003
Module: 000e2c3c
IsJitted: yes
m_CodeOrIL: 00920070
```

扩展命令 dumpmodule 不仅能够转储出模块特定的信息，还可以输出在模块中定义和使用的所有类型。要获得这些信息，可以使用-mt 命令开关，如下所示：

```
0:000> !dumpmodule -mt 000e2c3c
Name: C:\ADNDBin\02TypeSample.exe
Attributes: PEFile
Assembly: 002cd8b8
LoaderHeap: 00000000
TypeDefToMethodTableMap: 000e00c0
TypeRefToMethodTableMap: 000e00cc
MethodDefToDescMap: 000e00fc
FieldDefToDescMap: 000e010c
MemberRefToDescMap: 000e0120
FileReferencesMap: 000e0148
AssemblyReferencesMap: 000e014c
MetaData start address: 000c214c (1168 bytes)

Types defined in this module

      MT    TypeDef Name
-----------------------------------------------
```

```
000e30b0 0x02000002 Advanced.NET.Debugging.Chapter2.TypeSample
000e306c 0x02000003 Advanced.NET.Debugging.Chapter2.TypeSample+Coordinates

Types referenced in this module

      MT    TypeRef Name
-----------------------------------------------
790fd0f0 0x01000001 System.Object
790fd260 0x01000002 System.ValueType
```

2.6.5 元数据标记

到目前为止，我们已经看到了各种不同的运行时结构，例如程序集、模块、方法描述符、方法表等。所有这些结构的目的都是为了支持.NET 程序的类型系统和自描述属性。CLR 的这些元数据以表格的形式存储在运行时引擎中。总共有许多种不同的元数据表，而我们在这里也不可能全部讨论，重要的是要知道 CLR 如何使用这些元数据以及如何通过元数据标记来引用它们。简单来说，元数据标记是一个 4 字节的值，如图 2-10 所示。

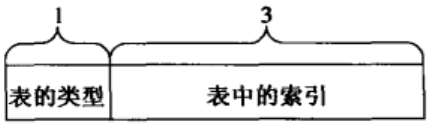


图 2-10 元数据标记的内存布局

高位的 1 个字节表示该标记所引用的表。在表 2-3 中给出了不同的表。

表 2-3 元数据表

类 型	描 述	类 型	描 述	类 型	描 述
0x00000000	模块	0x0c000000	自定义属性	0x23000000	程序集引用
0x01000000	类型引用	0x0e000000	许可	0x26000000	文件
0x02000000	类型定义	0x11000000	签名	0x27000000	导出类型
0x04000000	域定义	0x14000000	事件	0x28000000	清单资源
0x06000000	方法定义	0x17000000	属性	0x2a000000	泛型参数
0x08000000	参数定义	0x1a000000	模块引用	0x2b000000	方法规范
0x09000000	接口定义	0x1b000000	类型规范	0x2c000000	泛型参数约束
0x0a000000	成员引用	0x20000000	程序集		

例如，值为 06000001 的元数据标记可以解释为指向方法定义表（高位字节为 0x06）中的第 1 个索引。有没有某种方法能够观察其中一些表的内容？有的，在前面已经介绍过，DumpModule 命令的输出中包含了一组常见表映射。下面是 DumpModule 命令的输出示例。

```
0:000> !DumpModule 00292c3c
Name: C:\ADNDBin\03ObjTypes.exe
Attributes: PEFile
Assembly: 0033d2e8
LoaderHeap: 00000000
TypeDefToMethodTableMap: 002900c0
TypeRefToMethodTableMap: 002900d8
MethodDefToDescMap: 00290124
```

```
FieldDefToDescMap: 00290150
MemberRefToDescMap: 00290174
FileReferencesMap: 002901b4
AssemblyReferencesMap: 002901b8
MetaData start address: 013d2330 (2424 bytes)
```

让我们来看第一个映射。TypeDeftoMethodTableMap 位于地址 0x002900c0 处，它将类型定义映射到相应的方法表。如果通过命令 dd 将这个映射转储出来，那么将看到以下输出：

```
0:000> dd 002900c0
002900c0  00000000 00000000 0029313c 002930e8
002900d0  002931c0 002932e8 00000000 790fd0f0
002900e0  790fd260 7911a508 00000000 00000000
002900f0  00000000 00000000 79101118 00000000
00290100  79102290 00000000 00000000 790fea5c
00290110  00000000 00000000 790fd8c4 00000000
00290120  00000000 00000000 00293078 00293080
00290130  00293088 00293090 00293098 002930a0
```

从上面可以看到，在索引 2（0x0029313c）处存在一个方法表指针。我们可以通过命令 DumpMT 来验证所看到的信息：

```
0:000> !DumpMT 0029313c
EEClass: 002912e0
Module: 00292c3c
Name: Advanced.NET.Debugging.Chapter3.ObjTypes
mdToken: 02000002  (C:\ADNDBin\03ObjTypes.exe)
BaseSize: 0x1c
ComponentSize: 0x0
Number of IFaces in IFaceMap: 0
Slots in VTable: 10
```

位于索引 2 处的方法表对应于 Advanced. NET. Debugging. Chapter3. ObjTypes 类型的方法表，并且元数据标记为 02000002。标记的低位字节是正确的（索引 2），并且根据表 2-3 的内容，我们可以确认高位（02）表示一个类型。

2.6.6 EEClass

我们已经通过手动方式或者调试器和扩展命令观察了许多不同的 CLR 数据结构。在一些调试会话中提到了 EEClass 的概念，接下来将对其进行讨论。我们可以将 EEClass 数据结构看成是方法表的一个逻辑等价物，因此它可以作为实现 CLR 类型系统自描述性的一种机制。从本质来看，EEClass 和方法表是两种截然不同的结构，但从逻辑来看，它们都表示相同的概念，这样就带来了一个问题，为什么需要引入这种区分？这种区分是基于 CLR 使用类型域的频繁程度。频繁使用的域被保存到方法表中，而不太频繁使用的域被保存到 EEClass 数据结构中。

在图 2-11 中给出了 EEClass 数据结构中的大部分关键成员。

需要注意的是，虽然图 2-11 给出了 EEClass 结构中的一些域，但其中的偏移并不准确。

面向对象语言（例如 C#）中的层次结构属性在 EEClass 结构中同样适用。当 CLR 加载类型时，会创建一个类型的 EEClass 节点层次结构，其中包含了指向父节点和兄弟节点的指针，这样便可以通过一种高效的方式来遍历这个层次结构。EEClass 中的大多数域都很简单。其中一个重要的域就是方法描述块（methodDesc chunk）域，它包含了一个指针，指向类型中的第一组方法描述符，这使得能够遍历任意类型中的方法描述符。在每组方法描述符中又包含指向链表中下一组方法描述符的指针。要查看 EEClass 实例中的内容，一种最简单（也最安全）的方法就是使用扩展命令 dumpclass，如下所示。

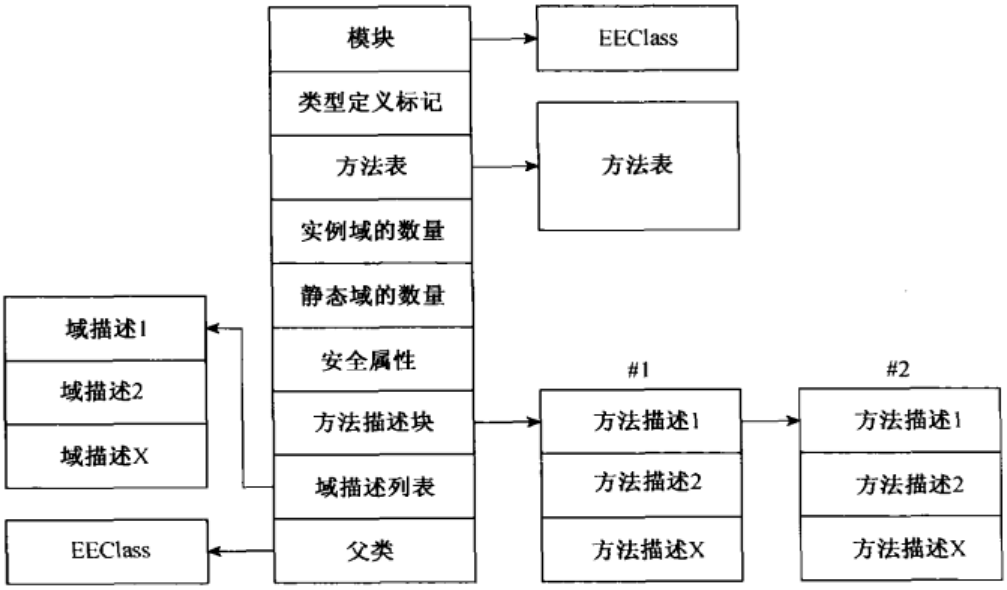


图 2-11　EEClass 的结构

```
0:000> !dumpmt 000e30b0
EEClass: 000e1244
Module: 000e2c3c
Name: Advanced.NET.Debugging.Chapter2.TypeSample
mdToken: 02000002  (C:\ADNDBin\02TypeSample.exe)
BaseSize: 0x14
ComponentSize: 0x0
Number of IFaces in IFaceMap: 0
Slots in VTable: 7
0:000> !dumpclass 000e1244
Class Name: Advanced.NET.Debugging.Chapter2.TypeSample
mdToken: 02000002 (C:\ADNDBin\02TypeSample.exe)
Parent Class: 790fd08c
Module: 000e2c3c
Method Table: 000e30b0
```

```
Vtable Slots: 4
Total Method Slots: 7
Class Attributes: 100000
NumInstanceFields: 1
NumStaticFields: 0
      MT    Field   Offset                 Type VT     Attr    Value Name
000e306c  4000001        4 ...ample+Coordinates  1 instance           coordinates
```

获得某个类型的 EEClass 指针的方法之一就是使用它的方法表。我们可以使用扩展命令 dumpmt 来转储出方法表，提取 EEClass 指针，然后将它作为参数传递给扩展命令 dumpclass。在扩展命令 dumpclass 的输出中将包含父类指针、模块、方法表、域的个数，以及每个类型域的详细信息。

2.7 小结

本章我们介绍了.NET 应用程序的基本概念，并通过调试器和相关的扩展命令进一步阐述了 CLR 中的一些基础构件。虽然我们没有详细地解释每一个调试器命令，但不必担心，在第 3 章中将详细介绍所有的命令。此外，还有其他一些关键的 CLR 组件没有讨论，例如垃圾收集器和程序集加载器。在本书的第二部分将介绍这些 CLR 组件以及在使用时经常会遇到的一些问题。

第 3 章　基本调试任务

在第 2 章中，我们通过调试器和相关的扩展命令介绍了一些关键的 CLR 内部机制。在本章我们将对调试器进行深入分析，并观察一些基本的调试任务，这些任务在本书的剩余部分中非常重要。我们将首先介绍一些基础知识，包括如何在调试器下启动进程、设置符号文件和源文件的路径、控制调试器的执行，以及如何对对象、线程和代码进行操作等。除了对这些命令给出详细描述外，我们还会通过不同的. NET 应用程序来说明这些命令的实际用法。需要注意的是，本章并没有对调试器的全部功能进行详尽描述，而只是为本书后面的内容提供必要的基础知识。

3.1　调试器以及调试目标

在任何一种调试中都包括两个主要的组件：调试器本身和调试目标（debugger target）。调试器表示一个引擎，而我们正是通过这个引擎与被调试的进程进行交互。所有与调试目标之间的交互操作（例如设置断点、观察状态等），都可以通过调试器命令来表示，而调试器将在调试目标的环境中执行这些命令。例如，如果要调试 02simple. exe，那么可以在调试器下启动这个程序，并且通过调试器进程来控制 02simple. exe 进程的执行，如图 3-1 所示。

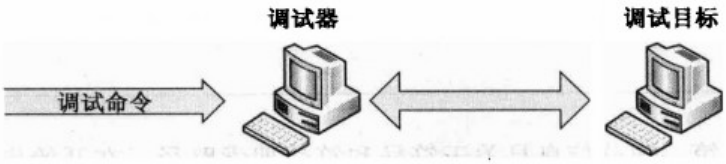


图 3-1　调试会话中的调试器与调试目标

从图 3-1 中可以看出，调试器和调试目标分别位于不同的机器上，这是一种远程调试方式，当我们无法从物理上访问调试目标时，这种方式将带来极大的便利。

为了更好地理解调试过程，我们来看一个示例。假定要通过调试器 ntsd. exe（包含在 Windows 调试工具集中）来调试应用程序 02simple. exe 。要启动调试器，只需在命令提示符中输入以下命令：

```
C:\> ntsd
```

如果没有指定任何参数，那么上述命令将只显示一组可用的选项。要指定某个需要调试的应用程序，只需将它作为命令行参数，如下所示：

```
C:\> ntsd c:\ADNDBin\02simple.exe
```

接下来，调试器本身在一个控制台窗口中启动，它会显示一系列的信息给用户。清单3-1给出了输出的信息。

清单 3-1 在调试 02simple. exe 时启动调试器

```
Microsoft (R) Windows Debugger Version 6.9.0003.113 X86
Copyright (c) Microsoft Corporation. All rights reserved.

CommandLine: c:\ADNDBin\02simple.exe
Symbol search path is: *** Invalid ***
*****************************************************************************
* Symbol loading may be unreliable without a symbol search path.            *
* Use .symfix to have the debugger choose a symbol path.                    *
* After setting your symbol path, use .reload to refresh symbol locations. *
*****************************************************************************
Executable search path is:
ModLoad: 00400000 00408000   02simple.exe
ModLoad: 7c900000 7c9af000   ntdll.dll
ModLoad: 79000000 79045000   C:\WINDOWS\system32\mscoree.dll
ModLoad: 7c800000 7c8f6000   C:\WINDOWS\system32\KERNEL32.dll
(1568.1600): Break instruction exception - code 80000003 (first chance)
eax=00251eb4 ebx=7ffdf000 ecx=00000000 edx=00000001 esi=00251f48 edi=00251eb4
eip=7c90120e esp=0013fb20 ebp=0013fc94 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for
ntdll.dll -
ntdll!DbgBreakPoint:
7c90120e cc              int     3
0:000>
```

在输出信息中，第一部分信息是关于符号和符号搜索路径。在开始启动时，调试器提示没有设置符号搜索路径。在本章的后面，我们将进一步介绍符号文件，但就目前而言，你可以把符号文件看成是与映像和二进制文件相关的一种元数据，它们能极大地提高调试效率。接下来列出了一组模块（例如 02simple. exe、ntdll. dll 和 mscoree. dll 等），这组模块表示应用程序到目前为止已经加载的所有模块。接下来需要注意的就是导致调试器停止执行的事件。在这里，我们可以看到这个事件是中断指令异常（break instruction exception）。每当在调试器下启动一个进程或者将调试器附载（attach）到一个进程时，调试器都会注入一条断点指令，这条指令将使调试目标停止执行。显然，断点指令的重要作用就是，使用户有机会与调试器和调试目标进行交互。接下来显示的是寄存器的值，以及导致最近一个事件发生的指令（这里是断点指令 int 3）。最后，调试器显示一个命令提示符，并等待用户输入命令以执行。注意，命令提示符的格式如下：

```
X:Y>
```

其中，X 表示当前正在被调试的活动目标（在大多数调试会话中，这个值都为 0），而 Y 表示导致调试器中断的线程的 ID。

此时，我们已经成功地在调试器下启动了一个应用程序，并且调试器也提示我们继续输入命令。当在调试器下启动有问题的应用程序时，这种形式的调试能够很好地发挥作用。然而，在某些情况下，应用程序已经处于运行状态，并且通常无法从头开始在调试器下启动这个程序。这种情况下，我们希望有一种机制能够将调试器附载到运行中的应用程序。例如，假设开发了一个 Web 服务，并且该服务以 Internet 信息服务（Internet Information Service，IIS）为宿主。随着时间的推移，这个 Web 服务开始表现出奇怪的行为，因此你希望当服务处于这种奇怪状态时进行调试。在这种情况下，你可以使用调试器支持的命令行参数-p。参数-p 告诉调试器希望调试一个运行中的进程，与它同时指定的还包括被调试进程的 ID。对于任意一个进程，可以通过多种方式找出该进程的 ID。其中一种简单的方式就是使用 Windows 调试工具集中的 tlist. exe。运行 tlist. exe 会输出当前所有的进程及其进程 ID。为了说明这种情况，假定需要调试一个运行中的进程 03simple. exe。首先，启动 03simple. exe，注意不要按任意键退出这个程序，然后运行 tlist. exe，会看到 03simple. exe 的进程 ID。清单 3-2 给出了在启动 03simple. exe 之后运行 tlist. exe 的输出信息。

清单 3-2　tlist. exe 的输出

```
C:\>tlist
   0 System Process
   4 System
 712 smss.exe
 768 csrss.exe
 792 winlogon.exe
 836 services.exe
…
…
…
5828 wlmail.exe        Inbox - Windows Live Mail
4288 OSE.EXE
2424 devenv.exe        03simple.cs - Microsoft Visual Studio
1540 03Simple.exe
1320 cmd.exe           C:\WINDOWS\system32\cmd.exe - tlist
3120 tlist.exe
```

可以看到，03simple. exe 的进程 ID 为 1540。现在，我们就可以通过这个 ID 将调试器附载到 03simple. exe 上，如下所示：

```
C:\> ntsd -p 1540
```

执行上面这条命令后，将再次显示一个新的调试器窗口，并输出一组初始信息，如清单3-3所示。

清单3-3 将调试器附载到运行中的03simple.exe实例

```
Microsoft (R) Windows Debugger Version 6.9.0003.113 X86
Copyright (c) Microsoft Corporation. All rights reserved.
*** wait with pending attach
Symbol search path is: *** Invalid ***
*****************************************************************************
* Symbol loading may be unreliable without a symbol search path.            *
* Use .symfix to have the debugger choose a symbol path.                    *
* After setting your symbol path, use .reload to refresh symbol locations. *
*****************************************************************************
Executable search path is:
ModLoad: 00400000 00408000   c:\ADNDBin\03Simple.exe
ModLoad: 7c900000 7c9af000   C:\WINDOWS\system32\ntdll.dll
ModLoad: 79000000 79045000   C:\WINDOWS\system32\mscoree.dll
ModLoad: 7c800000 7c8f6000   C:\WINDOWS\system32\KERNEL32.dll
ModLoad: 77dd0000 77e6b000   C:\WINDOWS\system32\ADVAPI32.dll
ModLoad: 77e70000 77f02000   C:\WINDOWS\system32\RPCRT4.dll
ModLoad: 77fe0000 77ff1000   C:\WINDOWS\system32\Secur32.dll
ModLoad: 77f60000 77fd6000   C:\WINDOWS\system32\SHLWAPI.dll
ModLoad: 77f10000 77f59000   C:\WINDOWS\system32\GDI32.dll
ModLoad: 7e410000 7e4a1000   C:\WINDOWS\system32\USER32.dll
ModLoad: 77c10000 77c68000   C:\WINDOWS\system32\msvcrt.dll
ModLoad: 76390000 763ad000   C:\WINDOWS\system32\IMM32.DLL
ModLoad: 79e70000 7a3d6000   C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727
\mscorwks.dll
ModLoad: 78130000 781cb000   C:\WINDOWS\WinSxS\x86_Microsoft.VC80.CRT
_1fc8b3b9a1e18e3b_8.0.50727.762_x-ww_6b128700\MSVCR80.dll
ModLoad: 7c9c0000 7d1d7000   C:\WINDOWS\system32\shell32.dll
ModLoad: 773d0000 774d3000   C:\WINDOWS\WinSxS\x86_Microsoft.
Windows.Common-Controls_6595b64144ccf1df_6.0.2600.5512_x-ww_35d4ce83\comctl32.dll
ModLoad: 5d090000 5d12a000   C:\WINDOWS\system32\comctl32.dll
ModLoad: 790c0000 79b90000   C:\WINDOWS\assembly\NativeImages_v2.0
.50727_32\mscorlib\1519aecdfda7ccb6075750b4429d2834\mscorlib.ni.dll
ModLoad: 774e0000 7761d000   C:\WINDOWS\system32\ole32.dll
ModLoad: 79060000 790b3000   C:\WINDOWS\Microsoft.NET\Framework
\v2.0.50727\mscorjit.dll
(604.168c): Break instruction exception - code 80000003 (first chance)
eax=7ffd4000 ebx=00000001 ecx=00000002 edx=00000003 esi=00000004 edi=00000005
eip=7c90120e esp=00c8ffcc ebp=00c8fff4 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000246
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for
C:\WINDOWS\system32\ntdll.dll -
ntdll!DbgBreakPoint:
7c90120e cc              int     3
0:003>
```

可以看到，显示出的大部分信息都类似于清单3-1中输出的内容，只不过在已加载模块

列表中包含了更多的模块。这是意料之中的，因为通常随着应用程序的执行，被加载模块的数量也会逐渐增加，因此肯定比初始中断时的模块要多。

非侵入式调试

到目前为止，我们已经介绍了调试目标应用程序的两种方式。一种是在调试器下启动目标应用程序，另一种是将调试器附载到运行中的应用程序实例。这两种方式都会中断应用程序的执行，并且使用户能够通过调试器与目标应用程序交互。此外，还有第三种方式，称之为非侵入式调试（noninvasive debugging）。如果以非侵入的方式将调试器附载到目标应用程序上，那么调试器并不会中断目标应用程序的执行，而是挂起目标应用程序中的所有线程，从而观察目标应用程序的状态。需要记住的是，由于非侵入式调试采取了挂起线程的方式，因此将无法使用一些可以控制目标应用程序执行的命令（例如命令 g 和 bp 等）。

至此，我们已经成功地启动了一个调试会话，并且对调试器输出的初始信息做出了一些解释。接下来该采取哪些步骤？如何开始调试目标应用程序？前面提到过，调试器给出提示信息，指出并没有设置符号路径和符号搜索路径。我在前面也提到了这些符号是一些有助于调试过程的元数据。在接下来的一节中，我们将详细介绍符号以及如何在调试器中使用它们。

3.2 符号

在介绍如何在调试器中使用符号文件和源代码文件之前，首先简要给出符号文件的定义。符号文件是一种辅助数据，它包含了对应用程序代码的一些标注信息，这些信息在调试过程中非常有用。如果没有这些辅助数据，那么所能获得的信息就只有应用程序的二进制文件。对二进制代码进行调试是非常困难的，因为你无法看到代码中的函数名、数据结构名等。这正是符号文件所要解决的问题。在符号文件中包含了额外的信息对应用程序的二进制代码进行标注，从而更易于调试。这种标注信息被保存为符号文件。当前，符号文件的扩展名通常是 pdb，调试器能够很好地解析这种文件格式，从而让工程师们能更容易地进行调试。

在调试工作中，符号文件对于调试任务的成功与否起着至关重要的作用。在调试托管代码时，尽管.NET 二进制文件比非托管的二进制文件有着更好的自描述性，但符号文件仍然非常重要。符号文件能极大地提高调试任务的成功概率。在符号文件中包含了许多重要的信息，例如行号和局部变量的名字等。

有两种不同类型的符号文件：私有（private）符号文件和公有（public）符号文件。私有符号文件是大多数开发人员在日常工作中使用的符号文件，其中包含了调试会话中需要的所有符号信息。公有符号文件只是有选择地包含了一些符号信息，这会使调试工作变得更困难一些。例如，在 Microsoft 的符号服务器上存储了一些公有符号文件。每当将调试器指向

Microsoft符号服务器时，都可以从服务器上下载这些公有符号并且调试会话中使用它们。你可能会问，为什么要将私有符号文件和公有符号文件区分开来？既然私有符号文件包含了更多的信息并且能够使调试工作变得更容易，那么为什么不只使用私有符号文件？答案在于要保护知识产权。私有符号中包含了大量关于底层技术的信息，这使得非常容易就能对应用程序进行逆向工程（reverse engineer）。为了更好地保护知识产权，软件公司通常会提供公有符号文件，这些文件能够帮助客户调试一些问题，但不会泄露核心的技术信息。

要在调试器中使用符号，我们必须首先告诉调试器这些符号文件位于何处。如果它们与应用程序位于相同的目录或者一些已知的目录中，那么调试器会自动加载它们。如果情况并非如此，那么也可以通过调试器提供的一组命令来设置符号文件路径。当设置了符号路径后，调试器将尝试从已知路径以及你所设置的符号路径中加载符号。首先，我们来看看元命令（meta-command）sympath 。如果在执行 sympath 命令时不带任何参数，那么调试器会输出当前设置的符号路径，如下所示：

```
0:000> .sympath
Symbol search path is: <empty>
```

初始情况下，符号路径为空，因此调试器只会从已知符号路径中加载符号。假定将某个应用程序安装在以下目录中：

```
C:\program files\My application
```

而且，假定需要调试这个应用程序，并且需要的符号文件位于以下文件夹：

```
C:\MySymbols
```

那么要使用这些符号，需要运行以下命令：

```
0:000> .sympath c:\mysymbols
Symbol search path is: c:\mysymbols
```

此时，调试器将记录上面新的符号路径，但并不会从这个路径中加载任何符号。要指示调试器加载符号，可以使用元命令 reload，这个命令能枚举出进程地址空间中所有已加载的模块，并且尝试找出与各个模块相关的符号文件。

```
0:000> .reload
Reloading current modules
....
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for
ntdll.dll -
```

如果调试器无法找到符号文件，那么它会输出上面的错误提示。在这里，除 ntdll. dll 模块外，调试器为其他的模块都成功地加载了符号。ntdll. dll 是 Windows 本身的一个模块，因此我们可以从 Microsoft 公有符号服务器上下载这个模块的符号。有一个简单的元命令 symfix，这个命令能自动将符号路径设置为 Microsoft 公有符号服务器，如下所示：

```
0:000> .symfix
No downstream store given, using c:\Program Files\Debugging Tools for
Windows (x86)\sym
```

"No downstream store..." 这条消息指的是，没有指定一个本地路径来缓存下载的符号。你可以在 symfix 命令中指定一个本地路径，用来存储下载的符号，这样调试器就不用每次都重新下载相同的符号，而只需从本地缓存中加载现有的符号文件。在默认情况下，如果没有指定本地路径（符号缓存），那么调试器将使用调试软件包安装路径下的 sym 文件夹。

到目前为止，我们知道了如何设置符号路径（通过 sympath），以及如何将符号路径设置为指向 Microsoft 公有符号服务器。sympath 和 symfix 等命令都有一种变化形式，即 sympath+ 和symfix+ 。这些命令不会覆盖现有的符号路径，而是把新的符号路径添加到现有的符号路径后面。例如，假定我们已经通过 sympath 命令将符号路径设置为 c:\mysym。如果想要添加另一个符号路径，就可以使用以下命令：

```
0:000> .sympath c:\adndbin
Symbol search path is: c:\adndbin
0:000> .sympath+ c:\mysym
Symbol search path is: c:\adndbin;c:\mysym
0:000>
```

到这里，关于如何对调试器进行设置以访问正确符号的介绍就可以告一段落了。如果没有正确的符号，调试工作会变得非常困难，而对于任何调试会话来说，知道如何使调试器访问正确的符号也是非常关键的。

3.3 控制调试目标的执行

在任何调试会话中，控制调试目标的执行是非常有用的功能。我们可能要先设置一些断点，然后恢复目标应用程序的执行直到触发断点，查看目标应用程序的状态，单步跟踪到函数内部，最后再次恢复执行等。在非托管调试器中提供了一组命令可以用来控制这种执行过程。在接下来的几节中，我们将介绍一些用于控制调试目标执行的常见命令。

3.3.1 中断执行

根据调试器的配置方式，可以有多种不同的方式来中断程序的执行。某些情况下，你希望通过手动方式来中断执行（例如在调试死锁问题时）并分析故障。要以手动方式来中断执行，可以使用组合键 CTRL + C。这个组合键将使调试器把一个线程注入到目标进程中，并且执行一条中断语句。调试器还可以通过其他的方式来中断执行，包括设置断点，在本章的后面将进一步介绍这种方式。通过设置断点，我们可以很方便地使调试器在执行流程中的任意位置上中断执行。最后，另一种使调试器中断执行的情况就是当发生异常时（第一轮异常或者第二轮异常）。在本章的后面同样将进一步介绍异常。

3.3.2 恢复执行

当调试器中断执行时（由于触发了断点或者其他的事件），调试器会给出一个命令提示符，此时可以通过在命令提示符中输入命令来与调试目标进行交互。通常，在这种情况下可以分析调试目标的某些状态，或者进一步设置更多的断点。当完成了对当前中断位置的分析后，你希望恢复调试目标的执行直到发生另一个调试事件。要恢复调试目标的执行，可以使用调试器命令 g（go）。如果命令 g 不带任何参数，那么只是恢复调试目标的执行，直到下一次发生某个调试事件。这种情况的一个常见示例就是，在调试器下启动进程。我们通过以下命令行在调试器下启动应用程序 03breakpoint. exe。

```
C:\ADNDBin\ntsd 03Breakpoint.exe
```

当执行上面的命令时，调试器在启动应用程序后将自动地中断并进入到调试器，如清单 3-4 所示。

清单 3-4 在调试器下启动进程时的初始断点

```
Microsoft (R) Windows Debugger Version 6.9.0003.113 X86
Copyright (c) Microsoft Corporation. All rights reserved.

CommandLine: 03Breakpoint.exe
Symbol search path is: *** Invalid ***
*************************************************************************
* Symbol loading may be unreliable without a symbol search path.        *
* Use .symfix to have the debugger choose a symbol path.                *
* After setting your symbol path, use .reload to refresh symbol locations. *
*************************************************************************
Executable search path is:
ModLoad: 00400000 00408000   03Breakpoint.exe
ModLoad: 7c900000 7c9af000   ntdll.dll
ModLoad: 79000000 79045000   C:\WINDOWS\system32\mscoree.dll
ModLoad: 7c800000 7c8f6000   C:\WINDOWS\system32\KERNEL32.dll
(dbc.a98): Break instruction exception - code 80000003 (first chance)
eax=00251eb4 ebx=7ffd5000 ecx=00000000 edx=00000001 esi=00251f48 edi=00251eb4
eip=7c90120e esp=0013fb20 ebp=0013fc94 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for
ntdll.dll -
ntdll!DbgBreakPoint:
7c90120e cc              int     3
0:000>
```

触发初始断点是调试器的默认行为，也是调试人员开始分析应用程序的最早时刻。此时，调试目标会停止执行并等待输入命令。当执行完所有需要执行的命令时（例如设置符号路径等），可以通过命令 g 来恢复调试目标的执行，如下所示：

```
0:000> .symfix
No downstream store given, using c:\Program Files\Debugging Tools for Windows\sym
0:000> .reload
Reloading current modules
....
0:000> g
ModLoad: 5cb70000 5cb96000   C:\WINDOWS\system32\ShimEng.dll
ModLoad: 77dd0000 77e6b000   C:\WINDOWS\system32\ADVAPI32.dll
ModLoad: 77e70000 77f02000   C:\WINDOWS\system32\RPCRT4.dll
ModLoad: 77fe0000 77ff1000   C:\WINDOWS\system32\Secur32.dll
ModLoad: 77f60000 77fd6000   C:\WINDOWS\system32\SHLWAPI.dll
ModLoad: 77f10000 77f59000   C:\WINDOWS\system32\GDI32.dll
ModLoad: 7e410000 7e4a1000   C:\WINDOWS\system32\USER32.dll
ModLoad: 77c10000 77c68000   C:\WINDOWS\system32\msvcrt.dll
ModLoad: 76390000 763ad000   C:\WINDOWS\system32\IMM32.DLL
ModLoad: 79e70000 7a3d6000   C:\WINDOWS\Microsoft.NET\Framework
\v2.0.50727\mscorwks.dll
ModLoad: 78130000 781cb000   C:\WINDOWS\WinSxS\x86_Microsoft
.VC80.CRT_1fc8b3b9a1e18e3b_8.0.50727.762_x-ww_6b128700\MSVCR80.dll
ModLoad: 7c9c0000 7d1d7000   C:\WINDOWS\system32\shell32.dll
ModLoad: 773d0000 774d3000   C:\WINDOWS\WinSxS\x86_Microsoft.Windows.Common-
Controls_6595b64144ccf1df_6.0.2600.5512_x-ww_35d4ce83\comctl32.dll
ModLoad: 5d090000 5d12a000   C:\WINDOWS\system32\comctl32.dll
ModLoad: 60340000 60348000   C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727
\culture.dll
ModLoad: 790c0000 79b90000   C:\WINDOWS\assembly\NativeImages
_v2.0.50727_32\mscorlib\1519aecdfda7ccb6075750b4429d2834\mscorlib.ni.dll
ModLoad: 774e0000 7761d000   C:\WINDOWS\system32\ole32.dll
ModLoad: 79060000 790b3000   C:\WINDOWS\Microsoft.NET\Framework
\v2.0.50727\mscorjit.dll
Press any key to call an instance function
```

当执行命令 g 时，调试器会指示调试目标恢复执行，直到发生另一个调试事件。在前面的示例中，调试目标正在等待用户输入命令。

禁用初始和退出断点

如果不希望调试器在初始启动时停止程序的执行，那么在启动调试器时可以使用开关 -g。例如，在前面的示例中，可以通过以下命令来启动它：

```
ntsd -g 03Breakpoint.exe
```

通过使用-g 开关，调试器在启动程序时不会停止程序的执行。同样，每当调试目标准备退出时，调试器也将停止执行。这种行为同样可以通过使用 -G（注意是大写）开关来控制，这能避免在进程结束时触发最终的断点。

3.3.3 单步调试代码

通常，在调试器中分析代码时，我们需要单步调试（step through）代码。记住：如果在调试托管代码时使用非托管调试器，那么通常是对 JIT 编译器产生的机器代码进行单步调试。在少数情况下，例如当需要分析 CLR 本身的代码时，才可能需要对常规非托管代码进行单步调试。有两个主要的命令可用于单步调试代码，分别是命令 p(step)和 t(trace)。我们通过一个程序 03breakpoint. exe 来说明如何使用这些命令。在调试器下启动 03breakpoint. exe，并执行到程序提示按任意键。按下任意键继续执行，程序会要求再次按下任意键。此时，按下 CTRL + C 中断 Main 函数的执行。清单 3-5 给出了我们执行的命令，这些命令将得到 JIT 编译器在编译 AddAndPrint 函数时生成的机器代码，此外还演示了如何在这个函数上设置一个断点。

清单 3-5　在 AddAndPoint 函数中设置一个断点

```
Microsoft (R) Windows Debugger Version 6.9.0003.113 X86
Copyright (c) Microsoft Corporation. All rights reserved.

CommandLine: 03Breakpoint.exe

...
...
...
0:000> .symfix
No downstream store given, using c:\Program Files\Debugging Tools for Windows
(x86)\sym
0:000> .reload
Reloading current modules
....
0:000> g
ModLoad: 763c0000 76486000   C:\Windows\system32\ADVAPI32.dll
ModLoad: 77260000 77323000   C:\Windows\system32\RPCRT4.dll
ModLoad: 77980000 779d8000   C:\Windows\system32\SHLWAPI.dll
ModLoad: 76570000 765bb000   C:\Windows\system32\GDI32.dll
...
...
...
ModLoad: 790c0000 79bf6000   C:\Windows\assembly\NativeImages_v2.0.50727_32\
mscorlib\5b3e3b0551bcaa722c27dbb089c431e4\mscorlib.ni.dll
ModLoad: 79060000 790b6000   C:\Windows\Microsoft.NET\Framework
\v2.0.50727\mscorjit.dll
Press any key (1st instance function)
 Adding 10+5=15
Press any key (2nd instance function)
(1a48.1e70): Break instruction exception - code 80000003 (first chance)
eax=7ffda000 ebx=00000000 ecx=00000000 edx=77bcd094 esi=00000000 edi=00000000
eip=77b87dfe esp=01d7faa4 ebp=01d7fad0 iopl=0         nv up ei pl zr na pe nc
```

```
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
ntdll!DbgBreakPoint:
77b87dfe cc              int     3
0:004> .loadby sos.dll mscorwks
0:004> !name2ee 03breakpoint.exe
Advanced.NET.Debugging.Chapter3.Breakpoint.AddAndPrint
Module: 000c2d8c (03Breakpoint.exe)
Token: 0x06000002
MethodDesc: 000c3178
Name: Advanced.NET.Debugging.Chapter3.Breakpoint.AddAndPrint(Int32, Int32)
JITTED Code Address: 002a0178
0:004> bp 002a0178
0:004> g
(1a48.1e58): Control-C exception - code 40010005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=01c4fcbc ebx=00000000 ecx=00000000 edx=77b99a94 esi=00000000 edi=00000002
eip=77886da1 esp=01c4fcac ebp=01c4fd30 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
KERNEL32!CtrlRoutine+0xbf:
77886da1 c745fcfeffffff  mov     dword ptr [ebp-4],0FFFFFFFEh
ss:0023:01c4fd2c=00000000
0:003> g
gBreakpoint 0 hit
eax=01ee4344 ebx=001cf33c ecx=01ee4344 edx=00000064 esi=01ee4344 edi=01ee4344
eip=002a0178 esp=001cf2dc ebp=001cf320 iopl=0         nv up ei ng nz ac po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000293
002a0178 57              push    edi
0:000>
```

在清单 3-5 中，你可以看到我们首先设置了正确的符号路径，接下来在 AddAndPrint 函数上设置了一个断点。当恢复执行时，将立即触发断点，并且所处的位置是 AddAndPrint 函数的起始执行处。暂时先不考虑命令 bp 的细节内容，我们将在本章后面详细介绍它。我们如何得知当前处于代码的什么位置呢？答案在于指令指针（即 x86 系统上的 eip 寄存器）。如果想要查看接下来的一些指令是什么，可以使用命令 u，如下所示：

```
0:000> u
002a0178 57              push    edi
002a0179 56              push    esi
002a017a 53              push    ebx
002a017b 55              push    ebp
002a017c 83ec08          sub     esp,8
002a017f 890c24          mov     dword ptr [esp],ecx
002a0182 8bfa            mov     edi,edx
002a0184 833d582f0c0000  cmp     dword ptr ds:[0C2F58h],0
```

与大多数函数一样，开头的部分指令是函数的前导指令（function prolog），用于设置当前的栈帧。我们使用命令 p 来单步调试前面的一些指令：

```
0:000> p
eax=01ee4344 ebx=001cf33c ecx=01ee4344 edx=00000064 esi=01ee4344 edi=01ee4344
eip=002a0179 esp=001cf2d8 ebp=001cf320 iopl=0         nv up ei ng nz ac po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000293
002a0179 56              push    esi
0:000>
eax=01ee4344 ebx=001cf33c ecx=01ee4344 edx=00000064 esi=01ee4344 edi=01ee4344
eip=002a017a esp=001cf2d4 ebp=001cf320 iopl=0         nv up ei ng nz ac po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000293
002a017a 53              push    ebx
0:000>
eax=01ee4344 ebx=001cf33c ecx=01ee4344 edx=00000064 esi=01ee4344 edi=01ee4344
eip=002a017b esp=001cf2d0 ebp=001cf320 iopl=0         nv up ei ng nz ac po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000293
002a017b 55              push    ebp
0:000>
eax=01ee4344 ebx=001cf33c ecx=01ee4344 edx=00000064 esi=01ee4344 edi=01ee4344
eip=002a017c esp=001cf2cc ebp=001cf320 iopl=0         nv up ei ng nz ac po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000293
002a017c 83ec08          sub     esp,8
```

从上述清单可以看出，命令 p 执行了一条指令，并且显示出所有寄存器的结果。你可能会奇怪，为什么在示例中有的输入的命令为空。这是调试器的另一个强大功能。如果不输入任何命令，而只是按下 Enter 键，那么将再次执行最后一次执行的命令（这里是命令 p）。命令 p 的另一个变化形式为 pc，这个命令会重复执行所有指令，直到遇到下一个 call 指令。下面的清单中给出了一个示例。我们在已经打开的调试会话中执行这个命令。

```
0:000> pc
eax=00000096 ebx=01ee3dac ecx=79102290 edx=00000000 esi=01ee4344 edi=00000064
eip=002a01ae esp=001cf2c4 ebp=001cf320 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
002a01ae e8691ee1ff      call    000b201c
```

可以看到，命令 pc 执行了一些指令，直到到达 call 00b201c 这条指令。如果使用命令 p 来单步调试这个 call 指令，会出现什么情况？命令 p 会完整地执行完 call 指令，并且到达位于这个 call 指令之后的下一条指令。

```
0:000> p
eax=01ee4350 ebx=01ee3dac ecx=79102290 edx=003160f0 esi=01ee4344 edi=00000064
eip=002a01b3 esp=001cf2c4 ebp=001cf320 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
002a01b3 8bf0            mov     esi,eax
```

当然，我们也可以使用命令 t 来单步调试 call 指令，稍后会介绍这个命令。命令 p 的最后一种变化形式为 pt，它会一直执行指令，直到遇见一个 ret 指令：

```
0:000> pt
Adding 100+50=150
eax=00000001 ebx=001cf33c ecx=01ee3d1c edx=00000000 esi=01ee4344 edi=01ee4344
eip=002a01f5 esp=001cf2dc ebp=001cf320 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
002a01f5 c20400          ret     4
```

当执行这个命令时，我们可以看到 AddAndPrint 函数已经完成了它的工作（输出加法运算的结果），并且调试器停止在 ret 指令处，这条指令会把执行控制权返回给主调函数（Main）。

现在，让我们把注意力转向命令 t（trace）。与命令 p 非常相似的是，命令 t 执行单条指令，并显示所有寄存器的结果。关键区别在于命令 t 在执行 call 指令或者中断指令时的行为。重新启动示例程序 03breakpoint. exe，并且使用与清单 3-5 中相同的命令执行到 AddAndPrint 函数的起始位置。我们将单步调试 call 指令并跟踪该指令后面的代码，如下所示：

```
0:003> g
gBreakpoint 0 hit
eax=01d44344 ebx=0028f1ac ecx=01d44344 edx=00000064 esi=01d44344 edi=01d44344
eip=009f0178 esp=0028f14c ebp=0028f190 iopl=0         nv up ei ng nz ac po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000293
009f0178 57              push    edi
0:000> pc
eax=00000096 ebx=01d43dac ecx=79102290 edx=00000000 esi=01d44344 edi=00000064
eip=009f01ae esp=0028f134 ebp=0028f190 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
009f01ae e8691e8aff      call    0029201c
0:000> t
eax=00000096 ebx=01d43dac ecx=79102290 edx=00000000 esi=01d44344 edi=00000064
eip=0029201c esp=0028f130 ebp=0028f190 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
0029201c 8b4104          mov     eax,dword ptr [ecx+4] ds:0023:79102294=0000000c
0:000>
```

通过使用命令 pc，可以很快地执行到下一个 call 指令（call 0029201c），接着执行命令 t，这会单步调试位于地址 0029201c 上的代码。

命令 t 有多种变化形式，包括：

- ta < address > ：执行到 address 指定的地址，并且将包含被调用函数的单步执行显示出来。
- tc：执行到下一个 call 指令，并将包含被调用函数的单步执行显示出来。
- tt：执行到下一个 ret 指令，并将包含被调用函数的单步执行显示出来。

3.3.4 退出调试会话

在执行完一个调试会话后，可以有多种方法退出调试会话。如果希望结束调试会话并终

止调试目标，那么可以使用命令 q（quit）。通常，我们希望结束调试会话，但让调试目标继续运行。在这种情况下，可以通过命令 qd（quit and detach）将调试从调试目标程序上卸载（detach），而并不结束调试目标。只有在 Windows XP 和后续版本的操作系统上才支持 qd 命令。

3.4 加载托管代码调试的扩展命令

在非托管调试器中可以使用两种不同类型的命令。第一种被称为元命令。元命令是指在调试引擎中内置的命令。例如 help、sympath 以及 cls 等，都是元命令。当执行元命令时，必须在命令前面加上前缀“.”。例如，要设置符号路径，我们可以使用元命令 sympath，如下所示：

```
0:000> .sympath c:\adndbin
Symbol search path is: c:\adndbin
```

要得到元命令的完整列表，可以使用 help 命令。另一种类型的命令被称为扩展命令。扩展命令是在调试器引擎之外的独立 DLL 中实现的，这些 DLL 也被称为调试器扩展。在发布非托管调试器时，通常带有一定数量的调试器扩展 DLL，这些 DLL 由 Microsoft 的不同技术小组开发，分别用于对某种特定技术中的问题进行调试。除了随调试工具一起发布的扩展 DLL 之外，还可以通过调试器 SDK 来开发自己的扩展 DLL（请参见调试器帮助文档或者《Windows 高级调试》一书的第 11 章）。在执行扩展命令时，要在命令前面加上前缀“!”。例如，要执行扩展命令 htrace，可以使用以下形式：

```
0:000> !htrace -enable
Handle tracing enabled.
Handle tracing information snapshot successfully taken.
```

除了随 Windows 调试工具集发布的诸多扩展 DLL 外，还有许多其他的扩展 DLL。在调试托管代码时，有两个扩展 DLL 需要注意，它们分别是 SOS 和 SOSEX。

在使用这些扩展 DLL 之前，必须通过元命令 load 来通知调试器。元命令 load 的参数是需要加载的扩展 DLL 的路径。例如，要加载位于 c：\ adnbin 路径下的 myext. dll，可以使用以下命令：

```
.load c:\adndbin\myext.dll
```

我们来看看如何通过元命令 load 来加载 SOS 调试器扩展和 SOSEX。

3.4.1 加载 SOS 调试器扩展

SOS 调试器扩展的 DLL（sos. dll）与程序使用的 CLR 版本是相关的。因此，在发布每个 CLR 的主版本的同时，会发布一个新版本的 SOS 调试器扩展，以确保这个 DLL 可以使用该

版本 CLR 的新功能。SOS 调试器扩展是作为运行时的一部分发布的，它位于以下路径：

```
%systemroot%\Microsoft.NET\Framework\<framework version>\sos.dll
```

我们可以在上面的路径中指定需要调试哪一个框架版本，并且通过元命令 load 将它传递给调试器。例如，在我的系统上可以使用以下调试器命令来加载与 CLR 2.0 版本相对应的 SOS：

```
.load c:\windows\Microsoft.NET\Framework\v2.0.50727\sos.dll
```

为什么需要多个版本

为什么需要多个版本的 SOS 调试器扩展？因为 SOS 调试器扩展需要了解 CLR 的内部细节，所以每当对 CLR 进行修改或者增强时，就必须生成一个新版本的 SOS。

刚才给出的方法或许有些困难，因为需要找出应用程序所使用的 CLR 的正确版本，还要输入很长的路径。要解决这个问题，我们可以使用另一个元命令 loadby。元命令 loadby 的语法如下所示：

```
.loadby DLLName ModuleName
```

元命令 loadby 将尝试找出由 ModuleName 指定的路径（通过查看进程中的已加载模块列表），并且使用这个路径来加载指定的 DLLName（调试器扩展的 DLL）。在前面已经指出 SOS 调试器扩展与 CLR 版本是相关的，如果可以找到一个模块，并且 SOS 调试器扩展与这个模块位于同一个目录，那么就可以使用 loadby 命令来加载 SOS 调试器扩展。假设正在查找的模块是 mscorwks，这是 CLR 的主要引擎之一，那么只需执行以下命令：

```
.loadby sos.dll mscorwks
```

然后，调试器引擎将加载正确版本的 SOS 调试器扩展。

如果 mscorwks 模块还没有被加载，那么元命令 loadby 将提示以下错误信息：

```
0:000> .loadby sos.dll mscorwks
Unable to find module 'mscorwks'
```

如果需要在加载 mscorwks 模块时立即加载 SOS 调试器扩展，那么可以使用 sxe 命令。命令 sxe 用于控制在目标应用程序中的异常行为。我并不打算介绍命令 sxe 的所有选项，但其中有一个选项 sxe ld 是非常有用的，这个选项可以使得在加载某个特定的模块后，立即中断进入到调试器。我们可以通过 sxe ld 命令来告知调试器，当 mscorwks 被加载时中断调试器，然后加载 SOS 调试器扩展，如清单 3-6 所示。

清单 3-6 在 mscorwks.dll 被加载后立即加载 SOS 调试器扩展

```
Microsoft (R) Windows Debugger Version 6.9.0003.113 X86
Copyright (c) Microsoft Corporation. All rights reserved.
```

```
CommandLine: 01MDASample.exe
Symbol search path is: *** Invalid ***
...
...
...
(1ff0.1d6c): Break instruction exception - code 80000003 (first chance)
eax=00000000 ebx=00000000 ecx=002ff964 edx=77b99a94 esi=fffffffe edi=77b9b6f8
eip=77b87dfe esp=002ff97c ebp=002ff9ac iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for
ntdll.dll -
ntdll!DbgBreakPoint:
77b87dfe cc              int     3
0:000> .symfix
No downstream store given, using c:\Program Files\Debugging Tools for Windows
(x86)\sym
0:000> .reload
Reloading current modules
....
0:000> sxe ld mscorwks.dll
0:000> g
ModLoad: 79e70000 7a3ff000   C:\Windows\Microsoft.NET\Framework
\v2.0.50727\mscorwks.dll
eax=00000000 ebx=00000000 ecx=00000000 edx=4000001e esi=7ffdf000 edi=20000000
eip=77b99a94 esp=002fefd4 ebp=002ff018 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
ntdll!KiFastSystemCallRet:
77b99a94 c3              ret
0:000> .loadby sos.dll mscorwks
0:000>
```

从清单 3-6 中可以看出，当 mscorwks 被加载时，调试器中断执行，这样我们便可以通过元命令 loadby 来加载 SOS 调试器扩展。

SOS 与 Silverlight

Silverlight 使用了自带的 CLR，这个 CLR 是经过裁剪之后的版本，位于 coreclr. dll 中。如果要使用非托管调试器来调试 Silverlight 程序，那么必须下载正确版本的 SOS 调试器扩展。在 Silverlight 开发运行时中包含了在 Silverlight 中使用的 SOS，下载网址为 http://www. microsoft. com/silverlight/resources/tools. aspx。

在安装后，用于 Silverlight 的 SOS 调试器扩展位于 Silverlight 的安装文件夹中。例如，在我的机器上，这个文件的位置是：

```
c:\Program Files\Microsoft Silverlight\2.0.31005.0\sos.dll
```

3.4.2 加载 SOSEX 调试器扩展

SOSEX 是由 Steve Johnson 开发的一个调试器扩展，用于调试托管代码，可以免费下载（参见第 1 章，了解下载地址和安装指令）。SOSEX 增强了 SOS 的功能，它能使某些特定的调试任务更为高效。我们将在本章的后面介绍 SOSEX 命令的细节，但就目前而言，将只介绍如何在调试器下正确地加载 SOSEX。

当下载 SOSEX 并将其安装到指定的文件夹后，可以在调试器中使用元命令 load 来加载它。在前面已经提到过，需要为 SOSEX 所在的 DLL 指定完整的路径。另一种方式是将 SOSEX 的 DLL 复制到调试器的安装路径，这样可以避免指定完整路径。在接下来的示例中，我把 SOSEX 安装到 c:\myexts 文件夹下，并通过以下命令来加载它：

```
.load c:\myexts\sosex.dll
```

请注意，虽然在没有加载 mscorwkd. dll 的情况下也能加载 SOSEX 调试器扩展，但这些命令本身并不能工作，会输出以下错误信息：

```
Unable to initialize .NET interface. The CLR may not yet be loaded in the target
process.
```

如果需要在 mscorwkd. dll 被加载后立即使用某个 SOSEX 调试器命令，那么可以使用在前面 sxe ld mscorwkds. dll 中采用的相同技术。

3.5 控制 CLR 的调试

在调试. NET 程序时，调试器可以加载一个辅助 DLL，称为 mscordacwks. dll，这个 DLL 用于输出托管代码调试过程中的各种信息（例如 SOS 命令的输出）。加载 mscordacwks. dll 的路径取决于被加载到进程中的 mscorwks. dll 的路径。在实时调试（live debugging）中通常不存在问题，因为我们希望加载与已加载的 mscorwks. dll 相对应的 mscordacwks. dll 版本，但在事后调试（postmortem debugging）中则可能出现版本不匹配的情况。要想控制调试行为，我们可以使用元命令 cordll，并告诉调试器加载 mscordacwks. dll 的确切位置。例如，如果要指定一个新路径，那么可以使用以下命令：

```
.cordll -lp c:\x\y\z
```

这样便能告诉调试器从文件夹 c:\x\y\z 下加载 mscordacwks. dll。如果要卸载 mscordacwks. dll，可以使用 -u 开关。

在第 8 章中，我们将看到这个命令在调试崩溃转储（crash dump）时带来的潜在危险。

3.6 设置断点

设置断点的目的是为了告诉目标程序在执行到断点所在的位置时停止执行。断点使开发

人员可以分析程序在执行流中的状态，并使得更容易地对问题根本原因进行分析。在非托管代码调试中，断点是很容易设置的，因为（在大多数时候）我们知道需要进行分析的代码位置。设置断点的命令为 bp（breakpoint）。命令 bp 的最简单形式是只带一个参数，用于指定设置断点的位置。我们来看一个在非托管代码中设置断点的简单示例。清单 3-7 给出了在 notepad. exe 的 SaveFile 函数中设置断点的步骤。

清单 3-7 在 notepad. exe 中设置一个断点

```
Microsoft (R) Windows Debugger Version 6.9.0003.113 X86
Copyright (c) Microsoft Corporation. All rights reserved.
CommandLine: notepad.exe
...
...
...
ntdll!DbgBreakPoint:
7c90120e cc              int     3
0:000> .symfix
No downstream store given, using c:\Program Files\Debugging Tools for Windows\sym
0:000> .reload
Reloading current modules
..............
0:000> X notepad!*Save*
01001a28 notepad!NpSaveDialogHookProc = <no type information>
0100270f notepad!CheckSave = <no type information>
0100a528 notepad!g_ftSaveAs = <no type information>
01003a39 notepad!SaveGlobals = <no type information>
010012e4 notepad!_imp__GetSaveFileNameW = <no type information>
0100a540 notepad!szSaveFilterSpec = <no type information>
01004eae notepad!SaveFile = <no type information>
01009854 notepad!fInSaveAsDlg = <no type information>
0100136c notepad!s_SaveAsHelpIDs = <no type information>
01009090 notepad!szSaveCaption = <no type information>
0:000> bp notepad!SaveFile
0:000> g
ModLoad: 5cb70000 5cb96000   C:\WINDOWS\system32\ShimEng.dll
ModLoad: 6f880000 6fa4a000   C:\WINDOWS\AppPatch\AcGenral.DLL
ModLoad: 76b40000 76b6d000   C:\WINDOWS\system32\WINMM.dll
ModLoad: 774e0000 7761d000   C:\WINDOWS\system32\ole32.dll
...
...
...
ModLoad: 01790000 01a55000   C:\WINDOWS\system32\xpsp2res.dll
ModLoad: 73ba0000 73bb3000   C:\WINDOWS\system32\sti.dll
ModLoad: 74ae0000 74ae7000   C:\WINDOWS\system32\CFGMGR32.dll
Breakpoint 0 hit
eax=0007fbb0 ebx=00000104 ecx=00002bd2 edx=7c90e4f4 esi=00000000 edi=7c80ba7f
eip=01004eae esp=0007fb40 ebp=0007fdbc iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
notepad!SaveFile:
```

```
01004eae 8bff          mov     edi,edi
0:000>
```

在设置好断点并重新加载符号之后，可以使用命令 X（eXamine symbols）来显示包含“SaveFile”的所有符号。其中有一个符号非常匹配并且刚好也是一个函数。然后，我们通过命令 bp 在这个函数上设置一个断点（另一种方式是使用地址值 0x01004eae）。当恢复程序的执行后，点击 notepad 的“File“菜单，选择“Save”菜单项，调试器将中断执行并停在 SaveFile 函数处。命令 bp 的最简单形式是在非托管代码程序中设置一个断点，即首先找到需要分析的函数的地址，然后使用 bp 命令。

对于托管代码来说，情况有一些复杂。在第 2 章中，我们介绍了 CLR 通过 JIT 编译器在运行时将某个函数的中间语言（IL）转换为机器代码。CLR 可以自由选择将这段机器代码放在它认为合适的任何地方。然后，问题就变成了：如果要在函数 X 中设置一个断点，那么如何得知函数 X 的机器代码的确切位置来准确地设置断点？答案在于，我们要了解如何要求 CLR 给出函数被编译后的地址，然后再使用断点命令在这个地址上设置断点。请注意，根据这个函数是否已经被编译了，此方法有着不同的变化形式。我们可以通过清单 3-8 中的小程序来说明如何设置托管代码的断点。

清单 3-8 示例断点程序

```
using System;
using System.Text;

namespace Advanced.NET.Debugging.Chapter3
{
    class Breakpoint
    {
        static void Main(string[] args)
        {
            //
            //第一次调用实例函数
            //
            Console.WriteLine("Press any key (1st instance function)");
            Console.ReadKey();
            Breakpoint bp = new Breakpoint();
            bp.AddAndPrint(10, 5);
            //
            //第二次调用函数
            //
            Console.WriteLine("Press any key (2nd instance function)");
            Console.ReadKey();
            bp = new Breakpoint();
            bp.AddAndPrint(100, 50);
        }

        public void AddAndPrint(int a, int b)
```

```
        {
            int res = a + b;
            Console.WriteLine("Adding {0}+{1}={2}", a, b, res);
        }
    }
}
```

清单 3-8 中的源代码和二进制文件位于以下文件夹中：

- 源代码文件：C:\ADND\Chapter3\Breakpoint
- 二进制文件：C:\ADNDBin\03breakpoint.exe

3.6.1 在 JIT 编译生成的函数上设置断点

在前面已经介绍过，JIT 编译器编译了一个函数并将其放在内存中。如果我们知道 JIT 编译器保存机器代码的位置，就可以使用调试器的 bp 命令来设置断点。我们再次使用在前面介绍过的 03breakpoint.exe 程序，并且试验能否在 AddAndPrint 函数上设置一个断点。具体来说，我们希望在第二次调用这个函数时设置一个断点，以便分析其中潜在的错误。在调试器下启动 03breakpoint.exe，并继续执行直到程序提示按任意键。按下任意键，等待并直到第二次提示按下任意键。此时，按下 CTRL-C 进入调试器。这是在第二个 AddAndPrint 函数中设置断点的起始位置。第一个任务就是判断这个函数是否已经被 JIT 编译器编译，这可以通过 SOS 命令 name2ee 来进行判断。命令 name2ee 的形式如下所示：

```
!name2ee <module name> <type or method name>
```

参数 module name 是需要分析的模块，而 type or method name 则表示需要获得信息的类型名或者方法名。在这里，module name 是 03breakpoint.exe，而 type name 为 Advanced.NET.Debugging.Chapter3.Breakpoint.AddAndPrint。

```
0:004> !name2ee 03breakpoint.exe
Advanced.NET.Debugging.Chapter3.Breakpoint.AddAndPrint
Module: 00112d8c (03Breakpoint.exe)
Token: 0x06000002
MethodDesc: 00113178
Name: Advanced.NET.Debugging.Chapter3.Breakpoint.AddAndPrint(Int32, Int32)
JITTED Code Address: 003e0178
0:004>
```

可以看到，在最后一行输出中，这个方法的状态为 JITTED（表示已经被 JIT 编译器编译过了），并且编译之后的地址为 003e0178。我们可以通过命令 U 对这段代码做一个简单的完整性检查（sanity check，在后面将进一步讨论），如清单 3-9 所示。

清单 3-9　通过反编译 JIT 生成的代码对 JIT 地址进行完整性检查

```
0:004> !U 003e0178
Normal JIT generated code
Advanced.NET.Debugging.Chapter3.Breakpoint.AddAndPrint(Int32, Int32)
Begin 003e0178, size 80
>>> 003e0178 57              push    edi
003e0179 56              push    esi
003e017a 53              push    ebx
003e017b 55              push    ebp
003e017c 83ec08          sub     esp,8
003e017f 890c24          mov     dword ptr [esp],ecx
003e0182 8bfa            mov     edi,edx
003e0184 833d582f110000  cmp     dword ptr ds:[112F58h],0
003e018b 7405            je      003e0192
003e018d e8b581d479      call    mscorwks!JIT_DbgIsJustMyCode (7a128347)
003e0192 33d2            xor     edx,edx
003e0194 89542404        mov     dword ptr [esp+4],edx
003e0198 90              nop
003e0199 8b44241c        mov     eax,dword ptr [esp+1Ch]
003e019d 03c7            add     eax,edi
003e019f 89442404        mov     dword ptr [esp+4],eax
003e01a3 8b1d4c30c602    mov     ebx,dword ptr ds:[2C6304Ch] ("Adding {0}+{1}={2}")
...
...
...
003e01e2 8bd5            mov     edx,ebp
003e01e4 8bcb            mov     ecx,ebx
003e01e6 e8359c0079      call    mscorlib_ni+0x329e20 (793e9e20)
(System.Console.WriteLine
(System.String, System.Object, System.Object, System.Object), mdToken: 060007c8)
003e01eb 90              nop
003e01ec 90              nop
003e01ed 90              nop
003e01ee 83c408          add     esp,8
003e01f1 5d              pop     ebp
003e01f2 5b              pop     ebx
003e01f3 5e              pop     esi
003e01f4 5f              pop     edi
003e01f5 c20400          ret     4
0:004>
```

在反汇编代码的第一部分很清楚地表明了传递给命令 U 的地址参数是正确的。命令 U 告诉我们，位于这个地址上的代码是 JIT 生成代码。此外，下一行则告诉我们与这段代码相对应的方法名（这里是 Advanced. NET. Debugging. Chapter3. Breakpoint. AddAndPrint）。最后，第三行是起始地址（003e0178）和生成代码的大小（80）。在这些初始信息之后是反汇编后的指令。现在，我们可以找到动态生成代码的正确地址，并且在这个地址上设置一个断点。

如果恢复程序执行（不要忘记按下任意键来启动第二次调用），可以很快地触发断点，如下所示：

```
0:003> g
 Breakpoint 0 hit
eax=01cdbed8 ebx=0025f0fc ecx=01cdbed8 edx=00000064 esi=01cdbed8 edi=01cdbed8
eip=003f0178 esp=0025f09c ebp=0025f0e0 iopl=0         nv up ei ng nz ac pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000297
003f0178 57              push    edi
0:000> !ClrStack
OS Thread Id: 0xf7c (0)
ESP       EIP
0025f09c 003f0178 Advanced.NET.Debugging.Chapter3.Breakpoint.AddAndPrint(Int32,
Int32)
0025f0a4 003f0105 Advanced.NET.Debugging.Chapter3.Breakpoint.Main(System.String[])
0025f2f0 79e7c74b [GCFrame: 0025f2f0]
```

在触发断点后，可以使用 ClrStack 命令来确保触发的是正确的代码位置。

之前，我们希望在第二次调用 AddAndPrint 时设置一个断点，因为这次的调用中存在一个错误。而现在，我相信你已经知道了为什么要在第二次调用时设置一个断点，就是因为在一个由 JIT 编译的函数上设置断点是很容易的（在前一次调用该函数时会使这个函数被 JIT 编译）。我们可以通过 name2ee 命令来找出 JIT 编译后代码的地址，然后在这个地址上使用 bp 命令。

这是在设置断点时可以使用的另一种方法。SOS 调试器扩展包含了一个命令 bpmd，这个命令能够极大地简化工作，它能自动找出被 JIT 编译后代码的正确地址，并且可以仅根据完整的方法名来设置断点。

接下来，我们来看看如何在还没有被 JIT 编译的函数上设置断点。毕竟，我们不能总在被 JIT 编译后的函数上设置断点，有时候还需要在第一次调用函数时就设置断点，而此时函数通常还没有被编译。

3.6.2　在还没有被 JIT 编译的函数上设置断点

在前面一节中，我们介绍了在已被 JIT 编译的代码上设置断点的两种方法。第一种方法是首先找出被 JIT 编译后的代码地址，然后再通过命令 bp 来设置断点。如果代码已经被 JIT 编译了，那么这种方法是可行的，但如果 JIT 编译器还没有将代码编译为 IL，那么这种方法就不适用。这种情况通常发生在当你希望第一次调用某个函数时就设置断点的时候。由于在这种情况下无法找到被编译代码的地址，那么如何来设置断点？幸运的是，命令 bpmd 可以用来在还没有被 JIT 编译的代码上设置断点。它采取的做法是设置一个延迟断点（deferred breakpoint）。延迟断点也是一种断点，只不过在设置该断点时，断点的位置是未知的，只有在将来某个事件发生时，才会真正地设置断点。bmpd 是通过注册内部的 CLR JIT 编译通知（JIT Compilation Notification）来实现延迟断点的。当收到一个 JIT 编译通知时，它会检查这

个通知是否与现有的某个延迟断点相关，如果相关，那么在函数运行之前就会使断点生效。而且，命令 bpmd 还会接收模块加载通知（module load notification），这意味着在设置断点时甚至可以不需要加载程序集。当程序集被加载时，这个命令会再次得到通知并检查是否有某个延迟断点位于这个模块中，如果有便会激活这个断点。

bpmd 与模块加载通知

在编写本书时，在 bpmd 命令的模块加载通知机制中还存在一个错误，即 bpmd 会错误地接收通知，在这种情况下延迟断点将变成孤立断点（orphaned breakpoint），永远不会被触发。另一种方法是使用 SOSEX，在本章的后面将进一步介绍这个调试器扩展。

我们首先来看看如何通过 bpmd 命令在一个还没有被编译的函数（AddAndPrint）上设置断点，在这里使用的示例程序依然是 03breakpoint. exe。首先在调试器下启动这个程序，并且继续执行直到要求按下任意键（第一个实例函数）。接下来，按 CTRL-C 中断执行。我们要完成的第一个任务就是证明 AddAndPrint 方法还没有被编译。接下来，我们使用在前面介绍的 name2ee 命令来说明这个过程。

```
0:000> !name2ee
03breakpoint.exeAdvanced.NET.Debugging.Chapter3.Breakpoint.AddAndPrint
Module: 00912c24 (03Breakpoint.exe)
Token: 0x06000002
MethodDesc: 00912ff0
Name: Advanced.NET.Debugging.Chapter3.Breakpoint.AddAndPrint(Int32, Int32)
Not JITTED yet. Use !bpmd -md 00912ff0 to break on run.
```

从命令的输出中可以清楚地看到，方法 AddAndPrint 还没有被编译，因此可以继续通过 bpmd 命令和方法描述符来设置一个断点。如果使用 bpmd 来设置断点，可以看到以下信息：

```
0:000> !bpmd -md 00912ff0
MethodDesc = 00912ff0
Adding pending breakpoints...
```

在这里，命令 bpmd 很清楚地告诉我们，它所设置的是一个未决断点（pending breakpoint，也就是延迟断点）。如果恢复程序的执行，那么调试器会收到一个通知，并且在这个方法被调用时将中断执行：

```
0:003> g
 (8d8.b0): CLR notification exception - code e0444143 (first chance)
JITTED 03Breakpoint!Advanced.NET.Debugging.Chapter3.Breakpoint.AddAndPrint(Int32,
Int32)
Setting breakpoint: bp 00CB0138
[Advanced.NET.Debugging.Chapter3.Breakpoint.AddAndPrint(Int32, Int32)]
Breakpoint 0 hit
eax=00912ff0 ebx=0013f4ac ecx=01282e08 edx=0000000a esi=01282e08 edi=01282e08
eip=00cb0138 esp=0013f458 ebp=0013f490 iopl=0         nv up ei pl nz ac po nc
```

```
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000212
00cb0138 57              push    edi
```

需要注意的是 notification 部分的输出，我们从中可以看到，调试器收到了一个 CLR 通知异常（0xe0444143）。此外还可以看到，这个通知是在 JIT 编译器编译 AddAndPrint 方法时发生的，此时通过命令 bp 在代码地址 0x00cb0138 上设置了一个断点。如果对这个地址进行反汇编，那么可以很快地验证它对应于 AddAndPrint 方法。最后发生的事件是断点被触发并且调试器中断执行。最后需要注意的一点是：如果采用了 overloaded 来修饰方法的名字，那么 bpmd 将在所有的重载方法上设置断点。

bp 与之前的 CLR 版本

在 CLR 以前的版本（1.0 和 1.1）中，我们可以通过命令 bp 在一个还没有被 JIT 编译的方法上设置断点。首先找到 JIT 编译器保存已编译代码的地址，并在这个地址上设置一个写入断点，然后恢复程序的执行，直到 JIT 编译器完成编译过程并将编译后的代码保存到这个地址。此时，断点会被触发，并且你可以使用 JIT 编译器保存已编译代码的地址，并在该地址上设置一个断点。这是一个既漫长又费力的过程，幸运的是，在 CLR 2.0 中引入了 bpmd 命令，它能使设置断点的工作变得非常简单。

3.6.3 在预编译的程序集中设置断点

与所有代码一样，.NET 代码也需要在进程的上下文中执行。JIT 编译器将程序集的 IL 编译为机器代码。每当.NET 代码访问同一段代码时，CLR 首先检查它是否已经被编译了，如果是，则重用已编译的代码。当进程结束时，JIT 编译器生成的所有机器代码也将随之消失。当下一次需要调用程序集时，JIT 编译器再重新对相同的代码段进行编译。为了避免重新执行编译过程，CLR 支持预编译（或者叫做 NGEN'd）程序集。预编译程序集是与某个程序集对应的非托管映像，其中的全部代码都已经被转换为机器代码。如果 CLR 需要执行程序集中的某段代码，并且这个程序集在机器有一个非托管映像，那么将跳过 JIT 编译步骤，并直接从这个非托管映像中加载机器代码。虽然这看上去似乎是一个非常大的性能提升，但你可能会担心是否丢失了运行时 JIT 编译的某些优势。答案是，这些好处仍将存在，因为创建预编译程序集是在安装时进行的，而不是在编译时。因此，我们仍然可以获得在架构上运行 JIT 编译器带来的好处。关键区别在于，JIT 编译器把编译的结果保存在一个非托管的映像中，而不是保存在内存中。

如何将非托管映像与.NET 代码的调试关联起来？在对拥有非托管映像的.NET 代码进行调试时，并不存在太多的差异。事实上，有时候这种方式还使得调试工作更加容易，因为你不需要担心函数是否已经被 JIT 编译过。然而，还是有一些事项需要注意。当加载一个非托管映像时，你会看到另一个模块被加载到进程的地址空间中。这个模块是从非托管映像缓存

(native image cache) 中加载的。非托管映像缓存是系统上所有预编译程序集的集合，位置如下：

```
%windir%\assembly\NativeImages_v2.0.50727_<architecture>
```

其中，根据机器比特位数的不同，<architecture>可以是 32 或者 64。每当产生一个非托管映像时，都会放到这个位置中。

NGEN

在某些情况下，预编译程序集也被称为 NGEN'd 程序集。原因在于，生成非托管映像的工具的名字就是 ngen. exe（位于. NET 文件夹下，% windir% \microsoft. net\framework\v2. 0. 50727）。ngen. exe 负责生成非托管映像并将其放在非托管映像缓存中。

在前面的章节中，我们看到了一个示例：在尚未被 JIT 编译的代码上设置一个断点。让我们再来使用这个示例，但这次将首先预编译这个程序集，并且观察是否存在一些差异。使用的示例程序是 03ObjTypes. exe（见清单 3-10）。在使用这个程序之前，首先对其进行预编译：

```
%windir%\microsoft.net\framework\v2.0.50727\ngen install 03ObjTypes.exe
```

在完成预编译后，接下来在调试器下启动这个程序。调试器的初始输出如下所示：

```
Microsoft (R) Windows Debugger Version 6.9.0003.113 X86
Copyright (c) Microsoft Corporation. All rights reserved.

CommandLine: 03ObjTypes.exe
Symbol search path is: *** Invalid ***
*************************************************************************
* Symbol loading may be unreliable without a symbol search path.        *
* Use .symfix to have the debugger choose a symbol path.                *
* After setting your symbol path, use .reload to refresh symbol locations. *
*************************************************************************
Executable search path is:
ModLoad: 010b0000 010b8000   03ObjTypes.exe
ModLoad: 77890000 779b7000   ntdll.dll
ModLoad: 79000000 79046000   C:\Windows\system32\mscoree.dll
ModLoad: 77700000 777db000   C:\Windows\system32\KERNEL32.dll
(858.1138): Break instruction exception - code 80000003 (first chance)
eax=00000000 ebx=00000000 ecx=0020f594 edx=778e9a94 esi=fffffffe edi=778eb6f8
eip=778d7dfe esp=0020f5ac ebp=0020f5dc iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for
ntdll.dll -
ntdll!DbgBreakPoint:
778d7dfe cc              int     3
```

到目前为止，一切都在正常进行。程序集 03ObjTypes. exe 以及一些标准的 Windows 系统

DLL 被加载。如果恢复程序的执行，那么可以看到以下输出：

```
0:000> g
ModLoad: 76160000 76226000   C:\Windows\system32\ADVAPI32.dll
ModLoad: 774e0000 775a3000   C:\Windows\system32\RPCRT4.dll
ModLoad: 77330000 77388000   C:\Windows\system32\SHLWAPI.dll
ModLoad: 76110000 7615b000   C:\Windows\system32\GDI32.dll
ModLoad: 76450000 764ed000   C:\Windows\system32\USER32.dll
ModLoad: 777e0000 7788a000   C:\Windows\system32\msvcrt.dll
ModLoad: 77a10000 77a2e000   C:\Windows\system32\IMM32.DLL
ModLoad: 77260000 77328000   C:\Windows\system32\MSCTF.dll
ModLoad: 771b0000 771b9000   C:\Windows\system32\LPK.DLL
ModLoad: 77460000 774dd000   C:\Windows\system32\USP10.dll
ModLoad: 75110000 752ae000   C:\Windows\WinSxS\x86_microsoft.windows.common-
controls_6595b64144ccf1df_6.0.6001.18000_none_5cdbaa5a083979cc\comctl32.dll
ModLoad: 79e70000 7a3ff000   C:\Windows\Microsoft.NET\Framework\v2.0.50727
\mscorwks.dll
ModLoad: 753b0000 7544b000   C:\Windows\WinSxS\x86_microsoft
.vc80.crt_1fc8b3b9a1e18e3b_8.0.50727.1434_none_d08b6002442c891f\MSVCR80.dll
ModLoad: 766a0000 771af000   C:\Windows\system32\shell32.dll
ModLoad: 775b0000 776f4000   C:\Windows\system32\ole32.dll
ModLoad: 790c0000 79bf6000   C:\Windows\assembly\NativeImages_v2.0.50727_32
\mscorlib\5b3e3b0551bcaa722c27dbb089c431e4\mscorlib.ni.dll
ModLoad: 30000000 30012000   C:\Windows\assembly
\NativeImages_v2.0.50727_32\03ObjTypes.exe\
f322c1140e18febc7214db75604eca91\03ObjTypes.exe.ni.exe
```

正如我们预期的那样，加载了许多模块，包括 CLR 的一些模块，但只有最后一个模块是预料之外的：03ObjTypes. ni. exe。这是一个与 03ObjTypes. exe 应用程序对应的非托管映像（其中 ni 表示 native image），被加载到地址范围 0x30000000 – 0x30012000 中。此时，非托管映像已经被加载，并且我们看到了熟悉的提示“Press any key to continue（AddCoordinate）”。手动中断程序的执行，并判断 AddCoordinate 方法是否已经被编译。记得在之前的过程中，AddCoordinate 方法还没有被编译。

```
0:004> !name2ee 03ObjTypes.exe
Advanced.NET.Debugging.Chapter3.ObjTypes.AddCoordinate
Module: 30004000 (03ObjTypes.exe)
Token: 0x06000002
MethodDesc: 30004820
Name: Advanced.NET.Debugging.Chapter3.ObjTypes.AddCoordinate(Coordinate)
JITTED Code Address: 300025f8
```

这时，我们可以看到 AddCoordinate 方法已经被编译了（这与我们的预期相符，因为这个程序集已经被预编译了），并且这段代码位于地址 0x300025f8 处。这个地址位于预编译程序集的模块范围之内（0x30000000 – 0x30012000）。

3.6.4 在泛型方法上设置断点

泛型（generics）是在 CLR 中引入的一种机制，它使得开发人员可以将泛型性（generality）增加到类中。在引入泛型之前，要实现泛型性的一种方法就是，将所有类型都作为 Object（这是.NET 中所有类的基类），并且在必要时进行类型转换。虽然这种方法确实可行，但也存在一些缺点。缺点之一就是，这种方法将不同的类型都视为基类 Object，并且通过类型转换来具体化（specialize）类型。在这些过程中将丢失大量的编译时类型安全性。此外，如果使用了值类型，那么还会对性能造成一定的损失，因为每当将值类型转换为 Object（引用）类型时，都需要执行装箱（boxing）和拆箱（unboxing）等操作。泛型为开发人员提供了一种方式，在不牺牲类型安全性和性能的情况下就可以实现泛型性。在本章的这部分内容中，我们将分析在调试时，尤其在泛型类型和方法上设置断点时，会采取怎样的区别对待。我们将通过 03ObjTypes.exe 来说明如何使用 bpmd 命令在泛型类型上设置断点。

首先在调试器下运行 03ObjTypes.exe，直到出现以下提示：

```
Press any key to continue (Generics)
```

此时，手动中断进入到调试器，加载 SOS 调试器扩展，并且使用 bmpd 命令在 GreaterThan 方法（位于泛型类型 Comparer 中）上设置一个断点。调试会话如下所示：

```
Press any key to continue (Generics)
(a54.1fa4): Break instruction exception - code 80000003 (first chance)
eax=7ffdb000 ebx=00000000 ecx=00000000 edx=7791d094 esi=00000000 edi=00000000
eip=778d7dfe esp=040ef894 ebp=040ef8c0 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
ntdll!DbgBreakPoint:
778d7dfe cc              int     3
0:004> .loadby sos mscorwks
0:004> !bpmd 03ObjTypes.exe Advanced.NET.Debugging.Chapter3.Comparer`1.GreaterThan
Found 1 methods...
MethodDesc = 001a3188
Adding pending breakpoints...
```

你可以看到，这个过程非常类似于在非泛型的类型上设置断点的过程，只是有一点需要注意：在完整的方法名字上增加了“`1”。要通过 bpmd 命令在泛型类型上设置一个断点，需要知道正在设置的断点是哪种类型。具体来说，它需要知道所声明的泛型类型中的泛型成员的数量有多少。当多个泛型类型有着相同的名字但却不同数量的泛型成员时，这一点非常重要。这种情况下，泛型类型 Comparer 类的声明如下：

```
public class Comparer<T> where T: IComparable
{
    ...
}
```

我们可以看到，在 Comparer 类的实例中只有一个泛型类型需要具现化（instantiate）。如果存在其他也叫做 Comparer 的泛型类，并且它们包含两个具现化的泛型类型，那么可以使用下面这条命令来设置断点：

```
!bpmd 03ObjTypes.exe Advanced.NET.Debugging.Chapter3.Comparer`2.GreaterThan
```

3.7　对象检查

在大多数调试会话中，首先需要执行的任务之一就是分析应用程序的状态，以便确认程序的故障是由于某种无效状态造成的。通常，我们无法重新启动程序并且在跟踪问题时每次执行一行代码，这使得要找出问题的根源非常困难。在确认程序处于某种无效状态后，便可以开始分析程序是如何达到这种状态的。在本章的这部分介绍了一些命令，可以用来分析程序的状态。我们首先详细介绍在非托管调试器中最常用的一些命令，然后再介绍在 SOS 调试器扩展中针对托管代码调试的命令。再次以 03ObjTypes. exe 为例，如清单 3-10 所示。

清单 3-10　说明对象检查的示例程序

```
using System;
using System.Text;

namespace Advanced.NET.Debugging.Chapter3
{
    public class ObjTypes
    {
        public struct Coordinate
        {
            public int xCord;
            public int yCord;
            public int zCord;

            public Coordinate(int x, int y, int z)
            {
                xCord = x;
                yCord = y;
                zCord = z;
            }
        }
    private Coordinate coordinate;

    int[] intArray = new int[] { 1, 2, 3, 4, 5 };
    string[] strArray = new string[] {"Welcome",
                                      "to",
                                      "Advanced",
                                      ".NET",
                                      "Debugging"};
```

```
static void Main(string[] args)
{
    Coordinate point= new Coordinate(100, 100, 100);
    Console.WriteLine("Press any key to continue (AddCoordinate)");
    Console.ReadKey();
    ObjTypes ob = new ObjTypes();
    ob.AddCoordinate(point);

    Console.WriteLine("Press any key to continue (Arrays)");
    Console.ReadKey();
    ob.PrintArrays();

    Console.WriteLine("Press any key to continue (Generics)");
    Console.ReadKey();
    Comparer<int> c = new Comparer<int>();
    Console.WriteLine("Greater {0}", c.GreaterThan(5, 10));

    Console.WriteLine("Press any key to continue (Exception)");
    Console.ReadKey();
    ob.ThrowException(null);
}

public void AddCoordinate(Coordinate coord)
{
    coordinate.xCord += coord.xCord;
    coordinate.yCord += coord.yCord;
    coordinate.zCord += coord.zCord;

    Console.WriteLine("x:{0}, y:{1}, z:{2}",
                      coordinate.xCord,
                      coordinate.yCord,
                      coordinate.xCord);
 }
 public void PrintArrays()
 {
     foreach (int i in intArray)
     {
         Console.WriteLine("Int: {0}", i);
     }
     foreach (string s in strArray)
     {
         Console.WriteLine("Str: {0}", s);
     }
 }

 public void ThrowException(ObjTypes obj)
 {
     if (obj == null)
     {
         throw new System.ArgumentException("Obj cannot be null");
     }
 }
```

```
        }


        public class Comparer<T> where T: IComparable
        {
            public T GreaterThan(T d, T d2)
            {
                int ret = d.CompareTo(d2);
                if (ret > 0)
                    return d;
                else
                    return d2;
            }

            public T LessThan(T d, T d2)
            {
               int ret = d.CompareTo(d2);
               if (ret < 0)
                  return d;
            else
                return d2;
        }
    }
}
```

清单 3-10 中的源代码文件和二进制文件位于以下文件夹中：

- 源代码文件：C:\ADND\Chapter3\ObjTypes
- 二进制文件：C:\ADNDBin\03ObjTypes.exe

3.7.1 内存转储

在调试器中有许多命令都可以用来转储内存的内容。最常使用的命令是 d（显示内存）。命令 d 的最简单形式只带有一个参数，表示需要显示的内存地址。命令 d 能够以非常简洁的形式显示需要查看的内存，如下所示：

```
0:000> dd 0x01e06bec
01e06bec  001c30b0 00000000 00000000 00000000
01e06bfc  80000000 790fd8c4 00000014 00000013
01e06c0c  003a0078 0030007b 002c007d 00790020
01e06c1c  007b003a 007d0031 0020002c 003a007a
01e06c2c  0032007b 0000007d 00000000 00000000
01e06c3c  00000000 00000000 00000000 00000000
01e06c4c  00000000 00000000 00000000 00000000
01e06c5c  00000000 00000000 00000000 00000000
```

最左边的一列给出了每行内存的起始地址，后面是内存的内容。根据需要转储的数据类型的不同，命令 d 也有许多不同的变化形式。例如，命令 du 会把被转储的内存视作为 Uni-

code 字符：

```
0:000> du 7ffdfc00
7ffdfc00  "ADVAPI32.dll"
```

其他一些变化形式包括：

- da 把被转储的内存视作为 ASCII 字符。
- dw 把被转储的内存视作为字（word）。
- db 把被转储的内存视作为字节值和 ASCII 字符。
- dq 把被转储的内存视作为四字（quad word）值。

虽然，有时候直接把内存内容转储出来会很有用，但当调试托管代码时，如果能将 CLR 对象以一种更为结构化和更易于阅读的形式转储出来，会更为有用。例如，如果使用命令 d 来转储以下的内存，将得到以下输出：

```
0:000> dd 0x01c56bec
01c56bec  002130b0 00000000 00000000 00000000
01c56bfc  80000000 790fd8c4 00000014 00000013
01c56c0c  003a0078 0030007b 002c007d 00790020
01c56c1c  007b003a 007d0031 0020002c 003a007a
01c56c2c  0032007b 0000007d 00000000 00000000
01c56c3c  00000000 00000000 00000000 00000000
01c56c4c  00000000 00000000 00000000 00000000
01c56c5c  00000000 00000000 00000000 00000000
```

我们很难推断出这些内存表示的实际内容。现在，请看下面对同一地址转储出来的更多信息。

```
0:000> !dumpobj 0x01c56bec
Name: Advanced.NET.Debugging.Chapter3.ObjTypes
MethodTable: 002130b0
EEClass: 00211240
Size: 20(0x14) bytes
 (C:\ADNDBin\03ObjTypes.exe)
Fields:
      MT    Field   Offset                 Type VT     Attr    Value Name
0021306c  4000001        4 ...jTypes+Coordinate  1 instance 01c56bf0 coordinate
```

这时在输出中包含了更多的信息，例如它告诉我们这个地址表示一个 CLR 实例，类型为 Advanced. NET. Debugging. Chapter3. ObjTypes。此外，还给出了这个实例的一些基本信息，以及在实例中包含的各个域。在本书中，我们将直接内存转储技术与友好的转储命令结合起来使用。

在深入研究. NET 类型的转储命令之前，先来介绍 CLR 类型系统中的各种不同类型。共两种基本类型，分别为值类型和引用类型。值类型是从 System. ValueType 中派生下来的类型，通常在栈上分配。如果在引用类型（例如一个类）中包含了一个值类型，那么值类型会作为该引用类型的一部分在托管堆上分配。在 C#中，struct 类型就是一种值类型。引用类型

是从 System. Object 而不是 System. ValueType 中派生下来的类型（无论是直接派生还是间接派生的），并且通常是在托管堆上分配的。为什么要设计两种不同的类型？主要是为了提升程序的性能。在栈上分配小型的对象比在托管堆上分配它们要更高效。（我们将在第 5 章中讨论托管堆）。

我们知道,.NET 中所有类型的对象都是从 System. Object 中派生出来的，包括值类型。这意味着我们有一个一致的类型系统，其中所有对象都是同一个顶级父类中派生出来的。例如，这种一致性使我们能够编写像下面这样的代码：

```
int i = 10;
object o = i;                    // 隐式装箱
object oo = (object)i;           // 显式装箱
int ii = (int) oo;               // 显式拆箱
```

在这段代码中，我们把值类型 i 赋值给引用类型 o 和 oo。前面已经介绍过，值类型被分配在栈上，而引用类型则被分配在托管堆上。因此，如何把一个在栈上分配的值赋给一个在托管堆上分配的对象？答案是装箱操作。当把值类型赋给引用类型时，CLR 将自动执行以下任务：

1）在托管堆上分配内存。

2）将值类型的内存（从栈上）复制到托管堆上新分配的引用类型中。

与装箱相反的操作是拆箱，这个操作能将引用类型赋值给值类型。严格来说，拆箱操作的开销小于装箱操作的开销。因为当执行拆箱操作时，CLR 并不会复制被装箱的值类型的内容，而是返回一个托管指针，指向包含在引用类型中的值类型实例。然而，在赋值操作中同样也包含拆箱操作，从而需要将一个被装箱的值类型复制到栈上，这个操作也会带来一定的开销。

虽然这些操作对用户来说是透明的，但了解这些操作的内部机制是很重要的，因为它们可能对调试会话产生影响（并且，当存在大量的装箱和拆箱等操作时，会对性能产生影响）。

让我们开始介绍类型的转储过程，首先来看值类型。

3.7.2 值类型的转储

在讨论可用于对值类型进行转储的命令时，首先要理解值类型与引用类型之间的差异。假定有一个对象位于地址 0x00846710 上，我们如何知道这个地址上的对象是一个值类型还是一个引用类型？要判断一个指针指向的是否是值类型，最佳方式就是使用 DumpObj 命令，但它只对引用类型有效。DumpObj 命令的参数是一个指向引用类型的指针，能输出这个引用类型的内容。如果给定的指针指向一个值类型，那么 DumpObj 命令会给出以下错误提示：

```
0:000> !DumpObj 0x002bf0b4
<Note: this object has an invalid CLASS field>
Invalid object
```

在前面已经提到过，如果一个值类型没有作为某个引用类型的一部分（例如，在某个函数中声明一个局部变量或者一个值类型），那么它会被分配在栈上，并且可以使用前面介绍的 d 系列命令来转储值类型的内容。以 03ObjTypes. exe 程序为例，在调试器下运行它并且当提示按下任意键时手动中断程序的执行，如清单 3-11 所示。

清单 3-11　中断进入到 03ObjTypes. exe

```
0:000> .symfix
No downstream store given, using c:\Program Files\Debugging Tools for Windows
(x86)\sym
0:000> .reload
Reloading current modules
....
0:000> g
ModLoad: 763c0000 76486000   C:\Windows\system32\ADVAPI32.dll
ModLoad: 77260000 77323000   C:\Windows\system32\RPCRT4.dll
ModLoad: 77980000 779d8000   C:\Windows\system32\SHLWAPI.dll
ModLoad: 76570000 765bb000   C:\Windows\system32\GDI32.dll
ModLoad: 764d0000 7656d000   C:\Windows\system32\USER32.dll
ModLoad: 77330000 773da000   C:\Windows\system32\msvcrt.dll
ModLoad: 764a0000 764be000   C:\Windows\system32\IMM32.DLL
ModLoad: 77cd0000 77d98000   C:\Windows\system32\MSCTF.dll
ModLoad: 76490000 76499000   C:\Windows\system32\LPK.DLL
ModLoad: 779e0000 77a5d000   C:\Windows\system32\USP10.dll
ModLoad: 753d0000 7556e000   C:\Windows\WinSxS\x86_microsoft.windows.common-
controls_6595b64144ccf1df_6.0.6001.18000_none_5cdbaa5a083979cc\comctl32.dll
ModLoad: 79e70000 7a3ff000   C:\Windows\Microsoft.NET\Framework\v2.0.50727
\mscorwks.dll
ModLoad: 75670000 7570b000   C:\Windows\WinSxS\x86_microsoft
.vc80.crt_1fc8b3b9a1e18e3b_8.0.50727.1434_none_d08b6002442c891f\MSVCR80.dll
ModLoad: 76750000 7725f000   C:\Windows\system32\shell32.dll
ModLoad: 775f0000 77734000   C:\Windows\system32\ole32.dll
ModLoad: 790c0000 79bf6000   C:\Windows\assembly\NativeImages_v2.0.50727_32
\mscorlib\5b3e3b0551bcaa722c27dbb089c431e4\mscorlib.ni.dll
ModLoad: 79060000 790b6000   C:\Windows\Microsoft.NET\Framework\v2.0.50727
\mscorjit.dll
Press any key to continue
(1758.12b4): Break instruction exception - code 80000003 (first chance)
eax=7ffda000 ebx=00000000 ecx=00000000 edx=77bcd094 esi=00000000 edi=00000000
eip=77b87dfe esp=0419fed0 ebp=0419fefc iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
ntdll!DbgBreakPoint:
77b87dfe cc              int     3
0:004> .loadby sos mscorwks
```

首先，我们希望显示出托管调用栈以及相关的局部变量。可以通过 ClrStack 命令来获取这些信息，如下所示：

```
0:004> !ClrStack -a
OS Thread Id: 0x12b4 (4)
Unable to walk the managed stack. The current thread is likely not a
managed thread. You can run !threads to get a list of managed threads in the process
0:004>
```

ClrStack 命令提示了一个错误，指出当前的线程上下文并不是一个有效的托管线程。由于我们手动地中断程序的执行，调试器的线程上下文是在调试器线程上，而这是一个非托管线程，因此在运行 ClrStack 命令之前，必须首先切换到托管线程上下文。使用 ~ 命令将上下文切换到线程 0，然后再次执行 ClrStack 命令。

```
0:004> ~0s
eax=0012ed24 ebx=0012ed1c ecx=792274ec edx=79ec9058 esi=0012eb78 edi=00000000
eip=77b99a94 esp=0012eb28 ebp=0012eb48 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
ntdll!KiFastSystemCallRet:
77b99a94 c3              ret
0:000> !ClrStack -a
OS Thread Id: 0x10b4 (0)
ESP       EIP
0012ecf4 77b99a94 [NDirectMethodFrameSlim: 0012ecf4]
Microsoft.Win32.Win32Native.ReadConsoleInput
(IntPtr, InputRecord ByRef, Int32, Int32 ByRef)
0012ed0c 793e8f28 System.Console.ReadKey(Boolean)
    PARAMETERS:
        intercept = 0x00000000
    LOCALS:
        <no data>
        0x0012ed1c = 0x00000001
        <no data>
        <no data>
        <no data>
        <no data>
        <no data>
        <no data>
        <no data>
        <no data>

0012ed4c 793e8e33 System.Console.ReadKey()
0012ed50 002c00c1 Advanced.NET.Debugging.Chapter3.ObjTypes.Main(System.String[])
    PARAMETERS:
        args = 0x01b958ac
    LOCALS:
        0x0012ed54 = 0x00000064
        <CLR reg> = 0x00000000

0012ef94 79e7c74b [GCFrame: 0012ef94]
```

这时 ClrStack 命令给出了托管线程的栈回溯，包括每个栈帧的局部变量和参数。在调用

栈中，需要注意的栈帧是 Main 栈帧和位于地址 0x0012ed54 上的局部变量。由于我们不知道这个局部变量指向的是引用类型还是值类型，因此需要使用 DumpObj 命令来判断它是否能正确地将这个地址识别为某个指向托管堆中对象的引用：

```
0:000> !DumpObj 0x0012ed54
<Note: this object has an invalid CLASS field>
Invalid object
```

在 DumpObj 命令的输出结果中可以很清楚地看到，这个地址并非对应一个引用类型，因此我们可以假设这个地址是一个值类型。为了进一步验证它，来观察这个地址本身。记住，值类型被保存在栈上，如果发现某个地址位于当前栈指针的附近，那么就可以很好地证明这个地址上的对象是一个值类型。通过命令 r 将寄存器转储出来，观察 esp 寄存器（其中保存的是当前栈指针）：

```
0:000> r
eax=0012ed24 ebx=0012ed1c ecx=792274ec edx=79ec9058 esi=0012eb78 edi=00000000
eip=77b99a94 esp=0012eb28 ebp=0012eb48 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
ntdll!KiFastSystemCallRet:
77b99a94 c3              ret
```

在 esp 寄存器中包含的值是 0x0012eb28，这非常接近我们正在分析的地址：0x0012ed54。此时，我们可以得出结论：正在分析的地址事实上是栈上的一个地址，并且很可能是一个值类型。现在，可以使用命令 dd 来显示这个值类型的内容：

```
0:000> dd 0x0012ed54
0012ed54  00000064 00000064 00000064 0012ed9c
0012ed64  0012ed80 79e73560 00505408 00000000
0012ed74  79e7c74b 00000003 0012ed84 0012ee00
0012ed84  79e7c6cc 0012ee50 00000000 0012ee20
0012ed94  00000000 0022c040 0012edf0 79f07fee
0012eda4  0012ef94 7cc80b79 0012efe0 0012ee40
0012edb4  00000000 0012ef94 00505408 00000000
0012edc4  00000000 00000000 00000000 00000001
```

在内存转储结果中，前三个值为 0x64（100），分别对应于值类型 Coordinate 中的各个域。

到目前为止，我们已经看到了如何显示函数中声明的值类型的内容。通常，值类型被嵌入在引用类型中并且被保存在托管堆上。在这种情况下，我们不能使用直接内存转储命令，而是需要借助一些辅助命令来转储值类型。本过程的第一步就是通过 DumpObj 命令转储出引用类型本身的内容。再次使用 03ObjTypes. exe 示例。首先在调试器下运行这个应用程序，然后当提示按下任意键时手动中断执行。加载 SOS 调试器扩展，在 AddCoordinate 函数上设置一个断点，并且恢复程序执行直到触发这个断点。清单 3-12 给出了调试器会话过程。

清单 3-12 触发 03ObjTypes.exe 示例中 AddCoordinates 函数的断点

```
0:004> .loadby sos mscorwks
0:000> !bpmd 03ObjTypes.exe Advanced.NET.Debugging.Chapter3.ObjTypes.AddCoordinate
Found 1 methods...
MethodDesc = 000d3038
Adding pending breakpoints...
0:000> g
(232c.1d08): Control-C exception - code 40010005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=040dfdd4 ebx=00000000 ecx=00000000 edx=77b99a94 esi=00000000 edi=00000002
eip=77886da1 esp=040dfdc4 ebp=040dfe48 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
KERNEL32!CtrlRoutine+0xbf:
77886da1 c745fcfeffffff  mov     dword ptr [ebp-4],0FFFFFFFEh
ss:0023:040dfe44=00000000
0:003>
0:003> g
g(232c.209c): CLR notification exception - code e0444143 (first chance)
JITTED 03ObjTypes.exe!Advanced.NET.Debugging.Chapter3.
ObjTypes.AddCoordinate(Coordinate)
Setting breakpoint: bp 002A0180
[Advanced.NET.Debugging.Chapter3.ObjTypes.AddCoordinate(Coordinate)]
Breakpoint 0 hit
eax=000d3038 ebx=0040ef9c ecx=01de6bec edx=001b5408 esi=01de6bec edi=01de6bec
eip=002a0180 esp=0040ef40 ebp=0040ef80 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
002a0180 57              push    edi
0:000> !ClrStack -a
OS Thread Id: 0x209c (0)
ESP       EIP
0040ef40 002a0180
Advanced.NET.Debugging.Chapter3.ObjTypes.AddCoordinate(Coordinate)
    PARAMETERS:
        this = 0x01de6bec
        coord = 0x00000064

0040ef50 002a00f5 Advanced.NET.Debugging.Chapter3.ObjTypes.Main(System.String[])
    PARAMETERS:
        args = 0x01de58ac
    LOCALS:
        0x0040ef54 = 0x00000064
        <CLR reg> = 0x01de6bec

0040f18c 79e7c74b [GCFrame: 0040f18c]
```

在 ClrStack 命令的输出中，AddCoordinate 栈帧显示了一个 this 参数。this 指针指向当前的对象实例，类似于将非托管代码中当前实例的指针传递给类中的一个函数。可以在 this 指

针上使用 DumpObj 命令，从而显示出 ObjTypes 类的内容：

```
0:000> !DumpObj 0x01de6bec
Name: Advanced.NET.Debugging.Chapter3.ObjTypes
MethodTable: 000d30d0
EEClass: 000d1244
Size: 20(0x14) bytes
 (C:\ADNDBin\03ObjTypes.exe)
Fields:
      MT    Field   Offset                 Type VT     Attr    Value Name
000d3088  4000001        4 ...jTypes+Coordinate  1 instance 01de6bf0 coordinate
```

我们将在下一节更详细地讨论 DumpObj 的输出信息。就目前来说，最重要的是 Fields，它里面包含了这个对象的元数据。我们可以看到，在这个对象类型中只有一个域。在表 3-1 中详细说明了其中每一列的含义。

表 3-1 DumpObj 输出中各个域的含义

MT	这个域的方法表
Field	这个域的元数据。高顺序位（4）表示它是一个域，而低顺序位（1）是在元数据表中的偏移
Offset	这个域在引用类型内存布局中的偏移
Type	这个类型的缩写名字
VT	如果被设置为 1，那么表示这是一个值类型，如果为 0，则表示是一个引用类型
Attr	对象的属性。在前面的示例中，我们可以看到这个对象是一个实例
Value	域的值
Name	域的名字（在这里是成员域 coordinate 的名字）

如果要显示出这个域本身的内容，有两种选择。其中一种是使用 dd 命令，参数分别为引用类型对象的地址（0x01de6bec）和偏移（0x4），这个命令将转储出值类型的内容。

```
0:000> dd 0x01de6bec+0x4
01de6bf0  00000000 00000000 00000000 80000000
01de6c00  790fd8c4 00000014 00000013 003a0078
01de6c10  0030007b 002c007d 00790020 007b003a
01de6c20  007d0031 0020002c 003a007a 0032007b
01de6c30  0000007d 00000000 00000000 00000000
01de6c40  00000000 00000000 00000000 00000000
01de6c50  00000000 00000000 00000000 00000000
01de6c60  00000000 00000000 00000000 00000000
```

从上面的输出中可以很清楚地看到值类型内容。另一种更方便的方式是使用 DumpVC 命令，它能将引用类型中的值类型转储出来。DumpVC 命令的唯一作用就是显示值类型，它的形式如下所示：

```
!DumpVC <MethodTable address> <Address>
```

从 DumpObj 命令的输出中可以很容易地得到“Method Table Address”和“Address”。在前面的调试输出中，方法表被设置为 0x000d3088，地址为 0x01de6bf0（对应于 DumpObj 命

令 Fields 输出中的 Value 列）。DumpVC 命令的输出如下所示：

```
0:000> !DumpVC 0x000d3088  0x01de6bf0
Name: Advanced.NET.Debugging.Chapter3.ObjTypes+Coordinate
MethodTable 000d3088
EEClass: 000d12a8
Size: 20(0x14) bytes
 (C:\ADNDBin\03ObjTypes.exe)
Fields:
      MT    Field   Offset                 Type VT     Attr    Value Name
79102290  4000002        0         System.Int32  1 instance        0 xCord
79102290  4000003        4         System.Int32  1 instance        0 yCord
79102290  4000004        8         System.Int32  1 instance        0 zCord
```

可以看到，与命令 dd 对内存直接转储得到的信息相比，这个命令输出了更为详细的信息。除了显示值类型的所有成员外（以及相关的信息），还给出了一些基本的信息，例如值类型的名字和大小。最需要注意的是 Fields，它非常像 DumpObj 命令，列出了值类型中每个域的所有属性。我们可以看到这三个域都是值类型（因为它们都是 Int32 类型）。接下来可以再次使用 DumpVC 命令转储出各个域的内容。

到这里就结束了对值类型转储的讨论。我们介绍了如何转储出“单独的”值类型，以及如何转储出“嵌套”在引用类型中的值类型。接下来，我们将介绍如何转储引用类型。

3.7.3 转储基本的引用类型

在前面的章节中，我们已经看到了使用 DumpObj 命令可以转储出引用类型。DumpObj 命令的形式如下所示：

```
!DumpObj [-nofields] <object address>
```

其中 object address 是转储的对象地址。在默认情况下，DumpObj 会转储出类型的信息及其相关的域。如果只需要一般性的信息，那么可以使用-nofields 选项，它不输出域信息。

让我们进一步来观察 DumpObj 命令的输出。

```
0:000> !DumpObj 0x01de6bec
Name: Advanced.NET.Debugging.Chapter3.ObjTypes
MethodTable: 000d30d0
EEClass: 000d1244
Size: 20(0x14) bytes
 (C:\ADNDBin\03ObjTypes.exe)
Fields:
      MT    Field   Offset                 Type VT     Attr    Value Name
000d3088  4000001        4 ...jTypes+Coordinate  1 instance 01de6bf0 coordinate
```

DumpObj 命令给出的第一部分信息是类型的名字，即这里的 ObjTypes。接下来，它输出了类型方法表（MethodTable）和 EEClass 指针，以及大小（size）和类型定义所在的程序集。在表 3-1 中详细说明了域的输出中每一列的含义。由于每个域都可以是指向其他引用类型实

例的引用，因此 DumpObj 也可以用于显示被包含域的信息。

最后提示一点，DumpObj 命令可以通过缩写形式 do 来调用。

3.7.4 数组的转储

CLR 将数组作为第一级（first class）引用类型（所有数组都是从 System. Array 类型中派生下来的），因此可以使用 DumpObj 命令直接转储出数组的内容。再次以 03ObjTypes. exe 程序为例来说明数组的转储过程。首先在调试器下运行 03ObjTypes. exe，继续执行直到出现提示“Press any key to continue（Arrays）”。此时，手动中断执行并执行 ClrStack-a 命令，然后从这里开始讨论：

```
0:000> !ClrStack -a
OS Thread Id: 0xff8 (0)
ESP       EIP
001af3a8 77b99a94 [NDirectMethodFrameSlim: 001af3a8]
Microsoft.Win32.Win32Native.ReadConsoleInput(IntPtr, InputRecord ByRef, Int32, Int32
ByRef)
001af3c0 793e8f28 System.Console.ReadKey(Boolean)
    PARAMETERS:
        intercept = 0x00000000
    LOCALS:
        <no data>
        0x001af3d0 = 0x00000001
        <no data>
        <no data>
        <no data>
        <no data>
        <no data>
        <no data>
        <no data>

001af400 793e8e33 System.Console.ReadKey()
001af404 0097010c Advanced.NET.Debugging.Chapter3.ObjTypes.Main(System.String[])
    PARAMETERS:
        args = 0x01c758ac
    LOCALS:
        0x001af408 = 0x00000064
        <CLR reg> = 0x01c76c84

001af64c 79e7c74b [GCFrame: 001af64c]
0:000>
```

需要注意的栈帧是 Main 函数帧，其中列出了一个 < CLR reg > 指针值为 0x01c76c84。如果在这个指针上执行 DumpObj（见清单 3-13），可以很快地看到它对应于一个 ObjTypes 类型的指针，这个类型刚好定义了两个数组域。

清单 3-13 转储 ObjTypes 类型

```
0:000> !DumpObj 0x01c76c84
Name: Advanced.NET.Debugging.Chapter3.ObjTypes
MethodTable: 002a3114
EEClass: 002a129c
Size: 28(0x1c) bytes
 (C:\ADNDBin\03ObjTypes.exe)
Fields:
      MT    Field   Offset                 Type VT     Attr    Value Name
002a30c0  4000001        c ...jTypes+Coordinate  1 instance  01c76c90 coordinate
7912d7c0  4000002        4      System.Int32[]  0 instance 01c76d3c intArray
7912d8f8  4000003        8     System.Object[]  0 instance 01c76d5c strArray
```

从输出中可以看到，第一个数组域（intArray）的类型为 System. Int32 [ ]，而第二个数组域（strArray）的类型是 System. Object [ ]。第一个数组是整数数组，而第二个数组是对象数组。根据前面在转储基本引用类型中进行的讨论，我们可以在每个引用类型域（例如我们的数组域）上使用 DumpObj 命令来获得每个域上的进一步信息。在整数数组上运行 DumpObj 命令会产生以下输出：

```
0:000> !DumpObj 01c76d3c
Name: System.Int32[]
MethodTable: 7912d7c0
EEClass: 7912d878
Size: 32(0x20) bytes
Array: Rank 1, Number of elements 5, Type Int32
Element Type: System.Int32
Fields:
None
```

虽然 DumpObj 命令输出了一些有用的信息，例如数组维数、元素的数量、每个元素的类型，但却没有给出在每个数组位置上实际包含的内容。要获得这个信息，可以对数组指针本身使用直接转储命令，例如 dd，如清单 3-14 所示。

清单 3-14 对值类型数组进行直接内存转储

```
0:000> dd 01c76d3c
01c76d3c  7912d7c0 00000005 00000001 00000002
01c76d4c  00000003 00000004 00000005 00000000
01c76d5c  7912d8f8 00000005 790fd8c4 01c76ca0
01c76d6c  01c76cc0 01c76cd8 01c76cfc 01c76d18
01c76d7c  80000000 790fd8c4 00000014 00000013
01c76d8c  003a0078 0030007b 002c007d 00790020
01c76d9c  007b003a 007d0031 0020002c 003a007a
01c76dac  0032007b 0000007d 00000000 79102290
```

在任何数组的内存输出中，第一个值都是数组类型本身的方法表。将这个方法表的指针

传递给 DumpMT 命令，会显示关于这个类型的更多信息，如下所示：

```
0:000> !DumpMT 7912d7c0
EEClass: 7912d878
Module: 790c2000
Name: System.Int32[]
mdToken: 02000000  (C:\Windows\assembly\GAC_32\mscorlib\2.0.0.0__
b77a5c561934e089\mscorlib.dll)
BaseSize: 0xc
ComponentSize: 0x4
Number of IFaces in IFaceMap: 4
```

输出信息告诉我们，这个类型是一个整数数组（int32），但仍然没有显示出数组的所有元素。在清单 3-14 中，还要注意的值就是紧接着方法表指针的值，也就是数组本身的大小。在清单 3-14 中，我们可以看到数组的大小为 5。在数组大小后面是各个元素（1、2、3、4、5）。这就是显示值类型数组的完整过程。这里的关键点在于值类型数组的内存布局，如图 3-2 所示。

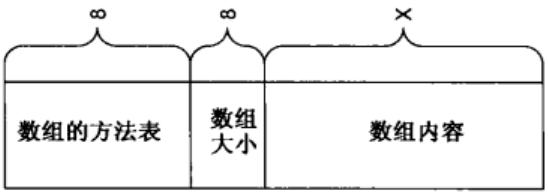


图 3-2　值类型数组的内存布局

如何转储引用类型数组？能否使用相同的方法？当然可以。然而，内存布局的关键区别在于，除了数组本身的方法表之外，还有一个方法表指针对应于数组包含元素的类型。在图 3-3 中说明了引用类型数组的内存布局。

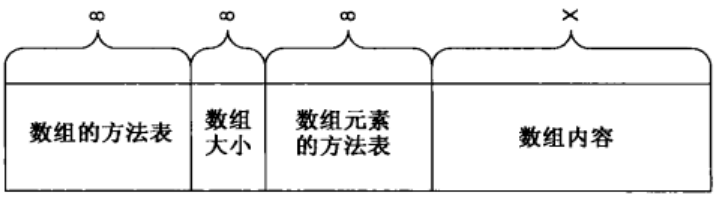


图 3-3　引用类型数组的内存布局

让我们继续之前的调试会话，并且观察如何转储 ObjTypes 类中字符数组的内容。从清单 3-14 中可以看到，指向字符数组的指针为 0x01c76d5c。在这个指针上使用命令 dd 会产生以下输出：

```
0:000> dd 01c76d5c
01c76d5c  7912d8f8 00000005 790fd8c4 01c76ca0
01c76d6c  01c76cc0 01c76cd8 01c76cfc 01c76d18
01c76d7c  80000000 790fd8c4 00000014 00000013
01c76d8c  003a0078 0030007b 002c007d 00790020
01c76d9c  007b003a 007d0031 0020002c 003a007a
01c76dac  0032007b 0000007d 00000000 79102290
01c76dbc  00000064 00000000 79102290 00000064
01c76dcc  00000000 79102290 00000064 00000000
```

我们已经知道第一个值（0x7912d8f8）对应于 Ssytem. Object［］的方法描述符，第二个值则是数组中元素的数量（5），第三个值对应于数组本身中元素的方法描述符。如果使用 DumpMT 命令，那么可以看到以下输出：

```
0:000> !DumpMT 790fd8c4
EEClass: 790fd824
Module: 790c2000
Name: System.String
mdToken: 02000024  (C:\Windows\assembly\GAC_32\mscorlib\2.0.0.0__
b77a5c561934e089\mscorlib.dll)
BaseSize: 0x10
ComponentSize: 0x2
Number of IFaces in IFaceMap: 7
Slots in VTable: 194
```

从输出信息中可以很清楚地看到，正在分析的是一个字符串数组。剩余的 5 个值分别指向数组中各个字符串的引用，可以在各个引用上使用 DumpObj 命令来输出字符串的值：

```
0:000> !DumpObj -nofields 01c76ca0
Name: System.String
MethodTable: 790fd8c4
EEClass: 790fd824
Size: 32(0x20) bytes
 (C:\Windows\assembly\GAC_32\mscorlib\2.0.0.0__b77a5c561934e089\mscorlib.dll)
String: Welcome
0:000> !DumpObj -nofields 01c76cc0
Name: System.String
MethodTable: 790fd8c4
EEClass: 790fd824
Size: 22(0x16) bytes
 (C:\Windows\assembly\GAC_32\mscorlib\2.0.0.0__b77a5c561934e089\mscorlib.dll)
String: to
...
...
```

注意，由于我们只对字符串值本身感兴趣，因此使用了 -nofields 选项，这个选项不会显示字符串类型中的各个域。

到这里，我们已经说明了如何转储值类型的数组和引用类型的数组。对于各种类型的数组，我们采用的方式都是直接转储内存的内容以及在引用上使用 DumpObj 命令（对于引用数组），这是个有些重复的过程。还有一个命令可以将转储数组的过程自动化：DumpArray。DumpArray 命令的常用形式如下所示：

```
!DumpArray
        [-start <startIndex>]
        [-length <length>]
        [-details]
        [-nofields]
        <array object address>
```

-start 选项能控制从数组中的哪个索引开始，而 - length 选项则控制显示数组中多少个元素。例如，如果希望转储出位于地址 X 上的数组的前三个元素，并从索引 2 开始，那么可以使用以下命令行：

```
!DumpArray -start 2 -length 3 X
```

如果增加 - details 选项，DumpArray 会输出更为详细的信息，在数组中的每个元素上都执行 DumpObj 和 DumpVC 命令。最后， - nofields 选项能使得 DumpArray 在与 - detail 选项一起使用的情况下不输出与域相关的信息。

我们再次使用已经启动的同一个调试会话，并且在清单 3-15 的所有数组上都使用 DumpArray，首先从位于地址 0x01c76d3c 上的整数数组开始。

```
0:000> !DumpArray -details 01c76d3c
Name: System.Int32[]
MethodTable: 7912d7c0
EEClass: 7912d878
Size: 32(0x20) bytes
Array: Rank 1, Number of elements 5, Type Int32
Element Methodtable: 79102290
[0] 01c76d44
    Name: System.Int32
    MethodTable 79102290
    EEClass: 79102218
    Size: 12(0xc) bytes
     (C:\Windows\assembly\GAC_32\mscorlib\2.0.0.0__b77a5c561934e089\mscorlib.dll)
    Fields:
          MT    Field   Offset                 Type VT     Attr    Value Name
    79102290  40003e9        0        System.Int32  1 instance        1 m_value
[1] 01c76d48
    Name: System.Int32
    MethodTable 79102290
    EEClass: 79102218
    Size: 12(0xc) bytes
     (C:\Windows\assembly\GAC_32\mscorlib\2.0.0.0__b77a5c561934e089\mscorlib.dll)
    Fields:
          MT    Field   Offset                 Type VT     Attr    Value Name
    79102290  40003e9        0        System.Int32  1 instance        2 m_value
[2] 01c76d4c
    Name: System.Int32
    MethodTable 79102290
    EEClass: 79102218
    Size: 12(0xc) bytes
     (C:\Windows\assembly\GAC_32\mscorlib\2.0.0.0__b77a5c561934e089\mscorlib.dll)
    Fields:
          MT    Field   Offset                 Type VT     Attr    Value Name
    79102290  40003e9        0        System.Int32  1 instance        3 m_value
[3] 01c76d50
    Name: System.Int32
```

```
    MethodTable 79102290
    EEClass: 79102218
    Size: 12(0xc) bytes
     (C:\Windows\assembly\GAC_32\mscorlib\2.0.0.0__b77a5c561934e089\mscorlib.dll)
    Fields:
          MT    Field   Offset                 Type VT     Attr    Value Name
    79102290  40003e9       0        System.Int32  1 instance        4 m_value
[4] 01c76d54
    Name: System.Int32
    MethodTable 79102290
    EEClass: 79102218
    Size: 12(0xc) bytes
     (C:\Windows\assembly\GAC_32\mscorlib\2.0.0.0__b77a5c561934e089\mscorlib.dll)
    Fields:
          MT    Field   Offset                 Type VT     Attr    Value Name
    79102290  40003e9       0        System.Int32  1 instance        5 m_value
```

DumpArray 命令可以很方便地显示数组中各个元素的值（1、2、3、4、5）。你可以看到，使用 DumpArray 比手动分析数组要方便得多。为了简便，我们为读者留了一个练习：在清单 3-15 的字符串数组上使用 DumpArray。注意，DumpArray 会自动识别出正在处理的是值类型还是引用类型。

3.7.5 栈上对象的转储

每个在 CLR 上下文中运行的托管线程都有一组相关数据。除了 CLR 维护的记录数据外，每个线程还包含了一些基本的信息，这些信息使 CLR 能够维持栈和参数以及局部变量的完整性。在大多数时候，我们可以使用 ClrStack 命令来找出每个栈帧的参数和局部变量，但有时候需要对栈进行更为深入地分析。SOS 中包含了一个命令 DumpStackObjects，它能对栈进行遍历，并输出栈上的所有托管对象。DumpStackObjects 命令的语法如下所示：

```
!DumpStackObjects [-verify] [top stack [bottom stack]]
```

如果没有指定任何参数，那么 DumpStackObjects 会输出当前线程的所有托管对象。如果希望对这个命令的输出进行限制，那么需要指定一个范围（栈顶［top stack］和栈底［bottom stack］）。verify 选项表示对每个找到的托管对象执行一个验证过程，这对判断对象是否被破坏来说是非常有用的。

我们来看一个 DumpStackObjects 的使用示例。在调试器下运行 03ObjTypes. exe 程序，继续执行直到看到以下输出：

```
Press any key to continue (Generics)
```

手动中断进入到调试器，并且通过 ~0s 将线程上下文切换到线程 0。在切换了线程上下文之后，加载 SOS 扩展并运行 DumpStackObjects 命令，如下所示：

```
0:003> ~0s
eax=00000422 ebx=002df18c ecx=7918b1dc edx=00000000 esi=002defe8 edi=00000000
eip=778e9a94 esp=002def98 ebp=002defb8 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
ntdll!KiFastSystemCallRet:
778e9a94 c3              ret
0:000> .loadby sos mscorwks
0:000> !DumpStackObjects
OS Thread Id: 0x10dc (0)
ESP/REG  Object   Name
002df0e4 01e25a80 Microsoft.Win32.SafeHandles.SafeFileHandle
002df1ac 01e26d64 Advanced.NET.Debugging.Chapter3.ObjTypes
002df1b0 01e26d64 Advanced.NET.Debugging.Chapter3.ObjTypes
002df1c0 01e258ac System.Object[]     (System.String[])
002df1ec 01e258ac System.Object[]     (System.String[])
002df2c4 01e258ac System.Object[]     (System.String[])
002df46c 01e258ac System.Object[]     (System.String[])
002df494 01e258ac System.Object[]     (System.String[])
```

在 DumpStackObjects 的输出中有三列信息：

- ESP/REG。该列表示托管对象所在的栈的位置。
- Object。该列是托管对象的地址。我们可以将这个地址传递给 DumpObj 命令，从而得到关于这个对象的更详细信息。例如，将前面清单中对象地址传 0x01e26d64 递给 DumpObj 命令，会输出以下信息：

```
0:000> !DumpObj 01e26d64
Name: Advanced.NET.Debugging.Chapter3.ObjTypes
MethodTable: 002e3124
EEClass: 002e12d8
Size: 28(0x1c) bytes
 (C:\ADNDBin\03ObjTypes.exe)
Fields:
      MT    Field   Offset                 Type VT     Attr    Value Name
002e30d0  4000001        c ...jTypes+Coordinate  1 instance 01e26d70 coordinate
7912d7c0  4000002        4       System.Int32[]  0 instance 01e26e1c intArray
7912d8f8  4000003        8      System.Object[]  0 instance 01e26e3c strArray
```

- 从 DumpObjects 的输出中可以看到，有几行的对象地址是相同的。这在预料之中，因为对象可能从一个函数（栈帧）传递给另一个函数，其中每个栈帧都包含了对同一个托管对象的引用。
- Name。在 name 列中给出了与托管对象对应的文本表示方式。

DumpStackObjects 的缩写形式

由于 DumpStackObjects 命令的拼写很长，因此有一个缩写形式：dso >。

3.7.6 找出对象的大小

对象的大小表示这个类型所占据的内存字节数量。我们已经看到了有多种方式可以获得对象的大小，例如使用 DumpObj 命令。DumpObj 命令的输出如下所示：

```
0:000> !DumpObj 01de1198
Name: System.SharedStatics
MethodTable: 790feba4
EEClass: 790feaf8
Size: 28(0x1c) bytes
 (C:\Windows\assembly\GAC_32\mscorlib\2.0.0.0__b77a5c561934e089\mscorlib.dll)
Fields:
      MT    Field   Offset                 Type VT     Attr    Value Name
790fd8c4  4000512        c        System.String  0 instance 00000000
_Remoting_Identity_IDGuid
79119d38  4000513       10 ...nizer+StringMaker  0 instance 01de25cc _maker
79102290  4000514       14         System.Int32  1 instance        0
_Remoting_Identity_IDSeqNum
790ffcc8  4000515        4         System.Int64  1 instance 0
_memFailPointReservedMemory
790feba4  4000511      120 System.SharedStatics  0   shared   static _sharedStatics
    >> Domain:Value  0017fd30:01de1198 <<
```

在这个示例中，DumpObj 报告对象的大小为 28（0x1c）个字节。当需要找出单个对象的确切大小时，它会很有用。通常，对象本身可能会引用其他的对象，而这些对象又可能再引用其他的对象，依此类推。有时候，知道对象的总体大小（包括遍历每个类型域的大小）能够有助于了解一些非常大的和复杂的对象。使用前面的相同引用，我们可以通过 ObjSize 命令来转储出对象的总体大小，如下所示：

```
0:000> !ObjSize 01de1198
sizeof(01de1198) =         9624 (      0x2598) bytes (System.SharedStatics)
```

总体大小为 9624 字节，这远远大于最初报告的 28 字节大小。如果在运行 ObjSize 命令时没有指定地址，那么这个命令将列出进程中所有托管线程中的所有对象的大小。

3.7.7 异常的转储

异常模型已经出现很长一段时间了，Windows 在实现异常模型时采用的方法之一就是结构化异常处理（Structured Exception Handling，SEH）。同样，.NET 异常模型也是构建在 Windows SHE 模型之上的，并提供了基于对象的异常模型，使其能够传播更详细的异常信息。CLR 在每个异常内携带的额外信息被保存在托管堆上。事实上，在 CLR 看来，异常是一种引用类型，因此可以使用前面介绍过的任何一种技术来转储出异常的详细内容。所有 CLR 异常都以 SHE 异常的形式出现，错误码为 0xe0434f4d，这就意味着每当异常被抛出时，调试器都将按照以下形式来报告异常：

```
(a98.cd0): CLR exception - code e0434f4d (first chance)
(a98.cd0): CLR exception - code e0434f4d (!!! second chance !!!)
```

在调试器中报告了异常后（假设这个异常没有被处理），那么程序会停止，从而使用户可以分析异常的来源。既然所有异常的错误码都为 0xe0434f4d，那么我们如何区分不同的 .NET异常？答案在于异常中保存的扩展异常信息。下面的清单给出了一个在抛出.NET 异常时的典型的非托管栈回溯：

```
ChildEBP RetAddr  Args to Child
0013f304 79f55b05 e0434f4d 00000001 00000001 KERNEL32!RaiseException+0x53
0013f364 7a0904d5 01283cac 00000000 00000000 mscorwks!
RaiseTheExceptionInternalOnly+0x226
0013f428 00cb0472 01282e54 01283cac 01282e54 mscorwks!JIT_Throw+0xfc
WARNING: Frame IP not in any known module. Following frames may be wrong.
0013f480 79e7be1b 0013f4cc 00000000 0013f510 0xcb0472
0013f490 79e7bd9b 0013f560 00000000 0013f530 mscorwks!CallDescrWorker+0x33
...
...
...
0013ff18 79edae8b 00400000 00000000 8ba1dfa4 mscorwks!
SystemDomain::ExecuteMainMethod+0x398
0013ff68 79edadf3 00400000 8ba1df7c 7c911440 mscorwks!ExecuteEXE+0x59
0013ffb0 79004044 0007f4cc 79e70000 0013fff0 mscorwks!_CorExeMain+0x11b
0013ffc0 7c817067 7c911440 0007f4cc 7ffdf000 mscoree!_CorExeMain+0x2c
0013fff0 00000000 79004010 00000000 78746341 KERNEL32!BaseProcessStart+0x23
```

在栈的顶部是一组栈帧，负责抛出异常。最值得注意的栈帧就是 mscorwks 模块中的 RaiseTheExceptionInternalOnly。具体来说，该函数的第一个参数是托管异常的地址（0x01283cac）。可以通过 DumpObj 命令来显示这个托管异常的所有属性。首先在调试器下运行 03ObjTypes. exe，并继续执行直到抛出一个 CLR 异常（记住所有 CLR 异常的异常码都是 0xe0434f4d）。当执行停止时，可以使用命令 kb 来输出调用栈：

```
0:000> kb
ChildEBP RetAddr  Args to Child
0013f304 79f55b05 e0434f4d 00000001 00000001 KERNEL32!RaiseException+0x53
0013f364 7a0904d5 01283cac 00000000 00000000 mscorwks!
RaiseTheExceptionInternalOnly+0x226
0013f428 00cb0483 01282e54 01283cac 01282e54 mscorwks!JIT_Throw+0xfc
WARNING: Frame IP not in any known module. Following frames may be wrong.
0013f480 79e7be1b 0013f4cc 00000000 0013f510 0xcb0483
0013f490 79e7bd9b 0013f560 00000000 0013f530 mscorwks!CallDescrWorker+0x33
0013f510 79e7bce8 0013f560 00000000 0013f530 mscorwks!
CallDescrWorkerWithHandler+0xa3
0013f650 79e7bbd0 00913178 0013f72c 0013f6e0 mscorwks!MethodDesc::CallDescr+0x19c
0013f668 79e802f4 00913178 0013f72c 0013f6e0 mscorwks!
MethodDesc::CallTargetWorker+0x20
```

```
0013f67c 79edb56e 0013f6e0 47284667 00000000
mscorwks!MethodDescCallSite::CallWithValueTypes_RetArgSlot+0x18
0013f7e0 79edb367 00913030 00000001 0013f81c mscorwks!ClassLoader::RunMain+0x220
0013fa48 79edb23c 00000000 47284bb7 00000000
mscorwks!Assembly::ExecuteMainMethod+0xa6
0013ff18 79edae8b 00400000 00000000 47284e2b
mscorwks!SystemDomain::ExecuteMainMethod+0x398
0013ff68 79edadf3 00400000 47284ef3 7c911440 mscorwks!ExecuteEXE+0x59
0013ffb0 79004044 0007f4cc 79e70000 0013fff0 mscorwks!_CorExeMain+0x11b
0013ffc0 7c817067 7c911440 0007f4cc 7ffde000 mscoree!_CorExeMain+0x2c
0013fff0 00000000 79004010 00000000 78746341 KERNEL32!BaseProcessStart+0x23
```

接下来，找出 RaiseTheExceptionInternalOnly 函数（0x01283cac）的第一个参数，并在这个地址上执行 DumpObj 命令来输出托管异常：

```
0:000> !DumpObj 01283cac
Name: System.ArgumentException
MethodTable: 7910139c
EEClass: 79101324
Size: 76(0x4c) bytes
 (C:\WINDOWS\assembly\GAC_32\mscorlib\2.0.0.0__b77a5c561934e089\mscorlib.dll)
Fields:
      MT    Field   Offset                 Type VT     Attr    Value Name
790f9244  40000b5        4        System.String  0 instance 00000000 _className
79107d4c  40000b6        8 ...ection.MethodBase  0 instance 00000000
_exceptionMethod
790f9244  40000b7        c        System.String  0 instance 00000000
_exceptionMethodString
790f9244  40000b8       10        System.String  0 instance 01283cf8 _message
79112734  40000b9       14 ...tions.IDictionary  0 instance 00000000 _data
790f984c  40000ba       18     System.Exception  0 instance 00000000 _innerException
790f9244  40000bb       1c        System.String  0 instance 00000000 _helpURL
790f8a7c  40000bc       20        System.Object  0 instance 01283d54 _stackTrace
790f9244  40000bd       24        System.String  0 instance 00000000
_stackTraceString
790f9244  40000be       28        System.String  0 instance 00000000
_remoteStackTraceString
790fdb60  40000bf       34         System.Int32  0 instance        0
_remoteStackIndex
790f8a7c  40000c0       2c        System.Object  0 instance 00000000 _dynamicMethods
790fdb60  40000c1       38         System.Int32  0 instance -2147024809 _HResult
790f9244  40000c2       30        System.String  0 instance 00000000 _source
790fcfa4  40000c3       3c        System.IntPtr  0 instance        0 _xptrs
790fdb60  40000c4       40         System.Int32  0 instance -532459699 _xcode
790f9244  40001e5       44        System.String  0 instance 00000000 m_paramName
```

从 DumpObj 的输出中，我们可以得到托管代码异常的所有信息，包括异常的类型（ArgumentException）以及与异常相关的所有域（栈回溯、消息、HRESULT、参数名字等）。可以进一步在_ message 域上使用 DumpObj 命令来找出异常消息：

```
0:000> !DumpObj -nofields 01283cf8
Name: System.String
MethodTable: 790f9244
EEClass: 790f91a4
Size: 54(0x36) bytes
 (C:\WINDOWS\assembly\GAC_32\mscorlib\2.0.0.0__b77a5c561934e089\mscorlib.dll)
String: Obj cannot be null
```

虽然通过 DumpObj 命令来转储异常是一种可行的方法，但 DumpObj 命令有些繁琐。在大多数时候，我们只希望显示异常的类型，栈回溯（包括内部栈回溯）以及消息等。在 SOS 调试器扩展中包含了一个命令 PrintExeption。PrintException 命令的参数是托管异常的地址，它能够以更容易理解的形式输出异常信息。如果使用前面相同的异常地址（0x01283cac），那么 PrintException 命令会有以下输出：

```
0:000> !PrintException 01283cac
Exception object: 01283cac
Exception type: System.ArgumentException
Message: Obj cannot be null
InnerException: <none>
StackTrace (generated):
    SP       IP       Function
    0013F430 00CB0483 Advanced.NET.Debugging.Chapter3.
ObjTypes.ThrowException(Advanced.NET.Debugging.Chapter3.ObjTypes)
    0013F444 00CB0133
Advanced.NET.Debugging.Chapter3.ObjTypes.Main(System.String[])

StackTraceString: <none>
HResult: 80070057
```

在分析异常时，另一个有用的命令就是 Threads，它能显示出系统中各个托管线程的信息，包括该线程抛出的最后一个异常。如果在之前的调试会话中执行 Threads 命令，那么将看到以下输出：

```
0:000> !Threads
ThreadCount: 2
UnstartedThread: 0
BackgroundThread: 1
PendingThread: 0
DeadThread: 0
Hosted Runtime: no
                                      PreEmptive   GC Alloc          Lock
       ID OSID ThreadOBJ    State     GC       Context       Domain   Count APT
Exception
   0    1  a0c 00190d50      a020 Enabled  00000000:00000000 0015bf28     0 MTA
System.ArgumentException (01283cac)
   2    2  580 0019aa88       b220 Enabled  00000000:00000000 0015bf28     0 MTA
(Finalizer)
```

从上面的输出可以看到，ID 为 1 的托管线程抛出了一个 System. ArgumentException。

最后需要注意的一个异常命令是 StopOnException 。这个命令与异常信息的转储并非紧密相关，它的作用是在抛出特定的异常时设置一个断点。StopOnException 命令的语法如下所示：

```
!StopOnException [-derived]
                 [-create | -create2]
                 <Exception>
                 [<Pseudo-register number>]
```

create 和 create2 这两个开关控制着断点是在第一次出现指定异常时触发，还是在第二次出现时触发。derived 开关会增加断点的范围，不仅包括指定的异常，而且还包括从指定异常派生出来的任意异常。pseudo-register 是可选的，表示命令将使用哪个伪寄存器来设置断点。如果没有指定伪寄存器，那么默认是 $ t1。我们来看一些示例。如果希望当第一次抛出 System. ArgumentException 时设置一个断点，那么可以使用以下命令：

```
!StopOnException -create System.ArgumentException
```

如果希望对从 System. Exception 中派生下来的任意异常都设置一个断点，那么可以使用以下命令：

```
!StopOnException -derived System.Exception
```

为什么调试器不在抛出 CLR 异常时中断

到目前为止，所有的示例都假设调试器已经设置成在第一次和第二次托管异常时发生中断。根据调试器的配置，这可能与你的需求相符，也可能不符。要在所有的托管异常上重新激活中断，可以使用 sxe 命令。sxe 命令能控制调试器在发生异常时的行为。通过在 sxe 命令中增加 clr 开关，告诉调试器在所有的托管异常上都停止执行：

```
sxe clr
```

3.8 线程的操作

在非托管代码调试中，所有与线程相关的调试器命令都是以非托管 Windows 线程为基础的。然而，托管代码调试给出了一种新的方法，因为托管代码线程采用了自己的线程结构，并且调试器本身无法对栈进行遍历。我们已经知道，CLR 能对托管代码调用进行动态转换，并且 JIT 编译器可以将生成的机器代码放在它认为合适的任意地方。非托管调试器并不知道关于 JIT 编译器的任何知识，也不知道它会把生成的代码放在何处，因此就无法正确地显示栈回溯。让我们来看一个示例。在调试器下运行 03simple. exe 程序，当按下任意键退出程序时，通过 CTRL + C 中断程序执行并进入调试器。接下来，对线程 0 进行转储，并观察托管代码调用栈在非托管调试器中的结构，如清单 3-15 所示。

清单 3-15 使用 kn 命令的托管代码调用栈示例

```
0:000> kn
 # ChildEBP RetAddr
00 0013f24c 7c90dacc ntdll!KiFastSystemCallRet
01 0013f250 7c912dc8 ntdll!NtRequestWaitReplyPort+0xc
02 0013f270 7c8743e8 ntdll!CsrClientCallServer+0x8c
03 0013f358 7c87450d KERNEL32!GetConsoleInput+0xdd
04 0013f378 0090a61c KERNEL32!ReadConsoleInputA+0x1a
WARNING: Frame IP not in any known module. Following frames may be wrong.
05 0013f400 793b8138 0x90a61c
06 0013f468 793b8043 mscorlib_ni+0x2f8138
07 0013f490 79e7bd9b mscorlib_ni+0x2f8043
08 0013f510 79e7bce8 mscorwks!CallDescrWorkerWithHandler+0xa3
09 0013f650 79e7bbd0 mscorwks!MethodDesc::CallDescr+0x19c
0a 0013f668 79e802f4 mscorwks!MethodDesc::CallTargetWorker+0x20
0b 0013f67c 79edb56e mscorwks!MethodDescCallSite::CallWithValueTypes_RetArgSlot+0x18
0c 0013f7e0 79edb367 mscorwks!ClassLoader::RunMain+0x220
od 0013fa48 79edb23c mscorwks!Assembly::ExecuteMainMethod+0xa6
oe 0013ff18 79edae8b mscorwks!SystemDomain::ExecuteMainMethod+0x398
of 0013ff68 79edadf3 mscorwks!ExecuteEXE+0x59
10 0013ffb0 79004044 mscorwks!_CorExeMain+0x11b
11 0013ffc0 7c817067 mscoree!_CorExeMain+0x2c
12 0013fff0 00000000 KERNEL32!BaseProcessStart+0x23
```

调用栈中的所有栈帧都在执行 mscorwks 和 mscorlib 等模块中的代码，这两个模块负责调用实际的托管代码（无论它位于内存何处）。栈帧 0xc 到 0x11 表示应用程序中的 Main 函数被调用了（通过 ExecuteMainMethod 函数可以看出来）。从 0x8 到 0xa 的栈帧负责解析 Main 函数的元数据，并在解析后进行调用。由于托管代码帧本身（0x5）没有相关的符号文件，因此会输出以下警告信息：

```
WARNING: Frame IP not in any known module. Following frames may be wrong.
```

调试器尝试解析位于地址 0x90a61c 处的代码，但却无法将它与任何现有的模块信息关联起来，因此显示出警告信息。这是意料之中的，因为托管代码是由 CLR 本身管理的，并且不属于非托管调用栈的范围之内。显然，在调试托管代码时，这个调用栈并不是很有用。如果有一个更为实际的托管代码调用栈，那么情况会好很多。幸运的是，SOS 调试器扩展非常了解托管代码调用栈，并且提供了一组线程命令，接下来我们将依次介绍它们。

3.8.1 ClrStack

ClrStack 命令可以用来获得一个托管代码调用栈，如清单 3-16 所示。

清单 3-16　使用 ClrStack 命令显示托管代码调用栈

```
0:000> !ClrStack
OS Thread Id: 0xb70 (0)
ESP       EIP
0013f418 7c90e4f4 [NDirectMethodFrameSlim: 0013f418] Microsoft.Win32.Win32Native.
ReadConsoleInput(IntPtr, InputRecord ByRef, Int32, Int32 ByRef)
0013f430 793b8138 System.Console.ReadKey(Boolean)
0013f470 793b8043 System.Console.ReadKey()
0013f474 00cb00a6 Advanced.NET.Debugging.Chapter2.Simple.Main(System.String[])
0013f69c 79e7be1b [GCFrame: 0013f69c]
0:000>
```

ClrStack 命令输出的第一部分信息是所显示线程的 ID。这是底层操作系统的线程 ID。接下来，它显示了托管调用栈的所有栈帧。这时，我们便可以更清楚地看到哪些函数被调用了。首先，Main 函数被调用，接下来是调用 ReadKey，这个调用将被转换为一组栈帧，其中最底层的栈帧被标注为 NDirectMethodFrameSlim。这表示正在执行托管代码与非托管代码之间的转换（在第 7 章中将详细讨论）。每个栈帧还给出了该帧所需的参数类型。

扩展命令 ClrStack 还包含一组开关，用于显示各种不同的信息。

ClrStack - l 用于显示局部变量信息（不包括名字），如下所示：

```
0:000> !ClrStack -l
OS Thread Id: 0x10ac (0)
ESP       EIP
0013f410 7c90e4f4 [NDirectMethodFrameSlim: 0013f410] Microsoft.Win32.Win32Native.
ReadConsoleInput(IntPtr, InputRecord ByRef, Int32, Int32 ByRef)
0013f428 793b8138 System.Console.ReadKey(Boolean)
    LOCALS:
        <no data>
        0x0013f438 = 0x00000001
        <no data>
        <no data>
        <no data>
        <no data>
        <no data>
        <no data>
        <no data>

0013f468 793b8043 System.Console.ReadKey()
0013f46c 00cb00c1 Advanced.NET.Debugging.Chapter3.Simple.Main(System.String[])
    LOCALS:
        <CLR reg> = 0x01281bac
```

我们可以看到，Main 函数帧中有一个局部变量位于地址 0x01281bac 上。现在，可以使用命令！DumpObj 来获得这个局部变量的扩展信息：

```
0:000> !DumpObj 0x01281bac
Name: System.Text.StringBuilder
MethodTable: 790f9664
EEClass: 790f95d0
Size: 20(0x14) bytes
 (C:\WINDOWS\assembly\GAC_32\mscorlib\2.0.0.0__b77a5c561934e089\mscorlib.dll)
Fields:
      MT    Field   Offset                 Type VT     Attr    Value Name
790fcfa4   40000b1        8        System.IntPtr  0 instance        0 m_currentThread
790fdb60   40000b2        c         System.Int32  0 instance 2147483647 m_MaxCapacity
790f9244   40000b3        4        System.String  0 instance 01281bc0 m_StringValue
```

这个局部变量的类型为 StringBuilder，并且最后一个域的名字为 m_ StringValue，值为 0x01281bc0。我们可以通过 DumpObj 命令来获得字符串的值：

```
0:000> !DumpObj 01281bc0
Name: System.String
MethodTable: 790f9244
EEClass: 790f91a4
Size: 146(0x92) bytes
 (C:\WINDOWS\assembly\GAC_32\mscorlib\2.0.0.0__b77a5c561934e089\mscorlib.dll)
String: Welcome to Advanced .NET Debugging!
Fields:
      MT    Field   Offset                 Type VT     Attr    Value Name
790fdb60  4000096        4         System.Int32  0 instance       65 m_arrayLength
790fdb60  4000097        8         System.Int32  0 instance       35 m_stringLength
790fad38  4000098        c          System.Char  0 instance       57 m_firstChar
790f9244  4000099       10        System.String  0   shared   static Empty
    >> Domain:Value  0015bf38:790d57b4 <<
79122994  400009a       14        System.Char[]  0   shared   static WhitespaceChars
    >> Domain:Value  0015bf38:012816c8 <<
```

这时，显示的对象类型是一个 string，并且同时给出了这个 string 对象的值：

```
Welcome to Advanced .NET Debugging
```

ClrStack 命令的另一个开关为 -p。开关 -p 将显示调用栈上每个托管代码栈帧的所有参数，如下所示：

```
0:000> !ClrStack -p
OS Thread Id: 0x10ac (0)
ESP       EIP
0013f410 7c90e4f4 [NDirectMethodFrameSlim: 0013f410] Microsoft.Win32.Win32Native.
ReadConsoleInput(IntPtr, InputRecord ByRef, Int32, Int32 ByRef)
0013f428 793b8138 System.Console.ReadKey(Boolean)
    PARAMETERS:
        intercept = 0x00000000

0013f468 793b8043 System.Console.ReadKey()
0013f46c 00cb00c1 Advanced.NET.Debugging.Chapter3.Simple.Main(System.String[])
```

```
    PARAMETERS:
        args = 0x01281b08

0013f69c 79e7be1b [GCFrame: 0013f69c]
```

从输出信息中，我们可以看到 Main 栈帧中有一个参数 args，值为 0x01281b08。从栈帧本身还可以看到 Main 函数带有一个参数 String[ ]。如果将这个地址上的对象转储出来，可以看到以下输出：

```
0:000> !DumpObj 0x01281b08
Name: System.Object[]
MethodTable: 7912254c
EEClass: 79122ac0
Size: 16(0x10) bytes
Array: Rank 1, Number of elements 0, Type CLASS
Element Type: System.String
Fields:
None
```

ClrStack 命令的最后一个开关为 -a，它只是将 -l 和 -p 合并起来。

请注意，如果 ClrStack 命令在非托管代码线程的上下文中运行，那么会显示一个错误信息，如下所示：

```
0:001> !ClrStack
OS Thread Id: 0x1608 (1)
Unable to walk the managed stack. The current thread is likely not a
managed thread. You can run !threads to get a list of managed threads in
the process.
```

可以看到，ClrStack 指出这个线程可能不是一个托管线程，并且提示可以通过 threads 命令来获得进程中的所有托管线程。

3.8.2 Threads

Threads 命令能枚举出进程中的所有托管代码线程。以 03simple.exe 为例，我们可以看到在进程中存在以下的托管代码线程：

```
0:001> !Threads
ThreadCount: 2
UnstartedThread: 0
BackgroundThread: 1
PendingThread: 0
DeadThread: 0
Hosted Runtime: no
                                    PreEmptive   GC Alloc           Lock
       ID OSID ThreadOBJ    State     GC       Context       Domain   Count APT
Exception
   0    1 10ac 00190d88      a020 Enabled  00000000:00000000 0015bf38     0 MTA
```

```
  2    2 10b0 0019aaf8      b220 Enabled  00000000:00000000 0015bf38     0 MTA
(Finalizer)
```

Threads 命令的第一部分输出信息是对进程中所有线程行为的总体介绍。接下来的部分更值得注意，其中包含了以下信息：

- 第一列（没有名字）是调试器线程 ID。在前面的示例输出中，第一个线程的 ID 为 0，而第二线程的 ID 为 2。
- ID 列是托管线程的 CLR 线程 ID。
- OSID 列是操作系统线程 ID。
- ThreadObj 是指向底层 CLR 线程数据结构的指针。

在 mscorwks. dll 的公有符号中并不包含线程数据结构。然而，我们仍然可以通过分析 SSCLI 源代码来了解线程数据结构的内部细节。具体来说，CLR 用于表示线程的数据结构位于以下位置：

```
rotor\sscli20\clr\src\vm\threads.h
```

我们不讨论 threads 类的所有细节，只需注意在成员 m_OSThreadId 之后有一个数据成员 m_ExposedObject。由于已经从 Threads 命令的输出中知道了 OS 线程 ID（10ac 和 10b0），那么可以很容易地转储出 ThreadObj 指针的内容，并且可以看到这两个操作系统线程 ID。在线程 ID 之后是 m_ExposedObject，如下所示。清单 3-17 给出了一个示例，从值为 00190d88 的 ThreadObj（threads 命令的输出）指针中找出托管代码线程对象。

清单 3-17　从线程对象指针中找出托管线程数据结构

```
0:001> dd 00190d88
00190d88  79f04514 0000a020 00000000 0013f410
00190d98  00000000 00000000 00000000 00000001
00190da8  00000000 00190db0 00190db0 00190db0
00190db8  00000000 00000000 baad0000 0015d110
00190dc8  00000000 00000000 00000000 00000000
00190dd8  00000000 00000000 00000000 baadf00d
00190de8  001688f0 001a0050 001a0168 000000f0
00190df8  001a0050 00000000 00000000 00000100
0:001> dd
00190e08  00000000 00000000 00000000 00000000
00190e18  00000000 00000000 00000000 dfca504a
00190e28  00000000 00000000 00000000 00000000
00190e38  00000000 baadf00d ffffffff ffffffff
00190e48  ffffffff ffffffff 00000000 00000000
00190e58  00000000 00000000 00191158 00000000
00190e68  0019abd8 00000000 00140000 00040000
00190e78  baadf00d baadf00d baadf00d baadf00d
0:001>
00190e88  baadf00d 00000000 00000764 00000000
00190e98  00000760 00000000 0000075c 00000000
```

```
00190ea8  00000758 00000000 baadf00d baadf00d
00190eb8  baadf00d 00000000 00000000 baadf00d
00190ec8  00000768 ffffffff ffffffff 00000001
00190ed8  000010ac 008f12fc 008f11f8 80000000
00190ee8  00000002 00000000 00000000 baadf00d
00190ef8  00000000 00000001 00000000 00000000
0:001> dd 008f12fc
008f12fc  01282ed4 00000000 00000000 00000000
008f130c  00000000 00000000 00000000 00000000
008f131c  00000000 00000000 00000000 00000000
008f132c  00000000 00000000 00000000 00000000
008f133c  00000000 00000000 00000000 00000000
008f134c  00000000 00000000 00000000 00000000
008f135c  00000000 00000000 00000000 00000000
008f136c  00000000 00000000 00000000 00000000
0:001> !do 01282ed4
Name: System.Threading.Thread
MethodTable: 790fa000
EEClass: 790f9f90
Size: 52(0x34) bytes
 (C:\WINDOWS\assembly\GAC_32\mscorlib\2.0.0.0__b77a5c561934e089\mscorlib.dll)
Fields:
      MT    Field   Offset                 Type VT     Attr    Value Name
791020b8  4000627        4 ....Contexts.Context  0 instance 00000000 m_Context
7910bdcc  4000628        8 ....ExecutionContext  0 instance 00000000
m_ExecutionContext
790f9244  4000629        c        System.String  0 instance 00000000 m_Name
790f9cac  400062a       10      System.Delegate  0 instance 00000000 m_Delegate
79122b40  400062b       14    System.Object[][]  0 instance 00000000
m_ThreadStaticsBuckets
79122414  400062c       18       System.Int32[]  0 instance 00000000
m_ThreadStaticsBits
790fe2e4  400062d       1c ...ation.CultureInfo  0 instance 00000000
m_CurrentCulture
790fe2e4  400062e       20 ...ation.CultureInfo  0 instance 00000000
m_CurrentUICulture
790f8a7c  400062f       24        System.Object  0 instance 00000000
m_ThreadStartArg
790fcfa4  4000630       28        System.IntPtr  0 instance  1641864
DONT_USE_InternalThread
790fdb60  4000631       2c         System.Int32  0 instance        2 m_Priority
79102238  4000632      160 ...LocalDataStoreMgr  0   shared   static
s_LocalDataStoreMgr
    >> Domain:Value  0015bf38:00000000 <<
790f8a7c  4000633      164        System.Object  0   shared   static s_SyncObject
    >> Domain:Value  0015bf38:01282ec8 <<
```

Threads 输出的 State 域给出了线程的当前状态。对于列出的两个线程状态（a020 和 b220），我们可以使用在 Rotor 源代码（sscli20\clr\src\vm\threads.h）中定义的位掩码，如下

所示：

```
TS_LegalToJoin = 0x00000020,     // 现在是否可以尝试执行方法Join()?

TS_Background = 0x00000200,     // 线程是一后台线程。
```

Preemptive GC 域表示这个线程是否由于执行垃圾收集操作而被抢先。

GC Alloc Context 域将在第 5 章中进行讨论。

Domain 域指出线程在哪一个应用程序域中运行。可以在这些指针上使用 DumpDomain 命令来获得应用程序域的扩展信息。

LockCount 域表示线程是否获得了一个托管锁。

最后，APT 域表示 Apartment（套间），指出线程处于哪一种 COM 套间模式中。

在 Threads 命令中包含了一组开关。具体来说，-live 开关将 Threads 命令限制为只输出那些处于活跃状态的线程的信息。-special 开关表示输出进程的所有“特殊的”线程，例如垃圾收集线程、调试器线程、线程池定时器线程等。

3.8.3 DumpStack

ClrStack 命令只给出托管代码调用栈，而 k 系列的命令只给出非托管调用栈。要同时转储出托管代码调用栈和非托管代码调用栈，可以使用 DumpStack 命令。为了说明 DumpStack 命令的输出内容，我们使用前面的同一个调试会话（当 03simple. exe 提示按下任意建时中断进入到调试器）。在清单 3-18 中给出了 DumpStack 命令的简单输出。

清单 3-18 DumpStack 命令的输出

```
0:000> !DumpStack
OS Thread Id: 0x10ac (0)
Current frame: ntdll!KiFastSystemCallRet
ChildEBP RetAddr  Caller,Callee
0013f244 7c90dacc ntdll!NtRequestWaitReplyPort+0xc
0013f248 7c912dc8 ntdll!CsrClientCallServer+0x8c,
 calling ntdll!ZwRequestWaitReplyPort
0013f268 7c8743e8 KERNEL32!GetConsoleInput+0xdd, calling ntdll!CsrClientCallServer
0013f2dc 79e85194 mscorwks!Thread::HandleThreadAbort+0x9c, calling
ntdll!RtlRestoreLastWin32Error
0013f2e0 79e85199 mscorwks!Thread::HandleThreadAbort+0xa1, calling
mscorwks!_EH_epilog3
0013f300 79e7b7f3 mscorwks!Module::IsJumpTargetTableEntry+0x26, calling
mscorwks!X86JumpTargetTable::ComputeSize
0013f314 79e84cf7 mscorwks!NDirectSlimStubWorker1+0xa9, calling mscorwks!RunML
0013f320 79e77d92 mscorwks!Thread::EnablePreemptiveGC+0xf, calling
mscorwks!Thread::CatchAtSafePoint
0013f350 7c87450d KERNEL32!ReadConsoleInputA+0x1a, calling KERNEL32!GetConsoleInput
0013f370 0090a61c 0090a61c
```

```
...
...
...
0013fba4 7c91003d ntdll!RtlFreeHeap+0x647, calling ntdll!_SEH_epilog
0013fbd4 7c91003d ntdll!RtlFreeHeap+0x647, calling ntdll!_SEH_epilog
0013fbd8 7c911432 ntdll!RtlpFreeDebugInfo+0x5c, calling ntdll!RtlFreeHeap
0013fbdc 7c911463 ntdll!RtlpFreeDebugInfo+0x77, calling
ntdll!RtlLeaveCriticalSection
0013fbe4 7c911440 ntdll!RtlpFreeDebugInfo+0x6a, calling ntdll!_SEH_epilog
0013fbfc 79e739ea mscorwks!UnsafeEELeaveCriticalSection+0x1d, calling  (JitHelp:
CORINFO_HELP_GET_THREAD)
0013fc04 79e739d7 mscorwks!UnsafeEELeaveCriticalSection+0xa, calling
ntdll!RtlLeaveCriticalSection
0013fc08 79e739ea mscorwks!UnsafeEELeaveCriticalSection+0x1d, calling  (JitHelp:
CORINFO_HELP_GET_THREAD)
0013fc0c 79e739af mscorwks!CrstBase::Leave+0x77, calling
mscorwks!UnsafeEELeaveCriticalSection
0013fc10 79e739cc mscorwks!CrstBase::Leave+0x96, calling mscorwks!_EH_epilog3
0013fc50 79e75877 mscorwks!EEHeapAlloc+0x12d, calling ntdll!RtlAllocateHeap
0013fc64 7c91003d ntdll!RtlFreeHeap+0x647, calling ntdll!_SEH_epilog
0013fc68 79e75923 mscorwks!EEHeapFree+0x83, calling ntdll!RtlFreeHeap
0013fc74 79e7593f mscorwks!EEHeapFree+0xa5, calling mscorwks!_EH_epilog3
0013fc88 79e75896 mscorwks!EEHeapAlloc+0x163, calling mscorwks!_EH_epilog3
0013fc8c 79e75848 mscorwks!EEHeapAllocInProcessHeap+0x51, calling
mscorwks!EEHeapAlloc
013fc9c 79e7593f mscorwks!EEHeapFree+0xa5, calling mscorwks!_EH_epilog3
013fca0 79e758f6 mscorwks!EEHeapFreeInProcessHeap+0x21, calling mscorwks!EEHeapFree
013fcb0 79e737aa mscorwks!SaveLastErrorHolder::~SaveLastErrorHolder+0x14, calling
ntdll!RtlGetLastWin32Error
0013fcb8 79e758da mscorwks!operator delete[]+0x41, calling mscorwks!_EH_epilog3
0013fce0 79e758da mscorwks!operator delete[]+0x41, calling mscorwks!_EH_epilog3
0013fce4 79e7fe39
mscorwks!SArray<CORBBTPROF_TOKEN_LIST_ENTRY,1>::~SArray<CORBBTPROF_TOKEN_LIST_ENTRY,
1>+0x21, calling mscorwks!operator delete
0013fce8 79e7fdeb
mscorwks!SArray<CORBBTPROF_TOKEN_LIST_ENTRY,1>::~SArray<CORBBTPROF_TOKEN_LIST_ENTRY,
1>+0x27, calling mscorwks!_EH_epilog3
0013fdb0 79e7b29b mscorwks!HardCodedMetaSig::GetBinarySig+0x146, calling
mscorwks!_EH_epilog3
0013fdb4 79e7ce70 mscorwks!Binder::FetchMethod+0x5a, calling
mscorwks!HardCodedMetaSig::GetBinarySig
0013fdcc 79e7c4bb mscorwks!Binder::CheckInit+0xb, calling
mscorwks!MethodTable::IsClassInited
0013fdd4 79e7ce07 mscorwks!Binder::GetMethod+0x63, calling
mscorwks!Binder::CheckInit
0013fdd8 79e7ce1b mscorwks!Binder::GetMethod+0x79, calling mscorwks!_EH_epilog3
0013fddc 79ede7a0 mscorwks!EEPolicy::GetActionOnFailure+0x8f, calling
mscorwks!_EH_epilog3
0013fe00 79e75a3a mscorwks!CLRException::HandlerState::CleanupTry+0x13, calling
mscorwks!GetCurrentSEHRecord
0013fe10 79ed32ec mscorwks!EEStartupHelper+0x7d5, calling
```

```
mscorwks!CLRException::HandlerState::CleanupTry
0013fe28 79ed3333 mscorwks!EEStartupHelper+0x904, calling
mscorwks!__security_check_cookie
0013fe4c 79f11b6a mscorwks!REGUTIL::InitOptionalConfigCache+0x186, calling
mscorwks!SString::CaseCompareHelper
0013feb8 7c9101bb ntdll!RtlAllocateHeap+0xeac, calling ntdll!_SEH_epilog
0013febc 79e75877 mscorwks!EEHeapAlloc+0x12d, calling ntdll!RtlAllocateHeap
0013feec 79ed3477 mscorwks!EEStartup+0x50, calling mscorwks!EEStartupHelper
0013fef0 79ed3496 mscorwks!EEStartup+0x75, calling mscorwks!_SEH_epilog4
0013ff18 79edae8b mscorwks!ExecuteEXE+0x59, calling
mscorwks!SystemDomain::ExecuteMainMethod
0013ff68 79edadf3 mscorwks!_CorExeMain+0x11b, calling mscorwks!ExecuteEXE
0013ff74 7c911440 ntdll!RtlpFreeDebugInfo+0x6a, calling ntdll!_SEH_epilog
0013ffb0 79004044 mscoree!_CorExeMain+0x2c
0013ffc0 7c817067 KERNEL32!BaseProcessStart+0x23
0013ffc4 7c911440 ntdll!RtlpFreeDebugInfo+0x6a, calling ntdll!_SEH_epilog
0:000>
```

你可以看到，DumpStack 命令的输出信息非常详细，在需要某种复合的调用栈视图时，这些信息会非常有用。

在 DumpStack 命令中使用 -EE 开关表示只显示托管函数。这个命令的执行结果与使用 ClrStack 命令的执行结果几乎完全相同，只不过还显示了方法描述符指针。

我们也可以为 DumpStack 命令指定一个栈范围。这有助于减少命令输出的信息量。

3.8.4 EEStack

有时候，我们需要获得进程中所有托管线程的调用栈，此时无需对 Threads 命令输出中的每个线程都使用 Threads 和 DumpStack 命令，而是可以使用 EEStack 命令。EEStack 命令会对进程中每个活跃的线程调用 DumpStack。在 EEStack 命令中可以使用两个开关，分别为：

- -short，这个开关只输出“感兴趣的”线程的调用栈。在这里，“感兴趣的”的定义如下：
  这个线程持有一个锁。
  线程被劫持以执行一个垃圾收集操作。
  线程当前正在托管代码中执行。
- -EE，这个开关会直接传递给 DumpStack 命令，表示只显示托管代码调用栈。

3.8.5 COMState

当与 COM 子系统一起使用时，重点是要知道 COM 提供的不同套间模型。COM 提供了两种主要的套间模型。第一种被称为单线程套间（Single Threaded Apartment，STA），第二种被称为多线程套间（Multiple Threaded Apartment，MTA）。每当一个线程希望使用 COM 对象时，它必须告诉 COM 子系统它需要使用哪一种套间模型。在使用.NET 互用性层时（在第 7

章中将详细讨论），在具体套间模型中对线程进行初始化的概念非常重要，此外，在调试COM互用性问题时，找出线程的套间模型是一个更为重要的方面。SOS调试器扩展提供了COMState命令，这个命令可以找出系统中每个线程的套间模型。下面给出了COMState命令的示例输出：

```
0:000> !COMState
     ID     TEB   APT    APTId CallerTID Context
  0  ff8 7ffdf000 MTA        0         0 00398fd8
  1 1554 7ffde000 Ukn
  2 2260 7ffdd000 MTA        0         0 00398fd8
  3 1eb8 7ffdc000 Ukn
  4 23a0 7ffda000 Ukn
```

ID列表示线程的ID。TEB列表示各个线程的线程环境块。通过teb命令和TEB指针可以获得线程的扩展信息（例如最近发生的错误和栈的大小限制等）。APT列表示线程是在哪一种套间模型中被初始化的。MTA表示多线程套间，而STA表示单线程套间。Ukn通常表示线程此时还没有初始化COM。

3.9　代码审查

任何调试会话的关键在于能够分析系统的状态，并推断出是什么样代码序列使程序进入当前的状态。同样非常重要的还要能分析执行的是哪段代码以及跟踪任意系统中的执行流。在本章的这部分中，我们将看到在执行代码审查时可用的一些命令。通过代码审查，可以从一段原始代码的字节流中推断出这段代码所要执行的任务。注意，这部分内容只是简要介绍了这些命令，在本书的随后章节中将进一步讨论它们的实际应用。

3.9.1　反汇编代码

在调试非托管代码时，一个很常用的命令就是u（unassemble）以及它的各种变化形式。从本质上看，命令u把代码字节流反汇编为汇编指令，因而能很容易推断出代码所要执行或者已经执行的功能。在托管代码中，我们仍然可以使用命令u对代码进行反汇编（假设知道代码的地址）。然而，有一个问题使这个工作变得略微复杂一些：命令u本身并不清楚托管代码，它将所有代码都视作普通的传统机器代码。下面给出了在一个托管函数上使用命令u的示例输出：

```
002c035d b990221079      mov     ecx,offset mscorlib_ni+0x42290 (79102290)
002c0362 e8b51ce5ff      call    0011201c
002c0367 8bf8            mov     edi,eax
002c0369 8b460c          mov     eax,dword ptr [esi+0Ch]
002c036c 894704          mov     dword ptr [edi+4],eax
002c036f 57              push    edi
0:000>
002c0370 8bd5            mov     edx,ebp
002c0372 8bcb            mov     ecx,ebx
002c0374 e8a79a1279      call    mscorlib_ni+0x329e20 (793e9e20)
```

你可以看到，有两个 call 指令。其中一个 call 指令调用到地址 0x0011201c ，而另一个 call 语句调用位于偏移 0x329e20 处的 mscorlib_ ni 模块（mscorlib 的预编译版本）。这种类型的反汇编代码并不能告诉我们太多的信息。位于地址 0x0011201c 上的代码是什么，并且调用的是 mscorlib_ ni 中的哪个函数？在 SOS 调试器扩展中包含了一个更为有用的命令，叫做 U 命令。由于 SOS 更清楚 CLR 的内部细节，因此命令 U 能够对程序集进行标注，这使得调试人员更容易理解代码流。如果在前面给出的代码上使用命令 U，那么可以看到以下输出：

```
0:000> !U eip
002c0362 e8b51ce5ff      call    0011201c (JitHelp: CORINFO_HELP_NEWSFAST)
002c0367 8bf8            mov     edi,eax
002c0369 8b460c          mov     eax,dword ptr [esi+0Ch]
002c036c 894704          mov     dword ptr [edi+4],eax
002c036f 57              push    edi
002c0370 8bd5            mov     edx,ebp
002c0372 8bcb            mov     ecx,ebx
002c0374 e8a79a1279      call    mscorlib_ni+0x329e20 (793e9e20)
(System.Console.WriteLine(System.String, System.Object, System.Object,
System.Object), mdToken: 060007c8)
```

在这里可以看到，在前面很难理解的 call 指令现在被表示为函数本身的文本。第一个 call 指令将调用 CORINFO_HELP_NEWSFAST 函数。第二个调用 mscorlib_ni 模块的 call 指令被标注为调用 System. Console. WriteLine 函数。

命令 U 以及方法描述符

在使用 U 命令时，除了指定代码地址外，还可以指定一个方法描述符。

3.9.2 从代码地址上获得方法描述符

在前面的示例中，我们看到了如何使用非托管的命令 u 来反汇编代码，还看到了命令 u 无法解析托管代码，因此需要采用命令 U 来获得对反编译代码的更详细标注。在命令 u 的输出中，有些指令基本上是无用的，例如下面这条指令：

```
call    mscorlib_ni+0x329e20 (793e9e20)
```

有一种机制可以将任意的托管代码地址转换为一个方法描述符，然后使用 DumpMD 命令获得进一步的信息。这个命令被称为 IP2MD（instruction pointer to method descriptor），它的语法如下所示：

```
!IP2MD <Code address>
```

如果在之前 call 指令的地址（0x793e9e20）上使用这个命令，可以看到以下输出：

```
0:000> !ip2md 0x793e9e20
MethodDesc: 79259d40
Method Name: System.Console.WriteLine
(System.String, System.Object, System.Object, System.Object)
Class: 79101018
MethodTable: 79101118
mdToken: 060007c8
Module: 790c2000
IsJitted: yes
m_CodeOrIL: 793e9e20
```

除了方法的名字外，它还显示了其他一些信息，例如方法描述符地址、方法表、JIT状态、代码地址等。要想快速找出汇编代码位于哪个函数中，使用 IP2MD 命令是很方便的。

3.9.3　显示中间语言指令

虽然能查看反汇编代码很有用，但有时候将反汇编代码与相应的 IL 指令关联起来同样有用。DumpIL 命令就可以实现这个功能。例如，位于地址 0x793e9e20 处的机器代码对应于 WriteLine 函数，因此可以使用 DumpIL 命令来查看相应的 IL 指令，如下所示：

```
0:000> !IP2MD 0x793e9e20
MethodDesc: 79259d40
Method Name: System.Console.WriteLine
(System.String, System.Object, System.Object, System.Object)
Class: 79101018
MethodTable: 79101118
mdToken: 060007c8
Module: 790c2000
IsJitted: yes
m_CodeOrIL: 793e9e20
0:000> !DumpIL 79259d40
ilAddr = 79b28110
IL_0000: call System.Console::get_Out
IL_0005: ldarg.0
IL_0006: ldarg.1
IL_0007: ldarg.2
IL_0008: ldarg.3
IL_0009: callvirt System.IO.TextWriter::WriteLine
IL_000e: ret
```

可以看到，DumpIL 命令以方法描述符的地址作为参数。要找出位于地址 0x793e9e20 处代码的描述符，可以使用 IP2MD 命令，并把得到的方法描述符传递给 DumpIL 命令，这个命令会输出相应的 IL。

3.10 CLR 内部命令

在本章中，我们已经看到了有多个命令可以帮助调试人员更深入地了解各种对象，CLR 通过这些对象能有效地执行托管代码。在本章的本节内容中，我们还将看到一些辅助命令，这些命令能帮助我们理解 CLR 的内部工作机制。

在对 CLR 内部命令的讨论中，我们使用 03ObjTypes. exe 示例程序，在调试器下运行这个程序，并在显示以下提示时中断进入到调试器：

```
Press any key to continue (AddCoordinate)
```

3.10.1 获得 CLR 的版本

要想快速找出当前在调试会话中使用的是哪个版本的 CLR，可以使用 EEVersion 命令。此外，这个命令还能给出正在使用的 SOS 的版本，以及 CLR 的当前运行模式（服务器模式还是工作站模式）。

```
0:000> !EEVersion
2.0.50727.1434 retail
Workstation mode
SOS Version: 2.0.50727.1434 retail build
```

3.10.2 根据名字找到方法描述符

在 SOS 调试器扩展中，有许多命令都需要使用方法描述符。对于任何一个方法，有多种不同的方式可以找出方法描述符。在给定了某个方法的名字后，找出方法描述符的最有用方法之一就是 Name2ee 命令。Name2ee 命令参数是模块的名字和方法的全名，这个命令将输出方法的一些信息，包括方法描述符。让我们来看看如何在示例程序 03ObjTypes. exe 中的一个方法上使用 Name2ee 命令。启动 Name2ee 命令，如下所示：

```
0:004> .loadby sos mscorwks
0:004> !Name2ee 03ObjTypes.exe Advanced.NET.Debugging.Chapter3.
ObjTypes.AddCoordinate
Module: 001e2c3c (03ObjTypes.exe)
Token: 0x06000002
MethodDesc: 001e3070
Name: Advanced.NET.Debugging.Chapter3.ObjTypes.AddCoordinate(Coordinate)
Not JITTED yet. Use !bpmd -md 001e3070 to break on run.
```

运行 Name2ee 命令将输出方法的信息，还有方法描述符地址（0x001e3070），在任何需要使用方法描述符的 SOS 命令中都可以使用这个地址。

Name2ee 命令还可以用于获得某个特定类型的详细信息。唯一的差别在于，我们需要指定一个完整的类型名，而不只是一个方法名。例如，使用之前的调试会话，我们可以找出关

于 ObjTypes 类型的进一步信息：

```
0:004> !Name2ee 03ObjTypes.exe Advanced.NET.Debugging.Chapter3.ObjTypes
Module: 001e2c3c (03ObjTypes.exe)
Token: 0x02000002
MethodTable: 001e3124
EEClass: 001e12d8
Name: Advanced.NET.Debugging.Chapter3.ObjTypes
```

3.10.3 对象同步块的转储

每个 CLR 托管类型都有一个相应的同步块，用于实现同步行为（在第 2 章中简要讨论过）。SyncBlk 命令可以用来获得这个同步块的详细信息，这个命令在分析死锁问题时非常有用。

在第 6 章中，我们将看到一些常见的同步问题，以及如何使用这个命令来找出这些问题的根源。

3.10.4 对象方法表的转储

每个托管对象都有一个相应的方法表，其中包含了该对象的一些信息（参见第 2 章了解对方法表的详细讨论）。DumpMT 命令可以用来显示方法表的信息，命令参数是方法表的地址。使用之前的调试会话，我们可以转储出类型 ObjTypes 的方法表信息，如下所示：

```
0:000> !Name2ee 03ObjTypes.exe Advanced.NET.Debugging.Chapter3.ObjTypes
Module: 001e2c3c (03ObjTypes.exe)
Token: 0x02000002
MethodTable: 001e3124
EEClass: 001e12d8
Name: Advanced.NET.Debugging.Chapter3.ObjTypes
0:000> !DumpMT 001e3124
EEClass: 001e12d8
Module: 001e2c3c
Name: Advanced.NET.Debugging.Chapter3.ObjTypes
mdToken: 02000002  (C:\ADNDBin\03ObjTypes.exe)
BaseSize: 0x1c
ComponentSize: 0x0
Number of IFaces in IFaceMap: 0
Slots in VTable: 9
```

3.10.5 托管堆和垃圾收集器信息的转储

CLR 垃圾收集器是一种高效的自动内存管理器，它负责确保内存实现最优的布局和管理。要高效地调试.NET 应用程序，我们必须分析垃圾收集器的内部状态。在 SOS 调试器扩展中定义了一些命令，可以完成这个任务。表 3-2 给出了其中一些命令以及相应的描述信息。

表 3-2 SOS 托管堆和垃圾收集器命令的示例

命 令	描 述
DumpHeap	遍历托管堆，收集并输出这个堆以及位于堆上所有对象的详细信息
GCRoot	显示对某个对象的引用（根对象）信息。当要找出某个对象为什么还没有被收集时，这将是非常有用的信息
TraverseHeap	遍历托管堆，并且把遍历结果输出到一个文件中，由 CLR 分析器来进行分析
VerifyHeap	与任何堆一样，托管堆可能被破坏。这个命令将验证托管堆的完整性

在第 5 章中我们将深入讨论表 3-2 中的各个命令，也会看到在错误使用托管堆和垃圾收集器时出现的一些实际问题。

3.11 诊断命令

到目前为止，除了讨论过的调试分析命令外，SOS 调试器还包含了一些非常有用的诊断命令，这些命令可以在调试会话中提供一些辅助信息。在本章的这部分内容中，我们就能看到一些有用的诊断命令以及讨论如何使用这些命令。

3.11.1 找出对象的应用程序域

在第 2 章中，我们曾介绍了应用程序域的概念，以及.NET 对象实例如何（在大多数时候）被绑定到某个特定的应用程序域。要找出指定的对象实例位于哪一个应用程序域中，可以使用 FindAppDomain 命令，如下所示：

```
0:000> !DumpStackObjects
OS Thread Id: 0x1964 (0)
ESP/REG  Object   Name
001aef54 01b85a80 Microsoft.Win32.SafeHandles.SafeFileHandle
001af01c 01b86d64 Advanced.NET.Debugging.Chapter3.ObjTypes
001af020 01b86d64 Advanced.NET.Debugging.Chapter3.ObjTypes
001af030 01b858ac System.Object[]     (System.String[])
001af05c 01b858ac System.Object[]     (System.String[])
001af134 01b858ac System.Object[]     (System.String[])
001af2e4 01b858ac System.Object[]     (System.String[])
001af30c 01b858ac System.Object[]     (System.String[])
0:000> !FindAppDomain 01b86d64
AppDomain: 0022fd30
Name: 03ObjTypes.exe
ID: 1
```

在知道了对象的应用程序域后，你可以使用 DumpDomain 命令来获取应用程序域的进一步信息。如果知道对象实例所在的应用程序域，那么就能获得关于对象来源的一些线索。

3.11.2 进程信息

在调试会话中，有时候获得关于被调试进程的更多信息是非常有用的。例如，内存使用

量、环境变量、处理时间等，这些数据都是非常有用的。要在调试时将这些信息转储出来，可以使用 ProcInfo 命令。它的语法如下所示：

```
!ProcInfo [-env] [-time] [-mem]
```

其中 -env、-time、-mem 等开关控制着所要显示的进程信息。如果没有指定任何开关，那么会显示所有这三类信息。以下是在示例程序 03ObjTypes. exe 上使用 PronInfo 命令的输出：

```
0:000> !ProcInfo
-----------------------------------------------
Environment
=C:=C:\ADNDBin
=ExitCode=E0434F4D
ALLUSERSPROFILE=C:\ProgramData
APPDATA=C:\Users\marioh\AppData\Roaming
AVENGINE=C:\PROGRA~1\CA\SHARED~1\SCANEN~1
...
...
...
USERDOMAIN=REDMOND
USERNAME=marioh
USERPROFILE=C:\Users\marioh.REDMOND
VS90COMNTOOLS=c:\Program Files\Microsoft Visual Studio 9.0\Common7\Tools\
windir=C:\Windows
-----------------------------------------------
Process Times
Process Started at: 2008 Dec  3 6:42:5.00
Kernel CPU time   : 0 days 00:00:00.03
User   CPU time   : 0 days 00:00:00.03
Total  CPU time   : 0 days 00:00:00.06
-----------------------------------------------
Process Memory
WorkingSetSize:      7528 KB       PeakWorkingSetSize:      8504 KB
VirtualSize:       103004 KB       PeakVirtualSize:       103004 KB
PagefileUsage:       7716 KB       PeakPagefileUsage:       7720 KB
-----------------------------------------------
72 percent of memory is in use.

Memory Availability (Numbers in MB)

                     Total        Avail
Physical Memory       3061          839
Page File             4095         3495
Virtual Memory        2047         1914
```

3.12　SOSEX 扩展命令

在 SOS 调试器扩展中包含了许多托管代码扩展命令，这些命令能使我们在调试托管代码时更加轻松。还有另一个 SOSEX 调试器扩展，它同样包含了在调试托管代码时需要的一些功

能。在接下来的章节中，我们将介绍在 SOSEX 中包含的扩展命令。

3.12.1 扩展的断点支持

SOSEX 引入了一组新的断点命令，这些命令可以管理断点列表（包括处于已启用状态和未启用状态的断点），并且可以在任意的源代码位置上设置断点。

断点列表

SOSEX 维持了一张断点列表，其中的断点都是通过 SOSEX 的断点命令设置的。它支持一组命令来管理这个断点列表。第一个命令是 mbl，它将显示所有的断点。例如，如果通过 SOSEX 命令来设置断点，那么 mbl 命令将输出以下信息：

```
0:000> !mbl
0 e : 03simple.cs, line 10: pass=1 oneshot=false thread=ANY
   03Simple!Advanced.NET.Debugging.Chapter3.Simple.Main(string[])+0x1(IL)
      0 e 00ac0085
```

还可以通过以下命令来管理列表中断点的状态：

- 命令 mbc 将从列表中清除指定的断点，或者清除所有的断点。
- 命令 mbd 将禁用列表中指定的断点，或者禁用所有的断点。
- 命令 mbe 将启用列表中指定的断点，或者启用所有的断点。

设置断点

SOSEX 引入了一个新的命令 mbp，这个命令可以在任意给定的源代码位置上设置断点。例如，假定要在 03sample. cs 的第 10 行代码上设置一个断点，即在输出“Welcome to Advanced . NET Debugging”字符串之前的位置。要完成这个任务，需要启动调试器，并将 03sample. exe 程序作为命令行参数，如清单 3-19 所示。

清单 3-19 使用 SOSEX 中的 bpsc 命令

```
Microsoft (R) Windows Debugger Version 6.9.0003.113 X86
Copyright (c) Microsoft Corporation. All rights reserved.

CommandLine: c:\ADNDBin\03Simple.exe
...
...
...
0:000> .symfix
No downstream store given, using c:\Program Files\Debugging Tools for Windows\sym
0:000> .reload
Reloading current modules
....
0:000> .load sosex.dll
0:000> !mbp 03simple.cs 10
0:000>
```

在设置好正确的符号路径后，可以使用 . load 命令来加载 SOSEX。然后，使用 mbp 命令来在源代码文件 03simple. cs 的第 10 行设置一个断点。清单 3-20 给出了在使用命令 g 恢复程序执行时得到的输出。

清单 3-20　在使用 bpsd 命令时触发断点

```
0:000> g
ModLoad: 5cb70000 5cb96000   C:\WINDOWS\system32\ShimEng.dll
ModLoad: 77dd0000 77e6b000   C:\WINDOWS\system32\ADVAPI32.dll
ModLoad: 77e70000 77f02000   C:\WINDOWS\system32\RPCRT4.dll
ModLoad: 77fe0000 77ff1000   C:\WINDOWS\system32\Secur32.dll
ModLoad: 77f60000 77fd6000   C:\WINDOWS\system32\SHLWAPI.dll
ModLoad: 77f10000 77f59000   C:\WINDOWS\system32\GDI32.dll
ModLoad: 7e410000 7e4a1000   C:\WINDOWS\system32\USER32.dll
ModLoad: 77c10000 77c68000   C:\WINDOWS\system32\msvcrt.dll
ModLoad: 76390000 763ad000   C:\WINDOWS\system32\IMM32.DLL
ModLoad: 79e70000 7a3d6000   C:\WINDOWS\Microsoft.NET\Framework
\v2.0.50727\mscorwks.dll
ModLoad: 78130000 781cb000   C:\WINDOWS\WinSxS\x86_Microsoft.VC80.CRT
_1fc8b3b9a1e18e3b_8.0.50727.762_x-ww_6b128700\MSVCR80.dll
…
…
…
(cfc.126c): CLR notification exception - code e0444143 (first chance)
Breakpoint 0 hit
eax=00912fe8 ebx=0013f4ac ecx=012819f4 edx=00000000 esi=00190de8 edi=00000000
eip=00cc0085 esp=0013f474 ebp=0013f490 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
00cc0085 8b0d3c302802    mov     ecx,dword ptr ds:[228303Ch]
ds:0023:0228303c=01281a04
0:000> .loadby sos.dll mscorwks
0:000> !ClrStack
OS Thread Id: 0x126c (0)
ESP       EIP
0013f474 00cc0085 Advanced.NET.Debugging.Chapter3.Simple.Main(System.String[])
0013f69c 79e7be1b [GCFrame: 0013f69c]
0:000>
```

当执行命令 g 时，调试器会中断并显示“Breakpoint 0 hit”消息。为了确认触发的断点正确，可以加载 SOS 扩展，并使用 ClrStack 扩展命令来显示当前所处位置的栈回溯。栈回溯很清楚地显示当前处于 Main 函数中。

除了 mbp 命令外，SOSEX 还包括一个功能强大的断点命令，mbm。这个命令能够在特定类型的指定 IL 偏移处设置一个断点。当无法获得源代码时，这个命令能带来极大的便利，例如动态生成代码的时候（Reflection. Emit）。为了说明 mbm 的使用方法，我们可以使用同一个 03simple. exe 程序。利用 mbm 命令（而不是之前的 mbp 命令）在 Main 方法的起始处设

置一个断点（位于偏移 0 的地方）。清单 3-21 给出了调试会话。

清单 3-21　使用 SOSEX 中的 mbmp 命令

```
Microsoft (R) Windows Debugger Version 6.9.0003.113 X86
Copyright (c) Microsoft Corporation. All rights reserved.

CommandLine: 03Simple.exe
Symbol search path is: *** Invalid ***
...
...
...
0:000> .load sosex.dll
0:000> !mbm *!Advanced.NET.Debugging.Chapter3.Simple.Main 0
The CLR has not yet been initialized in the process.
Breakpoint resolution will be attempted when the CLR is initialized.
0:000> g
ModLoad: 76160000 76226000   C:\Windows\system32\ADVAPI32.dll
ModLoad: 774e0000 775a3000   C:\Windows\system32\RPCRT4.dll
ModLoad: 77330000 77388000   C:\Windows\system32\SHLWAPI.dll
ModLoad: 76110000 7615b000   C:\Windows\system32\GDI32.dll
...
...
...
Breakpoint: CLR initialized. Attempting to resolve managed breakpoints.
ModLoad: 766a0000 771af000   C:\Windows\system32\shell32.dll
ModLoad: 775b0000 776f4000   C:\Windows\system32\ole32.dll
ModLoad: 790c0000 79bf6000   C:\Windows\assembly\NativeImages_v2.0.50727_32
\mscorlib\5b3e3b0551bcaa722c27dbb089c431e4\mscorlib.ni.dll
(1798.1ba8): CLR notification exception - code e0444143 (first chance)
(1798.1ba8): CLR notification exception - code e0444143 (first chance)
Breakpoint: Matching method Advanced.NET.Debugging.Chapter3.Simple.Main resolved,
but not yet jitted. Setting JIT notification...
ModLoad: 79060000 790b6000   C:\Windows\Microsoft.NET\Framework\v2.0.50727
\mscorjit.dll
(1798.1ba8): CLR notification exception - code e0444143 (first chance)
Breakpoint: JIT notification received for method
Advanced.NET.Debugging.Chapter3.Simple.Main(System.String[]).
Breakpoint set at Advanced.NET.Debugging
.Chapter3.Simple.Main(System.String[]).
Breakpoint 0 hit
eax=00113020 ebx=0024eddc ecx=01cd5890 edx=00000005 esi=004a5148 edi=00000000
eip=00380084 esp=0024eda4 ebp=0024edc0 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
00380084 90              nop
0:000> .loadby sos.dll mscorwks
0:000> !ClrStack
OS Thread Id: 0x1da8 (0)
```

```
ESP       EIP
002af174 002c0084 Advanced.NET.Debugging.Chapter3.
Simple.Main(System.String[])
002af3a4 79e7c74b [GCFrame: 002af3a4]
0:000>
```

在加载 SOSEX 后，我们使用 mpm 扩展命令在 Advanced. NET. Debugging. Chapter3. Simple 类型的偏移 0 处的 Main 方法上设置一个断点。当恢复程序的执行时，会触发断点，如果使用 SOS 的 ClrStack 命令，可以看到程序的执行是停止在 Main 方法的起始位置上。在 mbm 命令中还包含了一些选项：

```
!mbm <strTypeAndMethodFilter> <intILOffset> [Options]
```

其中 Options 开关的取值如下所示：

- /1。只触发这个断点一次。
- /p：<passcount>。表示在第几次执行到断点时才停止执行程序。
- /t：<threadid>。只有由 threadid 指定线程才可以触发这个断点。

mbm 命令还支持在 strTypeAndMethodFilter 中使用通配符。

SOSEX 提供的这两个断点扩展命令都非常有用，能使在源代码中设置断点更加容易。

3.12.2　托管元数据

要想通过非托管调试器来有效地调试. NET 程序，我们需要快速找出相应的元数据，例如类型名和方法名。为了满足这种需求，可以使用 mx 命令，命令的语法为！mx <Filter String>，其中参数“Filter String”表示查找元数据的通配符。例如，如果在调试器下运行 03simple. exe，并且希望找出 Simple 类型上的所有方法，那么可以运行以下命令：

```
!mx <Filter String>

0:000> !mx 03simple!*Simple*
03Simple!Advanced.NET.Debugging.Chapter3.Simple
03Simple!Advanced.NET.Debugging.Chapter3.Simple.
Advanced.NET.Debugging.Chapter3.Simple.Main(System.String[])
03Simple!Advanced.NET.Debugging.Chapter3.Simple.
Advanced.NET.Debugging.Chapter3.Simple..ctor()
```

另一个非常有用的命令是 mln，这个命令的参数是一个地址，可以识别出与该地址上内容相关的托管代码。它可以准确地识别出 JIT 已编译的代码，堆以及栈对象。例如，假定有一个地址 X，并且我们希望知道这个地址所表示的内容（从 CLR 的角度来看）。我们可以采用手动方式并使用各种不同的命令，例如 DumpObj（判断它是否是一个对象）或者命令 U 来反汇编这个地址来确认它是否包含了代码。如果有一种自动化的方式能找出这个地址的内容，那么将更加容易。例如，在调试 03ObjTypes. exe 程序时遇到了地址 0x01eb6d64，我们就可以

在这个地址上运行 mln 命令来找出该地址上的内容：

```
0:000> !mln 0x01eb6d64
Heap Object: 01eb6d64 [Advanced.NET.Debugging.Chapter3.ObjTypes]
```

可以看到，地址 0x01eb6d64 对应于一个托管堆对象，类型为 Advanced. NET. Debugging. Chapter3. ObjTypes。

3. 12. 3 栈回溯

到目前为止，我们已经看到了各种不同方式来显示托管堆线程的调用栈。非托管调试器命令 kb，它显示了非托管代码调用栈（对于托管代码调试来说并不是很有用）；ClrStack 命令，它能显示托管代码调用栈。当需要同时显示托管代码的调用栈和非托管代码的调用栈时，还可以使用 SOSEX 的 mk 命令。除了显示托管代码栈帧和非托管代码栈帧外，它还显示了栈帧的编号。下面给出的是当 03simple. exe 中断进入到调试器中时命令 mk 的输出：

```
0:000> !mk
*** WARNING: Unable to verify checksum for
C:\Windows\assembly\NativeImages_v2.0.50727_32\mscorlib
\5b3e3b0551bcaa722c27dbb089c431e4\mscorlib.ni.dll
00:U 0028ea68 778e8d94 ntdll!KiFastSystemCallRet+0x0
01:U 0028ea6c 778f9522 ntdll!NtRequestWaitReplyPort+0xc
02:U 0028ea8c 777a7e05 ntdll!CsrClientCallServer+0xc2
03:U 0028eb78 777a7f35 KERNEL32!GetConsoleInput+0xd2
04:U 0028eb98 003aa61c KERNEL32!ReadConsoleInputA+0x1a
05:U 0028ec20 793e8f28 0x003aa61c
06:M 0028ec88 793e8e33 mscorlib!System.Console.
ReadKey(Boolean)(+0x50 IL)(+0x6d Native)
07:M 0028ecb0 79e7c6cc mscorlib!System.Console.
ReadKey()(+0x0 IL)(+0x7 Native)
08:M 0028ecb0 79e7c6cc
03Simple!Advanced.NET.Debugging.Chapter3.Simple.Main(System.String[])(+0x17
IL)(+0x22 Native) [c:\Publishing\ADND\Code\Chapter3\Simple\03simple.cs, @ 12,13]
09:U 0028ed30 79e7c8e1 mscorwks!CallDescrWorkerWithHandler+0xa3
0a:U 0028ee74 79e7c783 mscorwks!MethodDesc::CallDescr+0x19c
0b:U 0028ee90 79e7c90d mscorwks!MethodDesc::CallTargetWorker+0x1f
0c:U 0028eea4 79eefb9e mscorwks!MethodDescCallSite::Call+0x18
0d:U 0028f008 79eef830 mscorwks!ClassLoader::RunMain+0x263
0e:U 0028f270 79ef01da mscorwks!Assembly::ExecuteMainMethod+0xa6
0f:U 0028f740 79fb9793 mscorwks!SystemDomain::ExecuteMainMethod+0x43f
10:U 0028f790 79fb96df mscorwks!ExecuteEXE+0x59
11:U 0028f7d8 7900b1b3 mscorwks!_CorExeMain+0x15c
12:U 0028f7e8 77744911 mscoree!_CorExeMain+0x2c
13:U 0028f7f4 778ce4b6 KERNEL32!BaseThreadInitThunk+0xe
14:U 0028f834 778ce489 ntdll!__RtlUserThreadStart+0x23
15:U 0028f84c 00000000 ntdll!_RtlUserThreadStart+0x1b
```

输出的每个栈帧都带有前缀 xx，其中 xx 表示该栈帧在调用栈上的帧编号。在接下来要

讨论的命令 mdv 中，mk 命令输出中的栈帧号非常重要。mdv 命令输出指定栈帧的局部变量。在默认情况下，它将输出最顶层栈帧的变量。如果继续前面的示例并运行 mdv 命令，那么就会显示出最顶层栈帧的变量（ntdll！KiFastSystemCallRet +0x0），输出如下所示：

```
0:000> !mdv
Frame 0x0: <Unmanaged Frame>
```

这些输出非常有意义，因为最顶层栈帧并不是一个托管代码栈帧。命令 mdv 还可以以栈帧编号作为参数，这样可以给出在 mk 命令输出的调用栈中的托管栈帧信息。例如，要了解第 8 栈帧的变量（.NET 程序的 Main 方法），就可以使用以下命令：

```
0:000> !mdv 8
Frame 0x8: (Advanced.NET.Debugging.Chapter3.Simple.Main(System.String[])):
[P0]:args:0x1d75890 (System.String[])
```

这个命令的输出表明了唯一可用的参数就是 Main 方法的参数（位于地址 0x1d75890 处）。命令 mdv 还支持 -w 开关，这个开关将遍历整个调用栈，并且转储出所有的参数和局部变量。

最后一个命令是 mframe，这个命令可以设置当前的托管栈帧（由 mdt 和 mdv 等命令使用）。

mk、mdv 和 mframe 这三个命令都是在分析托管代码调用栈以及它们相应状态时的功能强大的命令。

3.12.4　对象检查

在非托管代码调试中，一个非常有用的命令就是 dt。它可以显示局部变量，全局变量或者某个数据类型的信息。SOSEX 调试器扩展引入了一个类似的命令，称之为 mdt 命令。mdt 命令的语法如下所示：

```
!mdt [typename | paramname | localname | MT] [ADDR] [-r]
```

其中 typename、paramname 以及 localname 表示类型的名字（或者类型的方法表），而 ADDR 指定了需要分析的对象地址。选项 -r 表示 mdt 命令将递归地执行，当在对象中封装了另一个对象时，或在这个对象中又封装了其他对象时，这个命令会非常方便。我们无需为每个嵌套对象执行 mdt 命令，而是可以使用 -r 开关。接下来的示例是在某个地址上使用 mdt 命令，这个地址对应于在 03ObjTypes.exe 中定义的 ObjTypes 类型实例：

```
0:000> !mdt 0x1b06d64
01b06d64 (Advanced.NET.Debugging.Chapter3.ObjTypes)
   coordinate:01b06d70 (Advanced.NET.Debugging.Chapter3.ObjTypes+Coordinate)
   intArray:01b06e1c (System.Int32[], Elements: 5)
   strArray:01b06e3c (System.String[], Elements: 5)
```

我们可以看到，在类型实例中包含了一个 coordinate 对象、一个正数数组、一个字符串

数组。我们已经知道在 coordinate 中包含了 x，y 和 z 等成员。要使 mdt 命令输出这些值，可以使用 -r 开关：

```
0:000> !mdt 0x1b06d64 -r
01b06d64 (Advanced.NET.Debugging.Chapter3.ObjTypes)
   coordinate:01b06d70 (Advanced.NET.Debugging.Chapter3.ObjTypes+Coordinate)
      xCord:0x64 (System.Int32)
      yCord:0x64 (System.Int32)
      zCord:0x80000000 (System.Int32)
   intArray:01b06e1c (System.Int32[], Elements: 5)
   strArray:01b06e3c (System.String[], Elements: 5)
```

现在，我们可以很清楚地看到 ObjTypes 类型中 coordinate 成员的值。

3.12.5 自动死锁检测

在开发多线程的程序时，死锁是一种很常见的问题。死锁的基本定义为：当两个线程各自持有不同的同步资源，并且彼此都在等待对方释放所持有的同步资源时，这两个线程都无法继续执行，这种情况就称之为死锁。分析死锁的关键在于，要正确识别出在死锁中涉及的各个线程，以及各个线程持有的同步资源。在识别死锁问题时，有一种手动方法是对进程中的每个线程进行转储，然后对调用栈进行分析以查看它是否在等待某一个锁，并且记录这个锁的信息。对于每个正在等待锁的线程，要找出哪个线程持有这个锁，并接着分析持有该锁的线程。最终，我们将找出出现问题的各个线程以及相应的锁。虽然这种方法能够找出哪些线程持有哪些锁，但却比较费力，因为通过手动方式来转储出进程中所有线程需要花费大量的时间并且很容易产生错误。幸运的是，在 SOS 和 SOSEX 调试器扩展中都包含了一些命令可以使这个过程变得更简单。

在第 2 章中已经介绍过，每个对象都包含了一个 syncBlk（同步块）。在同步块中包含了对象的锁状态信息（以及其他信息）。通过 syncblk 命令，我们可以显示在进程中持有的所有锁的信息，以及持有每个锁的线程的信息。当我们知道了哪些线程在哪些锁上等待以及哪些线程持有这些锁时，就可以使用 ClrStack 命令来进一步获得各个线程的信息，并找出为什么线程没有释放所持有锁的原因。下面是在发生死锁的进程上运行 syncblk 的示例输出：

```
0:005> !syncblk
Index SyncBlock MonitorHeld Recursion Owning Thread Info  SyncBlock Owner
    1 001c999c            3         1 001a5130  1af4   0   01ffab84 System.Object
    3 001c99fc            3         1 001c9380  2370   3   01ffab90 System.Object
-----------------------------
Total           3
CCW             0
RCW             0
ComClassFactory 0
Free            0
```

在 syncblk 命令的输出中，有两个对象处于锁定状态，此外还包括关于当前持有这些锁的线程的一些信息。根据输出的信息，现在可以使用 ClrStack 命令来获得每个持有锁的线程的栈回溯，并通过代码审查找出可能发生死锁的位置。在第 6 章中将进一步讨论 synblk 命令的输出。

syncblk 命令的输出使我们能够缩小在进程中搜索死锁的范围，但仍然需要大量的手动工作来找出哪些线程正在哪些锁上等待。SOSEX 调试器扩展更进一步增强了死锁检测的功能，并引入了 dlk 命令，这个命令可以把找出哪些线程可能发生死锁的过程自动化。使用前面相同的示例，dlk 的输出如下所示：

```
0:005> !dlk
Deadlock detected:
CLR thread 1 holds sync block 001c999c OBJ:01ffab84[System.Object]
             waits sync block 001c99fc OBJ:01ffab90[System.Object]
CLR thread 3 holds sync block 001c99fc OBJ:01ffab90[System.Object]
             waits sync block 001c999c OBJ:01ffab84[System.Object]
CLR Thread 1 is waiting at DeadLock!DeadLock.Program.Run()(+0x59 IL)(+0x87 Native)
CLR Thread 3 is waiting at DeadLock!DeadLock.
Program.DoWork()(+0x1d IL)(+0x25 Native)

1 deadlock detected.
```

我们可以很清楚地看到每个拥有锁的线程的详细信息，以及这个线程正在等待哪个锁。在这种情况下，线程 1 在对象 0x01ffab84 上持有一个锁并且正在等待获得对象 0x01ffab90 上的锁。接下来，线程 3 持有对象 0x01ffab90 上的锁（即线程 1 正在等待的锁），并且正在等待对象 0x01ffab84（这是由线程 1 持有的锁）。此外，命令 dlk 还给出了一个栈帧，其中每个线程都在等待锁被释放。最后一部分输出的信息是 dlk，指出已经检测到了一个死锁。你可以看到，命令 dlk 把死锁检测过程中的大部分手动工作都自动化了，因此在分析死锁的情况中应该尽可能地使用这个命令。

在本节中，我们简要介绍了在分析同步问题时可以使用的一些命令。具体来说，我们给出了一个简单的死锁问题，并且介绍了如何使用 syncblk 和 dlk 命令来获得进程中关于线程和锁的详细信息。在第 6 章中，我们会分析一些常见的同步问题，那时将进一步使用和解释这些命令。

3.12.6 托管堆与垃圾收集命令

SOSEX 调试器命令增强了 SOS 中的托管堆功能，并且提供了额外的一些命令，这些命令使得找出与托管堆和垃圾收集相关的问题更为容易。我们在本节中将简要介绍这些命令，并在第 5 章中进一步讨论它们，因此如果还不理解这里提到的一些概念，暂时也不必担心，后面会详细地对它们进行解释。

我们介绍的第一个命令是 gcgen。gcgen 命令的参数是托管堆上的对象地址，它能显示该

对象所属的代。例如，位于地址 0x01ffab64 上的对象的代可以通过以下命令来得到：

```
0:000> !gcgen 0x01ffab64
Gen 0
```

这个命令的输出表示这个对象属于第 0 代。当要找出一个对象的存活时间（或者它已经经过了多少次垃圾收集过程）时，这个命令将非常方便。

下一个命令与 gcgen 命令相关，叫做 dumpgen。dumpgen 命能输出指定代的所有对象。例如，运行 dumpgen 0 将给出以下（简化后的）输出：

```
0:000> !dumpgen 0
01fa1018              12 **** FREE ****
01fa1024              72 System.OutOfMemoryException
01fa106c              72 System.StackOverflowException
01fa10b4              72 System.ExecutionEngineException
01fa10fc              72 System.Threading.ThreadAbortException
01fa1144              72 System.Threading.ThreadAbortException
01fa118c              12 System.Object
01fa1198              28 System.SharedStatics
01fa11b4              34 System.String     STRVAL=c:\Zone\
01fa11d8              64 System.String     STRVAL=c:\Zone\DeadLock.config
01fa1218             100 System.AppDomain
TRVAL=file:///c:\Zone\DeadLock.exe
01fa1644              20 System.Security.Policy.Evidence
01fa1658              16 System.Security.Policy.Zone
01fa1668              24 System.Collections.ArrayList
01fa1680              16 System.Object[]
01fa1690              32 System.Collections.ArrayList+SyncArrayList
01fa16b0              12 System.Object
01fa16bc              32 System.Object[]
01fa16dc              12 System.Security.Policy.Url
01fa16e8              52 System.Security.Util.URLString
01fa171c              20 System.RuntimeType
01fa1730              64 System.IO.UnmanagedMemoryStream
01fa1770              36 System.Int64[]
01fa1794              12 System.Object
01fa17a0              32 Microsoft.Win32.Win32Native+OSVERSIONINFO
01fa17c0              46 System.String     STRVAL=Service Pack 1
01fa17f0              40 Microsoft.Win32.Win32Native+OSVERSIONINFOEX
01fa1818              46 System.String     STRVAL=Service Pack 1
```

dumpgen 命令的输出中包含了指定代中各个对象的地址，大小以及文本表示。

最后一个命令是 strings。strings 命令能够搜索托管堆上的任意字符串。在搜索过程中可以指定通配符、要搜索的代、最小长度和最大长度。下面给出了运行 strings 命令搜索字符串 Debug 的示例输出：

```
0:000> !strings -m:Debug
Address          Gen      Value
----------------------------
01fd0b04         0        System.Diagnostics.DebuggableAttribute+DebuggingModes
```

```
01fd11f8        0       DebuggingFlags
01fd15dc        0       m_debuggingModes
01fd1754        0       System.Diagnostics.DebuggableAttribute+DebuggingModes
790edba0        2       debug
790f0490        2       InvalidOperation_NotADebugModule
----------------------------
6 matching strings
```

在 strings 命令的输出中给出了已找到的所有字符串的地址，属于第几代以及字符串的值。要在指定的代中进行搜索，可以使用 -g 开关，此外还可以通过 -n 和 -x 等开关来指定最小长度和最大长度。

3.13 崩溃转储文件

到目前为止，我们分析的都是实时调试会话（live debug session）——“实时”意味着正在调试的是一个正在运行的物理进程，可以访问所有的进程状态和控制被调试进程的执行过程。有时候，实时调试会话并非总是可行的，因为无法访问被调试的机器。例如一些生产机器，这些机器位于被锁定的数据中心，并且进入许可非常严格。再例如，一些用户可能不愿意让调试工程师将调试器附载到他们的进程，因为这么做可能导致停机并付出高昂的成本，还可能在这些机器上留下一些操作痕迹。在这些情况下，我们需要采用事后调试（postmortem debugging）的方法。事后调试的工作原理是抓取指定进程的快照，并且以离线方式对快照进行调试。快照只是一个二进制文件，其中包含了抓取快照时进程所处的状态。在抓取了快照后，可以将这些快照交给工程师进行调试。我们可以使用在前面用到的非托管调试器来调试这些快照，只不过有些命令可能因为不是对实时目标进行调试而无法工作。

我们来看一个生成和调试快照的示例。在这个示例中，我们使用调试器本身来创建快照。还有其他一些功能强大的工具也可以用来抓取快照，我们将在第 8 章中更详细地介绍它们。首先在调试器下启动 03ObjTypes.exe 程序，并且恢复程序的执行直到调试器由于一个 CLR 异常而中断。此时，可以使用 dump 命令来生成一个转储文件，如下所示：

```
Press any key to continue (Exception)
 (2088.177c): CLR exception - code e0434f4d (first chance)
(2088.177c): CLR exception - code e0434f4d (!!! second chance !!!)
eax=0030f070 ebx=e0434f4d ecx=00000001 edx=00000000 esi=0030f0f8 edi=00135408
eip=777442eb esp=0030f070 ebp=0030f0c0 iopl=0         nv up ei pl nz ac po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000212
*** ERROR: Symbol file could not be found.
Defaulted to export symbols for C:\Windows\system32\KERNEL32.dll -
KERNEL32!RaiseException+0x58:
777442eb c9              leave
0:000> .dump /ma c:\dump.dmp
Creating c:\dump.dmp - mini user dump
Dump successfully written
0:000>
```

dumg 命令的参数是一个文件名，表示保存转储信息的文件。此外，dump 命令还可以带有一系列不同的开关，这些开关控制着将哪些进程状态保存到转储文件中。当然，保存的进程状态越多，那么调试的成功概率也就越大。我们将在第 8 章中介绍这些选项。现在，我们获得了一个转储文件，可以使用同一个调试器来调试它。要调试这个转储文件，我们要告诉调试器正在调试的是一个快照。首先使用调试开关 -z，紧接着是转储文件的路径，如下所示：

```
ntsd -z c:\dump.dmp
```

这个命令会使调试器读取转储文件，然后给出所有熟悉的调试提示符，如下所示：

```
Microsoft (R) Windows Debugger Version 6.9.0003.113 X86
Copyright (c) Microsoft Corporation. All rights reserved.


Loading Dump File [c:\dump.dmp]
User Mini Dump File: Only registers, stack and portions of memory are available
Symbol search path is: *** Invalid ***
Executable search path is:
Windows Server 2008 Version 6001 (Service Pack 1) MP (2 procs) Free x86 compatible
Product: WinNt, suite: SingleUserTS
Debug session time: Mon Dec  8 16:49:17.000 2008 (GMT-8)
System Uptime: not available
Process Uptime: 0 days 0:00:58.000
.....................
This dump file has an exception of interest stored in it.
The stored exception information can be accessed via .ecxr.
(2088.177c): CLR exception - code e0434f4d (first/second chance not available)
eax=0030f070 ebx=e0434f4d ecx=00000001 edx=00000000 esi=0030f0f8 edi=00135408
eip=777442eb esp=0030f070 ebp=0030f0c0 iopl=0         nv up ei pl nz ac po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000212
*** ERROR: Symbol file could not be found.
Defaulted to export symbols for kernel32.dll -
kernel32!RaiseException+0x58:
777442eb c9              leave
0:000>
```

在上面的调试输出中，第一部分信息表示调试器加载了一个转储文件，并且转储文件的类型是一个微型转储文件（mini dump file）。微型转储只是众多转储文件类型中的一种，它包含了有限的进程状态信息。接下来的重要信息是，在转储文件中包含了一个异常，并且这个异常信息可以通过 ecxr 命令来提取。执行 ecxr 命令，可以看到与这个异常相关的信息，例如在发生异常时寄存器的值，通过这些信息就可以获得线程栈信息。

在第 8 章中我们将更详细地介绍事后调试，也会看到一些功能更为强大的工具，这些工具能更容易地帮助生成转储文件并通过离线方式来调试问题。

3.14 小结

在本章中介绍了大量的基础知识，包括一些最基本的和最常见的调试任务和命令。本章还介绍了非托管调试器的基本功能和托管代码调试器扩展，并简要讨论了事后调试。本章可以作为对众多命令的一个简介。在随后的章节中，当分析实际的错误时，将更深入地介绍这些命令以及使用它们来分析各种错误的原因。

第二部分 调试实践

第4章 程序集加载器

到目前为止，对CLR的大多数讨论都是围绕着程序集展开的，程序集也是.NET的基础构件。第2章和第3章曾介绍了程序集的内部结构，但到目前为止，我们并没有介绍如何将.NET程序集加载到应用程序域中。.NET平台的目标之一就是消除所谓的DLL地狱（DLL Hell）问题。DLL地狱是指，软件在升级或降级的过程中错误地覆盖或删除了一些二进制文件。这种问题的表现形式通常是，应用程序无法启动或者应用程序表现出错误的行为（这种情况往往更为复杂）。为了避免DLL地狱问题，CLR为程序集的加载和管理等过程定义了严格的规则。在本章中，我们将介绍CLR加载器（代码名字为fusion），以及它是如何避免DLL地狱问题和与CLR加载器相关的其他一些陷阱。在本章中，我们还将通过调试器以及相应的工具来分析当错误使用CLR加载器时出现的各种问题。

4.1 CLR加载器简介

CLR提供了一种非常复杂的机制来加载.NET程序集。从宏观来看，.NET程序集可以是共享的或者私有的。共享的.NET程序集是指可以由同一台机器上的两个或者多个应用程序使用的程序集。共享程序集的好处是显而易见的，每个应用程序无需在本地路径中维持程序集，就可以共享程序集，因此只需在一个地方管理和维护它们。通常，共享程序集会被安装到全局程序集缓存（Global Assembly Cache，GAC）中。而私有程序集是指对于某个应用程序私有的程序集，它们通常保存在与应用程序本身路径相同（虽然这并非一种强制要求）的文件夹中（或者预定的文件夹中）。在面对不同类型的程序集时，CLR加载器如何判断从哪个位置加载程序集？答案在于加载上下文（Load Context）。图4-1给出了CLR程序集的加载过程。

在请求加载一个程序集时，CLR加载器将判断加载上下文（即这个加载请求是如何发出的），并且根据上下文采取不同的算法来判断程序集的位置。在进一步讨论各种加载上下文之前，首先要理解程序集标识（Assembly Identity）的概念。

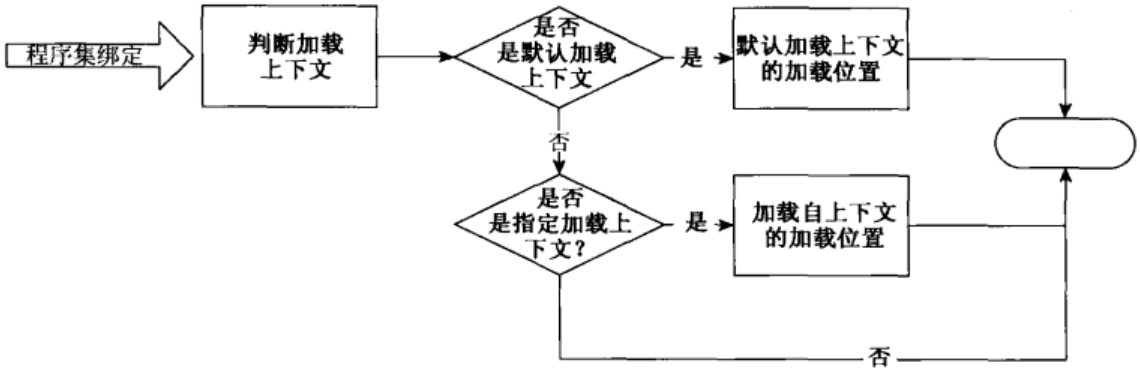


图 4-1　CLR 加载器流程概览

4.1.1　程序集标识

程序集标识定义了程序集的唯一性。当 CLR 判断加载哪个程序以及判断两个或多个已加载的程序集是否相等时，这个标识非常重要。考虑一个常见的程序集 workflow. dll。由于这个程序集的名字很常见，因此在系统中可能有多个不同的程序集的名字都是 workflow. dll。而且，如果有多个程序集使用了不同的 workflow. dll 实例，那么应用程序如何使用正确的 workflows. dll 实例，或者说 CLR 选择哪一个程序集来加载？回答这个问题需要考虑许多因素，但其中一个关键因素就是程序集标识。程序集标识由以下关键部分组成：

- 程序集的名字
- 文化
- 版本
- 公钥
- 处理器架构

程序集名字是指程序集的简单名字，通常是程序集的文件名（除去扩展名. exe 或者 . dll）。程序集的文化是指程序集的目标地域（例如中性文化）。版本属性表示程序集的版本，语法形式如下：< major >. < minor >. < build >. < revision >。

公钥属性是用于强名字的程序集，包含了公钥的 64 位散列值，这个公钥与用于对程序集签名的私钥是相对应的。最后，在 CLR2. 0 中引入的处理器架构属性指定了程序集的目标处理器架构。这五个属性合在一起就构成了程序集标识。在 CLR 中广泛地使用程序集标识来判断被加载的程序集的正确与否。如果所有五个属性都被指定了，并且使用默认加载上下文来加载程序集，那么 CLR 在判断应该加载哪个程序集时不会出现二义性。

需要注意的是，就程序集标识而言，当加载正确的程序集时，程序集的路径并不是 CLR 考虑的因素。如果两个相同的程序集位于不同的路径中，那么它们可以加载到同一个应用程序域中（虽然使用了不同的加载上下文）。即使这两个程序集是相同的，它们仍然

被视为两个不同的程序集。因此，在程序集中包含的类型同样被认为是不同的。这可能带来许多奇怪的类型转换异常。让我们来看一个示例，在清单 4-1 中给出了这种行为的示例程序。

清单 4-1　使用两个相同类型的应用程序

```
namespace Advanced.NET.Debugging.Chapter4
{
    class SimpleType
    {
        private int field1;
        private int field2;

        public int Field1
        {
            get { return field1; }
        }

        public int Field2
        {
            get { return field2; }
        }
        public SimpleType()
        {
            field1 = 10;
            field2 = 5;
        }
    }

    class TypeCast
    {
        static void Main(string[] args)
        {
            Assembly asmLoadFromContext = null;

            Console.WriteLine("Press any key to load into load from context");
            Console.ReadKey();

            asmLoadFromContext = Assembly.LoadFrom("04assembly.dll");

            SimpleType s = (SimpleType)asmLoadFromContext.CreateInstance(
                                "Advanced.NET.Debugging.Chapter4.SimpleType");

            Console.WriteLine("Press any key to exit");
            Console.ReadKey();
        }
    }
}
```

清单 4-1 中的源代码和二进制文件位于以下文件夹中：

- 源代码文件：C:\ADND\Chapter4\TypeCast
- 二进制文件：C:\ADNDBin\04TypeCast.exe

清单 4-1 中的源代码非常简单。在代码的第一部分定义了一个简单的类型 SimpleType。SimpleType 只有一个构造函数，它将初始化两个域。在接下来的代码中定义了一个类型 TypeCast，它包含一个 Main 方法。在 Main 方法中将通过指定加载上下文（在本章的后面将进行讨论）来加载程序集 04assembly.dll。清单 4-2 给出了 04assembly.dll 的代码。

清单 4-2　04assembly.dll 的源代码

```
using System;
using System.Text;
namespace Advanced.NET.Debugging.Chapter4
{
        class SimpleType
        {
            private int field1;
            private int field2;
            public int Field1
        {
             get { return field1; }
        }

        public int Field2
        {
            get { return field2; }
        }

        public SimpleType()
        {
            field1 = 10;
            field2 = 5;
        }
    }
}
```

清单 4-2 中的源代码和二进制文件位于以下文件夹中：

- 源代码文件：C:\ADND\Chapter4\TypeCast
- 二进制文件：C:\ADNDBin\04assembly.dll

可以看到，在清单 4-2 中的代码同样定义了一个类型 SimpleType（在与清单 4-1 相同的命名空间中）。事实上，这两个类型本质上是相同的，都带有一个构造函数和两个简单的域。在清单 4-1 的 Main 方法中，我们可以看到接下来的语句是创建一个在清单 4-2 中定义的 SimpleType 的实例。然后，将这个创建的实例赋值给一个 SimpleType 类型（在清单 4-1 中定义的）的

引用变量。由于这两个类型是相同的，因此这种类型转换没有问题。现在，我们来运行这个程序并看看会发生什么情况：

```
C:\ADNDBin>04TypeCast.exe
Press any key to load into load from context

Unhandled Exception: System.InvalidCastException: Unable to cast object of type
'Advanced.NET.Debugging.Chapter4.SimpleType' to type
'Advanced.NET.Debugging.Chapter4.SimpleType'.
   at Advanced.NET.Debugging.Chapter4.TypeCast.Main(String[] args) in
c:\Publishing\ADND\Code\Chapter4\TypeCast\04TypeCast.cs:line 41
```

值得注意的是，这个程序退出并抛出了一个异常 InvalidCastException。具体来说，这个异常表明了从类型 Advanced. NET. Debugging. Chapter4. SimpleType 到类型 Advanced. NET. Debugging. Chapter4. SimpleType 的转换操作是无效的。虽然这两个类型本质上是相等的，但 CLR 似乎将它们视为不同的类型定义，这正是混合使用不同加载上下文带来的危险性之一。在本示例中，我们在 04TypeCast. exe 中定义了一个类型，并在 04assembly. dll 中定义了一个相同的类型。而且，我们通过 Assembly. LoadForm 这个 API 加载 04assembly. dll，这表示这个程序集是在指定加载上下文中加载的，因而带有相同类型的两个不同程序集就加载到同一个应用程序域中，但每个类型却视为独立的实体。

可以看到，为了避免混合上下文加载的问题，了解程序集标识以及 CLR 的加载过程是非常重要的。在进一步讨论不同的加载上下文之前，我们首先要理解在 CLR 中引入的共享程序集机制，也叫做全局程序集缓存。

4.1.2　全局程序集缓存

我们曾经介绍了全局程序集缓存（GAC）的概念，但没有给出详细的定义。GAC 是一个保存所有共享程序集的常用位置。GAC 位于：

```
%windir%\assembly
```

在 GAC 文件夹下面是一组子文件夹：

- GAC. GAC 文件夹中包含了来自. NET 1. x 版本的程序集。
- GAC_32。包含了所有的 32 位程序集。
- GAC_64。包含了所有的 64 位程序集。

注意，如果运行的是 32 位版本的 Windows，那么没有这个文件夹。然而，如果运行 64 位版本的 Windows，那么 GAC_64 和 GAC_32 这两个文件夹将同时存在。

- GAC_MSIL。与架构无关的程序集。
- NativeImages_ < version > _ < architecture > 。这个文件夹包含的是预编译的共享程序集。

以 GAC 开头的文件夹是最重要的子文件夹，这些文件夹包含了目前所有的共享程序集。每

个共享程序集都有各自的子文件夹层次结构，如图 4-2 所示。

在 GAC* 文件夹下的每个文件夹都以共享程序集来命名。例如，在我的机器上，GAC 文件夹下有一个文件夹 System. Data，里面包含了指定共享程序集的所有不同版本。那么 GAC 如何对同一个共享程序的多个版本进行区分？

你可能会认为是通过程序集的标识。在每个程序集子文件夹下是另一个以程序集标识来命名的子文件夹。标识的形式类似于在 4. 1. 1 一节中介绍的形式：

```
<version>__<public key>
```

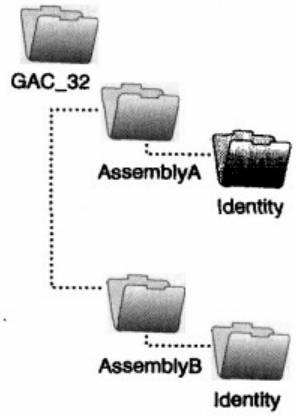


图 4-2　GAC 子文件夹层次结构

在这个标识文件夹下可以找到实际的程序集。总体来说，这就是 CLR 如何控制共享程序集不同版本的方式。每个程序集及其标识都位于通用的 GAC 文件夹中。注意，每个共享程序集都必须被强命名并且拥有唯一的版本号。

在共享程序集加载过程中（在默认的加载上下文中），CLR 加载器会首先检查 GAC 来确定这个程序集是否存在，然后再查看私有的（即使是预定义的）加载路径。你可以在调试器中通过命令 lm f（List Modules）来检查一个程序集是从什么位置被加载的。命令 lm f 还给出了目标进程中所有已加载的模块以及它们相应的加载路径：

```
0:004> lm f
start    end        module name
00710000 008c8000   mmc      C:\Windows\system32\mmc.exe
…
…
…
07030000 075b8000   System_Xml_ni C:\Windows\assembly\NativeImages_v2.0.50727_32\
System.Xml\02cf61328d59df9b3ec09544f449a781\System.Xml.ni.dll
…
…
…
4f400000 4f49c000   Microsoft_ManagementConsole_ni
C:\Windows\assembly\NativeImages_v2.0.50727_32\
Microsoft.Managemen#\b059345a9c2a126e320e17c2090dd354\Microsoft
.ManagementConsole.ni
.dll
…
…
…
C:\Windows\assembly\NativeImages_v2.0.50727_32\System.ServiceProce#\
80a3d0416c6660b86e245bd1f6b66fd8\System.ServiceProcess.ni.dll
```

我们可以看到，有多个程序集都是从 GAC 中的非托管映像文件夹中加载的。

通过 Windows 资源管理器来浏览 GAC

你还可以通过 Windows 资源管理器来浏览 GAC 的内容。当使用 Windows 资源管理器时，你可以查看定制的 GAC 视图，它是由 shell 扩展 shfusion. dll 实现的。

当处理依赖的程序集时，程序集的标识保存在主程序集的清单中，CLR 将通过这个清单来找出正确的程序集并加载。例如，下面（通过 ILDAMS 给出的输出）给出了 04TypeCase. exe 程序集清单（在这个清单中定义了依赖的程序集）：

```
.assembly extern mscorlib
{
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )                    // .z\V.4..
  .ver 2:0:0:0
}
```

我们可以看到，04TypeCast. exe 依赖于程序集 mscorlib，并且程序集的标识包含了公钥令牌和版本号。

请注意，尽管依赖的程序集是通过程序集标识来识别的，但也可以通过一些策略将依赖程序集重定向到其他有着不同标识的程序集版本。

现在，让我们来看看不同的加载上下文，并详细介绍 CLR 使用哪种加载算法来绑定到指定的程序集。

4.1.3　默认加载上下文

大多数程序集都使用默认加载上下文（Default Load Context），这也是最安全的方式，可以避免由于加载不正确的程序集而产生的问题。使用默认加载上下文的程序集是通过 Assembly. Load 及其变化形式来加载的。最安全的程序集加载方式意味着什么？在默认的加载上下文中，CLR 会执行所有的探测逻辑（见图 4-3），从而确保加载正确版本的程序集。此外，依赖的程序集也会在这个上下文中被自动找到。这与指定加载上下文（Load-From）或者无加载上下文（Load-Without）是不同的，在这些上下文中，调用者可以显式地选择程序集，因此更可能出现选择不正确版本的情况。在图 4-3 中给出了 CLR 在默认加载上下文中的探测逻辑。

从图 4-3 中可以看到，当一个程序集加载请求进入默认加载上下文时，CLR 加载器会首先判断这个程序集是否位于 GAC 中。如果是，CLR 加载器将直接根据所请求的程序集标识加载正确的版本。如果程序集不在 GAC 中，那么 CLR 加载器将探测其他路径，包括应用程序库所在的路径和私有二进制文件的路径等。如果在这两个位置的某一个中找到了该程序集，那么 CLR 加载器就会从这个位置加载。

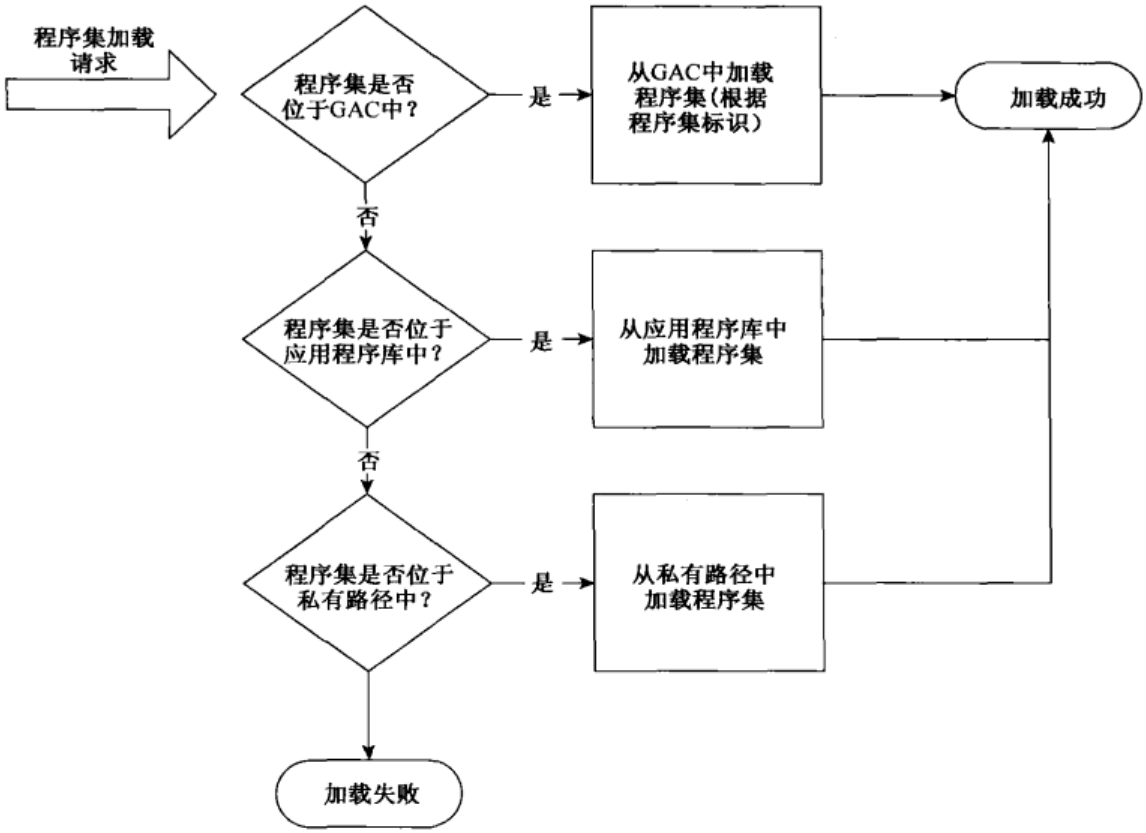


图 4-3 默认的加载上下文探测逻辑

4.1.4 指定加载上下文

当一个程序集被加载到指定加载上下文（Load-From Context）中时，通常是通过 Assembly. LoadFrom、AppDomain. CreateInstanceFrom、AppDomain. ExecuteAssembly 等 API 的某种变化形式来实现的，此时 CLR 不会执行探测逻辑，而是由调用者来负责消除程序集冲突。程序集依赖的所有其他程序集将从同一个路径加载。此外，被加载到指定加载上下文中的程序集可以使用默认加载上下文中的程序集。在图 4-4 中给出了指定加载上下文中一些值得注意的方面。

在图 4-4 中，我们可以看到标识相同但路径不同的程序集被视为同一个程序集，因此返回了一个引用指向已被加载的程序集。而且，如果一个程序集已经被加载到指定加载上下文，那么当再次尝试把同一个程序集加载到默认加载上下文时，就会导致一个错误。

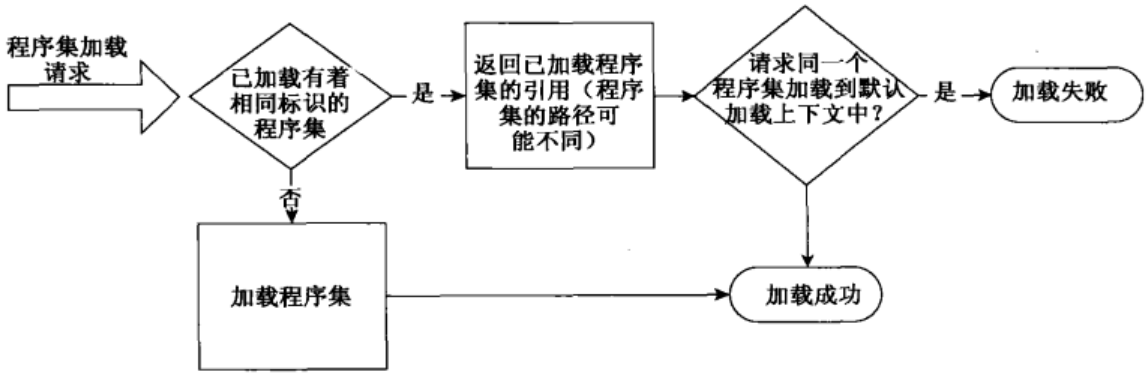


图 4-4　在使用指定加载上下文时值得注意的一些方面

4.1.5　无加载上下文

我们将要讨论的最后一个加载上下文是无加载上下文（Load-Without Context），即在加载时不存在上下文。这种上下文专门用于那些没有加载上下文的程序集。例如，一些通过 Reflection 命名空间和 Emit 生成的程序集。在这些情况下，CLR 不会执行任何探测操作。但是存在一种例外情况，即当通过策略把标识应用到生成的程序集上时，如果某个程序集位于 GAC 中，那么会使用这个程序集。

到目前为止，我们已经讨论了程序集标识，全局程序集缓存以及各种不同的加载上下文。当加载程序集时，可能会发生许多复杂和值得注意的问题（尤其是在混合使用多种加载上下文的情况下）。在本章的剩余部分，我们会分析一些实际问题，并介绍如何通过现有的工具、方法和调试器来分析问题的原因。

4.2　简单的程序集加载故障

我们首先来看一个简单的程序集加载故障，并介绍在分析程序集加载故障时可以使用的技术。我们使用与清单 4-1 和 4－2 中相同的程序，其中主程序集（04TypeCast. exe）只是加载另一个程序集 04Assembly. dll。这两个文件都位于目录 C:\adndbin 下。然而，在运行这个程序之前，请首先把 04assembly. dll 重命名为 04assembly. old。在重新命名后，可以继续执行程序集 04TypeCast. exe。

```
C:\ADNDBin>04TypeCast.exe
Press any key to load into load from context

Unhandled Exception: System.IO.FileNotFoundException: Could not load file or
assembly
'file:///C:\ADNDBin\04assembly.dll' or one of its dependencies.
The system cannot find the file specified.
```

```
File name: 'file:///C:\ADNDBin\04assembly.dll'
   at System.Reflection.Assembly._nLoad
(AssemblyName fileName, String codeBase, Evidence assemblySecurity, Assembly
locationHint,
StackCrawlMark& stackMark, Boolean throwOnFileNotFound, Boolean forIntrospection)
   at System.Reflection.Assembly.nLoad(AssemblyName fileName, String codeBase,
Evidence assemblySecurity, Assembly locationHint, StackCrawlMark& stackMark, Boolean
throwOnFileNotFound, Boolean forIntrospection)
   at System.Reflection.Assembly.InternalLoad(AssemblyName assemblyRef, Evidence
assemblySecurity, StackCrawlMark& stackMark, Boolean forIntrospection)
   at System.Reflection.Assembly.InternalLoadFrom(String assemblyFile, Evidence
securityEvidence, Byte[] hashValue, AssemblyHashAlgorithm hashAlgorithm, Boolean
forIntrospection, StackCrawlMark& stackMark)
   at System.Reflection.Assembly.LoadFrom(String assemblyFile)
   at Advanced.NET.Debugging.Chapter4.TypeCast.Main(String[] args) in
c:\Publishing\ADND\Code\Chapter4\TypeCast\04TypeCast.cs:line 39

WRN: Assembly binding logging is turned OFF.
To enable assembly bind failure logging, set the registry value
[HKLM\Software\Microsoft\Fusion!EnableLog] (DWORD) to 1.
Note: There is some performance penalty associated with assembly bind failure
logging.
To turn this feature off, remove the registry value
[HKLM\Software\Microsoft\Fusion!EnableLog].
```

在这个命令的输出中抛出了一个 FileNotFoundException 异常。虽然这些输出信息很有用，但通常需要能在调试器中收集关于这个故障的更多信息，而不仅仅是一个栈回溯。在调试器下重新运行这个程序，并执行命令 sxe 0xe0434f4d，当有任何 CLR 异常发生时都将停止执行。在清单 4-3 中给出了调试器会话。

清单 4-3　调试 FileNotFoundException

```
...
...
...
0:000> sxe 0xe0434f4d
0:000> g
ModLoad: 75e70000 75f36000   C:\Windows\system32\ADVAPI32.dll
ModLoad: 76330000 763f3000   C:\Windows\system32\RPCRT4.dll
...
...
...
ModLoad: 79060000 790b6000   C:\Windows\Microsoft.NET\Framework\v2.0.50727
\mscorjit.dll
Press any key to load into load from context
 ModLoad: 60340000 60348000   C:\Windows\Microsoft.NET\Framework\
v2.0.50727\culture.dll
(d1c.126c): C++ EH exception - code e06d7363 (first chance)
(d1c.126c): C++ EH exception - code e06d7363 (first chance)
```

```
(d1c.126c): C++ EH exception - code e06d7363 (first chance)
(d1c.126c): CLR exception - code e0434f4d (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0023efdc ebx=e0434f4d ecx=00000001 edx=00000000 esi=0023f064 edi=00345408
eip=773e42eb esp=0023efdc ebp=0023f02c iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
KERNEL32!RaiseException+0x58:
773e42eb c9              leave
0:000> kb
ChildEBP RetAddr  Args to Child
0023f02c 79f071ac e0434f4d 00000001 00000001 KERNEL32!RaiseException+0x58
0023f08c 79f9293a 01c978f8 00000000 00000000
mscorwks!RaiseTheExceptionInternalOnly+0x2a8
0023f0c4 79f933b6 0023f180 0035f260 82cf8ddc
mscorwks!UnwindAndContinueRethrowHelperAfterCatch+0x70
*** WARNING: Unable to verify checksum for
C:\Windows\assembly\NativeImages_v2.0.50727_32\mscorlib\5b3e3b0551bcaa722c27dbb089c4
31e4\mscorlib.ni.dll
0023f1fc 7937dd77 00000000 00000001 0023f284 mscorwks!AssemblyNative::Load+0x2d0
0023f2b0 79e7c74b 00000000 0023f2e8 0023f340 mscorlib_ni+0x2bdd77
0023f2c0 79e7c6cc 0023f390 00000000 0023f360 mscorwks!CallDescrWorker+0x33
0023f340 79e7c8e1 0023f390 00000000 0023f360
mscorwks!CallDescrWorkerWithHandler+0xa3
0023f478 79e7c783 002cc028 0023f540 0023f50c mscorwks!MethodDesc::CallDescr+0x19c
0023f494 79e7c90d 002cc028 0023f540 0023f50c
mscorwks!MethodDesc::CallTargetWorker+0x1f
0023f4a8 79eefb9e 0023f50c 82cf8a2c 00000000 mscorwks!MethodDescCallSite::Call+0x18
0023f60c 79eef830 002c3028 00000001 0023f648 mscorwks!ClassLoader::RunMain+0x263
0023f874 79ef01da 00000000 82cf8164 00000001
mscorwks!Assembly::ExecuteMainMethod+0xa6
0023fd44 79fb9793 00290000 00000000 82cf81b4
mscorwks!SystemDomain::ExecuteMainMethod+0x43f
0023fd94 79fb96df 00290000 82cf81fc 00000000 mscorwks!ExecuteEXE+0x59
0023fddc 7900b1b3 00000000 79e70000 0023fdf8 mscorwks!_CorExeMain+0x15c
0023fdec 773e4911 7ffd7000 0023fe38 7725e4b6 mscoree!_CorExeMain+0x2c
0023fdf8 7725e4b6 7ffd7000 7a3b7683 00000000 KERNEL32!BaseThreadInitThunk+0xe
0023fe38 7725e489 7900b183 7ffd7000 00000000 ntdll!__RtlUserThreadStart+0x23
0023fe50 00000000 7900b183 7ffd7000 00000000 ntdll!_RtlUserThreadStart+0x1b
0:000> .loadby sos mscorwks
0:000> !PrintException 01c978f8
Exception object: 01c978f8
Exception type: System.IO.FileNotFoundException
Message: Could not load file or assembly 'file:///C:\ADNDBin\04assembly.dll'
or one of its dependencies. The system cannot find the file specified.
InnerException: <none>
StackTrace (generated):
<none>
StackTraceString: <none>
HResult: 80070002
```

从清单 4-3 中可以看到，我们首先恢复了程序的执行，并直到发生异常时再停止程序的执行。当程序停止执行时，将调用栈转储出来，并从第二个栈帧的第一个参数（mscorwks! RaiseTheExceptionInternalOnly）中找到托管异常的地址。然后，将这个地址传递给 PrintException 命令以得到进一步的信息。需要注意的是，这个异常会列出一些关于故障的基本信息（例如哪个程序集加载失败以及相应的 HRESULT 值）。

如何通过 HRESULT 值得到进一步的信息

调试器包括一个命令 error，这个命令可以用来获得 HRESULT 的文本表示。将 HRESULT 传递给 error 命令，会显示以下信息：

```
0:000> !error 80070002
Error code: (HRESULT) 0x80070002 (2147942402) - The system cannot find the
file specified.
```

我们需要注意加载器实际加载程序集的位置。我们已经知道了其中一个路径（C:\adndbin\04assembly.dll），但这个程序集也可能位于其他地方。于是，这个问题就变成了：CLR 加载器能否告诉我们在查找程序集时使用的各个探测路径？答案是肯定的，在程序集绑定日志（assembly binding logging）中包含了这些信息，这是 CLR 的一个功能，即在绑定阶段进行跟踪。绑定日志功能在默认情况下是关闭着的，因此需要首先启用它。启用日志绑定的最简单方式是运行.NET 2.0 SDK 中的 fuslogvw.exe 工具。这个工具位于安装文件夹的 bin 目录下。例如，在我的机器上，这个文件的位置是：

```
c:\Program Files\Microsoft Visual Studio 8\SDK\v2.0\Bin
```

在图 4-5 中给出了启动工具后的主界面。

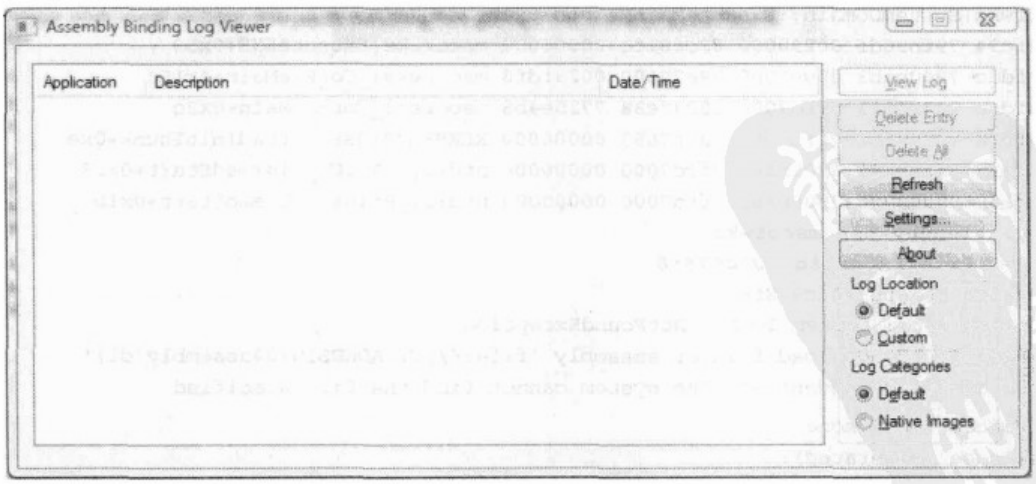


图 4-5 fuslogvw.exe 工具的主界面

主界面分为两个部分。第一部分包含了实际的日志项，共有三列：

- Application。该列给出日志项是来自哪个应用程序。
- Description。该列给出的是日志项的描述信息。
- Date/Time。该列给出的是日志项的时间戳。

在主界面右边包含了一组按钮，控制现有的日志项以及如何执行日志操作。Setting 按钮控制着如何执行日志记录以及何时执行，如图 4-6 所示。

图 4-6　Fuslogvw. exe 选项

有一组选项控制着如何执行日志操作。在默认情况下，日志操作是被禁用的，但我们也可以将其设置为以下行为：

- Log in Exception Text。当选择这个选项时，程序集绑定操作会被记录在异常中。
- Log Bind Failures to Disk。当选择这个选项时，只有发生故障的情况才会被记录到磁盘上。
- Log All Binds to Disk。当选择这个选项时，所有绑定操作都会被记录到磁盘上。
- Enable Custom Log Path。改变写入日志的路径。如果改变了日志路径，那么必须同时选择主界面上的 Custom Log Location 按钮。

自动启用绑定日志

在 fuslogvw. exe 中的这些设置是通过注册表中的以下注册键来实现的：

```
HKLM\Software\Microsoft\Fusion
```

在这个注册表路径下存在以下合法的注册键值：

- LogPath［SZ］。日志路径。
- ForceLog［DWORD = 1］。记录所有的绑定行为。
- LogFailures［DWORD = 1］。只记录失败的绑定行为。
- LogResourceBinds［DWORD = 1］。只记录卫星程序集[⊖]中的故障。

通过创建正确的注册键值，我们同样可以控制绑定日志功能。

我们通过启用“Log Bind Failures to Disk”这个设置来查看一个绑定日志功能的示例。当设置了这个选项时，运行 04typecast. exe 直到程序抛出异常。回到 fuslogvw. exe 工具并点击“Refresh”按钮。这时，会在 fuslogvw. exe 窗口中出现一个日志项。

```
04TypeCast.exe file:///C:/ADNDBIN/04Assembly.dll 1/1/2009/@1:32:08 PM
```

⊖ Satellite Assembly，一种不同主程序集的程序集，通常仅包含资源内容。——译者注

这个日志项对应于在加载 04Assembly. dll 时发生的故障。到目前为止，我们得到的故障信息非常少。要得到进一步的故障信息，双击日志项，会弹出默认的浏览器并给出详细的错误信息，参见图 4-7 中。

```
*** Assembly Binder Log Entry  (1/1/2009 @ 1:32:08 PM) ***

The operation failed.
Bind result: hr = 0x80070002. The system cannot find the file specified.

Assembly manager loaded from:  C:\Windows\Microsoft.NET\Framework\v2.0.50727\mscorwks.dll
Running under executable  C:\ADNDBin\04TypeCast.exe
--- A detailed error log follows.

=== Pre-bind state information ===
LOG: User =          marioh
LOG: Where-ref bind. Location = C:\ADNDBin\04assembly.dll
LOG: Appbase = file:///C:/ADNDBin/
LOG: Initial PrivatePath = NULL
LOG: Dynamic Base = NULL
LOG: Cache Base = NULL
LOG: AppName = 04TypeCast.exe
Calling assembly : (Unknown).
===
LOG: This bind starts in LoadFrom load context.
WRN: Native image will not be probed in LoadFrom context. Native image will only be probed in default load context, like
LOG: No application configuration file found.
LOG: Using machine configuration file from C:\Windows\Microsoft.NET\Framework\v2.0.50727\config\machine.config.
LOG: Attempting download of new URL file:///C:/ADNDBin/04assembly.dll.
LOG: All probing URLs attempted and failed.
```

图 4-7　详细的绑定日志错误信息

输出信息可以分为三个主要部分。第一部分信息包含了关于日志项的一般信息，例如操作状态（在这里是 failed）、HRESULT（包括文本表示）、CLR 加载器组件的路径（mscorwks. dll）以及可执行程序的路径。

下一部分信息包含了绑定之前（Pre-bind）的信息，这些信息值得我们注意。除了在运行程序时使用的用户名外，它还给出了尝试加载程序集的路径。在这部分信息中还包括了加载过程可能被探测的其他路径，例如应用程序库、私有路径、动态库以及缓存库等。这些不同路径表示 CLR 在绑定 04Assembly. dll 时探测的位置。

最后一部分输出的信息是实际的探测记录。从输出信息中可以得知它处于指定加载上下文中（因为使用了 LoadFrom）。这些信息还指出，没有找到应用程序配置文件，因此转而使用全局的配置文件。最后两行指出，在尝试从应用程序库目录中加载程序集时失败了。如果指定一个私有路径，那么 CLR 加载器同样会探测这个路径。

至此，我们完成了对一个简单程序集加载故障的分析。我们首先从应用程序发生故障时所输出的信息中得到了一些基本的诊断信息，接下来在调试器下运行这个有问题的程序，并观察抛出的异常，最后打开程序集绑定日志并找出 CLR 加载器将在哪些位置搜索被请求的程序集（在这种情况下，只有应用程序库这个路径）。

4.3　加载上下文故障

在本节中，我们将分析一个在加载依赖程序集时失败的问题。清单 4-4 给出了故障程序的源代码。请注意，为了将注意力主要放在调试方面而不是代码审查上，我们省略了其中一

部分源代码。

清单 4-4　存在程序集加载故障的应用程序

```
using System;
using System.Text;
using System.Runtime.Remoting;

namespace Advanced.NET.Debugging.Chapter4
{
    [Serializable]
    class Entity
    {
        public int a;
    }

    [Serializable]
    class EntityUtil
    {
        public void Dump(Entity t)
        {
            Console.WriteLine(t.a);
        }
    }


    class Program
    {
        static void Main(string[] args)
        {
            Program p = new Program();
            p.Run();
        }

        public void Run()
        {
            while (true)
            {
                Console.WriteLine("1. Run in default app domain");
                Console.WriteLine("2. Run in dedicated app domain");
                Console.WriteLine("Q. To quit");
                Console.Write("> ");
                ConsoleKeyInfo k=Console.ReadKey();
                Console.WriteLine();
                if (k.KeyChar == '1')
                {
                    RunInDefault();
                }
                else if (k.KeyChar == '2')
                {
```

```
                RunInDedicated();
            }
            else if (k.KeyChar == 'q' || k.KeyChar == 'Q')
                break;
        }

    }
        public AppDomain CreateDomain()
        {
            AppDomainSetup domaininfo = new AppDomainSetup();
            return AppDomain.CreateDomain("MyDomain",
                                          null,
                                          domaininfo);

        }

        public void RunInDefault()
        {
            EntityUtil t2 = new EntityUtil();

            Entity t = new Entity();
            t.a = 10;

            t2.Dump(t);
        }

        public void RunInDedicated()
        {
            AppDomain domain = CreateDomain();
                ObjectHandle h = domain.CreateInstance(
                            "04AppDomain",
                            "Advanced.NET.Debugging.Chapter4.EntityUtil");
                EntityUtil t2 = (EntityUtil)h.Unwrap();

                Entity t = new Entity();
                t.a = 10;

                t2.Dump(t);
            }
        }
    }
```

清单 4-4 中的源代码和二进制文件位于以下文件夹中：

- 源代码文件：C:\ADND\Chapter4\AppDomain
- 二进制文件：C:\ADNDBin\04AppDomain.exe

这个程序非常简单，它包含两个类，分别为 Entity 和 EntityUtil。Entity 类包含了一个公有域 a，而 EntityUtil 类包含一个方法 Dump，这个方法的参数是 Entity 类型，执行的操作是输

出这个公有域。当这个应用程序运行时，会显示一个菜单，它指定是在默认的应用程序域中运行还是在专门的应用程序域中运行。它还说明了如何在不同的应用程序域中使用 Entity 和 EntityUtil 等类。如果在默认的应用程序域中运行，那么这个类的所有实例都会在默认的应用程序域中创建，而如果在一个专门的应用程序域中运行，那么 EntityUtil 实例会在一个新的应用程序域中被创建，而 Entity 实例（被传递给 EntityUtil）会在默认的应用程序域中创建。由于在不同的应用程序域之间存在一个隔离层，因此如果要跨越应用程序域来传递对象，那么就需要使用列集（marshal）机制。在这里，我们将执行简单的按值列集操作（marshaling by value），并在目标应用程序域中创建对象的一个副本。现在，我们来运行这个程序并观察将发生什么情况：

```
C:\ADNDBin>04AppDomain.exe
1. Run in default app domain
2. Run in dedicated app domain
Q. To quit
> 1
10
1. Run in default app domain
2. Run in dedicated app domain
Q. To quit
> 2

Unhandled Exception: System.IO.FileNotFoundException:
Could not load file or assembly '04AppDomain' or one of its dependencies.
The system cannot find the file specified.
File name: '04AppDomain'
   at System.Reflection.Assembly._nLoad
(AssemblyName fileName, String codeBase,
Evidence assemblySecurity, Assembly locationHint,
StackCrawlMark& stackMark, Boolean throwOnFileNotFound,
Boolean forIntrospection)
   at System.Reflection.Assembly.nLoad
(AssemblyName fileName, String codeBase,
Evidence assemblySecurity, Assembly locationHint,
StackCrawlMark& stackMark, Boolean throwOnFileNotFound,
Boolean forIntrospection)
   at System.Reflection.Assembly.InternalLoad
(AssemblyName assemblyRef, Evidence assemblySecurity,
StackCrawlMark& stackMark, Boolean forIntrospection)
   at System.Reflection.Assembly.InternalLoad
(String assemblyString, Evidence assemblySecurity,
StackCrawlMark& stackMark, Boolean forIntrospection)
   at System.Activator.CreateInstance
(String assemblyName, String typeName, Boolean ignoreCase,
 BindingFlags bindingAttr, Binder binder,
Object[] args, CultureInfo culture, Object[] activationAttributes,
 Evidence securityInfo, StackCrawlMark& stackMark)
```

```
   at System.Activator.CreateInstance
(String assemblyName, String typeName)
   at System.AppDomain.CreateInstance
(String assemblyName, String typeName)
   at System.AppDomain.CreateInstance
(String assemblyName, String typeName)
   at Advanced.NET.Debugging.Chapter4.Program.RunInDedicated() in
 c:\Publishing\ADND\Code\Chapter4\AppDomain\04AppDomain.cs:line 77
   at Advanced.NET.Debugging.Chapter4.Program.Run() in
 c:\Publishing\ADND\Code\Chapter4\AppDomain\04AppDomain.cs:line 48
   at Advanced.NET.Debugging.Chapter4.Program.Main(String[] args) in
 c:\Publishing\ADND\Code\Chapter4\AppDomain\04AppDomain.cs:line 28

WRN: Assembly binding logging is turned OFF.
To enable assembly bind failure logging, set the registry value
[HKLM\Software\Microsoft\Fusion!EnableLog] (DWORD) to 1.
Note: There is some performance penalty associated with assembly bind failure
logging.
To turn this feature off, remove the registry value
[HKLM\Software\Microsoft\Fusion!EnableLog].
```

我们可以看到，只要在默认的应用程序域中运行这个程序，它就会成功执行。然而，如果在一个专门的应用程序域中运行这个程序（选项2），那么将抛出一个 FileNotFoundException 异常。为什么在默认应用程序域中可以运行，而在专门应用程序域中则运行失败呢？和前面一样，我们先来分析栈回溯。这个异常似乎来自 RunInDedicated 函数，它调用 CreateInstance 函数来创建一个 EntityType 类的实例。到目前为止，代码本身似乎一切正常。接下来在调试器下运行这个程序，并且观察异常：

```
0:000> !pe 01cb5dcc
Exception object: 01cb5dcc
Exception type: System.IO.FileNotFoundException
Message: Could not load file or assembly '04AppDomain' or one of its dependencies.
The system cannot find the file specified.
InnerException: <none>
StackTrace (generated):
    SP       IP       Function
    0017EE80 008A026C 04AppDomain!Advanced.NET.Debugging.Chapter4.
Program.RunInDedicated()+0x4c
    0017EE9C 008A01BC 04AppDomain!Advanced.NET.Debugging.Chapter4.Program.Run()+0xcc
    0017EEC8 008A00A7
04AppDomain!Advanced.NET.Debugging.Chapter4.Program.Main(System.String[])+0x37

StackTraceString: <none>
HResult: 80070002
```

在调试器中输出的异常并没有包含一些有价值的线索（只是报告有一个程序集加载失败）。从前面的章节中，我们知道可以使用 fuslogvw. exe 工具来找出 CLR 在哪些位置查找程序集。在图 4-8 中给出了在这个程序上运行 fuslogvw. exe 的结果。

```
*** Assembly Binder Log Entry  (1/3/2009 @ 12:54:43 PM) ***

The operation failed.
Bind result: hr = 0x80070002. The system cannot find the file specified.

Assembly manager loaded from:  C:\Windows\Microsoft.NET\Framework\v2.0.50727\mscorwks.dll
Running under executable  C:\ADNDBin\04AppDomain.exe
--- A detailed error log follows.

=== Pre-bind state information ===
LOG: User =          marioh
LOG: DisplayName = 04AppDomain
 (Partial)
LOG: Appbase = file:///C:/Windows/System32
LOG: Initial PrivatePath = NULL
LOG: Dynamic Base = NULL
LOG: Cache Base = NULL
LOG: AppName = 04AppDomain.exe
Calling assembly : (Unknown).
===
LOG: This bind starts in default load context.
LOG: Download of application configuration file was attempted from file:///C:/Windows/System32/04AppDomain.exe.config.
LOG: Configuration file C:\Windows\System32\04AppDomain.exe.config does not exist.
LOG: No application configuration file found.
LOG: Using machine configuration file from C:\Windows\Microsoft.NET\Framework\v2.0.50727\config\machine.config.
LOG: Policy not being applied to reference at this time (private, custom, partial, or location-based assembly bind).
LOG: Attempting download of new URL file:///C:/Windows/System32/04AppDomain.DLL.
LOG: Attempting download of new URL file:///C:/Windows/System32/04AppDomain/04AppDomain.DLL.
LOG: Attempting download of new URL file:///C:/Windows/System32/04AppDomain.EXE.
LOG: Attempting download of new URL file:///C:/Windows/System32/04AppDomain/04AppDomain.EXE.
LOG: All probing URLs attempted and failed.
```

图 4-8　AppBase 被设置为 C:\Windows\System32,04AppDomain. exe 的 CLR 加载器日志

从图 4-8 中 CLR 加载器日志中，可以看到 CLR 加载器尝试从路径 C:\Windows\System32中加载 04AppDomain. exe 程序集。为什么会选择这个路径？答案在于，当一个对象通过列集来跨越应用程序域的边界时，这个对象会被串行化（serialization），这意味着目标应用程序域必须访问包含这个类型定义的程序集。如果没有这个程序集，那么反串行化操作（deserialization）就会失败。还需要注意的是，应用程序域可以控制程序集探测路径的某些特定部分。具体来说，图 4-8 中的 AppBase 被设置为 C:\Windows\System32。如果进一步观察源代码中的 RunInDedicated 方法（没有包含在清单 4-4 中），那么会看到以下代码行：

```
AppDomainSetup domaininfo = new AppDomainSetup();
domaininfo.ApplicationBase = "C:\\Windows\\System32";
return AppDomain.CreateDomain("MyDomain", null, domaininfo);
```

事实上，在创建专门应用程序域的过程中，我们将应用程序库的路径设置为 C:\Windows\System32，这就意味着 CLR 无法加载任何依赖的程序集，除非它们也位于这个路径中。虽然这对应用程序域来说可能不是最优的方法，但这里只是为了说明应用程序中的一个常见问题，即 CLR 加载器选择不正确程序集的问题（或者在加载程序集时失败）。

FileNotFoundException 与 FusionLog 等属性

当打开程序集绑定日志功能时，你可能已经注意到在收集的日志项数据上设置了一个属性 FusionLog，它表示通过异常机制来传播故障。以下是当日志功能被打开时的 FileNotFoundException 及其 FusionLog 属性：

```
0:000> !do 01d863e0
Name: System.IO.FileNotFoundException
MethodTable: 791222c4
EEClass: 79122244
Size: 84(0x54) bytes
 (C:\Windows\assembly\GAC_32\mscorlib\2.0.0.0__b77a5c561934e089\mscorlib.dll)
 Fields:
       MT    Field   Offset                 Type VT     Attr    Value Name
 790fd8c4  40000b5        4        System.String  0 instance 01d6e1f8
 _className
 7910ebc8  40000b6        8 ...ection.MethodBase  0 instance 00000000
 _exceptionMethod
 …
 …
 …
 790fd8c4  4001bb5       4c        System.String  0 instance 01d6d18c
 _fusionLog
 0:000> !do 01d6d18c
 Name: System.String
 MethodTable: 790fd8c4
 EEClass: 790fd824
 Size: 2346(0x92a) bytes
  (C:\Windows\assembly\GAC_32\mscorlib\2.0.0.0__b77a5c561934e089\
 mscorlib.dll)
 String: Assembly manager loaded from:
 C:\Windows\Microsoft.NET\Framework\v2.0.50727\mscorwks.dll
 Running under executable  C:\ADNDBin\04AppDomain.exe
 -- A detailed error log follows.

 === Pre-bind state information ===
 LOG: User = REDMOND\marioh
 LOG: DisplayName = 04AppDomain
  (Partial)
 LOG: Appbase = file:///C:/Windows/System32
 LOG: Initial PrivatePath = NULL
 Calling assembly : (Unknown).
 ===
 LOG: This bind starts in default load context.
LOG: Configuration file C:\Windows\System32\04AppDomain.exe.config does not
exist.
LOG: No application configuration file found.
LOG: Using machine configuration file from
C:\Windows\Microsoft.NET\Framework\v2.0.50727\config\machine.config.
LOG: Policy not being applied to reference at this time
(private, custom, partial, or location-based assembly bind).
```

```
LOG: Attempting download of new URL
file:///C:/Windows/System32/04AppDomain.DLL.
LOG: Attempting download of new URL
file:///C:/Windows/System32/04AppDomain/04AppDomain.DLL.
LOG: Attempting download of new URL
file:///C:/Windows/System32/04AppDomain.EXE.
LOG: Attempting download of new URL
file:///C:/Windows/System32/04AppDomain/04AppDomain.EXE.

...
...
...
```

在处理加载器问题时，另一个有用的工具就是托管调试助手（Managed Debugging Assistant，MDA）。在第 1 章中给出了 MDA 的简要介绍以及如何启用它们。对于 CLR 加载器来说，有一个叫做 bindingFailure 的 MDA 可用于分析 CLR 加载器问题。要启用这个 MDA，可以将以下 XML 保存到 04AppDomain. exe. mda. config 文件中：

```
<mdaConfig>
  <assistants>
    <bindingFailure />
  </assistants>
</mdaConfig>
```

这个 XML 只是为程序 04AppDomain. exe 启用 bindingFailure。如果在调试器下重新运行 04AppDomain. exe，并重现这个问题（通过选择选项 2），那么可以看到以下输出信息：

```
> 2
ModLoad: 74e40000 74e7b000   C:\Windows\system32\rsaenh.dll
ModLoad: 60340000 60348000   C:\Windows\Microsoft.NET\Framework\v2.0.50727
\culture.dll
(1c60.18c0): C++ EH exception - code e06d7363 (first chance)
(1c60.18c0): C++ EH exception - code e06d7363 (first chance)
ModLoad: 60340000 60348000   C:\Windows\Microsoft.NET\Framework\v2.0.50727
\culture.dll
<mda:msg xmlns:mda="http://schemas.microsoft.com/CLR/2004/10/mda">
  <!--
      The assembly with display name '04AppDomain' failed to load in the
'LoadFrom'
      binding context of the AppDomain with ID 2. The cause of the failure was:
      System.IO.FileNotFoundException: Could not load file or assembly
'04AppDomain'
      or one of its dependencies. The system cannot find the file specified.
      File name: '04AppDomain'
      WRN: Assembly binding logging is turned OFF.
```

```
        To enable assembly bind failure logging, set the registry value
        [HKLM\Software\Microsoft\Fusion!EnableLog] (DWORD) to 1.
        Note: There is some performance penalty associated with assembly bind
failure
        logging.
        To turn this feature off, remove the registry value
        [HKLM\Software\Microsoft\Fusion!EnableLog].

   -->
  <mda:bindingFailureMsg break="true">
    <assemblyInfo appDomainId="2" displayName="04AppDomain" codeBase="" hResult="-
2147024894" bindingContextId="1"/>
  </mda:bindingFailureMsg>
</mda:msg>
(1c60.18c0): Break instruction exception - code 80000003 (first chance)
eax=00000000 ebx=00000000 ecx=7728861f edx=000d0c30 esi=0029b0a4 edi=00000200
eip=7a0c8c7c esp=0029a328 ebp=0029a7ac iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
mscorwks!MdaXmlMessage::SendDebugEvent+0x1e6:
7a0c8c7c cc              int     3
```

当调试器由于抛出异常而中断时，我们可以看到一些 XML 形式的信息。其中大部分信息都是我们之前通过观察异常本身得到的，除此之外还输出了其他一些有用的信息，例如应用程序域的 ID。我们可以使用 DumpDomain 命令来得到关于应用程序域的更详细的信息：

```
0:000> !DumpDomain
…
…
…
-------------------
Domain 2: 000bfcb8
LowFrequencyHeap: 000bfcdc
HighFrequencyHeap: 000bfd34
StubHeap: 000bfd8c
Stage: OPEN
SecurityDescriptor: 000c1f00
Name: MyDomain
Assembly: 000b45d8
[C:\Windows\assembly\GAC_32\mscorlib\2.0.0.0__b77a5c561934e089\mscorlib.dll]
ClassLoader: 000b4670
SecurityDescriptor: 000c3150
  Module Name
790c2000 C:\Windows\assembly\GAC_32\mscorlib\2.0.0.0__b77a5c561934e089\mscorlib.dll
0077239c C:\Windows\assembly\GAC_32\mscorlib\2.0.0.0__b77a5c561934e089\sortkey.nlp
00772010 C:\Windows\assembly\GAC_32\mscorlib\2.0.0.0__b77a5c561934e089\sorttbls.nlp
```

有没有其他有用的 MDA

答案是肯定的。当使用指定加载上下文时，loadFromContext 这个 MDA 可以发出警告。通常，人们会在无意中使用指定加载上下文，这将导致一些严重的问题。当某个程序集被加载到指定加载上下文时，这个 MDA 会发送一些警告信息。

4.4 互用性与 DllNotFoundException

CLR 提供了一个互用性层，使得. NET 代码能够与底层的非托管代码模块相互调用。当通过托管代码来调用遗留的非托管代码时，这个功能就显得非常重要。根据不同类型的非托管代码，可以有两种不同的互用性机制。第一种称为平台调用（Platform Invocation，P/Invokle），这种机制使开发人员能够调用非托管代码模块（或者 DLL）中导出的函数。要实现这种机制，必须定义与托管代码等价的方法原型，并使用 DllImport 属性来告知这是一个 P/Invoke 方法。第二种互用性称为 COM 互用性，它使得托管代码能够与非托管的 COM 对象一起工作。在本章的这部分，我们将介绍互用性在加载器环境中的含义。如果 CLR 在加载程序集时使用了不同的加载上下文，那么将带来某种程度的复杂性，这种情形与非托管代码平台互用性之间存在怎样的关联？在接下来的讨论中，我们将使用一个简单的程序，如清单 4-5 所示。

清单 4-5 P/Invoke 的简单示例

```
using System;
using System.Text;
using System.Runtime.InteropServices;

namespace Advanced.NET.Debugging.Chapter4
{
    class Interop
    {
        static void Main(string[] args)
        {
          Console.WriteLine("Press any key to P/Invoke");
          Console.ReadKey();
            Console.WriteLine();
            PrintMsg("Printed via P/Invoke");
        }

        [DllImport("04Native.dll", CharSet = CharSet.Auto)]
        internal static extern void PrintMsg(string message);
    }
}
```

清单 4-5 中的源代码和二进制文件位于以下文件夹中：

- 源代码文件：C:\ADND\Chapter4\Interop
- 二进制文件：C:\ADNDBin\04Interop.exe and 04Native.dll

源代码本身很简单。Main 方法只是调用一个 PrintMsg 的 P/Invoke 方法，这个方法将调用等价的非托管代码来输出传递给这个函数的字符串。运行这个程序，可以看到它将成功执行：

```
C:\ADNDBin>04Interop.exe
Press any key to P/Invoke

Printed via P/Invoke
```

为了说明 CLR 在无法找到某个非托管 DLL 时表现出的行为，我们将文件夹 C:\ADNDBIN 中的 04Native.dll 重命名为 04Native.old 并重新运行这个程序。

```
C:\ADNDBin>04Interop.exe
Press any key to P/Invoke

Unhandled Exception: System.DllNotFoundException:
Unable to load DLL '04Native.dll': The specified module could not
be found. (Exception from HRESULT: 0x8007007E)
   at Advanced.NET.Debugging.Chapter4.Interop.PrintMsg(String message)
   at Advanced.NET.Debugging.Chapter4.Interop.Main(String[] args) in
c:\Publishing\ADND\Code\Chapter4\Interop\04Interop.cs:line 14
```

这一次，我们可以看到抛出了一个 DllNotFoundException 异常。在本章的前面部分，我们介绍了 CLR 在绑定到正确程序集时使用的程序集加载逻辑。那么，这种逻辑是否同样适用于非托管 DLL？答案是不适用。在 P/Invoke 调用过程中使用的任何非托管 DLL，都将通过底层的 kernel32！LoadLibrary 来加载，这个 API 使用了与 Windows 加载动态库时相同的探测逻辑。关于加载逻辑的详细讨论，请参见 MSDN 文档中 LoadLibrary 部分。事实上，在前面分析加载问题时使用的 fuslogvw.exe 工作在分析 P/Invoke 加载问题时不会起任何作用。

COM 互用性的情况怎样

与 P/Invoke 依赖于 Windows 加载逻辑非常类似，COM 互用性层依赖于 COM 加载器。在 COM 对象能够由托管代码访问之前，必须在注册表中正确地注册它们，并且遵循 COM 子系统使用的相同加载逻辑。

4.5 轻量级代码生成的调试

CLR 提供了一种既方便又高效的机制来动态生成代码（也被称为 CodeGen）。在 CLR2.0 以前，所有动态生成代码的 API 都位于命名空间 System.Reflection.Emit 中。在代码生成过程中，开发人员必须维护在 CLR 上运行的任何代码段的结构完整性。例如，如果要定义一个简单的方法 Add，这个方法带有两个整数参数并且返回加和值，那么我们必须定义方法本身

（包括 IL）、这个方法使用的类型、这个类型定义所在的模块以及程序集所在的模块。虽然这种形式的代码生成能够提供一些特性，例如注入调试信息，但使用起来非常复杂，尤其是当你只希望生成一些非常简单的或小规模的方法时。为了减轻这方面的负担，CLR 2.0 引入了轻量级代码生成机制（Light Weight Code Generation，LCG）。当使用 LCG 时，将不再需要定义代码的各个方面（模块、类型、程序集等），而是只需定义代码本身，并且通过委托机制来调用它。由于在许多不同的系统中都大量使用了代码生成，因此理解如何对动态生成的代码进行调试就显得非常重要。毕竟，代码并不是在编译时生成的，因此没有相应的调试信息。你可能会发现，使用非托管调试器来调试动态生成的代码并不困难。为了更好地理解这个过程，我们来看一个简单的示例。清单 4-6 给出了一个非常简单的程序，这个程序将动态生成代码，把两个整数相加并返回结果。

清单 4-6　LCG 生成代码的简单示例

```
using System;
using System.Text;
using System.Reflection.Emit;

namespace Advanced.NET.Debugging.Chapter4
{
    class CodeGen
    {
        private delegate int Add(int a, int b);

        public static void Main()
        {
            Type[] args={typeof(int), typeof(int)};
            DynamicMethod dyn = new
               DynamicMethod("Add",
                             typeof(int),
                             new Type[] { typeof(int), typeof(int) },
                             typeof(CodeGen),
                             true);
            ILGenerator gen = dyn.GetILGenerator();
            gen.Emit(OpCodes.Ldarg_1);
            gen.Emit(OpCodes.Ldarg_2);
            gen.Emit(OpCodes.Add);
            gen.Emit(OpCodes.Ret);

            Add a= (Add) dyn.CreateDelegate(typeof(Add));
            int ret = a(1, 2);
            Console.WriteLine("1+2={0}", ret);
        }
    }
}
```

清单 4-6 中的源代码和二进制文件位于以下文件夹中：

- 源代码文件：C:\ADND\Chapter4\CodeGen
- 二进制文件：C:\ADNDBin\04CodeGen.exe

你可以看到，代码非常简单。我们定义了一个委托（add），这个委托带有两个整数并返回加和值。然后创建了 DynamicMethod 类的一个实例，并且使用 ILGenerator 来为 Add 方法生成 IL 指令。最后声明了生成代码的一个委托实例并调用它。如果运行这个程序，可以看到以下输出：

```
C:\ADNDBin>04CodeGen.exe
1+2=3
```

现在，问题变成了：如果想要调试这个程序去观察动态生成代码的运行过程，如何实现？例如，如果想在 Add 方法上设置一个断点，那么能否使用在 SOS 和 SOSEX 等调试器扩展中包含的断点命令？答案是可以，但需要一些额外的辅助工作。要想使用 bpmd 命令，首先需要找到设置断点的方法的描述符。通常这并不是一个问题，因为可以使用多种不同的命令来找到它。然而，对于 LCG 代码，我们则需要手动完成一些工作。具体来说，我们需要通过 JIT 组件中某个函数来获得方法描述符。这个函数就是 CILJit::compileMethod。如果在这个函数上设置一个非托管断点（通过 bp 命令），那么可以从这个函数的第三个参数中获得方法描述符。在调试器下运行 04CodeGen，并且当提示按下任意键时，手动中断进入到调试器，如下所示：

```
0:000> .symfix
...
...
...
(1b54.1df8): Break instruction exception - code 80000003 (first chance)
eax=7ffda000 ebx=00000000 ecx=00000000 edx=772ad094 esi=00000000 edi=00000000
eip=77267dfe esp=0441ff58 ebp=0441ff84 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
ntdll!DbgBreakPoint:
77267dfe cc              int     3
0:004> .loadby sos mscorwks
0:004> bp mscorjit!CILJit::compileMethod
0:004> g
Breakpoint 0 hit
eax=790b22a0 ebx=0031e80c ecx=790a60a0 edx=77279a94 esi=00555160 edi=00000000
eip=7906114f esp=0031e6c8 ebp=0031e730 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
mscorjit!CILJit::compileMethod:
7906114f 55              push    ebp
0:000> kb
```

```
ChildEBP RetAddr  Args to Child
0031e6c4 79f0f9cf 790b22a0 0031e80c 0031e898 mscorjit!CILJit::compileMethod
0031e730 79f0f945 0056e7a8 0031e80c 0031e898 mscorwks!invokeCompileMethodHelper+0x72
0031e774 79f0f8da 0056e7a8 0031e80c 0031e898 mscorwks!invokeCompileMethod+0x31
0031e7cc 79f0ea33 0056e7a8 0031e80c 00000000
mscorwks!CallCompileMethodWithSEHWrapper+0x84
0031eb84 79f0e795 001e3250 00000000 00107214 mscorwks!UnsafeJitFunction+0x230
0031ec28 79e87f52 00000000 00000000 60f1701d
mscorwks!MethodDesc::MakeJitWorker+0x1c1
0031ec80 79e8809e 00000000 60f1704d 0031ef8c mscorwks!MethodDesc::DoPrestub+0x486
0031ecd0 00190876 0031ed00 725139c9 00000002 mscorwks!PreStubWorker+0xeb
WARNING: Frame IP not in any known module. Following frames may be wrong.
0031ed70 79e7c74b 0031edbc 00000000 0031ee00 0x190876
0031ed80 79e7c6cc 0031ee50 00000000 0031ee20 mscorwks!CallDescrWorker+0x33
0031ee00 79e7c8e1 0031ee50 00000000 0031ee20
mscorwks!CallDescrWorkerWithHandler+0xa3
0031ef3c 79e7c783 001ec028 0031f004 0031efd0 mscorwks!MethodDesc::CallDescr+0x19c
0031ef58 79e7c90d 001ec028 0031f004 0031efd0
 mscorwks!MethodDesc::CallTargetWorker+0x1f
0031ef6c 79eefb9e 0031efd0 60f16c4d 00000000 mscorwks!MethodDescCallSite::Call+0x18
0031f0d0 79eef830 001e3028 00000001 0031f10c mscorwks!ClassLoader::RunMain+0x263
0031f338 79ef01da 00000000 60f16495 00000001
 mscorwks!Assembly::ExecuteMainMethod+0xa6
0031f808 79fb9793 010f0000 00000000 60f164c5
 mscorwks!SystemDomain::ExecuteMainMethod+0x43f
0031f858 79fb96df 010f0000 60f1643d 00000000 mscorwks!ExecuteEXE+0x59
0031f8a0 7900b1b3 00000000 79e70000 0031f8bc mscorwks!_CorExeMain+0x15c
0031f8b0 773e4911 7ffdf000 0031f8fc 7725e4b6 mscoree!_CorExeMain+0x2c
0:000> dd 0031e898
0031e898  001e3250 001e3a31 00571528 00000004
0031e8a8  00000002 00000010 00000000 00000000
0031e8b8  00000000 00020008 00000000 00000000
0031e8c8  00000000 00000000 0056f7eb 0056f7e8
0031e8d8  001e3a31 00000000 00000000 00000000
0031e8e8  00000000 00000101 00000000 00000000
0031e8f8  00000000 00000000 005773aa 005773a8
0031e908  001e3a31 00000000 00000000 0000003c
0:000> !DumpMD 001e3250
Method Name: DynamicClass.Add(Int32, Int32)
Class: 001e319c
MethodTable: 001e3200
mdToken: 06000000
Module: 001e2c3c
IsJitted: no
m_CodeOrIL: ffffffff
```

要找出方法描述符，我们需要将 CILJit :: compileMethod 的第三个参数转储出来，其中第一个 DWORD 包含的就是方法描述符，可以传递给 DumpMD 命令。在获得了方法描述符后，我们对其使用一些命令来进行分析。例如，如果在方法描述符上运行 DumpIL 命令，可以看到 Add 方法的 IL：

```
0:000> !DumpIL 001e3250
This is dynamic IL. Exception info is not reported at this time.
If a token is unresolved, run "!do <addr>" on the addr given
in parenthesis. You can also look at the token table yourself, by
running "!DumpArray 01e58738".

IL_0000: ldarg.1
IL_0001: ldarg.2
IL_0002: add
IL_0003: ret
```

有了方法描述符后，还可以通过 bpmd 命令在动态的 Add 方法上设置一个断点：

```
0:000> !bpmd -md 001e3250
MethodDesc = 001e3250
This DynamicMethodDesc is not yet JITTED. Placing memory breakpoint at 001ec45c
0:000> g
Breakpoint 2 hit
eax=001e3250 ebx=01e583cc ecx=00000001 edx=00000002 esi=01e587c8 edi=01e58038
eip=00aa00a8 esp=0031ed0c ebp=0031ed70 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
00aa00a8 56              push    esi
0:000> !ClrStack -a
OS Thread Id: 0x1e60 (0)
ESP       EIP
0031ed0c 00aa00a8 DynamicClass.Add(Int32, Int32)
    PARAMETERS:
        <no data>
        <no data>
0031ed10 00a702b9 Advanced.NET.Debugging.Chapter4.CodeGen.Main()
    LOCALS:
        0x0031ed40 = 0x01e5800c
        0x0031ed3c = 0x01e58064
        <CLR reg> = 0x01e583cc
        0x0031ed38 = 0x01e587c8
        0x0031ed60 = 0x00000000
        <CLR reg> = 0x01e58038

0031ef8c 79e7c74b [GCFrame: 0031ef8c]
```

到这里就结束关于如何调试 CLR 中轻量级代码生成机制的讨论。有许多组件都可以使用动态生成的代码，因此知道如何调试这些代码是非常重要的。

4.6 小结

在本章中，我们介绍了 CLR 加载器的内部机制，如何识别一些常见的误区以及在分析这些误区时可以采用的一些工具。我们简单介绍了 CLR 加载器、各种不同的加载上下文以及 CLR 在每种上下文中使用的探测逻辑。此外，还给出了一组常见的错误示例，并通过调试器 fuslogvw. exe 和 MDA 来分析这些错误。在使用 CLR 加载器时要非常小心，尤其要注意在加载程序集时所使用的加载上下文，这样才能尽量避免各种加载器问题。

第 5 章　托管堆与垃圾收集

在当前的应用程序中，如果以手工方式来管理内存，会很容易产生一些错误。事实上，有多项研究表明，一些最常见的错误都与手工方式管理内存有着密切联系。例如，其中一些问题包括：

- 悬空指针
- 重复释放
- 内存泄漏

如果采用自动内存管理机制（automatic memory management），那么将有助于消除这种复杂而又容易出错的手工管理内存机制。随着 Java 和. NET 平台的出现，自动化内存管理正获得越来越多的关注，但与之相应的概念和实现却早已出现。1959 年，John McCarthy 率先在 LISP 语言中提出了自动内存管理模式，此外还有其他一些语言同样实现了类似的机制。当前，人们普遍将垃圾收集器（Garbage Collector，GC）视为自动内存管理的一种实现。. NET 平台同样以自动内存管理机制为基础，并在内部实现了一个高性能和高可靠性的 GC。对于开发人员来说，尽管 GC 极大地简化了开发工作，使他们能够把更多的精力放在业务逻辑上，但如果对 GC 的运作原理能够有一些深入的理解，那么就能避开一些在垃圾收集环境中出现的问题。在本章中，我们将介绍 CLR 堆管理器和 GC 的内部工作机制，以及一些可能对应用程序造成破坏的常见错误。我们将通过调试器以及其他一组工具来说明如何分析这些问题的根本原因。

5.1　Windows 内存架构简介

在深入研究 CLR 堆管理器和 GC 的细节内容之前，我们先来回顾一下 Windows 的整体内存架构。图 5-1 给出了在一个进程中涉及的各种常见内存组件。

从图 5-1 中可以看出，在用户态（user mode）中运行的进程通常会使用一个或多个堆管理器。最常见的堆管理器就是 Windows 堆管理器（windows heap manager），很多应用程序中都大量地使用了这个堆管理器。另一个常见的堆管理器就是 CLR 堆管理器（CLR Heap Manager），它是在. NET 应用程序中使用的。Windows 堆管理器负责满足大多数的内存分配/回收请求，它从 Windows 虚拟内存管理器（windows virtual memory manager）中分配大块内存空间（称为内存段（Segment）），并且通过维持特定的记录数据（例如旁视列表（Look Aside List）和空闲列表（Free List）），以一种高效的方式将大块内存空间分割为许多更小的内存块来满足进程的分配请求。CLR 堆管理器的功能是类似的，它为托管进程中的所有内存分配请

求提供一站式服务。与Windows堆管理器相似的是，它同样从 Windows 虚拟内存管理器中分配大块内存（也被称为内存段），然后用这些内存段来满足所有的内存分配/回收请求。这两种堆管理器之间的关键差异在于，二者在维持堆完整性时使用的记录数据的结构是不同的。图 5-2 给出了 CLR 堆管理器的示意图。

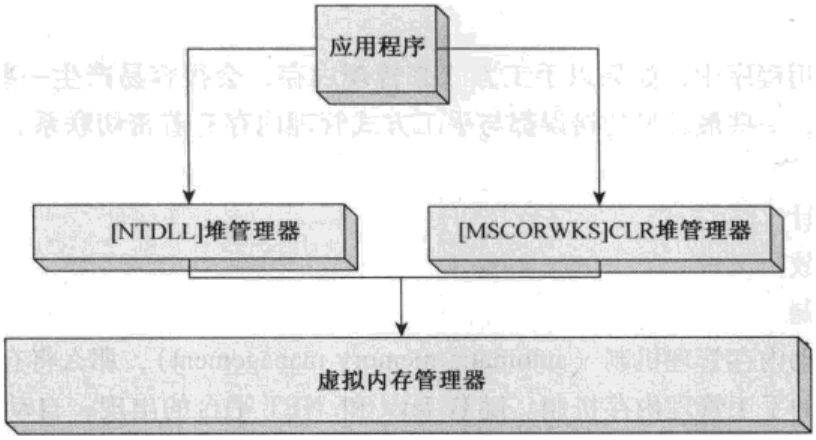


图 5-1　Windows CLR 内存架构概览

从图 5-2 中可以看到，CLR 堆管理器很小心地使用大块内存（内存段）来满足内存请求。从图中还可以观察到 CLR 堆管理器的运作模式，共有两种：工作站模式和服务器模式。就 CLR 堆管理器而言，服务器模式的主要特点是，它不是只有一个堆，而是每个处理器都对应一个堆，并且堆中内存段的大小通常大于工作站堆中内存段的大小（但是，这种差异只是一个实现细节，不必过多在意它）。从 GC 的角度来看，在工作站和服务器这两种模式中还存在其他一些重要差异，这些差异主要存在于 GC 线程模型，在服务器模式中，有一个专门的线程在管理所有的 GC 操作，而工作站模式中，则是在执行内存分配的线程上执行 GC 操作。

工作站模式和服务器模式的实现是否位于不同的二进制文件中

在 2.0 版本之前，工作站 GC 是在 mscorwks. dll 中实现的，而服务器 GC 则是在 mscorsvr. dll 中实现的。到了 2.0 版本时，这两种实现被合并到同一个二进制文件中（mscorwks. dll）。请注意，这种合并只是在二进制级别上的合并。

当加载 CLR 时，每个托管进程起初都有两个堆，并且每个堆都有各自的内存段。第一个堆被称为小对象堆（small object heap），这个堆中有一个初始内存段，在工作站模式中的大小为 16MB（在服务器模式中则更大）。小对象堆用来容纳大小不超过 85 000 字节的对象。第二个堆被称为大对象堆（Large Object Heap，LOH），它同样有一个初始内存段，大小为 16MB。LOH 用来容纳大小等于或超过 85 000 字节的对象。在随后讨论垃圾收集器的内部机制时，我们就能看到为什么要根据对象大小来区分不同的堆。需要注意的是，当创建一个内

存段时，不会立即提交段中的所有内存，CLR 堆管理器会保留一部分内存空间，并且根据需要来进行提交。当小对象堆中的内存都被耗尽时，CLR 堆管理器将触发 GC 操作；如果内存空间仍然不足，将对堆进行扩展。然而，如果大对象堆中的内存段被耗尽时，堆管理器将创建一个新的内存段来提供内存。相应地，当垃圾收集器释放内存时，内存段中的空间可以回收（decommit），当一个内存段中的所有空间都被回收时，这个内存段就会被彻底释放。

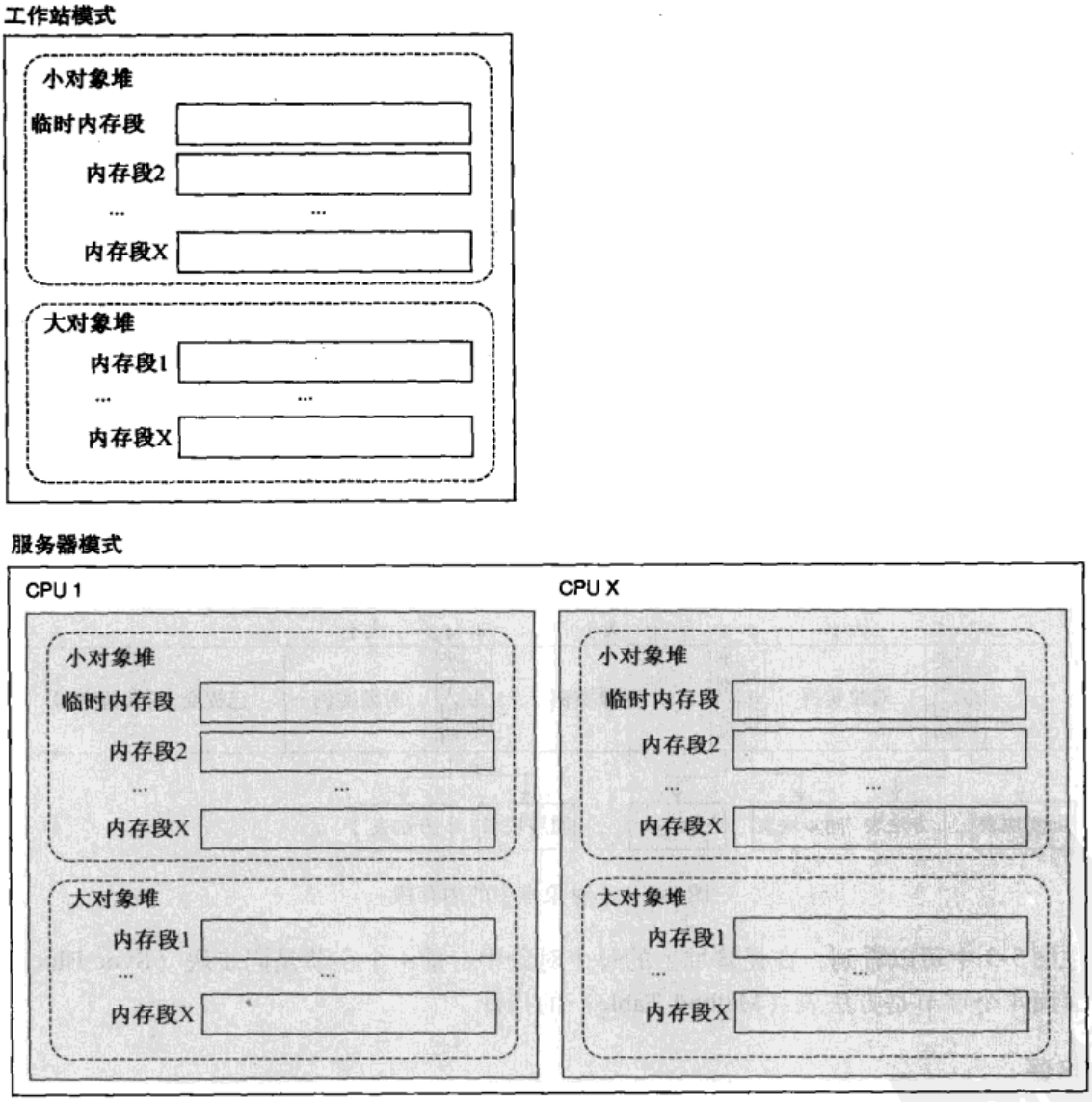


图 5-2　CLR 堆管理器概览

内存地址中的内容是什么

给定一个内存地址，那么是否有某种方式能够找出这块内存的状态？也就是，这块内存是被保留的还是已被提交的？是可写的还是只读的？回答这些问题的最佳方式是使用 address 命令。如果不带任何参数，那么 address 命令会给出进程中内存行为的详细信息和汇总信息。如果指定了一个内存地址作为参数，那么 address 命令也会尝试找出关于这个地址的一些信息。例如，假设有一个对象位于托管堆地址 0x01d96c58 上，如果在这个地址上运行 address 命令，将给出以下输出：

```
0:000> !address 0x01d96c58
 ProcessParameters 004d1668 in range 004d0000 0050b000
 Environment 004d0808 in range 004d0000 0050b000
    01d90000 : 01d90000 - 00012000
                        Type     00020000 MEM_PRIVATE
                        Protect  00000004 PAGE_READWRITE
                        State    00001000 MEM_COMMIT
                        Usage    RegionUsageIsVAD
```

这个地址是一个已分配的并且可访问的内存地址，这个内存地址是可读/写的，并且已经被提交。

在第 2 章中曾简要提到过，驻留在托管堆上的每个对象都附带有一些元数据。具体来说，每个对象的前面都有 8 个字节。图 5-3 给出了小对象堆中的一个内存段。

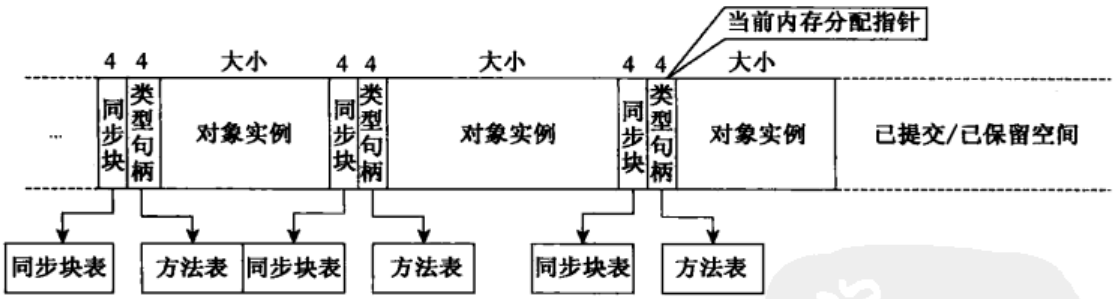


图 5-3　小对象堆中的内存段

在图 5-3 中可以看到，在托管堆上的每个对象中，前 4 个字节是同步块（Sync Block）索引，后面 4 个字节是方法表（Method Table）的指针。

分配内存

到目前为止，我们已经理解了 CLR 堆管理器中的内存层次结构，现在就来看看如何满足内存分配请求。我们已经知道在 CLR 堆管理器中包含了一个或者多个内存段，分配其中一个内存段并将它返回给调用者。然而这个内存分配过程是如何执行的？图 5-4 说明了当一个内存分配请求到达时 CLR 的完整处理流程。

在理想情况下，即当不需要执行 GC 就可以成功分配内存时，满足内存分配请求将是一个非常高效的过程。这种情况下会执行两个主要的任务，即递进内存分配指针和清空相应的内存区域。递进内存分配指针意味着新分配的内存将紧跟内存段中最后一个已分配的对象。当另一个分配请求被满足时，内存分配指针会再次被递进，依次类推。请注意，这种分配模式与 Windows 堆管理器中的模式有着极大的不同，因为 Windows 堆管理器并不会像这样来管理各个对象之间的位置。在 Windows 堆管理器中，当一个分配请求到来时，可以使用内存段中的任何一块空闲内存来满足这个请求。另一种需要注意的情况就是，当达到内存阈值时，将触发 GC 来执行垃圾收集动作。在这种情况下，会首先执行 GC，然后再尝试满足内存分配请求。在图 5-4 中需要注意的最后一点是，判断被分配的对象是否是可终结的。严格来说，这并非是堆管理器必须实现的一个功能，但它属于分配过程的一部分。如果对象是可终结的，那么将在 GC 中记录这个对象，以便正确地管理对象的生命周期。在本章的后面，我们将进一步讨论可终结对象。

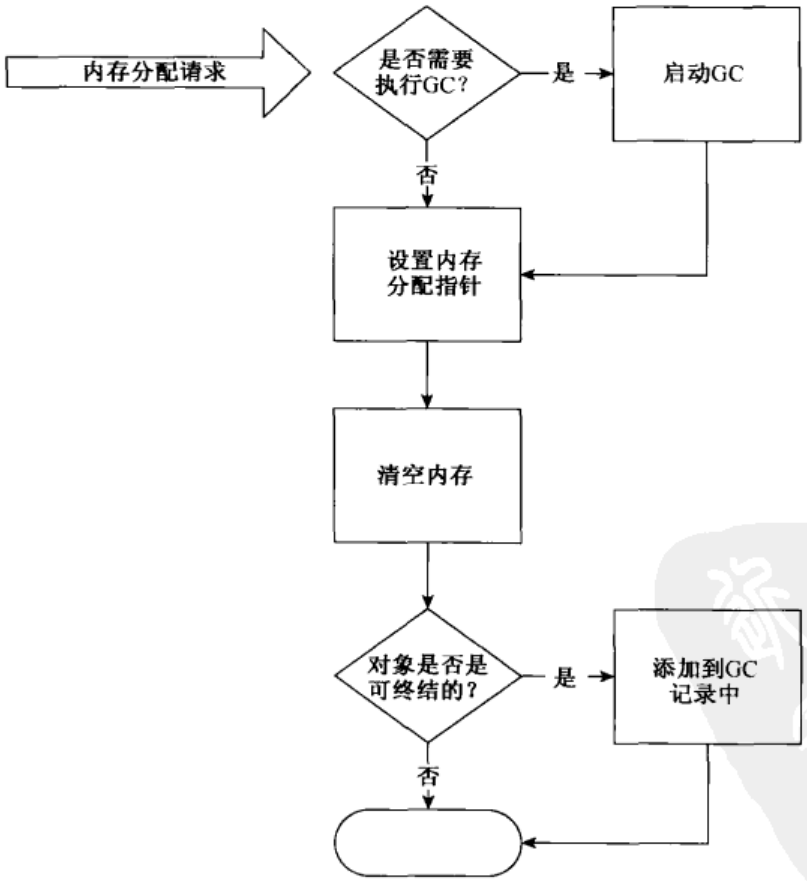


图 5-4　CLR 堆管理器的内存分配流程

在继续介绍垃圾收集器的内部机制之前，我们先来看一个执行内存分配操作的简单应用程序。在清单 5-1 中给出了程序的源代码。

清单 5-1　简单的内存分配

```
using System;
using System.Text;
using System.Runtime.Remoting;

namespace Advanced.NET.Debugging.Chapter5
{
    class Name
    {
        private string first;
        private string last;

        public string First { get { return first; } }
        public string Last { get { return last; } }

        public Name(string f, string l)
        {
            first = f; last = l;
        }
    }

    class SimpleAlloc
    {
        static void Main(string[] args)
        {
            Name name = null;

            Console.WriteLine("Press any key to allocate memory");
            Console.ReadKey();

            name = new Name("Mario", "Hewardt");

            Console.WriteLine("Press any key to exit");
            Console.ReadKey();
        }
    }
}
```

清单 5-1 中的源代码和二进制文件位于以下文件夹中：

- 源代码文件：C:\ADND\Chapter5\SimpleAlloc
- 二进制文件：C:\ADNDBin\05SimpleAlloc.exe

清单 5-1 中的代码非常简单，但需要注意的是如何通过调试器在托管堆中找到这块分配的内存。在 SOS 调试器扩展中包含了一些非常方便的命令，可以使我们得到关于托管堆的一些内部信息。在这个示例中，我们将使用 DumpHeap 命令。默认情况下，DumpHeap 命令会

列出在托管堆中的所有对象以及它们的地址、方法表和大小。首先在调试器下运行 05SimpleAlloc.exe，然后在出现提示信息“Press any key to allocate memory”时中断程序的执行。在中断到调试器后，运行 DumpHeap 命令。下面给出了这个命令的部分输出：

```
0:004> !DumpHeap
 Address       MT     Size
790d8620  790fd0f0       12
790d862c  790fd8c4       28
790d8648  790fd8c4       32
790d8668  790fd8c4       32
790d8688  790fd8c4       28
790d86a4  790fd8c4       24
790d86bc  790fd8c4       24
...
...
...

total 2379 objects
Statistics:
      MT    Count    TotalSize Class Name
79119954        1           12 System.Security.Permissions.ReflectionPermission
79119834        1           12 System.Security.Permissions.FileDialogPermission
791032a8        1          128 System.Globalization.NumberFormatInfo
79100e38        3          132 System.Security.FrameSecurityDescriptor
791028f4        2          136 System.Globalization.CultureInfo
791050b8        4          144 System.Security.Util.TokenBasedSet
790fe284        2          144 System.Threading.ThreadAbortException
79102290       13          156 System.Int32
790f97c4        3          156 System.Security.Policy.PolicyLevel
790ff734        9          180 System.RuntimeType
790ffb6c        3          192 System.IO.UnmanagedMemoryStream
7912d7c0       11          200 System.Int32[]
790fd0f0       17          204 System.Object
79119364        8          256 System.Collections.ArrayList+SyncArrayList
79101fe4        6          336 System.Collections.Hashtable
79100a18       10          360 System.Security.PermissionSet
79112d68       18          504
System.Collections.ArrayList+ArrayListEnumeratorSimple
79104368       21          504 System.Collections.ArrayList
7912d9bc        6          864 System.Collections.Hashtable+bucket[]
7912dae8        8         1700 System.Byte[]
7912dd40       14         2296 System.Char[]
7912d8f8       23        17604 System.Object[]
790fd8c4     2100       132680 System.String
Total 2379 objects
```

DumpHeap 命令的输出分为两部分。第一部分包含了当前位于托管堆中的所有对象。对于任何一个对象，都可以通过 DumpObject 命令来获得这个对象的详细信息。第二部分包含了关于托管堆行为的统计信息，其中相关的对象被归为一组，并给出了这组对象的方法表、对

象数量、总体大小以及对象的类型名。例如，在下面这行信息中：

```
79100a18       10          360 System.Security.PermissionSet
```

表示对象是 PermissionSet 类型，方法表位于0x79100a18，在托管堆中共有 10 个实例，总大小为 360 字节。在分析一个很大的托管堆以及需要找出哪些对象导致了堆空间增长时，这些统计信息非常有用。

然而，DumpHeap 命令输出的信息量非常大，因此我们很难从这些信息中找出特定的内存分配信息。幸运的是，DumpHeap 命令定义了许多开关选项来简化输出信息。例如，开关选项 -type 和 -mt 可以用来在托管堆中查找指定的类型名或者方法表的地址。如果在运行 DumpHeap 命令时使用了 -type 来查找程序中的内存分配信息，那么将得到以下信息：

```
0:003> !DumpHeap -type Advanced.NET.Debugging.Chapter5.Name
 Address       MT     Size
total 0 objects
Statistics:
      MT    Count    TotalSize Class Name
Total 0 objects
```

显然，从输出信息中可以看到，在当前托管堆中没有分配指定类型的对象。当然，这是因为在我们的程序中还没有执行内存分配操作。恢复程序的执行，直到看到提示信息“Press any key to exit”。然后再次中断到调试器，并且运行 DumpHeap 命令，同时指定 -type 开关：

```
0:004> !DumpHeap -type Advanced.NET.Debugging.Chapter5.Name
Address       MT     Size
01ca6c7c 002030cc       16
total 1 objects
Statistics:
      MT    Count    TotalSize Class Name
002030cc        1           16 Advanced.NET.Debugging.Chapter5.Name
Total 1 objects
```

这时，我们可以看到在托管堆中出现了指定类型的一个实例。输出信息的结构与 DumpHeap 命令的默认输出是一致的，首先给出的是这个实例的特定数据（地址、方法表和大小），然后是统计信息，指出在托管堆中只有一个这种类型的实例。

DumpHeap 命令还有其他几个有用的开关，可用于不同的调试任务中。表 5-1 详细列出了一些可用的开关。

表 5-1　DumpHeap 命令支持的开关

开　关	描　述
-stat	仅输出托管堆的统计信息
-strings	仅输出托管堆中的字符串
-short	仅输出托管堆中对象的地址
-min	仅输出大于指定最小对象大小的对象
-max	仅输出小于指定最大对象大小的对象

（续）

开 关	描 述
-thinlock	仅输出带有相关瘦锁（thinlock）的对象
-startAtLowerBound	从一个更小的下界开始遍历堆
-mt	仅输出拥有指定方法表的对象
-type	仅输出指定类型名字（匹配子字符串）的信息

至此，我们就结束了对 Windows 内存架构以及 CLR 堆管理器的讨论。我们已经清楚了 CLR 堆管理器如何组织内存以提供高效的内存管理模式，以及 CLR 堆管理器在满足内存分配请求时的完整处理流程。接下来将介绍 GC 自身的功能，它与 CLR 堆管理器的关系以及当内存被“丢弃”时如何释放。

有没有其他类型的 CLR 堆

除了执行“常规”内存分配的 CLR 堆之外，还有一些其他类型的堆。例如，当 JIT 编译器将 IL 转换为机器代码时会使用自己的堆，而在 CLR 加载器中使用了另一个堆。这些堆的内部信息大部分都没有公开，并且在调试会话中通常不会被用到（在 SOS 中没有考虑它们）。

5.2 垃圾收集器的内部工作机制

CLR GC 是一个高效的、可伸缩的以及可靠的自动内存管理器。研究人员在研究 GC 的最优化上投入了大量的时间和精力。在深入了解 CLR GC 的细节之前，我们首先要给出 GC 的定义和在设计和实现 GC 时遵循的一些假设。来看一些关键的假设：

- CLR GC 假设，如果没有特别声明，所有的对象都是垃圾。这意味着，除非特别声明，GC 会收集托管堆中的*所有*对象。从本质上来看，它为系统中所有活跃的（live）对象都实现了一种引用跟踪（reference tracking）模式（稍后将给出活跃的定义），如果一个对象没有任何引用指向它，那么这个对象就被认为是垃圾对象，并且可以被收集。
- CLR GC 假设托管堆上所有对象的活跃时间都是短暂的（或者说只能存活很短的时间）。换句话说，相对于长久活跃的对象来说，GC 将更为频繁地收集短暂活跃的对象，这种行为基于的假设是：如果一个对象已经活跃了一段时间，那么它很可能在更长一段时间内也是活跃的，因此不需要再次收集这个对象。
- CLR GC 通过代（generation）的概念来跟踪对象的持续时间。活跃时间短的对象被归为第 0 代，而活跃时间更长的对象被归为第 1 代和第 2 代。随着对象活跃时间的增长，与其相应的代也会不断递增。因此，我们可以认为代定义了对象的年龄（即活跃时间）。

基于上述假设，我们可以得出 CLR GC 的定义：它是一个基于引用跟踪和代的垃圾收集器。

接下来将更详细地介绍这个定义中包含的各个方面，首先来看如何通过代来定义对象的年龄。

5.2.1 代

CLR GC 定义了 3 个级别的代，分别称之为第 0 代、第 1 代和第 2 代。在每一代中都包含了特定年龄（age）的对象，其中第 0 代包含了新近分配的对象，而第 2 代包含了年龄最大的对象。一个对象可以从某一代迁移到下一代，以避免作为垃圾对象被收集。如果没有被收集，那么意味着在执行垃圾收集期间，这个对象仍将被引用（或者说仍然有根对象）。在任何时刻，每一代中的所有对象都可能作为垃圾对象，但执行垃圾收集操作的频率各不相同。在前面已经提到，CLR 基于的假设之一就是，它认为大多数对象都是短暂活跃的（即位于第 0 代中）。基于这个假设，第 0 代被收集的频率远远高于第 2 代被收集的频率，这是为了尽快地消除这些短暂存活的对象。图 5-5 给出了对各代中的对象进行垃圾收集的整体算法。

从图 5-5 中可以看到，当新的内存分配请求到来并且第 0 代中无法再容纳新的对象时，就会触发垃圾收集过程。如果是这种情况，那么垃圾收集器将收集所有不存在根对象（root）与之相关联的对象，并且将所有带有根对象的对象提升到第 1 代。正如在第 0 代中定义了预定空间容量，在第 1 代中同样也定义了一个预定空间容量，如果将第 0 代中的对象提升到第 1 代时超过了第 1 代的预定空间容量，那么 GC 将在第 1 代中收集没有根对象的对象，并将有根对象的对象提升到第 2 代。在第 2 代中会发生相同的动作。如果在提升到了第 2 代后，GC 仍无法收集任何对象并且超过了第 2 代的预定空间容量，那么 CLR 堆管理器会尝试分配另一个内存段来容纳第 2 代中的对象。如果在创建新的内存段时失败了，就会抛出一个 OutOfMemoryException 异常。如果内存段不再被使用，那么 CLR 堆管理器将释放它们，我们将在本章的后面详细介绍这个过程。

有没有其他的情况也会触发垃圾收集动作

除了由于在分配内存时超出了第 0、1、2 代的阈值而触发的垃圾收集动作外，还有其他一些情况也会触发 GC 动作。首先，可以通过 GC. Collect 以及相关的 API 来强制执行垃圾收集。其次，垃圾收集器非常清楚系统中内存的使用情况，通过与操作系统之间进行协调，当系统整体上存在着极大的内存压力时，垃圾收集器就会启动收集动作。

现在，我们来观察一个对象是如何被收集和提升的。清单 5-2 中的源代码很好地说明了代的概念。

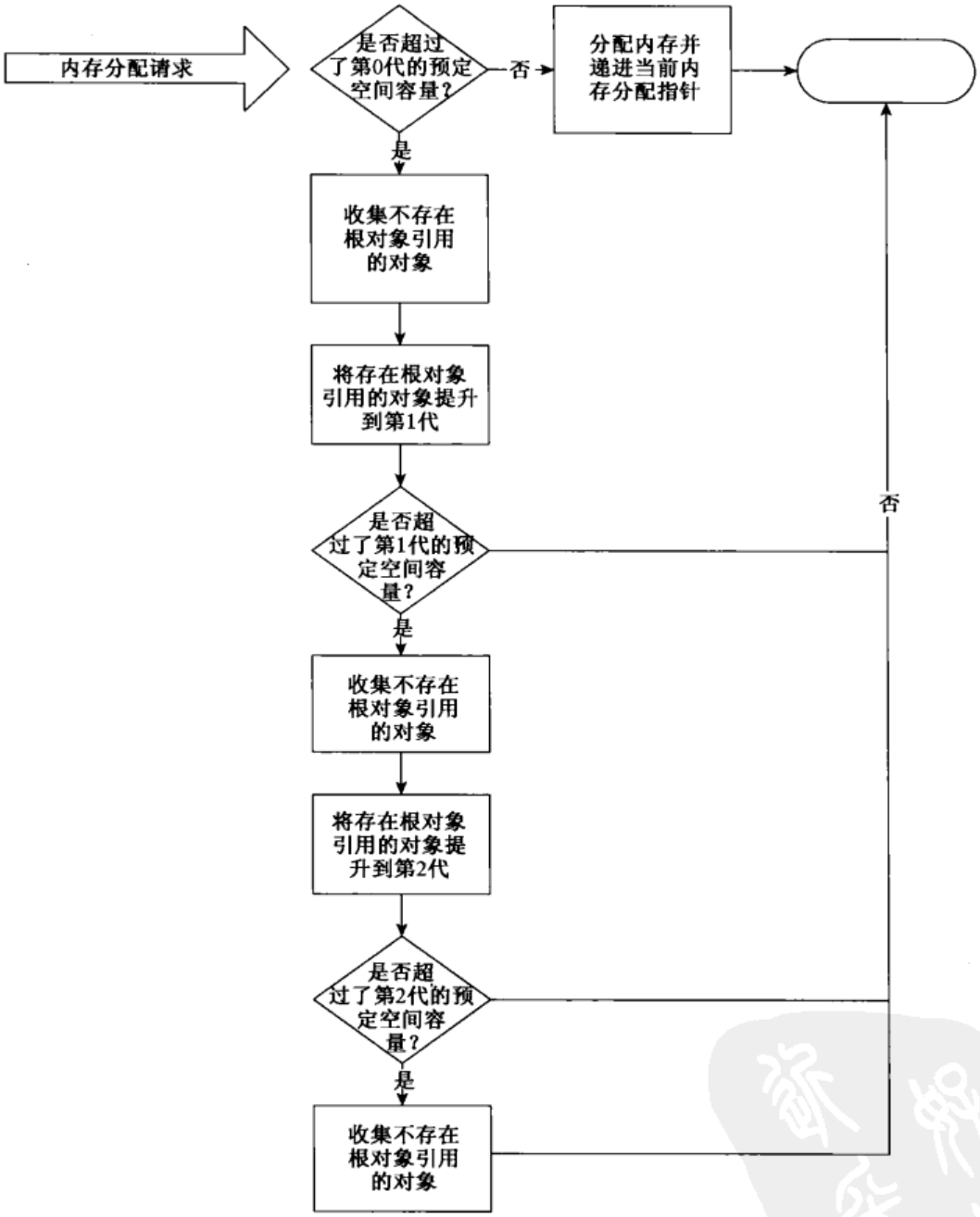


图 5-5　基于代的垃圾收集算法

清单 5-2　说明代的概念的示例代码

```
using System;
using System.Text;
using System.Runtime.Remoting;
```

```
namespace Advanced.NET.Debugging.Chapter5
{
    class Name
    {
        private string first;
        private string last;

        public string First { get { return first; } }
        public string Last { get { return last; } }

        public Name(string f, string l)
        {
            first = f; last = l;
        }
    }
    class Gen
    {
        static void Main(string[] args)
        {
            Name n1 = new Name("Mario", "Hewardt");
            Name n2 = new Name("Gemma", "Hewardt");

            Console.WriteLine("Allocated objects");

            Console.WriteLine("Press any key to invoke GC");
            Console.ReadKey();

            n1 = null;
            GC.Collect();

            Console.WriteLine("Press any key to invoke GC");
            Console.ReadKey();

            GC.Collect();

            Console.WriteLine("Press any key to exit");
            Console.ReadKey();
        }
    }
}
```

清单 5-2 中的源代码和二进制文件位于以下文件夹中：

- 源代码文件：C:\ADND\Chapter5\Gen
- 二进制文件：C:\ADNDBin\05Gen. exe

在清单 5-2 中，我们定义了一个简单的类型 Name。在 Main 方法中创建了两个 Name 类型的实例，它们都将进入第 0 代。在用户看到提示信息“Press any key to invoke GC”并按下任意键之后，将实例 n1 设置为 null，这表示这个实例可以被收集，因为它不再拥有任何根对

象。接下来将执行垃圾收集动作，收集 n1 并将 n2 提升到第 1 代。最后一次垃圾收集动作会把 n2 提升到第 2 代，因为它仍然有根对象。

现在，让我们在调试器下运行这个程序，观察并验证我们对 n1 和 n2 的收集过程以及提升过程的分析。在调试器下运行程序，一直执行到第一次出现提示信息“Press any key to invoke GC”。此时中断程序的执行，并且找到这两个实例的地址，这可以通过 ClrStack 命令来实现，如下所示：

```
0:000> !ClrStack -a
OS Thread Id: 0x1c0c (0)
ESP       EIP
0028f3b4 77709a94 [NDirectMethodFrameSlim: 0028f3b4]
 Microsoft.Win32.Win32Native.ReadConsoleInput(IntPtr, InputRecord ByRef, Int32,
Int32 ByRef)
0028f3cc 793e8f28 System.Console.ReadKey(Boolean)
    PARAMETERS:
        intercept = 0x00000000
    LOCALS:
        <no data>
        0x0028f3dc = 0x00000001
        <no data>
        <no data>
        <no data>
        <no data>
        <no data>
        <no data>
        <no data>
        <no data>

0028f40c 793e8e33 System.Console.ReadKey()
0028f410 003000f3 Advanced.NET.Debugging.Chapter5.Gen.Main(System.String[])
    PARAMETERS:
        args = 0x01c55818
    LOCALS:
        <CLR reg> = 0x01da5938
        <CLR reg> = 0x01da5948

0028f65c 79e7c74b [GCFrame: 0028f65c]
```

这两个对象在托管堆上的地址分别为 0x01da5938 和 0x01da5948。如何找出托管堆上的这两个对象分别属于第几代？这就需要理解托管堆中的内存段与代之间的关系。在前面已经讨论过，每个托管堆都包含了一个或者多个内存段来容纳对象。而且，在这些内存段中有一部分空间是专门用于存放指定的代。在图 5-6 中给出了一个托管堆的内存段示例。

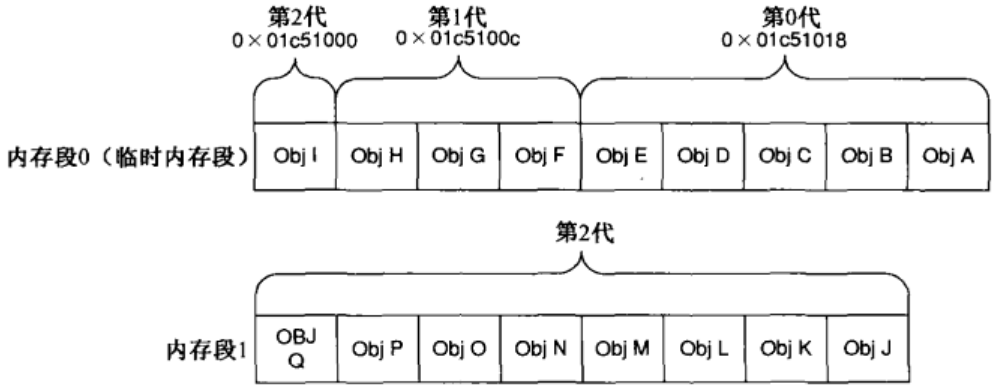


图 5-6　托管堆中的内存段

在图 5-6 中，托管堆内存段被划分为 3 部分，分别存放不同代的对象，其中每一部分都有自己的起始地址，这个地址由 CLR 堆管理器来管理。第 0 代和第 1 代属于同一个内存段，这个内存段被称为临时内存段（ephemeral segment），它包含的是短暂活跃的对象。由于 GC 假设大多数对象都是短暂活跃的，因此它认为大多数对象的活跃时间都不会超过第 0 代，最多不超过第 1 代。位于第 2 代的对象都是存活时间最长的对象，它们被收集的频率非常低。第 2 代的对象也可能位于这个临时内存段中，虽然第 2 代对象并不会被频繁地收集。通过查看对象的地址和了解存放每代对象的地址范围，我们可以找出这个对象属于哪一代。我们如何知道每一代在 CLR 堆管理器中的起始地址？答案就是使用 eeheap 命令。eeheap 命令能输出内部 CLR 数据结构的各种内存统计信息。在默认情况下，eeheap 将输出与 GC 和加载器相关的全部信息。如果只需输出关于 GC 的信息，可以使用开关选项 -gc。在现有的调试会话中运行这个命令，将看到以下输出：

```
0:004> !eeheap -gc
Number of GC Heaps: 1
generation 0 starts at 0x01da1018
generation 1 starts at 0x01da100c
generation 2 starts at 0x01da1000
ephemeral segment allocation context: none
 segment    begin allocated     size
002c7db0 790d8620  790f7d8c 0x0001f76c(128876)
01da0000 01da1000  01da8010 0x00007010(28688)
Large object heap starts at 0x02da1000
 segment    begin allocated     size
02da0000 02da1000  02da3250 0x00002250(8784)
Total Size   0x289cc(166348)
------------------------------
GC Heap Size   0x289cc(166348)
```

在输出信息中很清楚地给出了每一代的起始地址。如果再看看在示例程序的调试会话中输出的对象地址，我们可以得到以下信息：

```
<CLR reg> = 0x01da5938
<CLR reg> = 0x01da5948
```

这两个地址与程序中的对象一一对应，它们都位于第 0 代（起始地址位于 0x01da1018）的地址范围内，因此我们可以得出结论，这两个对象都属于第 0 代。这是很有意义的，因为在当前所处的代码位置上，这两个对象刚刚被分配，并且接下来会执行垃圾收集操作。恢复程序的运行，然后当再次看到提示信息“Press any key to invoke GC”时中断到调试器，我们会看到这些对象属于不同的代。如果观察源代码，可以看到在执行垃圾收集之前，我们将 n1 设置为 null，从而使这个对象没有根对象，因此将被作为垃圾对象收集。然而，n2 仍然有根对象，因此在垃圾收集操作中会被提升到第 1 代。再次重复前面相同的步骤：找到对象的地址，通过 eeheap 命令来找到每代的地址范围，并且查看这些对象都属于哪一代：

```
0:000> !ClrStack -a
OS Thread Id: 0x1910 (0)
ESP       EIP
0021f394 77709a94 [NDirectMethodFrameSlim: 0021f394]
 Microsoft.Win32.Win32Native.ReadConsoleInput(IntPtr, InputRecord
ByRef, Int32, Int32 ByRef)
0021f3ac 793e8f28 System.Console.ReadKey(Boolean)
    PARAMETERS:
        intercept = 0x00000000
    LOCALS:
        <no data>
        0x0021f3bc = 0x00000001
        <no data>
        <no data>
        <no data>
        <no data>
        <no data>
        <no data>
        <no data>
        <no data>
0021f3ec 793e8e33 System.Console.ReadKey()
0021f3f0 01690111 Advanced.NET.Debugging.Chapter5.Gen.Main(System.String[])
    PARAMETERS:
        args = 0x01da5818
    LOCALS:
        <CLR reg> = 0x00000000
        <CLR reg> = 0x01da5948

0021f644 79e7c74b [GCFrame: 0021f644]
0:000> !eeheap -gc
Number of GC Heaps: 1
generation 0 starts at 0x01da6c00
generation 1 starts at 0x01da100c
generation 2 starts at 0x01da1000
ephemeral segment allocation context: none
```

```
 segment    begin allocated     size
002c7db0 790d8620  790f7d8c 0x0001f76c(128876)
01da0000 01da1000  01da8c0c 0x00007c0c(31756)
Large object heap starts at 0x02da1000
 segment    begin allocated     size
02da0000 02da1000  02da3240 0x00002240(8768)
Total Size   0x295b8(169400)
------------------------------
GC Heap Size   0x295b8(169400)
```

在上述输出信息中，最需要注意的部分是 eeheap 命令的输出。现在，我们可以看到代的地址范围发生了一些变化。具体来说，第 0 代的起始地址由 0x01da1018 变成了 0x01da6c00，这实际上意味着第 1 代的空间将变得更大（由于第 1 代的起始地址保持不变）。如果将对象 n2 的地址（0x01da5948）与 eeheap 命令显示的代地址范围关联起来，我们可以看到对象 n2 位于第 1 代。这正是我们预期的结果，因为 n2 起初位于第 0 代，并且在执行垃圾收集时仍然存在根对象，因此会把这个对象提升到下一代。至于观察程序中最后一次垃圾收集后的结果，我将其作为一个练习留给读者。

尽管通过 SOS 调试器扩展可以找出某个对象属于哪一代，但这个过程也有些繁琐，因为它要求将对象的地址与托管堆段中的代地址进行比较，而这些代地址可能会发生变化。而且，没有任何方法能够列出某个代中的所有对象，从而就很难知道诸如每代的整体利用率等信息。幸运的是，SOSEX 调试器扩展能够解决这个问题，它提供了一个命令 dumpgen。通过使用 dumpgen 命令并将代作为命令的参数，可以很容易地得到某个代中的所有对象。例如，以清单 5-2 中的一个程序为例，以下是运行 dumpgen 之后的输出：

```
0:000> !dumpgen 0
01da6c00              12 **** FREE ****
01da6c0c              68 System.Char[]
2 objects, 80 bytes
0:000> !dumpgen 1
01da100c              12 **** FREE ****
01da1018              12 **** FREE ****
01da1024              72 System.OutOfMemoryException
01da106c              72 System.StackOverflowException
01da10b4              72 System.ExecutionEngineException
01da10fc              72 System.Threading.ThreadAbortException
01da1144              72 System.Threading.ThreadAbortException
01da118c              12 System.Object
01da1198              28 System.SharedStatics
01da11b4             100 System.AppDomain
...
...
...
01da5948              16 Advanced.NET.Debugging.Chapter5.Name
01da5958              28 Microsoft.Win32.Win32Native+InputRecord
01da5974              12 System.Object
```

```
01da5980          20 Microsoft.Win32.SafeHandles.SafeFileHandle
01da5994          36 System.IO.__ConsoleStream
01da59b8          28 System.IO.Stream+NullStream
...
...
...
```

这里我们并没有看到第 0 代中的所有对象，而是看到了许多第 1 代中的对象，包括位于地址 0x01da5948 上的实例 n2。在查看代的特定信息时，dumpgen 命令的确能使工作变得更加简单。

GC. Collect（）的作用是什么

你已经看到了，清单 5-2（以及在本章中从头到尾）的源代码中包含了对 GC. Collect（）的调用。GC. Collect 所实现的功能比其字面上表示的功能要多得多。它能强行触发一次垃圾收集操作，而不管实际上是否需要垃圾收集。这句话的后半部分非常重要："…而不管实际上是否需要垃圾收集"。在应用程序执行期间，GC 能不断地自我调节，以确保在应用程序的环境中表现出最优行为。然而，通过执行 GC. Collect( )来强制执行垃圾收集，有可能会破坏 GC 的自我微调算法。因此，在通常情况下，我们强烈建议不使用这个 API。在本书中使用这个 API 只是为了使示例程序表现出更具确定性的行为。

到目前为止，我们已经讨论了托管堆内存段中的对象如何被分成不同的代，这些对象是否仍然能被引用（或者说是否仍然存在引用它们的根对象），以及它们如何作为垃圾对象被收集或者被提升到下一代。还有一个问题没有回答，那就是对于一个对象来说，根对象引用的含义是什么？在下一节中将介绍根对象的概念，当 GC 判断一个对象是否需要被收集时，根对象将起到核心作用。

5.2.2 根对象

在垃圾收集中，一个最重要的功能就是要能判断哪些对象仍然被引用，哪些对象没有被引用，从而可以作为垃圾被收集。与通常看法不同的是，GC 本身并不会检测哪些对象仍然被引用，而是使用 CLR 中其他了解对象生命周期的组件。CLR 通过以下组件来判断哪些对象仍然被其他对象引用：

- 即时编译器（Just-in-time Compiler，JIT）。JIT 编译器负责将 IL 转换为机器代码，因此能够清楚地知道在任意时刻有哪些局部变量仍然被认为是活跃的。JIT 编译器将这些信息维护在一张表中，当 GC 在后面查询仍然存活的对象时，会用到这张表。

发行构建与调试构建

请注意，发行构建（retail build）与调试构建（debug build）在 JIT 编译器跟踪局部变量活跃性的方面存在一个差异。在发行构建中，JIT 编译器会变得非常主动，甚至在一个局

部变量离开作用域之前就认为它已经无效了（假设这个变量没有被使用）。在调试构建中，这种行为在调试时会带来一些问题，因此在调试构建中需要保持所有的局部变量都是活跃的，直到超出变量的作用域。

- 栈遍历器（stack walker）。当对执行引擎进行非托管调用时，将使用栈遍历器。在这些调用中使用的任意托管对象都必须属于引用跟踪系统（reference tracking system）。
- 句柄表（handle table）。CLR 为每个应用程序域提供了一组句柄表，在这些句柄表中包含了指向托管堆上固定（pinned）引用类型的指针。在 GC 查询期间，会通过这些句柄表来找出托管堆上对象的活跃引用。
- 终结队列（finalize queue）。前面曾经简要地讨论了对象终结器的概念，我们现在暂时把带有终结器的对象视为这样的对象：虽然从应用程序的角度来看，这些对象已经消亡了，但仍然能保持活跃以便被清除。
- 如果对象是上述任何一种类别对象中的一个成员。

在上面的探查阶段中，GC 还会根据对象的状态（存在根对象引用）来标记它们。一旦所有的组件都被探查了，GC 会对所有对象启动垃圾收集操作，并将所有仍然存在根对象引用的对象提升到下一代。如果给定托管堆上某个对象的地址，那么是否可以找出该对象是否存在根对象引用，如果存在的话，那么对象的引用链（reference chain）又是怎样？我们再次借助 SOS 扩展中的命令 gcroot。在 gcroot 命令中使用了一种类似于上面利用 GC 找出对象活跃性的技术。我们来看一段示例代码。清单 5-3 给出了一个程序的源代码，这个程序为不同作用域中的类型定义了一组类型和引用。

清单 5-3　说明根对象的示例程序

```
using System;
using System.Text;
using System.Threading;

namespace Advanced.NET.Debugging.Chapter5
{
    class Name
    {
        private string first;
        private string last;

        public string First { get { return first; } }
        public string Last { get { return last; } }

        public Name(string f, string l)
    {
        first = f; last = l;
```

```
    }
}

class Roots
{
    public static Name CompleteName = new Name ("First", "Last");

    private Thread thread;
    private bool shouldExit;

    static void Main(string[] args)
    {
        Roots r = new Roots();
        r.Run();
    }

    public void Run()
    {
        shouldExit = false;

        Name n1 = CompleteName;

        thread = new Thread(this.Worker);
        thread.Start(n1);

        Thread.Sleep(1000);

        Console.WriteLine("Press any key to exit");
        Console.ReadKey();

        shouldExit = true;

    }

    public void Worker(Object o)
    {
        Name n1 = (Name)o;
        Console.WriteLine("Thread started {0}, {1}",
                          n1.First,
                          n1.Last);

        while (true)
        {
            // 执行一些工作…
                Thread.Sleep(500);
                if (shouldExit)
                    break;
            }
        }
    }
}
```

清单 5-3 中的源代码和二进制文件位于以下文件夹中：

- 源代码文件：C:\ADND\Chapter5\Roots
- 二进制文件：C:\ADNDBin\05Roots.exe

在清单 5-3 的源代码中声明了一个 Name 类型的静态实例。程序在 Run 方法中声明了一个引用指向这个静态实例，然后启动一个线程并将这个引用传递给新创建的线程。在新线程执行的方法中使用这个传递进去的引用，直到用户按下任意一个键，此时工作线程和应用程序都会结束。在这个练习中，我们需要注意的对象就是静态域 CompleteName。从源代码中，我们可以得到 CompleteName 的以下特性：

- 在 Roots 类中有一个静态域指向这个对象实例，它是指向这个对象的第一个根对象引用。
- 在 Run 方法中，我们分配了一个句柄变量（n1）来引用这个对象实例，这是第二个根对象引用。在线程启动后，局部变量 n1 将不再使用，甚至在达到这个方法的作用域之前（在发行构建中）就已经失效。在调试构建中保证这个引用是有效的，直到达到作用域的末尾。
- 在 Run 方法中，我们将句柄变量引用 n1 传递给线程方法，这是第三个根对象引用。

在调试器下运行这个程序，并在出现提示信息“Press any key to exit”时中断程序的执行。我们需要找出的第一个信息就是被观察对象的地址（并转储出这个对象），然后在这个地址上运行 gcroot 命令：

```
0:005> ~0s
eax=002cef9c ebx=002cef94 ecx=792274ec edx=79ec9058 esi=002cedf0 edi=00000000
eip=77709a94 esp=002ceda0 ebp=002cedc0 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
ntdll!KiFastSystemCallRet:
77709a94 c3              ret
0:000> !ClrStack -a
OS Thread Id: 0x2358 (0)
ESP       EIP
002cef6c 77709a94 [NDirectMethodFrameSlim: 002cef6c]
 Microsoft.Win32.Win32Native.ReadConsoleInput(IntPtr, InputRecord ByRef, Int32,
Int32 ByRef)
002cef84 793e8f28 System.Console.ReadKey(Boolean)
    PARAMETERS:
        intercept = 0x00000000
    LOCALS:
        <no data>
        0x002cef94 = 0x00000001
        <no data>
        <no data>
        <no data>
        <no data>
        <no data>
        <no data>
        <no data>
        <no data>
```

```
002cefc4 793e8e33 System.Console.ReadKey()
002cefc8 00890212 Advanced.NET.Debugging.Chapter5.Roots.Run()
    PARAMETERS:
        this = 0x01c758e0
    LOCALS:
        <CLR reg> = 0x01c758d0

002cefe8 0089013f Advanced.NET.Debugging.Chapter5.Roots.Main(System.String[])
    PARAMETERS:
        args = 0x01c75888
    LOCALS:
        <CLR reg> = 0x01c758e0

002cf208 79e7c74b [GCFrame: 002cf208]
0:000> !do 0x01c758d0
Name: Advanced.NET.Debugging.Chapter5.Name
MethodTable: 001b311c
EEClass: 001b13a0
Size: 16(0x10) bytes
 (C:\ADNDBin\05Roots.exe)
Fields:
      MT    Field   Offset                 Type VT     Attr    Value Name
790fd8c4  4000001        4        System.String  0 instance 01c75898 first
790fd8c4  4000002        8        System.String  0 instance 01c758b4 last
0:000> !gcroot 0x01c758d0
Note: Roots found on stacks may be false positives. Run "!help gcroot" for
more info.
Scan Thread 0 OSTHread 2358
ESP:2cefbc:Root:01c758d0(Advanced.NET.Debugging.Chapter5.Name)
Scan Thread 1 OSTHread 1630
Scan Thread 3 OSTHread 254c
ESP:47df428:Root:01c758d0(Advanced.NET.Debugging.Chapter5.Name)
ESP:47df42c:Root:01c758d0(Advanced.NET.Debugging.Chapter5.Name)
ESP:47df438:Root:01c758d0(Advanced.NET.Debugging.Chapter5.Name)
ESP:47df4d0:Root:01c75984(System.Threading.ThreadHelper)->
01c758d0(Advanced.NET.Debugging.Chapter5.Name)
ESP:47df4d8:Root:01c75984(System.Threading.ThreadHelper)->
01c758d0(Advanced.NET.Debugging.Chapter5.Name)
ESP:47df4f4:Root:01c75984(System.Threading.ThreadHelper)->
01c758d0(Advanced.NET.Debugging.Chapter5.Name)
ESP:47df500:Root:01c75984(System.Threading.ThreadHelper)->
01c758d0(Advanced.NET.Debugging.Chapter5.Name)
ESP:47df5c0:Root:01c758d0(Advanced.NET.Debugging.Chapter5.Name)->
01c758d0(Advanced.NET.Debugging.Chapter5.Name)
ESP:47df5c4:Root:01c75998(System.Threading.ParameterizedThreadStart)->
01c75984(System.Threading.ThreadHelper)
ESP:47df754:Root:01c758d0(Advanced.NET.Debugging.Chapter5.Name)->
01c75984(System.Threading.ThreadHelper)
ESP:47df758:Root:01c75998(System.Threading.ParameterizedThreadStart)->
01c75984(System.Threading.ThreadHelper)
ESP:47df764:Root:01c75998(System.Threading.ParameterizedThreadStart)->
```

```
01c75984(System.Threading.ThreadHelper)
ESP:47df76c:Root:01c758d0(Advanced.NET.Debugging.Chapter5.Name)->
01c75984(System.Threading.ThreadHelper)
DOMAIN(0037FCF8):HANDLE(Pinned):a13fc:Root:02c71010(System.Object[])->
01c758d0(Advanced.NET.Debugging.Chapter5.Name)
```

从 gcroot 的输出中可以看到，这个命令扫描了各种不同来源去查找并构建指向这个对象的引用链。无论哪种来源，GCRoot 命令的输出都将遵循以下格式：

```
<root>-><reference 1>-><reference 2>-><reference X>-><object>
```

根据不同的来源，每个元素的格式会略有不同，如下所示：

- 线程栈上的局部变量。格式中的 roo 元素通常遵循以下格式：< stack register >：< stack pointer > :Root：< object >。其中“stack register”取决于架构。例如，在 x86 机器上，会显示 ESP，而在 x64 机器上则显示 RSP。“stack pointer”给出的是根对象引用在栈上的地址，“object address”中包含的是持有指向引用链中下一个对象的引用。我们来看一个示例：

```
ESP:47df428:Root:01c758d0(Advanced.NET.Debugging.Chapter5.Name)
```

  可以看到，在栈（ESP）的位置 0x047df428 上有一个局部变量。而且，输出信息指出这是一个根对象引用，指向地址 0x01c758d0 上的对象，而这个对象是一个指向 Advanced. NET. Debugging. Chapter5. Name 类型的引用。

- 句柄表。当 GCRoot 查找指定对象的引用时，会扫描所有的句柄表。如果找到了一个引用，那么命令的输出将遵循以下格式：

```
DOMAIN(<address>):HANDLE(<type>):<handleaddress>:Root:<object>。
```

其中“domain address”表示这个句柄引用所属的应用程序域的地址。“handletype”表示句柄的类型，它的取值包括 Weak、WeakTracResurrection、Normal 和 Pinned。

接下来是“handleaddress”，表示句柄本身的地址。请牢记，句柄类型是一种值类型，要想转储出句柄的内容，必须使用 DumpVC 命令而不是 DumpObj。最后，给出了根对象的地址。我们来看一个示例：

```
DOMAIN(002EFCD8):HANDLE(Pinned):2813fc:Root:02c81010
(System.Object[])->01c858d0(Advanced.NET.Debugging.
Chapter5.Name)
```

上面的输出表示：位于地址 0x01c858d0 上的对象存在一个根引用对象，它位于应用程序域的句柄表中，地址为 0x002efcd8。而且，存放这个引用的句柄值的地址为 0x002813fc，句柄类型为 Pinned。最后，存放引用的实际对象位于地址 0x02c81010 上，类型为 System. Object[]。

- F-Reachable 队列（F-Reachable Queue）。GC 通过扫描 F-Reachable 队列来查看是否存在指向这个对象的任何引用。如果在 F-Reachable 队列中找到了一个指向该对象的

根对象应用，那么会按照以下格式显示出来。

Finalizer queue:Root: < object address > (< object type >)。在输出信息的第一部分表示根对象引用来自于 F-Reachable 队列。接下来，输出的是被引用对象的地址以及对象的类型。下面是运行 GCRoot 命令的一个示例输出，其中命令的参数为 F-Reachable 队列上的一个对象：

```
Finalizer
queue:Root:01d15750(Advanced.NET.Debugging.Chapter5.Name)
```

从上面的输出我们可以看到，地址 0x01d15750 上类型为 Advanced. NET. Debugging. Chapter5. Name 的对象存在一个根对象引用，位于 F-Reachable 队列中。

- GCRoot 命令的最后一个输出来源是判断对象是否是上述任何类别对象中的一个成员。

在 GCRoot 和局部变量中还存在一个问题，即它输出的信息并不总是准确的，因此可能产生误报（False Positive）。为了保证在输出信息中列出的栈位置是准确的，我们必须检查每个栈位置并将其关联到源代码，这样才可以判断局部变量是否仍在引用这个对象。例如，假设有下面这个简单函数：

```
    public void Run()
    {
        Name n1 = new Name("A", "B");

        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
}
```

在上面的源代码中创建了一个 Name 类型的实例，并将其赋值给局部变量 n1。如果在 n1 上运行 GCRoot 命令，那么会看到在线程栈上只存在一个引用，如下所示：

```
0:000> !GCRoot 0x01e9580c
Note: Roots found on stacks may be false positives. Run "!help gcroot" for
more info.
Scan Thread 0 OSTHread 1638
ESP:1df29c:Root:01e9580c(Advanced.NET.Debugging.Chapter5.Name)
ESP:1df2a0:Root:01e9580c(Advanced.NET.Debugging.Chapter5.Name)
Scan Thread 2 OSTHread 14ac
```

输出信息清楚地表明了在线程栈上有两个引用指向该对象。为什么这种方式是可行的？GCRoor 命令的工作方式是，它假设栈上的每个地址都是某个对象的地址，并通过各种元数据信息来验证这种假设。按照这种假设，之前在栈上出现的对象将被认为是这些对象的引用，并且会在 GCRoot 的输出信息中列出。如果怀疑 GCRoot 的输出信息不正确，那么最好的方法就是通过命令 U 来反汇编栈帧（stack frame），并将 GCRoot 输出中的栈寄存器与反汇编代码关联起来从而判断哪些对象是真正有效的。

5.2.3 终结操作

到目前为止，我们所描述的垃圾收集机制假设被收集的对象不需要任何特殊的清理代码。

有时候，需要在对象中封装其他一些资源，并且在销毁对象时清除这些资源。一个常见的示例就是在对象中封装一个底层的资源，例如文件句柄。如果没有明确的清除代码，那么尽管托管对象的内存被 GC 清除，但对象所封装的底层句柄却不会被清除（因为 GC 并不知道这些句柄的存在）。显然，这样的结果造成了资源浪费。为了提供合适的清除机制，CLR 引入了终结器（finalizer）的概念。终结器的作用类似于 C++ 中的析构函数。当一个对象被释放时（或者被作为垃圾收集时），析构函数（或者终结器）就会运行。在 C#中，终结器的声明方式类似于 C++ 中的析构函数，使用的语法是 ~ <class name>()。如下所示：

```
public class MyClass
{
        ...
        ...
        ...
        ~MyClass()
        {
                //执行清除工作的代码
        }
}
```

当这个类被编译为 IL 时，终结方法会被编译为一个名为 Finalize 的函数。带有终结器的对象的关键点在于，垃圾收集器在处理这些对象时采用的方式与其他对象略有不同。由于垃圾收集器实际上是一个自动内存管理器，因此在垃圾收集过程中，它需要执行对象的终结代码。为了记录哪些对象拥有终结器，垃圾收集器维护了一个终结队列（Finalization Queue）。如果在托管堆上创建的对象中包含有终结器，那么在创建过程中将被自动放入终结队列中。需要注意的是，终结队列并没有包含那些被认为是垃圾的对象，而是包含了所有带有终结器并在托管堆上处于活跃状态的对象。如果某个带有终结器的对象不存在任何根对象引用了，并且此时启动了垃圾收集过程，那么 GC 会把这个对象放入另一个队列中，即 F-Reachable 队列。这个队列包含了所有带有终结器并且将被作为垃圾收集的对象，这些对象的终结器都将被执行。在 F-Reachable 队列上的所有对象都被视为仍然存在根对象引用。需要注意的是，在垃圾收集过程中并不会执行 F-Reachable 队列中每个对象的终结器代码，这些代码将在一个特殊的线程中执行，它就是每个.NET 进程的终结线程（Finalization Thread）。在收到 GC 的请求时，终结线程会启动并且查看 F-Reachable 队列的状态。如果在 F-Reachable 队列上有任何对象存在，那么它会依次执行这些对象的终结方法。

为什么不在垃圾收集过程中执行终结方法

由于在终结方法中包含了托管代码，并且在 GC 过程中托管代码线程被挂起，因此终结线程不在 GC 过程中运行。

在垃圾收集过程结束后，带有终结器的对象会出现在 F-Reachable 队列中（根对象引用存在且是活跃的），直到终结线程执行它们的 Finalize 方法。此时，对象将从 F-Reachable 队列中移走，并且这些对象也被认为不存在根对象引用，从而可以真正地被垃圾收集器回收。下一次启动垃圾收集过程时，这些对象会被收集。在图 5-7 中给出了终结过程的示意图。

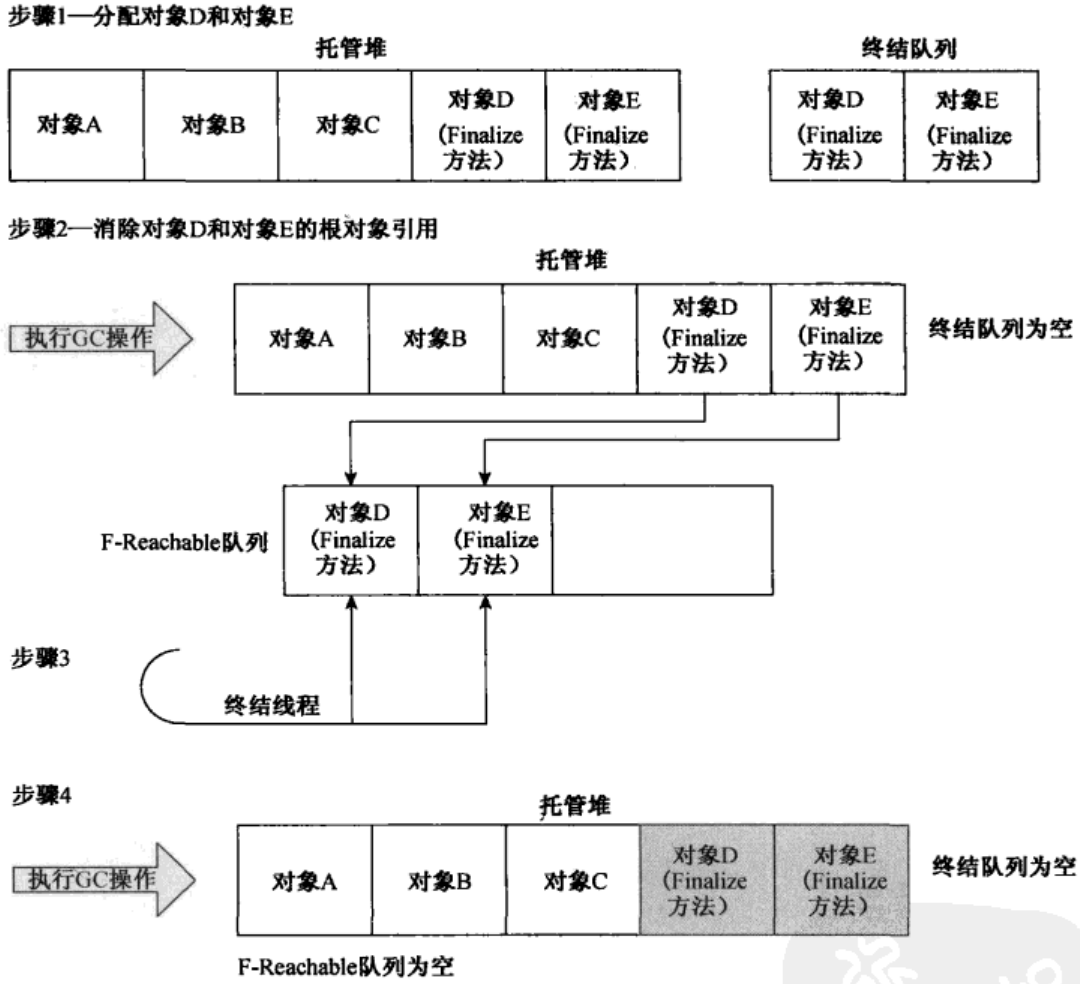


图 5-7　终结过程示例

在图 5-7 的步骤 1 中分配对象 D 和对象 E，它们各自带有一个 Finalize 方法。在分配过程中，这些对象除了被放在托管堆上外，还被放在终结队列中，表明这些对象不被使用时需要执行终结操作。在步骤 2 中，当垃圾收集过程启动时，Obj D 和 Obj E 都不存在根对象引用。此时，这两个对象将从终结队列中移动到 F-Reachable 队列中，表明可以运行它们的 Finalize 方法了。在接下来的某个时刻，步骤 3 会被执行，终结线程也会启动并且开始执行这两个对象上的 Finalize 方法。即使在终结器执行完成后，这两个对象仍然位于 F-Reachable 队列中。最后在步骤 4 中再次启动了垃圾收集过程，这些对象会被移出 F-Reachable 队列（不再有根对

象引用)，然后由垃圾收集器从托管堆中回收。需要注意的是，虽然有一个专门的线程来执行这些 Finalize 方法，但 CLR 并不保证这些线程将在何时启动并执行。因此，带有终结器的对象在被真正清除之前，可能需要等待一段时间。由于在对象中包含了一些资源和在等待资源被回收时需要的时间过长，所以这种方式或许行不通。在这种情况下，最好有一种明确的清除模式，例如 IDisposable 或者 Close 等模式。最后，通过一个专门的线程来执行 Finalize 方法意味着你将无法控制这个线程的状态，如果基于这个状态来做出一些假设，那么可能会对程序造成破坏。我们来看一个示例，在示例中有一个带有 Finalize 的对象，并且观察能否在垃圾收集期间跟踪对象的状态。清单 5-4 给出了示例程序的源代码。

清单 5-4　带有 finalize 方法的对象

```
using System;
using System.Text;
using System.Runtime.InteropServices;

namespace Advanced.NET.Debugging.Chapter5
{
    class NativeEvent
    {
        private IntPtr nativeHandle;

        public IntPtr NativeHandle { get { return nativeHandle; } }

        public NativeEvent(string name)
        {
            nativeHandle = CreateEvent(IntPtr.Zero,
                                       false,
                                       true,
                                       name);
        }

        ~NativeEvent()
        {
            if(nativeHandle!=IntPtr.Zero)
            {
                CloseHandle(nativeHandle);
                nativeHandle=IntPtr.Zero;
            }
        }
        [DllImport("kernel32.dll")]
        static extern IntPtr CreateEvent(IntPtr lpEventAttributes,
                                         bool bManualReset,
                                         bool bInitialState,
                                         string lpName);
```

```
        [DllImport("kernel32.dll")]
        static extern IntPtr CloseHandle(IntPtr lpEvent);
    }

    class Finalize
    {
        static void Main(string[] args)
        {
            Finalize f = new Finalize();
            f.Run();
        }

        public void Run()
        {
            NativeEvent nEvent = new NativeEvent("MyNewEvent");

            //
            // 使用 nEvent
            //

            nEvent = null;

            Console.WriteLine("Press any key to GC");
            Console.ReadKey();

            GC.Collect();

            Console.WriteLine("Press any key to GC");
            Console.ReadKey();

            GC.Collect();

            Console.WriteLine("Press any key to exit");
            Console.ReadKey();
        }
    }
}
```

清单 5-4 中的源代码和二进制文件位于以下文件夹中：

- 源代码文件：C:\ADND\Chapter5\Finalize
- 二进制文件：C:\ADNDBin\05Finalize.exe

在清单 5-4 的源代码中声明了一个类型 NativeEvent，这个类通过.NET 的互用性服务来包装 Windows 事件的创建过程。由于在创建事件时会返回一个句柄，因此在对象被销毁时必须关闭这个句柄，以免在程序中造成句柄泄漏。关闭句柄的操作是在 NativeEvent 的终结方法中实现的。程序的主体部分是在 Finalize 类中实现的。具体来说，Run 方法声明了一个 NativeEvent类的实例，并将这个句柄变量的引用设置为 null（表示它可以作为垃圾被收集），紧接着是强行启动垃圾收集操作。在第一次垃圾收集过程中，我们声明的 NativeEvent 实例会

发生哪些动作？根据前面的讨论，我们认为在垃圾收集操作执行之前，这个对象将位于终结队列中。而且，当垃圾收集操作发生时，这个对象肯定不存在根对象引用，并且被移动到 F-Reachable 队列，在这个队列中将维持一个指向该对象的引用，使得终结线程可以运行 Finalize 方法。需要记住的是，终结线程不会在垃圾收集过程中运行，而是在其他的某个时刻运行。在 Finalize 方法运行之后，这个对象将在下一次垃圾收集过程中被彻底回收。让我们来看看是否可以通过调试器来验证上述推论。在调试器下运行 05Finalize. exe，并且在出现提示信息“Press any key to GC”时中断程序的执行。在中断并进入到调试器后，可以使用 FinalizeQueue 命令来给出进程中可终结对象的状态：

```
0:004> !FinalizeQueue
SyncBlocks to be cleaned up: 0
MTA Interfaces to be released: 0
STA Interfaces to be released: 0
-------------------------------
generation 0 has 6 finalizable objects (003d3160->003d3178)
generation 1 has 0 finalizable objects (003d3160->003d3160)
generation 2 has 0 finalizable objects (003d3160->003d3160)
Ready for finalization 0 objects (003d3178->003d3178)
Statistics:
      MT    Count    TotalSize Class Name
00123128        1           12 Advanced.NET.Debugging.Chapter5.NativeEvent
7911c9c8        1           20 Microsoft.Win32.SafeHandles.SafePEFileHandle
791037c0        1           20 Microsoft.Win32.SafeHandles.SafeFileMappingHandle
79103764        1           20 Microsoft.Win32.SafeHandles.SafeViewOfFileHandle
79101444        1           20 Microsoft.Win32.SafeHandles.SafeFileHandle
790fe704        1           56 System.Threading.Thread
Total 6 objects
```

在输出信息中了包含了一些有用的内容。首先，给出了每一代的终结队列。在上面的输出中，第 0 代拥有 6 个可终结对象，而第 1 代和第 2 代没有任何可终结对象。对于每个终结队列，FinalizeQueue 对象还给出了队列本身的地址范围。例如，第 0 代的终结队列的起始地址为 0x003d3160 ，结束地址为 0x003d3178。可以使用 dd 命令来转储这个队列，如下所示：

```
0:004> dd 003d3160 l6
003d3160  01fc1df0 01fc5090 01fc5964 01fc5998
003d3170  01fc683c 01fc6850
```

可以通过 do 命令来进一步查看队列中的各个元素。例如，如果要查看地址 0x01fc5964 的对象的详细信息，可以使用以下命令：

```
0:004> !do 01fc5964
Name: Advanced.NET.Debugging.Chapter5.NativeEvent
MethodTable: 00123128
EEClass: 00121804
Size: 12(0xc) bytes
 (C:\ADNDBin\05Finalize.exe)
Fields:
      MT    Field   Offset                 Type VT     Attr    Value Name
791016bc  4000001        4        System.IntPtr  1 instance      1f0 nativeHandle
```

FinalizeQueue 命令中的另一个有用信息就是 F-Reachable 队列，如下所示：

```
Ready for finalization 0 objects (000c3178->000c3178)
```

这个输出信息表示此时没有任何对象可以执行终结操作。这是正确的，因为垃圾收集操作还没有被启动。

Finalize 命令输出的最后一部分是一些统计信息，其中包含了在终结队列或者 F-Reachable 队列中的所有对象。

在恢复程序执行之前，我们首先介绍在托管进程中包含的终结线程。这个线程的栈回溯会是怎样？要想知道答案，可以使用 ~ * kn 命令来显示进程中所有线程的栈回溯，包含栈帧的编号。在输出信息中，有一个线程值得我们注意：

```
      2  Id: 1a10.c10 Suspend: 1 Teb: 7ffdd000 Unfrozen
 # ChildEBP RetAddr
00 011cf604 77709254 ntdll!KiFastSystemCallRet
01 011cf608 7618c244 ntdll!ZwWaitForSingleObject+0xc
02 011cf678 79e789c6 KERNEL32!WaitForSingleObjectEx+0xbe
03 011cf6bc 79e7898f mscorwks!PEImage::LoadImage+0x1af
04 011cf70c 79e78944 mscorwks!CLREvent::WaitEx+0x117
05 011cf720 79ef2220 mscorwks!CLREvent::Wait+0x17
06 011cf73c 79fb997b mscorwks!WKS::WaitForFinalizerEvent+0x4a
07 011cf750 79ef3207 mscorwks!WKS::GCHeap::FinalizerThreadWorker+0x79
08 011cf764 79ef31a3 mscorwks!Thread::DoADCallBack+0x32a
09 011cf7f8 79ef30c3 mscorwks!Thread::ShouldChangeAbortToUnload+0xe3
0a 011cf834 79fb9643 mscorwks!Thread::ShouldChangeAbortToUnload+0x30a
0b 011cf85c 79fb960d mscorwks!ManagedThreadBase_NoADTransition+0x32
0c 011cf86c 79fba09b mscorwks!ManagedThreadBase::FinalizerBase+0xd
0d 011cf8a4 79f95a2e mscorwks!WKS::GCHeap::FinalizerThreadStart+0xbb
0e 011cf93c 76184911 mscorwks!Thread::intermediateThreadProc+0x49
0f 011cf948 776ee4b6 KERNEL32!BaseThreadInitThunk+0xe
10 011cf988 776ee489 ntdll!__RtlUserThreadStart+0x23
11 011cf9a0 00000000 ntdll!_RtlUserThreadStart+0x1b
```

从栈回溯中的第 6 个栈帧和第 7 个栈帧可以看出，这个线程实际上是进程中的终结线程。第 6 个栈帧尤其显示了，这个线程当前正在等待终结器事件（或者需要被终结的对象）。在第 6 个栈帧的返回地址处（0x79fb997b）设置一个断点，当终结线程执行工作时会触发这个断点：

```
bp 79fb997b
```

在设置了断点之后，恢复程序的执行，然后按下任意键来触发第一次垃圾收集操作。你会注意到，有一个断点被触发了，如下所示：

```
0:003> g
 Breakpoint 0 hit
eax=00000001 ebx=00000001 ecx=7618c42d edx=77709a94 esi=00000000 edi=00493a48
eip=79fb997b esp=00b7f768 ebp=00b7f770 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
mscorwks!WKS::GCHeap::FinalizerThreadWorker+0x79:
79fb997b 3bde            cmp     ebx,esi
```

这个断点就是之前设置的终结线程断点，表示终结器将在 F-Reachable 队列的对象上执行 Finalize 方法。如何找出 F-Reachable 队列中有哪些对象？答案是使用 FinalizeQueue 命令：

```
0:002> !FinalizeQueue
SyncBlocks to be cleaned up: 0
MTA Interfaces to be released: 0
STA Interfaces to be released: 0
-------------------------------
generation 0 has 0 finalizable objects (003d3170->003d3170)
generation 1 has 4 finalizable objects (003d3160->003d3170)
generation 2 has 0 finalizable objects (003d3160->003d3160)
Ready for finalization 2 objects (003d3170->003d3178)
Statistics:
      MT    Count    TotalSize Class Name
00123128        1           12 Advanced.NET.Debugging.Chapter5.NativeEvent
7911c9c8        1           20 Microsoft.Win32.SafeHandles.SafePEFileHandle
791037c0        1           20 Microsoft.Win32.SafeHandles.SafeFileMappingHandle
79103764        1           20 Microsoft.Win32.SafeHandles.SafeViewOfFileHandle
79101444        1           20 Microsoft.Win32.SafeHandles.SafeFileHandle
790fe704        1           56 System.Threading.Thread
```

这时，从输出信息中可以看出，在终结线程将要执行的 F-Reachable 队列中包含了两个对象，起始地址为 0x003d3160。如果将 F-Reachable 队列的内容以及每个对象都转储出来，那么可以看到以下内容：

```
0:002> dd 003d3170 l2
003d3170  01fc5090 01fc5964
0:002> !do 01fc5090
Name: Microsoft.Win32.SafeHandles.SafePEFileHandle
MethodTable: 7911c9c8
EEClass: 791fb61c
Size: 20(0x14) bytes
 (C:\Windows\assembly\GAC_32\mscorlib\2.0.0.0__b77a5c561934e089\mscorlib.dll)
Fields:
      MT    Field   Offset                 Type VT     Attr    Value Name
791016bc  40005c1        4        System.IntPtr  1 instance   3eab28 handle
79102290  40005c2        8         System.Int32  1 instance        4 _state
7910be50  40005c3        c       System.Boolean  1 instance        1 _ownsHandle
7910be50  40005c4        d       System.Boolean  1 instance        1
_fullyInitialized
0:002> !do 01fc5964
Name: Advanced.NET.Debugging.Chapter5.NativeEvent
MethodTable: 00123128
EEClass: 00121804
Size: 12(0xc) bytes
 (C:\ADNDBin\05Finalize.exe)
Fields:
      MT    Field   Offset                 Type VT     Attr    Value Name
791016bc  4000001        4        System.IntPtr  1 instance      1f0 nativeHandle
```

第一个对象是 SafePEFileHandle 类型，而第二个对象为 NativeEvent 类型，这正是我们需要

注意的对象。如果恢复程序的执行，那么终结线程会执行 NativeEvent 类的 Finalize 方法。当终结操作完成之后，F-Reachable 队列中的对象会发生怎样的变化？这些对象将从 F-Reachable 队列中移出，此时它们被认为是不存在根对象引用，并且在下一次垃圾收集过程中将被回收。

到这里也就结束了我们对终结操作的讨论。你可以看到，当使用终结类型时，在幕后将执行大量的操作。CLR 不仅需要额外的数据结构（例如终结队列或 F-Reachable 队列），而且还需要一个专门的线程来为每个被收集的对象运行 Finalize 方法。另外，带有 Finalize 的对象无法仅通过一次垃圾收集操作就被回收，而是需要两次，这意味着，带有 Finalize 方法的对象在它们被真正销毁之前，通常都会被提升到第 1 代，从而使它成为一种开销较高的对象。

5.2.4 回收 GC 内存

到目前为止，我们已经对 GC 进行了较为详细的讨论。现在，我们清楚地知道当一个对象被认为是垃圾对象时 GC 会采取的一些措施。还有一个问题没有被讨论：当对象被作为垃圾收集时，GC 将如何处理对象占据的内存？这块内存是否会被放入到某个空闲链表，然后在下次分配请求中被重用？这块内存是否被释放？在托管堆中，内存碎片是否会造成问题？答案都是肯定的。如果在第 0 代和第 1 代中执行的垃圾收集操作使得在托管堆上出现了内存空间缝隙，那么垃圾收集器将对所有的活跃对象执行紧缩（compact）操作，这样它们的内存位置可以彼此相邻，并将托管堆上的所有空闲块都合并成一块更大的空闲内存，这块内存就位于最后一个活跃对象之后（从当前分配指针开始）。在图 5-8 中给出了紧缩与合并操作的示例。

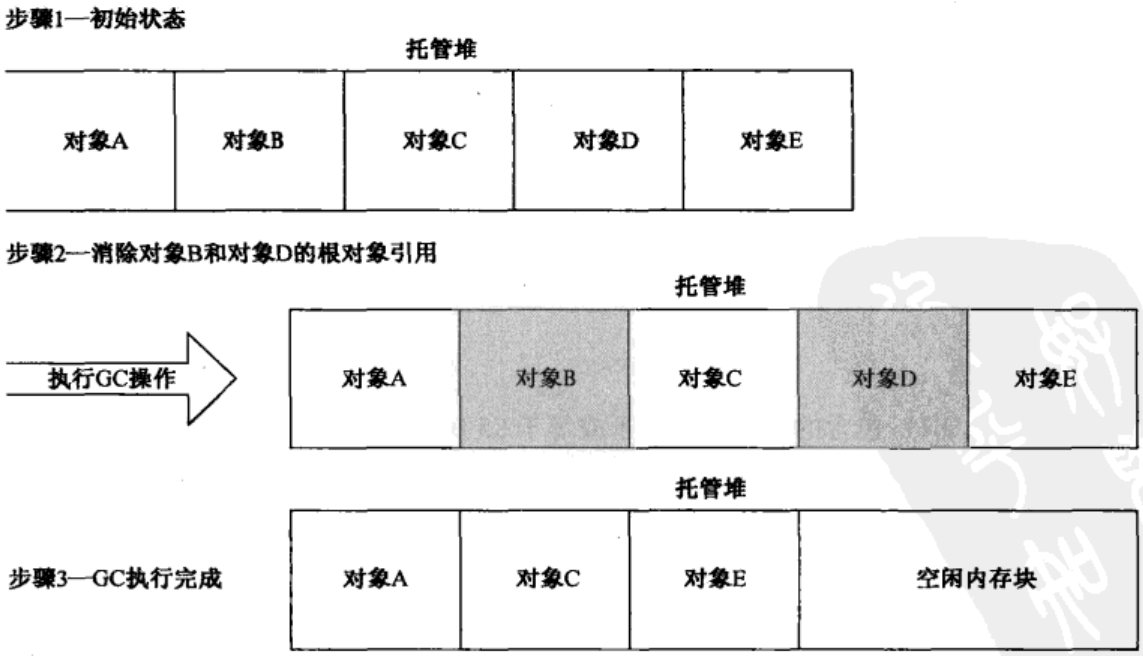


图 5-8 垃圾收集的紧缩与合并操作

在图5-8中，托管堆的初始状态包含了5个存在根对象引用的对象（从A到E）。在执行过程的某个时刻，对象B和D的根对象引用会被消除，因此这两个对象可以在下一次垃圾收集过程中被回收。当垃圾收集操作执行时，会回收对象B和D占据的内存，这将导致在托管堆上出现缝隙。为了消除这些缝隙，垃圾收集器把剩下的活跃对象（对象A，C和E）进行紧缩，并将多个空闲的内存块（保存对象B和D）合并成一个大的空闲块。最后，根据对象紧缩与合并的结果来更新当前的内存分配指针。在这个内存段中包含了第0代和第1代的对象（并且也是第2代的一部分），但第2代却可以包含多个托管内存段。随着越来越多的对象进入到第2代，增长第2代内存空间的需求也同样增长。CLR堆管理器增长第2代内存空间的方式是分配更多的内存段。当第2代中的对象被收集时，CLR堆管理器将回收这些内存段，并且当不再需要某个内存段时彻底释放它。在特定的情况和分配模式下，第2代将频繁地增长和收缩，从而大量地调用分配和释放虚拟内存的API（VirtualAlloc和VirtualFree等API）。这种方法存在两个缺陷：函数调用的开销很高（因为需要切换到内核态）以及可能在虚拟内存地址空间产生碎片。这样，CLR2.0引入了一种虚拟内存累积（VM Hoarding）机制，这种机制本质上并不会释放内存段，而是在一个单独的列表中记录它们，当需要更多的内存时就会使用这个列表。要使用虚拟内存累积这个功能，CLR宿主本身必须明确指出它需要启用这个功能。

完全垃圾收集与部分垃圾收集

如果由于达到了所有3个代的阈值而使得所有代中的对象都被收集，那么这种垃圾收集操作称之为完全垃圾收集（full garbage collection）。相反，仅在第0代或者仅在第0代和第1代中进行垃圾收集就称之为部分垃圾收集。由于执行紧缩操作的开销与对象的大小成正比（对象越大，那么紧缩操作的开销就越高），因此垃圾收集器引入了另一种类型的堆，称之为大对象堆。那些可能破坏紧缩操作性能的大型对象会被放入到LOH中，我们接下来将进行讨论。

5.2.5 大对象堆

大对象堆（LOH）包含的对象通常大于或等于85 000个字节。将这种大小的对象单独放入一个堆的原因是：在垃圾收集的紧缩阶段，在对某个对象执行紧缩操作时的开销与对象的大小成正比。因此，我们没有将大对象放在标准的堆上，而是创建了LOH。LOH最好被视为第2代内存空间的扩展，并且对LOH中对象的收集操作只有在收集完第2代中的对象之后才会进行，这也就意味着对LOH中对象的收集操作只会在完全垃圾收集中进行。因为对大对象进行压缩操作的开销是非常高的，因此GC会避免在LOH上进行紧缩操作，取而代之是执行一个清扫（sweeping）操作，在这个操作中会维持一个空闲链表，用于跟踪LOH内存段中的

可用内存。在图 5-9 中给出了带有两个内存段的 LOH 示例。

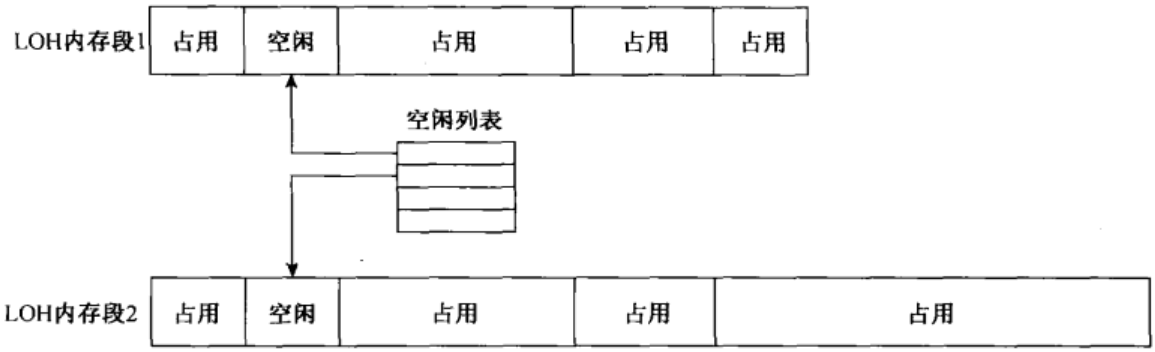


图 5-9 LOH 示例

请注意，虽然 LOH 并没有执行任何紧缩操作，但它将合并邻接的空闲内存块。也就是说，如果出现了两个空闲并且邻接的内存块，GC 就会将这两个内存块合并为一个更大的内存块，并把它添加到空闲链表中（同时消除了两个更小的内存块）。

要在调试器中找出 LOH 的当前状态，我们可以再次使用 eeheap-gc 命令，

```
0:004> !eeheap -gc
Number of GC Heaps: 1
generation 0 starts at 0x01fc6c18
generation 1 starts at 0x01fc100c
generation 2 starts at 0x01fc1000
ephemeral segment allocation context: none
 segment    begin allocated     size
00308030 790d8620  790f7d8c 0x0001f76c(128876)
01fc0000 01fc1000  01fc8c24 0x00007c24(31780)
Large object heap starts at 0x02fc1000
 segment    begin allocated     size
02fc0000 02fc1000  02fc3240 0x00002240(8768)
Total Size   0x295d0(169424)
------------------------------
GC Heap Size   0x295d0(169424)
```

在这个命令输出的 LOH 部分给出了 LOH 的起始地址以及每个内存段的信息，例如内存段的起始位置和结束位置，以及大小。从示例中我们可以看到 LOH 有一个内存段（0x02fc000），起始位置为 0x02fc1000，结束位置为 0x02fc3240，总大小为 0x00002240。最后一部分信息是 LOH 中所有内存段的总大小。与 LOH 相关的一个问题是，如何将 LOH 的内容转储出来。可以使用 DumpHeap 命令和两个相应的开关。第一个开关是 -min，它告诉 DumpHeap 命令只需要特定大小的对象。由于我们知道 LOH 对象都是大于或等于 85 000 个字节，因此可以使用以下命令：

```
0:004> !DumpHeap -min 85000
 Address       MT     Size
02c53250 7912dae8    100016
total 1 objects
Statistics:
      MT    Count    TotalSize Class Name
7912dae8        1       100016 System.Byte[]
```

在 LOH 中有一个大小为 100 016 的对象，可以通过查看地址来验证它是否位于 LOH。如果这个对象的地址位于 LOH 内存段地址中，那么它必定位于 LOH 上（除了空闲对象，它们既可以驻留在 SOH 中，也可以驻留在 LOH 中）。

下一个选项是为 DumpHeap 命令指定一个起始地址。如果指定了 LOH 的起始地址，那么可以利用这个命令将 LOH 上的对象转储出来。这个开关就是-startAtLowerBound，参数是一个地址值。在与前面相同的 LOH 上使用以下命令：

```
0:004> !DumpHeap -startAtLowerBound 02c51000
 Address       MT     Size
02c51000 002a6360       16 Free
02c51010 7912d8f8     4096
02c52010 002a6360       16 Free
02c52020 7912d8f8     4096
02c53020 002a6360       16 Free
02c53030 7912d8f8      528
02c53240 002a6360       16 Free
02c53250 7912dae8   100016
02c6b900 002a6360       16 Free
total 9 objects
Statistics:
      MT    Count    TotalSize   Class Name
002a6360        5           80      Free
7912d8f8        3         8720   System.Object[]
7912dae8        1       100016   System.Byte[]
Total 9 objects
```

我们再次看到了大小为 100 016 的对象，但更值得注意的是，我们看到一些小于 85 000 字节的对象也位于 LOH 上。这些是什么对象，为什么它们也位于 LOH 上？答案在于，这些对象是由 CLR 堆管理器放在 LOH 上的，有着特殊的用途。通常来说，你可以看到一些由 GC 专门使用并且小于 85 000 字节的对象。

我们来看一个示例程序，在这个程序中分配了一个 10 000 字节的大对象（参见清单 5-5）。然后，我们通过调试器来观察能否在 LOH 上找到这个对象，并且查看当这个对象被收集时所发生的情况。

清单 5-5　说明 LOH 的示例程序

```
using System;
using System.Text;
```

```
using System.Runtime.InteropServices;

namespace Advanced.NET.Debugging.Chapter5
{
     class LOH
    {
        static void Main(string[] args)
        {
            LOH l = new LOH();
            l.Run();
        }

        public void Run()
        {
            byte[] b = null;
            Console.WriteLine("Press any key to allocate on LOH");
            Console.ReadKey();

            b = new byte[100000];

            Console.WriteLine("Press any key to GC");
            Console.ReadKey();

            b = null;
            GC.Collect();

            Console.WriteLine("Press any key to exit");
            Console.ReadKey();
        }

    }
}
```

清单 5-5 中的源代码和二进制文件位于以下文件夹中：

- 源代码文件：C:\ADND\Chapter5\LOH
- 二进制文件：C:\ADNDBin\05LOH.exe

在调试器下运行这个程序，并在出现提示信息“Press any key to allocate on LOH”时中断程序的执行。此时还没有执行大内存分配，但可以看看当前在 LOH 堆上已经存在了哪些对象。

```
0:004> !eeheap -gc
Number of GC Heaps: 1
generation 0 starts at 0x01f01018
generation 1 starts at 0x01f0100c
generation 2 starts at 0x01f01000
ephemeral segment allocation context: none
 segment    begin allocated     size
```

```
004a8008 790d8620  790f7d8c 0x0001f76c(128876)
01f00000 01f01000  01f5c334 0x0005b334(373556)
Large object heap starts at 0x02f01000
 segment    begin allocated     size
02f00000 02f01000  02f03250 0x00002250(8784)
Total Size   0x7ccf0(511216)
------------------------------
GC Heap Size   0x7ccf0(511216)
0:004> !dumpheap -startatlowerbound 02f01000
 Address       MT     Size
02f01000 00496360       16 Free
02f01010 7912d8f8     4096
02f02010 00496360       16 Free
02f02020 7912d8f8     4096
02f03020 00496360       16 Free
02f03030 7912d8f8      528
02f03240 00496360       16 Free
total 7 objects
Statistics:
      MT    Count    TotalSize Class Name
00496360        4           64      Free
7912d8f8        3         8720 System.Object[]
Total 7 objects
```

首先通过 eeheap 命令找出 LOH 的起始位置。在这里，起始位置是 0x02f01000。然后，我们将该起始位置传递给 dumpheap 命令，并通过 -startlowerbound 开关来输出 LOH 上的所有对象。在输出信息中，我们可以看到在 LOH 上存在一组奇怪的对象，这些对象的大小都小于 85 000 字节。除此之外，没有其他的对象。接下来，恢复程序的执行并在出现“Press any key to GC”时再次手动中断执行。像前面一样执行 dumpheap 命令，并观察能否找出我们的 100KB 内存分配：

```
0:003> !dumpheap -startatlowerbound 02f01000
 Address       MT     Size
02f01000 00496360       16 Free
02f01010 7912d8f8     4096
02f02010 00496360       16 Free
02f02020 7912d8f8     4096
02f03020 00496360       16 Free
02f03030 7912d8f8      528
02f03240 00496360       16 Free
02f03250 7912dae8   100016
02f1b900 00496360       16 Free
total 9 objects
Statistics:
      MT    Count    TotalSize Class Name
00496360        5           80      Free
7912d8f8        3         8720 System.Object[]
7912dae8        1       100016 System.Byte[]
Total 9 objects
```

我们可以看到，分配的内存位于 LOH 的 0x02f03250 地址处。接下来，恢复程序的执行，直到出现提示信息“Press any key to exit”。此时将执行一次垃圾收集操作，因此，再次使用 dumpheap 命令来观察 LOH 的内容：

```
0:003> !dumpheap -startatlowerbound 02f01000
 Address       MT     Size
02f01000 00496360       16 Free
02f01010 7912d8f8     4096
02f02010 00496360       16 Free
02f02020 7912d8f8     4096
02f03020 00496360       16 Free
02f03030 7912d8f8      528
total 6 objects
Statistics:
      MT    Count    TotalSize Class Name
00496360        3           48      Free
7912d8f8        3         8720 System.Object[]
```

这时，我们可以看到对象从 LOH 中被移除，并且在执行完收集过程后出现了空闲内存块。

5.2.6 固定

我们在 5.2.4 一节中看到，垃圾收集器采用了一种紧缩技术来减少 GC 堆中的碎片。当发生一次紧缩操作时，对象可能在堆中移动，从而把它们放在一起以减少内存空间中的缝隙。在对象的移动过程中，由于对象的地址发生了变化，因此对这个对象的所有引用都会被更新。假设对这个对象的所有引用都包含在 CLR 中，那么这种移动不会带来任何问题，但对于 .NET 程序来说，经常需要通过互用性服务（例如平台调用或者 COM 互用性）在 CLR 范围之外工作，这就会产生问题。如果某个托管对象的引用被传递给一个底层的非托管 API，那么当非托管 API 正在读取或写入内存的同时移动了对象，就会导致严重的问题，因为 CLR 显然无法通知非托管 API 关于对象地址变动的情况。在图 5-10 中说明了这个问题。

从图 5-10 的流程图中，我们可以看到在托管堆中起初包含了 5 个对象，从位于地址 0x02000000 处的对象 A 开始。在某个特定时刻，通过平台调用来调用一个异步的非托管 API，同时将对象 C 的地址（0x02000090）传递给这个 API。在调用这个异步的非托管 API 时，发生了一次垃圾收集操作，使得对象 A 和对象 B 被收集。此时在托管堆中出现了两个空闲对象造成的缝隙，从而垃圾收集器会通过紧缩托管堆来解决这个问题，因此将对象 C 移动到了地址 0x02000000。此外，它还合并了这两个内存块，并将它们放在堆的末尾。在完成了垃圾收集操作后，之前进行的异步 API 调用决定写入到最初传递给它的地址（0x02000090），在这个地址上最初保存的是对象 C。你可以看到，在这个异步 API 写入到这个地址时，会发生堆破坏问题，因为被写入的内存已经不再由对象 C 占用。

由于调用非托管代码是一项很常见的任务，因此需要设计一种安全的解决方案，以确保

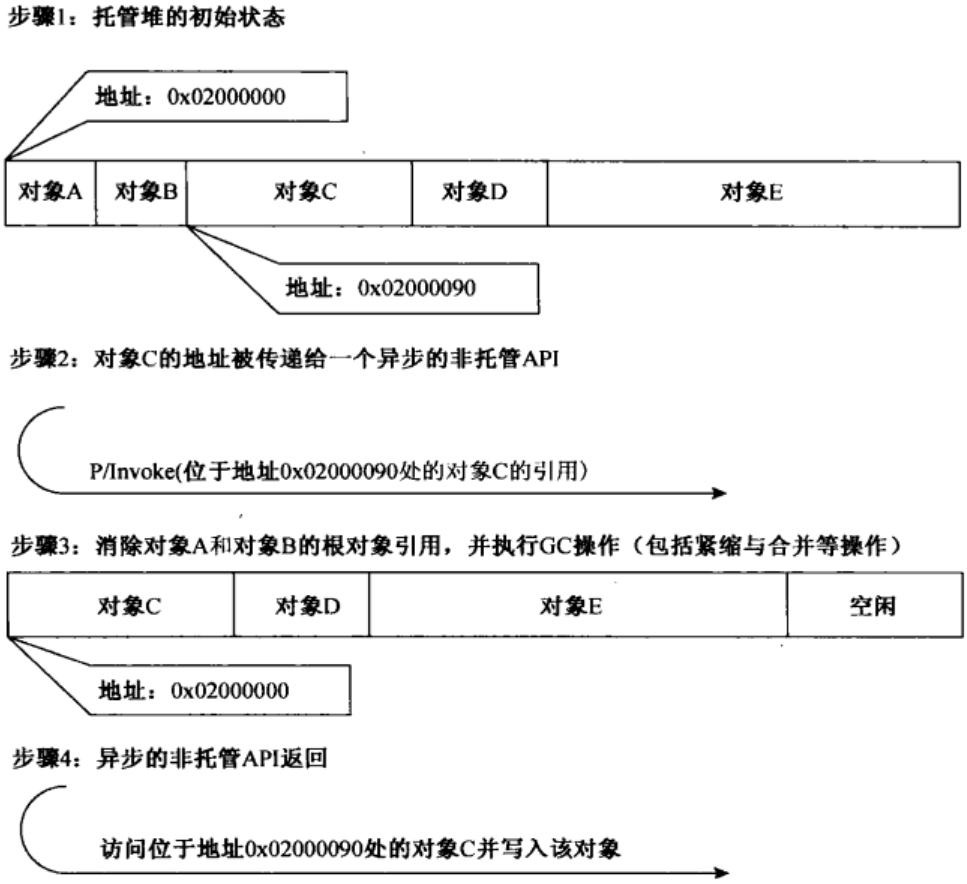


图 5-10　互用性服务以及 GC 紧缩带来的问题

在执行紧缩操作时仍能实现安全地调用。这种解决方案称为固定，指将托管堆上某些特定的对象固定住。当一个对象被固定住时，垃圾收集器不会移动这个对象，直到解除这个固定对象为止。如果图 5-10 中的对象 C 在调用异步的非托管 API 之前被固定，那么就不会对托管堆造成破坏，因为垃圾收集器在紧缩阶段不会移动对象 C。

在程序中固定对象的方法，参见清单 5-6 中给出的程序的源代码。

清单 5-6　在程序中固定对象的地址

```
using System;
using System.Text;
using System.Runtime.InteropServices;

namespace Advanced.NET.Debugging.Chapter5
{
    class Pinning
    {
```

```
        static void Main(string[] args)
        {
            Pinning p = new Pinning();
            p.Run();
        }
        public void Run()
        {
            SByte[] b1 = null;
            SByte[] b2 = null;
            SByte[] b3 = null;
            Console.WriteLine("Press any key to alloc");
            Console.ReadKey();

            b1 = new SByte[100];
            b2 = new SByte[200];
            b3 = new SByte[300];

            GCHandle h1 = GCHandle.Alloc(b1, GCHandleType.Pinned);
            GCHandle h2 = GCHandle.Alloc(b2, GCHandleType.Pinned);
            GCHandle h3 = GCHandle.Alloc(b3, GCHandleType.Pinned);

            Console.WriteLine("Press any key to GC");
            Console.ReadKey();

            GC.Collect();

            Console.WriteLine("Press any key to exit");
            Console.ReadKey();

            h1.Free(); h2.Free(); h3.Free();
        }

    }
}
```

清单 5-6 中的源代码和二进制文件位于以下文件夹中：

- 源代码文件：C:\ADND\Chapter5\Pinning
- 二进制文件：C:\ADNDBin\05Pinning.exe

在清单 5-6 中给出的示例程序说明了如何使用 GCHandle 类型来固定对象。在 Run 方法中声明了三个 SByte 类型的数组，并且为每个需要被固定的对象创建了 GCHandle。然后，这个程序执行一次垃圾收集操作并退出。在调试器下运行这个程序，查看能否找出所分配的内存以及它是如何被固定的。恢复程序的执行，直到看到提示信息“Press any key to GC”。此时我们手动中断执行，并使用命令 GCHandles。GCHandles 命令能显示进程中的所有句柄。

```
0:004> !GCHandles
GC Handle Statistics:
Strong Handles: 15
Pinned Handles: 7
Async Pinned Handles: 0
Ref Count Handles: 0
Weak Long Handles: 0
Weak Short Handles: 1
Other Handles: 0
Statistics:
      MT    Count    TotalSize Class Name
790fd0f0        1           12 System.Object
790feba4        1           28 System.SharedStatics
790fcc48        2           48 System.Reflection.Assembly
790fe17c        1           72 System.ExecutionEngineException
790fe0e0        1           72 System.StackOverflowException
790fe044        1           72 System.OutOfMemoryException
790fed00        1          100 System.AppDomain
790fe704        2          112 System.Threading.Thread
79100a18        4          144 System.Security.PermissionSet
790fe284        2          144 System.Threading.ThreadAbortException
7912ee44        3          636 System.SByte[]
7912d8f8        4         8736 System.Object[]
Total 23 objects
```

GCHanldes 命令将遍历句柄表，查看所有不同类型的句柄（Strong、Weak、Pinned 等），并且显示汇总信息以及统计信息，其中包含了所有已找到的类型的详细信息。在前面的输出中，我们可以看到有 15 个 Strong 类型的句柄，7 个 Pinned 类型的句柄以及 1 个 Weak Short 类型的句柄。此外，在 Statistics 部分还可以看到我们分配的且被固定的三个 SByte 数组。GCHandles 命令给出了进程中句柄行为的一些基本信息，但如果要获得进一步的信息，例如在 Statistics 部分中列出的各种类型的句柄信息，就需要使用另一个命令 objsize。objsize 命令的功能之一就是输出对象的大小。如果没有指定参数，那么该命令将扫描进程中所有被引用的对象，并输出对象的大小以及其他有用的信息：

```
0:004> !objsize
Scan Thread 0 OSTHread 2558
ESP:2fed54:  sizeof(01d9599c)  =          20 (        0x14) bytes
 (Microsoft.Win32.SafeHandles.SafeFileHandle)
ESP:2fee18: sizeof(01d96d9c) =          312 (       0x138) bytes (System.SByte[])
ESP:2fee20: sizeof(01d96c58) =          112 (        0x70) bytes (System.SByte[])
ESP:2fee24: sizeof(01d96cc8) =          212 (        0xd4) bytes (System.SByte[])
ESP:2fee30:  sizeof(01d958b4)  =           12 (         0xc) bytes
 (Advanced.NET.Debugging.Chapter5.Pinning)
...
...
...
Scan Thread 2 OSTHread 2c80
DOMAIN(004DFD10):HANDLE(Strong):1c119c: sizeof(01d958a4) =
           16 (        0x10) bytes (System.Object[])
```

```
...
...
...
DOMAIN(004DFD10):HANDLE(WeakSh):1c12fc: sizeof(01d91de8) =
          56 (        0x38) bytes (System.Threading.Thread)
DOMAIN(004DFD10):HANDLE(Pinned):1c13e4: sizeof(01d96d9c) =
          312 (       0x138) bytes (System.SByte[])
DOMAIN(004DFD10):HANDLE(Pinned):1c13e8: sizeof(01d96cc8) =
          212 (        0xd4) bytes (System.SByte[])
DOMAIN(004DFD10):HANDLE(Pinned):1c13ec: sizeof(01d96c58) =
         112 (        0x70) bytes (System.SByte[])
DOMAIN(004DFD10):HANDLE(Pinned):1c13f0: sizeof(02d93030) =
         708 (       0x2c4) bytes (System.Object[])
DOMAIN(004DFD10):HANDLE(Pinned):1c13f4: sizeof(02d92020) =
        4276 (      0x10b4) bytes (System.Object[])
DOMAIN(004DFD10):HANDLE(Pinned):1c13f8: sizeof(01d9118c) =
          12 (         0xc) bytes (System.Object)
DOMAIN(004DFD10):HANDLE(Pinned):1c13fc: sizeof(02d91010) =
       19332 (      0x4b84) bytes (System.Object[])
```

虽然这里的输出信息经过了一些简化，但显然可以看出我们的 SByte 类型已经被固定，因为可以看到 HANDLE（Pinned）。

虽然通过固定对象地址解决了在非托管代码调用期间的对象移动问题，但也给垃圾收集器带来了一个问题，即内存碎片（fragmentation），这正是紧缩操作需要解决的问题之一。如果在托管堆上存在大量的固定对象，那么就可能出现没有足够大而连续的空闲空间的情况。在图 5-11 中给出了由于大量固定对象而导致的托管堆碎片情形。

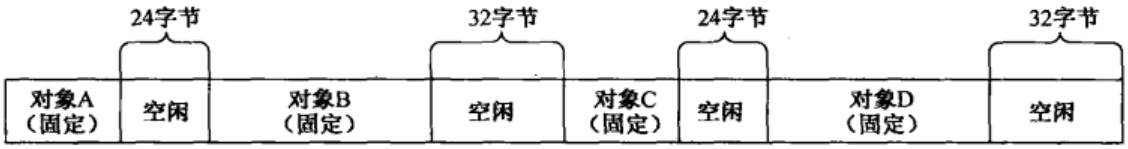


图 5-11　托管堆中的碎片

在图 5-11 的内存布局中，有数个空闲的小内存块与活跃的对象（对象 A 到对象 D）交错存放着。如果执行一次垃圾收集操作，那么托管堆的内存布局将保持不变。出现这种情况的原因很简单：由于所有活跃的对象都被固定住，因此无法移动，而垃圾收集器也就无法执行紧缩操作。由于这些空闲的内存块都不是邻接的，因此同样无法执行合并操作。这样，即使有多个空闲的内存块，只要内存分配请求超过 32 字节，也仍然会失败。在本章的后面，我们将看到一个实际的托管堆碎片问题。

LOH 中的碎片情况怎么样

我们曾经讨论了 LOH 以及它采用的清扫方式而不是紧缩方式。这从本质上意味着 LOH

上的对象永远都不会移动。那么可以忽略 LOH 上的固定对象？答案是不可以。如果没有固定住 LOH 上的对象，那么相当于一种非常危险的假设，即 LOH 绝不会使用紧缩操作。在不同的 CLR 版本之间，这种假设可能会发生变化。因此，在 LOH 上的对象必须被固定住，以免在不同版本的 CLR 实现之间发生变化。

5.2.7 垃圾收集模式

我们要讨论的最后一个主题就是垃圾收集器的运行模式。共有 3 种主要的运行模式：

- 非并发工作站
- 并发工作站
- 服务器

我们已经讨论了服务器模式和工作站模式之间的差异，服务器模式基本可归结为：为每个处理器创建一个堆和一个 GC 线程。垃圾收集操作的所有相关行为都是通过处理器上专门的 GC 线程来执行的。我们还没有讨论并发工作站和非并发工作站这两种模式。在非并发工作站模式中，垃圾收集器在整个垃圾收集过程中将挂起所有托管线程。只有当垃圾收集操作完成之后，才会恢复进程中所有托管线程的执行。在不需要快速响应的情况下，这种方式工作得很好，但在一些要求快速响应时间的情况下，例如 GUI 程序，这种方式并不适用。因此就引入了并发工作站模式，在垃圾收集操作中，托管线程在整个垃圾收集操作期间并非一直挂起，而是会定期醒来，并且在重新休眠之前执行一些必要的工作。这增加了程序的响应灵敏度，但却会使垃圾收集操作略微变慢。

5.3 调试托管堆的破坏问题

堆破坏是一种违背了堆完整性的错误，它会导致在程序中出现奇怪的行为。堆破坏问题的症状是多种多样的，包括一些随机的奇怪行为以及使程序崩溃。例如，在程序中有一个对象，该对象的状态控制着以何种频率从队列中取出工作项。如果某个线程由于破坏了对象的内存而在无意中改变了这个频率值，那么取出工作项的频率可能远远高于系统能够处理的频率，或者相反的是根本不取出任何工作项，那么就会导致处理过程延迟。在类似的情况下，要找出这种问题的根源是很困难的，因为在托管堆被破坏后展示出的行为是各不相同的。事实上，在处理堆破坏问题时，最好的情况就是出现程序崩溃，并且发生崩溃的位置与发生破坏的位置要尽可能地接近，这就不需要通过大量的回溯来找出最初发生破坏的位置。

由于堆破坏会表现出各种不同的症状，因此它也是最难调试的问题类型之一。首先，是什么导致了堆破坏问题？通常来说，堆破坏的原因可能有多种，但一个非常常见的原因是，没有正确地管理程序中的内存。例如，在释放内存后再次使用这块内存、悬空指针、缓冲区溢出等，都可能会导致堆破坏。然而，好在 CLR 通过高效地管理程序内存消除了许多问题。

例如，在一块内存被释放后，将无法再重用它，因为对象在仍然存在根对象引用时无法被收集，缓冲区溢出也会被识别出来并作为一个异常，同样也不太可能出现悬空指针。虽然 CLR 有效地消除了大量的堆破坏问题，但它需要代码运行在托管执行的环境中才能发挥作用。通常，在托管代码程序中会调用非托管的代码，并且将数据传递给非托管 API。当代码切换到非托管环境时，驻留在托管堆上并被传递给非托管代码的数据将不再受到 CLR 保护，如果在切换之前没有小心地管理，那么可能会导致各种各样的问题。例如，无法捕捉到缓冲区溢出错误，以及 GC 的紧缩操作使指针失效等。托管代码与非托管代码的交互是在托管环境中导致堆破坏的最主要原因之一。

在没有使用非托管代码的情况下是否也可能出现托管堆破坏问题

在没有使用任何非托管代码的情况下也可能出现托管堆破坏的情况，但这种情况极少发生，并且通常说明 CLR 本身存在一个错误。

在本章的这部分内容中，我们将观察一个发生堆破坏的示例程序。清单 5-7 给出了程序的源代码。

清单 5-7　出现堆破坏问题的示例程序

```
using System;
using System.Text;
using System.Runtime.InteropServices;

namespace Advanced.NET.Debugging.Chapter5
{
    class Heap
    {
        static void Main(string[] args)
        {
            Heap h = new Heap();
            h.Run();
        }
        public void Run()
        {
         byte[] b = new byte[50];
         for (int i = 0; i < 50; i++)
             b[i] = 15;


         Console.WriteLine("Press any key to invoke native method");
         Console.ReadKey();

         InitBuffer(b, 50);
```

```
            Console.WriteLine("Press any key to exit");
            Console.ReadKey();
        }

        [DllImport("05Native.dll")]
        static extern void InitBuffer(byte[] buffer, int size);

    }
}
```

清单 5-7 中的源代码和二进制文件位于以下文件夹中：

- 源代码文件：C:\ADND\Chapter5\Heap
- 二进制文件：C:\ADNDBin\05Heap. exe and C:\ADNDBin\05Native. dll

注意，为了更好地说明调试会话，在这里没有给出非托管源代码。

在清单 5-6 中的程序分配了一个字节数组（50 个元素），然后调用一个非托管 API，并将这个字节数组及其数组大小传递给这个 API 以初始化内存。如果在调试器下运行这个程序，那么会很快看到发生内存访问违例：

```
...
...
...
Press any key to invoke native method
 ModLoad: 71190000 711ab000   C:\ADNDBin\05Native.dll
ModLoad: 63f70000 64093000   C:\Windows\WinSxS\x86_microsoft.vc90.debugcrt
_1fc8b3b9a1e18e3b_9.0.21022.8_none_96748342450f6aa2\MSVCR90D.dll
(1b00.26e4): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=77767574 ebx=00000001 ecx=01c659a4 edx=01c66ad8 esi=01c66868 edi=00000017
eip=7936ab16 esp=0031edac ebp=00000017 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010206
*** WARNING: Unable to verify checksum for
C:\Windows\assembly\NativeImages_v2.0.50727_32\
mscorlib\5b3e3b0551bcaa722c27dbb089c431e4\mscorlib.ni.dll
mscorlib_ni+0x2aab16:
7936ab16 ff90a4000000    call    dword ptr [eax+0A4h] ds:0023:77767618=????????
0:000> !ClrStack
OS Thread Id: 0x26e4 (0)
ESP       EIP
0031edac 7936ab16 System.IO.StreamWriter.Flush(Boolean, Boolean)
0031edcc 7936b287 System.IO.StreamWriter.Write(Char[], Int32, Int32)
0031edec 7936b121 System.IO.TextWriter.WriteLine(System.String)
0031ee04 7936b036 System.IO.TextWriter+SyncTextWriter.WriteLine(System.String)
0031ee10 793e9d86 System.Console.WriteLine(System.String)
0031ee1c 00810171 Advanced.NET.Debugging.Chapter5.Heap.Run()
0031ee48 008100a7 Advanced.NET.Debugging.Chapter5.Heap.Main(System.String[])
0031f068 79e7c74b [GCFrame: 0031f068]
```

在这个内存访问违例中，需要注意的是线程的栈回溯。看上去，这个内存访问违例似乎

是发生在第二次调用 Console. WriteLine 方法时（刚好在调用了非托管的 InitBuffer 之后）。虽然我们假设在程序中发生了一个堆破坏问题，但为什么会在代码某个看似随机的位置上发生故障？要记住，当发生堆破坏时很少会立即中断程序执行，而是在后续执行流程中某个随机的位置上发生中断。这种现象肯定是随机的，因为我们没有预料到在调用 Console. WriteLine 时会出现一个内存访问违例。在知道了发生的是一个内存访问违例，并且这个问题发生在执行流中某个奇怪位置上，现在就可以推断可能发生了堆破坏问题。此时最大的问题是，如何验证这个假设？记得之前堆破坏的定义是：违背了堆的完整性。如果我们遍历堆上的所有对象，并对每个对象的有效性进行验证，那么可以判断是否违背了完整性。虽然我们可以手动遍历整个托管堆，但却是一个非常耗时的过程。SOS 中的 VerifyHeap 命令可以将这个过程自动化。VerifyHeap 命令会遍历整个托管堆，验证每个对象，并且报告验证过程的结果。如果在调试会话中运行这个命令，可以看到以下输出：

```
0:000> !VerifyHeap
-verify will only produce output if there are errors in the heap
object 01c65968: does not have valid MT
curr_object : 01c65968
Last good object: 01c65928
---------------
object 02c61010: bad member 01c65968 at 02c61084
object 02c61010: bad member 01c65984 at 02c6109c
object 02c61010: bad member 01c659fc at 02c61444
object 02c61010: bad member 01c659e4 at 02c61448
object 02c61010: bad member 01c659f0 at 02c6144c
object 02c61010: bad member 01c659c8 at 02c6158c
curr_object : 02c61010
Last good object: 02c61000
---------------
```

在前面的输出中，我们可以看到在托管堆中存在许多问题。具体来说，遇到的第一个错误是，位于地址 0x01c65968 处的对象的方法表（Method Table，MT）是无效的。我们可以很容易地验证这一点，即通过命令 dd 将这个地址的内容转储出来：

```
0:000> dd 01c65968 l1
01c65968  3b3a3938
0:000> dd 3b3a3938 l1
3b3a3938  ????????
```

位于地址 0x01c65968 处的对象的方法表似乎是 0x3b3a3938，这进一步说明了是一个无效地址。此时，我们知道堆被破坏了，并且这个堆的第一个对象是位于地址 0x01c65968 处，但我们并不知道这个堆是如何被破坏的。在这种情况下，一种有用的技术就是分析被破坏内存区域周围的对象。例如，前一个对象看上去像什么？在 VerifyHeap 的输出中，最后一个有效对象位于地址 0x01c65928 处。如果将它的内容转储出来，可以看到以下输出：

```
0:000> !do 01c65928
Name: System.Byte[]
MethodTable: 7912dae8
EEClass: 7912dba0
Size: 62(0x3e) bytes
Array: Rank 1, Number of elements 50, Type Byte
Element Type: System.Byte
Fields:
None
0:000> !objsize 01c65928
sizeof(01c65928) =          64 (        0x40) bytes (System.Byte[])
```

这个对象看上去是一个包含 50 个元素的字节数组，非常类似于我们在程序中创建的字节数组。而且，由于命令 do 可以显示这个对象的详细信息，因此这个对象的元数据似乎是结构完整的。请注意，在上面使用了命令 objsize 来获得对象的大小（包括对象的成员）。还需要注意的信息就是这个数组本身的内容。我们可以使用命令 dd 来显示整个对象的内容：

```
0:000> dd 01c65928
01c65928  7912dae8 00000032 03020100 07060504
01c65938  0b0a0908 0f0e0d0c 13121110 17161514
01c65948  1b1a1918 1f1e1d1c 23222120 27262524
01c65958  2b2a2928 2f2e2d2c 33323130 37363534
01c65968  3b3a3938 3f3e3d3c 43424140 47464544
01c65978  4b4a4948 4f4e4d4c 53525150 57565554
01c65988  5b5a5958 5f5e5d5c 63626160 67666564
01c65998  6b6a6968 6f6e6d6c 73727170 77767574
```

从输出信息中可以看到，这个对象的大小为 64 个字节，首先是一个方法表，表示数组的类型，接下来是数组中元素的数量，随后是数组本身的内容。下一个对象的起始地址是 0x01c65928（（前一个对象的起始地址）+0x40（对象的总大小））。观察最后一个有效对象（0x01c65928）的内容，可以看到在这个数组中包含的是递增的整数值。而且，当到达最后一个有效对象的末尾时，我们仍会看到一系列的递增整数值，覆盖了堆上下一个对象（0x01c65968）的空间。这个观察结果对于了解所发生的情况是一条非常重要的线索。如果地址 0x01c65928 的对象没有被正确地写入，并且在写入时越过了对象边界的末尾，那么将破坏堆中下一个对象。在图 5-12 中说明了这个问题。

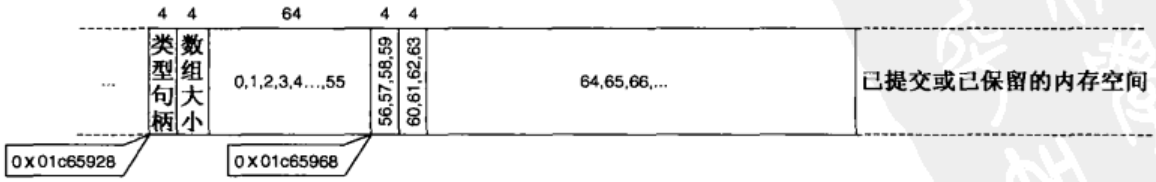


图 5-12　托管堆中的破坏

此时，我们很清楚地理解了调试器中给出的数据。通过对程序中处理字节数组的代码进

行分析，我们可以看到，当把字节数组传递给 InitBuffer 时，这个函数没有考虑对象的边界，并且在写入时越过了对象的末尾，因此导致堆上的紧接着该对象的下一个对象被破坏（在 VerifyHeap 命令的输出中）。VerifyHeap 命令显示的另一部分信息，如下所示：

```
object 02c61010: bad member 01c65968 at 02c61084
object 02c61010: bad member 01c65984 at 02c6109c
object 02c61010: bad member 01c659fc at 02c61444
object 02c61010: bad member 01c659e4 at 02c61448
object 02c61010: bad member 01c659f0 at 02c6144c
object 02c61010: bad member 01c659c8 at 02c6158c
curr_object : 02c61010
Last good object: 02c61000
```

VerifyHeap 告诉我们，在地址 0x02c61010 处有一个对象，包含了一个成员，这个成员引用了起始地址位于 0x01c65968 处的被破坏对象。事实上，在输出中有多行信息都表明了多个位于不同地址（0x01c65968、0x01c65984、0x01c659fc 等）的已破坏对象引用了同一个对象。VerifyHeap 不仅能告诉我们哪个对象被破坏，而且还会显示所有引用这个被破坏对象的对象，无论它们位于哪个堆上。

VerifyHeap 与 GC 的相互干扰

VerifyHeap 命令可以有效地分析托管堆破坏问题，它将遍历堆并且报告可能由于堆破坏而造成的不一致性。然而，有时候 VerifyHeap 也会产生一些并非由于堆破坏而导致的问题。例如，当 CLR 处于垃圾收集的过程中时。在垃圾收集过程中，GC 可能会对堆执行紧缩操作，这就涉及移动对象。例如，如果当前正在执行一个移动动作，那么 VerifyHeap 命令可能会失败，或者由于堆被重组而给出了不准确的信息。

在垃圾收集器中包含了许多内置的辅助诊断功能，其中之一就是在垃圾收集操作前后执行堆验证。要启用这个诊断功能，需要将环境变量 COMPLUS_ HeapVerify 设置为 1。

清单 5-7 中的示例程序是通过互用性服务来调用非托管代码。根据非托管代码破坏堆的方式，以及垃圾收集操作的执行时机，可能在非托管代码已经破坏之后的很长一段时间里都不会出现任何堆破坏现象，从而使得回溯问题的根源变得非常困难。为了简化这种问题分析过程，增加了一个名为 gcUnmanagedToManaged 的 MDA。这个 MDA 的作用是减少托管代码破坏与下一次执行 GC 操作的时间间隔。它的工作原理是，当互用性调用从非托管代码切换回托管代码时，强制执行一次垃圾收集操作，因此就可以更早地将进程中的问题暴露出来。启用这个 MDA（请参见第 1 章关于如何启用这个 MDA 的内容），并在调试器下重新运行示例程序来查看能否尽早地捕获堆破坏问题：

```
...
...
...
Press any key to invoke native method
 ModLoad: 71190000 711ab000   C:\ADNDBin\05Native.dll
ModLoad: 63f70000 64093000   C:\Windows\WinSxS\x86_microsoft.vc90.
debugcrt_1fc8b3b9a1e18e3b_9.0.21022.8_none_96748342450f6aa2\MSVCR90D.dll
(19d8.258c): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=3b3a3938 ebx=02d81010 ecx=00960184 edx=01d8598c esi=00020000 edi=00001000
eip=79f66846 esp=0025ec54 ebp=0025ec74 iopl=0  nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010202
mscorwks!WKS::gc_heap::mark_object_simple+0x16c:
79f66846 0fb708          movzx   ecx,word ptr [eax]      ds:0023:3b3a3938=????
0:000> k
ChildEBP RetAddr
0025ec74 79f66932 mscorwks!WKS::gc_heap::mark_object_simple+0x16c
0025ec88 79fbc552 mscorwks!WKS::GCHeap::Promote+0x8d
0025eca0 79fbc3c9 mscorwks!PinObject+0x10
0025ecc4 79fc37b9 mscorwks!ScanConsecutiveHandlesWithoutUserData+0x26
0025ece4 79fba942 mscorwks!BlockScanBlocksWithoutUserData+0x26
0025ed08 79fba917 mscorwks!SegmentScanByTypeMap+0x55
0025ed60 79fba807 mscorwks!TableScanHandles+0x65
0025edc8 79fbb9a2 mscorwks!HndScanHandlesForGC+0x10d
0025ee0c 79fbaaf8 mscorwks!Ref_TracePinningRoots+0x6c
0025ee30 79f669f6 mscorwks!CNameSpace::GcScanHandles+0x60
0025ee70 79f65d57 mscorwks!WKS::gc_heap::mark_phase+0xae
0025ee94 79f6614c mscorwks!WKS::gc_heap::gc1+0x62
0025eea8 79f65f5d mscorwks!WKS::gc_heap::garbage_collect+0x261
0025eed4 79f6dfa1 mscorwks!WKS::GCHeap::GarbageCollectGeneration+0x1a9
0025eee4 79f6df4b mscorwks!WKS::GCHeap::GarbageCollectTry+0x2d
0025ef04 7a0aea3d mscorwks!WKS::GCHeap::GarbageCollect+0x67
0025ef8c 7a12addd mscorwks!MdaGcUnmanagedToManaged::TriggerGC+0xa7
0025f020 79e7c74b mscorwks!FireMdaGcUnmanagedToManaged+0x3b
0025f030 79e7c6cc mscorwks!CallDescrWorker+0x33
0025f0b0 79e7c8e1 mscorwks!CallDescrWorkerWithHandler+0xa3
0:000> !ClrStack
OS Thread Id: 0x258c (0)
ESP       EIP
0025efdc 79f66846 [NDirectMethodFrameStandalone: 0025efdc]
 Advanced.NET.Debugging.Chapter5.Heap.InitBuffer(Byte[], Int32)

0025efec 00a80165 Advanced.NET.Debugging.Chapter5.Heap.Run()
0025f018 00a800a7 Advanced.NET.Debugging.Chapter5.Heap.Main(System.String[])
0025f240 79e7c74b [GCFrame: 0025f240]
```

我们可以看到，导致这个访问违例发生的非托管栈回溯与我们之前看到的栈回溯大不相同。现在，看上去似乎是在垃圾收集的过程中触发了这个问题。那么这个垃圾收集操作处于托管代码流中的什么位置？通过观察托管代码栈回溯，可以看到是在调用非托管的 InitBuffer

时出现了访问违例。

如果你怀疑在非托管的 API 调用过程中发生了一个堆破坏，那么使用 gcUnmanagedto-Managed 这个 MDA 就可以节约大量的调试时间。

5.4 调试托管堆的碎片问题

在本章的前面，我们曾介绍了一种现象称之为堆碎片，其中空闲内存块和被占用内存块在托管堆中交错存放，使程序出现了各种问题，通常表现为抛出 OutOfMemory 异常。虽然在实际情况中有足够多的空闲内存，但这些内存却不是连续的，从而需要 CLR 堆管理器通过紧缩与合并技术来减少发生堆碎片的风险。在本节中，我们将看到一个导致堆碎片发生的示例，以及如何使用调试器来找出堆碎片问题并且分析为什么会发生这个问题。清单 5-8 给出了这个示例。

清单 5-8 堆中的碎片

```
using System;
using System.Text;
using System.Runtime.InteropServices;

namespace Advanced.NET.Debugging.Chapter5
{
    class Fragment
    {
        static void Main(string[] args)
        {
            Fragment f = new Fragment();
            f.Run(args);
        }

        public void Run(string[] args)
        {
            if (args.Length < 2)
            {
                Console.WriteLine("05Fragment.exe <alloc. size> <max mem in MB>");
                return;
            }

            int size = Int32.Parse(args[0]);
            int maxmem = Int32.Parse(args[1]);
            byte[][] nonPinned = null;
            byte[][] pinned = null;
            GCHandle[] pinnedHandles = null;

            int numAllocs=maxmem*1000000/size;

            pinnedHandles = new GCHandle[numAllocs];

            pinned = new byte[numAllocs / 2][];
            nonPinned = new byte[numAllocs / 2][];
```

```
            for (int i = 0; i < numAllocs / 2; i++)
            {
                nonPinned[i] = new byte[size];
                pinned[i] = new byte[size];
    pinnedHandles[i] =
GCHandle.Alloc(pinned[i], GCHandleType.Pinned);
            }
            Console.WriteLine("Press any key to GC & promo to gen1");
            Console.ReadKey();

            GC.Collect();

            Console.WriteLine("Press any key to GC  & promo to gen2");
            Console.ReadKey();

            GC.Collect();

            Console.WriteLine("Press any key to GC(free non pinned");
            Console.ReadKey();

            for (int i = 0; i < numAllocs / 2; i++)
            {
                nonPinned[i] = null;
            }

            GC.Collect();

            Console.WriteLine("Press any key to exit");
            Console.ReadKey();
        }
    }
}
```

清单 5-8 中的源代码和二进制文件位于以下文件夹中：

- 源代码文件：C:\ADND\Chapter5\Fragment
- 二进制文件：C:\ADNDBin\05Fragment. exe

在这个程序中可以指定内存分配的大小，以及程序消耗的最大内存总量。例如，如果每次希望分配大小为 50 000 个字节的内存，并且总的内存消耗量的上限值为 100MB，那么可以像下面这样来运行程序：

```
C:\ADNDBIN\05Fragment 50000 100
```

应用程序会持续分配内存，每次分配指定大小的内存块，直到到达上限值。在执行了分配操作后，应用程序会执行两次垃圾收集操作将现有的对象提升到第 2 代，然后消除未固定对象的根对象引用，接下来再执行一次垃圾收集操作，并在这次操作中释放未固定对象。在调试器下运行这个程序，每次分配的大小为 50 000 字节，最大的内存阈值为 1GB。

在出现了提示信息“Press any key to GC and promo to Gen1”后，程序已经完成了所有的内存分配，可以通过命令 DumpHeap-stat 来观察托管堆：

```
0:004> !DumpHeap -stat
total 22812 objects
Statistics:
      MT    Count    TotalSize Class Name
79119954        1           12 System.Security.Permissions.ReflectionPermission
79119834        1           12 System.Security.Permissions.FileDialogPermission
791197b0        1           12 System.Security.PolicyManager
...
...
...
791032a8        2          256 System.Globalization.NumberFormatInfo
79101fe4        6          336 System.Collections.Hashtable
7912d9bc        6          864 System.Collections.Hashtable+bucket[]
7912dd40       10         2084 System.Char[]
00395f68      564        13120      Free
7912d8f8       14        17348 System.Object[]
791379e8        1        80012 System.Runtime.InteropServices.GCHandle[]
79141f50        2        80032 System.Byte[][]
790fd8c4     2108       132148 System.String
7912dae8    20002   1000240284 System.Byte[]
Total 22812 objects
```

在这个命令的输出中需要注意一些信息。由于我们正在观察堆碎片问题，因此对于任何标记为“Free”的内存块都必须仔细分析。在示例中，似乎有 564 个空闲内存块，占据的总大小为 13 120 字节。我们是否应该担心这些空闲内存块会造成堆碎片？通常来说，观察空闲块总大小与托管堆总大小的比值能够带来一些有用的信息。如果空闲块占据了大部分的堆，那么堆碎片可能会是一个问题，从而应该进一步分析。另一个重要的问题就是，堆碎片发生在哪一代。在第 0 代中通常没有碎片，因为 CLR 堆管理器可以使用任何可用的空闲内存块来进行分配。然而，在第 1 代和第 2 代中，使用空闲块的唯一方式是将对象提升到各自的代。由于第 1 代是临时内存段的一部分，并且只有一个临时内存段，因此在观察堆碎片问题时，通常需要分析第 2 代的内存空间。我们通过命令 eeheap-gc 观察堆：

```
0:004> !eeheap -gc
Number of GC Heaps: 1
generation 0 starts at 0x56192a54
generation 1 starts at 0x55d91000
generation 2 starts at 0x01c21000
ephemeral segment allocation context: none
 segment    begin allocated     size
003a80e0 790d8620  790f7d8c 0x0001f76c(128876)
01c20000 01c21000  0282db84 0x00c0cb84(12635012)
04800000 04801000  05405ee4 0x00c04ee4(12603108)
05800000 05801000  06405ee4 0x00c04ee4(12603108)
```

```
06a50000 06a51000  07655ee4 0x00c04ee4(12603108)
07a50000 07a51000  08655ee4 0x00c04ee4(12603108)
...
...
...
4fd90000 4fd91000  50995ee4 0x00c04ee4(12603108)
50d90000 50d91000  51995ee4 0x00c04ee4(12603108)
51d90000 51d91000  52995ee4 0x00c04ee4(12603108)
52d90000 52d91000  53995ee4 0x00c04ee4(12603108)
53d90000 53d91000  54995ee4 0x00c04ee4(12603108)
54d90000 54d91000  55995ee4 0x00c04ee4(12603108)
55d90000 55d91000  5621afd8 0x00489fd8(4759512)
Large object heap starts at 0x02c21000
 segment    begin allocated     size
02c20000 02c21000  02c23250 0x00002250(8784)
Total Size  0x3ba38e90(1000574608)
------------------------------
GC Heap Size  0x3ba38e90(1000574608)
```

输出信息的最后一行指出，GC 堆的总大小刚好在 1GB 左右。你还会注意到，有一个非常大的内存段列表。由于我们正在分配一块非常大的内存，因此临时内存段会被很快地填满，并且将创建新的第 2 代内存段。我们可以验证这种情况：观察前面输出中第 2 代的起始地址（0x01c21000）并将内存段列表中所有内存段的起始地址关联起来。回到前面观察到的空闲块。这些空闲块位于哪一代中？我们可以通过命令 dumpheap-type Free 来找出答案，它的简化输出如下所示：

```
0:004> !DumpHeap -type Free
 Address       MT     Size
01c21000 00395f68       12 Free
01c2100c 00395f68       24 Free
01c24c44 00395f68       12 Free
01c24c50 00395f68       12 Free
01c24c5c 00395f68     6336 Free
01e299d0 00395f68       12 Free
0202a6f4 00395f68       12 Free
0222b418 00395f68       12 Free
0242c13c 00395f68       12 Free
0262ce60 00395f68       12 Free
04801000 00395f68       12 Free
0480100c 00395f68       12 Free
04a01d30 00395f68       12 Free
04c02a54 00395f68       12 Free
04e03778 00395f68       12 Free
0500449c 00395f68       12 Free
052051c0 00395f68       12 Free
05801000 00395f68       12 Free
0580100c 00395f68       12 Free
05a01d30 00395f68       12 Free
05c02a54 00395f68       12 Free
05e03778 00395f68       12 Free
```

```
0600449c 00395f68        12 Free
062051c0 00395f68        12 Free
06a51000 00395f68        12 Free
06a5100c 00395f68        12 Free
06c51d30 00395f68        12 Free
06e52a54 00395f68        12 Free
07053778 00395f68        12 Free
0725449c 00395f68        12 Free
074551c0 00395f68        12 Free
07a51000 00395f68        12 Free
07a5100c 00395f68        12 Free
07c51d30 00395f68        12 Free
07e52a54 00395f68        12 Free
08053778 00395f68        12 Free
0825449c 00395f68        12 Free
084551c0 00395f68        12 Free
08a51000 00395f68        12 Free
08a5100c 00395f68        12 Free
08c51d30 00395f68        12 Free
08e52a54 00395f68        12 Free
09053778 00395f68        12 Free
0925449c 00395f68        12 Free
094551c0 00395f68        12 Free
09a51000 00395f68        12 Free
09a5100c 00395f68        12 Free
09c51d30 00395f68        12 Free
09e52a54 00395f68        12 Free
0a053778 00395f68        12 Free
0a25449c 00395f68        12 Free
0a4551c0 00395f68        12 Free
0aee1000 00395f68        12 Free
0aee100c 00395f68        12 Free
0b0e1d30 00395f68        12 Free
0b2e2a54 00395f68        12 Free
0b4e3778 00395f68        12 Free
...
...
...
55192a54 00395f68        12 Free
55393778 00395f68        12 Free
5559449c 00395f68        12 Free
557951c0 00395f68        12 Free
55d91000 00395f68        12 Free
55d9100c 00395f68        12 Free
55f91d30 00395f68        12 Free
56192a54 00395f68        12 Free
02c21000 00395f68        16 Free
02c22010 00395f68        16 Free
02c23020 00395f68        16 Free
02c23240 00395f68        16 Free
total 564 objects
```

```
Statistics:
      MT    Count    TotalSize Class Name
00395f68      564        13120      Free
Total 564 objects
```

通过查看每个空闲块的地址，并将该地址与 eeheap 命令输出的内存段地址关联起来，我们可以看到大部分的空闲对象都是位于第 2 代中。在堆中总共的空闲块大小为 13 120 字节，而堆大小大约在 1GB 左右，因而现在碎片占据的比例非常小，不需要担心碎片问题。恢复程序运行，并且在出现提示信息时持续按下任意键，直到看到“Press any key to exit”为止。此时，中断进入到调试器，并再次执行 DumpHeap-stat 命令来获得堆的信息：

```
0:004> !DumpHeap -stat
total 22233 objects
Statistics:
      MT    Count    TotalSize Class Name
79119954        1           12 System.Security.Permissions.ReflectionPermission
79119834        1           12 System.Security.Permissions.FileDialogPermission
791197b0        1           12 System.Security.PolicyManager
00113038        1           12 Advanced.NET.Debugging.Chapter5.Fragment
791052a8        1           16 System.Security.Permissions.UIPermission
79117480        1           20 System.Security.Permissions.EnvironmentPermission
791037c0        1           20 Microsoft.Win32.SafeHandles.SafeFileMappingHandle
79103764        1           20 Microsoft.Win32.SafeHandles.SafeViewOfFileHandle
…
…
…
7912d8f8       12        17256 System.Object[]
791379e8        1        80012 System.Runtime.InteropServices.GCHandle[]
79141f50        2        80032 System.Byte[][]
790fd8c4     2101       131812 System.String
00395f68    10006    496172124      Free
7912dae8    10002    500120284 System.Byte[]
Total 22233 objects
```

这时，我们可以看到空闲空间的总量极大地增长了。从输出信息中可以看到共有 10 006 个实例，占据了总量为 496 172 124 字节的内存。要找出总量与整体堆大小的关系，我们再次使用 eeheap-gc 命令：

```
0:004> !eeheap -gc
Number of GC Heaps: 1
generation 0 starts at 0x55d9100c
generation 1 starts at 0x55d91000
generation 2 starts at 0x01c21000
ephemeral segment allocation context: none
 segment    begin allocated     size
003a80e0 790d8620  790f7d8c 0x0001f76c(128876)
01c20000 01c21000  02821828 0x00c00828(12585000)
04800000 04801000  053f9b88 0x00bf8b88(12553096)
…
```

```
...
...
54d90000 54d91000  55989b88 0x00bf8b88(12553096)
55d90000 55d91000  562190b0 0x004880b0(4751536)
Large object heap starts at 0x02c21000
 segment    begin allocated     size
02c20000 02c21000  02c23240 0x00002240(8768)
Total Size  0x3b6725f4(996615668)
------------------------------
GC Heap Size  0x3b6725f4(996615668)
```

在输出中 GC 堆总大小为 996 615 668 字节。总体上，我们可以认为堆中有 50% 的碎片。通过 DumpHeap 命令的输出可以验证这一点：

```
0:004> !DumpHeap
 Address       MT     Size
...
...
...
55ff381c 7912dae8    50012
55fffb78 00395f68    50012 Free
5600bed4 7912dae8    50012
56018230 00395f68    50012 Free
5602458c 7912dae8    50012
560308e8 00395f68    50012 Free
5603cc44 7912dae8    50012
56048fa0 00395f68    50012 Free
560552fc 7912dae8    50012
56061658 00395f68    50012 Free
5606d9b4 7912dae8    50012
56079d10 00395f68    50012 Free
5608606c 7912dae8    50012
560923c8 00395f68    50012 Free
5609e724 7912dae8    50012
560aaa80 00395f68    50012 Free
560b6ddc 7912dae8    50012
560c3138 00395f68    50012 Free
560cf494 7912dae8    50012
560db7f0 00395f68    50012 Free
560e7b4c 7912dae8    50012
560f3ea8 00395f68    50012 Free
56100204 7912dae8    50012
5610c560 00395f68    50012 Free
...
...
...
```

从输出中，我们可以看到：首先一个大小为 50 012 字节的内存块被分配和使用，接着是一个同样大小的空闲内存块。我们可以在这个已分配的对象上使用 DumpObj 命令来找出更详细的信息：

```
0:004> !DumpObj 5606d9b4
Name: System.Byte[]
MethodTable: 7912dae8
EEClass: 7912dba0
Size: 50012(0xc35c) bytes
Array: Rank 1, Number of elements 50000, Type Byte
Element Type: System.Byte
Fields:
None
```

这个对象是一个字节数组，对应于程序中执行的内存分配。这种分配模式（已分配；空闲；已分配；空闲……）是如何形成的？我们知道垃圾收集器通常会执行紧缩与合并等操作来避免这种情况。导致垃圾收集器无法执行紧缩与合并的情况之一就是，在堆上存在一些被固定住（即不可移动）的对象。是否存在这种情况，我们需要判断在进程中是否存在类型为“Pinned”的句柄。可以使用 GCHandles 命令来获得进程中句柄的使用情况：

```
0:004> !GCHandles
GC Handle Statistics:
Strong Handles: 15
Pinned Handles: 10004
Async Pinned Handles: 0
Ref Count Handles: 0
Weak Long Handles: 0
Weak Short Handles: 1
Other Handles: 0
Statistics:
      MT    Count    TotalSize Class Name
790fd0f0        1           12 System.Object
790feba4        1           28 System.SharedStatics
790fcc48        2           48 System.Reflection.Assembly
790fe17c        1           72 System.ExecutionEngineException
790fe0e0        1           72 System.StackOverflowException
790fe044        1           72 System.OutOfMemoryException
790fed00        1          100 System.AppDomain
790fe704        2          112 System.Threading.Thread
79100a18        4          144 System.Security.PermissionSet
790fe284        2          144 System.Threading.ThreadAbortException
7912d8f8        4         8744 System.Object[]
7912dae8    10000    500120000 System.Byte[]
Total 10020 objects
```

从 GCHandle 命令输出可以看出，进程中共有 10 004 个类型为“Pinned”的句柄。而且，在 statistics 部分，我们可以看到共有 10 000 个这种句柄用于固定字节数组。此时，通过快速浏览代码，可以看到在程序中有一半的字节数组分配被固定住，导致在堆中产生碎片。

过多的或者过久的“固定”是造成托管堆中出现碎片的最常见原因之一。如果需要固定对象，那么开发人员必须确保固定的时间较短，从而不对垃圾收集器造成太多的影响。

多长时间才算是过长

在前面的示例中，最初堆的碎片量只占 1%。此时，我们无需过多关注碎片问题，因为它太小了。随后，我们注意到碎片的数量增长到了 50%，这才使我们开始进行深入分析以找出造成这种问题的原因。那么，是否存在一个阈值，在超过这个阈值之后才应该开始关注？事实上并不存在某个固定的值，通常来说，如果堆中碎片量在 10% 到 30% 之间，那么我们应该努力确保这种情况不会长时间存在。

在前面的示例中，除了与托管堆相关的碎片问题，还可能会遇到一些情况，例如在 Window 虚拟内存管理器管理的虚拟内存中发生碎片。在这些情况中，CLR 堆管理器可能不会通过增加堆容量长（例如分配新的内存段）来满足分配请求。可以使用 address 命令来获取系统虚拟内存状态的详细信息。

内存耗尽异常的调试

即使 CLR 堆管理器和垃圾收集器努力确保实现自动的内存管理，并且按照最有效的方式来使用内存，但一些糟糕的编程方式仍然会在. NET 程序中产生严重问题。在本章的这部分内容中，我们将讨论一个. NET 程序如何耗尽过多的内存，以至于在程序中发生 OutOfMemoryException 异常，此外还将讨论如何使用调试器来找出问题的根源。需要注意的是，这个程序仅说明了如何在托管环境下耗尽内存，而并没有介绍当通过互用性服务层来调用时如何在非托管代码中造成资源泄漏。在第 7 章中，我们将讨论一个非托管资源泄漏的示例，其中资源泄漏是由于托管代码的不正确调用造成的。

清单 5-9 给出了说明内存耗尽问题的示例程序。

清单 5-9 抛出 OutOfMemoryException 异常的程序示例

```
using System;
using System.IO;
using System.Xml.Serialization;

namespace Advanced.NET.Debugging.Chapter5
{

    public class Person
    {
        private string name;
        private string social;
        private int age;

        public string Name
        {
```

```
            get { return name; }
            set { this.name=value;}
        }

        public string SocialSecurity
        {
            get { return social; }
            set { this.social= value; }
        }

        public int Age
        {
            get { return age; }
            set { this.age = value; }
        }

        public Person() {}
        public Person(string name, string ss, int age)
        {
            this.name = name; this.social = ss; this.age = age;
        }
    }

    class OOM
    {
        static void Main(string[] args)
        {
            OOM o = new OOM();
            o.Run();
        }

        public void Run()
        {
            XmlRootAttribute root = new XmlRootAttribute();
            root.ElementName = "MyPersonRoot";
            root.Namespace = "http://www.contoso.com";
            root.IsNullable = true;

            while (true)
            {
                Person p = new Person();
                p.Name = "Mario Hewardt";
                p.SocialSecurity = "xxx-xx-xxxx";
                p.Age = 99;

                XmlSerializer ser = new
                    XmlSerializer(typeof(Person), root);
                Stream s = new
                    FileStream("c:\\ser.txt", FileMode.Create);

                ser.Serialize(s, p);
```

```
                s.Close();
            }
        }
    }
}
```

清单 5-9 中的源代码和二进制文件位于以下文件夹中：

- 源代码文件：C:\ADND\Chapter5\OOM
- 二进制文件：C:\ADNDBin\05OOM. exe

这个程序非常简单，包含了一个 Person 类和一个 OOM 类。在 OOM 类中包含了一个 Run 方法，它里面有一个循环，在这个循环中创建 Person 类的实例并将这个实例串行化到位于本地驱动器上的 XML 文件中。在运行这个程序时，我们需要监视内存的消耗量，观察它随时间稳定地增长，并且最终抛出一个 OutOfMemoryException 异常。哪些工具可以用来监视进程中的内存使用量？有多种选择。最基本的方式是使用任务管理器（按下组合键 SHIFT-CTRL-ESC）。任务管理器可以显示每个进程的内存信息，例如工作集（working set）、虚拟内存大小、页面缓冲池和非页面缓冲池。在默认情况下，只会显示内存使用（私有工作集）。要显示其他的进程信息，可以点击“查看”菜单下的“选择列”选项。Windows 任务管理器中包含了多个不同的标签页。在查看每个进程的详细信息时，最值得注意的标签页就是 Processes 标签页。Processes 标签页给出了多行信息，其中每行都表示一个运行中的进程，每列是关于进程的特定信息。在图 5-13 中给出了 Windows 任务管理器的一个示例，其中在 Processes 标签页中显示了一组不同的内存信息。

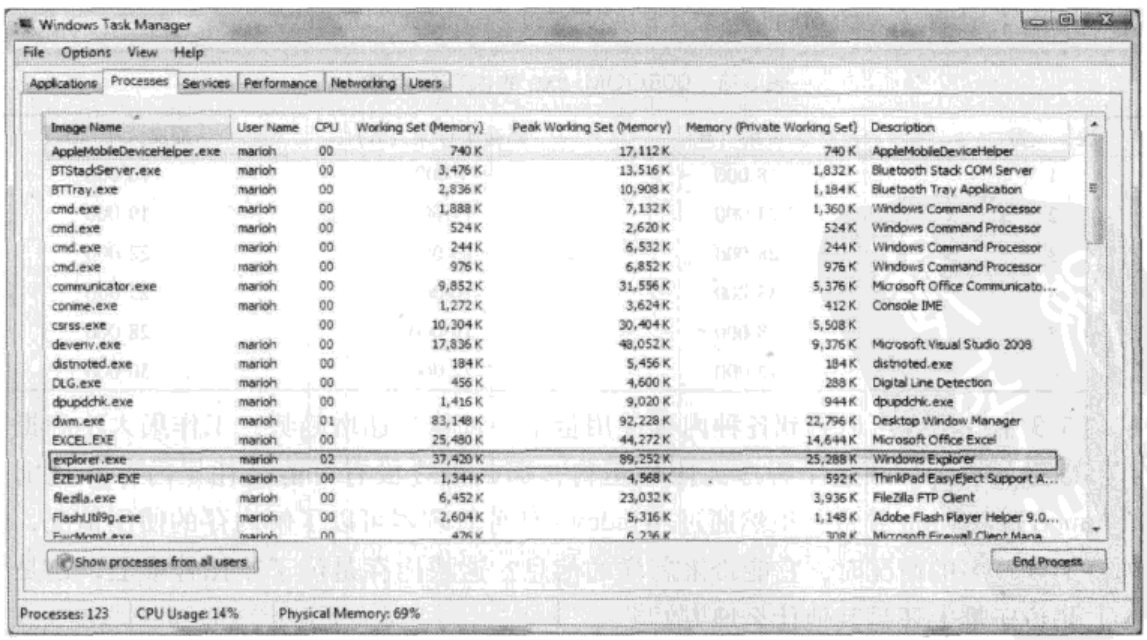


图 5-13　Windows 任务管理器中的 Processes 标签页

从图 5-13 中可以看出，explorer. exe 的工作集大小为 37 420K。在使用 Windows 任务器来分析与内存相关的问题之前，我们需要清楚地了解与内存信息相关的各个列的含义。在表 5-2 中详细给出了最常使用的列以及它们的描述。

Windows Vista 之前的任务管理器

在 Windows Vista 和后续的版本中对任务管理器进行了一些修改，以便更好地描述与内存相关的进程信息。而在此之前，Windows 任务管理器包含一列“虚拟内存大小”，表示进程正在使用的私有字节数量。同样地，“内存使用”列对应于进程的工作集（包括共享内存）。最后，我们在第 8 章中还将使用一个功能，即通过在进程项上点击右键，并在弹出的菜单中选择“创建转储文件”来生成转储文件。

表 5-2 Windows 任务管理器中与内存相关的列

列	描 述
内存使用（Memory-Woeking Set）	在私有工作集中和共享内存中的内存总量
内存使用高峰值（Memory- Peak Working Set）	进程使用的工作集的最大内存量
内存使用增量（Memory- Working Set Delta）	在工作集中变化的内存总量
内存使用－私有工作集（Memory- Private Working Set）	进程中使用的内存总量减去共享内存
内存－提交大小（Memory- Commit Size）	进程提交的虚拟内存总量

运行 05OOM. exe 并观察内存－工作集，内存－私有工作集以及内存－提交大小等列。在表 5-3 中给出了定期（大约 60 秒）采集的结果。

表 5-3 005OOM. exe 的内存使用量

次 数	工作集（K）	私有工作集（K）	提交大小（K）
1	18 000	7 000	16 000
2	24 000	11 000	19 000
3	28 000	13 000	22 000
4	33 000	16 000	25 000
5	38 000	19 000	28 000
6	42 000	22 000	30 000

从表 5-3 中，我们可以看到各种内存使用量呈一种稳步递增趋势，工作集大小和提交大小都在不断地增长。如果这个程序无限地运行，那么最终极有可能会耗尽内存并抛出一个 OutOfMemoryException 异常。虽然通过 Windows 任务管理器可以了解内存的使用情况，但在找出内存消耗过多的情况时，它能带来怎样的信息？这块内存是位于非托管堆上，还是托管堆上？它是位于堆上还是其他什么地方？

要找出这些问题的答案，我们需要一个更有用的工具：Windows 可靠性与性能监视器。

Windows 可靠性与性能监视器是一个功能强大且广泛使用的工具，可以用来分析系统的整体状态或者每个进程的状态。它使用了不同的数据源，例如性能计数器，跟踪日志以及配置信息等。在.NET 调试会话中，性能计数器是最常用的数据源，它能根据指定的时间间隔或者特定的条件来给出某个应用程序或服务的性能特征。例如，在信用卡事务的 Web 服务中可以提供一个性能计数器，给出在指定时间内有多少个失败的事务。Windows 可靠性与性能监视器工具知道在哪些位置上收集这个性能计数器数据，并以一种漂亮的图形和回溯视图来显示结果。要运行这个工具，点击 Windows 开始按钮，并在搜索工具中输入 perfmon.exe（在 Windows Vista 以前的版本中，选择“运行”，然后输入 perfmon.exe）。图 5-14 是工具起始画面的一个示例。

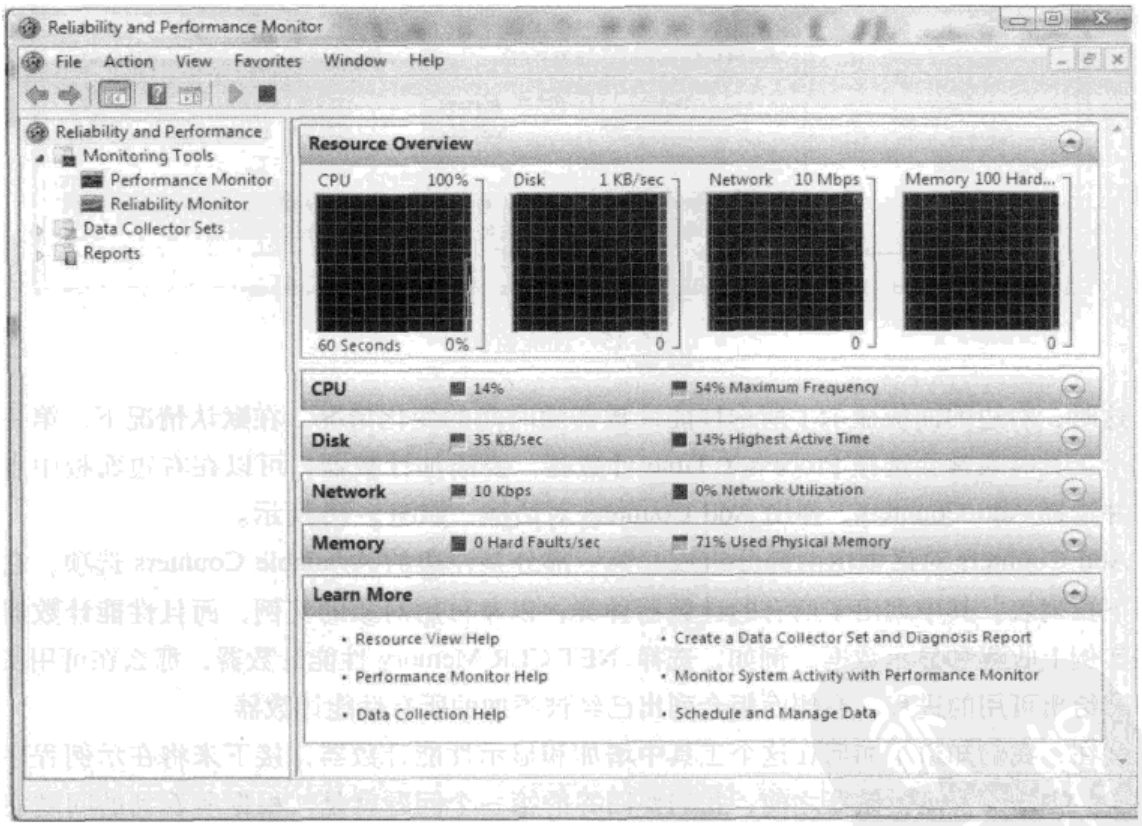


图 5-14　Windows 可靠性与性能监视器

在左边的面板中给出了该工具可用的各种数据源。前面曾经提到过，在诊断.NET 应用程序时将大量使用各种性能计数器，这些计数器位于 Monitoring Tools 节点中。在右边面板中给出了与当前所选数据源相关的数据。第一次启动时，它将显示系统状态，包括 CPU、磁盘、内存和网络等的使用情况。图 5-15 是选择了 Performance Monitor 项的界面。

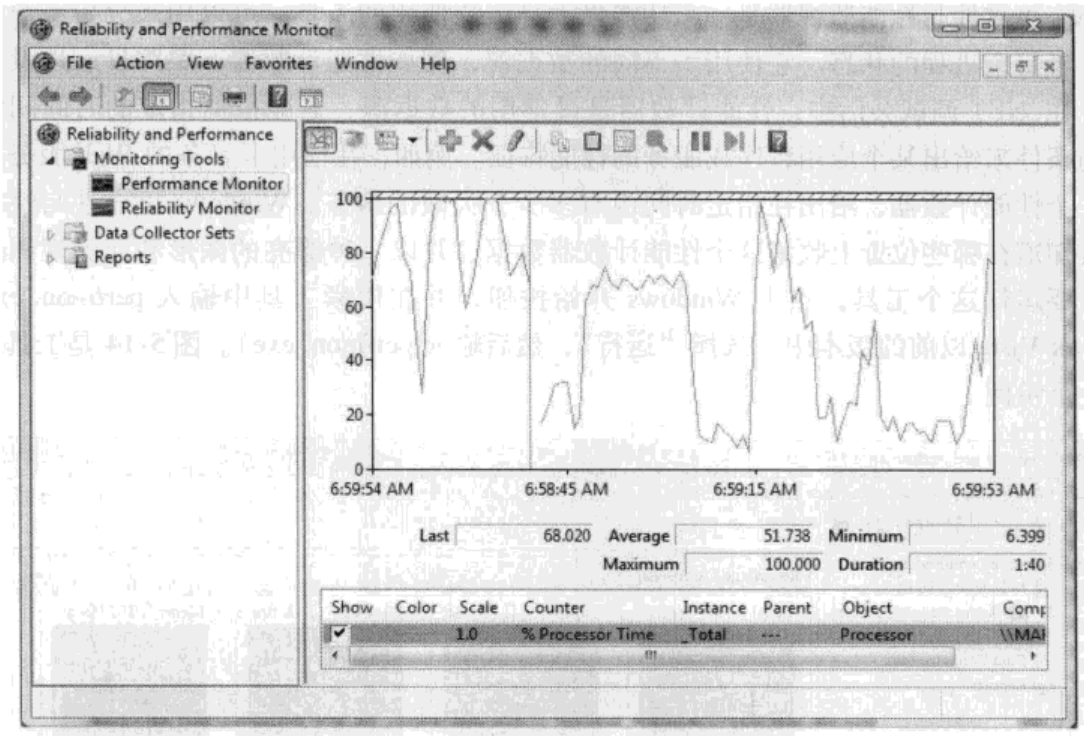


图 5-15 性能监视器

这时，右边的面板显示了所选性能计数器随时间的变化情况。在默认情况下，第一次启动这个工具时通常会选择 Processor Time 计数器。要添加计数器，可以在右边面板中点击右键，并选择 Add Counters，弹出 Add Counters 对话框，如图 5-16 所示。

Add Counters 对话框由两部分组成。第一部分是左边的 Available Counters 选项，它包含一个下拉列表，其中列出了所有的计数器种类，以及可用对象的实例，而且性能计数器将在这些实例上收集和显示数据。例如，选择. NET CLR Memory 性能计数器，那么在可用实例列表中会给出可用的进程。右侧面板会列出已经被添加的所有性能计数器。

现在，我们知道了如何在这个工具中增加和显示性能计数器，接下来将在示例程序上进行试验。在增加性能计数器之前，我们要回答的第一个问题就是，根据所看到的问题症状应该选择哪些性能计数器？表 5-4 列出了可用的 CLR 性能计数器种类，以及相应的描述信息。

表 5-4 CLR 性能计数器的种类

类 别	描 述
. NET CLR 数据	关于数据（例如 SQL）性能的运行时统计信息
. NET CLR 异常	关于 CLR 异常处理的运行时统计信息，例如所抛出的异常数量
. NET CLR 互用性	关于互用性服务的运行时统计，例如列集操作的次数
. NET CLR 即时编译器	关于即时编译器的运行时统计，例如被即时编译器编译的方法数量
. NET CLR 加载过程	关于 CLR 类/程序集加载器的运行时统计，例如加载器堆中的字节总数

（续）

类　别	描　述
.NET CLR 锁与线程	关于锁和线程的运行时统计，例如锁的竞争率
.NET CLR 内存	关于托管堆和垃圾收集器的运行时统计，例如每一代中收集操作的次数
.NET CLR 网络	关于网络的运行时统计，例如已发送和已接收的数据报
.NET CLR 远程操作	关于远程行为的运行时统计，例如每秒钟发生的远程调用次数
.NET CLR 安全	关于安全性的运行时统计，例如运行时检查的总次数

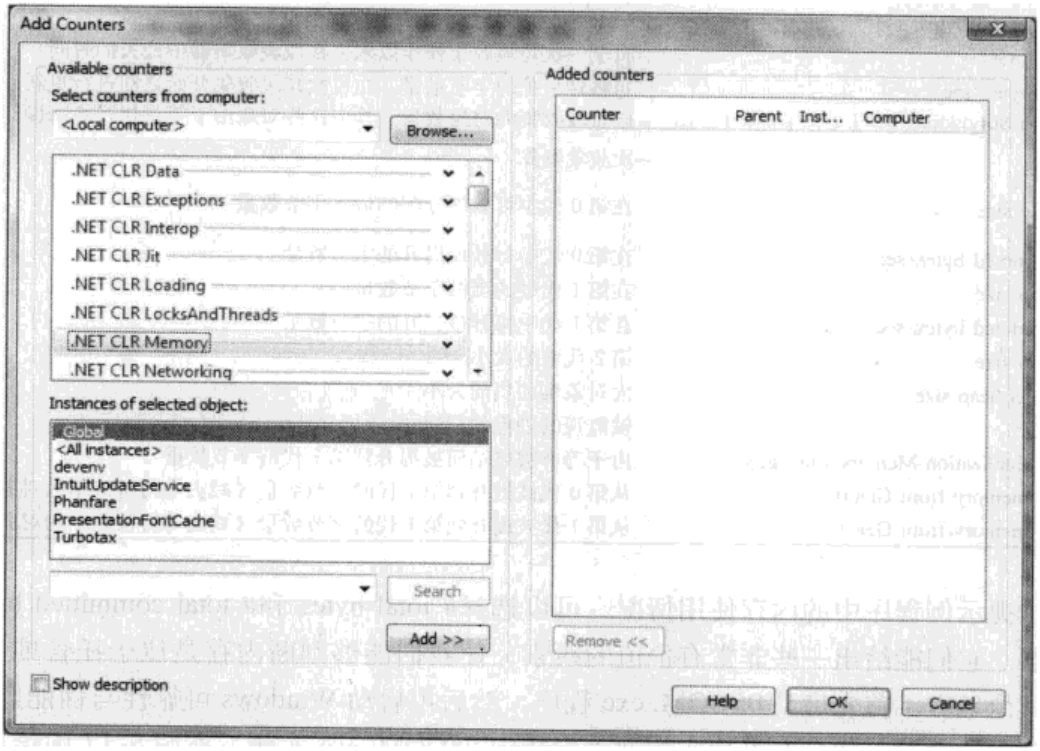


图 5-16　Add Counters 对话框

在这里的示例中，我们需要找出关于内存消耗量（.NET CLR 内存）的更多细节。在表 5-5 中列出了这个类别的计数器及其相应的描述信息。

表 5-5　.NET CLR 内存性能计数器

性能计数器的名字	描　述
# Bytes in all heaps	在所有堆（包括第 0，1，2 代堆以及大对象堆）中的总字节数
# GC Handles	GC 句柄的总数
# Gen 0 collections	在第 0 代中垃圾收集操作的总次数
# Gen 1 collections	在第 1 代中垃圾收集操作的总次数
# Gen 2 collections	在第 2 代中垃圾收集操作的总次数
# Induced GC	对 GC. Collect 的总调用次数

（续）

性能计数器的名字	描 述
# Pinned objects	在最近一次垃圾收集操作中，托管堆中固定对象的总数。请注意，它只会显示被收集代中固定对象的数量。因此，如果在垃圾收集操作中只收集了第0代，那么这个数值只是第0代有多少个固定对象
# of Sink blocks in use	当前有多少个同步块在使用。这个性能计数器在诊断与同步相关的性能问题时非常有用
# Total committed bytes	CLR 堆管理器已提交的虚拟字节总数
# Total reserved bytes	CLR 堆管理器已保留的虚拟字节总数
% Time in GC	自上一次垃圾收集操作以来，在垃圾收集器中经去的时间
Allocated bytes/sec	每秒钟分配的字节数量。在每次垃圾收集开始之前进行更新
Finalization Survivors	被垃圾收集的对象数量，其中这些对象由于等待执行终结操作而错过上一次收集操作
Gen 0 heap size	在第0代中可以被分配的最大字节数量
Gen 0 Promoted bytes/sec	在第0代中每秒被提升的字节数量
Gen 1 heap size	在第1代中的当前字节数量
Gen 1 Promoted bytes/sec	在第1代中每秒被提升的字节数量
Gen 2 heap size	第2代堆的大小
Large object heap size	大对象堆的当前大小
Process ID	被监视的进程的标识
Promoted finalization-Memory from gen 0	由于等待被终结而被提升到第1代的字节数量
Promoted memory from Gen 0	从第0代被提升到第1代的字节数量（减去等待被终结的对象的大小）
Promoted memory from Gen 1	从第1代被提升到第1代的字节数量（减去等待被终结的对象的大小）

要监视示例程序中的内存使用情况，可以选择# total bytes 和# total committed bytes 这两个计数器。它们能给出一些非常有价值的线索，使我们能够判断内存是位于托管堆上还是进程中的其他地方。首先启动 05OOM. exe 程序，然后再启动 Windows 可靠性与性能监测工具。添加这两个计数器，并且在可用实例列表中指定 05OOM. exe 实例。在图 5-17 中给出了这个工具在 05OOM. exe 运行两分钟之后的输出。

这个计数器看上去非常稳定，没有明显地上升。然而，如果在 Windows 任务管理器中查看 05OOM 进程，我们可以看到内存消耗量增加了不少。这些内存从何而来？此时，我们排除了托管堆是导致内存使用量增加的原因，并将观察策略改为通过其他的计数器来找出内存使用量上升的原因。例如，我们选择“bytes in loader heap”和“current assemblies”（二者都位于“. NET CLR 加载过程”类别下），并观察输出信息（见图 5-18）。

注意，根据程序执行时间的长短，你可能需要将纵坐标轴的最大值（在属性中）设置为一个更大的值。在图 5-18 中，纵坐标轴的最大值被设置为 5000。这时，我们可以看到一些值得注意的数据。在这两个性能计数器中的字节数都会随着时间缓慢地增长。我们的假设之一是可能存在潜在的程序集泄漏。为了验证这一点，我们可以将调试器附载到 05OOM. exe 进程（ntsd -pn 05OOM. exe），并执行命令 eeheap -loader：

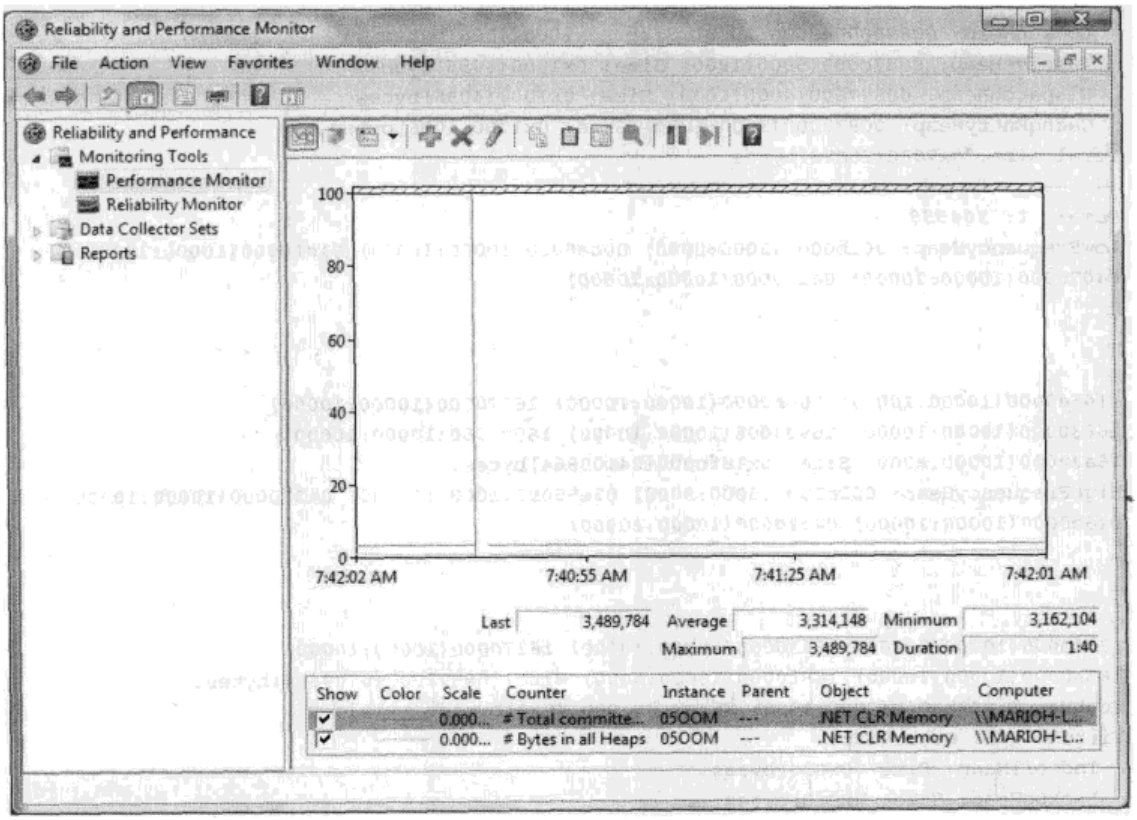


图 5-17　监视 05OOM. exe 的总字节数和已提交字节数

```
0:003> !eeheap -loader
Loader Heap:
-------------------------------------------

System Domain: 7a3bc8b8
LowFrequencyHeap: Size: 0x0(0)bytes.
HighFrequencyHeap: 002a2000(8000:1000) Size: 0x1000(4096)bytes.
StubHeap: 002aa000(2000:2000) Size: 0x2000(8192)bytes.
Virtual Call Stub Heap:
  IndcellHeap: Size: 0x0(0)bytes.
  LookupHeap: Size: 0x0(0)bytes.
  ResolveHeap: Size: 0x0(0)bytes.
  DispatchHeap: Size: 0x0(0)bytes.
  CacheEntryHeap: Size: 0x0(0)bytes.
Total size: 0x3000(12288)bytes
-------------------------------------------

Shared Domain: 7a3bc560
LowFrequencyHeap: 002d0000(2000:1000) Size: 0x1000(4096)bytes.
HighFrequencyHeap: 002d2000(8000:1000) Size: 0x1000(4096)bytes.
StubHeap: 002da000(2000:1000) Size: 0x1000(4096)bytes.
Virtual Call Stub Heap:
```

```
  IndcellHeap: 00870000(2000:1000) Size: 0x1000(4096)bytes.
  LookupHeap: 00875000(2000:1000) Size: 0x1000(4096)bytes.
  ResolveHeap: 0087b000(5000:1000) Size: 0x1000(4096)bytes.
  DispatchHeap: 00877000(4000:1000) Size: 0x1000(4096)bytes.
  CacheEntryHeap: 00872000(3000:1000) Size: 0x1000(4096)bytes.
Total size: 0x7000(28672)bytes
-----------------------------------------
Domain 1: 304558
LowFrequencyHeap: 002b0000(2000:2000) 00ca0000(10000:10000) 01cf0000(10000:10000)
04070000(10000:10000) 04170000(10000:10000)
...
...
...
 165e0000(10000:10000) 166b0000(10000:10000) 16770000(10000:10000)
16830000(10000:10000) 16900000(10000:10000) 169c0000(10000:10000)
16a80000(10000:a000) Size: 0x16fc000(24100864)bytes.
HighFrequencyHeap: 002b2000(8000:8000) 03e50000(10000:10000) 04370000(10000:10000)
046c0000(10000:10000) 04a10000(10000:10000)
...
...
...
15bf0000(10000:10000) 15f30000(10000:10000) 16270000(10000:10000)
165a0000(10000:10000) 168f0000(10000:a000) Size: 0x572000(5709824)bytes.
StubHeap: 002ba000(2000:1000) Size: 0x1000(4096)bytes.
Virtual Call Stub Heap:
  IndcellHeap: Size: 0x0(0)bytes.
  LookupHeap: Size: 0x0(0)bytes.
  ResolveHeap: 002ca000(6000:1000) Size: 0x1000(4096)bytes.
  DispatchHeap: 002c7000(3000:1000) Size: 0x1000(4096)bytes.
  CacheEntryHeap: 002c2000(4000:1000) Size: 0x1000(4096)bytes.
Total size: 0x1c71000(29822976)bytes
-----------------------------------------
Jit code heap:
LoaderCodeHeap: 165f0000(10000:b000) Size: 0xb000(45056)bytes.
LoaderCodeHeap: 15de0000(10000:10000) Size: 0x10000(65536)bytes.
LoaderCodeHeap: 15600000(10000:10000) Size: 0x10000(65536)bytes.
...
...
...
LoaderCodeHeap: 04710000(10000:10000) Size: 0x10000(65536)bytes.
LoaderCodeHeap: 009e0000(10000:10000) Size: 0x10000(65536)bytes.
Total size: 0x23b000(2338816)bytes
-----------------------------------------
Module Thunk heaps:
Module 790c2000: Size: 0x0(0)bytes.
Module 002d2564: Size: 0x0(0)bytes.
...
...
...
Module 168f8e40: Size: 0x0(0)bytes.
Module 168f93b8: Size: 0x0(0)bytes.
```

```
Module 168f9930: Size: 0x0(0)bytes.
Total size: 0x0(0)bytes
-----------------------------------------
Module Lookup Table heaps:
Module 790c2000: Size: 0x0(0)bytes.
Module 002d2564: Size: 0x0(0)bytes.
Module 002d21d8: Size: 0x0(0)bytes.
...
...
...
Module 168f93b8: Size: 0x0(0)bytes.
Module 168f9930: Size: 0x0(0)bytes.
Total size: 0x0(0)bytes
-----------------------------------------
Total LoaderHeap size: 0x1eb6000(32202752)bytes
=========================================
```

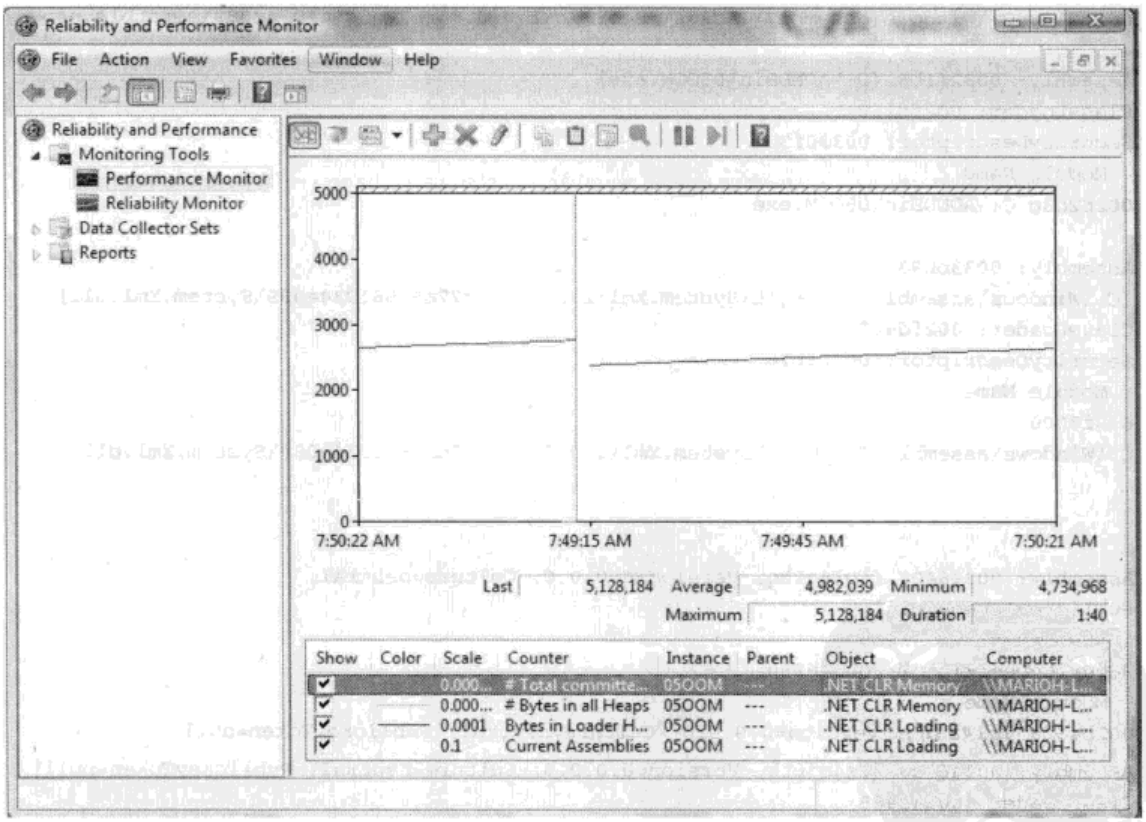


图 5-18　监视 loader heap 性能计数器中的 05OOM. exe 当前程序集和字节数

前两个应用程序域（系统应用程序域和共享应用程序域）看上去很正常，但在默认应用程序域中包含了大量的数据。具体来说，它包含了整个加载器堆（大小为 32 202 752）。为什么在应用程序域中包含了如此多的数据？我们可以进一步获得默认应用程序域的信息，使用 DumpDomain 命令并且将默认应用程序域的地址（可以在之前 eeheap 命令的输出中找到）作为命令的参数：

```
0:003> !DumpDomain 304558
-----------------------------------------
Domain 1: 00304558
LowFrequencyHeap: 0030457c
HighFrequencyHeap: 003045d4
StubHeap: 0030462c
Stage: OPEN
SecurityDescriptor: 00305ab8
Name: 05OOM.exe
Assembly: 0030d1b0
[C:\Windows\assembly\GAC_32\mscorlib\2.0.0.0__b77a5c561934e089\mscorlib.dll]
ClassLoader: 002fc988
SecurityDescriptor: 0030dfd8
  Module Name
790c2000 C:\Windows\assembly\GAC_32\mscorlib\2.0.0.0__b77a5c561934e089\mscorlib.dll
002d2564 C:\Windows\assembly\GAC_32\mscorlib\2.0.0.0__b77a5c561934e089\sortkey.nlp
002d21d8 C:\Windows\assembly\GAC_32\mscorlib\2.0.0.0__b77a5c561934e089\sorttbls.nlp

Assembly: 0032f1b8 [C:\ADNDBin\05OOM.exe]
ClassLoader: 002fd168
SecurityDescriptor: 00330f30
  Module Name
002b2c3c C:\ADNDBin\05OOM.exe

Assembly: 0033bb98
[C:\Windows\assembly\GAC_MSIL\System.Xml\2.0.0.0__b77a5c561934e089\System.Xml.dll]
ClassLoader: 002fd408
SecurityDescriptor: 00326b18
  Module Name
639f8000
C:\Windows\assembly\GAC_MSIL\System.Xml\2.0.0.0__b77a5c561934e089\System.Xml.dll
...
...
...
Assembly: 00346408 [4ql4a3hq, Version=0.0.0.0, Culture=neutral,
PublicKeyToken=null]
ClassLoader: 003423a8
SecurityDescriptor: 00346380
  Module Name
002b46f8 4ql4a3hq, Version=0.0.0.0, Culture=neutral, PublicKeyToken=null
Assembly: 003465a0 [lx4qjutk, Version=0.0.0.0, Culture=neutral, PublicKeyToken=null]
ClassLoader: 00342488
SecurityDescriptor: 00346518
  Module Name
002b4ce4 lx4qjutk, Version=0.0.0.0, Culture=neutral, PublicKeyToken=null

Assembly: 003466b0 [uds1hfbo, Version=0.0.0.0, Culture=neutral, PublicKeyToken=null]
ClassLoader: 003424f8
SecurityDescriptor: 00346628
  Module Name
```

```
002b5258 uds1hfbo, Version=0.0.0.0, Culture=neutral, PublicKeyToken=null

...
...
...
```

我们可以看到，有多个程序集被加载到默认的应用程序域中。而且，程序集的名字看似是随机的。为什么这些程序集会被加载？而清单 5-9 中的代码并没有直接加载任何程序集，这就意味着这些程序集一定是被动态加载的。为了进一步分析这些程序集所包含的内容，我们选择其中一个程序集，并通过 DumpModule 命令转储出相应的模块信息：

```
0:003> !DumpModule 002b5258
Name: uds1hfbo, Version=0.0.0.0, Culture=neutral, PublicKeyToken=null
Attributes: PEFile
Assembly: 003466b0
LoaderHeap: 00000000
TypeDefToMethodTableMap: 00ca2df8
TypeRefToMethodTableMap: 00ca2e10
MethodDefToDescMap: 00ca2e6c
FieldDefToDescMap: 00ca2ed0
MemberRefToDescMap: 00ca2ef8
FileReferencesMap: 00ca2fec
AssemblyReferencesMap: 00ca2ff0
MetaData start address: 00cc07c8 (4344 bytes)
```

接下来，使用命令 dc 并指定起始地址和结束地址（起始地址加上元数据的大小）来转储出模块的元数据：

```
0:003> dc 00cc07c8 00cc07c8+0n4344
00cc07c8  424a5342 00010001 00000000 0000000c  BSJB............
00cc07d8  302e3276 3730352e 00003732 00050000  v2.0.50727......
00cc07e8  0000006c 00000528 00007e23 00000594  l...(...#~......
00cc07f8  0000077c 72745323 73676e69 00000000  |...#Strings....
...
...
...
00cc0d58  00000000 6f4d3c00 656c7564 6475003e  .....<Module>.ud
00cc0d68  66683173 642e6f62 58006c6c 65536c6d  s1hfbo.dll.XmlSe
00cc0d78  6c616972 74617a69 576e6f69 65746972  rializationWrite
00cc0d88  72655072 006e6f73 7263694d 666f736f  rPerson.Microsof
00cc0d98  6d582e74 65532e6c 6c616972 74617a69  t.Xml.Serializat
00cc0da8  2e6e6f69 656e6547 65746172 7734164  ion.GeneratedAss
00cc0db8  6c626d65 6d580079 7265536c 696c6169  embly.XmlSeriali
00cc0dc8  6974617a 65526e6f 72656461 73726550  zationReaderPers
00cc0dd8  58006e6f 65536c6d 6c616972 72657a69  on.XmlSerializer
00cc0de8  65500031 6e6f7372 69726553 7a696c61  1.PersonSerializ
00cc0df8  58007265 65536c6d 6c616972 72657a69  er.XmlSerializer
00cc0e08  746e6f43 74636172 73795300 2e6d6574  Contract.System.
00cc0e18  006c6d58 74737953 582e6d65 532e6c6d  Xml.System.Xml.S
00cc0e28  61697265 617a696c 6e6f6974 6c6d5800  erialization.Xml
00cc0e38  69726553 7a696c61 6f697461 6972576e  SerializationWri
```

```
00cc0e48  00726574 536c6d58 61697265 617a696c  ter.XmlSerializa
00cc0e58  6e6f6974 64616552 58007265 65536c6d  tionReader.XmlSe
00cc0e68  6c616972 72657a69 6c6d5800 69726553  rializer.XmlSeri
…
…
…
```

现在，我们对问题应该有所了解了。从元数据的输出中，我们可以看到与程序集相关的模块中包含了对某种 XML 串行化的引用。而且，这个模块包含的 XML 串行化类型看上去与代码中 Person 类的串行化是相关的。根据这个事实，我们现在可以假设程序中 XML 串行化代码是导致这些动态程序集的原因。下一步就是查看 XmlSerializer 类的帮助文档。在 MSDN 中很清楚地指出，基于性能的原因，在使用 XmlSerializer 类时可能会动态创建一个专门的程序集来处理这种串行化操作。具体来说，在 XmlSerializer 的构造函数中，有 7 个构造函数将动态创建程序集，而剩下的 2 个构造函数则包含了一些重用逻辑来减少动态程序集的数量。

前面的内容介绍了如何使用 Windows 任务管理器来监视. NET 程序的整体内存使用量，以及如何使用 Widnows 可靠性与性能监视器工具来分析 CLR 的特性。这里，我们假设能够实时地运行和监视程序。在许多情况中，程序可能会持续运行直到耗尽内存并抛出一个 OutOfMemoryException 异常。如果让示例程序无限运行下去，那么将报告一个 OutOfMemoryException 异常，如下所示：

```
(1830.1f20): CLR exception - code e0434f4d (first/second chance not available)
eax=0027ed2c ebx=e0434f4d ecx=00000001 edx=00000000 esi=0027edb4 edi=00338510
eip=775842eb esp=0027ed2c ebp=0027ed7c iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for
kernel32.dll -
kernel32!RaiseException+0x58:
```

要获得关于托管异常的进一步信息，可以使用 PrintException 命令：

```
0:000> kb
ChildEBP RetAddr  Args to Child
0027ed7c 79f071ac e0434f4d 00000001 00000001 kernel32!RaiseException+0x58
0027eddc 79f0a780 51e10dac 00000001 00000000
 mscorwks!RaiseTheExceptionInternalOnly+0x2a8
*** WARNING: Unable to verify checksum for System.ni.dll
0027ee80 7a53e025 0027f14c 79f0a3d9 0027f338 mscorwks!JIT_Rethrow+0xbf
0027ef4c 7a53d665 51df597c 00000000 51de0050 System_ni+0xfe025
0027ef80 7a4d078a 51df597c 51de0050 638fcb39 System_ni+0xfd665
*** WARNING: Unable to verify checksum for System.Xml.ni.dll
0027efec 638fb6e5 00000000 51de02cc 00000000 System_ni+0x9078a
0027f078 638fa683 51ddff88 00000000 51de02cc System_Xml_ni+0x15b6e5
0027f09c 63960d09 00000000 00000000 00000000 System_Xml_ni+0x15a683
0027f0c4 6396090c 00000000 00000000 00000000 System_Xml_ni+0x1c0d09
0027f120 79e7c74b 00000000 0027f158 0027f1b0 System_Xml_ni+0x1c090c
00000000 00000000 00000000 00000000 00000000 mscorwks!CallDescrWorker+0x33
```

```
0:000> !PrintException 51e10dac
Exception object: 51e10dac
Exception type: System.OutOfMemoryException
Message: <none>
InnerException: <none>
StackTrace (generated):
    SP       IP       Function
    0027EE94 7942385A mscorlib_ni!System.Reflection.Assembly.Load
(Byte[], Byte[], System.Security.Policy.Evidence)+0x3a
    0027EEB0 7A4BF513 System_ni!Microsoft.CSharp.CSharpCodeGenerator.FromFileBatch
(System.CodeDom.Compiler.CompilerParameters, System.String[])+0x3ab
    0027EF00 7A53E025 System_ni!Microsoft.CSharp.CSharpCodeGenerator.FromSourceBatch
(System.CodeDom.Compiler.CompilerParameters, System.String[])+0x1f1
    0027EF58 7A53D665 System_ni!Microsoft.CSharp.CSharpCodeGenerator.System.CodeDom.
Compiler.ICodeCompiler.CompileAssemblyFromSourceBatch
(System.CodeDom.Compiler.CompilerParameters, System.String[])+0x29
    0027EF8C 7A4D078A System_ni!System.CodeDom.Compiler.CodeDomProvider.
CompileAssemblyFromSource(System.CodeDom.Compiler.CompilerParameters,
System.String[])+0x16
    0027EF98 638FCB39 System_Xml_ni!System.Xml.Serialization.Compiler.Compile
(System.Reflection.Assembly, System.String,
System.Xml.Serialization.XmlSerializerCompilerParameters,
System.Security.Policy.Evidence)+0x269
    0027F000 638FB6E5
System_Xml_ni!System.Xml.Serialization.TempAssembly.GenerateAssembly
(System.Xml.Serialization.XmlMapping[], System.Type[], System.String,
System.Security.Policy.Evidence,
System.Xml.Serialization.XmlSerializerCompilerParameters,
System.Reflection.Assembly, System.Collections.Hashtable)+0x7e9
    0027F094 638FA683 System_Xml_ni!System.Xml.Serialization.TempAssembly..ctor
(System.Xml.Serialization.XmlMapping[], System.Type[], System.String, System.String,
System.Security.Policy.Evidence)+0x4b
    0027F0B4 63960D09 System_Xml_ni!System.Xml.Serialization.XmlSerializer..ctor
(System.Type, System.Xml.Serialization.XmlAttributeOverrides, System.Type[],
System.Xml.Serialization.XmlRootAttribute, System.String, System.String,
System.Security.Policy.Evidence)+0xed
    0027F0E4 6396090C System_Xml_ni!System.Xml.Serialization.XmlSerializer..ctor
(System.Type, System.Xml.Serialization.XmlRootAttribute)+0x28
    0027F0F4 009201D6 05OOM!Advanced.NET.Debugging.Chapter5.OOM.Run()+0xe6
    0027F118 009200A7
05OOM!Advanced.NET.Debugging.Chapter5.OOM.Main(System.String[])+0x37
StackTraceString: <none>
HResult: 8007000e
There are nested exceptions on this thread. Run with -nested for details
```

此时，应用程序已经失败了，并且我们也无法通过运行监视工具来判断程序的内存使用量。在一些类似的情况中，我们只能利用调试器命令去分析在哪些地方消耗了内存。然而，在这个过程中没有固定的命令或者步骤可供遵循，根据经验法则，我们必须使用各种不同的诊断命令（例如 eeheap、dumpheap 以及 dumpdomain 等）来了解在哪些地方消耗了 CLR 内

存。当然，过多的内存消耗同样可能是由于非托管代码中的内存泄漏而引起的，我们将在第7章中看到一个示例。

在发生 OutOfMemoryException 异常时立即中断

当一个进程耗尽内存时，事情会变得非常复杂，并且程序不能正确地处理这种情况。由于 OutOfMemoryException 异常会在异常链中传播，并且只有在异常未被处理时才会导致进程故障，因此在回卷（unwinding）过程中仍要执行大量的代码，这在某些特定的情况中会进一步加大分析问题的难度。而且，如果这段代码位于一个它并不拥有的进程中，那么该进程可能会捕获所有类型的异常，并且继续运行而不会停止。为了确保在发生了 OutOfMemoryException 异常时立即进入调试器，CLR 引入了一个注册键 GCBreakOnOOM（DWORD），位于以下注册路径：HKEY_ LOCAL_ MACHINE \ SOFTWARE \ Microsoft \ .NETFramework。这个值可以设置为 1，从而会记录一个事件日志消息；也可以设置为 2，表示发生内存耗尽问题时将中断进入到调试器；或者也可以设置为 4，表示将写入更详细的日志，包括在发生内存问题时的内存统计信息。

5.5 小结

要想高效地调试程序中与堆和垃圾收集器相关的问题，就需要深入理解这些组件的内部工作机制。在本章中，我们详细介绍了 CLR 堆管理器以及垃圾收集器的工作流程。首先介绍了 Windows 内存架构以及 CLR 堆管理器在整体架构中的位置，接下来介绍了垃圾收集器使用的各种不同的概念（代、根对象、终结操作等）。我们给出了多个示例代码，并通过调试器和相应的工具来说明这些概念在实际中的应用。最后，我们讨论了各种常见的编程错误示例，以及它们在 CLR 中表现出的症状。这些示例包括，如何找出托管堆中堆破坏情况的根源，如何找出内存耗尽情况的根源以及如何调试错误的终结操作代码等。

第6章 同　　步

本章我们将介绍一些常见的同步（synchronization）问题，以及如何高效地分析和找出这些问题的根源。首先介绍 CLR 的基础同步知识，然后给出一些最为常见的同步问题及其调试过程，最后将介绍如何通过调试器来找出这些问题的根源。

6.1 同步的基础知识

Windows 操作系统是一个抢占式的多线程操作系统。多线程是指，能够并发地运行任意数量的线程。如果系统是一个单处理器机器，那么 Windows 会产生一种并发执行线程的假象：它让每个线程都运行一小段时间（也称之为时间片，time quantum）。当线程的时间片耗尽时，会被操作系统置入睡眠状态，同时处理器将切换到另一个线程（这个过程称之为上下文切换）。在多处理器机器上，两个或者多个线程可以真实地并发运行（每个物理处理器上运行一个线程）。

通过抢占式执行方式，系统中的活跃线程都必须能够在任意时刻将处理器的控制权交给另一个线程。由于操作系统可以取走一个线程的控制权，因此开发人员必须考虑当控制权被取走时线程的状态。

如果所有应用程序都是单线程的或者孤立运行的，那么不会带来任何同步问题。然而，为了提升程序的执行效率，相互依赖的多线程已成为当前的一种标准编程方式，同时也是程序中许多错误的根源。当两个或多个线程需要通过协作来完成某个任务时，就会出现相互依赖的多线程问题。例如，在执行某个任务时，代码可能会被分解到一个或多个线程（这些线程可能共享资源也可能不共享）中，因此线程需要根据线程执行的顺序与其他线程进行“通信”。这种通信称之为线程同步，它对于任何一个多线程程序来说都是至关重要的。

6.2 线程同步原语

Windows 操作系统在内部使用线程执行块（Thread Execution Block，TEB）这种数据结构来表示一个线程。该数据结构包含了线程的各种属性，例如线程标识、最近发生的错误、局部存储等。清单 6-1 给出了 TEB 数据结构中各个成员的简化输出。

清单 6-1　TEB 数据结构的简化输出

```
0:000> dt _TEB
ntdll!_TEB
   +0x000 NtTib            : _NT_TIB
```

```
+0x01c EnvironmentPointer : Ptr32 Void
+0x020 ClientId           : _CLIENT_ID
+0x028 ActiveRpcHandle    : Ptr32 Void
+0x02c ThreadLocalStoragePointer : Ptr32 Void
+0x030 ProcessEnvironmentBlock : Ptr32 _PEB
+0x034 LastErrorValue     : Uint4B
+0x038 CountOfOwnedCriticalSections : Uint4B
...
...
...
+0xfca RtlExceptionAttached : Pos 9, 1 Bit
+0xfca SpareSameTebBits : Pos 10, 6 Bits
+0xfcc TxnScopeEnterCallback : Ptr32 Void
+0xfd0 TxnScopeExitCallback : Ptr32 Void
+0xfd4 TxnScopeContext    : Ptr32 Void
+0xfd8 LockCount          : Uint4B
+0xfdc ProcessRundown     : Uint4B
+0xfe0 LastSwitchTime     : Uint8B
+0xfe8 TotalSwitchOutTime : Uint8B
+0xff0 WaitReasonBitMap : _LARGE_INTEGER
```

总的来说，在 Windows Vista 机器上，TEB 数据结构包含了大约 98 个不同的成员。尽管大多数成员通常在调试. NET 同步问题时不会被用到，但要清楚的是，Windows 为了精确地调度线程执行，需要了解关于线程的大量信息。Windows 通过线程数据结构来维护线程状态，CLR 也采用了类似的方式。CLR 中的线程数据结构称之为 Thread。虽然 Thread 类的内部信息并没有公开，但我们可以使用在第 3 章中介绍的 SOS 调试器扩展命令来获得 CLR 线程的内部信息。一个非常有用的命令就是 threads，它能输出当前进程中所有 CLR 线程的汇总信息以及每个线程的单独状态：

```
0:003> !threads
ThreadCount: 2
UnstartedThread: 0
BackgroundThread: 1
PendingThread: 0
DeadThread: 0
Hosted Runtime: no
                                      PreEmptive   GC Alloc           Lock
       ID OSID ThreadOBJ    State     GC       Context       Domain   Count APT
Exception
   0    1 108c 00191ec8      a020 Enabled  00000000:00000000 0015d080     0 MTA
   2    2  990 0019b900      b220 Enabled  00000000:00000000 0015d080     0 MTA
(Finalizer)
```

在第 3 章中给出了这个命令的详细输出。虽然 threads 命令能够给出线程在 CLR 中的某些信息（例如线程状态、CLR 线程 ID、操作系统线程 ID 等），但 CLR 却包含了更为丰富的线程信息。尽管线程在 CLR 中的内部没有被公开，但我们仍然可以通过 Rotor 源代码来获得

Thread 类的某些内部信息。相关的 Rotor 源文件为 threads. h 和 threads. cpp，分别位于文件夹 sscli20 \ clr \ src \ vm 之下。清单 6-2 给出了 Thread 类中的一些数据成员。

清单 6-2 CLR Thread 类的简化版本

```
class Thread
{
    ...
    ...
    ...
    volatile ThreadState m_State;

    DWORD m_dwLockCount;
    DWORD m_ThreadId;
    LockEntry *m_pHead;
    LockEntry m_embeddedEntry;
    ...
    ...
    ...
}
```

m_State 成员包含了线程的状态（例如活跃、废弃等）。m_dwLockCount 表示这个线程当前持有的锁的数量。成员 m_ThreadId 对应于托管线程的 ID，后两个成员（m_pHead, m_embeddedEntry）对应于线程的读/写锁的状态。如果要进一步观察 CLR 线程（包括上述成员），那么必须首先找出一个指向 Thread 实例的指针。这很容易做到，首先使用 threads 命令，并查看 ThreadObj 列，这列对应的就是 Thread 实例：

```
0:000> !threads
ThreadCount: 2
UnstartedThread: 0
BackgroundThread: 2
PendingThread: 0
DeadThread: 0
Hosted Runtime: no
                                        PreEmptive   GC Alloc          Lock
       ID OSID ThreadOBJ    State      GC        Context        Domain
   Count APT Exception
   0     1 47c8 003b4528       220 Enabled  01ebb070:01ebbfe8 003afe20     0 STA
XXXX     2 1120 003c3b28      b220 Enabled  00000000:00000000 003afe20
     0 STA (Finalizer)
0:000> dd 003b4528
003b4528  79f96af0 00000220 00000000 ffffffff
003b4538  00000000 00000000 00000000 00000001
003b4548  00000000 003b4550 003b4550 003b4550
003b4558  00000000 00000000 baad0000 003b1120
003b4568  01ebb070 01ebbfe8 0001afd0 00000000
003b4578  00000000 00000000 00000000 baadf00d
```

```
003b4588  003b4be8 003c90b0 003c91d0 000000e8
003b4598  003c90b0 003e3248 00000000 00000100
```

我们可以看到，threads 命令给出的第一个线程指针位于地址 0x003b4528 处。然后，我们可以使用 dd 命令来转储出这个指针的内容。如果要找出 m_State 成员的内容，那么该怎么办？要实现这个功能，首先要找出这个成员在对象内存布局中的偏移，可以使用两种不同的方法。第一种方法是查看类的定义，以及是否已经知道某些成员的偏移值。如果情况如此，那么可以将这个对象的内容转储出来，找出已知的成员，再通过相对偏移找到所要查找的目标成员。另一种方法是观察类中的所有成员，并把目标成员前面的所有成员的大小相加来作为目标成员的偏移。我们采用第二种方法来找出 m_State 成员。通过观察类的定义，我们可以看到 m_State 成员事实上是这个类的第一个成员。然后，在转储出这个线程指针的内容时，第一个域就是线程的状态：

```
0:000> dd 003b4528
003b4528  79f96af0 00000220 00000000 ffffffff
003b4538  00000000 00000000 00000000 00000001
003b4548  00000000 003b4550 003b4550 003b4550
003b4558  00000000 00000000 baad0000 003b1120
003b4568  01ebb070 01ebbfe8 0001afd0 00000000
003b4578  00000000 00000000 00000000 baadf00d
003b4588  003b4be8 003c90b0 003c91d0 000000e8
003b4598  003c90b0 003e3248 00000000 00000100
```

需要注意的是，第一个值（0x79f96af0）看上去并不像线程的状态。事实上，如果使用 ln（list near）命令，可以看到以下输出：

```
0:000> ln 79f96af0
(79f96af0)   mscorwks!Thread::'vftable'   |  (79f96b28)   mscorwks!Thread::Thread
Exact matches:
mscorwks!Thread::'vftable' = <no type information>
```

我们看到的是对象的虚函数表指针。虽然从调试的角度来看，这个指针并不需要太多的注意，但它却很清楚地告诉我们，它是一个指向有效线程对象的指针。就目前而言，我们可以暂时忽略这个指针，下一个值是 0x00000220。这个值很像某种位掩码。要按照线程状态来解释它，我们必须首先了解构成线程状态的各个位元（Bit）。在 Thread 类中包含了一个枚举类型 ThreadState，表示线程的状态。在调试同步问题时，这个枚举类型能够提供一些重要的线索。虽然在这个枚举类型中共包含了大约 100 个值，但在调试同步问题时，有些枚举值会更为重要。表 6-1 列出了 ThreadState 枚举类型中最值得注意的一些值。

表 6-1　ThreadState 枚举类型

域的名字	值	描　述
TS_Unknown	0x00000000	新近初始化线程的状态
TS_AbortRequested	0x00000001	请求了一个线程中止操作

（续）

域的名字	值	描 述
TS_GCSuspendPending	0x00000002	表示 GC 要求这个线程被挂起，并且这个线程正在努力找到一个安全的位置来挂起它自己
TS_LegalToJoin	0x00000020	线程处于可进入合并(Join)的状态
TS_Background	0x00000200	线程是一个后台线程
TS_Unstarted	0x00000400	线程从未启动过
TS_Dead	0x00000800	线程死亡了，意味着底层的操作系统线程已经消失了，但表示这个线程的托管数据结构仍然驻留在内存中
TS_ThreadPoolThread	0x00800000	该线程是线程池中的一个线程
TS_AbortInitiated	0x10000000	表示线程中止操作已经启动
TS_Finalized	0x20000000	底层的托管线程对象已经被终结了，并且可以被回收
TS_FailStarted	0x40000000	线程启动失败

根据表 6-1 以及之前得到的状态 0x00000220，可以推断出以下信息：

- 这个线程是一个后台线程（0x00000200）。
- 线程处于可以合并（Join）的状态（0x00000020）。
- 这个线程是一个刚被初始化的线程（0x00000000）。

Thread 类的内部结构

虽然观察线程的“内部信息”是很有用的，但重要的是要了解这些信息被保留在内部而没有通过 threads 命令暴露出来的原因。大多数信息都是一种实现细节，并且 Microsoft 可能在任意时刻修改它们。依赖这些内部信息是一种危险的行为，我们要尽力避免这种情况。其次，Rotor 只是一种参考实现，它并不能保证这些内部信息能准确地模拟 CLR 源代码。

现在，我们已经介绍了 CLR 如何在内部表示一个线程，接下来将介绍 CLR 公开的一些最常见的同步原语，以及如何在 CLR 中表示它们。

6.2.1 事件

事件是一种内核态的原语，可以在用户态中通过句柄来访问。事件也是一个同步对象，它可以处于两种状态之一：已触发（signaled）或未触发（nonsignaled）。当事件从未触发状态切换到已触发状态（表示发生了某个特定的事件）时，在这个事件上等待的某个线程会被唤醒并恢复执行。事件对象经常用于对多个线程之间的代码执行流程进行同步。例如，非托管的 Win32 API ReadFile 可以通过传递一个 OVERLAPPED 结构的指针并以异步方式读取数据。在图 6-1 中说明了这些事件的流程。

ReadFile 将立即返回到调用者，并在后台进行读取操作。然后，主调线程可以转而执行其他的工作。在调用者准备好接收读取操作的结果后，它把（通过 WaitForSignalObject）等

待事件的状态变成已触发。当后台读取操作成功时，这个事件将被设置为已触发状态，因此将唤醒主调线程，并使程序继续执行。

共有两种类型的事件对象：手动重置事件和自动重置事件。这两种类型之间的关键差异在于事件被触发时发生的操作。在手动重置事件中，事件对象保持为已触发状态，直到被手动重置，因此所有在这个事件对象上等待的线程都会被释放。与之相反的是，自动重置事件只允许其中一个等待线程被释放，然后又立即自动地回到未触发状态。如果没有任何等待的线程，那么这个事件将保持为未触发状态，直到第一个线程在这个事件上开始等待。在.NET框架中，手动重置事件对应于 System. Threading. ManualResetEvent 类，而自动重置事件对应于 System. Threading. AutoResetEvent 类。

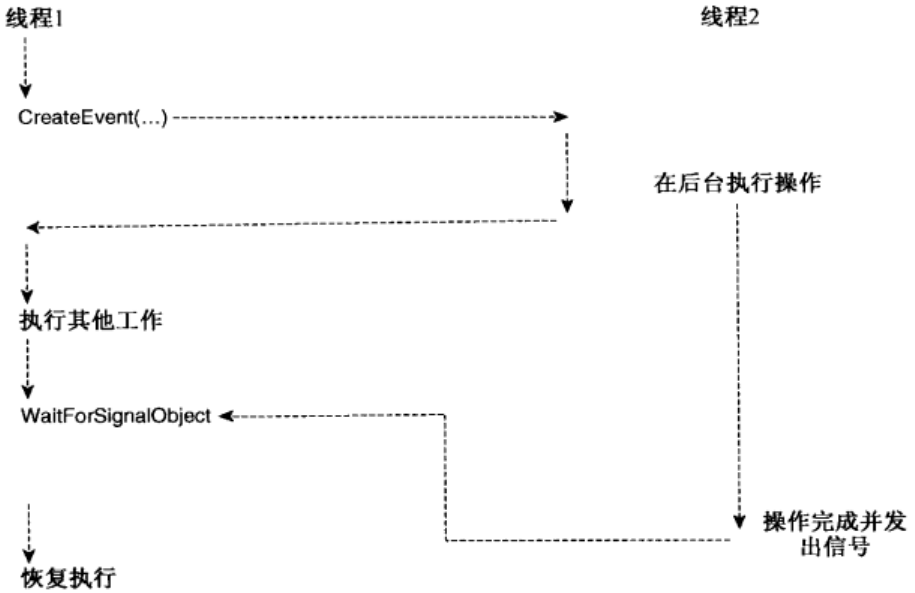


图 6-1　异步 API 的流程

要进一步观察这两种事件类的实例，可以使用 do 命令，如下所示：

```
0:000> !do 0x01c9ad4c
Name: System.Threading.AutoResetEvent
MethodTable: 791124e4
EEClass: 791f5ff0
Size: 24(0x18) bytes
 (C:\Windows\assembly\GAC_32\mscorlib\2.0.0.0__b77a5c561934e089\mscorlib.dll)
Fields:
      MT    Field   Offset                 Type VT     Attr    Value Name
790fd0f0  400018a        4       System.Object   0 instance 00000000 __identity
791016bc  40005b3        c      System.IntPtr   1 instance      204 waitHandle
79112728  40005b4        8 ...es.SafeWaitHandle  0 instance 01c9ad64 safeWaitHandle
7910be50  40005b5       10       System.Boolean  1 instance        0
 hasThreadAffinity
791016bc  40005b6      994       System.IntPtr   1   shared   static InvalidHandle
```

```
0:000> !do 0x01c9ad78
Name: System.Threading.ManualResetEvent
MethodTable: 7911a1e8
EEClass: 7911a170
Size: 24(0x18) bytes
 (C:\Windows\assembly\GAC_32\mscorlib\2.0.0.0__b77a5c561934e089\mscorlib.dll)
Fields:
      MT    Field   Offset                 Type VT     Attr    Value Name
790fd0f0  400018a        4       System.Object   0 instance 00000000 __identity
791016bc  40005b3        c       System.IntPtr   1 instance      208 waitHandle
79112728  40005b4        8 ...es.SafeWaitHandle   0 instance 01c9ad90 safeWaitHandle
7910be50  40005b5       10       System.Boolean   1 instance        0
 hasThreadAffinity
791016bc  40005b6      994        System.IntPtr   1 shared   static InvalidHandle
    >> Domain:Value  0016ff10:ffffffff <<
```

由于在 System. Threading 命名空间中的 Event 类只是对底层 Windows 内核对象进行了简单地封装，因此可以通过这些类的 waitHandle 成员来进一步了解底层内核态对象的更多信息。我们可以使用调试命令 handle，并将 waitHandle 值作为命令参数：

```
0:000> !handle 204 8
Handle 204
  Object Specific Information
    Event Type Auto Reset
    Event is Waiting
```

在这里，我们可以看到 waitHandle 的值为 204，对应于一个自动重置事件，并且该事件当前处于等待状态。

6.2.2 互斥体

互斥体（Mutex）是一个内核态的同步结构，既可以用于对某个进程内的线程进行同步，也可以在多个进程之间进行同步（通过在创建互斥体时指定名字）。通常来说，如果所有同步操作都位于同一个进程中，那么应该使用监视器对象或者其他的用户态同步原语。而另一方面，如果需要在多个进程之间进行同步，那么合适的方法是使用一个命名互斥体。由于互斥体是一种内核态结构，因此用户态代码需要通过 System. Threading. Mutex 类来访问互斥体。当在用户态中进行调试时，如果要获取关于互斥体的更多信息，可以使用扩展命令 dumpobj 来观察互斥体的各个域。参观以下代码：

```
0:000> !do 0x01eb58bc
Name: System.Threading.Mutex
MethodTable: 79117d00
EEClass: 791f94d8
Size: 24(0x18) bytes
 (C:\Windows\assembly\GAC_32\mscorlib\2.0.0.0__b77a5c561934e089\mscorlib.dll)
Fields:
      MT    Field   Offset                 Type VT     Attr    Value Name
```

```
790fd0f0  400018a        4       System.Object  0 instance 00000000
 __identity
791016bc 40005b3         c       System.IntPtr  1 instance      1fc
 waitHandle
79112728  40005b4        8 ...es.SafeWaitHandle  0 instance 01eb5958
 safeWaitHandle
7910be50  40005b5       10       System.Boolean  1 instance        1
hasThreadAffinity

791016bc  40005b6      994       System.IntPtr   1   shared   static InvalidHandle
    >> Domain:Value  003bfdc0:ffffffff <<
7910be50  40005fd      9a0      System.Boolean   1   shared   static dummyBool
    >> Domain:Value  003bfdc0:NotInit  <<
```

Mutex 类的结构非常类似于之前讨论的两个事件类。与事件类相同的是，在 Mutex 类中也包含了一个 waitHandle。这个 waitHandle 可以与 handle 命令放在一起使用来获得互斥体的更详细信息：

```
0:000> !handle 1fc 8
Handle 1fc
  Object Specific Information
    Mutex is Free
```

6.2.3 信号量

信号量（semaphore）是一种内核态同步对象，可以在用户态中访问。它类似于互斥体，因为它可以实现对资源的互斥访问。然而，二者之间的主要差异在于，信号量采用了资源计数，因此可以同时允许 X 个线程访问这个资源。信号量的一个应用示例是，假定系统中有 4 个 USB 端口，这些端口由一段代码来访问。由于共有 4 个 USB 端口，因此我们希望有 4 个线程并发地使用这些 USB 端口。要实现这个功能，首先需要创建一个信号量，并且将它的最大引用计数值设为4。当线程要获取这个信号量时，首先判断引用计数（初始化为4）是否大于0，如果大于0，那么将允许线程执行获取操作，并且递减引用计数值。当引用计数值为0时，线程在尝试获取信号量时会被阻塞并进入休眠状态，直到有一个线程释放这个信号量并且递增引用计数。与事件和互斥体一样，你可以使用命令 do，然后再使用调试器中的扩展命令 handle 来获得号量的扩展信息。参见以下代码：

```
0:000> !do 0x01d159ac
Name: System.Threading.Semaphore
MethodTable: 7a76397c
EEClass: 7a763904
Size: 24(0x18) bytes
 (C:\Windows\assembly\GAC_MSIL\System\2.0.0.0__b77a5c561934e089\System.dll)
Fields:
      MT    Field   Offset                 Type VT     Attr    Value Name
790fd0f0  400018a        4       System.Object   0 instance 00000000 __identity
791016bc 40005b3         c       System.IntPtr   1 instance      238 waitHandle
79112728  40005b4        8 ...es.SafeWaitHandle  0 instance 01d159c4 safeWaitHandle
```

```
7910be50  40005b5       10       System.Boolean  1 instance        0
 hasThreadAffinity
791016bc  40005b6      994       System.IntPtr   1   shared   static InvalidHandle
    >> Domain:Value  0018fdc0:ffffffff <<
79102290  4001066      8c8       System.Int32    1   static        0 MAX_PATH
0:000> !handle 238
Handle 238
  Type          Semaphore
0:000> !handle 238 8
Handle 238
  Object Specific Information
    Semaphore Count 0
    Semaphore Limit 3
```

6.2.4 监视器

监视器对象是一种对某个对象的访问操作进行监视的结构，它能在对象上创建一个锁，因而只有当持有该监视器对象的线程离开（或者解锁）监视器对象之后，其他线程才能进行访问。与同步原语不同的是，监视器并不是对内核 Windows 同步原语进行的简单封装，而是.NET 中定义的类，即 System.Threading.Monitor。Monitor 类不能被实例化，而是包含了一组静态方法，用于获取一个锁。最常用的两个方法就是 Enter 和 Exit。Enter 方法能获得指定对象上的互斥锁（假设这个对象还没有被锁定），而 Exit 方法能释放指定对象上的互斥锁。例如，下面这段代码是在对象 db1 上的锁定和解锁操作：

```
Monitor.Enter(db1);
//执行一些工作…
Monitor.Exit(db1);
```

在 C#中有一个更方便的 lock 语句，可以用于获取和释放互斥锁。虽然 lock 语句看上去与监视器没有任何关系，但实际上，它却是调用 Monitor.Enter 和 Monitor.Exit 等方法的一种快捷方式。这一点从 lock 语句生成的中间语言（IL）可以很容易看出来。在图 6-2 中给出了由以下代码声明的 IL：

```
lock (someObj)
{
    Console.WriteLine("I'm locked now");
}
```

在 IL 中可以看到，lock 语句自动进入一个监视器（如方框中的语句所示），并将保护区域内的代码封装在一个 try/finally 中，以确保监视器在作用域的结束位置被释放。

由于 Monitor 类是一个不能被实例化的对象，因此在调试器中无法看到它的任何状态。那么哪些锁是可用的，以及每个锁的状态又是什么？答案是，在每个被锁定的对象中记录了必要的信息来维持锁的完整性，这些信息将作为对象内存布局的一部分。在本章的 6.3 一节中将看到这种记录方式的详细信息。

```
IL_001e:  dup
IL_001f:  stloc.s    CS$2$0000
IL_0021:  call       void [mscorlib]System.Threading.Monitor::Enter(object)
IL_0026:  nop
.try
{
  IL_0027:  nop
  IL_0028:  ldstr      "I'm locked now"
  IL_002d:  call       void [mscorlib]System.Console::WriteLine(string)
  IL_0032:  nop
  IL_0033:  nop
  IL_0034:  leave.s    IL_003f
}  // end .try
finally
{
  IL_0036:  ldloc.s    CS$2$0000
  IL_0038:  call       void [mscorlib]System.Threading.Monitor::Exit(object)
  IL_003d:  nop
  IL_003e:  endfinally
}  // end handler
```

图 6-2　由 lock 语句生成的 IL

6.2.5　读写锁

Monitor 类每次允许一个线程独占地访问一个对象。虽然在写入操作非常频繁的情况下，Monitor 能够工作得很好，但当读取操作多于写入操作或者在锁上存在着高度竞争的情况下，Monitor 的性能将受到严重影响。为了解决这个性能问题，我们引入了读写锁，即 ReaderWriterLock 类。ReaderWriterLock 能够使多个线程并发地执行读操作，而每次只允许一个线程执行写操作。与 Monitor 类不同的是，ReaderWriterLock 类本身包含了状态来控制对锁的访问：

```
0:000> !do 0x01e86cdc
Name: System.Threading.ReaderWriterLock
MethodTable: 79108ba4
EEClass: 79108b40
Size: 44(0x2c) bytes
 (C:\Windows\assembly\GAC_32\mscorlib\2.0.0.0__b77a5c561934e089\mscorlib.dll)
Fields:
      MT    Field   Offset                 Type VT     Attr    Value Name
791016bc  4000626        4        System.IntPtr  1 instance        0 _hWriterEvent
791016bc  4000627        8        System.IntPtr  1 instance        0 _hReaderEvent
791016bc  4000628        c        System.IntPtr  1 instance        0 _hObjectHandle
79102290  4000629       10         System.Int32  1 instance        0 _dwState
79102290  400062a       14         System.Int32  1 instance        0 _dwULockID
79102290  400062b       18         System.Int32  1 instance        1 _dwLLockID
79102290  400062c       1c         System.Int32  1 instance        0 _dwWriterID
79102290  400062d       20         System.Int32  1 instance        1
_dwWriterSeqNum
7910480c  400062e       24         System.Int16  1 instance        0 _wWriterLevel
```

_hWriterEvent 和_hReaderEvent 这两个成员都是事件句柄，分别用来控制对读取队列和写入队列的访问。这些句柄都是普通的 Windows 事件。_dwState 域表示锁的各种不同内部状态（例如 reader、writer、waiting reader 和 waiting writer 等）。_dwULockID 和_dwLLockID 这两个域是持有锁的线程的内部标识。_dwWriterID 是持有锁的线程的 ID，而_wWriterLevel 则是持有写入线程的递归锁计数（recursive lock count）。

ReaderWriterLockSlim

在对关键代码的访问进行同步时，读写锁是一种非常好的机制。然而，与监视器相比，读写锁的性能较为糟糕。为了解决这个问题，CLR 引入了另一种读写锁，称之为轻量级读写锁，即 ReaderWriterLockSlim 类。除了提升读写锁的性能外，轻量级读写锁还极大地简化了在递归获取锁和升级/降低锁状态时的规则。CLR 强烈推荐开发人员使用这个更为高效的锁。

6.2.6 线程池

创建新线程的常用方式之一是使用 System. Threading. Thread 类，并且指定一个方法或者委托在线程启动后执行。对于简单情况来说，这种方法工作得不错，但当需要处理非常多的请求时，一种更好的方法是使用线程池（thread pool），并且由运行时来高效地管理这个线程池。System. Threading. ThreadPool 类为开发人员提供了这个功能。每个进程有且只有一个线程池，默认分配的线程数量为 250（1 000 个 I/O 完成线程）。在线程池中可用线程数量的最小值等于系统上处理器的数量。开发人员可以通过 ThreadPool 类中的 QueueUserWorkItem 方法将一个任务放入线程池中。需要注意的一点是，当线程被交还给线程池时，在线程上设置的任何状态都会被保留下来。如果同一个线程被用于服务另一个任务请求，并且该任务请求与线程状态不兼容，那么程序很可能会失败。

要在调试过程中找出线程池的状态，可以使用 SOS 调试器扩展命令 threadpool：

```
0:014> !threadpool
CPU utilization 10%
Worker Thread: Total: 9 Running: 9 Idle: 0 MaxLimit: 500 MinLimit: 2
Work Request in Queue: 92
--------------------------------
Number of Timers: 0
--------------------------------
Completion Port Thread:Total: 0 Free: 0 MaxFree: 4 CurrentLimit: 0
MaxLimit: 1000 MinLimit: 2
```

threadpool 命令输出的第一部分是工作线程的统计信息，即当前 CPU 的使用率（10%）、总工作线程数量（9）、正在运行的工作线程数量（9）以及空闲工作线程的数量（0）。此外，它还给出了线程池的最大线程数和最小线程数（500 和 2）。接下来的一行是队列中正在等待被服务的请求数量（92）。如果程序正在使用线程池来维持计时器，那么还会给出计时器的总数（0）。在输出的最后一部分给出了完成端口线程上的数据。

6.3 同步的内部细节

前面提到过，像监视器这样的线程同步原语需要包含一定数量的记录信息，包括判断这个监视器是否被锁定、持有这个监视器对象的线程、等待中的线程等。在本章的这部分内容中，我们将讨论 CLR 如何管理锁定信息。

6.3.1　对象头

第2章曾简要提到过，在托管堆中的每个对象都有一个相应的对象头，在对象头中包含了与对象状态相关的一组信息。在对象头中保存的信息包括散列码、锁定信息、同步块索引等。在图6-3中给出了一个托管堆的示例，其中每个对象都包含一个相应的对象头。

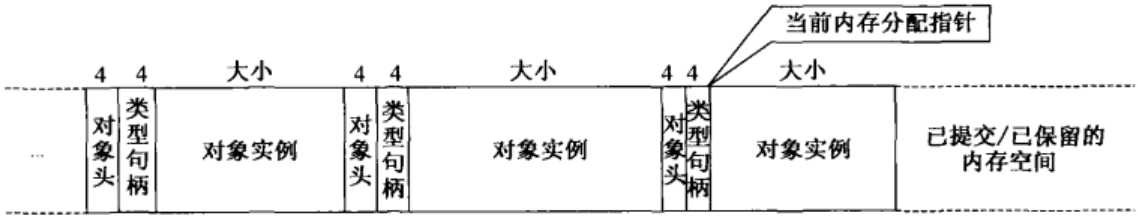


图6-3　托管堆以及对象内存布局的示例

在对象中需要保存的*所有*信息总量大于对象头本身的大小。这句话的意思是，任何一个对象都可能需要（也可能不需要）所有的信息，这取决于具体的执行流程。只要在执行操作中需要的信息（例如，只是获得对象的散列码）不超过对象头的大小，那么这些信息就会保存在对象头中。那么，当对象头中已经放满了信息时，还要增加额外的信息，会出现怎样的情况？在这种情况中，CLR会执行对象头膨胀操作（header infaltion），创建一个独立的同步块（sync block）数据结构，并将当前保存在对象头中的所有信息都复制到这个同步块中。同步块位于非GC的内存中，通过同步块表中的索引来访问。为了使对象能够找到与对象相关的同步块，现在对象头中保存的信息会被替换为同步块在同步块表中的索引。在图6-4中说明了对象头和同步块的概念。

在图6-4的步骤1中给出了一个对象，它包含一个对象头，值为0x0f78734a。在步骤2中，另一部分信息也需要保存到对象头中，但已经放不下了。因此，CLR将创建一个同步块，位于同步块表中索引为1的位置上。然后，对象头的值被设置为0x08000001，表示对象头现在包含的是一个同步块索引，值为1。CLR如何判断在对象头中保存的信息是一个同步块索引还是其他信息？答案在于对象头中位元的组织方式。如果在对象头中设置了掩码0x08000000，那么对象头的剩余部分包含的要么是一个散列码，要么是一个同步块索引。我们可以进一步判断是否同时设置了掩码0x04000000，这个掩码表示在对象头中包含的是一个散列码。如果没有设置这个掩码，那么在对象头中包含的就是一个同步块索引。在前面已经提到过，对象头中可以保存大量的不同信息。要深入了解各种不同类型的信息，请参见以下位置中的Rotor源代码：

```
sscli20\clr\src\vm\syncblk.h
```

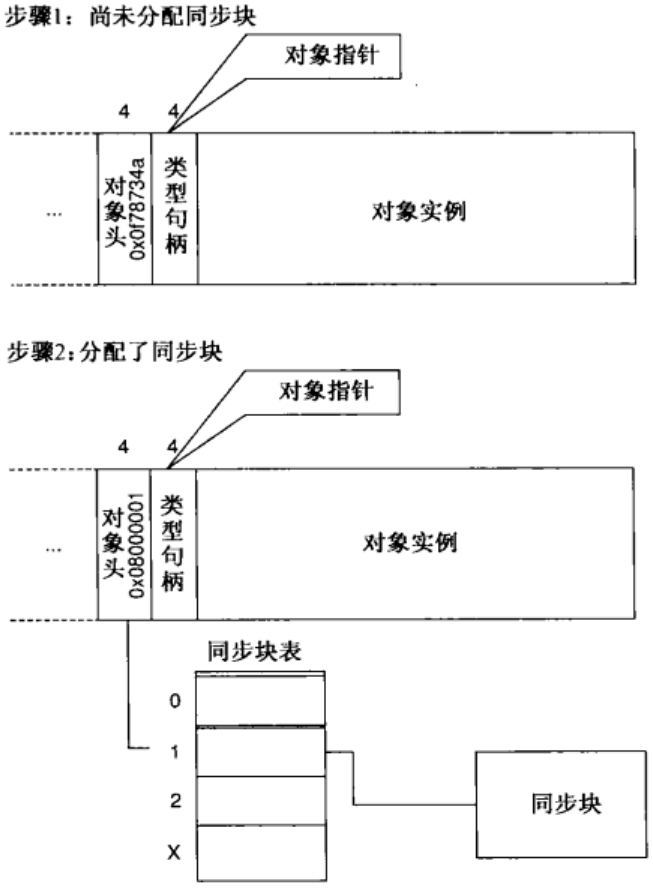


图 6-4 对象头和同步块

6.3.2 同步块

清单 6-3 给出了同步块的重要性以及它是如何与锁关联在一起的。

清单 6-3 一个已被获取的锁

```
using System;
using System.Text;
using System.Threading;

namespace Advanced.NET.Debugging.Chapter6
{
    class Simple
    {
        static void Main(string[] args)
        {
            Simple s = new Simple();
            s.Run();
        }
```

```
        public void Run()
        {
            this.GetHashCode();

            Console.WriteLine("Press any key to acquire lock");
            Console.ReadLine();
            Monitor.Enter(this);
            Console.WriteLine("Press any key to release lock");
            Console.ReadLine();
            Monitor.Exit(this);
            Console.WriteLine("Press any key to exit");
            Console.ReadLine();
        }
    }
}
```

清单 6-3 中的源代码和二进制文件位于以下文件夹中：

- 源代码文件：C:\ADND\Chapter6\Simple
- 二进制文件：C:\ADNDBIN\06Simple.exe

清单 6-3 中的示例代码非常简单。它只是在 Run 方法开头获取了 this 对象上的一个锁，并在这个方法的末尾处释放这个锁。暂时可以忽略 Run 方法起始位置的 GetHashCode，我们将在后面讨论这条语句的重要性。接下来，在调试器下运行这个程序，并在出现提示信息“Press any key to acquire lock”时中断程序的执行。此时，我们来观察 Simple 对象的同步块，并且验证没有任何锁被获取。由于同步块数据结构中的大部分内容还没有被公开，因此我们必须使用 syncblk 命令来获取更详细的信息。在执行 syncblk 命令时可以不带任何参数，这表示它将输出某个线程中所有对象的同步块；此外，也可以将某个同步块索引作为命令的参数。如何找出某个对象的同步块索引？请记住，对象指针指向的是类型句柄域，然后才是实际的对象数据。在类型句柄前的 4 个字节同样是对象布局的一部分，其中包含了对象的对象头。我们可以通过命令 dd 来转储出对象头索引的内容，只需将对象指针减去 4 个字节即可。例如，在调试会话中，要得到同步块索引，可以执行以下命令序列：

```
0:000> !ClrStack -a
OS Thread Id: 0x462c (0)
...
...
...
0018f080 793a3350 System.IO.StreamReader.ReadBuffer()
    PARAMETERS:
        this = 0x01e36cb8
    LOCALS:
        <no data>

0018f090 793aaa2f System.IO.StreamReader.ReadLine()
    PARAMETERS:
        this = 0x01e36cb8
```

```
    LOCALS:
        <no data>
        <no data>
        <no data>
        <no data>

0018f0a4 79497b5a System.IO.TextReader+SyncTextReader.ReadLine()
    PARAMETERS:
        this = 0x01e37028

0018f0ac 793e99f0 System.Console.ReadLine()
0018f0b0 0093011b Advanced.NET.Debugging.Chapter6.Simple.Run()
    PARAMETERS:
        this = 0x01e358a0

0018f0b8 009300a7 Advanced.NET.Debugging.Chapter6.Simple.Main(System.String[])
    PARAMETERS:
        args = 0x01e35890
    LOCALS:
        <CLR reg> = 0x01e358a0

0018f2e0 79e7c74b [GCFrame: 0018f2e0]
0:000> !do 0x01e358a0
Name: Advanced.NET.Debugging.Chapter6.Simple
MethodTable: 002e3038
EEClass: 002e1210
Size: 12(0xc) bytes
 (C:\ADNDBin\06Simple.exe)
Fields:
None
0:000> dd 0x01e358a0-0x4 l1
01e3589c  0f78734a
```

这里的命令序列可分为三个步骤：

1）通过 ClrStack-a 命令转储出这个线程的所有调用栈及其所有的参数。最底层的栈帧对应于 Main 方法，在这个方法中包含了一个局部变量，其中包含了对 Simple 类实例的引用。

2）为了确保正在操作的对象（类型为 Simple）是正确的，使用了 do 命令。

3）最后，使用 dd 命令来转储出对象头，它位于对象指针减去 4 个字节的位置上。

对象头的当前值（0x0f78734a）表示在对象头中包含的是对象的散列码（并不是一个同步块索引）。需要理解的是，我们还没有获取这个对象上的锁，因此还没有创建同步块。恢复程序的执行，直到看到提示信息“Press any key to release lock”，此时再次中断进入到调试器。这时，可以使用上述的 3 个步骤从 ClrStack 中获得指针，使用 do 来确保正在使用正确的对象，最后使用 dd 命令来转储出同步块索引。为什么要再次执行前两个步骤？对象指针难道不是保持不变么？原因是垃圾收集器可以在任意时刻移动这个对象，因此对象的位置可能会发生变化。为了简单起见，在下面清单的输出中只给出了同步块步骤：

```
0:000> dd 0x01e358a0-0x4 l1
01e3589c  08000001
```

这时，同步块索引看上去更加合理了（0x08000001）。根据前面对对象头的讨论，我们知道位掩码0x0800000表示正在处理的是一个同步块索引。如果使用syncblk命令，并将参数指定为0x1，那么可以看到：

```
0:000> !syncblk 0x1
Index SyncBlock MonitorHeld Recursion Owning Thread Info  SyncBlock Owner
    1 00322fd4            1         1 003044f0  462c   0   01e358a0
Advanced.NET.Debugging.Chapter6.Simple
-----------------------------------
Total           1
CCW             0
RCW             0
ComClassFactory 0
Free            0
```

在syncblk命令的输出中包含两部分信息。第一部分是持有锁的相关信息。在表6-2中给出了各列的含义。

表6-2　syncblk命令输出中各列的描述

列	描　述
Index	同步块索引
SyncBlock	同步块数据结构的地址（未公开）
MonitorHeld	持有的监视器的数量
Recursion	同一个线程获取这个锁的次数
Owning thread info	第一个数据项是指向内部线程数据结构的指针，第二个数据项是操作系统线程ID，第三个数据项是调试器线程ID
SyncBlock Owner	第一个数据项是指向持有锁的对象的指针，第二个数据项是锁所在的对象的类型

其他部分与锁并不直接相关，包括同步块表中同步块的总数量，运行时可调用包装（RCW）的数量以及COM可调用包装（CCW）的数量。最后，Free行表示在同步块表中有多少个同步块。

6.3.3　瘦锁

在处理一些常见的同步问题时，例如死锁，如果能了解在获取某个对象上的锁时发生的内部情况，那么会非常有用。syncblk命令可以快速找出哪个线程正在持有哪个锁，从而帮助我们判断死锁的来源。在介绍一些最常见的同步问题以及如何对它们进行调试之前，首先来解释清单6-3中的代码行：

```
this.GetHashCode();
```

将上面这条语句添加到示例代码中的原因是为了确保创建一个同步块。在前面提到过，可以通过对象头来管理对象上已经获取的锁。然而，这并不是对象头的唯一用途，它还可以

用来保存其他与对象相关的信息（例如，散列码，COM 互用性数据以及应用程序域索引等）。如果这个对象没有被同步（即没有获取任何锁），并且程序没有执行某个需要使用对象头的操作，那么在对象头中就不存在同步块。在前面的示例中，我们实际上是通过调用 GetHashCode 并强制创建一个同步块来说明 CLR 是如何管理同步块的。那么，能否在没有同步块的情况下对锁进行管理？当然可以，在 CLR 2.0 中引入了瘦锁，它实现了一种更为高效的机制来管理锁。在使用瘦锁时，保存在对象头中的唯一信息就是获取锁的线程 ID（即没有同步块）。根据这个信息，我们还可以推断出瘦锁只是一个自旋锁（spinning lock），因为如果要实现一个高效等待的锁，那么需要更多的信息。然而，这个瘦锁并不会无限地循环，而是当自旋到某个阈值时停止。如果在超过了阈值后还不能获取这个锁，那么接下来会创建一个实际的同步块，并将相应的信息保存下来来实现高效的等待（例如一个事件）。

CLR 通常采用以下算法来判断在什么情况下使用同步块和瘦锁：

- 如果同步块已经存在，那么使用同步块来存储锁信息。
- 如果同步块不存在，判断在当前对象头中是否可以容纳一个瘦锁：

如果可以容纳，那么将线程 ID 保存在对象头中。如果后面需要保存更多的信息，那么将自动创建一个同步块，并把当前对象头中的内容移动到新的同步块中。

如果不能容纳。那么创建一个新的同步块，将当前对象头的内容移动到新的同步块中，并保存锁。

验证这个算法很容易，只需将示例代码中的 GetHashCode 的调用移动到第一个获取锁的操作之后即可。为了简单起见，这里没有给出相应的源代码，它们位于以下位置：

- 源代码文件：C:\ADND\Chapter6\Simple2
- 二进制文件：C:\ADNDBIN\06Simple2.exe

在调试器下运行这个程序，并且通过以下步骤来验证前面的假设：

1）在获取锁之前，将同步块转储出来，并验证它为空。

2）获取这个锁，中断程序执行，并验证已经创建了一个瘦锁。

3）获得散列码，中断程序执行，并验证这个瘦锁已经被一个同步块替代。

第 1 步如下所示：

```
0:000> !do 0x01e1ad1c
Name: Advanced.NET.Debugging.Chapter6.Simple
MethodTable: 000d3188
EEClass: 000d1298

Size: 12(0xc) bytes
 (C:\ADNDBin\06Simple2.exe)
Fields:
None
0:000> dd 0x01e1ad1c-0x4 l1
01e1ad18  00000000
```

我们可以看到同步块的索引为0。接下来获取这个锁（没有调用 GetHashCode 方法），并验证得到的是一个瘦锁。

```
0:003> !do 0x01e1ad1c
Name: Advanced.NET.Debugging.Chapter6.Simple
MethodTable: 000d3188
EEClass: 000d1298
Size: 12(0xc) bytes
 (C:\ADNDBin\06Simple2.exe)
Fields:
None
ThinLock owner 1 (002f70e0), Recursive 0
```

需要注意的是，命令 do 检测到了一个瘦锁，如最后一行所示。这行输出只是告诉我们，在这个对象上获取了一个瘦锁，其中线程对象指针为0x002f70e0，且递归计数为0。我们还可以通过查看对象头来验证这一点，在对象头中包含了持有锁的线程 ID。

```
0:003> dd 0x01e1ad1c-0x4 l1
01e1ad18  00000001
0:003> !threads
ThreadCount: 2
UnstartedThread: 0
BackgroundThread: 1
PendingThread: 0
DeadThread: 0
Hosted Runtime: no
                                    PreEmptive   GC Alloc           Lock
       ID OSID ThreadOBJ    State     GC       Context       Domain
   Count APT Exception
   0    1 22cc 002f70e0      a020 Enabled  01c8bd38:01c8c328 0008fdf8     2 MTA
   2    2 cdf0 000a3ae0      b220 Enabled  00000000:00000000 0008fdf8
     0 MTA (Finalizer)
```

我们可以看到，所有者线程的 ID 为1，指向它的指针为0x002f70e0，这与命令 do 的输出是相对应的。

接下来执行 GetHashCode 方法并再次中断执行，然后检查同步块和瘦锁的状态：

```
0:000> !do 0x01e1ad1c
Name: Advanced.NET.Debugging.Chapter6.Simple
MethodTable: 000d3188
EEClass: 000d1298
Size: 12(0xc) bytes
 (C:\ADNDBin\06Simple2.exe)
Fields:
None
0:000> dd 0x01e1ad1c-0x4 l1
01e1ad18  08000001
0:000> !syncblk 0001
Index SyncBlock MonitorHeld Recursion Owning Thread Info  SyncBlock Owner
    1 003231fc            1         1 002f70e0  4c98    0   01e1ad1c
 Advanced.NET.Debugging.Chapter6.Simple
```

```
-------------------------------
Total           1
CCW             0
RCW             0
ComClassFactory 0
Free            0
```

正如我们所料，此次命令 do 的输出中没有包含瘦锁信息，表示这个锁可能已经移动到同步块中。然后，再次使用 synblk 命令来验证在同步块中确实包含了一个有效锁。

使用瘦锁的技巧之一就是将 -thinlock 与 dumpheap 命令结合起来使用，-thinlock 命令能找出托管堆上所有带瘦锁的对象。同样使用 06sample2.exe 程序，在获取了这个锁后运行这个命令：

```
0:000> !dumpheap -thinlock
 Address       MT     Size
01e1bcdc 79191d38       16      ThinLock owner 1 (002f70e0) Recursive 0
```

在输出中给出了持有锁的对象的地址（0x01e1bcdc）和方法表（0x79191d38），以及线程对象指针（0x002f70e0）。

6.4 同步任务

现在我们已经讨论了各种同步原语，用于分析这些原语的调试器命令以及它们的内部结构，现在就可以转而分析一些最常见的同步问题。首先来讨论一种简单的死锁情况。

6.4.1 死锁

开发人员在编写多线程程序时，死锁或许是最常见也最令人沮丧的问题。当两个或多个线程分别持有一些被保护的资源，并且都拒绝释放各自的资源而等待另一方释放资源时，就会产生死锁。由于这些线程都不愿意释放它们自己的保护资源，最终任何一个线程都无法继续执行下去。由于无限地等待对方做出让步，这样就产生了死锁。发生死锁的原因有多种，本章我们将讨论一些常见的情况。然而，在分析复杂的情况之前，我们首先来看一个简单的死锁情况，通过这个示例能够更好地了解死锁在调试器中的行为以及在分析死锁原因时的一些常用命令。

用来说明死锁的示例程序非常简单，如清单 6-4 所示。

清单 6-4 发生死锁的示例程序

```
using System;
using System.Text;
using System.Threading;

namespace Advanced.NET.Debugging.Chapter6
{
    internal class DBWrapper1
    {
        private string connectionString;

        public DBWrapper1(string conStr)
```

```
        {
            this.connectionString = conStr;
        }
    }

    internal class DBWrapper2
    {
        private string connectionString;

        public DBWrapper2(string conStr)
        {
            this.connectionString = conStr;
        }
    }

    class Deadlock
    {
        private static DBWrapper1 db1;
        private static DBWrapper2 db2;
        static void Main(string[] args)
        {
            db1 = new DBWrapper1("DBCon1");
            db2 = new DBWrapper2("DBCon2");

            Thread newThread = new Thread(ThreadProc);
            newThread.Start();

            Thread.Sleep(2000);
            lock (db2)
            {
                Console.WriteLine("Updating DB2");
                Thread.Sleep(2000);
                lock (db1)
                {
                    Console.WriteLine("Updating DB1");
                }
            }
        }
        private static void ThreadProc()
        {
            Console.WriteLine("Start worker thread");
            lock (db1)
            {
                Console.WriteLine("Updating DB1");
                Thread.Sleep(3000);
                lock (db2)
                {
                    Console.WriteLine("Updating DB2");
                }
            }
        }
    }
```

```
}
```

清单6-4中的源代码和二进制文件位于以下文件夹中：

- 源代码文件：C:\ADND\Chapter6\DeadLock
- 二进制文件：C:\ADNDBIN\06DeadLock.exe

这个程序是一个多线程程序（包括2个线程），其中使用了封装在辅助类中的两个不同数据库（DBWrapper1和DBWrapper2）。每个线程都需要访问这两个数据库从而执行各自的工作。由于底层的数据库访问API并非线程安全的，因此这个程序使用了两个监视器对象（通过lock语句），每个对象保护一个数据库。为了避免使示例代码复杂化，我们使线程休眠一定毫秒数来模拟对数据库的访问操作。运行这个程序后，我们很快会发现程序永远无法结束。虽然你可能已经通过阅读代码找出了程序中的问题，但我们还是将一个调试器附载到运行中的程序，并且观察所发生的情况。第一步就是把当前进程中运行的所有线程都转储出来，如清单6-5所示。

清单6-5 在进程中运行的所有CLR线程

```
0:004> ~*e!clrstack
OS Thread Id: 0xaec (0)
ESP       EIP
0013f328 7c90e514 [GCFrame: 0013f328]
0013f3f8 7c90e514 [HelperMethodFrame_1OBJ: 0013f3f8]
 System.Threading.Monitor.Enter(System.Object)
0013f450 00cb017b Advanced.NET.Debugging.Chapter6.Deadlock.Main(System.Strin g[]
0013f69c 79e7be1b [GCFrame: 0013f69c]
OS Thread Id: 0x2e8 (1)
Unable to walk the managed stack. The current thread is likely not a
managed thread. You can run !threads to get a list of managed threads in
the process
OS Thread Id: 0x9d4 (2)
Failed to start stack walk: 80004005
OS Thread Id: 0xe1c (3)
ESP       EIP
00dbf75c 7c90e514 [GCFrame: 00dbf75c]
00dbf82c 7c90e514 [HelperMethodFrame_1OBJ: 00dbf82c]
 System.Threading.Monitor.Enter(System.Object)
00dbf884 00cb02f4
Advanced.NET.Debugging.Chapter6.Deadlock.ThreadProc()
00dbf8b4 793d70fb
System.Threading.ThreadHelper.ThreadStart_Context(System.Object)
00dbf8bc 793608fd
System.Threading.ExecutionContext.Run(System.Threading.ExecutionContext,
System.Threading.ContextCallback, System.Object)
00dbf8d4 793d71dc System.Threading.ThreadHelper.ThreadStart()
00dbfaf8 79e7be1b [GCFrame: 00dbfaf8]
OS Thread Id: 0x5ec (4)
```

```
Unable to walk the managed stack. The current thread is likely not a
managed thread. You can run !threads to get a list of managed threads in
the process
```

请注意命令 ~* e!clrstack 的使用，这个命令将转储出进程中所有线程（包括非托管线程和托管线程）的栈回溯。这里的输出信息虽然简单，但却展示了一种常见的死锁识别技术。两个（或者多个）线程都在等待获取对方的锁，但却似乎没有发生任何事情。要验证关于死锁的假设是否正确，我们需要进一步观察对象。如何找出每个线程正在等待的对象的地址？可以使用两种不同的方法。第一种方法是手动分析每个正在等待的线程。首先从线程 0 开始：

```
0:004> ~0s
eax=00000000 ebx=0013f0c4 ecx=0013f390 edx=7c90e514 esi=00000000 edi=7ffd6000
eip=7c90e514 esp=0013f09c ebp=0013f138 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
ntdll!KiFastSystemCallRet:
7c90e514 c3              ret
0:000> !ClrStack -a
OS Thread Id: 0xaec (0)
ESP       EIP
0013f328 7c90e514 [GCFrame: 0013f328]
0013f3f8 7c90e514 [HelperMethodFrame_1OBJ: 0013f3f8]
System.Threading.Monitor.Enter(System.Object)
0013f450 00cb017b Advanced.NET.Debugging.Chapter6.Deadlock.Main(System.String[])
    PARAMETERS:
        args = 0x01281a00
    LOCALS:
        0x0013f458 = 0x01281ae0
        0x0013f454 = 0x01281ab4
        0x0013f450 = 0x01281aa8

0013f69c 79e7be1b [GCFrame: 0013f69c]
```

从输出信息可以看出 Main 方法正在尝试获取一个锁。然而，如果在 Main 方法中获取两个锁，那么这个调用栈对应于哪个 lock 语句？ClrStack 命令输出中的第三个栈帧告诉我们，在进入监视器对象时，EIP（指令指针）的值为 0x00cb017b。现在，我们可以通过命令 U 来反编译这个位置上的汇编代码及相应的注记。下面给出了一个简化后的输出：

```
0:000> !u 00cb017b
Normal JIT generated code
Advanced.NET.Debugging.Chapter6.Deadlock.Main(System.String[])
Begin 00cb0070, size 17c
00cb0070 55              push    ebp
00cb0071 8bec            mov     ebp,esp
00cb0073 57              push    edi
00cb0074 56              push    esi
...
...
...
```

```
00cb0142 a1cc1e2802      mov     eax,dword ptr ds:[02281ECCh]
00cb0147 8945d4          mov     dword ptr [ebp-2Ch],eax
00cb014a 8b4dd4          mov     ecx,dword ptr [ebp-2Ch]
00cb014d e8873b1c79      call    mscorwks!JIT_MonEnterWorker (79e73cd9)
00cb0152 90              nop
00cb0153 90              nop
00cb0154 8b0d44302802    mov     ecx,dword ptr ds:[2283044h]
00cb015a e8198d7078      call    mscorlib_ni+0x2f8e78 (793b8e78)
 (System.Console.WriteLine(System.String), mdToken: 06000764)
00cb015f 90              nop
00cb0160 b9d0070000      mov     ecx,7D0h
00cb0165 e806767278      call    mscorlib_ni+0x317770 (793d7770)
 (System.Threading.Thread.Sleep(Int32), mdToken: 0600125e)
00cb016a 90              nop
00cb016b a1c81e2802      mov     eax,dword ptr ds:[02281EC8h]
00cb0170 8945d0          mov     dword ptr [ebp-30h],eax
00cb0173 8b4dd0          mov     ecx,dword ptr [ebp-30h]
00cb0176 e85e3b1c79      call    mscorwks!JIT_MonEnterWorker (79e73cd9)
>>> 00cb017b 90              nop
00cb017c 90              nop
00cb017d 8b0d48302802    mov     ecx,dword ptr ds:[2283048h]
00cb0183 e8f08c7078      call    mscorlib_ni+0x2f8e78 (793b8e78)
 (System.Console.WriteLine(System.String), mdToken: 06000764)
...
...
...
```

根据调用 Enter 时的指令指针（0x00cb0176）以及命令 U 的输出，我们可以看到，在这个指令指针之前的位置上是另一个 Enter 调用，位于 0x00cb014d。此时，我们知道它是 Main 方法中的第二个 lock 语句，正是这条语句使程序无限地等待。通过简单的代码分析可以知道，这条语句正尝试获取对象 db1 上的锁，该对象类型为 DBWrapper1。db1 的地址是多少？由于 db1 是一个静态变量，因此可以通过 ClrStack 输出中的 locals 来找到。在每个地址上使用命令 do，我们可以发现 db1 实例是位于地址 0x01281aa8 处：

```
0:000> !do 0x01281aa8
Name: Advanced.NET.Debugging.Chapter6.DBWrapper1
MethodTable: 009130d4
EEClass: 009113a4
Size: 12(0xc) bytes
 (C:\ADNDBin\06Deadlock.exe)
Fields:
      MT    Field   Offset                 Type VT     Attr    Value Name
790f9244  4000001        4        System.String  0 instance 01281a10
connectionString
```

现在，我们已经得到了 Main 方法等待获取锁的对象的地址，接下来可以查看这个对象的同步块是否由另一个线程持有（记住，如果是由一个瘦锁持有，那么在命令 do 中的输出中已经显示了相应的信息）：

```
0:000> dd 0x01281aa8-0x4 l1
01281aa4  08000003
```

同步块索引为0003，我们将这个值传递给syncblk命令，如下所示：

```
0:000> !syncblk 0003
Index SyncBlock MonitorHeld Recursion Owning Thread Info  SyncBlock Owner
    3 001aa934            3         1 001aa218   f74   3   01281aa8
Advanced.NET.Debugging.Chapter6.DBWrapper1
-----------------------------------
Total           3
CCW             0
RCW             0
ComClassFactory 0
Free            0
```

从syncblk命令的输出中，我们可以看到这个对象实例已经被另一个线程锁定，并且这个线程的调试器线程ID为0x3。让我们切换到这个线程，并且查看是否有任何情况能阻止这个线程释放锁：

```
0:000> ~3s
eax=8bcf97d4 ebx=00dbf4f8 ecx=ffffffff edx=ffffffff esi=00000000 edi=7ffde000
eip=7c90e514 esp=00dbf4d0 ebp=00dbf56c iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
ntdll!KiFastSystemCallRet:
7c90e514 c3              ret
0:003> !ClrStack
OS Thread Id: 0xf74 (3)
ESP       EIP
00dbf75c 7c90e514 [GCFrame: 00dbf75c]
00dbf82c 7c90e514 [HelperMethodFrame_1OBJ: 00dbf82c]
 System.Threading.Monitor.Enter(System.Object)
00dbf884 00cb02f4
Advanced.NET.Debugging.Chapter6.Deadlock.ThreadProc()
00dbf8b4 793d70fb
System.Threading.ThreadHelper.ThreadStart_Context(System.Object)
00dbf8bc 793608fd System.Threading.ExecutionContext.Run(System.Threading.
ExecutionContext, System.Threading.ContextCallback, System.Object)
00dbf8d4 793d71dc System.Threading.ThreadHelper.ThreadStart()
00dbfaf8 79e7be1b [GCFrame: 00dbfaf8]
```

从ID为0x3的线程的调用栈中，我们可以看到它是程序中的工作线程，同样在等待获取一个锁。我们可以通过在前面介绍的相同过程来找出它正在等待的锁，如以下步骤所示：

1）通过命令U反编译ThreadProc方法。

2）找出使得线程停滞的那个Enter调用（记住在ThreadProc方法中有两条lock语句）。

3）将线程正尝试获取锁的对象的内容转储出来。

在调试会话中，工作线程正尝试获取db2对象上的锁。

```
0:003> !syncblk 0002
Index SyncBlock MonitorHeld Recursion Owning Thread Info  SyncBlock Owner
```

```
    2 001aa904              3          1 00190d88     dc    0    01281ab4
 Advanced.NET.Debugging.Chapter6.DBWrapper2
--------------------------------
Total            3
CCW              0
RCW              0
ComClassFactory  0
Free             0
```

从输出中可以很清楚地看到，持有这个锁的线程的调试器线程 ID 为 0x0。这个线程也是我们在前面看到的持有 db1 对象上锁的线程。现在情况就变得很清楚了。进程中的第一个线程正持有第二个数据库锁，同时在等待第一个数据库锁被释放。第二个线程持有与第一个数据库锁相关的锁，同时在等待第二个数据库锁被释放。结果就是产生了死锁。

现在，我们已经知道了哪些线程在哪些资源上发生了死锁，最后要做的就是通过分析源代码来打破死锁。从示例程序中可以很清楚地看到为什么会发生死锁以及如何打破这个死锁。

到目前为止，我们已经介绍了如何通过手动方法来找出死锁的原因，这是一个非常繁琐的过程。幸运的是，有一些命令可以使这个过程变得自动化。第一个命令就是 syncblk 命令。如果在运行 syncblk 命令时不指定任何参数，那么将给出整个同步块表的内容，这是很有用的，因为它给出了系统中被持有锁的信息。在示例程序中，命令 syncblk 的输出如下所示：

```
0:003> !syncblk
Index SyncBlock MonitorHeld Recursion Owning Thread Info  SyncBlock Owner
    2 001aa904              3          1 00190d88     dc    0    01281ab4
Advanced.NET.Debugging.Chapter6.DBWrapper2
    3 001aa934              3          1 001aa218    f74    3    01281aa8
Advanced.NET.Debugging.Chapter6.DBWrapper1
--------------------------------
Total            3
CCW              0
RCW              0
ComClassFactory  0
Free             0
```

在输出中显示了所有被持有的锁，如果怀疑存在死锁情况，那么很快就能根据进程的锁定状态找出有价值的线索。虽然 syncblk 命令给出了所有对象的整体锁定状态，但还是需要通过一些分析来判断是否存在死锁。例如，必须查看每个线程来判断它们是否发生了死锁，以及它们在哪些对象上发生了死锁。通过使用 SOSEX 调试器扩展中的 dlk（deadlock）命令，可以将这个手动过程进一步自动化。dlk 命令分析同步块内容，并在检测到死锁时输出相应的信息。在前面的示例中，命令 dlk 将产生以下信息：

```
0:003> !dlk
Deadlock detected:
CLR thread 1 holds sync block 001aa904
OBJ:01281ab4[Advanced.NET.Debugging.Chapter6.DBWrapper2]
            waits sync block 001aa934
OBJ:01281aa8[Advanced.NET.Debugging.Chapter6.DBWrapper1]
CLR thread 3 holds sync block 001aa934
```

```
OBJ:01281aa8[Advanced.NET.Debugging.Chapter6.DBWrapper1]
             waits sync block 001aa904
OBJ:01281ab4[Advanced.NET.Debugging.Chapter6.DBWrapper2]
CLR Thread 1 is waiting at
Advanced.NET.Debugging.Chapter6.Deadlock.Main(System.String[])(+0x67
 IL)(+0xd2 Native) [c:\Publishing\ADND\Code\Chapter6\Deadlock\06Deadlock.cs, @
45,17]
CLR Thread 3 is waiting at
Advanced.NET.Debugging.Chapter6.Deadlock.ThreadProc()(+0x30 IL)(+0x46 Native)
[c:\Publishing\ADND\Code\Chapter6\Deadlock\06Deadlock.cs, @ 59,17]
```

在输出信息中，可以很明显地看到检测出了一个潜在的死锁，线程 1 持有对象 DBWrapper2 上的锁，并正在等待 DBWrapper1，而线程 3 持有 DBWrapper1 上的一个锁，并正在 DBWrapper2 上等待。此外，最后一部分输出还给出了每个线程正在等待的准确位置。

请注意，之前的讨论都是假设这个锁被保存在同步块中。如果使用的是瘦锁，也可以使用相同的策略，只不过要通过 dumpheap-thinlock 命令（而不是 syncblk 命令）来找出哪些对象被锁定，以及哪个线程持有哪个锁。同样，在编写本书的时候，SOSEX blk 命令还不能处理瘦锁。

到这里就结束了对死锁调试的讨论。虽然我们以监视器对象为例来讨论死锁情况，但如果使用不当，任何同步原语都可能产生死锁。

6.4.2　孤立锁：异常

在编写代码时，要确保在出现异常的情况下代码仍能表现出良好的行为，这是一项困难的工作，尤其在多线程情况下。在本节中，我们将看到一个程序如何（糟糕地）使用异常。在清单 6-6 中给出了程序的代码。

清单 6-6　产生孤立锁的示例程序

```
using System;
using System.Text;
using System.Threading;

namespace Advanced.NET.Debugging.Chapter6
{
    internal class DBWrapper1
    {
        private string connectionString;

        public DBWrapper1(string conStr)
        {
            this.connectionString = conStr;
        }
    }

    class Exc
    {
```

```
        private static DBWrapper1 db1;

        static void Main(string[] args)
        {
            db1 = new DBWrapper1("DB1");

            Thread newThread = new Thread(ThreadProc);
            newThread.Start();

            Thread.Sleep(500);
            Console.WriteLine("Acquiring lock");
            Monitor.Enter(db1);

            //
            // 执行一些工作
            //

            Console.WriteLine("Releasing lock");
            Monitor.Exit(db1);
        }

        private static void ThreadProc()
    {
        try
        {
            Monitor.Enter(db1);
            Call3rdPartyCode(null);
            Monitor.Exit(db1);
        }
        catch (Exception)
        {
            Console.WriteLine("3rd party code threw an exception");
        }
    }
        private static void Call3rdPartyCode(Object obj)
        {
            if (obj == null)
            {
                throw new NullReferenceException();
            }

            //
            // 执行一些工作
            //
        }
    }
}
```

清单 6-6 中的源代码和二进制文件位于以下文件夹中：

- 源代码文件：C:\ADND\Chapter6\Exception

- 二进制文件：C:\ADNDBIN\06Exception. exe

你可以看到，清单6-6中的代码非常简单。在主线程中，首先创建了一个新的工作线程，在这个线程被成功创建后，它会尝试获取对象db1上的锁。而在获得了锁之后，又会释放这个锁并结束线程。

工作线程的任务是调用某个第三方代码（或许是一个动态加载的程序集），同时持有对象db1上的锁。这段代码还努力成为一段非常安全的代码：将调用语句封装在try/catch语句中，试图捕获所有被抛出的异常，并且在抛出异常的情况下将错误转储到控制台。

虽然这个程序的设计很糟糕，但它却说明了一个常见的问题。在深入分析程序中的所有问题之前，我们首先运行程序，并且看到它的最终输出结果是：

```
C:\ADNDBin>06Exception.exe
3rd party code threw an exception
Acquiring lock
```

看上去，程序似乎在试图获取一个锁的时候挂起了。我们来观察进程中各个线程的状态，将调试器附载到进程上，并且转储出所有的线程，如清单6-7所示。

清单6-7 示例的程序线程状态

```
0:004> ~*e!clrstack
OS Thread Id: 0x1530 (0)
ESP        EIP
0025ec6c 77d99a94 [GCFrame: 0025ec6c]
0025ed3c 77d99a94 [HelperMethodFrame_1OBJ: 0025ed3c]
 System.Threading.Monitor.Enter(System.Object)
0025ed94 01bf0111
Advanced.NET.Debugging.Chapter6.Exc.Main(System.String[])
0025efc4 79e7c74b [GCFrame: 0025efc4]
OS Thread Id: 0x1aec (1)
Unable to walk the managed stack. The current thread is likely not a
managed thread. You can run !threads to get a list of managed threads in
the process
OS Thread Id: 0x1e00 (2)
Failed to start stack walk: 80004005
OS Thread Id: 0x1640 (3)
Unable to walk the managed stack. The current thread is likely not a
managed thread. You can run !threads to get a list of managed threads in
the process
OS Thread Id: 0x15b0 (4)
Unable to walk the managed stack. The current thread is likely not a
managed thread. You can run !threads to get a list of managed threads in the process
```

清单6-7表示，主线程正在等待获取一个锁，但却没有成功（其余的线程都是非托管线程）。要找出不能成功的原因，我们必须转储出存在问题的锁，并且看看它能提供什么样的信息。要转储出这个锁，必须首先找到被分析的对象（类型为DBWrapper1），可以使用命令

dumpstackobjects，如下所示：

```
0:000> !dumpstackobjects
OS Thread Id: 0xd330 (0)
ESP/REG  Object   Name
ecx      01deadb0 Advanced.NET.Debugging.Chapter6.DBWrapper1
002df144 01deadb0 Advanced.NET.Debugging.Chapter6.DBWrapper1
002df198 01deadbc System.Threading.ThreadStart
002df19c 01deaddc System.Threading.Thread
002df1a0 01deaddc System.Threading.Thread
002df1a8 01deadbc System.Threading.ThreadStart
002df1b0 01deaddc System.Threading.Thread
002df1c4 01decc0c System.IO.StreamWriter
002df1e4 01deaeb4 System.Char[]
002df20c 01deaddc System.Threading.Thread
002df218 01deadb0 Advanced.NET.Debugging.Chapter6.DBWrapper1
002df234 01dead28 System.Object[]     (System.String[])
002df2f4 01dead28 System.Object[]     (System.String[])
002df4a8 01dead28 System.Object[]     (System.String[])
002df4d0 01dead28 System.Object[]     (System.String[])
```

然后，我们使用输出中的对象指针来获取对象头：

```
0:000> dd 01deadb0-0x4 l1
01deadac  08000002
```

可以看到，对象头已经创建了一个同步块。要获得锁信息，可以使用 syncblk 命令：

```
0:000> !syncblk 002
Index SyncBlock MonitorHeld Recursion Owning Thread Info  SyncBlock Owner
    2 00443994            3         1 00442208      0 XXX   01deadb0
 Advanced.NET.Debugging.Chapter6.DBWrapper1
-----------------------------
Total           2
CCW             0
RCW             0
ComClassFactory 0
Free            0
```

输出信息很清楚地告诉我们，这个对象被锁定了。在输出中需要注意的是 Owning Thread Info 列，其中 XXX 表示调试器线程 ID，0 表示操作系统线程 ID。XXX 的含义是什么？它只是表示 CLR 无法将操作系统线程 ID 映射到调试器线程。出现这种情况的一个常见原因就是，这个线程在某个时刻获取了该对象上的锁，然后这个线程消失了但却*没有释放*这个锁。为了进一步验证这种情况，可以使用 threads 命令：

```
0:000> !threads
ThreadCount: 3
UnstartedThread: 0
BackgroundThread: 1
PendingThread: 0
DeadThread: 1
```

```
Hosted Runtime: no
                                        PreEmptive   GC Alloc          Lock
       ID OSID ThreadOBJ    State      GC        Context        Domain
  Count APT Exception
    0     1 d330 00417130   200a020 Enabled  01deaee0:01dec328 0040fe20      0 MTA
    2     2 cb7c 00423b40      b220 Enabled   00000000:00000000 0040fe20      0 MTA
(Finalizer)
XXXX    3    0 00442208      9820 Enabled  00000000:00000000 0040fe20      1 Ukn
```

输出信息指出，有 1 个线程已经死亡了，并且这个线程在线程列表中用 XXX 来标记。只要没有执行终结操作，即使处于死亡状态的线程也会被输出。

下一个需要解决的问题就是，如何找出这个已死亡的线程是哪个线程？要回答这个问题，首先要进行一些代码分析，同时与调试器输出的数据结合起来。在这个示例中，唯一一个其他的线程就是工作线程，那么它是否是造成这种情况的主要原因？可能是，因为它肯定会尝试获取一个锁：

```
Monitor.Enter(db1);
Call3rdPartyCode(null);
Monitor.Exit(db1);
```

唯一的问题是，它在调用第三方代码后，同样正确地释放了锁，因此这个锁应该处于良好的状态。这可能是最初的印象，但通过进一步观察代码，我们发现问题在于锁定代码周围的 try/catch 块。事实上，由于线程正在调用某个第三方代码，因此它会尽量保证它自己不抛出任何类型的异常，并试图捕获所有可能出现的异常。虽然有人会认为这并不是保护自己的最佳方式，但实际上却带来了一个更大的问题。如果在第三方代码中抛出一个异常，那么将发生什么情况？“catch all“过滤器会被执行，并在输出一个错误信息到控制台后退出。我们是否释放了在调用第三方代码之前获取的锁？绝对没有。在这种情况下，唯一执行的代码就是 catch 过滤器，并且最终将导致一个孤立锁。现在，我们知道了为什么程序会挂起：工作线程使一个锁变成了孤立锁。除了试图通过捕获所有异常来保护它自己之外，还引出了另外一个问题，即在调用第三方代码时持有一个锁是否安全？通常来说，答案是否定的。因为你无法知道这个第三方代码会执行哪些操作，因此在调用它时持有一个锁可能会导致其他破坏性的问题。想象一下，假设这个第三方代码尝试回调某个 API，而这个 API 需要获取同一个锁（来自一个不同的线程）。如果这个 API 并没有按照可重入的方式来设计，那么就会产生一个死锁。通常来说，开发人员要非常小心地处理第三方代码并努力保护他们的代码不会发生各种问题。

如果编写清单 6-6 中代码的开发人员不同意对代码进行任何重大修改，那么能否通过某些其他方法使代码的行为变得更好一些？绝对可以。它可以确保在异常处理代码中释放这个锁，从而在出现异常的情况下也能正确地释放锁。要实现更强的保证，另一种方法就是添加一个 finally 块，并在这个代码块中释放锁。最后，要确保更高的代码可读性，还可以使用在前面介绍过的 lock 语句，它将被编译为 Monitor. Enter 和 Monitor. Exit（在 finally 块中）。

6.4.3 线程中止

在上一节中，我们介绍了在没有正确释放锁时造成的孤立锁问题。那么，在代码能够正确地释放锁的情况下，是否还存在其他情况也能造成孤立锁？答案是肯定的。最常见的就是调用了 Thread. Abort 方法。虽然 CLR 的线程中止（thread abortion）操作会干净地结束一个线程，但在许多情况下，中止线程会导致各种问题。我们来看其中一种情况，它与前面用来说明由于线程中止而造成的孤立锁的情况略有不同。清单 6-8 给出了程序的源代码。

清单 6-8 产生孤立锁的示例程序

```
using System;
using System.Threading;

namespace Advanced.NET.Debugging.Chapter6
{
    class Abort
    {
        public void WorkerThread()
        {
            try
            {
                //
                //执行一些工作
                //
            }
            finally
            {
                //在持有锁的情况下执行一些清理工作
            {
            }
        }
    }

    public static void Main(string[] args)
    {
        Abort abort = new Abort();
        Thread worker = new Thread(abort.WorkerThread);
        lock (abort)
        {
            worker.Start();

            Console.WriteLine("Acquired lock");
            Thread.Sleep(2000);

            Console.WriteLine("Aborting worker thread");
            worker.Abort();
        }
    }
}
```

```
}
```

清单 6-8 中的源代码和二进制文件位于以下文件夹中：

- 源代码文件：C:\ADND\Chapter6\Abort
- 二进制文件：C:\ADNDBIN\06Abort.exe

这段代码的执行流程是，主线程启动一个工作线程来执行一些后台工作。然后，主线程等待一段时间，直到它决定中止工作线程（或许认为工作线程停滞了）。如果执行这个程序，那么将看到以下输出：

```
C:\ADNDBin>06Abort.exe
Acquired lock
Aborting worker thread
```

程序总是停滞不前，并且永远无法结束。我们在程序上附载一个调试器，并且按照同样的过程来调试它。首先将所有的线程转储出来，查看是否有任何线程停滞不前（不考虑非托管线程）：

```
0:004> ~*e!clrstack
OS Thread Id: 0x135c (0)
ESP       EIP
0013f3e8 7c90e514 [HelperMethodFrame_1OBJ: 0013f3e8]
System.Threading.Thread.AbortInternal()
0013f440 793d750c System.Threading.Thread.Abort()
0013f450 00cb013d Advanced.NET.Debugging.Chapter6.Abort.Main(System.String[])
0013f69c 79e7be1b [GCFrame: 0013f69c]
OS Thread Id: 0x388 (3)
ESP       EIP
00dbf754 7c90e514 [GCFrame: 00dbf754]
00dbf824 7c90e514 [HelperMethodFrame_1OBJ: 00dbf824]
System.Threading.Monitor.Enter(System.Object)
00dbf87c 00cb0209 Advanced.NET.Debugging.Chapter6.Abort.WorkerThread()
00dbf8b4 793d70fb
System.Threading.ThreadHelper.ThreadStart_Context(System.Object)
00dbf8bc 793608fd
System.Threading.ExecutionContext.Run(System.Threading.ExecutionContex
t, System.Threading.ContextCallback, System.Object)
00dbf8d4 793d71dc System.Threading.ThreadHelper.ThreadStart()
00dbfaf8 79e7be1b [GCFrame: 00dbfaf8]
```

从输出信息中可以看到，有两个线程需要注意。第一个是主线程（id 为 0），似乎在调用 Abort 方法时停滞了，而第二个线程（id 为 3）像是在调用 Monitor. Enter 时停滞了。

为了更好地理解这两个线程的状态，我们需要知道当线程被中止时究竟发生了哪些操作。当线程中止时，在目标线程中抛出了一个 ThreadAbortException 异常，它使得线程在结束之前能够执行一些清理工作。对于那些需要确保清理工作被正确执行的线程来说，这种情况可以看成是一种合作式的线程中止（与 Windows 中的 TerminatedThread 不同）。我们知道本应该抛出一个 ThreadAbortException 异常，并且应该结束工作线程，那么为什么它仍然在等待监视器对象？我们要验证工作线程确实收到了一个中止请求：

```
0:004> !threads
ThreadCount: 3
UnstartedThread: 0
BackgroundThread: 1
PendingThread: 0
DeadThread: 0
Hosted Runtime: no
                                        PreEmptive   GC Alloc          Lock
       ID OSID ThreadOBJ    State      GC        Context        Domain
  Count APT Exception
   0     1 135c 00190d28   200a020 Enabled  00000000:00000000 0015bf10        1 MTA
   2     2 13d4 0019aa40      b220 Enabled  00000000:00000000 0015bf10        0 MTA
(Finalizer)
   3     3  388 001a9d30   200b021 Enabled  00000000:00000000 0015bf10        0 MTA
```

工作线程的状态为0x0200b021。关键在于第一个比特位（1）。参照表6-1中给出的描述，我们可以看到它已经收到了一个中止请求（TS_AbortRequested）。总结目前的发现，我们可以知道主线程启动了一个新线程，并尝试中止新线程（并且在执行这个操作时停滞了），而新线程收到了一个中止请求，但却拒绝结束。这个问题的关键在于，工作线程在执行finally代码块时确实收到了一个中止请求，但中止一个正在执行finally代码块的线程是不允许的，因此就产生了一个死锁。

lock语句与线程中止

在线程中止的情况下，lock语句能否总是释放这个锁？答案是有可能。关键是要在lock语句生成的IL代码上进行锁定。

```
IL_0008:  call       void [mscorlib]System.Threading.Monitor::Enter(object)
IL_000d:  nop
.try
{
  IL_000e:  nop
  IL_000f:  ldstr      "Test"
  IL_0014:  call       void
Advanced.NET.Debugging.Chapter6.Abort::Call3rdPartyCode(object)
  IL_0019:  nop
  IL_001a:  nop
  IL_001b:  leave.s    IL_0025
}  // end .try
finally
{
  IL_001d:  ldloc.0
  IL_001e:  call       void [mscorlib]System.Threading.Monitor::Exit(object)
  IL_0023:  nop
  IL_0024:  endfinally
}  // end handler
```

lock语句的基本思想是，首先获取锁（IL_0008），接着将代码区域封装起来，在try/finally的保护下执行。在finally语句中会释放这个监视器。现在，如果执行这段代码的线

程被中止了，那么 finally 语句将被调用，这个锁也会被释放。进一步观察 IL，可以看到一个值得注意的 nop 指令（IL_000d）。nop 指令是由编译器插入的，用于支持不同的调试情况（例如设置断点）。这里的关键点在于 nop 指令被插入在 try 块之前。这意味着，如果一个线程在执行 nop 指令时被中止了，那么这个线程中止异常将永远不会被捕获，从而监视器的状态仍将保持为锁定状态。请注意，这种情况只适用于调试构建（其中关闭了编译优化）。这是一种尤其要注意的情况，在这种情况下，你可能会遇到调试构建，并在产品构建中追踪这个死锁，而在产品构建中是永远都不会出现死锁的。

6.4.4 终结器挂起

在第 5 章中，我们介绍了 CLR 内存管理器并且观察了一些可能在程序中造成破坏的错误示例。在本章的这部分内容中，我们将讨论另一个值得注意的托管堆问题，它是由于线程之间的不良同步而造成的。清单 6-9 给出了说明这个问题的程序。

清单 6-9 出现内存泄漏的程序

```
using System;
using System.IO;
using System.Runtime.InteropServices;
using System.Management;

namespace Advanced.NET.Debugging.Chapter6
{
    class Wmi
    {
        private ManagementClass obj;
        private byte[] data;

        public Wmi(byte[] data)
        {
            this.data = data;
        }

        public void ProcessData()
        {
            obj = new ManagementClass("Win32_Environment");
            obj.Get();
        //
        // 使用数据成员引用
        //
    }
    ~Wmi()
    {
        //
        // 清除任何非托管资源
```

```
        //
    }
}

class Worker
{
    public Worker()
    {
        Init();
    }

    public void ProcessData(byte[] data)
    {
        Process(data);
    }

    ~Worker()
    {
        UnInit();
    }

    [DllImport("06Native.dll")]
    static extern void Init();

    [DllImport("06Native.dll")]
    static extern void UnInit();

    [DllImport("06Native.dll")]
    static extern void Process(byte[] data);
}

class OOMFin
{
    private Worker worker;

    static void Main(string[] args)
{
    if (args.Length < 1)
    {
        Console.WriteLine("06Finalize.exe <num iterations>");
        return;
    }

    OOMFin o = new OOMFin();
    o.Run(Int32.Parse(args[0]));
}

public void Run(int iterations)
{
    Initialize();

    for (int i = 0; i < iterations; i++)
    {
```

```
            byte[] b = new byte[10000];
            Wmi w = new Wmi(b);
            w.ProcessData();
        }

        GC.Collect();

        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }

    private void Initialize()
    {
        byte[] b = new byte[100];

        worker = new Worker();
        worker.ProcessData(b);

        worker = null;
        GC.Collect();
    }
  }
}
```

清单 6-9 中的源代码和二进制文件位于以下文件夹中：

- 源代码文件：C:\ADND\Chapter6\Finalize
- 二进制文件：C:\ADNDBin\06Finalize. exe and 05Native. dll

在这个程序中包含了一个 Worker 类，它通过互用性服务来处理收到的数据。驱动类是 OOMFin，其中包含了一个 Run 方法，在这个方法中使用了 Worker 类，并随后进入一个循环通过 Wmi 类从系统中提取环境块数据。迭代的次数是通过以下命令行来控制的：

```
C:\ADNDBin\06Finalize.exe <number of iterations>
```

程序的源代码很简单，其中没有任何情况表明在程序中存在与内存相关的问题。在调试器下运行这个程序，并且指定 10000 次迭代。在程序执行完毕后，我们怀疑它消耗了太多的内存，因而要通过调试器命令来找出内存的状态。首先使用命令 eeheap-loader 来观察在输出信息中是否包含任何异常：

```
0:007> !eeheap -loader
Loader Heap:
---------------------------------
System Domain: 7a3bc8b8
...
...
...
Total size: 0x3000(12288)bytes
---------------------------------
Shared Domain: 7a3bc560
...
...
```

```
...
Total size: 0x5000(20480)bytes
-------------------------------
Domain 1: 34fd48
...
...
...
Total size: 0x8000(32768)bytes
-------------------------------
Jit code heap:
LoaderCodeHeap: 00a60000(10000:1000) Size: 0x1000(4096)bytes.
Total size: 0x1000(4096)bytes
-------------------------------
Module Thunk heaps:
...
...
...
Total size: 0x0(0)bytes
-------------------------------
Module Lookup Table heaps:
...
...
...
Total size: 0x0(0)bytes
-------------------------------
Total LoaderHeap size: 0x11000(69632)bytes
```

加载器堆的总大小为69 632字节，这并不是非常高。因此，我们可以推断在加载器堆中没有出现问题。接下来，通过命令eeheap-gc来查看托管堆：

```
0:007> !eeheap -gc
Number of GC Heaps: 1
generation 0 starts at 0x125c2280
generation 1 starts at 0x123f63f4
generation 2 starts at 0x01fa1000
ephemeral segment allocation context: none
 segment    begin allocated     size
003872b8 7a733370  7a754b98 0x00021828(137256)
00367b38 790d8620  790f7d8c 0x0001f76c(128876)
01fa0000 01fa1000  02f9ee78 0x00ffde78(16768632)
054a0000 054a1000  0649f31c 0x00ffe31c(16769820)
07830000 07831000  0882e0d4 0x00ffd0d4(16765140)
0ada0000 0ada1000  0bd9f5cc 0x00ffe5cc(16770508)
0d250000 0d251000  0e24e0d4 0x00ffd0d4(16765140)
0f220000 0f221000  1021e0d4 0x00ffd0d4(16765140)
111f0000 111f1000  11d99830 0x00ba8830(12224560)
121f0000 121f1000  125c428c 0x003d328c(4010636)
Large object heap starts at 0x02fa1000
 segment    begin allocated     size
02fa0000 02fa1000  02fa4240 0x00003240(12864)
Total Size  0x6fb166c(117118572)
-------------------------------
GC Heap Size  0x6fb166c(117118572)
```

托管堆的总大小为117MB左右，这也不是非常高，但考虑到这个程序将要退出，并且它之前强行执行了一次垃圾收集操作，因此我们需要进一步分析这个统计数据。要获得关于托管堆的更多信息，可以使用DumpHeap-stat命令：

```
0:007> !DumpHeap -stat
total 226604 objects
Statistics:
      MT    Count    TotalSize Class Name
79119954        1           12
System.Security.Permissions.ReflectionPermission
79119834        1           12
System.Security.Permissions.FileDialogPermission
791197b0        1           12 System.Security.PolicyManager
...
...
...
6758707c    10000       120000
System.Management.IWbemClassObjectFreeThreaded
790fa9d8    10000       160000 System.__ComObject
000c3104    10000       160000 Advanced.NET.Debugging.Chapter6.Wmi
79104368    10003       240072 System.Collections.ArrayList
790fd8c4     4972       270224 System.String
67585310    10000       280000 System.Management.ManagementScope
67589900    20001       320016 System.Management.WbemDefPath
67584c28    20001       400020 System.Management.ManagementPath
67583f10    10000       480000
System.Management.ManagementNamedValueCollection
67584b74    10000       560000 System.Management.ConnectionOptions
79101fe4    10024       561344 System.Collections.Hashtable
67583650    10000       640000 System.Management.ManagementClass
7912d9bc    10024      1443528 System.Collections.Hashtable+bucket[]
67583fcc    60000      1920000
System.Management.IdentifierChangedEventHandler
003563b0    11301      9406484      Free
7912dae8    10003    100120360 System.Byte[]
Total 226604 objects
```

从输出信息中，我们可以看到在程序中仍然有10 000个Wmi类的实例存在。占据最大内存的类型是这个字节数组，它是由Wmi类型持有的。为什么会存在这些Wmi类型？使用这些类型的代码很简单，并且所有的Wmi实例都应该被作为垃圾收集：

```
for (int i = 0; i < iterations; i++)
{
    byte[] b = new byte[10000];
    Wmi w = new Wmi(b);
    w.ProcessData();
}
```

由于Wmi实例看上去仍然存在根对象引用，我们可以通过GCRoot命令来找出这个引用链，这个命令的参数是需要分析的对象地址。要找出这个地址，需要使用命令DumpHeap- type：

```
  0:007> !DumpHeap -type Advanced.NET.Debugging.Chapter6.Wmi
 Address       MT     Size
01fa72b4 000c3104       16
01fac2a8 000c3104       16
01faec9c 000c3104       16
...
...
...
125b2c9c 000c3104       16
125b73dc 000c3104       16
125bbb1c 000c3104       16
125c025c 000c3104       16
total 10000 objects
Statistics:
      MT    Count    TotalSize Class Name
000c3104    10000       160000 Advanced.NET.Debugging.Chapter6.Wmi
Total 10000 objects
0:007> !GCRoot 125bbb1c
Note: Roots found on stacks may be false positives. Run "!help gcroot" for
more info.
Scan Thread 0 OSTHread 2bdc
Scan Thread 1 OSTHread 23f4
Finalizer queue:Root:125bbb1c(Advanced.NET.Debugging.Chapter6.Wmi)
```

我们选择位于地址0x125bbb1c处的对象，并将这个地址作为GCRoot命令的参数。需要注意的是，这个对象的根对象引用是F-Reachable队列。这个对象位于F-Reachable队列中，这一点非常有意义，因为它带有一个Finalize方法。但我们并不清楚为什么这个对象还没有被回收，使用FinalizeQueue命令来观察F-Reachable队列的状态：

```
0:007> !FinalizeQueue
SyncBlocks to be cleaned up: 50000
MTA Interfaces to be released: 0
STA Interfaces to be released: 0
---------------------------------
generation 0 has 3 finalizable objects (10c0ec0c->10c0ec18)
generation 1 has 3 finalizable objects (10c0ec00->10c0ec0c)
generation 2 has 4 finalizable objects (10c0ebf0->10c0ec00)
Ready for finalization 29998 objects (10c0ec18->10c2c0d0)
Statistics:
      MT    Count    TotalSize Class Name
7911c9c8        1           20 Microsoft.Win32.SafeHandles.SafePEFileHandle
79112728        1           20 Microsoft.Win32.SafeHandles.SafeWaitHandle
791037c0        1           20 Microsoft.Win32.SafeHandles.SafeFileMappingHandle
79103764        1           20 Microsoft.Win32.SafeHandles.SafeViewOfFileHandle
79101444        1           20 Microsoft.Win32.SafeHandles.SafeFileHandle
79108ba4        1           44 System.Threading.ReaderWriterLock
790fe704        1           56 System.Threading.Thread
7910a5c4        1           60 System.Runtime.Remoting.Contexts.Context
6758707c    10000       120000 System.Management.IWbemClassObjectFreeThreaded
000c3104    10000       160000 Advanced.NET.Debugging.Chapter6.Wmi
67583650    10000       640000 System.Management.ManagementClass
Total 30008 objects
```

看上去在 F-Reachable 队列上有近 3000 个对象，但问题是为什么这些对象还没有被终结？一种可能的假设是，进程中的终结线程还没有醒来，因此也就没有运行 F-Reachable 队列上所有对象的 Finalize 方法。让我们来查看终结线程的状态：

```
0:007> !Threads
ThreadCount: 2
UnstartedThread: 0
BackgroundThread: 1
PendingThread: 0
DeadThread: 0
Hosted Runtime: no
                                    PreEmptive   GC Alloc           Lock
       ID OSID ThreadOBJ    State     GC       Context       Domain
  Count APT Exception
   0     1 2bdc 00354448      a020 Enabled  125c2e0c:125c4280 0034fd48     0 MTA
   1     2 23f4 00358d68      b220 Enabled  00000000:00000000 0034fd48     0 MTA
(Finalizer)
0:007> ~1s
eax=00358d68 ebx=00000000 ecx=041af6c0 edx=041af6cc esi=70627138 edi=00000000
eip=77419a94 esp=041af800 ebp=041af864 iopl=0         nv up ei pl nz ac po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000213
ntdll!KiFastSystemCallRet:
77419a94 c3              ret
0:001> k
ChildEBP RetAddr
041af7fc 77419254 ntdll!KiFastSystemCallRet
041af800 774033b4 ntdll!ZwWaitForSingleObject+0xc
041af864 7740323c ntdll!RtlpWaitOnCriticalSection+0x155
041af88c 706214e4 ntdll!RtlEnterCriticalSection+0x152
041af964 000ba26b 05Native!UnInit+0x34
WARNING: Frame IP not in any known module. Following frames may be wrong
041af9a8 79fbcca7 0xba26b
041af9c8 79fbcca7 mscorwks!MethodTable::SetObjCreateDelegate+0xc5
041afa2c 79fbcc15 mscorwks!MethodTable::SetObjCreateDelegate+0xc5
041afa4c 79fbcbb7 mscorwks!MethodTable::CallFinalizer+0x76
041afa60 79f6acb6 mscorwks!SVR::CallFinalizer+0xb2
041afab0 79f6abf7 mscorwks!WKS::GCHeap::TraceGCSegments+0x170
041afb38 79fb99d6 mscorwks!WKS::GCHeap::TraceGCSegments+0x2b6
041afb50 79ef3207 mscorwks!WKS::GCHeap::FinalizerThreadWorker+0xe7
041afb64 79ef31a3 mscorwks!Thread::DoADCallBack+0x32a
041afbf8 79ef30c3 mscorwks!Thread::ShouldChangeAbortToUnload+0xe3
041afc34 79fb9643 mscorwks!Thread::ShouldChangeAbortToUnload+0x30a
041afc5c 79fb960d mscorwks!ManagedThreadBase_NoADTransition+0x32
041afc6c 79fba09b mscorwks!ManagedThreadBase::FinalizerBase+0xd
041afca4 79f95a2e mscorwks!WKS::GCHeap::FinalizerThreadStart+0xbb
041afd48 77584911 mscorwks!Thread::intermediateThreadProc+0x49
```

值得注意的是，终结线程似乎正在执行一个 Finalize 方法。要想知道执行的是哪个对象上的 Finalize 方法，可以使用 ClrStack 命令来获得一个托管调用栈：

```
0:001> !ClrStack
OS Thread Id: 0x23f4 (1)
ESP       EIP
041af994 77419a94 [NDirectMethodFrameStandalone: 041af994]
Advanced.NET.Debugging.Chapter6.Worker.UnInit()
041af9a4 00a6030d Advanced.NET.Debugging.Chapter6.Worker.Finalize()
```

从输出信息中可以看出，终结线程正在执行 Worker 类型的 Finalize 方法。NDirectMethodFrameStandalone 表示正在执行从托管到非托管的转换。如果回到非托管调用栈，在栈的顶部可以看到：

```
041af7fc 77419254 ntdll!KiFastSystemCallRet
041af800 774033b4 ntdll!ZwWaitForSingleObject+0xc
041af864 7740323c ntdll!RtlpWaitOnCriticalSection+0x155
041af88c 706214e4 ntdll!RtlEnterCriticalSection+0x152
041af964 000ba26b 05Native!UnInit+0x34
```

终结线程正在执行 Worker 类型的 Finalize 方法，这个方法通过互用性服务来调用一个非托管模块（05Native），接着进入一个临界区，但似乎在执行这个过程时停滞了。通过进一步分析源代码，我们可以看到，在 OOMFin 类的 Initialize 方法中调用的 ProcessData 函数进入了一个临界区，但随后却没有释放这个临界区。UnInit 函数接着尝试进入这个临界区，但由于临界区已经被锁定了，因此这个函数调用将无限期地等待下去。由于在 Wmi 类之前使用了 Worker 类（同样包含一个 Finalize 方法），因此在 F-Reachable 队列上，Worker 类型实例位于所有 Wmi 实例之前，这同样意味着终结线程会首先选择 Worker 实例来执行，此时将立即停滞。由于终结线程以串行方式来依次取出 F-Reachable 队列上的对象，所以任何导致终结线程发生故障（或者停滞不前）的对象，都将使终结线程无法执行 F-Reachable 队列上其余对象的 Finalize 方法，从而使其他对象永远都无法被清除或者作为垃圾对象被收集。结果就是，内存消耗量无限地增长，并最终抛出 OutOfMemoryException 异常。

正如在本章这部分所看到的，要正确地使用 Finalize 方法并不容易。我们必须非常小心地确保 Finalize 代码不会在无意中出现错误（例如一个死锁），因为即使最小的错误都可能使特定的对象无法被清除，还会影响 F-Reachable 队列中的其他对象。

6.5 小结

多线程编程看上去是一种很容易的编程方法。首先创建一组线程，然后让它们并行工作以完成某项任务。但正如在本章中所看到的，并发和同步等领域的知识其实非常复杂，有人甚至认为这是最容易出现错误的领域之一。我们必须非常小心地确保所有线程都是活跃的并且能够和谐地在一起工作。在多线程逻辑中，即使一个小错误也可能导致严重的后果。

在本章中，我们介绍了一些在编写多线程程序和同步时常见的错误。首先简要介绍了 CLR 中不同的同步原语，以及这些原语在运行时中的定义。然后给出了一些示例情况，例如死锁、孤立锁，线程中止以及终结过程中的死锁等，此外还介绍了如何通过调试器来找出这些问题的根本原因。

第7章 互 用 性

当2002年发布.NET 1.0时,.NET团队面对的一个问题是：对于不是在托管运行时环境中编写的遗留代码，该提供怎样的支持？由于新平台的成功与否，在很大程度上取决于人们对平台的使用率，因此.NET团队决定对现有的非托管代码提供完全且无缝的集成。在两个主要的领域中，托管代码需要调用非托管代码（反之亦然），或者是通过COM，或者是调用从DLL中导出的函数。托管代码调用COM的情况（反之亦然）叫做COM互用性（也叫做COM interop），而调用导出DLL函数则称之为平台调用服务（Platform Invocation Services，P/Invoke）。本章我们将介绍COM互用性和平台调用服务的内部工作机制，以及当托管代码和非托管代码之间发生不正确交互时出现的一些问题。

7.1 平台调用

平台调用服务是CLR的一部分，负责确保托管代码可以调用从非托管DLL中导出的各种函数。虽然.NET框架做了大量的工作来封装Win32中的大多数函数，但仍然没有覆盖所有的函数。如果需要调用某个未被封装的函数，那么可以通过P/Invoke来实现。通过P/Invoke来调用函数的基本过程如下：

1）定义一个托管函数来与非托管的函数相对应。

2）用DllImport属性来修饰这个托管函数，表示它代表一个非托管函数。

3）调用托管代码函数，从而使CLR加载DLL并在调用阶段切换到非托管函数。

以下是一个简单的P/Invoke托管方法的示例，该方法对应于Win32的Beep函数（通过扬声器发出声音），其中Beep定义在kernel32.dll中。

```
BOOL WINAPI Beep(
    __in DWORD dwFreq,
    __in DWORD dwDuration
);

[DllImport("kernel32.dll", SetLastError=true)]
private static extern bool Beep(uint freq, uint dur);
```

DllImport属性用来表示这个函数对应于一个P/Invoke定义，并且这个指定的函数位于kernel32.dll中。而且，SetLastError表示这个函数在退出时设置最近的错误。我们接下来在调试器下观察这个P/Invoke函数的使用情况。清单7-1给出了一个简单的程序。

清单7-1 通过P/Invoke调用Beep函数的程序

```
using System;
using System.Text;
using System.Runtime.InteropServices;
using System.Runtime.Remoting;

namespace Advanced.NET.Debugging.Chapter7
{
    class BeepSample
    {
        static void Main(string[] args)
        {
            Beep(1000, 2000);
        }

        [DllImport("kernel32.dll", SetLastError=true)]
        private static extern bool Beep(uint freq, uint dur);
    }
}
```

清单7-1中的源代码和二进制文件位于以下文件夹中：

- 源代码文件：C:\ADND\Chapter7\Beep
- 二进制文件：C:\ADNDBin\07Beep.exe

在调试器下运行这个程序，并在kernel32.dll的Beep函数中设置一个断点。当断点触发时观察栈回溯，并分析运行时做了哪些工作来调用非托管代码：

```
0:000> bp kernel32!Beep
0:000> g
ModLoad: 75d10000 75dd6000   C:\Windows\system32\ADVAPI32.dll
ModLoad: 76fb0000 77073000   C:\Windows\system32\RPCRT4.dll
...
...
...
ModLoad: 76150000 76195000   C:\Windows\system32\iertutil.dll
ModLoad: 75b40000 75b54000   C:\Windows\system32\Secur32.dll
ModLoad: 77590000 775bd000   C:\Windows\system32\ws2_32.dll
ModLoad: 77560000 77566000   C:\Windows\system32\NSI.dll
ModLoad: 79060000 790b6000   C:\Windows\Microsoft.NET\Framework\
v2.0.50727\mscorjit.dll
Breakpoint 0 hit
eax=00123060 ebx=00448720 ecx=000003e8 edx=000007d0 esi=0029f160 edi=0029f390
eip=75e666e8 esp=0029f12c ebp=0029f148 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
KERNEL32!Beep:
75e666e8 6a2c            push    2Ch
0:000> !ClrStack
OS Thread Id: 0x146c (0)
```

```
ESP       EIP
0029f160 75e666e8 [NDirectMethodFrameStandalone: 0029f160]
 Advanced.NET.Debugging.Chapter7.BeepSample.Beep(UInt32, UInt32)
0029f170 00a80092 Advanced.NET.Debugging.Chapter7.BeepSample.Main(System.String[])
0029f390 79e7c74b [GCFrame: 0029f390]
```

从 ClrStack 命令的输出中可以看到，Main 方法正在调用我们的 P/Invoke Beep 函数。注意，与 Beep 方法对应的栈帧有着如下前缀：

```
[NDirectMethodFrameStandalone: 0029f160]
```

NDirectMethodFrameStandalone 表示正在执行从托管代码到非托管代码的切换。可以通过转储指定地址（0x0029f160）的内容来验证这一点：

```
0:000> dd 0029f160
0029f160  79e73620 0029f390 00123040 00a80092
0029f170  01e75874 79e7c74b 0029f180 0029f1c0
0029f180  0029f200 79e7c6cc 0029f250 00000000
0029f190  0029f220 00000000 0012c040 0029f1f0
0029f1a0  79f07fee 0029f390 be1faf39 0029f3dc
0029f1b0  0029f240 00000000 0029f390 00448720
0029f1c0  00000000 00000000 00000000 00000000
0029f1d0  00000001 00000000 79e7c1fa 0029f19c
```

第一个值看上去像一个代码地址，因此，我们使用 ln 命令来查看该地址能否被解析为代码：

```
0:000> ln 79e73620
(79e73620)   mscorwks!NDirectMethodFrameStandalone::'vftable'   |
  (79e73688)   mscorwks!NDirectMethodFrameSlim::'vftable'
Exact matches:
    mscorwks!NDirectMethodFrameStandalone::'vftable' = <no type information>
```

输出信息表明这个地址对应于对象 NDirectMethodFrameStandalone 的虚函数表。我们可以进一步将虚函数表转储出来，并在这个函数地址上使用 ln 命令来观察它所包含的内容：

```
0:000> dd 79e73620 l4
79e73620  79fc1a7d 79e730fc 7a0a3e0a 79e74034
0:000> ln 79fc1a7d
...
...
...
Exact matches:
    mscorwks!DelegateTransitionFrame::GcScanRoots = <no type information>
    mscorwks!NDirectMethodFrame::GcScanRoots = <no type information>
```

虽然 NDirectMethodFrame 的大部分信息并没有公开，但我们可以通过观察它所包含的各个函数来了解这个对象的信息。关键要记住，每当看到栈中有一条 NDirectMethodFrame 语句时，就应该知道这段代码刚刚从托管代码切换回非托管代码。

到目前为止，我们看到了一个非常简单的 P/Invoke 示例程序，以及在调试器中切换到非

托管代码的过程。切换栈帧（由 NDirectMethodFrame 对象表示）需要根据被调用非托管函数的复杂性来处理各种不同的模式。在这些模式中，或许最重要的就是在切换过程中发生的列集（marshaling）操作。列集是指在不同的数据表示形式之间的转换，这是因为在两种不同的环境（托管的和非托管的）中需要不同的表示形式。对于简单的数据类型，例如 int 和 bool，列集操作在很大程度上都是自动实现的，但对于其他一些更为复杂的类型，CLR 可能需要调用者来实现对数据的列集。我们来看一个稍微复杂的 P/Invoke 调用，如清单 7-2 所示。

清单 7-2 P/Invoke 示例

```
using System;
using System.Text;
using System.Runtime.InteropServices;
using System.Runtime.Remoting;

namespace Advanced.NET.Debugging.Chapter7
{
    class PInvoke
    {
        private const int TableSize = 50;

        [StructLayout(LayoutKind.Sequential, CharSet = CharSet.Ansi)]
        public class Node
        {
            public string First;
            public string Last;
            public string Social;
            public UInt32 Age;
        }

        [StructLayout(LayoutKind.Sequential)]
        public class Table
        {
            [MarshalAs(UnmanagedType.ByValArray, SizeConst = TableSize)]
            public IntPtr[] Nodes;

            public IntPtr Aux;
        }

        static void Main(string[] args)
        {
            PInvoke p = new PInvoke();
            p.Run();
        }
        public void Run()
        {
            Node[] nodes = new Node[TableSize];
            nodes[0] = new Node();
```

```
            nodes[0].First = "First Name 1";
            nodes[0].Last = "Last Name 1";
            nodes[0].Social = "Social 1";
            nodes[0].Age = 30;

            nodes[1] = new Node();
            nodes[1].First = "First Name 2";
            nodes[1].Last = "Last Name 2";
            nodes[1].Social = "Social 2";
            nodes[1].Age = 31;

            nodes[2] = new Node();
            nodes[2].First = "First Name 3";
            nodes[2].Last = "Last Name 3";
            nodes[2].Social = "Social 3";
            nodes[2].Age = 32;

            Table t = new Table();
            t.Aux = IntPtr.Zero;

            t.Nodes = new IntPtr[TableSize];
            for (int i = 0; i < TableSize && nodes[i] != null; i++)
            {
                int nodeSize = Marshal.SizeOf(typeof(Node));
                t.Nodes[i] = Marshal.AllocHGlobal(nodeSize);
                Marshal.StructureToPtr(nodes[i], t.Nodes[i], false);
            }

            int tableSize = Marshal.SizeOf(typeof(Table));
            IntPtr pTable = Marshal.AllocHGlobal(tableSize);
            Marshal.StructureToPtr(t, pTable, false);

            Myfunc(pTable);
        }

        [DllImport("05Native.dll")]
        private static extern void Myfunc(IntPtr ptr);
    }
}
```

清单 7-2 中的源代码和二进制文件位于以下文件夹中：

- 源代码文件：C:\ADND\Chapter7\PInvoke
- 二进制文件：C:\ADNDBin\07PInvoke.exe 与 05Native.dll

清单 7-2 中的源代码给出了一种略微复杂的 P/Invoke 调用。在程序中并不是将简单的数据类型传递给非托管函数，而是传递一个指向 Table 类型的指针，其中在 Table 类型中包含了一个数组，该数组的元素为 Node 类型。在 Node 类型中包含了一组基本的类型（string 和 int）。由于从托管代码传递到非托管代码的并不是简单的数据类型，因此我们不能再使用标准的列集器，而是必须提供特殊的指令来告知系统对这个复杂类型中的各个元素进行列集。

这些指令通常表现为类型或者成员的属性。例如，Node 类型和 Table 类型都有属性 StructLayout，表示类型的内存布局应该是串行的（意味着所有成员在内存中都是串行排列的）。此外，在 Table 类型中的 Nodes 数组有一个属性 MarshalAs，表示这个数组应该作为一个值类型数组。除了对传递给非托管代码的数据结构中的各个成员进行语法修饰外，我们还必须编写自定义的列集代码来确保为这个类型分配所需的内存，并调用方法 Marshal. StructureToPtr 将各个成员从这个数据类型复制到新分配的内存中。当编写完所有的必要代码后，就可以开始调用托管方法，并将传递执行 Table 类型的指针。显然，使用默认的列集操作要更为容易，但有时候仍然需要使用自定义的（或者显式的）列集操作，而编写自定义的列集操作是一项容易出错的任务。错误的列集操作将导致程序立即崩溃或者出现一些非常微妙的问题，并且这些问题不会立即显现出来。所以知道如何通过调试器来找出这些问题的根源是非常重要的。我们以清单 7-2 为例来说明调试列集代码的一些基本步骤。首先在调试器下启动 07Pinvoke. exe，并且在调用的非托管函数上设置一个断点（在模块 05Native. dll 中的 Myfunc 函数）。

```
0:000> bp 05native!MyFunc
Bp expression '05native!MyFunc' could not be resolved, adding deferred bp
0:000> g
ModLoad: 76730000 767f6000   C:\Windows\system32\ADVAPI32.dll
ModLoad: 761a0000 76263000   C:\Windows\system32\RPCRT4.dll
...
...
...
Breakpoint 0 hit
eax=000c3078 ebx=00249fe8 ecx=0026eb98 edx=79ec7f60 esi=001df258 edi=001df4bc
eip=6f2e1590 esp=001df228 ebp=001df240 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
05Native!Myfunc:
6f2e1590 55              push    ebp
```

在执行到这个断点后，可以观察传递给这个函数的参数：

```
0:000> dv
            ptr = 0x0026eb98
0:000> dd 0x0026eb98 l3
0026eb98  00270078 00270090 002700a8
0:000> dd 00270078 l4
00270078  00272a48 0025eb40 0025eda0 0000001e
0:000> da 00272a48
00272a48  "First Name 1"
```

命令 dv 将显示传递给这个函数的参数（0x0026eb98），该参数对应一个 Table 结构的实例。然后，我们使用命令 dd 来显示指向数组中各个节点的指针。接下来，再次使用命令 dd 来显示数组中第一个节点的内容，并使用命令 da 来显示第一个元素，这是一个字符串。要验证这个参数的完整性和正确性，可以继续使用相同的策略来验证剩余的节点和元素。在被调用的非托管代码中设置一个断点后，接着需要通过这组转储命令来验证所传递的数据是正确的。

到目前为止，我们已经介绍了调用非托管代码的 P/Invoke 方法，接下来将介绍另一个常用的层，称之为 COM 互用性层。

7.2 COM

组件对象模型（Component Object Model，COM）是 Microsoft 在 1993 年引入的一种二进制接口。它提供了一种通用的方式来定义与语言无关的组件，并且 COM 组件可以跨越机器的边界来创建和使用。COM 是作为一种*标准*引入的，它在 Windows 平台上取得了巨大的成功，因此 COM 对象的数量总是在持续不断地快速增长。基于 COM 成功应用以及.NET 的出现，Microsoft 引入了一种关键的功能来实现托管代码与现有 COM 对象的交互，这种功能也被称为 COM 互用性。COM 互用性是与非托管代码交互的一种方式，也就是被实现为 COM 对象的非托管代码。这种交互可以是双向的，因为托管代码可以调用现有的非托管代码 COM 对象，而非托管代码也可以调用以 COM 对象形式出现的托管对象。我们将侧重于介绍 COM 互用性的最常见类型，也就是.NET 如何实现托管代码调用非托管代码的 COM 对象。

运行时可调用封装

在本章的前面介绍了 P/Invoke 层以及通过 P/Invoke 层与非托管模块（DLL）进行交互的方法。在 P/Invoke 层中使用的算法如下所示：

1）将指定的模块（DLL）加载到进程的地址空间。

2）找到所需函数的地址。

3）对数据进行列集。

4）调用函数。

COM 互用性要略微复杂一些，因为它有自己的一套注册、实例化，生命周期管理以及列集等过程。在使用 COM 互用性层时，并不要求每个托管代码程序都必须实现相应的代码来查找，实例化和获取接口以及管理列集操作，COM 互用性层将生成所有必要的代码以及一种互调用程序集（interop assembly）。互调用程序集可以看成是一种托管代理（manager proxy），这种代理指向底层的 COM 对象。为了更好地理解这种机制的工作方式，我们使用一个简单的 COM 对象，在这个对象中导出一个方法 Add。我们无需了解实际的非托管代码，而只需列出这个 COM 对象的接口定义即可，如清单 7-3 所示。

清单 7-3 COM 对象的 IDL

```
interface IBasicMath : IUnknown
{
   [helpstring("method Add")]
   HRESULT Add([in] LONG num1, [in] LONG num2, [out] LONG* res);
};
```

清单7-3中的二进制文件位于以下文件夹中：

- 二进制文件：C:\ADNDBin\07PInvoke07ComObj.dll

注册COM二进制文件

所有的COM对象必须首先被注册到注册表中某个已知的位置上，这样COM系统才可以找到相应的二进制文件。可以使用regsvr32.exe来注册一个COM对象。例如，要使用清单7-3中的示例COM对象，必须首先运行以下命令：

```
regsvr32 07ComObj.dll.
```

Add方法的输入参数为两个数值，运算结果会被保存到输出参数中。在清单7-4的托管代码中使用了清单7-3中的COM对象。

清单7-4 使用COM对象的托管代码

```
using System;
using System.Text;
using System.Runtime.InteropServices;
using System.Runtime.Remoting;
using COMInterop;

namespace Advanced.NET.Debugging.Chapter7
{
    class COMInteropSample
    {
        [STAThread]
        static void Main(string[] args)
        {
            int result;
            BasicMathClass s = new BasicMathClass();
            s.Add(1, 2, out result);
            Console.WriteLine("Result= " + result);
        }
    }
}
```

清单7-4中的源代码和二进制文件位于以下文件夹中：

- 源代码文件：C:\ADND\Chapter7\COMInterop
- 二进制文件：C:\ADNDBin\07COMInterop.exe and 07Managed.dll

调用非托管代码COM对象是一种非常简单的过程。首先实例化一个BasicMathClass类型，然后调用相应的Add方法。BasicMatchClass这个类型定义在何处？答案是位于主互调用程序集（Primary Interop Assembly，PIA）中。在清单7-4中可以看到，我们通过一个using语句来引入COMInterop命名空间。这个命名空间及其包含的类型是通过使用工具tlbimp.exe（属于.NET SDK的一部分）来生成的。Tlbimp.exe可以利用这个COM二进制文件来生成一

个 PIA。在前面的示例中，可以运行以下命令来生成 07Managed. dll 的 PIA：

```
TlbImp.exe 07ComObj.dll /out:07Managed.dll /namespace:COMInterop
```

到目前为止，我们已经介绍了在 COM 互用性中包含的三个实体：COM 二进制文件、托管客户端、PIA。此外，还需要使用第四个实体，这个实体是在运行时被创建的，即运行时可调用封装（（Runtime Callable Wrapper，RCW）。图 7-1 说明了这四个实体一起实现互调用的过程。

如图 7-1 所示，首先是托管客户端调用在 COM 对象中定义的方法，该对象是在 PIA 中定义的。CLR 通过来自 PIA 的信息创建 RCW 的实例（每个 CoClass 创建一个实例）。然后，RCW 截获对这个方法的调用，将参数转义为非托管类型，切换环境，并且调用非托管代码中的方法。

RCW 的另一个功能是负责处理底层 COM 对象的生命周期。COM 对象的生命周期是通过一种引用计数模式来管理的，这意味着每当获取对象的一个接口时，引用计数就会递增。相反地，当不再需要一个接口时，引用计数会递减。当引用计数降为 0 时，就可以销毁对象。RCW 能跟踪引用的数量，并确保相应地递增/降低引用计数。当托管客户端使用完 RCW 并且不存在未释放的引用后，RCW 会被回收，而所有相关的 COM 对象都会被释放。

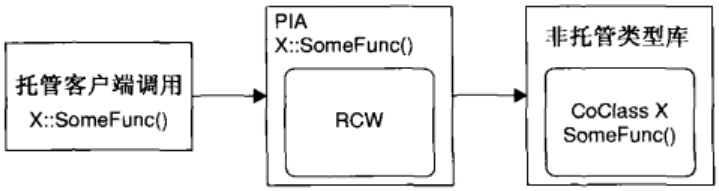


图 7-1　在 COM 中涉及的四个主要实体

可靠地释放 COM 对象

当不存在对 RCW 的引用后，RCW 会递减并且清除对底层 COM 对象的任何引用，因而 COM 对象直到垃圾收集器清除 RCW 才会被清除。有一种方式可以强制释放 COM 对象，即使用 Marshal. ReleaseComObject 方法。

在调试器下运行清单 7-4 中的程序（07ComInterop. exe），并且观察能否通过这个调试器来了解 COM 互用性调用的详细信息。首先在 COM 对象的 Add 函数的非托管函数上设置一个断点，待程序触发断点后，转储出托管调用栈：

```
0:000> bp 07ComObj!CBasicMath::Add
Bp expression '07ComObj!CBasicMath::Add' could not be resolved, adding deferred bp
0:000> g
ModLoad: 76730000 767f6000   C:\Windows\system32\ADVAPI32.dll
ModLoad: 761a0000 76263000   C:\Windows\system32\RPCRT4.dll
...
...
...
```

```
Breakpoint 0 hit
eax=5c701573 ebx=004d8610 ecx=00000003 edx=5c718704 esi=0025efa8 edi=0025f1ec
eip=5c707900 esp=0025ef60 ebp=0025ef90 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
07ComObj!CBasicMath::Add:
5c707900 55              push    ebp
0:000> !ClrStack -a
OS Thread Id: 0x26f4 (0)
ESP       EIP
0025efa8 5c707900 [ComPlusMethodFrameStandaloneCleanup: 0025efa8]
 COMInterop.BasicMathClass.Add(Int32, Int32, Int32 ByRef)
0025efc0 004800b8 Advanced.NET.Debugging.Chapter7.COMInteropSample.Main
(System.String[])
    PARAMETERS:
        args = 0x01f358c8
    LOCALS:
        0x0025efc4 = 0x00000000
        <CLR reg> = 0x01f36268
0025f1ec 79e7c74b [GCFrame: 0025f1ec]
```

在托管栈的最顶层栈帧中给出了对 PIA 中 Add 方法的调用，这将在后台处理非托管代码的切换（从 ComPlusMethodFrameStandaloneCleanup 可以看出来）。

一些 SOS 命令可以用来获取与 COM 互用性相关的信息。首先，本书曾经介绍过，有些 SOS 命令可以给出套间类型的信息。以下是在调试会话中使用 threads 命令的输出：

```
0:000> !threads
ThreadCount: 2
UnstartedThread: 0
BackgroundThread: 1
PendingThread: 0
DeadThread: 0
Hosted Runtime: no
                                        PreEmptive   GC Alloc           Lock
       ID OSID ThreadOBJ    State      GC        Context        Domain
   Count APT Exception
   0     1 26f4 004d8610      6020 Enabled  01f36278:01f38004 004d1328     0 STA
   2     2 2a00 004da6c0      b220 Enabled  00000000:00000000 004d1328     0 MTA
 (Finalizer)
```

套间是一种逻辑结构，与 COM 线程模型紧密相关。COM 的主要功能之一是，允许有着不同线程需求的组件在一起工作。例如，如果在编写某个 COM 组件时没有考虑并发调用情况，那么可以使用单线程套间（STA），这种套间会使 COM 子系统对所有这个组件的调用串行化。相反，能够处理并发调用的组件可以使用多线程套间（MTA）模型，在这种情况下，COM 将不会对这个组件的调用串行化。当任何一个线程使用 COM 组件时，它必须选择相应的套间模型。在默认情况下，所有的.NET 线程都在 MTA 模型中。在清单 7-4 中，我们修改了主线程的套间模型：在 Main 方法上使用 STAThread 属性（这是因为 COM 对象本身被注册

为使用 STA 模型）。在 threads 命令输出的 APT 列中，给出了线程正在使用的套间模型（STA，MTA）。

另一个可以给出与 COM 互用性相关信息的命令是 syncblk 命令。

```
0:000> !syncblk
Index SyncBlock MonitorHeld Recursion Owning Thread Info  SyncBlock Owner
-----------------------
Total           1
CCW             0
RCW             1
ComClassFactory 0
Free            0
```

在 syncblk 命令的输出中给出了 CLR 已经实例化的并且当前处于活跃状态的 RCW 数量。当希望快速了解当前 RCW 的使用情况时，这个数值很有用。

最后一个需要注意的命令是 COMState。COMState 命令能对进程中的每个线程输出详细的 COM 信息：

```
0:000> !COMState
     ID      TEB    APT    APTId CallerTID Context
  0 2ff8 7ffde000 STA      a84         0 0024b228
  1 2c14 7ffdd000 Ukn
  2  508 7ffdc000 MTA        0         0 0024b398
  3 2400 7ffdb000 Ukn
  4 2eb8 7ffda000 Ukn        0         0 00000000
```

在输出中，需要注意 APT 列，它列出了线程的套间模型。

现在，我们已经介绍了托管代码程序如何调用非托管代码 COM 对象，以及在调试器下观察与这个互用性相关的各种信息。接下来，我们将分析在使用 P/Invoke 和 COM 互用性层时最常见的错误，以及如何通过学到的知识和调试技巧来找出问题的根源。

7.3　P/Invoke 调用的调试

在本章的前面部分，我们介绍过托管代码如何通过 P/Invoke 层来调用非托管代码。调用过程的复杂性完全取决于被调用非托管函数的复杂性。例如，我们必须确保定义了正确的托管代码原型来表示非托管代码，或者在默认列集器无法满足参数传递的情况下，我们必须自行编写代码来实现列集操作。在编写这些自定义的列集代码时要非常小心，避免在程序中产生一些微妙的错误。在本书的这一节中，我们将讨论一些从托管代码中不正确地调用非托管代码的示例，这些问题表现出来的现象，以及如何通过调试器和工具来高效地找出问题。

7.3.1　调用约定

调用约定（calling conventions）是指在主调函数和被调函数之间的一种契约。调用约定包含了在实现正确的调用时主调方和被调方都认可的一组规则。表 7-1 给出了一些调用约定，

这些调用约定之间的主要差异在于如何将参数传递给被调函数以及如何对栈进行清理。

表7-1 调用约定

调用约定	参数传递	负责清理的函数	DllImport的CallingConvention域
Stdcall	栈（从右到左）	被调函数	StdCall
Cdecl	栈（从右到左）	主调函数	Cdecl
Fastcall	寄存器/栈（从右到左）	被调函数	FastCall（不支持）
Thiscall	寄存器/栈（从右到左）	被调函数	ThisCall

当使用P/Invoke来调用非托管函数时，一定要使用正确的调用约定，否则程序会出现一些难以发现的问题。在默认情况下，P/Invoke层使用Winapi调用约定，从严格意义上来说，这并不是一种调用约定，而是告诉运行时使用默认平台调用约定。例如，在Windows上，默认的平台调用约定是StdCall，而在Windows CE上则是Cdecl。此外，你还可以通过DllImport属性的CallingConvention域来指定一种不同的调用约定。在表5-1中的最后一列给出了在非托管调用约定和DllImport CallingConvention域之间的映射关系。

来看一个执行P/Invoke调用的程序。我们将使用与清单7-2相同的程序，只不过对P/Invoke原型做出了一些修改来说明程序在显式指定调用约定情况下的行为。

清单7-2中的源代码和二进制文件位于以下文件夹中：

- 源代码文件：C:\ADND\Chapter7\Sig
- 二进制文件：C:\ADNDBin\07Sig.exe and 05Native.dll

运行这个程序，很快便抛出了一个异常：

```
C:\ADNDBin>07Sig.exe

Unhandled Exception: System.AccessViolationException: Attempted to
 read or write protected memory. This is often an indication that
 other memory is corrupt.
   at Advanced.NET.Debugging.Chapter7.PInvoke.Myfunc(IntPtr ptr)

   at Advanced.NET.Debugging.Chapter7.PInvoke.Run() in
c:\Publishing\ADND\Code\Chapter7\Sig\07Sig.cs:line 72
   at
Advanced.NET.Debugging.Chapter7.PInvoke.Main(String[] args) in
c:\Publishing\ADND\Code\Chapter7\Sig\07Sig.cs:line 33
```

抛出的异常表示出现了一个访问违例。访问违例异常表示某个内存访问指令是无效的。调试这种类型问题的最简单方法就是在调试器下运行这个程序，直到发生访问违例，然后通过观察程序的状态来找出问题的根源：

```
...
...
...
(21a0.2608): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
```

```
This exception may be expected and handled.
eax=00000000 ebx=00078570 ecx=deda8173 edx=79ec7f60 esi=002deeb8 edi=002dee88
eip=6c7615c6 esp=002dedb0 ebp=002dee88 iopl=0         nv up ei pl nz ac po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010212
05Native!Myfunc+0x36:
6c7615c6 833c8100        cmp     dword ptr [ecx+eax*4],0 ds:0023:deda8173=????????
```

正如我们所料，当执行位于地址 0x6c7615c6 处的 cmp 指令时出现了第一轮访问违例异常。我们能否找出这个 cmp 指令对应哪一行源代码？在调试器中输出源代码行信息的最简单方式就是使用 lines 命令：

```
0:000> !lines
Line number information will be loaded
```

现在，每当要求调用栈的源代码位置和行号时，都会包含这些信息（如果存在私有符号）：

```
0:000> k
ChildEBP RetAddr
002dee88 001ba307 05Native!Myfunc+0x36
[c:\workzone\05native\05native\05native.cpp @ 75]
WARNING: Frame IP not in any known module. Following frames may be wrong.
002def00 79e7c74b 0x1ba307
002def10 79e7c6cc mscorwks!CallDescrWorker+0x33
002def90 79e7c8e1 mscorwks!CallDescrWorkerWithHandler+0xa3
002df0cc 79e7c783 mscorwks!MethodDesc::CallDescr+0x19c
002df0e8 79e7c90d mscorwks!MethodDesc::CallTargetWorker+0x1f
...
...
...
```

从输出信息中，我们可以看到源文件为 05native. cpp，行号为 75。以下是函数 Myfunc 的源代码：

```
__declspec(dllexport) VOID Myfunc(TABLE* ptr)
{
[75]          for(int i=0; ptr->Coll[i]!=NULL; i++)
      {
            printf("First: %s, Last: %s, Social: %s, Age: %d\n",
                  ptr->Coll[i]->First,
                  ptr->Coll[i]->Last,
                  ptr->Coll[i]->Social,
                  ptr->Coll[i]->Age);
      }
}
```

根据我们看到的 cmp 指令以及第 75 行对应的 for 循环，可以假设正在执行的是 for 循环的条件表达式（ptr -> Coll [i]! = NULL）。我们进一步分解这个条件表达式。指针 ptr 被作为参数传递给这个函数，因此我们可以使用命令 dv 来获得它的地址：

```
0:000> dv /V
002dee80 @ebp-0x08                i = 0
002dee90 @ebp+0x08              ptr = 0xdeda8173
```

从输出信息中，我们可以看到指针的值为0xdeda8173。由于这个指针值对应于TABLE结构，而这个结构又包含了一个节点数组，因此我们认为这个指针指向的内存包含了一定数量的指针，并且每个指针都对应于一个节点实例。如果将这个指针转储出来，可以看到以下内容：

```
0:000> dd 0xdeda8173
deda8173  ???????? ???????? ???????? ????????
deda8183  ???????? ???????? ???????? ????????
deda8193  ???????? ???????? ???????? ????????
deda81a3  ???????? ???????? ???????? ????????
deda81b3  ???????? ???????? ???????? ????????
deda81c3  ???????? ???????? ???????? ????????
deda81d3  ???????? ???????? ???????? ????????
deda81e3  ???????? ???????? ???????? ????????
```

这不是一种正常现象，输出信息中的问号表示内存的内容是无效的。那么这个指针值从何而来？我们知道，这个指针来自托管代码程序。如果观察非托管函数Myfunc的原型，可以看到函数的声明为：

```
__declspec(dllexport) VOID Myfunc(TABLE* ptr)
```

由于在函数原型中没有显式地声明调用约定，因此会使用默认的调用约定cdecl（参数按照从右到左的顺序传递到栈上）。根据命令dv的输出，我们认为这个参数会被放到栈位置ebp+0x8上。然而，我们清楚地看到这个指针值是无效的。最大的问题仍然存在：正确的指针位于何处？我们重新启动这个程序，并在发生访问违例之前在Myfunc上设置一个断点。当断点触发后，再次观察栈的状态：

```
...
...
...
Breakpoint 0 hit
eax=001d3078 ebx=002b8548 ecx=002d27f8 edx=79ec7f60 esi=001bef08 edi=001bf174
eip=6c761590 esp=001beedc ebp=001beef0 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
05Native!Myfunc:
6c761590 55              push    ebp
0:000> dv /V
001beee0 @ebp+0x08              ptr = 0xb87bc271
0:000> dd 0xb87bc271 l1
b87bc271  ????????
```

我们再次看到这个指针指向无效的内存（因为内存中的值都是问号）。然而，我们要注意寄存器中的值。例如，如果查看寄存器ecx指向的内存，可以看到以下信息：

```
0:000> dd 002d27f8
002d27f8  002dcd20 002d2678 002d2738 00000000
002d2808  00000000 00000000 00000000 00000000
002d2818  00000000 00000000 00000000 00000000
002d2828  00000000 00000000 00000000 00000000
002d2838  00000000 00000000 00000000 00000000
002d2848  00000000 00000000 00000000 00000000
002d2858  00000000 00000000 00000000 00000000
002d2868  00000000 00000000 00000000 00000000
0:000> dd 002dcd20 l4
002dcd20  002dab80 002d1148 002d1178 0000001e
0:000> da 002dab80
002dab80  "First Name 1"
```

ecx 指针的内容看上去非常类似于我们预想的函数参数，然而，这个函数似乎正从栈中获得一个无效的指针。在这种情况下，要牢记：在参数被传递给函数时，有一种调用约定会使用 ecx 寄存器，即 ThisCall。ThisCall 调用约定总是将“this”指针放在 ecx 寄存器中，并把剩余的参数放到栈上。是不是 ThisCall 调用约定导致了与非托管函数调用约定（cdecl）之间的不匹配？要回答这个问题，我们先来看 P/Invoke 函数的托管代码方法原型，如下所示：

```
[DllImport("05Native.dll", CallingConvention = CallingConvention.ThisCall)]
private static extern void Myfunc(IntPtr ptr);
```

从函数原型中可以清楚地看到，由于没有正确地指定 Myfunc 函数的调用约定，因而导致了调用约定之间的不匹配，使得非托管代码函数在栈上查找参数，而实际上这个参数是通过 ecx 寄存器来传递的。有大量的调试问题都是由于调用约定不匹配造成的。幸运的是，有一个 MDA 能够简化这种调试过程，即 pInvokeStackImbalance。这个 MDA 可以用来找出特定类型的调用约定不匹配问题，要启用它，可以使用以下配置文件（参见第 1 章来了解如何启动 MDA）：

```
<?xml version="1.0" encoding="UTF-8" ?>
<mdaConfig>
  <assistants>
    <pInvokeStackImbalance  />
  </assistants>
</mdaConfig>
```

请注意，这个 MDA 只会发现那些导致栈不平衡的调用约定不匹配问题，有些特殊类型的调用约定不匹配问题（例如 ThisCall 和 Cdecl）是无法发现的，在这种情况下就需要手动进行调试。

7.3.2　委托

我们已经讨论了如何与同步的非托管代码函数进行互调用，那么如何与异步的非托管代码函数进行互调用（在异步操作中需要通过一个函数指针来进行调用）？托管代码函数能否

作为一个指向非托管代码的函数指针来传递？绝对可以。P/Invoke 层可以获得一个托管代码委托，并将它转换为一个函数指针，然后由非托管函数来使用。我们来看一个示例。清单 7-5 是一个托管代码示例程序，它将调用以下异步的非托管方法：

```
typedef void (__stdcall  *PCALLBACK) (ULONG result);

__declspec(dllexport) VOID __stdcall AsyncProcess(PCALLBACK ptr)
```

这个非托管函数是 AsyncProcess，它的参数包括一个函数指针，指向的函数带有一个ULONG参数，表示操作的执行结果。AsyncProcess 函数的基本思想是，立即返回且并行地执行其他工作（创建另一个线程）。当执行完工作时，工作线程会用异步操作的结果来调用指定的函数。在清单 7-5 中给出了使用这个函数的托管程序。

清单 7-5 调用非托管异步函数的托管程序

```
using System;
using System.Text;
using System.Runtime.InteropServices;
using System.Runtime.Remoting;
using System.Threading;

namespace Advanced.NET.Debugging.Chapter7
{
    class PInvoke
    {
        public delegate void Callback(int result);

        public static void Main()
        {
            Callback c = new Callback(MyCallback);

            AsyncProcess(c);

            //执行一些工作
            Console.WriteLine("Press any key to exit");
            Console.ReadKey();

        }

        public static void MyCallback(int result)
        {
            Console.WriteLine("Result= " + result);
        }

        [DllImport("05Native.dll")]
        private static extern void AsyncProcess(Callback callBack);
    }
}
```

清单 7-5 中的源代码和二进制文件位于以下文件夹中：

- 源代码文件：C:\ADND\Chapter7\Callback
- 二进制文件：C:\ADNDBin\07Callback. exe and 05native. dll

这个程序很简单。它创建了一个委托实例来匹配非托管函数需要的函数指针，然后用这个委托实例来调用非托管函数，并在程序结束之前向用户提示“press any key to exit”。当非托管函数执行完成时将调用这个委托，指定执行操作的结果，然后将结果输出到控制台。请注意，在清单 7-5 中有意省去了托管程序中的部分代码以便更好地说明这个问题。

运行这个程序，可以看到它快速完成并提示“press any key to exit”。如果在程序完成后快速按下任意键，那么程序会很快结束而不会显示异步操作的结果。如果等待这个异步操作几秒钟的时间，那么也不会看到结果，而是出现对话框“07Callback. exe Has Stopped Working”。是什么问题导致了这个简单的程序崩溃？根据事件序列进行判断，程序似乎成功地调用了 AsyncProcess 函数，然后等待将结果传递给委托实例。为了了解更多的信息，我们在调试器下运行这个程序：

```
...
...
...
Press any key to exit
(2eb4.36a8) : Unknown exception - code c0000096 (first chance)
(2eb4.36a8): Unknown exception - code c0000096 (!!! second chance !!!)
eax=00000000 ebx=00000000 ecx=765ac379 edx=77629a94 esi=04b3f7a8 edi=04b3f874
eip=00bd0a22 esp=04b3f7a0 ebp=04b3f874 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
00bd0a22 ee              out     dx,al
```

从输出信息中可以看到抛出了一个异常，异常码为 0xc0000096。如何找出这个异常对应的信息？可以使用 error 命令，并将这个异常码指定为参数：

```
0:003> !error c0000096
Error code: (NTSTATUS) 0xc0000096 (3221225622) - {EXCEPTION}
  Privileged instruction.
```

这个异常表示由于正在执行某个特权指令而发生了错误。我们将出错线程的调用栈转储出来，并进一步了解这个特权指令的执行是从何而来的：

```
0:003> !lines
Line number information will be loaded
0:003> k
ChildEBP RetAddr
WARNING: Frame IP not in any known module. Following frames may be wrong.
04b3f79c 6de4169c 0xbd0a22
04b3f874 765a4911 05Native!Helper+0x3c
 [c:\workzone\05native\05native\05native.cpp @ 92]
04b3f880 7760e4b6 KERNEL32!BaseThreadInitThunk+0xe
04b3f8c0 7760e489 ntdll!__RtlUserThreadStart+0x23
04b3f8d8 00000000 ntdll!_RtlUserThreadStart+0x1b
```

从栈中可以看到，在非托管函数中创建的辅助线程执行了 Helper 函数，而这个函数（在第 92 行）又调用了位于地址 0xbd0a22 处的代码。如果查看 Helper 函数的代码，可以看到以下内容：

```
DWORD WINAPI Helper(LPVOID lpParam)
{
      Sleep(2000);
[92]  g_pfunc(5);
      return 1;
}
```

第 92 行对应一个函数调用，函数的地址保存在全局变量 g_pfunc 中。如果我们查看最初调用的 AsyncProcess 函数，那么会看到它把传递进来的参数的函数指针保存在全局变量 g_pfunc 中。根据这些观察信息，我们可以得出结论：这个托管程序调用 AsyncProcess 方法（将委托传递进去），把函数指针保存在全局变量中，创建一个新的线程来处理这个请求，然后立即返回。新的线程会执行这个工作，并且尝试回调（Call Back）保存在全局变量的函数指针。现在，让我们来看看导致异常抛出的代码：

```
0:003> u 0xbd0a22
00bd0a22 ee              out     dx,al
00bd0a23 fe              ???
00bd0a24 ee              out     dx,al
00bd0a25 fe              ???
00bd0a26 ee              out     dx,al
00bd0a27 fe              ???
00bd0a28 ee              out     dx,al
00bd0a29 fe              ???
```

我们可以看到，位于地址 0xbd0a22 处的指令是无效的（在汇编代码中看到??? 绝不是一件好事）。为什么最终会得到一个无效的函数地址？在调试器下重新运行这个程序，并在 AsyncProcess 函数上设置一个断点，查看传递进去的函数指针（也就是委托）是否正确：

```
0:000> bp 05native!AsyncProcess
0:000> g
...
...
...
Breakpoint 0 hit
eax=6de01050 ebx=00385c90 ecx=79ee2457 edx=80000000 esi=001df240 edi=001df1d8
eip=6de01d40 esp=001df194 ebp=001df228 iopl=0         nv up ei ng nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000286
05Native!AsyncProcess:
6de01d40 55              push    ebp
0:000> dv
            ptr = 0x008d0a22
        hThread = 0xffffffff
           dwId = 0x1df150
0:000> u 0x008d0a22
008d0a22 b8080a8d00      mov     eax,8D0A08h
```

```
008d0a27 e948000000      jmp     008d0a74
008d0a2c ab              stos    dword ptr es:[edi]
008d0a2d ab              stos    dword ptr es:[edi]
008d0a2e ab              stos    dword ptr es:[edi]
008d0a2f ab              stos    dword ptr es:[edi]
008d0a30 ab              stos    dword ptr es:[edi]
008d0a31 ab              stos    dword ptr es:[edi]
```

断点触发后，在局部变量的输出（通过命令 dv）中给出了这个函数指针为 0x008d0a22。对这个地址进行反汇编后，将看到一个 mov 指令以及紧接着的 jmp 指令。这看上去比之前的反汇编代码更为合理。这两个指令就形成了所谓的转换（thunk）代码，每当从非托管代码调用托管代码时，P/Invoke 层都会增加这段代码。这非常类似于从托管代码进入到非托管代码（例如列集数据）时 P/Invoke 层必须完成的工作。此时，我们已经验证了，当调用 AsyncProcess 时，从托管代码程序传递给非托管 AsyncProcess 函数的委托确实是有效的。在执行完调用后，函数指针指向的代码会发生变化，并在 Helper 函数使用它时使程序崩溃。我们在 Helper 函数上设置一个断点，再次判断函数指针的状态：

```
0:000> bp 05native!Helper
0:000> g
Breakpoint 1 hit
eax=765a48ff ebx=00000000 ecx=00000000 edx=6de011c2 esi=00000000 edi=00000000
eip=6de01660 esp=0496ff88 ebp=0496ff90 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
05Native!Helper:
6de01660 55              push    ebp
0:003> X 05native!g_pfunc
6de07138 05Native!g_pfunc = 0x008d0a22
```

在这个断点触发后，使用 X 命令来解析 g_pfunc 符号得到保存这个函数指针的地址（0x008d0a22）。接下来，对这个地址进行反汇编来验证这些指令仍然是正确的：

```
0:003> u 0x008d0a22
008d0a22 b8080a8d00      mov     eax,8D0A08h
008d0a27 e948000000      jmp     008d0a74
008d0a2c ab              stos    dword ptr es:[edi]
008d0a2d ab              stos    dword ptr es:[edi]
008d0a2e ab              stos    dword ptr es:[edi]
008d0a2f ab              stos    dword ptr es:[edi]
008d0a30 ab              stos    dword ptr es:[edi]
008d0a31 ab              stos    dword ptr es:[edi]
```

输出信息与之前的反汇编代码相同，并且可以得出结论，thunk 指令仍然有效。我们将执行的最后一个检查就是，在调用这个地址的位置上设置一个断点，再最后检查一次 thunk 代码：

```
0:003> u 05native!Helper
05Native!Helper:
6de01660 55              push    ebp
```

```
6de01661 8bec            mov     ebp,esp
6de01663 81ecc0000000    sub     esp,0C0h
6de01669 53              push    ebx
6de0166a 56              push    esi
6de0166b 57              push    edi
6de0166c 8dbd40ffffff    lea     edi,[ebp-0C0h]
6de01672 b930000000      mov     ecx,30h
0:003> u
05Native!Helper+0x17:
6de01677 b8cccccccc      mov     eax,0CCCCCCCCh
6de0167c f3ab            rep stos dword ptr es:[edi]
6de0167e 8bf4            mov     esi,esp
6de01680 68d0070000      push    7D0h
6de01685 ff158481e06d    call    dword ptr [05Native!_imp__Sleep (6de08184)]
6de0168b 3bf4            cmp     esi,esp
6de0168d e8a4faffff      call    05Native!ILT+305(__RTC_CheckEsp) (6de01136)
6de01692 8bf4            mov     esi,esp
0:003> u
05Native!Helper+0x34:
6de01694 6a05            push    5
6de01696 ff153871e06d    call    dword ptr [05Native!g_pfunc (6de07138)]
6de0169c 3bf4            cmp     esi,esp
6de0169e e893faffff      call    05Native!ILT+305(__RTC_CheckEsp) (6de01136)
6de016a3 b801000000      mov     eax,1
6de016a8 5f              pop     edi
6de016a9 5e              pop     esi
6de016aa 5b              pop     ebx
0:003> bp 6de01696
0:003> g
Press any key to exit
Breakpoint 1 hit
eax=00000000 ebx=00000000 ecx=765ac379 edx=77629a94 esi=0485fbd0 edi=0485fc9c
eip=6de01696 esp=0485fbcc ebp=0485fc9c iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
05Native!Helper+0x36:
6de01696 ff153871e06d    call    dword ptr [05Native!g_pfunc
 (6de07138)] ds:0023:6de07138=00ca0a22
0:003> u 0xbd0a22
00bd0a22 ee              out     dx,al
00bd0a23 fe              ???
00bd0a24 ee              out     dx,al
00bd0a25 fe              ???
00bd0a26 ee              out     dx,al
00bd0a27 fe              ???
00bd0a28 ee              out     dx,al
00bd0a29 fe              ???
```

这时，我们可以看到 thunk 代码已经无效。现在，我们可以得出结论，当 Helper 函数第一次执行时，一切都看似正常，但当实际调用它时，指针值发生变化而变得无效。为什么会出现这种情况？要理解所发生的问题，首先要记住委托同样是一个托管对象。委托实例位于

托管堆上，并且在垃圾收集操作期间要遵从相同的 GC 规则，即委托实例可能会被收集（如果不存在根对象引用）或者被移动到一个不同的位置。我们曾在清单 7-5 中省略了一些代码，这些代码包括：在调用了 AsyncProcess 之后，释放对这个委托实例的引用并且调用 GC，从而导致对象委托被清除。而且，委托实例的清除是在非托管的 Helper 函数执行回调之前进行的。这种行为导致的结果是，当执行回调时，会调用一块不再包含有效委托的内存。在这种情况下，我们实际上是非常幸运的，因为程序抛出了一个异常并结束。而在其他一些情况中，发生的问题可能是无效的代码对内存进行随机写入，并且不会直接表现为程序出错，这就会使问题的分析过程变得更为漫长。

这个问题的正确解决方案是什么？答案就是，在从托管代码切换到非托管代码的整个切换过程中，要确保所使用的任何对象都被固定住（请参见第 5 章了解关于固定对象的更多信息）。虽然 P/Invoke 层在执行 P/Invoke 调用时能自动地固定对象，但这些对象在函数调用完成之后会被自动解除固定并切换回托管代码。在同步调用的情况下，这种方式能够发挥作用，但在异步调用的情况下，在非托管代码中可能仍然存在对委托的引用，并在稍后的某个时间（例如在最初的调用返回之后）使用这个委托。

当非托管代码使用一个已被收集的委托时，表现出的问题现象很难琢磨，往往需要一定时间才能暴露出来，那么有没有其他方式能够尽快地将这个问题暴露出来？答案是肯定的，这要借助一个 MDA。callbackOnCollectedDelegate 这个 MDA 维持了一组已被收集委托的 thunk 代码，并且每当调用一个已被收集的委托时，MDA 就会检查这个 thunk 列表，如果发现调用了某个已收集的委托，那么会报告一个错误。要启用这个 MDA，可以使用以下配置文件：

```
<mdaConfig>
  <assistants>
    <callbackOnCollectedDelegate listSize="1500" />
  </assistants>
</mdaConfig>
```

listSize 属性对应于 MDA 在内存中维持的 thunk 数量。如果在程序中启用这个 MDA，并且在调试器中运行程序，那么当调用这个已被收集的委托时，会看到以下输出：

```
Press any key to exit
ModLoad: 60340000 60348000   C:\Windows\Microsoft.NET\Framework\
v2.0.50727\culture.dll
<mda:msg xmlns:mda="http://schemas.microsoft.com/CLR/2004/10/mda">
  <!--
       A callback was made on a garbage collected delegate of type
       '07Callback!Advanced.NET.Debugging.Chapter7.PInvoke+Callback::Invoke'. This
may
       cause application
crashes, corruption and data loss. When passing delegates to
       unmanaged code,
 they must be kept alive by the managed application until it is
       guaranteed that they will never be called.
   -->
```

```
  <mda:callbackOnCollectedDelegateMsg break="true">
    <delegate
 name="07Callback!Advanced.NET.Debugging.Chapter7.PInvoke+Callback::Invoke"/>
  </mda:callbackOnCollectedDelegateMsg>
</mda:msg>
```

在这个 MDA 的输出中很清楚地指出了，程序调用了一个已被收集的委托，并且给出了这个委托的类型。在调试任何与非托管代码互用性相关的问题时，我强烈建议启用这个 MDA，否则要找出由于互用性而导致的问题会非常困难。

委托的默认调用约定是什么

委托使用的调用约定是 stdcall，我们必须非常小心地确保非托管的回调定义与 stdcall 调用约定相匹配。

P/Invoke 日志

还有另一个 MDA 也很有用，即 pInvokeLog，它能输出托管程序中每个 P/Invoke 调用的详细信息。启用这个 MDA 的配置文件如下所示：

```
<mdaConfig>
  <assistants>
    <pInvokeLog>
      <filter>
        <match dllName="05native.dll"/>
        <match dllName="kernel32.dll"/>
      </filter>
    </pInvokeLog>
  </assistants>
</mdaConfig>
```

在 filter 部分可以指定要跟踪哪一个 DLL 中的 P/Invoke 调用。在前面的示例中，调试器将输出所有对 05native. dll 和 kernel32. dll 的 P/Invoke 调用。下面给出了一个示例输出：

```
<mda:msg xmlns:mda="http://schemas.microsoft.com/CLR/2004/10/mda">
  <mda:pInvokeLogMsg>
    <method name="07Sig!Advanced.NET.Debugging.Chapter7.PInvoke::Myfunc"/>
    <dllImport dllName="C:\ADNDBin\05Native.dll" entryPoint="Myfunc"/>
  </mda:pInvokeLogMsg>
</mda:msg>
```

7.4 互操作中内存泄漏问题的调试

在理想环境中，托管代码程序永远不需要（至少不是直接地）与非托管代码进行互操作。或者，对于现有的每个非托管代码组件都有一个经过完备测试的可靠的. NET 封装。然而，这种理想环境并不存在，因此必须使用互操作。在大多数时候，如果正确地编写一个非

托管库，那么在这个库周围创建一个托管封装器是非常简单的。但是，在某些情况中，非托管库会出现一些特殊的问题。这些问题包括功能代码中的错误以及严重的内存破坏。在本章的这部分中，我们将看到在使用非托管库时与内存泄漏相关的一些常见误区。请注意，在这里并不会详细介绍在调试非托管内存时的所有可用工具，而只是给出一些常用的工具和技术。要了解如何调试非托管内存泄漏的更详细信息，请参见 Mario Hewardt 和 Daniel Pravat 合著的《Windows 高级调试》中第 9 章的内容。

这里使用的示例托管代码程序位于以下文件夹中：

- 源代码文件：C:\ADND\Chapter7\Date
- 二进制文件：C:\ADNDBin\07Date.exe and 05native.dll

为了尽可能地说明问题，请暂时不要查看源代码。

这个程序的命令行语法如下所示：

```
07Date.exe <number of iterations>
```

迭代次数(number of iterations)是指托管程序执行 P/Invoke 调用的次数，程序最后会把当前日期输出到控制台上。你可以将这个程序看成是对 P/Invoke 调用的模拟压力测试。在运行程序之前，首先启动任务管理器(CTRL-SHIFT-ESC)并且确认显示“内存(私有工作集)”和“提交大小”这两列信息。(在任务管理器中选择“察看(View)”菜单，然后选择“选择列(select Column)”菜单项)。接下来，运行这个程序并每次指定不同的迭代次数，如表 7-2 所示。在每次程序运行快结束时(执行了垃圾收集操作之后)记录内存使用量。

表 7-2 基于迭代次数的内存使用量

迭代次数	内存使用－私有工作集	已提交
1 000	2 240K	8 784K
10 000	4 576K	11 124K
100 000	26 388K	32 948K
1 000 000	245 020K	251 588K

从表 7-2 中可以看到，随着迭代次数的增加，内存使用量也会增加。在最后一次运行中(1 000 000 次迭代)，已提交的内存为 250MB。我们怀疑程序在使用内存时出现了问题，那么如何来调试这个问题？在第 5 章中，我们已经看到了如何通过 eeheap 命令来做一些简单的分析。首先在调试器下运行这个程序(指定 1 000 000 次迭代)，在出现提示信息“Press any key to exit”时中断执行，并运行 eeheap-gc 命令：

```
0:004> !eeheap -gc
Number of GC Heaps: 1
generation 0 starts at 0x01d56510
generation 1 starts at 0x01d5100c
generation 2 starts at 0x01d51000
ephemeral segment allocation context: none
```

```
 segment    begin allocated     size
002e7828 790d8620  790f7d8c 0x0001f76c(128876)
01d50000 01d51000  01d5851c 0x0000751c(29980)
Large object heap starts at 0x02d51000
 segment    begin allocated     size
02d50000 02d51000  02d53240 0x00002240(8768)
Total Size   0x28ec8(167624)
------------------------------
GC Heap Size   0x28ec8(167624)
```

在输出信息的最后一行中，托管堆的大小为167 624字节，远远低于程序消耗的250MB内存。由于大部分内存消耗并不在托管堆上，因此我们使用命令address来了解进程中的内存使用情况：

```
0:004> !address -summary
 ProcessParametrs 002916e8 in range 00290000 002fc000
 Environment 00290808 in range 00290000 002fc000

--------------------- Usage SUMMARY ------------------------
    TotSize (      KB)   Pct(Tots) Pct(Busy)   Usage
    3f36000 (   64728) : 03.09%    17.31%    : RegionUsageIsVAD
   692c8000 ( 1723168) : 82.17%    00.00%    : RegionUsageFree
    2a7e000 (   43512) : 02.07%    11.64%    : RegionUsageImage
     4fe000 (    5112) : 00.24%    01.37%    : RegionUsageStack
       5000 (      20) : 00.00%    00.01%    : RegionUsageTeb
    fe70000 (  260544) : 12.42%    69.68%    : RegionUsageHeap
          0 (       0) : 00.00%    00.00%    : RegionUsagePageHeap
       1000 (       4) : 00.00%    00.00%    : RegionUsagePeb
          0 (       0) : 00.00%    00.00%    :
 RegionUsageProcessParametrs
          0 (       0) : 00.00%    00.00%    :
 RegionUsageEnvironmentBlock
       Tot: 7fff0000 (2097088 KB) Busy: 16d28000 (373920 KB)

---------------------- Type SUMMARY -------------------------
    TotSize (      KB)   Pct(Tots)  Usage
   692c8000 ( 1723168) : 82.17%    : <free>
    2a7e000 (   43512) : 02.07%    : MEM_IMAGE
    1e12000 (   30792) : 01.47%    : MEM_MAPPED
   12498000 (  299616) : 14.29%    : MEM_PRIVATE

--------------------- State SUMMARY ------------------------
    TotSize (      KB)   Pct(Tots)  Usage
   13ec7000 (  326428) : 15.57%    : MEM_COMMIT
   692c8000 ( 1723168) : 82.17%    : MEM_FREE
    2e61000 (   47492) : 02.26%    : MEM_RESERVE

Largest free region: Base 146e0000 - Size 41f50000 (1080640 KB)
```

在输出信息的Usage Summary部分，有一行RegionUsageHeap需要注意，在这行给出的内

存大小值为 260 544KB。它说明非托管 Windows 堆的大小大约在 260MB 左右，非常接近于在任务管理器中报告的内存使用量。此时，我们发现这个托管堆与整体内存使用量相比非常小，并且大多数内存都位于非托管的 Windows 堆上。此时的问题是，为什么所有这些内存都是在非托管对上分配的？你可能已经怀疑这种现象与在托管程序中执行的 P/Invoke 调用是相关联的。我们来看看程序的源代码，如清单 7-6 所示。

清单 7-6　使用 P/Invoke 的示例托管程序

```
using System;
using System.Text;
using System.Runtime.InteropServices;
using System.Runtime.Remoting;
using System.Threading;

namespace Advanced.NET.Debugging.Chapter7
{
    class PDate
    {
        public static void Main(string[] args)
        {
            if (args.Length != 1)
            {
                Console.WriteLine("Please specify number of iteratio
                return;
            }

            int it = Int32.Parse(args[0]);

            StringBuilder date=new StringBuilder(100);
            for (int i = 0; i < it; i++)
            {
                GetDate(date);
            }

            GC.Collect();
            Console.WriteLine("Press any key to exit");
            Console.Read();
        }

        [DllImport("05Native.dll", CharSet = CharSet.Unicode)]
        private static extern void GetDate(StringBuilder date);
    }
}
```

从清单 7-6 中可以看到，程序在一个密集循环中通过 P/Invoke 来调用非托管函数 GetDate。函数 GetDate 的参数是一个字符串，在调用成功的情况下包含了系统的当前日期。在完成了 P/Invoke 迭代后，将强行启动一个垃圾收集操作，并在退出之前给出提示。接下来，我们看看非托管代码函数：

```
__declspec(dllexport) VOID __stdcall GetDate(WCHAR* pszDate)
{
      SYSTEMTIME time;

      WCHAR* pszTmpDate=new WCHAR[100];

      GetSystemTime(&time);

      wsprintf(pszTmpDate,
         L"%d-%d-%d",
         time.wMonth,
         time.wDay,
         time.wYear);
      wcscpy(pszDate, pszTmpDate);
}
```

找出这个问题应该不难。这个函数使用了一个临时缓冲区来保存当前系统日期，然后将内容从这个缓冲区复制到作为参数传递进来的 string 指针。然而，开发人员忘记了释放与临时缓冲区相关的内存，从而造成了内存泄漏。从这个简单的内存泄漏程序中可以看到，当分析内存耗尽或者高内存使用量等问题时，我们必须非常小心。通常，简单地分析托管堆并不足以找出内存使用过多的原因。有时候，虽然托管堆看上去很正确，但我们需要在托管堆之外的其他地方进行分析，并判断在进程的整体内存使用量上是否存在某些值得注意的模式。

能否使用非托管内存泄漏检查工具

我们知道有一些功能非常强大的非托管代码内存泄检测工具（例如 UMDH 和 LeakDiag），它们不仅能告知潜在的内存泄漏问题，而且还能给出线程的完整调用栈。然而，这些工具在托管代码中不会带来很大的帮助，因为它们无法获得完备的托管代码调用栈。例如，在 UMDH 中的调用栈可能像下面这样：

```
002cefc0 003fa2ac 05Native!GetDate+0x67
002cf080 79e7c74b 0x3fa2ac
002cf090 79e7c6cc mscorwks!CallDescrWorker+0x33
002cf110 79e7c8e1 mscorwks!CallDescrWorkerWithHandler+0xa3
002cf254 79e7c783 mscorwks!MethodDesc::CallDescr+0x19c
002cf270 79e7c90d mscorwks!MethodDesc::CallTargetWorker+0x1f
002cf284 79eefb9e mscorwks!MethodDescCallSite::Call+0x18
002cf3e8 79eef830 mscorwks!ClassLoader::RunMain+0x263
002cf650 79ef01da mscorwks!Assembly::ExecuteMainMethod+0xa6
002cfb20 79fb9793 scorwks!SystemDomain::ExecuteMainMethod+0x43f
002cfb70 79fb96df mscorwks!ExecuteEXE+0x59
002cfbb8 70d1573d mscorwks!_CorExeMain+0x15c
002cfbc4 70d6ae72 mscoreei!_CorExeMain+0x25
```

```
002cfbd0 70d65121 mscoree!GetMetaDataInternalInterface+0x2d5
002cfbe4 7760e4b6 mscoree!CorExeMain+0x8
002cfc24 7760e489 ntdll!__RtlUserThreadStart+0x23
002cfc3c 00000000 ntdll!_RtlUserThreadStart+0x1b
```

这种调用栈并不是很有用，因为我们不知道哪一部分的托管代码会调用这个非托管函数。它的主要用处在于能够清楚地指出有一个托管代码线程通过 P/Invoke 调用了一个非托管的函数。然后，我们可以观察这些调用并查看能否找到出错的地方。

7.5 COM 互用性中终结操作的调试

在第 5 章中，我们看到了错误终结器带来的一些常见问题。在编写带有 Finalize 方法的对象时必须小心，要始终确保 Finalize 方法能够返回以避免终结队列中的对象累积，否则会最终导致发生内存耗尽的情况。在本章的这部分内容中，我们将看到一个使用 Finalize 方法和 COM 互用性的示例程序，并且观察是否会出现任何奇怪的问题。清单 7-7 给出了将要使用的示例程序。

清单 7-7 带有可终结对象的 COM 互用性示例

```
using System;
using System.Text;
using System.Runtime.InteropServices;
using System.Runtime.Remoting;
using System.Threading;
using System.Management;
using COMInterop;
namespace Advanced.NET.Debugging.Chapter7
{
    class Wmi
    {
        private ManagementClass obj;
        private byte[] data;

        public Wmi(byte[] data)
        {
            this.data = data;
    }

    public void ProcessData()
    {
        obj = new ManagementClass("Win32_Environment");
        obj.Get();

        //
        //使用数据成员引用
        //
    }
```

```
    ~Wmi()
    {
        //
        // 清除任何非托管资源
        //
    }
}

class Worker
{
    public Worker()
    {
        Init();
    }

    public void ProcessData(byte[] data)
    {
        Process(data);
    }

    ~Worker()
    {
        UnInit();
    }

    [DllImport("05Native.dll")]
    static extern void Init();

    [DllImport("05Native.dll")]
    static extern void UnInit();

    [DllImport("05Native.dll")]
    static extern void Process(byte[] data);
}

class Data
{
    BasicMathClass data;

    public Data(BasicMathClass data)
    {
        this.data = data;
    }

    ~Data()
    {
        int result;
        data.Add(1, 2, out result);
    }
}

class Fin
```

```
{
    private static BasicMathClass s;
    private static Worker worker;

    static void Main(string[] args)
    {
        if (args.Length != 1)
        {
            Console.WriteLine("07Fin.exe <num iterations");
            return;
        }
            Thread newThread =
                   new Thread(new ThreadStart(Helper));
            newThread.SetApartmentState(ApartmentState.STA);
            newThread.IsBackground=true;
            newThread.Start();

            Thread.Sleep(2000);
            Data d = new Data(s);

            d = null;
            GC.Collect();
            GC.Collect();

            Initialize();

            for (int i = 0; i < Int32.Parse(args[0]); i++)
            {
                byte[] b = new byte[10000];
                Wmi w = new Wmi(b);
                w.ProcessData();
            }

            GC.Collect();

            Console.WriteLine("Press any key to exit");
            Console.ReadKey();
        }
        private static void Initialize()
        {
            byte[] b = new byte[100];

            worker = new Worker();
            worker.ProcessData(b);

            worker = null;
            GC.Collect();
        }

        static void Helper()
        {
```

```
            s = new BasicMathClass();
            Thread.Sleep(60000*5);
        }

    }
}
```

清单7-7中的源代码和二进制文件位于以下文件夹中：

- 源代码文件：C:\ADND\Chapter7\Fin
- 二进制文件：C:\ADNDBin\07Fin.exe

这个程序执行了以下操作：

1）创建一个新线程，这个线程将实例化COM对象Math并进入睡眠状态。

2）创建一个Data类型的实例，并且快速释放对这个实例的引用。

3）强制执行垃圾收集操作（GC）。

4）创建一定数量的Wmi实例，其中实例的数量可以通过命令行来指定。

5）强制执行一个GC。

6）等待用户按下任意键退出。

根据前面的执行序列，程序的代码似乎一切正常。我们运行这个程序，并通过任务管理器来监视内存的使用情况。在表7-3中给出了在命令行上指定不同的迭代数量时观察到的内存消耗量（07Fin.exe < num iterations >）。

表7-3 07Fin.exe在不同迭代次数下的内存消耗量

迭代次数	内存使用－私有工作集	已提交
100	6 172K	15 832K
1000	27 044K	36 060K
10 000	224 228K	238 216K

尽管程序在退出之前强行执行了垃圾收集操作，但从表7-3中可以看出内存消耗量在稳步地增长。只需简单地观察不断增长的内存使用量就可以大胆地假设，如果指定足够大的迭代次数，那么程序肯定会抛出一个OutOfMemoryException异常。要分析内存使用量不断增长的原因，可以再次使用内存诊断策略：在调试器下运行程序，并且观察托管堆的统计信息。在接下来的调试输出中，迭代的数量被指定为1000：

```
...
...
...
Press any key to exit
(430c.477c): Break instruction exception - code 80000003 (first chance)
eax=7ffd8000 ebx=00000000 ecx=00000000 edx=7765d094 esi=00000000 edi=00000000
eip=77617dfe esp=0651fdcc ebp=0651fdf8 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
ntdll!DbgBreakPoint:
```

```
77617dfe cc              int     3
0:007> !eeheap -gc
Number of GC Heaps: 1
generation 0 starts at 0x028f540c
generation 1 starts at 0x028237b4
generation 2 starts at 0x01eb1000
ephemeral segment allocation context: none
 segment    begin allocated     size
004b6368 7a733370  7a754b98 0x00021828(137256)
00477790 790d8620  790f7d8c 0x0001f76c(128876)
01eb0000 01eb1000  028f7418 0x00a46418(10773528)
Large object heap starts at 0x02eb1000
 segment    begin allocated     size
02eb0000 02eb1000  02eb4240 0x00003240(12864)
Total Size  0xa8a5ec(11052524)
------------------------------
GC Heap Size  0xa8a5ec(11052524)
```

让程序一直运行直到出现提示信息“Press any key to exit”，此时中断进入到调试器，并执行命令 eeheap-gc 来查看托管堆的总体统计信息。报告的内存大小约为 11MB，而任务管理器报告的已提交内存大小为 36MB。由于托管堆只是整体内存的一部分，因此很可能存在非托管的资源：

```
0:007> !address -summary
 ProcessParametrs 004216e8 in range 00420000 00520000
 Environment 00420808 in range 00420000 00520000

---------- Usage SUMMARY --------------------------------------
    TotSize (      KB)   Pct(Tots) Pct(Busy)   Usage
    44c7000 (   70428) : 03.36%    45.47%    : RegionUsageIsVAD
   768b2000 ( 1942216) : 92.61%    00.00%    : RegionUsageFree
    379f000 (   56956) : 02.72%    36.78%    : RegionUsageImage
     7ff000 (    8188) : 00.39%    05.29%    : RegionUsageStack
       8000 (      32) : 00.00%    00.02%    : RegionUsageTeb
    12d0000 (   19264) : 00.92%    12.44%    : RegionUsageHeap
          0 (       0) : 00.00%    00.00%    : RegionUsagePageHeap
       1000 (       4) : 00.00%    00.00%    : RegionUsagePeb
          0 (       0) : 00.00%    00.00%    :
RegionUsageProcessParametrs
          0 (       0) : 00.00%    00.00%    :
RegionUsageEnvironmentBlock
       Tot: 7fff0000 (2097088 KB) Busy: 0973e000 (154872 KB)

---------- Type SUMMARY ---------------------------------------
    TotSize (      KB)   Pct(Tots)  Usage
   768b2000 ( 1942216) : 92.61%    : <free>
    379f000 (   56956) : 02.72%    : MEM_IMAGE
    2314000 (   35920) : 01.71%    : MEM_MAPPED
    3c8b000 (   61996) : 02.96%    : MEM_PRIVATE

---------- State SUMMARY --------------------------------------
```

```
    TotSize (       KB)   Pct(Tots)  Usage
    6a6f000 (   108988) : 05.20%    : MEM_COMMIT
   768b2000 ( 1942216) : 92.61%    : MEM_FREE
    2ccf000 (    45884) : 02.19%    : MEM_RESERVE

Largest free region: Base 06520000 - Size 51940000 (1336576 KB)
```

从 address 命令的输出中，我们可以看到在 Windows 堆中有 19MB 的使用量。虽然某些内存是由内部 CLR 数据结构占用的，但大多数都可以被认为是由于托管代码使用非托管资源而（间接地通过 System. Management 命名空间）消耗的。然而，值得注意的是，即使使用了非托管资源，也应该在程序结束并执行 GC 后被清除。关键要记住的是，如果在. NET 类型中封装了非托管资源，那么通常会使用 Finalize 方法来清除非托管资源。终结线程是否还没有终结所有的对象实例？此时可以使用 FinalizeQueue 命令，它能给出哪些对象还没有执行 Finalize 方法：

```
0:007> !FinalizeQueue
SyncBlocks to be cleaned up: 5000
MTA Interfaces to be released: 0
STA Interfaces to be released: 0
-----------------------------------
generation 0 has 3 finalizable objects (05bd1260->05bd126c)
generation 1 has 3 finalizable objects (05bd1254->05bd1260)
generation 2 has 5 finalizable objects (05bd1240->05bd1254)
Ready for finalization 3000 objects (05bd126c->05bd414c)
Statistics:
      MT    Count    TotalSize Class Name
000d3868        1           12 Advanced.NET.Debugging.Chapter7.Worker
79112728        1           20 Microsoft.Win32.SafeHandles.SafeWaitHandle
791037c0        1           20 Microsoft.Win32.SafeHandles.SafeFileMappingHandle
79103764        1           20 Microsoft.Win32.SafeHandles.SafeViewOfFileHandle
79101444        1           20 Microsoft.Win32.SafeHandles.SafeFileHandle
7911c9c8        2           40 Microsoft.Win32.SafeHandles.SafePEFileHandle
79108ba4        1           44 System.Threading.ReaderWriterLock
7910a5c4        1           60 System.Runtime.Remoting.Contexts.Context
790fe704        2          112 System.Threading.Thread
6758707c     1000        12000 System.Management.IWbemClassObjectFreeThreaded
000d376c     1000        16000 Advanced.NET.Debugging.Chapter7.Wmi
67583650     1000        64000 System.Management.ManagementClass
Total 3011 objects
```

在输出信息中，我们可以看到在终结队列上仍然存在 3000 个对象等待被清除。是什么原因阻止清除这些对象？毕竟，当强制执行 GC 时，进程的终结线程应该醒来，并开始以串行方式执行每个对象上的 Finalize 方法。我们来看看终结线程是否处于有效状态：

```
0:007> !threads
ThreadCount: 3
UnstartedThread: 0
BackgroundThread: 2
PendingThread: 0
```

```
DeadThread: 0
Hosted Runtime: no
                                        PreEmptive   GC Alloc          Lock
       ID OSID ThreadOBJ    State      GC        Context       Domain
  Count APT Exception
   0     1 42f8  004669a0       a020  Enabled   028f5f98:028f740c  00431288     0 MTA
   2     2 3bd8  00468868       b220  Enabled   00000000:00000000  00431288
     0 MTA (Finalizer)
   3     3 2904  00497248    2007220 Enabled   00000000:00000000  00431288     0 STA
0:007> ~2s
eax=feeefeee ebx=ffffffff ecx=0000032f edx=7fffffff esi=00000000 edi=00000000
eip=77629a94 esp=0404ee18 ebp=0404ee88 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
ntdll!KiFastSystemCallRet:
77629a94 c3              ret
0:002> k
ChildEBP RetAddr
0404ee14 77629254 ntdll!KiFastSystemCallRet
0404ee18 765ac244 ntdll!ZwWaitForSingleObject+0xc
0404ee88 765ac1b2 KERNEL32!WaitForSingleObjectEx+0xbe
0404ee9c 774a1f07 KERNEL32!WaitForSingleObject+0x12
0404eec0 775a7f3e ole32!GetToSTA+0xad
0404eeec 775a89d4 ole32!CRpcChannelBuffer::SwitchAptAndDispatchCall+0x132
0404efcc 774be6f5 ole32!CRpcChannelBuffer::SendReceive2+0xef
0404f044 774a0f4f ole32!CAptRpcChnl::SendReceive+0xaf
0404f098 7625364e ole32!CCtxComChnl::SendReceive+0x95
0404f0b0 762536af RPCRT4!NdrProxySendReceive+0x43
0404f0bc 762533e2 RPCRT4!NdrpProxySendReceive+0xc
0404f534 762535f4 RPCRT4!NdrClientCall2+0x5e9
0404f558 761ee20e RPCRT4!ObjectStublessClient+0x6f
0404f568 774a2ed7 RPCRT4!ObjectStubless+0xf
0404f5f0 774f2833 ole32!CObjectContext::InternalContextCallback+0x128
0404f640 7a0bf7db ole32!CObjectContext::ContextCallback+0x87
0404f68c 7a0c04b2 mscorwks!CtxEntry::EnterContextOle32BugAware+0x2b
0404f7ac 7a0bfa13 mscorwks!CtxEntry::EnterContext+0x322
0404f7c0 7a0bfe6f mscorwks!IUnkEntry::MarshalIUnknownToStreamCallback+0x34
0404f808 79ff0959 mscorwks!IUnkEntry::UnmarshalIUnknownForCurrContext+0x45
0404f814 79f831ce mscorwks!IUnkEntry::GetIUnknownForCurrContext+0x31
0404f840 79f823cc mscorwks!RCW::SafeQueryInterfaceRemoteAware+0x16
0404f884 79f82353 mscorwks!RCW::GetComIPForMethodTableFromCache+0x6b
0404f894 79f8243f mscorwks!RCW::GetComIPFromRCW+0x25
0404f8d4 79f82519 mscorwks!ComObject::GetComIPFromRCW+0x40
0404f93c 79f836ce mscorwks!ComObject::GetComIPFromRCWEx+0x66
0404f988 000ca3ee mscorwks!ComObject::StaticGetComIPFromRCWEx+0x50
WARNING: Frame IP not in any known module. Following frames may be wrong.
0404f9f8 79fbcca7 0xca3ee
0404fa18 79fbcca7 mscorwks!MethodTable::SetObjCreateDelegate+0xc5
0404fa7c 79fbcc15 mscorwks!MethodTable::SetObjCreateDelegate+0xc5
0404fa9c 79fbcbb7 mscorwks!MethodTable::CallFinalizer+0x76
0404fab0 79f6acb6 mscorwks!SVR::CallFinalizer+0xb2
0404fb00 79f6abf7 mscorwks!WKS::GCHeap::TraceGCSegments+0x170
```

```
0404fb88 79fb99d6 mscorwks!WKS::GCHeap::TraceGCSegments+0x2b6
0404fba0 79ef3207 mscorwks!WKS::GCHeap::FinalizerThreadWorker+0xe7
0404fbb4 79ef31a3 mscorwks!Thread::DoADCallBack+0x32a
0404fc48 79ef30c3 mscorwks!Thread::ShouldChangeAbortToUnload+0xe3
0404fc84 79fb9643 mscorwks!Thread::ShouldChangeAbortToUnload+0x30a
0404fcac 79fb960d mscorwks!ManagedThreadBase_NoADTransition+0x32
0404fcbc 79fba09b mscorwks!ManagedThreadBase::FinalizerBase+0xd
0404fcf4 79f95a2e mscorwks!WKS::GCHeap::FinalizerThreadStart+0xbb
0404fd90 765a4911 mscorwks!Thread::intermediateThreadProc+0x49
0404fd9c 7760e4b6 KERNEL32!BaseThreadInitThunk+0xe
0404fddc 7760e489 ntdll!__RtlUserThreadStart+0x23
0404fdf4 00000000 ntdll!_RtlUserThreadStart+0x1b
```

通过使用 threads 命令，可以很快地找出进程中的哪个线程对应于终结线程（调试器线程 ID 为 2）。在找出了终结线程后，我们可以切换到这个线程上下文并转储出调用栈。从调用栈中可以看到 FinalizerThreadWorker 栈帧，这表明我们正在观察的确实是一个终结线程，并且似乎正处于等待状态（可以通过最顶层栈帧看出来）。终结线程处于等待状态并没有什么奇怪之处，因为这个线程经常会等待直到某个对象被放入终结器队列上。正确的终结线程和这里的终结线程的主要区别在于栈帧 ole32！GetToSTA。这意味着终结线程正在调用一个 STA COM 对象，并且在调用过程中停滞了。我们重新观察清单 7-7 中的源代码，看看能否找出哪个对象的终结器调用了一个 COM 对象。在 Data 类型的终结器中调用了 COM 对象 BasicMathClass。通过观察这个终结线程的托管调用栈，我们可以验证这个对象正在被终结：

```
0:002> !ClrStack
OS Thread Id: 0x3bd8 (2)
ESP       EIP
0404f8f8 77629a94 [GCFrame: 0404f8f8]
0404f9d8 77629a94 [ComPlusMethodFrameStandaloneCleanup: 0404f9d8]
COMInterop.BasicMathClass.Add(Int32, Int32, Int32 ByRef)
0404f9f0 007602e5 Advanced.NET.Debugging.Chapter7.Data.Finalize()
```

在继续调试会话之前，重要的是要理解如何实例化 BasicMathClass 这个 COM 对象。在 Main 方法中启动了一个 STA 辅助线程，这个线程将接着创建 COM 对象。在创建对象后，辅助线程会休眠一段时间。这看上去是一种很正常的实例化过程。然而，关键并不在于如何创建 COM 对象，而是在于如何创建线程来实例化 COM 对象。在前面已经提到过，这个线程是在 STA 模型中创建的，因而所有对这个 COM 对象的调用都将被串行化。这种串行化的工作机制是，在后台创建一个隐藏的窗口，并将消息投递到这个窗口消息队列中，每个消息对应于一个方法调用。STA 线程以串行方式来提取窗口消息并执行 COM 方法。为了使这个 STA 线程获得窗口消息，它必须处理消息队列中的消息。如果这个线程无法这么做，那么就不会分发任何消息。现在，问题应该很清楚了：创建 COM 对象的 STA 线程同样很快地进入睡眠状态，从而无法处理消息队列中的消息，因此使得任何对这个 COM 对象的调用都被挂起。事实上，在这个 COM 对象上执行的所有调用都将挂起，直到消息被处理完毕。而且，由于终结线程以串行方式来选择对象执行终结操作，因此在终结队列中的所有对象将永远不会执

行终结操作，这就导致了内存使用量的增加（并且最终产生内存耗尽异常）。

我们已经看到了，一个看似无害的 Finalize 方法在调用 COM 对象时却导致了严重的可靠性问题。在与剩下的 COM 代码交互时，即使 COM 互用性层努力使工作变得尽可能容易，但还是必须非常小心地确保交互过程不会产生问题。

过早释放 COM 对象

在前面提到过，COM 对象的生命周期是由 RCW 来管理的。如果需要提前释放一个 COM 对象，那么可以使用 Marshal. ReleaseComObject 方法。为了避免发生内存破坏等问题，我们必须确保这个 COM 对象没有被使用。可以通过 raceOnRCWCleanup 这个 MDA 来识别这些问题。要启用这个 MDA，可以使用以下配置文件：

```
<mdaConfig>
  <assistants>
    <raceOnRCWCleanup/>
  </assistants>
</mdaConfig>
```

7.6 小结

在本章中，我们讨论了 CLR 如何通过一种简洁的方式与非托管代码交互。我们介绍了两种主要的互用性层：P/Invoke 和 COM 互用性。虽然 CLR 易于与非托管代码交互（最简单的情况就是通过属性），但必须深入理解互用性的本质才不会导致各种问题。我们给出了一些由于不正确的互用性而导致的可靠性问题。具体来说，我们介绍了一些编写错误的托管程序，例如无效的自定义列集操作、异步互用性，互用性中的内存泄漏以及 COM 互用性和终结操作等。最后我们给出了如何通过调试器以及相关的工具来快速找出问题的根源，并介绍了一组 MDA 可以帮助分析这些复杂的问题。

第三部分 高级主题

第8章 事后调试

本书介绍了开发人员在分析问题代码时可以使用的一些功能强大的工具。我们的最终目标是使开发人员在开发过程中能够使用这些工具，从而编写出高质量的代码。这些工具都是通过一些自动化的方法来找出错误，然而，它们并不能保证发布的程序中不包含任何错误。

在程序发布后，总会出现一些问题，并且这些问题的出现时机往往是最不凑巧的——大多数出现在用户使用它们时。它们可能对用户造成严重的影响，也可能只是一些令人讨厌的现象。无论是哪种情况，你都会收到某位沮丧用户的电话，询问为什么这个程序不能正确地工作。要补救这种情况并找出问题的原因，一种方式是通过远程方式来访问出现问题的计算机。虽然这种方式可行，但用户往往会有一些担心，因此在大多数情况下会拒绝采用这种方式。不允许通过远程方式来访问机器的原因有多种，通常包括：

- 用户所处的环境或者策略不允许远程连接。
- 远程调试要求将调试器附载到一个或多个进程上，这会导致机器停机。如果这个进程是一个关键服务器，那么用户不愿意停机。
- 通过用户态或者内核态来调试进程意味着，开发人员能够访问机器的完全状态，包括内存的内容。对于某些用户来说，这可能会导致隐私方面的问题。

如果客户拒绝对出现问题的机器进行实时调试，而又无法在本地重现问题，那么还能不能对问题进行调试？答案是肯定的，这个过程就称之为事后调试（Postmortem Debugging）。事后调试包括以下步骤：

1）触发故障的发生。

2）抓取系统在发生故障时的状态快照（根据不同的故障类型，在某些情况下还需要抓取故障发生前后的状态快照）。

3）将快照发送给工程师以做进一步分析。

在本章中，我们将讨论各种抓取快照的不同方式，不同类型的转储信息以及如何分析它们。此外，我们还将介绍一种功能非常强大的转储文件聚合服务，即 Window 错误报告

(Windows Error Reporting)。

我们首先来看转储文件的一些基本知识。

8.1 转储文件基本知识

在前面曾提到，转储文件是进程状态的外在表示。生成转储文件的主要目的是：在不需要对出现问题的计算机进行实时访问的情况下，就可以对程序故障进行分析。在生成了转储文件后，可以将这个文件发送给相应的工程师来分析故障，而无需访问出现故障的机器。因此，工程师可以在他自己的计算机上加载转储文件，并使用调试器的事后调试功能来分析故障。在转储文件中包含了哪些信息？这要取决于在生成转储文件时所指定的选项。共有两种类型的转储文件：

- 完全转储文件（full dump）
- 微型转储文件（mini dump）

在完全转储文件中包含了进程的整个内存空间、可执行映像、句柄表和调试器需要使用的其他信息。当使用完全转储文件时不能指定所要收集的数据量。但是，我们可以通过调试器将完全转储文件转换为微型转储文件。

微型转储文件的内容是可变的，并且能根据使用的转储文件生成器进行定制。在微型转储文件中可以只包含某个线程的信息，也可以包含被转储进程的详细信息。需要注意的是，在最大的微型转储文件中包含的内容要多于在完全转储文件中包含的内容。因此，本章将重点介绍微型转储文件的结构。

有一组工具可以生成转储文件，如表 8-1 所示。

表 8-1 生成转储文件的工具

名 字	描 述
Windows 调试器（Windows Debuggers）	Windows 调试器可以生成各种不同大小的转储文件，并且能够完全控制转储文件的生成过程
ADPlus	ADPlus 是 Windows 调试工具集中的一个。它的作用相当于一个进程监视器，每当发生崩溃或者挂起时，都能生成转储文件。此外，它还能将崩溃事件通知给用户
Windows 错误报告	Windows 错误报告是 Microsoft 提供的一种服务，用户通过这种服务注册到一个实时的错误报告站点。每当用户的某个应用程序发生错误时，都会将一个错误报告从发生崩溃的机器发送到 Windows 错误报告站点。然后，在进行事后分析时可以从 WER 服务中提取崩溃信息（包括转储文件）

在本节中，我们将首先介绍如何使用 Windows 调试器和 ADPlus 来生成转储文件，然后再介绍 Windows 错误报告。

为了更好地说明转储文件的生成过程，我们使用一个简单的示例程序，这个程序在堆上分配内存，写入这块内存，然后发生故障。清单8-1给出了程序的源代码。

清单8-1 一个简单的崩溃程序

```
using System;
using System.Text;
using System.Runtime.InteropServices;

namespace Advanced.NET.Debugging.Chapter8
{
    class SimpleExc
    {
        static void Main(string[] args)
        {
            SimpleExc s = new SimpleExc();
            s.Run();
        }

        public void Run()
        {
            Console.WriteLine("Press any key to start");
            Console.ReadKey();
            ProcessData(null);
        }

        public void ProcessData(string data)
        {
            if (data == null)
            {
                throw new ArgumentException("Argument NULL");
            }
            string s = "Hello: " + data;
        }

    }
}
```

清单8-1中的源代码和二进制文件位于以下文件夹中：

- 源代码文件：C:\ADND\Chapter8\SimpleExc
- 二进制文件：C:\ADNDBin\08SimpleExc.exe

程序的源代码很简单。在调用ProcessData时抛出了一个ArgumentException异常。我们首先说明如何使用调试器来生成转储文件。

8.1.1 通过调试器来生成转储文件

我们使用清单8-1中的程序来进行说明。在调试器下运行这个程序，直到发生异常：

```
...
...
...
ModLoad: 77bb0000 77bb6000   C:\Windows\system32\NSI.dll
ModLoad: 79060000 790b6000   C:\Windows\Microsoft.NET\Framework\v2.0.50727
\mscorjit.dll
(1860.958): CLR exception - code e0434f4d (first chance)
(1860.958): CLR exception - code e0434f4d (!!! second chance !!!)
eax=0020eaec ebx=e0434f4d ecx=00000001 edx=00000000 esi=0020eb74 edi=00416bd0
eip=767142eb esp=0020eaec ebp=0020eb3c iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
KERNEL32!RaiseException+0x58:
767142eb c9              leave
0:000> .loadby sos mscorwks
0:000> !ClrStack
OS Thread Id: 0x958 (0)
ESP       EIP
0020ebc4 767142eb [HelperMethodFrame: 0020ebc4]
0020ec68 00e10177 Advanced.NET.Debugging.Chapter8.SimpleExc.ProcessData
(System.String)
0020ec80 00e1010c Advanced.NET.Debugging.Chapter8.SimpleExc.Run()
0020ec88 00e100a7 Advanced.NET.Debugging.Chapter8.SimpleExc.Main
(System.String[])
0020eeac 79e7c74b [GCFrame: 0020eeac]
```

此时，我们希望生成一个转储文件来做进一步事后分析。在生成转储文件时，最大的问题之一就是要指定转储文件包含的信息量。作为一个经验法则，在转储文件中保存的状态越多，在进行事后调试时能获得的信息也就越多。当然，最大的限制因素就是转储文件的大小。有时候你会发现无法从一个高安全的服务器上获得很大的转储文件，而只能对一个删除了某些敏感信息之后的转储文件进行分析。

生成转储文件的命令是. dump。. dump/m 选项指示调试器生成一个微型转储文件。此外，. dump /m 命令还可以是其他的选项，如表 8-2 所示。

表 8-2　dump 命令的选项

选　项	描　述
a	生成一个完整的微型转储文件，启动所有选项。在这个文件中将包含完整的内存数据、句柄信息、模块信息、基本的内存信息和线程信息等。相当于使用 /mfFhut
f	生成一个微型转储文件，其中包含进程内所有可访问和已提交的内存页
F	生成一个微型转储文件，其中包含调试器在重构整个虚拟内存地址空间时需要的所有基本内存信息
h	生成一个微型转储文件，其中包含句柄信息
u	生成一个微型转储文件，其中包含未卸载模块的信息。注意，这个选项只能在 Windows Server 2003 上使用
t	生成一个微型转储文件，其中包含线程时间的信息。在线程时间信息中包括创建时间，以及在用户态和内核态中执行的时间

（续）

选项	描述
i	生成一个微型转储文件，其中包含辅助内存信息。辅助内存是指由栈指针或者后台存储使用的内存（及其周围的一小块内存）
p	生成一个微型转储文件，其中包含进程环境块和线程环境块
w	生成一个微型转储文件，其中包含所有已提交的读-写私有内存页
d	生成一个微型转储文件，其中包含映像中的所有数据段
c	生成一个微型转储文件，其中包含映像中的所有代码段
r	生成一个微型转储文件，适合于在需要保护隐私的情况中使用。这个选项将删除在重建栈时不需要的任何信息（将这些信息替换为0，包括局部变量）
R	生成一个微型转储文件，适合在需要保护隐私的情况下使用。这个选项将从微型转储文件中删除完整的模块路径，因此将确保用户目录结构的隐私性

除了各种控制转储文件内容的不同选项外，还必须指定转储文件的名字。如果在生成转储文件时没有指定完整的路径，那么会默认保存到调试器所在的目录下。下面的示例演示了如何生成一个带有完整内存信息的转储文件，并指定完整的路径名。

```
.dump /mf c:\08dumpfile.dmp
```

让我们在崩溃的程序上运行.dump命令：

```
0:000> .dump /mf 08dumpfile.dmp
Creating dumpfile.dmp - mini user dump
Dump successfully written
```

调试器会生成一个大约为64MB的转储文件。要使用这个转储文件，需要通过-z开关将其加载到调试器中。例如，要加载刚才生成的转储文件，可以使用以下命令：

```
c:\>ntsd -z 08dumpfile.dmp
```

在调试器加载了转储文件后，可以看到以下调试输出：

```
...
...
...
Loading Dump File [c:\08dumpfile.dmp]
User Mini Dump File with Full Memory: Only application data is available
Executable search path is:
Windows Server 2008 Version 6001 (Service Pack 1) MP (2 procs) Free x86 compatible
Product: WinNt, suite: SingleUserTS
Debug session time: Mon Mar  2 06:25:10.000 2009 (GMT-8)
System Uptime: 5 days 7:44:57.406
Process Uptime: 0 days 0:02:39.000
.............................
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for
ntdll.dll -
This dump file has an exception of interest stored in it.
```

```
The stored exception information can be accessed via .ecxr.
(1860.958): CLR exception - code e0434f4d (first/second chance not available)
eax=0020eaec ebx=e0434f4d ecx=00000001 edx=00000000 esi=0020eb74 edi=00416bd0
eip=767142eb esp=0020eaec ebp=0020eb3c iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
kernel32!RaiseException+0x58:
767142eb c9              leave
```

在靠近调试输出的顶部，我们注意到调试器给出了被加载转储文件的一些基本信息。这些信息包括转储文件的位置，转储文件的类型以及一些可用信息等。接下来需要注意的信息就是在调试器输出的末尾给出的故障原因（CLR 异常）。根据这个转储文件，你可以在任何一台机器上调试这个故障，而无需直接访问发生故障的机器。我们将在本章的后面详细介绍如何进行事后分析。

通过调试器来生成转储文件的一个难点就是调试器必须在合适的时候被附载到故障进程上。虽然这并不是一个大问题，但在某些情况下，崩溃情况只是每隔一段时间重现一次，因此很容易错失附载调试器的机会。如果能使 Windows 在发生进程崩溃时就使用调试器来生成转储文件，那么这个问题就能得到解决。这种机制确实存在，称之为事后调试器设置（Postmortem Debugger Setup）。在默认情况下，Windows 使用 Dr. Watson[⊖]作为事后调试器。每当进程崩溃时，Dr. Watson 都会生成一个转储文件，并询问用户是否将转储文件发送给 Microsoft 以做进一步的分析。我们可以通过使用表 8-3 中的命令行来修改事后调试器。

表 8-3 事后调试器的设置

命令行	注册键 Aedebug\Debugger 的值	描述
Windbg -I	windbg.exe -p %ld -e %ld -g	将事后调试器修改为 Windbg。注意 -I 必须大写
cdb -iae	cdb.exe -p %ld -e %ld -g	将事后调试器修改为 cdb
ntsd -iae	ntsd.exe -p %ld -e %ld -g	将事后调试器修改为 ntsd
drwtsn32 -i	drwtsn32 -p %ld -e %ld -g	将事后调试器修改为 Dr. Watson

转储文件的生成

在 Windows Vista 中修改了错误报告技术在本地机器上保存转储文件的方式。在 Windows Vista 之前，Dr. Watson 默认将生成的转储文件保存在本地机器上。这些转储文件可以由任何一个想要调试转储文件的用户访问。在 Windows Vista 中去掉了 Dr. Watson，而是引入了一种更为可靠和稳定的错误报告机制。其中的修改之一就是，生成的转储文件（在默认情况下）不会被保存到本地机器上。要改变这种默认行为，可以将注册表 ForceQueue 设置为 1，这将使所有转储文件在上传到 Microsoft 之前就在本地机器上排队。ForceQueue 注册键值位于以下注册路径：

⊖ 在 Windows Vista 中不推荐使用，而是建议使用另一项新技术。——译者注

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\Windows Error
Reporting
```

在注册键值 ForceQueue 被设置为 1 后，生成的所有转储文件都将被保存到以下位置：对于在系统上下文中运行或者被提升到系统上下文中运行的进程：

```
%ALLUSERSPROFILE %\Microsoft\Windows\WER\[ReportQueue|
ReportArchive]
```

对于所有其他的进程：

```
%LOCALAPPDATA%\Microsoft\Windows\WER\ [ReportQueue|
ReportArchive]
```

当执行表 8-3 中的命令行时，会发生什么情况？答案很简单。这些命令会修改一些注册键值，而 Windows 在检测到进程崩溃时将查看这些键值。用于设置事后调试器的注册路径如下所示：

```
HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\AeDebug
```

当调试非托管代码程序时，会用到 AeDebug 键值。然而，对于托管代码，可以使用其他两个注册键值来控制托管代码的调试，它们是 DbgManagedDebugger 和 DbgJITDebugLaunchSetting，分别位于以下键中：

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\.NETFramework
```

1. DbgManagedDebugger

如果通过 DbgJITDebugLaunchSettings 来启用事后调试，那么 DbgManagedDebugger 注册键值会指定当遇到未处理异常时应该启动哪一个调试器。例如，要将 ntsd 设置为当托管程序发生未处理异常时的默认调试器，那么应该把注册键值 DbgManagedDebugger 设置为：

```
c:\program files\debugging tools for windows (x86)\ntsd.exe -p %ld
```

当遇到未处理异常时，注册键 DbgManagedDebugger 中指定的调试器并不会被立即调用，而是显示一个消息框，由用户来选择是调试程序还是结束程序。

在事后调试中，一个很常见的问题就是，如何在程序发生故障时自动生成一个转储文件。实现这种机制的关键在于，要将注册键 DbgManagedDebugger 设置为以下值：

```
ntsd -pv -p %ld -c ".dump /u /ma <path to dump file>; .kill; qd
```

前面注册键值设置的含义是，当一个故障发生时，启动 ntsd 调试器，并且执行 dump 命令来生成一个转储文件，然后退出调试会话。

我们可以通过注册键值 DbgJITDebugLaunchSettings 来准确地控制未处理异常的行为，在

后面章节中将进行讨论。

2. DbgJITDebugLaunchSettings

注册键 DbgJITDebugLaunchSettings 用来指定发生未处理异常时的行为。如果这个值被设置为 0，那么将显示一个消息框，并由用户来选择对故障采用何种处理方式。请注意，只有在交互式进程的情况下才会显示这个消息框，而对于其他的进程（例如服务）则是直接结束。在图 8-1 中给出了示例的消息框。

这个事后调试消息框通知用户在程序 08SimpleExc. exe 中发生了一个问题，并且提示用户选择是调试这个故障（点击“Debug”按钮）还是结束这个程序（点击“Close Program”按钮）。如果点击了“Debug”按钮，那么将查询 DbgManagedDebugger 键的值，并启动在这个键中设置的调试器。

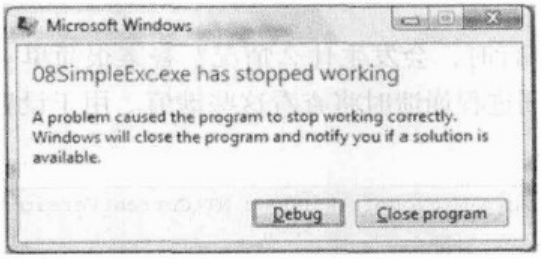


图 8-1　托管代码事后调试消息框

如果 DbgJITDebugLaunchSettings 被设置为 1，那么程序会直接结束并返回一个栈转储。

如果这个值被设置为 2，那么将立即启动在 DbgManagedDebugger 中指定的调试器，而不会显示消息框。

最后，如果这个值被设置为 16，对于交互式进程会显示前面的消息框，而对于非交互式进程则会直接启动在 DbgManagedDebugger 中指定的调试器。

虽然这些调试器提供了足够强大的功能和灵活性来生成转储文件，但还可以使用另外一个工具。这个工具就是 ADPlus。

8.1.2　通过 ADPlus 生成转储文件

ADPlus 工具能够监测和自动化一个或多个故障进程的转储文件生成过程，并且能够在发生崩溃时通知用户或者计算机。ADPlus 是一个命令行驱动的脚本，Microsoft 强烈推荐在 cscript. exe 解释器中运行它。事实上，如果默认的脚本解释器没有被设置为 cscript. exe，那么将出现一个对话框询问是否将解释器改为 cscript. exe。除了这些命令行选项外，ADPlus 还可以使用配置文件。配置文件能够对 ADPlus 的操作流程进行更细致地控制。

ADPlus 可以在以下两种模式中运行：

- 挂起模式（Hang Mode）用于分析出现挂起现象的进程（例如程序不执行或者

100%的CPU使用率）。ADPlus必须在进程挂起之后启动。

- 崩溃模式（Crash Mode）用于分析出现崩溃行为的进程。ADPlust必须在进程崩溃之后启动。

我们以崩溃模式为例来介绍如何通过ADPlust为08SimpleExc.exe生成一个转储文件。首先启动08SimpleExc.exe程序：

```
C:\ADNDBin\08SimpleExc.exe
```

在按下任意键恢复程序的执行之前，运行以下命令行：

```
C:\>adplus.vbs -crash -pn 08SimpleExc.exe -y
 SRV*c:\Symbols*http://msdl.microsoft.com/download/symbols
```

-crash开关将把ADPlus置为崩溃模式，-pn开关告知ADPlus需要监视的进程名字，而-y将设置在ADPlus执行过程中使用的符号路径。使用-pn开关的好处在于，它能够监视由name指定的进程的任意数量实例。

当执行完成后，ADPlus会把结果日志文件保存到Windows调试器的安装路径下。路径的名字符合以下格式：

```
<runtype>_Mode__Date_<date of run>__Time_<time of run>
```

例如，当ADPlus执行完成时，会创建以下路径：

```
c:\Program Files\Debugging Tools for Windows (x86)\Crash_Mode__Date_03-02-
2009__Time_08-31-43AM
```

注意，可以通过开关-o来改变默认的路径。

在前面的目录中包含了多个文件，其中最重要的是*.dmp文件。这些文件包含了从运行中得到的所有转储信息，你可以看到总共收集了多个转储文件。为什么在每次崩溃时会产生多个转储文件？这是因为ADPlus会自动执行转储文件的收集操作，从而，只要在执行期间触发了预定的条件时，就会生成转储文件。我们可以通过转储文件的名字来找出转储文件的生成原因。例如，在前面的程序运行中，ADPlus生成了以下转储文件：

```
PID-4448__08SIMPLEEXC.EXE__1st_chance_Process_Shut_
Down__full_1e20_2009-03-02_08-32-17-440_1160.dmp

PID-4448__08SIMPLEEXC.EXE__2nd_chance_NET_CLR__full_
1e20_2009-03-02_08-32-08-384_1160.dmp
```

当发生第一轮进程关闭事件时，ADPlus将生成一个完全转储文件，然后当发生.NET异常（第二轮）时再生成一个完全转储文件。在程序中是否需要所有这些转储文件？不是的。这里需要注意的转储文件是第二轮.NET异常。然而，在某些情况中，定期生成转储文件也是非常有用的，因为这使调试人员能够了解进程中问题的演化过程。ADPlus还为用户提供了一种功能更强大的方式来配置信息收集的频率以及在何种条件下收集信息，尤其是为调试器

提供了一个脚本前端。你可以在调试器的帮助文档中了解更多有关 ADPlus 脚本功能的内容。需要指出的是，ADPlus 并不会通过脚本引擎来实现任何“特殊的”功能。而只是采用了一种对用户友好的方式来执行调试器命令，并将它们的执行过程自动化。通过观察转储文件所在目录中的 CDBScripts 目录，你可以看到配置信息转换为调试器命令的过程。在这里的示例中，CDBScripts 目录下包含了一个文件 PID - 4448__08SimpleExc. exe. cfg，其中包含了在 ADPlus 会话中使用的所有调试器命令。

ADPlus 最后一个要点就是，如何控制在发生故障时生成的转储文件类型。共有四个命令行开关来控制这种行为：

- -FullOnFirst。这个开关使 ADPlus 在发生第一轮异常时生成一个完全转储文件。
- -MiniOnSecond。这个开关使 ADPlus 在发生第二轮异常时生成一个微型转储文件。
- -NoDumpOnFirst。这个开关使 ADPlus 在发生第一轮异常时不生成一个微型转储文件。这种方式在某些情况下是有用的，因为有时候程序生成的第一轮异常能够被妥善处理。
- -NoDumpOnSecond。这个开关告诉 ADPlus 在发生第二轮异常时不生成一个微型转储文件。

ADPlus 是一种方便的，功能强大的以及灵活的工具，它能有效地监视故障进程并从中收集数据。在本节中，我们介绍了这个工具的基本知识，此外你还可以研究这个工具的其他特性，例如脚本功能，以及设置自定义的异常处理器以便在发生异常时生成转储文件。

到目前为止，我们给出了生成转储文件的两种最常见方式，接下来将使用转储文件并介绍问题分析过程。

8.1.3 转储文件的调试

现在，假设已经获得了一个或多个转储文件，并且需要找出导致进程故障的根本原因，那么通过这些转储文件能够完成哪些工作？能否转储内存，查看句柄或者单步调试代码？记住，转储文件只是进程状态的一个静态快照。因此，我们无法在代码上设置断点以及单步调试代码。最好把转储文件看成是一种手动调试。通过手动方式，我们可以观察程序的状态，并手动地构造一些假设来分析代码是如何执行到这种状态的。显然，通过状态分析来构造代码的执行过程比实时调试会话要困难得多。而且，在使用转储文件时，仍然可以使用许多调试器命令，并且在大多数情况下，只要有足够的耐心，总是可以找出问题的根源。

在进一步观察转储文件之前，首先需要获得两个关键信息：符号文件和数据访问层（Data Access Layer，DAC）。由于在转储文件中不包含任何符号信息，因此当分析转储文件时，符号文件是非常重要的。我们已经在本书中看到了一些最常见的符号命令。另一个关键的信息就是 CLR 数据访问层，SOS 将通过这个信息来提供在调试会话中需要的所有数据。

8.1.4 数据访问层

在非托管调试环境中，许多信息都可以通过观察内存来收集，不同的是，在托管代码中，SOS 依靠 CLR 来提供我们预期的调试输出以及结果。为了使 SOS 能够正确解析（或翻译）传递给它的原始数据，SOS 将调用 CLR（即执行 CLR 代码）来辅助执行这个过程。CLR 中负责实现这个功能的组件就是数据访问层，它包含在 mscordacwks. dll 中。现在，随着 CLR 被不断地增强，底层的 DAC 同样随各个版本（包含补丁）的不同而变化。通过查看机器上每个. NET 版本的安装文件夹可以很容易地验证这一点。例如，在我的机器上，mscordacwks. dll 位于以下文件夹中：

```
C:\Windows\Microsoft.NET\Framework\v2.0.50727
```

然而，在运行 Visual Studio 2010 CTP 的机器上，这个文件位于下面的目录中，表示在 CLR 4. 0 中包含了一个新版本的 mscordacwks. dll。

```
C:\Windows\Microsoft.NET\Framework\v2.0.50727

C:\Windows\Microsoft.NET\Framework\v4.0.11001
```

由于调试器在其操作期间需要用到这个组件，因此要知道调试器这个文件的位置。在实时调试过程通常不需要关心这个问题，因为 SOS 能够从当前被调试的运行时所在位置上找到这个文件。在事后调试（或者转储文件）中，在程序中使用的 CLR 版本可能与转储文件所在机器上的 CLR 版本不同。再次重申，SOS 调试器扩展将调用 mscordacwks. dll 中的函数，这个动态库将执行 CLR 代码，因此为调试器指定正确的版本是非常重要的。由于 CLR 版本的正确与否对于调试成功与否非常关键，因此 Microsoft 公布了 mscordacwks. dll 的大部分符号，放在 Microsoft 共有符号服务器上。只要将调试器的符号路径指向公有符号服务器（使用 symfix 或者其他相关的命令），那么调试器就能找到这个文件。然而，在有些情况中，你仍然需要显式地告诉 SOS 扩展命令到什么位置上查找该文件。这些情况包括，当文件不在公有符号服务器上（这种情况很少发生）或者文件没有安装在与生成转储文件的机器上的同一个路径下。在这些情况中，我们可以使用 cordll 命令。cordll 命令能控制 mscordacwks. dll 的加载方式，并在处理版本不匹配问题时节约大量的时间。在表 8-4 中详细给出了 cordll 命令的各个开关。

表 8-4 Cordll 命令的开关

开 关	描 述
-l	在默认加载路径中搜索 DLL 并加载调试模块
-u	从内存中卸载调试模块
-e	启用 CLR 调试
-d	禁用 CLR 调试

（续）

开关	描述
-D	禁用 CLR 调试并卸载调试模块
-N	重新加载调试模块
-lp	指定调试模块的目录
-se	启用使用短名字版本的调试模块，mscordacwks. dll
-sd	禁用使用短名字的调试模块，mscordacwks.dll。如果指定了这个开关，那么调试模块要以以下格式加载： mscordacwks_<spec>.dll，其中<spec>的形式为<architecture>_<architecture>_<file version>，而<Architecture>可以是 x86 或者 amd64
-ve	启用 verbose 模式。当处理不匹配问题时，verbose 模式是非常有用的，因为它能给出调试器如何加载调试模块的信息
-vd	禁用 verbose 模式

如何知道是否需要使用 cordll 命令？通常，如果存在版本不匹配的 mscordacwks. dll，那么 SOS 调试扩展将输出以下错误信息（或者这些错误信息的某种变化形式）：

```
Failed to load data access DLL, 0x80004005
Verify that 1) you have a recent build of the debugger (6.2.14 or newer)
2) the file mscordacwks.dll that matches your version of mscorwks.dll is
in the version directory
3) or, if you are debugging a dump file, verify that the file
mscordacwks___.dll is on your symbol path.
4) you are debugging on the same architecture as the dump file.
For example, an IA64 dump file must be debugged on an IA64
machine.

You can also run the debugger command .cordll to control the debugger's
load of mscordacwks.dll. .cordll -ve -u -l will do a verbose reload.
If that succeeds, the SOS command should work on retry.

If you are debugging a minidump, you need to make sure that your
executable path is pointing to mscorwks.dll as well.
```

我们来仔细分析 SOS 调试器扩展输出的每条建议。第一条建议很简单，它要求确保运行调试器的新版本。第二条建议同样简单，只是要求检查并确保 mscordacwks. dll 的版本与所加载 mscorwks. dll 是相对应的。在前面讨论过，mscordacwks. dll 应该位于与 mscorwks. dll 相同的目录下。第三条建议值得注意，它要求确保 mscordacwks___. dll 位于符号路径中。mscordacwks___. dll 是什么？回到表 8-4，注意到 -sd 开关的作用是为 mscordacwks. dll 启用长名字。长名字只是将这个 DLL 对应的架构以及构建编号添加到 DLL 的名字中。然后，你可以更新符号路径，指向这个 DLL，并执行 cordll 命令来重新加载 mscordacwks. dll。例如，如果在生成转储文件时使用的 mscordacwks. dll 的版本为 1. 1. 1. 0，架构为 x86，那么可以将 mscordacwks. dll 重新命名为 mscordacwks_x86_x86_1. 1. 1. 0. dll，并将调试器的符号路径指向这

个重命名的位置，接着使用 cordll 命令来重新加载调试模块：

```
0:008> .sympath+ <path to renamed module>
0:008> .cordll -ve -u -l
```

第四条建议要求我们确保运行调试器所在的架构与生成转储文件的架构相同。由于调试器将执行 DAC 中的代码，因此用于调试转储文件的调试器的架构信息与创建转储文件时使用的调试器的架构信息要完全一样。例如，如果使用 64 位的调试器为某个 32 位的进程生成转储文件，并且这个进程是在 WOW64 的 64 位系统上运行，那么将不能调试这个转储文件。此时不应该使用 64 位调试器，而是确保要使用正确比特数（32 位）的调试器来生成转储文件。

最后，输出的最后一行要求可执行路径指向 mscorwks. dll。可执行路径可以在调试器中通过 exepath 命令来控制（如果要添加可执行路径，需要使用 exepath + 命令）。例如，如果在调试转储文件时，mscorwks. dll 位于 C:\windows\microsoft. net\framework\v2. 0. 50727 中，那么可以使用以下命令来确保正确地设置可执行路径，然后重新加载以确保调试器读取这个文件：

```
0:008> .exepath+ c:\windows\microsoft.net\framework\v2.0.50727
Executable image search path is: c:\windows\microsoft.net\framework\v2.0.50727
0:008> .reload
```

到目前为止，所有讨论的策略都假设 mscordacwks. dll 是位于某个位置（或者是公有符号服务器，或者在本地机器上）。如果无法找到在生成转储文件时使用的 DLL 的正确版本，那么最好要求生成这个转储文件的人员把相应的 mscordacwks. dll 发送给你。在收到这个文件后，再按照之前给出的策略来加载它。

有时候，确保加载正确版本的 mscordacwks. dll 很复杂，需要经过反复尝试。然而，在加载成功之后，SOS 调试器扩展会充分发挥其功能。

8.1.5 转储文件分析：未处理的. NET 异常

在前面一节中，我们为发生故障的程序生成了一个转储文件，现在的任务就是通过这个转储文件来找出问题的根源。

要使用这个转储文件，我们必须通过 -z 开关来告诉调试器分析一个转储文件：

```
C:> ntsd -z C:\08dumpfile.dmp
```

在调试器启动后，第一部分重要的信息就是 CLR 异常输出。

```
This dump file has an exception of interest stored in it.
The stored exception information can be accessed via .ecxr.
(2dc4.2a08): CLR exception - code e0434f4d (first/second chance not available)
eax=0024ef20 ebx=e0434f4d ecx=00000001 edx=00000000 esi=0024efa8 edi=002c43e8
eip=767142eb esp=0024ef20 ebp=0024ef70 iopl=0         nv up ei pl nz ac po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000212
kernel32!RaiseException+0x58:
767142eb c9              leave
```

这些信息告诉我们，正在分析的转储文件是由于发生了一个 CLR 异常而生成的。下一个步骤是查看异常的详细信息：

```
0:000> kb
ChildEBP RetAddr  Args to Child
0024ef70 79f071ac e0434f4d 00000001 00000001 kernel32!RaiseException+0x58
0024efd0 79f0a629 01b66c20 00000000 00000000
 mscorwks!RaiseTheExceptionInternalOnly+0x2a8
0024f094 01630197 01b658d0 0024f0e0 0024f0fc mscorwks!JIT_Throw+0xfc
WARNING: Frame IP not in any known module. Following frames may be wrong.
00000000 00000000 00000000 00000000 00000000 0x1630197
0:000> !pe 01b66c20
Exception object: 01b66c20
Exception type: System.ArgumentException
Message: Argument NULL
InnerException: <none>
StackTrace (generated):
    SP       IP       Function
    0024F09C 01630197
08SimpleExc!Advanced.NET.Debugging.Chapter8.SimpleExc.ProcessData
(System.String)+0x57
    0024F0B4 01630124
08SimpleExc!Advanced.NET.Debugging.Chapter8.SimpleExc.Run()+0x34
    0024F0C8 016300A7 08SimpleExc!Advanced.NET.Debugging.Chapter8.SimpleExc.Main
(System.String[])+0x37

StackTraceString: <none>
HResult: 80070057
```

从上述信息可以看出，程序由于发生了一个参数异常而产生故障：在方法调用中指定的参数是 NULL，这是不正确的。与异常一起给出的栈回溯提供了足够多的信息，通过这些信息可以进行一次简单的代码审查来找出问题。虽然这只是转储文件的一个简单示例，但却很好地说明了通过转储文件来调试托管代码问题是完全可行的，并且在 SOS 和 SOSEX 等调试器扩展中的一些有用的命令仍然可以继续使用。

8.2 Windows 错误报告

任何使用过 Windows 的人都遇到过图 8-2 中的消息框。

图 8-2 中给出了 Windows 错误报告的界面。当看到这个消息框时，用户可以选择是否将错误报告发送给 Microsoft。如果用户选择发送错误报告，那么将通过一个安全的通道（HTTPS）发送到 Microsoft 数据库，然后在数据库中对报告进行分类和存储以做进一步分析。在发送的错误报告中包含了一个转储文件，它有助于分析问题的人员找出问题的根源。能够参与到 Windows 错误报告流程中的程序并不仅限于 Microsoft 企业的产品。在 Windows 中的任何崩溃进程都可以使用这种模式。然而，要想访问与你的程序对应的错误报告，必须首先向 Windows 错误报告服务进行注册。在本节中，我们将看到 Windows 错误报告机制的工作原

理，在发送的错误报告中包含的内容，如何注册到 Windows 错误报告数据库以及如何查询这个服务以获得错误报告。

图 8-2 Dr. Watson 消息框

Windows 错误报告的架构

Windows 错误报告（Windows Error Reporting，WER）是一种聚合故障数据的服务，使得 Microsoft 和独立软件供应商（Independent Software Vendor，ISV）可以很容易地访问与他们程序相关的故障数据。在图 8-3 中给出了 WER 服务的操作流程。

图 8-3 包含了两个主要的实体：

- 计算机，运行存在问题的程序并且将错误报告上传到 WER。
- ISV，监视与他们相关的故障，报告到 WER。

假定在世界的某个地方，有一台计算机正在运行由 ADND 企业开发的一个程序（在图 8-3 中以“进程 X”表示）。假定这个程序崩溃了，且用户看到了 Dr. Watson 界面并被询问是否希望将错误报告发送给 Microsoft。用户选择了发送，并且错误报告将通过安全的通道（HTTPS）发送给 WER 服务。然后，WER 服务将收到的错误包括进行分门别类（每一类称为一个桶）并保存。要使用这些错误报告，来自 ADND 企业的用户需要查询 WER 服务，找出与其程序相关的崩溃并且获得报告的错误信息。如果 ADND 得到了这些错误报告，那么就可以修正这个问题，并且提供一个回应，这样下一次当用户遇到相同的崩溃情况时，Dr. Watson 将给出相应的回应。这个回应可以是一个补丁或者是其他一些帮助信息。你可以看到，WER 服务是一种功能非常强大的机制，它提供了对错误报告信息的聚合功能，ISV 可以通过查询这些信息来改进程序。此外，ISV 还可以对已知的问题提供回应，并且将这个回应集成到

WER 的反馈循环中，从而使用户可以很方便地得到回应。

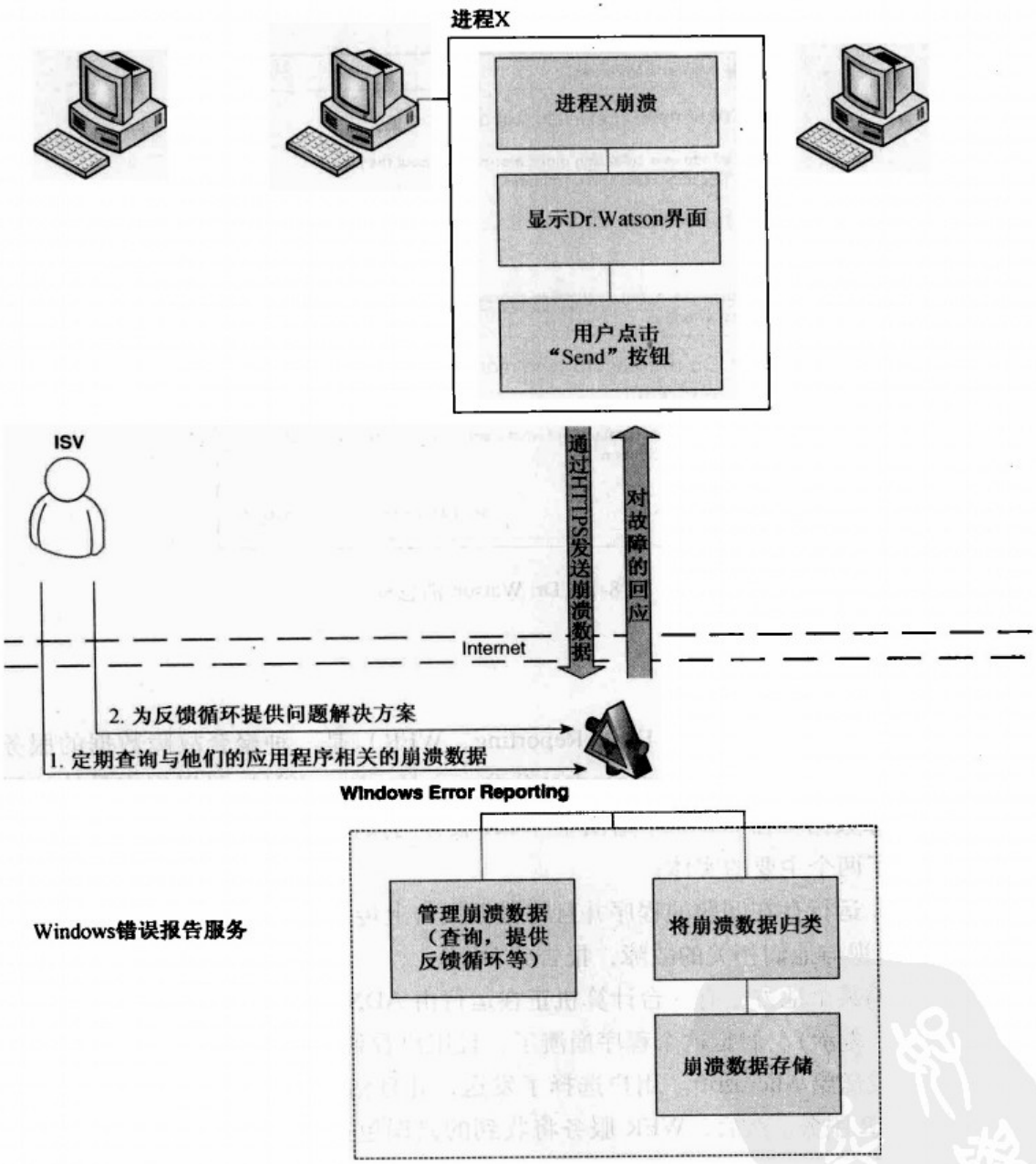


图 8-3　WER 功能概览

发送错误信息的重要性

当 Dr. Watson 界面出现并且提示某个程序崩溃时，你或许会问：为什么要发送这个信息？对这个问题可以做些什么？事实上，Microsoft 会非常认真地对待每个错误报告。毕竟，这就是为什么要实现这种服务的初衷。Microsoft 能主动地监测错误报告数据并将其转发到相应的产品团队。当开发出了一个补丁并且做好了发布准备时（通常是通过 Microsoft 升级），用户可以很容易地使用这个补丁。换句话说，用户对于错误的可见性有着直接的影响，因此应该确保将错误报告上传，这样 Microsoft 或者其他的 ISV 才有机会分析这个问题并提供补丁。

在接下来对 WER 的讨论中，我们将通过在前面章节中使用的程序 08SimpleExc. exe 来说明 WER 的使用过程。在使用 WER 时，第一步就是向 Windows 错误报告服务注册，接下来将进行详细描述。

1. 向 Windows 错误报告服务注册

要参与到（即查询错误报告）WER 中，必须首先完成一个注册过程。

注册过程可分为两个步骤：

- 创建一个用户账号
- 创建一个企业账号

跳转到以下 URL 开始注册过程，：

- https://winqual. microsoft. com/SignUp/

在加载页面后，会进入账号创建页面，如图 8-4 所示。

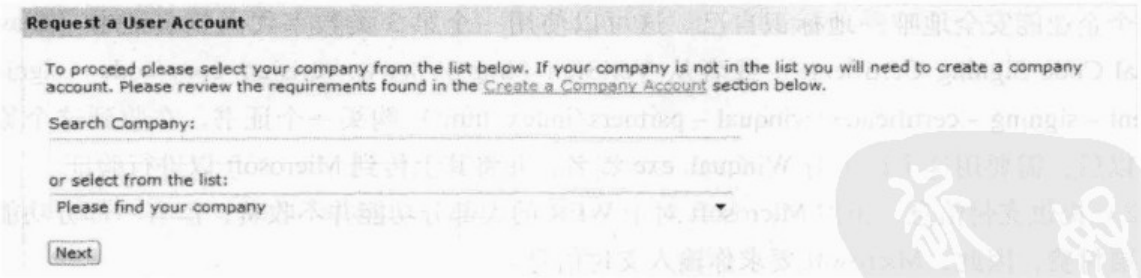


图 8-4 WER 注册过程的第一个页面

要创建一个用户账号，必须首先拥有一个企业账号。如果已经创建了一个企业账号，那么可以查找这个账号，或者在下拉列表中找到它。点击“Next”按钮进入到账号创建页面。由于我们还没有创建一个企业账号，因此点击“Create a Company Account”链接，这将开始企业账号的创建过程，如图 8-5 所示。

Create a Company Account

To establish a Winqual account for your company (a prerequisite for creating user accounts), you must establish your companies identity using a **VeriSign Certificate**. There are two Verisign certificates supported by Winqual for creating company accounts, and they are available at a discounted price through the links below:

- VeriSign Organizational Certificate **($99 USD)**
  This digital ID is only used by Winqual, and is only valid for establishing an account for your company in Winqual. Hardware submissions are not permitted with this digital ID.
  Users who wish to purchase a VeriSign Organizational Certificate after July 26, 2007 must first install a root certificate on the machine used for purchasing. Note the root certificate must also be installed on the code-signing machine in order to sign properly. Download the root certificate and instructions on how to install it here.
- VeriSign 'Microsoft Authenticode' Code Signing Digital ID **($399 USD)**
  The Code Signing digital ID is more versatile, and is the accepted standard for establishing ownership of code. Some applications within Winqual require the use of a Class 3 Code signing certificate (examples are: Hardware Logo signatures, Driver reliability signatures, and Driver Verification Testing). Using a code signing certificate enables you to digitally sign your 32-bit or 64-bit .exe (PE files), .cab, .dll and .ocx, files.

Winqual accounts are organized by company. To establish a Winqual account for your company, you will need to provide the following:

Winqual .exe — **Code signed Winqual.exe file**
The VeriSign ID will be analyzed and the company name and ID number will be extracted from the file.

Billing address — **Billing address**
Because there are fees for some submission types, we require a billing address to set up an account.

Contact data — **Contact data**
Create a user profile for the company account administrator.

Next

图 8-5 创建一个企业账号

在创建企业账号中包括三个步骤：

1）生成一个签名的 Winqual. exe 文件。对于要参与到 WER 中的企业来说，Microsoft 要求这个企业能安全地唯一地标识自己。这可以使用一个第 3 类数字代码签名证书（Class 3 Digital Code Signing Certificate）或者从 VeriSign（http://www. verisign. com/code - signing/content - signing - certificates/winqual - partners/index. html）购买一个证书。在收到这个签名证书以后，需要用这个证书对 Winqual. exe 签名，并将其上传到 Microsoft 以进行验证。

2）提供支付信息。虽然 Microsoft 对于 WER 的大部分功能并不收费，但有一部分功能仍然需要付费，因此，Microsoft 要求你输入支付信息。

3）这个过程的最后一部分将创建一个用于访问企业账号的用户账号。

我们首先从步骤 1 开始（签署 Winqual. exe 文件）。在前面提到过，出于安全性考虑，Microsoft 要求所有的 WER 企业账号都必须通过第 3 类数字代码签名证书或者某个正式的企业证书来标识。在剩下的章节中，假设已经从 VeriSign 获得了一个签名证书。第一个步骤就是从 Microsoft 下载需要签名的二进制文件。使用以下 URL 来下载 Winqual. exe：

https://winqual. microsoft. com/signup/winqual. exe

将文件保存到硬盘 C:\Sign。接下来下载对二进制文件签名的代码签名工具。通过以下

URL 来下载：

https://winqual.microsoft.com/signup/signcode.zip

将文件保存下来并解压到 C：\ Sign，此时会获得两个文件：

- Readme.rtf。这个文件包含了如何使用代码签名工具对二进制文件进行代码签名的方法。它还包含了一个密码，当解压 signcode.exe 文件（受密码保护）时会用到它，同样位于该 zip 文件中。
- Signcode.exe。这个程序用于对 Winqual.exe 文件进行签名。

将 signcode.exe 文件释放（解压密码时在 readme 文件中）到与 winqual.exe 文件相同的位置上（C:\Sign）。同样，确保将代码签名证书文件（扩展名为 .spc）和私钥（扩展名为 .pvk）复制到相同的位置上。然后，使用以下命令行来对 winqual.exe 文件签名：

```
C:\Sign>signcode.exe /spc myCert.spc /v myKey.pvk -t
 http://timestamp.verisign.com/scripts/timstamp.dll winqual.exe
Succeeded
```

记得将 mycert.spc 和 mykey.pvk 分别替换为你的证书文件名和私钥文件名。在签名过程中需要输入一个私钥密码。键入在购买证书时由 VerSign 提供给的密码。如果签名过程成功，那么将显示一个“Succeeded”消息。如果发生了错误，那么要检查是否正确地输入了证书文件名和私钥文件名，并且它们是否位于与 signcode.exe 文件相同的目录下。

下一个步骤就是将签署好的 winqual.exe 文件上传到 Microsoft 来进行验证。在图 8-5 中给出的页面上，点击 Next 按钮。在下一页中上传签署好的 Winqual.exe 文件，如图 8-6 所示。

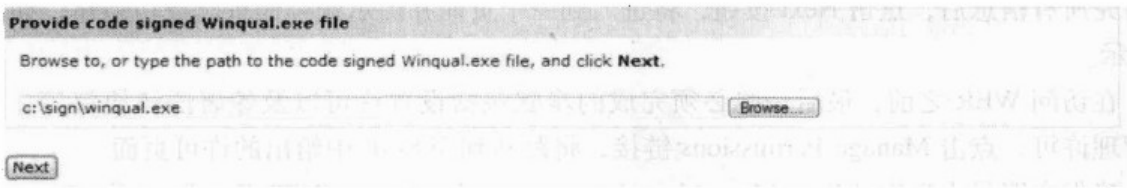


图 8-6 上传签署好的 Winqual.exe 文件

只需键入签署后的 winqual.exe 文件的路径，然后点击 Next 按钮开始上传文件。上传完文件后，下一个页面是 Billing 页面，如图 8-7 所示。

在前面提到过，WER 的大部分功能都是免费的，但也有一些功能是要收费的。如果用户使用了需要付费的 WER 服务，那么就要提供支付信息。

请键入企业的支付信息，然后点击 Next 按钮，会进入到账号创建页面，如图 8-8 所示。

用户名和密码这两项表示在访问 WER 站点时使用的登录信息。填写所有这些信息，并特别注意在底部给出的高强度密码要求。这些密码要求是非常重要的，它能确保企业错误报告信息的安全。

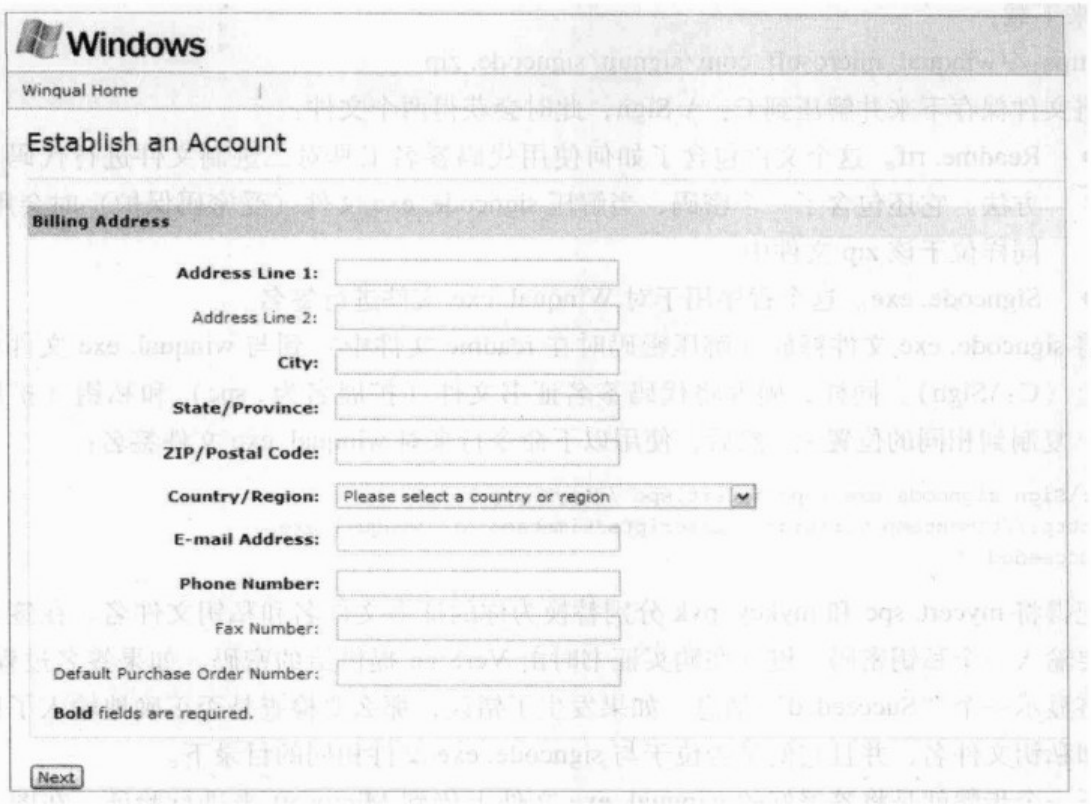


图 8-7 Billing 信息页面

填写完所有信息后，点击 Next 按钮，将进入到一个页面并提示账号信息已成功创建，如图 8-9 所示。

在访问 WER 之前，最后一些必须完成的步骤包括设置许可以及签署法律协议等。首先来管理许可。点击 Manage Permissions 链接，将跳转到图 8-10 中给出的许可页面。

确保在账号中选中“Sign Master Legal Agreements”，“View WER Data”以及“Download WER Data”等复选框，然后点击 Update 按钮。注意，在 WER 中可能有多个账号与一个企业账号相关。如果要使不同的用户拥有不同级别的访问权限（如图 8-10 所示），那么这是非常有用的。例如，一个用户可以访问错误报告，而另一个用户可以签署法律协议。

接下来，回到图 8-9 中的页面来签署 Microsoft 要求的法律协议。点击“Sign Legal Agreements”链接进入 Windows 错误报告的法律协议页面。仔细阅读给出的所有信息，如果接受这个协议，需要在最后一页的底部填写信息以签署这个协议。如果想获得协议的一份副本，可以在表格中键入企业的信息，然后打印一份副本。

Windows

Winqual Home

Establish an Account

Profile

Please make a note of your User Name and Password you will need to sign in if your account request is approved.

User Name:

Password:

Confirm Password:

Secret Question: What is my favorite movie?

Secret Answer:

Full Name:

Work E-mail Address:

Work Phone Number:

Work Fax Number:

Bold fields are required.

Password Requirements:
A password must:

- be at least 8 characters long and no longer than 16 characters
- have at least 1 lower-case alphabetic character
- have at least 1 upper-case alphabetic character
- have at least 1 number
- have at least 1 punctuation character/symbol
- have at least 1 non-alpha (number or a punctuation/symbol) within the 2nd to 6th character

Next

图 8-8 Profile 信息页面

Windows

Winqual Home Help

Establish an Account

Attention!

Your account has been successfully created!

Getting Started

There are two final steps which you need to perform before you can use many of the services available on Winqual:

1. Manage Permissions
   You must provision your own permissions for the following services:
   - **Legal** (Required to Sign Master Legal Agreements)
   - **Windows Error Reports** (Required to access WER Data)
   - **Driver Distribution Center** (Required to edit DDC Data)
   - **Submissions** (Required to access submission options)
2. Sign Legal Agreements
   A signed Legal Agreement might be required to access certain services available on Winqual such as Windows Error Reports. Only users with the **Sign Master Legal Agreements** permission have the authority to sign agreements for your company.

Learn more about the our services.

图 8-9 账号成功创建

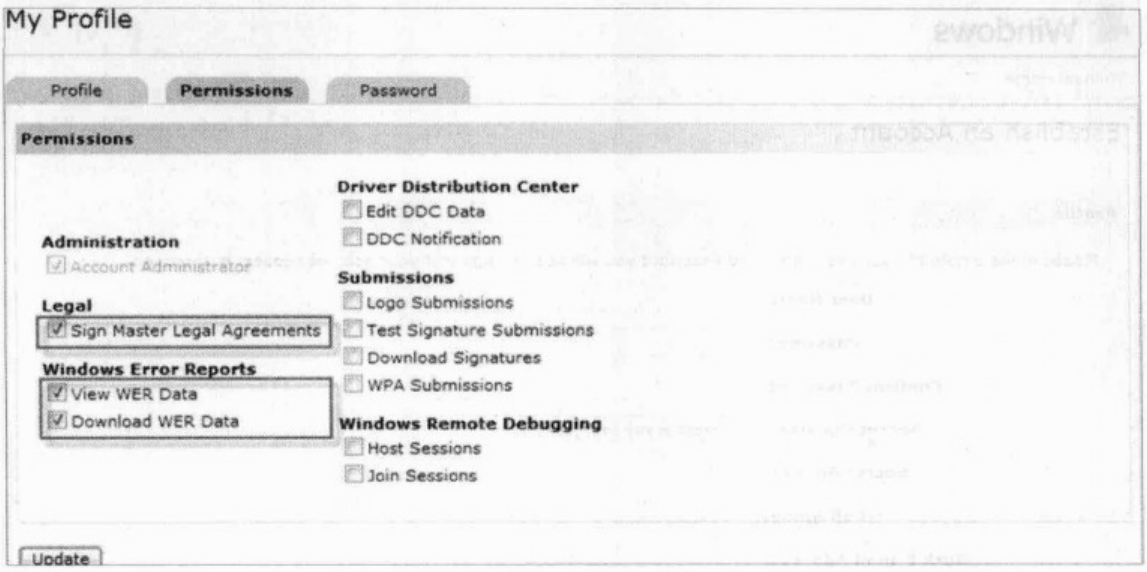


图 8-10 管理许可

现在，签署过程就完成了，你可以访问 WER 的所有功能，在以下 URL 用刚才创建的账号登录：

https://winqual.microsoft.com/default.aspx

2. 跳转到 WER 网址

当登录到 WER 网址时，你会看到一个包含最新 Winqual 声明的页面。这个声明的左边是一个面板，用于跳转到该站点的不同部分。在这个面板中包含了三个主要部分，分别是：

- Windows Logo Program
- Windows Error Reports
- Driver Distribution Center

在本章中，我们只介绍该站点的 Windows Error Reports 部分，也就是 WER 的 Software 部分。在图 8-11 中给出了在 Software 菜单中可用的选项。

在 Event Views 类别的 Product Rollups 选项中给出了一个视图，这个视图根据产品的名字和版本来组织错误报告。图 8-12 是 Product Rollup 页面。

在图 8-12 中给出了一个已注册的产品：Advanced .NET Debugging，其中有两列使你能够进一步了解与这个程序相关的事件（崩溃或者其他问题）。

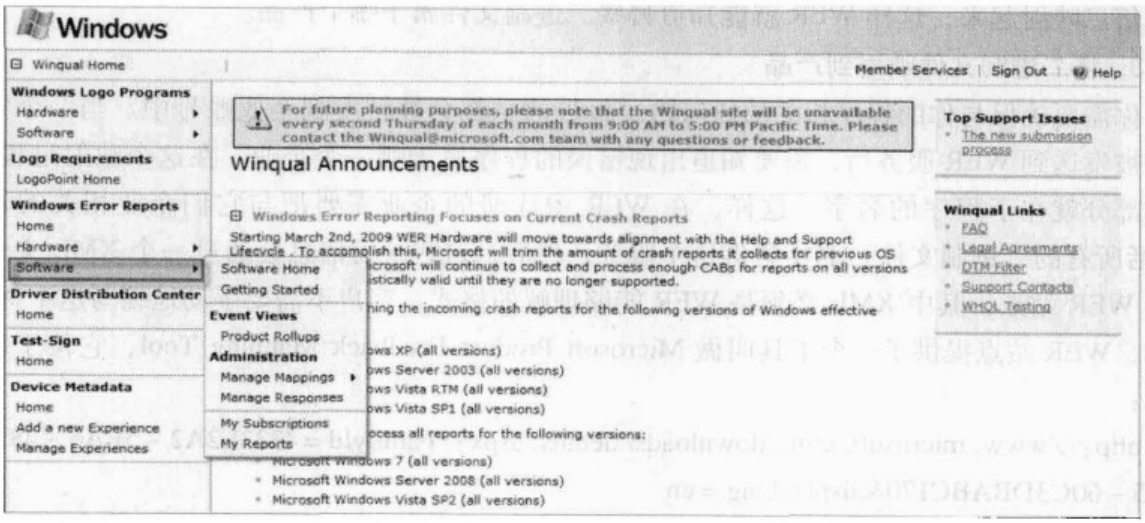


图 8-11 WER Software 选项

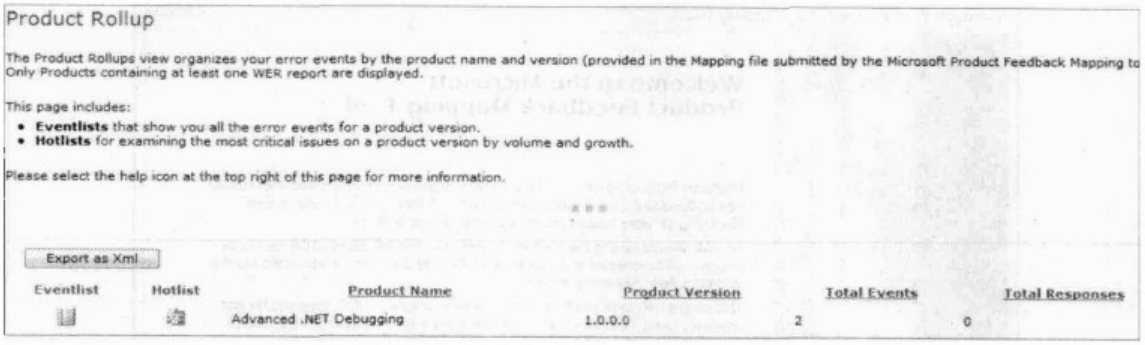


图 8-12 Product Rollup 页面

- Eventlist。点击这个图标会跳转到一个页面，其中详细给出了在程序中发生的事件列表。
- Hotlist。点击这个图标会跳转到一个页面，其中详细给出了所选程序在最近 90 天中发生最多的事件。

下一个菜单项是 Administration 类别。其中包含了以下选项：

- Manage Mappings。这个选项用于将二进制文件与产品映射起来，这样 WER 就知道哪些二进制文件属于哪个产品。在本章的后面，我们将讨论如何创建一个映射文件。
- Manage Responses。这个选项用于定义对用户报告问题的回应，这将形成一种反馈循环，在回应中可以包含任何东西，例如反馈信息或者补丁。在本章的后面，我们将讨论如何生成一个回应。

既然我们已经熟悉了 WER 站点的常规布局，现在就可以将产品的二进制文件与特定的

产品信息映射起来，这样 WER 就能知道哪些二进制文件属于哪个产品。

3. 将二进制文件映射到产品

你需要确保与你的程序相关的任何错误报告信息都会被转发到企业账号中。当一个错误报告被发送到 WER 服务后，需要知道出现错误的程序属于哪一个企业。在这个映射过程中，关键部分就在于程序的名字。这样，在 WER 中注册的企业需要把与它们企业相关的程序（包括所有的二进制文件）名字告诉这个服务。然后，这些映射信息会通过一个 XML 文件提交给 WER 站点，其中 XML 必须是 WER 能够理解的格式。客户不需要手动地编写这个 XML 文件，WER 站点提供了一个工具叫做 Microsoft Product Feedback Mapping Tool，它位于以下 URL：

http://www.microsoft.com/downloads/details.aspx? FamilyId = 4333E2A2 - 5EA6 - 4878 - BBE5 - 60C3DBABC170&displaylang = en

下载并安装这个工具。当安装完后，点击 Start -> Programs -> Microsoft Product Feedback Mapping Tool，会出现一个向导来指导你完成映射过程。向导的第一页如图 8-13 所示。

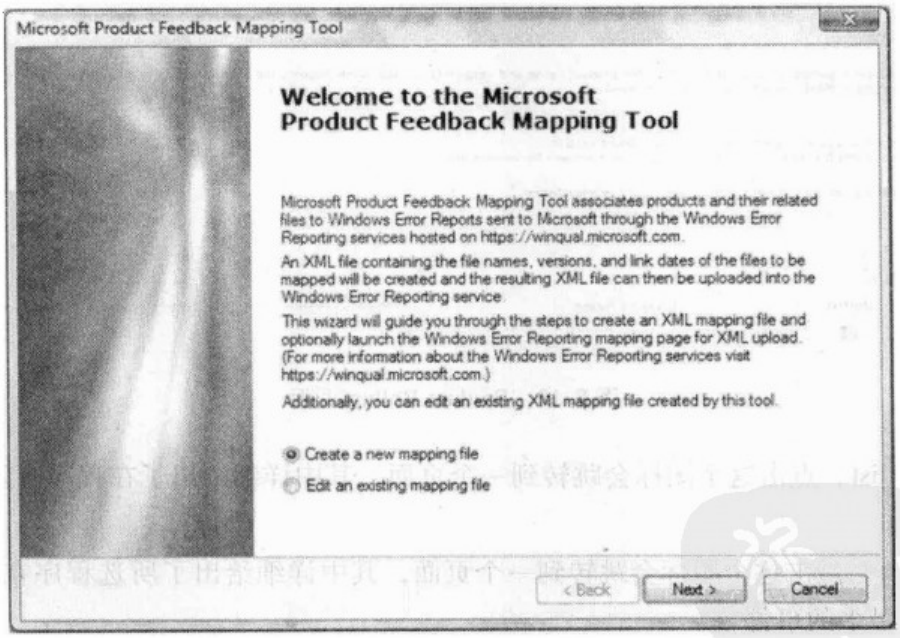


图 8-13 Microsoft Product Feedback Mapping Tool

为了说明设置映射文件的过程，我们使用在本章前面用过的程序 08SimpleExc.exe。首先确保选中了“Create a New Mapping File”，然后点击 Next。在图 8-14 中给出了“Gathering Product Mapping Information”页面。

在图 8-14 中包含了以下选项。请确保键入了如图所示的信息。

- Product File(s) Directory Path。指定想要映射的程序二进制文件的路径。

- Product Name。指定与这个二进制文件相关的产品名字。请注意，产品名字只是在WER站点上使用的一个友好名字，方便用户更有效地分组和搜索错误信息。
- Product Version。与这个二进制文件相关的产品版本。请注意，产品版本只是在WER站点上使用的一个友好版本，方便用户更有效地来分组和搜索错误信息。

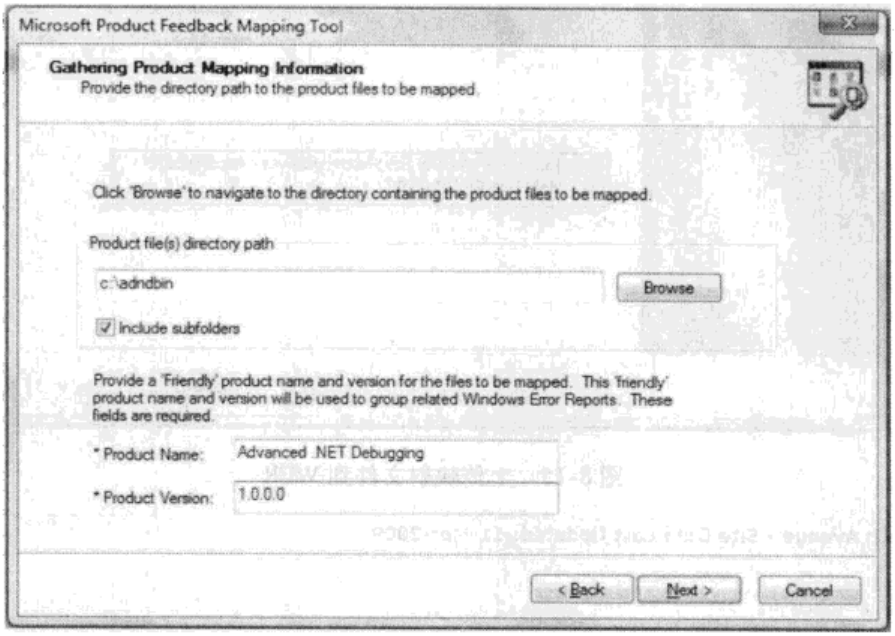


图8-14 收集产品映射信息

当键入所有信息后，点击Next，接下来点击下一个Next。现在，这个向导会要求你为映射文件指定一个文件名。键入以下映射文件路径，点击Next：

```
C:\testmap.xml
```

图8-15给出了这个过程的最后一步，即将映射文件上传到WER站点。

确保选中了这个复选框，接着点击Finish。现在，这个向导会启动浏览器，并给出File Upload页面，如图8-16所示。

键入刚才创建的映射文件的路径，然后点击Submit。在成功上传后，文件映射和上传的过程也就完成了。如果有多个产品，那么就要为每个产品都执行一次映射过程。

再回到主WER站点后，可以在左边导航面板中选择Software，然后点击Mange Mappins选择来管理产品和文件的映射。你可以选择管理产品和文件的映射，或者上传一个映射文件。例如，在上传映射文件后选择File Mapping链接，如图8-17所示。

图 8-15 上传映射文件到 WER

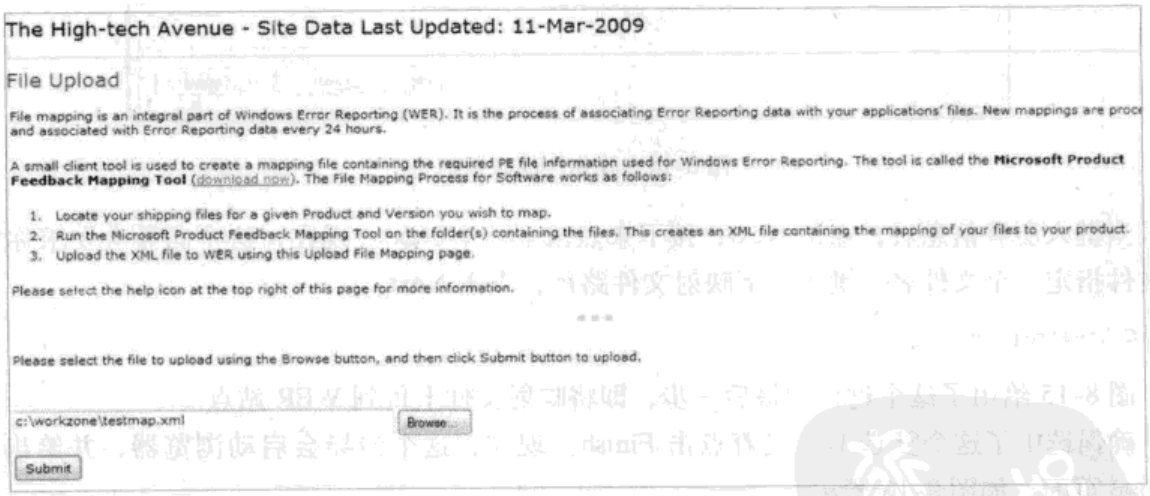


图 8-16 映射文件上传页面

从图 8-17 中，我们可以看到有多个文件映射，每个映射都有特定的属性（例如链接日期和映射日期）以及一些管理信息，例如谁创建了这个映射关系以及他的电子邮件地址。

现在，我们已经创建了一个产品和文件映射，接下来将介绍 WER 报告的生成。我们会看到如何生成用户发送的错误信息报告以及如何分析每个错误报告（例如崩溃转储）。

The High-tech Avenue - Site Data Last Updated: 11-Mar-2009

Manage File Mappings

Windows Error Reporting for Software applications provides two ways to view and manage file mappings: At the file level, and by the products the files are grouped by.

Files are mapped using the XML output of the Microsoft Product Feedback Mapping Tool. The XML mapping files are uploaded to Windows Error Reporting using the Upload File Mappings link under the Manage Mappings left navigational menu selection. When files are unmapped, they will appear disabled in the list until the system processes the chang Deleted files and products can take up to 24 hours to process and update the product rollups.

The Manage Product Mappings and Manage File Mappings pages work together:

- The **Manage Product Mappings** page lists all your company's products that have files related to them.
- The **Manage File Mappings** page displays all the files that are related to your company's products.

Please select the help icon at the top right of this page for more information.

Export as Xml

Prev Next Page 1 of 2

Product Mappings	File Name	File Version	Link Date	Map Date	Mapped By
	01MDASample.exe	0.0.0.0	06-Feb-09 03:20:07 PM	16-Mar-09 06:00:36 AM +00	manoh
	01sample.exe	1.0.0.0	26-Oct-07 06:23:24 AM	20-Nov-07 01:35:14 PM +00	manoh
	02ExcSample.exe	0.0.0.0	06-Feb-09 03:20:07 PM	16-Mar-09 06:00:36 AM +00	manoh
	02sample.exe	1.0.0.0	26-Oct-07 06:23:24 AM	20-Nov-07 01:35:14 PM +00	manoh
	02Simple.exe	0.0.0.0	06-Feb-09 03:20:07 PM	16-Mar-09 06:00:36 AM +00	manoh
	02TypeSample.exe	0.0.0.0	06-Feb-09 03:20:08 PM	16-Mar-09 06:00:36 AM +00	manoh

图 8-17 WER 文件映射

4. 查询 Windows 错误报告服务

在创建了一个账号并将 08SimpleExc. exe 映射到产品之后，现在我们来看看如何在 WER 中查询已上传的错误报告。运行几次 08SimpleExc. exe，并当程序崩溃时，告诉 Dr. Watson 将错误信息上传到 WER 站点。注意，在用户上传报告与在网上能查询到报告之间存在一定的时间延迟。

在上传并且可以看到错误报告后，你会在 Product Rollup 页面上看到一张产品列表，如图 8-18 所示。

Product Rollup

The Product Rollups view organizes your error events by the product name and version (provided in the Mapping file submitted by the Microsoft Product Feedback Mapping to Only Products containing at least one WER report are displayed.

This page includes:

- **Eventlists** that show you all the error events for a product version.
- **Hotlists** for examining the most critical issues on a product version by volume and growth.

Please select the help icon at the top right of this page for more information.

Export as Xml

Eventlist	Hotlist	Product Name	Product Version	Total Events	Total Responses
		Advanced .NET Debugging	1.0.0.0	2	0

图 8-18 Product Rollup 中的错事件

在图 8-18 中给出了已映射的产品（Advanced . NET Debugging）和已经报告的事件总数。此外，在 Eventlist 和 Hotlist 等列中包含了一些图标，这些图标显示了在这个产品上发生的所

有事件，以及在最近 90 天中发生的最多的事件，Hotlist 就是具有这种功能的一种快捷方式。在图 8-19 中给出了当点击事件列表图标时出现的事件列表页面。

Event List for product: Advanced .NET Debugging 1.0.0.0

The Event List page displays events for a specified product or search criteria. This page is useful for reviewing the complete list of events for a given product or search cri

Please select the help icon at the top right of this page for more information.

Show filter

Export as Xml

Event ID	Cabs	Responses	Total Hits	Avg. Hits	Growth Percent	Event Type	Application Name	Application Version	Module Name	Module Version
504156229			2	0.02		CLR20 Managed Crash	08SimpleExc.exe	0.0.0.0	08SimpleExc	0.0.0.0

图 8-19　Advanced . NET Debugging 的事件列表

事件列表页面中包含了一张表格，其中每一行都表示一个不同的错误事件。在图 8-19 中，我们可以看到只有一个事件的总数为 2。这张表格还给出了导致了这个错误报告的事件种类，这里的事件类型为 CLR20 Manager Crash，表示该事件是由于某个基于 CLR 2.0 的托管程序发生崩溃而产生的。如果点击这个事件 ID，那么将看到与这个特定事件相关的信息。事件的详细信息页面分为三个主要部分：

- Event Signature。由于一个产品可以有多个相关事件，因此每个事件都必须是唯一的。构成事件唯一性的信息包括：程序的名字和版本号，模块的名字和版本号以及导致事件发生的模块偏移。从图 8-20 中可以看到，发生崩溃的位置是在模块 08SimpleExc. exe 中的偏移 0x17 处。
- Event Time Trending Details。在 Event Time Trending Details 的图中给出了事件随时间的变化情况。在图 8-20 中，我们可以看到事件的数量在 3 月 16 日达到了顶峰，并且随时间的推移而逐渐递减。
- Platform details。最后一部分给出了这个事件的平台详细信息，其中包括操作系统和语言等信息。当要找出在特定配置情况下发生的问题时，这部分信息非常重要，并且能够提供一些有用的线索，例如只有在非英语版本的产品中才会发生的事件。

在图 8-20 中给出了为 08SimpleExc. exe 显示的数据。

在事件详情页面中还包含一个 Cab 数据收集部分，如图 8-21 所示。

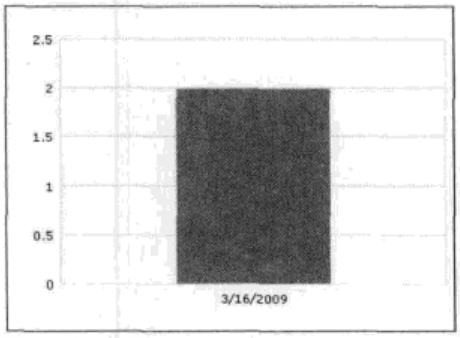
在数据收集部分中，可以通过点击 Cab Status 图标进入到特定事件的可用 Cab 列表，或者点击 Data Request 图标来修改特定事件的数据收集策略。图 8-22 给出了数据收集策略窗口。

Event Signature

Application Name	08SimpleExc.exe
Application Version	0.0.0.0
Module Name	08SimpleExc
Module Version	0.0.0.0
Offset	0x17
Exception Code	

Event Time Trending Details

Show filter



Total Hits:	2
First Seen:	16-Mar-09
Last Seen:	16-Mar-09
Average Hits:	0.02
Event Type Name:	CLR20 Managed Crash

Platform Details - 90 days

Operating System Breakout:

Windows Vista SP1 / Windows Server 2008 >99%

Language Breakout:

English - United States >99%

图 8-20 关于 08SimpleExc. exe 事件详细信息

Data Collection

Cab Status	
Data Request	

图 8-21 数据收集部分

记住，当程序发生错误时，发生错误的机器将与 WER 进行通信以检察数据收集策略。如果策略发生了变化，并且要求重新上传，那么客户端机器会创建一个与该策略相对应的 Cab，并且将其上传到 WER。在数据收集策略窗口中可以指定在这个过程中收集的额外信息。除了系统信息外，还可以收集堆信息，或者根据预定义的一组环境变量来收集额外的文件。最后，还可以通过这种新的策略指定能收集额外 Cab 的个数。

在图 8-19 中，最后一个重要的列是 Cabs 列。点击图标将得到这个事件的一个 Cab 列表。每个 Cab 包含了一组文件，表示由用户发送的事件信息（每次上传一个 Cab）。Cab 中最重要文件之一就是当发生故障时生成的转储文件。在前面已经解释过，当进行事后调试时需要使

用转储文件。

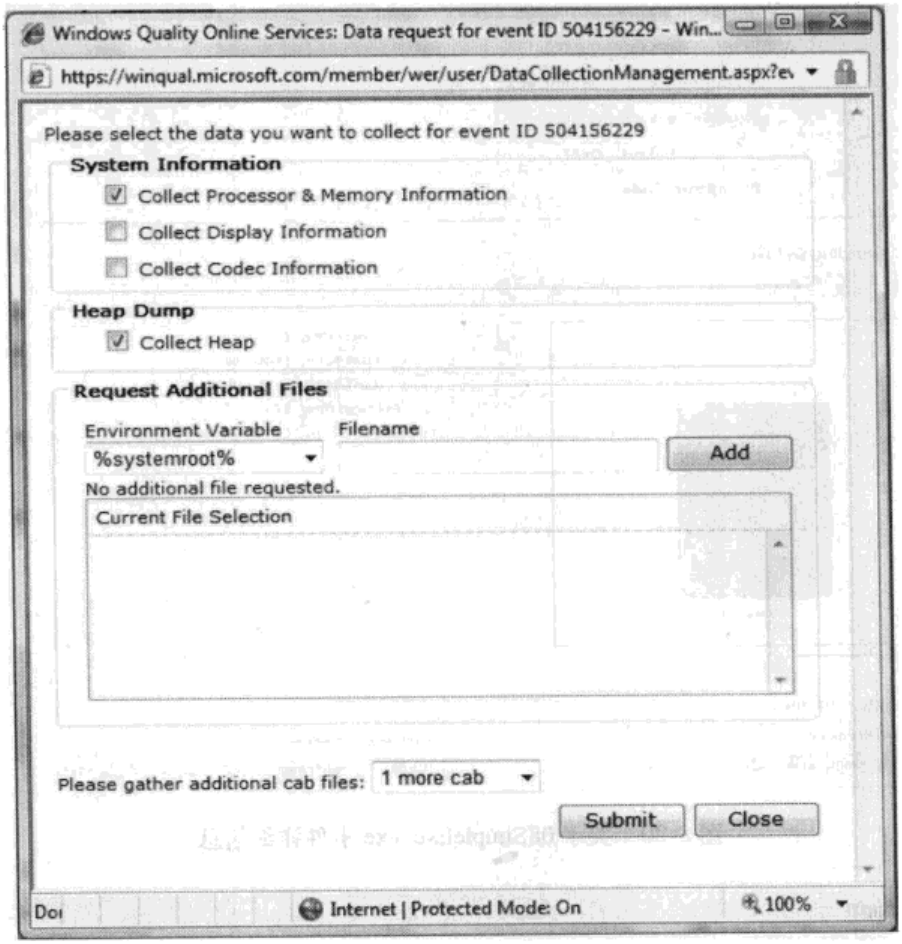


图 8-22 数据收集策略

现在，我们已经看到了通过 WER 站点可以访问的各种信息，并详细介绍了如何使用用户上传到 Microsoft 的信息，接下来将介绍这个过程中的最后一个关键步骤：在分析完问题后，如何为用户提供回应。

5. 提供回应

要针对某个事件为用户提供回应，你必须跳转到这个事件的 Event Details 页面。如果还没有为某个事件生成回应，那么在页面的最顶层部分提供了一些选项来注册回应，如图 8-23 所示。

可以在三种不同的级别上注册回应：

- Event。当单独提供一个补丁，并且这个补丁没有被集成到产品的升级包时，将使用这个级别。
- Application。如果在 Application 级别上提供回应，那么会创建一种基于规则的回应，

只有使用某个特定版本程序的用户才能看到这个回应。它的形式可以升级（例如一个新的版本）。

- Module。在 Module 级别上提供回应同样将创建一种基于规则的回应，只有使用某个特定版本程序的用户才能看到这个回应。它的形式可以升级（例如一个新的版本）。

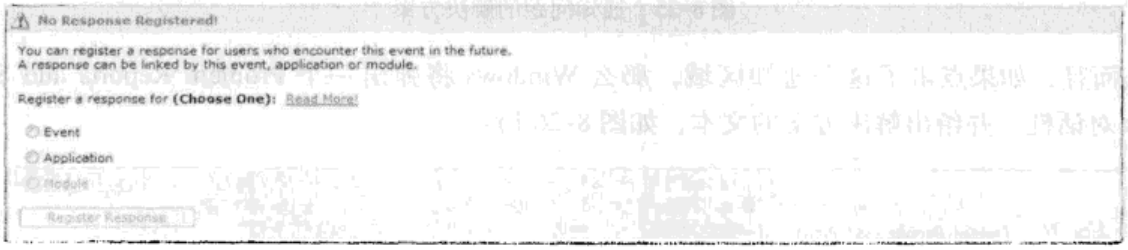


图 8-23 事件回应选项

对于这里的情况，我们选择使用基于事件的回应注册。选择单选按钮 Event，并点击 Register Response。接下来填写关于这个事件回应的详细信息。在注册回应之前，需要填写以下信息：

- Products。在 Product 域中键入产品的名字。
- URL of Solution/Info。键入这个回应的 URL。该 URL 应该指向一个页面，其中包含了这个回应需要的所有信息。
- Response Template。可以选择一个预定义模板来实现回应，或者使用自定义的模板。预定义的模板包括：System Does Not Meet Minimum Requirements、Product Upgrade、Upgrade to New Version 等。根据在下拉框中选择的不同版本，模板的“预览（Preview）”也将发生相应的变化。
- Response Template Preview：对回应中包含的信息进行预览。
- Additional Information：键入在这个回应中包含的任何额外信息。

当所有信息填写完毕后，继续注册这个回应，接下来会跳转到 Response Mangement 页面，并列出已经注册的所有回应。请注意，一个新注册的回应并不会立即生效，而是需要几天时间来完成一个审批过程。在 Response Management 页面中还可以管理已经创建的所有回应。你可以查看这些回应的详细信息，进行修改，或者删除某个不再适用的回应。

回应将如何提交给用户？当用户下一次遇到某个故障，并且有一个回应与这个故障相关时，他将看到一个对话框，如图 8-24 所示。

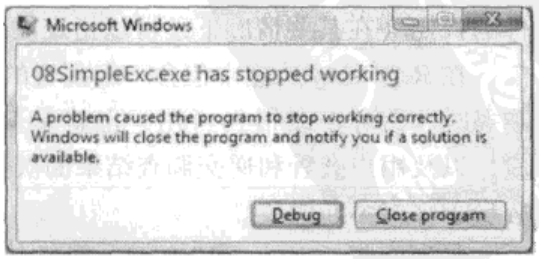


图 8-24 用户遇到一个带有回应的程序故障

如果用户点击 Close Program 按钮，会显示一个提示信息，提示用户遇到的问题有一个解决方案。在图 8-25 中给出了一个示例。

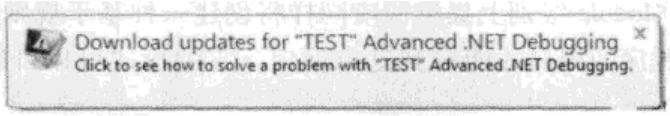


图 8-25　通知问题的解决方案

而且，如果点击了这个通知区域，那么 Windows 将弹出一个 Problem Reports and Solutions 对话框，并给出解决方案的文本，如图 8-26 所示。

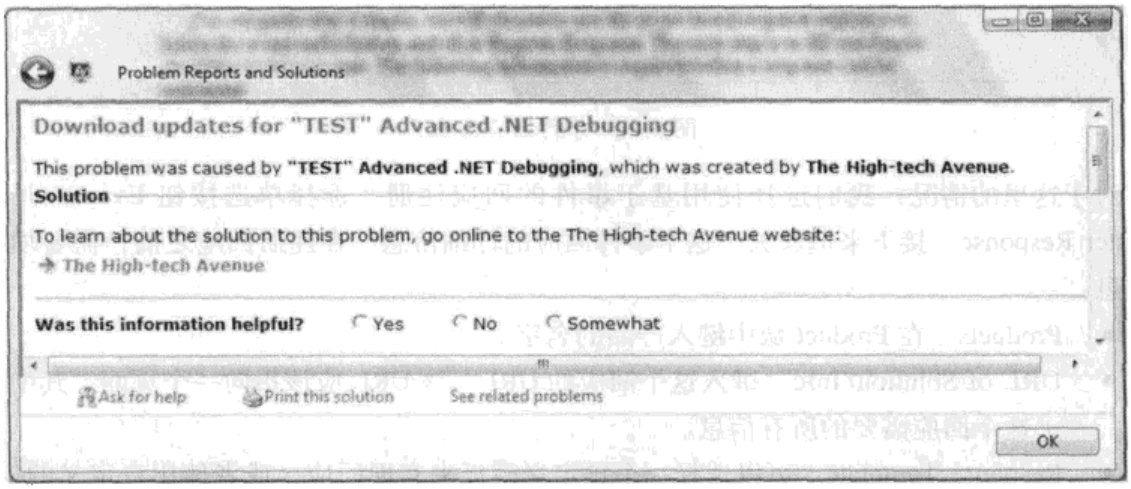


图 8-26　用户遇到一个带有回应的程序故障

用户可以点击链接（The High-Tech Avenue）跳转到一个包含详细解决方案的站点。

6. 报告与订阅

在 WER 服务中包含了两个新功能，分别为报告（Reporting）和订阅（Subscription）。订阅是指根据指定的条件来注册通知消息，并且可以在“Software”的“My Subscriptions”菜单项中访问。在图 8-27 中给出了各种不同的标准。

要激活一个订阅，请选中需要订阅消息旁边的复选框。每个订阅还可以指定发送电子邮件的日期（在某些情况下还可以指定发送频率）。

在 Reporting 功能中包含一组预置的报告，可以对 WER 中保存的数据进行分析。在撰写本书时，只有一个报告可用，也就是 Response Satisfaction 报告，它能给出整体的回应满意度，以及用户查看和递交调查结果的次数。报告是一个非常新的功能，我们可以预见报告的数量未来将不断增长。

正如你所看到的，WER 是一种功能非常强大的服务，它能够检测程序在实际应用中的表现行为。允许用户发送错误信息来进行分析，并且为发现的问题创建一个回应。这是一种非

常强大的技术，它有效地减轻了用户在遇到软件问题时的痛苦。

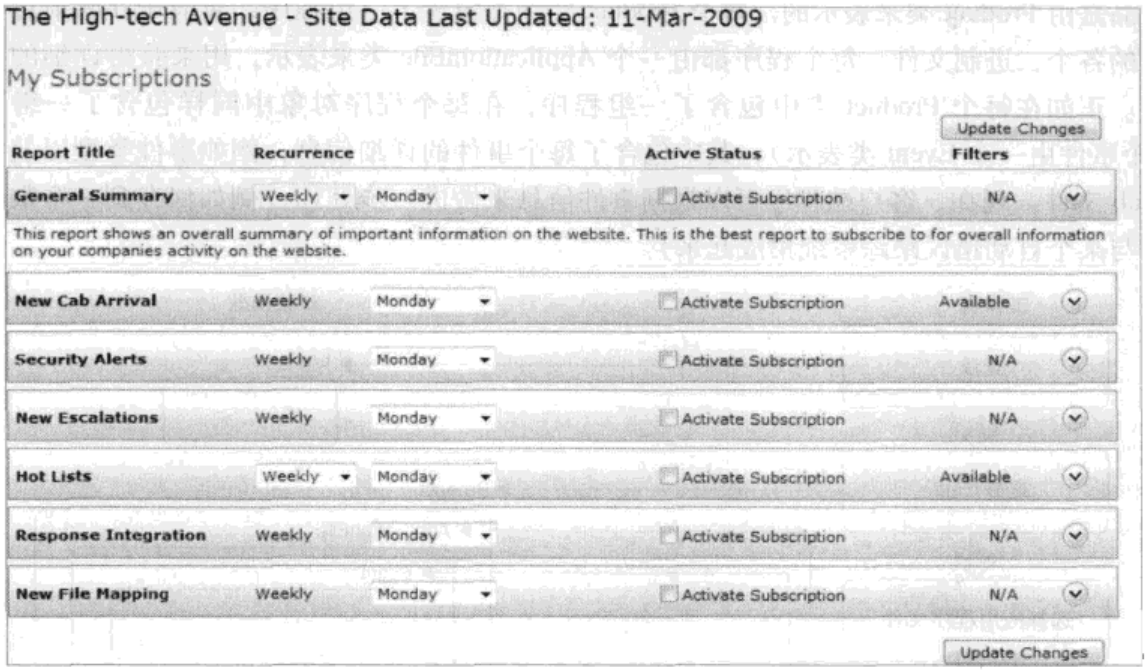


图 8-27 可用的订阅

7. Windows 错误报告的程序化访问

对于企业监视他们的程序在发布之后的行为以及如何对问题进行回应来说，Windows 错误报告服务是非常有用的。只需创建一个账号，并注册相应的产品和二进制文件，就可以得到实际应用中的大量故障信息。在使用 WER 时，需要注意的一个问题就是，用户需要定期地登录到这个站点并检查是否发生了任何新的事件。如果发生了，那么用户可以查看相应的事件数据，并下载与这个事件相关的 Cab 来进行事后分析。监视程序的更好方法是使用程序化访问（programmatic access），通过 WER 提供的 Web Service API 来实现。Web Service API 使企业能够创建自动化错误监测系统，在不需要人为介入的情况下就可以下载所有事件数据。WER 团队已经公布了对 Web Services 访问层（见 http://wer.codeplex.com/），并且包含了一组示例项目（包括从 WER 抽取信息的 Windows Vista Gadget）。此外，它还包含一个客户端程序集（见 http://wer.codeplex.com/Release/ProjectReleases.aspx? ReleaseId = 12825），这个程序集公开了 WER 对象模型，使开发人员能很容易地开始编写他们自己的监测程序。在图 8-28 中给出了 WER 工作流以及客户端对象模型的示意图。

每个 WER 工作流都是通过 Login 类登录到 WER 系统的。在登录过程中使用的证书是在该账号上某个授权用户（例如主管理员）的证书。在登录成功后，Login 对象通常被缓存在客户端程序中，并且在随后的 WER 操作中使用。接下来，客户端程序将枚举这个账号的可

用产品（之前是通过 Product Feedback Mapping Tool 映射的），并且可以选择感兴趣的产品。该产品是由 Product 类来表示的。每个 Product 实例都包含了一组程序，对应于在映射过程中映射的各个二进制文件。每个程序都由一个 ApplicatioinFile 类来表示，用来获得详细的程序信息。正如在每个 Product 类中包含了一组程序，在每个程序对象中同样包含了一组事件（每个事件由一个 Event 类表示），其中包含了每个事件的详细信息，例如事件类型以及相应的 Cab 文件。现在，客户端程序可以根据事件信息来做进一步处理（例如保存到一个数据库或者与某个自动错误跟踪系统集成起来）。

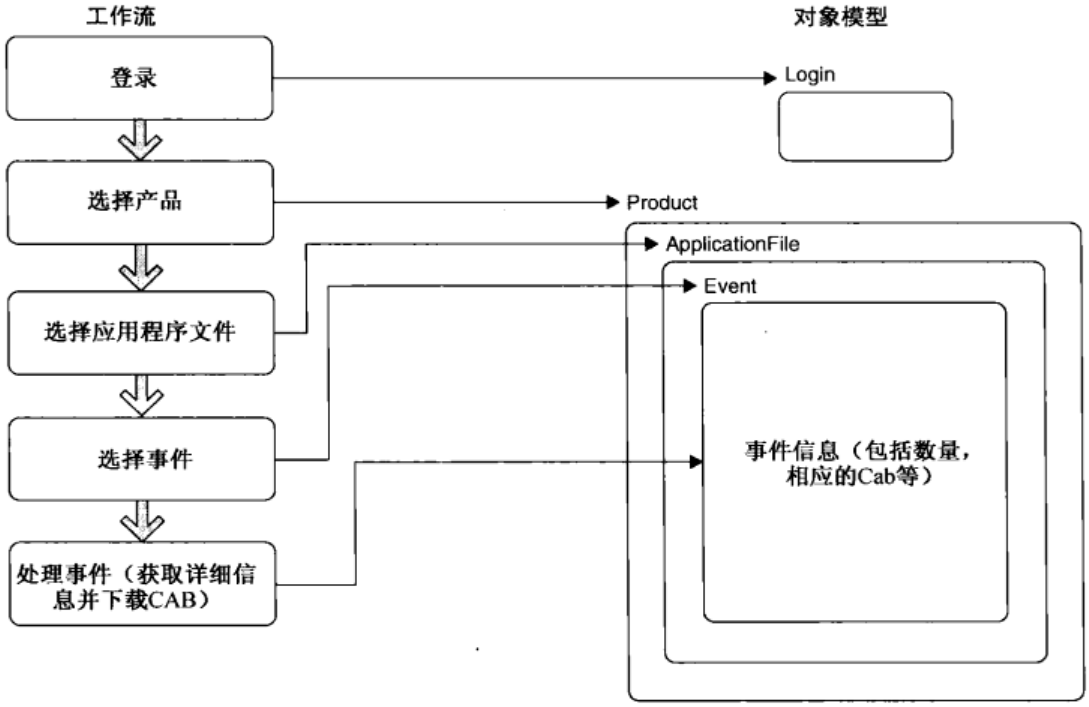


图 8-28　WER 工作流以及客户端对象模型

要更好地理解与 WER 集成的过程，我们来看一个基于控制台的客户端，它能够获得某个指定事件的详细信息。在清单 8-2 中给出了程序的源代码。

清单 8-2　与 WER 服务交互的示例程序

```
using System;
using System.IO;
using System.Text;
using System.Runtime.InteropServices;
using Microsoft.WindowsErrorReporting.Services.Data.API;

namespace Advanced.NET.Debugging.Chapter8
{
    class WerConsole
```

```
{
    static void Main(string[] args)
    {
        WerConsole s = new WerConsole();
        s.Run();
    }
    public void Run()
    {
        int eventId;
        string product, file, cabLoc, userName, password;
        Login login;

        Console.Write("Enter user name: ");
        userName = Console.ReadLine();

        Console.Write("Enter password: ");
        password = Console.ReadLine();

        Console.WriteLine("Login into WER...");
        login=WerLogin(userName, password);
        Console.WriteLine("Login succeeded");

        Console.Write("Enter Product: ");
        product=Console.ReadLine();

        Console.Write("Enter File: ");
        file=Console.ReadLine();

        Console.Write("Enter Event ID: ");
        eventId=Int32.Parse(Console.ReadLine());

        Console.Write("Enter Location to store CABs: ");
        cabLoc = Console.ReadLine();
        if (Directory.Exists(cabLoc) == false)
        {
            Directory.CreateDirectory(cabLoc);
        }
        Event e=GetEvent(product, file, eventId, ref login);
        Console.WriteLine("Event succesfully retreived");
        Console.WriteLine("Event ID: " + e.ID);
        Console.WriteLine("Event Total Hits: " + e.TotalHits.ToString());
        Console.WriteLine("Storing CABs...");
        foreach (Cab c in e.GetCabs(ref login))
        {
            try
            {
                c.SaveCab(cabLoc, true, ref login);
            }
            catch (Exception)
            {
```

```
            }
        }
        Console.WriteLine("CABs stored to: " + cabLoc);
    }

    public Login WerLogin(string userName, string password)
    {
        Login login = new Login(userName, password);
        login.Validate();
        return login;
    }

    public Event GetEvent(string pr,
                          string fi,
                          int eventId,
                          ref Login login)
    {
        foreach (Product p in Product.GetProducts(ref login))
        {
            if (p.Name == pr)
            {
                ApplicationFileCollection ac =
                          p.GetApplicationFiles(ref login);
                foreach (ApplicationFile file in ac)
                {
                    if (file.Name == fi)
                    {
                        EventPageReader epr=file.GetEvents();
                        while (epr.Read(ref login) == true)
                        {
                            EventReader er = epr.Events;
                            while (er.Read() == true)
                            {
                                Event e = er.Event;
                                return e;
                            }
                        }
                    }
                }
            }
        }
        throw new Exception("Event Not Found");
    }
  }
}
```

清单 8-2 中的源代码和二进制文件位于以下文件夹中：

- 源代码文件：C:\ADND\Chapter8\WerConsole
- 二进制文件：C:\ADNDBin\08WerConsole.exe

在编写示例程序时，WER 客户端程序集（Microsoft. WindowsErrorReporting. Services. Data. API. dll）位于文件夹 C:\ADND \Chapter8\WerConsole 中，并且在构建这个项目时会被自动放到文件夹 C:\ADNDBin 下。要获得最新版本的 WER 客户端程序集，可以从 WER CodePlex 下载，网址如下：

http://wer. codeplex. com/Release/ProjectReleases. aspx? ReleaseId = 12825

从清单 8-2 中可以看到，这是一个简单的命令行程序，提示用户输入以下信息：

- Username。用于与 WER 认证的用户名。
- Password。用户名对应的密码。请注意，在输出时，这个密码会显示在控制台上。

在输入用户名和密码后，程序使用 Login 方法，将输入的信息作为方法的参数。成功登录后，Login 方法将返回一个 Login 实例，表示新创建 WER 会话。这个实例将在随后的 WER 操作中使用。在连接到 WER 时，程序将提示输入进一步的信息：

- Product。产品的名字。
- File。程序文件的名字。
- Event ID。事件 ID。
- Location to store Cabs。与事件 ID 相关的所有 Cab 的存储位置。

在指定了上述信息后，程序将连接到 WER 并尝试查找这个事件。这个过程包括以下步骤：

1）枚举这个企业（与指定的用户名相关的）注册的所有产品并查找指定的产品。

2）枚举与在步骤 1 中找到的产品相关的所有程序文件并查找指定的程序。

3）枚举与在步骤 2 中找到的程序文件相关的所有事件并查找指定的事件。

4）当找到这个事件时，下载所有相关的 Cab 文件。

请注意，在任何需要调用 WER 的操作中同样需要将 Login 实例（即建立起会话）作为参数传递进去。

我们来看一个示例：

```
C:\ADNDBin>08WerConsole.exe
Enter user name: MarioH
Enter password: <password>
Login into WER...
Login succeeded
Enter Product: Advanced .NET Debugging
Enter File: 08SimpleExc.exe
Enter Event ID: 504156229
Enter Location to store CABs: c:\zone\CAB
Event successfully retrieved
Event ID: 504156229
Event Total Hits: 2
Storing CABs...
CABs stored to: c:\zone\CAB
```

请注意，根据下载的 Cab 的数量以及每个 Cab 包含的信息量，整个下载过程可能需要几分钟的时间。如果观察指定 Cab 位置，现在我们可以看到以下文件：

```
C:\ADNDBin>dir /B c:\Zone\cab
504156229-CLR20ManagedCrash-0605004230.cab
504156229-CLR20ManagedCrash-0605004408.cab
504156229-CLR20ManagedCrash-0605004551.cab
504156229-CLR20ManagedCrash-0605004647.cab
504156229-CLR20ManagedCrash-0605004808.cab
504156229-CLR20ManagedCrash-0605004930.cab
504156229-CLR20ManagedCrash-0605005030.cab
504156229-CLR20ManagedCrash-0605005112.cab
504156229-CLR20ManagedCrash-0606022813.cab
504156229-CLR20ManagedCrash-0606025125.cab
```

现在，我们可以解压下载的 Cab 文件，在调试器中加载相关的转储文件，并通过事后调试来找出在用户机器上的问题根源。

对 WER 的程序化访问是一种非常强大的功能，它使得企业开发出的监测程序与在线的错误跟踪系统能很容易地集成起来。人们也能很容易地开发一个服务，定期地查询 WER 服务以下载新的程序故障并保存到一个数据库中，将这个数据库与某个错误跟踪解决方案集成起来，从而使开发人员快速地获得故障通知，并为客户提供快速回应。

8.3　小结

事后调试是软件工程师日常工作的重要组成部分。在把程序交给用户使用后，通常很难对问题进行分析。如果能够对问题做出快速且准确的回应，那么将尽可能地减少用户遇到的麻烦，这对于企业高效管理用户投诉来说是非常重要的。

在本章中，我们介绍了为什么在某些情况下需要进行事后调试，在事后调试中需要用到的调试信息，以及可以通过哪些工具来收集信息。在获得了这些信息后，我们还介绍了如何使用调试器来分析这些调试信息并找出问题的根源。

此外，我们还详细介绍了 Windows 错误报告服务，这个服务能够监视程序在实际中的使用情况，甚至还能访问在程序中出现的每个问题的详细信息，并且提供问题的回应。此外，还讨论了 WER 服务的一些最新增强功能，例如程序化访问 WER 账号，它可以与错误跟踪集成起来从而更高效地分析程序中的错误。

第 9 章　一些功能强大的调试工具

当前，有许多调试工具都可以简化程序错误的分析过程。在本书中，我们已经介绍并使用了多种工具来分析程序中的错误。到目前为止，所介绍的大多数工具都是与非托管调试器相关的（例如 SOS 和 SOSEX 等调试器扩展），此外还有其他一些工具能极大地减少在调试上所花费的时间以及降低错误分析过程的难度。本章我们将介绍其中一些工具，包括：

- PowerDbg；
- SOS 与 Visual Studio 2008 的集成；
- Visual Studio 2008 与. NET 框架源代码级调试；
- Visual Studio 2010；
- CLR 分析器（CLR Profiler）；
- WinDbg 与 CmdTree 命令。

9.1　PowerDbg

PowerDbg 是一种脚本，它使调试人员能够通过 PowerShell 来控制非托管调试器的执行。PowerDbg 将非托管调试器的强大功能封装到 PowerShell 的丰富脚本语言中，这样既可以使用调试器的脚本功能，也可以使用 PowerShell 的脚本功能，因此能形成一些新命令，例如对调试器命令的输出进行后处理，并采用易于阅读的形式来显示结果。PowerDbg 既可以用于实时调试会话，也可以用于转储文件调试。

9.1.1　安装 PowerDbg

在安装 PowerDbg 的最新版本时要求使用 2.0 版本的 PowerShell。在编写本书时，PowerShell 2.0 还处于客户技术预览版本 3 状态（Customer Technology Preview release version 3，CTP3），可以从 Microsoft 网站上免费下载。在安装了 PowerShell 后，可以从以下位置下载 PowerDbg：

http://www.codeplex.com/powerdbg

下载的文件是 ZIP 格式，包含以下文件：

- Microsoft. PowerShell_profile. ps1。这个文件中包含了一些内置命令（cmdlet），用于对非托管调试器的命令输出进行解释，必须放在以下路径：

  %USERPROFILE%\Documents\WindowsPowerShell

- WinDbg. psm1。这个文件中包含了一些的内置命令，用于处理与非托管调试器的通信，必须放在以下路径：

  %WINDIR%\System32\WindowsPowerShell\v1.0\Modules\WinDbg

在使用 PowerDbg 之前，首先需要进行一些配置步骤。第一步就是告诉 PowerShell 允许执行脚本文件。可以从 PowerShell 窗口中运行以下命令来实现：

```
PS C:\Windows\System32> set-executionpolicy Unrestricted
```

上述命令将脚本的执行策略设置为：允许在系统上运行任意脚本。如果脚本是从 Internet 上下载的，那么 PowerShell 将在执行之前给出提示。

接下来，必须通过以下命令来导入 WinDbg. psml 模块：

```
Import-module WinDbg
```

请注意，导入的模块会被持久化，因此这个配置步骤只需执行一次。

接下来是将调试器配置为远程调试。执行这个步骤的原因在于，PowerDbg 的内置命令与非托管调试器之间的通信是通过非托管调试器中的远程功能来实现的。换句话说，要使 PowerDbg 能够工作，必须首先在调试器中设置远程管道（Remote Pipe）。要设置远程管道，可以在调试器中使用以下命令：

```
.server tcp:port=<port_number>
```

其中 port_number 是在通信过程中使用的 TCP 端口号。例如，如果使用 TCP 端口 8888，那么应该执行以下命令：

```
.server tcp:port=8888
```

接下来的步骤是确保在调试会话中设置了正确的符号路径。我们可以使用 WinDbg 中的 . symfix 命令将符号路径设置为 Microsoft 的公有符号服务器。

最后一步是通过 Connect-WinDbg 命令指示 PowerDbg 连接到调试会话。Connect – WinDbg 的命令参数是一个远程连接字符串，这个字符串是在 Windbg 中设置远程连接时形成的，字符串的语法如下：

```
tcp:Port=<port_number>,Server=<server_name>
```

其中 port_number 是在之前创建远程管道的 TCP 端口号，server_name 是运行 Windbg 会话的服务器名字。例如，假定 TCP 端口号为 8888 并且服务器名字为 MARIOH-LAPTOP，那么可以通过以下命令来连接：

```
Connect-WinDbg "tcp:Port=8888,Server=MARIOH-LAPTOP"
```

当前，每次只能支持一个连接，如果希望重新连接到另一个远程会话，那么必须首先执行 Disconnect-WinDbg 命令。

PowerDbg 是否仅支持 Windbg

并非如此。在 Windows 调试工具集中的所有调试器都共享同一个调试器引擎，因此 Windbg、cdb 或者 ntsd 等都可以用于创建远程连接。

前面给出的步骤就是开始使用 PowerDbg 之前的所有准备工作（启动调试器、创建远程连接、设置符号，并使用 Connect-WinDbg 命令来连接）。接下来，我们将介绍 PowerDbg 中的一些内置命令。

9.1.2　Analyze-PowerDbgThreads

Analyze-PowerDbgThreads 命令枚举进程中所有的线程，并显示它们的状态。在表 9-1 中给出了可用的状态。

表 9-1　内置命令 Analyze-PowerDbgThreads 支持的可用状态

状　态	描　述
UNKNOWN_SYMBOL	线程的状态未知
WAITING_FOR_CRITICAL_SECTION	线程正在等待某个临界区
DOING_IO	线程正在执行文件操作
WAITING	线程正在等待某个同步原语
GC_THREAD	线程正在执行垃圾收集操作
WAIT_UNTIL_GC_COMPLETE	线程正在等待垃圾收集操作执行完成
SUSPEND_FOR_GC	线程被垃圾收集器挂起了
WAIT_FOR_FINALIZE	线程正在等待某些对象被终结
TRYING_MANAGED_LOCK	线程正在尝试获取一个托管锁
DATA_FROM_WINSOCK	线程正在等待数据从 Windows 套接字（Socket）层传递过来

内置命令找出线程状态的过程是，发送一个 k 命令（显示栈回溯）到调试器会话，并且将栈上的符号映射到相应的状态。我们可以通过% USERPROFILE% \Documents \WindowsPowerShell\Microsoft. PowerShell_profile. psl 中的 Classify-PowerDbgThreads 函数来修改这个内置命令，并增加自定义的符号和状态。

以下是在 05Heap. exe 上执行这个命令时的输出：

```
Threads sorted by User Time...

Thread Number   User Time        Kernel Time          Activity
   0            0:00:00.031      0:00:00.078      Thread working and
 doing unknown activity.
   4            0:00:00.000      0:00:00.000      Thread in wait state.
   3            0:00:00.000      0:00:00.000      Thread working and doing unknown
activity.
```

```
   2              0:00:00.000     0:00:00.000     Thread waiting for the Finalizer
event. The Finalizer thread might be b
locked.
   1              0:00:00.000     0:00:00.000     Thread is the CLR Debugger Thread.
```

从输出中可以看到，除了每个线程的状态外，还显示了用户态时间和内核态时间，从而能很容易地判断某个线程是否是失控的或者被阻塞的。

9.1.3　Send-PowerDbgCommand

一些命令会执行完整的步骤来显示命令的执行结果，例如 Analyze-PowerDbgThreads。然而，在某些情况中，你可能只是希望发送某个命令到调试器，然后自行对命令的执行输出进行事后处理。在这些情况中，可以使用 Send-PowerDbgCommand。Send-PowerDbgCommand 的参数是一个字符串，其中包含了想要在调试器中执行的命令。例如，如果想要执行 kb 200 命令，可以键入以下命令：

```
PS C:\Windows\System32> Send-PowerDbgCommand "kb 200"
```

执行上述命令会生成一个文件 POWERDBG-OUTPUT. LOG，其中包含了调试器的输出结果。然后，我们可以通过这个文件对输出结果进行后处理，并以合适的格式显示数据。由于文件解析的过程可能非常麻烦，因此 PowerDbg 还包含了另一组内置命令，形如 Parse-PowerDbg*，其中*可以由相应的调试器命令替换。这些命令能分析由 Send-PowerDbgCommand 生成的文件内容，并将文件的内容格式化并保存到一个新文件（POWERDBG-PARSED. LOG）中。这个新文件可以与内置命令 Convert-PowerDbgCSVToHashTable 一起使用，它将把文件内容转换为一张散列表，从而更易于实现脚本过程。例如，非托管调试器命令 ~* kpn 1000（显示进程中所有线程的栈回溯）可以通过 PowerDbg 按照以下形式执行：

```
PS C:\Windows\System32> Send-PowerDbgCommand "~* kpn 1000"
PS C:\Windows\System32> Parse-PowerDbgK
PS C:\Windows\System32> $ht = @{}
PS C:\Windows\System32> $ht = Convert-PowerDbgCSVToHashTable
```

第一个命令将 ~*kpn 1000 发送到调试器，调试器会执行这个命令并将结果保存在 POWERDBG-OUTPUT. LOG 中。然后，Parse-PowerDbgK 会对文件中保存的执行结果进行解析，并且创建一个新文件 POWERDBG-PARSED. LOG，接下来 Convert-PowerDbgCSVToHashTable 处理这个文件并返回一个散列表以及被解析文件的内容。在获得这个散列表后，我们可以通过它以及 write-host 命令来显示不同的栈回溯：

```
PS C:\Windows\System32> write-host $ht["0"].Replace($global:g_frameDelimiter, "'n")
ChildEBP RetAddr
00 0017f060 778d8d94 ntdll!KiFastSystemCallRet
01 0017f064 778e9522 ntdll!NtRequestWaitReplyPort+0xc
02 0017f084 77507e05 ntdll!CsrClientCallServer+0xc2
03 0017f170 77507f35 KERNEL32!GetConsoleInput+0xd2
04 0017f190 001ca61c KERNEL32!ReadConsoleInputA+0x1a
```

```
Frame IP not in any known module. Following frames may be wrong.
05 0017f218 793e8f28 0x1ca61c
06 0017f280 793e8e33 mscorlib_ni+0x328f28
07 0017f2d0 79e7c6cc mscorlib_ni+0x328e33
08 0017f350 79e7c8e1 mscorwks!CallDescrWorkerWithHandler+0xa3
09 0017f490 79e7c783 mscorwks!MethodDesc::CallDescr+0x19c
0a 0017f4ac 79e7c90d mscorwks!MethodDesc::CallTargetWorker+0x1f
0b 0017f4c0 79eefb9e mscorwks!MethodDescCallSite::Call+0x18
0c 0017f624 79eef830 mscorwks!ClassLoader::RunMain+0x263
0d 0017f88c 79ef01da mscorwks!Assembly::ExecuteMainMethod+0xa6
0e 0017fd5c 79fb9793 mscorwks!SystemDomain::ExecuteMainMethod+0x43f
0f 0017fdac 79fb96df mscorwks!ExecuteEXE+0x59
10 0017fdf4 7900b1b3 mscorwks!_CorExeMain+0x15c
11 0017fe04 774a4911 mscoree!_CorExeMain+0x2c
12 0017fe10 778be4b6 KERNEL32!BaseThreadInitThunk+0xe
13 0017fe50 778be489 ntdll!__RtlUserThreadStart+0x23
14 0017fe68 00000000 ntdll!_RtlUserThreadStart+0x1b

PS C:\Windows\System32> write-host $ht["1"].Replace($global:g_frameDelimiter, "`n")
ChildEBP RetAddr
00 016ff95c 778d9244 ntdll!KiFastSystemCallRet
01 016ff960 774ac3e4 ntdll!ZwWaitForMultipleObjects+0xc
02 016ff9fc 774ac64e KERNEL32!WaitForMultipleObjectsEx+0x11d
03 016ffa18 79f4e8d8 KERNEL32!WaitForMultipleObjects+0x18
04 016ffa78 79f4e831 mscorwks!DebuggerRCThread::MainLoop+0xe9
05 016ffaa8 79f4e765 mscorwks!DebuggerRCThread::ThreadProc+0xe5
06 016ffad8 774a4911 mscorwks!DebuggerRCThread::ThreadProcStatic+0x9c
07 016ffae4 778be4b6 KERNEL32!BaseThreadInitThunk+0xe
08 016ffb24 778be489 ntdll!__RtlUserThreadStart+0x23
09 016ffb3c 00000000 ntdll!_RtlUserThreadStart+0x1b
```

表 9-2 详细列出了各种 Parse-PowerDbg* 内置命令。

表 9-2　形如 Parse-PowerDbg* 的内置命令

内置命令	描　述
Parse-PowerDbgDT	解析 dt 命令的输出
Parse-PowerDbgNAME2EE	解析 SOS name2ee 命令的输出
Parse-PowerDbgDUMPMD	解析 SOS dumpmd 命令的输出
Parse-PowerDbgDUMPMODULE	解析 SOS dumpmodule 命令的输出
Parse-PowerDbgLMI	解析 lmi 命令的输出
Parse-PowerDbgVERTARGET	解析 vertarget 命令的输出
Parse-PowerDbgRUNAWAY	解析 runaway 命令的输出
Parse-PowerDbgK	解析 k 命令的输出
Parse-PowerDbgSymbolsFromK	从 k 命令的输出中抽取符号信息
Parse-PowerDbgLM1M	解析 lm1m 命令的输出
Parse-PowerDbgPRINTEXCEPTION	提取异常信息
Parse-PowerDbgDD-L1	解析 dd 命令的输出
Parse-PowerDbgGCHANDLELEAKS	解析 SOS gchandleleaks 命令的输出
Parse-PowerDbgDUMPOBJ	解析 SOS dumpobj 命令的输出

9.1.4 扩展 PowerDbg 的功能

除了在 PowerDbg 工具中包含的内置命令之外，还可以使用现有内置命令的源代码来进行扩展。在本章的这部分内容中，我们将看到如何扩展 PowerDbg 的功能：通过编写一个内置命令来获得线程执行块（Thread Execution Block，TEB）。我们要做的第一件事情就是找出哪个调试器命令能够获得我们需要的结果。teb 命令可以用来获取线程执行块，这能显示 teb 中包含的所有数据。以下是 teb 命令的示例输出：

```
0:004> !teb
TEB at 7ffda000
    ExceptionList:        04a3f9a4
    StackBase:            04a40000
    StackLimit:           04a3c000
    SubSystemTib:         00000000
    FiberData:            00001e00
    ArbitraryUserPointer: 00000000
    Self:                 7ffda000
    EnvironmentPointer:   00000000
    ClientId:             0000196c . 000018ac
    RpcHandle:            00000000
    Tls Storage:          00000000
    PEB Address:          7ffdb000
    LastErrorValue:       0
    LastStatusValue:      0
    Count Owned Locks:    0
    HardErrorMode:        0
```

我们可以使用 Send-PowerDbg 命令把 teb 发送到调试器，并且在 POWERDBG-OUTPUT. LOG 文件中获得 teb 的执行结果。接下来的步骤是实现 Parse-PowerDbgHandle 命令，这个命令将读取 POWERDBG-OUTPUT. LOG 中的结果，并将它们按照 Convert-PowerDbgCSVToHashTable 的要求进行格式化，再把结果保存在 POWERDBG - PARSED. LOG 中。然后，可以使用 Convert-PowerDbgCSVToHashTable 将数据送入散列表中，并且通过其他的脚本来处理。其中，大部分工作都是用来实现 Parse-PowerDbgHandle 命令的，如清单 9-1 所示。

清单 9-1 Parse-PowerDbgHandle 内置命令

```
function Parse-PowerDbgTEB()
{
    set-psdebug -strict
    $ErrorActionPreference = "stop"
    trap {"Error message: $_"}

    # Extract output removing commands.
    $builder = New-Object System.Text.StringBuilder
```

```
    # Title for the CSV fields.
    $builder = $builder.AppendLine("key,value")
    foreach($line in $(get-content $global:g_fileCommandOutput))
    {
      if($line.Contains("TEB at"))
      {
      }
      else
      {
        $fields=$line.Split(":");
        if([String]::IsNullOrEmpty($fields[0]))
        {
        }
        if([String]::IsNullOrEmpty($fields[1]))
        {
        }
        else
        {
          $f1=$fields[0].Trim();
          $f2=$fields[1].Trim();
          $builder = $builder.AppendLine($f1 + $global:g_CSVDelimiter + $f2)
        }
      }
    }

    # Send output to our default file.
    out-file -filepath $global:g_fileParsedOutput -inputobject "$builder"
}
```

函数将遍历原始数据文件中的每一行，提取出每个域的名字和值，并将每个键－值二元组添加到一个字符串后面。在提取完所有域之后，会把这个字符串写入到被解析的输出文件中。随后，可以通过 Convert-PowerDbgCSVtoHashTable 将解析后的输出文件转换为一个散列表，如下所示：

```
PS C:\> $h={}
PS C:\> $h=Convert-PowerDbgCSVtoHashTable
```

要查看散列表中的值，需要指定这个散列表，PowerShell 会遍历所有的二元组并显示它们：

```
PS C:\> $h

Name                          Value
----                          -----
LastErrorValue                0
ExceptionList                 04a3f9a4
FiberData                     00001e00
StackLimit                    04a3c000
StackBase                     04a40000
Count Owned Locks             0
```

```
RpcHandle                    00000000
Self                         7ffda000
LastStatusValue              0
HardErrorMode                0
SubSystemTib                 00000000
ClientId                     0000196c . 000018ac
PEB Address                  7ffdb000
ArbitraryUserPointer         00000000
Tls Storage                  00000000
EnvironmentPointer           00000000
```

PowerDbg 库的可扩展性极高，能很容易地创建一些内置命令将调试命令输出的分析过程自动化，从而减少在调试特定类别错误上所花的时间。

9.2 Visual Studio

Visual Studio 是一种很常用的集成化开发环境（Integrated Developer Environment，IDE）。基于它的强大功能以及简单易用性，使其成为当前.NET 开发人员的首选开发环境。在 Visual Studio 中集成了一个非常完备的调试器，可以对各种不同的问题进行分析，例如进行源代码级调试、脚本调试以及 SQL 调试等。然而，Visual Studio 在产品级调试方面有所欠缺，因此我们要么需要对转储文件进行分析，要么需要访问 CLR 的内部细节（例如 SOS 命令）。Visual Studio 团队意识到了这个问题，并且已经解决了后一部分问题，即让 Visual Studio 能够与 SOS 调试器扩展兼容。在本章的这部分内容中，我们将看到如何配置 Visual Studio 2008 来使用 SOS 调试器扩展，以及如何配置 Visual Studio 来访问.NET 框架的源代码。最后，我们会简要介绍在 Visual Studio 2010 中将要包含的一些新的调试功能。

9.2.1 SOS 的集成

为了说明如何将 SOS 调试器扩展与 Visual Studio 集成起来，我们首先创建一个简单的 C# 项目（命令行程序就足够了）。然后，在代码的第一行设置一个断点，并且按下 F5 键开始调试。在图 9-1 中给出了在触发断点后的调试器。

接下来在 SOS 集成过程中打开即时窗口（Immediate Window）。即时窗口用于在调试会话中执行各种不同的命令（例如为变量赋值，表达式求值等）。展示如何打开即时窗口。

在启用即时窗口后，可以开始键入调试命令。其中一个可用的命令就是 load，它可以加载 SOS 调试器扩展。在图 9-3 中说明了执行 load 命令的结果。

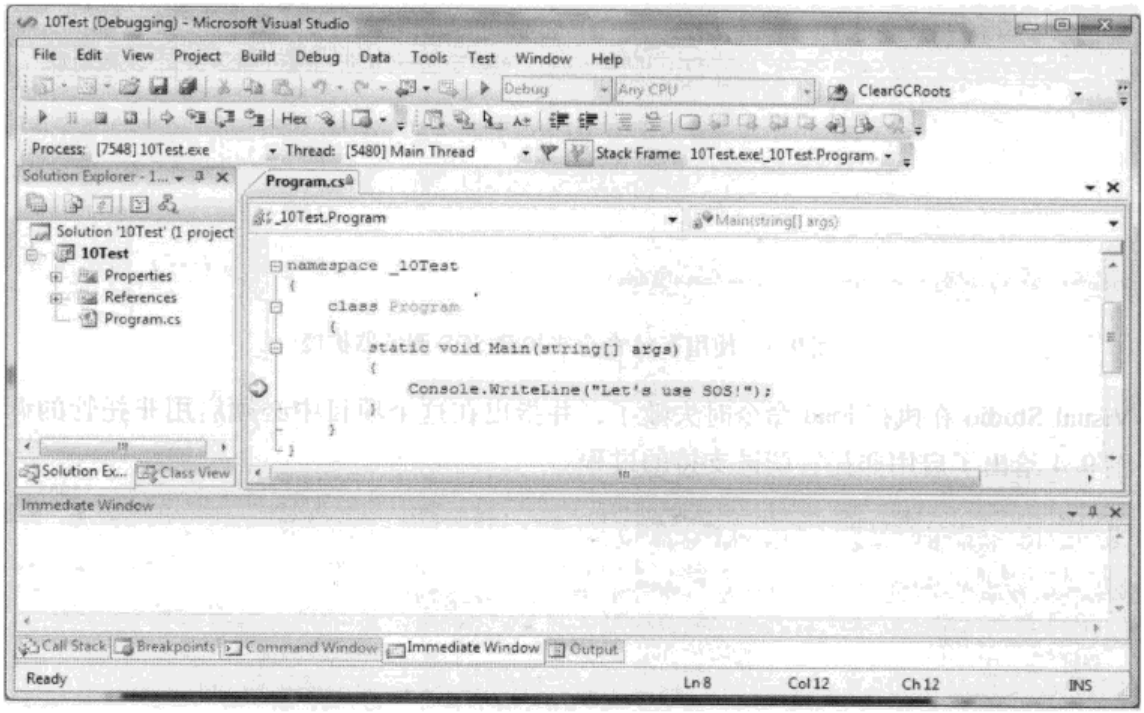


图 9-1　Visual Studio 中的断点触发

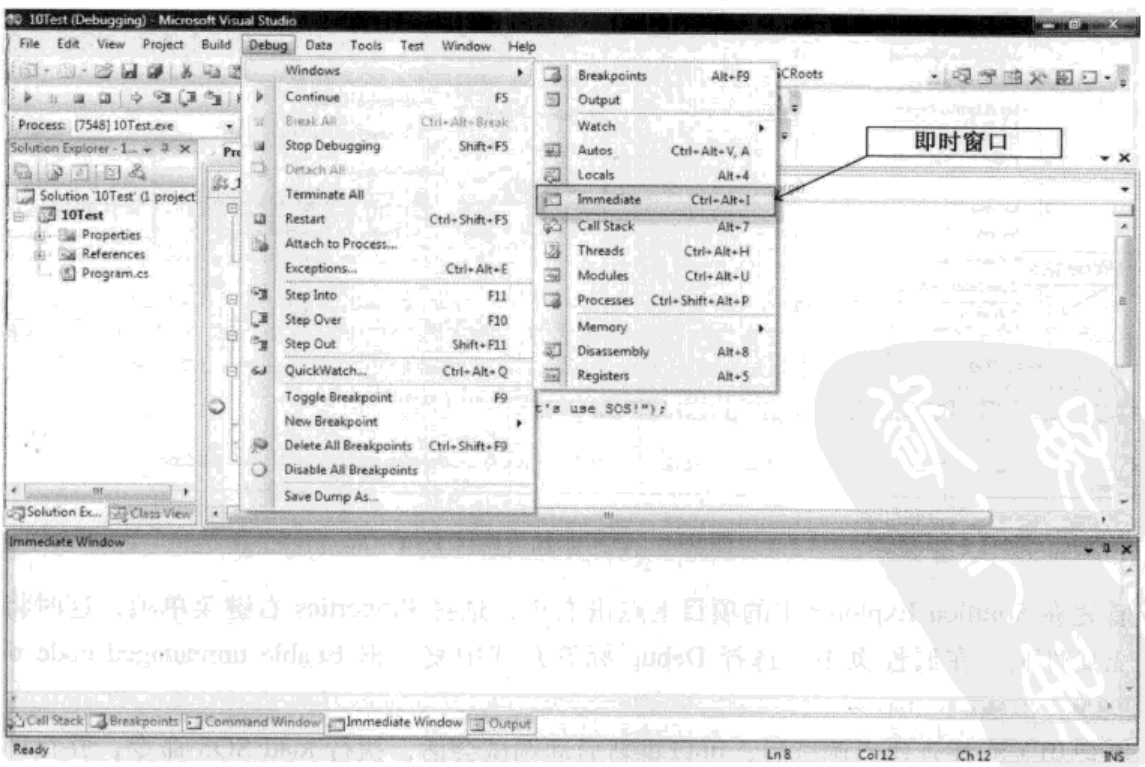


图 9-2　启用即时窗口

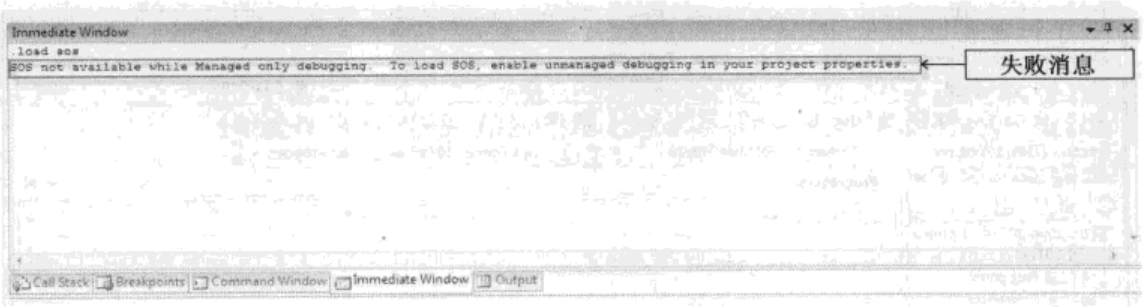


图 9-3　使用 load 命令来加载 SOS 调试器扩展

Visual Studio 在执行 load 命令时失败了，并指出在这个项目中必须启用非托管的调试支持。图 9-4 给出了启用非托管调试支持的过程。

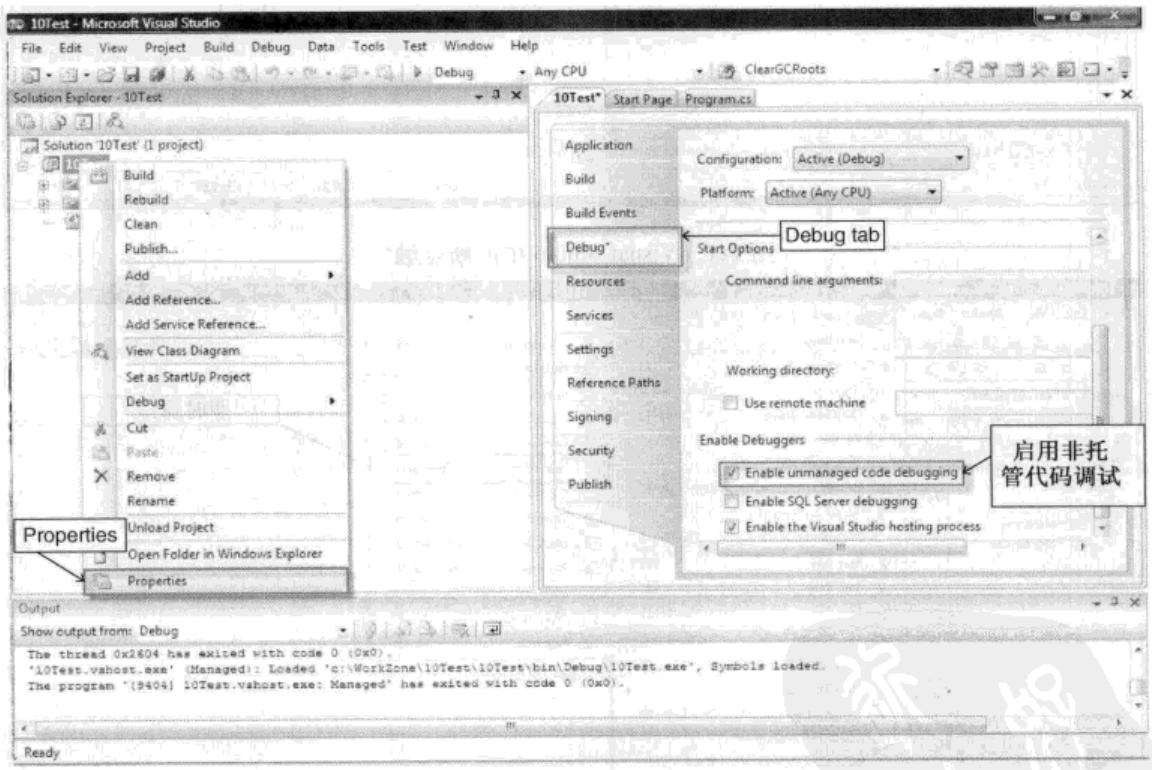


图 9-4　启用非托管调试支持

首先在 Solution Explorer 中的项目上点击右键，选择 Properties 右键菜单项，这时将在右侧弹出属性页。在属性页中，选择 Debug 标签并选中复选框 Enable unmanaged code debugging。

在启用了非托管代码调试后，可以重新启动调试会话，执行 load SOS 命令，并在即时窗口中使用 SOS 调试器扩展命令，如图 9-5 所示。

图 9-5　使用 SOS 调试器扩展

在图 9-5 中，我们使用了两个 SOS 命令（EEVersion 和 DumpHeap）来说明即时窗口中的输出信息。

将 Visual Studio 的强大功能和易用性与 SOS 对 CLR 的解析结合起来，就可以应对各种复杂的调试情况。

9.2.2　.NET 框架源代码级调试

通常，任何.NET 程序都会用到.NET 框架中定义的一组不同类型。这些类型既包括简单的数据类型，也包括复杂的 Web 服务绑定类型，它们能有效地屏蔽这些技术的底层复杂性。与任何抽象一样，如果要调试某个有问题的应用程序，那么调试过程也会因此而变得复杂。在分析程序为什么会失败时，如果能够直接对尝试对源代码进行分析，而不是使用逆向工程（即反汇编），那么调试过程会变得更为容易。Microsoft 意识到了这种需求，因此公开了.NET 框架的部分源代码。这些公开的源代码可以集成到 Visual Studio 中，从而使开发人员可以对已发布的.NET 框架源代码进行源代码级调试。接下来，我们将看到如何配置 Visual Studio 来实现无缝的.NET 框架源代码级调试。

Visual Studio 热补丁

要在 Visual Studio 中使用集成的.NET 框架源代码，必须首先安装热补丁（Hot Fix），下载地址为 https://connect.microsoft.com/VisualStudio/Downloads/DownloadDetails.aspx?DownloadID=10443&wa=wsignin1.0。

要实现源代码级调试，我们必须告诉 Visual Studio 调试器到什么位置去查找被调试模块的源代码。它从何处获得这个信息？答案是从相应的符号文件中获得这些信息，并且这些符

号文件是公开的。正如 Microsoft 将其他产品的公有符号在 Microsoft 符号服务器上公开，对于.NET 框架也是如此。然而，其中最大的差别在于在公开的符号文件中包含了多少信息。在公有符号服务器上的符号都被“裁剪过”，这意味着在服务文件中删除了某些特定信息（包括源代码级信息）。相反，.NET 框架的符号文件包含了完整的调试信息，包括源代码级的访问，然后 Visual Studio 会通过这些信息找出在 Internet 上公开的源代码。第一步是告诉 Visual Studio，在调试会话中将使用源代码服务器来访问源代码。依次选择 Tools -> Options -> Dubugging -> General 等菜单项，将弹出如图 9-6 所示的对话框。

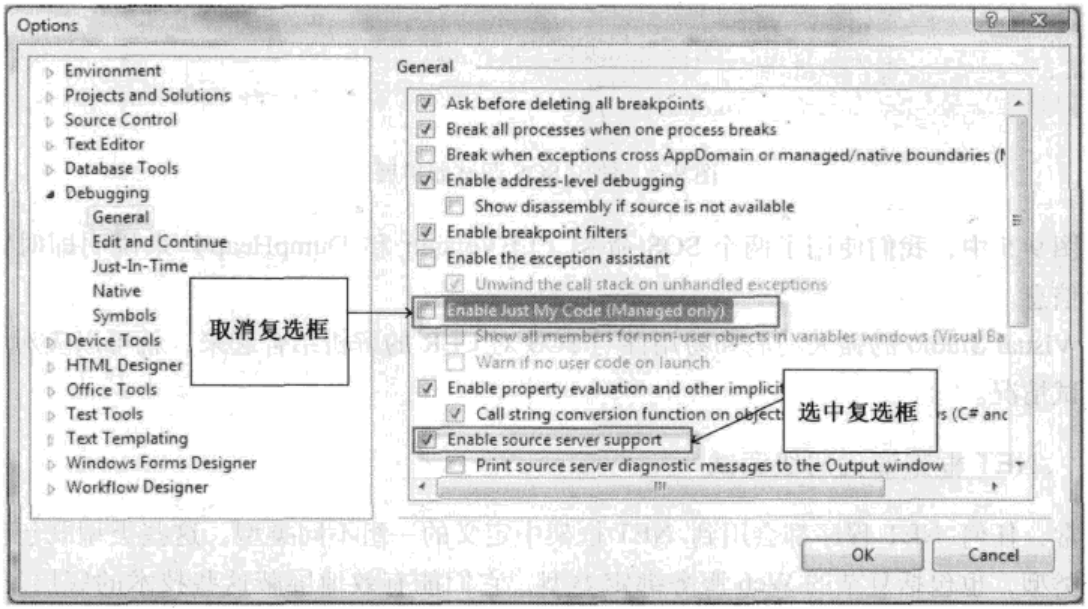


图 9-6 启用源代码级的调试

在 Debugging 的 General 属性页中，取消复选框 Enable Just My Code（Managed only）并选中复选框 Enable source server support。

接下来，跳转到 Symbols 页，指定想要使用的符号服务器的位置，如图 9-7 所示。

在这些符号文件位置中，我们添加了以下 URL，在这个 URL 中包含了.NET 框架符号：http://referencesource.microsoft.com/symbols。此外，我们告诉 Visual Studio 将所有下载后的符号文件缓存到文件夹 C:\Zone 中。这种缓存机制避免了每次启动一个调试会话时下载相同的符号文件，因此节约了大量的时间。最后，设置 Visual Studi 仅在符号被手动加载时才使用这些位置，也就是 Visual Studio 在支持.NET 框架的源代码级调试时需要的配置文件。我们来看一个简单的应用示例。创建一个 C#控制台程序，在 Main 方法中写入下面这行代码：

```
Console.WriteLine("Let's use SOS!");
```

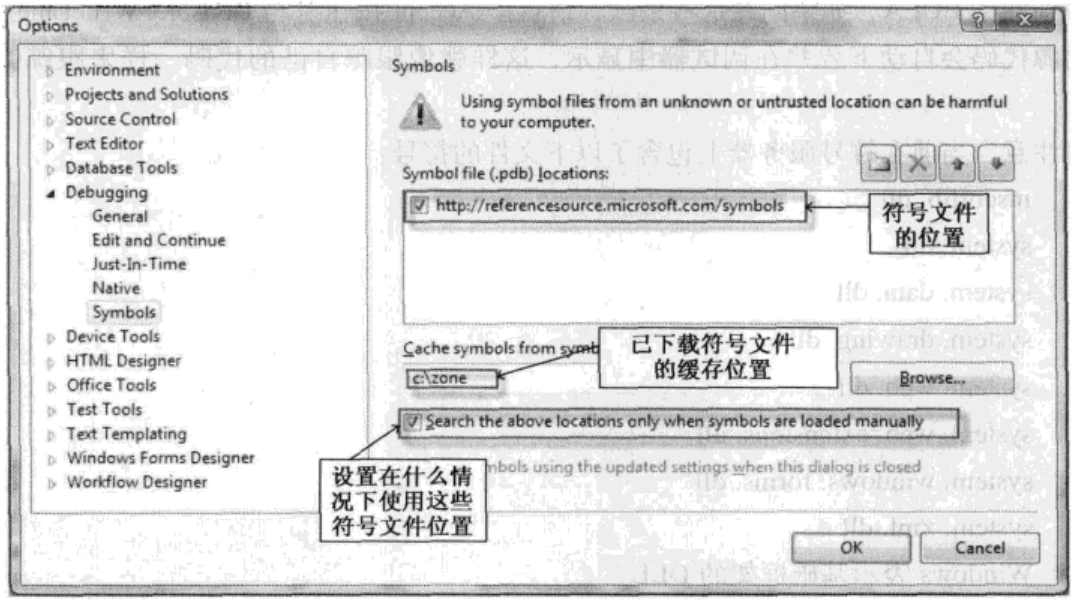


图 9-7　设置符号服务器

构建这个项目，并通过按 F9 键在上面的代码行上设置一个断点。接下来，按 F5 键调试这个程序，在触发断点后，使用 CTRL-ALT-U 弹出 Modules 窗口。找到 mscorlib. dll 模块，点击右键并选中 Load Symbols。这时会下载 mscorlib. dll 的符号来实现源代码级的调试。当下载完符号后，在 Symbol Status 列将显示 Symbols Loaded。现在，按下 F11 键进入到 Console. WriteLine 代码。你将注意到的第一件事情是，显示了一个终端用户许可协议（End User License Agreement ，EULA），如图 9-8 所示。

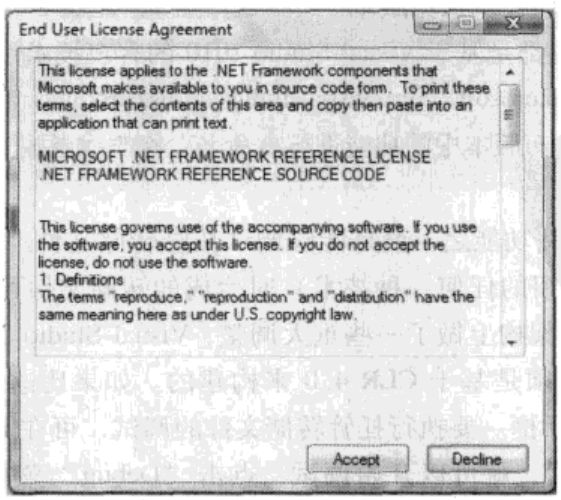


图 9-8　. NET 框架源代码的 EULA

仔细阅读 EULA，如果同意这些条款，请点击 Accept 按钮。与 Console. WriteLine 方法相对应的源代码会自动下载并在调试器中显示，这样就像跟踪自己的代码一样来跟踪 CLR 源代码。

请注意，当前在符号服务器上包含了以下文件的符号：

- mscorlib. dll
- system. dll
- system. data. dll
- system. drawing. dll
- system. web. dll
- system. web. extensions. dll
- system. windows. forms. dll
- system. xml. dll
- Windows 表示基础框架的 DLL
- Microsoft. visualbasic. dll

Microsoft 确保发布新符号和源代码的过程是完全自动的，我们今后还将看到越来越多的符号和源代码。

9. 2. 3 Visual Studio 2010

随着 Visual Studio 2010 发布日期的临近[⊖]，调试人员很高兴地了解到，Visual Studio 团队投入了大量的精力来弥补之前版本中存在的问题。

Visual Studio 2010 的客户技术预览版本

这里给出的所有信息都是基于 Visual Studio 2010 的客户技术预览版本（CTP），可以从 Microsoft（http://www. microsoft. com/visualstudio/en-us/products/2010/default. mspx）下载。Microsoft 保留了对最终发布版本中的功能进行重命名、修改或者删除的权利。

Visual Studio 2010 的新功能之一就是能够调试托管转储文件。现在，在抓取问题进程快照（可以使用第 8 章中介绍的任何一种技术）时生成的转储文件可以被加载到 Visual Studio 2010 中。由于在核心调试架构上做了一些重大调整，Visual Studio 2010 在调试转储文件时要求生成转储文件的程序必须是基于 CLR 4. 0 来构建的。如果比这个版本要早，那么 Visual Studio 将显示一个错误对话框。要执行托管转储文件的调试，可在 File 菜单中选择 Open Project 菜单项。在创建项目后，就可以开始调试，点击“Debug”菜单，并选择“Start Debug-

⊖ 已于 2010 年 4 月 12 日正式发布。——译者注

ging”菜单项。然后，Visual Studio 将给出导致调试器中断的最近事件，你也可以开始分析进程的状态。在 Visual Studio 2010 之前，所有的调用栈显示的是相应的非托管代码，而现在它们显示的是托管代码栈帧。其他功能和以前一样，可以在栈帧中双击进入到相应的托管源代码。其他一些功能，例如列出某个栈帧的局部变量，在托管代码上同样可以执行。但是存在一些限制，例如，如果使用的是经过优化后的二进制文件，那么给出的局部变量有可能不准确。在图 9-9 中给出了一个示例，对 05Heap. exe 生成的转储文件进行调试。

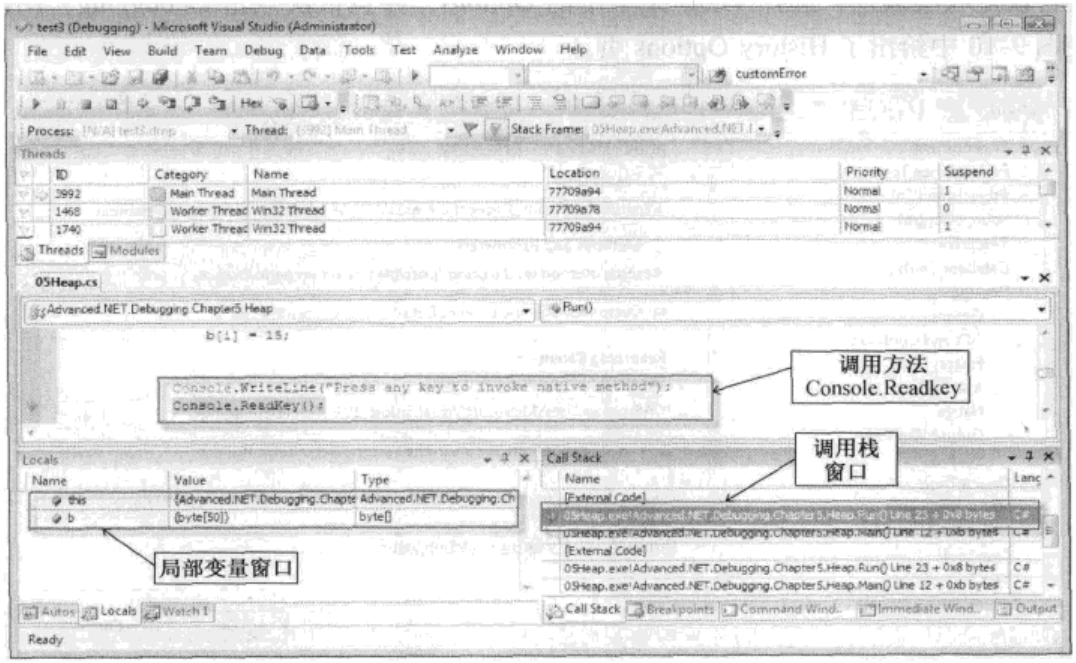


图 9-9　Visual Studio 2010 托管代码转储调试示例

在图 9-9 的示例中给出了当调用 Console. ReadKey 方法并中断进入到调试器中时调试器的内容。在右侧的 Call Stack 窗口给出了整个托管代码调用栈，而在左侧的 Locals 窗口给出了所选栈帧的局部变量。

要基于. NET 4. 0 来构建一个程序，可以使用 MSBUILD 构建系统，它属于. NET 4. 0 的一部分。请注意，必须使用/toolsversion：4. 0 开关。在默认情况下，MSBUILD 构建系统是基于 2. 0 版本的。

Microsoft 是否跳过了一个 CLR 版本

在 CLR 和. NET 框架的主修订号和次修订号之间，版本号变得有些混乱。CLR 的最后一次主要更新是版本 2. 0，而. NET 框架则更新到版本 3. 5。为了减少困惑，Microsoft 决定在 4. 0 版本中重新分配 CLR 和. NET 框架的版本。现在，CLR 和. NET 框架的版本都是 4. 0。

Visual Studio 的另一个新调试功能就是历史调试（Historical Debugging）。历史调试能够极大地减少开发人员在假设和验证程序如何进入到当前状态时所花费的时间。通过历史调试功能，开发人员不需要*手动地*回溯程序的执行历史，而是可以记录在程序执行过程中发生的重要事件，并且能够在调试会话中回溯这些事件。由于在收集程序执行的历史数据时需要付出很高的开销，并且会对性能产生负面影响，因此 Visual Studio 在默认情况下会使用相对较低的收集级别（Collection Level）。在默认情况下，只有重要的调试事件才会被记录。我们也可以自行设定收集级别，选择 Tools 菜单中的 Options，并且依次点击 Debugging，History 标签。在图 9-10 中给出了 History Options 页。

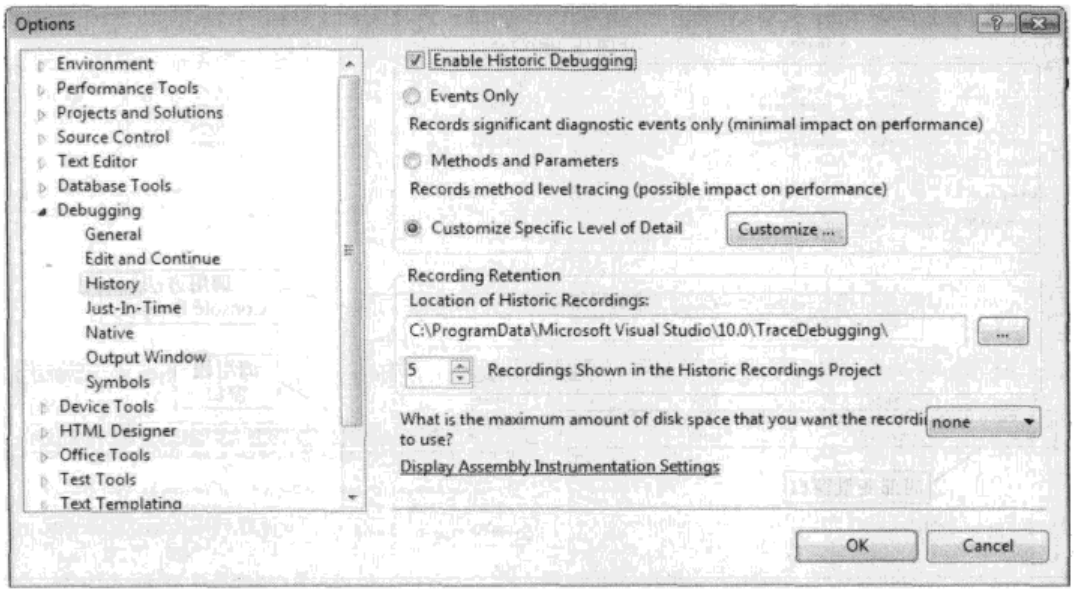


图 9-10 History 调试选项页

“Enable Historic Debugging”表示想要启用的调试级别。默认会选中 Events Only 单选按钮，并将收集级别设置为最低。单选按钮“Methods and Parameters”表示收集方法信息和参数信息，而“Customize Specific Level of Detail”则可以定义自己的收集级别。此外，还可以指定记录所在的位置以及可使用的最大磁盘空间，从而避免耗尽磁盘空间。我们通过一个简单的模拟示例来了解历史调试的工作原理。首先，确保将记录级别设置为 Methods and Parameters，然后从以下位置加载 History 项目：C:\ADNDSRC\Chapter9\History。这个程序非常简单，它只是将两个数值相加起来。按 F5 键开始调试这个程序，并在出现提示信息时输入 12 和 -6。很快可以看到抛出一个除零异常，并且程序执行停止在调试器中。启动 Debug History 窗口（依次选择 Debug Windows，Show Debug History Window）得到历史信息，如图 9-11 所示。

在 Debug History 窗口中包含了 3 个主要面板。最顶层的面板给出了导致故障的栈帧。在这里，线程入口调用了 Main 方法，而 Main 方法接着又调用了 Divide 方法。中间的面板给出了在所选栈帧中发生的诊断事件，以及其他被调用的函数。在这里唯一记录的诊断事件就是调用了 Divide 方法。最底层的面板给出了被调用的函数（Divide）及其相关的参数。在这里，我们可以看到键入的数值正在向函数传递。由于最底层的栈帧表示 Divide 方法，我们现在可以分析在这个方法中使用的局部变量（通过使用调试器的 Locals 窗口），并且可以看到 number1 和 number2 等变量分别被赋值为 12 和 0。由于第二个数值应该为 -6，我们可以很快地（通过代码审查）发现，由于任何负数值都被转换为 0，因此导致了一个除零异常。虽然这是一个非常简单的示例，但我们可以很容易看到这个功能的强大之处，它能够回溯并分析程序是如何进入到当前状态的。

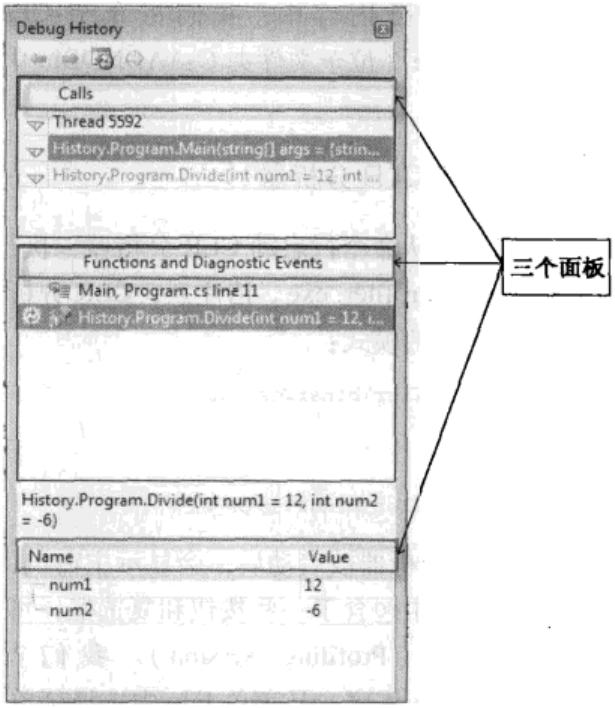


图 9-11　Debug History 窗口

在 Visual Studio 的早期版本中，只能在 x86 平台上执行混合模式的调试（托管的和非托管的）。随着 64 位计算已经变得越来越普遍，Visual Studio 团队增加了对 x64 混合模式调试的支持，这就使得在这种架构上的调试工作更为简单。

最后，CLR 修改了对某些特定异常的处理方式（虽然这本质上算不上某种调试功能）。具体来说，那些无法恢复的异常将不再被转换为 CLR 异常，从而避免这些异常在 catch - all 语句中被错误地捕获。这种情况的一个很好的示例就是访问违例异常。通常，捕获访问异常违例只会推迟将要发生的问题，因此当发生这种异常时立即停止程序执行是一种明智的做法。由于现在不再需要转换并捕获这种异常，因此开发人员能够更好地在测试阶段发现错误并进行修复。至此，我们就结束了对 Visual Studio 的简单介绍。Visual Studio 团队投入了大量的工作来增强调试功能，使得开发人员能更容易地找出程序中错误的根源。

9.3　CLR 分析器

在第 5 章，我们详细介绍了托管堆管理器和垃圾收集器。此外，我们还介绍了在跟踪内存相关问题时可以使用的一些工具。其中有一个工具没有介绍，即 CLR 分析器。CLR 分析器是一种功能非常强大的工具，它能够分析托管堆的行为，并且以各种不同的格式显示出结果。

在本章的这部分内容中，我们将介绍 CLR 分析器的基础知识，使用的示例程序是 05Fragment. exe（位于文件夹 C:\ADNDBin 中）。请参考第 1 章了解 CLR 分析器的下载和安装方法。

9.3.1　运行 CLR 分析器

我们可以从命令行启动 CLR 分析器：执行安装文件夹（在我的情况中是 C：\ CLRProfiler）中 CLRProfiler. exe。请注意，CLR 分析器同时支持 x86 和 x64 两种模式，因此必须确保启动了正确的模式：

```
C:\CLRProfiler\binaries\x86
```

或者

```
C:\CLRProfiler\binaries\x64
```

在 CLR 分析器被启动后，会显示主窗口，如图 9-12 所示。

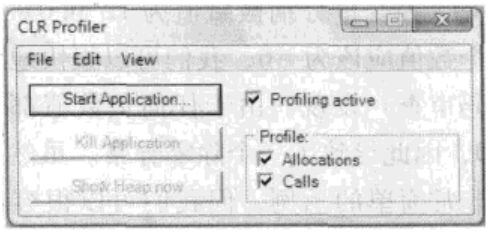


图 9-12　CLR 分析器的主窗口

在主窗口中包含了一些按钮和复选框，可以用于控制分析会话（Profiling Session）。我们首先指定 05Fragment. exe 程序。从菜单 File 中选择菜单项 Set Parameters，会弹出另一个窗口，用于指定与目标程序一起使用的命令行。05Fragment. exe 带有两个命令行参数，可以在编辑框 Enter Command Line 中指定它们。在这次运行中，我们选择的两个命令行参数分别为 50000（分配的大小）和 500（最大的大小）。点击 OK 按钮，再次回到主窗口。现在，点击 Start Application 按钮，会弹出一个对话框，选择要分析的程序。打开文件夹 C:\ADNDBin，并选择程序 05Fragment. exe。现在，程序开始运行，并且启用了两个其他的按钮（Kill Application 和 Show Heap now）。此时，你可以等待程序运行结束（在这种情况下会自动显示 Summary 视图），或者程序显示一个用户提示并且等待直到用户按下任意键。现在，你可以使用 View 菜单来选择任意一个视图，以便观察程序运行的详细信息。请注意，某些视图可能直到程序完成执行之后才能显示。此时，进入到 CLR 分析器启动的命令行，并且按下任意键直到程序执行完成。在接下来的内容中，我们将看到一些不同的视图，并介绍如何正确地解释给出的信息。

CLR 分析器将分析数据保存在什么位置

CLR 分析器将所有的分析数据保存在一个日志文件中，位于文件夹 C：\ Windows \ Temp 下。当分析会话开始时，通过查看执行 CLR 分析器的命令行可以找到这个日志文件名。例如，从 CLR 分析器中启动 05Fragment. exe 会在命令行上给出以下信息：

```
Log file name transmitted from UI is: pipe_78000.log
```

请注意，在收集分析数据的过程中可能会产生大量的日志文件，这会降低程序的执行速度。因此，在性能测试期间应该避免使用 CLR 分析器。

9.3.2　Summary 视图

在程序执行完成后，CLR 分析器会自动显示分析会话的 Summary 视图，如图 9-13 所示。

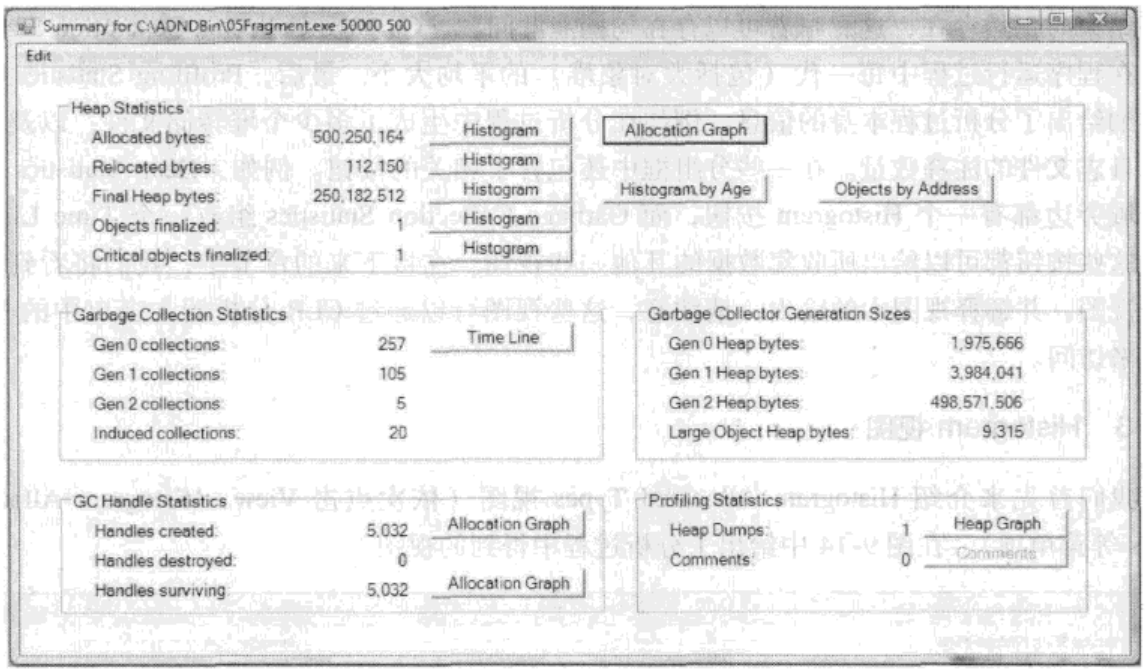


图 9-13　CLR 分析器的 Summary 视图

Summary 视图中包含了 5 组主要的信息。第一组 Heap Statistics 给出了在分析会话期间堆的总体信息。

- Allocated bytes 给出了在托管堆上分配的所有对象的总大小。
- Relocated bytes 给出了在垃圾收集期间被移动的对象的总大小。
- Final heap bytes 给出了当程序退出时托管堆上的所有对象的大小，其其中可能包含了带有终结器的对象，这些对象虽然不存在根对象引用，但还没有被收集。
- Objects finalized 给出了有多少对象成功地执行了 Finalize 方法。
- Critical objects finalized 给出了有多少对象成功地执行了临界 Finalize（Critical Finalize）方法。在这些对象中都包含一个临界 Finalize 方法，它使这些对象比其他普通的对象获得更强的保证。

如果观察在 Heap Statistics 部分中给出的数据，可以看到当程序执行完成时，大约有一半的内存被释放了。

下一组信息称之为 Garbage Collection Statistics。这部分信息非常简单，显示了在每一代（0－2）中发生的垃圾收集次数，并且 Induced collection 表示显式执行的垃圾收集操作（例如通过 GC. Collect 方法）次数。

GC Handle Statistics 部分包含了一些有价值的信息，这些信息在跟踪程序中的句柄使用情况时非常有用。它给出了在程序的生命周期中创建的句柄数量，所销毁的句柄数量，以及在程序运行结束时还有多少个句柄存在。在这里的情况中，我们可以看到所有被创建的句柄在程序结束时仍然存在，这意味着可以存在句柄泄漏问题。Garbage Collector Generation Sizes 给出了在程序运行过程中每一代（包括大对象堆）的平均大小。最后，Profiling Statistics 部分还详细给出了分析过程本身的信息，例如在分析过程中生成了多少个堆转储文件，以及被添加到日志文件的注释数量。在一些分组框中还包含了相关的按钮。例如，Heap Statistics 组在每个域旁边都有一个 Histogram 按钮，而 Garbage Collection Statistics 组有一个 Time Line 按钮。这些按钮都可以给出所收集数据的其他一些视图。在接下来的章节中，我们将看到其中一些视图，并解释视图中的输出。请注意，这些视图可以通过 CLR 分析器主窗口中的 View 菜单来访问。

9.3.3　Histogram 视图

我们首先来介绍 Histogram Allocated Types 视图（依次点击 View，Histogram Allocated Types 等菜单项）。在图 9-14 中给出了分析过程中得到的视图。

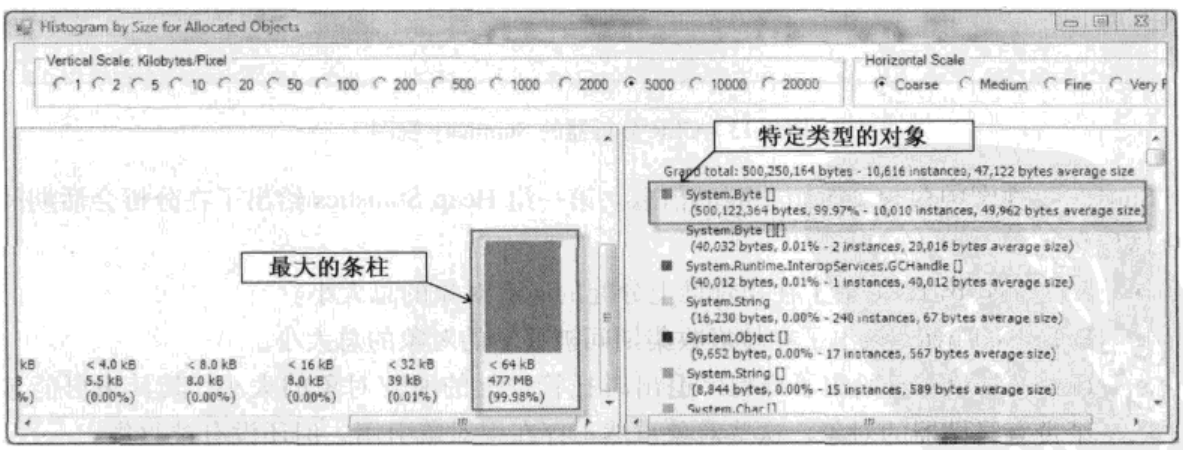


图 9-14　05Fragment. exe 的 Histogram Allocated Types 视图

Histogram Allocated Types 分为四个主要部分。柱状图根据大小给出了在程序中执行的内存分配类别。例如，有 39KB 个对象是大于 16KB 但小于 32KB 的。在分析过程中，最大的柱

状图表示小于 64KB 但大于 32KB 的对象。这个类别中的所有对象的总大小为 477MB，占据了托管堆中 99.98% 使用量。在右侧的面板中，可以通过柱的颜色与对象关联起来。在这里，柱为红色，表示一个 System. Byte ［］类型。最顶端的两个面板可以控制面板的纵坐标轴的最大值和横坐标轴的最大值。

通过分析 Histogram Allocated Types 视图，我们可以得出结论，托管堆中包含了大量的 System. Byte ［］。知道哪种类型占据了托管堆中的最大空间是一种非常有用的信息，此外在源代码中找出执行这些分配的位置同样是非常有用的。这可以很容易实现，在需要分析的柱上点击右键，并选择“Show Who Allocated“菜单项。这时出现一个 Allocation Graph 视图，如图 9-15 所示。

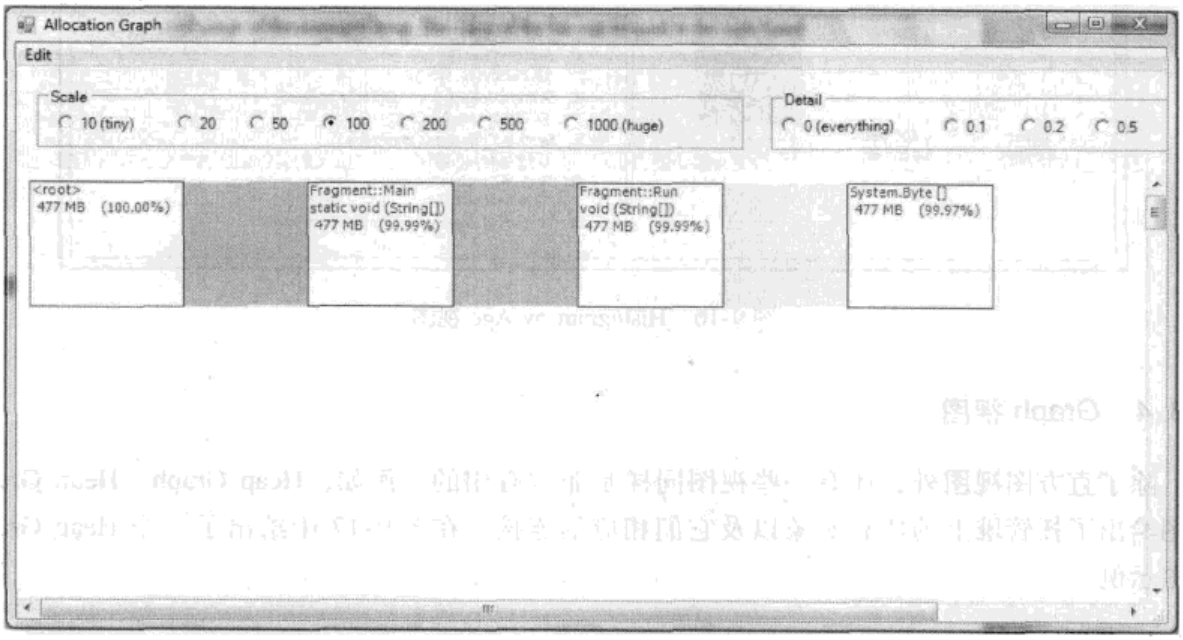


图 9-15　托管堆最大空间占有者的 Allocation Graph 视图

从图 9-15 中可以看到，分配图中指出根栈帧（main 入口点）调用了 Fragmen 类的 Main 方法，这个方法又调用 Run 方法，最终将分配一个 System. Byte ［］实例。请注意，为了节约屏幕空间，CLR 分析器截断了类型的名字。

另一个有用的信息是 Histogram by Age 视图，它将给出对象究竟存活了多长时间。在图 9-16 中给出了一个 Histogram by Age 视图示例。

这里，我们可以看到 239MB 的 System. Byte ［］已经存活了 100 到 150 秒。与其他的直方图视图一样，可以在柱上点击右键，并且选择“Show Who Allocated”菜单项来得到 Allocation Graph。

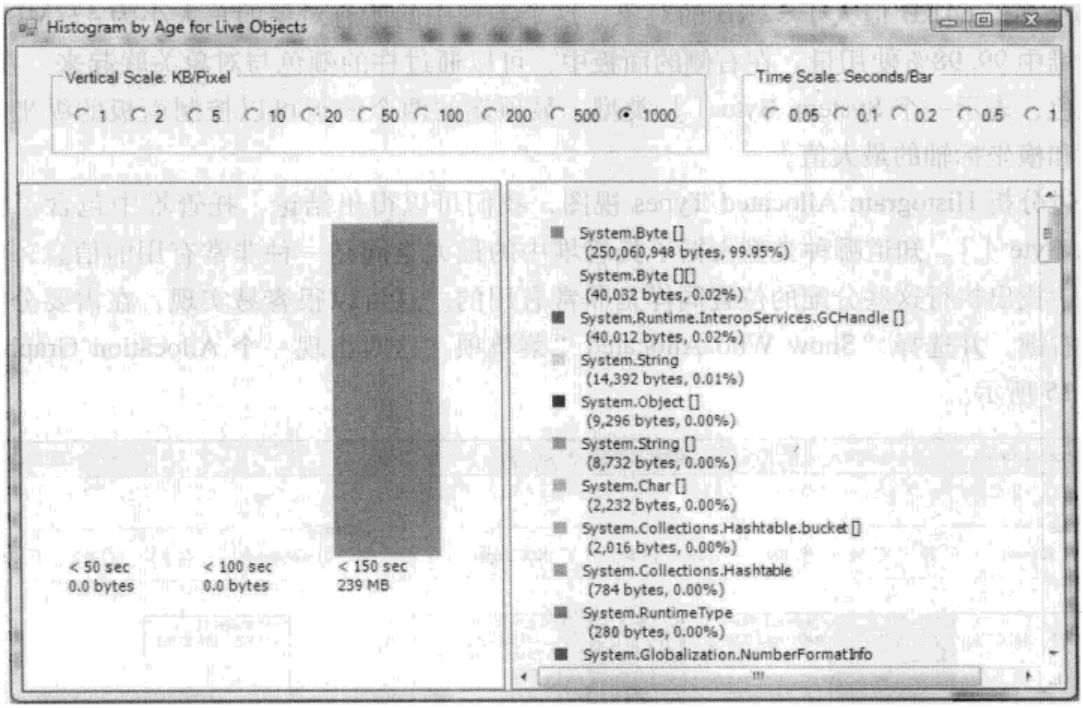


图 9-16 Histogram by Age 视图

9.3.4 Graph 视图

除了直方图视图外，还有一些视图同样是非常有用的。例如，Heap Graph。Heap Graph 视图给出了托管堆上的所有对象以及它们相应的连接。在图 9-17 中给出了一个 Heap Graph 视图示例。

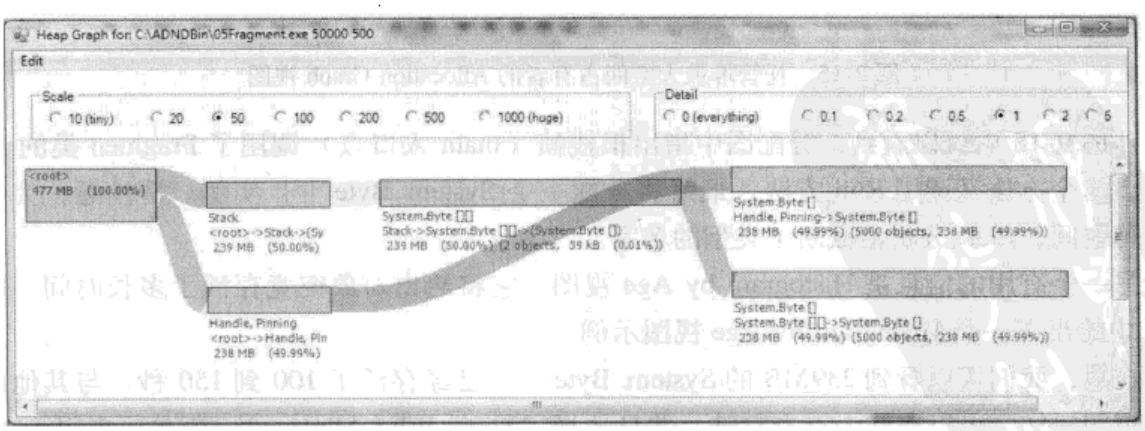


图 9-17 Heap Graph 视图

从图 9-17 中可以看到，System. Byte [ ] 占据了 239MB 的空间，而句柄（类型为 Pinned）占据了 238MB 的空间。

SOS 与 CLR 分析器

要生成 CLR 分析器能够理解的文件，可以使用 SOS TraverseHeap 命令。然后，可以将文件加载到 CLR 分析器中从而发挥 CLR 分析器的强大功能。

尽管 CLR 分析器是一个很成熟的工具，可以提供关于托管堆的大量信息，但还是存在一些局限：

- 降低程序的执行速度。当在 CLR 分析器下运行程序时，执行速度会降低 10 倍到 100 倍。
- 日志文件的大小。CLR 分析器收集了大量的数据保存在本地驱动器上。
- 程序必须由 CLR 分析器启动，而不能将 CLR 分析器附载到一个已经运行的程序。

9.4　WinDbg 和 CmdTree 命令

在本书中，我们使用了基于控制台版本的非托管调试器，分别是 ntsd 和 cdb。还有另一个非托管调试器 WinDbg，这是一个与控制台调试器对应的 GUI 版本。到目前为止我们讨论的这些命令在这三个调试器中都能使用，但也有些命令只能在 WinDbg 中使用，例如未公开的 cmdtree 命令。这个命令能使大量的键入工作自动化，如果没有它，那么在使用各种调试器命令时需要大量的手工键入工作。cmdtree 命令会显示一个窗口，在这个窗口中包含了一个层次式命令视图，用户可以双击这些命令并且执行。通常，命令是按照区域功能来分组的，例如线程、进程、模块等。在这个窗口中显示的命令可以通过编写一个简单的文本文件来定制，但这个文件要符合 cmdtree 命令的语法布局。在编写了这个文本文件后，可以使它通过 cmdtree < filename > 命令加载。例如，通过 cmdtree 命令来加载 sample. txt 数据文件如下所示。

```
windbg> .cmdtree c:\adndbin\sample.txt
```

在图 9-18 中给出了 cmdtree 窗口，作为之前命令的结果。

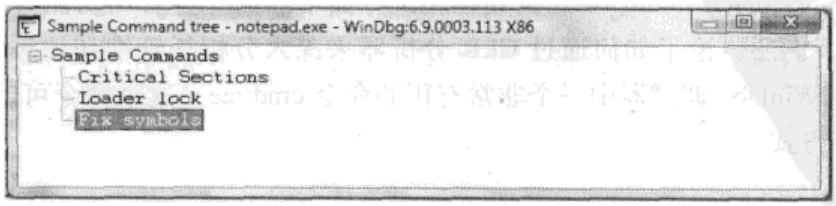


图 9-18　cmdtree 窗口示例

图 9-18 中的窗口包含了在 Sample Commands 节点下列出的三个命令。在这些命令中的任何一个上双击，会运行相应的命令，并在 WinDbg 命令窗口中输出结果。

在清单 9-2 中给出了 cmdtree 文件 sample. txt。

清单 9-2 示例 cmdtree 文件

```
windbg ANSI Command Tree 1.0
title {"Sample Command tree"}
body
{"Sample Commands"}
 {"Critical Sections"} {"!cs"}
 {"Loader lock"} {"X ntdll!LdrpLoaderLock"}
 {"Fix symbols"} {".symfix"}
```

第一行表示文件的头，指出这个文件符合 Command Tree 版本 1.0 命令的格式。title 命令表示命令窗口的标题，body 命令表示在实际窗口中显示的内容。在 body 部分中的每一行要么对应于一个树节点（在这种情况下没有指定相应的命令），要么是一个叶节点（在这种情况下，命令的名字后面紧接着实际的命令，在节点被双击之后由调试器来执行）。例如，在清单 9-2 中的 Loader lock 行将在调试器的命令窗口中执行 X ntdll! LdrpLoaderLock 命令。

可以看到，可以很容易对这个 cmdtree 文件进行扩充以包含其他非常有用的命令，这在调试时可以节约大量的时间。因此，我们不需要在命令窗口中键入命令，而是只需双击就可以快速并高效地得到结果。

9.5 小结

虽然非托管调试器包含了非常强大的功能，但并不妨碍你使用其他的调试工具。在某些情况下，如果使用某种不同的工具，甚至只使用一个辅助工具来增强非托管调试器，都会极大地提高调试的成功概率。在本章中，我们介绍了一些功能强大的工具，在调试过程中可以使用它们。我们介绍了 PowerDbg 库，它能够将 PowerShell 的脚本能力与非托管调试引擎结合起来，从而通过 PowerShell 来控制非托管调试器。我们还介绍了如何在 Visual Studio 2008 使用 SOS，以及如何访问. NET 框架源代码服务器来执行源代码级调试。此外，我们也简要介绍了即将发布的 Visual Studio 2010 及其在调试方面增加的一些有用功能（托管转储调试，历史调试等）。我们还讨论了如何通过 CLR 分析器来深入分析托管堆和垃圾收集器。最后，我们详细介绍了 WinDbg 调试器中一个非常有用的命令 cmdtree，这个命令可以为常用的调试器命令定义快捷方式。

第 10 章　CLR 4.0

.NET 框架 1.0 是在 2002 年发布的，在接下来的几年中，Microsoft 又陆续发布了几个包含新增功能的版本（.NET 1.1、2.0、3.0 和 3.5）。各个版本之间的差异主要在于框架中新增的功能，而对 CLR 的修改则越来越少。即将发布的.NET 版本为 4.0，这个版本对 CLR 的可靠性，性能以及功能等方面进行了重大的改进。在本章中，我们将介绍 CLR4.0 的一些新功能。在本章的每一部分都会提到前面章节曾介绍的一些 CLR 组件及其在 CLR 2.0 和 CLR 4.0 中的最主要区别

CLR 4.0 Beta

请注意，在编写本书时已经发布了.NET 4.0 的 Beta 1，本章给出的所有信息都是基于这个版本的。在正式产品发布之前，Microsoft 有权修改，增加和删除 Beta 1 中的任何功能。

10.1　工具

在本书中使用的工具集基本上保持不变，只不过对工具本身的一些功能进行了增强。

10.1.1　Windows 调试工具集

要在非托管调试器中使用 CLR 4.0，必须从以下位置下载最新的调试器。本章示例程序使用的 Windows 调试工具集版本是 6.11.0001.404。

http://www.microsoft.com/whdc/devtools/debugging/default.mspx

10.1.2　.NET 4.0 可再发行组件

.NET 4.0 Beta 1 可再发行组件可以从以下位置下载：

http://www.microsoft.com/visualstudio/en-us/try/default.mspx

请注意，如果正在使用 MSBUILD（基于控制台的构建系统）构建.NET 4.0 程序，必须使用/toolsversion 开关，并将目标版本指定为 4.0。例如，要使用 4.0 构建系统构建 05oom.exe，可以使用以下命令：

```
MSBuild.exe /toolsversion:4.0 build.xml
```

10.1.3 SOS

正如本书所介绍的，如果在调试.NET 程序问题时需要深入了解运行时本身，那么 SOS 调试扩展将是一种非常有用的工具。SOS 调试器扩展的早期版本与特定版本的 CLR 绑定在一起，并作为可再发行组件的一部分。由于调试器命令 loadby 可以找出 SOS 的正确版本（根据指定的 dll），通常就不需要知道 SOS 的准确位置（甚至不需要知道正在使用哪一个 CLR）。在 CLR 4.0 中需要注意，mscrowks.dll 被替换为 clr.dll。因此，要为 CLR 4.0 程序加载 SOS 调试器扩展，可以使用以下命令：

```
0:006> .loadby sos clr
```

在新的 SOS 调试器扩展中包含了一组新命令，如表 10-1 所示。

表 10-1 SOS 4.0 中的新命令

名 字	描 述
ThreadState	将线程状态转换为一个文本表示
COMState	输出进程中每个线程的套间模型
DumpSigElem	输出签名对象（Signature Object）中的单个元素
VerifyObj	检查传递进来的对象是否被破坏了
FindRoots	设置一个断点，当下一次垃圾收集操作（在任意代或者指定的代）发生时，调试器会停止
HeapStat	输出每一代堆的详细信息，包括每个堆的大小，总体大小以及空闲空间
GCWhere	给出托管堆上指定对象的位置（包括该对象位于哪一代）
ListNearObj(lno)	输出指定对象前后的对象
FindAppDomain	找出指定对象的应用程序域
AnalyzeOOM(ao)	输出最近一次在响应内存分配请求时出现的内耗耗尽异常

在本章后面将依次介绍各个新命令。

10.2 托管堆与垃圾收集

在 CLR 4.0 中对垃圾收集器进行了重大修改，增加了额外的诊断信息以及一种新的垃圾收集模式，即后台垃圾收集（background garbage collection）。在本章的这部分内容中，我们将首先看到这些新增的诊断信息，随后介绍后台垃圾收集模式。

10.2.1 扩展的诊断信息

在第 5 章，我们详细介绍了托管堆管理器以及垃圾收集器的内部工作机制。我们通过 SOS 调试器命令来了解 GC 的工作机制，并介绍了在分析程序问题根源时可以使用的一些信息。在 CLR 4.0 中，SOS 调试器扩展包含了一组新命令，可以进一步帮助我们找出程序中与 GC 相关的问题。在本节中，我们将讨论一些新的命令以及它们的使用方法。

1. VerifyObj

第一个需要注意的命令就是 VerifyObj，它的语法如下所示：

```
!VerifyObj <object address>
```

命令的参数是一个对象地址，它能判断对象是否被破坏了。检测对象是否破坏的算法是，判断在对象及其包含对象中的方法表是否完整。如果怀疑存在一个堆破坏，那么从这个命令的输出中可以很快判断出来。以下是一个被破坏的对象以及 VerifyObj 命令的输出：

```
0:000> !VerifyObj 0x02126804
object 0x2126804 does not have valid method table
```

2. FindRoots

要找出某个对象还没有被收集的原因，是一个复杂的过程。如果根对象很简单，那么要找出它引用的对象并不困难，但有的时候，要找出对象的根对象引用并不容易。例如，如果在对象中包含了跨代（Cross－Generational）的引用，并且引用的代还没有被收集，那么这个对象看上去仍然是存活的。当检测到这些跨代的引用时，为了使工作更简单，可以使用 FindRoots 命令：

```
!FindRoots -gen <N> | -gen any | <object address>
```

FindRoots 命令指示运行时设置一个断点，这个断点可以被设置为在指定的代中下一次发生垃圾收集时（使用 gen ＜N＞开关）触发，或者每当发生垃圾收集操作时就触发（使用 gen any 开关）。在断点被触发后，FindRoots 命令将得到一个对象的地址，命令的执行结果是显示这个对象的根对象。

在这个过程中，第一步通常是找出这个对象所属的代，可以使用 GCWhere 命令：

```
0:003> !GCWhere 00b08580
Address   Gen   Heap   segment   begin       allocated       size
00b0b400  0     0      00b00000  00b01000    00b0c010        0xc(12)
```

从输出中可以看出，位于地址 0x00b08580 的对象属于第 0 代。

接下来，我们使用 FindRoots 命令，并设置为当发生下一次垃圾收集操作时中断，然后恢复程序的执行：

```
0:003> !FindRoots -gen any
0:003> g
(710.970): CLR notification exception - code e0444143 (first chance)
CLR notification: GC - Performing a gen 2 collection. Determined surviving
objects...
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0013f118 ebx=00000000 ecx=00000000 edx=00000006 esi=0013f1dc edi=00000003
eip=7c812afb esp=0013f114 ebp=0013f168 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
KERNEL32!RaiseException+0x53:
7c812afb 5e              pop     esi
```

在断点被触发后，我们可以使用 FindRoots 命令以及对象的地址来找出这个对象的根对象。

```
0:000> !FindRoots 00b0b400
Scan Thread 0 OSTHread 970
ESP:13fac8:Root:  01b01010(System.Object[])->
  00b0ab38(System.Collections.Hashtable)->
  00b0ab70(System.Collections.Hashtable+bucket[])->
  00b0b400(System.Int32)
Scan Thread 2 OSTHread acc
DOMAIN(0016CB98):HANDLE(Pinned):9713fc:Root:  01b01010(System.Object[])->
  00b0ab38(System.Collections.Hashtable)->
  00b0ab70(System.Collections.Hashtable+bucket[])->
  00b0b400(System.Int32)
```

3. HeapStat

HeapStat 命令将给出每个托管堆上每一代中已使用字节数和空闲字节数的详细信息。此外，它还会给出小对象堆（SOH）和大对象堆（LOH）上已使用字节数与空闲字节数的比值（百分比）。这个命令的语法如下所示：

```
!HeapStat [-inclUnrooted | -iu]
```

命令的默认输出包含所有存在根对象引用的对象。inclUnrooted（或者简写为 iu）开关表示包含所有存在根对象引用的对象以及不存在根对象引用的对象。以下是在 05oom. exe 程序上运行 HeapStat 命令的输出：

```
0:004> !HeapStat
Heap       Gen0    Gen1     Gen2     LOH
Heap0   2166844 134200   159064   33328

Free space                                  Percentage
Heap0   1865804      12       36      96     SOH: 75% LOH:  0%
```

在输出中给出了内存的当前使用量：2.1MB 位于第 0 代，134KB 位于第 1 代，159KB 位于第 2 代，最后在 LOH 中有 33KB。“Free Space” 表示在第 0 代、第 1 代和第 2 代中分别有 1.8MB、12MB 和 36MB 的空闲空间。

4. GCWhere

在第 5 章中，我们介绍了如何找出某个对象属于第几代的过程。这个过程包括，将托管堆的内存段转储出来（通过 eeheap 命令），然后将对象的地址与输出中给出的堆内存段进行比较（在输出中给出了每个堆内存段对应的代）。如果只想找出一个或两个对象的代，那么这个过程或许管用，但如果要找出多个对象的代，那么这个过程将变得非常繁琐。在 SOS 4.0 中引入了一个命令，它将显示出对象的一些信息。GCWhere 命令的语法如下所示：

```
!GCWhere <object address>
```

下面是在 FileStream 对象上运行时的示例输出：

```
0:000> !GCWhere 0x01efd3f8
Address    Gen   Heap   segment    begin      allocated          size
01efd3f8   0     0      01ea0000   01ea1000   020f99cc        0x50(80)
```

在输出中给出了对象的地址（0x01efd3f8），这个对象属于第几代（Gen 0），托管堆（0），内存段指针（0x01ea0000），内存段的起始地址（0x01ea1000），在内存段上分配的字节数（0x020f99c），以及对象的大小（0x50）。请注意，这个大小并不是递归大小（即不包含子对象的大小）。

5. ListNearObj

我们可以使用 ListNearObj 命令来验证堆的一致性。这个命令的参数是一个对象地址，它将尝试验证在指定对象前后的对象。这个命令的语法如下所示：

```
!ListNearObj <object address>
```

例如，在某个有效对象（并且在该对象周围同样是有效对象）上运行 ListNearObj 命令将产生以下输出：

```
 0:000> !ListNearObj 0x01efd3f8
Before:  01efd3c8   48 (0x30)   System.Collections.Hashtable+bucket[]
Current: 01efd3f8   80 (0x50)   System.IO.FileStream
After:   01efd448   28 (0x1c)   System.String
Heap local consistency confirmed.
```

输出被分为 before，current 和 after，接下来是验证操作的结果。在 before，current 和 after 等部分中指定了对象的地址，大小以及类型。在前面的示例中，所有这三个对象都被认为是有效的，因此这个命令认为堆的一致性是完整的。另一方面，如果在一个已被破坏的对象（其中这个对象的大小被覆盖了）上运行这个命令，那么将看到以下输出：

```
0:000> !ListNearObj 0x01efd3f8
Before:  01efd3c8          48 (0x30)   System.Collections.Hashtable+bucket[]
After:   01efd448          28 (0x1c)   System.String
Heap local consistency not confirmed.
```

6. AnalyzeOOM

在第 5 章中，我们看到了在一些示例程序中抛出了内存耗尽的异常。我们还介绍了如何分析托管堆来获得关于内存耗尽异常的更多信息。在 SOS 4.0 中引入了一个新命令 AnalyzeOOM，它能在诊断内存耗尽问题的过程中提供帮助。这个命令的语法如下所示：

```
!AnalyzeOOM
```

我们通过一个程序 10OOM.exe 来说明如何使用这个命令。程序 10OOM.exe 执行一个密集循环，并且分配大量的内存，直到内存被耗尽。在调试器下运行这个程序，直到抛出内存耗尽异常，如下所示：

```
(2b14.281c): C++ EH exception - code e06d7363 (first chance)
(2b14.281c): CLR exception - code e0434352 (first chance)
ModLoad: 75370000 75378000   C:\Windows\system32\VERSION.dll

Unhandled Exception: OutOfMemoryException.
(2b14.281c): CLR exception - code e0434352 (!!! second chance !!!)
eax=0030edc0 ebx=00000005 ecx=00000005 edx=00000000 esi=0030ee6c edi=003fa160
eip=75e242eb esp=0030edc0 ebp=0030ee10 iopl=0         nv up ei pl nz ac pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000216
KERNEL32!RaiseException+0x58:
75e242eb c9              leave
```

接下来执行 AnalyzeOOM 命令：

```
0:000> !AnalyzeOOM
Managed OOM occured after GC #247 (Requested to allocate 1048576 bytes)
Reason: Low on memory during GC
Detail: SOH: Failed to reserve memory (16777216 bytes)
```

命令的输出告诉我们，在第 247 次垃圾收集操作后发生了内存耗尽问题，所请求的内存量为 1048576。它还给出了发生这个问题的原因，即在垃圾收集操作过程中内存不足。最后，在 details 部分指出，它在保留 16777216 字节的内存时失败了，这对应于小对象堆上最小内存段的大小。

在前面的示例中，我们很清楚为什么会发生内存耗尽的问题，但在实际情况中也可能是其他原因，例如 CLR 试图分配某个内部数据结构，或者其他组件抛出内存耗尽异常。

10.2.2　后台垃圾收集

在 CLR 4.0 以前，垃圾收集器可以在两种不同的模式下工作。第一种模式是工作站模式（或者叫做并发 GC），目标是在工作站上运行的程序，例如图形界面程序。第二种模式是服务器模式（或者叫做阻塞 GC），目标通常是不需要任何图形界面的服务器端程序。这两种模式的差异主要在于执行垃圾收集时的响应时间。在第 5 章中已经讨论过，在执行垃圾收集的过程中，执行引擎以及相关的托管线程必须被定期地挂起，以免触发另一次垃圾收集操作。显然，这种托管线程的挂起操作会带来短暂的停止，对使用程序的用户产生一定的影响。在带有图形界面的应用程序中，这可能导致界面闪烁，或者其他细微的现象，例如在用户点击按钮与获得响应之间出现延迟。在这些情况下，重要的是要确保线程处于挂起状态的时间尽可能地少。并发（或者工作站）GC 采用的方法是，在 GC 中将所有托管线程只挂起两次，而不是在整个生命周期中都挂起。在托管线程没有被挂起的时间里，它们能够持续分配内存，直到到达临时内存段的末尾。如果临时内存段的空间耗尽了，并且有一个并发 GC 正在执行，那么托管线程会被挂起，直到并发 GC 完成（将并发 GC 转换为一个阻塞 GC）。这意味着只要临时内存段没有被耗尽，就可以避免延迟。

另一方面，服务器 GC 并不用像工作站 GC 那么关心实时响应时间，因为大多数服务器

程序并不需要实时响应时间。服务器模式 GC 不允许在 GC 的执行过程中分配内存，而是在 GC 的整个执行过程中挂起所有线程。虽然这可能会在执行 GC 期间导致某种不可见的延迟时间，但好处是获得了更高的吞吐量，因为此时无需担心其他托管线程可能会在同时执行。此外，在服务器 GC 中，每个处理器都有一个专门的 GC 线程，因而可以同时运行 X 个 GC（其中 X 等于处理器的数量乘以每个处理器中核的数量）。并发 GC 的主要缺陷之一是，它通常只在小规模托管堆的程序上运行得较好（记住，只要临时内存段没有被耗尽，那么托管线程将持续分配内存）。在实际情况中，在程序中拥有数 GB 字节托管堆的情况并不多见。在这些情况下仍然存在延迟时间，因为当达到了临时内存段的限制时，并发 GC 将转变为一个阻塞 GC。为了解决这种低效问题，CLR 4.0 把并发 GC 替换为后台 GC。并发 GC 与后台 GC 之间的主要差异在于，后台 GC 允许 GC 和内存分配操作同时执行，并且允许收集第 0 代和第 1 代的对象。后台 GC 将定期检查是否有某个并发的内存分配导致了在临时内存段执行 GC 操作，如果是，将挂起自己，并且允许在临时内存段中的 GC 执行（前台 GC）。因此，虽然执行了一个完整的 GC，但我们仍然能够删除那些短暂存活的对象。由于后台 GC 允许在第 0 代和第 1 代中执行收集操作，那么在达到临时内存段的阈值时将发生什么情况？在这种情况下，前台 GC 将根据需要增加临时内存段。

总的来说，在 GC 执行过程中，服务器 GC 通常会阻塞。为了避免在这种挂起中造成延迟时间，引入了工作站 GC，它能把线程在 GC 过程中处于挂起状态的时间降到最低。虽然这种方法对于托管堆相对较小的程序能够起到作用，但如果临时内存段被耗尽了，那么仍然会造成延迟时间。这种缺点促使工作站 GC 演变为后台 GC，允许内存分配操作和临时内存段的收集操作真正地并发执行（如果需要的话还会扩展内存段）。

请注意，在 CLR 4.0 中，后台 GC 只有在工作站模式中才是有效的。

10.3 同步

在第 6 章中，我们看到了.NET 中一些最常用的同步原语以及 CLR 中的同步机制。在.NET 4.0 中对现有的同步机制进行了大幅度增强，此外还增加了新的同步原语。在本章的这部分内容中，我们将介绍.NET 4.0 中的一些新增功能。

10.3.1 线程池与任务

在 CLR 2.0 中引入的线程池提供了一种非常好的机制对工作进行排列，然后从队列中取出工作项并放到线程池的某个线程上执行。线程池的好处在于，可以由底层运行时来决定哪个线程执行哪个工作项。

这种方法工作得很好，因为线程池最清楚所要执行的任务，以及哪些线程最适合于服务其他的请求。尽管如此，运行时只知道它有一个工作项队列，以及一组线程来服务这些请求。

这可能无法实现最佳性能，例如，工作项之间是相关的，因此如果不能对工作项正确地排序，那么将无法实现最佳性能（毕竟线程池以 FIFO 形式来选择新的工作项）。而且，虽然工作项队列的数据结构是一个实现细节，但这个数据结构以及每当工作项入列和出列时执行的锁定操作，都增加了整体开销。为了解决这种锁定带来的低效问题，在. NET 4.0 中使用了一种无锁数据结构，从而避免了锁定和解锁等操作带来的开销。此外，. NET 4.0 还修改了数据结构，使其对 GC 更加友好，因此加速了这些数据结构的收集过程。虽然这两项改进能够提升性能，但仍然无法使得线程池了解工作项本身的内容。工作项只是被视为以 IFO 顺序排序的不透明数据。如果能在某种程度上将与工作项相关的信息告诉线程池，那么就可以对它们的执行顺序进行优化。. NET 4.0 通过引入任务并行库（Task Parallel Library，TPL）来解决这个问题。在这里不会介绍关于 TPL（System. Threading. Tasks）的太多细节，只能告诉读者，它公开了一组更为丰富的 API 来更清晰地定义工作项的细节，因此能够实现更高效的任务调度和执行。

10.3.2 监视器

在第 6 章中，我们讨论了在程序中使用 Abort 来结束线程，还介绍了线程中止操作的内部工作流程，以及为什么线程中止操作会在获取锁（使用了 lock 语句）的线程中导致问题。根本的问题在于编译器生成的 IL。具体来说，在 try 块之前插入了一个 nop 指令。结果就是，当执行 nop 指令时抛出了 ThreadAbortException，finally 语句永远都不被执行，因此将永远不会释放锁。要解决这个问题，. NET 4.0 引入了 Monitor. Enter 方法的一个重载版本：

```
public static void Enter(Object obj, ref bool lockTaken)
```

如果获取了锁，那么 lockTaken 参数就为 TRUE，否则为 FALSE。新的重载方法允许以下模式：

```
bool acquired=false;

try
{
    Monitor.Enter(objToLock, ref acquired);

    //在持有锁时执行一些工作
}
Finally
{
    if(acquired)
    {
        Monitor.Exit(objToLock);
    }
}
```

现在，编译器在为 lock 语句生成 IL 时同样进行了更新以遵循相同的模式，并且总是在 try 语句中获取锁，因此能够确保即使出现 ThreadAbortExceptions 情况仍然能够在 finally 语句中释放这个锁。

10.3.3 栅栏

栅栏对象（System. Threading. Barrier）最好被视为一种将一系列操作串行化的方法，在栅栏对象中包含了一个或多个检查点，在这个操作完成之前必须首先达到这些检查点。例如：假定某个任务需要填充一组缓冲区，再使用缓冲区中的数据来执行各种计算。在第二个阶段（计算节点）启动之前，需要首先填充完所有的缓冲区。我们可以创建一个包含 X 个参与者的栅栏对象，每个参与者在执行完每个阶段（write，read）后必须在栅栏处等待其他的线程执行完毕，然后再一起继续执行。

10.3.4 CountdownEvent

CountdownEvent 类（System. Threading. CountdownEvent）是一个计数事件，只有当它的计数值为 0 时才会触发。

10.3.5 ManualResetEventSlim

ManualResetEventSlim 类（System. Threading. ManualResetEventSlim）有些类似于现有的 ManualResetEvent。二者的关键区别在于，在等待获得某个已经被获取的锁时，新引入的 ManualResetEventSlim 原语将首先进行自旋。如果在指定的自旋时间内仍然无法获取这个锁，那么它将进入到等待状态（例如在 ManualResetEvent 中的情况）。由于在默认情况下并不会直接进入到等待状态，因此就不需要分配包含等待状态的数据结构，这也就是为什么在类的名字中包含了 Slim（意思为轻量级）。注意，与 ManualResetEvent 类不同的是，Slim 版本只能在跨进程的情况下使用。

10.3.6 SemaphoreSlim

与前面讨论的 ManualResetEventSlim 非常类似，Semaphore 类同样包含了高效的（自旋）版本，叫做 SemaphoreSlim（System. Threading. SemaphoreSlim）。注意，与 Semaphore 类不同的是，Slim 版本只能在跨进程的情况下使用。

10.3.7 SpinWait 和 SpinLock

当某个锁的平均持有时间很短时，自旋是等待锁被释放的一种更高效方式。主要原因在于，在自旋中不需要分配相应的资源（例如一个事件）来进入等待状态，线程会在自旋过程

中查看这个锁是否被释放。自旋的开销要远远小于在实现高效等待时需要的开销。如果只希望使用锁的自旋特性，那么可以使用 SpinLock 类（System. Threading），而如果希望在进入等待状态之前自旋一段时间，那么可以使用 SpinWait 类（System. Threading）。

10.4 互用性

正如 CLR 4.0 在一些领域中做出了重大改进，对互用性（interoperability）同样进行了重大修改。在执行 COM 互用性时，主要问题之一是使用主互用性程序集（PIA）。在第 7 章中我们已经详细地介绍了 PIA。简要地说，PIA 是一个独立的程序集，作为底层 COM 对象的托管代码“代理”。PIA 的最大缺陷在于，它是一个独立的程序集，必须与使用它的程序一起使用，这导致了在程序集的部署上将出现一些问题，例如 PIA 的版本和大小。要使用 PIA，必须在程序中部署整个 PIA，即使只需要一小部分。为了解决这个问题，CLR 通过 NoPIA（非主互用性程序集）引入 COM 互用性。这可以将调用 COM 对象时所需的信息嵌入到程序本身来实现，因此不需要单独部署 PIA。在第 7 章，我们看到了一个使用 PIA 的 COM 互用性示例（07ComInterop. exe），它要求了一个单独的 PIA 程序集。要在同一个示例程序中使用 NoPIA，我们可以使用完全相同的代码，只是需要对程序的构建方式做一些修改。现在不需要把 PIA 程序集作为构建的一部分，而是可以使用编译器开关/link（在 CLR 4.0 中新引入的）来指定一个 PIA 程序集，并且通知编译器将元数据嵌入到程序中。请注意，编译器只会把在程序中使用的元数据嵌入进去，因此如果只有一小部分 PIA 在使用，那么将极大地减少应用程序的大小。

另一个主要的增强功能就是，使开发人员更容易编写 P/Invoke 函数原型。在. NET 4.0 中包含了一种新工具叫做 P/Invoke 互用性助手（P/Invoke Interop Assistant），可以从 CodePlex 上下载：

```
http://www.codeplex.com/clrinterop
```

P/Invoke 互用性助手通过读取 windows. h 头文件中的信息来生成函数原型和封装器，在这个头文件中包含了 Windows 公开的大量函数。此外，它还能理解 SAL 注记，这些注记已经成为 Windows 中函数的标准部分，并使得这个工具更容易生成准确的函数原型和封装器。在图 10-1 中给出了这个工具的一个示例，以及为 CreateFileW 生成的代码。

此外，在 SOS 2.0 中可以使用却未被公开的 RCWCleanupList，在 SOS 4.0 已经被完全公开了。这个命令能显示所有不再被使用并将被清除的运行时可调用封装器。

```
0:000> !RCWCleanupList
MTA Interfaces to be released: 0
STA Interfaces to be released: 1
```

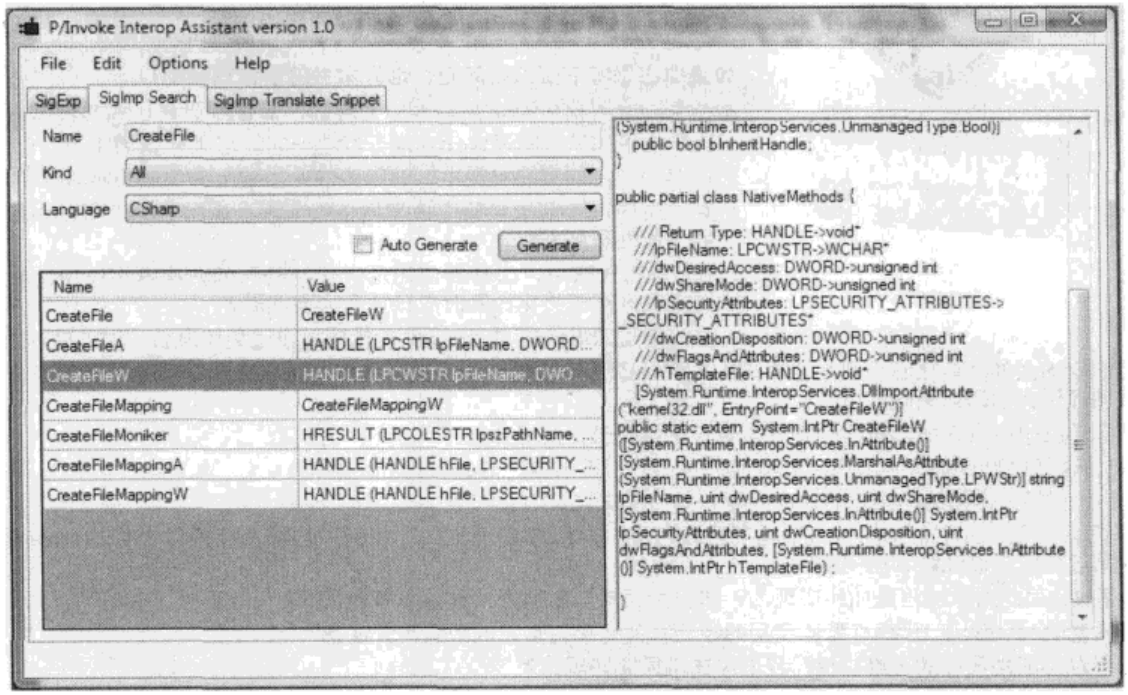


图 10-1 P/Invoke 互用性助手的示例

10.5 事后调试

在 CLR 4.0 的事后调试功能中也存在一些值得注意的变化。在第 8 章，我们介绍了为托管代码启用 JIT 调试器的各种不同方式，还讨论了如何通过 AeDebug 注册键来为非托管代码启用 JIT 调试功能，以及通过 DbgManagedDebugger 注册键来为托管代码启用 JIT 调试功能。在 CLR 4.0 中，这两个键被合并为一个键，只需查看 AeDebug 键来判断何时启动一个 JIT 调试器（非托管代码或者托管代码）即可。这些都极大地简化了将 JIT 调试过程自动化的过程。

10.6 小结

.NET 4.0 是对.NET 框架的最新升级，其中既包含了非常丰富的新功能，又包含了对运行时领域各个方面的一些增强功能，例如性能，可靠性以及安全性等。

在本书中，我们介绍了各种调试情况，并在本章中详细介绍了其中一些情况以及在使用.NET 4.0 时存在的差异。我们首先介绍了新的工具需求，然后给出了与某项特定技术相关的调试情况，例如托管堆和垃圾收集器、同步、互用性等。

Web前端开发者的内功修炼秘笈
4大社区鼎力推荐！

作者：曹刘阳 著　ISBN：978-7-111-30595-8　定价：49.00 元

如果你在思考下面这些问题，也许本书就是你想要的！

- 作为一名合格的Web前端开发工程师，究竟需要具备哪些技能和素质？为什么说如果要精通Web前端开发这一行，必须先精通十行？
- 在Web应用的实现代码中，有哪些技术因素会导致应用难以维护？
- 高质量的Web前端代码应该满足哪些条件？如何才能提高Web前端代码的可读性和可重用性？
- 在HTML代码中，为何要使用语义化标签？如何检查你使用的标签是否语义良好？语义化标签时应该注意哪些问题？
- 如何编写CSS代码和JavaScript代码可以避免团队合作时产生冲突？
- 如何组织CSS文件才能让它们更易于管理？如何让CSS模块化，从而提高代码的重用率？CSS的命名应该注意哪些问题？何谓优良的CSS编码风格？
- 如何在CSS编码中引入面向对象的编程思想？这样做有哪些好处？
- 原生JavaScript和JavaScript类库之间有何关系？如何编写自己的JavaScript类库？
- JavaScript有哪些常见的跨浏览器兼容问题？如何解决这些问题？
- 如何组织JavaScript才能让代码的结构更清晰有序，从而更易于维护？如何才能编写出弹性良好的JavaScript代码？编写过程中应该注意哪些问题？
- JavaScript的面向对象编程是如何实现的？如何用面向对象的方式重写原有的代码？
- 编写高质量的JavaScript代码有哪些实用的技巧？又有哪些常见的问题需要注意？
- 为了提高Web前端代码的可维护性，我们应该遵循哪些规范？

全面探讨Web 前端设计的方法、原则、技巧和最佳实践
5大专业社区一致鼎力推荐！

作者：赖定清 林坚 著　ISBN：978-7-111-31245-1　定价：59.00 元

如果你在思考下面这些问题，也许本书就是你想要的！

- 作为一名合格的Web前端开发工程师，究竟需要具备哪些技能和素质？为什么说如果要精通Web前端开发这一行，必须先精通十行？
- 前端设计者如何才能正确地理解自己的用户？如何理解并实践以用户为中心的设计原则？
- 原型设计应该注意哪些问题？如何更好地利用工具快速地进行原型设计？
- 可用性设计的关键要素是什么？如何设计高可用性的页面元素（导航、表单、链接等）？
- “可用性”的首要原则是“别让我思考”，你的网站如何才能做到不让用户思考呢？
- 可用性测试的5项目标是什么？如何通过可用性测试发现问题现象背后的本质？
- 如何保持设计的一致性？一致性设计的三项原则是什么？
- 如何理解“样式就是设计”这句话？有哪些样式技术是前端开发者和设计者必须掌握的？样式究竟有哪些功能？
- 如何编写易于管理、维护和复用的JavaScript代码？JavaScript有哪些最佳实践？
- 如何理解HTML文件、CSS文件和JavaScript文件之间的关系？如何良好地组织这些文件从而让它们更易于管理、复用和维护？
- 如何平衡网站的色彩？如何让你的网站设计简洁而美观？页面排版的艺术你知多少？
- Web前端设计领域有哪些经典的设计思维？如何才能掌握这些设计思维的本质？
- 如何测试前端的性能？前端性能优化的基本原则是什么？如何进行页面内容的优化和服务器端的优化？如何利用SEO技术让你的网站更容易被发现？
- CSS 3与HTML 5将带来哪些全新的设计方式？
- Web 3.0真的来了吗？Web 3.0的先驱者们有哪些杰出的表现？Web前端开发与设计的未来会怎样？

这是一次软件开发者的心灵沟通之旅 10大社区一致鼎力推荐！

作者：倪健 著
ISBN：978-7-111-30103-5
定价：55.00 元

多年以来，不管是从事一线的软件开发工作，还是从事管理工作，作者一直在思考这样一个问题：业界有这么多知识财富，可是在实践中真正能够被吸收和应用的却很少，这些知识财富的价值是毋庸置疑的，软件开发人员的热情和渴求也是有目共睹的，可问题究竟出在哪里呢？作者最后得出的结论是：这个问题要归结于思想和文化。

无论方法论也好，管理理论也好，都是技术层面的，它们来自于大师们的总结和提炼，本身是非常好的，但是它们有复杂的上下文，例如在敏捷开发中没有提到或者强调的——注重个体文化、专业化思想，以及多元化价值认同等，这些都属于思想和文化的范畴，它们是方法论和管理理论的运行环境。换句话说，如果割裂这些上下文，机械地运用那些技术层面的东西，效果就总是不好。

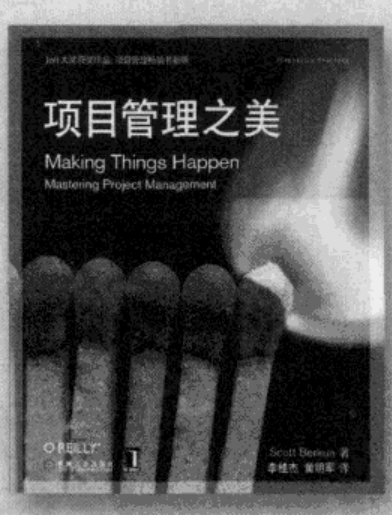
所谓机械运用，就是追求形式化的执行，而没有站在以人为本的角度展开思考。以人为本的核心就是对思想和文化的关注。成熟的思想和文化，可以使软件开发中的每一个细节都变得浑然天成；相反，不注重思想和文化的建设，就只能依赖无法预料的外部约束，例如，沉迷于方法论的技术细节而无法自拔。事实上，在软件开发领域，没有解决问题的银弹，没有提升效率的短期计划，也没有获得成功的操作指南，一切都依赖于人。

《项目管理之美》引人思考、生动有趣、坦诚直率、引人注目，是你和你的团队在当前和未来的项目中所必备的参考。通过《项目管理之美》，你可以从一位经验丰富、从事多年软件开发和Web开发的经理那里学习如何计划、管理和领导项目。书中的那些宝贵而有用的建议，是作者十多年经验的积累，从很多复杂的概念和挑战中提炼而来。

包括如下主题：如何制定好的决策、想法以及如何处理、领导力和信任、当事情出错时该怎么办。

新版包括：120多道新的练习题考察你所学到的知识、用于和团队一起使用《项目管理之美》的讨论指南、每章都有修订和改进的建议。

《项目管理之美》抓住了我们平时在项目管理过程中所涉及的重点，以轻松朴实、容易理解的叙述方式告诉我们：什么是项目管理？相关管理应该做哪些事情？为什么做这些事情？怎么做这些事情？在具体的环境下应该重点关注什么？忽略什么？



作者：赖定清 林坚 著
ISBN：978-7-111-31245-1
定价：59.00 元

专业成就人生
立体服务大众

HZ BOOKS

www.hzbook.com

填写读者调查表　加入华章书友会
获赠精彩技术书　参与活动和抽奖

尊敬的读者：

感谢您选择华章图书。为了聆听您的意见，以便我们能够为您提供更优秀的图书产品，敬请您抽出宝贵的时间填写本表，并按底部的地址邮寄给我们（您也可通过www.hzbook.com填写本表）。您将加入我们的“华章书友会”，及时获得新书资讯，免费参加书友会活动。我们将定期选出若干名热心读者，免费赠送我们出版的图书。请一定填写书名书号并留全您的联系信息，以便我们联络您，谢谢！

书名：　　书号：7-111-(　　)

姓名：	性别：☐男　☐女	年龄：	职业：
通信地址：		E-mail：	
电话：	手机：	邮编：	

1. 您是如何获知本书的：

☐朋友推荐　☐书店　☐图书目录　☐杂志、报纸、网络等　☐其他

2. 您从哪里购买本书：

☐新华书店　☐计算机专业书店　☐网上书店　☐其他

3. 您对本书的评价是：

技术内容	☐很好	☐一般	☐较差	☐理由_____
文字质量	☐很好	☐一般	☐较差	☐理由_____
版式封面	☐很好	☐一般	☐较差	☐理由_____
印装质量	☐很好	☐一般	☐较差	☐理由_____
图书定价	☐太高	☐合适	☐较低	☐理由_____

4. 您希望我们的图书在哪些方面进行改进？

_____

_____

5. 您最希望我们出版哪方面的图书？如果有英文版请写出书名。

_____

_____

6. 您有没有写作或翻译技术图书的想法？

☐是，我的计划是_____　☐否

7. 您希望获取图书信息的形式：

☐邮件　☐信函　☐短信　☐其他_____

请寄：北京市西城区百万庄南街1号　机械工业出版社　华章公司　计算机图书策划部收
邮编：100037　电话：（010）88379512　传真：（010）68311602　E-mail: hzjsj@hzbook.com