

# Debugging Strategies for .NET Developers

## .NET 开发人员 调试策略

- ❖ 用作者在实际工作中遇到的大量调试实例来介绍如何根除 Bug
- ❖ 介绍如何最大限度地利用 Visual Studio .NET 调试器
- ❖ 介绍如何在远程客户站点和本地计算机上调试应用程序
- ❖ 指导读者如何思考调试过程，使实际的调试工作更有效

(美) Darin Dillon 著  
张楚雄 刘 剑 译



清华大学出版社

# **.NET 开发人员调试策略**

(美) Darin Dillon 著

张楚雄 刘剑 译

**清华大学出版社**

北 京

## 内 容 简 介

在整个程序开发周期中,调试是一项繁琐但又必须执行的任务。.NET 中新增了许多调试技术和工具,本书通过实际工作中的大量示例介绍了一些根除 Bug 的方法,并介绍了如何利用 VS.NET 调试器在远程客户站点或本机上进行调试。此外,本书还谈到了在调试时需要注意的各项事宜,从而使开发人员在实际调试程序时能够更全面地考虑问题。

本书适用于 .NET 平台下的开发人员以及希望了解 .NET 调试技术的相关人员。

EISBN: 1-59059-059-7

Debugging Strategies for .NET Developers

Darin Dillon

Original English language edition published by Apress L. P., 2560 Ninth Street, Suite 219, Berkeley, CA 94710 USA. Copyright ©2003 by Apress L.P. Simplified Chinese-Language edition copyright ©2004 by Tsinghua University Press. All rights reserved.

本书中文简体字版由 Apress 出版公司授权清华大学出版社出版。未经出版者书面许可,不得以任何方式复制或抄袭本书内容。

北京市版权局著作权合同登记号 图字: 01-2003-7382

本书封面贴有清华大学出版社激光防伪标签,无标签者不得销售。

### 图书在版编目(CIP)数据

.NET 开发人员调试策略/(美)狄龙(Dillon, D.)著;张楚雄,刘剑译.—北京:清华大学出版社,2004.2

书名原文: Debugging Strategies for .NET Developers

ISBN 7-302-08071-2

I. N… II. ①狄…②张…③刘… III. 计算机网络—程序设计 IV. TP393

中国版本图书馆 CIP 数据核字(2004)第 008221 号

出 版 者: 清华大学出版社  
http://www.tup.com.cn  
社 总 机: 010-62770175

地 址: 北京清华大学学研大厦  
邮 编: 100084  
客户服务: 010-62776969

组稿编辑: 曹康  
文稿编辑: 崔伟  
封面设计: 康博  
版式设计: 康博  
印 刷 者: 北京市季蜂印刷有限公司  
装 订 者: 三河市化甲屯小学装订二厂  
发 行 者: 新华书店总店北京发行所  
开 本: 185×260 印张: 12.25 字数: 254 千字  
版 次: 2004 年 3 月第 1 版 2004 年 3 月第 1 次印刷  
书 号: ISBN 7-302-08071-2/TP·5840  
印 数: 1~4000  
定 价: 28.00 元

本书如存在文字不清、漏印以及缺页、倒页、脱页等印装质量问题,请与清华大学出版社出版部联系调换。联系电话: (010)62770175-3103 或 (010)62795704

# 目 录

|                                                                 |    |
|-----------------------------------------------------------------|----|
| 第 1 章 调试简介 .....                                                | 1  |
| 1.1 明确问题 .....                                                  | 1  |
| 1.1.1 三思而后行 .....                                               | 2  |
| 1.1.2 是否值得这样做 .....                                             | 2  |
| 1.1.3 提高调试技能 .....                                              | 3  |
| 1.2 .NET 新增的调试特性 .....                                          | 4  |
| 1.2.1 内存问题不再出现 .....                                            | 4  |
| 1.2.2 语言无关性 .....                                               | 5  |
| 1.2.3 远程调试和跨机调试 .....                                           | 6  |
| 1.2.4 从进程中分离 .....                                              | 7  |
| 1.2.5 ASP.NET .....                                             | 7  |
| 1.3 其他内容 .....                                                  | 8  |
| 第 2 章 调试过程中的 6 点建议 .....                                        | 9  |
| 2.1 在检查 Bug 时, 不要忽视在此之前出现的任何 Bug .....                          | 9  |
| 2.2 不要忘记最终目标是让程序正确执行——修正 Bug 只是手段,<br>并非最终目的 .....              | 11 |
| 2.2.1 当目标依赖许多子目标, 而这些子目标又依赖更多的子目标时 .....                        | 13 |
| 2.2.2 当进行下一步很困难时, 可寻找一种简单的办法 .....                              | 13 |
| 2.3 当代码在一种方式下正常运行, 而在另一种方式下出现故障时,<br>应将注意力集中在导致代码出现故障的方式上 ..... | 14 |
| 2.3.1 将注意力集中在不同点上 .....                                         | 15 |
| 2.3.2 当一些程序以某种方式运行时 .....                                       | 16 |
| 2.4 在没有合理的证据时, 不要怀疑问题出在硬件、API 或 OS 上 .....                      | 17 |
| 2.5 在一些测试计算机上不要安装调试工具——以免发生异常循环 .....                           | 19 |
| 2.5.1 在非开发环境下检验代码的工作情况 .....                                    | 20 |
| 2.5.2 确保能够在客户站点上调试您的产品 .....                                    | 21 |
| 2.5.3 为您的团队提供使用其他调试方法的实践机会 .....                                | 22 |
| 2.6 编码后, 立即在调试器上逐步验证所有代码 .....                                  | 22 |
| 2.6.1 忘记补充函数细节 .....                                            | 23 |
| 2.6.2 发现处理错误的代码隐藏着一个严重的问题 .....                                 | 23 |

|       |                           |    |
|-------|---------------------------|----|
| 2.6.3 | 把修改代码也当作一次测试, 但是偶尔也应该检查一下 | 24 |
| 2.6.4 | 错过优化代码的机会                 | 25 |
| 2.6.5 | 正确使用逐步调试                  | 26 |
| 2.7   | 小结                        | 26 |
| 第 3 章 | 制定计划                      | 28 |
| 3.1   | 蛮力调试                      | 28 |
| 3.2   | 3 个猜测                     | 30 |
| 3.2.1 | 还是不相信我吗? 让我证明给您看          | 31 |
| 3.2.2 | 构造一个测试案例                  | 33 |
| 3.2.3 | 聪明地使用猜测                   | 33 |
| 3.2.4 | 以前曾经见过与之类似的 Bug 吗         | 33 |
| 3.3   | 手机问题                      | 34 |
| 3.3.1 | 一旦做出假设, 我们就很可能做蠢事         | 35 |
| 3.3.2 | 收集信息——到何时一切才能正常运行         | 36 |
| 3.3.3 | 推理实现细节                    | 39 |
| 3.3.4 | 深入问题                      | 41 |
| 3.3.5 | 决定性的问题——故障的根源是什么          | 43 |
| 3.4   | 小结                        | 44 |
| 第 4 章 | 断言调试                      | 45 |
| 4.1   | 断言                        | 45 |
| 4.1.1 | 关于消息窗口的问题                 | 46 |
| 4.1.2 | 使用断言来启动调试器                | 47 |
| 4.2   | 我经常验证代码, 为什么还需要断言呢        | 48 |
| 4.2.1 | 找到出错点                     | 48 |
| 4.2.2 | 断言的部分优点                   | 50 |
| 4.3   | .NET 的调试和跟踪类              | 51 |
| 4.4   | 积极地使用断言                   | 53 |
| 4.5   | 断言性能瓶颈                    | 55 |
| 4.6   | 不要断言合理的情况                 | 57 |
| 4.7   | 当不能简单地使用断言时               | 58 |
| 4.7.1 | 关于 Windows 服务的断言          | 58 |
| 4.7.2 | 关于 ASP.NET 页面和 Web 服务的断言  | 59 |
| 4.7.3 | 关于远程对象的断言                 | 61 |
| 4.8   | 使用 TraceListener 定制断言     | 61 |

|                                       |            |
|---------------------------------------|------------|
| 4.8.1 启动定制的断言 .....                   | 63         |
| 4.8.2 在服务中使用定制的断言 .....               | 64         |
| 4.9 小结 .....                          | 65         |
| <b>第 5 章 用日志调试 .....</b>              | <b>66</b>  |
| 5.1 Printf 调试 .....                   | 66         |
| 5.2 程序日志 .....                        | 67         |
| 5.2.1 日志中应该包含的信息 .....                | 67         |
| 5.2.2 使日志易读 .....                     | 71         |
| 5.3 .NET 对记录的支持 .....                 | 76         |
| 5.3.1 .NET 中的侦听器 and 开关 .....         | 78         |
| 5.3.2 XML 配置文件 .....                  | 80         |
| 5.4 Windows 事件日志 .....                | 82         |
| 5.5 小结 .....                          | 84         |
| <b>第 6 章 ASP.NET 调试和 SQL 调试 .....</b> | <b>85</b>  |
| 6.1 ASP.NET 和调试器 .....                | 85         |
| 6.1.1 连接调试器 .....                     | 87         |
| 6.1.2 远程调试 .....                      | 89         |
| 6.1.3 分离调试器 .....                     | 89         |
| 6.2 ASP.NET 中的日志 .....                | 90         |
| 6.2.1 ASP.NET 跟踪数据 .....              | 91         |
| 6.2.2 启用 ASP.NET 跟踪 .....             | 96         |
| 6.3 调试 SQL 存储过程 .....                 | 100        |
| 6.3.1 直接逐步执行存储过程 .....                | 100        |
| 6.3.2 在应用程序中调试存储过程 .....              | 102        |
| 6.3.3 SQL 远程调试 .....                  | 102        |
| 6.4 小结 .....                          | 103        |
| <b>第 7 章 调试远程客户站点 .....</b>           | <b>104</b> |
| 7.1 调试远程客户问题如此困难的原因 .....             | 104        |
| 7.1.1 配置问题 .....                      | 105        |
| 7.1.2 您必须设计再现 Bug .....               | 106        |
| 7.2 为什么要问我这么多问题? 只要修正它不就行了 .....      | 107        |
| 7.2.1 客户的期望 .....                     | 107        |
| 7.2.2 为什么不能实现客户的愿望 .....              | 108        |
| 7.3 自动诊断实用程序 .....                    | 108        |

---

|              |                                     |            |
|--------------|-------------------------------------|------------|
| 7.3.1        | 自动收集数据 .....                        | 109        |
| 7.3.2        | 诊断实用程序的设计决策 .....                   | 109        |
| 7.4          | 帮助您观察客户再现 Bug 的第三方工具 .....          | 111        |
| 7.4.1        | 亲眼观察 Bug .....                      | 111        |
| 7.4.2        | 控制用户的电脑 .....                       | 112        |
| 7.5          | 对于非常严重的 Bug, 可以让开发人员参与技术支持工作 .....  | 112        |
| 7.5.1        | 参与的人员越多, 遗漏的信息越多 .....              | 113        |
| 7.5.2        | 开发人员应该对客户说的话 .....                  | 113        |
| 7.6          | 构建一个带有更多日志的新版本——以及验证它是否被安装的方法 ..... | 114        |
| 7.7          | .NET 的安全性 .....                     | 115        |
| 7.7.1        | 限制访问 .....                          | 116        |
| 7.7.2        | CAS 的工作方式 .....                     | 117        |
| 7.7.3        | 处理 SecurityExceptions .....         | 119        |
| 7.7.4        | 根据每个程序集授予权限 .....                   | 120        |
| 7.8          | 小结 .....                            | 122        |
| <b>第 8 章</b> | <b>多线程调试 .....</b>                  | <b>123</b> |
| 8.1          | 多线程的概念 .....                        | 123        |
| 8.2          | 在调试器中查看线程 .....                     | 125        |
| 8.3          | 常见线程问题的原因 .....                     | 126        |
| 8.3.1        | 竞争条件 .....                          | 127        |
| 8.3.2        | 死锁 .....                            | 130        |
| 8.3.3        | 资源匮乏 .....                          | 131        |
| 8.4          | 调试线程问题 .....                        | 133        |
| 8.4.1        | 调试竞争条件 .....                        | 135        |
| 8.4.2        | 调试死锁 .....                          | 139        |
| 8.5          | 关于线程的思考 .....                       | 145        |
| 8.6          | 小结 .....                            | 145        |
| <b>第 9 章</b> | <b>错误跟踪程序 .....</b>                 | <b>146</b> |
| 9.1          | 需要错误跟踪软件的原因 .....                   | 146        |
| 9.1.1        | 您不知道的细节 .....                       | 147        |
| 9.1.2        | 错误跟踪程序的概念 .....                     | 148        |
| 9.2          | 开发工作流程 .....                        | 150        |
| 9.2.1        | 常见的工作流程问题 .....                     | 151        |
| 9.2.2        | 执行工作流程 .....                        | 152        |

|                                       |            |
|---------------------------------------|------------|
| 9.2.3 处理工作流程中的反弹 .....                | 153        |
| 9.3 搜索和报告 .....                       | 154        |
| 9.3.1 重复的 PR .....                    | 154        |
| 9.3.2 为解决当前 Bug 而从以前的 Bug 中查找线索 ..... | 155        |
| 9.3.3 PR 统计报告 .....                   | 156        |
| 9.4 和队员交流 .....                       | 158        |
| 9.4.1 与其他开发人员合作 .....                 | 159        |
| 9.4.2 与测试人员合作 .....                   | 160        |
| 9.4.3 与技术支持部门合作 .....                 | 161        |
| 9.5 了解您的工具 .....                      | 162        |
| 9.6 小结 .....                          | 163        |
| <b>第 10 章 源代码管理调试 .....</b>           | <b>164</b> |
| 10.1 源代码管理介绍 .....                    | 165        |
| 10.1.1 使用 Visual SourceSafe .....     | 165        |
| 10.1.2 允许开发人员协同工作 .....               | 166        |
| 10.1.3 查看文件变化的历史记录 .....              | 168        |
| 10.1.4 取回某一版本或者分离一个副本 .....           | 170        |
| 10.1.5 防止意外删除文件 .....                 | 171        |
| 10.2 用 SourceSafe 进行调试 .....          | 171        |
| 10.2.1 用修改历史进行调试 .....                | 172        |
| 10.2.2 在分支管理中修正 Bug .....             | 177        |
| 10.3 习惯于使用 SourceSafe .....           | 182        |
| 10.4 小结 .....                         | 182        |
| <b>结束语 .....</b>                      | <b>184</b> |

# 第1章 调试简介

您听说过一个程序员在海边溺水身亡的故事吗？一群水兵就在他附近，但是没有一个人救助他，因为他们搞不懂这个程序员为什么在喊“F1, F1!”

——佚名

面试时，我最喜欢问的一个技术问题是：“假设您编写了一个编译器，但有人告诉您有一个 I/O 库函数(System.Console.WriteLine、cout、System.Out.WriteLine 等)无法工作。您将怎么办？”如果有人这样回答：“哦，我会启动调试器，并逐步执行代码，”那么您的团队可能不会聘用这种人。用调试器来逐步遍历代码太耗费时间，因此这只能作为最后一个手段。

但令人惊讶的是，有许多开发人员会选择将遍历代码作为解决这个 Bug 的首选方法。其实有很多更好的方法完全可以解决这个 Bug。举例来说，我会很高兴地听到一位面试官这样回答：“嗯，首先我将了解所谓的‘无法工作’到底是什么样的情况。它会崩溃？输出错误？还是其他什么结果？它是持续出现还是偶尔发生？在所有的机器上都这样，还是只在某台机器上出现这种情况？是只对这一类输入能正常工作，但对其他输入才发生故障吗？”总之，要想修复一个 Bug，您首先要真正了解它。

本书针对使用 Microsoft 的 .NET Framework 编写的应用程序介绍了一些调试策略。首先我们分析一下如何更精彩地回答这个面试问题。

## 1.1 明确问题

调试过程中最重要的是我们要理解这个问题。在启动调试器之前，我们要通过大量提问来明确症结所在：“您肯定正确使用了函数吗？您能发给我一份演示这个问题的测试程序吗？您使用什么版本的操作系统？Bug 是不是只出现在多线程代码中，或是出现在使用非默认 I/O 驱动程序的系统上，还是有其他的异常发生？”

运气好的话，其中的某一个会提供有用的线索——或者使工程师意识到：“哦，编写代码的时候我没考虑到多线程；我相信问题就出在没有使用信号量(semaphore)来保护变量……”

在弄清楚问题之后，现在我们可以用调试器遍历代码了吗？目前还不可以。经验丰富的工程师会接着建议尝试以下操作：“我要检查这个 Bug 是不是由我们已经发现并

修正过的某个 Bug 引起的，我将通过追踪数据库来检查 Bug，或者请同事来看是否熟悉这个问题——也许其他人修正过它。”

### 1.1.1 三思而后行

如果以上办法都不起作用，那么可以遍历代码了吗？可以，但如果我们先尝试再现这个 Bug 可能会更好。毕竟如果连 Bug 都无法再现，要调试器给我们答案就更不可能了。一位出色的面试者会这样回答：“我会编写一个非常简单的程序来测试函数，看 Bug 是否会出现。如果没有，我将设法找出我的测试程序与客户端的程序有何不同，因为这可以提供线索，告诉我应该在何处查找 Bug。”

当然，如果有人注意到大多数 I/O 库函数都可能是用低级汇编语言编写的，他将得到我们的奖励。所以，既然我们是在编写编译器，那么这个 Bug 就未必(但也可能)是我们的问题，因为汇编语言代码是不需要编译的。因此，也许我们应该首先检验汇编程序或者函数代码本身。

无论如何，接下来我们应该花几分钟考虑收集到的所有信息。脑海中很可能会突发一个灵感，“嗨，我打赌它是 XYZ！看看我将如何证明它。”如果推测正确，我们就不需要再做乏味的调试工作。当然，不可能总是那么幸运，我们经常还是要靠调试器来完成工作。但至少我们已经清楚了 Bug 是什么样子，什么时候会产生 Bug，什么时候不会产生 Bug。甚至可能对 Bug 产生的大致区域已经有了一两种猜测。然后，我们可以从这些区域开始搜索，而不需要从第一行开始逐行遍历代码。

### 1.1.2 是否值得这样做

听起来调试 Bug 需要进行很多工作，幸好我们还没有这样做。做这么多工作值得吗？当然，比起刚开始时，我们对这个 Bug 将有更多的了解，可是这值得花费这么多额外的时间吗？许多常见的 Bug 实际上很容易解决，并不需要深入分析。有些 Bug 要花几天时间才能找到，不过大多数 Bug 只是一些简单的输入错误，只要一打开调试器就能发现。有些 Bug 可能是因为你忘了递增循环计数器。综上所述，对出现的每个 Bug 都花费额外的时间处理是否值得呢？为什么有些时候我们不能直接调试呢？

首先，您确实需要提高调试技能，以便为无法使用调试器时做准备。具体是什么时候呢？迟早会有客户报告，在他(她)机器上产生的 Bug 是您无法再现的。有时候，您或许可以使用 Visual Studio .NET 的远程调试功能，但这个功能有一些严格的限制，下文我们将会谈到。应该凭借逻辑思维来查找这些不可再现的 Bug，否则您将疲于应付许多愤怒的客户，再往后就不会有生意上门了。

但是，这并不是处理 Bug 问题的惟一途径。在开始调试之前花时间思考到底值不值得这么做？我们现在给出这个问题的正确答案：有时候不值得这样做。大部分有关调试的参考资料都强调他们的 12 步骤法是有效的，但事实是没有什么能够永远有效。有时候将额外的工作花在从多角度处理 Bug 上确实是在浪费时间。如果开发人员在开始调试之前多做一些计划，那么他们将会做得更好，不过如果说大量的调试从来都是不适用的，那也是在说谎。

简单的错误应该用简单的方法解决。要分辨哪些 Bug 需要深究，而哪些可以直接动手处理，经验和直觉是最好的方法，而积累经验和直觉的最好办法是学习他人的经验。研究此类调试经验是本书的主要目的。

### 1.1.3 提高调试技能

许多有关调试方面的参考书都把重点集中在晦涩难懂的调试工具上：如何使用核心调试器、如何读取 .NET 中类似汇编语言的微软中间语言 (MSIL) 以及如何利用寄存器 (register) 显示窗口。不过，如果离开实际的应用示例，这些晦涩难懂的诀窍是毫无价值的。于是，工程师最后会得到一份关于 Bug 难以解决的报告——当然，您可以看到所有的寄存器值并读取想要的 MSIL 代码，可是之后该怎么办？如果您的团队花了两周时间都无法找出该 Bug，那又如何跟踪这个 Bug 呢？怎样才能捕捉到这个无法在您的计算机上再现的 Bug 呢？

Microsoft 的 .NET 激励开发人员不断提高他们的调试技能。Visual Studio .NET 不仅具有简化调试任务的新特性，还提高了开发人员的工作效率，使开发人员可以编写更多的代码。还记得 Visual C++ 中的 MFC (微软基本类) 消息映射机制和对话框数据交换 (DDX) 多么让人沮丧吗？还记得在 Visual Basic 中调用组件对象模型 (COM) 对象的所有管道 (plumbing) 代码是多么令人懊恼吗？.NET 的优点在于它免除了大多数令人头疼的事情——现在您可以专注于业务逻辑的代码，而所有的图形用户界面 (GUI) 和组件都会按照您在第一次尝试中所期望的方式正常工作。

本书假定您已经了解了一些基本工具，使用过调试器并知道如何设置断点。本书还假定您不需要复习编程语言，而是着眼于我们所见的实际 Bug、用于解决这些 Bug 的调试技巧，以及我们所公认的最合理方法。下面我们来探讨几个遭遇过的 Bug 实例及其解决步骤，但这些 Bug 都很难跟踪。根据获得的信息，可以分析当时在思考什么，从中学到了什么，以及 Bug 的真正起因。总之，我们将讨论如何成为一名优秀的调试人员。

## 1.2 .NET 新增的调试特性

如果是刚开始使用.NET,那么您会发现.NET 开发环境在调试和编写新代码方面有许多优点。您可曾因为 MFC 和 ATL(动态模板库)离开向导后无法进行实际应用而感到沮丧?您可曾为花费整整一周时间来跟踪代码中的内存泄漏而头痛不已?或者觉得这些 Windows API 像是由 100 个人随意拼凑在一起的,而这些人彼此之间明显没有任何沟通?所有这些问题在.NET 中都得到了解决。

您可曾因为不得不去处理 VARIANT 类型和 SAFEARRAY 类型而感到非常苦恼?这些类型在 Visual Basic 中看起来很简单,但在其他编程语言中使用却要耗费大量工作。如果您是一位 VB 程序员,您可曾渴望拥有多线程、对象继承性以及可以直接访问 C++特有的 Windows API 函数?事实上,为什么编程语言之间首先就有这么大的区别呢?为什么我们不能用 JScript 编写一个简单的函数来调用传统的 C++和 COBOL 函数,从而无需编写大量的“管道”代码呢?所有这些问题在.NET 中也得到了解决。

在学习本书其他部分之前,我们先就以下这些对调试工作特别有益的新特性作一个快速介绍。

- 内存问题不再出现
- 语言无关性
- 远程调试和跨机调试
- 从进程中分离
- ASP.NET

### 1.2.1 内存问题不再出现

.NET 的核心是称为公共语言运行库(Common Language Runtime, CLR)的虚拟机。.NET 编译器生成一种可以被即时(Just-In-Time)解释的中间语言,称为 MSIL(微软中间语言),这样就不需要将代码编译为汇编语言。每个人首先都会问该虚拟机是否会影响性能?的确,当前版本的.NET 公共语言运行库会影响到速度,如果有人再说别的什么就是在骗人了(ASP.NET Web 页例外,它比传统的 ASP 还要快)。幸运的是,这个性能影响对于多数应用程序来说都可以忽略不计。此外,您不能只看到性能损失而无视 CLR 带来的额外优点。这些优点有很多,其中最重要的两个是类型安全(type-safety)和垃圾回收(garbage collection)。

JScript 和 VB 程序员基本上无须考虑内存管理方面的问题,但 C++ 程序员就要面对众多的内存问题。您可曾在使用 COM 的 `AddRef()` 和 `Release()` 方法时有过误算? 您可曾超出数组定义的范围? 您可曾访问过已删除对象的方法? 通过使用 .NET, 就不必为低级内存管理而厌烦。您可以自由分配对象, 当不再需要它们时就会将其自动删除。而且您也不用担心内存崩溃——超出数组范围依然是个错误, 但至少错误会在该 Bug 出现时持续显示为异常, 而不是在很久之后显示为全局保护错误(Global Protection Fault)。

最好的情况是, 大部分 .NET 语言自动支持类型安全和垃圾回收功能, 因而不需要您做任何事情。无须设法使对象在不需要时自动释放——CLR 会自动处理所有的低级细节。在特殊情况下, 您可以使用 C# 中的 `unsafe` 关键字来部分关闭自动内存管理功能。不过, 您很可能会发现类型安全和垃圾回收如此有用, 以至于再也离不开它们。

## 1.2.2 语言无关性

我们无法知道 .NET 到底使用了哪种语言。微软的确选择了 C# 和 VB.NET 作为大多数 Microsoft .NET 代码示例的编程语言, 不过任何面向对象的语言都可以插入 .NET Framework 并成为其中的一部分, 而且这种语言可以与其他语言一样有效访问所有 .NET API 库。与 Visual Studio 的前一个版本不同, VS.NET 对所有语言都使用一种集成开发环境(Integrated Development Environment, IDE), 如 C++、VB、JScript、C# 和 HTML 等。开发人员不需要为了 Visual Basic IDE 学习一组命令, 而为了 Visual C++ IDE 再学另外一组命令。

您甚至可以同时使用多种语言。有这样一个示例使得 .NET 给人的印象最深刻: 在 VB .NET 中先编写一个类, 然后编写一个 C# 对象来继承这个 VB 类, 并重写其中一个方法, 之后又使用其他语言(例如 C++) 编写代码来实例化这个 C# 对象, 并在没有添加任何管道代码的情况下调用它的方法。您可以用一种语言编写应用程序的绝大部分代码, 不过如果您想使用其他语言编写的样本代码, 那也没有关系。您只需直接添加那段代码, 两种语言就能自动地无缝交互。

事实上, 甚至可以跨语言进行调试。在 C++ 代码中, 可以在调用 C# 对象的地方设置断点。可以像对其他常规函数一样逐步调用这个 C# 方法, 而且调试器将自动转到正确的代码, 就好像始终在调试 C# 一样(如图 1-1 所示)。如果这个方法调用了某个 VB.NET 的方法, 调试器同样也会自动转到此处。在 .NET 出现之前, 调试跨语言的程序是可行的, 不过非常困难。但现在, 跨语言调试十分简单。

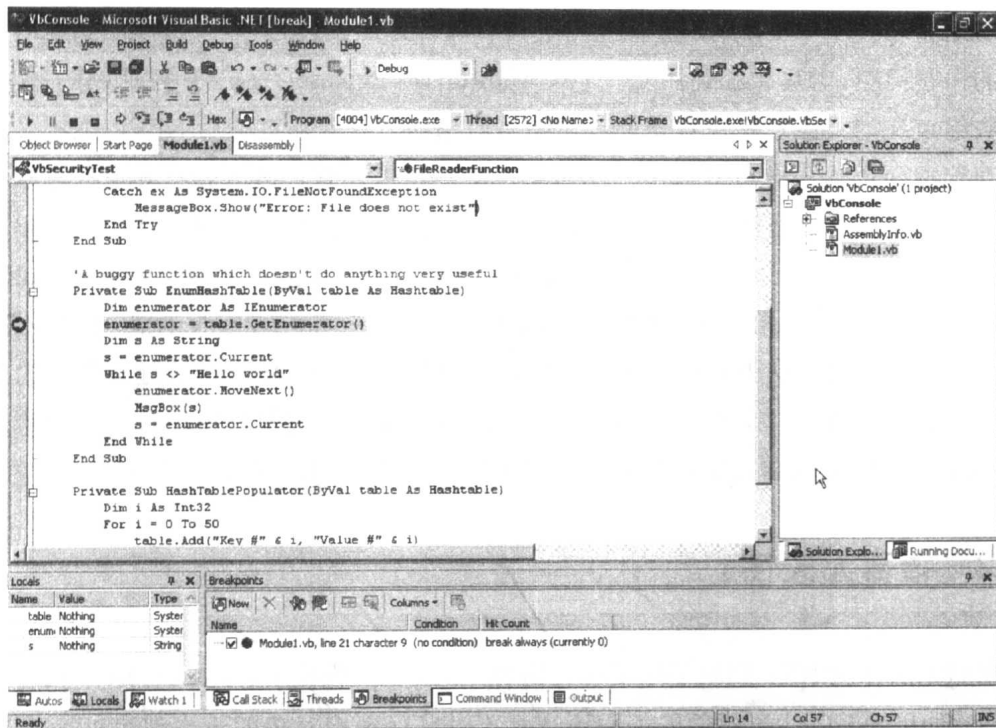


图 1-1 Visual Studio .NET 调试器的工作界面

### 1.2.3 远程调试和跨机调试

只有少数开发人员知道 Microsoft Visual C++ 6.0 配有远程调试器。大多数人从未听说过这件事，因为这个远程调试器使用起来非常困难。为了使用它，首先必须在远程机器上安装特定的调试系统动态链接库(DLL)。但是在何处能找到这些 DLL 呢？它们在 Visual Studio 光盘上的某个目录下吗？多数人从未找到过它们。此后，您还需要运行某些复杂的配置细节。不过最大的障碍还是第一个步骤。

在 Visual Studio .NET 中，Microsoft 大幅度地改进了远程调试器。您仍然需要在远程机器上安装那些 DLL，但现在简单多了。将 VS.NET 光盘放入驱动器，出现的第一个画面就会问您是安装完整的 Visual Studio .NET，还是只安装远程调试组件(如图 1-2 所示)。

一旦解决了这个问题，其他问题就好办了。您必须留意一些限制，但是它们都不难理解。这个功能的出众之处就是跨机调试。假定您在正确的位置安装了所有的调试符号文件，那么 VS.NET 调试器就允许您遍历机器上的代码，然后逐步调用远程机器上的函数。调试器将“神奇般地”逐步执行该函数，看起来就好像函数在本地一样。如果不是通过调试器来区分，本地和远程的机器没有任何不同。

我们将在第 6 章详细阐述远程调试器。

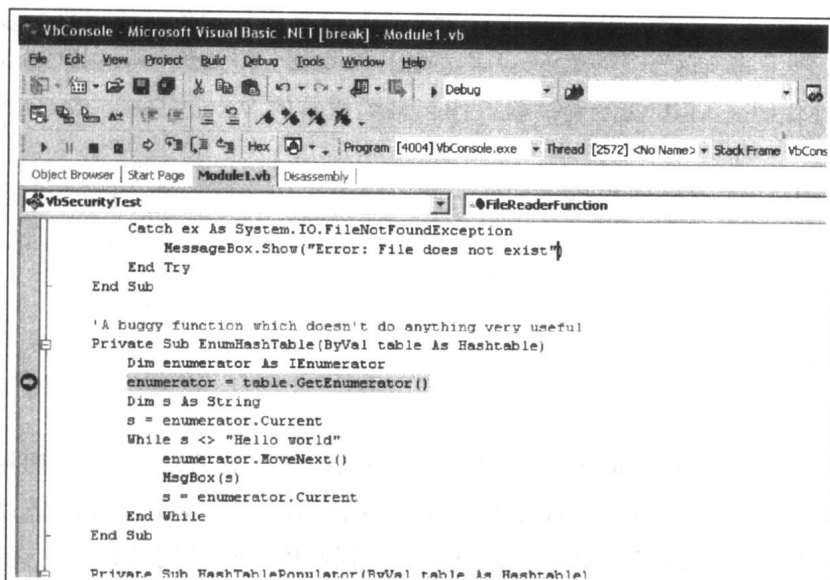


图 1-2 安装远程调试组件

## 1.2.4 从进程中分离

将调试器连接到运行中的进程上是一个很好的办法。如果程序在某人的计算机上发生了故障，那么只需要连接上调试器并设置断点就可以进行局部调试。不过，在 Visual Studio 6.0 中有一个问题，VS6 支持连接调试器的功能，但不支持分离功能。所以您可以和进程连接并进行调试，不过一旦完成调试，关闭调试器会导致这个进程关闭。而且 VS6 中没有什么变通的方法能解决这个问题。

Visual Studio .NET 增加了一个很重要的新特性——现在调试器既可以连接进程，也可以从进程中分离。这种特性在调试 ASP.NET Web 页时最经常使用。如果您只想观察调试器中 ASP.NET Web 服务的行为，但又不想改变任何设置或停止服务，这时就可以使用分离调试器的新特性。它允许使用全新的无干扰调试。

## 1.2.5 ASP.NET

在 ASP.NET 中，Microsoft 重写了动态服务器页(Active Server Pages, ASP)，使其在各个方面都变得更加完善。单单 ASP.NET 就可以写一整本书，不过总的来说，ASP.NET 是使用 .NET 最无法令人抗拒的理由之一。这不仅是因为它比 ASP 更快捷和扩展性更强，同时它需要的代码也更少。您可以随意迁移您的站点，而无须对现有的 ASP 页面做任何修改，因此升级到 ASP.NET 将会非常容易。

和 ASP 不同，ASP.NET 可以使用与 VB、C# 或 C++ 相同的调试器来进行调试。您可曾对 ASP 使用过 Microsoft 的脚本调试器(Script Debugger)? 您是否感到被欺骗了?

通过使用 ASP.NET, 可以在 Web 页上设置断点, 检查变量的值, 还可以逐步执行函数和跳出函数。您的页面使用了 .NET 或者 COM 组件吗? 没问题——调试器同样可以无缝地逐步执行代码。ASP.NET 调试器工作起来同正常的 Visual Basic 或 C# 调试器没有什么区别。

其实, ASP.NET 甚至还提供了其他一些很吸引人的调试特性。例如, 在 Web 页中添加一条线将启动大量的自动性能和诊断日志。长话短说, 您以前用于调试 ASP 的所有技术依然有效, 同时还有很多新功能可以使用。

## 1.3 其他内容

上述内容只不过是 Microsoft .NET 新增调试特性的概括性介绍。我们将在本书中讨论许多工具、技术和示例: .NET 宣称的所有内容和新 SQL 调试器的日志功能, 还包括调试远程客户站点的技术, 以及如何在没有源代码的情况下调试系统。在第 2 章中, 我们首先对从多年的调试经历中获得的难得教训开始分析。

## 第2章 调试过程中的6点建议

有位作家打算创作一部重要的著作，他希望这部著作备受读者青睐并能给读者带来喜怒哀乐。他最终实现了这个夙愿，现在他在为 Microsoft 编写错误报告。

——转寄 E-mail，大概在 2001 年

通过阅读 Microsoft 的产品手册，您可以了解有关调试方面的知识，而登录到网站 <http://msdn.microsoft.com> 则可以获得有关调试方面的最新技术。但是，有些调试技术只能通过自己的经验积累来获得，或者由身边经验丰富的同事告诉您。在开始调试前，必须掌握一些原则，这些原则将有助于您更有效地调试相关的问题。

不应该把注意力只放在 .NET 技术上，这一点非常重要。例如，Visual Studio .NET 现在能为 SQL 服务器存储过程提供功能强大的调试工具，我们将在第 7 章中详细介绍这些工具的使用方法。这些调试器更容易跟踪存储过程中的问题。然而，在有些情况下您将发现，调试存储过程也无法解决问题。通常，您应该先试用一种较简单的方法。在讨论 .NET 的新技术之前，了解如何准确定位问题无疑是很有帮助的。

本章我们将介绍调试过程中的 6 点建议：

- 在检查 Bug 时，不要忽视在此之前出现的任何 Bug。
- 不要忘记最终目标是让程序正确执行——修正 Bug 只是手段，并非最终目的。
- 当代码在一种方式下正常运行，而在另一种方式下出现故障时，应将注意力集中在导致代码出现故障的方式上。
- 在没有合理的证据时，不要怀疑问题出在硬件、API 或 OS 上。
- 在一些测试计算机上不要安装调试工具——以免发生异常循环。
- 编码后，立即在调试器上逐步验证所有代码。

### 2.1 在检查 Bug 时，不要忽视在此之前出现的任何 Bug

设想您正在检查一个生产小配件的装配线，假定整个过程包括 10 个步骤：第 1 步是切割金属，第 2 步将边缘进行平滑处理。有一天您被告知所生产的产品有缺陷。仔细查看整个生产过程后，发现在第 6 步时产品被挤压变形。于是将工程师找来，让他对第 6 步进行全面检查以找出 Bug。然而，接下来您注意到第 5 步也同样有问题。在第 5 步中，配件被倒转放在传送带上，因此导致产品在到达第 6 步进行处理时不能按照预定的方式执行，因此认为：“毫无疑问第 5 步有 Bug，必须解决这个 Bug，但是刚才我们将注意