

The Developer's Guide to Debugging

软件调试实战

[德] Thorsten Grötker Ulrich Holtmann
Holger Keding Markus Wloka 著

赵俐 译



人民邮电出版社
POSTS & TELECOM PRESS

软件调试实战

The Developer's Guide to Debugging

软件调试是软件开发中最令人苦恼的环节。反复思考每个假设，反复斟酌从需求到实现的每个步骤，将耗费大量时间。最糟糕的是，调试根本无法预测，我们永远无法知道修复一个 bug 需要多长时间，甚至根本不知道是否还能修复它。然而，如果采用系统的方式并配备合适的工具，调试也会充满乐趣，成功的调试就像解出难题、猜出谜语或破获奇案一样令人激动。这本书就能帮你实现这一惊天逆转。

本书是4位深谙软件调试之道的资深开发人员的实战经验总结，不仅讲述了简单的源代码调试，还涵盖了各个领域的最常见的实际问题，包括程序链接、内存存取、并行处理和性能分析。最后几章讨论了静态检查器，介绍了一些较好地运用了调试技巧的代码编写方法。书中讲述的调试技术不仅可以用于C/C++程序，还可以用于其他语言编写的程序。

- Amazon 五星图书
- Synopsys 公司专家的调试经验总结
- 软件调试权威指南



本书相关细心请访问：图灵网站 <http://www.turingbook.com>

读者/作者热线：(010)51095186

反馈/投稿/推荐信箱：contact@turingbook.com

分类建议 计算机 / 软件工程

人民邮电出版社网址 www.ptpress.com.cn



ISBN 978-7-115-21885-8



ISBN 978-7-115-21885-8

定价：45.00元

The Developer's Guide to Debugging

软件调试实战

[德] Thorsten Grötker Ulrich Holtmann 著
Holger Keding Markus Wloka

赵俐 译

人民邮电出版社
北京

图书在版编目（CIP）数据

软件调试实战 / (德) 格勒特克 (Grötker, T.) 等著;
赵俐译. —北京: 人民邮电出版社, 2010.2
(图灵程序设计丛书)
书名原文: The Developer's Guide to Debugging
ISBN 978-7-115-21885-8

I. ①软… II. ①格…②赵… III. ①软件 - 调试
IV. ①TP311.5

中国版本图书馆CIP数据核字 (2009) 第227533号

内 容 提 要

本书主要讲述 C/C++ 程序的调试和分析, 书中的调试技术也可以应用于其他语言编写的程序。本书在讲述简单的源代码分析和测试的基础上, 讲述了现实的程序中经常遇到的一些问题 (如程序链接、内存访问、并行处理和性能分析) 并给出了解决方案。

本书适合软件开发人员、调试人员阅读和使用。

图灵程序设计丛书

软件调试实战

◆ 著 [德] Thorsten Grötker Ulrich Holtmann
Holger Keding Markus Wloka
译 赵 俐
责任编辑 傅志红
执行编辑 杨 爽
◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街14号
邮编 100061 电子函件 315@ptpress.com.cn
网址 <http://www.ptpress.com.cn>
北京隆昌伟业印刷有限公司印刷
◆ 开本: 800×1000 1/16
印张: 12.75
字数: 302 千字 2010年2月第1版
印数: 1~3 000册 2010年2月北京第1次印刷
著作权合同登记号 图字: 01-2009-4819号
ISBN 978-7-115-21885-8

定价: 45.00元

读者服务热线: (010)51095186 印装质量热线: (010)67129223

反盗版热线: (010)67171154

版 权 声 明

Translation from the English language edition: *The Developer's Guide to Debugging*, by Thorsten Grötker, Ulrich Holtmann, Holger Keding, Markus Wloka.

Copyright © 2008, as a part of Springer Science+Business Media.

All Right Reserved.

本书简体中文版由Springer Science+Business Media授权人民邮电出版社独家出版。未经出版者书面许可，不得以任何方式复制或抄袭本书的任何部分。

版权所有，侵权必究。

译者序

软件并不是从一开始就完美无缺，它包含各种各样的缺陷和错误，因此需要不断调试，找到问题，然后修改代码。好的调试技巧可以帮助我们创建高质量的软件，因此在软件开发项目中，花时间进行全面而细致的调试工作是非常值得的。

软件调试早已经不是什么陌生的字眼了，大多数开发人员都或多或少做过一些调试工作，例如源代码调试、内存调试、性能调试等。然而要在实际工作中出色地完成调试工作却不容易，因为调试需要耗费大量时间，而且具有很大的不确定性。在很多情况下，我们很难预知修复一个bug需要多长时间，甚至根本不知道能否修复它。由于这些原因，我们必须采用系统性的调试方法并配备正确的工具，这正是本书的核心。

本书主要以C和C++程序为例详解如何通过调试来分析和改进程序代码。书中揭示了大量的软件缺陷并介绍了各种调试技术，同时给出了如何编写可调试代码的建议。相信所有开发人员，特别是调试人员，都能够从本书中获益匪浅。

翻译从来都是一项充满挑战和趣味性的工作，本书的翻译更是如此，每当我字斟句酌地思考应该如何用最准确简练的语言表达几位作者思想的时候，总是能够从他们的智慧和洞察力中得到启示，从而对调试有了越发深刻的认识，这使得北京炎热的夏天也成了最美好的时光。我把这本书当成了我的老师，也希望它能够成为各位读者的老师，把几位作者的思想精华传授给你们。

最后，衷心感谢北京双飞软件公司的同仁们在翻译中给予的帮助，以及在一些高级调试主题方面的技术支持。由于译者水平有限，在翻译过程中难免会出现一些错误，恳请读者批评指正。

2009年9月

序

在所有软件开发工作中，调试或许是最令人苦恼的。调试工作极易受到指责，因为技术失败即意味着做人失败，而且矛头直指调试人员，昭示出他们所犯下的错误。由于必须反复思考每个假设，反复斟酌从需求到实现的每个步骤，因此调试将耗费大量时间。最糟糕的是，调试还无法预测，我们永远无法知道修复一个bug需要多长时间，甚至根本不知道是否能够修复它。

问一下开发人员他们生活中最沮丧的时刻，大多数回答将与调试有关。也许现在正是深夜11点，你仍在忙着调试，正当对程序进行走查时，你的家人打电话给你，问你到底什么时候才能回家，而你只希望尽快放下电话，因为好不容易得到的观察结果和推断正要从脑中溜走。此时，你可能最后有两种选择：一是重新调试，二是过后再试图重修旧好。据我个人估计，调试是导致程序员离婚的第一大原因。

然而，调试也蕴含着乐趣，就像解出难题、猜出谜语或破获谋杀案一样令人激动，但前提条件是必须采用系统性的方式并配备正确的工具。这正是本书的用武之地。本书四位作者直接与那些固持己见的开发者对话，直截了当地提出解决调试问题的建议，并给出了真正快速的解决方案。无论是解决内存问题，调试并行程序，还是处理工具链引入的问题，本书都能够提供“急救措施”，书中的建议都是经过反复尝试和验证的。

如果我最初开始调试程序时就能有这样一本书，该多好啊！我想我会屏息注视，看看这些调试工具将带给我什么惊喜，而且采纳书中的建议，必然会节省大量手工调试的时间，并可将这些时间投入到其他工作中。譬如说，可以使代码更可靠，这样最后可能根本不必做任何调试了。

当然，这是专业编程的长期目标，即从一开始就编写正确的代码，通过某种确认或验证方法来杜绝所有错误（或至少检测到错误）。目前，断言和单元测试已经在提高程序可信度方面提供了很多帮助。未来可能会有一些用于行业级系统的成熟验证方法。我们现在尚未实现这样的方法，这可能需要很多年，而且当达到这个目标时，所实现的方法肯定不会适用于现在的编程语言。在处理当今的程序，特别是那些C和C++程序时，我们仍将在调试上花费一定时间——这正是本书的宝贵价值所在。

Andreas Zeller
2008年春于萨尔兰德大学

前　　言

在创作本书时，我们几位作者都就职于一家生产软件产品的技术公司，只是所属的项目或产品不同。然而，我们都曾经被派去支持客户和同行解决C和C++程序的调试问题，这是我们软件工程工作的一部分，因为我们开发了一些用来帮助用户编写优化仿真程序的工具，或者只是因为我们凑巧开发了调试工具。在这个过程中，我们一次又一次地重复着相同的基本技巧，因为在调试方面没有好的图书可供参考。

现在就不一样了。

本书的网站

我们建立了一个网站来补充本书的内容，网址为<http://www.debugging-guide.com>，站点上列出了软件调试主题的最新参考资料，包括工具、图书、杂志、研究论文、会议、教程和Web链接。站点还提供了本书中的示例和更多可供下载的资料。

致 谢

没有众人的帮助，本书将无法出版。

首先感谢来自Springer的Mark de Jongh先生，他总是耐心地鼓励和支持我们创作本书，他的耐心经历了我们几个人无数次的考验。

还要感谢许多人，其中包括我们在Synopsys公司的同事，感谢他们源源不断地提出软件调试领域的问题，并教给我们攻克难关的技巧。这些帮助正是本书萌芽的温床。在此，请恕我们无法完整地列出所有人员的名单。

这里要特别提一下Andrea Kroll，因为她是第一个要求我们编写用于调试仿真的结构化方法的人，还要特别感谢Roland Verreet给予我们的鼓励和他在营销方面的真知灼见。另外感谢Joachim Kunkel和Debashis Chowdhury提供的支持。

软件有bug，书也不例外，特别是早期的草稿。感谢那些勇敢提出意见的人们，他们使得本书的质量和可读性有了很大提高。感谢以下人员在此过程中做出的贡献（以姓氏字母排序）：Ralf Beckers、Joe Buck、Ric Hilderink、Gernot Koch、Rainer Leupers、Olaf Scheufen、Matthias Wloka和Christian Zunker。

另外，感谢Scott Meyers在如何组织各章内容以及如何呈现关键材料方面提出的建议。

还要感谢Andrea Höltner，她在反复审阅本书的过程中写下了很有见地的修改意见。

同时感谢Mike Appleby、Simon North和Ian Stephens，他们帮助我们将大量零散的信息组织为较易理解的内容，而且对我们的很多英语表述错误给予了纠正。本书剩下的任何错误和缺点当然归咎于我们自己。

最后，如果没有我们各自家庭的不懈支持，本书也将无法出版。

感谢你们！

目 录

第 1 章 谁编写软件，谁制造 bug （为什么需要本书）	1
第 2 章 系统性调试方法	3
2.1 为什么要遵循结构化的过程	3
2.2 充分利用机会	3
2.3 13条黄金规则	5
2.3.1 理解需求	5
2.3.2 制造失败	6
2.3.3 简化测试用例	6
2.3.4 读取恰当的错误消息	6
2.3.5 检查显而易见的问题	6
2.3.6 从解释中分离出事实	7
2.3.7 分而治之	7
2.3.8 工具要与bug匹配	8
2.3.9 一次只做一项更改	9
2.3.10 保持审计跟踪	9
2.3.11 获得全新观点	9
2.3.12 bug不会自己修复	9
2.3.13 用回归测试来检查bug修复	10
2.4 构建一个好的工具包	10
2.4.1 工具箱	11
2.4.2 每天运行测试，防止出现 bug	11
2.5 认清敌人——遇到bug家族	13
2.5.1 常见bug	13
2.5.2 偶发性bug	13
2.5.3 Heisenbug	13
2.5.4 隐藏在bug背后的bug	14
2.5.5 秘密bug——调试与机密性	14
2.5.6 更多读物	15
第 3 章 查找根源——源代码调试器	17
3.1 可视化程序行为	17
3.2 准备简单的可预测的示例	18
3.3 使调试器与程序一起运行	18
3.4 学习在程序崩溃时执行栈跟踪	21
3.5 学习使用断点	21
3.6 学习在程序中导航	22
3.7 学习检查数据：变量和表达式	22
3.8 一个简单示例的调试会话	23
第 4 章 修复内存问题	27
4.1 C/C++中的内存管理——功能强大 但很危险	27
4.1.1 内存泄漏	27
4.1.2 内存管理的错误使用	28
4.1.3 缓冲区溢出	28
4.1.4 未初始化的内存bug	28
4.2 有效的内存调试器	28
4.3 示例1：检测内存访问错误	29
4.3.1 检测无效的写访问	30
4.3.2 检测对未初始化的内存的读取 操作	30
4.3.3 检测内存泄漏	31
4.4 示例2：对内存分配/释放的不完整 调用	31
4.5 结合使用内存调试器和源代码调试器	33
4.6 减少干扰，排查错误	33
4.7 何时使用内存调试器	34
4.8 约束	34
4.8.1 测试用例应该有很好的代码 覆盖率	34

4.8.2 提供更多计算机资源	35	6.1.10 算法与实现之间的差异	56
4.8.3 可能不支持多线程	35	6.2 使用剖析工具	58
4.8.4 对非标准内存处理程序的支持	35	6.2.1 不要编写自己的剖析工具	58
第5章 剖析内存的使用	37	6.2.2 剖析工具的工作原理	58
5.1 基本策略——主要步骤	37	6.2.3 了解gprof	59
5.2 示例：分配数组	38	6.2.4 了解Quantify	63
5.3 第1步：查找泄漏	38	6.2.5 了解Callgrind	64
5.4 第2步：设置期望值	38	6.2.6 了解VTune	66
5.5 第3步：测量内存使用	39	6.3 分析I/O性能	68
5.5.1 使用多个输入	39		
5.5.2 在固定时间间隔停止程序	39		
5.5.3 用简单工具测量内存使用	40		
5.5.4 使用top	40		
5.5.5 使用Windows Task Manager	41		
5.5.6 为testmalloc选择相关输入值	42		
5.5.7 确定机器上的内存是如何被释放的	42		
5.5.8 使用内存剖析工具	43		
5.6 第4步：查明大部分内存被哪些数据结构占用了	44		
5.7 综合练习——genindex示例	45		
5.7.1 核实没有大的内存泄漏	46		
5.7.2 估计内存使用	46		
5.7.3 测量内存使用	46		
5.7.4 查找使用内存的数据结构	47		
第6章 解决性能问题	51		
6.1 分步查找性能bug	51		
6.1.1 执行前期分析	51		
6.1.2 使用简单的时间测量方法	52		
6.1.3 创建测试用例	52		
6.1.4 使测试用例具有可再现性	53		
6.1.5 检查程序的正确性	53		
6.1.6 创建可扩展的测试用例	53		
6.1.7 排除对测试用例的干扰	54		
6.1.8 用time命令测量时可能会发生错误和偏差	54		
6.1.9 选择一个能够揭示运行时间瓶颈的测试用例	55		
		第7章 调试并行程序	71
		7.1 用C/C++编写并行程序	71
		7.2 调试竞争条件	72
		7.2.1 使用基本调试器功能来查找竞争条件	73
		7.2.2 使用日志文件来查找竞争条件	74
		7.3 调试死锁	76
		7.3.1 如何确定正在运行的是哪个线程	77
		7.3.2 分析程序的线程	78
		7.4 了解线程分析工具	78
		7.5 异步事件和中断处理程序	80
		第8章 查找环境和编译器问题	83
		8.1 环境变更——问题的根源	83
		8.1.1 环境变量	83
		8.1.2 本地安装依赖	84
		8.1.3 当前工作目录依赖	84
		8.1.4 进程ID依赖	84
		8.2 如何查看程序正在做什么	84
		8.2.1 用top来查看进程	84
		8.2.2 用ps来查找应用程序的多个进程	85
		8.2.3 使用/proc/<pid>来访问进程	85
		8.2.4 使用strace跟踪对操作系统的调用	85
		8.3 编译器和调试器也有bug	87
		8.3.1 编译器bug	87
		8.3.2 调试器和编译器兼容性问题	88

第 9 章 处理链接问题	89	10.5.3 在静态初始化之前连接调试器	118
9.1 链接器的工作原理	89	10.6 使用观察点	119
9.2 构建并链接对象	89	10.7 捕捉信号	120
9.3 解析未定义的符号	91	10.8 捕获异常	122
9.3.1 丢失链接器参数	91	10.9 读取栈跟踪	124
9.3.2 搜索丢失的符号	91	10.9.1 带调试信息编译的源代码的栈跟踪	124
9.3.3 链接顺序问题	92	10.9.2 不带调试信息编译的源代码的栈跟踪	124
9.3.4 C++符号和名称改编	93	10.9.3 不带任何调试信息的帧	125
9.3.5 符号的反改编	94	10.9.4 实际工作中的栈跟踪	125
9.3.6 链接C和C++代码	94	10.9.5 改编后的函数名称	126
9.4 具有两个定义的符号	95	10.9.6 被破坏的栈跟踪	126
9.5 信号冲突	96	10.9.7 核心转储	127
9.6 识别编译器和链接器版本不匹配	96	10.10 操纵正在运行的程序	128
9.6.1 系统库不匹配	97	10.10.1 修改变量	130
9.6.2 对象文件不匹配	97	10.10.2 调用函数	131
9.6.3 运行时崩溃	98	10.10.3 修改函数的返回值	132
9.6.4 确定编译器版本	98	10.10.4 中止函数调用	132
9.7 解决动态链接问题	100	10.10.5 跳过或重复执行个别语句	133
9.7.1 链接或载入DLL	100	10.10.6 输出和修改内存内容	133
9.7.2 无法找到DLL文件	101	10.11 在没有调试信息时进行调试	135
9.7.3 分析载入器问题	102	10.11.1 从栈读取函数参数	137
9.7.4 在DLL中设置断点	103	10.11.2 读取局部/全局变量和用户	138
9.7.5 提供DLL问题的错误消息	104	定义的数据类型	138
第 10 章 高级调试	107	10.11.3 在源代码中查找语句的大	139
10.1 在C++函数、方法和操作符中设置		概位置	139
断点	107	10.11.4 走查汇编代码	140
10.2 在模板化的函数和C++类中设置断点	109		
10.3 进入C++方法	110		
10.3.1 用step-into命令进入到隐式			
函数中	112		
10.3.2 用step-out命令跳过隐式函数	112		
10.3.3 利用临时断点跳过隐式函数	113		
10.3.4 从隐式函数调用返回	113		
10.4 条件断点和断点命令	114		
10.5 调试静态构造/析构函数	116		
10.5.1 由静态初始化程序的顺序			
依赖性引起的bug	117		
10.5.2 识别静态初始化程序的栈			
跟踪	118		
第 11 章 编写可调试的代码	143		
11.1 注释的重要性	143		
11.1.1 函数签名的注释	144		
11.1.2 对折中办法的注释	144		
11.1.3 为不确定的代码加注释	144		
11.2 采用一致的编码风格	144		
11.2.1 仔细选择名称	145		
11.2.2 不要使用“聪明过头”的			
结构	145		
11.2.3 不要压缩代码	145		

11.2.4 为复杂表达式使用临时变量	145
11.3 避免使用预处理器宏	146
11.3.1 使用常量或枚举来替代宏	146
11.3.2 使用函数来替代预处理器宏	148
11.3.3 调试预处理器输出	149
11.3.4 使用功能更强的预处理器	150
11.4 提供更多调试函数	151
11.4.1 显示用户定义的数据类型	151
11.4.2 自检查代码	152
11.4.3 为操作符创建一个函数，以便帮助调试	153
11.5 为事后调试做准备	153
第 12 章 静态检查的作用	155
12.1 使用编译器作为调试工具	155
12.1.1 不要认为警告是无害的	156
12.1.2 使用多个编译器来检查代码	158
12.2 使用lint	158
12.3 使用静态分析工具	158
12.3.1 了解静态检查器	158
12.3.2 将静态检查器检测到的错误减至（接近）零	160
12.3.3 完成代码清理后重新运行所有测试用例	160
12.4 静态分析的高级应用	161
第 13 章 结束语	163
附录 A 调试命令	165
附录 B 工具资源	167
附录 C 源代码	179
参考文献	189

第1章

谁编写软件，谁制造bug (为什么需要本书)

本书讲的是C和C++程序的分析和改进，是由软件开发者写给软件开发者的。

在软件开发工作中，我们经常被派去帮助客户和同行发现bug。他们都记得在学校中学过的一些概念，例如面向对象、代码评审和黑盒与白盒测试，但大部分人都不怎么了解调试工具，甚至有时会犯糊涂，例如不知道某一特定方法应该在何时使用，或者不清楚在调试工具给出混淆或错误结果时应该如何处理。

因此，我们一次又一次地发现不得不教会人们如何追踪bug。奇怪的是，很多程序员都没有将调试转化为一种系统性的方法。虽然软件开发中的很多步骤可以归纳到某一过程中，但当谈到调试时，多数人的观点都是这不仅要求对代码有深入洞察力，而且在追踪bug时还需要灵光一现。遗憾的是，理查德·费曼（Richard Feynman）的“写下问题、努力思考、写下答案”的方法并不是修复软件问题最有效和最成功的方法。

当意识到我们总是不断地写下相同的步骤规则，并为每份bug报告解释相同工具的操作和限制时，我们萌生了一个念头，应该总结所有实践经验、收集这些建议并编纂成书——结果，就是读者现在手捧的这本书。现在，在有人面对查找bug的任务时，我们就可以将本书推荐给他了。我们也相信这样一本讨论系统性调试模式的书将是编程课程的有趣补充，或者作为解决软件问题的课程的补充。原因其实很简单：

软件有bug。仅此而已。

很遗憾这么说，但事实的确如此。甚至每位C和C++程序员都熟知的“hello, world”程序也不例外^①。软件开发意味着必须处理各种缺陷，包括旧缺陷、新缺陷、开发人员自己制造的缺陷以及其他人在代码中留下的缺陷。

^① 如果在调用printf()期间，程序接收到异步信号，而又没有代码检查其返回值，那么就可能产生不完整的输出。

软件开发人员每天都要调试程序。

因此，好的调试技巧是一项必备技能。虽说如此，理工科院校中却很少开设调试课程，这不免令人遗憾。

本书既适合希望扩充自己技能的软件开发人员，也适合想要从基础开始学习一门专业技能的学生。书中提供了很多小的示例和练习，因此非常适合作为计算机科学课程的补充。同时，也可以将本书作为参考指南，随时解决出现的问题。

本书不仅仅讲述简单的源代码调试，还涵盖各个领域最常见的实际问题，包括程序链接、内存存取、并行处理和性能分析。最后几章还讨论了静态检查器和一些较好地运用了调试的代码编写技巧。

但是，本书并不能替代调试器手册，它也不是一本主要讨论Microsoft的Visual Studio或GNU的GDB的书，尽管书中频繁地提到这些工具。实际上，在可能的情况下，我们尽量介绍独立于操作系统和编译器/调试器组合的基本调试和高级调试。当然，在比较依赖编程环境的地方我们也会指出来。

大多数示例使用GCC编译器和GDB调试器。原因很简单：这些工具是免费的，而且可以在很多系统（包括UNIX、Linux、Windows和大量嵌入式平台）中使用。大多数示例可以用附录A中的表A-1来“翻译”，这张表给出的是等效的Visual Studio命令。当无法进行这种简单转换时，表A-1会提供更多的说明。

那么，阅读本书的最好方法是什么呢？这要取决于具体情况。

如果想从基础学起，从头到尾阅读本书不失为一种好方法。第2章概述了收集信息和分析问题的各种机会。第3章更详细地介绍一些关键技术，例如在调试器中运行程序、分析数据以及控制执行流。接下来，第4章介绍如何处理那些由于内存bug而莫名其妙失败的程序。接下来的两章重点关注广义上的优化。第5章讨论内存消耗。第6章解释如何分析执行速度。第7章介绍与多线程程序和异步事件相关的一些难点。接下来是第8章，介绍查找环境和编译器问题。随后的第9章介绍如何处理程序无法开始链接的情况。第9章还讨论链接程序时可能会遇到的其他问题。至此，就可以准备解决一些问题了，例如分析初始化时间问题或者调试那些在没有调试信息情况下编译的代码，这些内容将在第10章中讨论。第10章还将介绍一些其他技术，例如条件断点、观察点和捕获异步事件。最后，第11章和第12章可以帮助读者正确地编写自己的源代码。

此外，如果受到某种实际调试问题的困扰，读者可以在本书中找到相关部分来解决问题。遇到问题时翻阅一下第4章是个不错的想法，特别是当所面对的问题看起来无法用逻辑规则解决的时候。

系统性调试方法

2.1 为什么要遵循结构化的过程

理查德·费曼是个妙人，他的传奇故事读起来总是妙趣横生。他所提出的那个著名方法帮助他解决了大量问题。

诺贝尔物理学奖得主默里·盖尔曼在《纽约时报》上这样总结费曼的方法：“写下问题、努力思考、写下答案。”

这种方案不无感召力。它是一种简单的通用方法，只需要纸笔和一个善于思考的大脑。

将这种方法应用于软件调试时，几乎可以不必知道任何与系统性调试过程或工具有关的事情。但是，必须要全面了解问题。

如果问题（软件）过大，或者软件是由其他人编写的，那么这种方法就不适用了。而且它也不是一种经济的做法，因为此项目知识无法用于彼项目，彼项目只能“重回起点”。

如果想以软件开发谋生，那么在系统性调试方法上加大投入是值得的。事实上，我们将体会到投资回报是相当可观的。

2.2 充分利用机会

本章概述bug查找过程的结构。解决各类问题的细节将在后续各章中讨论。

首先让我们在图2-1所示的简化流中找出调试的机会。

源代码（包括头文件）或多或少地都可以用一种可调试的方式来编写(1)。开发人员也可以编写额外代码，通常称为“插装”（instrumentation），来提高软件的可观察性和可控制性(2)。通常，这是通过宏定义(3) 实现的，宏定义被提供给负责处理源代码和包含头文件的预处理器。编译器标志(4)可用于生成代码和信息，这些代码和信息是源代码调试和剖析工具所必需的。

除了对编译器警告信息加以注意之外，也可以运行静态检查器工具(5)。在链接时，可以选择

带有一些调试信息的库(6)。可以使用链接器选项，比如，可以强制将更多测试例程链接到最后得到的可执行程序中(7)。也可以使用一些对可执行程序进行自动插装的工具，插装是指添加或修改用于分析性能或内存存取的对象代码(8)。

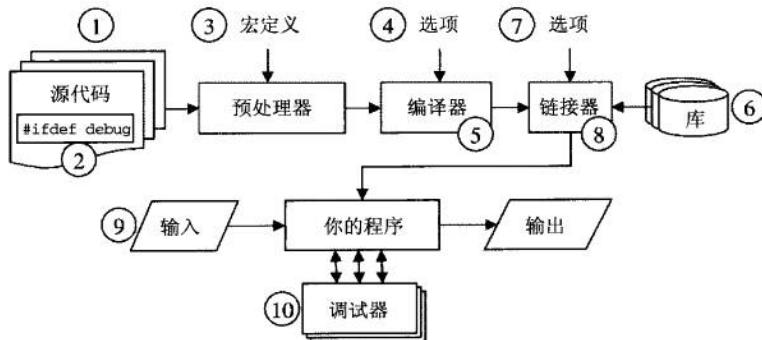


图2-1 简化的构建和测试流

有了可执行程序之后，就可以考虑如何改进它了(9)。选择好的测试用例可以大大减少分析时间。在运行时可以使用各种调试工具(10)，包括源代码调试器、剖析工具、内存检查器，以及一些可以跟踪OS调用的程序。一些工具甚至可以用于“事后剖析”，即在可执行程序已经运行（或崩溃）之后进行剖析。

现在，请花点时间考虑以下三个方面。这将有助于充分利用本书。

(1) 构建和测试过程中有调试机会，如图2-1所示。每个人用来构建和测试软件的具体流程可能略有不同，但基本元素都应该有。

(2) 有13条黄金规则通常适用于构建和测试过程中的任何阶段。这些规则将在2.3节中介绍。

(3) 后续各章将讨论可能遇到的具体问题。例如，第3章讨论源代码调试，第9章讨论链接器问题。

请注意本书是按照面向解决方案的方式组织的，既包括基本技巧，也涉及高级主题。各章的顺序并未采用图2-1所示的流程顺序。以下是各个主题对应的章。

发现bug的机会

- (1) 可调试的源代码：第11章。
- (2) 插装：第5、6、7、11章。
- (3) 宏定义：第11章。
- (4) 编译器标志：第3、6、8、9、12章。
- (5) 静态检查器：第12章。

- 2
- (6) 选择的库: 第4、5、6、11章。
 - (7) 链接器选项: 第9、11章。
 - (8) 代码插装工具: 第4、5、6章。
 - (9) 测试用例/输入数据: 第2章。
 - (10) 调试器。
 - (a) 源代码: 第3、10章。
 - (b) 剖析: 第5、6章。
 - (c) 内存存取: 第4、5章。
 - (d) OS调用跟踪器(例如truss或strace): 第8章。

当然, 利用哪些机会取决于具体问题。因此, 我们无法定义一个通用的、一步一步的调试过程。

既然知道了到哪里去查找bug, 接下来需要确定查找方式。如上文所述, 我们将通过两个步骤来查找。首先, 我们将在下一节列出一组“黄金规则”。这些都是指导方针, 如果运用得当的话, 可在各种调试情况中起到帮助作用。后续几章将讨论面向解决方案的方式中的具体问题。

2.3 13 条黄金规则

经验告诉我们, 不应忽视很多普遍适用的规则。“13条黄金调试规则”可以被看做是对D.J. Agans在[Agans02]中所阐述的“用于发现最难捕捉的软件和硬件问题的9条必备规则”的一个扩展。

13条黄金调试规则

- (1) 理解需求。
- (2) 制造失败。
- (3) 简化测试用例。
- (4) 读取恰当的错误消息。
- (5) 检查显而易见的问题。
- (6) 从解释中分离出事实。
- (7) 分而治之。
- (8) 工具要与bug匹配。
- (9) 一次只做一项更改。
- (10) 保持审计跟踪。
- (11) 获得全新观点。
- (12) bug不会自己修复。
- (13) 用回归测试来检查bug修复。

2.3.1 理解需求

在开始调试和修复任何错误之前, 一定要保证理解需求。有没有标准文档或规格说明可供查阅? 有没有其他文档? 或许软件根本就没有故障, 只是产生了误解, 而不是bug。

2.3.2 制造失败

我们需要一个测试用例。使程序运行失败，然后亲自观察。测试用例是必不可少的，原因有三。

(1) 如果没有看到程序最后可以工作了，怎么能够知道已经修复了问题呢？

(2) 为了遵守规则13“用回归测试来检查bug修复”，需要一个测试用例。

(3) 必须理解造成软件失败的所有因素，并从假设中分离出事实。造成失败的因素有很多，例如环境变量、操作系统或窗口管理器。

bug报告与交通事故或犯罪现场的目击者报告比较相似：通常，事实和解释混杂在一起，而且常常会丢失信息的关键部分——尽管目击者有着最好的意愿，并且确信他们已经描述了完整且未删减的事实。

2.3.3 简化测试用例

下一步是简化测试用例，主要为了如下几个目的。

- 排除不起作用的因素。
- 减少测试用例的运行时间。
- 最重要的是，使测试用例更容易调试。试想，谁愿意处理那些填充了数百或数千项的数据容器呢？

2.3.4 读取恰当的错误消息

某个地方出错时，满屏都是错误消息。

应该重点关注哪些消息呢？

令人吃惊的是，很多人的回答都是错误的。

正确答案是：首先出现的那些消息！^①

但这不一定是最先看到的消息，必要时需要向前滚动。第一个错误之后所发生的每件事都应该用一种怀疑的眼光来看待。第一个问题可能导致程序处于不正常状态。

因此，要事第一——按照出现顺序来修复问题，只有理由很充分时，才能打破这条规则。

2.3.5 检查显而易见的问题

接下来，检查一些显而易见的问题。当发生问题时，软件的所有部分是否都已启动和运行？是否有正确的权限？是否有足够的磁盘空间？在所有相关的文件系统上是否有足够空间（包括C:\WINDOWS、/tmp和/var这些目录）？系统是否有足够内存？

^① 当然，没有任何规则是绝对的。但一般情况下这条简单规则是正确的。

对照10种常见错误，并确信没有人犯这些错误。

2.3.6 从解释中分离出事实

不要直接下结论。整理一份清单，列出对某一事实已知的情况和原因。问一问自己“能否证明它？”行为是不是可再现的？

判断事实的依据是什么？bug报告中可能会这样说：“当选择蓝色项时，软件失败；当选择红色项时，软件总是正常工作”。那么，错误行为是取决于颜色吗？大概不是吧。可能用户选择蓝色时是通过鼠标单击选择的，而其他项则是通过键盘指定名称来选择的。

2.3.7 分而治之

美国国家标准与技术研究所（National Institute of Standards and Technology, NIST）将分而治之方法定义为一种算法技术，它按如下方式解决问题：“如果问题很容易解决，则直接解决；否则就将其分解为两个或多个更小的问题。”而且进一步指出“将各个解决方案整合到一起，形成原始问题的解决方案。”

这种策略可以成功地用于处理多种因素共同作用的复杂调试。在较大的软件项目中，问题通常是由于并发开发活动的冲突引起的，特别是当许多开发人员在同一个源代码库上工作的时候。程序大体上仍然可以运行，但某一特定功能出了问题。例如，程序在上星期三还正常工作，但现在却停止了，而且根本看不出是哪项更改导致了失败。这时该怎么办？下面介绍一种可行的分而治之方法。

分而治之101^①

- 整理一份清单，列出潜在问题以及如何调试它们。
- 将环境更改和源代码更改区分开。
 - 跟踪环境的更改。
 - 撤销源代码的更改。
- 放大并治之。
 - 内存调试。
 - 常规的源代码调试。
 - 同步调试。

1. 整理一份清单，列出潜在问题以及如何调试它们

显然，第一步是理解如何分割问题。首先整理一份清单，列出可能的问题以及如何调试它们。

^① 美国大学生的第一门课程经常编号为101，所以101一般指入门课程。——译者注

源代码库的更改是突发bug的一个原因，但并非是唯一原因。编译器是否被修改了？第三方库文件是否被修改了？是不是程序调用了源代码控制系统外部的其他程序，而这些程序当中的一个发生了更改？上星期三运行的某一功能现在是否还在同一台主机上运行？操作系统库文件是否被修改了？环境变量是否发生变化？等等。与环境相关的问题有很多。最后，调试归根结底是一种试错法（trial-and-error）。要试着理解某些更改导致问题的可能性，以及为测试目的而撤销这些更改的代价。

2. 将环境更改和源代码更改区分开

有一种实用方法可用来查明某一项源代码更改是否与bug有关。获取一个对应于上星期三源代码库的程序修订版本。我们将它称为“好的”修订版本。现在，在完全相同的环境中（相同的主机和环境变量设置，如果可能的话，使用相同的命令行解释器）运行“好的”版本和当前版本，来测试相同的用例。如果这两个版本都失败了，则说明环境的某项更改可能是导致bug的原因。下面是接下来要检查的事项。

- 是否知道哪些因素可能导致失败？参见第8章中与环境有关的bug的例子。
- 是否能确定什么发生了更改？通常环境的更改并不会在修订控制系统的更改中反映出来。是否更改了计算机账户的设置？IT团队是否执行了任何升级？
- 是否能恢复初始环境？用VMware构建一个机器和环境的虚拟副本（参见附录B.3.2），并且在每次修改操作系统、工具和已安装的软件之前存储一个检查点，这样就很容易退回到先前的机器状态了。

但是，如果老的源代码修订正常工作，而最新的修订无法工作，那么源代码库中的更改可能是导致bug的原因。如果是这样，可以尝试通过后退构建（back-out build）工作来缩小搜索范围并隔离有问题的更改。基本思想很简单：用修订控制系统来确定最新的正确工作的源代码版本，检出带有不同时间标签的完整配置并构建它们。同理，确定第一个有问题的版本。然后分析这两个版本之间的源代码更改，bug就隐藏在其中。

3. 放大并治之

可能此时导致问题的原因仍不明显。运行内存调试器（参见第4章）和常规的源代码调试（参见第3章）可能无法解决问题。这时就必须咬紧牙关，面对冗长乏味的同步调试任务，即同步比较两个版本的数据、日志文件和控制流。

2.3.8 工具要与 bug 匹配

不要嫌麻烦，要调试的是出现问题的地方，而不是便于调试的地方。

在特定情况下，某些调试工具比其他工具更易于使用。但所有工具的功能并不相同。使用最熟悉的工具和过程很正常。但应该遵循以下原则：关注那些最有可能找到bug的方面，即使这可能会使工作非常乏味或进入一个不熟悉的领域。

例如，软件开发人员总是不愿使用内存调试器。一种常常听到的借口是“它们总是产生大量奇怪、含义模糊的输出”。即使是在一些显而易见有内存问题的情况下，例如间歇故障和无法解释的随机行为。开发人员仍然不愿使用内存调试器。

2.3.9 一次只做一项更改

尽可能一次只做一项更改，然后检查它是否有意义。如果没有意义，则返回到原来状态，再尝试下一个思路（没有任何规则是绝对的，2.5.4节讨论隐藏在bug背后的bug时，将打破这项规则——代价是做更多的记录工作）。

在调试期间为修改的源代码加注释指明修改的类型和原因是一个好习惯。记住，任何代码修改都可能引入新的问题。调试时要尽量一次解决一个问题。

2.3.10 保持审计跟踪

我们经常必须处理一些涉及多个参数的问题。这时需要尝试多种组合，很容易失去对更改的跟踪。

所以，一定要保持审计跟踪！

这对于假故障尤为重要。在进行手工测试时，要记下都做了哪些事情、执行的顺序以及出现了什么结果。让程序创建日志文件并输出状态消息。一旦找到bug，笔记和日志可能是唯一能够将bug与环境关联起来的信息。实际上，假故障通常并不是随机发现的。它们是由明确定义的事件（但可能是隐晦的事件）触发的，只是这些事件对调试者来说是未知的。

2.3.11 获得全新观点

当陷入僵局时，可以找别人讨论一下。一定要在事实（以及事实根据）与你的理论之间划出明确的界限。很可能是你的理论不够完善。

向别人解释问题的过程可能会帮助认清事实，而且会形成一种全新的观点。当然，建议与专家进行讨论。然而，一些不是专家的人也可能会带来很大帮助，因为我们需要解释更多东西。

2.3.12 bug 不会自己修复

有时，在修改了某些语句之后，bug可能会消失。但是，除非有很充分的理由说明修复很有效，否则最好假设bug仍然存在，并且未来还会发作。源代码的更改可能只是改变了环境，因此也改变了bug出现的概率。

即使有很好的解释，也要验证修复的有效性，方法是取消修复，并检查bug是否重新出现。在做出更改后从头构建程序也是一种好的做法。构建过程中的依赖性可能不够完善，因此，对象代码可能并不完全符合我们的本意。

2.3.13 用回归测试来检查 bug 修复

今天修复了问题，但明天会怎样呢？

为了彻底修复bug，应该将简化的测试用例（规则3）转化为回归测试。可以将回归测试看做是保险螺栓。它可以防止访问源代码库的其他人无意之间破坏我们花费很大力气修复的某一功能。客户也会赞赏这样做，因为没有比挥之不去的bug更令人讨厌的了。

如果以前从未听说过回归测试，可以先看一下2.4.2节的第一小节。此外，自动测试也是必不可少的。当然，必须花费一些力气来保证软件的可测试性，并维护一个回归测试系统，这是专业软件开发的一个不可分割的部分。

软件开发不仅仅是编写代码

软件开发不仅仅是编写源代码，还要求具备其他各种能力，例如软件架构、剖析、调试，等等。一个容易被忽视的方面是对未来的规划，包括付出更多努力来开发可测试的软件和回归测试。

在某些方面软件就像葡萄酒，多年窖酿的美酒才有价值。能够产生数百万美元收入的软件不是一夜间就能创建的，它需要几年时间。

读者现在是否理解了回归测试？

以上就是13条黄金规则。在介绍特定调试和优化问题的解决方案之前，先来看一下最后一些通用的建议。

2.4 构建一个好的工具包

在软件发生大的崩溃之前，应该安装并测试一系列完整的调试工具。这样可以解决各种浅显的问题，客户和经理也就不会总是问这样的问题，因此工作会轻松得多。

好的软件开发人员具有能工巧匠和艺术家的某些风范。你需要一个工具箱（workshop），它装有各种有用且易于找到的工具，而且必须熟练使用这些工具。

保持工具箱的整齐

- 为正在开发的软件安装和测试以下调试工具（只有10行的测试程序不算在内）。
 - 源代码调试器（参见第3章）。
 - 内存调试器（参见第4章）。
 - 剖析工具（参见第5、6章）。
- 确保调试器与编译器兼容。
- 运行并维护单元测试和系统测试。
- 自动执行一些常规任务。

2.4.1 工具箱

读者可能会想“我不需要内存调试器”，这时一定要三思。第4章将阐述软件开发人员为什么需要它。记住，到真正需要时才购买救生设备往往为时已晚。

不需要剖析工具吗？同样的道理。大多数平台上都有一些有用的免费软件解决方案。如果其他选项不适用，可以考虑这些解决方案。

当更换编译器版本时，开发人员应负责检查整个调试工具套件是否仍然工作。确保它对正在开发的软件有效。在只有10行的测试程序上运行调试工具是无效的，要在整个软件上运行，测试每一项功能。

在大型组织中，可能不要求每位软件开发人员都反复测试所有的调试工具。但切莫认为这是别人的问题，因为别人不会修复你的bug，而且当问题得不到修复时，不会有人打电话给他们。实际上，当问题变得更加严重时，“别人”可能甚至不会在场。因此我们要经常检查自己的工具，特别是当环境发生变化的时候。

2.4.2 每天运行测试，防止出现 bug

本书并不是一本讲软件测试的书，但软件测试是一个重要的主题。本节只是浅尝辄止，从最广泛的意义上指出与调试有关的几个方面。更多深入信息可以在[Meyers04]中找到。

1. 回归测试

如果不测试它，它就不会工作。

大量事实可以证明这个结论。当然，这最初听起来有些奇怪。为什么必须要通过测试来保证正常工作呢？

让我们想一下。如何使自己确信某一新代码片段确实按照预期工作了？方法是测试。

那么又如何保证它在一整年中都能运行呢？方法是经常测试。回归测试这个术语是指检查昨天的功能在今天是否仍然工作良好。

这些测试应该是自动进行的，以便能够经常且高效地执行。而且测试应该进行自我检查。即应该有一个回归测试系统，通常这是一组脚本的集合，它执行稳步增加的测试集。结果是两个清单，一个列出通过的测试，另一个列出失败的测试。每次添加新功能或修复了一个bug时，都应该增加测试。

2. 单元测试和系统测试

区别开回归测试的两种类型是有意义的，即单元测试和系统测试。系统测试是将软件作为整体进行测试。这些测试是必需的，它们模拟正常的操作并确保最终用户的功能。

图2-2显示了一个简化的示例，其中假设程序只由三个软件模块组成：读取输入数据的模块、处理数据的模块和生成输出文件的模块。一个简单的回归测试用一组给定的输入文件来运行软

件，然后将输出与一组“黄金输出文件”进行比较。

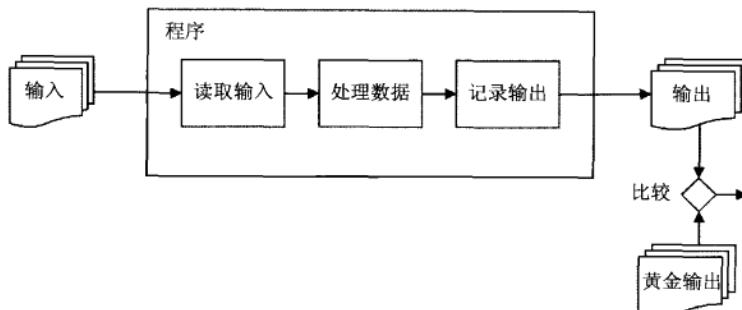


图2-2 简单的测试系统

相反，单元测试关注单个软件构建块。通常，它们需要付出更多工作来开发更多测试程序和测试接口，但这些投入将得到回报。

- 在整个系统可运行之前就可以开发单元测试。
- 单元测试接口可以提高软件的可观察性和可控制性。
- 最后但并非不重要，它们简化了调试任务。

想象一下如果图2-2所描述的系统测试失败了。如果从一开始就可以缩小范围会不会更好？程序组件的单元测试就能够做到这一点。有了正确的基础架构后，可以快速查明数据输入、数据处理和数据输出中哪个模块导致了失败。

图2-3显示了一种可行的方法。程序中内置了单元测试支持，因此可以存储每个单元的输出。此外，可以通过将数据直接输入到很难到达或控制的单元中来绕过常规的数据流。也可以为图2-3所示的三个功能单元创建独立的单元测试程序。

在单元测试中，应该将白盒测试与黑盒测试区分开。黑盒测试的主要目的是验证组件的预期功能，而忽略其实际的实现。黑盒测试的优点是具有很好的可移植性，即使实现发生了变化，这些测试仍然可以正确工作。而白盒测试主要测试实现的边界情况以及一些“弄巧成拙的错误”。它们充当了很好的监督员。

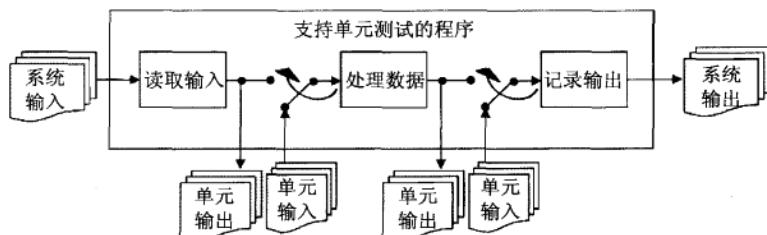


图2-3 支持单元测试的简单测试系统

2.5 认清敌人——遇到 bug 家族

“宽恕你的敌人，但永远不要忘记他们的名字。”

——约翰·F. 肯尼迪

我们可以区分不同类型的bug，它们的修复方法各不相同。下面几节就介绍分类，但这些分类并不是绝对的。

2.5.1 常见 bug

常见bug存在于源代码中。虽然令人讨厌，但它们的行为是可以预测的。模棱两可的规格说明和测试计划中的漏洞通常会导致这种bug的繁殖。常见bug有几个同样令人讨厌的同类bug，这些bug将在后面介绍。

2.5.2 偶发性 bug

虽然在给定正确测试用例的情况下，可以预见到常见bug的发生，但偶发性bug就不同了。它并不会轻易地浮出水面，而且常常是在我们没有准备的情况下发动攻击。成功防范此类bug的关键是如下几点。

□ 设置一个捕猎器，即在程序中添加“看门狗”代码，当有bug试图绕过它时，发出警报。

当然，我们需要保存并检查日志文件，否则这些代码等于是做无用功。

□ 找到正确的诱饵。保持耐心，做好记录。记住，微小的变化也可能事关成败。

如果回归测试受到这类bug的影响，则可以考虑保留^①核心文件，并用源代码调试工具进行事后分析。

2.5.3 Heisenbug

Heisenbug这种bug可能令人崩溃。其名字来自于维尔纳·海森堡（Werner Heisenberg）在量子物理学中提出的测不准原理。

“位置测得越准，此时刻的动量（质量乘以加速度）就越测不准，反之亦然。”

在软件调试中有这样一个类比：越是努力地调试，就越有可能隐藏特定的bug^②。

□ “当我在编译器中打开调试信息时，bug消失了。”

□ “我添加了一条printf()语句，程序正常，而去掉printf()时程序失败了。可是我不能

① 在必要的情况下，也可以强制生成一个核心转储。

② 注意偶发性bug与Heisenbug之间微妙但重要的区别。偶发性bug可能很难定位。然而，一旦找到了一个测试用例，就可以再现这些bug。但Heisenbug就不是这种情况。

把生成调试输出的程序发给客户啊。”

□“在我将测试代码链接到可执行程序中之后，就无法再捕捉到bug了。我甚至都没有调用过任何一个例程。这真是怪事。”

我们必须理解Heisenbug的特性，才能成功地对付这个难缠的角色。通常，它要么是由于资源争用引起的，要么是由于非法使用内存引起的，要么是由于优化错误引起的。

1. 资源争用

竞争条件（race condition）是Heisenbug的常见原因。这是指程序行为依赖于特定代码段执行顺序的情况，而且这种顺序没有明确的定义。那些使用了错误的内部任务通信机制或缺少同步方式的并行多线程程序往往会展现出这样的问题（参见第7章）。但即使是单线程程序也可能会受到影响，初始化顺序的问题只是一个例子（参见第10章）。

所有情况的共性是（至少在某种程度上）未定义执行顺序。因此，任何小的更改，例如添加一条printf()语句或生成带有调试信息的“更慢的”代码，都可能会改变程序的行为，并导致Heisenbug类型的bug。

2. 非法使用内存

内存访问冲突也可能导致Heisenbug，例如读取未初始化的变量、野指针或数组越界问题。看起来一些毫无关联的更改（如添加一个逻辑变量或移动代码）都有可能导致轻微改变堆或栈的内存分配。好消息是内存调试器对于解决这些类型的问题很有帮助（参见第4章）。

3. 优化错误

这种类型的bug是在抽象数据类型和算法的优化中产生的，在相应代码执行某种非法的快捷操作时发作。

当然，读者可能立刻会想到编译器优化。的确，编译器优化时常出现bug。但可能程序中的优化并不是罪魁祸首。不管怎样，一些小的改动可能会改变优化的效果，甚至完全中止了优化。

初级分析在这些情况中并不难做。如果怀疑某项优化是导致问题的部分原因，可以关闭它。

2.5.4 隐藏在 bug 背后的 bug

要经常考虑有多个bug的可能性。在某些情况下，为了避免程序行为的异常，需要修复多个问题。

如果怀疑有多个bug，那么就要考虑违反规则9“一次只做一项更改”了，这样做的代价就是记录的工作量加倍了（规则10）。

2.5.5 秘密 bug——调试与机密性

常见bug的另一个讨厌的同类是“秘密bug”。它在客户正在使用软件时发作。然后客户会告诉我们无法提供测试用例，要么是因为机密性，要么因为客户没有权力提取、打包和发送所有相

关信息。而我们同样无法发送一个包含源代码在内的可调试的软件版本。结果就是进退两难。

这时应考虑三种选择：尝试自己再现这一问题、在客户方进行现场调试或者使用一个安全连接进行远程调试。

1. 自己尝试再现相同的bug

通常需要索取测试用例或精简版的测试用例。但如果由于保密原因或时间约束无法获取测试用例，则需要自己尝试再现相同的bug。

首先，尝试从客户那里获取一个恰当的、清晰的错误描述，索取日志文件，并利用内存检查器（参见第4章）和跟踪工具（例如

2. 现场调试

另一种方法是现场调试。编译带有调试信息的软件，将其发送给客户，但删去所有源代码文件^①。符号调试信息在对程序进行反向工程时是有一定价值的，但当然不如源代码有用。现在带上装有相关源代码部分的笔记本电脑去访问客户，源代码最好是加密的。由于程序已经有符号信息，因此在客户计算机上运行程序的调试器将会很好地工作，我们可以在函数中设置断点、走查程序的行、获取栈的跟踪信息、获取或设置变量。但是，它不会显示源代码，而只是给出文件名称和行号，这时就必须查看笔记本电脑上的源代码了。

记得带手机，打电话给同事可以节省大量时间。

这种类型的调试代价很高并且耗时（需要旅行），而且压力也不小，一方面是时间压力，一方面是客户就站在背后观察。但这可能是最后也是唯一的选项。

3. 使用安全的连接

可以使用一种比较安全的远程桌面访问机制来优化现场调试这种方法，例如使用SSH连接的WebEx（参见附录B.8.7）或VNC（参见附录B.8.6）。情况大致是相同的：客户的计算机运行可调试的软件，但源代码仍然在调试者的计算机上。调试者仍然能看到软件的运行情况，或许甚至可以与软件和调试器交互。这样就不必出门了，而且可以访问自己公司的计算机网络。

2.5.6 更多读物

David J. Agans所著的*Debugging: The Nine Indispensable Rules for Finding Even the Most Elusive Software and Hardware Problems*[Agans02]是一本非常有趣的调试入门书，而且不仅限于软件。

^① 在Solaris操作系统上还需要发送对象代码文件，因为大部分调试信息保存在这些文件中，而不是位于程序中。

我们使用Andreas Zeller的*Why Programs Fail: A Guide to Systematic Debugging*^① [Zeller05]一书作为bug查找技术的参考书，该书也代表了自动调试领域的科学进展。在该书的网站上可以找到示例和课程材料，网址为<http://www.whypartoffail.com>。

如果读者正在使用嵌入式系统，那么特别推荐M. Barr和A. Massa所著的[Barr06]一书。书中有一节是专门讲调试的，介绍二值图像的下载、远程调试器和仿真等内容。另外推荐S. Ball所著的*Debugging Embedded Microprocessor Systems*[Ball98]。

有关各种调试主题的更多阅读资料可以在以下书籍中找到：[Brown88]、[Ford02]、[Kaspersky05]、[Lencevicius00]、[Metzger03]、[Pappas00]、[Stitt92]、[Telles01]和[Rosenberg96]。要了解目前的研究主题，请参考[Zeller05]和[Fritzson93]。

建议读者熟悉自己的调试工具文档。GDB调试器手册 [Stallmann02]已经出版。可以到附录B.2.3中了解如何在网上查找GDB文档。Visual Studio的文档名为“MSDN Library for Visual Studio”，在安装软件时可以获得它。更多信息参见附录B.1.1。

① 本书中文版名为《Why Programs Fail——系统化调试指南》。

3.1 可视化程序行为

可视化程序行为的最快速有效的工具是调试器，它是一种测试和调试其他程序的程序。如果调试器可以显示程序源代码内部的当前执行点或程序错误的位置，这种调试器就称为源代码调试器（source code debugger）或符号调试器（symbolic debugger）。

有了源代码调试器（以下简称调试器）以后，就可以逐行地走查源代码，查看程序的条件语句和循环语句都经过了哪些路径，显示哪些函数被调用了，以及显示目前正处于函数调用栈的什么位置。我们可以检查变量值，并在个别的代码行中设置断点，然后让程序运行，直至到达此行。这是在复杂程序中进行导航的便利方法。

调试器将显示程序都执行了哪些操作，这是修复任何bug的先决条件。如果代码是自己编写的，调试器可以帮助实现预期行为，即代码的功能。如果是别人编写的，调试将呈现出代码执行的动态视图，以补充静态代码检查。

本章将介绍基本的源代码调试器功能，并讲述如何用它们来查找C和C++程序中的bug。同时介绍独立于特定的计算机平台或工具。示例中将使用两个非常常见的调试器：GDB和Visual Studio。同时列出GDB和Visual Studio访问每个被讨论的特性的命令。为了节省篇幅，这里只展示节选的GDB输出并简要介绍Visual Studio如何输出结果，不会给出屏幕截图。

GNU调试器GDB代表从具有命令行接口的命令行解释器（command shell）运行的调试器。GDB与GCC编译器一起使用，已经被植入很多操作系统中，如Windows、Solaris、UNIX、Linux和用于嵌入式系统的操作系统。有关GDB的下载信息和文档，参见附录B.2.3。

Microsoft Visual Studio是用于Microsoft Windows系统的调试器，它与Visual C++编译器一起使用。Visual Studio是GUI（Graphical User Interface，图形用户界面）程序，是IDE（Integrated Development Environment，集成开发环境）的一部分。有关Visual Studio的更多信息，请参见附录B.1.1。

大多数源代码调试器共享一个公共的特性集，因为这些调试器具有类似的命令。附录A中的

表A-1给出了GDB和Visual Studio这两种常见调试器的命令转换表。dbx是用于Sun Solaris的命令行调试器。TotalView（参见附录B.2.5）是用于Linux和MacOS的基于GUI的调试器，它能调试基于线程、OpenMP和MPI的并行程序。ARM RealView Development Suite和Lauterbach TRACE32是使用ARM CPU的系统的调试器。

建议读者通过调试一个简单的、可预测的示例来熟悉任何新调试器及其基本特性。本章将介绍如何构建一个小的测试程序，并将它与调试器一起运行。示例将演示一些重要的特性，包括在程序崩溃时执行栈跟踪、设置断点、代码走查以及变量检查。

第10章将介绍高级特性，例如修改正在运行的程序的状态以及从调试器调用函数。

3.2 准备简单的可预测的示例

本章以计算任意非负整数 n 的阶乘函数 $n!$ 为例， $0! = 1$ ，当 $n > 0$ 时， $n! = n \times (n-1)!$ 。

图3-1显示了计算阶乘函数的程序factorial的源代码。这里将函数factorial(int n)实现为它自己的一个递归调用。注意，代码的安全性很差，因为没有 n 的负值保护，也没有由于 $n > 12$ 时32位整数溢出而导致错误结果的保护。本例就是为了演示用来查找这些bug的基本调试器特性。

```

1  /* factorial.c */
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  int factorial(int n) {
6      int result = 1;
7      if(n == 0)
8          return result;
9      result = factorial(n-1) * n;
10     return result;
11 }
12
13 int main(int argc, char **argv) {
14     int n, result;
15     if(argc != 2) {
16         fprintf(stderr, "usage: factorial n, n >=0\n");
17         return 1;
18     }
19     n = atoi(argv[1]);
20     result = factorial(n);
21     printf("factorial %d = %d\n", n, result);
22     return 0;
23 }
```

图3-1 factorial.c：计算 $n!$ 的递归函数

3.3 使调试器与程序一起运行

为了开始进行调试，需要使程序与调试器一起运行。必须指示编译器将调试信息放到程序的

对象代码中。这些调试信息也称为调试符号（debug symbol）或符号信息（symbolic information），它们包含函数和变量的名称以及CPU指令、源文件和行号之间的关系。注意，大多数编译器在默认或优化模式下未启用调试，因为对象代码中的调试信息会使程序更大。此外，大多数编译器优化在调试模式下是禁用的，因此程序会运行得较慢。

对于GNU编译器GCC和大部分其他编译器，进行调试的编译器标志是-g。用GCC编译factorial程序的方法如下：

```
> gcc -g -o factorial factorial.c
```

对于Visual Studio调试器，构建和调试程序的最方便方法是创建项目。参见下面的“如何在Visual Studio 2008中构建和运行程序”，其中介绍了用于创建项目、构建程序、向控制台程序输入命令参数以及运行程序的指令。

下一步是将程序载入调试器，并运行它。调试器将一直处于运行模式，直到被打断、程序崩溃或程序退出。

对于GDB调试器，需要键入命令gdb，并输入程序名称作为它的第一个参数。GDB将启动一个命令行解释器，在这里可以输入命令来控制调试器。运行程序的命令是run，后跟要传递给程序的命令行参数。在GDB中运行我们的示例，方法如下：

```
> gdb factorial
<lots of copyright stuff..>
(gdb) run 1
<messages about Loaded symbols..>
factorial 1 = 1
Program exited normally
```

如何在Visual Studio 2008中构建和运行程序

创建项目：

- 将factorial.c文件放到一个新目录中。
- 启动Microsoft Visual Studio。
- 单击菜单项File/New/Project From Existing Code...（文件/新建/从现有代码创建项目）创建一个新项目。
- 选择Visual C++作为项目类型，单击Next（下一步）。在Project file location（项目文件位置）输入源代码文件factorial.c的路径，在Project name（项目名称）位置输入factorial。
- 项目创建中的下一个对话框是Specify Project Settings（指定项目设置）。在Use Visual Studio, Project type（使用Visual Studio/项目类型）项下，选择Console application project（控制台应用程序项目）。不要选择Add support for ATL（添加对ATL的支持）或任何其他工具包。

□ 单击**Finish**（完成），向导将创建factorial项目。

□ 检查一下在**Solution Explorer**树视图中是否看到了factorial项目。

构建程序：

□ 按F7键或单击菜单项**Build/Build Solution**（生成/生成解决方案）来编译程序。默认得到一个调试版本。调试版本的配置将包含所有正确的调试标志和库。

□ 检查程序是否正确编译。

添加命令行参数：

□ 右键单击选中factorial项目，并打开菜单项**Project/Properties**（项目/属性），显示factorial属性页。打开**Configuration Properties/Debugging**（配置属性/活动(Debugg)），在属性表中找到**Command Arguments**（命令参数）项。输入factorial命令行参数所需的值，单击**OK**（确定）关闭对话框。

运行程序：

□ 按F5键或选择菜单项**Debug/Start Debugging**（调试/启动调试）来运行程序。

附加一个已经运行的程序：

□ 单击菜单项**Debug/Attach to Process...**（调试/附加到进程），然后单击要调试的程序进程。

在Visual Studio中调试控制台应用程序的一个问题是命令行控制台的处理。在我们的示例中创建了一个控制台窗口，并弹出来，程序运行并在窗口中输出stdout，然后控制台就完全消失了。一个好的解决方法是在程序的最后一行（第22行）上右击，并选择**Breakpoint/Insert Breakpoint**（断点/插入断点）。

后面将详细介绍断点，但现在这种方法足以使我们看到程序的输出：

```
factorial 1 = 1
```

此时，就可以用调试器来做一些有用的测试了。用参数n=-1来运行程序，这将导致递归不能停止：

```
(gdb) run -1
Program received signal SIGSEGV, Segmentation fault.
factorial (n=-103583) at factorial.c:6
6      int result = 1;
```

上面的GDB输出告诉我们程序在factorial函数的第6行发生了一个内存错误，原因是它尝试再次调用它自己。GDB显示了发生错误的实际源代码行。

在Visual Studio中，将程序参数更改为-1，然后按F5再次运行程序，将出现有以下消息的新窗口：

```
Unhandled exception at <...> in factorial.exe: Stack overflow
```

为了在Visual Studio中找到崩溃位置，单击此窗口中的**Break**（断点）。调试器将显示源代码窗口，窗口中显示factorial.c文件和一个表示当前程序位置的黄色箭头，它指向第9行，即factorial函数的开始。

3.4 学习在程序崩溃时执行栈跟踪

Visual Studio已经说明了崩溃的原因：栈溢出。C/C++程序的栈是一个内存片段，用来存储每个活动的函数调用的栈帧（stack frame）。栈帧由返回地址、函数的参数和局部变量组成。栈跟踪（stack trace）是一个实际的栈帧链，这个链从调试器当前停止或暂停的最顶部函数开始，向下一直到main()函数。当嵌套函数调用的链过长，造成栈没有足够内存来存储当前栈帧时，就发生了栈溢出。

除了在源代码中显示程序崩溃的位置以外，调试器还显示栈帧和崩溃的栈跟踪。栈跟踪是用于调试崩溃位置的有用信息，因为它可以告诉我们导致崩溃的函数调用链。

GDB调试器通过编号来引用栈帧，其中当前栈帧的编号为0，main()函数的栈帧编号最高。这个数字也是调用栈的大小。GDB的栈跟踪命令是bt、backtrace或where。栈跟踪确定本例中的错误是程序的栈溢出，原因是对factorial()函数进行了过多的递归调用：

```
(gdb) backtrace
#0 <..> in factorial (n=-105582) at factorial.c:9
<lots of frames..>
#103581 <..> in factorial (n=-2) at factorial.c:9
#103582 <..> in factorial (n=-1) at factorial.c:9
#103583 <..> in main (argc=2, argv=0x761ce8) at factorial.c:20
```

可以在调用栈中进行上下导航，并检查函数参数和局部变量的值。在GDB中，可以使用命令up或down在栈中移动。

在Visual Studio中，可以看到名为**Call Stack**（调用堆栈）的窗口。此窗口只显示前1000个帧。单击**Call Stack**窗口中的行可以在调用栈中导航。

3.5 学习使用断点

交互式调试要求能够在程序终止之前挂起程序的执行，并且能够以可控制的方式在程序代码中导航。这是通过断点实现的。调试器提供了一系列断点命令，如下。

- 行断点（line breakpoint）——当到达源代码中的指定行时，暂停程序。
- 函数断点（function breakpoint）——当到达指定函数的第一行时，暂停程序。
- 条件断点（conditional breakpoint）——如果特定条件保持为真，则暂停程序。
- 事件断点（event breakpoint）——当发生特定事件时，使程序进入暂停模式。支持的事件包括来自操作系统的signals和C++异常（exception）。

附录A中的表A-1提供了全部调试器命令。第10章将讨论事件断点的更多细节：10.4节讨论条件断点、10.6节讨论观察点（watchpoint）、10.7节讨论捕获信号（catching signal）、10.8节讨论捕获异常（catching exception）。

3.6 学习在程序中导航

上面介绍了调试器有一些用于运行和暂停程序的命令。

- **run**——run命令将开始程序。可以通过命令行参数或环境变量来控制和更改运行程序的环境。
- **start**——start命令将运行程序，直到main()的第一行，然后停止程序的执行。这样就不必搜索包含main()函数的文件，并在其第一行设置显式的断点了。
- **pause**——pause命令将中断一个正在运行的程序。在某些调试器中，键入Ctrl-C或单击Pause键将起到相同的效果。
- **continue**——调试器命令continue使暂停的程序恢复执行。

为了理解复杂代码片段的行为，需要逐行遍历源代码。调试器提供了逐行步进（stepping）的功能。有3种不同的步进模式，即step-into、step-over和step-out，它们的区别在于处理函数调用的方式。下面将介绍它们的工作原理。建议读者在factorial示例上试一下这3种模式，以便熟悉通过它们在程序中导航的方法。

- **step-into**——调试器命令step-into（在GDB中是step）的作用是移动到下一个可执行的代码行。如果当前行是一个函数调用，则调试器将进入函数，并停止在函数体的第一行。step-into可以帮助初步揭开代码位置的谜团。例如，函数调用和函数本身可能在不同的文件中。10.3节详细介绍了C++隐式函数调用中的step-into。
- **step-over**——调试器命令step-over（在GDB中是next）的作用是在同一个调用栈层中移动到下一个可执行的代码行。如果当前行是一个函数调用，则调试器将在函数调用之后的下一条语句停止。调试器不会进入函数体。如果当前行是函数的最后一行，则step-over将进入下一个栈层，并在调用函数的下一行停止。
- **step-out**——调试器命令step-out（在GDB中是finish）的作用是在栈中前进到下一层，并在调用函数的下一行停止。

GDB和Visual Studio中的最常见命令可参见附录A中的表A-1。

3.7 学习检查数据：变量和表达式

显示变量值的方法为：在图3-1的factorial示例中的第7行设置一个断点，然后启动程序。当程序在第7行暂停时，在GDB中键入print n。

- **print**——**print**命令将输出变量或表达式的当前值。
- **显示表达式**——继续显示表达式的值。每次程序暂停执行时，值都会更新。

在Visual Studio中，将光标置于函数参数n上，会弹出一个小的**DataTip**（数据提示）窗口，显示变量名和值。也可以使用**Debug/Windows/Locals**（调试/窗口/局部变量）菜单项来创建一个窗口，显示当前函数或方法的局部变量。要在Visual Studio中显示全局变量，可以单击**Debug/Windows/Watch**（调试/窗口/监视）菜单项，打开一个**Watch**（监视）窗口，然后在**Name**（名称）栏中输入要查看的全局变量的名称。

在很多情况下，输出表达式的值是很方便的。调试器将使用来自当前栈帧的变量值来计算表达式，并在输出区域中显示结果，或显示为控制台输出。表达式可能由变量名称构成，也可能由运算符、变量、类型转换和函数调用构成。表达式计算结果的输出格式取决于表达式的数据类型：float类型的表达式的输出结果带小数点，而int类型的表达式的结果则为整数。可以为**print**命令选择不同的输出格式。例如，含有状态位的变量可以用二进制或十六进制格式来显示。

如果必须分析和反复输出相同的表达式，则可以在调试器每次暂停时，使用**显示**（display）特性来计算并自动输出表达式的结果。使用显示特性可以在调试器中对程序进行导航的同时有效地监视感兴趣的变量。在GDB中，命令名称是**display**。在Visual Studio中，可以打开一个**Watch**（监视）窗口，显示变量和表达式。

调试器有一个显示表达式的清单，类似于断点清单。我们可以在显示清单中禁用、启用、添加或删除表达式，具体方法参照调试器文档。注意不可过度地使用显示，因为如果调试器显示的数据表达式过多，就会隐藏相关细节。附录A中的表A-1列出了GDB和Visual Studio中最常用的输出和显示命令。

3.8 一个简单示例的调试会话

下面来简单回顾一下前几节介绍的调试器命令和特性。以图3-1所示的**factorial**为例，用调试器来查明为什么当n >= 13时，程序返回错误值。程序计算13!的返回结果是1932053504，而正确的值是13! = 6227020800。

第一项任务是用参数13来运行程序，并在**main()**的第一行停止，然后逐行前进，直到到达第20行，即**factorial()**函数被调用的地方。这样做的目的是检查命令行参数是否被正确解析，以及整数值13是否被传递给**factorial()**函数。然后进入**factorial()**函数。用GDB来调试这个示例：

```
> gdb factorial  
...  
(gdb) start 13
```

```
(gdb) next
...
(gdb) next
20      result = factorial(n);
(gdb) print n
$1 = 13
(gdb) step
factorial (n=13) at factorial.c:6
6      int result = 1;
```

我们需要检查递归调用是否正常工作，因此在第8行设置一个断点，以便检查当 $n = 0$ 时，函数不再调用自己。另外在第10行设置一个断点，以便能够查看每次调用factorial的返回值。这里不使用步进，而是使程序继续第一个断点。

```
(gdb) break 8
Breakpoint 2 at <...>: file factorial.c, line 8
(gdb) break 10
Breakpoint 3 at <...>: file factorial.c, line 10
(gdb) continue
Continuing.
Breakpoint 2, factorial (n=0) at factorial.c:8
```

输出变量n和result，检查函数是否正确工作。还可以使用变量result上的显示命令，这样就不必重复键入print命令。

```
(gdb) print n
$1 = 0
(gdb) print result
$2 = 1
(gdb) display result
1: result = 1
```

当继续重复continue命令时，得到当 $n = 0, 1, \dots, 13$ 时程序第10行的变量result的输出结果。
当 $n=13$ 时，程序暂停。

```
(gdb) continue
Breakpoint 3, factorial (n=1) at factorial.c:10
1: result = 1
(gdb) continue
...
Breakpoint 3, factorial (n=12) at factorial.c:10
1: result = 479001600
(gdb) continue
Breakpoint 3, factorial (n=13) at factorial.c:10
1: result = 193053504
(gdb) print 13 * 479001600
$5 = 193053504
```

注意， $n=12$ 时的结果仍然是正确的，但它与13相乘得到了一个错误结果。重复第9行的计算，输出表达式 $13 * 479001600$ ，看到它再次产生计算错误。错误原因是 $13!$ 的值太大了，无法存储在int类型的32位变量中，因此整数溢出导致了错误的结果。

经验总结

- 使用源代码调试器来可视化程序的行为。
- 准备一个简单的示例来熟悉调试器的特性。
- 使调试器与程序一起运行。
- 学习在程序崩溃时对栈跟踪进行分析。
- 学习使用断点。
- 学习在程序中导航。
- 学习检查变量和表达式。
- 在一个简单的示例上进行一次调试会话。



本章介绍如何在内存调试器的帮助下查找C/C++程序中的bug。内存调试器是一个运行时工具，它的用途是跟踪和检测C/C++内存管理和访问中的bug，但并不能取代常规调试器。本章将介绍C/C++程序中经常出现的内存访问bug，并介绍内存调试器，给出使用这些工具来查找bug的两个示例。然后介绍如何同时运行内存调试器和源代码调试器，如何通过编写一个排查文件来处理无用的错误消息，以及需要考虑哪些约束。

4.1 C/C++中的内存管理——功能强大但很危险

C/C++语言能够管理内存资源，并且可以通过指针直接访问内存。有效的内存处理和“靠近硬件的编程”是C/C++取代汇编语言开发大型软件项目（如操作系统）的原因；在这些项目中，性能和低开销非常重要。程序员可以控制C/C++中的动态内存〔也称为堆内存（heap memory）〕分配。可以使用malloc()等函数和各种形式的new操作符来分配新内存，未使用的内存则可以通过free()或delete返回。

C/C++中的内存处理具有很高的自由度、可控度和性能，但也伴随着高昂的代价，即内存访问频繁地发生bug。最常见的内存访问bug是内存泄漏（memory leak）、内存管理的错误使用（incorrect use of memory management）、缓冲区溢出（buffer overrun）和读取未初始化的内存（reading uninitialized memory）。

4.1.1 内存泄漏

内存泄漏是一种在运行时分配但当程序不再需要它时未释放的数据结构。如果内存泄漏频繁发生或者很大，那么计算机中所有可用主内存都将被耗尽。程序首先会变慢，因为计算机开始将页面转换为虚拟内存，最后以内存不足（out-of-memory）告终。常规调试器很难找到内存泄漏，因为没有明显的错误语句，而是丢失或忘记了调用某条语句。

4.1.2 内存管理的错误使用

内存管理的错误调用涉及一大类bug，例如多次释放一个内存块、在释放一个内存块之后又访问它或者释放一个从未分配的内存块。此类错误还包括在C++中使用`delete`而不是`delete[]`来取消数组的分配，以及将`malloc()`与`delete`混用，或者将`new`与`free()`混用。

4.1.3 缓冲区溢出

缓冲区溢出是指在已分配内存外部的内存被改写或破坏。很多地方都可能发生缓冲区溢出，包括全局变量、栈上的局部变量，以及通过内存管理在堆上分配的动态变量。

内存破坏的一个不良后果是在改写内存的那条语句中看不到bug。只有以后程序中另一条语句访问这个内存位置时，bug才会显露。由于内存位置有一个非法值，所以程序可能会产生多种错误行为，例如程序可能会计算出错误结果，或如果非法值在一个指针中，则程序将尝试访问受保护的内存并导致崩溃。如果函数指针变量被改写，则程序将跳转，并将数据当做程序代码来执行。关键是导致内存破坏的语句与触发可见bug的语句之间并没有明确的关系。

4.1.4 未初始化的内存 bug

由于C/C++允许创建没有初始值的变量，所以有可能发生读取未初始化的内存的事件。程序员全权负责初始化所有全局变量和局部变量，可以通过赋值语句或各种C++构造函数来完成。内存分配函数`malloc()`和操作符`new`不会初始化或清除已分配的内存块。未初始化的变量将包含无法预料的值。

4.2 有效的内存调试器

解决上述几种内存访问bug需要适当的调试工具。事实证明，GDB这样的传统调试器无法有效地找出与内存泄漏、破坏或未初始化内存有关的bug。处理大型软件项目中的内存泄漏问题，很多程序员都有共同的想法，即创建一些带有特殊插装的内存管理函数或操作符，来跟踪内存块被分配到何处，以及在程序结束时每个内存块是否被正确释放。

由于每个人在C/C++程序中都有相同的内存bug，而且每个人都编写自定义插装来跟踪一些bug，这就催生了内存调试器（memory debugger）市场。最有名的工具是Purify，它是Pure Software公司在1991年发布的。自此，Purify的名字就成了内存调试的代名词。其他工具还有Insure++、Valgrind和BoundsChecker等。参见附录B.4中的工具，此外[Luecke06]对各种特性做了调查比较。

内存调试器详细记录所有已分配和释放的动态内存。它们也拦截和检查对动态内存的访问。一些内存调试器可以检查对栈上局部变量和静态分配内存的访问。Purify和BoundsChecker使用程

序链接时的对象代码插装(object code instrumentation)进行检查。Insure++使用源代码插装(source code instrumentation)，而Valgrind则在虚拟机上执行程序，并监视所有内存事务。利用代码插装，调试工具可以定位发生内存bug的源代码语句。

内存调试器可以检测以下bug。

- 内存泄漏。
- 访问已释放的内存。
- 多次释放同一个内存位置。
- 释放从未分配的内存。
- 混用C中malloc()/free()和C++中new/delete。
- 对数组使用delete，而没有使用delete[]。
- 数组越界错误。
- 访问从未分配的内存。
- 读取未初始化的内存。
- 读或写空指针。

下一节将演示如何将内存调试器与程序连接起来，以及调试工具如何查找和报告bug。

4.3 示例 1：检测内存访问错误

第一个示例是：为数组分配动态内存，然后访问最后一个数组元素之外的元素，读取一个未初始化的数组元素，最后没有释放分配给数组的内存。这里使用Linux上的公共工具Valgrind作为内存调试器，并演示它如何自动检测这些bug。程序main1.c的代码如下：

```

1  /* main1.c */
2  #include <stdio.h>
3  int main(int argc, char* argv[]) {
4      const int size=100;
5      int n, sum=0;
6      int* A = (int*)malloc( sizeof(int)*size );
7
8      for (n=size; n>0; n--) /* walk through A[100]...A[1] */
9          A[n] = n;           /* error: A[100] invalid write */
10     for (n=0;n<size; n++) /* walk through A[0]...A[99] */
11         sum += A[n];       /* error: A[0] not initialized */
12     printf ("sum=%d\n", sum);
13     return 0;             /* mem leak: A[] */
14 }
```

编译带有调试信息的程序，在Valgrind中运行：

```
> gcc -g main1.c
> valgrind --tool=memcheck --leak-check=yes ./a.out
```

下几节将通看一下Valgrind报告的错误清单。

4.3.1 检测无效的写访问

第一个（也可能是最严重的）错误是缓冲区溢出，即对数组元素A[100]的意外写访问。由于数组只有100个元素，因此最大的有效索引是99。A[100]指向未分配的内存位置，这个位置刚好在为数组A分配的内存后面。因此，Valgrind报告了“invalid write”错误：

```
==11323== Invalid write of size 4
==11323==   at 0x8048518: main (main1.c:9)
==11323== Address 0xBB261B8 is 0 bytes after a block
==11323==   of size 400 alloc'd
==11323==   at 0x1B903F40: malloc
==11323==   (in /usr/lib/valgrind/vgpreload_memcheck.so)
==11323==   by 0x80484F2: main (main1.c:6)
```

字符串==11323==是指进程ID，当Valgrind检查多个进程时，它很有用^①。重要的信息是main1.c的第9行发生了无效写操作。其他信息还揭示了被分配的最近的内存块地址，以及它是如何分配的。内存调试器猜测第9行的无效写操作与此内存块有关。猜测是正确的，因为这两个内存块都属于同一数组A。

注意，只有当使用malloc()或new为数组分配动态内存时，Valgrind才能够捕获到数组越界错误。本例正是这种情况，发生在第6行：

```
6     int* A = (int*)malloc( sizeof(int)*size );
```

如果将第6行改写为int A[size]，则A就是一个位于栈上的局部变量，而不是堆上。事实证明，Valgrind检测不到这样的错误，但Purify能够捕获它。这说明并非所有内存调试器都能够准确地报告相同的错误。

4.3.2 检测对未初始化的内存的读取操作

main1.c中的下一个错误是读取了未初始化的内存。由于第8行的索引计算错误，element A[0]一直未被写入初始值。后面，第11行的语句sum += A[n]要读这个元素，这意味着变量sum也“被感染”了。

要指出的重要一点是Valgrind何时报告这个未初始化的内存错误。sum的感染发生在第11行。但错误却不是在该时刻报告的，而是在后面第12行输出变量值时才报告的：

```
==11323== Use of uninitialised value of size 4
==11323==   at 0xBA429B7: (within /lib/tls/libc.so.6)
==11323==   by 0xBA46A35: __IO_vfprintf (in .../libc.so.6)
==11323==   by 0xBA4BDAF: __IO_printf (in .../libc.so.6)
==11323==   by 0x804855C: main (main1.c:12)
==11323==
```

^① 默认情况下，Valgrind只检查已被调用的第一个进程（即父进程）。使用选项--trace-children=yes可以检查所有子进程。

```

==11323== Conditional jump or move depends on
==11323== uninitialized value(s)
==11323==     at 0x1BA429BF: (within .../libc.so.6)
==11323==       by 0x1BA46A35: __IO_vfprintf (in .../libc.so.6)
==11323==       by 0x1BA4BDAF: __IO_printf (in .../libc.so.6)
==11323==       by 0x804855C: main (main1.c:12)

```

遗憾的是，Valgrind没有给出sum被感染位置的详细解释，因此必须自己查找。分析源代码是一种方法。还有一种临时方法是在for循环中的第11行后面添加一条空语句：if (sum>0) do nothing()。它的作用是观察前面的sum值。

4.3.3 检测内存泄漏

最后一个bug是内存泄漏。程序的第6行通过malloc(sizeof(int)*size)语句为数组A分配了内存，但一直没有相应的free(A)语句来删除它。在程序结束时，内存调试器报告了这个泄漏错误：

```

==11323== 400 bytes in 1 blocks are definitely lost
==11323==     in loss record 1 of 1
==11323==     at 0x1B903F40: malloc
==11323==       (in /usr/lib/valgrind/vgpreload_memcheck.so)
==11323==     by 0x80484F2: main (main1.c:6)

```

4.4 示例 2：对内存分配/释放的不完整调用

第二个示例main2.c演示了如何在一个使用了char*类型C字符串的程序中查找内存分配bug：

```

1  /* main2.c */
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <string.h>
5  int main(int argc, char* argv[]) {
6      char* mystr1=strdup("test");
7      char* mystr2=strdup("TEST");
8      mystr1=mystr2;
9
10     printf ("mystr1=%s\n", mystr1);
11     free(mystr1);
12
13     printf ("mystr2=%s\n", mystr2);
14     free(mystr2);
15     return 0;
16 }

```

strdup()函数返回其字符串参数的一个副本。它在堆上分配了必要的内存。调用函数过会儿负责释放这些内存。现在编译并运行程序：

```

> gcc -g main2.c
> valgrind --tool=memcheck --leak-check=yes ./a.out

```

实际的错误发生在第8行的语句mystr1=mystr2。编码风格显示出在第8行之前，mystr1和mystr2都有单独分配的内存。之后，这两个变量指向了同一个内存位置。但是，程序员并没有考虑这一点，因此随后发生了几个错误。

第一个错误发生在第13行，即当输出mystr2的时候。由于第11行已经释放了分配的内存，因此mystr2现在是一个指向已释放内存的指针。在printf()语句中访问它就形成了一个无效的读取访问：

```
--=11787== Invalid read of size 4
--=11787==      at 0xBA71903: strlen (in .../libc.so.6)
--=11787==      by 0xBA4BDAF: _IO_printf (in .../libc.so.6)
--=11787==      by 0x8048554: main (main2.c:13)
--=11787==      Address 0xBB26060 is 0 bytes inside a block
--=11787==      of size 5 free'd
--=11787==      at 0xB9040B1: free (in ...memcheck.so)
--=11787==      by 0x8048541: main (main2.c:11)
```

注意，内存调试器给出了错误语句的位置（第13行），以及在何处进行了无效的内存释放（第11行）。

下一个bug在第14行，它释放了mystr2指向的内存。这是非法的，因为在第11行已经释放了该内存：

```
--=11787== Invalid free() / delete / delete[]
--=11787==      at 0xB9040B1: free (in ...memcheck.so)
--=11787==      by 0x8048562: main (main2.c:14)
--=11787==      Address 0xBB26060 is 0 bytes inside a block
--=11787==      of size 5 free'd
--=11787==      at 0xB9040B1: free (in ...memcheck.so)
--=11787==      by 0x8048541: main (main2.c:11)
```

最后，还有一个内存泄漏。在第8行变量mystr1被更改之前，它一直指向在第6行为其分配的内存。这个内存一直未释放。由于不再有任何指针指向第6行中分配的这个内存位置，因此它成为一个内存泄漏。

```
--=11787== 5 bytes in 1 blocks are definitely lost in
--=11787== loss record 1 of 1
--=11787==      at 0xB903B7C: malloc (in ...memcheck.so)
--=11787==      by 0xBA7163F: strdup (in .../libc.so.6)
--=11787==      by 0x8048504: main (main2.c:6)
```

经验总结

- 内存调试器很容易使用。
- 内存调试器是检测内存泄漏的最有效工具。
- 内存调试器可以检测内存访问错误：缓冲区溢出、错误的数组索引和空指针访问。
- 内存调试器可以发现对未初始化内存的读取错误，从而防止程序发生不可预测的行为。
- 使用内存调试器来检测内存管理例程的错误使用。

- 在使用C字符串和<string.h>中的函数 `strdup()`或 `strcpy()`时，应使用内存调试器来调试程序。

4.5 结合使用内存调试器和源代码测试器

当内存调试器报告一个错误时，它会给出上下文信息，例如调用链（向上到一个特定长度）。这些信息可能并不足以理解为什么会发生bug。因此，内存调试器提供了用来连接源代码调试器（如GDB）的挂钩，以便进行更深入的分析。

下面在我们的示例中使用源代码调试器。Valgrind报告的第一个错误是`main1.c`的第9行 (`A[n] = n`) 的无效写操作。这条语句位于`for`循环中，因此无法立即看出变量`n`的值，这时可以利用源代码调试器来查询变量值。

在使用Valgrind时，可以使用`--db-attach=yes`选项来连接源代码调试器。使用此选项后，Valgrind将在每个报错的地方停止程序的执行，并询问用户是否连接到源代码调试器：

```
> valgrind --tool=memcheck --leak-check=yes \
--db-attach=yes ./a.out
```

在Valgrind停止并询问是否连接源代码调试器时，确认连接，调试器开始工作。然后查询`n`值，结果为100。默认调试器是GDB。使用`--db-command=<command>`选项可以指定另一个调试器。

Purify有一个类似的概念，即在报告错误之后立即调用`purify_stop_here()`函数。启动源代码调试器，在`purify_stop_here()`中设置一个断点，运行程序。源代码调试器将直接停止在引起内存错误的函数中。Insure++也有一个相同的概念，即调用`_Insure_trap_error()`函数。

结合使用内存调试器和源代码调试器

- 如果单独使用源代码位置和栈跟踪无法提供足够信息来定位bug，则可以结合使用源代码调试器和内存调试器。
- 所有的内存调试器都有用于在需要时连接源代码调试器的API。可以阅读用户指南来查看如何使用。
- 使用源代码调试器来检查变量值和调用调试函数。

4.6 减少干扰，排查错误

内存调试器都有排查错误消息的机制，方法是将它们从生成的报告中过滤出来。利用这种机制，可以编写精确的排查规则。这些规则遵守某种语法，而且有两种生成方法，一是用交互工具生成，二是作为文本输入到排查文件中。在大多数工具中，可以通过以下几种方式过滤。

- 错误类型——示例：排查所有对未初始化的内存的访问。
- 函数调用链——示例：排查调用链...->`A()`->`B()`->`C()`中发生的所有数组越界写错误。

- 对象或源文件名称。

例如，在Purify中，排查规则存储在主目录或程序目录的.purify文件中。规则如下所示：

```
suppress UMR          # uninitialized memory reads, all
suppress ABW ...;A;B;C # array bounds write, on call chain
suppress MLK "myleak.c" # memory leaks, in file
```

现在，理论上一个好的想法是要求应用程序及其所有组件“完全纯净”，或是要求内存调试器实现这种类似的结果。最好的方法是分析每个内存错误、修复它并重复这个过程，直到没有错误。

在实践中，这通常是不切实际的。应用程序将包含一些无法查看源代码的操作系统库和第三方库，以及一些包含开发人员不愿意或无法修复的已报告的内存错误的库。在运行内存调试器时，这样的库可能会产生数千条错误消息。在这样大量的错误消息中筛选出与自己的代码有关的错误是不切实际的。此外，如果错误报告总是包含数千条消息，就无法快速检测到是否有新的内存bug，因为这些消息将被淹没在大量干扰消息中。因此，大多数软件项目都会创建一个排查文件，排查那些无害的或无法修复的错误消息。

但是，不要自动排查所有与第三方库有关的错误，原因如下。我们可能以错误的方式使用了库函数。例如，可能使用了错误的函数参数，或者未遵守正确的调用顺序。有些函数必须在调用之前分配缓冲区，有些函数则自己分配内存，但程序员可能负责释放为返回参数分配的内存，或调用清除函数。因此，在过滤每组错误消息之前要再次检查它们。

此外，在使用排查文件时，可能会不小心过滤掉了合法的错误消息。每隔一断时间，在更新程序中的库或使用新的编译器或内存调试器版本时，都要重新构建排查文件。

4.7 何时使用内存调试器

在整个软件开发过程中使用内存调试器可以提高软件质量，因为在将软件交付给客户之前内存调试器会捕获很多bug。同时，正如前面演示的，使用内存调试器只需做很少的工作。建议在以下几种情况中使用内存调试器。

- 将软件移植到新操作系统上时。
- 程序崩溃时。
- 开始调试一个“奇怪的”bug时，即问题有些类似于“这根本不可能”、“这是一个多么奇怪的栈跟踪”或“这个函数从来没发生过问题”。
- 为回归测试的一部分。

4.8 约束

4.8.1 测试用例应该有很好的代码覆盖率

内存调试器是用于查找运行时bug的工具。测试用例必须具有很好的代码覆盖率（code

coverage)，因为只有当程序执行到有bug的语句时，工具才能检测到bug。

4.8.2 提供更多计算机资源

内存调试器是一种资源密集型的工具。由于在每个数据结构中都添加了调试记录信息，因此动态内存的使用显著增加，一般会增加2~4倍。用于检查所有内存访问的调试记录还增加了程序的运行时间（通常是一个数量级）。最后，如果内存调试器基于对象代码插装，则需要大量磁盘空间来缓存系统和程序的运行库。

4.8.3 可能不支持多线程

多线程程序可能会干扰内存调试器。像POSIX Threads这样的常用线程包可能受支持，但不常使用的线程包则可能会导致问题。

4.8.4 对非标准内存处理程序的支持

程序可能包含其自己的低层内存处理程序，其中定义了malloc()和free()这样的函数。使用此内存处理程序时，系统可能在内部调用sbrk()或mmap()函数，其作用是从操作系统接收内存块。

使用自定义内存处理程序有3个原因，一是在特定应用领域具有更好的性能，二是需要更少的空间，三是某一程序模块可能很不完整，或者无法维护，以至于用free或delete无法安全地释放内存的分配。而一旦模块完成，内存管理器中的全部内容在一个步骤中释放，并不会考虑用内存管理器分配的任何数据结构。

内存调试器无法检测到非标准的内存处理程序的使用，而且无法可靠地报告新内存管理器分配的那些数据结构所发生的内存泄漏或访问冲突。但是，内存调试器通常有一个API，利用此API可以将调用注册给自定义内存处理程序，以便检测内存泄漏、缓冲区溢出和未初始化的内存读取。更多细节可参考Purify、Insure++和Valgrind的文档。

经验总结

- 使用排查文件来减少那些来自于无害或无法修复的bug的错误消息。
- 内存调试器是运行时工具，它只能在程序执行含有bug的语句时才能检测到bug，因此测试用例必须具有很好的代码覆盖率。
- 内存调试器将极大减慢程序的执行速度，并增加内存的使用。
- 某些线程包与内存调试器不兼容。
- 可以使用API来集成非标准内存处理程序。



本章介绍内存剖析，也就是调试那些占用过多内存的程序。过量的内存使用有多种原因，例如无效的数据结构、忘记了释放内存，或只是对程序所需内存量估计不足。过量使用内存还会增加程序的运行时间，原因是程序必须访问内存而不是速度更快的缓存，甚至可能是主存储器（分页）溢出，因此本章内容可看作是对第6章介绍查找性能bug的提前预热。

5.1 基本策略——主要步骤

这里介绍一种内存剖析方法，它分4个步骤。

内存剖析的分步方法

- 第1步 检查是否有大的内存泄漏。
- 第2步 估计预期的内存使用。
- 第3步 用多个输入来测量内存使用随时间的变化。
- 第4步 查找使用内存的数据结构。

第1步是检查是否有内存泄漏或破坏。如果能够用内存泄漏检查器来直接搜索，就不要系统性地全面搜索程序内存使用的时间和位置。如第4章所述，可以使用像Purify或Valgrind这样的工具，并且在程序中尽早释放不再需要的数据结构。

在检查完泄漏之后，第2步是粗略估计程序的内存使用。这个估计可作为后面测量的合理性检查。

第3步是查明内存的使用时间，并大体判断在程序流中使用内存的位置。测量总的内存使用如何随时间增加，以及它与输入数据有何关联。注意，要使用有意义的输入数据进行这些测量。

最后是第4步，识别使用内存的语句和数据结构。

作为这种分步内存剖析方法的一部分，本章将讨论可用的工具，并介绍如何为程序编写插装代码，以及如何解释结果。最后举例演示这种方法。

5.2 示例：分配数组

第一个示例是testmalloc.c，它是一个非常简单的程序。此程序执行一个循环，并在动态内存中分配一个大的数组，然后释放该数组。完整的源代码在附录C.1中，代码中包含#ifndef USE_NEW语句，以使用C++的new/delete或C的malloc()/free()语句。本章中将使用C++的new/delete语句，用-DUSE_NEW标志编译代码。在移除#ifndef语句之后，这些都是重要的语句：

```

42  for(i=0; i<iterations; i++) {
43      wait_for_input("before malloc: ", ...);
45      myarray = new int*[n];
49      for(j=0; j<n; j++)
51          myarray[j] = new int[blocksize];
56      wait_for_input("after malloc: ", ...);
57      for(j=0; j<n; j++)
59          delete [] myarray[j];
66      delete [] myarray;
70  }

```

程序有3个参数，当第一个参数被设置为i时，程序将在每个内存分配和释放步骤之前等待。第二个参数n通知程序应该分配多少个大小为1024（blocksize=1024）的整数块，第三个参数通知程序应该执行多少次循环迭代。每次迭代结束都会释放所有数组，并在下次迭代开始时再次分配。

5.3 第1步：查找泄漏

第1步是对怀疑使用了过多内存的程序运行内存检查器，如第4章所讲的Purify或Valgrind。如果发现存在内存泄漏，则应该修复bug，或者至少估计内存泄漏的数量。对于无法修复的bug，例如系统库中的内存泄漏，可能泄漏的数量只有几个字节，这完全可以忽略。我们通过Valgrind运行testmalloc程序，并验证没有内存泄漏。

5.4 第2步：设置期望值

第2步是对程序的内存使用做一次粗略估计。这一步应该在为软件编写插装代码或用调试工具进行实际测量之前完成。这个估计是一种实际性检查，以防万一后面的测量得出意外结果：测量工具可能没有正确工作、插装代码放错了地方，或者程序在未预计到的位置使用了内存。如果内存使用的测量结果明显高于估计值，则数据存储方式可能有bug，或者存储了一些冗余数据。

我们知道，在很多情况下很难估计内存的使用，如程序可能很大、很老、不是模块化的或者混乱。只有了解了一些程序功能、其内部工作原理和所使用的数据结构的知识后，才能够做出很好的估计。但是，要想获取这些知识，需要进行充分的内存剖析。通常，通过一个迭代过程进行

估计：首先通过检查问题说明、实现说明和源代码做出一个最初的猜想。然后进行一系列的测量，查明在内存中使用最多的数据结构。

那么`testmalloc`示例的估计值是什么呢？内存使用几乎完全取决于第二个参数 n ，它通知程序要分配多少个大小为1 024的整数块。每个整数块分配 $1\ 024 \times 4\text{ B}$ ，因此当 $n = 100\ 000$ 时，程序将分配 $100\ 000 \times 1\ 024 \times 4 = 409\ 600\ 000$ 个字节，约为390.6 MB（1 MB = $1\ 024 \times 1\ 024\text{ B}$ ）。

示例中还有其他需要使用内存的数据结构，即栈和用于保存已分配块的指针的`myarray`数组。但是，与整数块相比，它们很小，可以忽略不计。

5.5 第3步：测量内存使用

本节以`testmalloc`为例显示如何测量程序实际的内存使用。这里将显示内存使用如何随着输入值而变化。

我们并不是只想知道程序的最大内存使用量。在执行期间，程序将经过多个时间间隔或阶段。可以将程序阶段与内存分配和释放关联起来，方法是在各个阶段的边界进行测量。利用经典的分而治之策略，可以对测量进行优化，即查找内存使用的增加和减少最显著的阶段。在这些阶段中插入更多测量点，更容易查找的内存分配和释放的位置。

5.5.1 使用多个输入

在测量程序的内存使用之前，要确保与程序有相关的输入数据。一定要有足够的理由来执行内存剖析，例如，理由是程序“在输入文件很小时运行良好，但在输入文件很大时却很糟糕。”一定要使用能够模拟糟糕行为的输入数据，否则所有后续操作只测量了一些无关数据，而且我们会将注意力集中在程序的无关部分上。

输入数据和内存使用之间有某种关联。第6章将更详细解释对数增加、线性增加、平方增加或指数增加的区别。至少要保证使用3个数据样本。例如，如果只使用两个数据样本，则无法区分线性增加和平方增加。

理解了输入数据与内存使用之间的关联之后，将它与第2步得出的估算进行比较。如果差别很大，例如平方增加或指数增加，而不是线性增加，则将有助于找出程序出错的位置。

此外，应该记录内存测量，并根据初始估计来检查这些值。记录很重要，它们有多种用途，包括检查测量错误、在问题审查期间提供证据，而且以后可以用它们来检查bug修复或软件重构是否实际改进了内存使用，这是最重要的。

5.5.2 在固定时间间隔停止程序

在程序流中找到相关位置（即内存使用最多的地方）的方法是插入停止点并测量内存使用。那么如何插入停止点呢？

在预定的时间间隔观察程序的两种最简单的方式是在源代码调试器中使用断点，或者插入以下代码段：

```
if(getenv("MEM_DEBUG")) {
    char c;
    printf("stage xxx\n");
    printf("hit return to continue\n");

    fflush(stdout);
    c = getchar();
}
```

如果设置了环境变量MEM_DEBUG，那么程序将停止，直到按下Return键。如果未设置环境变量，则程序将照常执行，不会停止。在UNIX中设置环境变量的方法是，csh脚本使用setenv MEM_DEBUG 1命令，sh和bash脚本使用MEM_DEBUG=1; export MEM_DEBUG。在Windows中设置环境变量的方法是使用**Control Panel/System/Advanced/Environment Variables**。

当然，在程序的哪些位置插入停止点由调试者决定。每个主要模块的开始是一个好的停止点。接下来，使用分而治之策略来定位内存使用增加最多的程序段，并在这些程序段中添加更多停止点。

5.5.3 用简单工具测量内存使用

建议首先使用简单工具（例如UNIX实用程序top或Windows任务管理器）来测量总体内存使用。还有一些高级工具，如内存剖析工具（将在5.5.8节讲解），但不会马上用到。开始时，它们生成的数据量可能非常大，而且数据也可能被错误地解析。而利用简单工具，可以得到一个数字，即程序分配的总体内存量。

5.5.4 使用 top

在Linux或UNIX机器上显示内存使用的最简单方法是使用实用程序top。在Windows机器上可以使用Task Manager（任务管理器），或使用top命令，它是Cygwin环境的一部分，隐藏在system/procps包中。为了将程序的输出分开，应该在命令解释器（shell）中运行top，命令解释器与被观察的程序是分开的。相关的列是RSS，它显示了任务的物理内存大小，由代码和数据组成。

在testmalloc示例中，将第二个参数设置为100 000。我们已经估计这个值需要分配 $100\ 000 \times 1\ 024 \times 4 = 409\ 600\ 000\ B$ （或390.6 MB）的内存。在分配内存之前在程序testmalloc i 100000 8上执行top命令的输出如下：

```
> top
...
PID USER PRI SIZE RSS SHARE STAT %CPU %MEM TIME CPU COMMAND
475 wloka 21 312 312 268 S 0.0 0.0 0:00 0 testmalloc
```

在调用malloc()分配了 $100\ 000 \times 1\ 024 \times 4\ B$ 的内存之后的输出如下。RSS列的值从312 B增加到391 MB，符合预期：

```
PID USER PRI SIZE RSS SHARE STAT %CPU %MEM TIME CPU COMMAND
475 wloka 16 391M 391M 300 S 4.0 15.6 0:00 1 testmalloc
```

在调用free()释放动态内存后的输出如下：

```
PID USER PRI SIZE RSS SHARE STAT %CPU %MEM TIME CPU COMMAND
475 wloka 16 488 488 300 S 0.0 0.0 0:00 1 testmalloc
```

5.5.5 使用 Windows Task Manager

在Windows中，比使用top命令更简单的方式是同时按Ctrl-Alt-Delete键打开Task Manager程序，单击**Process**（进程）选项卡，再单击**Image Name**（映像名称）使进程按字母排序，然后查看**Mem Usage**（内存使用）列。这里的内存使用单位是KB。为了查看内存使用随时间的变化，单击Task Manager中的**Performance**（性能）选项卡，并查看**Page File Usage History**（页面文件使用记录）图。图5-1显示了运行`./testmalloc i 100000 8`的输出，并每5秒钟按一次Return键，使程序前进。注意，图5-1显示了计算机上所有任务使用的内存，而不只是testmalloc。只有在所有其他任务的内存使用均保持稳定的情况下，才能清晰地看到testmalloc的运行结果。

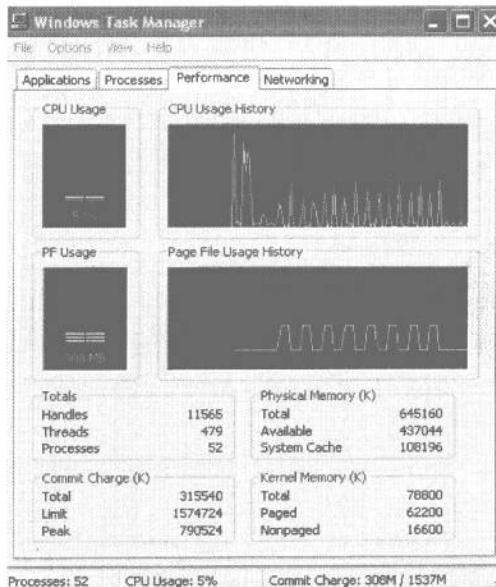


图5-1 运行`./testmalloc i 100000 8`程序时，Windows Task Manager的结果

5.5.6 为 `testmalloc` 选择相关输入值

用不同输入值反复进行测量。将块数减小为原来的1/10，即 $n=10\ 000$ ，则内存使用按预期减少为39 MB。另外，增加或减小迭代次数对内存使用峰值没有影响。这也与估计吻合。

从现在起，我们使用输入值 $n = 100\ 000$ ，迭代次数为8，以触发程序的“正常”行为。几乎所有已分配的391 MB内存都是由于这个 n 值引起的。

5.5.7 确定机器上的内存是如何被释放的

在讨论内存使用的测试工具之前，有必要指出在某些机器上，在释放动态内存时，程序的进程实际上不会将内存还给操作系统。这取决于机器上安装的操作系统、编译器和C/C++运行库。表5-1列出了一些典型开发系统中的内存释放行为。在前面的示例中，机器将内存释放给操作系统。

表5-1 `free()`和`delete`在不同操作系统平台上的行为

平 台	<code>free()</code> / <code>delete()</code> 的效果
SunOS 5.8、GCC 3.3.2	内存不释放给操作系统
Linux Red Hat Enterprise 3.0、GCC 3.3.2	内存释放给操作系统
Linux Red Hat 7.2、GCC 3.3.2	内存释放给操作系统
Suse 10.0、GCC 4.0	内存释放给操作系统
Windows XP、VC++ 8.0 SP1 (.Net 2005)	内存释放给操作系统
Windows XP、Cygwin 1.5.25、GCC 3.4.4	内存不释放给操作系统

在一些平台上，第一次分配之后，程序所使用的内存保持为一个常量（约为391 MB）。`free()`以及所有后续的`malloc`和`free()`调用在top中都没有明显效果，我们仍将看到相同结果：

```
PID USER PRI SIZE RSS SHARE STAT %CPU %MEM TIME CPU COMMAND
475 wloka 16 391M 391M 300 S      4.0 15.6 0:00   1 testmalloc
```

这种行为并不是bug，而是在各种操作系统与编译器组合中动态内存实现方式上的区别。当调用`malloc()`分配内存元素时，C或C++运行库将向操作系统请求一个大内存块，内存元素被置在内存块中的某处，并记录分配了多少内存，以及内存元素位于何处。所有后续的`malloc/free()`调用都发生在这个预分配的大内存块内部。如果这个块中已没有可用空间，则将再次调用操作系统来扩展内存块或分配另一个内存块。

动态内存行为区别形成的原因是内存分配方式的选择。在调用`free()`后，产生了一个完全空的内存块。我们来看一下在这之后将发生什么事情，如`testmalloc.c`示例所示。

一种可能是将内存块还给操作系统，以便其他程序和用户使用。这可以提高内存使用效率，但交还内存块的系统调用要花费时间。在最坏的情形下，在内存块边界上反复调用`malloc/free()`可能导致大量系统调用。因此，处理大内存块的另一种选择是不将它们还给操作系统。这样，我

们会观察到程序的内存使用一直随程序的运行时间而增加。在大多数程序中，C运行库的动态内存释放方式几乎没有实际区别。但是，如果程序在启动时需要大量内存，但在长时间运行中只需较少内存，则忘记释放内存将导致极大开销。

建议读者将编译和运行testmalloc.c程序作为一个快速测试，来看一下操作系统和编译器平台如何处理内存释放，这样就不会从top和Task Manager的结果得出错误结论了。

经验总结

- 在一些操作系统/编译器组合中，用free()/delete释放内存可能不会立即看到效果。
top报告的内存使用并未减少。
- 如有疑问，可以使用testmalloc.c程序来查明当前操作系统/编译器组合是如何处理内存释放的。

5.5.8 使用内存剖析工具

用top来测量内存使用的优点是易于实现，并且在其他工具不可用时，可以使用这种方法。但是，添加插装代码（如使程序等待按键的代码）和做手工记录需要大量工作，特别是当程序中有很多不同模块使用动态内存的时候。利用内存剖析工具能减少这些手工工作量。

内存剖析工具是详细记录内存使用的工具。由于大部分内存剖析工具只是监视通过malloc()/new在堆上分配的动态内存，因此它们也被称为堆剖析工具（heap profiler）。内存剖析工具记录动态内存分配的时间、由谁（调用栈）分配的、大小以及何时、由谁释放的。在程序结束后，内存剖析工具输出图表和日志文件，它们展示了内存使用的细节，并使得内存更容易分配给最大的内存使用者。

附录B中给出了很多可用工具和它们的简介。在Windows中，可以使用AQtime（附录B.5.4）性能和内存调试器。AQtime可以单独使用，也可以集成到Visual Studio中。在Linux上，建议使用Massif（附录B.4.2），它是Valgrind调试和剖析工具套件的一部分。Mpatrol（附录B.5.5）既可在Windows上使用，也可在UNIX/Linux上使用，它是一个开源工具，但功能较少。

本章后面将使用Massif，因为它在内存剖析工具中很有代表性，它是开源工具，其使用模式比较简单。没有特殊的编译器标志，也不需要重新编译或重新链接程序。在执行程序之前，很容易在命令解释器中添加命令。如果用调试信息标志-g编译程序，则Massif将支持在统计输出中引用具体的行。

```
> valgrind --tool=massif ./testmalloc n 100000 8
```

图5-2显示了testmalloc的内存分配和释放行为。图5-2是直接由Massif生成的Postscript文件。程序执行8次循环迭代。在每次迭代中，为一个大数（100 000）分配一个整数数组，然后再释放，因此通过图5-2中的8次峰值可以看出这些迭代。由于内存分配不是通过一次调用malloc

完成的，而是通过一个调用序列（100 000次）完成的，因此峰值具有很陡的斜率。如果在一次调用中完成内存分配，则图形将更像一个方波。

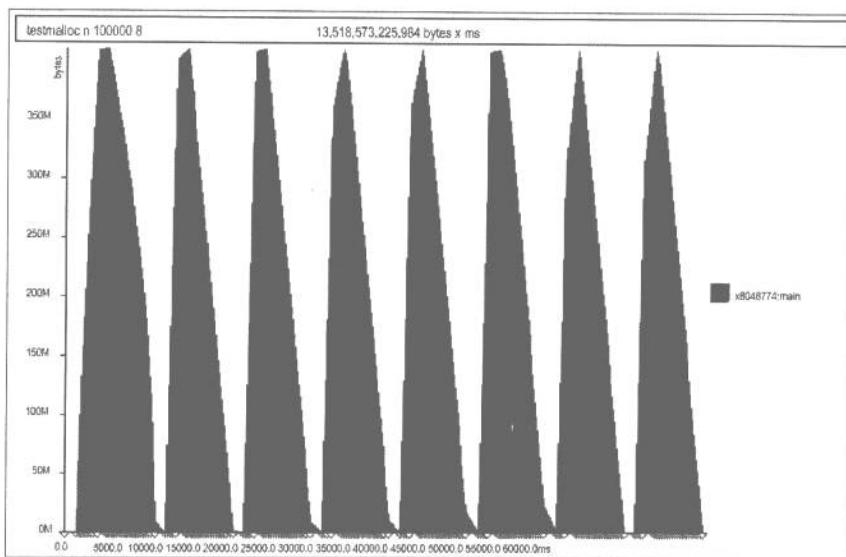


图5-2 valgrind --tool=massif ./testmalloc n 100000 8的Massif输出

Massif的文本输出给出了内存分配的信息。对于每次分配，Massif以函数名形式列出调用链，如果编译时带有调试信息，还会列出源文件名和行号。5.7节将讨论如何使用Massif的这个特性来测量数据结构的内存使用。

以下是testmalloc程序的Massif文本输出：

```
Command: testmalloc n 100000 8
Heap allocation functions accounted for 88.8% of measured
spacetime

Called from:
 84.9% : 0x8048773: main (testmalloc.c:53)
  3.9% : 0x8048746: main (testmalloc.c:47)
...
```

Massif的输出显示大部分内存（约85%）是在第53行分配的，这里调用malloc()分配了一个1 024 KB的整数块。第47行使用了一个较小的内存块，这里分配了指向内存块的指针数组。

5.6 第4步：查明大部分内存被哪些数据结构占用了

第4步的作用是在程序中找到与总体内存使用最密切相关的数据结构和代码位置。至少有两

种详细分析方法：为数据结构编写插装代码，或使用内存剖析工具的输出。

内存剖析工具很容易使用，因为它不需要修改源代码，并且提供了准确而详细的反馈。然而，如果剖析工具检查所有内存分配语句，可能会给出过多的详细数据。

为数据结构编写插装代码

另一种测量内存分配语句（以及分配的大小）的方法是为源代码（特别是数据结构）编写插装语句。辅助函数可以跟踪数据结构的使用频率、大致需要的内存大小，如果可能的话，还可以跟踪哪些语句分配了数据结构。例如，可以为C++类添加一个方法，它给出类对象内存使用的大致值（包括成员的内存使用）。另一个例子是计算活动的类对象数，方法是定义类的所有new和delete操作符并用静态计数器来计算增减。

testmalloc程序太简单了，显示不出来有意义的插装代码，因此下面将在一个更复杂的示例上解释数据结构插装的概念，这个示例就是genindex程序。

5.7 综合练习——genindex示例

该程序的源代码可以在附录C.2中的C++文件genindex.cc中找到。genindex的初始需求是读取一个文本文件，并输出文件中的单词索引。索引是唯一词（unique word）的排序列表，其中每个词后面有一个用逗号隔开的列表，列出单词出现的行号。

正如实际工作中的其他程序一样，genindex有一个需求说明，其中包含开发过程中添加的特性。genindex最初是一个简单的UNIX过滤器程序，它打开一个文本文件，使用一个简单的状态机标记器（tokenizer）将文本分解为单词，并存储每个单词以及它在索引数据结构中的当前行号。

随着时间的推移，增加了几个需求：首先，在任意时刻处理多个文件，其次，在对标记器进行一些问题调试后，添加一个函数来交叉检查生成的索引的正确性。此外，在考虑了动态内存泄漏等一些问题，以及对DIY列表和散列表数据结构的合理怀疑之后，重写程序以使用C++ Standard Library中的容器来实现索引数据结构。

然后在样例输入文件的回归测试数据库上对程序进行测试，用Purify来检查内存泄漏和内存破坏（如第4章所述），并同时通过gcc -Wall和静态分析程序Coverity（将在第12章介绍）来运行它。程序通过了所有测试，但仍然有些问题导致程序运行得很慢，特别是当文件很大和输入文件很多的时候。起初，问题被归咎于用户缺乏耐心和较老的计算机硬件上，但后来我们决定做一次快速的程序分析。

首先，反复执行分析策略的前3步，即确定没有（大的）内存泄漏、计算预期使用并用多个输入来测量内存使用随时间的变化。然后，将注意力集中在第4步上，也就是引入这个示例的实际原因，具体方法是编写插装代码，并测量哪些数据结构和代码位置正在使用内存。

接下来的4小节将依次介绍每个步骤。

5.7.1 核实没有大的内存泄漏

第1步是再次运行Purify测试，这一步通过了。这并不奇怪，因为程序使用了C++ Standard Library中的数据结构，其中并没有任何malloc()或new命令，因此也就不会产生内存泄漏。

5.7.2 估计内存使用

第2步是估计内存使用。这并不像第一个示例那样简单。程序需要在索引表中为文件中的每个唯一词存储一项。这个项由单词和一个整数列表组成，整数代表了该词在输入文件中的位置。假设使用一个双向链表，则每个列表项需要 $2 \times \text{size(pointer)} + \text{size(int)} = 12 \text{ B}$ ^①。在处理输入文件的下一个单词时，有两种选择——如果单词已知，则只需在索引表中的现有项后面添加一个新的列表项，这需要12 B；如果单词未知，则需要用一个新项来存储单词及其第一个列表项，这将需要12 B外加单词的字符串大小。第二种选择需要更多内存，因此最坏的情况是所有单词都是唯一的。

假设所有单词都是4个字符并且都是唯一的，那么一个 $n \text{ B}$ 的文件需要多少数据呢？需要存储文本（ n 个字符）外加 $n/4$ 个列表项（每个项12 B），共计 $n+12 \times (n/4) = 4 \times n \text{ B}$ 。此外还需要一些内存，以可搜索的方式来存储单词，但这个搜索表的开销与项比起来小多了。最后，我们估计所需的内存是输入文件的几倍。

第二个重要的估计是内存不应随着输入文件的数量增加，原因是在两个文件的处理之间没有需要存储的数据。在每个文件结束时输出索引，并且在该时刻所有已分配内存都应该被释放。只有文件名是在整个运行时存储的，但这项开销远远小于处理文件所需的内存。

5.7.3 测量内存使用

第3步是用不同的输入文件来测量内存随时间的使用情况。插入并调用wait_for_input()代码，可以在处理完每个文件之后暂停程序。此代码是通过编译时使用PAUSE INDEX选项来启用的。

```
> g++ -g -DPAUSE_INDEX -o genindex genindex.cc  
> ./genindex input1.txt input1.txt input1.txt input1.txt
```

在读取第一个文件之后，top的输出（在这里没显示）是4.6 MB，每读取一个20 KB的input1.txt文件增加2.3 MB，最后的总数是11.4 MB。测量的每个文件的内存使用是2.3 MB，大于估计值80 KB（输入文件大小为20 KB）。此外，测量的内存使用随着文件数的增加而增加，与估计的保持稳定不符。这两个不符之处说明有错误。

^① 在我们的示例中假定使用32位指针，如果是64位指针，则需要20 B。

5.7.4 查找使用内存的数据结构

接下来，详细看一下哪些数据结构正在使用内存以及使用了多少，首先用Massif进行测试，然后再用插装代码进行测试。调用Massif，用4个输入文件来运行genindex：

```
> g++ -g -o genindex genindex.cc
> valgrind --tool=massif \
./genindex input1.txt input1.txt input1.txt >log
```

图形输出如图5-3所示。Massif确认了程序的内存使用量快速增加，而且总的内存使用量大于11 MB。Massif的文本输出显示了大部分内存被string数据类型使用了：

```
Called from:
93.0% : 0x3AA1F460: std::string::_Rep::_S_create(unsigned,
unsigned, std::allocator<char> const&)
(in /usr/lib/libstdc++.so.6.0.6)
```

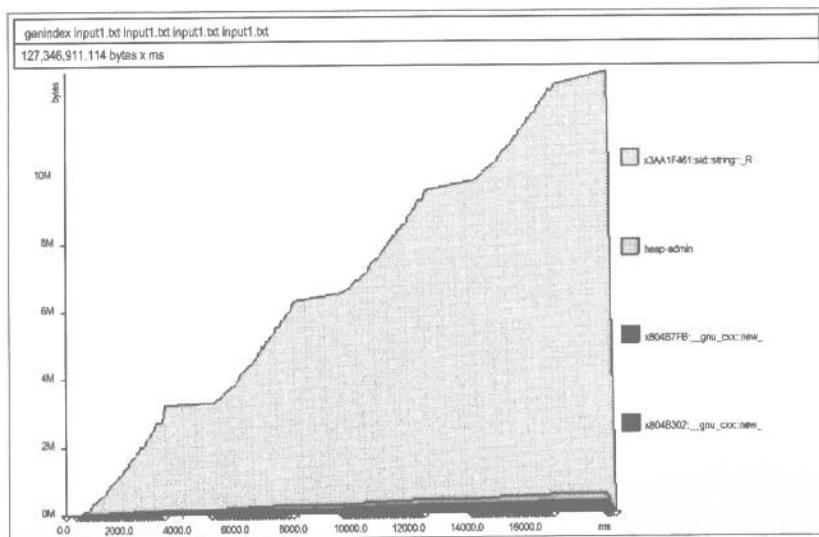


图5-3 genindex的Massif输出：内存使用增加很快，每20 KB的输入文件增加2.3 MB

为了分析哪种数据结构正在使用内存，在程序中插入插装代码来估计内存的使用。为了保持简单，`print_memory_stats()`方法在每次读取文件之后遍历主数据结构`wordindex`。粗略地估计内存使用的下限：假设主要负载是索引图的新增元素的大小，新增元素是由字符串和整数列表组成的一对元素。这种近似忽略了这样一件事：索引图中通常会创建一个红黑树数据结构，支持对关键元素的快速搜索。对于双向链表容器，像前面那样假设每个列表元素的主要负载是一个整数和两个指针。

程序生成以下调试输出：

```
-- memory size for index of 'input1.txt' file size=20016:
--     filename=14 wordindex=47211 lines=2340979 total=2388204
...
-- memory size for all data structures: 9552816 bytes
```

大部分内存被用于存储文本行的成员变量使用了。此变量是在开发验证代码期间添加的，这里验证代码用成员函数`verify_index()`对索引进行交叉检查。一个有20 016个字符的文件需要2 300 000 B是不合理的。

代码检查揭示的问题是唯一对`lines`数据结构执行实际写操作的代码位于`scan_file()`的第126行，即：

```
126             lines.push_back(buffer);
```

变量`buffer`存储当前输入行，因此它随着从输入文件读取的每个字符而增长：

```
105 int FileIndexType::scan_file(char *fname) {
...
112     string buffer;
...
119     while(1) {
120         c = getc(fp);
121
122         if(c == EOF || c == '\n') {
123             add_to_index(newword, current_line);
124             newword = "";
125             current_line++;
126             lines.push_back(buffer);
...
133         else if(c == ' ' || c == '\t' || c == '\r') {
...
137             buffer = buffer + (char) c;
138         }
139         else {
...
141             buffer = buffer + (char) c;
142         }
143         filesize++;
144     }
```

在输入文本行的末尾没有重置`buffer`变量，因此它不仅包含当前行，还包含所有前面的行。更糟的是，`lines`数组的内存大小将呈平方增加，因为每个新元素将包含前面所有的行。

注意，这个bug并不影响`verify_index()`的错误检查，因为`verify_index()`的任务是核实索引中的每个单词是否出现在它在文本文件中对应的行上。字符串搜索可能要花费很长时间来运行。

调试`genindex`的内存使用的下一步是修复bug，即在行的末尾重置缓冲区：

```
119     while(1) {
120         c = getc(fp);
121 }
```

```

122     if(c == EOF || c == '\n') {
123         add_to_index(newword, current_line);
124         newword = "";
125         current_line++;
126         lines.push_back(buffer);
127         ...
128         buffer = ""; // ----- this is the bug fix

```

附录C.2中列出的代码已准备好了bug修复，用-DFIX_LINES标志来编译即可激活它。接下来，用Massif来重新分析插装代码。修复bug之后的Massif输出如图5-4所示。

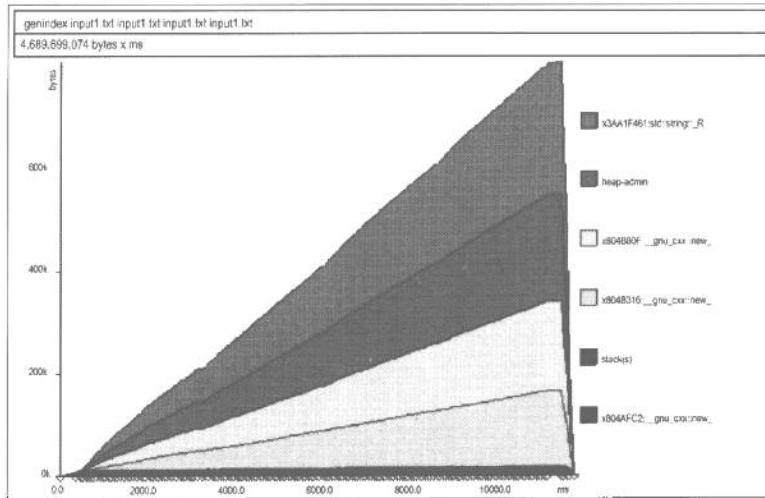


图5-4 用-DFIX_LINES编译genindex后的Massif输出：每个文件的内存使用量减少到200 KB，但总的内存使用仍在增加

```

> g++ -g -o genindex -DFIX_LINES genindex.cc
> ./genindex input1.txt input1.txt input1.txt >log
-- memory size for index of 'input1.txt' file size=20016:
--   filename=14 wordindex=47211 lines=19318 total=66543
...
-- memory size for all data structures: 266172 bytes

```

可以看到，现在每个文件的内存使用是合理的，运行Massif时每个文件使用的内存均约为200 KB，运行插装函数时约为68 KB。

注意，我们对map数据结构的开销估计过低，仅为真实开销的1/3。再来看一下图5-4所示的Massif图，程序显然不具有好的可扩展性，因为处理每个文件都会导致内存增加。进一步的代码检查显示了原因，即为每个文件创建的C++对象在程序运行时一直保持。

接下来，修改程序的执行流，立即为每个文件创建、验证和输出索引。然后，清除容器释放内存。用-DCLEAR_INDEX标志编译可激活修改后的代码。

```
> g++ -g -o genindex -DFIX_LINES -DCLEAR_INDEX genindex.cc
> ./genindex input1.txt input1.txt input1.txt input1.txt >log
-- memory size for index of '' file size=0:
--     filename=4 wordindex=0 lines=0 total=4
...
-- memory size for all data structures: 16 bytes
```

Massif的输出如图5-5所示。注意，我们仍在向索引容器添加元素，因此看到在Massif中内存的使用随时间略有增加。由于插装代码只测量负载，而负载已被清除，所以它无法精确地测量这么小的增加。进一步修改代码，移除`FileTypeIndex`数组，并保持重用一个`FileTypeIndex`类型的变量作为存储单元。

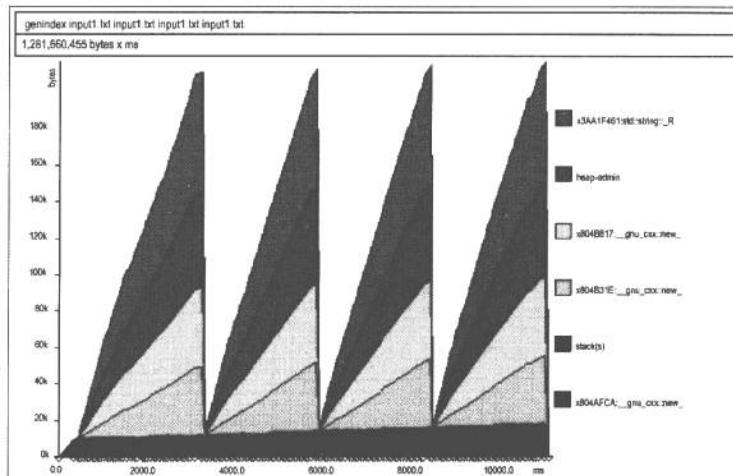


图5-5 用`-DFIX_LINES -DCLEAR_INDEX`编译`genindex`的Massif输出：总内存使用和每个文件的内存使用都减少了

经验总结

- 使用4步方法来查找内存bug。
 - (1) 首先确定没有内存泄漏（第1步）。
 - (2) 估计每个输入文件为20 KB的一个文件序列约需要80 KB内存（第2步）。
 - (3) 实际的内存使用（第3步）远远超过估计值，而且随着输入文件数的增加而增大。
 - (4) Massif内存剖析工具（第4步）查明大部分内存被`lines`变量使用了，代码插装也证实了这一点。随后，代码评审指出了实际的bug，即丢失了重置行缓冲区的语句。在修复bug之后，实际内存使用接近估计值，结果可接受。
- 第二轮测量和代码修改实现了进一步改进。

解决性能问题



本章讨论如何查找与程序运行时性能有关的bug。最常见的情况是程序运行时间过长，因此需要查明如何提高程序的运行速度。在有些情况下，bug导致程序运行得过快。例如，由于程序的某些函数没有被调用，因此程序运行得非常快，但没有执行预期操作。

本章首先介绍如何分析性能问题。除了简单的时钟测量方法和time命令以外，还将讨论一些独立于任何调试工具的常见技术。接下来介绍一些性能分析工具，也称为剖析工具(profiler)，并举例说明它们的应用。最后以I/O性能问题的调试结束本章。本章重点强调并详细讨论适用于多种性能问题的重要经验教训。

6.1 分步查找性能 bug

软件开发人员可以使用一些高级工具来可视化和分析程序的运行时行为，6.2节将讨论这些工具。但经验表明，做一些前期的基本分析也是很重要的，而不能只依靠特殊工具。

在使用剖析工具之前，做一些前期分析

- 第1步 创建一组大小不等的有意义的测试用例。确定程序能正确运行：没有崩溃，并得到正确的输出值。
- 第2步 用简单工具（如/usr/bin/time）来测量运行时间与测试用例大小之间的关联。确定程序使用了正确的算法。
- 第3步 选择一个能够揭示主要的运行时瓶颈的测试用例。继续用剖析工具来识别产生瓶颈的原因。

6.1.1 执行前期分析

通过执行前期分析来回答以下问题。

- (1) 是否有能够再现性能问题的测试用例？
- (2) 程序是否计算出正确结果？

(3) 是否能够修改测试用例来观察问题规模、输入数据和环境的影响？

(4) 性能问题是否确实是由于程序引起的，而不是某些其他因素的副作用，如网速慢、许可服务器速度慢或内存低（分页/交换）？

(5) 是否使用了适合问题规模和输入数据的正确算法？

执行前期分析并回答上述问题的优点是可以合理地确定是否有bug存在，并且可以用测试用例来可靠地再现bug。产生bug的一个原因可能是使用了错误的算法。通过分析程序运行时间如何随着输入规模增加，可以发现算法错误。性能bug的另一个原因是实现问题，这些问题可以利用后面介绍的性能测量工具来查找。

前期分析只需用纸或文件来记录结果，用一种简单方法来测量时间，还需要有一个运行程序的环境（很多时候要运行很多次）。

6.1.2 使用简单的时间测量方法

最基本的时间测量工具是能够显示秒的时钟。在有些情况下，如后文介绍的一些高级工具无法工作，或测量结果不准确需要交叉检查时，使用此方法。

`time`是一个易于使用的命令，它随同Cygwin包一起安装在UNIX、Linux或Windows计算机上。该命令位于`/bin`或`/usr/bin`目录中，或者内置在命令行解释器中。将`time`命令放在程序名前面，如下所示：

```
> time <program> [<program_args...>]
```

`time`命令运行程序，并得到以下输出：

```
9.179u 1.363s 0:10.89 96.6%
```

第一个值u是程序的用户CPU时间（user CPU time），第二个值是系统调用所花的时间，第三个值是程序开始和结束之间占用的实际时间。注意，准确的输出取决于操作系统以及使用的是`time`命令的哪个变体，`sh`、`csh`和`/usr/bin/time`的内置`time`命令行解释器命令都使用不同的格式，而且显示结果的顺序也不同。选项`-p`强制大部分（但并非所有）变体使用相同的格式。

本章中的测量表将使用CPU时间，因为这是我们最感兴趣的量，即程序在计算中用的时间。系统调用的效果对本例不起作用。实际的时间值包括被其他程序打断的时间或程序的等待时间，如等待用户输入、网络传输完成或内存分页的时间，等等。

本章使用的示例是`isort.c`程序，见附录C.3。该程序有两种用于对浮点型数进行排序的算法，分别是插入排序（insertion sort）和快速排序（Quicksort）。

6.1.3 创建测试用例

如果测试用例能反复运行，而且可以在不同机器和环境中向其他人演示，那么会更容易找到

问题。如果没有这样的测试用例，那么很可能虽然改进代码段时做了大量工作，但对总体运行时间却没有明显影响。实际上，看起来很合理的代码修改也许会使得程序运行得更慢。

`isort.c`示例中有一个从两种算法中进行选择的命令行参数，一个定义要对多少个数字进行排序的参数，还有一个在多次迭代中运行排序算法的参数。此外，还有生成要排序的随机数数组的代码。

6.1.4 使测试用例具有可再现性

为了在示例的运行过程中获得可再现的时间测量结果，随机数总是从相同的种子值开始，因此`isort.c`程序总是具有相同的输入值。如果不这样，不同的运行将具有不同的开始条件（输入值），并且需要不同的运行时间，因为每次排序的难易程度各有不同。这将在测量中带来一些“噪音”。如果在这种噪音干扰下测量优化效果（运行时间上的改进），则需要运行很多次测试来得到平均值。

6.1.5 检查程序的正确性

开发过程中经常会出现bug，而且也很容易发生计算错误。例如，在`isort.c`程序中，假设`isort()`函数顶部的条件是错的：

```
if(n >= 1) return;
```

这样，无论数组多大，程序的运行时间都相同。这种情况的症状是很明显的，即结果数组中的数字未被排序。在本示例中，解决方法是多用点时间编写一个单独的函数，验证数组结果中的数字是否已排序。只有合理地确定程序正常运行之后，才能继续进行性能分析和优化。

6.1.6 创建可扩展的测试用例

常见的错误是只根据一个测试用例来分析性能问题，然后修改代码以提高程序运行速度。测试的设置应该支持多个测量，允许通过改变问题的规模、更改输入数据或环境来更改程序的参数。修改测试设置时，应该坚持可重复和可再现的原则。

利用可扩展的测试用例，我们可以在正确的操作点上进行测试。在实践中，可能必须根据问题规模来选择完全不同的算法。对于给定示例，应该改变输入数组的大小，以观察问题规模对排序算法的影响。从排序算法的文献中可以知道，当问题规模较小时($n \approx 10$)，插入排序比快速排序更有效。此外，当给出部分排序的输入数组时（我们示例中就是这样实现的），快速排序的性能将更加恶化，因此通常需要添加一个用于随机选择主元素（pivot element）的机制。

在极端情况下， n 的值可能非常大，数组无法载入计算机的缓存或主内存中。在这种情况下，程序的运行时性能将大幅降低，因为部分数组将从主内存中被交换到硬盘的虚拟内存中。一种解决方案是更改算法，将数组分割为适合载入主内存的片段，然后对各个片段进行排序，再将排序

后的片段合并为一个排序后的数组。

6.1.7 排除对测试用例的干扰

在创建了一组可再现的测试用例后，可以试着进行一些小的、可控制的更改，并观察它们对程序运行时间的影响。能够可靠地再现试验结果是很重要的，这样可以确定结果只取决于程序，而不是其他因素。以下是可能影响运行时测量的一些因素。

- **运行时间过短：**确保程序至少运行5秒钟（用户时间加上系统时间），最好是运行几分钟。如果程序运行时间过短，`time`命令测得的结果就不够准确。建议将运行时间保持在10分钟以内。如果运行时间超过1小时，则测试用例将很难使用。
- **文件I/O：**在很多情况下，程序需要执行一些文件I/O操作，例如将一些调试消息输出到`stdout`或日志文件。应该尽可能关闭输出语句。在最坏的情况下，执行输出语句比执行程序核心功能花费更多时间。如果执行了过多的输出语句，那么实际测量的将是磁盘速度，或图形用户界面的绘图性能，而不是算法的CPU时间。
- **系统调用：**如果可能的话，避免在测量运行时间时调用操作系统的例程，如`getenv()`调用和`system()`、`exec()`、`fork()`调用（这些调用将生成其他进程），等等。很难预计这样的操作系统调用需要用多长时间，因此为测量带来了一些不确定性。
- **没有足够的主内存：**确保运行程序的机器有足够的内存。如果内存少，操作系统可能会临时将部分程序从内存移到磁盘上，然后再移回来（分页、交换）。这可能间接地影响要测量的CPU时间，而且肯定会增加实际的运行时间。
- **CPU时钟频率不稳定：**一个例子是现代CPU可以自动以较低的时钟频率运行，以减少热量的产生，但这会导致性能测量完全混乱。应该禁用这项特性，至少运行某种监视软件，确保CPU频率在测试期间保持稳定。在程序开发期间，还有其他一些不可预见的因素会影响性能。
- **其他进程：**确保在测试运行时间时，计算机上只有较少的用户（最好是没有）。随着计算机上负载的增加，报告的实际时间必然会增加。更糟的是，由于缓存未命中和内存交换，用户和系统运行时间也会增加。

6.1.8 用 `time` 命令测量时可能会发生错误和偏差

计算机上的时间测量也可能发生错误和偏差。在现代操作系统上，许多程序以时间片(`time-sliced`)方式在同一个CPU上运行。其他用户程序、设备驱动程序、服务和(我们最喜欢的)屏保都在争用同一个CPU的时间。

`time`命令将通过报告在CPU上所用的时间来排除其他程序的大部分影响，但应注意到这是不准确的。随着计算机上负载的增加，缓存未命中和内存交换将增加`time`报告的运行时间。

作为一种最低限度的保护，每次测量应至少重复3次，注意结果并检查变化。当测量结果变化较大时，可以通过从计算机上移除用户或任务来排除系统负载，也可以在另一个系统中运行测量。如果怀疑网络负载影响了测量，可以断开机器的网络连接，或在网络流量较小时运行测量。

经验总结

- 用 `/usr/bin/time` 或类似工具测量运行时间可能有错误。
- 环境可能对程序有很大影响，并且使实际运行时间（挂钟时间）的测量失去意义。要注意文件I/O、系统调用、低内存（交换/分页）、CPU时钟频率不稳定或其他进程的影响。
- 每个测试至少运行3次，并检查变化。
- 选择一个使程序至少运行5秒钟而小于10分钟的测试用例。

6.1.9 选择一个能够揭示运行时间瓶颈的测试用例

一旦有了好的、可扩展的测试用例和可靠的数据，就应该测量运行时间与输入和问题规模有何关联。图6-1显示了几种典型情况。很多简单程序的运行时间与输入数据或问题规模(n)呈线性函数关系 $O(n)$ ，如曲线A所示。曲线B显示了运行时间的快速增加，如平方($O(n^2)$)或指数增加($O(2^n)$)，这一般会涉及优化问题。曲线C显示了对数增加($O(\log(n))$)，通常是由搜索算法造成的。我们预计每条曲线都有某个常量偏移，因为即使对于最小的测试用例，运行时间也永远不会减小到0。

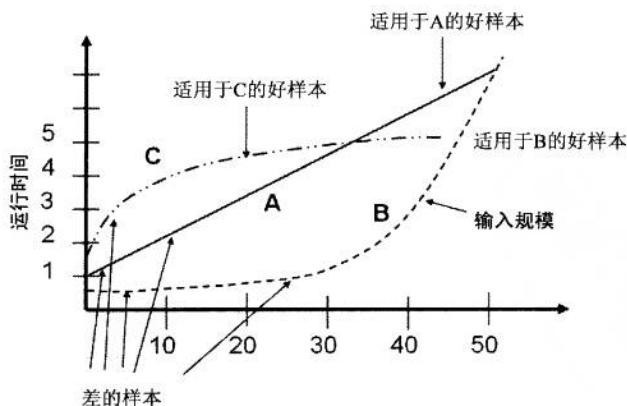


图6-1 选择一个能够很好地揭示运行时间瓶颈的测试用例

在现实世界中，程序的运行时间与输入规模之间的关联更为复杂。可能很难测量输入规模与运行时间之间的关联，而且要收集每个数据点可能需要大量工作，但这些工作是值得的。即使程

序只是大致接近图6-1中曲线B的形状（即平方增长），那么在不断增大输入规模时，将会收集到更多的测量结果。在运行时间开始显著增加时，程序也就失去了它的使用意义。这就是出现性能bug的地方，并且值得我们花力气去分析。适用于曲线B的好测试用例（输入数据样本）是这样的：测试用例的运行时间开始失控，但仍然足够小，以便可以用剖析工具来处理。

随着输入规模的不断增大，符合对数曲线C的程序的性能变化也非常明显。如果是这样，曲线开始变得平坦的地方就是用于剖析的好测试用例（数据样本）。

在线性增加的情况下（曲线A），当选择用于剖析的测试用例时，应保证总运行时间至少比运行最小测试用例所需时间高出一个数量级。

如果不知道出现的是哪种类型的曲线，则要确保使用至少3个数据样本。通过对isort.c示例以及与性能测量有关的其他问题进行分析，可以得出以下结论。

- 如果只做一次测量，则所有程序都运行常量时间。
- 如果只做一次或两次测量，则所有程序都呈现出线性增加，即运行时间是问题规模的函数（曲线A）。

假设从曲线B上任意取两个样本，然后像曲线A一样在它们之间画一条直线。如果只取这两个样本，则无法区分A、B和C。只有再增加一个样本，才能明确区分它们。

6.1.10 算法与实现之间的差异

如果读者读过某本基础的算法书（例如[Cormen01]和[Sedgewick01]）的介绍性章节，那么应该知道为什么要选择插入排序作为例子来证明前期性能问题分析的必要性。原因是插入排序对于合理的问题规模来说是一种错误的算法（合理规模是指对10个以上的数字进行排序）。但是，性能测量工具不会告诉我们这一点。相反，这样的工具以及逐步精化方法将给出以下建议。

- 优化insert()函数，使其尽早退出for循环，性能可提高50%。
- 将less()和swap()编写为内联函数可提高性能。
- 将insert()编写为内联函数并删除从isort()的递归调用，即将算法编写为一个包含两个for循环的函数可提高性能。
- 用汇编代码或宏代码来编写sort()函数可提高性能。

问题在于，以上列出的任何一项改进都无法得到满意的结果。为了查看插入排序算法出现了什么问题，需要通过不同的问题规模进行一些实验。

首先，运行程序，对大小为100的输入数组进行一次排序，表6-1显示了结果。可以看到两个结果：在任何合理的时间测试中程序都完成得过快，而且负责将随机数载入输入数组的测试代码可能影响了测量。因此，迭代参数（排序的次数）被设置为“合理的”大小。我们的示例选择10000。这样选择有两个目的，一是使排序次数远远大于生成输入数字的次数，二是程序在PC机上至少运行数秒钟。10000这个值不只是根据示例选择的，而是综合考虑了示例和所使用的计算

机而选定的值。

表6-1 插入排序的运行时间采样，选择的运行时间过小

程 序	命令行参数			用户运行时间 (s)
	算法	数组大小	迭代次数	
isort	i	100	1	0
isort	i	800	1	0

表6-2中的结果告诉我们，插入排序算法出现了严重的错误，因为当问题规模增加2倍时，运行时间增加了4倍。我们可以计算一下插入排序的运行时复杂度，或参考上面提到的两本算法书，就会发现这实际上并不奇怪：算法复杂度是 $O(n^2)$ ，其中 n 是要排序的数组的大小。

表6-2 插入算法的运行时间采样，选择的运行时间是正确的

程 序	命令行参数			用户运行时间 (s)
	算法	数组大小	迭代次数	
isort	i	100	10000	1.9
isort	i	200	10000	6.8
isort	i	400	10000	25.7
isort	i	800	10000	102

对此算法做再多调整也无济于事。对于大数组，运行时间将变得非常长。上面提到的两本算法书中介绍了快速算法的复杂度更低一些，即 $O(n \log(n))$ 。这意味着随着输入数组的增大，排序时间比线性增加略差一些。

使用参数q可以将isort.c程序切换为使用快速算法。将迭代次数提高10倍，这使得程序至少运行数秒钟。结果如表6-3所示。

表6-3 快速算法的运行时间采样

程 序	命令行参数			用户运行时间 (s)
	算法	数组大小	迭代次数	
isort	q	100	100000	3
isort	q	200	100000	6.2
isort	q	400	100000	15
isort	q	800	10000	33

注意，示例中的quicksort()实现肯定不是最优的。为了保持示例简洁且便于阅读，没有进行以下改进。

- 将partition()编写为内联函数。
- 将swap()编写为内联函数。
- 随机选择partition()中的主元素，确保要排序的数组被划分为两个大小相等的部分。

在给定的实现中，我们使用数组的最后一个元素。这种方法可以有效地用于随机数的排

序，而部分排序的数组会使运行时间达到 $O(n^2)$ 。

6.2 使用剖析工具

剖析工具显示了在执行过程中程序的时间都用在了什么地方。剖析工具的一些其他常见特性还包括列出每个函数的调用次数、调用的分布情况、函数被调用的时间以及每个函数的调用者。测量的精确程度各有不同，从子函数（所有工具都支持）到程序块、行、表达式，再到个别的机器指令。

正如下面几小节所讲的，剖析工具的特性集和精确度各有不同，这是由剖析工具开发人员的实现决策决定的。下面将简要介绍gprof、Callgrind和Quantify，以便读者能够从一个小示例入手。此外，我们将给出一些文献参考，以提供更多细节信息。重点讨论如何系统性地使用剖析工具查找性能bug，并专门介绍一下剖析工具在哪些地方的测量结果准确，在哪些地方的测量结果不准确（后者更加有趣）。

6.2.1 不要编写自己的剖析工具

程序员常犯的一个错误是忽略了一些高级剖析工具（将在后面的内容中介绍），而手工创建自己的剖析工具，在应用程序中插入时间测量代码。我们建议“不要这样做”，除非有商业动机来重复实现典型剖析工具中提供的多数技术。

自己编写的剖析工具通常使用系统调用time()或ctime()来测量时间。这些系统调用的问题是开销很高，而且准确度低。经验表明，不要用这些系统调用来测量函数或程序块，而且当事件的运行时间少于0.1秒时，也不要用它们来测量这些事件之间的时间差。

自己编写的剖析工具的另一个问题是在剖析期间要收集大量数据。除非特别下了一番功夫来设计用于存储剖析数据的数据结构，否则在存储期间的存储数据计算工作和内存流量可能比程序本身的计算和内存流量大得多。潜在开销和测量的不准确性使得自己编写的剖析工具没有实际使用意义。

关于运行时剖析工具

- 不要编写自己的运行时剖析工具。
- 有大量的商业剖析工具或公共剖析工具可供使用。当然，理解它们的输出格式可能需要下一番功夫，但这是值得的。

6.2.2 剖析工具的工作原理

剖析工具的作用是给出一个准确的概览，指明程序的运行时间都用在何处，同时提供有关特定函数甚至个别语句的准确且详细的数据。剖析工具有两个核心功能：收集数据和呈现数据。

1. 收集数据

剖析工具使用不同的技术来判断运行时间都用在何处。主要技术是采样和插装。

采样 (sampling) 方法很简单：在固定时间间隔（采样间隔）拦截程序，观察调用栈，并跟踪每个函数在调用栈顶端出现的频率。需要运行时间越长的特定函数，在调用栈顶部出现的频率越高。采样的优点是对程序运行速度影响较小，并且不需要修改程序。主要缺点是固有的不准确性。由于采样是根据挂钟时间进行的，因此程序速度的微小变化（如由于缓存未命中引起的变化）意味每次运行程序时，都是在不同的程序位置上做的采样。因此，报告的结果可能有偏差，特别是那些运行时间很短的函数，由于这个原因，它们只在很少的样本中出现。如果要通过采样来剖析程序的某个特定部分，一定要确保这个部分的总运行时间至少超过剖析工具采样间隔两个数量级。

剖析工具使用的第二种技术是插装 (instrumentation)。插装就是在程序中插入用于跟踪函数调用频率及调用者的语句。插装代码可以在编译期间 (`gprof`、`gcc`标记-`pg`)、链接期间 (`Purify`、`VTune`) 或模拟程序期间 (`Callgrind`) 添加。也可以在函数内部添加插装代码，如解释每个基本块的插装代码。基本块是一个语句序列，其中没有任何条件或跳转语句。基本块的执行时间永远不发生变化，而且可以静态计算出来（在忽略缓存效果的情况下）。程序的总执行时间的计算方法是每个基本块的时钟周期数乘以相应的执行次数。插装的主要优点是可以精确到CPU周期。主要缺点是与采样相比对程序速度影响更大。

2. 呈现数据

剖析工具收集了大量数据，最后需要以适当方式呈现它们，呈现形式既要适合浏览，还要提供有关特定函数的详细反馈。剖析工具几乎都能以一种直观形式成功地呈现数据。

一般情况下，要呈现两类数据，一是平面剖析 (flat profile) 数据，它显示了每个函数本身所用的时间，二是调用图 (call graph) 数据，它显示了函数如何互相调用，以及调用的频率如何。关键是如何将这两类数据合并在一起。例如，平面剖析可能显示75%的运行时间花费在函数 `strcmp()` 上，且调用图显示在对 `strcmp()` 的100 000个调用中有95 000个来自 `func1()`。这说明仔细检查 `func1()` 是有意义的。下一节将以 `gprof` 为例进一步解释平面剖析和调用图。

6.2.3 了解 `gprof`

GNU剖析工具 `gprof` 是一个测量工具，它测量程序的每个函数占用多长时间，以及每个函数被其他函数调用的次数。此外，如果程序是用调试信息编译的 (`gcc`选项-`g`)，还可以看到程序的每个基本块用的时间。使用 `gprof` 需要以下3步。

- (1) 用 `-pg` 标志编译并链接程序。
- (2) 运行程序，剖析文件数据将被写入 `gmon.out` 文件中。
- (3) 运行 `gprof <program> gmon.out` 命令生成剖析报告。

第1步是用 `-pg` 编译器标志编译程序的源文件。这会把特殊的插装代码插入到生成的每个函

数的对象代码中。每当函数被调用时，都会生成一个包含一对数据项（函数和调用者）的记录。

第2步是运行程序，在程序启动时插入一种机制，它在固定时间间隔记录程序计数器目前正在哪个函数中。

第3步对收集到的剖析数据进行排序，并计算统计结果，后面几段会讲到。在gprof文档中可以找到有关gprof程序选项的详细信息。这些选项可用来更改报告的数据格式和类型。

了解gprof的一些实现细节有助于理解工具的优缺点和不准确之处。在gprof文档的Details of Profiling/Implementation节中可以找到有关gprof实现的描述。参见附录B.5.1了解如何在网上查找与gprof有关的信息。

gprof的优点是很容易在新的机器架构上实现，因此可以在大部分平台上使用。使用gprof能限制剖析的范围，方法是只在感兴趣的文件上使用`-pg`编译标志。

gprof的一个缺点是不剖析调用操作系统所花的时间，而且可能无法处理共享库(`lib*.so`)。其另一个缺点是插入的剖析代码极大地降低了程序的运行速度，这也是所有剖析工具的共同缺点。当在CPU为2 GHz的Linux和GCC 3.X机器上运行`isort.c`程序时，程序的运行速度降低为原来的1/4。虽然这种影响已经很大了，但其他工具（如Quantify或Valgrind）则可能导致程序运行得更为缓慢。

此外，采样方法并不完全准确。因为gprof并不测量函数所用的准确时间，而是记录在固定时间间隔内程序计数器位于哪个函数中。这些时间间隔比单个CPU指令的持续时间要长得多，以便减少用在测量上的时间，并减少收集的数据量。此外，gprof必须猜测函数在其调用者上的时间分布。这是一个很大的限制。

1. 警惕gprof的测量错误

一般来说，应该确保剖析数据是在运行一个有代表性且有意义的测试期间生成的。这包括准备一个适当的输入数据集，并为程序提供正确的参数。由于gprof使用基于采样的方法，因此测量错误取决于在特定函数中用的总时间，而不是执行一次函数所用的时间。这是很重要的，因为如果运行次数足够多，则可以准确地分析那些运行时间很短的函数。相反，如果执行次数很少，那些运行时间较长的函数可能产生更大的错误。在实际调试中，应该建立适当的测试用例，使其至少运行5秒钟。

在某些情况下，有必要增加用gprof插装的程序的运行时间，以便软件的每个相关部分的运行时间至少超过剖析工具的采样间隔两个数量级。在`isort.c`示例中，排序算法在相同输入数据上重复数千次。

2. 平面剖析

下面的例子说明了如何使用gprof来剖析`isort.c`程序：

```
> gcc -o isort -pg isort.c
> ./isort i 100 100000
> gprof isort gmon.out >report.txt
```

gprof报告的第一个重要数据是报告文件顶部的平面分析，如图6-2所示。记住，主程序将在一个包含100个样本的数组上运行插入排序函数100000次。还要注意gprof没有计算isort()函数对自己的递归调用。

Each sample counts as 0.00195312 seconds.						
% cumulative	self	self	total			
time	seconds	seconds	calls	ns/call	ns/call	name
66.79	22.47	22.47	495000000	45.40	45.40	less
25.56	31.07	8.60	9900000	868.45	3356.02	insert_value
6.41	33.22	2.16	219300000	9.83	9.83	swap
1.03	33.57	0.35	100000	3476.56	335722.66	isort
0.21	33.64	0.07				main

图6-2 isort.c的gprof平面剖析

总的运行时间33.64秒比采样间隔0.00195312秒高4个数量级，因此完全符合我们的建议（高出两个数量级）。

图6-2中的函数是根据第3列（self seconds）排序的，也就是函数调用本身所用的时间。第4列（calls）给出了函数的调用次数。第6列（total ns/call）给出了此函数及其所有后代中每个调用花费的平均时间，这是此表中唯一使用了调用图数据的字段。其他列的用处较小，并且不含有新信息，只是用不同方式呈现了相同数据。

注意图6-2的第一行，此行给出了采样之间的时间。记住，在第3列中所有与采样时间数量级相同或小于采样时间数量级的测量结果都是不可靠的。

图6-3显示了根据第1~n-1个元素已排序这个事实，修改insert_value()函数之后的平面剖析表。函数在找到第一对元素*i, i-1*之后退出，这对元素不必交换。正如报告所示，运行时间减少，因为在insert_value()的for循环中对less()函数的调用次数减少了。

Each sample counts as 0.00195312 seconds.						
% cumulative	self	self	total			
time	seconds	seconds	calls	ns/call	ns/call	name
62.07	10.27	10.27	229000000	44.86	44.86	less
22.96	14.07	3.80	9900000	383.92	1626.03	insert_value
12.23	16.10	2.02	219300000	9.23	9.23	swap
1.94	16.42	0.32	100000	3203.12	164179.69	isort
0.80	16.55	0.13				main

图6-3 在加速insert_value()之后isort.c的gprof平面剖析

3. 调用图

在gprof所生成的报告中，第二个有用的表是调用图。图6-4显示了修改后的insert_value()函数的调用图。最初看起来，输出格式有些含义模糊，而且需要一些实际经验才能理解。其他一些带有图形前端（如Quantify）的工具可以用更易看懂的方式来呈现这数据。但是，它们都呈现相同的数据，即哪个函数调用了哪个函数、调用的频率以及调用花费了多长时间。

```

granularity:
each sample hit covers 4 byte(s) for 0.01% of 33.64 seconds

index %time self children    called      name
                                         <spontaneous>
[1] 100.0   0.07  33.57          main        [1]
     0.35  33.22  100000/100000  isort       [2]
-----
                           99000000           isort       [2]
                           0.35  33.22  100000/100000  main        [1]
[2]  99.8   0.35  33.22  100000+9900000  isort       [2]
     8.60  24.63  9900000/9900000  insert_value [3]
     99000000           isort       [2]
-----
                           8.60  24.63  9900000/9900000  isort       [2]
[3]  98.8   8.60  24.63  9900000  insert_value [3]
     22.47  0.00  495000000/495000000 less        [4]
     2.16   0.00  219300000/219300000 swap       [5]
-----
                           22.47  0.00  495000000/495000000 insert_value [3]
[4]  66.8   22.47  0.00  495000000 less        [4]
-----
                           2.16   0.00  219300000/219300000 insert_value [3]
[5]   6.4    2.16   0.00  219300000 swap       [5]

```

图6-4 isort.c的gprof调用图

gprof调用图是按块组织，并用虚线分开的。每个块处理一个函数，显示了该函数的调用者以及调用频率。这个块还显示了函数自身用的总时间（self time），以及被调用的子函数和在这些子函数调用中所用的时间（descendant time）。块是按照累积的（self+descendant）时间排序的。首先显示的是那些本身或其后代占用大部分运行时间的函数。main函数通常是最高的。

我们通过第2块（它处理isort函数）来对格式进行解释。以方括号开始、索引数字在最左端的行（[2]...isort [2]）是函数本身。该行上面的一行是一些调用isort的函数，在本例中就是main和isort函数。调用图还包含函数调用频率的信息。可以看到，main函数调用isort 100 000次，isort又对它自己进行了9 900 000次递归调用。[2]...isort [2]下面的一行代码是isort所调用的后代函数。可以看到，isort对自己进行递归调用，而且调用了insert_value。

函数调用计数的格式是一对数字，即<来自该函数的调用次数/总的调用次数>。在本例中，isort调用insert_value的次数为9 900 000/9 900 000，这意味着总共有9 900 000次调用，其中9 900 000次（即所有）调用都来自isort。

图数据包括函数调用次数和所有调用所需的总时间。看一下第4块，它处理less，可以看到此函数需要22.47秒，占总运行时间的66.8%。它的全部调用都来自insert_value，而且（平均）每次调用insert_value都导致insert_value调用less约50次。

记住，gprof猜测函数总时间在其调用者中的分布。调用的分布是准确的，但运行时间的分布只是根据“所有调用都需要相同运行时间”这个假设来猜测的。如果这个假设错误，则

`totalns/call`中呈现的数据将是错误的，而且更糟的是，调用图在运行时间分布方面可能会给出误导信息。

经验总结

- 每个函数的平面剖析显示了它被调用的频率，以及占用CPU的总时间。
- 调用图解释了函数之间是如何互相调用的，以及调用的频率。
- 关键是将平面剖析与调用图结合起来分析：在平面剖析中查明所有时间都用在何处，并通过分析调用图来理解为什么会这样。

4. 如果有些库不是用`-pg`编译的该怎么办

如果可能的话，用编译器标志`-pg`来编译所有源文件。然而，这并不总是可行的，例如当第三方库只能作为对象文件的时候。此时，`gprof`在某种程度上可以处理这种情况。好消息是平面剖析仍然可用，但有如下两个注意事项。

- 未用`-pg`标志编译的所有函数的`calls`、`self ns/call`和`total ns/call`字段均保持为空。调用的总数是未知的。`self seconds`字段包含有用信息。
- 未用`-pg`编译的函数并不会减少`gprof`的运行时间开销。由于`gprof`在报告结果时无法去除这些开销，因此未用`-pg`编译的函数看起来比用`-pg`编译的函数运行得更快，即使它们的运行时间实际上是相同的。

问题在于调用图变得更不完整，而且每个未用`-pg`编译的函数的可信度更低。调用图可能不能完整地记录“哪个函数调用了哪个函数”。未用`-pg`编译的函数的调用者被报告为`<spontaneous>`。未用`-pg`编译的子函数不出现在子函数列表中，即使子函数存在，也不会给出提示。这样，很难（即使有可能）得出运行时间用在何处的准确结论。

经验总结

- 当使用`gprof`时，要确保用`-pg`编译器标志来编译所有源文件。
- 如果未用编译器标志`-pg`编译的函数占用了大部分运行时间，则要检查平面剖析。如果情况如此，在解释调用图时要多加注意，因为它可能给出误导信息。

6.2.4 了解 Quantify

Quantify是一个功能强大的商业剖析工具，IBM将它作为IBM Rational软件质量工具家族的一部分出售。附录B.5.2给出了相关文档和如何获得它的更多信息。

Quantify与`gprof`相比的最大优点是有一个易于使用的图形用户界面，它提供了多种方式来查看测得的剖析数据，并且提供了一种更准确的时间测量和记录方法。Quantify的工作原理是将测量指令插入到对象代码中，以计算指令和每个指令用的周期数。与`gprof`在固定时间间隔对程

序计数器进行采样的方法相比，这给出了更精确且可重复的数据。Quantify的另一个优点是它记录了每个函数调用的时间和调用者。实际上，这可能是与gprof最大的区别，即能够准确地显示在调用者和被调用函数之间的时间分布。

Quantify的缺点是插入的代码导致实际执行时间显著增加。在下面的示例中，运行时间增加了50倍。由于不涉及随机采样，所以Quantify并不需要很长的测量周期。减少总的程序运行时间也能够得到同样准确的结果，这样可以弥补Quantify的一些运行时间开销。

使用Quantify的步骤如下。

(1) 将quantify添加到程序的链接命令中。

(2) 运行程序。弹出一个图形用户界面，程序运行结束时，此界面中显示收集到的数据。

这里再次使用了isort.c示例：

```
> quantify cc -o isort isort.c
> ./isort i 100 10000
```

Quantify将剖析代码插入到程序的对象文件和库（包括所有OS和C++运行库）中。Quantify不必使用源代码文件。因此，当无法访问某些库的源代码时，可以方便地使用Quantify对其进行性能分析。建议在使用Quantify GUI时采用以下设置。

- 忽略**Call Graph**显示。
- 在**Function List**窗口中，将**Display data**选项设置为**Function+descendants**，并将**View/Scale Factors**设置为**seconds**。

Function List窗口将显示在每个函数中用了多长时间。以下是用Quantify分析示例程序得到的**Function List**数据：

```
0.59  main
0.59  isort
0.58  insert_value
0.20  less
0.13  swap
```

单击函数列表中的**insert_value()**项，将弹出**Function Detail**窗口。**Function Detail**窗口中的内容如图6-5所示。图6-5显示了**insert_value()**被调用的次数，以及在函数中用了多长时间。而且也显示了后代函数**swap()**和**less()**对**insert_value()**的调用次数，以及**insert_value()**的调用者和调用频率。在本示例中，**insert_value()**只有一个调用者，即**isort()**函数。

Quantify将剖析数据保存到扩展名为.qv的文件中。过后显示剖析数据时不必返回程序，只需运行以下命令：

```
> quantify -view <file>.qv
```

6.2.5 了解 Callgrind

Callgrind（参见附录B.4.2）是Valgrind调试和剖析工具套件的一部分，它可以收集准确的运

行时间数据和调用图信息。使用Callgrind的步骤如下。

```
Function name:           insert_value
Called:                 990000 times
Function time:          0.25 secs (41.84% of .root.)
Function+descendants time: 0.58 secs (98.60% of .root.)

Distribution to callers:
990000 times  isort
Contributions from descendants:
49500000 times (35.09%) less
22870000 times (22.48%) swap
```

图6-5 isort.c的Function Detail窗口

- (1) 用Callgrind运行程序，剖析数据将被写入callgrind.out.<id>文件中。
- (2) 运行callgrind annotate callgrind.out.<id>命令生成一个剖析报告，或用KCachegrind来查看结果（参见附录B.4.3）。

注意，不必用特殊的标志或工具来编译程序。Callgrind使用现有的可执行程序，这是一种非常方便的使用模式。

这里再次使用isort.c程序作为示例。注意，我们对callgrind_annotate的输出稍微做了修改，删除了不相关的所有行和字符串。结果是以时钟周期数报告的：

```
> gcc -o isort -O isort.c
> valgrind --tool=callgrind ./isort i 100 10000
> callgrind_annotate callgrind.out.31612
 919,788,380  PROGRAM TOTALS
 545,100,000  insert_value
 198,000,000  less
 153,510,000  swap
 16,770,000   isort'2
```

测量结果具有可再现性，因为它们是基于Valgrind引擎计算的，此引擎模拟处理器；它不涉及在随机时间采样。就像Quantify一样，这能够在缩短测量周期的情况下测得同样准确的数据，因此弥补了运行时间开销。在本例中，Callgrind使程序速度降低为原来的1/40，这是一个严重的缺点，这个数字远远高于gprof对程序速度的影响（速度降低为原来的1/4）。

--tree=both选项将显示调用图，列出每个函数的调用者以及它调用的其他函数。
--auto=yes选项输出带注释的源文件（这些文件是用调试信息编译的）。

Callgrind有一些有用的API，可用来获取基本数据，并且仅对感兴趣的特殊区域进行剖析。这减轻了对那些无关区域的速度影响。虽然可以以文本形式读取调用图，但通常更容易的方法是通过图形前端KCachegrind来查看：

```
> kcachegrind callgrind.out.31612
```

图6-6显示了KCachegrind的快照。通过配置右侧的两个窗口，可以显示调用者列表、被调用者列表、调用图或CPU时间在各函数间的图形分布。

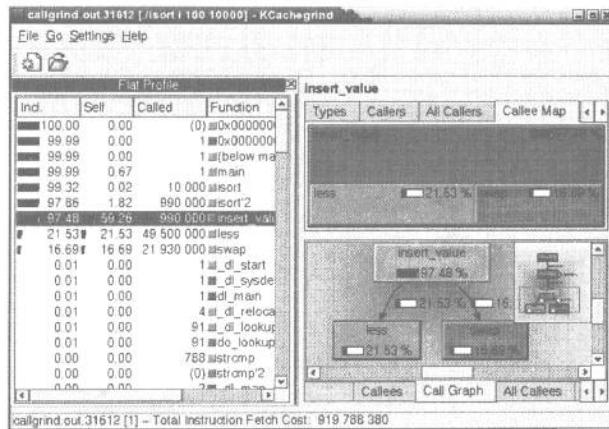


图6-6 用KCachegrind前端显示的isort的Callgrind剖析数据

6.2.6 了解VTune

Intel VTune(参见附录B.5.3)是一个适用于x86和x64 CPU的商业剖析工具,它可以在Windows和Linux平台上运行。VTune有两种模式:采样和调用图,前者类似于gprof,后者类似于Quantify。在Windows平台上使用VTune的步骤如下。

- (1) 启动VTune。
- (2) 在Easy Start向导中,选择New Project,然后选择Sampling Wizard或Call Graph Wizard。
- (3) 选择Windows...profiling。
- (4) 在Application to Launch中输入应用程序名称,并在Command Line Arguments字段中输入命令行参数。

(5) 保持所有其他剖析选项为默认值。有很多测量选项可用来检查文档。

(6) 在向导完成之后,VTune将自动运行程序。

首先选择Sampling Wizard。这里再次以isort.c程序为例,用命令行选项i 100 100000来运行程序。如前所述,在调试模式中构建isort,并用VTune运行它。程序运行结束时,VTune将显示一个窗口,显示在运行时所有活动的进程。选择isort.exe进程,然后单击Drill Down to new Window按钮。这个按钮很难找到,而且遗憾的是没有菜单项。继续选择isort.exe进程,并单击Drill down...按钮,直到显示isort.exe中的函数。采样模式中的VTune给出了CPU采样的统计信息和每个函数用时的百分比。

Name	CPU%
less	52.29%
swap	31.57%
insert_value	11.99%
isort	1.09%
main	0.35%

注意，采样模式中的VTune并不记录函数调用数，也不记录某个函数在调用其他函数时所用的时间。为了测量这些数据，需要使用调用图模式。

重复这个示例，这次使用**Call Graph Wizard**。注意，为了使用调用图，在构建isort.exe时需要启用调试功能，并且链接程序需要有一个选项来生成可重定位的代码。命令行选项如下（也可以将它们输入到Visual Studio项目配置中）：

```
> export LINK=/fixed:no
> cl -o isort isort.c /Zi
```

图6-7显示了程序执行完成之后的VTune。右上方窗口显示了每个函数的运行时间统计。右下方窗口显示了调用图。可以调整这两个窗口，使其仅显示main中的函数及其后代函数。调用图模式中的VTune给出了统计信息，包括调用数、函数本身所用时间以及函数所用的总时间。

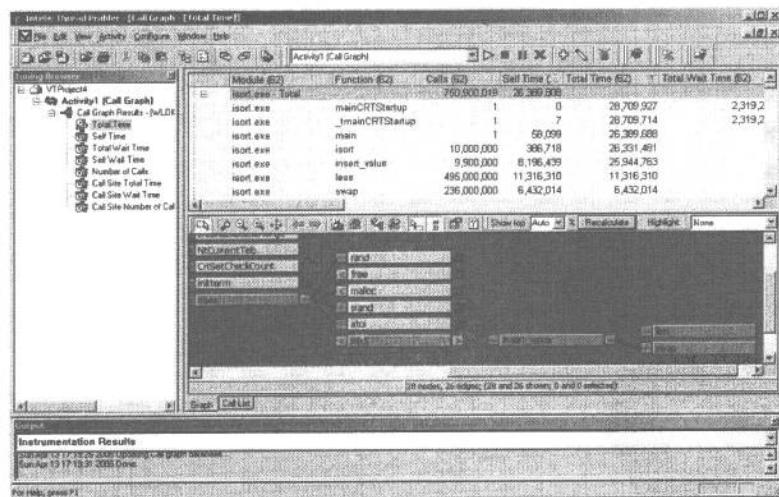


图6-7 在调用图模式下，isort.c的VTune剖析数据

经验总结

- 剖析工具有多种不同的数据收集方法。不同方法在设置、精确度、可再现性和运行时间开销方面各有不同。
- 采样：在固定时间间隔拦截程序，并分析调用栈。这种方法设置简单，开销低，但精确

度不高，而且可再现性不高（每次运行报告的数字均不同）。gprof和VTune在采样模式中使用这种技术。

- 为对象文件编写插装代码，这种方法的设置较复杂、运行时间开销较高（使速度降低为原来的1/10），但准确度较高，而且可以得到再现的数字。Quantify和VTune在调用图模式中使用这种技术。
- 在CPU模拟时间进行插装，这种方法设置简单，运行时间长（使速度降低为原来的1/10），但准确度高，而且可以得到再现数字。Callgrind使用这种技术。

6.3 分析I/O性能

本节介绍如何分析由错误或低效的外部内存系统I/O（读/写）操作导致的性能问题。这些内存系统可能是硬盘、USB记忆棒、闪存、CD ROM驱动器和网络文件服务器。这些设备的共同特性是为大量数据提供了永久存储，但它们的速度比计算机的寄存器、缓存和主内存慢几个数量级。

这里将用一个简单示例来一步一步演示分析过程。参见附录C.4中的filebug.c程序。当用以下命令调用它时：

```
./filebug s xxx.log 2000000
```

程序将使用缓冲的I/O函数fopen()、fclose()和fputc()把2 000 000个字符写入xxx.log文件中。由于使用了参数s，程序在写入每个字符之后调用fflush()函数，这样内置缓冲机制就失去作用，从而导致每个字符的写入都产生单独的系统调用和磁盘I/O操作。

这个程序显然非常简单，通过代码评审很容易发现可以不必调用fflush()。但是要记住，在实践中这样的问题往往隐藏在大型软件项目的代码中，因此需要用一种系统性的方法来定位和排除问题。

第一步是用time命令进行测量：

```
> time ./filebug s xxx.log 200000000
73.724u 444.128s 8:40.78 99.4%
```

从上面的测量可以看出：用户进程只占总运行时间（ $73.7+444.1=517.8$ s）的14%（73.7 s），其他都是系统调用时间。这与isort.c示例的测量结果有明显区别，isort.c只执行计算，而不发生I/O操作。

检查测量的合理性

根据上面的测量，从粗略的合理性检查计算可以看出，测试程序在520 s内写入了2亿个字符，即写入速率为0.38 MB/s。在撰写本书时，计算机的磁盘I/O速度是这个数字的10~100倍，因此这个数据写入速率是有问题的。进一步的调查显示xxx.log文件的位置在远程文件服务器上，计算机通过网络连接到文件服务器。

为了查明是否是网络带宽或文件服务器的低性能造成了影响，我们在另一台使用本地硬盘的计算机上重新进行了实验。独立的磁盘基准测试工具在这台计算机上测得的最大传输速率是40 MB/s。实际的运行时间是451s。这个结果略有提高，但0.46 MB/s的速率仍然低于预期值。

下一步是再次实验，并用剖析工具来观察程序内部的运行情况。如前所述，gprof剖析工具并不为系统调用编写插装代码，因此它不是分析I/O问题的好工具。因此，接下来使用Quantify。由于我们知道Quantify会减慢程序运行速度，因此将输出文件的大小减少为原来的1/100，即200万，以使测试用例更可控且更快得到结果。**Function List**的数据视图如图6-8所示。可以看到fflush()占用了程序的大部分执行时间。

```
Function+descendants time (secs)
4.77  main
4.70  fflush
0.05  fputc
0.00  fopen
0.00  fclose
```

图6-8 I/O性能示例的Quantify Function列表

如图6-9所示，**Function Detail**显示了有关函数调用数的更多数据。这证实了程序包含对fflush()的调用代码，且调用频率与fputc()的调用频率相同，因此程序对每个字符都进行一次缓冲区刷新，这些操作产生了很高的开销。

修改代码，使其不再调用fflush()。在示例程序中，这项修改很简单，只需将第一个命令行参数由s更改为f。然后再次运行实验（这一步很重要），以验证代码修改确实提高了程序速度。对于filebug.c示例，使用网络磁盘重新运行第一个实验，结果如下：

```
> time filebug f xxxx.log 200000000
9.179u 1.363s 0:10.89 96.6%
```

```
Function name:                      main
Distribution to callers:
1 time  __libc_start_main
Contributions from descendants:
2000000 times (98.64%)  fflush
2000000 times ( 1.07%)  fputc
1 time   ( 0.06%)  fopen
1 time   ( 0.01%)  fclose
```

图6-9 I/O性能示例的Quantify Function Detail

结果显示测得的实际运行时间快了47倍，磁盘传输速率达到18.2 MB/s。同样，在使用本地磁盘的计算机上再次运行程序，运行时间为24 s，即快19倍。然后在Quantify剖析工具中再次进行实验，可以看到程序总的运行时间减少了，而且大部分时间用在fputc()中，如图6-10所示。

```
Function+descendants time (secs)
0.08  main
0.07  fputc
0.00  fopen
0.00  fclose
```

图6-10 删除fflush()之后，I/O性能示例的Quantify Function List

对于这里的小示例，不再需要更多修改，但在大型应用程序中，通常有多个性能瓶颈，需要一个一个地解决。

经验总结

- 剖析I/O性能与剖析运行时间很类似，即选择一个好的、可扩展的测试用例，在使用剖析工具之前，进行前期分析/合理性检查。
- 确保所选择的剖析工具能够很好地分析操作系统调用所花的时间。

第7章

调试并行程序

本章介绍C或C++并行程序的调试。首先讲解两类最主要的并行编程错误：竞争条件和死锁。然后讨论线程分析工具，本章章末讲述异步事件和中断处理程序。

7.1 用 C/C++编写并行程序

软件中的并发有很多种形式。低层的形式有位级（bit-level）和指令级（instruction-level）的并发机制。这样就有了一些为专用系统（如矢量处理器或阵列处理器）编写的程序。但是，本章将主要讲述更粗粒度的并行机制，并特别考虑三类程序，一是用C或C++编写的可在不同主机上运行的多路通信程序，二是多线程程序〔包括运行在嵌入（实时）操作系统上的应用程序〕，三是如信号或中断处理程序这样的代码（它们在本质上就是异步的）。

有很多库和语言扩展可用于编写并行程序。更多信息、参考和文档链接可参见附录B.7。Posix Threads和Windows threads库提供了对现代操作系统线程功能的低层访问。TBB（Threading Building Blocks）是一个C++模板库，可用于编写独立于OS的多线程代码。OpenMP是一个基于预处理器指令的C/C++和Fortran语言扩展，它通常是在线程上实现的。线程库和OpenMP基于共享内存的概念，即每个并行线程都可以访问所有全局变量和堆上的所有动态内存。每个线程都有栈，栈中有局部变量。

除了共享内存以外，还有协作进程，MPI（Message Passing Interface）标准就基于协作进程，每个进程都有其自己的内存，运行在一个机器集群上。Google的MapReduce是一个用于并行计算的框架，它将计算分布在相隔很远的集群上（这样的集群具有不可靠的节点）。

这些并行应用程序的调试更为复杂。**bug**可能导致无法预料的行为，也会影响一系列的程序。此外，我们还面临新的挑战。两类最重要的**bug**是竞争条件和死锁。

在一些系统中，计算结果严重依赖于正确的计算步骤，从而经常发生竞争条件**bug**。破坏这种排序可能导致错误的数据或行为，或者只是带来麻烦，例如每次运行程序时日志文件的内容不断发生变化。竞争条件是由于在共享变量和程序代码上使用了错误的同步或未使用同步而导致的。有一些机制可以防止竞争条件，例如互斥（mutex）、信号量（semaphore）、临界区域或边界，

以及原子操作。有关这些机制的一般介绍、等价性证明和实现细节，可参考两本权威著作 [Tanenbaum01] 和 [Silberschatz04]。7.2节将讨论如何对竞争条件进行调试。

死锁是一种典型的线程或进程问题。最常见的情况是一系列互相等待的进程。其他情况还包括进程终止过早和过度触发的进程，其中进程终止过早通常是由于在线程顶层运行时意外返回导致的。这些内容将在7.3节介绍。

7.2 调试竞争条件

如果两个或多个线程共享一个变量，而且当一个进程在共享变量上执行原子事务（atomic transaction）时没有同步机制来阻止另一个线程，就有可能发生竞争条件。原子事务是指一个不能被打断的操作序列。

图7-1显示了一个简单的数据竞争条件（data race condition）示例，其中两个并行线程 t_beancounter0、t_beancounter1访问变量beans，此变量是在第19行分配的。访问发生在第11行的累加语句(*beans)++中。

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <pthread.h>
4
5 void* beancounter(void* beans_arg)
6 {
7     int counter;
8     int* beans = (int*)beans_arg;
9
10    for (counter=0; counter<100000; counter++) {
11        (*beans)++;
12    }
13    return beans_arg;
14 }
15
16 int main(void)
17 {
18    pthread_t t_beancounter0, t_beancounter1;
19    int beans = 0;
20
21    pthread_create(&t_beancounter0, 0, beancounter, (void*)(&beans));
22    pthread_create(&t_beancounter1, 0, beancounter, (void*)(&beans));
23    pthread_join(t_beancounter0, 0);
24    pthread_join(t_beancounter1, 0);
25
26    printf("The sum of all beans is %d\n", beans);
27    return 0;
28 }
```

图7-1 从两个不同线程调用beancounter()函数

当运行程序时，结果偶尔会小于期望值200 000。原因在于第11行中的语句不是原子操作：它

被转换为一个机器操作序列，即读取变量、在CPU寄存器中执行加法运算，然后写入变量。C/C++语言并不保证语句（甚至是变量累加）作为原子操作来执行。现在，第11行将发生以下操作。

- t_beancounter1执行读操作。
- t_beancounter2访问共享变量，并执行一次或多次累加。
- t_beancounter1完成累加并写回结果。
- t_beancounter2所执行的累加丢失。

注意，失败的频率取决于环境，例如操作系统、编译器、CPU负载等。我们需要反复运行测试，可以通过脚本运行，或使用UNIX csh命令repeat：

```
repeat <nr_or_repetitions> <path_to_exe_file>
```

数据竞争的典型症状是偶尔发生不可预测的行为。如果在一个含有并行进程的程序中看到这种失败，首先要识别共享的内存变量，然后在访问这些变量的代码中寻找潜在的缺陷。

7.2.1 使用基本调试器功能来查找竞争条件

在并行程序中走查或设置断点将产生干扰，改变程序的时序。如果在调试器的控制下运行应用程序，它可能会产生不同行为。类似地，添加跟踪代码或I/O代码（例如printf()语句）也会改变程序的时序。

当使用源代码调试器对代码进行走查并显示共享变量的内容时，不要希望得到太多信息。连接到调试器后，线程的调试行为可能发生极大变化，以至于观察不到错误。但这仍然是要尝试的第一步。将调试器连接到程序也可能会使错误发生得更频繁，因此可以识别bug。

大部分调试器支持对程序的线程进行分析。这包括对当前线程ID、线程栈和其他有用信息的分析。

我们可以模拟线程的调度（mimic the scheduling），即从一个线程任意切换到另一个线程。如果能够在程序中把对公共变量进行读写操作的几个点找出来，将有助于识别潜在问题。我们应该用调试器来走查代码，并在每一步提出以下问题：

“如果在这里调度程序发生改变，将会发生什么情况？”

如果无法回答这个问题，则可以切换到另一个线程，看一下发生什么情况。

但是，如果逐行走查上面的示例，将会发现可以在任意位置或任意时间切换线程，而不会破坏程序的正确功能。我们正在走查的是C/C++代码，而调度程序可以在机器代码的任意位置切换线程（包括在C/C++语句的中间）。

我们可以令调试器显示和走查机器代码，而不是C/C++代码。在使用GDB时，disassemble命令显示程序计数器当前位置周围的内存的反汇编后的内容，使用stepi i命令可以前进到下一条机器指令。在Visual Studio中，使用Disassembly（反汇编）窗口，参见10.11.4节。

以下是在Intel x86 CPU上执行(*beans)++操作的机器操作序列：

```
0x08048452 <beancounter+30>: mov    0xffffffff8(%ebp),%eax
0x08048455 <beancounter+33>: incl    (%eax)
0x08048457 <beancounter+35>: lea     0xfffffff8(%ebp),%eax
```

如果令调试器在第2步和第3步之间进行线程切换，则可以再现错误结果。当再次切换到被挂起的线程时，它将从被挂起的地方继续执行。它将写回局部寄存器的内容，这将覆盖另一个进程所做的任何修改。

7.2.2 使用日志文件来查找竞争条件

如7.2.1节所述，与顺序程序的调试不同，并行程序的调试通常不会按照我们预期的那样工作，因为连接调试器、暂停程序和走查代码将严重影响应用程序的时序。这是Heisenbug的一种典型情况，即越是努力地调试，就越有可能隐藏bug。

此外，在很多情况下调试器的视图范围过窄，也过于详细了，特别是当不知道要查找什么问题的时候。更有意义的方法是不打断程序的运行，收集所有需要的运行时数据，然后对收集到的数据进行事后处理。这可以通过日志文件（log file）或跟踪文件（trace file）完成，也可以在控制台窗口中显示调试信息。要使用这种方法，必须为日志指令（logging instruction）插装代码，如图7-2所示。但仍然可能会出现Heisenbug，即在添加插装代码后，程序的失败频率减小了，或观察不到错误。

```
1 void* beancounter(void* beans_arg)
2 {
3     int counter;
4     int* beans = (int*)beans_arg;
5
6     for(counter=0; counter<100000; counter++) {
7         printf("%x| before next bean: %d - ",clock(),(*beans));
8         fflush(stdout);
9         (*beans)++;
10        printf("%x| after next bean: %d\n",clock(),(*beans));
11        fflush(stdout);
12    }
13 }
```

图7-2 使用日志指令来收集动态调试信息

`printf()`命令显示了用于观察程序执行流和直接结果的相关数据，这样就能够从日志文件或模拟运行之后的最终输出来定位异常行为。当对代码进行插装时，应遵守以下规则。

- 对并行代码进行插装时，一定要使用线程安全的原子函数和命令，否则最后调试的就是
 - 插装代码，而不是实际的程序。
- `printf()`和`fprintf()`可在大多数操作系统上可靠工作。在每条I/O语句后面（这些语句将流缓冲写到终端或文件中），立即调用`fflush()`。错误流`stderr`是未被缓冲的，因此

最适合用来记录。C++流是最不适合用来记录的，因为它们可能产生来自不同线程的完全交插的文本输出。

- 当转储大型程序中的线程信息时，应使用时间戳。这有助于将来检查和处理收集到的信息。可以使用挂钟计时信息（使用`time()`函数），或使用CPU时间（使用`clock()`函数）。`time()`和`clock()`函数都在标准的C库中，而且头文件`time.h`中声明了它们的函数原型。从`time()`函数测得的挂钟时间具有每秒最粗的粒度，对某些程序来说这可能不够精确，但当对在多个处理器上运行的程序进行调试时，挂钟时间是一个很好的参考。从`clock()`函数测得的CPU时间具有更细的粒度，但可能不适合作为多处理器系统的公共时间排序参考。
- 两台不同机器上的挂钟时间永远不会同步，因此当合并来自两台机器上的日志文件时，无法用它来重构事件的排序。检查并行处理库的文档，看看是否提供了时钟同步机制。
- 如果经常调试并行程序，一种好的方法是创建辅助工具，即跟踪函数（trace function）或跟踪缓冲区（trace buffer）变量，用于自动捕获一些有用信息，如时间戳或线程ID。
- 在某些情况下，使用断言（断言是一种可立即检查程序数据的正确性的代码）是有意义的，特别是如果不使用断言，日志数据量变得很大的时候。但是，要注意调试代码越多，对程序运行时间的影响就越大，因此越不容易找到原来的错误。有时这导致无法追踪问题的根源。另一种选择是使用脚本或实用程序对数据进行事后处理和过滤，特别是当收集到的数据量很大的时候。

在很多情况下，记录的数据可以准确指向问题根源，而有些情况下，它可能只是为我们提供了更好的思路，帮助判断程序在哪个具体位置出现了异常行为。我们应该在这个位置使用代码插装或调试器进行深入挖掘，偶尔还需要重构代码。

图7-2的示例生成的日志显示了以下产生竞争条件的模式：

```

...
1 7530 | before next bean: 791 - 7530 | after next bean: 792
2 7530 | before next bean: 792 - 7530 | after next bean: 793
3 7530 | before next bean: 793 - 7530 | after next bean: 794
4 7530 | after next bean: 790
5 7530 | before next bean: 790 - 7530 | after next bean: 791
6 7530 | before next bean: 791 - 7530 | after next bean: 792
7 7530 | before next bean: 792 - 7530 | after next bean: 793
...

```

从日志可以看出，第3行和第4行连续执行了图7-2中的第10行代码，这意味着程序切换了线程。从日志文件的第4行可以看出，`beans`变量被重置为790。这足以使我们怀疑程序第9行的`(*beans)++`操作的线程安全。下一步就是使用调试器来分析机器代码，并按7.2.1节所讲的，通

过模拟线程调度来理解问题的性质，以便修复它。

一些调试器提供了内置支持——可以在程序执行期间收集数据。我们可以使用GDB的trace和collect命令来设置跟踪点（tracepoint），然后就可以指定每次到达跟踪点时应该记录的数据。可以记录到达跟踪点的次数，并收集其他有用信息来进行事后处理。

也可以用断点来实现类似的功能，在断点处设置转储所需数据的命令，然后继续执行程序。但要注意，这些调试方法将产生干扰，并影响被调试程序的时序。在遇到竞争条件时，这可能导致不同行为。在实践中，绝大多数经验丰富的开发人员同时使用调试器跟踪点和printf()。

7.3 调试死锁

程序中的死锁是指某个线程或进程所需的资源未被释放，并因此导致该线程或进程永远等待资源。在使用了互斥和信号量机制时，如果未正确解锁，就会发生死锁条件。在大多数情况下，死锁的根本原因为如下几点。

- 循环互斥锁定，其中两个或多个线程开始时都占有了资源，但它们无法继续执行，因为互相等待当前被对方锁定的资源。循环互斥锁的最著名示例当属“哲学家就餐问题”。

在哲学家就餐问题中有5位哲学家，他们围坐在一张圆桌餐旁准备就餐。桌中央有一盘意大利面，每两人之间放一把叉子，这样每位哲学家左右各有一把叉子。Edsger Dijkstra假设用一把叉子是不能吃意大利面的，为了吃意大利面，每位哲学家必须拿到两把叉子，并且每个人只能直接从自己的左边或右边去取叉子。如果这5位哲学家同时取走了放在他们右侧的叉子，并等待左手边的叉子，那么他们将都被饿死。

- 协议不匹配，其中两个或多个进程或线程通过互斥或信息量机制保持同步，而互斥或信号量未按要求解锁。在“生产者-消费者”情况中，经常遇到这种情况。如果有一个固定大小的通信缓冲区，则必须保证当缓冲区已满时，生产者停止写入，而且当缓冲区为空时，消费者停止读取。例如，这可以通过信号量机制来实现。如果某个协议错误或竞争条件导致信号量的不明增加或减少，则它可能导致线程当中的一个永远休眠。

当程序陷入死锁时，分析工作的第一步是使用调试器来查看程序和各个线程都处于何种状态，如7.3.2节所述。如果使用交互式调试器得到的信息量不足，例如，由于将调试器连接到程序而使错误无法再现，则可以使用7.2.2节所讲的日志方法。

下面举例说明如何使用调试器来查找死锁的原因，这里使用与图7-1基本相同的代码，唯一不同的是使用互斥机制来消除竞争条件。修改后的beancounter()函数如图7-3所示。这里的错误是在第12行对互斥进行锁定后，没有在第16行中对其解锁，这是一个调试死锁的典型例子。

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <pthread.h>
4
5 pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
6
7 void* beancounter(void* beans_arg)
8 {
9     int counter;
10    int* beans = (int*)beans_arg;
11
12    pthread_mutex_lock(&mutex);
13    for (counter=0; counter<100000; counter++) {
14        (*beans)++;
15    }
16    /* oops, forgot to unlock the mutex here... */
17    return beans_arg;
18 }
```

图7-3 定义和使用全局互斥

在修改beancounter()函数之后运行程序，它将进入死锁状态，而且永远不会退出。在调试器中启动程序将进入同样的死锁状态。我们不必立即在调试器中启动程序，而是可以将调试器连接到正在运行的程序。

7.3.1 如何确定正在运行的是哪个线程

调试器可以告诉我们CPU最后执行的是哪一个线程，以及该线程的栈是什么状态。这样就很可能立即发现死锁的原因。如果在GDB中暂停程序（如按Ctrl-C），并运行info stack命令，将看到当前线程正在等待对mutex进行解锁：

```

^C
Program received signal SIGINT, Interrupt.
[Switching to Thread -172106832 (LWP 18839)]
0x.. in __l11_mutex_lock_wait() from /lib/tls/libpthread.so.0

(gdb) info stack
#0 0x.. in __l11_mutex_lock_wait () from ...
#1 0x.. in _L_mutex_lock_28 () from ...
#2 0x.. in __JCR_LIST__ () from ...
#3 0x.. in ?? ()
#4 0x.. in ?? ()
#5 0x.. in beancounter at beancounter_deadlock.c:12

(gdb) frame 5
#5 0x.. in beancounter at beancounter_deadlock.c:12

(gdb) list 12,15
12    pthread_mutex_lock(&mutex);
13    for (counter=0; counter<1000; counter++) {
14        (*beans)++;
15    }
```

上述调试器信息显示出当前线程正在等待一个互斥命令，而包含此命令的用户代码行是beancounter_deadlock.c文件的第12行。

7.3.2 分析程序的线程

调试器可以告诉我们每个线程的状态，以及当程序挂起时，每个线程正在执行哪段代码。这对于了解程序的总体状态很有帮助，而且往往揭示了在程序流中哪个线程正在其他线程后面运行，这个进程就是进一步调查的最佳候选。

在GDB中可以使用info threads命令。它列出所有线程和它们的唯一ID，并用*标记当前活动的线程。在我们的示例中，当前线程的ID为3，它位于beancounter_deadlock.c文件的第12行（这一点也可以从上一节的栈跟踪看出来），并且第二个线程位于pthread_join函数中。显然，这个线程正在等待另一个线程结束。也可以使用GDB命令thread从一个线程切换到另一个线程，方法是在此命令后面输入要切换到的线程ID。这样就能够分析所有其他线程的栈。

```
(gdb) info threads
* 3 Thread in __lll_mutex_lock_wait () from ...
  1 Thread in pthread_join () from ...

(gdb) thread 1
[Switching to thread 1] #0 in pthread_join () from ...
(gdb) info stack
#0  0x.. in pthread_join () from /lib/tls/libpthread.so.0
#1  0x.. in main () at beancounter_deadlock.c:45
```

根据上面显示的信息，很容易知道Thread 3正在等待mutex解锁，而Thread 1在第12行对mutex进行了锁定。下一个逻辑步骤就是分析为什么Thread 1一直未解锁mutex。解决了这个问题，也就找到了问题的答案，即图7-3所示的函数beancounter()完全忘记了解锁mutex的语句。在第15行后插入一条pthread_mutex_unlock(&mutex)语句即可修复此问题。

7.4 了解线程分析工具

市场上有很多分析并行程序和查找潜在错误源的高级工具。线程分析工具可以帮助发现隐藏的错误，如死锁和数据竞争，并指明出现这些问题的内存位置和代码行。

Intel公司提供了一些有用的商业线程分析工具，其中最具代表性的是Thread Checker工具（参见附录B.7.6），此外开源工具Helgrind也是一个很好的工具（参见附录B.4.2）。

Helgrind是Valgrind插装框架的一部分，在撰写本书时，它仍处于实验阶段，尽管如此，它仍是一个非常有用工具。在Helgrind中运行图7-1所示的beancounter.c程序将会给出一个有用的揭示，即有一个潜在的数据竞争：

```
> gcc -g beancounter.c -lpthread -o beancounter
> valgrind --tool=helgrind beancounter
```

```

== Helgrind, a thread error detector.
== Using valgrind-3.3.0
==
== Thread #2 was created
==     at 0x48CDD0C: clone (in /lib/tls/libc-2.3.2.so)
==     by 0x80484BB: main (beancounter.c:40)
 ==
== Thread #3 was created
==     at 0x48CDD0C: clone (in /lib/tls/libc-2.3.2.so)
==     by 0x80484DD: main (beancounter.c:41)
 ==
== Possible data race during write of size 4 at 0xFEFF6D4C
==     at 0x8048475: beancounter (beancounter.c:11)
==     by 0x48CDD19: clone (in /lib/tls/libc-2.3.2.so)
== Old state: shared-readonly by threads #2, #3
== New state: shared-modified by threads #2, #3
== Reason:    this thread, #3, holds no consistent locks
== Location 0xFEFF6D4C has never been protected by any lock
The sum of all beans is 200000
 ==
== ERROR SUMMARY: 1 errors from 1 contexts

```

在本例中，Helgrind正确地指出：beancounter.c程序第11行的累加运算对同一个内存位置进行共享访问，而没有任何锁为此访问提供保护。

使用图7-3所示的死锁示例beancounter_deadlock.c，也可以从Helgrind得到一些有价值的信息：

```

> valgrind --tool=helgrind beancounter_deadlock

== Helgrind, a thread error detector.
== Using valgrind-3.3.0
==
== Thread #2 was created
==     at 0x48CDD0C: clone (in /lib/tls/libc-2.3.2.so)
==     by 0x8048507: main (beancounter_deadlock.c:25)
 ==
== Thread #2: Exiting thread still holds 1 lock
==     at 0x47E5E24: start_thread (/lib/tls/libpthread-0.60.so)
==     by 0x48CDD19: clone (in /lib/tls/libc-2.3.2.so)
 ==
...

```

一个重要的提示是Thread #2已退出，但它仍保持了一个锁。这种情况通常都不是有意而为之的，很可能成为死锁的原因。

Helgrind的基本机制是记录所有内存访问。Helgrind能够识别被多个线程访问的内存位置。在每个内存位置上，Helgrind记录程序中的哪个锁被正在访问的线程占用。

一个小的难题是即使使用了互斥锁，也有可能会发生数据竞争。使用互斥锁本身就意味着从多个线程访问同一内存位置，否则就没必要使用互斥锁了。由于对此种锁定机制的访问一般无法用另一个锁来保护，因此Helgrind将这种模式报告为潜在的数据竞争（至少有一个最外层的锁定

机制无法用它周围的锁来保护)。这意味着即使在一个使用了锁定而几乎没有bug的程序中，也会得到大量的错误报告。还好，在Helgrind报告中很容易定位那些除潜在数据竞争之外的其他错误，因为它们都指向用于设置锁的方法(`pthread_mutex_lock`)，并指向这个锁定函数在程序代码中被调用的位置。在下面的示例中，这个位置就是`beancounter_fixed.c`文件的第12行：

```
> valgrind --tool=helgrind beancounter_fixed

== Helgrind, a thread error detector.
== Using valgrind-3.3.0
==
== Thread #2 was created
==     at 0x48CDD0C: clone (in /lib/tls/libc-2.3.2.so)
==     by 0x8048553: main (beancounter_fixed.c:25)
 ==
== Thread #3 was created
==     at 0x48CDD0C: clone (in /lib/tls/libc-2.3.2.so)
==     by 0x8048575: main (beancounter_fixed.c:26)
 ==
== Possible data race during write of size 4 at 0x80497F0
==     at 0x47E78D1: pthread_mutex_lock
==     by 0x80484EB: beancounter (beancounter_fixed.c:12)
==     by 0x48CDD19: clone (in /lib/tls/libc-2.3.2.so)
==     Old state: shared-readonly by threads #2, #3
==     New state: shared-modified by threads #2, #3
==     Reason:    this thread, #3, holds no consistent locks
==     Location 0x80497F0 has never been protected by any lock
 ==
== Possible data race during write of size 4 at 0x80497E8
==     at 0x47E78D4: pthread_mutex_lock
==     by 0x80484EB: beancounter (beancounter_fixed.c:12)
==     by 0x48CDD19: clone (in /lib/tls/libc-2.3.2.so)
==     Old state: owned exclusively by thread #2
==     New state: shared-modified by threads #2, #3
==     Reason:    this thread, #3, holds no locks at all
 ==
...
...
```

通常，在线程创建和退出期间、锁的获取和释放期间，或线程通信期间，Helgrind报告的很多潜在数据竞争不必引起注意。

7.5 异步事件和中断处理程序

什么样的程序才算是真正的并行程序呢？这取决于具体的标准。人们可能会说，在简单的单核处理器系统中，多线程程序实际上不是并行的。毕竟，在任意给定时刻只有一个活动的线程。真正的问题在于任务切换是异步的，除非采取特殊保护措施，否则无法保证源代码逐行执行而不发生中断。

这就引入了其他类型的异步事件以及它们所触发的上下文切换。两个典型例子是信号处理函

数和低层中断服务例程。信号处理函数是由操作系统调用的函数，操作系统在将一个信号传递给进程时调用此函数（参见10.7节）。信号是异步的，例如，它们可能是由于用户或其他程序试图与你的程序（SIGINT、SIGIO、SIGTERM……）通信而生成的。因此，相应的信号处理函数可能在任意时刻中断程序，除非（临时）阻止相应的信号。

现在，事情变得有些复杂了。一方面，我们正在处理异步执行，这要求正确保护对共享资源的访问。另一方面，中断处理程序和信号处理函数的运行时间应该很短暂，以防止更多事件的传递被屏蔽。这意味着不能使用任何缓慢的操作，如使用OS调用来访问信号量，或者I/O操作。这使我们陷入进退两难的境地，一方面必须处理异步访问，一方面又不能依靠标准的同步方案，而且传统的printf()调试也是有问题的。

本书不介绍如何编写好的信号处理程序。请参考操作系统开发方面好的图书。在涉及调试时，我们将给出揭示。

首先，我们可以跳转到10.7节看一下，其中解释了从源代码调试器可以得到哪些类型的支持。当然，我们总是可以在处理中断或OS信号的例程中设置断点。在这种环境中，断点命令可能是一个功能强大的工具。其次，可以考虑使用观察点（参见10.6节来确定程序的其他部分是否以及何时访问相同的变量）。变量集应该非常小，否则信号处理函数的准确性就会受到影响。硬件辅助的观察点能够有效地减少出现Heisenbug的可能性。

经验总结

- 使用跟踪文件来调试竞争条件。
- 确保跟踪机制是线程安全的。
- 在使用时间戳对事件进行排序之前，检查并行处理库是否有时钟同步机制。
- 使用简单的示例来了解如何切换线程，并在调试器中检查同步代码。
- 了解线程分析工具：Helgrind、Intel Thread Checker。

查找环境和编译器问题



程序的行为不仅依赖于代码、输入数据文件和用户输入，而且还依赖于运行的环境。例如，程序在受控的开发环境中运行良好，但在一些用户系统上可能会发生故障。诸如OS更新、新软件安装或开发系统服务包更新等维护工作将导致看起来毫无关联的程序错误。本章讨论环境变更的影响，并列举一些可以显示程序进程正在执行哪些操作的工具。此外还将讨论编译器bug以及调试器与编译器之间不兼容的问题。

8.1 环境变更——问题的根源

程序对环境的依赖以很多不可预知的方式表现出来。以下几节将列出典型的环境依赖，包括环境变量和安装依赖。此外，还将介绍如何诊断这样的依赖，以及如何在程序中预测或避免它们。

8.1.1 环境变量

程序在不同系统上、对不同用户有不同行为的常见原因是用户定义的环境变量(*environment variable*)。在UNIX操作系统中，PATH和LD_LIBRARY_PATH是对程序行为影响最大的两个环境变量，而在Windows操作系统上则是PATH、LIB和INCLUDE。PATH用于定位其他程序。在UNIX上，LD_LIBRARY_PATH变量用于定位动态载入程序共享库。在Windows上，操作系统使用PATH变量来查找程序和DLL(*dynamic link library*，动态链接库)。选错了DLL版本可能会导致严重后果。

如果程序在一种环境中正确运行，而在另一种环境中改变了行为，那么该程序可能是根据一个或多个环境变量而改变了内部行为。多次运行程序，确定这种差别是一致且可再现的。输出所有变量，并在两种环境中比较它们的设置。在UNIX上，使用env命令。在Windows上，点击start/Run(开始/运行)，并输入cmd命令打开一个DOS窗口。在DOS窗口中，使用set命令。比较这些变量的两种设置，逐个修改不同的变量，并在每一次修改后检查程序的行为是否发生变化。如果找到了导致程序行为变化的变量，一定要把依赖记录下来，要么修改程序去掉依赖，要么提前做好准备。

8.1.2 本地安装依赖

如果操作系统、补丁、工具、C/C++运行库等在机器上的本地安装方式不同，可能会触发程序的bug。当发生这种情况时，可以试着找到明显的安装区别，并构建一个类似的系统来进行内部测试。要确保能够再现错误。然后，要么在程序中增加代码来支持新的安装，要么添加一个保护机制来阻止程序在新安装中执行，并记录下不兼容问题。一个典型示例是Windows Vista SP1操作系统，它警告用户不支持Visual C++ 7.0编译器。

8.1.3 当前工作目录依赖

程序的行为可能依赖于当前工作目录。在UNIX上，从哪个目录启动的程序，该目录就是工作目录。目录被注册到环境变量CWD中。例如，程序要读写的文件可能都需要在某一特定工作目录中。要使程序从其他目录工作，要么使用绝对路径名称，要么更改程序的当前工作目录。

8.1.4 进程ID依赖

当前进程的ID（PID）可能也会产生影响。PID是一个整数，当程序启动时，它被自动分配给程序的进程。UNIX程序偶尔会创建临时文件，例如/tmp/myprog.<pid>。程序可能不会总是删除这些临时文件，而且错误地假设在打开文件并写入内容之前，不存在同名文件。在运行程序时，如果正好有相同的PID，则程序将无法创建文件，从而失败。

在发生PID问题时，可以使软件发出错误消息，以便修复问题，这样可以建立一种偶发冲突的跟踪机制。此外还应该修改软件，使用更完善的唯一标识符。

8.2 如何查看程序正在做什么

应用程序可能不只是一个程序。它可能由多个协同工作的程序和脚本组成。为了调试应用程序，需要确定哪个程序正在运行，程序参数是什么，以及应该将调试器连接到哪个进程。

8.2.1 用 top 来查看进程

在UNIX和Linux上，可以使用实用程序top（参见附录B.8.5）来查看正在运行的进程清单。进程清单每几秒钟更新一次。该实用程序显示哪些程序占用了大部分CPU时间或内存。在Windows上，可以使用任务管理器来显示进程。

```
Tasks: 78 total, 4 running, 74 sleeping, 0 stopped, 0 zombie
Cpu(s): 60.5%us, 38.1%sy, 0.0%ni, 0.0%id, 0.0%wa, 1.3%hi, 0.0%s
Mem: 276616k total, 271224k used, 5392k free, 48328k buffers
Swap:321260k total, 0k used, 321260k free, 111272k cached

 PID USER PR VIRT RES SHR S %CPU %MEM TIME+ COMMAND
 7606 zaphod 23 26704 22m 2256 R 54.7 8.2 0:22.44 havoc
```

```

7491 arthur 25 2252 680 592 R 14.0 0.2 0:16.40 survive
5597 root 15 34096 13m 2716 S 1.7 5.0 0:09.85 X
5959 mice0 15 31772 15m 12m S 0.3 5.7 56:06.00 find_pol
7604 mice9 19 5120 2352 1588 S 0.3 0.9 0:00.01 test.human
1 root 16 680 248 216 S 0.0 0.1 0:00.62 init
...

```

top的输出如上所示。

8.2.2 用 ps 来查找应用程序的多个进程

UNIX实用程序ps（参见附录B.8.3）显示了计算机上正在运行的进程快照。ps具有命令行参数，使用这些参数可以显示不同类型的信息。注意，在不同的操作系统上，实际的命令行参数有所不同。ps的一个非常有用的特性是检查某一特定用户的进程的父/子关系，并显示进程的命令行参数。在Linux上，这是通过ps -u <username> -H -opid,cmd实现的，其中<username>是用户名。

在下面的示例中，程序myprogA用了很长时间才返回，我们想知道原因。

```

> ps -u someone -H -opid,cmd
...
8804      ./myprogA MyFile 123
8805      /bin/csh -f ./myshellB MyFile 123
8806      ./myprogC MyFile abc 123
8807      sleep 100
...

```

缩进的部分显示出myprogA调用myshellB，后者又调用myprogC，而myprogC执行了sleep命令。

8

8.2.3 使用/proc/<pid>来访问进程

一些Linux和UNIX操作系统允许通过文件系统/proc/...来访问进程。每个进程在文件系统中都有自己的目录。注意，在不同的操作系统上，/proc/...目录下可用的信息类型和访问方式有很大不同。proc功能在Windows上不可用。

在Linux Red Hat和Suse上，可以访问命令行参数（/proc/8806/cmdline）和环境变量（/proc/8806/environ），此外还提供了一个到可执行程序的符号链接（/proc/8806/exe）以及其他信息。

8.2.4 使用 strace 跟踪对操作系统的调用

Linux实用程序strace（在Solaris上是truss）能够记录对操作系统的所有调用，例如内存分配、文件I/O、系统调用和子进程的启动。使用选项-f还可记录与任何子进程有关的信息。在Windows上，安装Cygwin即可使用strace命令。

strace是一个可对那些没有源代码的程序或链入的库进行调试的工具。使用它很简单，不需要对源代码做大的改动，譬如说，无需在每个系统调用周围放置代码包装器和跟踪代码。此外，它确实能够捕获对操作系统的所有调用。相比之下，当在每个操作系统调用周围放置手写的包装器时，不得不担心跟踪代码的完整性和正确性。

stracer的主要缺点是它产生大量难以读懂的输出。**strace**将显示系统调用的原始错误代码，因此我们必须查阅手册，或是上网搜索，才能理解错误。

建议在进行以下调试时使用**strace**。

- 文件I/O。使用**strace**来查明哪些文件被打开了。程序在初始化期间通常使用安装文件，并且以静默方式打开它们。
- 在OS例程中未捕获的错误或中断。**strace**将显示OS调用的返回状态。一个常见的bug是忽略了OS调用的返回状态，因此**strace**查找返回的错误值，并再次核对源代码是否正确处理了这些值。
- OS调用的频率。当调试性能问题时，**strace**可以指明瓶颈是否是由于过多OS调用引起的（这些调用占用了大量CPU时间）。重构代码来删除内部循环中的OS调用可能会起作用。
- 内存分配/释放/映射。**strace**将显示对动态内存管理的调用。当内存调试器（如Purify或Valgrind）不可用时，可以使用**strace**。

在下面的示例中，我们将使用**strace**来获取有关**myprogA**及其子进程的更多信息。注意，这里对输出进行了删减，只显示大部分有关的活动：

```
> strace -f -o strace.log ./myprogA MyFile 123

File trace.log:
8804 execve("./myprogA", ["../myprogA", "MyFile", "123"],
             /* 82 vars */) = 0
8804 uname({sys="Linux", node="linux", ...}) = 0
8804 brk(0)                           = 0x804a000
8804...
8804 open("MyFile", O_RDONLY)         = -1 ENOENT
                                         (No such file or directory)
8804 fstat64(1, {st_mode=S_IFCHR|0600, ...}) = 0
8804...
8804 write(1, "About to call ./myshellB MyFile "...)= 36
8804...
8804 clone(child_stack=0, flags=CLONE_PARENT_SETTID|SIGCHLD,
          parent_tidptr=0xbffffed68) = 8805
8804...
```

`open("MyFile", O_RDONLY)`这一行显示了**myprogA**正在试图打开一个文件，但未找到文件。接下来程序将`About to call ./myshellB MyFile`打印到标准输出，然后克隆自己。进程8805被激活，并生成更多子进程：

```

8805 execve("/bin/sh", ["sh", "-c", "./myshellB
    MyFile 123\n"], /* 82 vars */) = 0
...
8805 read(16, "#!/bin/csh -f\n./myprogC $1 abc $"..., 
4096) = 34
8805...
8805 clone(child_stack=0, flags=...) = 8806
8806 execve("./myprogC", ["../myprogC","MyFile","abc","123"],
    /* 82 vars */) = 0
8806...
8806 write(1, "C called with \'MyFile\', \'abc\'", '\0') = 37
8806...
8806 clone(child_stack=0,flags=...) = 8807
8807 execve("/bin/sh", ["sh", "-c", "sleep 100"],
    /* 82 vars */) = 0
8807...
8807 nanosleep({100, 0}, NULL) = 0

```

8.3 编译器和调试器也有 bug

开发系统是程序环境的一部分。本节将讨论由开发系统导致的bug的两个主要来源，即编译器bug和调试器与编译器之间的不兼容问题，调试这两种bug将耗费大量时间。第9章将讨论开发链中的第三个bug来源，即链接器。

8.3.1 编译器 bug

如果在C++中使用新的或高级语言特性时遇到bug，应该考虑编译器本身有bug的可能性。过去这是很常见的，当时C++正在经历大量的新开发和标准化，但现在偶尔也会发生这种情况。当怀疑存在编译器bug时，可以采用以下方法。

- 利用分而治之的方法来减小测试用例的规模。尝试隔离有问题的编译器特性，最好是能精确到几行代码。
- 尝试各个编译器选项，查看问题是否消失：关闭优化或切换到一个较低的优化水平。打开/关闭调试支持，这也会改变内存布局和变量的初始化。
- 使用先前已发布的语言特性示例来交叉检查，以确定自己在特性的使用上没有错误。
- 上网搜索一下，看看是否有其他开发人员也遇到同样的问题。
- 试一下不同的编译器，看一下问题是否仍然存在。
- 报告编译器bug。

一旦确定了问题是由编译器bug导致的，就必须重新编码。通常修复编译器bug需要一定时间，而项目无法停下来等待bug的修复。这时需要为自己的软件找到一种折中方法，可以不使用特定的语言特性，或者调整编码风格以限制有问题的语言特性的使用。此外还要将问题记录下来，并

保存测试用例。当新的编译器版本发布后，可以快速测试bug是否已修复。这样，就不必永久地保持这种折中方法和编码风格限制。

8.3.2 调试器和编译器兼容性问题

C/C++编译器和调试器有很多种，而且每种编译器和调试器都有很多主要版本和次要版本。并非所有编译器和调试器组合都能够一起工作。测试所有组合是不可能的，而且也不能总是保证向后和向前的兼容性。调试器与OS、系统库（如

- 调试器在载入程序的同时（或者在调试会话的中间）对核心文件进行了转储。
 - 调试器无法在通过函数名指定的C++类成员中设置断点。
 - 调试器在指定的源文件/行位置设置了断点，但在执行到该位置时却不能暂停。
- 在发生编译器与调试器的不兼容问题时，有很多办法解决，如下所述。
- 检查文档：推荐的调试器版本是哪个？是否有已知的不兼容问题？
 - 试一下安装几个不同的调试器版本，并记录发生的不兼容问题。这种方法通常适用于GDB和GCC用户。在Visual Studio中，调试器和编译器是集成在一起的，因此用户只能使用另一个主要版本。
 - 记录哪种组合是有效的。兼容性问题往往是由于开发系统的大规模升级引起的，包括编译器、OS和调试器的升级。我们可能必须回滚软件升级，以确保正确的调试环境。
 - 不要认为随同OS一起发布的默认调试器和编译器就适合你的项目。

经验总结

- 程序bug可能是由环境引起的：环境变量、OS安装依赖、中断处理程序，等等。在将新软件部署到开发系统之外的计算机上时，要考虑这些bug来源。
- 有一些工具可用来调试程序的环境，例如top、ps、env、/proc/<pid>和strace。
- 错误的动态库搜索路径可能是与环境有关的程序崩溃的原因。查找错误或丢失的PATH和LD_LIBRARY_PATH变量值。
- 编译器也有bug。利用分而治之方法来隔离问题，并找到折中办法来继续开发项目。
- 调试器可能与特定的编译器、语言特性、OS和系统库版本不兼容。在对开发系统进行大的变动之前，要记录哪些组合是有效的。



本章介绍如何发现在程序链接阶段引入的bug。链接问题的数量巨大、发生频繁，而且修复这类问题将耗费大量时间，此外调试工具对链接问题的支持也不够完善。本章将给出链接问题的示例，这些问题在链接时无法检测出来，但会导致程序崩溃或操作错误。此外，在实践中，链接器错误消息可能含义模糊，它们只指出了程序组成部分之间不匹配的地方，但通常不会进一步解释这种不匹配的来源。正是由于这些原因，我们拿出一章的篇幅来详细讨论链接问题。

本章首先介绍链接的一些基本原理，然后逐步解释最常见的链接问题，并演示如何检测和修复它们。

9.1 链接器的工作原理

链接（linking）是基于对象文件和对象文件库构建可执行文件的过程。编译器或汇编程序将源代码转换为机器代码，并输出对象文件。对象文件中包含符号，符号表示源代码中定义的函数或变量。如果一个符号被关联到相同对象文件中的一个地址和代码段（其中包含该符号的定义），那么这个符号就被称为已定义或已解析的符号。尚未与地址关联的符号被称为未定义或未解析的符号。链接的基本思想是将每个已解析的符号与其对应的定义关联起来。这个过程也称为符号解析。有关链接器工作原理的更多细节，可参考[Levine00]。

链接器将已解析和未解析的符号保存在两个列表中。链接器的主要工作就是在将对象文件和库汇编成程序时，把未解析符号转换为已解析的符号。在最后的可执行程序中，每个符号都必须有一个有效的定义，否则就会出现问题。

9.2 构建并链接对象

下面几节将使用一段修改后的factorial.c代码作为示例，它与第3章中使用的示例类似，不同的是将main()函数和factorial()函数放到了不同的源文件中。

```

1  /* factorial.c */
2
3  int factorial(int n) {
4      int result = 1;
5      if(n == 0)
6          return result;
7      result = factorial(n-1) * n;
8      return result;
9  }

1  /* main.c */
2  #include <stdlib.h>
3  #include <stdio.h>
4  #include "factorial.h"
5
6  int main(int argc, char **argv) {
7      int n, result;
8      if(argc != 2) {
9          fprintf(stderr, "usage: factorial n, n >=0\n");
10         return 1;
11     }
12     n = atoi(argv[1]);
13     result = factorial(n);
14     printf("factorial %d!=%d\n", n, result);
15     return 0;
16 }
```

将这段源代码文件编译为对象文件和随后的链接工作可以在一个步骤中完成：

```
> gcc -g -o calc_factorial factorial.c main.c
```

尽管如此，我们可以将编译和链接分为两步，而且在大型项目中，以模块化方式来构建可执行文件通常是一个好的思想，也就是说，将每个对象文件及随后的可执行文件或库的构建阶段与对象文件明确地分开。如果不这样，修改某个源代码文件将要求重新编译所有源代码，而使用模块化方法则可以只重新编译那些受到代码修改影响的部分。

像gcc这样的命令实际上并不是一个单纯的编译命令，它管理程序构建过程中的多个阶段，包括预处理、编译、汇编和链接。它通常被称为编译器驱动程序。编译器驱动程序调用链接器ld，并且为链接器提供相关参数（这些参数是从编译器驱动程序的参数派生得到的），此外它还提供了很多特定于平台的对象文件和系统库。

在上面的示例中，将对象文件创建与链接步骤分开就是指首先将源文件编译为对象文件，然后再将这些对象文件链接为一个可执行文件。在下面的命令行中，使用了GCC编译器的-c标志来构建对象文件。然后，将对象文件链接为一个可执行文件，即calc_factorial：

```
> gcc -g -c factorial.c    (builds object file factorial.o)
> gcc -g -c main.c        (builds object file main.o)
> gcc -o calc_factorial factorial.o main.o
```

9.3 解析未定义的符号

最常见的链接问题是由于未解析的符号引起的。下面几小节将介绍一些最常见的情况。

9.3.1 丢失链接器参数

如果链接器的命令行参数中丢失了某个对象文件，而它当中包含所需的符号定义，那么链接器将报告丢失的符号。例如，如果遗漏了参数factorial.o，将得到以下信息：

```
> gcc -o calc_factorial main.o
main.o: In function 'main':
[...]/linking_issues/example1/main.c:13:
      undefined reference to 'factorial'
```

我们看到链接器提供足够信息来说明丢失了什么，如下所述。

- 链接错误是由于一个未解析的符号引起的，它的名称是factorial。
- 该符号未在对象文件main.o中解析。
- 对象文件main.o属于源文件main.c。
- 此源文件的第13行使用了factorial()函数。

在以下示例中，我们编译了不带调试信息的源文件，以证明链接器错误消息仍包含足够信息，可以搜索到哪个对象文件或库中包含丢失的符号的定义。可以看到，链接器错误消息的细致程度减小了：未提供源文件的位置，而且行号被对象文件中的位置取代了。

```
> gcc -c main.c
> gcc -o calc_factorial main.o
main.o: In function 'main':
main.c:(.text+0x4): undefined reference to 'factorial'
```

9.3.2 搜索丢失的符号

根据项目规模的不同，查找丢失的符号定义可能非常困难。在最好的情况下，程序员知道丢失了哪个符号以及它是在哪里定义的，不是在对象文件或库中，就是在某个仍需要编译的源文件中。

如果不知道符号或它的定义位置，那么第一个方法就是搜索它。如果符号是一个函数或方法，则应该试着在文档中查询它。好的搜索目标是计算机系统文档、项目文档或第三方软件的文档。或者用计算机来搜索。在UNIX上，可以使用实用程序grep和find来定位文件。在Windows上，可以使用浏览器的搜索对话框。

有一些实用程序可用来分析对象文件、库和程序中的符号。在UNIX上，实用程序nm提供了一个符号清单。在Windows上，可以使用实用程序DUMPBIN。需要解析的符号用U或UNDEF来标记，而已定义的符号前面带有一个标记，用来指明包含符号定义的对象文件部分。例如，字母T对应

对象文件的文本部分，也就是对已编译代码的说明部分。表示函数的符号包含在对象文件的文本部分中。

`factorial`示例很小，因此可以在未经过滤的对象文件上使用实用程序`nm`，并读取整个符号表。在实际的项目中，这几乎是不可能的，因此需要在程序上使用搜索或过滤实用程序，如`grep`：

```
> nm -o *.o | grep factorial
factorial.o:00000000 T factorial
main.o:           U factorial
```

在本例中，实用程序`nm`显示了当前目录中所有对象文件中的符号的清单，而实用程序`grep`过滤出了我们感兴趣的函数名称。这里使用`nm`命令的`-o`选项将文件名显示在行首，否则`grep`命令的结果就不是特别有用了。结果显示，符号`factorial`是在对象文件`factorial.o`中定义的。将此文件添加到链接器的参数列表中，链接器就能够正确构建程序了。

如果项目的对象文件或库中均不含有丢失的符号，那么可以使用相同的策略在以下位置搜索定义。

- 在程序的源代码中。如果代码未被编译到对象文件中，则需要修复构建系统。
- 在系统库中。由于计算机上通常有大量的系统库，所以很容易查询联机文档。例如，如果在UNIX上有一个未定义的符号`sqrtf`，则`man sqrtf`命令将显示此函数在`math`库中，并且需要链接器参数`-lm`。
- 在第三方软件库中。如果程序中计划包含其他公司或开发人员的软件，则丢失的符号可能会在它们的软件库中找到。
- 在计算机或网络外部。如果系统定义所在的对象文件不在我们自己的系统上，则可能需要用因特网搜索引擎来查找丢失的库或符号。你可能会惊奇地发现很多程序员遇到并解决了与你同样的问题。

9.3.3 链接顺序问题

如果直接将对象文件作为链接器的参数来提供，那么它的所有符号都将被链接到最后的可执行文件中，这是一个重要方面。尤其是当链接那些包含各种服务函数（这些函数一般用于多个项目）的对象文件时，可能会得到我们不想要的结果，即产生了一个不必要的、过大的可执行文件。为了避免此问题，可以使用库或归档文件（archive file）来代替对象文件。链接器可以从库中挑选那些包含当前未定义符号的对象文件。要注意的重要一点是，大多数链接器对提供给它的对象文件和库只进行一次处理，在处理时依据以下算法。

- 首先，链接器载入对象文件，对象文件包含程序的初始化代码。初始化代码也调用C程序的`main`例程，也就是说，链接器至少以未定义的符号`main`开始。编译器驱动程序自动将包含初始化例程的对象文件提供给链接器。

- 然后，链接器按照链接行参数的顺序继续走查指定的对象文件和库。链接器从每个库中将那些至少包含一个未解析符号的对象文件挑选出来。它从对象文件中挑出所有符号，无论是否对链接器列表中标记为未定义的符号进行解析。当将新的对象文件载入链接器时，可能会产生新的未定义符号。

- 如果在链接行的结尾，所有符号均已解析，则链接过程成功。

在我们的示例中，使用UNIX实用程序ar从对象文件factorial.o生成了库libfactorial.a。ar是一个用来创建或修改库的程序，它还可以从库中提取对象。在下面的链接步骤中，暂时忽略链接器参数正确排序的重要性。即使链接器所需的各部分信息都存在，它还是会报告丢失了符号：

```
> ar -r -o libfactorial.a factorial.o
ar: creating libfactorial.a

> gcc -o calc_factorial libfactorial.a main.o
main.o:main.c:(..): undefined reference to 'factorial'
```

知道了编译器的工作原理以后，这种行为就很好解释了：当链接器到达libfactorial.a库时，其列表中只有未定义的符号main。在libfactorial.a中，没有为此符号提供定义的对象文件，而且库中定义的factorial符号未被挑选出来，因为链接器中没有它的未解析符号。下一步，链接器载入对象文件main.o，这里，main符号已定义，但main()例程还调用函数factorial()。因此，它在未定义的符号列表中创建一个新的未解析的项。由于链接器现在已到达参数列表的末尾，所以程序终止，并给出一条错误消息和一个丢失的符号列表。

如果改变对象文件main.o和库libfactorial.a的顺序，则链接过程将成功完成：

```
> gcc -o calc_factorial main.o libfactorial.a
```

9.3.4 C++符号和名称改编

C++的命名空间、类、模板或函数重载等特性使得编译和链接更为复杂。这使得符号不匹配更有可能引发链接问题。甚至用C和C++编译器编译相同的C源代码也会得到不同的符号：

```
> gcc -c main.c
> nm main.o | grep factorial
U factorial

> g++ -c main.c
> nm main.o | grep factorial
U __Z9factoriali
```

虽然在C中将factorial()函数编译为名为factorial的符号就足够了，但在C++中却并非如此。C++允许重载函数，即可以有多个名为factorial的函数，它们的参数个数和类型各有不同，返回值的类型也不相同。为了生成C++代码元素的唯一符号，编译器使用一种称为名称改编

(name mangling) 的技术，它将对象或函数的准确规格说明（如命名空间和函数参数的个数及类型）编码到符号中。遗憾的是，几乎每个编译器都有自己的名称改编规则集，甚至同一个编译器的不同版本也可能产生互不兼容的对象代码。

9.3.5 符号的反改编

为了跟踪这些不兼容性，必须仔细观察经过名称改编的符号。这样的符号读起来很单调乏味，而且易出错，因此更好的方法是对这些符号进行反改编（demangle）。有两种反改编方法，一是使用分析实用程序（analysis utility），它们可以在改编和反改编的输出之间切换，二是使用过滤器（如c++filt）。通过选项-C或--demangle可以调用nm实用程序，得到实际的用户级别的符号名称。这些过滤器将加密的参数和类型编码转换回源代码，因此很容易看到符号的原始形式。

注意，名称改编依赖于编译器，因此我们必须保证使用匹配的实用程序和工具：

```
> nm main.o | grep factorial  
U __Z9factoriali  
  
> nm main.o | grep factorial | c++filt  
U factorial(int)
```

9.3.6 链接 C 和 C++ 代码

正如我们看到的，用C++编译器和C编译器编译同一段代码得到了不同符号，这常常也会导致链接错误：

```
> gcc -c factorial.c  
> g++ -c main.c  
> gcc -o calc_factorial main.o factorial.o  
main.o:main.c:(...): undefined reference to `factorial(int)'
```

值得注意的是，仅使用C编译器驱动程序并不能保证源代码实际上被编译为C代码，编译器驱动程序可能根据文件扩展名来决定生成C还是C++风格的符号。例如，如果调用编译器驱动程序gcc来编译扩展名为.c的文件，则此文件被编译为C代码，而扩展名为.cc的文件中的相同代码将被编译为C++代码。C++文件经常使用的扩展名还有.c、.cpp或.cxx。

```
> gcc -c main.c  
> nm main.o | grep factorial  
U factorial  
> cp main.c main.cc  
> gcc -c main.cc  
> nm main.o | grep factorial  
U __Z9factoriali
```

为了避免上述链接问题，最好严格限定文件扩展名，或给出明确的编译器指令来选择目标格式。

尽管如此，我们并不是总能够用同一个编译器来编译所有源代码。C++程序经常需要链接C对象文件中定义的符号，这往往导致上述链接问题。解决办法是明确地通知C++编译器不对那些需要由C对象文件中的符号来解析的未定义符号进行名称改编。这可以通过在相应元素的声明中使用`extern "C"`编译器指令来实现。在图9-1的示例中，在`extern "C"`代码块中只给出了`factorial()`函数的声明，这样就能够用C++编译器来编译`main.cc`，而且得到的对象文件仍然可以被链接到C库（此库中定义了所需的符号`factorial`）。这样，链接程序时就不会再生成未定义的符号。

```
> g++ -c main.cc
> nm main.o | grep factorial
    U factorial

1 extern "C" {
2     unsigned factorial(unsigned n);
3 }
```

图9-1 头文件`factorial.h`的`extern "C"`声明

9.4 具有多个定义的符号

与丢失符号相反的是，链接器在链接阶段发现同一符号的多个定义。在最好的情况下，链接器报告这一点，从而在链接程序时能够定位bug的来源。但在很多情况下，链接器只选择它在命令行所指定的对象文件和库文件中找到的第一个定义。如果幸运的话，第一个定义恰好是正确的，这样程序将按预期工作，但通常并不会如此幸运。

下面举例说明，假设除了在库文件`libfactorial.a`中有一个`factorial`定义之外，在`sfactorial.c`文件中还定义了一个桩函数`factorial()`。

```
1 #include <stdio.h>
2
3 int factorial(int n)
4 {
5     fprintf(stderr,"Stub for factorial is called\n");
6     return 0;
7 }
```

如果在指定链接器参数的顺序时，强制链接器同时选择这两个符号，那么它将产生一条错误消息，指明同一名称的两个符号产生冲突，并给出这两个符号都是在哪里定义的：

```
> gcc -o calc_factorial main.o libfactorial.a sfactorial.o
sfactorial.o:sfactorial.c: multiple definition of 'factorial'
libfactorial.a(factorial.o):factorial.c:4: first defined here
```

9.5 信号冲突

下面改变命令行参数的顺序，让链接器自由选择第一个符号定义，并忽略所有后面的规定。链接过程可以成功完成，但最后的程序行为不符合预期：

```
> gcc -o calc_factorial main.o sfactorial.o libfactorial.a
> ./calc_factorial 5
  Stub for factorial is called
  factorial 5!=0
```

在这种情况下，无论输入什么数字，`calc_factorial`程序的结果总显示为0。

以上代码片段还说明了另一个好的思想，即用一些安全机制，如断言、调试输出或无效返回值，清晰地标记那些不应被调用的桩函数。如果不这样做，可能要花费非常多的调试时间来查找被错误执行的函数。

这个问题就称为符号冲突（symbol clash），即相同符号存在多个定义，而在链接时并未检测到这种歧义。

那么，如何分析符号冲突呢？

调用错误的函数可能会导致内存破坏。其症状是没有任何意义的奇怪的核心转储，程序执行到一个看起来毫无问题的函数时突然崩溃，类对象的数据成员的值突然被破坏，而内存检查器又无法找到哪条语句破坏了内存。

如果函数是C函数，则调用者和被调用者的参数的成员和类型不同。如果符号是一个类方法，则访问任何类成员都将发生错误，因为这两个类的内存布局几乎是完全不同的。这可能立即导致一个错误，例如分段错误（segmentation fault），或者是很久之后才能检测到的内存破坏。

在应用程序中发现符号冲突之后，有以下几种解决方法。

- 明显的解决方案（特别是在能够访问源代码的时候）是重新命名那些导致符号冲突的变量和函数。
- 也可以考虑引入C++命名空间。
- 可以在源代码（或者在开发流后面的链接器标志）中使用`static`限定符来限制特定符号的导出。这个过程也称为符号的本地化。
- 最后可以借助类似`objcopy`（GNU binutils的一部分）的工具，在Windows上则可以使用`EDITBIN`或`LIB`。除了对符号进行本地化以外，这些工具还可用于重新命名符号引用和定义。

9.6 识别编译器和链接器版本不匹配

当谈到编译器或链接器版本不匹配时，实际上是指这样一个事实：当使用不同的ABI（application binary interface，应用程序二进制接口）时，需要使用不同的C++编译器来生成代码。

ABI是一组运行时惯例，并且带有用于处理程序二进制表示的所有工具，包括编译器、汇编程序和链接器。不同版本的C++编译器之间的一个明显的区别是使用的名称改编惯例不同，另一个区别是随同C++编译器和链接器一起安装的库（例如libstdc++.a）中定义的系统库符号不同。在链接期间，定义的符号与使用的符号将不再匹配。链接器将报告未定义的符号，尽管所有链接器参数和它们的顺序看起来都是正确的。为了演示这一点，本节使用不同的编译器/链接器版本，即将现在的版本和过去已淘汰的版本作为对比。当然，“现在”是指撰写本书的时候，但无论是现在还是将来，使用不同编译器和版本的原则（及问题）都是相同的。

值得注意的是，名称改编中的这些不兼容性常常是编译器开发人员有意引入的，目的是防止编译器用户遇到更隐晦的ABI不匹配问题。这样，不兼容的库就可以在链接时检测出来，而不是程序在运行时莫明其妙地崩溃。

符号不匹配主要发生在C++中，因为C编译器和链接器不必支持函数重载、模板等概念和所有其他特性，而这些特性正是为C++编译器引入名称改编的原因。但即使在纯粹的C编译和链接中，仍可能会遇到不一致的ABI。^①

9.6.1 系统库不匹配

当用同一个编译器编译所有对象文件，但用另一个不同编译器（版本）来链接的时候，就会发生系统库不匹配问题。

C++编译器通过集成系统库实现了一些特定于C++的特性，例如动态转换。GCC 3.x编译器将libstdc++库链接到编译后的可执行程序中。为了使用动态转换特性，对象文件和库必须匹配。

当用GCC 2.95.3生成对象文件，而用GCC 3.3.5进行链接时，将会产生大量未定义的符号，像下面这样：

```
9
undefined reference to '__throw'
undefined reference to '__builtin_delete'
undefined reference to '__builtin_new'
undefined reference to '__rtti_user'
undefined reference to 'cout'
undefined reference to 'endl(ostream &)'
undefined reference to 'ostream::operator<<(char const *)'
```

如果链接器报错，指出丢失了内部使用的函数和方法（如cout或__throw）的符号，那么很可能是因为对象文件与系统库之间不匹配。C++编译器内部使用的函数通常以双下划线为前缀，例如__dynamic_cast。

9.6.2 对象文件不匹配

当试图链接用不同编译器（或版本）生成的对象文件时，就会发生对象文件不匹配问题。最

^① 引自GNU在线文档，网址为<http://gcc.gnu.org/onlinedocs/gcc-4.0.3/gcc/Interoperation.html>。

常见的情况是必须使用那些无法获得源代码的对象文件或库，例如第三方库文件。在这种情况下，简单的修复方法不再有效，即无法使用相同编译器和链接器从头构建一切。

编译器不匹配的症状是链接器报错（有符号丢失），但这些符号在对象文件或库中都存在，而且链接顺序看起来也是正确的。

使用nm（在UNIX上）或DUMPBIN（在Windows上）工具对两个对象文件的符号表进行分析：一是包含未解析符号的表，二是具有相应定义符号的表，将得出这样一个结果，已反改编的用户级别的符号是相同的，而经过名称改编的符号不同。下一步就是找出用于生成不同对象文件的编译器版本（参见9.6.4节），并决定使用哪个编译器来编译和链接所有对象文件和库。

注意，即使微小的差别（如编译器的发布版本号不同）也可能引起上述链接问题。

9.6.3 运行时崩溃

即使程序在链接时未报告错误消息，链接过程中仍有可能会发生错误，通常这种错误导致程序在运行时产生一个核心转储。那么，如何将这些核心转储与其他问题（例如第4章讲过的内存问题）引起的核心转储区分开呢？核心转储通常是由于使用了特定的C++函数引起的。这些函数以双下划线为前缀。也就是说，我们应该在核心文件的栈跟踪或者在调试器中运行程序时检查这样的函数。当然，栈跟踪中的特定C++函数并不总是表明编译器或链接器不匹配。我们仍然需要像第4章所讲的那样用内存调试工具检查源代码。

例如，当用GCC 2.96编译源代码，而用GCC 2.95.3进行链接时，在使用动态转换时就可能触发核心转储。

修复这个问题的方法很简单：用同一个编译器重新编译所有源文件，并且在链接时也使用这个编译器。

9.6.4 确定编译器版本

一个问题是是如何知道现有对象文件是用哪个编译器版本生成的。特别是当无法获取这些对象文件或库的源代码，而且无法用同一个编译器重新构建一切时，该如何知道编译器版本呢？

一种几乎适用于所有UNIX风格的平台的方法是在对象文件中搜索特定字符串，这些字符串给出了有关编译器版本的暗示。可用的方法是使用UNIX工具（如strings和grep）或编辑器（如emacs）。

那么搜索什么字符串呢？这取决于使用的编译器，这里无法给出所有过去、现在和未来编译器的完整清单，但可以给出一些关于最常用的编译器的提示。在使用其他编译器时，也可以搜索同类信息。

- GNU编译器包含字符串“(GCC)”，它后面跟着版本号。
- Sun编译器包含字符串“Forte Developer”或“Sun”，它后面跟着编译器版本号或补丁号。

- HP编译器中没有直接的版本字符串，但通常给出系统文件所在的完整路径名，路径中含有编译器的版本号。

以下是在UNIX/Linux上搜索编译器版本字符串的示例：

```
> strings -a factorial.o | egrep "(GCC|GNU|Sun|Forte|HP)"
GCC: (GNU) 3.3.2
```

另一种猜测编译器版本的方法是查看它在对象文件中生成的符号，例如，可以使用UNIX实用程序nm来查看。不同的编译器在编译特定的C++语言结构或标准的C++类时使用不同的符号。在有疑问的情况下，可以创建一个如图9-2所示的小测试文件，使用在整个程序中使用的编译器来编译它，并用nm检查生成的对象文件。如果对象文件和测试文件中显示了相同种类的符号，则此编译器极有可能也是被用于创建对象文件的编译器。

```
1 #include <iostream>
2
3 struct B { virtual int foo(); };
4 struct D: public B { virtual int foo(); };
5 void test() {
6     std::string S("abc");      // std::string::string(...)
7     std::string S2(S);         // ditto
8     int* a = new int[4];       // operator new[]
9     delete a[];               // operator delete[]
10    D* d1 = new D;
11    B* b = d1;
12    D* d2 = dynamic_cast<D*>(b); // dynamic_cast
13 }
```

图9-2 示例文件test_file.cc

编译文件并用nm检查符号名称：

```
> g++ -c test_file.cc
> nm test_file.o | grep " T "           # output in mangled form
  000000c4 T _Z4testv
> nm -C test_file.o | grep " T "        # output in C++ notation
  000000c4 T test()
```

不同的编译器家族（如GCC和Sun Forte编译器cc）具有不同的名称改编模式。每种编译器都有自己的c++filt版本，而且工作方式各不相同。这可能已经给出了有关编译器的线索。如果用C++环境中当前使用的c++filt工具来转换对象文件中已改编的符号，而得到的结果并不像是一个C++签名，那么对象文件很可能是用另一个不同编译器编译的。

在下面的示例中，test_file.cc文件已经用Sun Forte编译，而正在使用GCC安装中的c++filt实用程序。

```
> /opt/SUNWspro/bin/CC test_file.cc
> nm test_file.o | grep " T "
  00000010 T __lcEtest6F_v_
> nm test_file.o | grep " T " | c++filt
```

```
00000010 T __lcEtest6F_v_
> nm test_file.o | grep " T " | /opt/SUNWspro/bin/c++filt
00000010 T void test()
```

这个例显示了使用不匹配的c++filt版本没有任何效果。有些情况下，c++filt会发出错误消息，指出没有关于如何处理当前格式的可用信息。

c++filt只能可靠地测试不同编译器产品家族之间的版本不兼容。当测试同一编译器家族（如GCC）的不同版本时，要格外小心，因为一些c++filt版本具有智能功能，它们自动检测输入的格式。因此，即使有版本不匹配问题，也会在c++filt输出中看到正确的C++名称。

9.7 解决动态链接问题

我们可以不使用静态链接的方法，而是将含有对象代码的库动态链接到可执行程序中。在UNIX和Linux上，这样的DLL或共享对象文件（shared object file）的扩展名一般为.so，在Windows上的扩展名则是.dll。本书不讲解各种编译器如何创建DLL，相关信息可参考编译器文档。

DLL有几个优点，如链接时间较短，并且得到的可执行程序更小。它的缺点是略微延缓了程序的执行速度，此外还有一些DLL特有的链接和运行时问题。本节主要讨论这些问题。

9.7.1 链接或载入 DLL

可以使用两种方式将DLL链接到可执行程序，一是直接链接，二是在运行时通过显式调用载入。

以下示例给出了在GCC中将DLL指定为链接器参数的方法：

```
> gcc -o myprog main.o libA.so
```

链接器不会将libA.so的实际内容链接到myprog程序中，而是创建到它的一个引用。每当程序启动时，都会自动载入libA.so。

在Visual Studio中，不使用实际的DLL（文件扩展名为.dll）来链接，而是使用一个专门的导入库，它与静态库具有相同的扩展名（.lib）。

在Linux上，可以通过dlopen()在运行时访问DLL，在Windows上具有同样功能的函数是LoadLibrary()。这样，用户就能够控制是否载入DLL，以及何时载入。用户可以选择在程序刚刚启动时或运行过程中载入，也可以不载入。

在载入DLL之后，就可以通过系统调用访问其中的符号了，在UNIX或Linux上的调用函数是dlsym()，在Windows上的调用函数是GetProcAddress()。这些函数返回指向各自变量和例程的指针，如果DLL没有导出具有给定名称的符号，则返回空指针。

图9-3中的示例显示了如何在GCC中使用这两种技术：函数show1()位于libutil1.so中，libutil1.so被直接链接到程序中。函数show2()位于libutil2.so中，libutil2.so在main.c

程序的第13行被载入。图9-4中的命令显示了DLL和myprog程序的编译和链接命令行。如果想用GCC来构建共享的对象文件，必须使用-fPIC标志来创建与位置无关的代码，这样的代码适合动态链接。要使用GCC生成共享的对象，必须使用-shared标志。注意，在链接时只有libutil1.so是作为链接器参数提供的，而libutil2.so是被载入的。

```

1 /* dll_issue_example.c */
2 #include <dlfcn.h>
3 #include <stdio.h>
4 void show1(char* msg, int value);
5
6 int main(int argc, char* argv[])
7 {
8     void *dll;
9     void (*fn)(char*,int);
10
11    show1("pol", 42);
12
13    if((dll=dlopen("./libutil2.so",RTLD_NOW|RTLD_GLOBAL))!=0) {
14        fprintf(stderr,"%s\n",dlerror());
15        return 1;
16    }
17    if((fn = dlsym(dll,"show2")) == 0) {
18        fprintf(stderr,"%s\n",dlerror());
19        return 2;
20    }
21    (fn)("pol", 42);
22    return 0;
23 }
```

图9-3 通过链接和载入使用DLL

```

> gcc -g -c -fPIC util1.c
> gcc -shared -o libutil1.so util1.o
> gcc -g -c -fPIC util2.c
> gcc -shared -o libutil2.so util2.o
> gcc -o myprog -g main.c libutil1.so -ldl
```

图9-4 构建和链接共享对象

9.7.2 无法找到 DLL 文件

使用DLL的一个很常见的问题是程序在启动期间给出一条错误消息，即无法载入或找到DLL或共享对象。被链接到程序中的DLL也会发生这种情况。例如，运行图9-3所示的代码时，就会发生此错误：

```

> ./myprog
./myprog: error while loading shared libraries:
libutil1.so: cannot open shared object file:
No such file or directory
```

当程序启动时，一个启动例程（startup routine）自动尝试载入所有被链接到程序中的DLL。那么，程序如何知道载入哪些DLL呢？到哪里去寻找它们？大多数开发环境都有默认的库目录，程序在这些目录中搜索DLL。此外，在构建程序时还可以为链接器指定更多搜索路径。最后（但并非不重要），还可以设置环境变量，如在UNIX上可以使用LD_LIBRARY_PATH，在Windows上可以使用PATH，这些环境变量的作用是在载入器的搜索路径中添加目录。

用于控制DLL搜索顺序的准确规则更为复杂，但环境变量对于用户定义的DLL来说是最重要的，即UNIX上的LD_LIBRARY_PATH和Windows上的PATH。

实用程序可以帮助查明程序中使用了哪些DLL。例如，在UNIX上，ldd命令输出一个清单，列出程序都需要哪些动态库，以及它们都来自何处。

```
> ldd myprog
    linux-gate.so.1 => (0xfffffe000)
    libutil11.so => not found
    libdl.so.2 => /lib/libdl.so.2 (0x4002d00)
    libc.so.6 =>/lib/tls/libc.so.6 (0x4003100)
    /lib/ld-linux.so.2 (0x4000000)
```

注意，除了libutil11.so未找到以外，其他所需的库都位于系统上。

如果某个特定的库未找到，则应该检查它是否确实不在库的搜索路径上。DLL搜索路径包括：

- 由环境变量LD_LIBRARY_PATH或PATH指定的一个目录列表；
- 与操作系统或机器有关的一个系统库目录（如/lib或/usr/lib）列表。

注意，错误的搜索顺序也可能导致问题，特别是当系统上存储了相同DLL的不同版本的时候。更多信息可参考链接器文档。

9.7.3 分析载入器问题

如果没有适当的工具，很难调试9.7.2节所讲的库搜索路径丢失或错误问题。大多数UNIX风格的系统都支持在运行时对载入器进行分析，即记录或跟踪在程序运行期间载入器的每个步骤。举例来说，在这些跟踪文件或日志文件中，可以看到搜索了哪些目录，以及最后在哪里找到了DLL。通过设置环境变量LD_DEBUG即可启用这项载入器调试功能。如果将环境变量设置为libs，然后再用载入器来调用程序，则它将报告与查找所需DLL有关的每个步骤。

将环境变量LD_DEBUG设置为libs，然后运行9.7.1节给出的程序myprog，将得到类似于下面这样的跟踪记录。第一个数字表示使用载入器的进程ID。接下来列出此进程所需的每个DLL及其搜索路径。然后载入器在所有目录中搜索，直到它找到所需的DLL，如果未找到则报告一个错误。

```
> setenv LD_DEBUG libs
> ./myprog
2420:  find library=libutil11.so [0]; searching
2420:      search path=.:/tls/i686/sse2:/tls/i686:
                  .:/tls/sse2:/tls:/i686/sse2:
```

```

        ./i686./sse2.. (LD_LIBRARY_PATH)
2420:   trying file=./tls/i686/sse2/libutill.so
...
2420:   trying file=./i686/libutill.so
2420:   trying file=./sse2/libutill.so
2420:   trying file=./libutill.so
2420:
2420:   find library=libdl.so.2 [0]; searching
2420:   search path=./tls/i686/sse2./tls/i686:
        ./tls/sse2./tls./i686/sse2:
        ./i686./sse2.. (LD_LIBRARY_PATH)
2420:   trying file=./tls/i686/sse2/libdl.so.2
...
2420:   trying file=./i686/libdl.so.2
2420:   trying file=./sse2/libdl.so.2
2420:   trying file=./libdl.so.2
2420:   search cache=/etc/ld.so.cache
2420:   trying file=/lib/libdl.so.2
2420:
...

```

大多数载入器提供了更多分析功能，将环境变量LD_DEBUG设置为不同的值即可启用这些功能。如果将LD_DEBUG设置为help，可以显示载入器支持特性的简短清单：

```

> setenv LD_DEBUG help
> ./myprog
Valid options for the LD_DEBUG environment variable are:
libs      display library search paths
reloc     display relocation processing
files     display progress for input file
symbols   display symbol table processing
bindings  display information about symbol binding
versions  display version dependencies
all       all previous options combined
statistics display relocation statistics
unused    determine unused DSOs
help      display this help message and exit

```

也可以不使用标准输出，而是将调试信息输出到文件中，方法是使用环境变量LD_DEBUG_OUTPUT，并指定一个文件名。

9.7.4 在 DLL 中设置断点

如前所述，DLL可以链接到程序中，也可以在运行时通过显式的dlopen()调用载入。大多数调试器都有不同的DLL使用模式和命令。

默认的设置是尽可能早地为DLL载入调试信息。链接到程序中的DLL的调试信息是在程序启动时读取的，而运行时载入的DLL的调试信息是在调用dlopen()或LoadLibrary()打开DLL时读取的。当调试器被链接到程序时，将为此时已打开的所有DLL读取调试信息。GDB命令shared library可以显示哪些DLL已经载入，并且为尚未载入调试信息的DLL载入调试信息。在Visual

Studio中，可以暂停程序的执行，并通过菜单项**Debug/Windows/Modules**（调试/窗口/模块）打开一个窗口，此窗口中显示所有已载入的DLL。

由于DLL的载入具有动态性，所以包含DLL的源代码只有在载入DLL之后才能进行调试。在此之前，没有可用的调试信息，因此也无法直接设置断点。当开始调试全新的、不甚了解的程序时，这特别容易引起混淆。此外，我们经常不易区分哪些源代码被编译到程序的静态部分中，哪些被编译到DLL中。如果在特定函数中设置断点时发生不明原因的失败，则说明此函数被编译到DLL中。

大部分较新的调试器都提供了一些用来解决此问题的功能，即将断点设置为挂起状态。无论何时，当载入一个新的DLL时，调试器尝试解析挂起的断点。

可以用图9-3中的代码来试一下此功能：在show1中设置断点可以立即完成，而在show2中设置断点则是挂起状态，直到第13行载入DLL：

```
> gdb myprog
...
(gdb) break show2
Function "show2" not defined.
Make breakpoint pending on future shared library load?
(y or [n]) y

Breakpoint 1 (show2) pending.
(gdb) run
Starting program: [...]/linking_problems/example4/myprog
Breakpoint 2 at 0xa7a882: file util2.c, line 5.
Pending breakpoint "show2" resolved
util1, pol:value=42

Breakpoint 2, show2 (msg=0x.. "pol", value=42) at util2.c:5
5         printf("util2, %s:value=%d\n", msg, value);
(gdb)
```

9.7.5 提供 DLL 问题的错误消息

如果载入动态库或访问其符号失败，可以使用错误报告函数来生成人们可读的错误解释。在UNIX上，错误处理函数是**dllerror()**。在Windows上，可以使用**GetLastError()**和**FormatMessage()**函数来提供有用错误消息。也可以从调试器调用这些错误报告函数。

在图9-3的示例中，第14行和18行使用了**dllerror()**。如果在启动myprog程序之前删除第13行需要载入的DLL，将看到以下结果：

```
> rm libutil2.so
> ./myprog
util1, pol:value=42
./myprog: error while loading shared libraries: libutil2.so:
cannot open shared object file: No such file or directory
```

如果没有第14行的`dllerror()`发出的错误消息，程序将默默地从`main()`例程返回，不会有任何错误消息。

经验总结

- 如果链接器报告了未定义的符号，一定要确定这些符号是在哪里定义的，并确保提供正确的编译单元（对象文件或库）作为链接器参数。
- 链接器参数的顺序很重要，对于编译单元（如对象文件和库）和搜索路径来说都是如此。
- 用C和C++编译器编译相同函数会产生不同符号。
- 如果链接器报告了未定义的符号，而又找不到任何明显的链接错误，则错误可能是由于在编译单元中使用了不同的ABI引起的。当使用不同的编译器或不同的环境设置来创建编译单元时，可能发生此问题。
- 上网搜索一下丢失的符号，很有可能其他开发人员也曾遇到相同的链接问题。
- 具有多个定义的符号也会产生问题，尤其是运行时符号冲突更难分析。在源代码中重新命名符号、引入命名空间或在编译后改编名称可以作为部分解决方案。
- DLL提出了特殊的问题。要注意链接器和运行时载入器给出的搜索路径和调试帮助。通常，只有在载入相应的DLL之后，才能设置断点。



本章讨论一些高级调试内容，每节介绍一个主题。其中有些主题是通用的，而有些则是个别调试器所特有的。

10.1节至10.5节介绍如何用调试器在C++代码中导航，重点讨论函数重载、隐式函数、模板和静态构造/析构函数。10.6节介绍与数据有关的断点，即观察点。随后的几节介绍如何调试信号和异常、如何读栈跟踪、如何修改正在运行的程序，以及如何调试那些不带调试信息的程序。

10.1 在 C++ 函数、方法和操作符中设置断点

使程序在相关位置暂停是一个重要的调试技巧。正如第3章所述，我们通过在具体函数或行中设置断点使程序暂停。但是，在C++方法、函数和操作符中设置断点时，调试器可能会拒绝设置断点，并报告没有这样的函数，或有多个函数而无法选择。

C++支持函数重载，因此可能有多个同名C++函数，这些函数具有不同的参数类型和参数个数。为了找到函数，调试器必须知道它的准确签名（signature）。在C++中，函数签名包含名称、类名称、所有参数的类型和命名空间。在模板函数特化（template function specialization）中，签名还包括模板参数。签名的用途是使调试器能够区分函数的多个版本。在链接程序时，也会用到签名（参见第9章）。

将正确的C++函数签名输入到调试器中是很费时间的，因此应该让调试器帮助我们选择所需的C++函数名称。GDB和Visual Studio提供了几种有用的特性，下面举例说明：

```
1 class C {
2 public:
3     C(int a)           : n(a) {}
4     int foo(int a)      {return n=a;}
5     int foo() const    {return n;}
6     int foo(char c, bool b) {return n+=b+c;}
7     C& operator=(const C& r) {n=r.foo(); return *this; }
8 private:
9     int n;
10};
```

```

11
12 int main(int argc, char* argv[]) {
13     C ca(0);
14     C cb(1);
15     ca.foo(-23);
16     cb=ca;
17     return cb.foo('A',false);
18 }
```

在GDB中启动程序，并在第4行的成员函数foo中设置一个断点。首先尝试使用break C::foo命令来设置断点：

```
(gdb) break C::foo
[0] cancel
[1] all
[2] C::foo(char, bool) at main.cpp:6
[3] C::foo() const at main.cpp:5
[4] C::foo(int) at main.cpp:4
```

方法名称foo并未指定参数，所以名称是不明确的，GDB输出所有匹配方法清单。一种解决办法是从GDB提供的候选签名中选择一个签名输入。例如break C::foo(int)将使程序在第4行停止。

也可以键入部分签名，然后让GDB来完成其余工作。GDB提供了一种方便的机制，可以自动完成签名，这类似于命令行解释器自动完成文件。在函数前面加单引号，后跟Tab键或ESC ?，即可触发自动完成机制。例如，键入break 'C::foo，后跟Tab键。

使用GDB的ptype命令可以找到类方法的更多信息。此命令列出类的所有元素（包括方法）：

```
(gdb) ptype C
type = class C {
    private:
        int n;

    public:
        C(int);
        int foo(int);
        int foo() const;
        int foo(char, bool);
        C & operator=(C const&);
}
```

GDB命令info functions <expr>将查找与<expr>匹配的所有全局函数和成员函数：

```
(gdb) info functions C::foo
All functions matching regular expression "C::foo":

File main.cpp:
int C::foo() const;
int C::foo(char, bool);
int C::foo(int);
```

在Visual Studio调试器中，打开**Debug**(调试)菜单，单击菜单项**Breakpoint/Break at Function...**

(切换断点/在函数处中断), 然后输入c::foo。这将建立3个断点, 每个断点对应一个名为c::foo的函数。要为重载的函数设置断点, 可以输入函数名称和参数类型。例如, c::foo(int)将在第4行设置一个断点。

经验总结

- 由于C++支持函数重载, 所以在C++设置断点更为复杂。我们可以通过简单的C++示例来熟悉调试器命令和特性, 列出完整函数签名, 并处理不完整的签名。

10.2 在模板化的函数和C++类中设置断点

GDB的一个特有问题是处理模板中的断点。当在模板化的函数或C++类中的特定行设置断点时, 程序可能不会在那里停止。下面解释其中的原因, 以及如何克服这个问题。考虑以下示例, 它包含一个模板化的函数myFunction:

```

1 #include <iostream>
2
3 template <class T>
4 void myFunction(T value)
5 {
6     std::cout << "got " << value << std::endl;
7 }
8
9 int main(int argc, char* argv[]) {
10    myFunction(100);
11    myFunction('A');
12    myFunction(true);
13    return 0;
14 }
```

我们希望在每次调用myFunction时停止, 因此在第6行设置一个断点。GDB接受了断点命令, 但当运行程序时, 调试器没有停止三次, 而只停止一次:

```

(gdb) break main.cpp:6
Breakpoint 1 at 0x8048882: file main.cpp, line 6.
(gdb) run
got 100

Breakpoint 1, myFunction<char> (value=65 'A') at main.cpp:6
6     std::cout << "got " << value << std::endl;
(gdb) cont
got A
got 1
```

为什么GDB没有在每次调用myFunction时停止呢? 当用新的参数初始化模板时, C++编译器将创建不同的对象代码。每个模板实例中的调试信息将回头引用模板代码中的相同源代码行。在模板中的某一行设置断点将导致GDB只在一个实例中设置断点。在上例中, break main.cpp:6

命令在`myFunction<char>`中设置了一个断点，而其他实例（例如`myFunction<int>`）都没有被设置断点，因此GDB在执行到这些实例时不会停止。

应该使用`info breakpoints`命令来查明GDB实际上选择了哪个模板实例：

```
(gdb) info breakpoints
Num Type Disp Enb Address What
1 breakpoint keep y 0x.. in void myFunction<char>(char)
                           at main.cpp:6
breakpoint already hit 1 time
```

为了在执行到不同的模板实例时停止，必须在函数签名（而不是源代码行）中设置断点。参见10.1节中有关在C++函数中设置断点的内容。

```
(gdb) info functions myFunction
All functions matching regular expression "myFunction":

File main.cpp:
void void myFunction<bool>(bool);
void void myFunction<char>(char);
void void myFunction<int>(int);
```

按照这种方法，可以在`myFunction`的3个模板实例中设置断点。现在，可以看到GDB在3个函数调用中都停止了：

```
(gdb) break void myFunction<bool>(bool)
Breakpoint 2 at 0x80488b6: file main.cpp, line 6.
(gdb) break void myFunction<int>(int)
Breakpoint 3 at 0x80488e8: file main.cpp, line 6.
```

注意，这里讨论的是在GDB 6.5及更早版本中存在的问题，GDB的较新版本可能已经修复此问题。但是，不管使用哪种调试器，都要注意适用于模板实例化和断点的微妙规则。不同的实例化将会导致在不同内存地址有多个函数。一个特定断点只能用于一个实例。

如果使用Visual Studio在第6行设置了断点，则它将创建3个断点，每个断点对应一个模板实例，这比GDB更直观。

经验总结

- 在模板中的某一行设置断点可能不够明确，通过函数签名设置断点能解决此问题。
- 对模板化的函数或成员要多加注意。GDB断点只能用于模板的一个实例，而并非所有。

10.3 进入C++方法

在C++中，一条简单的C++代码语句可能包含很多隐式或“隐藏的”函数调用。例如，语句`A=B;`可能包含对赋值操作符或类型转换操作符的调用。利用类构造函数、操作符、隐式转换等功能，可以用非常紧凑的源代码来表示复杂操作。这可以实现很好的抽象和很高的代码密度，而且

可以用类库来扩展语言。紧凑的C++源代码的一个缺点是不便于调试，如果隐式函数未按照我们的预期工作，则必须使这些函数可视化。本节介绍如何找到被调用的隐式函数，以及如何用调试器在特定函数中导航。

考虑以下示例，我们有意编写了一个不完善的字符串类STR：

```

1  /* class STR example */
2  #include <string>
3  #include <iostream>
4
5  class STR {
6  public:
7      STR(const char* a) {
8          s=a;
9          num++;
10     }
11     STR(const STR& a) {
12         s=a.s;
13         num++;
14     }
15     ~STR() {
16         num--;
17     }
18     const char* c_str() const {
19         return s.c_str();
20     }
21     const STR& operator+ (const STR& a) {
22         s += a.s; return *this;
23     }
24     int num_objs() const {
25         return num;
26     }
27     STR operator* (int num_copies) const {
28         std::string tmp("");
29         for (int n=0; n<num_copies;n++)
30             tmp += s;
31         return STR(tmp.c_str());
32     }
33 private:
34     static int num; // total #objects of this class
35     std::string s;
36 };
37 int STR::num=0;
38
39 void show( STR z )
40 {
41     std::cout <<z.num_objs() <<": " <<z.c_str() <<std::endl;
42 }
43
44 int main(int argc, char* argv[]) {
45     STR x="abc";
46     show(x);

```

```

47     show((x+"def") * 3);
48     return 0;
49 }

```

10.3.1 用step-into命令进入到隐式函数中

看一下第46行的show(x)语句。看起来函数main()是直接调用函数show()，但它对类STR的构造函数有一个隐藏的调用。调用此构造函数的原因在于函数show()使用了STR参数，而不是使用STR对象的引用，因此编译器在调用函数之前必须创建参数的一个副本。在调试器前进到第46行，并进入到函数调用中，STR构造函数被调用了。然后继续使用step-into命令来前进，直到到达函数show()。

```

...
46     show(x);
(gdb) step
STR (this=0xbffffe00, a=@0xbffffe10) at main.cpp:11
11     STR(const STR& a) {
...
(gdb) step
show (z= ... 0x804b014 "abc" ...) at main.cpp:41
41     cout <<z.num_objs() <<": " <<z.c_str() <<endl;

```

step-into命令可以进入任何隐式函数调用中，因此它是查找隐式函数的好方法。但要理解的重要一点是，step-into命令通常只会在带有调试信息的地方停止。这意味着不带调试信息的隐式函数不容易找到。

注意，step-into命令有时会进入C++编译器提供的函数或方法，或者进入程序所包含的各种库中。在上例中，当步进到类std::basic_string<...>的构造函数中时，就发生了这种情况。

10.3.2 用step-out命令跳过隐式函数

下面对10.3节中的示例第47行的show(x + "def")函数调用进行调试。第47行实际是一个隐式函数调用序列，它等价于（但并不一定完全等于）图10-1中标记为47.1…47.6的伪代码语句。

```

47     show((x+"def") * 3);
47.1     tmp1=STR::STR("def");
47.2     tmp2=STR::operator+(x,tmp1);
47.3     operator*(tmp2,3);
47.4     show(tmp2);
47.5     STR::~STR(tmp2);
47.6     STR::~STR(tmp1);
48     return 0;

```

图10-1 隐式函数调用

通过反复执行step-into命令，可以到达show()的函数体，但这将依次经过函数体的语句47.1至47.3，包括这些行中执行的任何函数调用。这样就需要执行大量的step-into命令，因此很容易

迷失在低层实现细节中。那么，有没有更快的方法可以从第47行直接进入第41行的show()函数中呢？

一种快捷办法是离开当前函数，在GDB中可以使用finish命令，在Visual Studio中则可以使Step-Out（跳出）按钮。

让我们来看一下工作原理。从第47行开始，也就是从假设的伪代码行47.1开始。step-into命令进入到第7行的STR构造函数中。step-out命令离开构造函数，并在第47.2行停止。注意，GDB报告的位置是0x<address> ... at main.cpp:47。所有伪代码行47.1 ... 47.4都属于同一个源代码行47，但它们具有不同的程序地址，所以可以区分它们。下一个step-into命令进入47.2的函数调用。

很难跟踪哪个隐式函数被调用了，但执行一系列step-into、step-out、step-into……命令后，最终就会进入到函数show()中。如果失去线索，可以查看调用栈。在本例中，进入show()函数需要执行4次step-into命令和3次step-out命令，这比执行大量连续的step-into命令要快多了。

10.3.3 利用临时断点跳过隐式函数

在有些情况下，我们可能希望从第47行直接进入到函数show()中，而跳过所有计算函数参数的隐式函数调用。这可以通过设置临时断点来实现，方法是使用GDB命令tbreak：

```
(gdb) tbreak show
Breakpoint 2 at 0x8048a06: file main.cpp, line 41.
(gdb) cont
show (z=..."abcdefabcdefabcdef"...) at main.cpp:41
41    cout <<z.num_objs() <<": " <<z.c_str() <<endl;
```

注意，GDB在第一次遇到临时断点时会自动删除它们。GDB调试器在遇到它们时也不会报告断点数。

在Visual Studio中，通过在第41行单击鼠标左键设置断点，然后按F5键继续。当调试器在第41行停止时，再次单击此行即可删除断点。

10.3.4 从隐式函数调用返回

如果在嵌套的隐式函数调用中失去线索，也可以使用临时断点。再次从第47行开始，反复使用step-into命令，直到到达第7行中的第二个构造函数调用。假设现在失去了线索，因此只想返回第47行，以便再次使用step-into命令进入隐式函数。这时可以结合使用栈导航命令up和tbreak前进到第47行的下一条伪代码语句，具体方法如下：

```
47      show((x+"def") * 3);
(gdb) step (repeatedly until you reach line 7 again)
...
(gdb) step
7      STR(const char* a) {
```

```
(gdb) where
#0  STR (... "abcdefabcdefabcdef"...) at main.cpp:7
#1  in STR::operator* (...) at main.cpp:31
#2  in main (...) at main.cpp:47
(gdb) up
(gdb) up
(gdb) tbreak
Breakpoint 4 at 0x8048b1d: file main.cpp, line 47.
(gdb) cont
47      show((x+"def") * 3);
(gdb) step
show (z=..."abcdefabcdefabcdef"...) at main.cpp:41
41      cout <<z.num_objs() <<" : " <<z.c_str() <<endl;
```

当不带参数时，tbreak命令通知调试器在选中栈帧的下一个地址停止，这刚好是伪代码的第47.4行。

经验总结

- 要定位隐藏的函数调用，在编译所有代码时应带有调试信息，并反复使用step-into命令。
- 要跳过对隐式函数所有源代码行的走查，反复执行step-out、step-into命令。
- 要跳过隐式函数，可以在目标函数中设置临时断点，然后继续。
- 如果在隐式函数中失去线索，可以在调用栈中向上导航，直到返回调用语句，然后设置一个临时断点。

10.4 条件断点和断点命令

我们经常会遇到这种情况：在到达要调试的位置之前，程序在很多断点处停下来。问题是如何才能使程序在我们最感兴趣的地方停止，而自动跳过其他断点。答案是使用条件断点和断点命令，这就是本节要讨论的主题。

首先以10.3节的STR类为例，解释在GDB中如何使用条件断点。该类有一个用于字符串字面值的STR构造函数。字符串字面值是源代码中const char*类型的常量，例如第45行中的“abc”。我们使用断点命令来找到构造函数被调用的时间和位置。每个构造函数调用都输出一行，然后程序自动继续执行。接着添加一个条件断点，使程序在用“def”参数调用构造函数的地方暂停。

首先用GDB调试器执行此操作。使用commands命令指定一个命令序列，每次到达一个特定断点时，都将执行此序列。这个命令序列的第一条命令是silent，它的作用是不输出后续命令的执行结果，以免屏幕输出过多。我们希望只输出一行，并使用GDB的printf命令来创建整齐的格式化输出。continue命令恢复程序的执行。此命令序列以end命令终止：

```
(gdb) break main.cpp:7
Breakpoint 1 at 0x8048c3a: file main.cpp, line 7.
(gdb) commands 1
```

```
Type commands for when breakpoint 1 is hit, one per line.
End with a line saying just "end".
> silent
> printf "STR CTOR with a=0x%x \"%s\"\n", a, a
> continue
> end
(gdb)
```

commands的参数1是指断点个数为1。每个断点只能有一个命令序列。如果再次指定commands 1，它将覆盖前面的命令序列。

现在运行程序将输出所有STR构造函数的内存地址和字符串字面值，可以看到，有3次构造函数调用：

```
(gdb) run
...
STR CTOR with a=0x8048f3b "abc"
2: abc
STR CTOR with a=0x8048f3f "def"
STR CTOR with a=0x804b51c "abcdefabcdefabcdef"
3: abcdefabcdefabcdef
```

在Visual Studio中，创建断点的方法是在示例的第7行单击鼠标右键并选择**Breakpoint/Insert Breakpoint**（断点/插入断点）。然后在**Breakpoints**（断点）窗口中，在断点上右击鼠标，选择**When Hit...**（命中条件）对话框。选择**Print a message**（打印消息）。记住要勾选**continue execution**（继续执行）复选框。

接下来在GDB示例中，我们将使用GDB的condition命令创建一个条件断点，以便在用字符串字面值“def”调用的地方停止：

```
(gdb) break main.cpp:7
...
Breakpoint 2 at 0x8048c3a: file main.cpp, line 7.
(gdb) condition 2 a[0]=='d' && a[1]=='e' && a[2]=='f' &&a[3]==0
(gdb) run
...
Breakpoint 2, in STR (... a=0x8048f3f "def") at main.cpp:7
7      STR(const char* a) {
```

因为有了一个字符串字面值，所以选择正确的条件有些复杂。条件表达式a=="def"永远不为真，因为调试器将指针a的值与另一个字符串字面值"def"的地址进行比较。这两个字符串字面值都是"def"，但它们存储在不同的地址中。我们也无法在条件中调用函数strcmp()。一个折中办法是比较字符串的各个字符。

在Visual Studio中，右击第7行的断点并选择**Breakpoint/Condition**。在对话框中输入a[0]=='d' && a[1]=='e' && a[2]=='f' && a[3]==0。

在GDB中创建条件断点的另一种方法是使用一个常规断点，然后附加一个包含条件语句的命令：

```
(gdb) break 7
Breakpoint 2 at 0x8048c3a: file main.cpp, line 7.
(gdb) commands 2
Type the commands for when breakpoint 2 is hit, one per line.
End with a line saying "end".
> silent
> print strcmp(a,"def")
> if $$0==0
>   print "FOUND the right BP!"
> else
>   printf "STR CTOR with a=0x%x \"%s\"\n", a, a
>   continue
> end
> end
(gdb)
```

表达式\$\$0引用前一行返回的结果，在本例中就是print strcmp(a, "def")命令。

注意，条件断点或断点命令可能会减慢程序的执行速度。每当到达断点时，GDB都需要占用CPU时间，而不管条件是否满足。

经验总结

- 使用断点命令，可以在无需重新编译程序的情况下添加输出语句。
- 条件断点可以捕捉特定函数调用，而忽略所有其他函数调用。
- 断点可能极大地减慢程序的执行速度。

10.5 调试静态构造/析构函数

本节讨论如何调试静态构造函数和析构函数的问题。在C++中，如果程序中含有静态类成员或全局实例化的类对象，那么程序在启动时会调用该对象的构造函数。此外，在C++中可以使用一种称为静态初始化程序（static initializer）的函数或方法来初始化全局对象和静态对象。下面是一个示例：

```
 MyClass otherGlobal;
int MyClass::veryImportantStatic = somefunction();
```

注意这些函数和构造函数是在何时被调用的。它们是在main()函数被调用之前的静态初始化期间被调用的。类似地，静态或全局类对象的析构函数是在main()函数返回之后的程序关闭期间被调用的。

如果静态构造函数是共享库的一部分，则它们在运行时载入共享库的时候被调用。这可能发生在程序启动时，如果共享库是通过程序代码显式载入的，也可能发生在启动之后。析构函数则是在程序退出或共享库被显式地关闭时调用的。

静态初始化程序可能引起非常严重的bug，因此带来了特殊的调试难题。最常见的bug来源是

静态初始化代码，这些代码对初始化的顺序有特定要求。由于这些代码不是从main()函数调用的，因此栈跟踪看起来有所不同。最后，当把调试器连接到正在运行的程序时，情况将更加复杂，因为当连接调试器时，初始化可能已经完成。

10.5.1 由静态初始化程序的顺序依赖性引起的 bug

在下面的示例中，有两个带有静态初始化程序的变量：全局变量otherGlobal和静态变量veryImportantStatic：

```

1  /* MyClass.h */
2  class MyClass {
3  public:
4      MyClass::MyClass() {
5          int i;
6          for(i=0;i<10;i++)
7              a[i]= veryImportantStatic+i;
8      int a[10];
9      static int veryImportantStatic;
10 }
11
12 /* MyClass.cc */
13 #include "MyClass.h"
14
15 int somefunction()
16 { return 42; }
17
18 int MyClass::veryImportantStatic = somefunction();
19
20 /* static_conflict.cc */
21 #include <stdio.h>
22 #include "MyClass.h"
23
24 MyClass otherGlobal;
25
26 int main() {
27     printf("otherGlobal.a[3]=%d\n", otherGlobal.a[3]);
28     return 0;
29 }
```

这两个变量都是在程序启动期间通过函数调用初始化的。一个隐含的要求是veryImportantStatic必须在otherGlobal之前被初始化。但是，程序忽略了这项要求，因此初始化顺序有可能发生错误。

初始化顺序除了依赖编译器之外，还依赖于链接器和链接顺序。链接器在安排静态初始化程序代码时，并没有固定的标准，而是会发生改变。在上例中，GCC 3.4.4首先初始化veryImportantStatic，然后计算结果。在Visual Studio 2008中，计算结果则取决于源文件在项目中的列出顺序。记住，程序也有可能会产生循环依赖，这样就永远无法工作。

10.5.2 识别静态初始化程序的栈跟踪

构造函数或析构函数的顺序问题可能非常模糊。因此它们不易识别。在搜索这些bug时，最有用的信息就是栈跟踪：

```
> g++ -g -o static_conflict static_conflict.cc MyClass.cc
> ./static_conflict
otherGlobal.a[3]=45
> gdb ./static_conflict
...
(gdb) break MyClass.cc:5
Breakpoint 1 at 0x4010e0: file MyClass.cc, line 5.
(gdb) run
(gdb) where
#0  somefunction () at MyClass.cc:5
#1  0x... in __static_initialization_and_destruction_0 (
    __initialize_p=1, __priority=65535) at MyClass.cc:7
#2  0x... in global constructors keyed to _Z12somefunctionv ()
    at MyClass.cc:8
#3  0x... in do_global_ctors () from /usr/bin/cygwin1.dll
#4  0x... in __check_for_executable () from /usr/bin/cygwin1.dll
```

注意，栈跟踪中并没有main()函数，因为它尚未被调用。somefunction()的调用是通过编译器和操作系统所提供的函数初始化的。根据编译器和OS的不同，这些函数的确切名称也不相同，但都类似于上面的形式。从现在开始，就可以在构造函数和初始化例程中设置断点并进行常规调试了。

10.5.3 在静态初始化之前连接调试器

在某些情况下，可能必须将调试器连接到正在运行的进程，然后对初始化例程进行调试。由于静态初始化例程是在程序启动阶段被调用的，所以在连接调试器时，它们已经执行。因此在初始化阶段应减慢程序的执行速度。添加另一个静态初始化调用，延迟程序的执行，这样就有足够时间连接调试器。以下代码片段提供了一个示例：

```
File initial_delay.cpp
1 #include <unistd.h>
2
3 static int delay_done=0;
4 static int ask_mice() {
5     while(!delay_done)
6         sleep(10);
7     return 42;
8 }
9 static int pol = ask_mice();
```

连接正在运行的进程，在调试器中设置所有需要的断点。然后更改delay_done变量，使程序继续执行。

```
> g++ -o test -g main.cpp init_delay.cpp
> ./test &
[1] 20189

> gdb test 20189
(gdb) ...
(gdb) set var delay_done=1
(gdb) continue
```

注意，这种方法还依赖于静态初始化代码的特定执行顺序，即`ask_mice()`首先被调用。链接器影响顺序，因此可能需要做几次实验，以便将这段代码放到首先被链接器初始化的程序中。

经验总结

- C++使用函数来初始化全局对象。这些函数是在`main()`之前被调用的，而且没有预定义的执行顺序。使用提供的示例来了解如何调试这些初始化函数。
- 使用提供的代码段来连接正在运行的进程，然后调试静态初始化函数。

10.6 使用观察点

观察点或数据断点的作用是在表达式值发生变化时停止程序的执行。在GDB中，`watch`命令的表达式可以是变量、内存地址或任意的复杂表达式。调试器将不间断地监视表达式，并在表达式值发生更改的语句停止程序的执行。

在Visual Studio中，使用**Debug/New Breakpoint/New Data Breakpoint**（调试/新建断点/新建数据断点）对话框来设置断点。在Visual Studio中无法为局部变量设置观察点，而只能观察地址。

在下面的示例中，将再次使用10.3节的STR类程序。目的是找到变量`x.num`发生更改的所有地方。每次`x.num`更改时，调试器都应该停止，并输出哪条语句更改了它。这里将使用GDB。

```
(gdb) start
(gdb) watch x.num
Hardware watchpoint 2: x.num
(gdb) cont
...
9 num++
(gdb) where
#0 0x08048c62 in STR::STR (...) at break_str.cc:9
#1 0x08048a92 in main (...) at break_str.cc:45
```

10

可以看到，当观察点表达式的值更改时，程序暂停。

如果可能的话，GDB调试器将用硬件辅助（hardware assistance）来实现观察点。如果观察点没有硬件辅助，它可能严重影响程序的速度，而硬件辅助的断点几乎不会减慢执行速度。

注意，如果在局部变量中设置了观察点，则当程序离开这个局部范围时，GDB将删除这些观

察点。例如，如果在一个构造函数中停止，并在num上设置观察点，则一旦离开此构造函数，该观察点将丢失。

在GDB和Visual Studio中，都可以在需要观察的变量的地址上设置观察点。这个表达式适用于整个程序，因此观察点不会被删除：

```
(gdb) start
... run program till line 13
13         num++;
(gdb) p &num
$1 = (int *) 0x804a360
(gdb) watch *0x804a360
Hardware watchpoint 2: *134521696
```

不要忘记在地址前面加*。如果不小心键入了watch 0x804a360，而不是watch *0x804a360，那么观察点将检查一个十六进制的常量地址值，它永远不会发生变化。

对程序栈上的对象（如局部变量）进行观察是没有意义的。一旦退出了当前函数，栈就会用于存储其他变量。另外也不会保证下次调用函数时本地变量还会被分配相同的栈地址。

经验总结

- 利用观察点可以有效地找到哪些语句对特定变量做了修改。
- 未使用硬件辅助的观察点对程序性能有很大影响。
- 建议不要对局部变量使用观察点。

10.7 捕捉信号

本节讨论如何对信号进行调试。进程之间通过信号通信。信号从一个进程被发送到另一个进程，或者在一个进程内部传递。信号是异步的。不同的操作系统具有不同的可用信号集。Windows上的可用信号较少。在UNIX或Linux上，可以使用命令kill -l来列出所有可用信号。信号的一个众所周知的应用是Ctrl-C组合键，它拦截正在运行的程序。在命令行解释器中按Ctrl-C键会把SIGINT信号发送给前台正在运行的程序。

接到信号的进程必须做出响应。进程可能有一个称为信号处理程序的C函数，此函数已经被某种特定信号类型注册。每当这种类型的信号到达时，如果没有被临时阻止，则将调用信号处理程序，而不管当前正在执行哪个进程。在用户未定义处理程序时，操作系统将采取预定义的操作，具体操作取决于信号，要么中止程序，要么忽略信号，要么挂起程序。

与信号处理有关的问题很难调试，因为我们无法预测这些事件何时发生。系统调用可能被打断，而且可能返回无法预测的错误代码。此外，我们可能正在构建一个数据结构，因此信号处理程序对此数据结构的读写可能产生无法预料的结果。

大多数调试器允许用户指定如何处理每个接收的信号。在GDB中，这是通过handle命令实现的。可以选择将消息输出到屏幕上，或者停止程序的执行。也可以通知调试器忽略信息，这样就由注册的中断处理来处理它。还有一个命令可以通知调试器生成一个信号，并将它传送给程序。在GDB中，这个命令就是signal。

在调试信号问题时，可以采用以下策略。

- 提高可见性。令调试器在收到信号时输出一条消息（例如，SIGUSR1），然后将此消息发送给程序。信号将在屏幕上留下明确的线索，这样更容易理解正在发生什么事情。示例：handle SIGUSR1 print nostop pass。
- 禁用干扰性信号。信号处理程序可能对程序产生影响。可以令调试器忽略所有进入的信号，这样就不再调用信号处理程序。示例：handle SIGUSR1 noprint nostop nopass。
- 引发bug。生成一个信号，并将其发送给程序，以测试它是否触发bug。示例：signal SIGUSR1。

以下示例是一个简单的信号处理程序，它在程序处于内部延时循环繁忙期间计算信号SIGUSR1的接收频率。注意，此程序在Windows上无法编译，因为Windows平台不支持SIGUSR1信号。

```

1 #include <signal.h>
2 #include <stdio.h>
3
4 static int num_sigusr1=0;
5
6 void handler(int sig) {
7     num_sigusr1++;
8     signal(SIGUSR1,&handler);
9 }
10
11 int main(int argc, char** argv) {
12     int n,m, pol=0;
13     signal(SIGUSR1,&handler);
14     printf("- program starts\n"); fflush(stdout);
15     for (n=0; n<10; n++) {
16         raise(SIGUSR1);
17         for (m=0; m<10000000000; m++)
18             pol++;
19     }
20     printf("- program ends: received SIGUSR1 %d time(s)\n",
21           num_sigusr1);
22     fflush(stdout);
23     return 0;
24 }
```

程序将向它自己发送10个SIGUSR1类型的信号。在程序运行期间，使用UNIX命令kill

-USR1 <pid>从另一个命令行解释器发送更多信号。在GDB中运行程序，并使用GDB命令handle来配置GDB对信号的响应方式。首先，将信号输出到屏幕：

```
(gdb) handle SIGUSR1 print nostop pass
Signal  Stop Print Pass to program Description
SIGUSR1  No    Yes   Yes           User defined signal 1
(gdb) run
```

一旦处理程序接收到信号，GDB输出一条消息：

```
Program received signal SIGUSR1, User defined signal 1.
Program received signal SIGUSR1, User defined signal 1.
Program received signal SIGUSR1, User defined signal 1.
...
program ends: received SIGUSR1 10 time(s)
```

接下来，禁用SIGUSR1，这样它不再到达程序：

```
(gdb) handle SIGUSR1 noprint nostop nopass
Signal  Stop Print Pass to program Description
SIGUSR1  No    No    No           User defined signal 1
(gdb) run
program ends: received SIGUSR1 0 time(s)
```

最后，使用GDB的signal命令来生成一个信号，以便测试程序如何响应：

```
(gdb) break handler
Breakpoint 1 at 0x894842f: file main.c, line 7.
(gdb) run
... Interrupt program with Ctrl-C
(gdb) signal SIGUSR1
Continuing with signal SIGUSR1

Breakpoint 1, handler (sig=10) at main.c:7
7      num_sigusr1++;
...
program ends: received SIGUSR1 11 time(s)
```

可以看到，处理程序在第7行接收到信号，并计算收到的次数。

经验总结

- 调试器能够处理信号，例如SIGUSR1、SIGALRM、SIGSEGV、SIGINT。
- 可以采用多种信号处理方式，包括输出信号、过滤掉信号或是用调试器来生成信号。

10.8 捕获异常

C++提供了一种称为异常的特性，它既可用于错误处理，也可用于异常情况的处理。异常在同一个进程中被抛出（生成）和捕获（使用）。当使用GDB的catch throw命令时，调试器在抛出任何异常时停止。当使用catch catch命令时，调试器在捕获任何异常时停止。考虑

以下示例：

```

1 #include <iostream>
2
3 void f1() {
4     throw 42;
5 }
6 void f2() {
7     throw "pol";
8 }
9
10 int main(int,char**) {
11     try {
12         f1();
13     } catch (int E) {
14         std::cout << "caught E=" << E << std::endl;
15     }
16     f2();
17     return 0;
18 }

(gdb) start
...
(gdb) catch throw
Catchpoint 2 (throw)
(gdb) catch catch
Catchpoint 3 (catch)
(gdb) cont

Catchpoint 2 (exception thrown)
0x400clea5 in __cxa_throw () from /usr/lib/libstdc++.so.5
(gdb) where
#0 0x400clea5 in __cxa_throw () from /usr/lib/libstdc++.so.5
#1 0x08048955 in f1 () at main.cpp:4
#2 0x800489a2 in main () at main.cpp:12

```

注意，只有在载入了C++运行库（DLL）之后，才能使用GDB的catch和throw命令。因此，一定要在运行程序之后使用这些命令。一种简便的方法是用start命令使程序在main()函数的第一行停止。

如果调试器不可用，一种替代方法是修改程序，加入一条catch语句。当异常被抛出时，程序继续执行，直到到达一条包含匹配类型的catch语句。注意，catch(...)将捕获任何类型的异常。这告诉我们程序中发生异常，如果异常类型是已知的，还会报告它的类型。但是，这种方法不会显示抛出异常的位置，因此使用调试器还是首选。

在Visual Studio中，打开**Debug**（调试）菜单，选择**Exceptions...**（异常）。此时将弹出一个对话框，可以选择在抛出特定异常时中断程序的执行。选择**C++ Exceptions**可以为代码中的C++ throw语句使用程序断点。

经验总结

- 可以使用GDB的catch命令来调试C++异常。
- 在Visual Studio中，通过Debug（调试）菜单的Exceptions...（异常）窗口来使用断点（在抛出异常时）。

10.9 读取栈跟踪

本节主要讨论读取栈跟踪的技巧。第3章和第4章已经介绍了这些概念，并演示了如何用调试器在栈帧中导航。栈跟踪是一个帧列表，其中每个帧对应一个被调用的函数。函数名称可能是可读的，而且帧可能还包含与函数参数有关的更多信息。

10.9.1 带调试信息编译的源代码的栈跟踪

这里仍以10.3节的STR类为例。修改第22行中的operator+代码，使其引用一个空指针，从而引起一个分段错误：

```
file main.cpp
21 const STR& operator+ (const STR& a) {
22     int* dummy=0;
23     (*dummy)++;
24     s += a.s; return *this;
25 }
```

编译带调试信息的程序，然后运行它，直到崩溃：

```
...
Program received signal SIGSEGV, Segmentation fault.
0x08048d24 in STR::operator+ (...) at break_str.cc:23
23         (*dummy)++;
...
(gdb) where
#0  0x.. in STR::operator+ (...) at break_str.cc:23
#1  0x.. in main (argc=1, argv=0xbffffe44) at break_str.cc:51
```

信号SIGSEGV是一个分段错误，当程序试图访问没有权限的内存地址时，就会发生此错误。在本例中，我们访问了地址0x0，这是分段错误的一个常见原因。其他致命的信号还有SIGBUS（非法总线地址）、SIGILL（非法指令）和SIGFPE（浮点异常）。

栈跟踪有两个帧，每个帧都有源代码和行信息以及实际的参数列表。可以看到，在第23行函数main()正在调用函数STR::operator+。这是栈跟踪的一个典型示例。

10.9.2 不带调试信息编译的源代码的栈跟踪

栈跟踪不包含过多信息也是很常见的。如果编译不带调试信息的相同代码，则栈跟踪将像下面这样：

```
> g++ -o break_str break_str.cc
> gdb ./break_str
...
(gdb) where
#0  0x08048d24 in STR::operator+ ()
#1  0x08048b11 in main ()
```

注意，源代码和行信息没有了，这使得更难找到错误的源代码行。好消息是我们可以明显看出函数main()正在调用函数STR::operator+。接下来，应该搜索STR类定义的源代码，然后在其中搜索operator+，这就将我们引导到第23行附近。有关如何调试不带调试信息编译的源代码的更多内容，参见10.11节。

10.9.3 不带任何调试信息的帧

调试器有可能不会找到与某个特定帧有关的任何信息。删除所有调试信息，就出现了这种情况：

```
> g++ -o break_str -g break_str.cc
> strip break_str
> gdb break_str
...
Program received signal SIGSEGV, Segmentation fault.

0x08048d24 in ?? ()
(gdb) where
#0  0x08048d24 in ?? ()
#1  0x08048b11 in ?? ()
#2  0x4012ae80 in __libc_start_main () from /lib/tls/libc.so.6
#3  0x08048961 in ?? ()
```

这样的栈跟踪信息几乎没什么用处，因为除了__libc_start_main之外就没有函数名称了。所有其他帧报告的函数名称都是??，这表明调试器在给定地址上没有任何调试信息。

注意，栈帧的数量增加到了4个。这是正确的，下面的两个帧是两个C/C++标准库中的函数，它们调用main()函数。调试器一般不报告比main()还低的帧，因此前面未报告它们。

当调试器在特定地址上没有对象代码的相关信息时，这个帧就会报告??。在很多情况下可能发生这个问题，例如：

- 出于某种目的，程序被删除了，而且所有调试信息也被移除了；
- 调试信息在共享库中，而调试器尚未载入它；
- 调试器无法理解调试信息。当混用不同的编译器时，例如将Sun Forte和GCC on Solaris混用，经常发生这种情况；
- 存储栈帧的内存被破坏了。

10.9.4 实际工作中的栈跟踪

下面是实际应用程序中的一个栈跟踪示例。猜一下它是哪个应用程序（提示：从底部开始读），

现在正在做什么（提示：从顶部开始读）。

```
#0 0x.. in ?? ()
#1 0x.. in ?? ()
#2 0x.. in ?? ()
#3 0x.. in ?? ()
#4 0x.. in select () from /lib/tls/libc.so.6
#5 0x.. in XtAddTimeOut () from /usr/X11R6/lib/libXt.so.6
#6 0x.. in _XtWaitForSomething () from ..
#7 0x.. in XtAppProcessEvent () from /usr/X11R6/lib/libXt.so.6
#8 0x.. in emacs_Xt_handle_widget_losing_focus ()
#9 0x.. in event_stream_resignal_wakeup ()
#10 0x.. in Fnext_event ()
#11 0x.. in Fcommand_loop_1 ()
#12 0x.. in Fcommand_loop_1 ()
#13 0x.. in condition_case_1 ()
#14 0x.. in Frecursive_edit ()
#15 0x.. in internal_catch ()
#16 0x.. in initial_command_loop ()
#17 0x.. in xemacs_21_5_b18_i386_suse_linux ()
#18 0x.. in main ()
```

第17帧和第8帧显示了这是一个XEmacs编辑器。第4~6帧表明编辑器正在等待新的输入，例如按键操作。注意，第5~7帧属于X库。函数select()是标准C库的一部分，它可能正在调用操作系统的某个函数。第0~3帧可能属于OS，而且它们没有可用的调试信息。

10.9.5 改编后的函数名称

如果使用不同的编译器编译C++源代码，则某些帧的函数名称可能被报告为改编后的形式。调试器通常可以自动对这些名称进行反改编，但偶尔会产生混乱。有关如何读取改编后的C++函数名称的更多内容，参见9.3.4节。

10.9.6 被破坏的栈跟踪

当程序的内存被bug破坏时，它也可能改写调用栈。固定大小的局部数组经常发生这种情况。越界写数组将对栈造成破坏，而且程序可能很快遇到致命的bug。此外，调试器总是报告破坏的数据，而不是正确的调用栈信息。下面的示例就是这种情况：

```
1 int n;
2 int *p;
3 int F(int a) {
4     p = &a;
5     for (n=a; n>0; n--)
6         *p-- = 0x42;
7     return a;
8 }
9 int main(int argc, char* argv[])
```

```
10  {
11      return F(10);
12  }

> g++ -o broken_stack -g broken_stack.cc
> gdb broken_stack
...
(gdb) run
...
Program received signal SIGSEGV, Segmentation fault.
0x00000002a in ?? ()
(gdb) where
#0 0x00000042 in ?? ()
#1 0x00000042 in ?? ()
#2 0xbffffee4c in ?? ()
#3 0xbffffedb8 in ?? ()
#4 0x08048491 in __libc_csu_init ()
Previous frame inner to this frame (corrupt stack?)
```

重要的一点是GDB报告栈可能被破坏。注意第0~1帧的值为42，这更说明栈已被破坏。

经验总结

- 学习读取栈跟踪。
- 破坏的栈跟踪可能是内存破坏的一个症状，例如越界写一个局部数组。检查方法是使用内存调试器运行程序。

10.9.7 核心转储

如果程序遇到bug并触发了一个分段错误，则操作系统将创建一个核心转储文件（core dump file）。核心转储为事后调试提供了数据，并且可用来查找程序失败的原因。

以下程序将引发一个分段错误：

```
1  /* answer.c */
2  #include <stdio.h>
3
4  void runexperiment() {
5      int *answerp;
6      answerp = (void *) 42;
7      printf("The answer is %d\n", *answerp);
8  }
9
10 void createplanet() {
11     runexperiment();
12 }
13
14 int main() {
15     printf("Hello Universe! Computing answer ...\\n");
16     fflush(stdout);
```

```

17     createplanet();
18     return 0;
19 }

```

运行程序，第7行将发生分段错误：

```

> ./myprog
Hello Universe! Computing answer ...
Segmentation fault

```

注意，如果cshrc命令行解释器初始化文件中包含`limit coredumpsize 0`命令，则不会生成核心转储文件。检查命令行解释器的文档，了解如何启用核心转储。

我们使用`gdb myprog core`命令将核心转储和程序一并载入调试器。

```

> gdb myprog core
...
#0 0x080c1368 in runexperiment ()
#1 0x080bff8f in createplanet ()
#2 0x080bfc0f in main ()
(gdb)

```

核心转储包含程序已分配的所有数据的一个副本。栈数据也是其中的一部分，利用栈数据可以在程序终止后使用调试器来查看栈跟踪。

如果在Windows上运行程序，它将由于分段错误崩溃。这时将弹出一个窗口，询问是否使用选定的调试器进行调试。单击**Yes**（是）启动Visual Studio。在已崩溃的进程上可以执行两种操作，一是立即调试，二是生成一个核心转储文件，过后再调试。在Visual Studio中保存核心转储文件的方法是打开菜单项**Debug**（调试），选择**Save Dump As...**（将转储另存为），这将生成一个.dmp文件。过后，双击.dmp文件即可将它重新载入Visual Studio。

经验总结

- 使用核心转储文件可以进行事后调试。可以查看何处发生了分段错误，还可以查看完整的栈跟踪，其中列出了导致崩溃的函数，以及调用该函数的所有函数。
- 还可以上下移动栈帧，并查询变量和内存的值。

10.10 操纵正在运行的程序

调试器除了可用于跟踪程序的控制流和分析数据以外，还可以通过更改变量内容或从调试器内部调用函数来操纵程序的行为，这就是本节要重点讨论的内容。这些调试器功能可以帮助测试bug是否已修复，而免去了冗长的重新编译过程，此外，这些功能还可以对测试过程未涵盖的一些区域进行调试。

改变程序行为的标准方法是修改源代码，然后重新编译。我们可以利用调试器得到相同的结果。如果重新编译程序需要很长时间，或者很难重新建立当前的程序状态，就可以利用调试器作

为一种替代方法。需要记住的是，当调试器退出时，通过它执行的操纵也将丢失，而且通过调试器实现的操纵是有限的，但我们不妨了一下都能做什么。

调试器通常提供所有（或至少是一部分）以下命令，这些命令用于操纵程序的行为。

- 操纵数据

- 修改变量值或函数的实际参数。在GDB中，使用`set var <varname>=<expr>`命令，例如`set var MyVar=17`。在Visual Studio中，可以在**Variables**（变量）窗口中编辑值。
- 修改函数的返回参数值。在GDB中，跳到函数的最后一条指令，然后使用`return <expr>`命令。
- 修改堆内存的内容，如10.10.6节所述。

- 修改环境变量。在GDB中，使用`set environment <var> <value>`命令。在Visual Studio中，单击菜单项**Project**（项目），然后单击**Properties / Configuration Properties / Debugging / Environment**（属性/配置属性/调试/环境），这样就可重写环境变量了。修改后的环境变量在下次启动程序时生效。

- 操纵流的控制

- 调用函数。在GDB中，使用`call`或`print`命令。在Visual Studio中，使用**Immediate**（即时）窗口。
- 从函数退出，跳过剩下的语句。在GDB中，使用`return [<expr>]`命令。在Visual Studio中没有等效的命令。
- 跳过或重新执行当前函数中的语句。在GDB中，使用`jump`命令。在Visual Studio中没有等效的命令。

通过将这些命令添加到断点中可以进行“半永久型”的修改。只要不退出调试器，这些修改就是有效的。参见10.4节的条件断点了解具体方法。

现在通过以下示例解释调试器的操纵。示例中的程序搜索输入字符串中的名字，然后将它们转换为大写字母，并计算更改的字符串个数：

```

1  /* capitalize.c */
2  #include <stdio.h>
3  #include <ctype.h>
4  #include <string.h>
5  #include <stdlib.h>
6  #ifdef _MSC_VER
7  #define strncasecmp strnicmp
8  #endif
9
10 void change_word ( char* str, int len ) {
11     int i;
12     for ( i=0; i<len; i++ )
13         str[i]=toupper(str[i]);

```

```

14 }
15
16 int capitalize_str( char* str, const char* name ) {
17     int n;
18     int hits=0;
19     int len=strlen(str);
20     int len_name=strlen(name);
21     int lastpos = len-len_name;
22     for (n=0; n<=lastpos; n++) {
23         if (strncasecmp(str+n,name,len_name)==0) {
24             change_word( str+n, len_name );
25             n += len_name;
26             hits++;
27         }
28     }
29     return hits;
30 }
31
32 int main( int argc, char* argv[] ) {
33     int hits_total=0;
34     int na;
35     char *mycopy;
36     if (argc<3) return 1;
37     mycopy = strdup(argv[1]);
38     for(na=2; na<argc; na++)
39         hits_total += capitalize_str( mycopy, argv[na] );
40     printf("Total %d hits:\n", hits_total);
41     printf("original: %s\n", argv[1]);
42     printf("modified: %s\n", mycopy);
43     free(mycopy);
44     return 0;
45 }

```

运行程序，显示出一个bug：

```

> myprog "Foofoo, foobar and Bar!" foo bar
Total 4 hits:
original: Foofoo, foobar and Bar!
modified: FOOfoo, FOOBAR and BAR!

```

程序没有将“Foofoo”中的第二个“foo”转换为大写。原因是第25行有个bug，也就是在计算第22行中的循环增量n++时，n的值增大len_name，而不是len_name-1。

10.10.1 修改变量

对这个小程序来说，修改源代码然后重新编译是很快的。但在大型程序中这可能要花费很长时间。在开始缓慢的重新编译过程之前，有必要测试一下bug是否已正确修复。在示例中，测试方法是通过调试器来减小变量n的值。程序显示出正确结果，这样就可以确定bug已修复。

```

(gdb) run "Foofoo, foobar and Bar!" foo bar
... run program until it stops in line 23 for the 2nd time
23         if (strncasecmp(str+n,name,len_name)==0) {

```

```
(gdb) print n
$1 = 4
(gdb) print str+n
$2 = 0x804a00c "oo, foobar and Bar!"
(gdb) set var n=3
(gdb) print str+n
$3 = 0x804a00b "Foo, foobar and Bar!"
(gdb) continue
...
Total 5 hits:
original: Foofoo, foobar and Bar!
modified: FOOFOO, FOOBAR and BAR!
```

所有调试器都提供了某种用来修改变量的方式，因为这是一项重要且常用的特性。在Visual Studio中，在**Immediate**（即时）窗口中输入n=3，或者在**Variables**（变量）窗口修改值。

10.10.2 调用函数

调试器提供的另一个重要特性是通过调用C或C++函数来操纵程序。在GDB中，可以使用call或print命令。在Visual Studio中，使用**Immediate**（即时）窗口。下面通过多调用一次capitalizestr函数将字符串and转换为大写来演示此特性：

```
... navigate the program to line 39
39          hits_total += capitalize_str( copy, argv[na] );
(gdb) call capitalize_str( copy, "and" )
$3 = 1
(gdb) cont
Continuing.
Total 4 hits:
original: Foofoo, foobar and Bar!
modified: FOOfoo, FOOBAR AND BAR!
```

注意，AND现在是大写的了。使用call或print命令可以调用任何C/C++函数。在调用C函数时，必须保证参数个数和类型都正确，因为GDB不能检查这一点，至少不能检查C函数的参数个数和类型是否正确。如果调用导致了致命错误，那么GDB有两种不同的响应模式，具体取决于unwindonsignal的设置，如下所述。

□ **set unwindonsignal off**: 当程序崩溃时，GDB停止。当需要测试函数以了解崩溃原因时，这很有用。栈帧将显示调用是从何处开始的（从名为<function called from gdb>的帧开始）。如果想继续执行程序，可以使用前面讲过的return命令从停止处返回。

□ **set unwindonsignal on**: GDB将打印它收到了一个信号，然后自动返回。

在某种程度上，也可以调用C++类的成员函数。例如，可以调用像mystr.c str()这样的简单成员函数，它们不带参数，或只带有ANSI C类型的参数。那些以类对象作为参数的成员函数则很难调用，而类构造函数或操作符则几乎无法调用。

10.10.3 修改函数的返回值

接下来，在第30行函数刚要返回时，使用return命令重写函数capitalizestr的返回值：

```
... run program till line 30
30      }
(gdb) return 40
Make capitalize_str(char*, char const*) return now? (y or n) y

#0 0x080486cd in main (argc=4, argv=0xbffffe44) at main.c:39
39          hits_total += capitalize_str( copy, argv[na] );
(gdb) cont
Continuing.
Total 42 hits:
original: Foofoo, foobar and Bar!
modified: FOOfoo, FOOBAR and BAR!
```

正常的返回值是4，而程序现在输出42。这说明操纵返回值是有效的。

10.10.4 中止函数调用

使用GDB的return命令可以进一步操纵程序的行为。它可以返回选中栈帧之外的帧，从而跳过所有内部的栈帧。当程序执行到change_word的中间时，可以使用此特性从change_word和capitalizestr直接返回到main：

```
... run the program until it reaches line 12 for the 2nd time
12      for (i=0; i<len; i++)

(gdb) up
#1 0x08048639 in capitalize_str (... ) at main.c:24
24          change_word( str+n, len_name );

(gdb) return 0
Make capitalize_str(char*, char const*) return now? (y or n) y

#0 0x080486cd in main (...) at main.c:39
39          hits_total += capitalize_str( copy, argv[na] );

(gdb) cont
Continuing.
Total 2 hits:
original: Foofoo, foobar and Bar!
modified: FOOfoo, fooBAR and BAR!
```

return命令将清除栈帧，因此程序很有可能会继续运行。记住，当使用return命令从帧中退出时，程序将不再执行这些帧中的任何语句。这可能产生内存泄漏或破坏类对象，因为可能跳过了隐式的析构函数调用。

10.10.5 跳过或重复执行个别语句

使用GDB的jump命令可以从当前的栈帧跳到任意一行。可以向前跳跃，这样将跳过某些语句，也可以向后跳跃，从而重复执行某些语句。我们将使用此命令跳过第24行，从而避免修改word(...)调用：

```
... navigate program to line 24
24      change_word( str+n, len_name );
(gdb) jump 25
Continuing at 0x804863c.
Total 4 hits:
original: Foofoo, foobar and Bar!
modified: Foofoo, FOOBAR and BAR!
```

注意，修改后的字符串现在以“Foofoo...”开头，而不是“FOOfoo...”，因此jump命令有效地跳过了第24行的函数调用change_word(...)

10.10.6 输出和修改内存内容

本节介绍如何输出和修改内存内容。所有调试器都提供了某种用于检查（有时是修改）内存内容的特性。GDB中的相关命令是what is、print、x和set var。在Visual Studio中，可以使用显示变量和内存的窗口，或Immediate（即时）窗口。内存位置可以用0xbff7a5b4这样的形式表示，也可以通过指向堆内存的变量表示，例如argv[1]。

这里将再次以10.3节的STR类为例来演示如何使用调试器的这些特性来操纵程序的行为。我们的目标是读取并修改main()函数的argc和argv变量。变量argc是一个整数，因此很容易修改。但argv是在内存中分配的一个char*字符串数组，所以更难修改。

第一步是使用what is命令确定argv的类型，再用print命令确定它的当前值：

```
(gdb) start "foofoo, FooBar and Bar!" foo bar
...
(gdb) whatis argv
type = char **
(gdb) print argv
$1 = (char **) 0xbffffee54
```

变量argv是一个字符串数组，因此可以使用表达式argv[<n>]来访问它的个别数组元素：

```
(gdb) whatis argv[0]
type = char *
(gdb) print argv[0]
$2 = 0xbffff05e "/home/someone/myprog"
(gdb) print argv[1]
$3 = 0xbffff08a "foofoo, FooBar and Bar!"
```

10

@操作符后面紧跟参数5可以通知GDB调试器argv是一个数组，并通知GDB显示前5个元素。注意，数组以一个空指针终止，因此实际上有argc+1个元素：

```
(gdb) print argc
$4 = 4
(gdb) print *argv@5
$20 = {0xbffff05e "/home/someone/myprog",
        0xbffff08a "foofoo, FooBar and Bar!",
        0xbffff0a2 "foo",
        0xbffff0a6 "bar",
        0x0}
```

也可以用x命令直接访问内存，此命令只需要地址和一个可选的格式：

```
(gdb) x/5xw 0xbffffee54
0xbffffee54: 0xbffff05e 0xbffff08a 0xbffff0a2 0xbffff0a6
0xbffffee64: 0x00000000
```

参见GDB文档或键入help x命令查看格式的描述。x/5xw命令将输出5个大小为4字节的单词。也可以用x/40c命令输出字符串argv[1]，此命令输出40字节的字符：

```
(gdb) p argv[1]
$38 = 0xbffff08a "foofoo, FooBar and Bar!"
(gdb) x/40c 0xbffff08a

0xbffff08a: 102 'f' 111 'o' 111 'o' 102 'f' 111 'o' 111 'o' 44 ',' 32 '
0xbffff092: 70 'F' 111 'o' 111 'o' 66 'B' 97 'a' 114 'r' 32 ',' 97 'a'
0xbffff09a: 110 'n' 100 'd' 32 ',' 66 'B' 97 'a' 114 'r' 33 '!' 0 '\0'
0xbffff0a2: 102 'f' 111 'o' 111 'o' 0 '\0' 98 'b' 97 'a' 114 'r' 0 '\0'
0xbffff0aa: 76 'L' 69 'E' 83 'S' 83 'S' 75 'K' 69 'E' 89 'Y' 61 '='
```

现在通过改变“Foo”和“Bar”的顺序来为argc和argv变量创建新值，并且在“Foo”和“Bar”之间添加一个新的参数。先前为argv分配的内存无法容纳argv[5]，因此必须调用malloc重新为argv分配内存，使其可以容纳6个4字节单词。

```
(gdb) set var argc=5
(gdb) print/x malloc(6*4)
$23 = 0x804a048
(gdb) set var argv=0x804a048
```

接下来设置argv中的所有指针。复制原来的argv[0]和argv[1]指针，并为其他参数使用字符串字面值。注意，GDB在堆上分配表达式中使用的字符串字面值，而且不释放它们，因此不必担心内存突然被释放。

```
(gdb) set var argv[0]=0xbffff05e
(gdb) set var argv[1]=0xbffff08a
(gdb) set var argv[2]="BAR"
(gdb) set var argv[3]="and"
(gdb) set var argv[4]="Foo"
(gdb) set var argv[5]=0
(gdb) print *argv@6
$24 = {0xbffff05e "/home/someone/myprog",
        0xbffff08a "foofoo, FooBar and Bar!",
        0x804a068 "BAR",
        0x804a078 "and",
        0x804a088 "Foo",
        0x0}
```

最后一步是将字符串“`foofoo, FooBar and Bar!`”修改为“`ABCDEF, FooBar and Bar!`”。前两个字符通过访问`argv[1][0]`和`argv[1][1]`来重写，这两个元素均指向单个字符。这种方法虽然可行，但过于烦琐。因此接下来调用`strncpy`修改剩余的4个字符：

```
(gdb) set var argv[1][0] ='A'
(gdb) set var argv[1][1] ='B'
(gdb) call strncpy(argv[1]+2, "CDEF", 4)
$25 = -1073745779
(gdb) print *argv@6
$27 = {0xbfffff05e "/home/someone/myprog",
        0xbfffff08a "ABCDEF, FooBar and Bar!",
        0x804a068 "BAR",
        0x804a078 "and",
        0x804a088 "Foo",
        0x0}
```

现在完成了内存修改，运行程序，查看结果是否正确：

```
(gdb) cont
Continuing.
Total 4 hits:
original: ABCDEF, FooBar and Bar!
modified: ABCDEF, FOOBAR AND BAR!
```

从上面的讨论可以知道，内存的修改并不限于少数几个变量。付出足够的努力和细心，就可以修改全部内存内容。当然，也有可能会意外地破坏内存。

经验总结

- 调试器可以通过多种方法改变正在运行的程序，包括修改变量、调用函数、重写函数的返回值、中止函数调用、跳过或反复执行某些语句，以及直接修改内存内容。
- 将上述特性与断点命令结合起来使用，就得到功能强大的调试技术。

10.11 在没有调试信息时进行调试

有些程序在编译部分或所有源代码时不带有调试信息，本节就讨论如何调试这样的程序。当用GDB或其他调试器调试程序时，通常会编译带有调试信息的程序，例如使用`gcc -g`选项。但是，有时候程序的调试版本不可用，而只有优化版本（即非调试版本）可用。当在客户现场进行调试时，有时来不及创建一个调试版本，这个例子就适用于这样的场景。

假设我们必须用手头上现有的工具调试这样一个程序。虽说这是一种无奈的情况，但并非毫无希望。本节将给出一些提示来说明如何最大限度地调试这种程序，以及如何使用调试器提取尽可能多的信息。这里将使用以下示例来输出函数的参数值、在源代码中查找语句的大概位置，以及对源代码进行走查。

```

1 #ifndef NODEBUG_H
2 #define NODEBUG_H
3 /* nodebug.h */
4 int size(const char* S);
5 int F (int A, const char* B, char* C);
6 #endif

1 /* nodebug.c */
2 #include "nodebug.h"
3
4 int F (int A, const char* B, char* C) {
5     int R = size(B+A);
6     R *= size(C+A);
7     return R;
8 }

1 /* main.c */
2 #include "nodebug.h"
3 #include <string.h>
4
5 int size(const char* S) {
6     return strlen(S);
7 }
8
9 int main(int argc, char* argv[]) {
10     return F (2,           /* A */
11                "AABBCCDD",    /* B */
12                argv[1] );      /* C */
13 }

```

在此示例中，编译main()和size()时带有调试信息，而编译F()时不带调试信息。选择命令行参数，使程序崩溃：

```

gcc -O -c nodebug.c
gcc -g -c main.c
gcc -o test main.o nodebug.o
./test
Segmentation Fault

```

调试器显示崩溃发生在函数strlen()中：

```

gdb test
(gdb) run
Program received signal SIGSEGV, Segmentation fault.
0x400938db in strlen () from /lib/tls/libc.so.6
(gdb) where
#0 0x400938db in strlen ()
   from /lib/tls/libc.so.6
#1 0x080483cd in size (S=0x2 <Address 0x2 out of bounds>)
   at main.c:6
#2 0x0804842f in F ()
#3 0x08048405 in main (argc=1, argv=0xbffffeeb4),
   at main.c:10

```

函数F()通过非法参数s=0x2调用了函数size(), 而函数F()又被main()调用。首先用调试器来检查main()是否正确工作, 因为main()是带调试信息编译的, 因此易于调试。由于strlen()是标准C库中的函数, 已经过很好的测试, 因此一般不会有bug。这使得F()成为唯一的bug来源。这就需要尽可能多地收集有关F()内部工作的信息。

10.11.1 从栈读取函数参数

首先需要检查F()函数的参数。调试器无法直接显示它们, 因为F()是不带调试信息编译的。但调试器仍然能够提供足够信息, 将这些信息与源代码信息综合到一起, 可以得到相同结果。信息的连接方式取决于CPU类型、操作系统和编译器。本例是在Pentium CPU上编译的, 使用了Suse9 Linux操作系统和GCC 3.3.5编译器。在这种情况下, 所有函数参数都存储在栈上。首先, 使用GDB的info frame命令定位栈所在的内存地址:

```
(gdb) frame 2
#2 0x0804842f in F ()
(gdb) info frame
Stack level 2, frame at 0xbffffee00:
  eip = 0x804842f in F; saved eip 0x8048405
  called by frame at 0xbffffe30, caller of frame at 0xbffffede0

  Arglist at 0xbfffffedf8, args:      <== MOST INTERESTING PART
  Locals at 0xbfffffedf8, Previous frame's sp is 0xbffffee00

  Saved registers:
    ebx at 0xbfffffedf0, ebp at 0xbfffffedf8, esi at 0xbfffffedf4,
    eip at 0xbfffffedfc
```

帧从地址0xbffffee00开始。F()的三个参数都在这个帧上, 而且它们的地址是连续的。可以使用GDB的print sizeof(int)和print sizeof(char*)命令查看参数需要多大空间。每个参数需要4字节, 因此通过访问从0xbffffee00开始的三个单词就可以找到所有数据:

```
(gdb) p sizeof (int)
$1 = 4
(gdb) p sizeof (char*)
$2 = 4
(gdb) x/3x 0xbffffee00
0xbffffee00: 0x00000002 0x08048558 0x00000000
```

第二个参数是char* B, 因此还要检查一下字符串的内容:

```
(gdb) x/s 0x08048558
0x8048558 <_IO_stdin_used+4>: "AABBCCDD"
```

现在得到了所需的全部信息:

```
0xbffffee00: int A    = 2
0xbffffee04: char* B = 0x08048558 "AABBCCDD"
0xbffffee08: char* C = 0x00000000
```

在不同的CPU体系结构、操作系统和编译器中，调用函数将参数传递给被调用者的方式也不同。可能需要进行一些实验才能发现函数参数的存储位置。参数可能存储在栈上的主内存中，也可能存储在CPU寄存器中。如果参数存储在栈上，则可以使用上面讲过的命令来找到它的值。如果存储在寄存器中，则使用GDB命令info reg可以输出它们。遗憾的是，在函数执行期间，编译器出于优化的缘故可以自由改写寄存器，这意味着仅当函数刚开始执行时，才能安全地访问其初始值（从调用函数传递的值）。

这里有一些平台依赖的例子：在处理器为Sparc V9 CPU、操作系统为Solaris 5.10、编译器为GCC 3.3.5的机器上，参数存储在寄存器i0、i1和i2中。在GDB中访问它们的方法是使用前缀\$，例如print \$i0。在处理器为64位AMD Opteron、操作系统为Linux Red Hat 3.0、编译器为GCC 3.3.5的机器上，参数存储在寄存器rdx、rsi和rdi中。在这两种环境中，函数F()在执行期间寄存器都将被改写，因此需要在F()中设置一个断点，并在到达该断点时立即读取参数值。

这说明并没有简单、通用的方法可用来查找函数参数。环境可能发生改变，而且上面提到的寄存器名称可能不会适用于未来的CPU类型和编译器。但是，在内存和寄存器中搜索变量数据的方法应该仍然有效。可以在调试会话中使用这些命令来查找参数值：

```
(gdb) break strncmp
Breakpoint 17 at 0x400451fa
(gdb) call strncmp("AAAAAA", "BBBBBB", 0x77777777)
Breakpoint 17, 0x400451fa in strncmp () ...
(gdb) info frame
...
(gdb) info reg
...
(gdb) return
(gdb) delete 17
```

函数strncmp()可能被链接到程序中，此函数是标准C库中的函数，大多数程序都包含标准C库。因此不必修改和重新编译程序。在内存转储中很容易找到如0x77777777这样的值。一旦发现了参数是如何被传递给strncmp()的，就可以用相同的方法为其他函数传递参数。不要忘记调用return语句，它结束对strncmp()的调用，并将栈返回到先前状态。

如果程序在编译时不带调试信息，且只有一个核心转储文件可用，那么这种检索函数参数值的方式在事后调试中特别有用。有关核心转储文件的调试，参见10.9.7节。

10.11.2 读取局部/全局变量和用户定义的数据类型

遗憾的是，在读取局部变量值时并没有简单的方法。编译器可以将变量自由地存储在任意栈上或寄存器中。在Pentium CPU、Suse 9、GCC 3.3.5的机器上，局部变量s存储在寄存器eax中，可以用p \$eax命令输出此寄存器。但并不保证在函数执行期间此变量都保存在相同的寄存器中。

全局变量存储在固定的内存地址中，调试器可以通过变量名找到它们。如果地址是已知的，

则在整个程序执行期间都可以通过这个地址访问全局变量。

读取用户定义的数据类型也是一个难点。我们或许可以访问每个字节，但将每个字节与相应的字段关联起来则相当耗时，而且易出错。

10.11.3 在源代码中查找语句的大概位置

接下来，需要查明F()中的哪条语句导致了崩溃。做一次快速的源代码检查，再结合函数参数，就可以找到根源，但出于演示的目的，这里使用一种更难的方法。

where命令已经显示出F()中的当前地址是0x0804842f。现在，对整个函数F()进行反编译，并检查这个地址在什么地方：

```
(gdb) disassemble F
0x0804840c <F+0>: push    %ebp
0x0804840d <F+1>: mov     %esp,%ebp
0x0804840f <F+3>: push    %esi
0x08048410 <F+4>: push    %ebx
0x08048411 <F+5>: mov     0x8(%ebp),%ebx
0x08048414 <F+8>: sub    $0xc,%esp
0x08048417 <F+11>: mov     0xc(%ebp),%eax
0x0804841a <F+14>: add    %ebx,%eax
0x0804841c <F+16>: push    %eax
0x0804841d <F+17>: call    0x80483bc <size> <== 1st size() call
0x08048422 <F+22>: mov     %eax,%esi
0x08048424 <F+24>: add    0x10(%ebp),%ebx
0x08048427 <F+27>: mov     %ebx,(%esp)
0x0804842a <F+30>: call    0x80483bc <size> <== 2nd size() call
0x0804842f <F+35>: imul   %eax,%esi           <== current adr
0x08048432 <F+38>: mov     %esi,%eax
0x08048434 <F+40>: lea     0xffffffff8(%ebp),%esp
0x08048437 <F+43>: pop    %ebx
0x08048438 <F+44>: pop    %esi
0x08048439 <F+45>: pop    %ebp
0x0804843a <F+46>: ret
0x0804843b <F+47>: nop
0x0804843c <F+48>: nop
0x0804843d <F+49>: nop
0x0804843e <F+50>: nop
0x0804843f <F+51>: nop
End of assembler dump.
```

虽然大部分汇编指令看起来很隐晦，但有些指令还是很易分辨的。例如，对函数size()的调用就很容易查找，它位于F()的nodebug.c文件的第5行和第6行。这个帧的当前地址刚好在第二个调用之后，而且下一条指令是一个乘法运算。这样，就可以确定崩溃发生在第6行。

如前所述，不同的主机CPU类型的汇编代码将有很大不同，因此读取汇编代码总是一个难题。其他复杂因素还包括内联函数的存在、编译器可以自由安排基本块，等等。但是，我们总是有机会找到发生崩溃的语句。

10.11.4 走查汇编代码

调试过程的最后一步是走查F()的代码。我们无法直接走查C源代码，因为没有关于地址与源代码行之间链接的调试信息。但是，我们可以走查机器指令，并像上面讲的那样将机器指令与源代码关联起来。所有好的调试器都具有对机器指令进行反汇编和走查的特性。在GDB中，反汇编命令是disassemble、stepi、nexti。

GDB中有一个有用的命令set step 1，使用此命令可以从带调试信息的代码进入到不带调试信息的函数调用中。将此命令与反汇编和机器代码走查结合起来，就可以用下面的方法来走查F()中的代码：

```
(gdb) start
main (argc=1, argv=0xbffffeeb4) at main.c:10
10    return F (2,             /* A */
(gdb) set step 1
(gdb) step
0x0804840c in F ()
(gdb) disas $pc $pc+5
0x0804840c <F+0>: push    %ebp
0x0804840d <F+1>:  mov     %esp,%ebp
0x0804840f <F+3>:  push    %esi
0x08048410 <F+4>:  push    %ebx
(gdb) nexti
0x0804840d in F ()
(gdb) nexti
0x0804840f in F ()
(gdb) nexti
0x08048410 in F ()
```

这种调试方法较为烦琐，但可用它来跟踪程序的执行流，从而帮助找到bug。

下面介绍一下如何在Visual Studio中调试上述程序。首先在程序的项目上单击鼠标右键，选择**Properties**（属性）。在**Properties**对话框中，找到**Configuration Properties/ C/C++/General**（配置属性/C/C++/常规），并将**Debug Information Format**（调试信息格式）设置为**Disabled**（禁用）。这样编译后的程序就不带调试信息。要在Visual Studio中调试不带调试信息的程序，单击**Step- Into**（逐语句）启动程序。这时出现一个选项卡窗口**Call Stack**（调用堆栈），在程序上单击右键，选择**Go To Disassembly**（转到反汇编），弹出另一个窗口，其中显示了反汇编后的代码。表示函数main和F的项分别是：

```
_main:
...
_F:
...
```

可以走查代码和设置断点，还可以通过将光标指向相应的项来检查寄存器和内存的值，也可以更改汇编代码。

经验总结

- 即使在没有调试信息的情况下，也可以使用调试器来调试程序。可以读取函数参数、检查调用栈以及走查代码。
- 如果你恰好精通所使用的处理器的汇编语言，并且了解C/C++函数的调用惯例，那么可以说你已经有了所需的一切。局部变量的地址可能很难确定，但或许能够找到与它们相邻的函数调用参数。

编写可调试的代码

在程序开始投入使用后，很有可能bug也会接踵而至。因此，我们需要对程序进行调试，而此时程序的可调试性就会呈现在人们面前。并非所有程序都具有很高的可调试性。本章将给出一些提示和技术，讲解如何使程序从一开始就具有较高的可调试性。

调试在很大程度上意味着执行反向工程，特别是在程序不是调试者编写的情况下。11.1节和11.2节讨论注释和编码风格。源代码越容易理解，可调试性就越高。如何编写源代码取决于个人风格和喜好，本章将解释一些明显的技巧。

本章以类似#define INCR 20这样的宏为例。这样的宏会妨碍交互式调试，11.3节介绍如何用不同的语言结构来替代它们。

11.4节介绍如何在程序中多构建一些起到辅助作用的调试函数。如果可以预见到将来会在程序调试上花费大量时间，那么为什么不尽早做些准备工作呢？未来的某一天会证明这些辅助函数是很有价值的。

最后，11.5节介绍如何为程序的事后调试做好准备，也就是将一些与程序活动有关的提示写到日志文件中，这样在程序发生崩溃时可以提供足够的信息。

11.1 注释的重要性

在源代码注释以及它们是否有助于调试过程这个问题上，有两种截然不同的观点。第一种观点认为源代码注释是软件必不可少的部分，如果没有这些注释，将很难理解和维护软件。第二种观点则认为注释非常危险。注释可能是针对第一个源代码版本编写的，这些注释很快就过时了，因为当修改代码时，不一定会同步修改注释。这通常导致错误的注释，比没有注释还要糟糕。因此，第二种观点主张以一种自文档化的方式来编写代码，放弃不必要的注释。我们认为应该折中这两种极端观点，即源代码应尽可能做到自我解释。但不管怎样，只要注释为自文档化的源代码增加了信息，它们对于可维护性和可调试性就是至关重要的。这也说明注释必须满足以下条件：它们描述了函数或代码片段的作用（做什么），并潜在地表示了选择特定解决方案的动机（为什么）。源代码中的不太明显的技巧也值得用一些注释加以说明。

调试在很大程度上需要反向工程的技巧，因为我们经常要对别人以前编写的源代码进行调试。好的注释总是受人欢迎的，因为它们使源代码更好理解。

以下几小节给出一些示例，说明注释在调试中的作用。

11.1.1 函数签名的注释

注释应清晰地表述以下内容。

- 函数要做什么。这样的行为描述应该与实际实现无关。例如，函数`data* get(content *list, const char* key)`的注释应该解释清楚这样一件事：此函数在与提供的字符串值匹配的数据结构中检索第一个地址项，相反，此注释不应该详细描述C++标准库中的基本映射算法。
- 说明函数参数，以及在异常情况下如何处理这些参数。示例：`data* get(content *list, const char* key)`允许`list`为NULL指针，并在这种情况下返回NULL。
- 接口使用上的假设。函数`data* get(content *list, const char* key)`中所提供的字符串关键字是否有特殊的格式要求（如禁用空格和换行）？是否有编程顺序的要求（如只能在调用`init()`之后才能调用`get()`）？
- 内存分配。如果函数返回一个指向类对象的指针，那么该对象有多长的生命期？谁负责删除它？
- 副作用。有了副作用就已经很糟了，如果不在注释中记录它们则问题更加严重。
- 记录所有已知的陷阱和临时折中办法。

11.1.2 对折中办法的注释

为了解决库函数、编译器或硬件bug引起的问题，程序员有时需要在代码中使用特殊的折中办法才能使程序工作。一旦修复了原来的bug，这些数据结构有时就成了有缺陷的代码。用一段注释来说明折中办法都做了什么，以及为什么要这样做，将为调试人员提供极大帮助。

11.1.3 为不确定的代码加注释

程序员应该实事求是地表述对特定代码片段的怀疑，这将大有帮助。程序员应该为一些不完全清楚的事情加上注释，以便在调试时注意它们。例如：

```
delete[] a; /* not sure if we can really deallocate here */
```

这样的注释就是追查分段错误时的一个很好的提示。

11.2 采用一致的编码风格

有些人过度追求某种编码风格——使用大写、缩进、括号，等等。重要的不是采用哪种编码

风格，而是总体风格应该便于阅读和理解代码。一致的编码风格有助于理解别人编写的源代码。这就是为什么在修改现有软件项目时应该保持一致编码风格的原因，只有当开发新项目或新的软件模块时，才能引入新的编码风格。

11.2.1 仔细选择名称

变量和函数名称应该具有很好的说明性。采用这条原则后，就可以得到自文档化的代码。在命名惯例上没有统一的标准。尽管如此，仍有一条事实上的标准，即遵守C++标准库的规则，如下所述。

- 类、结构、枚举、`typedef`、函数、变量、常量和命名空间均使用小写，单词之间用下划线隔开。
- 模板参数名称以大写字母开头，后面每个单词均以大写字母开头。
- 宏的名称全采用大写，单词之间以下划线隔开。
- 不要创建以下划线开头的标识符，这样的标识符留给编译器、系统库和操作系统。

11.2.2 不要使用“聪明过头”的结构

Brian W. Kernighan曾经写道：

“调试要比编码难两倍。因此，如果你已经付出了全部的聪明才智编写代码，在调试上就无能为力了。”

记住，软件有bug，因此需要有人来调试它，而调试者往往不是代码的编写者。调试工作主要是对代码执行反向工程，以查明代码的功能是什么。代码越“聪明”，就越难进行反向工程。

11.2.3 不要压缩代码

不要将过多的语句压缩到一行中，因为这不利于调试。在下面的代码示例中，`if`语句和`then`子句在同一行中：

```
1 if (is_this_true() || is_that_true()) expr1 = expr2++;
```

在走查源代码时，无论`if`语句的`then`子句是否执行，调试器在第1行只停止1次。这为查看是否满足条件设置了不必要的复杂性。更好的方法是将代码改写为两行：

```
1 if (is_this_true() || is_that_true())
2     expr1 = expr2++;
```

现在，调试简单多了。当调试器进入第2行时，表达式为真。

11.2.4 为复杂表达式使用临时变量

考虑由子表达式组成的复杂表达式，其中有些子表达式是函数调用或操作符调用。那么，如

何才能找到子表达式的值呢？一种方法是用调试器的step-into和step-out命令来跟踪程序的执行流，但这非常耗时。另一种方法是直接输出子表达式的值，但当调用内联函数或操作符时可能无法使用这种方法。

在下面的示例中，`is_this_true()`和`is_that_true()`这两个函数的返回值不易分析，而且一旦经过了第1行之后，根本不可能再分析它们：

```
1 if ( is_this_true() || is_that_true() )
2     expression1 = expression2++;
```

如果重写上面的代码，用临时变量保存这两个函数的返回值，那么分析就简单多了：

```
1 int this_is_true = is_this_true();
2 int that_is_true = is_that_true();
3
4 if ( this_is_true || that_is_true )
5     expression1 = expression2++;
```

注意，更改后的代码与原来的版本并不完全相同。原来，仅当`is_this_true()`返回`false`时，才调用`is_that_true()`，现在总会调用它。这个区别可能影响运行时速度。如果被调用的函数有副作用，这还会影响程序的行为。

11.3 避免使用预处理器宏

宏会增加调试的难度，因此应尽力避免使用预处理器宏。在代码进入编译器之前，预处理器会替代宏的内容，也就是说，宏的内容在进入编译器之前就已经不存在了，因此对调试器来说是不可见或未知的。在走查源代码的过程中，当到达一个使用了宏的位置时，实际上完全是遇到了一个盲点。

尽管有时使用宏是最实用的解决方案（例如当使用拼接“##”时），但一般都可以通过其他方法实现相同目的。以下几小节将介绍这些替代方法，并解释为什么它们更适用。

11.3.1 使用常量或枚举来替代宏

一种很常见的编程习惯是使用预处理器的`#define`语句定义要在一个或多个代码文件中使用的常量。预处理器在编译之前会替代它们，因此，宏的名称并不表示调试器的符号，因此不能显示。一种替代方法是使用枚举或常量。

以下示例定义了4个宏：`NUM_REGISTERS`、`INCR`、`DECR`和`INVERT`。它们在函数`execute_unary()`中使用，此函数在CPU仿真程序中执行一元运算：

```
1 #include <assert.h>
2 #include <stdlib.h>
3
4
5 #define NUM_REGISTERS 8
```

```

6 #define INCR          16
7 #define DECR          20
8 #define INVERT         22
9
10 int* registers;
11
12 void execute_unary( int opcode, int reg )
13 {
14     assert(0 <= reg && reg < NUM_REGISTERS);
15
16     if (opcode == INCR)
17         ++registers[reg];
18     else if (opcode == DECR)
19         --registers[reg];
20     else if (opcode == INVERT)
21         registers[reg] = ~registers[reg];
22     else
23         assert(0); /* illegal unary opcode */
24 }

```

这4个宏对调试器都是不可见的：

```

(gdb) ... run till line 14 ...
14     assert(0 <= reg && reg < NUM_REGISTERS);
(gdb) p NUM_REGISTERS
(gdb) No symbol "NUM_REGISTERS" in current context.
(gdb) p opcode
(gdb) $1 = 20
(gdb) p INCR
(gdb) No symbol "INCR" in current context.

```

现在重写上面的程序，以便使它更易于调试。用一个常量（`const int num_registers = 8`）来替代宏`NUM_REGISTERS`，并用枚举来替代宏`INCR`、`DECR`和`INVERT`：

```

1 #include <assert.h>
2 #include <stdlib.h>
3
4
5 const int num_registers = 8;
6 enum UnaryOpcodeEncoding {incr=16, decr=20, invert=22};
7
8 int* registers;
9
10 void execute_unary( enum UnaryOpcodeEncoding opcode, int reg )
11 {
12     assert(0 <= reg && reg < num_registers);
13
14     if (opcode == incr)
15         ++registers[reg];
16     else if (opcode == decr)
17         --registers[reg];
18     else if (opcode == invert)
19         registers[reg] = ~registers[reg];

```

```

20     else
21         assert(0); /* illegal unary opcode */
22     }

```

现在调试器完全可以访问这4个定义了：

```

(gdb) ... run till line 12 ...
12 assert(0 <= reg && reg < num_registers);
(gdb) p num_registers
(gdb) $1 = 8
(gdb) p opcode
(gdb) $2 = decr
(gdb) p (int)incr
(gdb) $4 = 16

```

以枚举形式来定义常量不仅使调试更简单，而且还可以提高源代码的可读性，因为类型 `UnaryOpcodeEncoding` 给出了用途提示。

但是，使用 `const` 定义也有几个小的弊端。`const` 定义会在符号表中增加一个符号，因此生成的对象文件将略微增大。在链接期间，此符号还可能与同名符号发生冲突，但可以通过 C++ 的命名空间或者将常量声明为 `static`（也就是将常量限定在文件范围内使用）来避免冲突。另一方面，用 `#ifdef` 定义的宏也可能与另一个宏定义发生冲突，因此在引起冲突这个问题上，这两种方法或多或少是相同的。

11.3.2 使用函数来替代预处理器宏

比使用预处理器定义常量更坏的习惯是用预处理器定义函数。正如前一小节所讲的预处理器常量一样，调试器无法找到宏函数的符号，而且无法走查函数代码，因此使调试变得很困难。使用预处理器宏函数的原因可能是它的执行速度比一般函数快，因为预处理器将函数内联到代码中。在 C++ 中可以利用 `inline` 函数实现相同的效果，它指示编译器内联函数的代码。下面的示例显示了一个宏函数 `MAX` 和一个 `inline` 函数 `min`，编译器都可以有效地将它们内联到代码中：

```

1 #define MAX(x,y) ((x)>(y))?(x):(y)
2
3 inline int min(int x, int y)
4 {
5     if (x<y) return x;
6     else      return y;
7 }
8
9 int main()
10 {
11     int a1 = 0, a2 = 42, a3;
12
13     a3 = MAX(a1,a2);
14     a3 = min(a1,a2);
15
16     return 0;
17 }

```

使用预处理器宏函数的另一个原因是它们支持不同数据类型的参数，只要这些数据类型支持函数内部的操作即可。在上例中，宏函数MAX可以使用所有的本地C数据类型，而函数min只支持int数据类型。如果可以使用C++编译器，则最好的解决方案是使用模板，这可以提高可维护性、调试性和灵活性。下面就将min函数定义为一个模板函数：

```
1 template<class T>
2 inline const& T min(const T & x, const T & y)
3 {
4     if (x<y) return x;
5     else      return y;
6 }
```

这个模板可以使用不同的数据类型，只要参数x和y具有相同类型即可。

11.3.3 调试预处理器输出

如果必须使用宏，而又需要调试它，可以直接观察预处理器的输出。通常，编译器调用预处理器，而预处理器的输出不可见。但是，大多数编译器支持-E标志，这样在调用预处理器后就可以输出结果，然后停止。下面仍以MAX宏为例加以说明：

```
File main.c:
1 #define MAX(x,y) ((x)>(y)?(x):(y))
2
3 int main()
4 {
5     int a1 = 0, a2 = 42, a3;
6
7     a3 = MAX(a1,a2);
8 }
```

现在，输出预处理器结果：

```
> gcc -E main.c > main.post.c
```

预处理器的输出如下^①：

```
File main.post.c:
1 # 1 "main.c"
2 # 1 "<built-in>"
3 # 1 "<command line>"
4 # 1 "main.c"
5
6
7 int main()
8 {
9     int a1 = 0, a2 = 42, a3;
10
11    a3 = ((a1)>(a2)?(a1):(a2));
12 }
```

^① 不同编译的实际输出也不同。

第11行是MAX宏的展开，这正是我们要调试的对象。

注意第1~4行以#开头的编译器指令# <line number> <filename>。它们通知编译器（因此也通知调试器）将当前源文件（main.post.c文件）向回映射到一个不同的源文件。当编译并调试main.post.c文件时，调试器将显示main.c文件，而不是main.post.c。如果不希望这种结果，可以删除第1~4行，并重新编译。

注意，生成的预处理器输出可能很长，因为它包含在所有层次上展开的文件。像#include <iostream>这样的一条语句可能会展开成数千行^①。

11.3.4 使用功能更强的预处理器

尽管许多宏可以用更便于调试的代码结构来替代，但预处理器宏仍具有自身的优势。例如，当编写需要在不同平台上运行的代码时，当将一些运行时任务拿到构建过程中完成时（从而优化程序执行速度），或者当保护特殊的调试代码时，都可以使用预处理器宏。在这些情况下，建议使用功能更强的宏预处理器，例如m4（参见附录B.8.2），而不要使用开发环境中的预处理器。第一个原因是m4比大多数集成在编译器包中的预处理器的功能更强。第二个原因是有利于调试，因为m4预处理器在最终源代码编译之前被调用。因此，可以直接调试最终的C/C++源代码，而没有隐含的宏展开（macro expansion）妨碍调试。以下代码示例演示了如何在设计流中使用m4预处理器：

```
File matrix.m4cpp:
1 include(forloop.m4)
2 define('identity_matrix',
3     'int id$1[$1][$1] = {forloop('ii',0,eval($1-1),
4         forloop('jj',0,eval($1-1),`eval(jj==ii),'))})dn1
5
6 int main (int argc, char* argv[])
7 {
8     int count = 0;
9     identity_matrix(5);
10    ...
```

这段代码中混合了C/C++代码和m4宏。宏identity_matrix定义并初始化了变量维度的单位矩阵（identity matrix）。在本例中，它的参数是5。m4对文件matrix.m4cpp进行预处理，并将结果写到宏展开所得到的文件中：

```
> m4 matrix.m4cpp > matrix.cpp
> cat -n matrix.cpp
1
2 int main()
3 {
4     int count = 0;
```

^① 当使用GCC 3.3.5编译器和Suse 9操作系统时，约为30000行。

```

5     int identity_matrix5[5][5] = {
6         1,0,0,0,0,
7         0,1,0,0,0,
8         0,0,1,0,0,
9         0,0,0,1,0,
10        0,0,0,0,1,};
11    ...

```

然后，对得到的文件matrix.cpp进行编译和调试。这里的关键在于，调试器将引用matrix.cpp，它是一个纯的C代码文件。如果使用了宏的程序出现异常行为，可以分两步来调试宏。首先，比较文件matrix.m4cpp和matrix.cpp，检查宏是否创建了符合预期的C代码。然后，用调试器来检查C代码是否按预期工作。当发现bug时，可以手工修改C代码，直到它正确工作，然后调整宏，让它创建所需的C代码。

11.4 提供更多调试函数

当开发或调试复杂程序时，有时需要反复显示正在处理的一些程序状态或数据信息。开发人员经常要花费一些力气在调试器中收集所需信息，或者过滤出所需的信息。为了加快这个过程，一个好的思想是提供特殊的调试函数。它们可以是代码的集成部分，也可以从外部链接到程序中。这些调试函数的唯一目的就是提取或过滤调试所需的信息。其他的应用例子还有用于检查数据完整性和一致性的例程，包括分析例程，例如显示散列表中的数据分布。

11.4.1 显示用户定义的数据类型

与本机的C/C++数据类型不同，在C/C++中无法将用户定义的数据类型直接传递给I/O例程，例如printf或cout。用户必须定义适当的显示方法或流方法（streaming method）。如果数据类型很复杂，那么这些方法应该是可配置的。例如，显示一定程度的细节（冗长级别），或配置输出的位置（stdout、stderr或用户定义的FILE指针）。这些自定义的I/O方法也可在调试器中进行交互式调用。

示例：假设有一个C++类People，它存储了一个人员数据列表（姓名、地址、电话、出生日期）。现在从调试器内部调用调试函数People_debug_print(People* obj, int level)。通过参数level可以选择输出信息的详细程度：不输出统计数字（level=0）、只输出每个人的姓名（level=1）、输出所有可用信息（level=2）：

```

(gdb) call People_debug_print(my_close_friends,0)
3 persons

(gdb) call People_debug_print(my_close_friends,1)
3 persons
Meggy Meyers
Paul Smith
Martha Miller

```

```
(gdb) call People_debug_print(my_close_friends,2)
3 persons
Meggy Meyers
  addr: 112 Flynn Ave / 12345 LittleTown
  phone: 123-7654-321
  DOB: 1967-02-24
Paul Smith
  addr: 496 Thompson Lane / 12345 LittleTown
...
```

以下是有关调试函数的一些建议。

- 一致地应用于多个类。在具有很多个类的复杂程序中，应该使用一致的命名模式，并将这些调试函数应用于所有的类。例如，为所有类提供一个`debugprint(int verbosity)`方法，它将类对象的相关信息打印到标准输出，其中当参数`verbosity=0`时，只输出摘要，当`verbosity=1`时，输出最有用的信息，当`verbosity=2`时，输出所有信息。
- 内联。如果可能，不要将调试函数编写为内联函数，因为调试器可能无法调用它们。
- 由环境变量触发。一个典型的调试特性是从程序调用调试函数（在设置了特殊环境变量的条件下）。
- 调试与效率。添加调试函数将增大最后的可执行程序。如果在最终的产品版本中无法接受这项开销，则可以使用编译器指令`#ifdef ENABLE_DEBUG_CODE`阻止调试函数的主体或整个定义。
- 链接。程序代码一般不能直接使用调试函数，因此链接器可能会忽略它们。有几种方法可以避免此问题，其中有两种比较简单的方法，一是将调试函数编写到一个已链接到程序的源文件中，二是使用链接器标志^①来强制链接。

11.4.2 自检查代码

通过添加断言和分析函数使程序进行自我检查。与前面讲的显示函数不同，分析函数还检查数据或控制状态信息。当很难维护复杂数据的一致性时，这具有特别重要的意义。当跟踪对象内容是否被破坏，以及何时被破坏的时候，从调试器调用分析函数是一种便捷的方法。自检查代码和断言的详细内容可参见[Zeller05]。

在设计分析函数时，令其在内部状态正确的情况下返回1，而不产生输出。接下来，在适当的位置添加断言语句`assert(analysis fn(my object))`。当分析函数第一次失败时，程序将自动中止。在运行回归测试时这是一项有用的检查。

在哪里以及何时调用分析函数，需要有良好的判断。在较多调用分析函数和较少调用之间有一个折中，前者将带来更好的调试机会，后者则可以减少运行时间开销。正如可以关闭自检查代码一样，在必要情况下也应该使用`#ifdef`命令阻止分析函数和断言。

^① 例如，在GCC编译器中可以使用标志`-u <debug-function>`。

11.4.3 为操作符创建一个函数，以便帮助调试

C++支持将特定操作符映射到用户定义的代码。有时，我们无法从调试器调用特定的操作符。为了绕过这项限制，可以创建一个从内部调用操作符的全局函数。现在就可以从调试器调用此函数了，参见下面的示例：

```
class myClass
{
    ...
    double operator() (int factor) const;
    ...
};

double
myClass_op_parenth (const myClass &c, int factor)
{
    return c(factor);
}
```

现在就可以从调试器调用`myClass op_parenth(my object, 5)`来验证操作符`my_object(5)`的行为。

11.5 为事后调试做准备

最有效的调试方法是用一个小的测试用例对程序进行交互式调试。但是，有时却无法进行这样的调试，例如程序只是偶尔才发生崩溃，或者由于机密性原因无法得到测试用例。

这就要求我们必须采用事后处理的模式来进行调试，这意味着必须在程序完全结束之后查明都发生了哪些事情。如果实际的bug或崩溃事件只留下很少的数据，那么事后调试实质上是不可能的，至少是非常低效的。可能会有一个核心文件显示了崩溃时的调用栈、某些变量的值和内存内容。但是，可能只有极少的数据记录了崩溃之前都发生了什么事情。唯一留下的内容就是命令行解释器中的几行输出，再有就是用户所记得的一点东西。

生成日志文件

航空业面临着同样的困境。如果飞机失事了，必须要查明确切原因，以确保问题（bug）不再出现。人们采取了很多措施，包括在飞机上安装黑匣子，用于记录最重要的活动和飞机的状态信息。数据记录一直在进行着，但只有最后半小时的记录才能保存下来。在飞机失事的分析中，这些通常是最重要的数据。

软件中的黑匣子就是日志文件。程序员负责创建日志文件，并定义记录哪些类型的信息。通常在运行时间/磁盘空间开销与调试的难度间存在一个折中的选择。

如果不对日志文件的大小加以限制，它最终将占满整个磁盘，从而自身就构成一个bug。通

过限制文件大小可以解决此问题，例如，只保存最后1000行或100 KB数据。最简单有效的办法是使用两个日志文件，每当当前文件到达定义的限制时，就切换到另一个文件。

要确保经常刷新输出缓冲，否则在程序发生致命错误时可能会丢失最后几行输出，而这可能是最重要的信息。示例如下：

```
...
FILE* log_file;
...
void calibrate(...)
{
    ...
    if(log_file) {
        fprintf(log_file, "Calibrating phalanx %s\n", ...);
        fflush(log_file); /* important or lines may be lost */
    }
    ...
}
```

经验总结

- 用注释说明源代码中一些不太明确的地方。描述代码的用途，以及为什么要以特定方式编写它。
- 在命名常量、类、成员和变量时要使用一致的命名惯例，并仔细选择名称。
- 不要使用预处理器宏。有一些很好的替代方法，例如枚举、常量、内联函数或模板。
- 添加可以从调试器调用的调试函数。这些函数可以正确输出用户定义的数据类型，并检查数据库的完整性。
- 为事后调试做准备：创建一个可选的日志文件，记住要经常调用fflush()，并使代码具有自我检查功能。

静态检查的作用

12

本章介绍一些称为静态检查器的工具。静态检查器可以在不运行程序的情况下对其源代码进行分析。理想情况下，这些工具作为常规软件构建过程的一部分运行，用于查找一些可通过静态源代码分析发现的特定bug。典型的检查包括语法错误、内存的错误使用和不可达代码（unreachable code）。本章后面在讨论各种可用工具时，将给出一个更详尽的错误清单，这些错误都可以通过静态分析来发现。

通常，我们不会在调试过程中使用静态检查器，因为实践表明，用静态检查器找到特定bug根源的可能性是很小的。但是，如果静态检查器在软件中发现很多冲突，或者编译器发出很多警告，那么这样的软件有可能含有大量bug。这些bug可能在软件维护期间显现，例如，当编译器升级到新版本时，当软件被移植到新的CPU平台时，或者当软件以新的方式被重用时，都可能出现bug。因此，有些软件开发团队制订了一条明确的规则，即源代码库必须通过一个或多个静态检查器的检查，而且错误应接近于零。应通过两种方法来定期排查冲突和警告，一是清理代码，二是使用防御性编码实践，从而避免使用高级或易出错的编码风格。

12.1节介绍如何将C/C++编译器作为最基本的静态检查器使用。随后几节介绍更复杂的检查器，并列举它们的优点，以及要想正确使用它们需要做哪些工作。

12.1 使用编译器作为调试工具

近年来，C和C++编译器有了很大的发展和完善，它们不仅报告语法错误，而且还具有一些静态代码检查特性。除了报告常规错误以外，现代的编译器还将报告如下这些警告。

- 在枚举类型的分支语句中丢失了case。
- 未使用的函数、函数参数或标签。
- 对register变量进行了取址操作。
- 整数被零除。
- 死代码（即不可达代码）。

- 丢失了函数声明和return语句。
- 内存的错误使用：未使用的或未初始化的变量。
- 未来将与C++标准不兼容的地方。
- 与64位CPU不兼容。

一个广泛应用并获得成功的软件项目编程规则是启用编译器中的所有警告，并在常规开发期间修复软件，或在特殊的清理项目中修复，做到（几乎）不再出现警告。

首先讲一下如何启用编译器警告。机制通常是相同的：编译命令接受命令行上的编译标志，编译标志决定了如何处理警告——不显示所有警告、启用所有警告或将所有警告作为错误来处理。此外，通常还有一些标志可用来启用/禁用特定类型的警告。编译器手册会列出这些与警告有关的标志的清单。表12-1列出了在三种常用编译器中用于控制警告显示的最常见标志。

表12-1 用于警告消息的编译器标志

	GCC	Sun CC	Visual C++
启用所有警告	-Wall	+w	-Wall
不显示所有警告	-w	-w	-w
将警告转换为错误	-Werror	-xwe	-WX

12.1.1 不要认为警告是无害的

下面给出第一个由编译器报告的警告的例子，程序中有两个问题：

```

1 /* testinit.c */
2 int main() {
3     int v[16];
4     int i, j, k;
5     j = i;
6     v[i] = 42;
7     return 0;
8 }
```

第一个问题是变量k未使用。第二个（也是更严重的）问题是变量i作为数组索引被使用之前，没有一个初始化值。

当用GCC编译器的默认设置编译testinit.c示例时，结果是没有警告。但使用-Wall标志时将产生以下警告：

```
testinit.c:4: warning: unused variable 'k'
```

当使用Sun编译器编译testinit.c示例时，结果显示以下警告，与是否使用+w标志无关：

```
testinit.c", line 5: Warning: The variable i has not yet
been assigned a value.
```

当使用Visual C++ 7.0、8.0和9.0版本的默认设置时，将发现变量i的问题。当使用-Wall标志

时，两个问题都可以找到。

```
testinit.c(4): warning C4101: 'k' : unreferenced local variable
h:\src\testinit.c(5): warning C4700: local variable 'i' used
without having been initialized
```

建议不要忽略编译器警告。如果警告表明有一个实际或潜在的bug，则应该修复它。如果bug是无害的，或无法修复，应在代码中用注释进行解释。

下面给出另一个示例，它表明相同类型的警告可能是无害的，也可能是一个严重的bug。在我们的示例中，有两个指针P和Q指向some_class对象。代码必须检查P和Q是否指向同一对象。遗憾的是，源代码中有一个录入错误，即第8行中的比较符==被误写成赋值符=：

```
1  /* myfile.cpp */
2  some_class* P = ...
3  some_class* Q;
4  if ( Q=find_pointer(...) )
5      // do something with Q
6  if (P=Q)           <----- typo, should have been P==Q
7      // they are equal, do something special
8  ...
9  ... further processing of P, Q ...
```

当使用-Wall标志时，GCC编译器产生以下警告：

```
> gcc -Wall myfile.cpp
myfile.cpp:4: warning: suggest parentheses around
                  assignment used as truth value
myfile.cpp:6: warning: suggest parentheses around
                  assignment used as truth value
```

注意，如果不使用-Wall标志，将不会得到任何警告，这正是为什么要始终启用最高的警告级别的原因。

第4行的警告是无害的，这条语句首先为Q赋值，然后检查它是否为NULL指针。这正是程序员想要做的，因此不是真正的bug。但是，第6行的语句是一个实际的bug，这条语句并未执行程序员所设想的操作。它没有比较两个指针，而是首先将P重写为Q的值，然后与一个NULL指针进行比较，因此if语句匹配了一种完全不同的条件。更糟的是，P已经被意外修改了，这可能导致后面产生更多副作用。

这两条if语句看起来都是可疑的，触发了完全相同的警告，而且我们必须检查它们的警告消息。这里的建议是修改第4行的语句，使得调试器能够更明确地理解其意图。同时应修复第6行中的bug：

```
4  if ( 0 != (Q=find_pointer(...)))
5  ...
6  if (P==Q)
```

以上示例说明不应认为警告是无害的，还说明花少量时间来清除在软件中遇到的警告消息是

值得的，因为这将减少后面的调试会话的需要。

12.1.2 使用多个编译器来检查代码

从上面的testinit.c示例可以看到，并非每个调试器都能够给出每个可能问题的警告。因此，好的做法是不要使用一个调试器来开发软件，而应该将C或C++代码编写为与多种编译器兼容，并且在多个编译器中编译时均不产生警告消息，如果能在多平台上工作就更好了。

12.2 使用 lint

lint是C语言最早使用的静态检查器之一，也是最著名的一个。lint是在1979年作为UNIX操作系统的一部分出现的，它仅限于C语言，而不支持C++。这里不对它进行详细讨论，因为像GCC这样的编译器已包含lint中内置的大部分检查功能。更多信息参见附录B.6.2。

Splint（Secure Programming Lint）是一个基于最初的UNIX lint的静态检查器，它增加了更多检查功能和源代码注释功能。更多信息参见附录B.6.3。

12.3 使用静态分析工具

附录B.6列出了一组利用静态代码分析技术的工具。以下是其中最重要的一些工具（在本书创作时），它们可用于对C/C++代码进行规则检查。

- Coverity Prevent是一个商业静态分析工具，它在C/C++代码中执行规则检查。参见附录B.6.1。
- PC-lint（在非Windows平台上也称为FlexeLint）是一个商业静态分析工具，它用于检查C和C++代码。参见附录B.6.7。
- QA C++是一个商业静态分析工具，它用于检查C和C++（也可检查其他语言），主要是进行规则检查。参见附录B.6.8。
- Codecheck是一个商业静态分析工具，它检查C和C++代码的规则冲突。参见附录B.6.9。
- Visual C++ 8.0企业版（适用于团队开发）支持/analyze代码分析选项，此选项被集成到Visual Studio中。参见附录B.6.4。
- Parasoft C++test是一个静态分析工具，它用于规则检查，并且能够自动创建测试工具。参见附录B.6.12。

所有这些工具都具有类似的特性和使用模式，因此下面以Coverity为例来解释它们。Coverity发现的bug类型以及此工具的使用方法在静态检查器中具有代表性。

12.3.1 了解静态检查器

静态检查器的工作原理是解析程序的C或C++源代码，然后通过在代码上执行规则检查来进

行静态分析。静态分析不必运行程序，因此无需提供测试用例和输入数据。

静态检查器能够检测到的第一类问题是内存错误，包括未初始化的变量、内存和文件句柄泄漏、缓冲区溢出和破坏、空指针访问、使用已释放的变量，以及重复调用`free/delete`。静态检查器的功能与动态内存检查器（例如第4章中讨论过的Valgrind或Purify）有一些重合。静态检查器的优点是不必为了执行代码而构建测试用例，也无需提供输入值来触发有问题的代码。它的缺点是在发生内存泄漏这样的问题时，不能提供足够信息来确定某一代码段是否正确。

Coverity能够检查的其他问题还包括死代码、多余代码（如重复的null检查），以及API使用错误，如STL使用错误和不正确的错误处理。

要运行Coverity，必须将它集成到应用程序的构建系统中。由于C/C++语言有多个版本，因此Coverity在检查代码的正确性之前，首先需要确定具体的编译器版本、编译标志、系统头，等等。

很多工具中都已经集成了Coverity，例如Linux的make构建命令、Visual Studio、Eclipse和很多其他IDE。附录B.6.1列出了Coverity文档的链接。产品文档列出了当前支持的构建系统，并给出了如何作为构建系统的一部分来运行Coverity的细节和示例。

为了证明Coverity的价值，下面给出一个检测死代码的示例。死代码是指在程序执行期间永远无法到达的源代码。死代码在大型软件项目中是很常见的。有时，由于这些代码是先前软件版本的残留物，因此不再需要执行，而且没有人记得它的用途是什么。在这种情况下，应该删除死代码，或者将它注释掉。

在大多数情况下，死代码是由错误的条件表达式引起的。下面的程序给出了一个示例，它有两个嵌套的`if()`表达式，这两个表达式重叠了，因此第二个`if()`中的语句永远不会执行。

```

1  /* testdead.cc */
2  int main() {
3      int v[16];
4      int i;
5
6      for(i=0; i<16; i++) {
7          if(i > 8) {
8              v[i] = i;
9              if(i <= 8) {
10                  v[i] = -i;
11              }
12          }
13      }
14      return 0;
15 }
```

在上例中，当第一个`if()`表达式`i > 8`保持为真时，第二个`if()`表达式`i <= 8`将永远不为真，因此其中的代码永远不可达。下面是Coverity生成的报告：

```
6      for(i=0; i<16; i++) {
```

```

Event between: After this line, the value of "i" is between 9 and 15
Event new_values: Conditional "i > 8"
Also see events: [dead_error_line] [dead_error_condition] [new_values]

7     if(i > 8) {
8         v[i] = i;

Event dead_error_condition: On this path, the condition "i <= 8"
could not be true
Also see events: [dead_error_line] [between] [new_values] [new_values]

9         if(i <= 8) {

Event dead_error_line: Cannot reach this line of code
Also see events: [dead_error_condition] [between] [new_values] [new_values]
10        v[i] = -i;

```

上面报告了if语句中的代码不可达，注意这是如何报告的。这里，条件表达式的精确分析使我们很容易理解死代码的原因。

12.3.2 将静态检查器检测到的错误减至（接近）零

当第一次使用静态检查器对大型应用程序进行检查时，警告和错误的典型分布情况如下。

- 40%的错误是假阳性报告，实际上它们是正确的代码。
- 40%的错误是同一个问题的多次出现。
- 10%是较小或较浅显的问题。
- 10%是严重的bug，通过其他方法很难发现它们。

从以上错误分布可以看出，在最初的问题分析上需要付出很多工作。不管静态检查器报告了什么类型的错误，修复所有错误总是一个好的做法。在某些情况下，检查器可能产生假阳性，即把正确的代码标记为错误。大多数静态检查器都有防止假阳性的机制——或是利用在代码中加注释，或是将有错误的函数加入黑名单。有关Coverity黑名单项的确切格式，可参考它的文档。

在实践中，要排除假阳性和重复问题报告是很简单的工作，但需要耗费大量时间。当把大量无需修复的错误分离出来后，一小部分严重bug就会浮出水面了。应该注意到这些问题不是偶然出现的，也不应仅仅修改代码使得错误消息不再出现。相反，我们需要检查代码，以确定它的功能，然后才能进行修改。

建议对静态检查器做一次清理，全部排除它所报告的错误，或者是减至一个很小的数量，使之对代码不构成影响。当将Coverity报告的错误减至这个基线时，新的错误就只会在修改代码时出现。在大型软件项目中，新代码占总代码的比例很小，因此不必在每日的“编辑-编译-运行”周期中进行静态检查。在夜间的软件生成期间运行Coverity并安排好常规的Coverity清理过程就足够了。

12.3.3 完成代码清理后重新运行所有测试用例

根据静态检查器的错误报告对错误进行修改之后，必须执行两项任务。第一任务是重新运行

检查器，检查错误是否已修复，而且未产生新的错误。第二项任务（也更为重要）是执行所有动态测试用例，以确保程序正常工作。

12.4 静态分析的高级应用

静态源代码分析的其他应用还有如下几个。

- 可移植性：检查代码的可移植性问题，例如大端（big endian）与小端（little endian）之间的可移植性、内部（非标准的）头文件的使用、32位与64位之间的可移植性。示例为 Codecheck，参见附录B.6.9。
- 反向工程：创建类文档，用于说明类的结构、类之间相互的使用关系，以及软件模块之间的依赖性，等等。示例为Understand C++，参数附录B.6.14。
- 代码统计：通过建立统计数字计算软件的整体质量。典型的应用是查找重复的代码段或接口分析。示例为Axivion Bauhaus Suite，参见附录B.6.10。
- 安全性：通过专用于Web应用程序的规则检查器检查缓冲区溢出和SQL查询。示例为 Klocwork（参见附录B.6.5）、Fortify（参见附录B.6.6）。

经验总结

- 不要忽略编译器警告，即使它们看起来是无害的。
- 使用多个检查器检查代码。
- 了解静态检查器。
- 将静态检查器报告的错误减至（接近）零。
- 完成代码清理后重新运行所有测试用例。
- 坚持常规的源代码清理，这将实现长期回报。

如 果读者已经从头至尾读完了本书，了解了书中介绍的各种问题和解决方案，那么应该说已经掌握了一些调试技术了。本书介绍了几种重要调试工具的使用时机和方式，包括静态检查器、链接器、源代码调试器、内存调试器和剖析工具。更重要的是，你已经了解到，最强大的工具不在计算机中，而是调试者的判断力和分析技巧。

以下是为读者提供的一些建议。

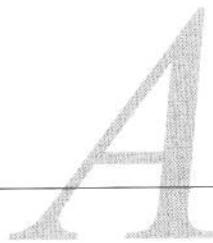
- 调试是一个多层面的问题，调试工作也不仅限于使用源代码调试器。
- 调试是软件开发人员的重要技能，是控制大型软件工程项目的先决条件。
- 调试是一个永恒的话题。随着软件和硬件复杂度的增加，以及重用性（必须使用别人编写的软件）和并行度的提高，问题也将随之增多。
- 过段时间后请重新温习本书。新的理解可能会改变你的想法，并为你提出新的挑战。

本书并未介绍所有调试技术。如果读者认为书中遗漏了重要内容，请发邮件至authors@debugging-guide.com。本书有一个勘误表供读者参考，网址为<http://www.debugging-guide.com>。如果书中有一些“bug”未在勘误表中纠正，也请发邮件给我们，不胜感激！

最后祝读者在bug狩猎游戏中好运！

附录 A

调试命令



表A-1 两种调试器的自检查功能一览

命令名称	GDB	Visual Studio
运行程序	run [args]	F5: Start Debugging (开始调试)
启动程序	start [args]	F10: Step over (逐过程)
暂停	Ctrl-C	Ctrl-Alt-Break: Break All (全部中断)
继续运行	cont	F5: Continue (继续)
step-over	next	F10: Step over (逐过程)
step-into	step	F11: Step into (逐语句)
step-out	finish	Shift + F11: Step out (跳出)
断点	break file:lineno	右击 Breakpoint/Insert Breakpoint (断点/插入断点)
跟踪点	watch file:lineno	右击 Breakpoint/Insert Tracepoint (断点/插入跟踪点)
观察点	watch expr	Debug/New Bkpt/New Data Breakpoint (调试/新建断点) /新建数据断点)
栈跟踪	bt, where	Call Stack (调用堆栈)
输出表达式	print expr	ImmediateWindow (即时窗口)
显示表达式	display expr	Watch (监视) 窗口
设置变量	set var var=expr	Variables (变量) 窗口
设置环境变量	set env var[=val]	Properties/Debugging/Environment (属性/调试/环境)
显示机器代码	Disassemble	右击 Go to Disassembly (转到反汇编)
在机器代码中执行step-over	nexti	F10: Step over (逐过程)
在机器代码中执行step-into	stepi	F11: Step into (逐语句)

表A-2 本书中使用的其他调试命令

命令名称	GDB	Visual Studio
条件断点	condition <i>bnum</i>	右击 Breakpoint/Condition (断点/条件)
事件断点	handle, signal	Debug/Exceptions... (调试/异常)
异常断点	catch, throw	Debug/New Bkpt/Break at Function (调试/新建断点/ 在函数处中断)
函数断点	break function	
临时断点	tbreak	F9: Debug/Toggle Breakpoint (调试/切换断点)
列出所有断点	info breakpoints	Breakpoints (断点) 窗口
将命令连接到断点	commands <i>bnum</i>	右击 Breakpoint/When Hit (断点/命中条件)
输出到命令行	printf	右击 Breakpoint/When Hit (断点/命中条件)
查找函数	info functions <i>expr</i>	Debug/New Bkpt/Break at Function
调用函数	call <i>expr</i>	(调试/新建断点/在函数处中断)
修改函数返回值	return <i>expr</i>	Immediate Window (即时窗口)
输出类型	whatis <i>arg</i>	右击 Go To Declaration (转到声明)
输出类型描述	ptype <i>arg</i>	右击 Go To Definition (转到定义)
输出内存内容	x <i>arg</i>	ImmediateWindow (即时窗口)
选择栈帧	frame <i>arg</i>	Call Stack 右击 Switch to Frame
输出帧描述	info frame	Call Stack (调用堆栈)

附录 B

工具资源



本附录列出了到哪里能够找到工具、文档、更多阅读资料和其他与调试有关的资料。由于有些信息经常更新和更改，因此会在本书的后续版本中修订这个附录。同时在我们的站点上发布此附录的最新版本，网址为<http://www.debugging-guide.com>。

B.1 IDE、编译器、构建工具

B.1.1 Microsoft Visual Studio

Visual Studio是微软公司的一个IDE。它是一个功能强大且被良好集成的环境，提供了适用于C++、Java和Visual Basic的编译器，此外还提供了源代码查看器和几种调试工具。在撰写本书时，有一个受限的编译器和IDE版本可供免费下载，它的名称是Visual C++ 2008 Express Edition。使用时请检查最新的限制许可协议。本书的所有示例都用Visual C++ 2005 SP1 (VC++8.0) 和Visual C++ 2008 (VC++9.0) 进行了测试。Visual Studio文档的名称是MSDN Library for Visual Studio，它是软件安装中的一部分。

微软的网站提供了有关Visual C++和Microsoft C++编译器的信息，进入主页后到Developer tools和Visual C++的栏目中查找：

<http://www.microsoft.com>

B.1.2 Eclipse

Eclipse是一个开源IDE，它支持Java、C++和其他语言。它可以在Linux和Windows上运行，并支持插件扩展。CDT是C/C++为Eclipse提供的IDE。

<http://www.eclipse.org>
<http://www.eclipse.org/cdt>
http://en.wikipedia.org/wiki/Java_eclipse

B.1.3 GCC

GCC (GNU Compiler Collection) 是一个高质量的编译器，它带有用于C++、C、Java和Fortran

的前端。由于GCC是开源的，因此几乎可用于任何CPU，而且非常模块化且便于移植。GCC是大多数Linux的一部分，也是Windows平台上Cygwin的一部分（参见附录B.3.1）。本书示例中使用的GCC版本为3.2.3和4.2。有关GCC软件、文档和源代码的信息可通过以下网址找到：

<http://gcc.gnu.org>

B.1.4 GNU Make

GNU Make（也称为gmake）是一个用于构建软件的应用程序。它是Linux和Cygwin的一部分。有关GNU Make的文档可通过以下网址找到：

<http://www.gnu.org/software/make>

B.2 调试器

B.2.1 dbx

dbx是一个用于C和C++的源代码调试器，它最初是作为Berkeley UNIX的一部分开发的，可以在Solaris和BSD Unix平台上使用。可以通过以下网址获得更多信息：

<http://developers.sun.com/sunstudio>
<http://docs.sun.com/app/docs/doc/819-5257>
http://en.wikipedia.org/wiki/Dbx_debugger

B.2.2 DDD

DDD是一个用于GDB和dbx的图形调试器前端。可以通过以下网址获得更多信息：

<http://www.gnu.org/software/ddd>
<http://en.wikipedia.org/wiki/Ddd>

B.2.3 GDB

GDB是GNU项目的源代码调试器。GDB手册已作为图书[Stallmann02]出版，并且提供了Web文档：

<http://sourceware.org/gdb>

B.2.4 ARM RealView

ARM RealView是一个用于ARM CPU的软件开发工具集，包括编译器、汇编器、链接器和调试器。调试器支持多核CPU、跟踪、剖析、CPU模拟，还可以将调试器连接到嵌入式系统板上。可以通过以下网址获得更多信息：

<http://www.arm.com/products/DevTools>
http://en.wikipedia.org/wiki/ARM_Ltd

B.2.5 TotalView Debugger

TotalView Debugger是一个商业C/C++调试器。它可以感知STL，并且具有良好的数据可视化能力，此外可以调试那些使用了线程、MPI或OpenMP的并行程序。有关TotalView的更多信息、文档、演示版和定价通过以下网址获得：

<http://www.totalviewtech.com>

B.2.6 Lauterbach TRACE32

Lauterbach TRACE32是一个用于嵌入式系统的源代码调试器。TRACE32支持ARM、MIPS、PowerPC和x86 CPU，以及各种DSP。可以通过以下网址获得更多信息：

<http://www.lauterbach.com>

B.3 环境

B.3.1 Cygwin

Cygwin是一个Linux风格的环境，它可以在Windows操作系统上使用。Cygwin提供了一个bash命令行解释器、几个编辑器（如vi和xemacs）以及很多UNIX和Linux风格的命令。用户可以选择安装GCC编译器（gmake）和调试工具GDB（gprof和Valgrind），因此很容易获得Linux风格的软件开发系统。更多信息、文档和下载说明可以通过以下网址获得：

<http://www.cygwin.com>

B.3.2 VMware

VMware Workstation是一个在主机工作站上提供虚拟机的软件包。它允许安装多个独立的Windows和Linux工作站，所有这些工作站都可以在一台主机上运行。在需要用多个编译器或操作系统版本对程序进行测试时，或者某一平台上仅能使用某个特定调试工具时，通常可以使用VMware。

VMware的检查点特性非常有用，利用此特性可以存储和恢复几个已知的稳定状态。此特性可有效地实现一个干净且可重复的调试环境。例如，如果软件修改或破坏了虚拟机，可以使用检查点特性，在软件维护时，也可以使用此特性保护正在使用的调试环境。有关VMware的更多信息、文档、演示版和定价通过以下网址获得：

<http://www.vmware.com>

B.4 内存调试器

B.4.1 Purify

Purify是一个商业内存调试工具，它可在Linux、Windows和Solaris平台上使用。Purify的工作方式是在链接阶段插装程序的对象代码。它不需要使用源代码，没有特殊的编译器标志，也无需重新编译对象文件。可以通过以下网址获得更多信息：

http://en.wikipedia.org/wiki/IBM_Rational_Purify
<http://www-306.ibm.com/software/awdtools/purifyplus>

B.4.2 Valgrind

Valgrind是一个开源软件。现在可以在Linux上使用，适用于x86和PowerPC处理器。Valgrind的使用模式很简单。Valgrind解释对象代码，因此不需要修改对象文件或可执行程序，从而不需要特殊的编译器标志，也不需要重新编译或重新链接程序。在执行程序之前，只需在命令行中输入valgrind命令。Valgrind最突出的优点是不需要源代码，因此可用来分析第三方的黑盒软件模块（这些模块的代码是保密的，因此无法访问）。

Valgrind有一个工具集：

- Memcheck：内存检查器
- Callgrind：规则剖析工具
- Cachegrind：缓存剖析工具
- Helgrind：查找竞争条件
- Massif：内存剖析工具

有关Valgrind的文档和下载说明可通过以下网址获得：

<http://valgrind.org>

B.4.3 KCachegrind

KCachegrind是一个图形前端，用于显示Valgrind/Callgrind的剖析结果，它是一个开源软件。KCachegrind显示剖析所生成的跟踪记录，包括树型图和调用图。有关KCachegrind的文档和下载说明可通过以下网址获得：

<http://sourceforge.net/projects/kcachegrind>

B.4.4 Insure++

Insure++是一个用于检测运行时内存错误的商业工具。Insure++使用源代码插装，即在将源代码传递给编译器之前对其进行动态修改。这种使用模式要求重新编译源文件。它还提供了一些

支持对象代码库的功能这些库的源代码不可用。可通过以下网址获得更多信息：

<http://www.parasoft.com>

B.4.5 BoundsChecker

BoundsChecker是一个商业内存检查工具，适用于在Windows平台上检查Visual C++。BoundsChecker有两种使用模式，即ActiveCheck和FinalCheck，前者监控对操作系统和内存管理例程的调用，后者在对象代码中加入插装代码，以检查缓冲区溢出和未初始化的内存读取。可通过以下网址获得更多信息：

<http://en.wikipedia.org/wiki/BoundsChecker>

<http://www.compuware.com/products/devpartner/visualc.htm>

B.5 剖析工具

B.5.1 gprof

gprof是一个开源剖析工具，通常作为GCC编译器的一部分。在下载和安装GCC编译器时，可能必须选择gprof作为一个额外的选项或包。有关gprof剖析工具的文档可通过以下网址获得：

<http://sourceware.org/binutils>

B.5.2 Quantify

Quantify是IBM出售的一个功能强大的商业剖析工具，它是IBM Rational软件质量工具家族的一部分。Quantify是Rational PurifyPlus工具套件中的一个工具。有关Quantify的信息可以在IBM的网站上找到，进入主页后搜索Software and Quality Products。

<http://www.ibm.com>

B.5.3 Intel VTune

Intel VTune是用于x86和x64处理器的性能分析工具。它可以在Windows和Linux上使用。更多信息、文档、演示版和定价可通过以下网址获得：

<http://www.intel.com>

B.5.4 AQtime

AQtime是AutomatedQA出售的一个商业工具。它是一个运行时和内存剖析工具，适用于Windows平台上的Microsoft、Borland、Intel、Compaq和GNU编译器。Microsoft Visual Studio和Borland Developer Studio中集成了AQtime。可通过以下网址获得更多信息：

<http://www.automatedqa.com/products/aqtime>

B.5.5 mpatrol

mpatrol是一个开源软件内存调试器，它也有内存剖析功能。它是一个链接到可执行程序中的库，拦截对malloc()、free()和类似函数的调用。其使用模式类似于gprof。可通过以下网址获得更多信息：

<http://sourceforge.net/projects/mpatrol>

B.6 静态检查器

B.6.1 Coverity

Coverity Prevent是Coverity公司出售的一个商业静态代码检查器，它基于美国斯坦福大学计算机系统实验室（Computer Systems Laboratory）开发的bug查找技术。更多信息、文档、下载说明和定价可通过以下网址获得：

<http://www.coverity.com>

B.6.2 Lint

lint是1979年出现的，当时它是UNIX操作系统的一部分。尽管它最初只能在C语言上工作，但现在lint已成了各种计算机调语言的代码检查的代名词。Solaris和Linux等一些操作系统发行版中都有lint。`man lint`命令可以输出它的使用和选项信息。它的主页地址为：

<http://web.gat.com/docview/lint.html>

手册可以通过以下网址找到：

<http://www.thinkage.ca/english/gcos/expl/lint/manu/manu.html>

教程可以通过以下网址获得：

<http://www.pdc.kth.se/training/Tutor/Basics/lint>

B.6.3 Splint

Splint（Secure Programming Lint）是一个静态检查器。它基于最初的UNIX lint，并扩展了更多检查功能和源代码注释功能。Cygwin包中也提供了Splint。它的文档、源代码和二进制库可到以下网址下载：

<http://www.splint.org>

B.6.4 Visual Studio 企业版中的/analyze 选项

Visual Studio企业版（团队开发版本）的价格更高，此版本支持/analyze静态代码分析选项，此选项已被集成到Visual Studio IDE中。

B.6.5 Klocwork

Klocwork是一个商业静态分析工具，用于解决Web安全、软件架构问题，以及修复常规的代码缺陷。更多信息、文档、定价和试用版可通过以下网址获得：

<http://www.klocwork.com>

B.6.6 Fortify

Fortify Source Code Analysis Suite是一个商业静态分析工具，主要用于查找Web应用程序中的安全漏洞，例如缓冲区溢出。可通过以下网址获得更多信息：

<http://www.fortifysoftware.com>

B.6.7 PC-lint/FlexeLint

PC-lint（在非Windows平台上也称为FlexeLint）是一个商业静态分析工具，用于检查C和C++代码。此软件是Gimpel Software公司出售的产品。可通过以下网址获得更多信息：

<http://www.gimpel.com>

B.6.8 QA C++

QA C++是一个商业静态分析工具，用于检查C、C++及其他语言代码中的规则冲突。它是Programming Research公司出售的产品。可通过以下网址获得更多信息：

<http://www.programmingresearch.com>

B.6.9 Codecheck

Codecheck是一个商业静态分析工具，主要用于检查C和C++代码的规则冲突。它是Abraxas软件公司出售的产品。可通过以下网址获得更多信息：

<http://www.abraxas-software.com>

B.6.10 Axivion Bauhaus Suite

Axivion Bauhaus Suite是一个商业静态分析工具，它分析源代码的体系结构。它的特性包括克隆检测（那些被复制并稍加修改的源代码）和接口分析（不同模块如何相互通信）。它是Axivion

GmbH公司出售的产品。可通过以下网址获得更多信息：

<http://www.axivion.com>

B.6.11 C++ SoftBench CodeAdvisor

CodeAdvisor是一个商业静态分析工具，用于检查C/C++代码的规则冲突。它是惠普公司的SoftBench工具套件的一部分。有关更多信息，请访问惠普网站，搜索CodeAdvisor。

<http://www.hp.com>

B.6.12 Parasoft C++test

C++test是Parasoft公司出售的一个商业工具，用于测试C/C++代码。它是一个静态分析工具，提供了规则检查功能，也可以自动创建测试工具（test harness）。可通过以下网址获得更多信息：

<http://www.parasoft.com>

B.6.13 LDRA 工具套件

LDRA工具套件是一个商业静态分析工具，提供规则检查和其他代码分析功能。它是LDRA公司出售的产品。可通过以下网址获得更多信息：

<http://www.ldra.com>

B.6.14 Understand C++

Understand C++是一个静态代码分析工具。它的主要功能是进行反向工程和文档化源代码。它是Scientific Toolworks公司出售的产品。可通过以下网址获得更多信息：

<http://www.scitools.com>

B.7 并行编程工具

B.7.1 Posix Threads

Posix Threads是Linux和Solaris平台上的线程标准，它也有一个Windows实现。更多信息参见[Butenhof97]，也可以通过以下网址获得：

http://en.wikipedia.org/wiki/Posix_threads
<https://computing.llnl.gov/tutorials/pthreads>

B.7.2 OpenMP

OpenMP API支持使用C、C++和Fortran语言做共享内存的并行编程，适用于Linux、Windows、

Solaris和MacOS平台。更多信息参见[Chandra00]、[Chapman07]、[Eigenmann01]，也可通过以下网址获得：

<http://www.openmp.org>
<http://en.wikipedia.org/wiki/Openmp>

B.7.3 Intel TBB

Intel TBB（Threading Building Blocks）是一个基于模板的C++类库，它在原始操作系统线程周围提供了抽象层。TBB有一个带有内存分配器的运行库，因此是线程安全的，并且避免了缓存冲突（cache conflict）。它可以在x86（Pentium 4及更高）和兼容的处理器上运行，支持Linux、Windows和MacOS平台。TBB还有一个开源版本。更多信息参见以下网址[Reinders07]，也可通过以下网址获得：

<http://www.intel.com>
<http://threadingbuildingblocks.org>
<http://en.wikipedia.org/wiki/TBB>

B.7.4 MPI

MPI（Message Passing Interface）是一个消息传递接口，适用于大型计算机集群的编程。MPI与语言无关，为C、C++、Fortran、Java、Perl和Python等语言提供了实现。可通过以下网址获得更多信息：

<http://www-unix.mcs.anl.gov/mpi>
<http://www.mpi-forum.org>
http://en.wikipedia.org/wiki/Message_Passing_Interface

B.7.5 MapReduce

Google MapReduce是一个并行计算框架，它将计算分布在相隔很远的具有不可靠节点的集群上。可通过以下网址获得更多信息：

<http://labs.google.com/papers/mapreduce.html>
<http://en.wikipedia.org/wiki/MapReduce>

B.7.6 Intel 线程分析工具

Intel Thread Checker是一个用于检测死锁和数据竞争的工具。它可以将潜在错误定位到内存引用和源代码行。其结果被分为不同的安全级别，如注释、警告和错误。Intel Thread Checker可在Linux和Windows平台上运行。

Intel Thread Profiler是一个用于测量和显示并行线程执行情况的工具。它与OpenMP和TBB兼容，目前只能在Windows上使用。更多信息、文档、演示版和定价可通过以下网址获得：

<http://www.intel.com>

B.8 其他工具

B.8.1 GNU binutils

GNU binutils是一个二进制工具集，适用于UNIX、Linux和Cygwin平台。一些有用的调试命令是：`nm`（列出来自对象文件的符号）、`objdump`（显示对象文件信息）、`strings`（列出二进制文件中可输出的字符串）和`strip`（删除来自对象文件的符号）。更多信息参见以下网址：

<http://www.gnu.org/software/binutils>

B.8.2 M4

GNU M4宏处理程序是一个开源软件。有关文档和下载资源可通过以下网址找到：

<http://www.gnu.org/software/m4>

B.8.3 ps

`ps`是一个UNIX、Linux和Cygwin实用程序，它显示主机上运行的进程的当前状态。通过命令行参数可以控制显示不同类型的信息。注意，在不同的操作系统上，命令行参数也不同，使用`man ps`命令可以列出更多信息。

B.8.4 strace/truss

Linux实用程序`strace`（在Solaris上是`truss`）记录对操作系统的所有访问，例如内存分配、文件I/O、系统调用和子进程的启动。在Linux上使用`man strace`或在Solaris上使用`man truss`可以列出更多信息。

B.8.5 top

`top`是一个UNIX、Linux和Cygwin实用程序，它以简单的图形形式显示主机上正在运行哪些进程，并给出内存使用、CPU时间消耗、优先级等详细信息。它还显示主机状态的汇总，例如根据“用户/内核/空闲”时间列出总的内存使用和总体CPU时间使用。利用这个实用程序可以快速了解主机的使用情况。使用`man top`命令可以列出更多信息。

B.8.6 VNC

VNC是一个远程控制软件，可以通过网络查看并控制计算机的桌面。更多信息、文档和软件下载可以通过以下网址找到：

<http://www.realvnc.com>
<http://www.tightvnc.com>

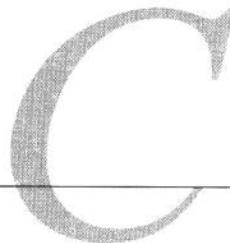
B.8.7 WebEx

WebEx是一个远程会议软件，利用它可以查看和共享远程桌面的应用程序。可以通过以下网址获得更多信息：

<http://www.webex.com>
<http://en.wikipedia.org/wiki/Webex>

附录 C

源 代 码



C.1 testmalloc.c

第5章使用这个示例来检查free()/delete是否将未分配的内存返回给操作系统。

```
1  /* testmalloc.c Copyright 2007 Groetker, Holtmann, Keding, Wloka */
2  #include <stdio.h>
3  #include <stdlib.h>
4  #ifdef _MSC_VER
5  #define sleep(x) _sleep(1000*(x))
6  #endif
7
8  #define blocksize 1024
9
10 /* make the program wait, to inspect process for memory use */
11 void wait_for_input(const char *prefix, int is_interactive) {
12     char c;
13     if(is_interactive) {
14         printf("%s hit return to continue\n", prefix); fflush(stdout);
15         c = getchar();
16     }
17     else
18     {   sleep(1); }
19 }
20
21 /* program entry point */
22 int main(int argc, char **argv) {
23     const char *usage = "usage: testmalloc i[interactive]\n n iter\n";
24     int n, i, j, iterations, is_interactive = 0;
25     int **myarray;
26
27     if(argc != 4) {
28         fprintf(stderr, usage);
29         return 1;
30     }
31
32     if(argv[1][0] == 'i')
33         is_interactive = 1;
34
35     n = atoi(argv[2]);
36     iterations = atoi(argv[3]);
```

```

37     if(n <= 0 || iterations < 0) {
38         fprintf(stderr, usage);
39     return 2;
40     }
41
42     for(i=0; i<iterations; i++) {
43         wait_for_input("before malloc: ", is_interactive);
44 #ifdef USE_NEW
45         myarray = new int*[n];
46 #else
47         myarray = (int **) malloc(n * (sizeof(int *)));
48 #endif
49         for(j=0; j<n; j++) {
50 #ifdef USE_NEW
51             myarray[j] = new int[blocksize];
52 #else
53             myarray[j] = (int *) malloc(blocksize * sizeof(int));
54 #endif
55         }
56         wait_for_input("after malloc: ", is_interactive);
57         for(j=0; j<n; j++) {
58 #ifdef USE_NEW
59             delete [] myarray[j];
60 #else
61             free(myarray[j]);
62 #endif
63         }
64
65 #ifdef USE_NEW
66     delete [] myarray;
67 #else
68     free(myarray);
69 #endif
70     }
71     return 0;
72 }

```

C.2 genindex.c

第5章使用这个示例来演示如何测量数据结构上的内存消耗。

```

1  /* genindex.cc Copyright 2007 Groetker, Holtmann, Keding, Wloka */
2  #include <stdio.h>
3  #include <string>
4  #include <list>
5  #include <vector>
6  #include <map>
7  using namespace std;
8
9  /* make the program wait, to inspect process for memory use */
10 void wait_for_input(const char *prefix) {
11     char c;
12     fprintf(stderr, "%s hit return to continue\n", prefix);
13     fflush(stderr);
14     c = getchar();

```

```
15 }
16 // word index data structure: word as key, list of integers
17 // each integer stores the line number where the word occurs
18 typedef map<string,list<int>,less<string> > WordIndexType;
19
20 // wrapper class, stores one index per text file
21 class FileIndexType
22 {
23 public:
24     FileIndexType();
25     ~FileIndexType();
26     int scan_file(char *fname);
27     int add_to_index(string &key, int l);
28     void print_index();
29     int verify_index();
30     void clear();
31     void clear_lines();
32     int print_memory_stats();
33 protected:
34     string filename;
35     int filesize;
36     WordIndexType wordindex;
37     vector<string> lines;
38 };
39
40 // constructor
41 FileIndexType::FileIndexType()
42 {}
43
44 // destructor
45 FileIndexType::~FileIndexType() {
46     clear();
47 }
48
49 // clear the index
50 void FileIndexType::clear() {
51     filename.clear();
52     filesize = 0;
53     wordindex.clear();
54     clear_lines();
55 }
56
57 // clear the lines buffer
58 void FileIndexType::clear_lines() {
59     lines.clear();
60 }
61
62 // generate a report of memory usage
63 int FileIndexType::print_memory_stats() {
64     unsigned i;
65     int mem_filename = sizeof(string) + filename.size();
66     int mem_wordindex = 0;
67     int mem_lines = 0;
68     int mem_total = 0;
69
70 // compute size of wordindex data structure
```

```

71 // Note: very rough approximation, measures the payload, not
72 // the internal search structure of map.
73     WordIndexType::const_iterator it;
74     list<int>::const_iterator wt;
75     for(it = wordindex.begin(); it != wordindex.end(); it++) {
76         mem_wordindex += it->first.size();      // add size of word key
77         for(wt = it->second.begin(); wt != it->second.end(); wt++)
78             // double-linked list element size is at least 2 pointers plus content
79             mem_wordindex += sizeof(int) + 2 * sizeof(void*);
80     }
81
82 // compute size of lines data structure payload
83     for (i=0; i < lines.size(); i++)
84         mem_lines += lines[i].size();
85
86     mem_total = mem_filename + mem_wordindex + mem_lines;
87
88     fprintf(stderr, "-- memory size for index of '%s' file size=%d\n",
89             filename.c_str(), filesize);
90     fprintf(stderr, "-- filename=%d wordindex=%d lines=%d total=%d\n",
91             mem_filename, mem_wordindex, mem_lines, mem_total);
92     fflush(stderr);
93     return mem_total;
94 }
95
96 // add a (word, line) pair to the index
97 int FileIndexType::add_to_index(string &key, int l) {
98     if(key.size() == 0)
99         return 0;
100    wordindex[key].push_back(l);
101    return 0;
102 }
103
104 // open file, break text into words, add words to index, close file
105 int FileIndexType::scan_file(char *fname) {
106     filename = fname;
107     filesize = 0;
108     FILE *fp = 0;
109     int c = 0;
110     string newword = "";
111     int current_line = 1; /* start counting lines at value 1 */
112     string buffer;
113
114     if(NULL == (fp = fopen(filename.c_str(), "r"))) {
115         fprintf(stderr, "-- error: cannot read file '%s'\n",
116                 filename.c_str());
117         return 1;
118     }
119     while(1) { // very simple tokenizer to break text into words
120         c = getc(fp);
121
122         if(c == EOF || c == '\n') {
123             add_to_index(newword, current_line);
124             newword = "";
125             current_line++;
126             lines.push_back(buffer);

```

```
127 #ifdef FIX_LINES
128         buffer = "";
129 #endif
130         if(c == EOF)
131             break;
132     }
133     else if(c == ' ' || c == '\t' || c == '\r') {
134         add_to_index(newword, current_line);
135         newword = "";
136         if(c != '\r')
137             buffer = buffer + (char) c;
138     }
139     else {
140         newword = newword + (char) c;
141         buffer = buffer + (char) c;
142     }
143     filesize++;
144 }
145 fclose(fp);
146 return 0;
147 }

148 // output of the program: a word index
149 void FileIndexType::print_index() {
150     WordIndexType::const_iterator it;
151     list<int>::const_iterator wt;
152     printf("index of file '%s'\n", filename.c_str());
153     for(it = wordindex.begin(); it != wordindex.end(); it++) {
154         printf("%s", it->first.c_str());
155         for(wt = it->second.begin(); wt != it->second.end(); wt++)
156             printf(" %d", (*wt));
157         printf("\n");
158     }
159 }
160 fflush(stdout);
161 }

162 // verification code: cross-check generated index
163 int FileIndexType::verify_index() {
164     int result = 0;
165     WordIndexType::const_iterator it;
166     list<int>::const_iterator wt;
167     string w;
168     int i;
169     for(it = wordindex.begin(); it != wordindex.end(); it++)
170     {
171         w = it->first;
172         for(wt = it->second.begin(); wt != it->second.end(); wt++) {
173             i = (*wt);
174             if(string::npos == lines[i-1].find(w))
175                 return 1;
176         }
177     }
178 }
179 return result;
180 }
181
182 /* program entry point */
```

```

183 int main(int argc, char **argv) {
184     const char *usage = "-- usage: genindex filename [filename...]\n";
185     if(argc < 2) {
186         fprintf(stderr, usage);
187         return 1;
188     }
189     vector<FileIndexType> fileindex(argc-1);
190     int result = 0;
191     int total = 0;
192     int i;
193
194     // for each file, compute an index
195     for(i = 0; i<argc-1; i++) {
196         result = fileindex[i].scan_file(argv[i+1]);
197         if(result)
198             return result; // something went wrong with file read
199         result = fileindex[i].verify_index();
200         if(result)
201             fprintf(stderr, "-- error: index verification failed.\n");
202         return result;
203     }
204 #ifdef CLEAR_INDEX
205     fileindex[i].print_index();
206     fileindex[i].clear();
207     total += fileindex[i].print_memory_stats();
208 #endif
209 #ifdef PAUSE_INDEX
210     wait_for_input("-- done generating index: ");
211 #endif
212 }
213
214 #ifndef CLEAR_INDEX
215     // for each file, output index
216     for(i=0; i<argc-1; i++) {
217         fileindex[i].print_index();
218         total += fileindex[i].print_memory_stats();
219     }
220 #endif
221     fprintf(stderr, "-- memory size, all data structures: %d bytes\n",
222             total);
223     return result;
224 }
```

C.3 isort.c

第6章使用这个示例进行一般剖析。

```

1 /* isort.c Copyright 2007 Groetker, Holtmann, Keding, Wloka */
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 typedef double Stype;
6
7 /* Print an array of the given size on stdout. */
8 void print_array(const char* prefix, const Stype *array, int size) {
```

```
9     int i;
10    printf("%s:", prefix);
11    for(i=0; i<size; i++)
12        printf(" %f", array[i]);
13    printf("\n"); fflush(stdout);
14 }
15
16 /* swap 2 elements */
17 void swap(Stype *a, int i, int j) {
18     Stype tmp;
19     tmp = a[i];
20     a[i] = a[j];
21     a[j] = tmp;
22 }
23
24 /* check if 'a' is less than 'b' */
25 int less(Stype a, Stype b) {
26     return (a < b) ? 1 : 0;
27 }
28
29 /* insert a[0] into pre-sorted array a[1]...a[n-1] */
30 void insert_value(Stype *a, int n) {
31     int i;
32     for(i=1; i<n; i++)
33         if(less(a[i], a[i-1]))
34             swap(a, i, i-1);
35 #ifdef ISORT_FAST /* compile with -DISORT_FAST to speed up */
36     else return;
37 #endif
38 }
39
40 /* toplevel routine for isort */
41 void isort(Stype *a, int n) {
42     if(n <= 1)
43         return;
44     isort(a+1, n-1);
45     insert_value(a,n);
46 }
47
48 /* partition array a, so that all values smaller than pivot a[n-1]
49  are placed in a[0]..a[result-1], the pivot is in a[result],
50  and the values larger than the pivot are in a[result+1]..a[n-1]
51 */
52 int partition(Stype *a, int n) {
53     int i, result = 0;
54     for(i = 0; i<n-1; i++) {
55         if(less(a[i], a[n-1])) {
56             swap(a, result, i);
57             result++;
58         }
59         swap(a, result, n-1);
60 #ifdef DEBUG
61         print_array("partition:", a, n);
62         printf("pivot at: %d\n", result);
63 #endif
64     return result;
```

```
65 }
66
67 /* toplevel routine for quicksort */
68 void quicksort(Stype *a, int n) {
69     int i;
70     if(n <= 1)
71         return;
72     i = partition(a, n);
73     quicksort(a, i);
74     quicksort(a+i+1, n-(i+1));
75 }
76
77 /* program entry point */
78 int main(int argc, char **argv) {
79     const char *usage = "usage: isort i|q n iter, n>0, iter>0\n";
80     int n, i, j, iterations, use_isort = 1;
81     Stype *input, *result;
82
83     if(argc != 4) {
84         fprintf(stderr, usage);
85         return 1;
86     }
87
88     if(argv[1][0] == 'q')
89         use_isort = 0;
90
91     n = atoi(argv[2]);
92     iterations = atoi(argv[3]);
93     if(n <= 0 || iterations <= 0) {
94         fprintf(stderr, usage);
95         return 2;
96     }
97
98     input = (Stype *) malloc(n * sizeof(Stype));
99     result = (Stype *) malloc(n * sizeof(Stype));
100    if(input == 0 || result == 0) {
101        fprintf(stderr, "out of memory\n");
102        return 3;
103    }
104
105    srand48(1); /* always generate the same random sequence */
106    for(i=0; i<n; i++)
107        input[i] = drand48();
108
109    for(j=0; j<iterations; j++) {
110        for(i=0; i<n; i++)
111            result[i] = input[i];
112        if(use_isort)
113            isort(result, n);
114        else
115            quicksort(result, n);
116    }
117 #ifdef DEBUG
118     print_array("input", input, n);
119     print_array("result", result, n);
120 #endif
```

```

121     free(input);
122     free(result);
123     return 0;
124 }
```

C.4 filebug.c

第6章使用这个示例进行I/O问题剖析。

```

1  /* filebug.c Copyright 2007 Groetker, Holtmann, Keding, Wloka */
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  /* program entry point */
6  int main(int argc, char **argv) {
7      int i, n, use_flush = 1;
8      const char *usage = "usage: filebug f[ast]|s[low] file n\n";
9      char *filename;
10     FILE *fp;
11     char c;
12
13     if(argc != 4) {
14         fprintf(stderr, usage);
15         return 1;
16     }
17
18     if(argv[1][0] == 'f')
19         use_flush = 0;
20
21     filename = argv[2];
22
23     n = atoi(argv[3]);
24     if(n < 0) {
25         fprintf(stderr, usage);
26         return 2;
27     }
28
29     if(!(fp = fopen(filename, "w"))) {
30         fprintf(stderr, "can not open file '%s' for write\n", filename);
31         return 3;
32     }
33
34     /* write n characters to file, to observe effect of fflush() */
35     for(i=0; i<n; i++) {
36         c = 'a' + (i % 26);
37         fputc(c, fp);
38         if(use_flush)
39             fflush(fp);
40     }
41     fclose(fp);
42     return 0;
43 }
```


参 考 文 献

- [Agans02] D.J. Agans, *Debugging: The Nine Indispensable Rules for Finding Even the Most Elusive Software and Hardware Problems*. American Management Association, 2002
- [Ball98] S. Ball, *Debugging Embedded Microprocessor Systems*. Newnes, 1998
- [Barr06] M. Barr, A. Massa, *Programming Embedded Systems*. O'Reilly, 2nd Edition, 2006
- [Brown88] M.H. Brown, *Algorithm Animation*. The MIT Press, 1988
- [Butenhof97] D.R. Butenhof, *Programming with POSIX Threads*. Addison-Wesley, 1997
- [Chandra00] R. Chandra, R. Menon, L. Dagum, D. Kohr, D. Maydan, J. McDonald, *Parallel Programming in OpenMP*. Morgan Kaufmann, 2000
- [Chapman07] B. Chapman, G. Jost, R. van der Pas, *Using OpenMP: Portable Shared Memory Parallel Programming*. The MIT Press, 2007
- [Cormen01] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, *Introduction to Algorithms*. The MIT Press, 2nd Edition, 2001
- [Eigenmann01] R. Eigenmann, M. Voss (Editors), *OpenMP Shared Memory Parallel Programming*. International Workshop on OpenMP Applications and Tools, WOMPAT 2001. (Lecture Notes in Computer Science). Springer, 2001
- [Ford02] A.R. Ford, T.J. Teorey, *Practical Debugging in C++*. Prentice Hall, 2002
- [Fritzson93] P.A. Fritzson (Editor), *Automated and Algorithmic Debugging*. First International Workshop, Aadebug '93 Linkoping, Sweden, May 3–5, 1993: Proceedings (Lecture Notes in Computer Science). Springer, 1993
- [Kaspersky05] K. Kaspersky, *Hacker Debugging Uncovered*. A-List Publishing, 2005
- [Lencevicius00] R. Lencevicius, *Advanced Debugging Methods*. Springer, 2000
- [Levine00] J. Levine, *Linkers & Loaders*. Morgan Kaufmann, 2000
- [Luecke06] G.R. Luecke, J. Coyle, J. Hoekstra, M. Kraeva, Y. Li, O. Taborskaia, Y. Wang, *A Survey of Systems for Detecting Serial Run-Time Errors*. Concurr. Comput. : Pract. Exper. 18(15): 1885–1907, Dec. 2006

- [Metzger03] R.C. Metzger, *Debugging by Thinking: A Multidisciplinary Approach*. Digital Press, 2003
- [Meyers04] G.J. Meyers, C. Sandler, T. Badgett, T.M. Thomas, *The Art of Software Testing*. John Wiley & Sons, 2004
- [Pappas00] C.H. Pappas, W.H. Murray, *Debugging C++*. Osborne Publishing, 2000
- [Reinders07] J. Reinders, *Intel Threading Building Blocks: outfitting C++ for Multi-core Processor Parallelism*. O'Reilly Media, 2007
- [Rosenberg96] J.B. Rosenberg, *How Debuggers Work: Algorithms, Data Structures, and Architecture*. Wiley, 1996
- [Sedgewick01] R. Sedgewick, *Bundle of Algorithms in C++, Parts 1–5: Fundamentals, Data Structures, Sorting, Searching, and Graph Algorithms*. Addison-Wesley Professional, 3rd Edition, 2001
- [Silberschatz04] A. Silberschatz, G. Gagne, P.B. Galvin, *Operating System Concepts*. John Wiley & Sons Inc, 2004
- [Stitt92] M. Stitt, *Debugging: Creative Techniques and Tools for Software Repair*. John Wiley & Sons Inc, 1992
- [Stallmann02] R.M. Stallmann, R.H. Pesch, S. Shebs, *Debugging With GDB: The Gnu SourceLevel Debugger*. Free Software Foundation, 2002
- [Tanenbaum01] Andrew S. Tanenbaum, *Modern Operating Systems*. Prentice Hall PTR, 2001
- [Telles01] M.A. Telles, Y. Hsieh, *The Science of Debugging*. Coriolis Group Books, 2001
- [Zeller05] A. Zeller, *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann, 2005