

程序调试 思想与实践

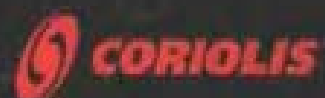
The Science of Debugging

[美] Matt Telles, Yuan Hsieh 著

邓劭生 等译



中国水利水电出版社
www.waterpub.com.cn



责任编辑：晓渊工作室

程序调试 思想与实践

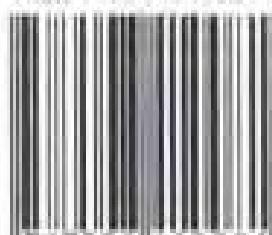
教你如何调试作为一门可反复使用的学问

“在这个世界上，除了死亡和税收之外没有确定的东西。”软件的bug也是如此。有效的调试不仅仅包括用调试器浏览代码。本书认识到了这一点，并且教你如何可靠地找到并修复bug。其出发点是通过提供你在迅速地识别、跟踪和修复bug时所需的知识和技术，使得调试更加容易、快捷和有效。然后又给出了关于如何将bug极小化和使它们出现之后更加容易发现的实际经验。要虚心学习使用那些有经验的人十几年来积累的技术！

本书的对象

专业的软件开发人员，软件管理人员，软件测试人员，专业调度员。

ISBN 7-5084-1008-4



9 787508 410081 >

本书将会帮助你

- 开发科学的、应用于查找和删除软件 bug 的方法
- 理解著名的 bug 以及它们是如何被发现和排除的
- 学习不同的使 bug 易于重现、提高效率、以及其他的技术
- 将有害的 bug 与危险的 bug 区别开来
- 用以 bug 为中心的观点来进行软件开发
- 理解 bug 的生命周期以及它们是如何从测试中溜走的
- 学习 bug 预防和测试技术
- 理解不同的 bug，包括它们的最一般环境、症状和例子
- 学习深入调试的方法和专业技术
- 成为一名专业调试员

本书给你带来什么

- 这本书用以 bug 为中心的观点来进入软件开发世界
- 明白 bug 为什么会出现以及在软件开发过程中如何做才能避免出现重复的 bug
- 学习正确进行调试的方法



北京万水电子信息有限公司

Beijing Wan Shui Electronic Information Co., Ltd.

地址：北京市西直门外榆树馆一世永康商务写字楼
邮编：100044

电话：(010)6835.9286, 6835.9167

传真：(010)6835.9284

E-mail: twchannel@public3.bta.net.cn

ISBN 7-5084-1008-4/ TP·390

定价：40.00 元

程序调试思想与实践

[美] Matt Telles, Yuan Hsieh 著

邓劲生 等译

中国水利水电出版社

内 容 提 要

本书将调试作为一门专业的学科进行研究和分析, 提供大量的代码实例和问题描述, 对调试的各个方面进行细致而深入的阐述和讨论。全书以 bug 为中心, 围绕调试这一主题进行组织。第 2 章到第 5 章包括 bug 的诊断、分类以及它们的症状。第 6 章到第 10 章讨论那些处理 bug 的策略, 包括可以使用的工具以及在不同情况下如何最有效地使用它们。第 11 章到第 16 章包括对专业调试的解释以及如何成为一个出色的调试员。

本书主要面向的读者群是软件项目的开发人员、调试人员、测试人员以及管理人员。

Original English language edition published by The Coriolis Group LLC, 14455 N. Hayden Drive, Suite 220, Scottsdale, Arizona 85260 USA, telephone (480) 483-0192, fax (480) 483-0193. Copyright © 2001 by The Coriolis Group. Simplified Chinese Language edition copyright © 2002 by China WaterPower Press. All rights reserved.

北京市版权局著作权合同登记号: 图字 01-2001-2599

图书在版编目 (CIP) 数据

程序调试思想与实践/ (美) 特列斯 (Telles, M.), (美) 元赫 (Yuan Hsieh) 著; 邓劲生等译. —北京: 中国水利水电出版社, 2002

书名原文: The Science of Debugging

ISBN 7-5084-1008-4

I. 程… II. ①特… ②元… ③邓… III. 程序设计—研究 IV. TP311

中国版本图书馆 CIP 数据核字 (2002) 第 014414 号

书 名	程序调试思想与实践
作 者	[美] Matt Telles, Yuan Hsieh 著
译 者	邓劲生 等译
出版、发行	中国水利水电出版社 (北京市三里河路 6 号 100044) 网址: www.waterpub.com.cn E-mail: mchannel@public3.bta.net.cn (万水) sale@waterpub.com.cn 电话: (010) 68359286 (万水)、63202266 (总机)、68331835 (发行部) 全国各地新华书店
经 售	北京万水电子信息有限公司
排 版	北京市天竺颖华印刷厂
印 刷	787×1092 毫米 16 开本 22.75 印张 491 千字
规 格	2002 年 3 月第一版 2002 年 3 月北京第一次印刷
版 次	0001—5000 册
印 数	40.00 元
定 价	

凡购买我社图书, 如有缺页、倒页、脱页的, 本社发行部负责调换
版权所有·侵权必究

译者序

自从有了计算机以来，甚至更早的时候，bug 和 debug 的斗争就一直纠缠在每个编程人员的心头。但是过去 30 年以来程序员们并没有真正做出什么改变。那些在软件工业发展的初期就存在的问题，现在仍然存在。软件的 bug 和在早期一样，仍然在当今的软件中出现，但是数量已经大不一样了。过去 bug 只出现在独立的单个用户或者一组用户之中，现在它们却出现在整个世界。那些过去仅需要手工重新制作报告并重新输入数据的问题，而现在可能意味着一个组织的生存。

尽管调试在软件的各个领域都极其重要，但它仍然被看作软件工程师的例行工作，这种状况越来越明显地需要改变。调试必须和设计 and 需求分析一样进行考虑：就是说，作为它本身的工作。用特殊的态度来考虑如何编写系统的设计，并不起多大的作用；但是如果用特殊的态度来考虑如何维护和修正系统的代码，则不然。

在目前的各种软件开发书籍中，大量的篇幅被用于讨论关于设计软件、编写代码和配置系统的更好方式等内容，通常到最后才考虑调试。但是它是软件开发生命过程中的一个自然的部分，是不容忽视地存在于我们每个软件项目中的，并且关系到整个项目的成功与否。调试和测试介入项目的时间越早，最后成品软件的质量就越高。

本书的作者都在软件领域工作多年，有着丰富的工程实践经验。本书将调试作为一门专业的学科进行研究和分析，提供大量的代码实例和问题描述，对调试的各个方面进行细致而深入的阐述和讨论，系统而有条理地解决了许多我们长期思考但仍然迷惑不解的问题。阅读本书，就像一名长者正在娓娓道来，真有一种如沐春风的感觉。

全书以 bug 为中心，内容围绕调试这一主题进行组织，大致可以划分为三个部分。第 2 章到第 5 章包括 bug 诊断、分类以及它们的症状。第 6 章到第 10 章讨论那些处理 bug 的策略，包括可以使用的工具以及在不同情况下如何最有效地使用它们。第 11 章到第 16 章包括对专业调试的解释以及如何成为一个出色的调试员。每章的最后都有一个 bug 问题留待读者思考，并将解答放在附录中。

本书主要面向的读者群是软件项目的开发人员、调试人员、测试人员以及管理人员。本书由邓劲生、刘锋、陈永然、陶阳等完成主要章节的翻译工作，参加录入、校对和排版的其他人员还有王江书、刘培英、何易非、任芳、金姜等，全文由邓劲生统校。作为译者，我们逐字逐句地细细研读本书，感觉从中受益匪浅。但是由于我们水平有限，书中难免出现一些不当甚至错误之处，恳请读者批评指正。

译者

2001 年 10 月

作者简介

Matt Telles 是软件开发界一名有 15 年经验的老手。他经历过 FORTRAN、COBOL 和其他奇怪的语言。现在是 Microsoft 公司的高级工程师。他的时间主要花在查找和修复别人产生的 bug 上。除此之外，他还喜欢将他掌握的技术教给其他开发人员。拥有了对应用程序进行编程、设计、做文档和调试的专门技术之后，他已经到达了当今程序员的巅峰：能够为专著写自己的简历。Matt 还出版过其他 5 本书，他居住在 Colorado 州的 Lakewood，正在忙着玩他心爱的 DEC 10。

Yuan Hsieh 还记得那些能够在互联网上阅读每个新闻组的日子。他花了 12 年来制造 bug。同样也花了 12 年修复这些 bug。他的修复工作非常好。目前他是 InfoNow 公司的软件开发主管。他不再制造和修复 bug 了，而是看别人制造和修复 bug。现在他把大部分时间花在如何将开发人员集中而作为一个统一而高效的编程团体，以能够及时在预算内产生无 bug 的软件。他现在住在 Colorado 州的 Lone Tree，将大部分闲暇时间用来在山上游玩。

致 谢

我要感谢我的雇主——Microsoft 公司给我时间和空间来完成本书。而且我要感谢我的妻子 Dawwna 和三个孩子能容忍来自我的办公室的激昂的讨论。最后，任何书的完成都离不开幕后人员的工作。Tom Lamoureux、Kevin Weeks、Peggy Cantrell、Jodi Winkler、Quentin Correll、Bob Arnson 和 Anne Marie Walker 不知疲倦地将我的空泛的文字转换成像样的英文。

——Matt

有人说一个艺术家通常在思维被打乱的情况下才发挥得最好。虽然这是一本技术类的书，也需要一定的思维和灵感。像我这样一个学者，没有比经历约定的典礼更能打乱思维了。这样，我就要感谢所有的过去和现在的出现在我生活中的异性，谢谢那些焦虑、灵感和不眠的夜晚（这时是写作的最好时候、独自在黑暗中发泄、涌现有哲理的词句）。

严肃地讲，我必须要感谢一些人。感谢 Kim Wright 帮我校对一些早期的章节。感谢 Stephen Shiller 爽快地为本书提供了令人惊讶的例子。感谢在 Coriolis 的人们：Kevin Weeks、Tom Lamoureux、Peggy Cantrell、Jodi Winkler 和技术参考人员 Quentin Correll，以及副本编辑者 Anne Marie Walker，谢谢他们的帮助。而且有许多与我一同工作的人员以及我所工作过的公司和地方。这些经历是无价的。

最后，感谢 Matt 为我提供了与他一起编写此书的机会。

——Y'

目 录

译者序

作者简介

致谢

第 1 章	Debug 简介	1
1.1	本书的内容	2
1.2	为什么要关注 bug	3
1.3	什么是 bug	4
1.4	本书的对象	4
1.5	本书的组织	5
1.6	调试简史	6
1.7	小结	8
1.8	bug 问题	8
第 2 章	研究著名的（以及不太著名的）bug 例子	10
2.1	大致情况	11
2.1.1	现实生活中的分布式系统	11
2.1.2	Therac-25	15
2.2	bug 实例 #1	19
2.3	bug 实例 #2	22
2.4	bug 实例 #3	24
2.5	bug 实例 #4	28
2.5.1	AT&T 电话中断	29
2.5.2	缓冲区溢出	32
2.6	小结	35
2.7	bug 问题	38
第 3 章	什么是 bug	39
3.1	什么是 bug	39
3.1.1	什么是软件缺陷	41
3.1.2	调试不是什么	42
3.1.3	什么是调试	43
3.2	为什么要考虑 bug	44

3.2.1	bug 的信心代价	44
3.2.2	名声和形象的代价	45
3.2.3	bug 的财产损失	46
3.3	bug 的本质	47
3.3.1	bug 的发生都有原因	47
3.3.2	bug 是可以重现的	47
3.3.3	bug 通常在有变化时出现	48
3.3.4	bug 也会产生 bug	49
3.3.5	bug 吸引 bug	49
3.3.6	证明缺乏理解	49
3.3.7	难的代码对大家都一样难	51
3.3.8	在软件生命周期中不同阶段的 bug 有不同的性质	51
3.3.9	稳定的系统中的 bug 的原因可能比错误的系统中的 bug 的原因 更难以识别	53
3.4	小结	54
3.5	bug 问题	54
第 4 章	bug 的生命周期	55
4.1	为什么会出现 bug	55
4.1.1	复杂性	55
4.1.2	现实	61
4.1.3	人类的弱点	63
4.2	bug 是如何产生的	64
4.2.1	对软件做修改	64
4.2.2	拙劣的描述	67
4.2.3	方法的复杂性	68
4.2.4	缺少一致的观点	70
4.2.5	程序员错误	71
4.3	bug 是如何躲过测试的	74
4.3.1	遵循形式过程代价太大	75
4.3.2	政策/市场决策	75
4.3.3	时间不充分	75
4.3.4	缺少重现能力	75
4.3.5	自负	76
4.3.6	差劲的描述/不知道要测试什么	76
4.3.7	缺乏测试环境	77

4.4	小结	77
4.5	bug 问题	77
第 5 章	bug 的分类	79
5.1	bug 的种类	80
5.1.1	需求阶段的 bug	80
5.1.2	设计阶段的 bug	80
5.1.3	实现阶段的 bug	81
5.1.4	处理阶段的 bug	83
5.1.5	编译的 bug	85
5.1.6	配置的 bug	86
5.1.7	未来计划 bug	86
5.1.8	文档 bug	87
5.2	严重性	88
5.3	bug 分类法	88
5.3.1	名字	89
5.3.2	描述	89
5.3.3	最一般的环境	89
5.3.4	症状	89
5.3.5	例子	89
5.4	bug 的分类	89
5.4.1	内存或资源泄漏	89
5.4.2	逻辑错误	91
5.4.3	编码错误	92
5.4.4	内存侵占	94
5.4.5	循环错误	96
5.4.6	条件错误	97
5.4.7	指针错误	99
5.4.8	分配/释放错误	100
5.4.9	多线程错误	102
5.4.10	定时错误	103
5.4.11	分布式应用程序错误	105
5.4.12	存储错误	107
5.4.13	集成错误	108
5.4.14	转换错误	109
5.4.15	硬编码长度/尺寸	110

5.4.16	版本 bug	112
5.4.17	不恰当地重用 bug	113
5.4.18	布尔 bug	114
5.5	为什么分类重要	116
5.6	小结	116
5.7	bug 问题	117
第 6 章	检测工作	118
6.1	整体调试	118
6.1.1	复制和粘贴错误	119
6.1.2	全局变量	120
6.1.3	副作用	121
6.1.4	观察意外的消息或结果	123
6.1.5	跟踪诊断显示	124
6.2	调试方法	124
6.2.1	科学方法	124
6.2.2	直觉	125
6.2.3	思维跳跃	125
6.2.4	诊断	126
6.3	商业技巧	126
6.3.1	内嵌式调试器	127
6.3.2	日志对象	127
6.3.3	跟踪对象	128
6.3.4	隐藏的诊断屏	129
6.3.5	为以后的程序运行保存 bug 数据	129
6.4	可重现实例	130
6.4.1	测试实例	130
6.4.2	数据依赖	130
6.4.3	从根本原因中分离症状	131
6.4.4	收集观察结果	131
6.4.5	统计/公理	132
6.5	小结	133
6.6	bug 问题	133
第 7 章	调试工具及其使用时机	135
7.1	测试和调试环境	135
7.1.1	测试组	135

7.1.2	测试套	136
7.1.3	过时的 bug 组	136
7.1.4	日志	137
7.1.5	跟踪	138
7.2	中级调试技术	138
7.2.1	内存漏洞检测工具	138
7.2.2	交叉索引及工具用法	140
7.2.3	调试器	141
7.2.4	Heisenberg 的不确定原理	141
7.2.5	嵌入式诊断	142
7.2.6	断言的弊端	143
7.2.7	同用户一起工作	143
7.2.8	bug 跟踪	144
7.2.9	代码覆盖范围分析	144
7.2.10	编译器	145
7.3	小结	147
7.4	bug 问题	147
第 8 章	调试的一般过程	148
8.1	识别问题	148
8.1.1	这是 bug 吗	148
8.1.2	为什么它是一个 bug	149
8.1.3	程序应该在做什么	150
8.1.4	程序到底在做什么	151
8.2	收集信息	152
8.2.1	用户对问题的描述	152
8.2.2	日志文件	154
8.2.3	新自观察	156
8.2.4	症状	156
8.2.5	失败的测试实例	157
8.2.6	相似的问题	157
8.2.7	近期变化	158
8.2.8	运行的环境信息	159
8.3	形成假设	161
8.4	测试假设	166
8.5	崩溃的 Web 服务器示例	166

8.6	重复直到某个假设被证实	168
8.7	提出解决方法	169
8.8	测试解决方法	170
8.9	重复直到某个解决方法被证实	170
8.10	回归测试	171
8.11	小结	171
8.12	bug 问题	172
第 9 章	调试技术	173
9.1	插入式调试和非插入式调试	173
9.2	短期调试和长期调试技术	174
9.3	成品软件调试的折衷办法	175
9.4	调试技术介绍	176
9.4.1	面对实际用户	176
9.4.2	查看观察日志	176
9.4.3	记录调试代码和过程	177
9.4.4	模拟代码和问题	177
9.4.5	简化重现性	178
9.4.6	把问题简化成最简单的元素	178
9.4.7	代码消去法	179
9.4.8	简化法	180
9.4.9	使用调试器	181
9.4.10	跳跃	182
9.4.11	分解法	183
9.4.12	种植错误法	184
9.4.13	编译检测法	185
9.4.14	整体考虑法	185
9.4.15	使用不同操作系统上的另一个编译器	186
9.4.16	每次改变一个变量	187
9.4.17	数字命理学和边界条件	188
9.4.18	检查最近的修改	189
9.4.19	清除系统中的“死代码”	190
9.4.20	问题假设法	192
9.4.21	检查未测试的代码	194
9.4.22	不变式法	195
9.4.23	存储器使用情况	196

9.4.24	互斥	197
9.4.25	显示系统的运行情况	198
9.4.26	和工作的系统进行代码比较	199
9.4.27	理解算法	199
9.4.28	检查连通性	200
9.4.29	核心文件	201
9.4.30	增加跟踪	202
9.4.31	数据关系检查	203
9.4.32	重放能力（记录动作）	204
9.4.33	生成系统的副本	205
9.5	小结	205
9.6	bug 问题	206
第 10 章	不同应用系统的调试	207
10.1	小规模单机系统	207
10.1.1	成为系统的用户	208
10.1.2	复制环境	208
10.1.3	提防 DLL Hell	209
10.1.4	输入/输出错误	209
10.2	中规模单机系统	210
10.3	中规模客户/服务器系统	211
10.3.1	生成后端数据库的一个简单描述	212
10.3.2	保留数据的使用情况来发现哪些数据经常被使用	212
10.4	大规模系统	213
10.4.1	为测试安装一个“后门”	213
10.4.2	观察外部条件的改变	214
10.5	实时系统	214
10.5.1	注意添加调试语句	214
10.5.2	监视硬件软件的冲突	215
10.5.3	时间问题	215
10.6	嵌入式系统	216
10.6.1	模拟器问题	216
10.6.2	抑制中断	216
10.6.3	协议错误	216
10.6.4	看门狗时钟	217
10.6.5	调试嵌入式系统	217

10.7 分布式系统	217
10.7.1 中间设备错误	218
10.7.2 预测错误	218
10.7.3 连接错误	218
10.7.4 安全错误	219
10.7.5 信息数据库	219
10.7.6 记录的事后调查分析	220
10.8 模拟系统	220
10.8.1 封装硬件接口	220
10.8.2 把模拟调用封装成预测错误返回	220
10.8.3 把实际系统模拟器的问题简化成最简单的形式	221
10.9 小结	221
10.10 bug 问题	221
第 11 章 调试之后	223
11.1 是不是在其他地方犯同样的错误	224
11.2 bug 背后隐含了什么	225
11.3 怎样预防同类 bug	226
11.3.1 理解原因	226
11.3.2 保留测试用例	226
11.3.3 利用 bug 为将来的设计	226
11.4 怎样更容易发现类似的 bug	227
11.4.1 创造工具	227
11.4.2 记录 bug	228
11.4.3 预留调试框架	229
11.5 我是否有所进步	231
11.5.1 bug 量度	232
11.5.2 bug 跟踪	232
11.5.3 怎样处理数据	233
11.6 小结	234
11.7 bug 问题	234
第 12 章 bug 预防	236
12.1 什么是 bug 预防	236
12.2 一般技术	238
12.2.1 预防 bug 的人	238
12.2.2 问题原因分析	241

12.2.3	检测错误	243
12.2.4	重用	247
12.2.5	减小复杂度和管理复杂度	253
12.2.6	现实世界的文档	257
12.2.7	内部基础结构支持	260
12.3	需求中的 bug 预防	262
12.3.1	理解问题	263
12.3.2	正确获得需求	265
12.4	设计中的 bug 预防	266
12.4.1	不同的设计产生不同的 bug 类型	266
12.4.2	设计接口	269
12.4.3	设计包含文档的代码	274
12.5	实现中的 bug 预防	275
12.5.1	理解开发工具	275
12.5.2	防御编码	276
12.5.3	实现选择	280
12.5.4	逐步测试	282
12.6	小结	283
12.7	bug 问题	284
第 13 章	测试	285
13.1	单元测试	285
13.1.1	测试什么	285
13.1.2	单元测试为什么重要	285
13.1.3	怎样应用到调试中	286
13.1.4	单元测试有什么重要问题	286
13.2	验证测试	286
13.3	质量保证测试	287
13.4	测试方法	288
13.4.1	路径测试	288
13.4.2	事务处理测试	290
13.4.3	输入验证	290
13.4.4	算法测试	293
13.4.5	决策表	294
13.4.6	状态机分析	294
13.4.7	综合测试	296

13.4.8	自上向下测试和自下向上测试	297
13.4.9	配置调试	298
13.4.10	恢复崩溃和掉电测试	299
13.4.11	安全性测试	300
13.4.12	第三方测试	301
13.4.13	多用户测试	302
13.4.14	负载和性能测试	303
13.4.15	测量和统计	304
13.5	小结	304
13.6	bug 问题	305
第 14 章	维护	307
14.1	什么是软件维护	307
14.1.1	维护工作	308
14.1.2	维护的挑战	308
14.1.3	软件的退化需要软件维护	309
14.2	创建一个可维护的软件系统	309
14.2.1	创建可理解的软件	309
14.2.2	可维护设计	313
14.2.3	提供一个回归测试环境	316
14.2.4	创建一个可维护的环境	316
14.3	维护现有的软件	317
14.3.1	修改现有软件系统	317
14.3.2	围绕现有的需求和设计 bug 展开工作	318
14.3.3	彻底的回归测试	319
14.3.4	保持修改跟踪	319
14.4	什么时候要放弃	320
14.5	小结	321
14.6	bug 问题	321
第 15 章	专业调试	323
15.1	学习成为一名调试员	323
15.1.1	在软件维护和改进组中工作	323
15.1.2	学习编写源代码的一个好办法	325
15.2	什么地方需要专业调试员	325
15.2.1	在设计阶段	326
15.2.2	在需求阶段	326

15.2.3	在代码审查阶段	327
15.2.4	在代码移交阶段	328
15.2.5	项目阶段总结	329
15.3	优秀专业调试者的特点	329
15.3.1	老练	329
15.3.2	耐性	330
15.3.3	探测技巧	330
15.3.4	处理压力的能力	331
15.3.5	工程的/科学的方法	332
15.3.6	忘我	332
15.3.7	坚定不移	333
15.4	专业调试者的一天	334
15.5	小结	336
15.6	bug 问题	336
附录 A	bug 问题的答案	337
附录 B	附加的阅读资料	344

第 1 章 Debug 简介

Benjamin Franklin 说过，“在这个世界上，除了死亡和税收之外没有确定的东西。”但是如果你处在软件开发行业，就可以修正这句话。生活中除了死亡、税收和软件 bug 之外没有确定的东西。如果你把自己冷藏起来，就可以无限期地延迟死亡。如果你去一个没有所得税的国家，就可以通过不购买任何东西逃避交税。但是如果你开发软件，还没有什么补救措施可以避免软件 bug 的灾难。

这本书是关于 bug 的——大的 bug、小的 bug、严重的 bug 和著名的 bug，讨论了所有类型的 bug。如果你曾经写过一个有 bug 的程序（就是说，如果你曾经写过一个软件程序），就可能在这里发现 bug 的一个版本。在本书中不止会学到什么是 bug 以及如何发现和修复 bug，还会学到如何在它们成为成品代码以前检测到它们。

大多数程序员会发现一件最令人沮丧的事情是：软件开发类的书籍都认为软件世界中有正确的设计、正确的代码和正确的测试计划后，就不会有什么 bug 了。你可以肯定这与事实相去甚远。无论如何“正确”的代码，如果它不可能做到用户想让它做到的事情，那么它就可能含有 bug。这并不是说，它不起作用；多数情况下还是正常工作的，但并不能完全按照希望的那样做。

软件的 bug 可能会很有意思。例如，Microsoft Windows 附件中计算器的早期版本里有这样一个 bug，它把 2 的平方根计算成为一个无穷数。这显然和实际的计算器的结果有差别。这样的 bug 通常会成为新闻，并招来嘲弄和耻笑；而且它们本来也值得这样，它们通常会损害编写这些应用程序的程序员们的形象。这些 bug 并不会带来真正的损失，但它会使人们质疑这些程序员所写的其他的、更重要的应用程序。例如，我们是否可以认为，在平方根函数中犯下错误的程序员，也会在操作系统的文件系统的代码中犯下类似的（甚至更严重的）错误？如果这些错误能使得所有的文件从硬盘驱动器上消失的话，就没那么好笑了。

其他类型的 bug 可能对程序员来说有点好笑，但会浪费用户的钱。例如，Sun Microsystems 制作的一个软件中的一个“故障”，曾导致 eBay 在线拍卖陷于 22 小时的瘫痪。不仅公司在这 22 个小时中的收入全都损失，而且它们的形象也受到严重的损失，在这个例子中，Sun 公司承认这样的 bug 的存在，但也声明有一个“补丁”（修补软件）来修正这个 bug，只是还没有被安装。当然，对于 eBay 的用户来说，是哪一个公司造成的错误并不重要，他们只知道他们不能再使用这个系统。

软件可能不只耗费资金，它们可能也会影响你在现实世界中的信用和名誉。例如，东北大学曾无意地招收了超出自己所能处理的人数 25% 的秋季班新生。以前，这个大学曾苦于许多潜在的申请人在其新的软件系统初次安装时被遗漏，这个损失迫使学校开展了积极的招

生活动，以招收那些一开始就被录取的考生，这导致了第二个问题：太多的学生。你能够想象那些没有被录取的学生的感觉吗？东北大学当然因这个失败而声望下降。

软件在我们的社会中越来越势不可挡。许多以前纯粹依靠人的交互而运行的系统，现在则彻底由软件系统来运行。这就意味着，生与死可能真的需要依赖于软件能否按照原来的期望而运行。一个软件的 bug 或者缺陷，会使得程序产生意想不到的结果或做一些错误的事情。在最坏的情况下，可能会导致死亡。举个例子，在 1985 年至 1987 年间，一台 Therac-25 放射治疗仪中的错误的软件，使得若干癌症患者因致命剂量的放射线而死亡。虽然这对于本来想帮助人们摆脱疾病的死亡威胁、却害死了人的机器来说，是一个巨大的讽刺，但它本身更是一个莫大的悲剧。虽说在医疗仪器的软件中发现的 bug 的数量非常少，但这个案例本身就说明当软件 bug 出现在成品代码中时会有什么样的结果。

1.1 本书的内容

总的来讲，本书是关于软件 bug 的——如何检测 bug，如何修复 bug，以及如何杜绝产生 bug 的根源。在完成本书的阅读以后，你将会知道为什么会产生 bug，怎样做可以避免它们产生，以及在现有代码中如何发现和修复 bug。但本书更多是“以 bug 为中心”。我们从 bug 早就存在、仍然存在、将来仍会存在这个角度来研究问题。谈论“无 bug”的软件很容易，因为这样的软件几乎不存在。我们将关注存在的东西——有 bug 的软件。

在软件开发书籍的历史中，大量的时间和篇幅被用于讨论关于设计软件、编写代码、配置系统的更好方式。在所有这些研究中，通常到最后才考虑调试。调试是软件行业中的一个脏污的小秘密，是一个被忽略的被放在地下室的后柜中的东西。现在应该把调试从柜子中取出，并按其自身特点来对待它。它是软件开发生命过程中的一个自然的部分。

bug 可以在应用软件的开发生命过程中的任何阶段潜伏下来。我们将考虑在开发的需求阶段中 bug 是如何产生的。在该阶段 bug 不需要多少修补，但如果最终出现在成品中则会耗资巨大。同样，我们也考虑在开发过程中外部原因如何产生 bug，也叫做“特征潜伏”。软件 bug 常常由于不同的原因而溜过了开发过程中的测试阶段。只有理解其发生的原因，才可以更好地设计、应用和测试系统。

始终贯穿本书的一个主题是关于如何避免 bug，修复它们来验证所做修复的思想。你需要理解系统的“蓝图”。我们可以通过书中的手段，来检验你能够更加理解软件系统的能力，提供如何更好地备案所编写代码的信息，以及研究一个几乎没有或有很少经验的系统的思想。虽然我们通常考虑让原来的开发人员来调试系统，但让新的开发人员加入到一个已经存在的系统中调试问题也是正常的。

而且，我们也将考虑当今软件工程领域“专业调试员”的角色。尽管调试在软件一切领域都极其重要，但它仍然被看作软件工程师的例行工作，这种状况越来越明显地需要改变。调试必须和设计与需求分析一样进行考虑：就是说，作为它本身的工作。用特殊的态度来考

虑如何编写系统的设计，并不起多大的作用；但是如果用特殊的态度来考虑如何维护和修正系统的代码，则不然。我们的软件工业应该开始理解这个问题了。

我们同时也将关注一个好的调试员的特点，并且阐述如果你正在从事调试工作，应该如何加强这些特征。我们将着眼于你可以通过哪些工作，而使自己成为一个更好的调试员，包括那些并非在修正错误时或在 bug 出现在系统之前所采用的技术。

1.2 为什么要关注 bug

2001 年早些时候，纽约州承认：由于一个关于对存储在数据库中的指纹进行数字分配的问题，2001 年底以前将用完可能采用的数字。为了修正这个“bug”，该州决定从 1 开始循环使用数字。假设，在某个时候，你在纽约州的某次犯罪调查中被提取了指纹，或者因为你曾经要访问国会大厦而在经过允许后指纹被记录在案，会有问题吗？应该不会的。

在犯罪发生后，指纹被提取，然后运用指纹识别来跟踪那些被调查和被指控的人。假设你的指纹碰巧与一个新指纹拥有同样的标记数字，而该新指纹正好吻合一个犯罪现场的指纹。由于这两个数据库系统因近期的数字重用而没有得到更新同步，你就会发现自己被法庭传讯。

还有人到现在仍不确信 bug 是重要的，请看这个例子：在 1972 年，一个软件杂志（已经不再出版了，可悲）叫做《EDP 分析者》，它认为在多数组织中 50% 的预算用于维护和调试。在 19 世纪 80 年代后期，一些组织发现这个数字上升到 80%（同样出自《EDP 分析者》）。考虑到一个一般的 IT 部门的预算大部分被用于设备和工资，这意味着本来可以用于提升工资和购置新的设备的钱，实际上都被用于修复问题。后来 Capers Jones 等三人的研究结果可能会让人欣慰些，目前该数字又降到了 50%。

另一条有趣的信息来自于 Jones 的研究：大约有 20% 的维护预算被用于修正那些在系统的初始开发中提到的缺陷。这给了我们什么启示呢？这意味着如果你正在从事修正软件问题的职业，你今后将永远不会失业，臭名昭著的“千年虫”问题已经证明了这一点。未来的问题已经浮出水面，将使“千年虫”问题变得逊色得多。在本书中我们将考查其中的一些问题。

另一个考虑 bug 的重要理由是“silver bullet”问题。从外部来讲，人们相信可以有简单的神奇公式应用于软件的开发中，在所有的 bug 出现之前消灭它们。这完全是无稽之谈。过去 30 年以来程序员们并没有真正做出什么改变。那些在软件工业发展的初期就存在的问题，现在仍然存在。软件的 bug 和在早期一样，仍然在当今的软件中出现，但是问题的数量已经大不一样。过去 bug 只出现在独立的单个用户或者一组用户之中，现在它们却出现在整个世界。那些过去仅意味着要手工地重新制作报告，并重新输入数据的问题；而现在可能意味着一个组织的生存。

如果以上所说的这些问题，还没有使你相信软件 bug 是一个严重的问题而且需要着重强调的话，那可能就没有什么可以说服你了。要得到更多的例子，我们建议阅读第 2 章来看看

“现实世界”中出现了什么样的问题，以及在金钱和荣誉上给工业造成了什么样的损失。如果在阅读完所有这些研究的例子之后，你还不相信软件 bug 是一个问题，请合上本书去做别的事情。但当你不得不在一周内花 7 天时间去修正一个到达危险地步的问题时，请不要吃惊。

1.3 什么是 bug

我们用很多名字称呼它：软件缺陷、软件 bug、软件问题甚至软件“症状”。无论怎么称呼，它们实际上就是软件做了没有期望它去做的事（或者相反，软件没有做到所期望它去做的事）。软件 bug（本书使用这个名词）包括从软件失效到返回错误信息到混乱的显示。

在本书中，软件 bug 是软件没有做到所期望做的事情的例子。在第 2 章我们将更加详细讨论这些例子。一个 bug 的严重性要从它所有的事情如何糟糕来计算。我们认为一个 bug 的“耗费”是它对用户造成的破坏的严重性，加上它对调试员和开发人员修补问题时所耗费的时间和精力。

本书中所用的其他描述 bug 的术语包括分类、可重现性、边缘效应和可升级性。我们将讨论如何发现 bug，如何修复一个 bug，以及如何判断一个 bug 是否已经被修复。调试不被重视的一个原因是，如果不能确定不再发生的话，它对于“修复”就没什么作用。

1.4 本书的对象

虽然看起来本书只是针对那些开发人员，但仍可供更多其他人来阅读。开发人员肯定会学会理解大型应用程序中会出现什么样的 bug，如何检测它们，以及如何在它们出现之前就阻止它们。同样，调试员会学会如何对他们所发现问题的类型进行分类和标识，以及如何科学地根除和销毁 bug，特别是在他们所负责的应用程序中。

管理人员也应该读这本书。如果不理解产生软件 bug 的过程和问题，以及那些设计用来最小化软件 bug 的过程，这些问题将永远不能解决。人们将做那些管理人员最强调的事情。如果只想更快、更便宜地得到软件，那么只将会得到更大更多的 bug。如果你需要愉快的客户，那应该平衡更好的错误探测器和诊断工具与其他列表之间的关系。这并不是一本代码书。虽然在许多章节列出了代码的例子，但也同样有非技术性的例子。

测试人员当然也应该对本书感兴趣。测试人员建立一些基于人们在过去发现的各种问题的例子。因而，通过学习已经发现的 bug 的种类，并判定这些 bug 所产生的根本原因，测试人员可以建立更多的有效测试例子。测试并不能消灭 bug，它只能在它们中的一些溜入代码之后，出现在用户前面之前阻止它们。太多的人相信测试与阻止 bug 是并行的事情。阻止 bug 的工作只在于系统设计和应用层次。

很明显，开发人员也可以使用这本书。通过了解那些存在于别人的软件中的问题，你将

能避免在自己的软件中出现类似的问题。正如 George Santayana 所说,“那些不记得过去的人,会再经历一次的”。除了阻止 bug,通过本书的学习,开发人员还可以理解最好的策略,以在成品代码中发现和修复 bug。虽然本书非常适合于专业的调试人员,但我们都知道更多情况下还是初始的开发人员(或刚雇用的入门水平的人员)来修复 bug 问题。通过理解那些所能够掌握的适当的策略和技术,就会减少查找和修复 bug 所需要的时间量。

对本书不太感兴趣的一个群体可能是系统体系结构人员,虽然他们同样能够通过理解 bug 产生的根本原因,以及如何避免 bug 之后得到帮助。更一般地,我们在软件开发行业中责备那些代码写得差劲的问题。如果需求分析和设计很差劲的话,最终会得到一个很差劲的系统。代码可以写得很漂亮,但如果它内在设计非常差劲,或很难被应用这个软件的人所理解的话,将不会起什么作用。

另外,我们将新概念“专业调试员”作为一个真正的专业来考虑。这个特定角色在一个公司的招聘单上不会有一个“公开的需求”,但我们认为应该有。调试是开发和移植过程中的一个重要部分,而且它应该得到这样的对待。

1.5 本书的组织

本书被组织为三个不同的部分。第2章到第5章包括诊断和分类的 bug 类型,以及它们的症状。第6章到第10章讨论那些处理 bug 的策略,包括可以使用的工具,以及在不同情况下如何最为有效地使用它们。第11章到第15章包括对专业调试的解释,以及如何成为一个出色的调试员,即使这并不是你在公司里的原始位置。

因为第2章到第5章包括诊断和分类的工具,我们把这部分称为“检测”部分。这些章节开始包括判断什么是 bug,什么不是 bug,从 bug 的症状判断其根本原因,以及建立一些必要的假定来修复问题。一旦判断出所观察应用中 bug 症状的真正原因,解决问题就不只是描述出你想让这个应用如何工作那么简单。

诊断工作包含了调试工作中的主要工作,而且涵盖了这本书的主要内容。虽然你可能认为调试是一个简单的事情,就是被告知出现了什么现象然后去修复,但实际上根本没那么简单。测试员和用户所报告的被我们称为“bug”的东西,通常是一些简单的症状,而其内在另有根本原因。例如,考虑一个程序,当过程 A、B 和 C 按用户手册上描述的顺序运行而程序崩溃,看起来可能是由于 A、B 和 C 之间的交互造成的。但是更深入的分析会发现,无论 A 和 C 按什么顺序运行都会导致出现问题。这可以明显地看出有一个介于两个错误之间未能解决的错误情况:一个内存失效错误或者仅仅一个未初始化的内存地址出现在 A 进程中,而后来该地址又用于 B 进程中。类似的问题很容易发现,但很难修正,因为其根本原因与其症状没有什么关系。在第一部分中会有我们要考查的 bug 种类。

第6章到第10章包括如何处理所遇到的 bug,和利用在检测阶段发现的信息诊断 bug 的策略和技术。在本书的这个部分,我们将考虑一些使用诊断技术以及所发现症状的方法,

来发现那些确实能够修复问题的方法。但一般的修复和寻找 bug 的技术，在大多数的操作系统、程序设计程序和应用程序中都有效。

调试和 bug 修复是计算程序设计世界中的“幕后技术”。虽然是出于无心，多数程序员并不想与别人分享交流那些他们修复所遇到问题的方法。这种交流和科学方式的缺乏，导致在每个组织的开发人员中都有上百个低效且过于复杂的方法。如果我们用对待设计和使用的态度来对待调试的话，就会发现软件 bug 虽然的确不只是过去的东西，却只需耗费在该领域中非常小部分的时间。这样就能带来更加稳定的应用程序和对用户更加友好的特征，而且许多时间被节省下来不用去修复 bug，而是用来去做更多有意义的事情（当然，从最终用户的角度看）。

第 11 章到第 15 章讨论专业调试员的角色应是什么样的。如果在我们的专业中有这样一个领域，在该领域中我们必须为每个项目组分配一个专家，那么调试就当然是这个领域。我们将说明如果调试员是开发组的标准成员，整个软件开发过程将会得到改进。调试员可以不只把他们的技术用于修复问题，他们同样可以使用这些技术去设计搜集需求、测试和建档。所有这些技术对软件开发的多数部分是重要的，但调试会将这些技术组成为一个包。我们将考虑如何成为一个好的调试员，以及如何在调试中使用学到的技术来增强个人的技能。

1.6 调试简史

看起来，自从有了计算机，就有了计算机 bug。但是并非如此。即使已知的最早期计算机中出现的错误，在当时并没有被称作 bug。一个叫做 Admired Grace Hopper 的女士发明了“bug”这个术语。

Grace 在 1906 年 12 月 9 日出生于纽约城。她是三个孩子中最大的，童年中大多数时间都在玩玩具。在 7 岁时，她拆开了 7 个闹钟来看它们究竟如何工作。这些经历，在她将来工作在计算机世界中时，对她帮助很大。

1928 年从 Vassar 毕业之后，她去了耶鲁大学攻读硕士学位。从耶鲁毕业后，她留校为数学教授。1943 年她离开耶鲁，作出了一个改变自己的未来同时改变计算机世界的未来的决定。随着第二次世界大战的爆发，Grace 决定为海军工作。因为她的年龄（34）和体重（105 磅），她并没有被海军部门认可。但政府当时认为像她这样的数学家对战争很重要。这样，Grace 的申请得到了认可，并于 1943 年 12 月正式编为海军一员。

她的第一个工作是在 Howard Aiken 中校的带领下在哈佛大学的顺序计算部门工作。她是 Mark II 上的程序员，Mark II 是世界上第一台自动的顺序数字计算机，用来判断大炮的射击角度，而且是在战争期间不同的气候条件下。它被认为是当时最重要的程序。

正是她在 Mark II 上工作期间，她针对计算机问题创造了“bug”这个术语。第一个“bug”实际上是个飞蛾，这个飞蛾飞过了一个打开的窗户并飞到 Mark II 的一个中继器上。这个飞蛾造成它短路，导致系统暂时关闭。这个飞蛾后来被弄走（调试？）然后粘贴到项目的日志

本上。从那时起，如果她带领的队伍在产生数字或工作于代码，那他们就说是在“debugging”系统。

从这个戏剧性的开端起，计算机调试在几年内发展成为一种随意性的过程。早期的调试行为以系统的数据堆或使用的输出设备，例如打印机、显示灯为中心，来表明一个错误出现了。程序员将逐行逐行地检查代码，直到判断出问题的位置。

在计算机程序经过了纸带卡片打孔时代，到纸带输出和最后 CRT 终端普及之后，调试取得了巨大的进展，计算机程序也同样跟着变化。不同于面向批处理的程序（即夜间运行连白天也运行的程序），程序已经变成用户交互式的了，要求用户在开始时或者甚至在程序运行期间有真正的输入，一直到程序结束。这被看作计算机程序设计的一个巨大的进步。其实他们并不知道多少-----

调试的革命的下一个步骤是命令行调试器的进展。这些简单的程序对程序员来说是一个令人震惊的进步。他或她不再需要猜测程序中内存地址的值，调试器在一个给定的内存位置中可以给出值。这就允许工作于流水线上的程序员，可以直接查看寄存器和包含程序中局部值的内存块。虽然仍然不太好用，但这些程序都体现了最初的真正目的——把调试从偶然性的事情变为一个可重现的过程。能够使某些东西重现，就是使其迈向成为一个科学工程的过程的第一步。

在调试器成为程序员的工具的一部分之后，软件就会开始考虑其他能够使得程序更容易被调试的方法。软件项目将变得更火，那些在小项目上运行良好的技术，在程序达到一定大小之后就不再有效了。可扩充性，看起来即使在软件复杂度曲线的开始处，也都是个问题。编译程序厂商已经发现，他们所拥有的在分析诸如 C、FORTRAN、COBOL 的高级语言时得到的信息，可以被保存在一个独立的文件里，叫做一个特征映像，它可以将程序中变元的名字“映像”到它们在运行时所被加载到的物理存储器地址。查看变元名字的能力，以及将它们映射到存储器地址的能力，允许程序员按名字来访问内存。这些调试器称为“符号调试器”，因为它们的操作是从符号到允许开发者观察数据。

在这个阶段的另一大进步是设置断点的能力。名词“断点”的意思是在一个给定地点中断代码并停止程序执行的能力。程序仍然被加载到内存，它只是在遇到断点的时候不再继续运行而已。这个进步是允许他们在程序崩溃时，或者在做了什么没有期望去做的事情时，可以查看变元（符号）的内容。在使用断点之前，程序实际上只能在两种状态下查看：在运行之前的初始状态，和结束或崩溃后的最终状态。如果在运行期间发生了什么事情，程序员就只能猜测其原因。有了在代码关键连接处设置断点的能力，程序员就可以发现有些事情到底为什么发生。当然这还不是最圆满的解决方法，因为他们必须要知道他们想让程序在什么地方停止，但这使得当时我们这些编程人员有一个巨大的进步。

调试过程就和软件世界本身一样，开始进步得越来越快。有了查看原始代码行以设置断点的能力。更好地访问内存以及查看内存中的变化的方式也有了。而且，许多调试器都有了情况（condition）断点。情况断点允许设置一个情况，诸如某个给定的变元等于零，如果已

经在那个特定的行上设置了一个断点，那么在这个情况出现后程序就会停止。在 Borland 推出 Turbo Pascal 的时候（20 世纪 80 年代早期），调试器就与许多集成开发环境（Integrated Development Environment, IDE）一起进入了可视化阶段。在这个令人吃惊的环境中，可以在同一个系统中编辑、编译、链接和调试代码，不需要运行分离的程序，不需要加载特殊的符号表或使用特殊的编译器选项。Turbo Pascal 标志调试进入了现代化阶段的黎明。

当 Microsoft Windows 在 20 世纪 80 年代中后期推出的时候，就需要一个新种类的调试器。在过去，尤其是 MS-DOS 时期，调试器只能做一个单独的任务，调试一个程序。Microsoft Windows 把这些都改变了。虽然 Unix 中已经有了线程调试器，但它们很笨拙庞大，而且难以在一个真正多线程任务的系统中使用。虽然 Microsoft Windows 下的应用在初期并不是多任务的，但它们很好地模拟了多任务这个概念。开发人员不光要注意他们自己的应用程序，而且要留意那些同时在机器上运行的其他应用程序。以前的应用调试器只能解决基于文本的应用，Microsoft Windows 是一个图形用户接口（GUI）。这就要求对调试有新的思想。毕竟，如果应用程序的输出是图形的，你也要能够知道它是否正常运行。现有的调试器并不允许这一点，你可以在应用和调试器之间进行屏幕切换，但当应用程序在屏幕上“画”的时候并不起什么帮助作用。如果应用程序没有运行，屏幕就是空白的。

所有这些特征只适用于实际地调试应用程序。然而通过其他方式，我们也在学习更多的关于“专业调试”和“bug 预防”的艺术，即将 bug 在到达最终代码以前清除。在本书中将学到的技术都是建立在先前的程序员们多年的汗水和痛苦之上的。花点时间通读本书，理解调试技术的方式，以及如何才能使用这些技术在自己的软件中修复 bug。这是值得花点时间和努力的。

1.7 小结

在本章中，你开始涉足到关于 bug 的神秘世界。初步认识了我们在本书中将要讨论的问题的种类，以及一些用来解决这些问题的技术样本。

bug 在软件开发工业中是一个重要问题。因为我们存在糟糕的开发过程、对系统的误解以及不恰当的需求定义，所以 bug 会一直纠缠我们。通过阅读本书余下的部分，你将不仅会知道如何识别和解决你所遇到的问题，也会学会如何在 bug 首次出现前阻止它们。

1.8 bug 问题

为了更好地理解什么是 bug，以及应该如何解决它们，我们决定在每一章中都会添加一个 bug 的例子。这些问题的答案可以在附录 A 中找到。虽然一些例子包含代码，但是许多例子将只是简单地描述导致报告的问题的情况。还需要使用提供的线索和自己的探检技术，来判断在每个例子中到底发生了什么事情。我们可以从那些例子考虑现实世界中的 bug。

在一个普通的 Internet 商店里有一个系统，它不断运行以使客户和股市信息保持一致。这个专用的系统在大多数时候都正常运行，但通常在更新用户的报价申请时停止，并只是简单地返回无效或过期的数据。程序员花了相当多的时间，来判断为什么系统会偶然地启动和停止。同时，操作人员要花大量的时间来描述问题的症状。最后，问题变得很严重，以致系统每天开始崩溃好几次。程序员被难倒了，操作人员也发怒了。

最后，解决的方法却不只是简单，而且几乎不需要对任意一方修复。在到书的末尾寻找答案之前，问问自己将会采取什么过程去解决问题？最后，假想你是作为一个专业的调试人员来解决这个秘密。

实际的答案并没有完全包含在线索中。但是那些应该遵循的过程如果做得正确的话，会使你更加迅速而容易地找到办法。在阅读“真正”的答案之前和之后，要仔细地思索这个例子。然后开始阅读本书的第一部分，在那里你将知道这个问题在成为一个严肃的问题之前是如何产生的。

要记住本书是围绕着调试的“科学”的。科学包括那些能够用规则来解释的可重复出现的例子。调试没有什么不同。bug 是偶然出现的，它们或者存在或者不存在。我们将提供需要的工具和技术，来理解 bug 为什么会出现在代码中，以及如何在破坏应用程序的前提下消灭它们。

第 2 章 研究著名的（以及不太著名的）bug 例子

假设今年是 1986 年，并假设你是后来的 Richard Feynman，电话响了，然后你接。这是来自工作在 NASA 的 William 打来的电话，他是行政主管。你知道这并非一个普通的社会性电话，因为挑战者航天飞机几天前在 1986 年 1 月 28 日爆炸了。你的猜测是对的。他打电话来是请你成为委员会中的一员，来调查挑战者号灾难的原因。实际上这是 NASA 在邀请你这个著名的诺贝尔奖获得者，作为关于 NASA 近代历史上最著名的事件的高级职业调试员，你接受了这个邀请。当时你对航天飞机的组件和系统还不熟悉，你也不知道挑战者这次致命的飞行中记录了什么数据。惟一的线索是在录像中发现的烟雾，看起来好像出自右侧的燃料箱，那么你将如何去完成调查失事原因的任务呢？

事实上，Feynman 博士不屈服于政治压力，飞到全国各地与工程师和操作人员交流以发现事故的原因。他最终在一个现场直播的记者招待会上进行了桌面实验，使用了航天飞机的模型和一杯冷水来证明他的理论。

谢天谢地，我们作为调试员的工作对象是软件系统，比 Feynman 博士所面对的要容易得多。在多数情况下，我们可以重现 bug，以保证我们能完全理解它的原因和影响。NASA 的科学家们可没有这个条件。毕竟，他们不可能通过再爆炸一架航天飞机来再现 bug。而且 Feynman 博士必须提交一份正式报告来描述他的发现和分析，为公众提供一个解释；而且更坏的是，来自于 NASA 的政治压力。我们在修复 bug 时做任何一件事情的最后期限是什么？在软件工业中，我们对 bug 的分析很少。我们并不问诸如 bug 如何被发现，bug 如何发生，以及我们如何做以避免 bug。如果我们真的做一些认真的分析，那么我们几乎肯定会记录下我们的发现，以使大家能够共享它。

从各方面来讲，本书荟萃了我们已知的和经历的许多年来关于 bug 的认真的分析。在第 12 章，我们将讨论如何分析 bug，如何增加关于 bug 形成过程的知识库，又如何使用这些知识来帮助避免类似的错误，以使得在将来能更好地生产软件。我们认为答案是“行”。在 1981 年，NEC 实施了一个计划来帮助软件工程师和项目管理员学习 bug。这个项目罗列了那些来自于许多合作项目的 bug，鼓励软件工程师去发现软件失效的原因，并防止它再次发生。在项目开始的 10 年里，他们得到了许多教训，并且能够应用这些教训来提高他们的生产能力，降低 bug 产生的频率。

最初是 NEC 开始将出现问题和解决方法进行分类，然后通过研究这些问题之间前因后果的关系，软件工程师就可以准确地找出解决这些问题的对策。下面我们也将使用这个方法。

2.1 大致情况

在本章，我们收集了一些 bug 来进行研究。许多这些 bug 是很有名的，而且记录得非常好，因为它们的出现是公开的事情，而且许多都有大的影响。这些 bug 无论被引用的次数多少，它们都是比较恰当的研究例子。问题是并非每个人都要涉及到大规模的、对安全性要求较高的软件项目，所以这些开发者遇到的困难的程度，看起来可能不同于你在日常工作中所遇到的困难。因而，我们也收集了一些来自我们的软件工程师的例子。

这些个人的 bug 更加现实一些，不会导致爆炸和数十亿美元的损失。还有，本书中我们也提供了别人编写的 bug 来展示不同的问题。我们不想把这些 bug 作为难题。我们学习这些事情的目的是，就是研究著名的和不著名的 bug，并从中学习到一些东西。

2.1.1 现实生活中的分布式系统

我们先看来自于我从前的工作中的两个 bug 的例子。第一个 bug 持续了几个小时，而后被迅速解决。这对第二个 bug 来说是一个很好的教训。4 个月后，即在我刚刚开始另一个工作的时候，遇到了一个相似的 bug。但是，在这个新的公司里，这个 bug 在我到来之前已经活动了两个月。第一个 bug 的严重性是它影响了整个系统。第二个 bug 虽然持久，但不会导致停机；它只是一个小的麻烦，并不能导致大的灾难。

故事：Y 公司

在 1999 年的后半年，我在为一家向互联网提供金融报告的网络公司工作。让我们把它叫做 Y 公司。提供金融报告的系统的体系结构很简单，网络请求来自于多台 Web 服务器，Web 服务器把请求交给一个负载管理器。负载管理器把请求送到一个环形结构的四台报告服务器中。报告服务器就从数据库中提取数据，产生报告。

一天，警报器开始左右闪烁。报告服务器一台接一台地崩溃。报告服务器会突然崩溃，几分钟之后重启，然后几分钟之后再度崩溃，所有的服务器都是这样。请求的频率不高，也不反常。重新启动机器看来并没有什么作用。网络操作人员弄不清楚是什么引起了这次混乱。开发人员被叫来了，因为我设计和实现了内部服务器所使用的结构方式。别人问我以求得帮助。开发人员怀疑通信结构中的负载均衡模块发生了错误，而报告服务器是由于负载过重而崩溃的。

通信结构中有一个内置的日志和跟踪功能，但在产品环境中一般没有被使用。我做的第一件事就是启动日志并跟踪输出来观察系统的动作。报告服务器所产生的日志对我并没有起多大作用。日志中有许多错误和警告信息，这些信息都说明不了什么问题。然而，在每个实例中，在报告服务器崩溃和重启之前，有一个日志条目指出报告服务器正在中止运行。我问了一个工程师关于这个日志的问题。但他的反应非常模糊：它应该是由程序被设计成在它认为出现了问题时，自己重新启动所导致的结果。在那时，当查看来自于负载管理器的日志

文件时，我发现这个错误同一时刻只发生在一台服务器上。我立即认为是负载均衡机制没有正常运转，负载服务器使得报告服务器超载，从而导致其重启。我告诉了开发人员我的设想，回到桌子旁查看代码，寻找潜在的错误。它们并没什么特别的。我想起了工程师的描述：报告服务器被设计成在它认为有问题发生时就重新启动。我决定查明到底是什么种类的问题导致报告服务器自己重启。一般地，报告服务器有 20 个线程，每个对报告服务器的请求占用一个线程。正常情况下，每个线程要在 1 秒或 2 秒内完成处理。所以在服务器内部就建立了一个“安全”逻辑，这样当所有的 20 个线程都繁忙而第 21 个请求到来时，第 21 个请求最多会等待 5 秒钟直到某个进程完成。如果第 21 个请求超时，报告服务器会认为有些东西出了差错，这样 20 个正在运行的线程就会挂起。这样，惟一可采取的行动就是终止，然后重新启动服务器。

听到这些之后，我回到办公桌，再考虑一遍。几分钟之后，一切都明白了。我用日志和跟踪文件证明了我的推测，并向服务器工作人员描述我的假设。

- ◆ 加载管理器在一个时刻只使用一台服务器，其原因并非是因为负载均衡算法的错误。它之所以在一个时刻只使用一台服务器，是因为其他的报告服务器不可用：它们正在重新启动。通过观察加载管理器和报告服务器的日志文件时间戳，可以轻易地验证这一点。

- ◆ 报告服务器的日志文件表明“安全”逻辑已经被触发，这意味着每台报告服务器必须包含 21 个并发的请求。这说明有时系统中会有 84 个请求，因为当时有四台报告服务器在操作。

- ◆ 最后，报告服务器会因为违反了“安全”逻辑而重新启动。这样的结果就将系统的容量减少到了 63 个请求。但是输入请求的频率仍然没有改变。这就增加了剩下三台报告服务器的负载，而且最终导致其他的某台服务器，也因“安全”逻辑而重新启动。系统的容量被削减到 42 个请求。到第一台报告服务器的重启完成之后，第三台服务器也将崩溃和重启。只要系统的输入频率和响应频率不变，这种现象就将一直延续。

这个假设与所观察的服务器经常崩溃的现象不太一致，而且服务器不可能稳定地运转。但是，我们仍然需要判断 84 个请求的备份日志的根本原因。我们已经发现负载并不是非常重。这样只剩下一种可能。问题肯定和那个每个请求在 1 秒或 2 秒之内完成、而且第 21 个请求只等待 5 秒这个假设有关。这些选定的值看起来有很大的余地来发生错误。但是，这个假设在正常的操作情况下是有效的。

在这个描述的开始，我指出报告服务器要依赖数据库来产生报告。在这个情况下，数据将要经历某种我们将永远不能判断出错的原因。一个简单的 SELECT 语句要 10 秒钟才能完成，而不是只用 0.8 秒。数据库性能的降低触发了设计缺陷，情况就发生了迅速的变化。因为每台崩溃的报告服务器都会降低可以使用的容量，那么更多的压力就会转而加到剩下的服务器上面。如果在整个系统中负载能减少到少于 20 个并发请求，问题就能很轻易地解决。

既然断定是由于数据库服务器失效造成的，那么只要重新启动 SQL 服务器就会解决问题。

故事：X 公司

在 2000 年中期，我开始在一家提供计划和业务信息以及电子商务的网络公司上班。让我们把它叫做 X 公司。X 公司的架构与 Y 公司的大致相同：Web 服务器与加载管理器通信，而加载管理器与后端的代理通信，后端代理与数据库通信。但是有一个小的冲突。X 公司同时提供交互式语音响应（IVR）界面，允许顾客通过一般的电话访问系统。应用 IVR 方法，只是用一个 IVR 服务器替代 Web 服务器，用与 Web 服务器同样的方式与负载管理通信。另一个区别是 X 公司使用了两个彻底独立的数据中心。而且，IVR 服务器将其中一个数据中心作为它的主要活动系统来使用。当活动数据中心发生冲突时，IVR 服务器则切换到备用的数据中心。

一天，X 公司的 CTO 来到我的工作室，请我与 IVR 工作人员交谈，以帮他们调查最近出现的服务器问题。几天前，IVR 系统停止运行达 8 个小时。当我去和 IVR 系统人员交谈时，并没有得到什么有价值的信息。存在有 IVR 服务器出错时的日志文件，但是它们与网络操作中心的一些网络问题有关，而这并不能解释为什么没有启用备份的数据中心。通过进一步调查，发现 IVR 服务器存在有与数据中心之间的问题，这个问题在过去两个月内一直存在。

因为我对系统并不熟悉，我向工作人员索要所产生的日志文件。我想能够在中断再次出现时捕捉到系统的行为。日志文件虽然在表示错误时是有用的，但对于描述系统的行为来说是无用的。它只能表示每个请求在完成之后的状态。如果一个特殊的请求会导致失效，日志不会表示出 IVR 服务器在崩溃时的状态。而且，IVR 服务器在出错期间并不崩溃。并没有我们可以使用的核心文件。但是，日志文件的确记录了当 IVR 服务器切换到另一个数据中心时，有相当大数量的事件。每个数据中心切换事件与一个用-3 表示的错误代码代表的一系列三个请求有关，这个代码表示在 IVR 服务器试图与负载管理器通信时有一个请求已经超时。在 X 公司，每个请求有一个惟一的请求代号，它在请求所访问的每个服务器上都作了日志，所以可以用它来跟踪系统中的每个请求。但是，IVR 服务器并不在自己的日志中记录请求的代号。所以我们无法调查请求为什么超时。

根据我从 Y 公司学到的经验，我问了工作人员关于 IVR 服务器失效时的策略。后来发现 IVR 有个特殊的性能要求——不同于访问一个 Web 地址，人们并不喜欢在电话上等待，所以 IVR 必须在特定的时间内返回。IVR 被设置成只在 3 秒钟以内接收来自负载管理器的数据。在这三个决定性的尝试中，IVR 会自动地切换到备份的数据中心。在备份的数据中心，IVR 服务器会尝试再次连接到主数据中心，一般是在 2 小时以后。如果成功，它将重新建立它与主数据中心之间的连接。

这时，我有了两个合理的假设，可以满足所有的观察结果。①因为 IVR 服务器存在有与数据中心之间的问题，所以 IVR 服务器实际上是在没有一个备份的情况下运行的。如果所连接的数据中心出了什么问题，IVR 服务器就不可使用了。②IVR 服务器与数据中心之间无法连接的原因，被精确地记录在现有的日志文件里。对它的请求超时了。为了判断出它为

什么频繁地超时，我们可能要跟踪那些导致系统超时的问题的请求。

但是，并没有日志和其他信息能证明我的假设，所以我与 IVR 工作人员一道去做出对调试更有用的一个日志。一旦我们能够跟踪系统中的请求流，就能很容易地发现瓶颈在什么地方，后端服务器通常要用一定的时间来处理一个请求。这个后端服务器向位于同一台机器上的、作为服务器的数据库发送一些 SQL 请求。数据库请求比预期的时间要长，因为后端服务器会有比较重的 CPU 负载。一个推测是机器负载过重，需要添加硬件。另一个推测是 SQL 语句的效率低，这就需要调查。几天之后的调查结果发现数据库在事故之前被升级了。但数据库管理员却忘记了升级索引，这样每个请求将耗费比原来大 10 倍的时间来进行处理。CPU 的沉重负载，只是反映了一个未索引的数据库中运行的 SQL 语句的效果。“起硬件作用的”数据中心的数据库被升级然后重新做了索引，这样就不会有什么问题了。因为没有可以供进行调查所用的信息，我们将永远不能恰当地解释那些失效问题，但是缺少备份的数据中心是关于失效的一个主要原因。

教训

前面两个例子中的教训是惊人的。两个例子中都涉及到了长期操作的系统。两个系统都按设计的要求进行，而且没有应用 bug。在这个例子中的修复（简而言之）只是重新启动或对数据库重新索引。两个问题都是因为数据的响应时间比预期的长。问题的原因在日志文件中说得很明白。但是要查明日志并完全理解应用却很难，为什么？

- ◆ 组件思维——在两个例子中，初始的工程师们主要关注的是组件；他们从不考虑对整体系统上下文环境的观察。分布式计算环境中的调试要求有整体思维。系统中的每个组件都可以影响该系统中的另一个组件。系统中的组件之间的相互关系可能是高度复杂而无法查明的。当你正奇怪为什么一个组件会失效时，应该问一下这个问题，系统中心哪些其他组件可以使这个组件失效？

- ◆ 考虑次要线索——几年前，我参加一个雪崩训练课。课程中的一项是进行一个指示器搜索。指示器搜索后面的思想是：当一个受害者被埋之后，它的雪崩指示器会发射出无线电信号，这样另一个雪崩指示器就会收到，无线电信号在接收方的指示器上被转换为音频信号。指示器之间的通信规则是靠音频信号的大小来区分的；声音越大，受害者就越近。一开始，我嫌这种搜索有点麻烦。我向四处张望以发现视觉上的线索，而没有注意听跟踪信号。一旦我能够放弃所有的其他感觉而只是跟踪音频信号，我的生存时间就会戏剧性地增加。在调试中同样的尝试也是有用的。在两个例子中，初始的工程师都得到了他们系统中提供的线索。他们不使用这些线索，而去猜想次要的线索，并得出不正确的假设，而且与所有的观察并不一致。在建立假设时，必须考虑所有的线索。

- ◆ 忽略线索——重要的线索被忽略了。在 X 公司，应该要调查后端服务器的超乎寻常的负载，因为机器的输入请求频率没有超出正常范围。在出现问题的两个月期间内，一直都记录着后端服务器处理一个请求和对数据库的访问所用的不可忍受的时间。有许多关于可能发生潜在错误的警告，但由于缺少时间、资源和基础组织，这些警告一直没有被留意，直

到发生了大的灾难。

这两个事件并没有涉及到传统意义上的 bug，但是开发人员还是被找来了。X 公司和 X 公司所面对的问题，在操作人得到足够的提醒时是可以避免的。

◆ 更好的管理工具——在两个例子中，如果操作人员能够检测并留意到数据库服务器的反常现象，他们可能就已经能够采取行动修正数据库服务器。那么所发生的问题可能就得到了避免。

◆ 按照过程进行——在 X 公司，标准过程是在数据库服务器升级时，必须重新对数据库进行索引。在这个例子中，就没有进行重新索引这个步骤，而且也没有关于升级的知识或记录。缺乏记录使得他们很难判断环境和系统中的变化以帮助调试过程。

2.1.2 Therac-25

Therac-25 可能是近年来损失最大的一个 bug。在 1985 年 6 月和 1987 年 1 月之间，已知有 6 个病人被 Therac-25 治疗过量，导致其中 3 人死亡。对于这个报道，我们主要是注意由 Leveson 和 Turner 经几年调查工作后在 1993 年做出的一份报告。这份报告证据充分，包含了许多 Therac-25 事件前因后果的详细而彻底的描述，讨论了系统安全问题的所有方面。但是因为本书是关于软件 bug 的，所以在此我们将把描述和分析限制在与软件有关的问题里。

故事

Therac-25 是一个计算机化的，由加拿大自动能源有限公司（Atomic Energy of Canada Limited, AECL）制造的放射治疗仪器。Therac-25 家族包括 Therac-6，它是一个六百万电子伏特（MeV）的加速器，只能产生 X 射线，Therac-20 是一个 20MeV 的 X 射线和电子加速器，由三种仪器都使用 DEC PPP1 微处理器来完成软件的操作，Therac-6 和 Therac-20 都使用一些硬件措施来避免不安全操作。Therac-6 中的一些软件模块也被用于 Therac-20 和 Therac-25。而且 Therac-25 也使用了来自于 Therac-20 上的关于电子模式的软件模块。

Therac-25 是对 Therac-20 的改进。它站在多个能量水平上传递 25MeV 的光子或电子。它很小，有更多的特征，更加容易使用。它也被设计成比它的前身们更加容易地接受计算机的控制。Therac-25 上的软件用于控制和监视硬件。这样设计人员决定去掉硬件的安全特征，并依靠软件来保证系统的安全。

Therac-25 在 1982 年后期才开始走入市场，在北美洲安装了 11 台，5 台在美国，6 台在加拿大。在 1985 年到 1987 年间发生了 6 起 Therac 过量治疗事故。

Kennestone 地区肿瘤中心，Marritta，1985 年 6 月

一个 61 岁的妇女在经过手术后被送到肿瘤中心接受后来的 Therac-25 的治疗。病人被认为是接受了一剂或两剂的射线，范围介于 15,000 到 20,000 rad（射线吸收量）之间。比较而言，一般的单独治疗剂量是在 200 rad 范围内。Kennestone 自 1983 年以来就一直使用 Therac-25，从没有事故发生，技术人员和 AECL 不相信她的问题是 Therac-25 引起的。最后，

病人由于射线燃烧失去了一个乳房，而且肩部和手臂都已不能使用。

Ontario 癌症基金会，Hamilton，Ontario 加拿大，1985 年 7 月

Hamilton 设备部门在出现过剂量事件之后，已经使用 Therac-25 达 6 个月了。一个 40 岁的妇女来到这里进行她的第 24 次 Therac-25 治疗。Therac-25 在操作员启动机器 5 秒钟之后停止了。很明显，Therac-25 的操作人员已习惯于按照常识操作机器；这些误操作表面上看起来对病人没有坏的影响。因为机器上指示没有放出射线，操作人员就一次又一次地输送射线。持续了 5 次，在第 5 次之后，连技术师也被叫来了，但还是没有发现机器有什么问题。

这个事件被报告到 AECL，但是他们并没有重现出这次误操作，也没有对 Therac-25 的行为做出总结性的解释。但是，从这个报告中，AECL 确实发现了设计上的失误和一些潜在的技术性问题，这些都存在于 Therac-25 的可旋转定位上。并且 AECL 做了一些改进来纠正这个问题。病人在 5 个月后死亡了。医生认为她是因癌症而死亡的，而不是因为过剂量的射线。但医生也表示病人臀部的损伤是过量的放射线造成的，后来估计病人当时受到了 13,000 到 17,000 rad 的辐射。

Yakima Valley 纪念医院，Yakima，华盛顿，1987 年 12 月

一个妇女在 Therac-25 上治疗之后，皮肤上过度发红。医院人员怀疑这是 Therac-25 引起的。但是，他们无法重现出她治疗期间所使用的硬件配置，以和她的症状相联系。医院的工作人员通知了 AECL 关于潜在的过量问题。AECL 再次认为 Therac-25 不可能对一个病人过剂量治疗。医院人员并不知道前面的两次事故，而且也没有能力自己对事故进行调查，所以在得到 AECL 的答复之后就不再追究了。这位病人显然没有前两个病例中的病人受到的射线多，她在过量治疗后并没有出现大的问题。

东 Texas 癌症中心，Tyler，Texas，1986 年 3 月

东 Texas 癌症中心自 1984 年使用 Therac-25 治疗了 500 多例病人。在 1986 年 3 月 21 日，一个男性病人按照要求进行治疗。操作人员已经对 Therac-25 相当熟悉并熟练各种使用方法。当她开始输入病人和剂量数据时，犯了一个错，但立即更正了，然后开始治疗。过了一会，机器就停止运行并显示了错误信息。这个错误信息是软件系统所独有的，别人都不能破译的代码；控制台上显示的“错误操作 54”被翻译为“表单上的‘剂量输入 2’有错误”。东 Texas 医院没有其他的文件来解释“剂量输入 2”的含义。但控制台显示剂量不足，而且由于操作员习惯忽视 Therac-25 的毛病与错误操作，她立即要求用 Therac-25 继续治疗。机器马上又停止而且显示了同样的错误信息。

AECL 被告知这个问题。AECL 派到东 Texas 癌症中心的技术人员不能再产生这种误操作。AECL 仍然认为 Therac-25 不会对病造成过量治疗。在事故发生 5 个月之后，病人因为过量治疗的复杂反应而死亡。

东 Texas 癌症中心，Tyler，Texas，1986 年 4 月

三周后，另一个男性患者来用 Therac-25 治疗。这次负责治疗的还是上次那个操作员。与上一次相同，她又在输入数据时犯了错误，而且很快又改正错误并继续治疗。机器显示了

“错误操作 54”之后关闭了。但是，病人已经治疗过量，操作人员立刻寻求帮助。医院暂停使用 Therac-25，通知 AECL 又一个过量治疗事故发生了。

医院的内科医生 Fritz Hager 与操作人员一起工作，研究如何重现“错误操作 54”信息。很明显，处方数据如果编辑得太快，就会发生过剂量问题。AECL 最终重现了这个错误，并承认 Therac-25 可能会导致过剂量治疗。

过于迅速的编辑变化，为何能导致过剂量呢？Therac-25 可以工作于光子或电子模式。在电子模式中，操作人员必须输入能量水平；但如果选择光子模式，光子的默认能量水平是 25MeV。操作员在两个事故中所犯的错误的相同的。两位受到过量治疗的病人都必须用电子模式治疗。但大多数病人必须用光子模式，操作人员就默认选择光子模式。在这两个事故中，操作人员开始选择的是光子模式，然后又更正了自己的错误。新的物理校准和设置需要 8 秒。在模式和能量参数被设置后，检验数据输入是否完成的软件模块会与磁场校准过程一起运行。另一个软件模块负责管理任何数据条目的改变，这样当一个改变发生时，磁场被重新设置，校准会用新的数据来再次启动，但是，由于这个模块中的一个逻辑错误，使得系统在模式和能量水平被初始输入后 8 秒钟之内，不能发现编辑的变化（这是对软件算法的极其概括的描述，建议那些想知道细节的读者去看他们的报告）。在这两个例子中，当操作员在开始选择光子模式时，机器就开始准备输送能量水平为 25MeV 的光子模式的射线。但是如果操作人员在 8 秒钟以内就把模式改为电子模式的话，磁场就没有被重新设置，因而输送了过剂量的能量。

在知道这个 bug 之后，AECL 立即向用户组织发信建议一种临时的修复方法，即在数据条目操作期间禁止模式编辑。

病人在事故三个星期之后，由于太过量的辐射而死亡。

Yakima Valley 纪念医院，Yakima，华盛顿，1987 年 1 月

到这个时候为止，Therac-25 在公众中引起了巨大的注意；至少是在用户中间，操作人员知道了在编辑中不能做改动。在此期间，AECL 被要求在出现第 6 次事故时，提交一份正确的关于 FDA 的行动计划。在这次事故中，病人将要接受 3 个剂量。头两剂大约要求 3 或 4 rad。第三剂是 79 rad 的光子治疗。头两剂进行得非常顺利。做完第二剂之后，操作人员来到治疗间旋转旋钮以保证射线方案的位置与病人的身体相符合。操作员按下了“Set”按钮以查看位置是否确实恰当，该按钮与旋钮之间很近。当旋钮被设置之后，操作人员继续进行治疗。这一次，机器再次中断并显示“flatness”理由，也即关闭的理由。当时怀疑用户已经过量治疗；但是控制台显示上两次治疗只用了 7 rad 的剂量。

一周后，AECL 在软件中发现了一个可以解释这种行为的缺陷，导致这次事故的软件缺陷在细节上与东 Texas 癌症中心的大不相同。但是，两个缺陷都由软件模块中未预料到的特殊情况所导致。在这次事故中，有个叫做 class3 的共享变量，是一个单字节的值，表示机器参数与治疗的是否一致。如果 class3 的值非 D，那么就有不一致存在，然后就收回光束。变量 class3 的初始化位于那个设置机器进行治疗的模块中。但是初始化靠增加 class3 的值来完

成。因为变量值的长度是一个字节，每当在第 256 次循环时，它的值都将变为 0。进行初始化的模块依赖于系统中的事件而持续运行。当“set”按钮恰恰在变量 class3 返回到 0 时被按下，那么非持续性的检验并没有运行，那么机器就被允许对病人使用 25MeV 的射线柱。

病人在 1987 年 4 月由于过剂量治疗的复杂反应而死亡了，机器在后来就很快被收回了。
教训

在 Therac-25 的生命历程中有两个缺陷被发现。

- ◆ 一个逻辑错误，使得操作员改变了机器状态之后，没有升级机器参数。
- ◆ 当一个 8 位的参数溢出切换到 0 时，安全检查被跳过了。

但是，从一个系统的安全角度讲，最大的缺陷是要依靠软件来保证系统的安全性。显然，Therac-20 中也有与 Therac-25 中一样的错误。那个使得 Therac-25 使用了过剂量治疗的错误，也曾经使得 Therac-20 停机，但不会对病人过量治疗，因为 Therac-20 有独立的硬件安全装置阻止过量的放射线。

作为极特殊情况结果的 bug，是相当难以识别和重现的。在第一个例子中，要求有两个精确的事件来触发 bug：

- ◆ 操作员必须对机器的模式和能量水平作改变。
- ◆ 操作员必须在 8 秒钟之内接受改变。

在第二个例子中，必须要有巧合：bug 只在一个按钮在计数器达到 0 的同时按下。这就是为什么尽管它有内在缺陷而只有 6 个病例被发现。如果 bug 很明显而且很容易触发，那么毫无疑问 AECL 早在其正常测试和质量保证过程中就已经发现它了，bug 就永远不会公开出现了。bug 难于触发的事实，使得在正常测试中很难发现。如果难以让某个 bug 重现，那么它就很难被发现。

注意：许多事件是系统事件；就是说，它们存在于不同的组件和行为之间的复杂的交互中。将一个错误归于一个简单的原因通常是错误的。

除了前面提到的那两个主要缺陷之外，在故事的描述中可以发现大量的问题，它们可能将来造成 bug。

◆ 有一段时期，Therac-25 曾出现错误操作而没有带来什么损失，所以操作人员已经学会了忽略那些奇怪现象。这就好像一个男孩喊“狼来了”——而全镇子的人都习惯性地不理他那个故事一样。频繁的机器关闭和暂停给操作人员带来了不便。操作员从来不会关心由于这些关闭而给病人带来的不良影响，所以他们学会忽略这些问题，一些可能引起严重后果的事情也这样被忽略了。经常性的错误运行，也证明了机器内部的不稳定性和不安全性。

◆ 当错误操作发生时，Therac-25 产生不便理解的信息，给操作人员提供不了任何关于系统错误的原因和结果的线索。系统的设计并没有提供足够的反馈，来让操作人员知道到底在发生什么事情，信息从操作员输入到软件再到硬件。当存在一个断线时，用户无法知道硬件的状态，因为软件没有被设计成允许查询硬件。数据库条目 bug 就是来自缺少反馈的后果的一个清楚例子。例如，如果不能允许操作人员在参数作了改变之后开启光柱，软件就

可以设计成：软件查询完硬件的状态并向操作员报告硬件的配置情况以求验证之后，治疗才能运行。

◆ 当医院向 AECL 联系关于潜在的问题时，AECL 并没有很严肃地看待这个问题，因为 AECL 也无法重现这个问题，那么看起来就会是机器没有这个问题。而且，因为 Therac-25 是基于 Therac-6 和 Therac-20 的，而后两者提供了一般的记录，AECL 开始满足于忽略潜在的问题。尽管 bug 很难检测和重现，但如果给了足够的考虑，那么它就容易识别了。1985 年和 1987 年的 Yakima 事件就是很清楚的例子。当 1985 年报告一个问题时，AECL 进行了例行的检查。AECL 通知医院 bug 是不可重现的，所以没有问题。但是在 1987 年，AECL 就严肃地对待报告了，而且在几个星期内发现了错误行为可能性的原因。只要 AECL 接受自己的产品中可能含有缺陷这个事实，它就能认真地看待问题报告了。

◆ 当数据条目 bug 被发现时，看起来并没有谁去评价整个系统的安全性。在当时总共报告了 5 个病例，数据条目只能解释其中两个事故，都是在东 Texas 癌症中心的，前三个病例并不能用这个 bug 来解释。这就表示系统中还有一些没有描述到的缺陷，机器将来可能会因为一个完全不同的原因而对另一个病人使用过量的治疗。8 个月后 Yakima 遇到了第二个病例。

◆ Therac-25 重用前身们的模块，软件重用使得 Therac-25 更早地投入市场，但也使得 AECL 对系统的稳定性过于自信了。软件重用被软件工业中的人们作为一个增强生产能力和降低缺陷数目的方法。这是对的，但是另一方面，软件重用也意味着 bug 和缺陷是延续的。而且，软件重用会在新的环境让两个组件间建立新的交互关系。新的交互关系可能暴露隐藏的缺陷和新的 bug。在一次使用中没暴露出来的缺陷，在另一个环境中可能突然出现在面前。而且软件通常没有很好的文档，用户很难理解重用模块的意思。有些重用的软件模块可能还会导致拙劣而危险的设计。

2.2 bug 实例 #1

在 2000 年 4 月，一个敏感的 9 页北约文件被发送到伦敦发表，上面的日期是 1999 年 9 月 23 日。显然，在科索沃冲突期间，北约的计算机被塞尔维亚用病毒攻击了。北约的科学家想办法让自己一方的计算机免于更多的攻击，创建了病毒来模拟攻击方式。但实验出麻烦了。实验病毒确实按照设计去运行了。它们从被感染计算机的硬盘中找文档，使这些文档在电子邮件中作为不可见的附件，这样敏感的文档就被公开了。

这个例子中的 bug 并非北约科学家所设计和应用的病毒。这个 bug 是北约的网络环境以及允许发布文档的测试过程。当你的计算机确实按照你所期望的去做时，你恨它吗？

2.2.1 Intel 奔腾处理器的 FDIV bug

在 1993 年，Intel 公司推出了 Pentium™ 芯片，当时认为是 PC 市场的最好的芯片。一年

后这个芯片公布了，Lynchburg 大学的 Thomas Nicely 教授发现了一个 bug 并报告给 Intel。这给 Intel 带来了—个噩梦时期，而且最后 Intel 同意按用户要求更换芯片。公众媒体会让你相信这个 bug 产生的原因是 Intel 的工程师和测试人员的愚蠢——毕竟，测试一个表格中的每个值是多么难啊？和往常一样，现实远远比报纸上的新闻评论要复杂得多。

故事

Thomas Nicely 教授是一个数学家。在 1993 年和 1994 年间，他在做一个计算数学理论领域中的一个研究项目。他的一个目标是演示桌面 PC 的工具。为了进行研究，他购买了大量的 PC 来枚举正整数中 6×10^{12} 以下的所有的素数、双素数、素数三元偶和素数四元偶。素数是只能被 1 和其本身整除的正整数（如 3、…）。双素数是连续两个奇数都是素数（如 3 和 5…）。素数三元偶是连续三个奇数都是素数（如 3、5 和 7）。在计算过程中，他的程序也要进行一定数量的检验，计算那些已公布的值，来看正在进行的计算是否正确。在 1994 年添加了一台 Pentium 计算机，在 1994 年 6 月 13 日，一次检验中出现未预期到的值。经过几个月的调试，Nicely 教授已经能把问题跟踪到 Pentium 机器的浮点处理单元（FPU）。在 1994 年 12 月 9 日的一个公开发言中，他描述了他的实验，以及在判断错误的原因时遇到的困难。

当他对白己的分析有把握之后，Nicely 教授向 Intel 的技术支持部门递交了一份 bug 报告，时间在 1994 年 10 月 30 日。Nicely 向他的几个同事写信指出他发现了 Intel 的 Pentium 芯片中明显的浮点除法的 bug。在电子邮件中，他描述了他的发现：

“简而言之，Pentium 的 FPU 对某些除法返回了错误的值。例如：

1/824633702441.0

这个计算的结果是错的（第 8 位之后的数值都是错的）。”

很快，bug 报告在 CompuServe 论坛和互联网新闻组之间传播。Alexander Wolfe 是《EE Times》的一名记者。他看到了这个故事，并写了一篇文章在《EE Times》1994 年 11 月 7 日的一期中发表了。作为反应，Intel 宣称它在 1994 年夏季就已经发现了 bug，在后来推出 Pentium 芯片中已经作了修复。但是 Intel 无法判断已经出售的有缺陷芯片的数量，而且他们准备忽略这个 bug 带来的影响。

Intel 的 Smith 先生强调少量的异常不会影响全部的用户，谈起 Nicely 时，Smith 说：“他是最特殊的用户。他把所有的时间都用来计算倒数。他工作了几个月之后的发现只是除了 8 位正确，1 位不正确而已。所以你可能在尾数右面第 9 位得到一个错误，我想即使你是一个工程师，你也不会注意这些的。”

CNN 在 1994 年 11 月 22 日广播了这个故事，然后这个报道就出现在各大主要新闻媒体中，例如《New York Time》和联合新闻。当接受采访时，Intel 再次重申以前的立场，即这个 bug 对一般用户没有影响。

Intel 说它不认为芯片需要回收，宣称一般的用户由于这个错误而面对一个不精确结果的机会小于 90 亿分之一，这样对商业和家用计算机不会造成什么影响。实际上，这个公司说它将继续向计算机制造商出售那些在发现缺陷之前制造的芯片。

在 1994 年 11 月 28 日,“Vitesse 半导体”的 Tim Coe 在 comp.sys.intel 新闻组中发表了一篇文章,文章中他研究了算法并提出了 Pentium 芯片的一个行为模型。几天后,一个关于 Pentium 的 FDIV bug 的软件和硬件工作组成立。在 1994 年 12 月 3 日,Stanford 大学一位教授 Vaughan R. Pratt 在 comp.arch 和 comp.sys.intel 新闻组中发表了一封信,信中他讨论了 Intel 的“出现 bug 的可能性是每 27000 年一次”的观点。他能够证明他可以在合理的情形下将这个 bug 每 30 毫秒触发一次,而且看起来平常的一个除法 $4.999999/14.999999$,用有 bug 的 Pentium 芯片计算出的结果是错误的 0.00000407。

在 1994 年 12 月 12 日,IBM 发表了一个报告,报告中质疑了 Intel 的“出现错误的机率是 90 亿分之一”的说法。

1994 年 12 月 14 日,Intel 发表了标记日期为 1994 年 11 月 30 日的白皮书。

这篇白皮书检讨了这个 bug 以及它的影响,显然已经招致了许多批评。在这篇文章中,认为遇到这个 bug 的可能性是 90 亿分之一,而且由这个 bug 导致错误的平均时间应该是 27000 年一次。而且,Intel 的工程师选择了一个基数 4 的 SRT 算法作为 Pentium 处理器的除法算法,这样 Pentium 芯片除法的处理速度就比 486 处理器快了一倍。对 Pentium 芯片中使用的 SRT 算法的详细描述已经超出了本书的范围,有兴趣的读者可以参阅 Edelman、Sharangpani 和 Bryant 三人的书。

但是这个 bug 的症结在于,SRT 算法要求要有一个详细的查看表单以判断结果。表单中的值是计算出来的,并被输入到可编程的参照阵列中。脚本中的一个缺陷导致了查看表中一些条目被忽略掉。当一些除法需要用到这些已经被忽略的条目时,就会得到一个错误的值,从而计算结果的精确度就下降了。

最后,Intel 在 1994 年 12 月 20 日,宣布它将开始使用更高的要求来制作 Pentium 芯片,从而最终使得关于 Pentium 的 bug 的问题消失。但这并没有阻止人们对 Pentium 的 bug 的分析。在 1995 年 9 月 19 日,Alan Edelman 发表了一篇报告,报告中他对 Pentium 的 FDIV bug 进行了一次详细的分析。

在报告中他判断出,在计算中查看表中的有 bug 的条目,只有两种被访问和使用的方式。有缺陷的条目只有当除法器遇到从第 5 位到第 10 位都连续的数时才会被访问。而且有缺陷的条目不可能都被随机地访问。在选择测试例子中一个正常的测试可能比随机的测试要好得多,他同时指出这个 bug 产生的差错的绝对值的最大值不大于 0.00005。

教训

Intel 的 Pentium 是由一个容易做到却不容易发现的错误导致的。这是由于两个原因。第一,这个错误是一个最大误差为 0.00005 的 Pentium 芯片运行的除法所导致的。我们有多少人会在意 0.33332922 和 0.33333329 之间的区别呢?而且,如果我们使用一个将其自动舍入取为两位有效数字的应用程序时呢?我们甚至不会注意到小的差别。就和 Vaughn Pratt 写的那样:

“这个 bug 非常之隐秘,它们可能在人们浏览数据时几乎不会被触发。这样,十万分之

一的小错误可以在相当长的时期内在上万亿次计算中溜掉，而且由于很少做大量详细监控的能够遇到 Pentium FDIV bug 而对可靠的浮点处理是完全不必要的一些运算，使得没有能够发现这些 bug 现实的途径。”

这个 bug 难以捕捉的第二个理由是缺陷很难触发的。

在一个随机性的测试中，想要访问到有缺陷表的条目，其机会是很低的。Intel 是在它发行了有缺陷的芯片将近一年之后，才自己独立地在测试中发现了 bug。这就说明了测试和捕捉 bug 的难度性。

对于 Nicely 教授，bug 识别过程的开端是因为他发现了细微的不一致，但是他坚持注意这个不一致性的原因，是他在代码中添加了检验程序以确定计算的正确性。他在程序中提供了一些机制，使得自己能控制程序的正确性。如果没有这些检验功能，他可能不会这么早就发现这个 bug，而且他所枚举的素数会有错。在发现了存在的问题之后，他就开始系统地消除可能产生 bug 的原因，并将原因限制到 Pentium 的 FPU 上。

从 Intel 的损失中，我们可以得到的教训很多：

- ◆ 测试不仅需要遵循要求，它还要描述所使用的算法来保证最大的完备性。
- ◆ 我们在程序中必须建立一些机制来持续地监视程序的正确性。这会提供给我们尽早尽可能地识别潜在 bug 的机会。
- ◆ 要对所有出乎意料的结果进行质疑和探测，直到我们能够解释其理由。可能会有隐藏的 bug。
- ◆ 发现一个 bug 的潜在的根本原因是一个系统化的过程。首先假设出来一个潜在的原因，然后开始实验以验证这个假设。
- ◆ 建立一个简单的可重现的实例，有时能带来巨大的区别。Nicely 教授能够用一个简单的除法触发 bug，这就使得他能测试其他的系统和配置，帮助他排除错误的假设。

2.3 bug 实例 #2

根据传言，早期的鱼雷上设计有一个自动防故障装置，以阻止在鱼雷发射后破坏潜艇。发射之后，鱼雷被设计成在发生 180 度转向时自毁。这个设计背后的理论是：如果鱼雷在发射后转向 180 度，则有可能攻击发射自己的潜艇。一天，一个潜艇上尉决定发射一枚鱼雷。但是，鱼雷被卡在发射舱里，不能被释放了。上尉决定让潜艇返回港口以进行修理。当潜艇自身进行 180 度的转弯时，鱼雷自毁了，并且将发射自己的潜艇也报销了。

这个例子中的 bug 存在于鱼雷的安全性设计中。有时候，一些人认为在鱼雷中加入一些自动故障装置是一个不错的想法，因为这样它就不会破坏发射它的潜艇。这个想法听起来有意义，但设计上不只这么简单。设计者无法考虑到潜艇下沉时的具体情形。但是对于我们来说，这个事件是否是早期潜艇武器设计的一个 bug 的证明，或者这个事件只是一个传言，这还不很清楚。但我们在讨论这个事件时，会觉得既可笑又现实。

2.3.1 Ariane 5 操作数错误

Ariane 5 的小故事是在将一个 64 位浮点值转换为 16 位的有符号整数值时，因一个没有被解决的异常而引起的事故。引起这次异常的浮点值其大小超出了 16 位整数的表示范围（这次例外在官方的调查报告中被称为操作数错误）。这个故事的更长的版本非常有趣而且有教育意义。

故事

按照欧洲空间机构的期望，Ariane 5 是要成为商业太空市场的主要发射器。Ariane 5 耗资 70 亿美元，历时 10 多年才建造成功，能够将 3 吨重的卫星带到轨道。

在它的首次飞行中，Ariane 5 火箭 1996 年 6 月 4 日发射的 40 秒之后爆炸了。对飞行数据的分析后，很快就发现它在突然改变路径并自毁之前一直运行正常。测试飞行的那天上午的天气还算可以，所以天气不是一个因素。飞行数据还显示活动的主惯性导航系统在火箭自毁以前都几乎同时失灵了，而此系统主要影响固体调压器喷口的控制。

在事故发生后，组织了一个调查委员会来调查这次失败。调查委员会拥有火箭的遥测数据，来自雷达站的轨道数据、发射的观测信息以及收回的碎片，还有收回的惯性导航系统。委员会就依靠这些完成查找 bug 的任务。而且，调查委员会也有用来测试和参考的单个组件和软件系统。有了这些信息之后，它就可以重建 1996 年 6 月 4 日的事件序列。

1. 最终发生错误的软件模块是重用自 Ariane 4 火箭的。这个模块负责管理惯性平台队列，这样惯性导航系统中所进行的计算的精度就可以被估算出来。发射之后，这个模块对 Ariane 5 没什么意义。但在 Ariane 5 上，这个模块还要工作满 50 秒。Ariane 5 的飞行轨道的早期部分与 Ariane 4 的很不相同，而且这个软件模块从未正式地测试过。

2. 发射之后不久，这个错误的软件模块想要计算一个基于火箭的水平速度的一个值。但这个值在 Ariane 5 上比在 Ariane 4 上要大得多，这样在后备的和主要的惯性导航系统上都出现了一个操作数错误。转换的有效性没有被检查，因为它从未想到会发生。

3. 关于对系统错误进行处理的描述，表明失败时的上下文环境可以在关闭处理器之前存入 EEPROM 存储器中。在出现操作数错误之后，惯性导航系统按照所描述的情形设置了失败时的上下文环境，而这些将被火箭上的计算机读取。根据这些值，火箭上的计算机向固体稳压器喷嘴和主发动机发送了一个命令。这个命令使得喷嘴完全偏斜，导致火箭飞上了一个奇怪的轨道。

4. 新的轨道使火箭处于极端的空气动力负载之下，这样火箭就开始分裂了。调压器从主火箭中分离，从而触发了自毁系统。

不可否认，这是一个无声的错误，但有趣的问题是：这个错误是如何通过测试的呢？空间工业显然有很高的标准和详细的过程与程序来保证安全性，因为失败时的代价是巨大的。调查委员会也提出同样的问题，但 Ariane 5 的工作人员提供了如下的基本原理：

- ◆ Ariane 5 的工作人员决定不去保护一些可能有潜在操作数错误的变量。因为他们感

觉这些变量的值在物理上是受限的，或处于不影响安全性的范围。

- ◆ Ariane 5 的工作人员决定不把轨道数据作为惯性导航系统的功能要求的一部分。这样 Ariane 5 的轨道数据在测试中就没有被使用。

- ◆ 很难使用惯性导航系统进行现实的飞行，因为有物理上的规则。在飞行软件的功能模拟测试中，工作人员决定在测试里不加入实际的惯性导航系统，主要的原因是惯性导航系统应该已经在装备测试中验证过了，而且在模拟测试中如果使用了真正的惯性导航系统，就很难达到期望的程序。

教训

就和我们前面所讨论的 bug 一样，Ariane 5 的失败并只是因为一个原因。在整个开发和测试过程中有许多阶段里可以发现一些缺陷。

- ◆ 一个软件模块被重用于一个新的环境中，此环境中的操作情况与原来的软件模块中的要求不一样。这些要求没有被注意到。

- ◆ 系统捕捉和识别了错误。不幸的是，关于错误运行的机制的描述是不恰当的，而且导致了最终的灾难。

- ◆ 错误的模块在新的环境中从没有被恰当地测试——在设备水平没测试，在系统集成水平也没有测试。这样，错误的设计和应用就永远也没有暴露出来。

调查委员会的报告包括以下观察，我们相信它们是高度正确的，不仅对于 Ariane 5 的设计者，对整个软件工业也是这样。

“Ariane 5 的开发时一个潜在的问题是减轻随机失败的偏见……这次所发生的意外不是因为随机的错误，而是因为设计上的失误。这个错误被检测到了但没有被恰当地处理，因为当时认为软件在没有表现出错误时，就可以算作是正确的……委员会却支持相反的观点，即在应用目前所能得到的最好的经验模式可以证明软件是正确的之前，软件应看作是有错误的。”

但是，调查委员会能够成功地找出错误的原因是拥有大量的测量数据、模拟的环境和文档。没有这些数据，就很难将天气的影响排除在外。没有遥测和飞行数据，将很难判断轨迹变化的时间、信息以及错误的惯性导航系统，它使得委员会能立即缩小潜在错误的范围。飞行后的模拟是通过使用来自于火箭飞行的实际数据进行的，而且这次模拟令人信服地再现了导致系统失败的事件链。所以调查委员会能够再现这个 bug！

2.4 bug 实例 #3

互联网上有大量的信息，很遗憾，这个“大量”不是用来描述信息质量的形容词。相反地，“大量”意味着信息的数量——太多的信息。有些信息在一些环境下是不需要的，我们称之为“Spam”。在其他的例子中，对诸如色情图片之类的特殊形式信息的访问，因为许多原因要受到限制和审查。无论如何，都有这样一些工具，其职责是作为过滤器，删除那些潜

在不希望的邮件并阻止对某些网络站点的访问。例如：

“刚刚在 CBS 网络收音机上听到这样一个报告说，国内网上的橄榄球迷们无法得知关于星期天的超级碗杯的情况。看来安装在浏览器或其他东西中的网络过滤器软件发现了“超级碗杯 XXXIV”中的“XXX”，并把它解释为一个成人色情站点。”

在另一个例子中，对潜在可能发生问题的词语的删减，可能会使一个本来很明白的消息变得不可理解甚至是有趣。例如，BBC 的一个互联网服务删减了下面的句子：

“I hope you still have your appetite for scraps of dickens, when I bump into you in class in Scunthorpe,Essex ,on Saturday”

变为：

“I hope you still have your appetite for s****s of dickens, when I ****p into you in s****horpe,Es***,on Sa*****ay”

但是最无辜的情况是些反垃圾软件，它们可以删除重要的邮件，因为软件把它们错误地认为是垃圾。

我最近升级到了 MS Office 2000，其中允许同时拥有 8 个活动的邮件过滤器。在兴奋中，我设置了所有的东西，包括垃圾邮件过滤器。意外的是，我发现了 8 至 10 个重要的消息，它们都是来自我对个人邮件列表中的人所发查询的答复——都被扔在了垃圾邮件文件夹中。

怎么回事？我当时工作在一个慈善机构，我需要向他们告知我的请求，即我需要他们的资金。所以我给他们发了邮件，以更改我的训练状态，并请他们寄来支票。很明显，这个邮件里面至少有一个美元符“\$”——而且因为我是一个容易激动的家伙，我在里面至少加入了一个双重感叹号“!!”，而且我在最后责备了我的管理员，希望他认真对待我对他所作承诺的热烈期盼。

“>>Mark，你不是曾经许诺给我 5,000\$吗？”

……我们也同时过滤掉了“0,000”这个词。

现在，如果这三个元素出现的话，你就有了一个垃圾，实际的规则（他们网站上的）是：

“正文包括“0,000”者，正文包含“!”者以及正文包含“\$”者。”

谁猜得到呢？实际上，即使我可以查看他们的过滤器列表，仍然花了很长时间来看究竟引发了哪个过滤器（是的，有时候我的确很迟钝）。

这些是 bug 呢？还是用户的错误？在这三个例子中，软件按照算法的要求正常运行，但软件的制作人不会将这些例子看作 bug。但是软件的效果，有时候并不像最终用户所期望的那样。所以作为最终用户，我们是关掉过滤器和反垃圾软件来做到我们想做的事情，还是想办法修改算法的错误呢？在第 3 章中，我们将研究 bug 的本质，这将能够帮助我们判断什么是 bug。

火星气象卫星

“火星气象卫星”是火星探测（Mars Surveyor）计划的一部分，目的是探测和描述火星。

火星探测计划持续 10 年,平均每年发射一个飞行器。头两个飞行器在 1996 年发射,分别是“火星探路者 (Mars Pathfinder)”和“火星全球探测器 (Mars Global Surveyor)”。“火星气象卫星 (Mars Climate Orbiter)”在 1999 年 1 月 3 日发射。“火星气象卫星”和“火星极地登陆者 (Mars Polar Lander)”在它们到达火星之后不久就都消失了。这两个空间飞行器的建造和操作花费 NASA 大约三亿二千七百六十万美元。这两次失败迫使 NASA 对其目标、过程和火星计划进行重新评估,以保证其将来的飞行器的成功。“火星极地登陆者”失败的原因仍然不能确定,而导致“火星气象卫星”丢失的错误原因已经确定。接下来我们就重点研究“火星气象卫星”。

故事

“火星气象卫星”在 1998 年 12 月 11 日,在 Florida 的 Canaveral 角的航空站由一个 Delta I 火箭运载发射。经过 9 个半月的空间飞行,计划于 1999 年 9 月 23 日进入预定轨道,但是当计划的时间到来时,有些东西出了问题。

“今天上午早些时候,大约在太平洋时间凌晨 2:00,人造卫星点燃了它的主发动机进入火星的轨道。所有来自飞行器的信息都表明一切正常。发动机按计划再次点燃是在卫星将要飞到从地球所看的火星背面之前 5 分钟。飞行控制器再也没有探测到一点在卫星将从火星背后出来时的信号。”

“火星气象卫星”由科学家和工程师组成的队伍开发,分两个地点:喷气推进实验室 (Jet Propulsion Laboratory, JPL), 位于 California, 而 Lockheed Martin 空间飞行学 (Lockheed Martin Astronautics, LMA) 工作室, 位于 Colorado。LMA 负责“火星气象卫星”开发中的飞行系统的集成与测试和发射操作支持。JPL 负责项目管理、项目系统工程、飞行设计、导航设计、飞行器操作系统开发,地面数据系统开发以及飞行的保证。

在它的 9 个半月的飞行中,地面工作人员跟踪了飞行器的观察轨道,并将其与期望的轨道比较。而且地面人员也跟踪了卫星上面所发生的所有的事情,其中一个角度动力衰减事件 (Angular Momentum Desaturation, AMD), 当卫星点燃推进器来消除调速轮中内在的角度动力时,发生了一次 AMD 事件。基本上,这是一次对动力的校准,以保证系统的组件在一个期望的范围内运行。当一个 AMD 事件发生时,会发生以下的动作序列:

1. 卫星向地面场地发送数据。
2. 数据被一个叫做“SM_FORCE”的软件模块处理。
3. 该软件模块的处理结果存储为一个叫做 AMD 的文件。
4. AMD 文件中的数据被用来计算卫星的速度变化。
5. 计算出的速度变化被用于建模卫星的轨道。

根据描述, SM_FORCE 模块可以使用公制单位产生数据并存到 AMD 文件里,即牛顿。但是因为某种原因,地面站的 SM_FORCE 模块用英制单位输出数据,即磅。(官方的报告并未指明原因,所以我们也不便去推测。)计算速度变化的软件模块使用了 AMD 文件中的数据,根据描述,是使用公制单位。这就导致了飞行器和地面站之间计算出的轨道间的差别。

而且地面站计算出的轨道的值，比飞行器乘以 4.45 计算出的值远远地要小，因为 1 磅等于 4.45 牛顿。

飞行器定期向地面站传送它计算出的轨道模型的数据以作比较。从理论上讲，一个对飞行器中的模型和地面站之间的模型之间的迅速比较，可以导出一个警告，但是一些复杂的因素使得地面工作人员没有意识到错误。

- ◆ 在地面站的软件中有一些 bug，所以地面工作人员无法使用 SM_FORCE 模块计算卫星的轨道。这些 bug 在飞行了 4 个月之后，即 1999 年 4 月才得到修复。

- ◆ 负责飞行器导航的地面工作人员不清楚来自飞行器的速度变化数据是否可以用于比较，这情况一直持续到发射了很久以后。

- ◆ 飞行器和地球间的视觉限制使得地面人员无法利用观察对飞行器的轨道精确地建模。

如果 AMD 事件很少发生，那么 4.45 倍的差别就不会造成这么大的影响。但是由于飞行器的外形，AMD 事件发生的是预期的几倍到 14 倍，而且，当地面站模型、飞行器模型和观察信息三者之间的差异被发现时，这个情况是非正式地用电子邮件而不是按照这个项目的标准过程报告的。最后，在飞行器消失之前，这个问题一直没有解决。

在 1999 年 9 月 8 日，地面人员计算了将卫星送入火星轨道的某个动力，这次计算使用了错误的模型。这个动力的目的是校正卫星的轨道以保证离火星的最近距离是 226 公里。在 9 月 23 日，这个动力被开始执行。在 09:00:46（统一时间 UTC）启动了发动机，这个时间亦即格林威治时间（Greenwich Mean Time, GMT）。4 分 6 秒之后，地面站就看不到卫星了。信号的丢失是因为卫星已经到火星的背面，但这比预期轨道模型中描述的要早 49 秒。因为使用了错误的轨道模型，飞行器实际上离火星的表面比预期的近得多。卫星到火星表面之间的最短距离，实际上只有 57 公里。这种高度，对于卫星来说实在是太低了。

教训

“人们经常犯错误”，NASA 的空间科学副总管 Edward Weiler 博士说。这里的问题并不是错误，这是 NASA 系统工程的失败，而且是我们处理过程中的检查和平衡以及对错误检测的失败。

在软件的某个部分出现了一个错误。但是在 9 个半月的期间内出现的一系列的错误最终导致了“火星气象卫星”的灾难。对初始错误根本原因的探索发现，实际上这个事件并不是一个单独的错误的结果，这次卫星的失败有三个主要因素：

- ◆ 如果进行了测试，这些错误可以迅速被发现和纠正。

- ◆ 但是，由于地面站的软件没有正确地测试，所以卫星发射之后，出现了另外的 bug，这些 bug 延误了对问题的观察。这些 bug 是后来出现的，没有给工作人员以 9 个月的时间来发现它，而是将地面人员跟踪异常现象的时间缩短到只有 5 个月。

- ◆ 当发现到异常时，它没有被恰当地解释。初始的 bug 也从来没有被确定，而它本来可以早就提醒地面人员这个灾难的。

调查委员会的第一阶段报告也指出了导致灾难的以下因素，并提供了一些附加的建议。这些程序大部分与系统的某些程序和过程有关，就是那些地面软件系统。

◆ 交流——在任何大规模项目中，交流始终是一个大问题，无论是在软件还是其他项目中。在“火星气象卫星”的例子中，项目的工作人员之间并没有进行有效的交流。操作人员在发现轨道的差异后，并没有把这个顾虑与飞行器操作人员和项目管理人员交流。一个工作组中关键性的知识，没有充分地输送或分享给另一个工作组，这在一定程度上导致了失败。

◆ 从开发到操作之间的培训和过渡——轨道模型中的问题一直到事故发生后才得到解释，其中的一个原因是操作人员的培训不够；从开发项目到操作项目之间的迁移过渡，并没有被详细地解释或执行。在一个软件项目环境中，这就相当于软件开发人员将完成的系统直接交给网络操作人员，而没有提供给他们那些工具和针对系统特征和行为的培训。

◆ 分析什么会出差错——调查人员认为，对系统的失败情况的预先分析，会帮助避免未来的事故。这类似于网络中常见的分布式计算环境中进行的失效恢复、容错和容量计划的分析。

◆ 自负——调查委员会发现，项目负责人认为将一个飞行器发送到围绕火星的轨道是一个容易的任务，因为 JPL 有 30 年的行星间飞行导航经验，并且历史上没有任何导航的错误。这种自负心理也许可以解释为什么软件模块缺少恰当的测试。

2.5 bug 实例 #4

这个事件与任何类型的软件都没什么关系。实际上，这个故事根本不涉及什么软件。但是它很有趣，很有教育意义，在本章中值得一提。这个故事的简明版本说的是一个转包商，他从事为本地电话公司铺设光缆的工作。在事故发生前，他弄断了电话线，并用一台挖沟机在地上挖沟。但是，由于早上的露水弄湿了地面，挖沟机从路上滑到了—一个沟里，把这位转包商撞到了 10~15 英尺深的沟里。看到这个事故后，一位旁观者跑到最近的房子去拨打 911。但电话不能用了，为什么？因为这位转包商为了施工将电话线弄断了。令这个事故更加可笑的是，这位转包商的卡车上有一个手提电话，但一起工作的人也受伤了，在发生事故时彻底忘了这个电话。最后，另一个旁观者在另一所房子里成功地拨打了 911 电话，这样这个事件就有了一个愉快的结局。受伤的人被送到医院、没有大伤。挖沟机从沟里吊了上来，光缆也铺成了。

我们从这个事件中可以学到哪些教训，以应用于软件调试中呢？主要是——必须有一个备份。通常情况下，在我们热情洋溢地修改和变化代码时，我们对自己所做的改动总是很自信，认为可以不需要测试。要认识到改动是很难恢复原样的；而如果存在问题的话就麻烦了，这就好像那个转包商一样，在换上新的电话线之前弄断旧的电话线，而没有考虑到可能发生的问题。到了发生问题后，根本没有电话可以用。

对源代码要做一个备份，对那些要替换掉的代码作上注释，而不是删除它们；对旧的函

数作一份拷贝，并取个不同的名字，以及使用一个配置管理工具。上述这些方法是成功地测试完所有改变之前所应采取的工作。但是备份在被完全忘了的情况下，是没什么意义的。在这个故事里，如果有人能记得卡车里的手提电话的话，救援工作能够更快地开展起来。在我们的工作中，危险并不在于我们忘记了备份，而在于试图发现哪个版本的备份（或者来自于不同模块的版本的组合）中包含原先了解的工作系统。

2.5.1 AT&T 电话中断

在远程通信工业中，服务的丢失被叫做中断（outage）。对于我们大多数人来说，当发生一次中断时，我们就丢失了电话服务；我们既不能打电话，也不能接电话。中断被人们所接受，并且在工业中被认为是危险的。狂风可以吹断电话线，地下电缆可以被人为原因或地震破坏，或者在网络的管理软件中可能有 bug。

在 1990 年 1 月 15 日，AT&T 发生了全国范围内持续 9 个小时的电话系统中断，其原因是软件中的一个程序错误。该错误的出发点是让系统更加有效。8 年后，在 1998 年 4 月 13 日，AT&T 在其帧中继网络中又出现了一个巨大的错误，影响了 ATM 机器、信用卡交易和其他的商业数据服务，这次失效持续了 26 个小时，这次的 bug 也是在一次软件升级中出现的。

难道 AT&T 两次都犯了同样的错误？除了这些 bug 还有其他因素吗？

故事：1990 年的中断

在 1990 年 AT&T 电话网络由 114 个相互连接的 4ESS 计费交换系统组成（大致如此）。为了便于讨论，我们从概念上将其网络建模为一个图。在图中有 114 个节点，每个节点代表 1 个 4ESS 交换机。节点间画的线条代表节点间的管理通信。

在这个电话网络中，当任何一个节点经历一个问题时，发至错误的节点就向与它连接的所有节点发送“请勿打扰”的信息。这个消息通知那些相邻的节点，这个发生错误的节点不能处理新的呼叫，并让它们把自己标记为不可用节点。同时，发生错误的节点启动一个错误恢复过程，发送一个“初始地址消息”（Initial Address Message, IAM），发给与它相邻的节点，告诉它们自己的新状态，并让它们把正在送来的呼叫转向“正在工作（now-working）”的节点。

在 1989 年 12 月中旬，AT&T 在 4ESS 交换机中升级了软件以提高网络的性能，并且增加了在网络出现问题时启动更快的失效恢复过程的功能。在 1990 年 1 月 15 日美国东部时区大约下午 2:30 的时候，纽约的 4ESS 交换机发生了一个小的硬件问题，所以它就如前所述启动了错误恢复过程。在纽约的交换机修复了自己的问题之后，它向相邻的节点发送了一个 IAM 告知它们自己可以重新服务了。但是 12 月中旬的软件升级中出现了一个缺陷。这个缺陷发作的时机是：当一个交换机在 1/100 秒的间隔内接到两个 IAM 时——交换机中的一些数据被破坏，交换机也暂停工作以重新初始化。当相邻的交换机停止服务时，它们也进行相同的恢复程序。因为所有的动作都相同，那么相同的事件流从一个交换机流向另一个交换机，

这样就使整个网络瘫痪了。

在一整天中，AT&T 的工程师们通过减少网络负载来稳定网络。到美国东部时间晚上 11:30 为止，他们清除掉了网络中所有的连接，系统就运转正常了。

在 1990 年 1 月 16 日，星期二，工程师们已经能够识别和分离出来 bug 了，它是一段错误的代码。错误的代码在交换机恢复过程中被激活。导致中断的代码片断如下：

```

1. do {
2.     ...
3.     switch (expression) {
4.         case (value 0): {
5.             if (logical_test) {
6.                 ...
7.                 break;
8.             } else {
9.                 ...
10.            }
11.            ...
12.            break;
13.        }
14.        ...
15.    }
16.    ...
17. } while (expression);

```

在这个例子中，第 7 行的中断语句是罪魁祸首。正如应用中表现的那样，如果第 5 行的逻辑测试成功的话，程序就会从第 6 行开始执行。到第 7 行时，中断语句使得程序跳出了第 3 行和第 15 行的 switch 块，去执行第 16 行的代码，但这并不是程序所期望的执行路径。编程人员的本意是程序执行第 7 行之后，期望执行是第 11 行，表 2.1 说明了实际中和期望的程序路径的区别。

表 2.1 导致 1990 年 AT&T 中断的实际执行的指令与期望的指令

流	实际	期望
步骤 1	行 2	行 2
步骤 2	行 3	行 3
步骤 3	行 4	行 4
步骤 4	行 5	行 5
步骤 5	行 6	行 6
步骤 6	行 7	行 7
步骤 7	行 16	行 11 (错误)
步骤 8	行 17	行 12 (错误)
步骤 9	—	行 16 (错误)

教训：1990 年的中断

导致 1990 年的中断的程序设计的错误，是一个典型的新手错误。但我们都可能犯错误，即使一个有 20 年经验的老手也有可能偶尔发生可笑的错误。既然错误是不可避免的，问题就是我们怎样才能在错误出现在公众面前之前把它找出来呢？我们并没有关于 AT&T 在 1990 年的内部的软件开发过程中的第一手资料；所以我们也无法评价其他的因素。官方的 AT&T 报告指出：

“我们相信我们所使用的软件设计、开发和测试过程是基于坚实高质量的基础上的。所有已经发行的软件将继续进行严格的测试。我们将利用在这个问题中得到的经验，在将来改进我们的程序。”

我们无法确信能够找到 AT&T 在 1990 年的中断中软件开发过程的缺点，也没有理由怀疑它在其软件的升级版本中是否会严格地进行测试。事后再看，显然如果开发人员只是测试了该软件，那么他们将来可能就会遇到 bug。如果他们能够进行一次代码检查，他们将能够发现这个缺陷。在这个例子中代码检查可能会起到作用。但是通过代码检查发现这个 bug 的惟一途径是：另一个工程师发现了这行特殊的代码，并向原来的编程人员询问运行代码的意图。而他问这个问题的惟一理由，是他对这个代码块是否很熟悉。

这个特征的 bug 通常容易测试，在一个正常的测试环境中也容易发现。如果你能理解它，并可以产生触发它的必要的事件序列的话，就可以让它再现了。但是随机也能产生正确的事件序列的机会很小，尤其是系统将使用于一个实时的、大规模的、难以在实验室中模拟的环境中时。而且新的软件已经正确地运行了一个月，处理了上 10 亿个的电话业务。软件的确存在一个缺陷，但这个缺陷存在的前提是一组特定的事件和因素。

- ◆ 这个 bug 需要在网络上有一个固定的负载。负载减少时，这个 bug 的影响就消失了。
- ◆ 这个 bug 时间很强。两个来自于同一交换机的 IAM 需要在 10 毫秒内接收才能触发这个 bug。
- ◆ 所有的交换机都有相同的软件，这就使得系统的大小可以变化，但是，风险是所有的交换机有同样的缺陷，对同一个 bug 敏感。

故事：1998 年的中断

1998 年 4 月 13 日，下午 2:30，一个技术师被派去升级一个装在 Cisco stratacom BPX 帧中继交换机上的主干卡。这种帧中继交换机有一对冗余的主干卡，一个是活动的，另一个作为备份。AT&T 使用了两个过程来进行主干卡的升级。一个过程假定交换机当前是活动的并与网络连接，另一个过程假设交换机独立于网络之外。

在第一种情况中，交换机与网络互连，该过程要求技术人员首先换下备用的卡。一旦确定新卡是稳定的之后，旧的活动卡将被设为备用，而新换的卡成为活动的。之后技术人员就可以升级那个剩下的卡了（即现在的备用卡）。第二个过程假定交换机没有与网络连接，技术师就可以自由地同时换下这两块卡。

当技术师来到设备旁边准备进行升级时，技术师认为待升级的交换机不在网络上，因为

它看起来并没有什么网络负载。但是这个交换机确实是在网络上，而且是活跃的。不幸的是两个卡都有缺陷。在主干卡被安装并激活以后，它们立即向交换机发送了一系列的错误消息。来自主干卡的错误消息触发了交换机软模块中的一个缺陷。这个缺陷导致了错误的消息向网络中其他交换机的传播，包括 145 个即所有的交换机。错误消息的数量太大了，所有的交换机很快就超载，最终在下午 3:00 的时候弄垮了整个网络。

当时并没有关于主干卡或 Cisco 交换机软件错误的信息。Cisco 系统的软件质量工程的管理者 Alka Travis，在 1998 年 3 月 29 日的国际软件质量周会（International Software Quality Week session）上指出，导致这次网络中断的代码是一些旧的代码。

AT&T 立即迅速地将出错的交换机分离，到晚上 11:00 时错误的网关已经从网络分离了。剩下的任务就是重建整个网络系统了。到 1998 年 4 月 14 日凌晨 2:30，99.9% 的帧中继网络再次运行。但是还是花了大约一个星期的时间才判断出中断的原因。4 月 22 日，AT&T 发布了一个报告，描述了这次网络失效的原因。

教训：1998 年的中断

虽然在 1998 年的网络中断中有软件缺陷的因素，但还有些导致中断的其他因素。这次中断和 1990 年不同的是，在升级过程中的一个程序错误触发了潜在的软件缺陷。但是它们还是有很多的相似性的。

- ◆ 新软件的安装是一个触发因素。软件无论是新的还是旧的，在正常的软件测试过程中，都有一些潜在的缺陷没有被发现。在操作环境中发生了一些变化，从而触发了那个能使得网络中断的缺陷。

- ◆ 错误的代码没有被恰当地审查。在 1990 年的中断中，它是来自于 AT&T 的新代码。在 1998 年的中断中，错误的代码来自于 Cisco 系统中的旧代码。

- ◆ 两个例子中的软件缺陷，都有一些界限情况比较模糊的问题，很难于测试，而且可能永远无法测试。

1998 年的 AT&T 网络中断，同时也暴露了该公司在网络操作过程与程序中的大量缺陷，证明了设计和维护一个有生命力的网络的困难性。显然，AT&T 从它的错误中已经学到了教训，并已经纠正了大量的过程缺陷，制定了大量的灾难恢复计划，以避免下一次全部网络失效的危机。

2.5.2 缓冲区溢出

2000 年 7 月 18 日，在“Bug Tray”邮件列表中公布了“Microsoft Outlook”和“Outlook Express”中的安全防范的详细描述。Bug Tray 是一个邮件列表组，主要是关于对计算机安全与防护的讨论。这个防护实际上包含一个软件工业界里人所共知的缓冲区溢出（buffer overflow）bug。在其最简单的形式中，当程序试图将数据放到一个大小不够的内存区域中时，就会发生缓冲区溢出的情况。下列的 C 语言代码就是一个例子。

```
1. char Array[10];
```

```
2. strcpy( Array, "This will cause a buffer overflow" );
```

数组的大小是 10，而消息 “This will cause a buffer flow” 有 33 个字符。strcpy() 试图将 33 个字符复制到一个只能容纳 10 个字符的空间中，所以一部分数据就写在了有效区域的外面。当发生这件事情时，程序可能会崩溃。而在其他时候，程序可能会没有任何影响地继续执行一段时间。但如果消息是按某种方式构造的，它可以使程序执行捆绑在消息中的病毒，从而使机器被攻击性的信息破坏。

到现在，多数对 PC 的安全考虑都包括病毒的变种，用户某些不得不做的工作帮助了入侵。例如，1999 年的 “ILOVEYOU” 病毒或 1998 年的 “Melissa” 病毒。在两个例子中，为了激活病毒，用户在受骗的情况下打开一个文件或执行一个程序。但是，Outlook 中的缓冲区溢出 bug 不需要用户的交互。你只需接收一个 E-mail，就可以受到影响，而且这种攻击非常难以防范。

不同于本章中其他的 bug 的情况，这个 bug 不局限于 Microsoft Outlook。作为 bug，缓冲区溢出已经存在了 10 年，它是一个著名的安全隐患。在网络安全问题之外，缓冲区溢出也是一个非常普通的 bug——好像每个人在其职业生涯的某些时候都发生过这样的错误。只要使用一个固定大小的存储变量，例如数组，就有面对缓冲区溢出错误的危险。因为这个问题是不可避免的，我们下面先不讨论 Microsoft Outlook 的 bug 细节，而是讨论缓冲区溢出 bug 的一般问题——错误如何造成的，它们带来什么安全隐患，以及我们将如何做来避免它。

故事

缓冲区溢出的原因是很容易理解的。它们在向一个固定大小的存储区域（缓冲区）中存放超过其大小的数据时发生。为了理解缓冲区溢出如何使程序崩溃，或者如何成为安全隐患，我们应钻研一下计算机的内部工作。

当你用诸如 C 语言这样的高级语言写程序时，编译器会把它翻译为汇编代码。汇编代码实际上是原始指令和数据的序列。例如，考虑如下的主程序：

```
1. main() {
2.     printf( "Hello World" );
3. }
```

这个著名的 “Hello World” 程序在 Windows NT 上用 Gnu ‘C’ 编译器 (GCC) 编译后成为 300294 个字节的二进制执行序列，看起来如下所示：

```
0x401000  5b 89 e5 83 ec 10 83 3d 00 20 40 00 00 74 01 cc
0x401010  d9 7d fe 66 8b 45 fe 25 c0 f0 ff ff 66 89 45 fe
...
0x401040  48 6b 6c 6c 6f 20 57 6f 72 6c 64 00 55 89 55 89
0x401050  eb e8 94 01 00 00 68 40 10 40 00 e8 92 01 00 00
0x401060  83 c4 ..
```

这些是相当难懂的，是不是？计算机使用这些数字来执行你所写的计算机程序。前面代码中的数字是用 16 进制表示的。第一列表示程序将要驻留的存储器的地址。后面的列表示那些地址的存储器内容。每一列将地址增加 1，所以在存储器地址 0x401000 中，存储的值

是 55。在 0x4001 中，值是 89，以此类推。其中的一些数字是告诉计算机要干什么；有些是计算机所要使用的数据。在这个例子中，计算机取出第一个值 55，把它解释为一个命令去完成一定的任务。然后到第二个值 89。计算机知道 89 这个命令需要一个参数，存储在邻近的存储器地址中有一个值是 e5。所以计算机执行的命令是 89 e5，然后取 e5 后面的命令。这样，计算机就同时执行了这些值，在“Hello World”程序中，字符串 Hello World 就是数据。如果你能够阅读 16 进制数并知道其 ASCII 表示，就可以发现“Hello World”这个串被存储在 0x401040 中。

0x401040 48(H) 65(e) 6c(l) 6c(l) 6f(o) 20() 57(W) 6f(o) 72(r) 6c(l) 64(d) 00

这组数字看起来与其他组的数字不可区分。是什么阻止计算机把这些值作为一个命令序列来执行呢？是程序中的控制流程，这正是缓冲区溢出的 bug 和隐患的关键之处。如果你犯了一个错误，计算机会误以为这些数据值是命令，并继续执行。没有办法知道它们是数据。在这个例子中，命令是非法的，程序很可能会崩溃，所以那些想成为入侵者的人的任务就是使用这个 bug，来诱使计算机执行他们提供的数据。所以，那些想侵犯你的计算机并使用你的软件的人需要做两件事：

- ◆ 将他的代码加入内存。
- ◆ 诱使你的计算机执行他的代码。

缓冲区溢出给黑客提供了做到这两件事的方法。因为程序在将数据写入缓冲区之前并不检查其大小，入侵者可以在与缓冲区邻接的地方放入自己的内容——这只是其工作的一半，另一半工作是计划出如何诱导计算机去执行他们已经在程序中加入的代码。

教训

在本章所研究的例子中，缓冲区溢出可能是最简单的容易理解和表示的 bug。它永远存在，仍然很流行。我们曾经听说过作为工业的东西吗？为什么我们老是犯同一个错误呢？

◆ 语言不好。缓冲区溢出在编程语言中是一个传染性的问题，没有内置的检查机制，例如 C 和 C++。这些程序设计语言为程序员提供随意操纵计算机的能力。程序员被赋予了这个能力，并被期望安全而正确地编制程序。他们也被期望进行清楚而必要的边界检验。比较而言，Java 这样的语言进行精确的边界检查，类似于缓冲区溢出这样的东西就很少了。但是 Java 限制了程序员的行为能力。它不允许程序员直接操作机器的存储器空间。但是这并不是说 Java 程序就不会出现缓冲区溢出的情况。这只是意味着程序员必须走一些弯路，才能在 Java 中造成缓冲区溢出。但当这件事确实发生了时，通常很难判断其原因。而且因为多数的 Java 虚拟机是用 C 或 C++写的，所以在 Java 虚拟机上就可能出现缓冲区溢出。

◆ 程序员不能被信任。程序员也是人，是人就会犯错。我已经在从前不经意地出现过无数次缓冲区溢出的情况。很多次，这些 bug 出现的原因是我先制作软件模块原型，而边界检查是我最后考虑的事情。但是原型可以流入成品中，其他情况是我对代码和使用做了假设，认为不可能出现缓冲区溢出，这时就产生了 bug。和通常情况一样，当我假定什么事情时，马上就会有证明我写的是错的。通常的情况下，发生 bug 是因为我懒惰，多数情况下编

写检查缓冲区溢出的代码是很容易的，但处理发生缓冲区溢出的错误时却非常困难和讨厌。当边界检查失败时，应该让程序中止吗？需要向调用函数报告这次失败吗？调用函数又如何处理这个失败呢？应该在调用函数中停止吗？如果不理它并认为它不会发生的话将会更轻松一些。当测试人员证明我的错误时，我会处理它的。

◆ 语言构造使得它很难。用一定的语言构造进行边界检验可能很难——一般会想到用 `sprintf()`。因为 `sprintf()` 接受可变数量的参数，要求程序员利用这些参数计算缓冲区的长度。这也很容易产生缓冲区溢出，因为不知道语言的细节。在 C 和 C++ 中，字符串“Hello World”要求 12 个字符的存储空间，因为这种语言把字符串定义为以空字符结尾。用其他的话来说，在字符“d”的后面还有一个空字符。如果只给它分配一个 11 字符的缓冲区，认为它可以存储整个字符串，并使用字符串函数，如 `strcpy()`，那么这里就会有一个缓冲区隐患，在你不注意的时候，就可能会咬你。

讨论如何避免缓冲区溢出 bug 的方法是很容易的。例如，停止使用比如 C 和 C++ 这样的“不安全”的语言并建立程序保护的规则。我们知道这个说起来比做更难。如果认为缓冲区溢出的 bug 是不可避免的，而且需要使用安全的语言，那么惟一的方法就是 bug 检测。越早检测和发现 bug，将来的灾难就越少。

从安全性角度看，我们不关心程序是否崩溃，我们关心的是程序的 bug 是否被发现。大量的研究计划和商业产品被设计用来处理这些问题。建议有兴趣的读者参考那些文章。

2.6 小结

本章所讨论的 bug，只是一小部分曾经存在或正在存在的 bug。现在成百上千的工程师正在加倍工作去发现 bug。一些 bug 很有意思，其他一些则很严肃或可悲。无论如何，在周末花力气修正 bug 是没什么意思的。但是在软件工程中修复 bug 所需要的代价，比起这些 bug 的影响来说，是微不足道的。本章中我们知道 bug 的损失可以是天文数字。价值上百万美元的设备 and 人力就突然消失了。人类的痛苦、灾难甚至死亡，都可能因为软件中一个细小的错误而起。

我们知道尽管有许多的测试和过程，bug 还是难以避免的。在产品环境中发作的 bug 可以是一系列错误的结果，一般没有单独的原因。实际上，在致命的结局之前，各有一系列的原因使得 bug 难以探测和消除。如，在 1999 年的中断中有一个简单的程序设计上的错误。但是如果运行了一个测试以检查代码的执行路径，并且经过了错误代码段，那么这个缺陷可能会更早地出现，并能够避免 AT&T 出现全国大范围内电话中断的尴尬。bug 的根本原因，是那些编写了错误代码的程序员呢？还是那些使得缺陷溜进产品环境中的测试部门呢？

我们知道，测试 bug 的能力与避免 bug 的能力一样重要。越早发现，修复的代价就越低。我们知道软件可以按照要求去运行，而且，当它的确按照要求去做时，我们还应该认为它是有缺陷的。我们知道必须意识到那些系统提供的线索，必须经常质疑、评价和理解线索背后

的理由。但最重要的是我们在 bug 面前必须虚心。如果没有发现可能的错误的能力，不对产品进行质疑，那么就会经常面临软件失效。因为自负是一个致命的过失。

我们已经确定了本章中的 bug 的许多原因，这些原因被总结如下：

- ◆ 缺少对工具和语言构造的理解：

例子——AT&T1990 年的中断，缓冲区溢出

描述——当程序员不能正确地理解语言构造时，就会有缺陷。在 AT&T 的 1990 年的中断中，其错误是对 break 语句的错误放置。一个典型的缓冲区溢出的 bug，可以因为不理解标准 C 和 C++ 函数中的操作包而引起。

- ◆ 在不理解的情况下重用：

例子——Ariane 5

描述——软件模块被重用，但缺少对所重用软件模块的要求的理解。没有去评价重用的恰当性。

- ◆ 软件变化和升级：

例子——AT&T1990 年的中断和 1998 年的中断

描述——升级的目的是使系统更加高效，但升级过程中及之后出现了 bug。

- ◆ 忽视不可预见的结果：

例子——火星气象卫星，Therac-25

描述——有许多不同的警告及线索，预示着在两个例子中会有差错。这些警告作为未预见的结果和行为出现。但线索照例被忽视，没有进行正确的调查。在火星气象卫星中，问题是飞行器与地面站之间的计算模型不一致。在 Therac-25 中，问题是 Therac-20 与 Therac-25 之间连续的误操作。

- ◆ 不理解线索：

例子——火星气象卫星，Therac-25，X 公司，Y 公司

描述——那些指出了潜在的 bug 的线索一直没有被恰当地解释，导致所有的最终错误。在 X 公司和 Y 公司中，线索正确地描述了症状的原因，但一开始就没被注意。

- ◆ 缺少相关的管理工具：

例子——火星气象卫星，Therac-25，X 公司，Y 公司

描述——好的管理工具可以帮助操作人员，在 bug 成为严重问题之前将其发现和解决。将火星气象卫星的例子归入这一类可能是不恰当的，因为科学家要局限于技术和他们的观测能力的限制。但是他们不能观察到飞行器的实际轨迹，这就限制了他们判断其模型正确性的能力。在 Therac-25 中，错误操作的描述不够详细，不足以警告操作人员机器上的潜在问题，而且很明显地缺少一个反馈，以使得操作人员知道机器到底在做什么。X 公司和 Y 公司的问题中，如果操作人员在失效之前能够发现数据库性能的降低，其问题就会大大减轻。

- ◆ 不正确的或糟糕的描述：

例子——Ariane 5

描述——Ariane 5 的错误处理软件说明错误的值被置于由导航计算机使用的存储器内容中。软件实际上也符合这个描述。导航计算机使用错误的值进行了火箭的机动，导致灾难。在这个例子中，错误的描述不正确从而导致失事。

◆ 不遵循规范：

例子——火星气象卫星

描述——火星气象卫星的规范很清楚，换算单位应该是公制的而不是英制的。

◆ 重用 bug：

例子——Therac-25，AT&T 1998 年的中断

描述——重用的软件模块中包含一些在其早期运用中没有被检测出来的 bug。

◆ 没有按照正确的程序：

例子——AT&T 1998 年的中断，X 公司

描述——设计出了阻止我们犯错的程序，当程序没有被遵守，而采取了捷径时，就会出现未料及的结果。

◆ 不安全的工具：

例子——缓冲区溢出

描述——缓冲区溢出的一个主要原因是 C 和 C++ 语言的“不安全性”，使得软件依赖于程序员精确地进行边界检查，以避免缓冲区溢出。

◆ 人为因素：

例子——缓冲区溢出

描述——我们都知道如何避免缓冲区溢出，而且我们一直在犯这个错误。

◆ 骄傲和自负：

例子——Therac-25，火星气象卫星

描述——自负可以妨碍对系统的评价，也可以使我们不去怀疑我们的假定。这些缺点的结果是我们可能不去留意被提供的线索以及潜在的软件缺陷。

◆ 不完整的测试：

例子——Ariane 5，火星气象卫星，Pentium，AT&T 1990 年的中断

描述——事后聪明是没用的，多数的这些 bug 本可以通过“正确”的测试来发现。在 Ariane 5 的例子中，需求的不完全和模拟操作环境能力的限制，意味着工程人员不得不预先进行测试，以发现 Ariane 5 中的 bug。还不能确定在火星气象卫星上是否做了对 bug 的测试，因为工程师当时仍然在修复软件。对于 Pentium 的 bug 和 AT&T 1990 年的中断所进行的测试，没有考虑到算法的本质，所以一些代码没有被测试。在 Pentium 的 bug 例子中，随机测试对于测试参照表中所有条目的有效性来说是不充分的。在 AT&T 1990 年的中断中，测试的例子没有全面地考虑交换机恢复的情况。

2.7 bug 问题

你正在测试公司网站上的-组网页。你点击了对网页 A 的链接，每件事看起来都不错。你继续测试网站的其他网页，很偶然地你又回到了网页 A。但是这一次，上面的内容算过期的吗？你有点怀疑，然后重新加载了它，它现在看起来正常了。你现在还是不太相信，又加载了一次网页，网页看起来正确。经过了三次，你自己觉得如果网页已经正确了，那么刚才所见到的只是一个例外而已。你重新加载了网页，网页的内容再次过期。你把头埋向显示器，在一次又一次地加载这个网页。10 分钟后，你发现这个网页正确加载了 34 次，错误 12 次，这个 bug 的原因可能是什么呢？

这里有些线索来帮你推理这个 bug 的原因。系统地讲，有 5 个网络服务器处理浏览器请求。每个网络服务器可与 4 个应用服务器之一通信。应用服务器从数据库中取得数据并应用业务规则。为了使速度更快，每个应用服务器在一个本地的高速缓存器中存放数据。

第3章 什么是 bug

1964 年在 Ohio 州的 Jacobellis，后来的高级法官 Potter Stewart 写下了他的观点：

“今天我不想再深入地把我所理解原材料的种类定义在一个简单的描述里，而且我可能永远也无法深入理解它。但我知道当我看见它的时候，我就明白了，而且例子中的动作图并非是这样的。”

Stewart 法官是针对赤裸裸的色情文学而写下上述文章的，但是他同样善于写关于 bug 的东西。bug 也有相似的虚假性。我们看到一个东西很容易，但很明白地定义它却很难。在一些例子中，我们都怀疑某些 bug 是不是 bug。

3.1 什么是 bug

当我们阅读这本书的时候，可以使用迅速的信息传送功能，与别人或本书的思想进行交流。（或者，我们主要用它来打发时间而不必去干什么活）。一天，我们正在进行一次关于生命意义的在线会议……

Matt：这样你就有了<a>,<c>。

[Yuan 非常迷惑，但只是认为 Matt 不会输入这些内容。]

Matt：注意不要用 HTML 标记。

[Yuan 笑了。]

Yuan：你输入了<a><c>，但系统漏掉了！

(在'<','b','>'之间有空格)

[5 分钟之后，在 Yuan 的脑袋中突然灵光一闪。]

Yuan：这是不是 bug？

Matt：我不这样认为。我认为系统是想解析 HTML 标记。

Yuan：但它不再让我写了。

Matt：<pre></pre>

Matt：嗯，我认为<pre>被系统认为是封闭的文本。我写了<pre></pre>。也许我的标记有问题。

[Yuan 处理一个在线的 HTML 文档站点。]

Yuan：<pre>是正确的标记，我认为这个 IM 没有解析所有的 HTML 标记，只是一个子集。我想这是一个 bug。

Matt：我不确定，我要看看说明。

这是一个 bug 吗？当一个程序崩溃时，很容易说它是一个 bug。但是当软件没有按照期望去做时，它就有 bug 吗？什么是 bug？

作为一个词，bug 被软件开发组织用来描述一些需要修复的问题。但是，在多数学术研究和工业文献中，好像一般都避免使用 bug 这个词。理由可能是这个词过于通俗，而且对于不同的人有不同的含义。但这本书是关于 bug 和调试的，不是关于软件缺陷（或错误，或失败等）。所以我们将要仔细地研究软件工作人员日常生活中所说的 bug 的定义。

Bug 的一个简单定义：bug 是未预料到的系统行为。

这个简单的定义的一个问题是，一个人眼里的 bug，可能是其他人眼里的特征。像一说法说的那样，bug 存在于旁观人的眼里，在快速信息传递的例子中，对于用户不能传递未解析的 HTML 标记，我们认为是始料未及的事（虽然我们内部还不能确定它是否是一个 bug）。开发人员工作时所使用的说明可能只定义了解析 HTML 标记的需求，而不是传递未解析的 HTML 标记的方式。所以，工具开发人员可能认为这是个小问题。毕竟，程序是按照说明要求运行的，而且它的运行也是按照期望的。在项目的管理部门得到通知以前，这个未预期行为还不能被认为是系统的 bug。即使项目管理人员发现了这个特征，也不会很重视，他们更可能同意开发人员的观点，即这只是一个小程序问题。

现在，快速消息传递的用户已经学会适应这些小问题，而且学会了处理它们。（我们在潜在的 HTML 标记字符之间加入了空格，这样，就能够传送了。）我们先前考虑的未预期行为的问题已经快消失了。但是当用户向管理人员抱怨时，相应的改变就和重要的客户同样重要了。这时，这个特征就会被汇入 bug 的数据库并被重视，这就给我们带来了 bug 的第二个定义。

Bug 的一个严格定义：bug 是系统中的这样一个现象，即本来应该有的功能却没有。

这是对 bug 和软件工业的一个沮丧悲观的看法，因为它暗示 bug 是主观主义的，并没有标准，像第 2 章中的 Pentium 的 bug 的例子那样。Intel 作为一个大公司，会希望我们把芯片中的浮点缺陷看作一个小小的不便。在所有的可能性中，Intel 的芯片设计者可能会认为这个缺陷是个 bug，因为它没有按照期望去做。用户组织则明确认为这是个 bug，并进行了大量的讨论迫使 Intel 承认浮点缺陷是一个 bug。在这个例子中，Intel 的工程师们已经知道了这个缺陷，并正确地修复了。调试对于 Intel 来说只不过是为用户组织更换芯片。

的确，很多例子中我们没有能将 bug 说出来，或者一个 bug 被一些奇怪意义的操作定义为不存在。例如，你在一个公司中为另一个公司编写软件，让我们称其为公司 A。在你递交初始版本时，公司会因为不喜欢其中的一些特征而称其为 bug。你还有选择吗？如果在这件事中，技术要求很明确或在初始的要求中特定的特征不明确，那就没什么问题。你最终可能要修复这个“bug”。至少在商业软件、工业中这是现实。在多数情况下，一个 bug 如果确实是 bug，那么它对于相关的公司来说，应该是很清楚的。作为软件产品和服务的用户，我们都明确知道什么是 bug。系统中有些行为，对于我们用户来说是不可接受的。我们也作过开发人员和测试人员。在这些角色中，我们也直觉地在发现 bug 时，承认它是一个 bug，因为

它使得系统没有按我们开发和测试人员的期望去做。所以，当双方对没有实现的行为达成一致时，这个没有实现的问题就基本上是个 bug 了，我们对 bug 的定义作出新的改进如下。

Bug 的一个以人为本的定义：bug 是系统没有实现软件开发人员（开发者、测试人员、项目管理者）和用户之间达成一致的行为。

当相关的组织无法对某个特征是否能实现达成一致时，会怎么样呢？这时，完全取决于哪个组织最有影响力，这意味着我们回到了中心定义。幸运的是，在多数情况下，相关的组织对特征和 bug 的定义会一致，不一致通常围绕于不直觉的界面，想要的特征和可用性，以及不匹配的期望。

这个对 bug 的定义对我们有什么意义吗？当然，即使它只说明了 bug 是人们达成一致叫做 bug 的东西，这个定义也是有用的。它表明了有一组良好设计的需求说明的重要性，因为一个需求说明应该列出在达成一致的系统中所期望和不期望的特征。它强调了软件开发人员理解业务需求和其客户过程的重要性。它说明软件开发人员必须把他们对系统的看法与客户交流。它也强调我们相互之间应该共享各自的概念和想法。

3.1.1 什么是软件缺陷

传统情况下，在软件工业中，bug 这个词可以与“软件缺陷 (defect)”相互使用。但是，我们必须承认在软件缺陷和 bug 之间的确有一些的差别，bug 是软件缺陷的实际证明。

Intel Pentium 的 bug 清楚地表明了这一点。Intel 公司同意用户组织的关于芯片有缺陷的看法。区别是 Intel 不认为这个缺陷是 bug。公司感觉普通的用户只会在 27,000 年里遇到一次；这样这个缺陷不会造成什么后果，而且不必为普通用户修复它。但是用户组织也不同意。最后 Intel 让步了，并改变了对这个缺陷的看法。缺陷成为 bug，而且 Intel 建立了处理这个 bug 和用户组织的相应模型。我们把软件缺陷定义如下：

软件缺陷：软件缺陷是一个软件系统中的需求、体系结构、设计和应用上的错误。

我们对软件缺陷的定义比工业标准要松得多。我们认为软件缺陷可以出现在软件开发的全过程中，而不只是与应用过程中出现的错误相关联。如果一个系统是按照一组相互冲突的需求而设计应用的，最后也就会出现一些未预料的问题从而需要更正。在这个例子中，软件缺陷存在于需求说明中，因为如果要修正错误的行为，必须先解决出现的需求冲突。如果一个系统要同时满足 1000 个需求，但设计时只满足 500 个需求，而且不是可扩展的，我们就说系统的体系结构有缺陷。如果在系统组件的应用中并不检查存储器分配语句的正确性，而是认为分配都是成功的，那么这个系统就有一个缺陷最终会导致系统崩溃或反常。不管缺陷在什么地方，它都可能给开发人员和客户带来不应有的困难。

软件缺陷有成为 bug 的可能性，但并非所有的软件缺陷都产生 bug。平均而言，每 1000 行代码内有 25 个缺陷，一个 500,000 行的程序，就意味着可能有 12,500 个缺陷，我们到现在仍然在依赖于计算机，简直是个奇迹。实际上，软件缺陷只不过是那些存在于软件中的缺陷。多数缺陷从不现身。为什么？因为它们相当一部分只是那些很少运行的逻辑，因为程序

的运行总没有编写得那么长，因为程序的使用方式，或是因为我们走运。不幸的是我们无法发现关于判断缺陷的数量与 bug 的数量之间关系的研究，所以我们不能知道缺陷水平与 bug 水平之间的确切关系。但是，假设缺陷水平和 bug 数量成正比应该不会有错。（就是说，缺陷的数量与 bug 的数量一起增加。）

考虑如下例子：

```
int main( int argc, char **argv ) {  
}
```

这个小的什么也不做的主程序有一个应用缺陷，它在离开主函数时并不返回整数值。在多数情况下，这个缺陷不能算作一个 bug。但是，在一个系统中，如果该程序是一个组件，而且程序的退出情况被捕捉和使用（如在一个 Unix 的命令解释脚本中），那么这个过失最终可能会使其成为一个 bug，一个可能会使得可怜的调试员努力寻找和修复的 bug。

现在我们已经知道什么是 bug，那么应该怎样对待它呢？调试它！这是当然的。

3.1.2 调试不是什么

我有一个朋友，工作于一个大型远程通信公司，负责处在一个标准的三级的客户机/服务器系统的一台客户机。他的组件使用于由服务器开发组开发的应用程序编程接口（API）中。一天，他的组件开始崩溃，虽然他没做什么变化，只是从服务器组中链接到了新的 API 分支。在检验了对客户机系统的输入是可靠的之后，他开始钻研代码。他很快地发现了崩溃的地方，令他吃惊的是，那是由服务器开发人员添加的一句很明白的退出语句（一个 `assert(false)` 调用），而它在以前的版本中根本没有。当然，代码附近没有任何注释告诉他理由。几分钟的电话就迅速解开了这个谜。虽然，服务器开发人员有一份关于那段代码的 bug 报告。由于不能发现问题的真正原因，服务器开发人员决定在代码运行到不正常状态时就停止程序，因为服务器开发人员认为任何“正确的”对 API 的使用都不会导致这个问题。

这个例子并非是服务器人员有意放置了一个 `assert()` 来在调试模式下捕捉这个错误，然后偶然发布了调试版。这是一个故意的决定。开发人员的理由是如果这个 `assert` 被触发，这就是程序员出了错误，而且程序应该结束。希望你也认为服务器工作人员没有调试，它只是屏蔽了这个来自于另一个过程的 bug 而已。

实际上，服务器工作人员修复了初始 bug 的特征，满足了另一个用户的要求。但是，工作人员并没有查出 bug 背后的原因，没有理会所做的结果。经过“修复”之后，初始的 bug 仍然存在，而且可能在其他某次机会发作。“修复后”的系统中，仍然有一个服务器工作人员造成的附加 bug。不管你相信与否，这是对特征所做修改的一个非常好的结果，系统自动退出并通知其他的用户和开发人员发生了解蔽。在没有人能够“修复”时，对特征的修复就派上用场了。

考虑一个存储器错误的 bug 的例子。你认为它是个存储器错误 bug，因为当你在正确的位置添加一个赋值语句时（如 `X=20`），程序就不再发生崩溃。很轻松地，你提交了这个变

化。发生了什么呢？在所有的可能性中，你恰恰使这个 bug 更加难于发现了。你添加的赋值语句并不能改变导致存储器错误原因的存在。你只是延迟了它或除掉了它的替身，几个月后，程序可能在另外的地方又崩溃而那时你可能已经忘记了所作的变化。到那时，谁知道在应用程序上又加进什么复杂的东西？bug 随着代码的逐渐改动从而越埋越深。

但是，在一个周转过程中处理特征性的东西仍然有些价值。在第 9 章，我们将介绍一种能够描述出代码的技术，以帮助观察和理解代码的行为。这个技术在一个控制环境中完成，而且变化对于回溯检查错误的原因没有影响，也不是一个伪装的修复。

在体系结构或设计层次修复特征是一个很大的错误。在这个层次修复特征的主要后果是：它导致不稳定的不可扩展的、不可重用的系统和组件。Blob 和瑞士军刀的反仿制，就是在体系结构或设计水平修复特征，而不修复原因所导致问题的例子。

Blob 反仿制是在一个单独的类垄断了模块中所有过程时出现的。瑞士军刀反仿制可以描述为一个单独的类中多余的复杂的接口。当在初始的设计中要求了某种功能却没有说明时，开发者会想办法在新的功能中最容易分解的地方，而不是逐步回溯而且重新考虑设计。通常使用一个方便的类来提供新的特性或模式。几次反复后，这个类就与一个 Blob 或瑞士军刀相似了；它能够做所有事情。

一个流行的误解是：在应用中缺少某些特征就是一个 bug。有些人认为调试就是加入新的特征。缺少的特征一般是由于对需求说明的忽视，而且应在设计层次解决。但是，如果接受我们对 bug 的定义，缺少特征实际上并不是“系统不合适行为”。它们是系统的“合适的但并不存在的行为”。而且添加特征并不修改什么，我们并非在消除 bug 和缺陷。

可以肯定地说，对已经存在的产品质量的软件添加新的特征并不是一个小任务，尤其是在体系结构或设计层次做改动时。我们对软件保护保留了一章，而且单独处理这个主题。这个误解可能已经出现了，因为我们喜欢将添加新特征的任务作为对体系结构和设计进行大检查的理由。这是一个修复体系结构和设计 bug 的机会。这个误解的另一个原因是在一些情况下，在 bug 查询系统中跳过一个特征需求而不是通过从上到下的正常渠道的话，要更快一些。开发人员观察那个漏掉了 XYZ 特征的 bug 并应用了 XYZ。每个人都很高兴，只是这个新特性从未在需求文档或设计文档中出现过。很快，我们手头上就有了一个难于处理的麻烦。

3.1.3 什么是调试

我们已经讨论了修正特征和添加新的特征不是调试。显然，添加新的特征就对基本的代码添加了代码，并不是删掉了系统不应有的行为。修复特征主要是因为缺少对系统的理解，并且不做什么去除潜在缺陷的工作。知道了这些，我们就可以定义调试过程了。

调试 (debugging)：调试就是理解系统的行为以利于去除 bug 的过程。

在调试时，我们试图根除那些导致系统出现未预期行为的根本原因。为了理解其根本原因，我们需要理解系统的运行情况。这里的关键字是“运行”和“理解”。我们需要理解系统做什么——不是我们认为系统应该做什么。也不是另外什么人说系统在做什么——系统实

际上在做什么。而且，如果不理解系统，就不能指望改动系统让它完成我们想做的事情。我们所做的任何改变系统行为的事情，都有破坏另外一些事情的危险，也有只改动了特征而带来了更多的 bug 的危险。

调试并不只是一个发现和纠正系统中软件错误的过程。一些 bug 不是由于软件缺陷造成的。软件开发人员倾向于认为每个 bug 都可以被跟踪到代码一级。在多数情况下是这样。但是，我们所构建的系统和应用必须运行在一个有其他系统和应用与之并存的环境中。当出错时，我们必须着手解决问题，并考虑问题的所有因素。一个电缆可以被拔下来，从而导致系统没有反应。有些人可能安装了新的不兼容的操作系统包，使得一个稳定的应用在没有明显原因的情况下发生混乱。一个标准的过程或程序可能被忽略了，导致服务中戏剧性地停机。在分布式计算环境中，组件中一个微小的故障可以导致在另一个看起来无关的组件中出现 bug。在第 2 章中的分布式计算环境的例子中，读者可以发现数据库所发生的意外的响应缓慢，可以导致整个系统出现问题。在那个系统中有许多隐藏的缺陷，而仅通过关注一个组件而想去修复 bug 几乎是不可能的。

调试的科学和艺术，是我们在本书中的主要内容，包括所有这些方面。我们涵盖了与软件和应用相关的 bug，而且也触及那些并非由软件缺陷而导致的 bug。我们的目标是帮助读者理解 bug 产生的原因从而可以阻止它们。并且提供一些有效的诀窍、技术和过程等武器，以帮助使得调试过程更加有效而不那么痛苦。

3.2 为什么要考虑 bug

我们担心 bug，因为它们的代价很高。bug 可以耗掉我们的金钱，打击相关的人，并破坏我们的形象和名誉。当然，我们注意 bug 也因为我们骄傲，希望尽可能做出最好的软件。

有一天我正在往邮局走，想去买一张汇票。当我到达那里时，发现灯关了。门上贴着一张启事：“因为电力中断，我们今天不接收现金和信用卡业务。”现金？信用卡我可以理解，但电力中断会影响现金这种事我还是不相信，我给他们钞票，他们给我票据，还有比这简单的吗？我们的社会已经进化得过于依赖技术，特别是计算系统。通信、数字投影和录像、金融交易、计算机为基础的飞行控制、交通指示灯、内置的诊疗自动机、微波、苏打机——无论往哪里看，都有一个计算机（无论多小）为你而工作。一个最不合适的 bug，发生在错误的时间和错误的地点，那么其后果可能是很可怕的。千年虫问题就是一个例子。

在软件短短的历史中，许多的 bug 已经直接或间接地影响了我们的生活。为什么要考虑 bug？在极端情况下，你的生活或你的生活方式可能会面临危险。但是这些惊险的战略谈得已经够多了，让我们看一看为什么一个公司要关心 bug，看其三个主要原因。

3.2.1 bug 的信心代价

bug 的最容易被忽略的一个代价，是与项目中涉及到的人的信心相关的代价，没有人想

和比较劣质的软件有什么牵扯。我们都希望能为自己的产品骄傲。软件中持久的 bug 对我们的自身价值来说，是公开的污辱。在未检查的情况下，有 bug 的软件可以而且必将降低软件开发人员的积极性，从管理者到开发人员，从测试人员到客户服务人员。

软件工业中的每个人在其软件产品中都会有一些 bug。在当前的环境中，一个软件开发组都会而且确实在花主要的时间去处理 bug，直到其软件的第二个版本推出。但是，bug 在其暴露的时候已经开始成为一股令人丧气的力量，尤其是当你以为已经修复了 bug，却眼睁睁地看它自己再次得逞时。它可以到达这样一个程度：一个微小的变化，甚至是对 bug 的修复，都可以导致另一个 bug 的产生。当发生这样的事情时，你常常会感到悲观，认为没什么办法了。

人们已经厌倦了重复处理相同的问题。开发人员想搞新的东西，一些有趣的东西。他们想使用最新的技术和工具设计最新的流行产品。客户服务人员已经厌倦了因为同样的问题而一次又一次地被叫走。项目管理者厌倦了让开发人员去修复 bug，而且面临最后期限。像开发人员一样，他们也想搞新鲜一些的项目。测试人员的角色比较可怜，他们眼看着 bug 列表随着时间增加。沮丧来了，然后又去了另一家公司或去了另一个项目里，成为很令人注意的东西。开始有人指指点点，会议越来越长。这些都无助于一个良好的工作环境。

令人失望的软件开发组的结局，是低的生产能力和高的周转率。像许多管理书籍所说的那样，一个令人失望的团队，最后可能会耗掉组织的一大笔资金。这个团队中的成员可能会找捷径，以使得他们能避免碰到什么问题，而不是用自己最大的能力来做工作。虽然捷径在短期内可能有效，它们在长期的耗费可能是很大的，会出现新的问题，而现已有的问题可能就被掩盖了。另外，一个简单的问题可能会待在那里好几天，而没有人有兴趣解决它。周转意味将有新的成员加入，而且培训耗费是比较大的。新的员工需要培训，而且刚开始他们的生产能力很低，从而导致更多的不必要的耗费。

3.2.2 名声和形象的代价

软件工业在消费品工业中是独一无二的——我们训练客户像正常操作过程一样接受我们的产品中的问题。而且，当我们产生有缺陷的产品时，我们会给用户补丁和修理工具。但是最好的是，我们迫使用户与我们签订协议，声明我们不能保证所提供的都是可用的，他们也不能因为损失和可靠性而起诉我们。这样我们还需要担心 bug 会如何破坏我们的名声和形象吗？当然要谢谢上帝。

虽然用户已经被说明接受有 bug 的软件，但并不表示他们喜欢它。在 1999 年 1 月，《PC World》对用户进行了一次关于有 bug 软件的调查。结果表明：“64% 的被调查的用户认为如果允许软件有 bug，那么他们将购买那些特征比较少的软件程序”。这说明除非用户必须买某一公司生产的软件，否则他们将会寻找那些有更少特征的软件。

与其他的组织相同，软件组织的成功依赖于许多因素，市场运营中的一个关键的动力是品牌的声誉。当提起处理数字图像的软件包时，我们会想起 Adobe Photoshop。办公产品就

当属 Microsoft Office 系列了。Quicken 是重要的个人财务软件包。这些关联都是来之不易的。在品牌创建的早期中一个重要过失，会带来不可挽回的损失。软件市场中的另一个力量是占据锁定 (lock-in)，这是指用户主要是由于对数据格式的依赖性和熟悉的界面，不得不局限于从一个固定的公司购买产品。锁定之后，那个软件公司就有了损害形象的一个回旋的余地。

但是，互联网可能是改变用户愿望的一个主要力量，而让他们不再接受差的软件产品和服务。如果有一个电子商务网站，人们无法再忍受长的等待时间、差的性能和丢失的订单；他们就会尝试另一家卖相同货物的网站，人家可能有更好的服务。进入互联网/电子商务的障碍是很低的，一个品牌就可以帮你搞定。这里并没有锁定的概念，用户也没有理由再依赖那些质量差的产品和服务。一旦客户离开，让他们再回来的可能性很小，因为他们认为你的网站不可信。得到一个客户的耗费，通常比留住一个客户的耗费要高。如果你的业务是向网络公司卖电子商务软件，这些公司将不能忍受低质量，因为这影响到他们的底线。

在考虑互联网之前，网络安全对于大的公司来说是个主要的顾虑，包括军队和工业系统。现在谁都有一个电子商务网站，而且网络安全已经成为一个关键的问题。潜在入侵者一般在系统中或代码中开发一些 bug，来获得对特殊区域的访问权。像第 2 章中描述的那样，网络安全的最大威胁来自缓冲区溢出 bug。但是，黑客并不是顾虑的惟一原因。大家也十分关心一些内部的 bug，它们可能会错误地将私人的财务信息公布给错误的人。而且如果客户的需求过量或他们的订单不明确，那么业务就会丢失。例如，如果顾客想买 ABC 产品，而你运给他们 XYZ 产品并收取了他们 10 倍的钱，他们就再也不会和你合作了。

3.2.3 bug 的财产损失

最终，任何事情都归结到了钱。高级员工的周转要钱。坏的形象意味着一个公司必须花费更多的钱去取得客户。但即使没有直接损失，一个有 bug 的软件的项目对组织的底线也有直接的影响。运行一个软件项目并产生一个工作系统也是很昂贵的。当系统有 bug 时，就面临一个需要更长时间的风险；需要更多的时间，就意味着产品就要更晚一些推出，而且需要更多的资源来开发它。每天都在延误软件的投资。最后，软件投资的收益是 0，项目被取消。但是一个从开发过程中溜出来的进入现实世界的 bug，可以使得这个被取消的项目比较起来像一个廉价的商品。

1994 年，Capers Jones 对大约 4000 个软件项目进行了调查。该调查向被调查者询问了超出日程表的原因。最常见的一种原因就是质量差。而且他指出几乎一半被取消的项目的质量都比较差。

有 bug 的软件维护起来也比较昂贵。一个对 63 个开发项目和 25 个维护项目的分析显示，开发可靠的软件的费用是不太可靠的软件的两倍。但是低可靠性的软件维护起来比高可靠性的要昂贵得多。

现实中 bug 的代价可能是天文数字。为了平息客户对于 Pentium bug 的抱怨，Intel 同意更换所有有缺陷的芯片，耗资 400 多万美元。Denver 国际机场的旅行包自动处理系统的

开发就有一些 bug 问题。原计划 1993 年投入使用，但机场因为旅行包处理系统中的 bug 而直到 1995 年才开放。每天都在延误开放，它耗掉了机场和城市大约 1,100,000 美元。因为有人忘记了模块中的单位转换，NASA 的火星气象卫星在火星的表面撞出了一个坑。还有“火星极地登陆者”，这个项目的投资的回报几乎是 0。投资是多少？只是 327,600,000 美元而已。软件 bug 最大的损失莫过于生命、人类和其他东西。Therac-25 是最常举的例子——那些被设计用来治疗的软件导致过剂量的辐射。这个 bug 导致了 3 个人的死亡。

3.3 bug 的本质

每个 bug 都是独特的，但是 bug 又都是相同的。古语说：“要洞悉你的敌人”。在我们力争程序的功能性、可操作性和有效性的长期斗争中，我们的敌人是那些迫使我们一直坐着的 bug。敌人披着不同的外衣，有些 bug 就像核武器——一次爆炸就让计算机不能用了。其他的 bug 则比较轻，它们慢慢地破坏薄弱环节，一次破坏一点，直到使系统失效。有些 bug 则类似于双重间谍。它们作为一个意想不到的有用特征出现，等待最好的时机发作。有些 bug 有着共同的性质，供我们考虑。当你能够用杀虫剂突然杀掉 99% 的臭虫时，为什么还要去一个一个地捏死呢？这对杀虫的人有效，对我们也肯定有效。我们怎样产生出这些类似于化学药品的东西呢？我们先来研究和分析 bug 的一般特征。

3.3.1 bug 的发生都有原因

bug 并非像人们所想的那样，抬起头来向人们发难。bug 不是硅中的小鬼产生的。除非在空间飞行器上或地面站上处理卫星发射，bug 一般也不是随机的太阳耀斑或宇宙射线造成的；相反，bug 只是系统逻辑的奴隶。

软件 bug 在软件中产生。软件在计算机中每次运行一个指令（不包括多处理器）。计算是不做任何空想的。它不要概括，也不假设。它不作任何观察。这当然离不开你所写的代码。它是你的产品，是你的奴隶。它忠诚地按照代码执行到最后。如果你告诉它跳过一座桥，它就跳（假设你不是在一个有自卫本能的模块中编程，而且代码中没有 bug）。我们的问题在于我们不像计算机那样想和做。所以 bug 产生的原因常常是神秘的。为了成为一个高效的调试员，必须有一个清醒的头脑。我常想建立一个 ESP 模块，让它做我让它做的事（对于 40 多岁的人来说，这就是所谓[DWVN]命令）。但即使到现在我已经有了这么多的成就，也只能去习惯生存于这个不完美的世界。所以就不要往这方面想了。

3.3.2 bug 是可以重现的

如果 bug 因为某种原因而发生，就意味着必然有一些原因使它发生。如果能重现出这些原因，必然就能够再现出这些 bug。听起来很简单，是不是？如果你已经在软件工业中干上了一段时间，你会知道这并非那么简单，因为让 bug 重现是很困难的。

有些 bug 比其他 bug 的重现要难得多。这些 bug 看起来无法重现，只是因为我们缺少工具来模拟操作环境并捕捉应用的输入历史。一些不能重现 bug 的理由包括：

- ◆ 客户可能正在以一种未预期的方式使用产品。当他们报告一个 bug 时，他们会忘记说一些你以为不用问他们就会说的东西。所以你按照他们的描述试着重现 bug，但却始终无法重现出这个问题。

- ◆ 软件可能工作于实时性的大规模环境中，而且使用对时间有依赖性的数据。你只是无法捕捉所有的数据以保证没有 bug。如果一个 bug 使得软件崩溃，而且没有比较好的诊断信息，将很难用任何有说服力的方式重现这个 bug。

- ◆ 你可能有一个多线程的系统，有很多计时器。如果没有得到与时钟相一致的输入事件序列，就不能重现这个 bug。但这并不意味着这个 bug 是不可重现的。如果想重现这个 bug，最好在系统中建立一些支撑环境。

重现 bug 所遇到的困难并不表示 bug 是不可重现的。但有时重现的代价太大。其他情况下，我们不理解系统中相互关联的所有前因后果，从而难以顺利地重现它。通常只要你发现并理解了它，就很容易再次观察这个 bug。但是，你也没有义务去说明你知道它的原因，而且你已经解决了它。如果不考虑义务，本书就会很薄。

3.3.3 bug 通常在有变化时出现

在对模块做的改变和改进的数量，与模块中缺陷的数量之间有很强的联系。为判断几个设计规格和软件的缺陷水平之间的关系，有人对一个包含 70000 行代码的软件组件进行了分析，这个研究能够估算出每个对软件组件的改变和改进导致大约 0.628 个缺陷。

另一项对于来自一个大的远程通信系统的大约 100 万行的可执行代码的研究，再次表明代码变化会使效果更加不稳定。在这项研究中，作者试图判断重用错误的软件模块所带来的影响。他们的系统中大约有 1980 个功能模块。他们发现如果没有改变，他们的每个模块中大约会有 0.52 个错误。但是，如果重用的模块被改动了，错误率可能会上升到 1.753。最后，如果又产生了一个新的模块，大约平均会有 2.294 个错误。

这个研究的基本意义很容易理解。改变代码会增加出现错误的机会。理想情况下，当需求发生改变时，设计者和开发者会坐下来重新考虑系统。但是，我们往往只是考虑那些会被新特征所影响的模块，只在局部进行变化，而没有全局观点。这可以破坏现有的界面。需求的变化常常是在软件开发日程表之内发生的。这意味着设计人员和开发人员必须在不现实的时间限制下工作，使他们不得不走捷径。而且，软件不是静态的，也不是单独运行的：磁盘空间可以用尽，网络可以瘫痪，另一个应用程序可能争夺同样的资源。如果不对这些环境变化进行编程，可能就会有一个将来导致 bug 的缺陷。

软件也与人交互。人们会犯错，那些不预先考虑和阻止人为错误的软件系统肯定会崩溃。例如，人们可能在加进新的数据之后，忘记了重新索引数据库，这样数据库的查询时间将增大 10 倍。一个体系结构依赖于快速数据库的系统会突然发生问题。根本原因可能就是这个

未索引的数据库，而且无论怎样看，这实际上都不是软件的错误（不同于体系结构的容错能力很强的情况）。但这样会使作为系统的软件开发者的你，在半夜五点被叫醒去修复“bug”。

3.3.4 bug 也会产生 bug

我们取笑这样一个情况，当我们升级软件或下载补丁来修复一个已有的 bug 时，我们实际上就是在将那些已知的 bug 升级成未知的 bug。撇开笑话的成分，它的确有一些有说服力。高质量的实验和个人的实际经验表明，当修改代码时，就有使软件出现更多 bug 的危险。当调试系统时，就是在改变系统，因为 bug 可以在一个多变的系统中存活：那么在试图修复系统时，弄出或发现新 bug 的机会是很多的。现实的经历表明，在修复一个“错误”时，弄出一个或更多额外“错误”的可能性，依情况不同而估计有 15% 到 50%。

这更说明在大的程序中，修正原始“错误”期间增添出严重的“错误”的机会非常大，以至于只有一小部分的原始“错误”需要修正。

另一种 bug 产生 bug 的方式是：软件中有问题部分仍然有问题。如果你曾工作于一块代码或一个模块，而它由于倾注了大量的人的时间及无穷的心血，从而拒绝修正，那么你就能明白我们的意思了。一个对一纽约有 70000 行代码的模块研究表明，一个模块中以前的缺陷水平，对该模块现在的缺陷水平来说，是个很好的预测。换句话说，在对软件模块的评价中，从前的缺陷水平与现在的缺陷水平之间有很密切的关系。

3.3.5 bug 吸引 bug

如果 bug 有一个产生其他 bug 的趋势，那么这就可以解释为什么在你修复一个模块中的 bug 时，会在同一模块中弄出另外的 bug。一段时间之后，这些模块就被知道是易于出错的模块了。有时候，用类似焦土政策的技术通过消灭 bug 的生存环境来消灭 bug（也即，重新设计和应用这个模块），这样也许会更容易和更便宜一些。

对某个用 Ada 写的包含 282 个模块的命令控制通信的军事系统的一项研究表明，整个系统中有 20% 的模块包含 80% 的错误。而且，5% 的模块包含 42.5% 的错误，而 52% 的模块里没有错误。

有证据显示 20% 的问题和 bug，耗费 80% 的修复 bug 的开销。这就好像拿苹果比桔子一样，但让我们用来向上述两项中的数字来比较，看看有错误倾向的软件金钱耗费是多少。如果 5% 的模块包含 42.5% 的问题且 20% 的问题占去了 8% 的耗费，那么我们就可以估计出修复这些模块的耗费会超过整体耗费的 80%。有 bug 的软件模块确实是昂贵的。

3.3.6 证明缺乏理解

你能够解决一个你不理解的问题吗？你能够编程序解决一个你不理解的问题吗？都不是？好，那么我们就一致了，如果这两个问题中你至少回答了一个“是”，那么我们就不会急诊你的答案。但是我们要提出的问题是①你的解决方法的正确性和完备性如何？②你确定

你是在解初始的问题，还是在解决你根据不完整的理解而优化改进的问题？

一项对开放资源软件处理的研究（Apache 组织）表明，开放源码的版本和同类的商业产品相比，缺陷少的一个理由是开放资源的开发者也是这个软件的用户。这样，他对所要解决的问题就有了大量的知识。

我们的直觉告诉我们：这是一个合理的假设，而且与我们的经验相吻合。对所要解决的问题的了解，以及实际使用自己创建的产品，会有助于更好地解决问题。

作为软件开发人员，我们经常去对一些我们并不完全理解的东西编程，而且用于理解的时间从不被划入过程中。所以设计会在项目中间改变，现有的需求也改变，而且出现新的需求。那么，软件出现 bug 还有什么疑问吗？

你能够回忆起你编的第一个程序吗？我们一般不会记住的，因为会觉得尴尬。当我们是新手时，一个简单的程序可能要花上 10 倍的时间。我们也会犯最笨的错误，然后一直追踪这个 bug，都是因为对编程语言不熟悉。1990 年的 AT&T 电话系统的中断其根本原因，是不正确地旋转了一个“break”语句，因为程序员对语言的构造不太熟悉。下面的代码介绍在典型的 C++ 类定义中，新手所犯的一个经典错误。

```
1. Class ABuggyClass {
2.     public:
3.     ABuggyClass();
4.     ABuggyClass(const ABuggyClass &src);
5.     virtual ~ABuggyClass();
6.     private:
7.     int SomeDataMemberVariable;
8.     };
```

你发现问题了吗？你能推测出这个 bug 将如何发作以及特征是什么样的吗？修正很简单，但是除非你真正理解 C++ 如何工作，否则你是不会明白的。你只会盯着那些提示，并且抱怨编译器的 bug。除了用正确的方法修复其根本原因之外，你可能会做许多事情去修复特征。

注意：构造函数副本的参数应该是引用方式的，而不是源模式的。这样，正确的修复方式是：

```
ABuggyClass(const ABuggyClass &src);
```

否则，就会出现对构造函数副本的递归调用，直到堆栈溢出和内存用尽。为什么呢？这问题留作练习。

但即使你是有 20 多年工作经验的老手，当你闭着眼走进一个有 10000 行的代码的系统时，在进行正确的修改时，也仍然会出现问题。当对一个现存的系统不太理解时，是很难作修改的。你会害怕带来 bug，因为你不知道所做的变化会如何影响系统中其余的部分。为弥补这种情况并开始熟悉代码，你会去寻找文档，寻找那些写初始代码的人。并问许多问题。你需要理解你所修改的模块如何与系统的其余部分交互。

3.3.7 难的代码对大家都一样难

制作容错软件的一个技术是 N 版本编程 (N-Version Programming, NVP)。

在 NVP 中, 程序员中划分为多个开发组。每个组都被给了相同的需求描述, 并按事实描述相互之间独立地创建一个联系。NVP 背后的假设是: 如果每个组都独立地开发系统, 一个组中可能不再会犯另一个组中的错误。另一个假设是: 被不同的组开发的软件如果能够组合, 那么组合软件可能比独立开发的软件还要稳定。

有很多实验评价了 NVP 的作用。这些实验表明“错误是互相独立的”这个假设是不正确的。在一个实验中, 有 27 个为同一需求而独立开发的软件版本进行了一百万个实例测试。这 27 个软件版本中的每一个的可靠性都超过了 99%, 其中的 26 个版本的可靠性超过了 99.77%。实验的结果表明有 1255 个测试例子 (一百万个之中) 中有一个以上的软件版本失败。在另一个极端, 在 2 次测试中有 8 个版本失败。表面上, 一百万个测试中的 1255 次显得很低, 但是每个版本都有 99% 的可靠性, 这意味着每个版本最多只能有 10000 个错误。在现实中, 只有 4 个版本有 1000 多个错误。认识到这些, 1255 个普通的错误看起来就很严重了。通过对这些独立的可能进行建模, 报告的作者证明, 从实验中编程出现的问题看, 错误的独立性的假设可以有 99% 的把握被否定。

在 20 世纪 80 年代, 开发出了波音 777 客机。波音公司决定对其飞行控制软件采用 NVP 技术开发, 以产生一个更加可靠的系统。三个独立的组受命按相同的需求去开发这个软件, 使用三种不同的编程语言: Ada、C 和 PL/M。结果证明, 独立编写的程序在相同的地方会犯相同的错误。对于一个人来说, 难写的东西对另一个也难, 即使是使用不同的语言。最后, 波音放弃了 NVP 计划, 而把资源集中于系统中更加难的地方。

而且, 一定的编程语言对错误各有一定的影响。高度结构化的和类型分明的语言, 如 Ada, 其错误就比较好。但是, 对语言的选择一般并不影响那些与描述和设计相关的错误。相似的错误的产生与语言的选择无关。

如果不同的人根据相同的要求解决相同的问题时产生了相同的 bug, 这意味着:

- ◆ 要求的描述不清楚, 有问题。
- ◆ 我们都在相同的地方犯相同的使用或设计错误。

无论用哪一种方式, 代码的关键部分会很难, 因为它很难使用, 或者它很难被描述清楚。无论什么原因, 它只是意味着我们会犯同样的错误。

3.3.8 在软件生命周期中不同阶段的 bug 有不同的性质

一个典型的软件开发过程有 4 个不同的阶段: 需求分析、设计、编程实现和测试。每个阶段都会产生 bug。在一项研究中, 发现主要的缺陷 (65% 以上) 是在编程实现阶段产生的, 如表 3.1 所示。但是, 另一项对一个包含 150 万行代码的产品的研究展示另一个不同的结果, 如表 3.2 所示。

表 3.1 出现的缺陷和修复这些缺陷的耗费

阶段	项目 #1		项目 #2	
	出现缺陷的百分率	平均修复时间	出现缺陷的百分率	平均修复时间
需求分析	—	—	3	575 小时
设计	20	315 小时	15	235 小时
实现	75	80 小时	65	118 小时
测试	5	46 小时	17	74 小时

表 3.2 软件生命周期中不同阶段的错误类型

开发周期中的阶段	缺陷数量	漏过率	错误率	额外率
需求分析	203	63	34	3
开发计划	451	61	33	6
产品描述	1627	56	40	4
设计描述	503	53	39	8
Alpha 测试计划	472	71	26	3
Beta 测试计划	290	51	39	10
代码列表	228	26	45	29

这些区别的原因很明显。我们的推测是，这是由于不同的错误检测和计算技术，以及问题领域中的问题所引起的。这个差异不是为了告诉我们，为实验软件工程研究所做的一致而有意义的统计是多么难。但是，这两项研究确实利用软件生命周期，对 bug 及其关系进行了深入的观察。

表 3.1 说明，修复一个 bug 所要求的平均时间，在开发周期中的开始阶段是非常长的。比末尾时要长得得多，这表明修复需求分析中的 bug 比修复一个应用 bug 要昂贵得多。这个结果证明了早期的观察报告，即修复和改写软件在软件开发周期中的早期耗费，大约是在后期的 50 到 200 倍。

这里的含义是显而易见的。处理需求和设计阶段的代码的代价比较高。至少，早点对需求和设计的 bug 进行检测和纠正会带来好处。

表 3.2 说明，需求分析和设计阶段中的 bug，主要是因为忘记需求、忘记特征、忘记描述造成的。在实现阶段出现的主要错误可以归为差错，所以问题就是不正确的实现。令人惊奇的是（或不至于如此），实现阶段中 29% 的错误被归结为额外的特征，从而为系统的开发人员和用户添加了额外的复杂性，而且它们有潜在的破坏产品的概念上整体性的可能，而且需要额外的努力。他们对用户和维护人员来说都是潜在的问题。

3.3.9 稳定的系统中的 bug 的原因可能比错误的系统中的 bug 的原因更难以识别

从一开始创建一个系统时，就有出现 bug 的可能性。创建过程中，你一直在修复 bug，系统越来越稳定，系统就成为产品。在初始版本发行 2 周后，你会收到一系列的 bug 报告，所以你辛苦地修复 bug。慢慢地而且必然地，bug 报告的数量减少了，系统变得越来越可信。你以为最终可以把该项目放在一边并去进行别的事情。系统成为产品已经有 6 个月了，一切正常。突然，整个系统崩溃，而且你急于发现问题，这是一个软件工程师生命中平常的一天，对不对？

在创建系统时，你会专门使用一个调试器或内置一些诊断信息，以帮助发现 bug。在开发阶段中期，你对所开发的代码和系统已经很熟悉了，所以 bug 就容易发现了。

一旦你将系统发布给测试人员，你就会失去其中的一些内置的帮助工具。但在此期间你的努力不应增加过大，因为你可以访问测试者进行测试的机器，也可以与测试者交谈以知道他们在做什么，这样，你就可以重现这个 bug。你可能也有辅助的调试工具，这样系统就可以继续产生详细的日志，或者测试人员正在运行系统的调试版本，调试器仍可以使用。这时候，软件在你的脑海中仍然清晰，所以这个时候的 bug 不是什么大问题。

当软件在现实世界里发布以后，你可能仍然不能确定它的稳定性，所以你在系统中保留了帮助调试的东西以备用。但这时用户报告了一个 bug。你很难与用户直接见面并重现这个 bug。而且用户运行软件时的支撑系统可能与你的不同，你开始对软件陌生，bug 变得少见，并且难以分离。

一旦你认为一切都可以了，并开始删除那些帮助你发现 bug 并跟踪系统的辅助内容，问题就会变得更加复杂。而且，在这个时候，你已经不像从前那样对软件那么熟悉了，代码中的注释不再像以前那么有用了，而且你需要重新研究系统以确定问题是怎么回事。这个阶段 bug 通常十分模糊，而且难以触发，bug 的重现开始变得困难。第 2 章中描述的 bug 大多是从稳定的系统中报告的。表 3.3 总结了发现 bug 可以使用的工具以及要求的努力程度，这是从软件的不同成熟水平来比较的。

表 3.3 在不同的成熟水平调试软件所需的努力

bug 报告自	识别原因所需要的工作量	用来帮助识别的工具
开发	低	调试器，代码的跟踪和日志工具
测试	低—高	相似的系统？调试器？代码的跟踪和日志工具
初始版本	中	调试器？跟踪？日志？
稳定的系统	高	日志？

我们不会去相信有 bug 的系统，所以为那些系统保留了工具以帮助发现 bug。而且因为我们通常在软件内部修复 bug，我们就对系统比较熟悉，并且能够更快地发现原因。另一方面，一个稳定的系统不需要开发人员过问，所以我们不会太注意它。我们转向别的事情或去

掉用于调试的东西，因为它们影响系统的性能。但当有些东西的确出现问题时，就很难跟踪 bug 了。是不是有点讽刺意味？稳定性也需要代价。

3.4 小结

我们对 bug 的分析，现在已经为我们提供了许多有意思的发现：

- ◆ 一个软件缺陷是我们在描述、建造、设计或实现一个系统时犯了错误所造成的。
- ◆ 当软件缺陷存在于软件系统中时，系统可以展示未预期的行为。
- ◆ bug 是软件所不期望的行为，而且我们必须答应排除它。
- ◆ 调试是识别和纠正产生 bug 的根本原因的过程。
- ◆ bug 并不是随机的和神秘的事件。bug 是能够重现的，但一些 bug 可能几乎不可重现。
- ◆ 人们对于给定的相同的问题会产生相同的错误。
- ◆ bug 可以在一个变化的环境中生存和重现。
- ◆ 在软件开发过程的不同阶段中产生的 bug 有不同的特征。
- ◆ bug 产生是因为我们并不理解描述或语言结构。
- ◆ 一个易产生 bug 的模块包含系统中的大多数的 bug，而且耗费了用于调试的大部分力量和资源。但在一个稳定的系统中发现 bug 是很困难的。

对这些观察结果浏览之后，就能看出许多潜在的避免和识别 bug 的方法。这些发现会作为本书中许多技术处理过程的基础。为什么产生 bug？如何产生 bug？又是如何让 bug 从测试中溜走的？在下一章，将继续对 bug 分析。既然现在对 bug 已经有些了解了，我们就能够提出更多揭示性的问题，这些有助于今后形成更好的策略以写出更好的软件。

3.5 bug 问题

我们在本章开始处描述的“快速消息传递”，也有一个另外的特殊行为。“快速消息传递”允许你发送统一资源地址（URL），包括你的朋友或其他终端。朋友可以点击这个 URL，只要它已经出现，然后就能显示出你所发送的 URL 所定义的网页。这个快速消息传递在 Windows 环境下运行，而且有时候当你点击这个链接时，浏览器会被唤醒来显示这个网页。其他一些时候，“快速消息传递”被挂起没有反应。可以点击 URL 来锁定整个机器。有时候，只有浏览器失常而且变得没有反应。其他时候，“快速消息传递”会在你选择 URL 后消失。如何解决这个问题？

第 4 章 bug 的生命周期

“我怎么会这么傻？”我的邻居在经过 18 个小时的连夜寻找产品系统中的 bug 之后，足足抱怨了整整一个早上。他的神经被含咖啡因的饮料刺激得兴奋了，眼睛血红，他的桌子上堆着相当厚一层头发，这是他从已经谢顶的头上拽下来的。“我到底在想什么？”他咕哝着说，气愤地编写出对 bug 的修复，并发布了补丁。这天上午，在彻底筋疲力尽而崩溃之前，他向项目管理者递交了关于 bug 及其解决的报告。听到这些，管理者摇头说：“这个 bug 怎么可能通过了测试部门呢？”疑惑之后，她又说：“他们真的测试了所有的东西吗？”

这是一个常见的情形吗？在我的软件工程师生涯中，我也曾一次又一次地说过这些话。倒霉的开发人员和迷惑的管理者之间的感觉是不同的。但问题还是没有答案，为什么会出现 bug？bug 是如何产生的？bug 又是如何躲过测试的？

4.1 为什么会出现 bug

每天，我们每个人、所有人都会产生 bug。看起来，好像我们正把每天的大部分时间花在处理各种形式的 bug 上了。“是人就会犯错”，这句谚语是个忠告。作为软件工作人员，我们犯过太多的错误。为什么？

4.1.1 复杂性

我们之所以犯错，是因为人类的思考过程对于复杂系统和实时交互来说不太够。对人类计划和决策过程的研究，显示了人们在处理复杂问题时遇到的困难。

什么是复杂性呢？复杂性和系统中独立变量的数量相关。在系统中这些变量之间的大量变化及其相互依赖关系，导致了一个更加复杂的系统。

软件程序复杂吗？一个 59 行的程序在所有的输入下，有可能的 36000 个执行路径。多数人都无法去计算和思考这些组合。记得我们最后一次编 59 行程序是什么时候吗？现在一个平常的程序也要 100,000 行以上，Windows 2000 有大约 36,000,000 行代码。想想这样的程序会有多少可能的执行路径，再想想所有可能出错的地方。对任何人来说这都是一个很难的事情，而且人多了之后这也不会变得更简单。

在软件工程中，复杂性存在于多个层次。问题本身可能就很难，使得想解决问题的人很难理解并去考虑其中的变元和因素。一个复杂的问题也会使得与需求之间的清晰的通信变得很难，问题解决起来也很难，意思是可能很难选择和使用一个解决措施，而且一旦某个方法被运用了，就很难维护和支持这个方法。用来实现这个方法的工具可能很难，而且使得这些

工具很难正确而有效地使用。团体中成员之间的交流也会很难。如何有效而正确地播发消息？如何调整工作努力和相互接触？如何处理不同的人格和行政日程？而且，软件开发过程可以是复杂的，从而成为 bug 的根源。

问题的复杂性

几年以前，我在一家合同公司工作，在那里为一个大型企业做事。我被分配了一个项目，按要求开发软件并为用户提供相应的软件包。软件背后的思想听起来很简单。企业出售大量的产品和服务。其中的一些产品和服务可以被加到包里，销售人员再将这个包推荐到只需要个人产品的客户那里。软件要求考虑用户的大体情况，评估一个包，并告诉销售人员哪些包可以销售。你每天都处在这种情况下。例如你去一个随时快餐店，只要一个汉堡包和一杯苏打水。但在走进商店中之后，就会发现再加 99 美分就可以买一份很好的午餐，里面不只有汉堡包和苏打水，而且有些法式油炸品。

这个概念听起来简单，毕竟，这种情况是我们日常生活的一部分。它每天都发生，而且可以发生在任何地方。但如果要将这些市场规律完全写成代码可能就是一场噩梦了。规则太多了，而且规则的例外又太多了，使得很难做出一个令人接受的需求描述。只有顾客已经拥有产品 C 时，才能将产品 A 与产品 B 组合。如果一个顾客生活在一个特殊的区域，他就不会买产品 D，除非他已经是这个公司的俱乐部的成员。我还能举出许许多多这样的相反的商业规则。没有人能知道这些规则的前身是什么，而且看来没有人能够了解所有这些规则，也无法列出所有这些可能的规则。来自于国内的一个地区的商业人员会希望一个事情这样，而来自另一个不同地区的人可能会希望那样，没有人理解我们奉命解决的问题，即使那些需要这个软件产品的用户也不会理解。更坏的是，这个大公司有三个独立的系统包含有客户信息，其他的四个公司包含有不同的产品和包的信息，没有人知道我们从哪里可以得到所有被要求的信息。我们花费几个月的工作找出需求。最后，客户发现了情况的困难性，并无情地枪毙了这个项目。

注意：我从这个项目中学到的教训是，不要用软件解决市场问题。人类的思维太过于有创造力了，以至于不能被一组算法所局限。

到底是什么使得一个问题变得复杂起来呢？在变元以及影响问题的因素数量超过了我们的处理能力时，这个问题就是复杂的了。在 1956 年，George A Miller 观察到，人类的思维在一个时刻只能处理大约 7 个概念。

这表明在处理那些需要同时应付 7 个变元或因素的问题时，我们会感到困难。如何判断一个软件项目是否复杂呢？下列特征也许会有助于识别一个复杂系统。

- ◆ 需求文档的页数比较多——如果我们只能同时处理 7 条信息（在这个例子中是 7 个需求），而且需求文档项目数多于 7，那么在我们考虑解决方案时就要放弃一些需求。有意或无意地放弃一些需求，并不是降低复杂性的办法，因为那些被放弃的需求在后来会体现出来的。如果是无意的，意味着你已经把它忘记了，而且它最后会成为一个 bug；但即使一个需求是有意去掉的，那么你在设计中就不会考虑这个需求。那么在后来的需求中又浮现出来

时，这就会是一个麻烦。

◆ 经常改变需求——如果需求老是改变，那么显然客户本身就不太理解他们的需求，而且问题很复杂。如果客户理解自己的需求，那么他们在清晰地表达需求时就不会有任何问题。如果他们转回来告诉你一些新的需求，那通常表示他们以前漏掉了一些需要的特征。为什么呢？因为他们被 7 条信息的智力水平所限制。

◆ 不停的询问—回答—询问过程——如果开发人员老是跑到项目管理人员或需求分析人员那里，询问如果这种或那种情况发生时他们应该怎么做，那么这个问题就太复杂了。这常常意味着需求不完全，需求不完全的主要原因是问题的复杂性。

这三个准则看起来比较适合于我曾经遇到过的每个软件开发项目。有没有什么我们可以做的，以减少和管理复杂问题？当然，我们在日常的问题处理工作中无时不在做这些。在认知心理学中，“chunking”这个词被用来解释人们如何把一个复杂的问题分成若干可以管理的小片。而复杂是一个主观的衡量。我们从经验中学习。我们的经验形成一个完全形态，其中一些变元被压缩为一个，当我们必须减少要考虑的变元数量时，我们就降低了复杂性，对于一个职业滑冰选手来说，滑冰并不比走在大街上难。对于一个初学者，在两个冰刀上掌握平衡而不摔倒是个艰巨的事情。当我们对一个问题比较熟练了以后，它就没那么复杂了。我们可以发现一些能够降低可感知复杂性的方法。在第 12 章，我们将提供一些关于如何降低软件工程问题复杂性的思想。

方法的复杂性

很不幸，复杂的方法有时候也是一些复杂的问题。网络商务系统就带来了复杂的问题。网络商务系统不仅要使用买卖商品的业务逻辑，它还要处理安全性、性能和容量。系统必须处理大量的交易，同时每天和每星期都要保持一些最低的性能标准。

我们如何解决网络商务系统带来的问题呢？方法是多线程分布式计算机系统，这就和在嵌入式系统或并行系统时做软件工程一样难。多线程应用系统很难调试和应用，因为有时和同步问题，另一个原因是开发者和调试者很难在某个给定时刻看到系统的所有状态。分布式计算带来了大量的计算机互联工作的思想，而且将“系统工程”的概念引入了纯软件世界。软件工程师不仅要考虑运行在单个计算上的应用中的 bug。我们还不得不在没有训练或知识背景的情况下，学习做传统意义的软件工程师的工作。比如失效、冗余、容错性、可扩展性以及生存能力等，都成为了必须面对的问题。我们的方法中会出现更多的相互依赖的变元（比如计算机的数量）。

如果体系结构的决策会给方法带来复杂性，那么方法的设计也可以。一个对 ADA 程序设计的研究表明，“上下文结合”的设计方法与缺陷的数量有关。

什么是“上下文结合”？两个软件模块在需要相互交互的时候是上下文相结合的。例如，一个软件模块调用另一个软件模块中的函数和方法，或者一个模块使用另一个模块中的资源。假定软件模块就是软件系统中的变元，增加系统中的上下文相关的方法，就意味着增加系统中变元之间的相互关联。从对复杂性的定义来看可以得出结论，即增加设计复杂性，就

可能增加缺陷和失效的数量。

还有许多其他人试图评估一个软件系统的复杂性，以及表示出这些评估与软件系统中缺陷的数量之间有什么关系。代码行数是最简单评估的方式。有一种复杂性评估方法，到现在都一直经受住了时间的考验，它就是 McCabe 的记转器复杂性评估方法。

它最初在 1976 年提出，主要以软件系统中的判断路径的数目来评价。许多其他的评估方法到现在也存在。但是，不管软件的复杂性如何评价，所有这些研究都支持相同的结论，即增加复杂性会增加缺陷的数量。

在第 12 章我们也讨论了一些方法，能够有助于降低和控制方法里可能出现的复杂性。

工具的复杂性

在研究完处理复杂问题的复杂方法之后，让我们选择一个难以使用的工具。不管如何挖苦，软件工程的辅助工具在一方面变得越来越好，另一方面却越来越差。高档次的开发语言，如 ADA、Java、Visual Basic 和 C++，比起往日的汇编语言要好多了，那些代码产生工具，如 C++ Builder，减轻了工程产生冗长代码的痛苦。对现成产品的重用，如数据库服务器（Oracle、Informix 等），中间件（企业资源规划系统）和类库（MFC、Rogue Wave 的工具类、Java 基础类以及 Java Swing 工具包 API 等），它们降低了重新开发一些东西的需求。配置管理和 bug 跟踪工具帮我们管理产品的进化。调试工具帮我们检查 bug 的源代码。这些工具很有效，并且提高了我们的生产能力。

但是软件工业中没有被经常讨论的一个细节，是使用这些工具时的折衷。我们认为，如果正确使用，这些工具能够确实提高生产能力，而且最终能够带来良好的效果。但是我们发现很少有关于这些工具的非金钱消耗的严肃讨论。至少，利用我们所选择的每个工具时，我们要花费一定的代价去学会它。我们不光要学会如何用它，我们还要知道它的优点和缺点。而且我们要知道使用这些工具的恰当时间和地点。盲目地信任市场上的小册子，肯定会带来灾难。

有许多的训练教程来教你如何写 C++ 或 Java 代码，或者如何成为某家工具厂商的认证工程师。但是其中没有什么教程对这些工具进行比较，或者揭示使用某种工具时潜在的缺陷。这些教程主要集中于一个方面，即如何使用这个工具。我们知道，了解每个工具的优点和危险会给我们的日常工作带来奇妙的帮助。

调试器的伟大之处在于发现了一定种类的 bug，但正如我们要在第 7 章中介绍的那样，它们也有自己的缺点，它们可以改变那些被调试程序的行为。

好像你的软件还不够复杂似的，对调试器的使用又增添了一个层次的变元，而且在评价调试结果时必须考虑它。如果依靠调试器来修复所有的 bug，那么早晚都会遇到一个调试器所无法修复的 bug。

现成的软件产品，如数据库服务器和企业资源规划（Enterprise Resource Planning, ERP）系统，提供了大量的预先做好的功能。它们如果从零开始做的话是很难的。但是，这些产品总是要对所有的人提供所有的东西，使得他们因为多样性而失去了简单性。它们很难配置、

调整和维护。由于这些系统还是专用的，而且作为黑箱来使用，当有 bug 的时候，就很难分离出根本原因。如果这些系统并不复杂，那么我们为什么还要付给数据库管理人员和 ERP 系统顾问们这么高的薪水呢？其他的现成产品，如类库，在降低复杂性方面表现得并不怎么样。类库迫使你必须遵从它们的框架和哲学，它们经常迫使编程人员整个地或部分地使用它们的包。如果能够理解他们的设计背后的框架和哲学，那么这就不是一个问题了。但是，如果你与类库不相一致，那么就好像试图把一个方形的楔子钉进一个圆洞中一样。图形用户界面的开发人员可以一下子轻易地做出一个图形界面的原型。这里点击一下，那里拖放一下，再看看就已经有了自己的一个对话框。这是非常棒的——直到出现了一个 bug。这时你将不得不分析那些并不属于你的代码，解释那些从前根本不知道其存在的类和函数，并且正确地描述代码如何工作。你曾经试图用 Visual C++ 的 COM 向导产生一个 COM 组件吗？当用一个界面试图创建一个 COM 对象时，是很容易的。但是如果想用 COM 做更多的事情，而且这常常是必要的，那么向导可能就完全无用了。这就是代码产生器的另一个问题。有时候它并不为你做所有的事情，结果，你就不得不在这里把一段自动生成的代码去掉，在那里又要自己定义一个类，以使得它能够正确地工作。当你想做一些事情而代码产生器的作者不想让你做时，工作的复杂性就增加了。你不光要知道如何做开始的工作，还要知道代码产生器想为你做和不想为你做的事情，以及如何对待这些限制。

对编程语言的比较，可以比作一次激烈的圣战。我们会避免这些，避免使这本书因此成为 2000 页的大部头。但还是要使它能说明每个编程语言都有自己的优点和缺点。这样，在设计中，要尽力使用语言的力量并设计软件，以免陷入软件的缺陷中。例如，Java 的垃圾收集机制对开发者而言是一件好事。垃圾收集使得他们不用担心不稳定的指针、破坏的内存空间和存储器漏洞。但反过来，Java 中的资源遗漏很难发现，而且垃圾收集理所当然地有些不足之处。Java 中的资源遗漏是在你忘记删除一个对象时产生的。例如，你创建了一个对象，里面有数组，数组中包含一些元素，但如果不彻底地将元素从数组中删除的话，那么这些元素就可能无法再分配。Java 中的资源遗漏也发生在全局静态类中；所以，在设计时和使用某个单一模式时要注意。

虽然 Java 的垃圾收集算法是很高效的；但是它可能会在最不合适的时候执行，并且使用 Java 对实时系统可能带来危险。

注意：以上这些可能是正确的。Java 一直在进步和变化，所以很难知道它目前的性能描述和特征。虽然 Java 一直比较新颖和令人兴奋，但持续的发展也使得语言更加复杂。有许多诸如“我在用哪一个 JDK？”“安装了哪一个 JAR？”以及“我如何表示？”之类的疑问，证明了在使用该语言时附加的复杂性。

还有许多其他的例子，但是我们的目标不是贬低软件工具。在平常的环境和过程下，它们可以极大地提高生产能力以及工作质量。我们只是想指出在使用这些工具时的潜在规律，这样你就可以考虑这些问题，从而判断哪一个工具适合在项目上使用，并且判断在选择某工具集的时候可能遇到和需要解决的危险。

人的复杂性

人的复杂性，在单个人完成的软件项目中并不是什么问题。但是，由单个人完成的项目很少。一个典型的软件项目要有一个项目经理、一个技术领导、一组开发人员和一组测试人员。所有这些人必须学会与其他人交流协作。项目经理必须与工作组交流该项目的问题、需求和目标。测试人员需要理解软件试图解决的问题，这样才能测试正确的行为。开发组之间必须相互交流，这样他们的每个部分都能整体化并且作为一个单元共同工作。

交流在任何工程中都是一个问题。但是这个问题在软件世界里面就比较大了。软件工程作为一个规则，并没有哪种标准的“蓝点”语言，能让我们相互共享以便交流手上的任务和设计工作。设计模式（design pattern）的进步以及统一建模语言（Uniform Modeling Language, UML）的出现，已经帮助人们降低了许多交流上的问题；但是它们并没有解决这个问题，而且还带来了自身的缺陷。我们所拥有的用来交流的最精确的语言，很不幸地，就是程序语言本身。我们能够用一个编程语言帮我们交流吗？答案是可以的。我们将在第 12 章研究其可行性。

人们可能会为不同的原因而制作一个复杂的项目。有时候，这些原因可能甚至会是“利人主义”。项目管理者可能对工作过于热情并接受了每一个需求变化或附加特征，因为他认为这对公司有好处，这样整个工作组就要遭罪了。我们常常听到一个特殊的特征必须马上实现，否则就是末日了。即使这个特征需要花 10 个月才能够实现，公司仍然会要求在 2 周内完成。

实际上复杂性也会来自技术组中固执和草率的成员。这些人可能会在项目中较劲，因为他们总以自己为中心，而且他们认为任何不满足自己提议的方法都是错的。

另外一些人想把问题和项目弄复杂，因为他们只想满足自己的行政目标。可知情的软件项目成员往往是处于错误的地方和错误的时机。如果你正在给某个部门编写一个会替代他们一半人员的软件，那么他们把你的工作变得容易的动机是什么？所以，那些将会被你替代掉的工作组成员会尽可能地把项目变得对于你来说非常难。他们会一直改变需求。他们会保留资源并且拒绝回答某些问题。

软件工业中人人都谈及文档的重要性。但是开发人员写文档的目的是什么呢？为什么他们要让自己被他人挤掉呢？当公司不给他们足够的时间来写的话，他们为什么要花时间去写一份他们知道没有人会读的文档呢？

注意：为什么有些人会想使你的生活困难呢？有无数的原因。有时，这些理由不是个人的，有时是你可能以为这些话题与一本关于 bug 和调试的书没什么关系。它们可能不会对你修复 bug 有直接的帮助；但是理解这些问题，会有助于理解你所接手的系统为什么会是这样，并帮你判断那些可能的选择。有时候，bug 的根本原因可能与你所检查的代码没有什么关系。bug 产生的原因可能就是你的项目经理或者同事，或者公司本身。你想（或者说你敢）修复这样的 bug 吗？如果你只处理其表面症状，你感觉自在吗？早点意识到这些，会避免在后来遇到大量不开心的事情。

过程的复杂性

一天，我们正在产品站点升级我们的网络服务器，突然一切都陷入混乱。每样东西都开始乱，即使将更改过的东西复原也没什么用。经过两个小时的苦战，我们已经能将问题分离到一个 JAR 文件中了，而该文件我们原以为是删除了的。恢复脚本并没有删掉这个 JAR 文件，因为脚本并不知道我们要删掉该文件。安装脚本也不知道这个 JAR 文件的存在，所以它不会警告任何人说系统中被加进了不该有的外来库。

共享库、动态链接库（DLL）和 JAR 文件的初衷是使得新软件的开发变得更容易。这些 DLL 的一个优点是执行体积更小。因为库不被编译到可执行文件中，所以可执行文件就比较小。但是这些库还仍然要存在；所以如果此 DLL 被一个单独的可执行文件使用的话，整个的体积差别就不是巨大的了。但是通过将库与可执行文件相分离，现在已经能够在多个执行文件中共享库了。这样两个或更多的可执行文件就可以共享相同的一组库了，而且可执行文件的整体体积可以缩小了。使用 DLL 的另一个优点是：对软件的升级和修改可以成为对库的升级和修改，只需更换一个特定的库模块而无须更换整个程序。而且如果升级只是修复 bug，那么所有使用这个模块的程序会同样从修复中得到好处。

但另一方面，如果没有一个用于控制软件发布的好的过程，那么在升级 DLL 时可能会毁坏整个系统。二进制的配置管理，和源代码的配置管理一样重要，并不是一个简单的任务。某一天，我正在使用 Java 的 XML 解析器。每次试图编译代码时，我就会遇到一个错误，提示说一个正在使用的类的方法并不存在。但这个方法在 Java 文档中已经定义了，在我将类对象反汇编之后，那个方法也存在。我挠了一阵子头发，然后开始排除 CLASS PATH 中的变元直到能够编译代码。后来发现在 CLASS PATH 设置下有包的不同版本，而那个包里而并没有所定义的方法。如果这是发生在运行时，程序肯定会提示出现异常，而且操作人员很难发现其原因。

源代码的配置管理，在只有一个开发人员和一组源代码时是容易的。但是有些时候，一组软件开发人员开发“多个”一次性产品，他们工作在相同的源代码上。这就意味着要维护一个程序的多个版本，它们的行为相似但不完全相同。这就导出了分支和合并的概念。如果某个分支中发生变化，那么另一个分支发生变化的可能性就会增加，因为我们很少从软件的另一个分支来考虑去验证这个变化。但是当一组人在同一时刻工作于相同的计算机之上时，过程的复杂性会显著地增加。你能阻止别人改写你的内容吗？你能保证你的版本始终与“正式”发布的相同；以便当你回查一些东西时，这些改变不破坏系统吗？你如何在同一时刻同步所有的变化，你如何保证另外没有人正在改变一个你所依赖的资源。有一些软件工具能帮助我们减轻来自于配置管理的风险；但是，它们也仅仅只是工具而已，而且人们将仍然迷惑并犯错误。

4.1.2 现实

我们都会犯错误，因为现实是苛刻的。更便宜、更好、更快、市场时间、现在——这些

词语都很紧迫并迫使我们走捷径。我们所写的原型，作为概念的一个证据已经被导入产品。紧要的最后期限很容易使得我们只考虑外部特征并让客户和管理者不再嘲笑我们。而且我们通常只在上午的几个小时里修复；谁能够一直思考到凌晨 3:00 呢？市场部门不断地加过来越来越多的新的特征，一直到代码固定的时候。所以我们就放弃了优秀的体系结构和设计，而发疯般地不加思考和控制地堆加一个个的特征。单元测试被紧催，全速前进。文档？什么是文档啊？

Foote 和 Yoder 提出了一个现实驱动的软件体系结构，叫做“泥中的大球 (Big Ball of Mud)”，而且还有从它所导出的一组相关的反模型。“泥中的大球”的体系结构包括一些随意的结构和空心类的代码；它容易令人想起传送管带和轨道线。开发的系统代表临时暂住区，在这里堆积了许多的变化和软件模块，没有考虑对整体蓝图的影响。“泥中的大球”体系结构是不规则的和不太美观的，然而却是流行的和普遍的。它是不规则的和不太美观的，因为该系统是权宜改变的结果。但是它又是流行的和普遍的，因为它提供的功能服务于每个目的，而且它们是不可避免的。这些系统是现实的压力产生的：时间、耗费、经验和变化。

- ◆ 时间不充足——一项对一个商业系统中的 40800 行代码的研究显示，在开发时间被延长以后，缺陷的数量就会减少。

- ◆ 在这项研究中，将开发时间延长了 13 个月或 15 个月会将缺陷的数量减少到一半。当有了更多的时间以后，工作人员就有更多的时间做“正确”的事情。通常永远也没有足够的时间来把所有的东西都纠正，我们的经验与这项研究是一致的。即使管理者很善良和开明，而且同意开发人员按自己的计划工作；开发人员作为一个整体，也有低估所需劳动量的趋势。开发过程中往往有一些不可预料的突然麻烦：bug 的数量比预想的多、不好的设计决策、新的需求或顽固的无法分离出来的 bug。下一个所知道的事情就是将要比计划晚 3 周。再下一个所知道的事情就是将要比计划再晚 3 周。当没有时间的时候，所有的礼节都被扔到了窗外，因为管理者和市场部门不能再等了。这样软件就必须立刻完成，bug 也随之出现了。

- ◆ 没有足够的钱——当你给开发人员更多的时间去思考设计和实现时，你将增加项目的开发耗费。当你增加更多的形式过程时，比如接见会议，你就占去了工程师的时间，开发的耗费又增加了。培训会消耗掉钱，工具也要钱，正常的测试也要钱。在任何组织中，及时按预算完成任务的压力是巨大的；所以一个诱惑就是采取捷径来满足这些限制。尽管会有潜在的长期而更高的耗费，这个诱惑仍然是不可阻挡的。

- ◆ 持续的变化——在第 3 章，我们指出 bug 趋向于与变化相联系：对系统的变化，对环境的变化，对所涉及的人的变化。为什么？软件系统对问题的设计是一个单独的版本。系统声明：“这就是世界的行为。”但是当出现一个需求变化时，世界就不会再按照期望的那样进行了。当环境变化时，世界就会变化；而当人变化时，这个声明本身就变化了，因为项目的新负责人的看法会不同于以前的负责人。当这些基本的事件发生时，正确的事情就是重新思考和重新设计系统。这些都很少发生。实际中，系统趋向于被改变和干扰以满足需求。“这就是世界的行为”这个句子就可以变形为“这就是系统在没有下列特例时的行为……”系统

变得越来越复杂而且越来越难于理解——随之 bug 就繁荣了。

◆ 缺少经验——在没有去过一个地方时，是很难知道其最优路径的。缺少对一个问题领域的知识，可以是一个重要因素；缺少对提出方法的经验，则是产生方法中的复杂性和 bug 的另一个重要因素。在一个软件开发和设计过程中，有无数的方案和设计决策要做。如果以前从未经历过这些，那么在开始做这些事情之前，是很难理解这些决策的后果的。我们都明白让一组 C 程序员去做一个 Java 项目上的工作，是会带来灾难的。同样，当你硬让开发人员接受一个他们根本不了解的设计模式时，也会带来灾难。你可以给他们所需的所有的训练和教程，但教程不能代替经验。教程并不能告诉你什么时候使用某个设计模式是恰当的，也不能告诉你会遇到什么风险。这是通过经验学来的。人们常常需要一个实验，或者一个试验错误时期，以学习那些问题和方法相关的东西。但由于时间和耗费的限制，这些其实都很少做。所以我们一般会使用一些路径，从一点折到另一点，看起来好像没有什么理由似的，而不是使用一个单独而完美的方法。判断变得更加困难了，方法也发生变化，bug 会慢慢地滋生出来。

在开发过程中采取捷径，会增加方法的复杂性并降低系统的可理解性。维护人员由于没有时间理解软件，可能会一直降低软件的结构完整性。这种情况会一直持续到软件的管理人员认为对软件的维护耗费远远超出了创建新程序的费用为止。受现实左右的软件开发和维护过程的最终结果，使软件系统的可理解性降低。

那些由受现实左右的体系结构和设计产生的软件，会不会比由定义的能够区分代理和现实的程序和过程开发出的软件的 bug 多呢？很难靠经验来通过什么方式给出这个问题一个具体的答案，因为大多数的软件项目都要受现实左右。惟一的例外可能是在类似 NASA 和军队中的情况，在那些地方出现任何错误的代价都是天文数字，因而开发人员无法赔偿错误。一些新闻事件表明，在一个典型的软件产品中，每 1000 行代码大约有 25 个缺陷，而 NASA 的航天飞机软件一般每 10000 行代码只有不到 1 个错误。但是生产 NASA 软件的耗费，则是生产一个典型的商业软件产品的耗费的许多倍。有报告说航天飞机软件每行代码大约耗费 1000 美元，而商业软件则只要每行 5 美元。

这是现实导致的，不是吗？

4.1.3 人类的弱点

我们犯错误，因为我们脆弱，会遗忘，并且不坚定。我们常常将“==”错写成“=”。我们会忘记一个曾经分配的指针。我们太懒了，不想输入更多行的代码来对函数输入的有效性进行检查。我们使文档和实现不一致，我们理解错了文档。我们写一些无用的文档。我们不考虑边缘效应，我们也过于自负，以自我为中心。我们不考虑那些当前还没出现的问题。我们认为某些问题是琐碎的，所以对它就不怎么在意。我们以为自己知道如何解决一个问题，其实还不理解问题本身。我们容易出错，因为我们是人。

有上百万条理由来说明人们为何出错。我们没有正确的经历。我们没有恰当的技巧集。

问题太难了，我们困了。在此我可以一直举下去，事实就是我们都犯错，而且这些错误会变成 bug。如何避免愚蠢的错误呢？第一步是承认我们是会失败的。一旦接受了这个简单的事实，我们就能够接受那些能够避免错误并使得我们能更快地发现错误的修正和新思想。有些错误可以轻易避免，另一些可能比较难；有许多可以使用的步骤和过程能够避免愚蠢的错误，也有许多可以使用的工具有助于识别和发现 bug。在第 12 章，我们将详细浏览这些工具和技术。

4.2 bug 是如何产生的

复杂性、现实和智力缺陷是产生 bug 的三个基本要素。在软件工程中，这些要素相互组合，并且从各种方式中产生 bug。每当修改一行代码时，都要冒给代码增加 bug 的危险。bug 的产生可能是由于没有一个好的需求说明来告诉我们要写什么东西。我们尽量聪明起来，然后提出一个很难实现的方法，开发人员会对系统有一个不同的智力模型，并带来介于组件间的整体性错误。最后，我们还有比较愚笨的人为差错和 bug。

4.2.1 对软件做修改

在修改代码的时候，就可能给代码带来缺陷和病毒。很简单，在第 3 章中我们列举了大量的源于实验的结果，证明了修改和软件缺陷数量之间的关系。在修改中有三个主要的导致软件缺陷的原因。主要的原因是对被修改的软件的理解力不够，这一点可以影响到软件生命周期中的开发和维护阶段。一个组件的设计和开发过程中缺少与另一个系统组件之间相联系的考虑，称为视野狭隘。最后，特征的变化或需求的修改，也能对软件的缺陷水平造成影响。

理解力不够

理解力不够可以在两个方面影响 bug。第一个是对问题的理解不够，以及在开始的设计和开发行动中对方法的理解不足。这可以导致差劲的描述，在后面会讲到。另一个方面是发生在维护阶段中需要对系统作修改时。当你并不知道所做修改将如何影响系统时，是很难正确地修改软件的。理解得不好，可能是因差劲的文档或设计和使用导致的。

什么是文档？文档是一个书面报告，能够帮助描述软件系统的使用、设计、目标和描述。代码是软件的最后仲裁工具。当所有其他的文档资源对我们都没什么作用时，我们就要靠代码来理解系统了。所以，我们应该把源代码看作文档的一部分。

差劲的文档有如下特征：

- ◆ 缺少文档——许多组织并没有设计文档需求。许多程序员都不写注释来解释代码的期望行为。理论上，代码必须有自身的文档。我们认为每个开发者应该能够阅读代码以明白代码的行为，但代码的实际行为可能会与代码的期望行为很不一致。这里任何一种对问题的描述都是有幫助的。

- ◆ 误导的文档——对代码作修改后，却往往忽略了修改代码中相应的注释。维护设计

和需求文档可能需要大量的工作，而且这些工作在软件的整个生命周期中很少能做得好。当文档和注释过期时，就意味着它如同红色的鱼引导狂乱的鹅那样误导我们。当我们认为代码中的注释是正确的，而且作了相应的修改后，我们就要依赖于文档的正确性了。如果它是错的，我们就会造成一个潜在的 bug。

◆ 写错了对象的文档——我们通常是那些已经对软件比较熟悉的人写文档，但并不为那些对系统不太了解的人写文档。

Java 文档是直接从源代码中建立应用程序接口（API）的一个很好的工具。它鼓励开发人员写文档，因为它使得写文档很容易并满足其过程。作为一个参考工具，它是很棒的。但是这些 API 文档在你还不熟悉如何使用 Java 时，并不起什么作用。Java 网站通过提供 Java 教程来解决这个问题，给初学者一个挑战的机会，当我们写文档时，我们趋向于为“熟练”的程序员写文档。这些“熟练”的程序员已经对软件比较熟悉，而文档只相当于一个参考工具。软件项目几乎不对新雇员提供教育性的产品，但这些指导的用处不只在训练新的雇员。指导教程中的例子和讨论，也会帮助开发人员对他们的设计哲学和软件的期望用途进行交流。软件的期望用途是一条被大量执行过的代码路径。所以，使用这个大量测试的代码路径时遇到缺陷的机会就很小了，这意味着最终产品中有更少的 bug。缺陷的数量可能是相同的，但 bug 会更少。

视野狭隘

在编写软件的时候，我们通常只考虑目前正在处理的特定代码段。但是我们所实现的功能和开发的软件模块并不是单独起作用的。相反，它们通常只是系统的一部分并持续地进化，当我们不考虑那些其他的变元时，就将增加产生 bug 的可能性。

在软件的设计和构造阶段中，通常不考虑易于维护性。我们通常不考虑如何使得程序对于那些可能需要维护软件的人来说易于理解和修改。考虑下面的网络服务器的体系结构。系统预计要为一些客户提供一个服务。对每个客户的核心服务是相同的，由一些后台服务 API 调用组成。但是每个客户都要在其服务中加入自己的特别的业务需求。在图 4.1 中的体系结构中，每个 Java Servlet 页（JSP）都实现了客户的特殊的服务，方法是使用后台服务器的 API 调用。

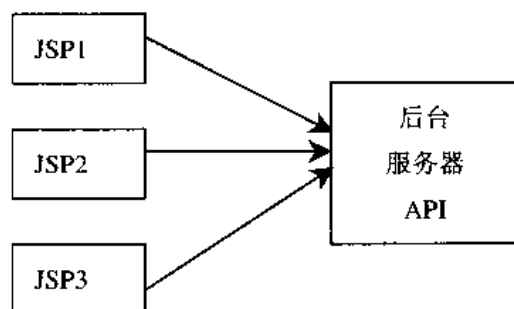


图 4.1 一个 JSP 网络服务器体系结构的例子

在这个体系结构中, 如果后台服务器 API 发生变化, 维护人员将回溯所有使用这个已经变化了的 API 的 JSP 页并修改它们。当只有三个页时, 这还没什么。但是如果业务比较好, 而且有 20 到 30 个 JSP 页时, 维护的工作就变得比较繁重了。而且每个 JSP 页应该因后台服务器 API 的变化而进行测试。一个能够避免这些麻烦的方法是提供一个附加的层包装那些通用服务组件, 如图 4.2 所示。

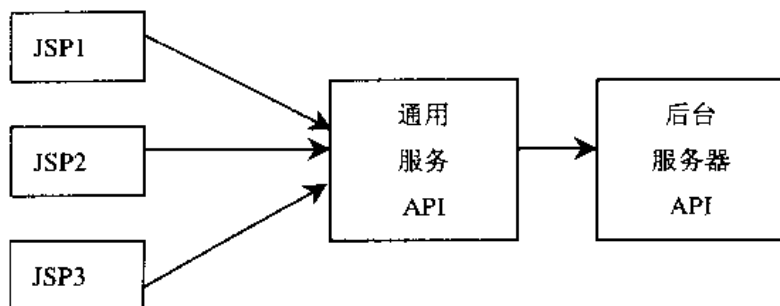


图 4.2 一个更好的 JSP 网络服务器的体系结构的例子

在这个体系结构中, 通用服务 API 对 JSP 页隐藏了后台服务器 API 的复杂性。对后台服务器 API 的修改, 只需在通用服务 API 中测试和修改。如果通用服务 API 没有改变, 则只需检查一个组件, 而不用修改和测试三个 JSP 页。系统的可维护性提高了。

一天, 我们收到了一个请求, 内容是一个客户想拥有一个使用 Java Servlet 技术的 XML 界面, 而不是通过 JSP 页访问服务。添加这个请求的难易程度, 依赖于通用服务 API 的设计如何通用化。如果通用服务 API 特别为 JSP 页和 HTML 页流设计, 就很难使用通用服务 API 作为 XML Servlet 界面了。

这个例子的教训是如果不考虑潜在的变化和系统的使用, 将来就要面对一些有风险的变化。当系统不能轻易地扩展时, 对系统的修改的数量和难度可能很大, 导致增加了出现软件缺陷的可能性。

特征蠕变

提起特征蠕变的时候, 我们会联想起一个泥泞的生意和销售人员。即使这个景象与现实的差距不远, 我们也不将蠕变作为人来考虑。实际上, 在这里特征蠕变指的是在软件开发过程中的设计、实现和测试阶段对软件系统添加新特征的过程。

特征蠕变背后的主要原因实际上在于销售和市场部门。通过对市场的研究、用户访谈以及一些领域知识, 他们搜集了所有的该公司应该有的特征, 以增强其在这个新的工业中的竞争能力。其他时候, 特征蠕变可能源于高级的管理部门。由于许多原因, 一些人认为产品必须支持 X、Y 和 Z; 他们成为公司里的高级决策人。特征蠕变还可以因用户而产生。初始的需求描述可能不准确, 所以漏掉某些期望的特征。后来这些特征又被加上, 但就开发人员而言, 这些都是未期望的特征。特征蠕变同样也可以由开发人员产生, 一般是因为他们认为产品应该完成一些需求中没有要求的任务。由开发人员导致的特征蠕变的主要内在原因是一些

好的愿望，如可重用性、可扩展性、完备性以及其它技术完美性概念。在实现过程中，开发人员会有一些醒悟，认识到如果他们不这样做，许多东西就会变得更好、更容易。所以系统的设计会在中途变化。但是“好的企图”可能会导致灾难。通过改变软件设计，我们增加了带来软件 bug 的机会。修改后的影响通常没有被充分分析，因为已经做过分析了（毕竟，我们是处在项目的实现阶段，是不是？）

4.2.2 拙劣的描述

多数 bug 的主要原因是描述太差。有许多关于需求与 bug 之间的关系的研究。这些研究表明，在一些例子中软件系统 50% 以上的 bug 可以归结于需求分析阶段的错误。不正确的需求描述甚至是遗漏的描述，这两者是最一般的需求分析的错误，导致不一致和模糊。

一个描述应该是完备的、正确的和清晰的。这是一个很值得努力的目标，但难于实现。描述的目的是给开发人员和测试人员一个对需要实现什么东西的完全的理解。开发人员需要知道编写什么，测试人员需要知道测试什么，就如同第 2 章中提到的那样，Ariane 与火箭的毁灭可以部分地归罪于描述的不准确。

很明显，一个不正确的描述会导致不正确的实现，从而导致 bug。

模糊的描述中会包含互相冲突的需求，导致系统有错误的行为。模糊的描述也可能有不够详细的要求。模糊而不够详细的需求是主观的，需求分析中的短语如：“系统必须是对用户友好的”，并没有给设计人员提供系统目标的客观描述以使用。在这样的例子中，开发人员不得不做出假设，使用自己对需求的主观理解来生产软件。但最可能的是，一个描述的模糊是因为它是使用自然语言写的，比如英语。我们都使用相同的单词来表示不同的东西，而又用不同的词表示相同的东西。我们往往编写出复杂的句子，目的是使文档更加“可读”，但却很难表达准确的含义。

但是，在对描述进行浏览后发现的主要错误是漏掉了需求，就是说描述是不完整的，不完整的描述是指描述可以导致某些特征被忘记。面对不完整的描述，开发人员不得不猜测系统的正常行为。这个过程很可能最后在软件中做不到被期望去做的事情。

最关键的描述需求的一个条目是错误处理。一般地，需求分析在功能层描述对错误的处理。但是，编程阶段的错误处理的描述会变得更加困难，而且通常不在需求描述中定义。问题和不确定性就出来了，如同下面例子中一样：

- ◆ 如何处理存储器分配错误？
- ◆ 如果用尽了磁盘空间怎么办？
- ◆ 如果流在中间被截断会怎样？
- ◆ 是自己处理这个错误呢？还是捕捉到它并向调用的程序报告呢？
- ◆ 要捕捉哪些异常，而放走哪些？
- ◆ 是应该从错误中返回，还是报告它？

我们的经验是不正确的。对错误的处理是一种巨大的潜在错误，一般在产品阶段的几周

或几个月后表现出来，因为它们很少被测试，而且难于触发。许多时候开发人员只是加进一些东西，因为他们不知道如何用一种现实的方式处理错误，或者根本不知道会发生错误。很不幸，软件工程也无法脱离 Murphy 法则。

4.2.3 方法的复杂性

我曾经与一些人一起工作，他们认为程序的性能就是一切。现在流行用 C 语言编程，他们使用一些让人提心吊胆的指针操作来让程序加速几个毫秒。他们在一个句子中进行多个操作以节省几个毫秒。他们为能够在办公室中得到最简练的和最快的程序而自豪。很不幸，他们的程序也是最难于维护的，看起来都头痛，无法调试，易于出错。我就如同躲避瘟疫一样躲避他们的程序。考虑下面的例子：

例 1：

```
1. #define SIZE 10
2. int count;
3. int sum = 0;
4. int array[SIZE];
5. ... // array was filled
6. for ( count = 0; count < SIZE; count++ )
7.     sum += array[count];
```

例 2：

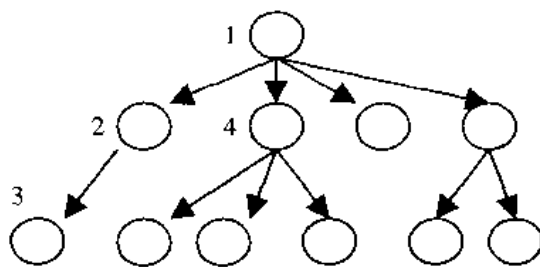
```
1. #define SIZE 10
2. int sum = 0;
3. int array[SIZE];
4. int *ptr0 = &array[0];
5. int *ptr1 = &array[SIZE-1];
6. ... // array was filled.
7. while ( ptr1 != ptr0 )
8.     sum += *ptr0++;
```

这两个例子虽然不是非常复杂，但也能说明问题了。两个例子都是完成同样一个功能——把一个数组里的 10 个数加起来，在我们的观点里，for 循环结构更容易理解，更少出错；而 while 循环结构则比较复杂，而且更有可能出现 bug。在例 1 中，显然循环会反复 10 次执行直到满足第 6 行的条件。在例 2 中，while 循环只告诉我们，循环会一直执行到 ptr1 的值等于 ptr0 的值。我们需看第 4 行、第 5 行和第 8 行，才能理解 ptr1 在什么情况下才能等于 ptr0 中的值。循环的终止依赖于初始化和指针递增的正确性。而那些对 C 语言中的指针操作不太熟悉的人，可能很难理解句子的正确性。例 2 比例 1 有更多的出错的机会。

差劲的设计也可以使一个系统难以理解、实现和维护。我记得曾经在某个人做的中间件系统中寻找 bug。设计者是个狂热的设计模式迷，而且这些模式在整个系统里都得到了使用。有命令模式、观察模式、正面模式、制作模式和单元模式。如果这些模式使得恰当的话，这并不是一个大问题。但系统有 7 个单元模式和 2 个命令模式。这个单元模式遍布于整个系统并且在不同的地方初始化、改变和使用，很难由代码追踪理解这些单元规则的使用和状态。

基本上，设计者将单元模式作为全局变量的外观来使用而不实现它。命令模式使得数据流变得复杂，而且很难看出数据如何从一个类流入另一个类。所有这些模式出现在大约 20 个文件里。谈论使用模式语言的系统是一件大事，你可以把每个模式看作黑箱来谈论。但是，在实现和调试中，我们不能假定这些模式都正确地实现了并且没有什么 bug。当你不得不走进代码中去理解它时，模式语言的优点就没有了。这并不是说明设计模式不好，而且应该避开，这并不是我们的意思；我们也不希望你丢开设计模式。设计模式只是一个工具，像其他工具一样，它也可能被滥用。在滥用的时候，它会使得一个方法变得更复杂，这就意味着可能带来更多的 bug 和大量的调试工作。

回调函数很流行，因为它们很有用，能够帮助我们掌握程序中事件传播的概念。当一个软件组件对另一个组件中的事件感兴趣时，它会将这个事件登记在源代码中。当事件发生在事件源中时，事件源通知所有对此感兴趣的组件。一旦得到通知，感兴趣的组件就会去执行那些依赖于这个事件的任务。从中我们可以得出一个因潜在的复杂性而导致缺陷的抽象。图 4.3 显示了一个例子。在这个图中，每个圆圈代表一个软件组件。顶部的圆圈代表事件源。从事件源中（组件 1），伸出了一组箭头指向那些对这个特定事件感兴趣的软件组件。在一个事件发生后，程序的控制将转向它下面的一个组件（例如组件 2）。如果组件 2 中有一个组件 3 所等待的事件，那么组件 2 就把控制交给组件 3。这个过程一直持续到所有的组件都有机会处理这个事件。最后控制转向组件 1。



这些是流行的设计方法和它们产生潜在的 bug 的可能性的例子。在本书的研究中，我们注意到不同的设计方法有不同类型的 bug。回调函数所导致的 bug 的类型，不会出现在不使用它的程序里。在第 12 章，我们会详细讨论不同的 bug 和相关的 bug 类型。

在经验中，我们可以观察到人们喜欢把事情复杂化。人们一般不用一个单独的类做一件事，而是或者用一个类做多个事情，或者用许多类做一个事情。有时候是以灵活性的名义，其他时候则是以重用的名义。两者都是不错的理由，但是如果做得不正确，就会产生复杂性。我们前面讨论了复杂性是产生 bug 的主要原因。“让它简单些，笨蛋”这是总统竞选中的忠告，这在软件工程中同样适用。

4.2.4 缺少一致的观点

软件在开始设计时，开发人员有机会到一起尽可能地谈论问题和方法。这个过程导致一种思维混合，保证了组中的成员达成一致。单个成员可能会反对或不同意某个被采纳的解释和设计，但他们理解这些设计决策的基本原理，并且保留不同意见去工作。在这个思维混合过程之后，工作组将分为更小的组来工作于问题的各个部分，集中的观点将从他们的脑海中消退并被他们自己的观点所取代。任何后来的变化，都不会共享相同的设计观点，一致的设计整体性被许多有竞争力的设计元素所取代。这个问题在项目的维护阶段变得更为严重。通常，被叫去对系统修改和维护的人与初始的设计选择和折衷并不一致，而且修改常常不考虑系统的初始观点。

缺少共享的观点将如何影响 bug 的产生呢？还没有人直接对这个事情进行研究，但是有一些具体环境中的证据表示这个关系。Joseph Dvorak 提出“概念熵 (conceptual entropy)”这个词来描述开发者在类层次上相互间的概念上的不一致性。在他的研究中 Dvorak 证明：当程序员们进入到类层次的深度时，对子类的选择几乎就没什么一致性了。开发人员根据他们的个人观点来区分概念和类，而不是对类的表示坚持一个统一的观点。

Daly 等学者提出了一个经验研究的结果，证明了进行维护工作时概念层次的影响。他们的研究表明：“可以证明，有三个继承深度水平层次的面向对象软件，能够比基于对象的没有继承的软件，用更少的时间完成维护任务。但相比较而言，那些有五个继承深度水平层次的面向对象软件，却没有这个特点。课题的完成时间、源代码中的方法和听取询问，实际上就表明课题因为这个更深的层次而遇到了困难。”

而且，在另一项研究中表明“程序员容易对对象、层次和重用关系产生误解”，而且会在对软件维护工作的研究中出现差错。

概念共享和程序理解并不只是面向对象结构中的问题。概念熵的定义不能被用于软件整体的设计和实现。从原理和形式上共享同样的思想集合，确实能提高生产能力和减少 bug。我有过与一些和我差不多的人共同工作的美好经历。我无须猜测他们的代码，也无须寻找函数名和函数标记；我心里明白他们的代码是什么样的，而不用与他们交谈，因为我们共享了关于如何解决问题和如何编写代码的概念。一次，当我和朋友忙一个项目的时候，他为一个

遗传程序算法编写 API，而我将使用它。我对他的接口编码时他就坐在我旁边，除了对他正在为算法编写 API 之外，再没有关于设计和实现的讨论。无须看一行他的代码，我就可以用与他的 API 相一致的代码写出一半的程序。我居然能够预料他的 API，这让我们都很吃惊。当有这种思维混合时，就很少有人会因为某个人误解了软件的目标和用途而犯错误了。

一个共同的观点有各种形态，它可以如同一个代码形式那么简单，或者如同软件开发过程和方法的选择那样复杂。它可以如同对组件的界面达成的一致那样实在，也可以如同对设计构造的原理的一致和对优先的折衷选择的理解那样抽象。一般地，软件开发组中的一致观点，在缺少组中成员间的交流和对新成员的培训时就会垮掉。

因为我们没有人会有超感官的感觉，我们无法相互之间用心灵感应来交流想法和思想。所以我们必须依靠口头的和文字的交流，来增强观点和目标的一致性。当交流中止时，成员之间关于某些事情如何做的不一致就出现了，而且永远无法解决。当有不一致的时候，人们习惯于按照自己的观点来解决问题。当不同的人工作于同一个项目解决一些稍微不同的问题，与整体性相关的 bug 出现的机会就会增加。火星气象卫星的损失是一个活生生的例子。

当新的成员加入到组中时，他们会带来新的特征。这些特征包括他们的习惯、经历、知识和模式。我们想平衡他们的经验和知识，但不想把这些加进工作组中。如果没有一个训练过程的话，就只能是一个缓慢的偶然过程。所以，一个新的成员用这种方式工作，另一个开发者用另一种方式工作，所以很快地就有多种方式和模式。理解和学习一个人的方式已经不容易了，何况 5 到 10 个不同的人。Java 软件开发包 (Software Development Kit, SDK) 的 API 有一个很详细的设计和接口原则。对于任务如何做和如何用 Java API 做得一致，它考虑得很多。设想如果不是这样，Java 就不会像今天这么流行了，因为学起来代价太高。当一个项目有许多模式时，程序理解所需的耗费就会随着缺陷的数目增加。

4.2.5 程序员错误

我曾经在 C 程序设计语言新闻组中的一个贴子上发现过下面的 bug：

```
1. while( I = 0; I < 10; ++I ) {  
2.     printf("%d\n", I );  
3. }
```

这个贴子抱怨说这段程序不能编译。它得到大约 5 个回复，只有一篇指出了这段代码中的根本错误。显然，这应该是一个 for 循环结构而不是 while 循环结构，有些读者可能会立即指出错误，而有些读者则要花点时间。这是一个小的编程错误，却可以使一个新手花一段时间才能发现。我试了两次，因为我只是大致看了一下认为结构是对的。我并没有真正阅读代码。为什么会这样？谁知道？这个程序员可能正在想是用 for 循环还是用 while 循环，并在这两者之间犹豫。

在这个例子中，问题是关于句法的，所以编译器会阻止 bug 流入现实世界。但是，相似的错误可以发生在语义水平上，这将会使得 bug 难于发现和识别。下列潜在的 bug 可以逃过

直接的检查:

```
1. std::vector<int>::iterator iter = alist.begin();
2. while ( iter != alist.end() ) {
3.     cout << (*iter) << endl;
4.     if ( (*iter) == 5 )
5.         break;
6. }
```

这是另一个简单错误的例子。发现这个 bug 的时机，依赖于我们触发适当的情况的时机 (iter 需要在第 6 行之前增加)。如果 alist 的值通常开始于 5，那么在 bug 发生前应该有一个 while。如果 if 语句变成如下的话，问题就严重了。

```
4a.     if ( (*iter) = 5 )
5a.         break;
```

我们可能认为已经修复了初始的 bug，只是遗漏了第 4a 行的第二个 bug (“= ”应该是 “=”)。你可能认为很容易发现这类 bug，在你读书时候，思维严谨，它可能显得简单。但当你一直编写代码并盯着它看了几个小时以后，就很容易遗漏这样的错误了。被放错地方的 “break” 语句曾使 AT&T 在 1990 年发生中断，而这样小的编程错误却从未被注意过，因为它看起来没什么危险，软件工程师也看不到什么别的。

人的错误通常是因为不注意造成的。1962 年 7 月 22 日，到金星的 Mariner 探测器失踪了，因为 “DO3I=1,3” 被错写成 “DO3I=1.3”，这是 FORTRAN 程序中的错误。结果，FORTRAN 编译器没有执行 DO 命令，而将语句解释为 “将 1.3 赋给变元 DO3I”。

注意：无论如何，这个故事是有些真实性的，官方的 NASA 网站描述了 “在编码计算机结构中连接号的遗漏” 是主要的原因。要得到关于计算机轶事的其他信息，请参阅《Risk Digest》关于这个主题的文章，第五卷，第 64、65、66 和 73 期。

但是还有能够影响和导致愚蠢的编程错误的其他因素。

缺少对工具的理解

在第 2 章，我们描述了 AT&T 在 1990 年长距离的电话服务中断。其原因是 “break” 语句的放置不正确。我们不知道程序员的确犯了一个错误呢，还是不理解语言的结构。但是，如果这个程序老是犯这样的错误也是不希奇的。我曾与一些自称是某些语言方面专家的人合作过，发现他们对该语言使用的缺乏理解是惊人的。例如，我曾与一个 C 程序员合作，他喜欢将一个函数参数反引用两次而没有什么明显的理由。对于下面代码：

```
1. Function_Foo( int *a ) {
2.     --
3.     *a = 10;
4.     --
5. }
```

他常常写成：

```
1. Function_Foo( int **a ) {
2.     --
3.     *(*a) = 10;
```

```
4.    _  
5.    )
```

这两段代码完成同样的事情，参数被赋值为 10。但是，在第二个例子中，参数的反引用多了一次，可以成为潜在的混淆或错误的原因。对于多数 C 程序员来说，指针到指针再到变元，和指针到变元，有不同的语义。我可以想象什么情况下你可能多次引用一个参数，我也与他多次争论这个事情。但他从没有说清楚理由，最后我们的结论是“格式上的区别”。在我知道他并不精通指针之后就不觉得惊奇了，他的试验和错误经历证明当他这样做时是有效的。所以他常常将一个值反引用两次而并不理解内在的原因。

C++ 并没有使编码容易些。实际上，它可能使得事情更加困难了。它不只是仍然存在内存管理和指针问题，而且面向对象技术的引入也增加了语言的复杂性。传递对象时有通过值、通过引用和通过指针几种容易混淆的形式。继承和多态引入了一堆关于复杂性的问题。当我刚开始作 C++ 程序员时，我经常在重载构造和析构时犯错误，因为我不理解 C++ 所有的规则和细节。我可以通过任何面试，而且可以背诵程序员手册上的所有规则。但是我还没有理解语言的原理，而且我并没有按原意去使用 C++。我只是根据自己的感觉使用 C++，而那是错误的。

我听说过许多关于由于初学者的错误导致 bug 的故事。他们将数据返回到栈并且使用，好像数据已经分配了。他们嵌套了 11 层循环因为他们不理解在一个 if 语句中使用多种情况表达式的概念。他们忘记在使用 C 中的字符串函数时应该事先分配一个空字符空间。缓冲区溢出也是新手常见的错误。所有这些事情新手都很少调试，因为他们不知道自己在制造 bug。他们只知道程序不起作用，而且症状往往在几百行代码之外出现。

使用不恰当的工具

使用工具时一个你所不理解的边缘效应是你可能遇到使用错误工具的危险。我们仍然会使用错误的工具，因为我们想使用，或者有别的原因。可能是因为这个工具是最新潮的；可能是因为我们想更了解它并使自己更加市场化；可能是因为我们已经学会怎么使用了，而不想花时间学其他的了；也可能因为我们不了解这个工具是干什么用的。这些工具是什么？它们可能是调试器、编程语言、方法学、现成的商业组件和应用，或者类库和组件。

在第 2 章中，我们给出了两个事件，其 bug 可以部分地归罪于对工具的不正确使用——Ariane 5 和 Therac 25。在两个例子中，导致问题的软件组件被不正确地重用了。在 Ariane 5 的例子中，对组件的描述重用自 Ariane 4 火箭，而其有与 Ariane 5 不同的性能结构和描述。

但是区别没有被考虑，在 Therac 25 的例子中，重用的软件组件是有 bug 的；导致过量辐射的 bug 同样存在于 Therac 20，但是 Therac 20 有一个硬件保护避免 bug 导致事故。

考虑另一个例子。在互联网软件开发环境中使用瀑布方法，可能会带来问题。瀑布方法要求，在开发过程中一个阶段的描述，在下一阶段开始之前就应该清楚和明确。在互联网时代，一个更加反应性的方法，如一个螺旋型的或进化式方法，应该更加恰当。

面向对象的设计已经成为软件结构事实上的标准。C++ 和 Java 的流行性，使得面向对

象成为设计模式的首选。但是能够在每个项目中都用面向对象技术吗？结构化程序设计有什么毛病吗？自上向下设立不对吗？继承和多态虽然功能强大，但使得软件更难以理解。生成和设计一个正确的类层次和关系是比较难的，而且能使软件没有必要地复杂化。但是一个设计良好的对象，可能有利于使用和理解。自上向下的设计可能不太吸引人，但用起来可能相当容易。

注意：工具的选择应该根据具体的问题而定。

懒惰

懒惰可能是一个好的软件工程师的好的行为。好的软件工程师懒于重新发明一些东西。他们想重新使用软件组件和模块。他们寻找最简单的方法，并避开不必要的复杂性，因为他们不想再经历智力上的训练。但是懒惰可能也是一个不称职的软件工程师的标志，不称职的软件工程师剪切和粘贴代码时不考虑所有被重用的代码的正确性和描述。他们不去检查错误情况，而且使得方法比所需要的更加复杂，因为他们懒得去找一个简单途径。

一个粗心的软件工程师的懒惰，也表现在懒惰的代码中。什么是懒惰的代码？你可能已经见识过或者写过。我知道我也曾经写过。懒惰的代码能误导其他的开发人员，它们可以有多种形式。懒惰的代码可以包含一个定义的却未使用的函数。它可以是一个从未使用过的变量。它可以是一个错误的或过期的注释。它可以是一个错误的变量名因为你在改变它的用途时没有改变名字。它可以是一个短的但没什么含义的变量名。不管它是什么，它使代码混乱并降低了信噪比。其他人理解起来越来越困难，产生 bug 的机会也增加了。

4.3 bug 是如何躲过测试的

这个世界上有两种 bug：内部 bug 和公开 bug。内部 bug 是那些我们在开发过程中发现的 bug。这些 bug 是通过浏览、开发和测试发现的。一般地，只有项目内部的人才知道内部 bug 的存在。发现内部 bug 是好事。它表示我们在自己家门口发现了错误，我们的愚蠢不会被别人嘲笑。另一方面，公开 bug 是坏事。公开 bug 发生在产品环境中，并且一般被用户报告，实在令人尴尬。

我们相信 bug 的产生是软件开发过程不可避免的结果。我们的软件应尽可能少地产生公开 bug，而不是产生没有 bug 的软件。达到这个方法是在开发过程中识别尽可能多的 bug。

bug 在软件开发过程的不同阶段产生，如在需求分析、系统设计和实现阶段。我们在开发过程中有许多机会去识别和纠正错误。但许多的 bug 都逃过了正常的检测。这时我们通常归因于不正确的或不完全的测试。在本书中，我们不把测试作为软件开发过程中的独立部分。我们相信测试是由项目管理者、实现者和测试人员在整个开发过程中完成的。在软件工业，质量保证（Quality Assurance, QA）是用来描述在软件开发生命周期中排除 bug 的过程。当一个 bug 躲过测试时，它就躲过了 QA 过程，而不只是测试者。很明显地，当代码没有被

测试时，bug 通常躲过了。但测试过程和环境可能包含一些导致 bug 能躲过去的缺陷。

4.3.1 遵循形式过程代价太大

检查、审查和测试是 QA 过程的组成部分，但进行这些过程的代价比较大。为了正确检查和审查软件文档和产品，就要使用合格的检查者。合格的检查者应该是高水平的分析员和工程师，所以他们的时间很少，很宝贵。文档和代码应该被收集并提前交给检查者。这意味着开发人员和检查者要有更多的时间和精力。检查要花费一定的时间，要不断地开会交流检查和审查的结果。附加的工程时间将转化为附加的费用。

4.3.2 政策/市场决策

bug 的发布可能是由于一些明显的和技术人员无关的原因。一个组织可能会严格地按照过程做。但有时，公司会觉得对附加 bug 进行维修的时间耗费将会太延迟软件的发布。软件足够好，而且由于一些预知的市场优势，软件就要按计划发表了。所以软件发行时就带有已知的 bug。

在安全攸关的工业中，如医疗和航空系统，发布带有已知 bug 的软件并不是好的想法，但对于其他的工业，它可能是可行的。发布带有已知 bug 的软件，要考虑 bug 的严重性、环境和对于用户来说的可能结果。在第 2 章，我们给出了 Intel Pentium 的 FDIV bug。拿它作为例子，即使 Intel 在软件发布时不知道这个 bug，它后来也的确发现了这个 bug 并且保留了信息，因为它认为 bug 的危险性很小。这个决策对 Intel 的代价比较大，但对这个 bug 不加理睬的决策曾经被认真衡量和考虑过。

4.3.3 时间不充分

测试要花时间，QA 过程要花时间，而且当日程表已经过了之后，分配给 QA 的时间通常就被压缩了。代码审查过程被取消了。只进行了更少的测试实例。测试在代码修改的同时进行，所以测试很可能被用于不正确的版本上。当时间不够时，很难做对事情，很难优先测试那些对用户比较关键的功能和代码路径。

乐观而不现实的时间表是主要的出错原因。软件开发人员会高估自己的能力并低估问题的难度。时间表根据市场角度来决定，而不是根据技术可能性来决定。开发人员可能会遇到阻碍并走许多弯路。开发人员计划重用的软件组件可能会出现 bug。必要的软件可能来得比较晚。人们可能重病或有急事。你的宠物狗可能咬了你的笔记本，而里面可能有关键的软件，硬盘可能崩溃了。一直要记住任何可能出现错误的地方，软件在时间不充裕时就会出错。

4.3.4 缺少重现能力

一个测试人员的噩梦通常包括不可重现的或半可重现的 bug，当一个 bug 很不容易重现时，它通常难以发现。测试人员写出关于这个 bug 的报告并交给开发人员。开发人员无法重

现这个 bug 并把报告还给测试人员。测试人员也不能重现 bug，这样这件事就停止了。……一直到大约 1 星期之后，当测试人员要日夜工作，最终得到可以重现的情况。他们这是在调试软件。

bug 的可重现性，与导致这个 bug 的原因有着内在的联系。那些不能持续重现的 bug 可以让一个开发人员心痛。这意味着你对系统的了解并不够，系统是有自己的特点的。说明你并没有能够预见到代码中断的环境和可能性。然后你开始担心是不是有了一个无法发现的 bug，实际上可能有无数的根本不知道其存在的 bug。

bug 的重现困难性有很多理由。有可能是测试人员没有正当地或正确地跟踪和记录，可能是软件应用的物理限制。测试航空软件是很难的，因为要模拟出许多的硬件；Ariane 5 的事故就是一个例子。也可以是数据和事件驱动。AT&T 在 1990 年的中断和 Pentium FDIV 的 bug 都曾逃过了测试，因为触发事件是数据驱动的，而且比较少见。系统可以对环境和配置区别敏感。软件在不同机器上的运行可能不同。bug 可以是顺序相关的组件，而且 bug 的触发可以依赖于按照正确的同步执行正确的顺序。

4.3.5 自负

有一天我看了《Titanic》，这是一艘号称永不沉没的船最后沉没的故事；显然 Titanic 的沉没是因为它撞了冰山。但是人类的骄傲和自负使得这个事件成为可能，而且使得灾难加剧。由于对技术和人类能力的信任，对可能出现冰山的警告被忽略了。因为人的自负，船上没有放足够的救生艇以疏散 2200 个人。在 Titanic 不会沉没的时候，为什么还要浪费钱搞救生艇呢？Titanic 是那个世纪的最严重的海难。然而，我们一样可以在软件系统中造成 Titanic 那样的灾难。

开发人员是非常不称职的测试者。你一定听过多少次某个程序员说“它在我的机器上正常啊”。当然，总是听说，但客户却不是使用你的机器，所以要把自负放在一边，并看看他们到底在抱怨什么，不只开发人员有自负的现象，项目管理者可能对他的员工和产品自信，并且不会注意去测试。公司可能过于自豪，不相信会出现有 bug 的产品。当你不相信你的软件里有 bug 时，就不会去查找它们。如果不去找，你可能就不会发现它们，Therac 25 的例子就给出了相关的例子。

满足是自负的一个结果，而且被认为是第 2 章中所说的火星气象卫星和 Ariane 5 中的 bug 的原因。当我们觉得满足时，我们并不质疑所有假设，我们做了错误的假设。我们假设所有的功能和特征都正确。我们没有详细地测试。我们关心每个细小的不一致性，这样缺陷就从这些缝隙里溜走了。

4.3.6 差劲的描述/不知道要测试什么

就像开发人员需要知道写什么一样，测试人员需要知道要测试什么。一个好的描述可以做到这两点。测试计划可以直接受影响于一个好的描述，一个测试人员可以从描述中挖掘出

需求并测试它们。一个好的描述也可以告知测试人员模块的算法，这样测试人员就可以设计测试情况和实例。

在开发 Pentium 芯片时，Intel 依靠随机测试来测试 Pentium 芯片中的 FDIV 算法。但随机测试不能按正确的方式到达所有可能的代码路径，所有得到的结果对 Pentium 芯片中应用的 FDIV 算法具有代表性，如果所采用的测试能够真正描述 FDIV 算法，那么 Intel 在测试日期发现 FDIV bug 的可能性将会增加。

4.3.7 缺乏测试环境

一个好的测试环境包括三个特征：有良好工具的测试人员，在一台与客户系统有相同的系统配置的计算机上的软件，用户的使用被理解并且被良好地建模。当缺少其中一个特征时，bug 就有机会逃过测试过程。

- ◆ 缺少正确的工具——有时候，测试人员只是没有能够正确地进行测试的工具。当测试不当时，我们可以得出关于软件可靠性的错误结论。例如，在一个大规模的产品网站中，系统在正常负载下的性能是很重要的，而且应该首先得到考虑。但是，如果测试人员没有合适的测试工具来进行负载测试，或者缺少硬件资源，那么就很难恰当地模拟负载情况。

- ◆ 缺少恰当的系统配置——创造软件所使用时的环境也很难。在单个应用中，测试环境包括计算机和软件配置。为应用建立一个测试环境看起来没什么，但当你认识到客户可能使用的硬件和软件配置时就明白了。在硬件和嵌入式系统中，很难正确地模拟硬件接口。

- ◆ 缺少使用模型——建模和测试系统的实际使用的能力，能够减少公开 bug 的数量，因为可以找出大多数潜在的 bug。一个软件系统可能有很多的缺陷，但是如果用户从来不使用有 bug 的代码路径，这些缺陷将永远不会成为公开的 bug。

4.4 小结

bug 存在的原因可以被归结为一个词汇——人。人是创造者和破坏者。人犯错，但也发现和修改错误。如果我们想使软件更加稳定，我们必须从人开始着手。我们必须理解为什么我们会犯错，以及如何犯错误。只有这样我们才能设计防御性行为，并创造能够提供帮助的工具。我们也应知道如何更容易地发现和纠正错误。在本章，我们区分了一些导致 bug 的因素，它们伴随着复杂性的根本原因、现实和人类的弱点。然后我们展示了这些原因如何相互作用从而产生出容易出现 bug 的环境。最后，我们讨论了我们是如何让 bug 溜进现实世界的。我们将在第 12 章重新看一下这些理解和发现，以讨论有效的方法。

4.5 bug 问题

为什么会出现 bug？有时根本没什么好的理由，试试看下面这个现实生活中的 bug 问题。

你在一个网络公司中面临着被炒鱿鱼的问题，因为你正处理公司的一个新站点的性能，它向消费者提供一定的服务。服务本身并不重要，但 CEO 对系统缓慢的响应时间却很恼火，你要么修正这个问题，要么走人。

你对产品站点进行了一系列的实验，而且看到了站点性能的巨大变化，即使是在加载相同的页面时。所以你建立了一个脚本以在每天每分钟内加载一次欢迎页面。结果显示欢迎页面的加载时间为 4.54 秒到 34.31 秒，平均时间是 16.42 秒，标准偏差是 9.43 秒。你很困惑，不明白为什么网页的加载时间会有如此大的波动。然后你在开发人员的机器上做了相同的实验。但是，开发人员的机器上的代码和配置有一点区别（因为它们毕竟是开发人员的）。在开发人员的机器上你发现加载时间为 3.23 秒到 4.76 秒，平均 3.56 秒，标准偏差是 0.21 秒。

欢迎网页是用 Java Servlet Pages (JSP) 编写的，与后台服务通信建立显示在客户浏览器上的最终的 HTML 页。你检查了后台服务器在测试中的性能，日志显示产品服务器和开发服务器都在 3 秒内持续响应 JSP 请求。问题是什么？提示：图 4.4 表示了系统的概念体系结构/网络布局。

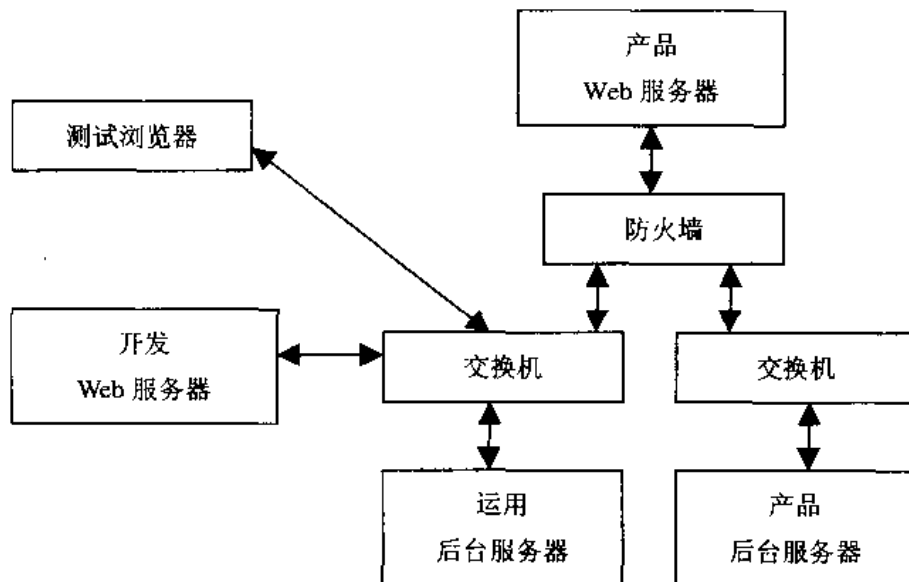


图 4.4 bug 问题中所描述的系统概念的、体系结构的和网络的布局

第 5 章 bug 的分类

当科学家谈论现实的 bug，就是那些晚上爬在房子里的臭虫时，他们要将其进行分类 (taxonomy)。辞典中 taxonomy 的定义如下：

taxonomy：自然科学家对动植物的类别进行的区分：分级原理的规则。

当我们处理技术上的 bug 时，这又表示什么呢？这时的 bug 毕竟不是动植物。由于 bug 不属于这两类，我们可参考第二个定义——分级原理的规则。

为什么考虑对 bug 的分类很重要？最重要的原因是各个组的 bug 有重要的相似性。在理解这些相似性之后，就可以通过检查其症状，很快地断定所遇到 bug 的类型，然后到分类中查找它们。

调试中最关键的（并且是最容易忽视的）一个方面是：理解 bug 为什么会发生，以及来自何处。如果能够判断一个系统中有什么类型的 bug，而且能够将它们与开发过程中一定的行为联系起来，那么我们就能够改进清除 bug 的过程了。为了做到这一点，我们将这些 bug 区分为一般的种类，看这些种类之间的关系是否有助于更好地理解在过程中 bug 产生的根源。所以通过建立一个 bug 分类，我们希望能创建一般的分类列表，以能够观察到 bug 的产生源。

目前，有两种通用的分类标准。第一个由 IEEE 发表，题目是“ANISI/IEEEP1044-1994：软件异常的标准分类”。这个分类主要考虑产生 bug 的原因，并准确描述项目的软件开发过程中发生 bug 的地方。虽然这样开始寻找问题比较好，但对于做调试工作的软件工程师，或者那些想把某一 bug 归到一个类别中的测试人员来说，没什么用。

第二个可以使用的一般分类是在 Boris Bezier 的《软件测试技术》第二版中发布的。幸运地，Bezier 将他的分类公开了。这些可以在他的书中或者网上查到。

在分类中，我们要考察一些不同的资源和原因。我们不仅要考虑那些在代码中发现的 bug，还要考虑那些在软件开发过程其他部分中发现的 bug。在开发过程的每个阶段，从项目的定义到最后的配置，bug 都可以潜伏在系统中。发现和修复这些 bug 是软件工程师在多数例子中的职责，对于另外一些例子中的操作人员和项目系统人员也是如此。通过理解 bug 从何处进入系统以及在何处修复，我们可以恰当地将 bug 分配给合适的人而不至于浪费公司里别人的时间。

区分 bug 的种类时，需要哪些信息？首先必须知道 bug 发生在开发过程中的哪个阶段。其次，我们要知道是谁造成了 bug（或没有捕捉 bug）。最后，我们要考虑最后谁负责修复系统中的 bug。作为一个调试员，你可能会认为产品中最终出现的 bug 都是自己的问题，但并不尽然。例如，某个问题只有重新设计才能真正“解决”，那么方法就在于所使用的体系结

构层。如果问题是因为对部分用户缺少了解，那么方法就在于编写帮助的人员或系统的培训人员。不要老是认为关于来自 bug 数据库中的 bug 报告是软件工程师的问题。本章不考虑的问题是过程的最后一步，即发现和修正 bug。在第 8 章讨论了发现 bug 的方法学，第 9 章讨论了几几乎所有种类的 bug 发现和识别技术。对任何超出分类问题的问题请参阅这两章。

bug 趋向于出现在开发过程的特定区域。让我们看看是哪些区域，以及其中的 bug 到底是什么。

5.1 bug 的种类

每种 bug 都包含开发过程的一个阶段和一组相关的问题。不同的阶段有不同类型的解决方法 and 不同层次的风险。从前面的研究中我们知道，在那些产生它们的层次之上出现的 bug，比起那些在同一层次上产生、捕捉和消灭的 bug，修复起来要难得多。调试一般不单指修复代码。它可以包括许多其他层次的工作。在下面即将看到。

5.1.1 需求阶段的 bug

项目中的软件开发过程中的需求阶段，被认为是对系统的全面成功最重要的。如果你想了解这方面的内容，可以看一下软件工程的书。本书的参考书目中就有许多。但是，当我们从以 bug 为中心的角度来看需求阶段时，就要变换视角了。需求阶段的 bug 是目前最难发现最难修复的，后面将会看到。这就是为什么需求阶段的 bug 比编码阶段的 bug 的修复费用要贵 15~75 倍的原因。

需求阶段的 bug 有几个出现的原因。第一，一个需求可能在系统的描述中忽略。这导致用户不满，最终用户可能认为软件有 bug。需求忽略是目前需求阶段 bug 中的主要问题。

第二，需求阶段的 bug 类型可能是在描述中的相互冲突的需求。比如有两个必须被满足的需求，他们有相反的含义。我们将相互冲突的需求问题作为一个特定的 bug 类型。

第三，有些需求很模糊。有些需求虽然与其他需求没有冲突，它不够清晰，不能导出一个设计决策。当设计选择是基于一个不完整的或模糊的需求上时，最终用户常称其为一个 bug。

大量的研究表明，需求阶段的 bug 修复代价最高，跟踪难度最大。理由很简单。如果一个 bug 存在于需求阶段，那么就没有“问题代码”可以检测。没有测试实例可以说明问题，因为测试实例是按要求描述的。只有最终用户才能指出需求问题，然后不直接地指向系统中的一个 bug。

5.1.2 设计阶段的 bug

设计是指将模糊而广义的需求描述语句，转换成关于系统内部如何工作的更加精确的技术性描述。和任何翻译系统一样，这个过程中肯定会出现问题。设计上的 bug 比需求中的 bug

更易于捕捉，但最后的修复代价很高。设计上的 bug 的修复代价高的主要原因，是它们趋于作为一个整体来影响系统。显然，修复整个子系统或应用系统要更难一些，比修复几行错误的代码或替换一个模块更加容易带来错误。作个比较，如果需求代表整个汽车，设计则代表汽车的发动机。即使整个汽车耗资 20,000 美元，更换发动机也需要 12,000 美元。显然，没有令人满意的情况。你会奇怪为什么没有对应用中的需求和设计的保证呢？

设计上的 bug 通常归于三类中的一种：忽略、冲突和翻译错误。忽略意味着没能将一个或更多的需求加入应用的最终设计中。用另一个比喻，这就好比与一个未来的房主谈一所房子，而忘记在最终的蓝图加入正确数量的卫生间。很不幸，就像房子的错误那样，捕捉到被忽略的部分是很难的，即使是你与用户一起参考说明。当然，这暗示着描述必须能读而且必须去读。但在保护别人方面这就足够了。一旦忽略进入最终的设计，它将成为最终产品中的 bug。

冲突错误发生在两个设计特征有相冲突的设计时。这可能很简单，只是一个完全的冲突，其中设计规定一个文件处在文件单元中时可以被两个不同的用户同时打开，而底层的文件单元规定只允许单个用户访问。在这个层次的冲突中导致的 bug，如果不彻底改写系统中的模块，几乎是很难修复的。这是这个层次的 bug 的修复代价高的一个原因。几乎没有一种简单的方法去修复它们。

翻译 bug 是所有设计层的 bug 中最阴险的。翻译 bug 产生的原因是需求没有被正确解释，至少是从最终用户的角度讲。如果设计人员没有正确阅读需求或没有与用户一样阅读需求，那么这个应用的设计过程中就会出现错误。举个例子。考虑用户想拥有“在应用中查找文件”的能力。这是一个很模糊的需求，可以随意解释。设计人员会理解为通过名字、作者和文件创建或最后修改时的日期来“发现”文件。如果当时设计的描述完备，用户不会抱怨设计者的解释。但是，当最终的应用提交到用户手里时，用户指出没有“查询”功能。用户需要有一个能对文件进行全文本搜索的功能。显然，由于用户并没有得到该功能，设计就有缺陷了。你可能会认为这只是个遗漏性的错误，因为这个特征在设计中没有考虑。但是这个功能的确已经存在于最终的设计中了，只是形式不同或不正确。修复一个 bug 时，通常要浏览好几个层次才能正确地解决问题。需求文档通常为将来的软件开发项目所重用。

5.1.3 实现阶段的 bug

实现阶段的 bug 就是人们通常谈论的软件“bug”或软件缺陷。虽然一个软件缺陷可能实际上是一个需求或设计上的 bug，但软件 bug 通常是一个实现中的问题。实现阶段的 bug 是系统中最普通、最一般、最容易修复的 bug。但对于软件工程师来说不是如此。你仍然要费力去查找、修复和检验系统中实现阶段的 bug。

我们本章主要讨论各类实现阶段的 bug，因为它们最容易分类。不像设计 bug 和需求 bug 那样与应用相关，自计算机时代开始以来程序员们在对应用进行编码时就一直犯着相同类型的错误。但有些这样的 bug 也是和其出现的环境密切相关的。

我们可以将实现中的 bug 分为 5 种广义的类型。包括：

- ◆ 错误消息
- ◆ 图形用户界面错误
- ◆ 在实现阶段遗漏了功能
- ◆ 程序崩溃或其他严重缺陷
- ◆ 其他实现中的错误

第一种类型是那些其他功能正常而向用户显示了错误的消息的 bug。虽然这些消息可能是合理的，用户却认为这是个 bug，因为他们操作正确却得到了错误的显示。一般地，当用户遵循规则时，他们希望得到一个合理的结果。一个错误的消息不满足一个合法的显示模式。这种错误通常是由资源错误和数据相关而导致的。

第二种 bug 即图形用户界面错误的 bug。这些 bug 或者是对应用 UI 做的修改不太标准，使得用户无法正确工作；或者是它们虽然可以作为 UI 中的标准组件，但使用方式与用户所期望的不同。

GUI 的 bug 由于其容易表达和追踪，常常很容易修复，但在应用中的比例也相当大。例如，考虑一个例子中屏幕上包括几项输入和两个按钮：OK 和 Cancel。用户一般可能认为在屏幕上输入一些数据后按 OK 就能进入到下一个屏幕，而按 Cancel 则能回到以前的屏幕或退出。如果程序将 Cancel 解释为接受所有的修改，那么当然就要被看成 bug 了。

第三种包括实现阶段遗漏的功能，通常被看作 bug。如果设计描述指出你能够导入 X、Y 和 E 三种格式，而你的系统却只能导入 X，那么这一般就是实现的 bug 了。同理，如果描述指出你的系统可以接受 10000 个条目到一个表格，而实际数目只有 500，这一般也是一个 bug。功能遗漏 bug 通常在早期由针对初始的系统需求制定的测试实例来捕捉。

许多测试人员和 beta 版用户会报告一些 bug，而它们很多其实却是为了性能提高。作为一个软件工程师，处理这些 bug 不是你的工作，但它们会存在于数据库中。对于这些 bug，你的工作是显示出描述中所没有的合法的遗漏问题。如果你没做好，这些问题就被看作“特征潜伏”。我们并不将这种 bug 列入分类，但你要明白许多人会这样。

程序崩溃、挂起和其他严重的缺陷属于另一种软件 bug——即我们所说的第四种。当程序崩溃或退出以及破坏了自己所使用的数据文件时，一般人都会认为这是一个严重的 bug。但是，这个部分的 bug 归于那些“未期望的行为”问题。这些问题包括系统的用户和测试人员在操作软件时所发现的任何不满意的事情。

许多人认为语义错误、链接错误以及其他这样的问题都是实现中的 bug。这并不是因为系统从不会带着这样的问题发布。假设你可以看到系统中的每个特征和所有特征，系统的工作就是完备的了。我们将假设系统在实现层次是完备的，而且被正确地构建。在构建阶段的任何其他 bug 属于构建 bug。

有一种形式的语义错误问题经常被提及。这是在运行时出现的脚本语言错误。这种 bug 非常类似于由编码错误导致的运行时问题，例如除零错误或空指针赋值错误。所有这些可以

而且应该在程序的源代码中捕捉，但有时不是。程序的构建和配置都正确，但是也会运行错误。

5.1.4 处理阶段的 bug

处理阶段是软件开发周期中的一部分，包括配置管理、源代码控制、建立新项目、生成文档以及利用脚本使工作自动化等，而且是重复性的。处理也可以是大规模客户服务器或者基于网络的系统的一部分。包括诸如备份文件、解释日志、组织通信以及操作任务等。

你可能会认为过程 bug 不会对最终的开发工作造成影响，但你错了。实际上，许多 bug 的产生是因为低层次的处理错误传播到最终产品中而造成的。

让我们看两个简单的处理错误导致 bug 的例子。第一个是某种产品，第二个是某个大规模网站。在这两个例子中，我们将看到过程错误是如何对最终用户造成巨大影响的。

例 1：不正确的 DLL

很久以前，我在一个公司中工作，该公司生产能够帮助用户在 CD-ROM 上浏览、搜索和读取数据的产品。这的确是很久以前了，但概念仍然相同。这个项目是用 VC 在 Windows 3.1 中做的。在 Windows 3.1 环境中工作的最重要的一个事情是，所编写的代码要调用动态链接库（DLL），而这些文件在用户的硬盘上作为文件存在。DLL 的含义是可以让使用相同函数的多个程序共享代码。这样不管几个程序使用这个 DLL，内存中都只有它的一个拷贝。这在内存有限的系统中是一个好的策略。Windows 3.1 就是这样的系统。

但是，DLL 的一个最大的问题在于：它们与加载它们的程序之间的通信。Windows 提供在应用程序中使用所谓的“导入库”来建立存根，操作系统在运行时将其链接到加载的模块。DLL 文件本身是通过使用 Windows 环境中定义的路径，按所要求的库的名字找到第一个匹配的文件而加载的。现在你已经有了背景知识，我们再来看看实现的问题。

当同事和我开发应用的时候，我们遇到了一个持续性的问题。程序在开发人员的机器运行正常，但在测试机上却常常失败或崩溃。测试者就是我们老板，但这也不一定。项目开发人员设计了安装脚本（在自动安装工具之前），管理者和测试人员可以用它得到程序的最新拷贝并将它放入硬盘。然后由脚本建立不同的图标，这些是启动程序所必须的。

在安装脚本完成之后，管理者和测试人员则只需点一下图标，应用程序然后就能运行了。它可以运行在开发机上，可以运行在我们用来对最终产品收尾测试的计算机上，可以运行在产品的最终用户的计算机上，他们则会尽力看机器是否能按照他们的要求做。但是这个愚蠢的程序唯独不能在测试机上运行。

在安装东西时，你会尽力保证所有必须的组件都已安装到系统中。在我们的例子中，我们很清楚 DLL 问题，因为以前在安装时出现过这个问题。因此我们的安装脚本不仅要检查 DLL 是否存在，还要检查它的版本。如果版本数字（嵌入在文件内部）不够高，脚本会将新版本的 DLL 复制到机器上。这样，所有机器上的所有东西都是相同的，这是过程控制的基础。注意在尝试新的东西时，应保证所有的元素都相同。

因为我们知道库文件必须相同，而且管理者也正确地运行着软件，那只能在这个机器上寻找其他一些不同的东西了。最后，在一次实时分离中，我决定用实验证实是否什么东西都正确。对一个程序员来说，这相当于无奈时去翻手册或者停下来问方向。

当验证每个和所有的文件时，我发现应用系统中的 DLL 写得不正确。老版本的库依然存在。这其实没什么。但这的确是我们的脚本的行为。现在证明我们对这个情况中的因素并不十分清楚。看起来当一个 DLL 文件已存在并且正在被使用时，Windows 并不重写它。其他的机器都没有使用 DLL，但管理者的计算机还没有精明到能判断出它所执行的函数是错的，所以它的 DLL 从来没有更改。这是一个处理 bug 的典型例子。

例 2：被破坏的数据库

回到 20 世纪 90 年代后期，我当时在为一家互联网公司工作，就和世界上当时其他的开发人员一样，我的工作是一忙着一项能够带来世界性革新的产品。其中，我们在一个网站上使用了“前沿技术”。我不知道还有没有比它更危险、更加容易产生 bug 的词语。忽略小的细节，我使用的是面向对象的数据库、分布式组件对象模型（DCOM）服务器、活动服务器页面（ASP），从前到后地实现我的大部分功能。剩下的工作是在我们的网站上使用三个部分的工具。啊，年轻人的愚蠢。幸运地，那个时代已经过去了。

现在回到我们手上的工作中。有一天我们发现备份网站上的（所幸我们有备份网站）登录系统对内部的用户工作不正常。后来发现对外部用户也不正常。实际上，几乎所有的用户都无法登录。只有一个小组的人员能登录，但我们不知道为什么。登录所使用的过程如下：

- ◆ 检查用户是否已经登录。如果是，则拒绝访问，因为同一时刻每个人只允许登录一个账号。
- ◆ 如果用户未登录，则检查用户名和密码。如果它们有效且匹配，则将其登录。将用户登录的时间分配为其起始时间。
- ◆ 检查用户的定制属性，以决定首先将哪个屏幕显示给用户。在每个屏幕的顶部都有验证用户是否已登录的标记，否则用户只能在内部的页面上作书签并跳进去，安全地通过登录系统。这个特征并不怎么样，但确实能起作用。
- ◆ 登录过程的一部分判断用户发呆的时间。这通过其上次动作时间来算。每次用户在系统中做什么事情时，就修改一次上次动作时间。

我和同事在研究站点的 ASP 代码时发现，系统会因用户发呆时间过长自动与用户断开。这说明不了什么，因为用户才刚刚登录，并没怎么空闲。

判断问题的原因很花时间，不知道问题在何处。虽然，整个问题最奇怪的方面是对象数据库的组件和它的界面，都与产品系统中的没什么区别。实际上，备份站点是产品站点的一个映像。这是初始建立它的目的，即得到一个没有变化的运行备份。

我研究了这个问题。开始我怀疑操作部门的一些人没有正确地设置界面。我错误地提出了这个看法，立即遭到驳斥。所以我就检查了机器设置中所涉及的列表项内容。看起来没什么问题。Web 服务器安装了所有相应的软件，后端数据库的访问和安全设置也正确，一切

看来正常。但是数据库接口软件返回了用户上次登录时间的秒数，数值巨大。

只有从这里下手了。我逐行逐行地浏览代码，以发现数据到底从哪里来，以及通过了什么信息。登录时间表来自 Web 服务器，存储在后端数据库。Web 服务器然后查明后端数据库上次访问时间以及当前的系统时间。为了计算空闲时间，它只是从当前时间减去了上次访问时间。明白了，当前服务器时间。这个例子中当前服务器是后端数据库。初始的登录时间来自于数据库。一般这不是问题，不同的服务器自建立起就不断更新，而且我们保证它们与网络上的主服务器同步。问题是，有一个备份系统，它没有一个主服务器来同步时间。实际上，设置过程的一部分就是检查每个机器时间是否一致。我询问了操作人员，他们说保证已经检查过每个机器，它们的时间都是相同的。

那么这个过程到底出了什么错？简单地说，就是假定了太多。所有的机器的时间在几秒内确定相同。但数据库机器上的日期都比其他的机器早了一天。当它计算当前时间时，它使用上次启动时的秒数，像多数计算机那样。但在这个例子中，其时间比其他的机器超前了 84000 秒。84000 秒正好是一整天的秒数。将日期重新设置以后，问题解决。

这个例子中的问题在哪里？是因为软件报告了一个无效的秒数吗？当然不是。软件正确，并且在其他系统上运行良好。是因为 Web 服务器没有检查日期与数据库服务器是否相同吗？不是。该过程本来是要保证机器时钟的同步的。它的失败是因为有 bug。

5.1.5 编译的 bug

编译时出现的 bug 是因为编译过程中出现错误，或编译引擎出现了一个导致产品失效的错误。编译中的 bug 可以因源代码控制系统中的不正确的标记而起，或者因 make 文件的不正确的设置，等等。编译错误一般直接因处理错误而发生，而且常常通过对处理的清理来解决，以避免再次发生。

在几年前我曾遇到过一个编译错误。当时我在一家为用户做定制的软件的公司工作。定制的软件是用核心产品做的。在核心产品上，定制部分一般通过条件编译和内部的文件标志添加。当为一个客户开发一个新版本软件的时候，编译人员接到了一个新的要求，声明对这个客户如何设置新的软件。这个要求被存储在计算机里，用客户的名字来标志。

碰巧有一个新的客户与已有的客户名字相同。例如，称原来的客户为 XYZ 公司，新的客户为 XYZ 有限公司。很不幸，工程师接到的要求中一般不给出公司的全名，因为一直如此。他打开机器选择了 he 看见的第一个 XYZ。命该如此，公司列表里列出了第一个 XYZ 公司而没在屏幕上显示第二个公司。所以，他制作了一个类似的但不合要求的产品。

当产品送出后，我们收到了客户的一个传真，指出产品缺少一些我们许诺过的而且演示过的一些特征。实际上，客户说，如果我们只能做到如此，他们则另找人做。我们无话可说。我们对软件已经尽力，把大量的时间花在如何将要求的功能加在软件上。幸运地，错误不久就被发现，并且重新向用户提供产品。用户和开发人员都很满意，工程师们也懂得了在编译阶段也会有 bug。

5.1.6 配置的 bug

在世界上所有的软件开发中，你会认为配置过程中的 bug 是最少的。在多数例子中，这是对的。当软件将要面临客户或准备销售时，会得到注意。但是，当它在内部应用之间转移或客户机服务器的服务器边缘应用中转移时，bug 的数量将剧增。在一个配置 bug 中，软件是正确的，编译过程是正确的，但最后的软件安装不正确。

在我的职业生涯里，经历了无数的配置 bug。奇怪的是，系统越重要，安装越关键，在其配置中越可能出现错误。我所见过最重大的配置错误，也许发生在我在一家通过车载卫星系统发送数据的公司工作时。

在一个特定的情况下，系统预期要发送一个夜间下载，其中包含对卫星接收盒的软件升级。这个升级过程过去一直正常运行。但这次，却很快突然进入上载期。因为升级过程发生紊乱，工作人员由于上级压力而设计的封装格式，在从主局到上行设施之间的复制过程中损坏了。

被破坏的下载结果，导致所有的卫星接收盒有了无数的问题，从不能浏览卫星信号到失去盒中的所有信息。许多客户打电话抱怨这个问题。这问题如果一开始就按正常程序来，则可以轻易避免。但过程的好坏依赖于人的操作。如果你迫使他们走捷径，问题就会发生。这给了我们什么教训呢？很少。下次上载一样地操作，在修复时还会走捷径。这没有导致什么问题，更加不幸。我确信这样的公司还会遇到类似的灾难。他们忽略了那些早期建立的用以保证上载被正确安装而且不会给用户带来问题的过程。这是一个典型的配置 bug——当出现麻烦时避开过程。

5.1.7 未来计划 bug

如果设计者或开发人员能够看得长远一点，则未来计划 bug 就可以避免。关于未来计划 bug 的一个最好例子是千年虫问题。程序员开发的系统在存储器中用两位数表示年份，使得 1900 年 2000 年平安无事，但如果有人早就指出来并命令设计人员考虑的话，这个问题早就可以避免。当然，问题在于如何使设计人员因此而修改设计。如果设计人员和实现人员没有交流，就会有设计错误。对于一个明智的设计者来说，考虑别人的知识和经验是很重要的。第 15 章里，这些就可以在一个职业调试员的帮助下做好了。未来计划 bug 可以由在内存、磁盘空间和接口上取捷径而导致。一个简单的例子是在内存中进行排序，因为设计者没想到会有更多的条目。当条目的数量超过一百万时，这种排序在一般的计算机上就无法完成了。

你可能认为这样的 bug 已经过去了，因为千年虫问题导致了巨大的忧虑，长时间高消耗的、涉及了全世界许多开发人员的重复工作。但这还没完。几个类似的 bug 已经浮现。你要避免让你的系统上也出现类似问题。一些例子如下：

例如，考虑社会安全数字问题（SSN）。社会安全数字的格式是 xxx-xx-xxxx，x 代表一位数，从 0 到 9。那么是 300,000,000 人。所以你会认为如果出生率保持很低的话，我们在

未来的 200 年也是安全的。但没这么简单。SSN 永不重用。这意味着你可能有 100 亿个数字。自 1920 年起有多少人拥有过 SSN？（SSN 创建于那个时期）。问题变大了，是不是？有点严重了。每年当人们死亡，这个数字就跳出了循环。因为 SSN 不能重用，100 年后则没有 SSN 可以使用。另外 SSN 在每个地区是按“街区”分配的，所以你会发现 SSN 用得更快。用尽之后如何呢？有多少使用 SSN 作为惟一标识的系统需要改写？千年虫问题和它比较起来就没什么了。

这对于开发者和设计者而言意味着什么？假设应用中的数字有 9 位硬件编码的域。再假设扩大这个域以及修改现有数字所需要的大量的工作。害怕了？应该的。从千年虫问题可以学会一些东西。将 SSN 域设置为可变长度，不要写那些要求这个域是 9 位的代码。最简单的方法是允许字母，这将会大大延长 SSN 的生存期，但将扰乱当今的所有软件。

如果不能确信未来计划问题是重要的，那么考虑电话号码。电话号为 10 位，其中有区号、前缀和数字，意味着有 100 亿个可能的组合。假设人口 3 亿，这表示人均最多拥有 40 个号码。这可以嘛！很少有人有 33 个号码。等一下，这并不包含商业，不包含无线电话、传真等的号码。这样 100 亿好像不怎么高。而且由于许多数字不能用（如前缀是被限制的），这将大大减少数字量。在许多州都出现了号码短缺和区号分割。什么时候才能让这些“绷带”式的方法停止呢？

分给电话的代码的数量令人吃惊。我们有识别地区的代码和判断位置（城市、州）的代码。我们写出了让人容易地输入电话号码的 GUI 界面。我们有查找人们的电话号码的目录和系统。当号码系统改变时，这些代码会如何？

未来计划错误本来可以避免，但你如果没有认识到和考虑到，则会显得不熟练。在千年问题之后还有多少程序用两位数表示年份？爆发过多少公司内部关于他们的第三方工具是否与千年问题相适应的争论？看到这些类似的情况，而且如果处在类似于 SSN、电话号码、ISBN 和 UPC（条形码）的环境中时，就需要考虑其将来。不要弄出今后 20 年内发作的 bug。考虑将来，不要假定今天的工作不会在 20 年后影响应用代码。

5.1.8 文档 bug

文档 bug 的定义很容易。系统这样做而文档则说那样做。哪一个“正确”？许多例子中，文档是对的，因为它是来自于系统原始描述的。当然有理想的情况。许多例子中人们根据事实写文档，包括他们所认为的系统工作的方式。但另一些例子中，bug 却没有被追踪到代码，而是文档。文档有许多形式：用户指南、对系统进行定义的内部文档、屏幕的帮助文件和提示。其中任何一种都可产生 bug。

文档 bug 分三类：模糊、不完全、不准确。模糊 bug 指用户没有得到充分的信息来判断如何正确地做事情。例如一个库的例子，一个模糊的文档可能有关于如何调用函数的信息，而没说明首先调用什么以及边界效应（如内存分配）。

不完全的文档留下的信息不足以满足用户完成任务所需。例如，在一个字处理软件中，

用户可能要完成四五个任务才能检查作品的语法和拼写。这可能会使用户认为任务已经完成，而实际上还没有。这样的 bug 会导致用户不满，即使应用系统能达到用户目的。

不精确的文档则是最差的。这里，文档是错的，没有可弥补性。经常发生在后期对系统或描述的修改中，而且这些修改没有与文档的作者联系。不精确的 bug 困住了用户。并不是告诉你如何做特定的工作，你只会发现文档中的某些东西在最后的应用中并不存在。

最后，值得一提的是文档和需求问题都模糊的描述和语言的结果。计算机中用 0 和 1 工作。而人类使用语言。在语言中一个简单的误解，不管是在文档中还是初始问题的描述中，都可以轻易地在最终产品里导致严重的 bug。

5.2 严重性

对 bug 的完全讨论，必须涉及 bug 的不同的严重性水平。实际的严重性水平在公司间有所不同，但一般地，我们可以分出三个水平，如果你要加上对系统增加一个改进需求的 bug 的话就是四个。公司经常有很多的严重性水平，但我发现分成三个水平更简单。所以考虑如下三个严重性水平。

- ◆ 小的 bug——一般用户不会注意和发现的。小 bug 可以包括例如屏幕上的拼写错误，没有帮助主题、屏幕上的标签顺序问题和其他不能确定是否是 bug 的问题。例如，一个测试人员可能会在一个时间显示在一个屏幕上而不显示在另一个屏幕上时认为那是 bug，而许多人只会认为是用户界面的设计不同。这样的 bug 可能永远不会影响系统。假设影响度为从 1 到 10，小 bug 大约为 1 或 2。

- ◆ 讨厌的 bug——影响用户，但能够提供相应的功能。它们就像页面间错误的链接、屏幕上有歧义的文本或者是在难以重现的情况下崩溃而不会影响大量的用户。在发布软件时必须面对现实；因而一些“坏”的 bug 溜进了最终版本。这个范围内能够修复的 bug，应该在发布时间表上留出足够的时间来修复。如前，这种 bug 的影响度为 3 到 8。

- ◆ 严重的 bug——那些阻碍了产品发布的 bug。包括那些在按错了键或输入错误数据时，使系统崩溃或数据丢失的 bug。每人都理解，因为它们使你不能发布软件。影响度为 9 或 10。

严重性在讨论一个系统中的 bug 的试验时很重要。修复问题的时间和资源（人力）都有限。所以应该将 bug 分成水平，这样就可以判断什么是可修复的和什么是不可修复的。在已经判断出一个一致的能够修复的 bug 列表，就可以为它们分配资源并估计所需的时间。

5.3 bug 分类法

在我们的分类计划中，对世界上存在的任何类型的 bug 都有一个条目。注意这些分类与语言定义不同。例如，你不会发现 C++ 中的“assigning a bad pointer to a variable”。这些 bug

都被聚集在一个压缩性的题目“内存分配错误”下面。对于每种 bug 分类，我们考虑下面的元素。

5.3.1 名字

每个 bug 分类都有一个名字，这样我们就无须再用其全称来引用它了。我们可以引用“存储 bug”，而且能返回来看到底是什么。

5.3.2 描述

描述只是一个关于 bug 的症状的更长的列表。描述中包含的内容，足以用来向遇到这类 bug 的人做出解释。

5.3.3 最一般的环境

这是指哪些环境下在其中最容易发现 bug。环境值可以是“任何”，表示这个 bug 可以发生于任何系统中。其他的值有单机、客户/服务器、Web 服务器、嵌入式、产品系统和前台客户。

5.3.4 症状

多种 bug 都有独特的症状，但并不意味着它独有。虽然分类常常包含症状集，但许多不同类别的 bug 可能会有些相同的症状。调试技术实际上就是排除其他 bug 分类的可能性，从而判断问题到底是什么。这并不是说“啊！这一定是一个 (XYZ) bug。”如果希望仅仅通过对症状的观察来判断 bug 的分类，就错了。看起来不错，可在 bug 根本原因之间的分类中没这么简单，所以症状看起来可能简单。

5.3.5 例子

每个 bug 都有一个例子，所以即使无法从我的描述中看明白，也可以从所举的例子中领悟。在多数例子中，大部分取自现实生活，所以至少可以从我以前的遭遇中明白一些。

5.4 bug 的分类

bug 可以根据其症状和原因分为不同的种类。在这部分，我们研究不同种类的 bug，并看一下与每个类最为关联的环境和症状。

5.4.1 内存或资源泄漏

内存泄漏 bug 指一个内存由操作系统或内部存储器“池”分配，但是用完后从未收回。这实际上是资源回收的失职。在这个例子中，它既可以说是内存也可以是系统中的其他资源。

你可能认为内存泄漏只在支持存储器分配并提供内存重分配方法的语言中出现。例如，FORTRAN、VB 和 Java 不会导致这种错误，因为它们不支持内存分配，或者有一个垃圾收集机制，即在内存区长期不使用时自动释放。事实不只这么简单。资源泄漏可以出现于任何语言，而且常常是因为编程错误而起。

最一般的环境

内存或资源泄漏可以发生于任何环境。最常见于个人或大型的计算机上，以及支持内存分配和回收的语言里。在嵌入式系统中则比较少见。

症状

其症状往往是系统变慢或者突然崩溃，以及其他奇怪的症状。最容易发现的是系统的存储器（或资源）在一段时间内减少而且不恢复。你可以用三个部分的管理工具来验证这些，以观察程序是否有内存泄漏。

嵌入式资源的症状更难发现。例如，假设有一个可以从中分配对象的对象库，可以使用对象并交还给库。如果没能交还给库，那么它立刻孤立并且不可被再调用。只要通过观察库的使用模式（分配和回收的日志条目），就能看到持续减少的对象数量。如果正在使用一个管理工具并输出了日志，将会看到资源的持续丢失。

内存泄漏的其他症状包括对“确定性东西”进行分配的失败。例如如果正在分配一块 20 字节的内存却失败，必然要怀疑系统出现了严重的内存泄漏。

在 Windows 环境中，你可以使系统资源的数量设置限制。例如，你可能打开用过的设备环境（DC），而没能关闭。或者，在 Windows 或 Unix 环境中，也可能打开了过多的文件句柄。在每个例子中，分配新资源的企图将会失败。如果发现资源分配突然失败了，就应该怀疑内存泄漏。

例子

让我们看一个 C 程序中常见的内存泄漏的错误。这种代码在程序论坛上很常见。这是最简单的例子，内存分配了却没有回收。这种 bug 最讨厌，因为内存有时被回收，有时却没有。

```
char *buffer = new char[kMaxBufferSize+1];
memset( buffer, 0, kMaxBufferSize+1 );

// Do some stuff to fill and work with the character buffer

if ( IsError( nCondition ) ) // Did we get an error in the processing piece?
{
    Message( ("An error occurred. Skipping final stage") );
    return FailureCode;
}

// Final stage of code
```



```
// Free up allocated memory
delete buffer;
return OkCode;
```

应注意的是,在许多例子中,这段代码工作正常。如果处理阶段没有错误,内存就会正确地释放。但如果出现错误,则不能被释放,就出现了泄漏。这个例子中,你可以使用“聪明的指针”,它们可以自动地释放自己所指向的内存。如果指针是全局的,在那只能靠编程人员自己手工来做到这些了。

下面这段代码是我曾写过的一段 C++ 代码,有一个不明显的 bug。我对它感到很惭愧,但它确实说明了问题,值得一提。

```
class Foo
{
private:
    int nStringLength;
    char *sString;
public:
    Foo()
    {
        nStringLength = 20; // Default
        sString = new char[ nStringLength + 1];
    }
    ~Foo()
    {
        delete sString;
    }
    void SetString( const char *inString )
    {
        sString = new char[ strlen( inString+1 ) ];
        if ( inString != NULL )
            return;
        strncpy( sString, inString, strlen(inString) );
        nStringLength = strlen(inString)+1;
    }
};
```

我们看看发生什么,在多数例子中, Foo 对象被创建了,没其他问题。但如果调用 SetString 方法,问题就出来了。先前分配的串被后来新分配的串重写了,以前的内存还在。我们将重复出现错误,因为它先分配块,然后才检查输入串的正确性。我的建议是不要做这样的蠢事。如果在类中发现一个分配,要检查串在分配时是否会出现那样的问题。

5.4.2 逻辑错误

逻辑错误是目前最棘手的编程错误。当代码的语法正确而没做到期望的事情时,就是出现了逻辑错误。其原因是程序录入错误或者没有考虑问题。逻辑错误无法由编译器、调试器或代码分析器捕获,因为代码正确,并按照所写的来运行。

最一般的环境

任何操作系统、任何编程语言中，都会出现。

症状

最明显的症状是看见一个程序员挠头说：“但我确实让它那样做了。”逻辑错误的发现是由于代码执行了难以解释的错误动作。程序没有崩溃，但程序进入代码中不正确的分支。结果与期望相反。输出看起来奇怪，但没明显破坏的症状。

例子

我曾经浏览过下面这段有注释的代码，它来自于我做的维护工作中的应用。如果能看到那位写这段代码的哥们儿的脸色就有趣了。

```
// Make sure that the input is valid. For this value, valid ranges
// are 1-10 and 15-20
if ( ( input >= 1 && input <= 10) && ( input >= 15 && input <= 20 ) )
{
    // Do something for valid case
}
else
{
    //Do something for invalid case
}
```

下面有等价的 VB 代码：

```
IF ( (INPUT >= 1 and INPUT <= 10) AND (INPUT >= 15 and INPUT <= 20) )
    BEGIN
        ; Do something for valid case.
    END
ELSE
    BEGIN
        ; Do something for invalid case
    END
ENDIF
```

在任何语言中看这段代码，问题都一样。正确情况下，数字应是介于 1 和 10 之间而且要介于 15 到 20 之间，可能吗？好像不可能。这是一个典型的逻辑错误。

5.4.3 编码错误

这是在编写代码时的小问题。它可能是没有能检查错误的返回值，没能检查有效的情况，或没考虑系统的其他部分。但另一个形式的编码错误是错误的参数传递和无效的强制返回类型。

最一般的环境

有些语言本来就容易出现错。而且有些编码技术更加容易出现编码错误。类型严格的语言极少犯参数传递错误。不允许轻易地强制转换的语言极少犯类型强制的错误。VB 是类型较弱的语言，在类型转换时经常出错。C 程序员常因为参数中的变量数目遇到问题。Java 程序

员可以避免处理许多错误如异常等，但是如果处理不当，也能出现问题。当然如果使用 FORTRAN 编程，而且坚持用一般的块，那么如果不按照标准习惯的话就会遇到问题。

症状

黑箱测试（不理解软件的内部功能结构时进行的测试）中所没有解释的错误常常由编码错误而引起。其他的症状有编译器警告、不正常的中止以及无法看到正常的程序流。

编码错误其实处于程序问题的根部。毕竟，如果编码“正确”就不会有问题。我们将编码错误与代码中的其他错误分开的原因，是我们想识别那些单纯由于缺少注意那些可能易于在代码中产生的问题而导致的错误。

例子

在下面的例子中，函数接受了一个输入的整数，并将其转化为包含这个整数的串。不幸的是程序员没有考虑所有的事情。让我们看函数执行的结果：

```
procedure ConvertToString(InInteger: Integer, OutString: String,
    OutLength : Integer);
begin

    case InInteger of
        1:
            OutString := "One";
            OutLength := 3;
        2:
            OutString := "Two";
            OutLength := 3;
        3:
            OutString := "Three";
            OutLength := 5;
        4:
            OutString := "Four";
            OutLength := 4;
        5:
            OutString := "Five";
            OutLength := 4;
        6:
            OutString := "Six";
            OutLength := 3;
        7:
            OutString := "Seven";
            OutLength := 5;
        8:
            OutString := "Eight";
            OutLength := 5;
        9:
            OutString := "Nine";
            OutLength := 4;
```

```
end case;

end procedure;

(* My call of the function *)
var
    Length : Integer;
    Name   : String;
begin
    Length := 22; (* Initialized for another use *)
    ConvertToString( 0, Name, Length );

    For I := 1 to Length
        Write( Name[I] );
        WriteLn;
    end;
end;
```

上面的代码中有几个地方值得注意。`ConvertToString` 函数没有处理所有可能的输入。例如输入一个 0。而且函数没有初始化输出变元，让它们成为随机值。输出到函数并不成为问题，但输出长度的确是个问题。当函数存在时，长度没有重置，使其保持初始值。在例子中是 22。输出串没有被使用，所以一直空白。for 循环执行时，程序会进入除字符串部分之外的所有内存，并将异常地终止。虽然终止的实际原因是内存耗尽，但根本原因在于一些编码错误。

5.4.4 内存侵占

这发生在使用那些不属于自己的内存时。原因可以是超出了数组界限，或向一个内存区域复制了过多的数据。内存侵占在编程领域曾经非常普遍，因为无法知道某些东西的实际大小。注意只是访问不允许访问的内存，不一定会产生问题。实际上，遍历一个比估计要短的数组时，也不一定有问题，虽然它肯定会返回一个 `false` 的值。问题常常发生在想去修改那些无效的内存块时。

最一般的环境

内存侵占可以在所有的环境中出现，单机、客户/服务器、基于 Web 的系统。FORTRAN、C 和 C++ 之类的语言最容易导致这些错误，虽然 C++ 通过提供向量和字符串的类来改进了这个问题，C++ 中的这些机制能在它们产生之前捕捉它们。

症状

内存侵占的症状比较少。最主要的是程序崩溃。如果在某种给定的例程被调用时就崩溃，那就要看看它是否有内存侵占的情况。如果该例程没有问题，最可能的原因就是另一个先前被调用的例程破坏了变元或内存块。检查被调用例程的跟踪日志，找到那个有问题的例程，常常会看出错误。另一个有用的症状是程序中的变元的值被神秘地改变。例如，如果你在检

查调试器中的一个值，而它在 8 行之后没做修改的情况下发生了变化，这时应该怀疑是否发生了内存侵占。这是因为修改了超过数组长度的值，所以改变了一些东西。这些东西往往是内存中的下一个变元。当你看到一个变元神秘地改变时，检查离它最近的数组定义，往往发现这个数组在程序的某一时刻发生改写。

许多好的调试器可以捕捉到内存侵占，而且几个部分的内存检测例程也可以捕捉内存侵占。如果怀疑有这样的问題，可以打开其中的一个工具。如果显示有内存侵占，那么至少知道从何处下手了。

例子

我在做软件维护时，认为遇到最多的还是内存侵占 bug。其中一些已经在代码中存在许多年了。以下是个例子。为篇幅起见，我作了删减，但我认为它最好的一方面是修复人员加入了一个字符缓冲区以使它正常运行。我经常遇到这样的问题。

```
const kMaxEntries = 50;
int gArray[ kMaxEntries ];
char szDummyBuffer[ 256 ]; // This was inserted to fix a weird error
int nState = 10;

int ZeroArray ( int *pArray )
{
    for ( int i=0; i<100; ++i )
        pArray[i] = 0;
}

// Much later in the program

int Initialize()
{
    ZeroArray( gArray ); // Make sure that the array is cleared out to start.
    nState=10; // We need to keep doing this because it keeps
    // getting cleared someplace.
}
```

这个例子不只说明了内存侵占（一个严重的例子），而且说明了在应用中程序员如何修复它。我看完之后，对自己大笑。看一下 `Initialize` 函数，它一开始就将数组全置为 0，我不知道为什么要这样，但开发人员肯定知道。问题立即出现了。`ZeroArray` 函数通过对 100 个独立的入口初始化超过了数组边界。问题是它越过界后访问了不该访问的东西。程序员显然也发现了，他提供了注释，说明 `nState` 需要重置，因为它也会被清 0。当然，数组侵占正在清除它，因为在存储器中它们相邻。很奇怪这个程序一开始就能工作。

奇怪的是它是如何修复的。代码中，软件维护人员注意到 `nState` 受到了破坏。他不去弄清到底是怎么回事，而是在程序中其定义处的前面加上了一个大的字符缓冲区。这就解决了问题，因为每次被破坏的都是缓冲字符了。这并没有修复什么，只是将问题转到另一处。像我们以前提到的那样，这样的修复将问题掩埋得更远——而不是修复。它们是反向修复，因

为使得发现真正的问题越来越难了。

5.4.5 循环错误

一个循环错误发生在程序中的循环结构中。它有几个子类型。例如，我们有无限循环、未处理循环、减 1 循环和不正确退出循环。我们简单看一下。

无限循环是那些有 exit 的循环。它可以是设计的症状，例如一个循环在一个程序中一直处理信息。这时它们不是 bug。其他例子中，程序员想让这个循环可以跳出，但没有能给出一个完备的跳出条件。无意的死循环如此紧密，以致程序会锁定所运行的系统，即使程序和系统都正确。

减 1 循环很特殊，只在程序设计领域出现。“减 1”是根据一般人从 1 数到某个值的概念而进行的，比如 10。但在程序设计中一般是从 0 数到最终数字之前的值，比如 9。从 0 数到 9 而不是从 1 数到 10，这就是减 1。这些问题在 C 之类的语言里很常见，那里的数组索引从 0 开始。

不正确退出循环是那些过早或过晚退出的循环。它们可能是减 1 循环的变种，例如程序员想在到达 9 的时候退出，而程序却数到 10。

最一般的环境

循环错误可以发生在任何环境。一些循环错误，尤其是无限循环，很少嵌入到那些有监视器程序超时中断它们的环境中。因为这个中断，循环错误在嵌入式环境中更容易检测和消除。不正确退出条件的错误、减 1 错误和其他种类的循环错误更加复杂，除非加倍地调试，否则很难发现。

症状

不同种类的循环错误有不同种类的症状。如果程序只是锁定，重复显示相同的数据或消息框，就要立即怀疑这是一个循环错误。

减 1 循环错误在那些计算处理中很常见，如果手工计算发现整体或最终的值不正确，马上应该想到减 1 循环错误。同样，如果使用图形软件而且在屏幕上发现了所有的点，但是最后两个不相连，就应该怀疑是减 1 错误。

不正确退出循环，通常可以通过观察其出乎意料的终止来发现。在循环相互嵌套的嵌入式循环中很常见。一个不正确的条件，会使得只有在它执行完之后才能回到上层循环。

例子

因为这三种循环错误互不相同，让我们看一下各自的例子。下面是 VB 中一个无限循环的例子。它看来不一定是无限循环的，但多数情况下是：

```
Dim DoneFlag As Boolean
DoneFlag = false;
Do While Not DoneFlag
Get lFileNum, , baChunk
If baChunk = "EDN" THEN
```

```
DoneFlag = True;  
Loop.
```

这段代码想从一个文件中读一些行，直到有一行包含字符串“END”。有两个原因导致其无效。第一，要比较的字符串不是“END”，而是“EDN”。程序可能永远也找不到这个串，所以将持续下去。第二，程序没有检查文件条件的结束，这最终将导致读文件的错误，从而使得程序突然中止。

你可能认为减 1 循环只限于 C 和 C++，错！下面是 Delphi 中的减 1 错误。

```
var  
anArray : Array[1..50] of Integer;  
Index : Integer;  
begin  
Index := 50;  
While ( Index >= 0 ) do  
Begin  
anArray[Index] := 0;  
Index := Index - 1;  
End;  
end.
```

在这个例子中，程序员想聪明些，他按自己的方式向后遍历数组，认为不会有错。当然，如果你检查了循环，会发现它被执行了 51 次。但数组对象里只有 50 个元素。如果它不从 Delphi 系统中触发一个立即性异常的话，这当然将在后面的程序中导致问题。

最终的情况是不正确的循环退出条件。这可以用一组嵌套循环来表示。在下面的例子中，我们想在内层循环中反复计算一些东西。

```
int nIndex = 0;  
for ( int I=0; I<kMaxIterations; ++I )  
{  
while ( nIndex < 20 )  
{  
ComputeSomething( I*20 + nIndex );  
nIndex ++;  
}  
}
```

代码看来会去做那些我们让它做的事，内层循环 20 次。但问题是，nIndex 应该小于 20。外层循环第一次后，变量值成为 21，内层循环结束。这是正确的，也是你所期望的。但问题是下一次循环时，内层循环不会被执行。显然这不是我们期望的。为什么会这样？一旦 nIndex 到达 21，内层循环就停止。但循环变元没有被重置，所以循环还没开始就已经满足了退出条件，这将导致问题，而且结果是正确地退出循环条件。

5.4.6 条件错误

条件错误由于条件语句书写不好或考虑不周而导致。在多数语言中，条件语句通过 if 语句或条件循环来实现，比如 while 和 do 循环。条件错误可以是对布尔代数的误解或条件

嵌套错误造成的。

条件错误与逻辑错误类似，与循环错误也相似。但也有自己的独特性。使用一个使用嵌套的 `if` 语句来完成任务的语句，可能会导致开发人员没有考虑条件路径的正确性。而且，开发人员可能会处理一个给定的经过条件语句的特殊例子，但没考虑另外的条件，比如 `else` 语句的例子。

追踪条件错误可能是调试人员的一个典型的痛苦。因为开发人员会下意识地认为程序的逻辑正确而忘了什么事情，所以就会经常忽略条件错误。

最一般的环境

因为条件存在于任何操作系统、语言、平台，那么随处都可以发现它。

症状

最明显的是程序流没有到达期望的地点。应用系统的跟踪日志经常显示出那些本应到达都没有到达的函数和方法。使用调试器之后发现应用程序跳过了应该执行的语句。最简单的识别方法是在代码加上调试逻辑，以显示执行了哪些分支。如果错在条件，则它将显示一些与正常预期不相符的步骤。

例子

让我们看一个看起来正确的条件错误。在这个例子里，程序员检查一些来自用户的不同输入。每个例子中，程序都进行一个步骤。在这个例子中，我们使用 C 语言的 `switch` 作条件操作符：

```
int nOper = GetUserInput();
switch ( nOper )
{
    case kOpenFile:
        DoOpenFile();
        break;
    case kCloseFile:
        DoCloseFile();
    case kDeleteFile:
        DoDeleteFile();
        break;
}
```

这段代码有两处错误。首先没有处理缺省的 `case`。如果 `GetUserInput` 返回了一个不期望的值，我们必须处理。它后面的代码会依赖于 `nOper` 的值，如果有些东西不知如何处理，那就必须尽快显示出来。

第二个问题是语法条件错误。注意，在 `kCloseFile` 的情况，调用 `DocloseFile()` 函数之后没有 `break` 语句。这样程序将继续执行下面的 `kDeleteFile` 情况，这就把问题推到了 `DoDeleteFile()`。就是说关闭文件就会删除文件。操作顺序决定了这些，所以第一次运行时这就是一个 bug。

条件错误可以出现在 `if`、`switch`（或 `case if`）、`while`、`do`、`for` 之类的语句和 C 语言的三

元操作符中。任何在给定条件下决定做什么事的语句，都会隐含有条件错误。

5.4.7 指针错误

指针错误是在用一些东西作为另一个条目的非直接指针时出现的。许多程序员把它作为存储错误。在分配一块内存并给定一个指针之后发生意外，就有了不正确的指针。但不一定都是这样。也可能是数组的索引被破坏或重新赋值了，然后又被使用。这个索引是数组的指针，用来访问数组的元素，但它已经是无效的了，所以就产生了问题。

对于内存指针，其指针错误有三种。第一种是未初始化。指针被用来指向某些东西，但我们没有赋值给它。它们是随机的，而且有奇怪的结果。第二种是删除的指针，但仍然使用。这是在分配给一个内存块一个指针之后，删去内存块但没删除指针造成的。指针指向一个随机的区域，里面可能有想再使用的东西，这样当它再次使用时，会包含大量的垃圾。

第三种指针问题是无效指针。和前二者不同，无效指针指向一块内存区域，的但里面并不是所期望的值。无效指针最难跟踪，因为使用的语言会将所指向的数据类型强制转换为想要的类型。但强制并不能使数据转换为有效的类型，它只是到了相同的边界。这很麻烦，例如所指向的数据类型包含有自己的指针的情况。无效指针问题在类型强的系统中很常见。

最一般的情况

最常见于支持动态存储分配的语言里。而且，在支持动态存储分配的操作系统里也很常见。许多嵌入式系统和脚本环境中没有指针错误问题。

症状

指针错误有一些奇怪的症状。第一，程序往往崩溃，或者以一种不可预见的、令人莫名其妙的方式运行。第二，一般会看到栈破坏、内存分配失败或变量值奇怪地改变。

如果问题是一个指向随机的区域，改变一行代码的指针可以改变问题的发生地点，这就是处理指针问题时的第一线索。如果在被认为有问题的代码处放置了一个打印语句或一个新的变元之后问题就消失了，那么这可能是因为一个错误指针的问题。

那些发生在删除掉指针所指的内容之后的指针问题，比较难以检测。当重新使用删除掉的指针时，会出现一些症状。第一个症状是分配会因为尾指针的破坏而失败。另一个是问题可以重复出现，但没什么意义。你会发现程序会在 C++ 对象的一个方法中，或在指针（C、Pascal、Java）被再引用的时候，发生崩溃。Java 之类的语言比较礼貌，只会在这个时候发出一个异常信息。

例子

让我们看一看最一般的指针错误的形式，删除一个指针后重新使用。如下例，当编码不正确时也不太明显。

```
void cleanup_function( char *ptr )
{
    // Save the pointer to disk
    SaveToDisk( ptr );
}
```

```
// Now get rid of it so that we have room for others.
delete ptr;
}
int func()
{
    char *s = new char[80];

    // Do some things with s

    // Cleanup s
    cleanup_function( s );

    // Do some more things

    // We want to make sure that there are no memory leaks, so we will clean
up
    // after ourselves
    delete s;
}
```

可能在重新使用一个已经删除的指针时，看起来不会太明显。在这个例子中，程序员为求简洁，他开始的代码可能是要把指针分配在函数的顶端，然后由于习惯，在函数末尾删除。这是防止内存泄漏的一个好的编程习惯。但问题在于 `cleanup_function` 例程。该函数可能是后来另外的人写的，用来保证所有的指针都存放在永久驱动器上，然后再释放，以免有任何泄漏。实际上，问题中的代码来自于从内存池系统中重用的一组函数，它们在破坏指针前将数据存入磁盘，这样就可以从永久池中取出另一个指针。问题是，重用没有考虑现有的系统。只重用系统的一部分是正常的。

当第二个删除出现在 `func()` 的底部时，结果是不可预测的。最好是 `delete` 函数发现这个指针已经被破坏，而且在程序的后来发生存储器崩溃，这是最难发现和修复的。

5.4.8 分配/释放错误

这个错误指分配和释放的顺序不正确。显然正确的顺序应该是先分配后释放。但有其他的可能发生：

- ◆ 释放一个东西，然后再分配它。这种情况出奇地多。这通常是在重新分配一块内存前，没有确保以前的分配被清除以避免内存泄漏而导致的结果。
- ◆ 分配、再分配然后释放。这是一个典型的内存泄漏。
- ◆ 分配一些东西，但没有释放。这是另一种典型的内存泄漏。

而且对于特定的语言和操作系统，有第四种错误，即用一种内存管理方式分配，但用另一种方式释放。但幸运的是随着内存管理的标准化，这种情况减少了。

混合和匹配分配是如何发生的呢？这种错误有许多原因。一个是没有对内存分配初始化或检查初始值。例如某个对象是由一个指向一块内存的指针初始值为零（通常是 `null` 或 `nil`），

就需要检查看看那个指针在释放以前是否已初始化。虽然这听起来是显然的，但在许多应用中还是出现问题。另一个导致内存分配/释放的严重问题的是：库或工具函数内的隐性边缘作用。例如一个应用函数可能会分配一块内存并将其返回给调用例程。如果没有工具例程的源代码或足够的文档，就很容易在调用例程中使用完后没有将其释放。同样，如果假定一个工具函数分配了一块内存并释放了它，那么当工具函数只将指针返回给一个静态变元或者一个全局的对象池而它们从没被分配过时，就会出现严重的问题。

最后，这个 bug 的一个变种是存储器并没有真正地被分配或释放，却保留在可用对象池里。如果不能将一个对象返回池里或返回了两次，就会导致类似的问题。

最一般的情况

如同指针错误，分配/释放错误也发生在支持动态、存储分配的操作系统中。释放应用程序中所分配的内存方面的错误，多数发生在没有内置垃圾回收机制的系统中。存储器池错误是指用完对象后没有返还给存储器池，或者返回了两次，多数发生在不真正分配内存的嵌入式系统中。

症状

释放分配的内存所发生的问题常表现出相同的症状，即内存泄漏错误。释放分配的内存时表现为指针错误，将对象返回到存储器发生失败时表现为内存泄漏，导致程序最后跳出了全局对象。

例子

以下的代码段中有一个讨厌的 bug，有两个特点使其难于发现。看代码：

```
class xString
{
private:
    char *sString;
public:
    xString( void )
    {
    }
    xString( char *s )
    {
        sString = new char[ strlen(s)+1 ];
        strcpy ( sString, s );
    }
    ~xString( void )
    {
        if ( sString )
            delete sString;
    }
};
```

代码中有两个不同的问题，一个比较明显。注意如果使用类中的空构造函数（无参数），那么 sString 成员变量将不会被分配或者赋为空值。对象销毁时，调用析构函数（~xString）。

因为先前没有初始化 `sString` 的成员变量，将不会通过 `if` 语句，所以它将在没有分配的情况下删除。

另一个问题则更含糊了。如果在自己的机器上编写和测试这个代码，将不会有什么问题。因为多数开发人员是用其内置的调试信息编译和链接程序的。这样做的一个边缘效应是存储器被初始化为 0，与 C++ 的 `null` 值相同。如果不是为将这个类投入产品或发布，将永远看不到这个错误，而它在投入到产品环境中之后会马上出现。

注意：要在产品环境中测试软件。

5.4.9 多线程错误

多线程错误是因为在一个实现了多线程的程序中，两个线程试图同时访问或修改相同的存储器地址。这是计算机界的禁忌，多数环境中遇到这种现象时会把它定为异常，或终止程序。

注意：例如，假定要往一个数组里加一个数。有一个专门的函数修改数组元素的数目以及内容。这样的函数被称为不可重入函数，因为它们在函数期间内依赖于对数组的整体访问。如果在第一个函数修改数组时，有另外一个线程也来修改，那就很难说系统会出现什么样的差错了。有许多的方法可以避免这样的修改，并有一些如信号量、临界区和 `mutex`（互斥）的装置。所有这些装置都避免了一个线程访问被另一个线程锁定的临界区。

这个问题很难检测和修复。一些系统会显示错误信息或者异常，以说明出现了问题。另外一些系统中，会有奇怪的行为，因为数据被多方修改。由于少有可靠的多线程调试器，所以很难追踪线程错误。多数的调试是跟踪当前每个函数中的线程的 ID，或者将它与最后已知的 ID 比较，或者把它放到调试器中观察。不管如何去寻找，都很难证明它确实是多线程错误。只能提取有嫌疑的代码段，然后在一个多线程访问管理中运行它，才有可能检测到。

最一般的情况

显然必须是在支持多线程的环境或应用中。客户/服务器应用、Web 服务器和一些特殊的单机应用会出现这个问题。当然用在 Web 服务器中的 COM 组件也有这个问题，其他一些可能的情况是一些线程化的应用，如后台排序或存储，或那些与外部应用之间的 socket 连接的应用。

症状

最显著的症状是：问题不发生的时候就是应用是单线程或只有一个用户的时候。例如，如果从一个没有问题的单用户网络应用转向多用户的应用时马上遇到了问题，那么就可能有一个多线程 bug。

另一个症状是：两个不同的线程想访问函数的同一方法时，会发生“挂起”。这是函数不可重入的一个标志，而且应该想到用一个信号保护装置（临界区，互斥量等），以避免今后再发生。

例子

以下的代码段中包含一个比较明显的多线程错误。其中，在一个应用中用了“标记”。一个用于“锁定”程序对数据库的访问，另一个要“锁定”用于应用中的整个线程的全局空间内的一个数组。下面是它们二者冲突的情况。

```
int func1()
{
    WaitForLock( &arrayLock ); // Wait for the array to be free
    // Do something with the array
    WaitForLock( &dbLock );    // Wait until we have access to the database
    // Do some stuff with the database
    ReleaseLock( &dbLock );
    ReleaseLock( &arrayLock );
}

int func2()
{
    WaitForLock( &dbLock );
    // Do something with the db lock
    WaitForLock( &arrayLock );
    ReleaseLock( &dbLock );
    // Do something with the array
    ReleaseLock( &arrayLock );
}
```

你可能认为这段代码没有问题。毕竟每个函数都等待相应的锁定（信号量或互斥），做一定的事情，然后解除锁定。问题是 func1()和 func2()，同时被调用后会怎么样？在这个例子中，func1()得到数组锁定，func2()得到 dblock 锁定。这样它们就相互等待对方释放的锁定之后才能运行。这时这个死锁使得两个函数相应地相互等待对方的锁。但在查看每个函数的代码时，我们所看见的是它们得到相应的锁之后完成工作，再释放。错误是因多个线程同时调用两个函数。

多线程错误一般是因为程序员没有阻止两个并发进程同时相撞而引起的。如果要写多线程代码，就应小心地避免函数模块间的高度聚集。否则将会面临难以检测的危险，而且要调试系统。

5.4.10 定时错误

曾经尝试在着陆和起飞时建立两个飞机间的紧密的联系吗？如果是，那你肯定遇到过经典的同步 bug。在纸上看起来不错的连接方案在现实中无效，因为不可能在按照连接中描述的时间要求内，从一个关口到另一个去。

定时错误发生在一个事件被设计和实现成以特定的频率，或在另一个事件错误的空隙内发生的时候。在编写中断服务例程时，没有在给定的时间内完成所做的处理时，最容易出现定时错误。在这个例子中，可能会因为另一个中断的产生而使得例程终止。

最一般的情况

定时错误在中断函数和基于定时器的函数中最容易出现。往往会在操作系统服务例程中或与硬件设备打交道的嵌入式系统中遇到它。虽然在多用户或线程的环境中也可能得到定时错误，但这个 bug 通常归入多线程 bug。它限定于那些时钟非常重要的环境中。

症状

定时问题最好通过观察。如果一个错误只在某些数据被处理或发生某些事件时才出现，那就可能是定时问题。例如，如果中断处理器一直正常运行，但每天失效一次，那么就应该检查在那个每天一次的处理中还有什么其他因素。可能会发现中断处理触发了一个备份请求，而往缓冲区加入了太多的数据。如果是这样，那么这个请求会占用很长时间，另一个中断将触发，然后程序崩溃。

检测定时错误，要求有内置的诊断工具并有深入的考虑。统计数据，比如给定时间内的中断或处理的事件的数量，是很有用的，因为它们能显示出处理曲线中易导致问题的峰值。在第 9 章，我们讨论如何实现自己的内部调试工具。对于定时问题，这是必要的。

定时问题的最一般的症状是：

- ◆ 看起来与代码无关的中断。
- ◆ 输入和输出被挂起而不能重入的问题。
- ◆ 那些经常或平衡或突然的数据流。
- ◆ 对服务器例程加入调试代码，如输入打印语句时，应用程序突然崩溃。

例子

几年前，我处理一个嵌入在电视卫星盒中的系统。该系统负责收集来自卫星的输入数据存到磁盘，然后送到盒子的输入端，然后与电视连接。这个系统允许我们完成诸如将程序存到硬盘以及停止“现场直播”电视等工作。几乎所有时候系统都正常工作。但有一次在调试模式下，程序很奇怪地崩溃。这从未发生过。我和同事都无法判断发生了什么。

盒子的调试模式和产品模式间的主要区别是：在调试模式下会看到诊断信息从串行线被传到计算机。显然，在此这样的信息不应被发送，因为它能让别人看出盒子如何工作，而且会使系统变慢。问题是发送这样的信息通过电缆需要 500 毫秒。每个发送的消息都要这么多的时间来格式化、预备和发送到串口。如果有大量的信息，盒子最终将关闭自己，因为它要把所有的处理时间都花在通过导线发送信息上。工作在这个系统上时会迅速发现应该保持消息最小化，并在每条消息中加入尽可能多的信息。

问题在于，为什么盒子在调试模式和给定的时间才出现问题呢？答案在于消息处理系统。在代码的特定的时间区域内，中断服务例程将会在数据来到系统时，向缓冲区倾倒入数据。在正常基础上，我的同事打印了诊断信息，例如所处理包的数目、所检测错误的数目等，打印到了控制台。诊断的定时是基于处理的数据量，这样就不至于太影响系统了。这部分运行正常，没有错误。但当卫星广播通往盒子的数据压缩率变为更高的时候，系统就开始经常地崩溃了。

当我们开始查看问题后，立刻明白了。收集数据的中断例程的速度，远远超过了处理新的压缩率的速度。但消息系统依然按照旧的压缩率输出消息。这导致消息的数量急剧增加，消息输出时系统变慢。因为消息上没有时间标记，所以不能看出它们在新版中出来的速度更快。结果在控制台上就花了好几个小时来观察输出流，并且与以前的输出相比较，以发现新版本中有更多的输出。我们改变了输出率以匹配输入率，问题消失了。

应该说我们做的并非真正的“修复”。问题是如果数据率提高还会出现问题。由于一些行政上的原因，“正确”的修复被认为花费太高太久。这样，可以看出政治是如何在调试领域里乱搅和的。

5.4.11 分布式应用程序错误

某些错误可发生于任何环境：内存泄漏、缓冲区溢出、编码问题，等等。但某些错误只发生在特定的环境。分布式应用程序错误就是一例。虽然这些错误的根本原因在任何环境中都相同——拙劣的编码和设计，或者差劲的需求分析——分布式应用程序有一些全新的问题，因为它们组合成新的应用程序，可能运行在不同的主机上，就很难抓住链上的某个错误。而且对系统的修改的影响比较大，开发人员也意识不到，那么对这类错误的测试就是一个噩梦。

分布式应用程序错误可以定义为分布式系统中两个应用程序的接口中的错误。其形式可以是从小方到另一方的未处理的消息，或者对分布式应用程序崩溃的测试。多数分布式应用程序错误是由检查错误的返回值时的失败，或格式化应用程序间的消息时的变化而引起的。有时候，两个应用程序之间不直接“交谈”，而是共享一些永久的媒介，如数据库。这时，问题很容易因媒介中放入了被破坏的数据或应用程序之间的锁定问题而引起。

另一种分布式应用程序错误是由组件的动态加载引起的，如 Windows 中的 COM 对象。许多问题是界面中的变化或作用在组件上的外部原因引起的。例如，如果一个 COM 组件崩溃，使用这个组件的应用程序（尤其是 DCOM）一定会检查并处理。当然，不只是 COM 有这个问题，其他如打印假脱机系统或通信处理等也有。但在 COM 中，如果系统不正常工作，问题就比较明显了。

最一般的环境

分布式应用程序错误显然只在有分布式特点的系统中。真正的单机系统永远不会出现这个错误，嵌入式系统也不会。只有那些含有并发运行并为系统的整体任务分担职责的应用程序系统中才会有这些错误。如果要写一个 Unix 的控制台应用，除非使用数据库或管道与其他应用程序相连，否则不需要考虑分布式应用程序错误。

症状

根据其种类，症状可以是程序崩溃到挂起等情况，即看起来没什么在运行。而且可以在分布式链中的一些应用程序中看到一些奇怪的消息，这是因为缺少应用程序之间或与系统之间的连接。当工作在分布式环境中时，如果看到某个程序的运行突然变得很奇怪，就应该

怀疑是分布式问题。要寻找程序链中的应用程序之间的连接或死锁情况。

例子

考虑下面的代码，来自于一个 COM 对象，被 Web 服务器环境中的一些 ASP 代码使用。

```
STDMETHODIMP IConnectObj::CreateUser(BSTR org, BSTR libCardID,
BSTR name, BSTR passwd, VARIANT *vAccountID,
VARIANT *vSessionID)
{
    Guard protect(mutex);
    LOG_METHOD_START("IConnectObj::CreateUser")
    long Acct = -1;
    long SID = -1;

    HRESULT hr = GetAccountServer()->CreateUser(org, libCardID,
        name, passwd, &Acct, &SID);
    if ( hr != S_OK )
    {
        Logger::log(ERROR_LEVEL, IConnectObj::CreateUser",
            "Unable to create user. Error: %ld", 1, hr);
        hr = E_FAIL;
    }

    if ( hr == S_OK )
    {
        // Do some stuff.
    }

    return hr;
}
```

代码中，COM 对象通过 GetAccountServer()函数与远端的 DCOM 服务器相连。这个函数通常会返回一个错误消息说明无法与远端 DCOM 相连。这里，我们记录一个错误消息，内容是请求的动作无法执行，然后返回调用的 ASP 脚本一个错误，说明动作失败。

正常情况下，你会认为这是不错的防错代码。它检查错误，返回调用者，正确地处理异常。但当你处在 ASP 一方时，就不一样了。实际上代码看起来如同这样：

```
ON ERROR GOTO HandleError
CreateUser organization, libraryCard, name, password, accountID, sessionID
; Do something when we create the new user.

..
HandleError:
    PrintError "Unable to create user. Probably a duplicate user name"
```

注意上面的代码假定了一些与底部的 COM 代码不同的东西，如果用户不能由数据库创建，这个假定就是用户名已经在系统中了（系统中的一个错误）。如果到远端数据库或 DCOM 对象的连接失败，错误就应该是有一个与终端用户名字重复的名字。

5.4.12 存储错误

这是在永久存储设备遇到错误不能再进行时发生的问题。这可以是文件系统、数据库或一些备份设备。在任何情况，如果错误不能解决，都将影响到整个系统。

当一个程序与一个数据库交互时，人们期望数据库能够安全地进行保存。在特殊情况下，如电源中断或磁盘损坏，就麻烦了。其他情况下，与数据的连接会失败，要求换一个新的备份设备。无论怎样，应用程序必须面对所需要的数据已不存在的事实。

有时，你会依赖于用于其他用途的存储设备，如配置文件或备份文件。例如假设依赖一个配置文件中的数据，但它已经删除。没有这个配置文件，应用程序中会有足够的信息吗？加载配置信息时发生的失败，会使程序崩溃吗？如果程序使用外部存储机制，那就要保证这些。对存储设备的简单测试是：删除需要的配置文件、数据库或备份存储设备，以看能否导致问题。如果是这样，就很可能是用户错误产生的原因。

最一般的环境

存储错误发生在程序使用存储机制时。一般地，它在往输出文件里存放数据时，从输入文件中输入数据时，或使用备份机制以保证数据一致并免于破坏时发生。

在使用数据库来恢复或存储信息的系统中，存储错误一般发生在后台服务器。当数据存储在远端数据库服务器时，本地存储一般不会出问题。

对于配置文件之类，有两种情况最容易遇到存储错误，它们是用户删除文件，或磁盘变满或者被破坏时。如果没有文件管理系统，这可能发生在客户服务器或产品 Web 服务环境中。当然在一个单机系统中，如果不首先检查配置文件的存在，就可能发生任何情况。

症状

存储错误有许多症状。多数操作系统在即将用尽磁盘空间时会警告你，所以这类错误很容易检测。

当然这是假定运行应用程序的计算机也被管理了。如果不是，那么就需要有一些远程控制系统来保证机器不会用尽内存或磁盘。

对于数据库，每天或每周（依赖于系统的负载）的整体检查，应该足以能够保证其正常性，以及没有磁盘错误使得它不可使用。对于配置文件，你可能会注意到程序没有正常响应。一个简单的检查是在系统中嵌入一个诊断屏，显示出当前的配置设置。我们会在第4章更多地讨论内置诊断及其应用性。如果系统已经正常运行，但突然不能访问数据库、备份机或其他服务器了，可能就要怀疑是否有一个配置问题。如果代码对你开放的话，可能需要检查代码以判断它在哪里读取磁盘上的配置信息。检查配置文件是否存在、是否可读以及有没有数据。如果有一点不对，就可能是存储错误。

最后，如果一个程序在同样的环境和配置下运行时，只在其中一台机器上发生失效，而程序对磁盘没有访问权时，会发生奇怪的存储问题。

例子

让我们看一个简单的例子，这是在操作文件时一个未解决的错误。这里，我们只是打开服务器上的文件，从中读一些信息，然后用读出的数据做一些事。这个 Java 例子说明如果处理不当，那么解决异常也不是最好的办法。

```
import java.io.*;
import java.util.*;

public class StudyFileStream {
    public static void main(String args[]) {
        try {
            FileInputStream theInput = new FileInputStream( args[0] );
            System.out.println( "Copying from file: " + args[0] );
            theOutput.close();
        } catch (ArrayIndexOutOfBoundsException ie) {
            System.out.println("No filename is given");
        } catch (FileNotFoundException filenotfound) {
            System.out.println( "Could not open files." );
        } catch (Exception e) {
            System.out.println(e);
        }
        int valueRead;
        while ( (valueRead = theInput.read()) != -1 )
            System.out.println( valueRead );
        theInput.close();
    }
}
```

注意在这个例子中，我们打开一个文件，从中读出一系列的整数。这些整数是给那些读取控制台输出的用户显示的。我们正确地处理了文件不存在或者名字错误的情况。但在处理完错误后，我们又继续读文件。这样，程序当然就会出现异常和崩溃，这是最可能的结果。但如果程序没有用错误的输入进行测试，潜在问题可能就不会被发现。

存储错误对程序员很常见。如果没有充分的测试实例来处理存储系统中的潜在的错误，这些错误就会进入产品系统，这时只能责怪自己。

5.4.13 集成错误

集成错误指两个子系统已经分别测试和验证，组合后在其交互时出现了问题。一种形式的接口错误是刚才所说的存储错误。但多数的接口错误是因为开发人员对接口中某一端的假设。

接口错误中一般的症状是开发人员期望对他的例程的输入是“干净”的，即没有任何错误或无效输入。如果一个例程假定传过来的指针非空，而另一个例程传给它一个空值，那么前者将会崩溃。这并不是第一个开发人员的错误，因为他明确地说明例程不能正确处理空的指针值。也不是第二个开发人员的错误，但可能没接到关于空指针导致问题的信息。

最一般的环境

集成错误最多发生在有多个独立的开发人员，或使用第三方软件的系统中。它们可以发生在任何系统，从单机系统到多用户的客户机/服务器系统。集成错误常因缺少对子系统之间的整体性测试，所以任何对子系统做了测试而缺少在子系统间的测试的系统，都会出现这个问题。

症状

最一般的症状是错误只出现在产品测试中，而不出现在给定模块的单元测试中。证明集成错误的一个好的方法是：将单元测试与最终的产品系统比较。如果产品测试计划显示了一些在单元测试计划中没有处理的情况，可能就是对整个测试考虑不够。

例子

假设要写一个 Delphi 例程，计算一组数的平均值。可能需要使用一个叫做 Array 的对象，包括一组数；和一个叫做 Count 的函数，返回元素的数量。下面这样写函数：

```
Function Average ( anArray : ArrayObject ) : Integer;  
Var  
    I : Integer;  
    Total : Integer;  
Begin  
    Total := 0;  
    For I := 1 to anArray.Count() do  
        Total := Total + anArray.GetAt(i);  
    Average := Total / anArray.Count();  
End;
```

看起来能工作，是不是？你传过来一个数组，里面有其所有的值，并返回其平均值。还可以使用负数、正数以及正负数的组合进行测试。

假定现在来了另一个开发人员 Charlie。他喜欢这个函数并用在自己的代码里。只是有个问题。他向这个函数传递用户在对话框中输入的数。那么他就没理会用户到底往对话框里输入了什么。所以传递到函数的数组对象可能为空。看起来不严重，是不是？现在问题清楚了，是不是？如果数组对象为空，则 Count 方法将返回一个 0。函数最后的除法将产生测试过程从未出现的除 0 错误。现在就进入集成错误了。你写函数时的假定是数组非空。当 Charlie 使用函数时，他不知道。噢，出现瞬间的 bug。

5.4.14 转换错误

计算机世界中的最一般的错误是：数据形式很少与所希望的相同。用户往往用文本输入数据，那么文本必须被转换为数字或其他的数据输入。程序中一种格式的数据，往往要转换为另一种。程序间的数据交换，必须经历一方的“标准格式”到另一方的“标准格式”。还是别从对标准的“扩展”谈起。

转换错误发生在将一个类型的数据转换为另一类型发生错误时。原因有许多，可能有一

个差劲的转换过程，或数据可能包含不能被正确转换的信息。例如“ABCD”转成整数是什么？我不知道正确的答案，但知道有无数错误的答案。其他转换包括数据类型之间的隐性转换。例如一个函数可能需要一个浮点类型的数。当传递 123 时，就隐性地转换为 123.00。这样通常都不错，但有时没那么简单。例如计算 12/24 的值。

在多数语言中，12/24 后得 0.5。但 C 和 C++ 则不然。如果你将 12 除以 24 并赋为整数值，就是整数除法。两整数相除结果也为整数。这样 12/24 就得出 0 而不是 0.5。在阅读那些使用这种语句的代码时，往往很难认识到这是一个整数除法。这是转换问题的根本原因——没有发现转换。

最一般的环境

转换错误最容易发生在进行隐性转换的操作系统和语言中。类型安全的语言很少有这个问题，但以前也有过。嵌入式系统和脚本语言中更少，虽然在给定的条件集里面也会发生。

症状

转换错误的最一般的症状是错误的输出格式或未预期的算术错误。除 0 最为常见，但许多系统中还有 NaN 值 (Not a Number)。对于特定的转换，如从一个标准格式到另一个标准格式，最常见的症状是由未预期的值导致的错误的输出格式。

例子

转换错误是我最熟悉的 bug，在代码中最难跟踪。在下面代码中，将要显示数据库中的无效字符，最终结果导致用户无法从某些地方登录系统。看一看问题的伪代码，判断一下。

```
Loop
  If InputString.CharacterAt(Index) < "A" OR InputString.
    CharacterAt(Index) > "Z"
    SkipIt
  Else
    OutputString := OutputString + InputString.CharacterAt(Index)
  While Index < Length(InputString)
```

这段代码的目的是只接收登录的名字，并传到数据库以寻找相应的密码。但在某些系统上则没用，为什么？是因为有转换的问题。在一些系统上，使用某些操作系统时，西班牙语中的一些元音和辅音，都无法处理。问题是它们只在某些例程中转换。每个字符都无法通过“是否字符”的测试，但却由另一个（低级的）例程传递，而且使用了与测试不同的转换方法。

这段代码的另一个问题是它假定所工作的系统中的“I”比“A”大。并不都是这样，而且字符集并非都从 A 到 Z。这样的强制性的转换检查常导致一些不可预料的错误。

一旦开发人员认识到发生了什么，他们就会删除有问题的代码并且换成系统中处理各种情况的惟一的例程。另一个转换问题无所谓。

5.4.15 硬编码长度/尺寸

在我的经历中，一个经典的 bug 源是应用程序中的硬编码数。这些所谓的“魔鬼”数通

常是没有任何关于位置和重要性的原因的线索的情况下，放在代码中的。而且这些数在整个源树上抖动，而不是应用程序中的一个常量。这很难保证在修改一个数时，修改了它的所有出现情况。

为什么硬编码的长度会导致 bug？一般地，是因为有些东西的改变使得长度和尺寸无效。因为系统的一部分，如产生数据和确定其长度的那部分，可能与另一部分之间独立地改变，这样导致 bug，因为公司中不同的组之间的交流往往是个问题，所以变化不会传播到每个需要知道的人。而且如果问题源自外部的销售者，那么在出现问题前将永远不会认识到变化的发生。

最一般的环境

硬编码长度/尺寸一般发生在客户/服务器环境，那里的数据来自外部源，或者在单机应用中。奇怪的是，Web 服务器和嵌入式系统很少有这个问题。这可能是因为嵌入式系统中升级的难度，从而避免了对输入的修改。Web 服务器因为趋向于脚本环境，故而很少有这样的问题。

症状

一个典型的症状是当系统的输入改变时系统中止。另一个明显的症状是检查代码后会发现分散在整个系统中的大量的常量。包含数字、嵌入式链表和其他类似值的源代码，应首先得到怀疑。当程序突然改变而且代码停止运行时，应该认真地查看整个源树，以发现这些值在哪里使用。

另一个症状是当遇到有相同名字而不同大小的变量名时。例如，可能某个数组有 20 个元素，而在代码的另一个地方有一个同名同类型的包含 22 个元素的数组。可能是当时的工作人员修改了其中一个的长度，却忘了修改另一个导致后来的问题。

例子

下面的代码显示了一个常见的错误。用的是 C 语言，但它暴露的底层问题可以发生在任何操作系统和任何语言上。

```
void ClearArray( int *Array )
{
    for ( int I=0; I<100; ++I )
        Array[I] = 0;
}
```

在例子中可以看出，假定代码中数组或其他结构的长度是很正常的。在这个例子中，程序员可能一直在自己的系统中将使用的数组长度设为“100”。结果在清除数组以备后来使用时，他自动地使用常量 100 来遍历数组。代码有什么问题？假定你正在使用这个看起来安全的函数，而且是用在另一个程序员编写的另一部分代码中。你的代码可能如下：

```
int anArray[ 50 ];
ClearArray( anArray );
```

当这个代码执行时，它将会毁坏 anArray 变元后的所有存储区。毁坏内存是指在不意的情况下（可能是无权的）写了一些内存区。这是因为开发人员假定数组的尺寸是不变的，而

且可以安全地作为硬编码数用在应用程序中。

值得一提的是硬编码数可能引起别的问题。例如在调试时遇到这样的代码段：

```
mStartESCR = mFirstVideoPTS - 3*3003;
```

你能知道这段代码是干什么的吗？假定即使当时知道 `mStartESCR` 和 `mFirstVideoPTS` 的用处，那么 3 和 3003 又表示什么？

如果代码写成下面这样，你可能就会明白了。

```
mStartESCR = mFirstVideoPTS - kAudioFrames*FRAMESIZE;
```

在这个例子中，程序员显然是在备份缓冲区的一些帧，给定了变量的名字后，你就更明白了，而且不一定要尝试数字的各种可能性以改变应用程序的行为。

5.4.16 版本 bug

版本 bug 是在软件的两个版本之间改变了功能和文件存储格式时，结果向后不兼容或在将来的发布中没有正确处理。最明显的例子是：文件格式因软件应用程序的永久存储机制的变化而改变。如果先前的文件格式没有正确地处理或转换，就将得到无用的数据。曾经有用的无用数据是比较令用户烦恼的事。

版本错误也出现在用户接口问题中。例如当许多应用程序从基于文本的 MS-DOS 界面转移到 MS Windows 图形用户界面时，就会有許多 bug，因为用户不能发现和使用以前版本中正常的特征。当然其中的一些 bug 只是文档错误，或对操作系统的工作原理有误解。其他错误的原因是版本的开发人员忽略了，某些按键或功能键仍然使用在初始的应用程序中。

最一般的环境

实际上任何发生变化的环境都会有版本 bug。但最一般是那些包含用独特的格式存储的用户数据的应用程序。单机程序最容易出现这个问题。在客户/服务器、Web 服务器或嵌入式系统中，测试过程中的存储问题太普遍了，不能忽视。而且，这些环境中的应用程序安装时数据发生转换是正常的。所以在现实中一般是单机系统发生这样的问题。

症状

版本问题通常在使用以前的数据或功能时，才表现出来。如果要加载软件以前版本中产生的一个文件，而它不工作或工作不正常，那就应该是遇到了一个版本问题。回归测试应得到修改，以测试以前的版本和功能，尽快看出是否有版本问题。

另一种形式的版本问题更讨厌，比特殊的格式问题还要严重。这发生在使用有多个版本的标准格式时检查不出数据的正确版本的时候。例如有一个 HTML 文件支持 HTML 3.2 标准，而编写的应用程序只能从 3.1 版加载数据。检测未来版本中元素的失败，就好比加载过去版本的失败。

发现软件中的版本问题关键要看文件中没有版本标记。如果不能看出是哪个版本的软件产生了这个文件，以及文件的格式是否已改变，那就立刻知道有一个问题。要检查所有文件格式的版本信息。

例子

一个我曾工作过的公司有一些二进制文件。就是说它们只有二进制信息，只能被产生它们的应用程序解释。一个应用程序会写出一个文件，而另一个应用程序则读这个文件并做处理，以供将来的信息和显示使用。文件包括一个头块和一系列的信息块。每个信息块包括一个“标签”，表明这个块中存有某种数据，后面是块自身的信息。所有这些都工作正常，而且运行了一段时间。一天，一个新的开发人员加入项目。他比较热情，决定加一些新的标签，显示一些工作人员在会议中讨论的信息。不幸地，他没有与部门的任何人说明新的标签，而是想用他的工作使我们惊讶。一般地，这种问题发生在有些人不知道系统要作改变时。这当然没有异常，即使过程和文件都良好地定义。

他在输出文件系统中加入了新的标签，然后系统就整个崩溃了。为什么？实际上原因很简单，而且导致了这个部门做事方式上的巨大变化。记住公司的开发人员写的标签后就是数据。并没给标签留出长度，也没分隔符来指示标签的起止。所以一个程序不会理解一个给定的标签，也无法跳过它。由于那位工作人员不知道还有其他的系统在读这些数据，因而没有修改用来读这些文件的 API；任何读这个文件的程序都无法经过这个它所不理解的标签，只能显示一个错误，指出文件损坏，然后退出。这可不是什么好的行为。

我和同事发现问题后，我们开发了一个系统，其中的文件包含一个文件头块，后面是一个小的头块，其中有版本信息数字、数据类型的标签以及长度，指出标签后有多少数据。这样，任何不理解给定的标签的应用都可以跳过它来到下一个数据位。当然也要求开发人员提供一个工具过程（程序脚本、指导手册）来升级现有的文件。一旦在版本操作过程中失败，就会发现遇到了大量的 bug。

5.4.17 不恰当地重用 bug

大量的 bug 是因为对代码或组件在新系统中不正确地重用而造成的。当代码或组件的重用方式与原来的设计者或开发者的意图不同时，重用可能就是不恰当的。例子很多，第2章就有一个比较大的例子，但是问题的基础比较复杂。

如果重用代码，就有继承原来的 bug 的危险。这是生活中一个简单的事实。最好的例子是重用一個完成简单动作的代码片断。编译器销售商早就知道这一点，为他们的系统发布了标准的库，完成标准的动作。C/C++ 有自己的标准的模板库。VB 有自己内置的组件和函数。Java 有自己的 Swing 组件和标准的库。我们重用这些标准组件时不会多想，因为它们被试验过、测试过和调试过，已经过了很多年了。但有些代码里面还是会有 bug。发现这些 bug 的最好的方法是用一种新的方式使用这些组件，这时要记住你是在尝试初始的设计者没有考虑的事情。

最一般的环境

不恰当的重用发生在支持重用的语言里，因为至少能在系统间剪切和粘贴，这说明不恰当的重用可以发生在任何地方。因为代码多数本身不依赖于所运行的环境，所以重用错误没

有一致的环境。

症状

调用代码是对重用 bug 的最简单的检测。如果重用代码的应用程序的反应与原来的不同,就可发生了重用问题。这种问题一旦重用就能检测出来。

另一个检查代码是否工作正常的方法是:重用所“借”代码的测试实例。如果代码没有测试集,就应该建一个并将其运用于原来的版本和复制的版本。

重用代码不能如愿的第一个原因是代码本身的副作用。例如,如果代码依赖系统中其他完成特定任务的东西,而新的应用程序不完成那个任务,代码就不会正常工作。在一个更低的层次上,如果新的应用程序中定义了某个给定的常量,但却不是被重用的模块所期望的,将会发现在自己的应用程序中重用代码已经导致产生了一大堆 bug。

例子

在 Microsoft Windows 操作系统世界里,每天都发生着不恰当重用的最典型的例子。当然,我指的是 Microsoft 基础类库 (MFC),它是与 Visual C++ 一起销售的。虽然 MFC 是开发 Windows 程序时一个快速跳跃的方法,但如果重用不当也会有麻烦。

我在职业生涯早期遇到的一个应用程序是一个允许人们浏览和查看文档的系统。系统使用“多文档窗口”,允许不同组的窗口同时在屏幕上显示,并允许用户在其中切换。MFC 库中有大量的处理多窗口的代码,我的同事认为最好使用现成的框架而不从零开始创建。我当然同意,不想做重复性工作。

需求严格化之后问题马上来了。对系统的许多需求,如查看文档应有多个方式(列表、搜索、文档视图),对发布的 MFC 来说是不可能的。(有趣的是,它们后来成为了系统的一部分,但我离题了)。到让系统在框架中工作时,我和同事发现我们被迫在 MFC 系统的许多内部函数之间辗转,才能得到想要的值。虽然这在当时不是大问题,但在将来会成为一个可怕的决定。

当下一版的 MFC 出现时,它支持新的 32 位操作系统,叫做 Windows 95。我们立刻决定把软件转换到新的框架,以取得立刻拥有一个产品的商业能力的优势。当然,因为我们已经不正确地重用了代码,我们发现所做的每件事,即那些依靠系统内部实现的事,都不正常了。总之,将现有的 16 位应用程序转换为 32 位系统下的应用程序所花的时间,比一开始就编写它所花的时间长多了。都是因为我们想重用代码以节省时间。

如果想重用代码并修改它,最好“克隆”它并把它当作独立的产品。不恰当的重用会浪费时间、金钱和解决问题时抓掉的头发。

5.4.18 布尔 bug

布尔 bug 是因为布尔代数的数学精度与等价的英语单词没有任何实际关系导致的。当我们说“and”时,我们实际的意思是布尔中的“or”,反之亦然。这将导致用户和编码者的误解。而且布尔错误是因为代码中 true 和 false 的含义之间的误解造成的。

让我们假定正在读一段对代码的描述。描述中说用户必须在应用程序开始时在 A 和 B 之间做出选择。在布尔代数里, A and B 表示 A 与 B 都为 true 时表达式才能为 true。但这没用。但需求中的含义是在 A 或 B 中做出选择才能继续。你不能都选。这是使用英语描述问题时潜在的缺点。

true 和 false 是如何导致问题的? 毕竟, true 就是 true, false 就是 false, 两者永不混淆, 是不是? 是的, 但这并不表示一个人的 true 就是另一个人的 true。例如, 考虑一个函数的返回值。如果一个函数返回一个布尔值, 这意味着 true 表示成功, false 表示失败吗? 不一定。许多开发人员愿意返回 0 作为成功 (或 false), 返回 1 作为失败 (或 true)。也可能会返回 -1 表示错误, 或一些其他的负数作为错误代号。

如果大家习惯一直相同, 可能在遵循这个习惯的组成员之间不会遇到什么麻烦。但是如果习惯没有备案而来了新的成员的话, 你们可能就有问题了。

最一般的环境

两种与布尔错误最相关的 bug 是需求错误和逻辑错误。它们发生于任何环境。这种问题看起来在嵌入式系统中比较可怕, 那里布尔代数用得比任何环境都多。

症状

布尔错误最明显的症状是程序做了与期望相反的事情。例如, 你可能认为你需要在其中只选择一个条目。然而, 程序会一直等你选择了多个之后才会继续。而且它还会一直告诉你只选一个!

对 true/false 问题, 通常会看到一些调试输出指出函数中有一个错误, 而调用函数好像什么也没发生一样继续执行。相当成功时调用者还可能会认为过程失败了。这通常说明你错误地解释了函数的结果。仔细看到底返回了什么, 而不要假定成功或失败的值。

例子

以下的代码是一个简单的例子, 函数返回一个与直觉相反的值, 并说明了这个值如何在剩下的程序代码中导致问题。这个例子执行一个动作并告诉用户动作是否成功。

```
Function DoSomeAction( InputNum : Integer ) : Integer
Begin
    If ( InputNum < 1 or InputNum > 10 ) then
        DoSomeAction := 0;
    Else
    Begin
        PerformTheAction( InputNum );
        DoSomeAction := NumberOfActions + 1;
        NumberOfActions := NumberOfActions + 1;
    End;
End;
```

看过函数之后, 问自己——成功时返回什么值? 失败时呢? 通过对函数声明的浏览, 你怎么就能够知道呢?

代码后面, 可能会这样做。

```
Var Value;  
Var Ret;  
Begin  
    Value := 11;  
    Ret := DoSomeAction( Value );  
    If ( Ret )  
        Println("Error in DoSomeAction");  
End;
```

如果在调试这个应用，你会开始肯定怀疑它为什么正常运行。进一步观察就会发现，错误的语句永远不会触发，尽管例程会遇到错误。为什么会这样？这是因为失败时例程不会返回非零值，它在成功时才返回非零值。在这个函数的给定的情况下布尔值不是直觉的，而且容易让使用它的程序员在代码中做出错误的假定。这里到底哪一个程序员有责任？这是一个好问题，想想吧。

5.5 为什么分类重要

如果在自己的组织中跟踪 bug 的种类，将会发现它们都属于某种可预期的模式。许多这些 bug 都可以通过消除其模式和那些导致 bug 出现的思维过程而排除。例如，如果发现问题主要来自于内存分配泄漏，就应考虑另外的能够提供“聪明的指针”的库，使得内存及分配和释放不需要猜测。如果发现错误主要来自于定时或多线程问题，就应该考虑一个管理系统，以保证写的代码是单线程的，从而将你的错误分离。（这种系统有 Microsoft 事务服务（MTS），是 Microsoft 为 COM 对象做的。）

永远也不会完全消灭 bug。几乎不会有一个不产生 bug 的程序，因为正如我们所讨论的，程序总会进化到一个包含 bug 的状态的。但是通过仔细的编码，可以减少 bug 的数量。你的职责是想出如何更加小心和吸取教训。诚实地衡量错误的惟一方法是尽量完全地对它们分类。bug 跟踪系统中的条目应该有 bug 描述、bug 类型和 bug 的严重性。这个系统应允许在调试过程中优先修改问题。更重要的是这个系统应允许进行有效的事后检查，而不只是指责。

5.6 小结

总之，我们认为 bug 可分到几个不同的领域。通过理解这些分类，我们可以预测不同的 bug 导致的问题的种类。通过对给定的项目的开发过程每个阶段的检查，才可以看出每个阶段都有什么样的 bug。

项目的阶段有：需求、设计、实现、处理、编译、配置和未来计划。每个都处于开发过程的一定时间内，而且都有某种类型的 bug，以及对该 bug 进行修复所需的耗费。

通过对在自己的开发和调试过程中遇到的 bug 进行分类，我们会有一个发现和预测 bug 的更好的方法。如果精确地预期了一个项目中将来的 bug 的种类，就将能轻易地减少测试和

调试所必需的时间。节省时间和金钱，最终带来产品的高质量。

5.7 bug 问题

一个 bug 在什么时候不算 bug？当然是当它是一个设计缺陷的时候。这个 bug 问题正好解释了现实世界应用中的环境。在一个我曾经参与的项目中，我们有一个优先的方法在硬盘上保存关于系统的信息。这个例子中数据是一组视频信息记录。硬盘开始容量是 8GB，后来增加到 20GB。

驱动器布局如下：

一开始格式化后分几个区。第一个是索引区，有 5MB，通常是首先创建的区。其余的空间分成 4 个 GB 大小的区，叫做视频区。这些区中有原始的视频数据。

第一个分区，即索引区，包含指向存储在视频数据中的每个程序的指针。每个条目包含程序信息、标题、日期和描述等等。而且每个条目包含驱动器原始视频数据的块的列表。当然存储的程序越大，用的块越多。

系统是一个消费电子设备，说明用户不需要懂太多的计算机知识。对系统错误的处理必须允许系统不管这个错误而继续运行。

现在是对 bug 的描述：许多用户对系统中的磁盘太小不满意，自己更换了大的。许多是 40GB、50GB 甚至 80GB 的。但后来这些设备看来都经历了相同的问题。它们会停止、自动重启以及丢失所有存储的程序，这显然很让用户失望。

这次调试中我们开始的假设是包含视频信息的分区可能被破坏了。有一个单独的对所有分区的一个映像，我们怀疑这个映像可能损坏了，导致无法找到分区。这个设备比较小，在启动时会检查映像，而且会创建仍然在映像里找不到的分区，这看来是个合理的假设。

但进一步观察发现，在所有记录都丢失后，分区映像仍然有效。对系统的日志文件的检查表明，设备知道自己在做什么，也检测到了错误。查看内部文档后，发现这个错误是“分区已满错误”。当遇到这个错误时，该设备会删掉有问题的区并自动重启，修复所有错误。

当然，问题是为什么分区会满，磁盘上还有足够的时间来添加记录。实际上，这个错误一般发生在有 60GB 的硬盘而只用了三分之二的空间的设备中。这就是我们发现和解决问题之前的所有信息，甚至比当时还多，因为这不是真正的 bug，而是设计缺陷。问题是：是什么导致了这样神秘的程序丢失的现象？

第6章 检测工作

既然大家已经理解了 bug 的基本属性、它们的形成过程以及它们在程序运行时所表现的特征，那么现在要做的工作就是在开发过程中如何发现 bug 了。正如前面所提到的，这包括大量的检测工作。在本章中，我们开始探讨如何利用不同的方法，排除应用程序中所表现症状的根本原因。在这个问题得以解决之后，我们就可以提出一些技术，用来构造假设，并从不同角度加以证明，这样这个问题才能够圆满解决。

在调试中最难的问题就是非重现 bug 问题。某顾客反映说：我只做了一点点事情，程序就出现了许多错误。你回到桌面上，顺着顾客提供的指令，按同样的执行顺序走一遍，却发现一切如常。这之所以使维护人员感受挫折，就是因为他们被告知，他们看到的和所做的是错的。这之所以使维护人员感到挫折的是他们不得不返回去，告诉顾客该问题不会再重现，因此不能修补。最后，使调试者感受挫折的原因是，总是难以使自己相信顾客根本不明白他们对自己的所见有何思考。总之，这种事情导致的结果是令顾客不高兴的，在你记忆的深处也会有责备挑剔的疑点，那就是这个问题确实存在，简单依赖于在 a、b、c 之前的所作所为，或者关系到客户环境，或者依赖于硬件配置，或者……好了，你已经明白了这个意思。

为了真正搞清楚顾客是对还是错，就必须知道程序是否可以和报告中所说的一样。如果报告中说程序有问题，那么你可以相信这是真的。要知道所有的程序只要是某种特定的环境，都会有问题。问题的存在取决于是什么样的环境。要想完全理解问题的范围，你必须完全理解自己应用程序的局限性。这就是我们在检测工作要做的第一步——整体调试。

6.1 整体调试

字典中对 holistic 是这样定义的：

ho•lis•tic (h-lstik) *adj.*

1. 整体主义的，与整体主义相关的

a. 强调整体的主要性和部分的互相依赖性；

b. 根据整体来考虑，不从部分分析：holistic medicine ; holistic ecology

我们把焦点集中在 holistic 强调整体作用和整体中各部门的互相依赖性上。这就是说一个具有整体作用的处理方法主要集中在系统的全局而不是各部分。例如 holistic medicine（整体医药），它所考虑的是把身体的状况作为一个整体，而不是正在发作的疾病。同样地，程序员与调试员必须明白不能治标不治本，必须集中在应用系统的整体考虑。

有趣的是，字典中还有一个调试条目。让我们快速看一下这个定义：

de-bug(d-bg) *v.tr.* de-bugged, de-bug-ging, de-bugs

1. 删除一个隐藏的电子设备，比如麦克风，来自：debug a conference room。
2. 使（如一个隐藏的麦克风）无效。
3. 搜索并排除故障元素或错误：debug a spacecraft before launch ; debug a computer program。
4. 灭昆虫，比如使用杀虫剂。

当然在我们这种情况下，我们感兴趣的是第三个定义。可以发现字典注意的是问题的排除。在这一节中，我们来讨论为什么字典中（和大多开发人员）对调试人员工作地位的理解是错误的。

既然已经理解了 holistic 的意思，那么如何把整体这个概念应用到软件程序和过程的调试工作中呢？让我们首先从一个应用程序的调试意义上的单个程序块开始考虑。在调试一个应用程序时，你经常会简单地考虑用户报告回来的问题，以及如何消除这个问题。在医学领域，类似的就是只治疗病症，而从不考虑为什么会出现该症状。

来看一个简单的例子，考虑这种情况：程序每次运行时总是在代码的某个特定点处神秘地失效。代码失效的地方根本没什么特别的。例如在 C++ 中，可能会建立如下的一个类似的成员函数：

```
void SetX ( int X)
{
    mX=X ;
}
```

程序总会在 X 赋值给成员变量 mX (mX=X) 这一行中出错。来看看这段代码，我们很难看出程序会在这个地方出现问题。奇怪的是，我们发现这样一个事实，插入一条输出语句（C 语言中使用 printf，Delphi 或 Pascal 中使用 Println，Basic 中使用 PRINT），问题不再发生了，而接下来的语句确实是重复出错的几步。这个问题算是解决了吗？

大多数有经验的程序员、管理员或是调试员可能会说问题并没有解决，尽管很少有人能说出确切的理由。这就是我们治标不治本的事实。尽管这个例子有点极端，这种只治疗症状而忽略全局的问题，在我们行业中是常有的。

6.1.1 复制和粘贴错误

关心全局系统而不只是解决手头问题的另一方面，就是我们所指的“剪切和粘贴”问题的例子。在这种情况下，程序员复制了许多代码块（作为最坏的情况）或者相同的基本代码（最好的情况）到应用程序的许多地方。等到某处出现了问题时，因为该复制的代码而造成的问题往往会传播到应用程序的其他地方。

复制和粘贴错误，尽管不是真正的复制，也会导致不同种类的问题。在下面的 BASIC 程序中，在给定的示例段中使用了某种编码技术。我们发现一个问题，当这个示例段中的问题解决以后，由于其他地方使用了与之相同的技术，所以在代码的其他地方出现了数以百计

的问题。

```
ON ERROR GOTO 100
; This is a global array of 100 integers
DIM X(100)
; Before this point, we have put something somewhere in X, now we are looking
  for it.
FOR I=0 TO 100
  IF X(I) = myValue THEN
    DoSomething()
  END IF
NEXT I

100 PRINT "DONE"
```

显而易见，如果你是一个 BASIC 程序员就会知道，这段代码的问题产生的原因就是某人很可能习惯于使用 C 或 C++ 工作。BASIC 语言中数组下标是从 1 开始的，而不是 0。当开发者使用命名常量时，比如 `kStartloopIdx` 和 `kEndloopIdx`，这个问题会更严重，因为调试员必须找到或知道运行和调试时它们的值，因为程序员从来不测试这个循环，所以也不会发现它有问题。在多数 BASIC 变量中使用下标为 0 的数组元素，将会造成程序异常。因为在程序循环体开始前要处理错误，所以程序继续执行 `PRINT` 语句。在用户看来就好像程序做了该做的事。然而，该循环从来没有被执行，用户也找不到他们要找的东西。这种“沉默的”错误蔓延到整个程序。如果我们简单地看待这种初始化问题（嘿！从来找不到我的值！），我们就会忽略这样的事实：这个问题会到处存在。

顺便说一下，这个例子阐述了调试和维护的整体方法的另一个方面。一旦发现写这段代码的程序员使用了 0 作为数组索引下标，就可以通过搜索代码来找出同样的问题做出纠正。然而，这样就完事了吗？没有。你可能会在前面的代码中发现其他更严重的问题。在这种情况下，根本就没有真正地处理错误！实际上，把错误隐藏起来，让程序表面看起来正确，反而会使错误更加严重。关于“整体系统”方法的另外两个方面：全局变量和副作用，会表明这个方法为什么很必要。

6.1.2 全局变量

除了 C 和 Java 外，大部分语言都支持全局变量。全局变量不属于哪一个范围，也不会某个类或结构中存在。它们是程序员的祸根，因为我们无法控制如何访问它们或如何改变它们。在使用全局变量时，必须将程序的控制权提供给开发者，让他们有选择地在其代码中使用该变量。

为了充分理解含有全局变量的系统中全局变量如何作用于程序，就需要确认修改变量的每一处。而且，在每一处，都需要弄清楚该变量或其他变量得以被修改的环境。这样做的工作量比较大。可以不使用全局变量，来使程序变得更容易理解和调试。也可以提供访问函数、方法或对象来处理这些变量。考虑下面的 C 代码：

```
int globalInt = 0;
int TestFunction()
{
    switch ( globalInt )
    {
        case 0:
            Initialize();
            break;
        case 1:
            GetInput();
            break;
        case 2:
            Finalize();
            break;
    }
}
```

显而易见，写这段代码的程序员依靠全局变量 `globalInt` 来告知他处在执行某一任务的什么阶段。为什么会存在问题呢？是这样的，如果另外一个程序员见到了变量 `globalInt`，认为由于在某种情况下没有初始化，所以他可以给它赋值为 1 从而创建一个“捷径”，这样也就很快产生了一个 bug。如果有新的成员变量或其他初始化数据被加入到 `Initialize()` 函数中的初始化部分时，那么也就马上等于在系统中引入了一个新的 bug。

提示：在应用程序中保持一些能清除有用的静态和全局变量的函数或方法。

全局变量很有用。但是，一般地，它们是有待发作的 bug。当程序中含有某种全局变量时，就需要使用一种有用的调试技术，维持某种函数或方法，以把这些变量打印出来，这样就很容易每隔一定的时间间隔检查它们。它们也可以很方便地用来开始查看什么时候、什么局部的变化影响程序的其他区域。

提示：当局部变化影响程序其他部分时，检查全局变量。

6.1.3 副作用

如果说全局变量是令调试员第一个最头痛的东西的话，那么副作用就是第二个。副作用可以是任何动作，也可以是一条语句或是一个函数，总之是其行为非期望的。看一个简单例子，一个用来统计字符串中空格个数的函数可能执行别的工作，比如增加了该子字符串的关系数量。这种副作用即使已经说明，也会在阅读代码查找问题时，在探索问题上产生很大的麻烦。

为了理解副作用是什么样子，我们来考虑下面的 Java 代码程序，这个类方法的作用是转化并返回一个字符串的大写形式。

```
public string UpperCaseString( string InString ){string s = new string;
    for ( int i=0; i<InString.length(); ++i )
        if ( !sLower( InString.getAt(i) ) )
            s += ToUpper( InString.getAt(i);
```

```

        else
            s += InString.getAt(I);

        InternalReferenceCount++;

        return s;
    }

```

这个函数好像做了所期望它完成的大部分任务。它可以把输入的字符串转化为输出的字符串，返回该字符串的一个副本。源字符串好像没有被修改。然而，请注意对 `InternalReferenceCount` 递增的调用。这条语句的作用清楚吗？不完全清楚，确实如此。这个方法被调用多少次，它就会简单地递增多少，或者说它就会跟踪这个对象的引用次数。如果不知道写这段代码时在思考什么，那么它的作用就很难说了。

揭开这种增加发生的原因的工作就是调试者的推测工作，这可能不重要，也许只是用于跟踪这个对象初始化代码中的变量。也可能非常重要，可能会代表一个对象的引用次数，这个对象从来不出范围，而是一个内存漏洞。总之，这是一个副作用，在它出现的函数的范围内令人惊讶。这种副作用是良性的。下面的代码中包含一个十分糟糕的副作用，它实际上是标准 C 库的一部分（这个说明不是库里的，只是它的简化）。

```

// strdup: make a copy of the input string and return it.
char *strdup( const char *instring )
{
    // Allow room for the trailing null
    char *temp = new char[ strlen(instring)+1];

    // Copy all including trailing null
    memcpy( temp, instring, strlen(instring)+1);

    return temp;
}

```

这个函数复制了一个字符串，然后返回它。这段代码中好像是没有副作用，只是根据函数说明那样来执行任务。这么简单的函数哪里可能会出错呢？要理解这一点，就必须从用法的观点来看它。程序员调用这个函数，以某种方式复制一个输入字符串，比如：

```

char *s = strdup("This is a test");
FunctionThatModifiesString(s);
PrintString(s);
return 0;

```

这会发生什么情况呢？程序员使用这个函数也正如它的用法一样，是不是呢？他复制了输入串，传送给某个函数修改，然后输出了结果。他做完了工作，通过返回给调用程序一个代码从而结束了函数调用，表明在函数中没有错误。如果用标准 C 库的背景知识来简单看待这个函数，也看不出什么毛病。然而，它确实存在一个相当严重的问题。函数 `strdup` 分配得一块内存，用于复制字符串，然后 `strdup` 的调用程序获得一个返回的指针。从这个时刻起，该已分配的内存块就一直起着调用函数的作用。在函数调用中，没什么地方声明释放

该内存块。我们创建了一个内存漏洞，只是不表现在应用程序的源代码中而已。这种函数对于调试和修补来说有时简直就像一场噩梦。

提示：在代码中观察和说明无法预料的或隐藏的副作用。

6.1.4 观察意外的消息或结果

生意上最容易使用的技巧之一，是仅仅观察意外的结果。在运行程序时，应该观察那些被显示出来的古怪消息。例如当试图保存所做工作时，可能已经习惯了那种在应用程序中警告你不要覆盖文件的对话框。在给定的情况下，如果当你知道该文件存在时，而对话框却没有出现，你可能会立即觉得奇怪。或许对话框出现了，但显示的消息不正确。假如不是询问你是否想要覆盖文件“XYZ”，而是出现一个对话框问你是否想要覆盖文件“*&^(*&^。”你不可能使用这么一个文件名称，所以很容易知道程序出了问题。仔细观察诸如此类的情况有助于你以后调试程序。

对于应用程序中的大多数问题，用户还是不能真正地诊断的。他们只会观察发现一些看起来不太正确的问题，然后向你报告。经常地，当他们发现这样的事情并告诉你时，问题的细节却在交流中失去了。因此，拥有一个可重现的实例是很重要的，在这种实例中可以观察程序运行时的反馈情况。而且，知道程序是如何按预期目的地工作是很重要的。因此，可以很快知道程序是不是工作正确。

看一个用户报告的程序 bug 的简单例子。用户写道：“当我试图把节目录入到磁盘时，程序却拒绝这样做。我录制节目一整天了，为什么突然不让我录制了呢？”这种情况，节目出问题就好似是一个 VCR。调查表明，用户一整天不停地从电视上录制节目。突然，程序拒绝录制了。这是一个 bug 吗？回答却令人惊讶：“也许”。在这种情况下，节目录制事件达到了它预定的极限 50。简单检查一下这个应用程序的源代码，会发现，这是一个预先硬定义的录制事件发生次数的极限。这是一个 bug 吗？不是。需求并不要求这么一个极限，但需求在这样的问题上往往是含糊的。如果通过告知用户有这么一个极限，来解决刚才的问题，那么用户一定会很明白的。所以，为什么却回答说“也许”呢？因为没有反馈给用户以表现录制事件被拒绝的理由。如果用户不知道这么个限制，你也不告诉他，这将成为一个没有得到解释的问题。用户会继续尝试录制，然后失败；然后他会继续抱怨。

现在，让我们看一个相似的例子，说明这个相同的基本问题，阐述一个更严重的 bug。当观察从电视录制节目到磁盘的程序的输出日志信息时，你会注意到有这么一行：“错误：不能为事件分配额外空间”。

电视节目继续播放，好像是录制到了硬盘上似的。但是，当你把节目倒回去时，节目却停止了，很长时间也放映不出来。不停地试图播放，结果却是一样的；一定时间后，也没有什么显示。回过头来看看观察结果，就可以猜到问题的实际原因：磁盘空间用完了，所以电影停止录制。这是不是一个真正的 bug，完全依赖于应用程序的需求定义。在这个例子中，并不是磁盘空间不够的问题，而是错误没有得到正确处理的事实。要解决这个问题，要么将

其他的节目从磁盘中删除，为当前的录制释放空间；要么应用程序首先要检查对于当前的录制工作是否有足够的空间。无论用户（和程序员）第一次运行程序，还是运行了千百次，都不应当对它的行为感到惊奇。

6.1.5 跟踪诊断显示

在为一个应用程序正确制作诊断程序时，仔细观察这些诊断是很重要的，无论人工的或是自动的。如果不注意这些诊断显示，那么它们也没什么大的用处。它们的重要性就好像旧式轿车里的油灯。人们理所当然地认为这样一个事实：它们是存在的。“哦，是的，它总是燃着的，我认为它就是动力燃料。”呀！真有点奇怪！如果这样的灯不能指示问题，那为什么制造商要提供它呢？同样可以说，在一个应用程序运行过程中弹出一个对话框，如果不指示一个严重的磁盘错误，你大可以忽略它。但是，为什么你想知道程序是不是有问题呢？有两种情况，如果没有严重的磁盘错误，程序包含有一个 bug，应当被修正。如果有严重的磁盘错误，你可能要使机器工作，并不只是程序，就如履薄冰。无论哪一种情况，忽略诊断都会是一个很糟糕的想法。

对用户来讲，诊断并不总是可见的，有时只有程序员才可看到。例如系统日志、调试输出以及其他的日志信息，用于诊断问题发现原因时可以隐藏在“幕后”。有些信息用于跟踪可能发生的问题，但在事件发生时对用户没有什么用处。这样的信息经常在诊断输出中出现，应该一丝不苟地进行检查。这样在问题形成危机之前即可当场诊断。

6.2 调试方法

既然你明白了一些用于调试应用程序的技术，下面将着重研究一些应用于调试应用程序的一般化方法。每种方法都有其优点和弱点，使用每种方法也存在合适的时机和不合适的时机。经常由你自己来决定，在职业或职位上根据给定的情况选择一种比较合适的方法使用。

6.2.1 科学方法

在真正的工程领域，常使用科学方法来解决复杂的问题。该方法描述很简单：先做出假设，提供证据证明或反驳假设，重复这样做，直到找到一种假设适合所有可知证据，且不与任何观察矛盾。

在调试领域中，科学方法真正有用于重现问题的情况。在问题发生前、发生时以及发生后都要仔细观察问题，注意问题的一切症状。而且，在问题发生期间要收集一切诊断显示。

在手头有了证据之后，就可以立即做出各种可能的假设，来解释问题出现的原因。比如，可以假设在搜索操作中耗尽磁盘空间，从而导致程序失效。看看手头所拥有的证据。磁盘空间有多大？搜索到了多少数据？这样的搜索创建了多少临时文件？在问题发生时要仔细考虑所做的观察。听到磁盘不停地转吗？磁盘指示灯闪亮不止吗？如果系统支持警报指示，有没

有看到警告对话框，指示说磁盘空间不足了呢？

看看观察结果是不是合乎假设。一般情况下，有些是支持假设的，因为开始时就使用它们来生成了该假设。有没有使用前的证据可作为假设的弥补？要仔细检查在观察过程中记下的每一条证据。某些症状是不是直接与问题相矛盾呢？看看搜索的例子，能够发现这样一条证据吗？就是程序与磁盘完成后，开始在内存中排序，这可望反驳磁盘空间耗尽的论断。如果能找到至少一条决定性好的证据来反驳原有假设，就需要后退一步，建立一个新的假设了。也有可能一个假设的层次结构，需要以一个一般假设开始集中于问题的一点，然后“演练”到一个具体的问题。

某种意义上讲，还是希望能找到一个不与任何有效证据相矛盾的假设。当然，这也不能说明理论就是正确的。在这一点上，它是一条理论。如果理论正确，就可以建立关于假设问题的解决方法来解决此类问题。要做到这一点，就要从假设的观点角度来考虑问题。如果理论正确，就提供解决该类问题的方法。从这一点上看，还需要写出实际代码或者执行应用程序来使你的解决方法可行。当代码或过程比较恰当，如果问题不再出现时，就说明已经解决了问题，是不是？不是，还没有解决问题——至少说没有完全解决。要理解问题是否真正得到了解决，还需要完成一些其他的任务来修正 bug 补丁。这些任务会在第9章中讨论。

6.2.2 直觉

直觉对熟悉代码的调试者来讲，可能是使用得最普遍的调试方法。直觉简单地说就是从观察到的症状中，可“得知”问题可能存在于什么地方。尽管直觉并不比彻底了解代码和系统整体工作方式好，但对于那些不了解系统的人来说，还是比较有用的。

直觉并非一般化调试技术的一个好的标志。为了有效地使用它，就必须对系统有个整体认识。如果你是系统的底层开发者，这项要求并不算难。但是对于不是长期的系统开发人员来说，直觉只能依靠相似系统的相关知识。

概括地讲，直觉能够让我们看到问题的症状；能够迅速地找到出现问题的代码区域，它还要求有足够知识来理解应用程序中的算法，并能够对产生症状的错误进行分类。

为了更好地理解直觉方法，让我们来看一个简单例子。假设你在运行一个程序，这个程序可提供硬件设计（比如吸盘）的实时数值。程序开始运行比较顺利，一会儿后就开始返回假的值给应用程序用户。作为一个中间介入的开发者，你可能要钻研代码，了解算法的结构和运行情况。另一方面，对于原始程序的开发者来讲，他们可能会看输出结果，就会发现与硬件设备的连接断了且检测不到其状态，返回的数字只是侦听端口的线噪声。

没有教授直觉的方法。它主要来自对代码的彻底理解，以及对应用程序工作方式的熟悉。单独把它提出来的目的，就是更好地理解人们如何使用这些知识来做各种事情。

6.2.3 思维跳跃

思维跳跃可以简单地说是问题的有理由的猜测，也可能是当今第二种最普遍的调试方

法了。开发者不需要检查所有的症状，而是从中选取一些，对可能是什么问题做个迅速的猜测。然后，开发者直接修改代码来测试理论，所修改的代码主要是他认为存在问题的部分，然后观察问题是否得以解决。

思维跳跃常指作“暗创”。尽管思维跳跃可以发现并修补问题，但也很有可能误入歧途，搞得越来越糊涂。思维跳跃的一般作用直接与你所知道的系统信息的多少成正比。如果你完全了解系统了，那么思维跳跃也就接近直觉了。另一方面，如果你几乎不了解系统，思想跳跃则很接近 WAG (Scientific Wild A**Guessing)。

调试最差的可能情况，就是第一次试图使用思维跳跃。如果是这种情况，你可能要忽略科学方法，进一步使用思维跳跃。于是你跳跃次数越多，成功的机会就越少；尽管有时你可能认为自己快要成功了。而科学方法成功的概率比较大。所以，一般地，你要三思而行。

6.2.4 诊断

纯粹的诊断有时称作“提前推敲测试”，是对应用程序中可能发生的问题种类的提前预言和追踪，所以最终程序员能够最终找到问题并修补问题。尽管在工作中很少使用诊断调试作为惟一的调试形式，但这种方法已经广为熟知。诊断是我们所拥有的最好工具，因为它们由问题发生时收集的事件和统计数字组成，而不只是来自观察和后来的改造。然而，诊断不是调试的万能药。当诊断显示告诉你发生了什么错，在什么地方出错时，它并没有首先告诉你产生错误的原因。比如，得知一个底层例程接收到了一个空指针，这是一个好的关于系统产生错误的原因，但这却不能帮助你理解为什么指针为空。在任何时候都要遵循假定的模型，即使是你得到了极好的诊断信息。

如果这是真的，那么为什么调试者不使用诊断调试作为惟一的调试形式呢？原因十分简单。要识别程序中的每个可能的错误条件输出和每个变量的值，那是不可能的。在第 10 章，我们会看到许多诊断数据元素，那些都是需要监视的；但是现在，你要接受限制的事实。而且，每一个保存的诊断要占用磁盘空间，而磁盘空间又是一个十分有限的资源。如果系统没有太多磁盘空间，比如嵌入式系统，则境况会更糟糕。如果不知道程序生成日志文件的原因，用户则很有可能会删除它们，以图收回这些资源。还有，诊断会利用有限的时间来显示和写入到磁盘上。这段过程所使用的时间对应用程序会有严重的影响。你不想耗尽用户的整个磁盘空间，而让用户使用一个指甲大小的程序。即使这种应用程序没有 bug，也不会有用户去使用。

6.3 商业技巧

对每种调试方法，每项调试技术，都有一些技巧，可以用来改善生活。在这里我们会介绍一些。

6.3.1 内嵌式调试器

在应用程序中嵌入一个调试系统是可能的，也是必要的。这样的系统同使用外部调试器不同；它同外部程序相比，既有优点也有不足。调试器应该在系统执行时引入并运行，允许在应用程序运行时进行调试。在下一章中我们会看到内部调试器相对于外部调试器的另一个更大优点，也就是内部调试器不改变应用程序的运行条件。

要实现一个内部调试器，需要创建一个内部代码的专门化的方法。这往往通过使用密码机制来实现，尽管通过使用登录方法很容易修改它。无论哪种情况，最好不要授权临时用户使用该调试器，因为他们很可能会搞糊涂，甚至无意地破坏了数据。

在设置了调试器界面之后，就需要决定要查看什么信息。不像普通的外部调试程序，内部调试器只完成程序编入其中的任务。要使它更一般化，就要求利用用户的计算机比程序平时更多的资源。由于这个原因，内部调试器需要跟踪预定义的变量。为了具体说明某些变量，可以“注册”希望查看的变量，或者通过伴随调试器的程序在运行时跟踪例程。这就可使得决定想看什么就看什么，想什么时候就什么时候看。这个方法有一点不利之处就是，如果后来决定获取更多的、没有提前定义的变量的信息，它就无能为力了。但是，你可以首先退回前面，添加这些变量，再返回应用程序。

内部调试器在应用程序初始化设计期间必须被考虑。在程序完成之后添加一个调试系统往往是件十分困难的事，只是在检查系统时，并且有时间正确添加调试系统时，才会考虑在程序中添加这样的调试系统。一个写得不好的调试系统，实际上是弊大于利的。

6.3.2 日志对象

到目前为止，最有用的调试诊断是了解一个程序信息的某个时刻做了些什么。要做到这一点，输出日志采用跟踪应用程序所使用的信息。日志对象可以很容易地用于各个应用程序中，这也只是添加它们在设计 and 开发中的开销而已。日志对象简单地说就是一个对象（一个 C++ 类、Java 类或 Delphi 类，或者是 VB 或 ASP 中的一个元素），可控制包含应用程序运行的特定信息的日志。

重要的是要考虑以下问题：什么时间写入日志对象（或者使用一个已写好的），在运行时可否开关日志，用户是否可控制日志内容，该对象是否可用于类似系统中断处理程序的组件中，在什么地点写数据是临界的等等。

警告：创建日志对象时，要考虑它们在中断或时间临界区的使用。

不要将日志对象与下一小节将讨论的跟踪对象相混淆。日志对象是一般目的的对象，应当能够通过用户命令或进程定义进入系统的入口点，然后把该命令或进程名字传递给调用它的下一级对象。在使用跟踪对象时，将会看到调用所经过的每一级以及与对象调用匹配的代码所专用的分支。

6.3.3 跟踪对象

跟踪对象就是在程序运行时可用于跟踪程序流的对象。与日志对象不同，跟踪对象试图在程序运行时模拟程序流。要充分利用跟踪对象，你就应当能够通过一个用户命令或进程，定义一个系统的入口点，然后把该用户命令或进程的名称，传递给被调用的底层对象。当你使用对象时，你要看一下传递调用的每一层次，通过与对象调用的代码进行匹配，看看采用的是哪一支。

跟踪对象有利于程序调试的整体方法，可看到系统正在做什么，而不需要坐下来劳苦地跟踪代码的每条命令或进程。尽管跟踪对象最好是在程序执行初始化时运行，但也可以在此之后很容易地添加。所必需的就是在代码的每一行新建函数方法或对象中放置合适的对象。看个例子，C++中有一个跟踪对象，它的作用如下：

```
class Trace
{
private:
    std::string mFunctionName;
    Trace( const char *funcName)
    {
        mFunctionName = funcName;
        printf("Entering %s", funcName);
    };
    ~Trace(void)
    {
        printf("Exiting function %s", mFunctionName.c_str() );
    };
};
```

然后将这个简单类以某种方式嵌入到代码中：

```
int function1(void)
{
    Trace trace("Function1");
    // Do something
}

int Function2( void )
{
    Trace trace("Function2");
    function1();
}

int Function3(void)
{
    Trace trace("Function3");
    Function2();
}

int main(void)
```

```
{  
    Function3();  
}
```

单个程序运行输出如下:

```
Entering function Function3  
Entering function Function2  
Entering function Function1  
Exiting function Function1  
Exiting function Function2  
Exiting function Function3
```

通过观察输出结果和主函数 `main()`, 可以很容易地跟踪程序工作。显而易见, 这个简单设计的例子很平常, 但已足够观察其代码本身了。在使用 C++、Delphi 或 Java 语言编程时, 会很容易迷失在集成和重载虚拟函数的层次中, 不知道现在正处于什么调用中。跟踪对象可以有助于快速解决这个问题。

提示: 在编写代码时, 添加跟踪对象。

6.3.4 隐藏的诊断屏

使用隐藏的诊断屏是在运行时调试程序的另一种有用的技术, 尤其在一个产品环境中。隐藏的诊断屏和内嵌的调试器很相似, 但是比较简单, 只提供一两张列表, 列出程序在程序运行想要跟踪的所有主要的症状诊断, 表中的某些信息可能包括程序运行的环境、可用内存的大小、可用磁盘空间的数量以及应用程序各种组件的版本信息。

诊断屏相对于内嵌调试器的优点, 就是它们的尺寸以及可在事实之后添加这种情况。如果你想在产生环境中程序运行时得到诊断信息, 又没有你所讨厌去修改的底层代码, 那么最好的方法就是勾画出想要跟踪的点滴, 然后添加隐藏的诊断屏到单独的位置。这些屏一般情况下对用户是不可见的, 可以通过使用某些奇怪的按键组合, 或者特定的用户/密码组来标识它们, 一般用户可能无权使用。

提示: 添加只能由专门的键盘命令或用户登录机制访问的隐藏的诊断屏。

关于诊断屏, 最后要注意一点: 它们可用于历史数据, 在调试时可能需要这种用法。例如, 你可能要跟踪用户创建的新文件的数量, 与其打开的现存文件数作比较。这对于一些 bug 产生原因的追踪也是有用的。比如, 若观察到用户已经打开了一个现存文件, 也就没有必要在该用户的创建例程中寻找 bug 了。而且, 通过跟踪使用的样式, 就可以决定在什么地方花费精力改进和提高应用程序最好。噢! 作为最后一条注意, 如果注意到用户总是在访问帮助屏, 那就值得看看这个应用程序的可用性了。

6.3.5 为以后的程序运行保存 bug 数据

对在应用程序中产生 bug 的数据进行跟踪, 是另外一个技巧。程序员们最奇怪的特征之一, 是总会在许多地方犯下相同的错误。同样地, 只要他们犯了一次错误, 他们就很容易在

后面的代码设计中犯同样的错误。如果跟踪第一次产生某种类型 bug 的数据和进程，就可以在以后的回归测试和新特点测试中使用这项数据。

考虑一个例子，假定有一个接受日期的输入屏。程序里这个屏的作用是检查“有效”日期，不允许无效的月份、无效的天数（指那些超过 1~31 范围的）以及无效的年份如负值。然而，程序员却不能检查闰年。你可能需要注意这一点，为未来工作活动的一个测试实例寻找闰年项。这将节省花费在探索问题上的大量精力和时间。

可能值得一提的是在发现一个问题实例时，比如闰年问题，就应当把完成日期修正的代码调到一个单独的位置。这样，一旦问题解决了，也就永远解决了。而且，如果想获得应用程序域的所有变更的全局域时，只需要跟踪很少的代码就可以了。

6.4 可重现实例

到目前为止，试图解决一个 bug 时，收集证据的最好方法就是拥有一个可重现实例，来举例说明正在试图修正的难题。让我们来看一些创建和寻找可重现实例的问题。

6.4.1 测试实例

要想能够真正地创建一个可重现实例，必须首先拥有一个阐述问题的测试实例。测试实例一般由软件测试人员编写，也可由开发人员和调试人员创建。要能够很容易地进入创建 bug 症状的进程内部，这一点是很重要的；所以要确定测试实例写得比较好，容易上手。

看一个例子，测试实例表达为“启动程序，做事情直到屏幕变蓝”，就不可能产生任何可用的可重现实例。到底需要做哪类“事情”？做多长时间？还需要做指定的“事情”吗？所有这些都是值得问的问题。

与第一个例子比较，看下面的测试实例。“启动程序；选择文件I打开，从目录\\testcases share 中选择文件 file001a.dat。点击 OK，移动到该文件的 33 行，键入 Hello world。仔细地观察程序状态栏，注意列指示器随着键入而移动。”这个命令会产生一条 bug 报告：“当运行测试实例<×××>时，列指示器没有正确指示你所在的列。”显然，这是一个很好的说明，可以让调试器迅速且容易地重现问题，从而有希望解决问题。

跟踪测试实例是很重要的，这样开发人员能够看出产生问题的事件类别。同跟踪 bug 一样，跟踪测试实例可提高软件质量，因为在设计和调试过程中，可以不断地解决问题。

6.4.2 数据依赖

最坏的 bug 是那种依赖于特定输入数据表单的 bug。因为只有特定的进程跟在特定的数据集之后，才能重现 bug，所以要解决这些问题往往非常困难。数据依赖错误也可以被修正，因为该数据集不同于重现它们。同寻找可重现实例一样，寻找可重现数据集也常常是一个严肃的问题，在调试问题中也会有很大的帮助。

发现并报告错误时，要首先跟踪用于创建该错误的数据，这是很重要的。如果是你为该应用程序创建的数据，那么做到这一点是很容易的。但是如果数据来自外部资源，比如一个实时数据流，那就比较困难了。在这些情况下，使用大量的日志和跟踪，可有助于正确判断什么类型的数据产生了问题。

一般地，数据依赖问题出现在使用程序中算法或极限的边界条件时。发现这些边界条件的最好的方法之一，是充分理解系统中的极限。例如，一个系统可能包含入口数量的专门的极限，它可能存在于文件中。知道了这个极限，就可以很容易地测试和修正当文件为空或入口太多时产生的问题。要知道一些应用程序中的硬性限制，比如“最多 50 个文件”或“只允许介于 0.4 和 1.2 之间的信号。”

6.4.3 从根本原因中分离症状

在调试一个应用程序时，一般会看重所观察到的症状，当症状消失时就称为问题解决了。不幸的是，常常有这种情况，即真正的 bug 往往与其表现出来的症状无关。关于这种情况最清楚的例子就是：在许多语言中常发生的指针丢失问题。

指针丢失问题常发生在没有对指针初始化的时候，对那些不使用带有指针的编程语言的人来说，指针是一个变量，指向某特定的内存块。在使用指针的语言中，不能将变量指向给定的内存块，导致结果是指针内存中的随机块。这样的变量被称为未初始化的。当这样的指针在内存临界区中指向自己时，就会产生不可预料的程序结果，比如冲突。在程序中添加代码（或添加新变量），可以改变指针被指向的地方，并且好像是修复了问题。当然，这并没有修复任何东西，只是把问题移到了一个新的目前尚未被发现的地方。

那么怎样才能证实确实修复了错误呢？让我们回头看看假设和解决方法的思想。如果你创立了一个假说，说明问题是一段糟糕的代码，你修复该段代码，那么从理论上讲问题应该是消除了。然而，如果在代码中再添加两条调试语句，来看看发生了什么（日志语句）以及问题是否消除，很有可能仍然会发现根本没有修复任何东西。只有通过确定问题，才能够证实你已经发现并修复了根本原因。

6.4.4 收集观察结果

观察产品的使用情况，是决定错误产生的时机和场所的最好方法之一。首先，这是一个获知程序如何从外部工作的优秀方法，并且可以更好地理解为什么程序以内部方式进行工作。第二，通过观察程序正常工作的时机，就可以更容易地预料程序不正常工作的时机。

在收集关于程序运行的观察结果时，写下收集时采取的步骤，这是更重要的。收集观察结果需要通过测试实例，在思想上已很少面向目标问题。如果可能，不但要记下正在做什么，还要记下输入了什么数据到程序中。然后使用这些观察结果来跟踪代码，用所输入的数据来看结果是什么。如果发现显示反常或者程序没有按期望地进行，就要在查看代码之前收集大量的关于其用法的信息。同样地，发生问题的观察区域可使你明白产生问题的数据种类通过

的原因。比如，虽然在应用程序算法内部有了问题，但却很容易通过不接受那些产生问题的数据入口，从而有效地得到修复。

在说明观察结果是对或错方面，保持一个项目记录本记下系统应该如何从外部工作，将是一项有价值的工作。你可以拿着项目记录本，与需求定义（如果有的话）作比较，或者与系统设计（如果有的话）作比较，看看每一步应该发生什么。正如在本书前面看到的，并不是所有不按预期方式工作的东西都是 bug。通过观察结果鉴别 bug，在调试的检测工作中是一种强有力的手段。

6.4.5 统计/公理

对应用程序保持统计和公理数据简直就是两刃刀。一方面，可以由此知道是不是在克服应用程序中的问题，更能够识别最好进行重检或重写的程序段。另一方面，公理和统计常常用于政治性的激发运动，来表示说明一件事或另一件事依赖于掌握统计的人。正如一句古语所说：“水能载舟，亦能覆舟”。这并不是说，我们有很好的理由不主张统计和公理了，而且，要注意一点：在调试过程中所有的工作并不总是技术问题。

所以，在程序调试时，什么类型的统计和公理有用呢？重要的是，要经常跟踪最后更新的模块，以及每个模块改变的次数。当错误突然出现时，问题很可能发生在最近更新上。至少说该变更值得怀疑。尽管 bug 本身可能不在那个模块中，但该变更影响到了另一个模块，从而引起了那里的问题。

另一个有用的公理就是：要考虑任何给定模块中代码的复杂性。尽管并不是完全由于复杂性才出现问题的，但是在 bug 的复杂性与可能性之间有一定的联系。看个例子，考虑下面两段 Visual Basic 代码：

```
FOR I = 1 TO 100
    CALLPROC( I, I*I )
NEXT I
```

以及

```
FOR I = 1 TO 100
    FOR J = 1 TO 100
        X = X + SQR(Y) + I*X + X * (I*1)
        Z = Y + X
        DIF = Z - (X+R)
    NEXT J
NEXT I
```

即使你不知道这两段代码是做什么的，你会认为哪一段更可能包含 bug 呢？第二段要比第一段的可能性大一些，这一点也不奇怪。衡量复杂性是件十分困难的工作，但已存在有许多可用的方法了。McCabe 复杂性值是其中的一个好方法。它可使用于评估代码中功能集的复杂程度。函数中的路径条数，可用于计算该函数的复杂性量度。

公理另一个有用的地方，就是在跟踪代码的问题数量过程中。随着代码的完成，在项目

的生存同期内如果每天都测量修复错误的数量，就会发现这类似于一条贝尔曲线，也就是说在开始时修复的错误比较少，因为开发还在进行，测试刚刚开始。然后，发现并修复的错误急剧上升，由于测试过程已经全面展开，在代码最后完成后，调试工作占了上风。最后，由于产品的质量大大提高，发现和修复的 bug 数也渐渐少了。通过研究一些工程以及一些代表工程的曲线，可以更好地了解代码中预期的 bug 数量，以及可能花费多长时间修复它们。

最后一个值得遵守的公理，也是必须小心对待的一个，就是每位开发者跟踪 bug 数量。通过跟踪产生错误的模块，以及在那些模块中出现的错误类型，可把每个错误分配给开发相应模块的开发者去处理。如果知道开发者 A 很少在代码中出错，而 B 却会产生 10% 以上的错误，那么就可以知道应该在什么地方开始测试和调试了。另外，如果发现开发者 A 有 45% 的错误出现在内存管理中，而观察到的错误症状是与内存管理相关，就会很好地想到要到什么地方去查找问题了。但是要记住，没有人喜欢别人指出自己的问题。这个公理只应该由调试器使用，而不是管理器。

最后的一点提出了关于调试器相对于程序开发的一个有趣的观察结果，在调试程序时，首先应该检查一下自己。谁出了问题或者是谁修复了问题并不是重要的，主要的是找到了问题并修复了问题。在调试问题时切忌说三道四。

6.5 小结

总而言之，这一章我们学习了如何在调试过程中使用检测技巧。如果不考虑系统的等级，科学方法是用于调试程序的极其重要的方法。从微小的控制台程序到大规模的分布式系统，科学方法会有效地发现和修复存在于项目中的问题。

在学习了发现问题之后，我们找出解决方法的许多步骤。首先必须观察问题的症状，仔细了解系统正在做什么。然后，根据问题提出假说，仔细考虑所有的观察结果，使假说不与任何已观察到的症状相矛盾。第三步，测试假说，不但要看看它是不是同观察到的症状相违背，还要看看如果使用其他的输入设定，会不会有其他的症状出现。如果假说满足可得到的认证必要条件，那么第四步提出一种解决方法，解决指定的假说中的问题。第五步，验证该方法是否解决了问题且没有出现新问题。

希望读者已经理解了实际问题的重要性，而不只是简单地对付应用程序中观察到的症状。再强调一次，简单地给“受伤”的应用程序加上“绷带”，从长远来看只会使问题更加严重。

6.6 bug 问题

为了更好地学习本章中检测工作的思想，让我们来看一个例子，使用检测工作方法正确判断应用程序在最终用户端出现了什么问题。

应用程序是你的公司开发的，当研究它的 bug 报告时，遇到了一点古怪的东西。用户报告说在每一天的相同时间程序出现故障一次。而且，用户还说以前程序是这样的，后来神奇般地“自动修复”了，可现在同样的问题又出现了。

在追踪系统“自动修复”与现在时间的差别时，你发现只有三点不同。第一，程序的修复是由于使用了新的打印机，安装了一套该打印机的一套最新版本的驱动程序。第二，错误修复了是因为自动备份系统，以前它一直不能正确工作，现在工作良好了。最后，对内存管理系统进行了改动，使得在其空闲期内碎片收集做得比较好了。

假设程序包含有日志信息，指导在一定时间内发生了什么主要事件。进一步假设，你可以完全访问程序源代码，可以检查代码的任何一般问题。最后，假设你在实验中的“重现系统”不能够重现该问题。请你决定要发现该问题你应采取哪些步骤。考虑各种可能的问题假说，看看你如何修复这个问题。

第 7 章 调试工具及其使用时机

第一次调试程序时，有时就好像很无助。可能还会感到对于某个特殊的应用程序没有工具可用，没有现成的文档对正在查找并试图修复的 bug 进行说明。然而，根据面临的问题和工作的条件，还是有很多工具可用的。在这一章中，我们分析一些可用的工具，决定什么时候使用它们或者使用更重要的工具，以及建议什么时候不使用它们。

理解工具的局限性，同理解它们的好处几乎是一样的重要。大量的文档描述了调试工具的使用方法。几乎没有文档来描述在使用这些工具时可能面临的问题，或者说明工具有时却使工作更加复杂而不是简单的原因。

7.1 测试和调试环境

我们要讨论的第一个调试工具是测试环境。测试环境包括一套硬件、软件和测试实例，可用于模拟、调查和重现那些用户所看到的问题。没有固定的方法来判断问题的实质，你只是在黑暗中摸索寻找错误的原因。有一个固定的环境，你可以很容易地测试假说并修复错误。我们将按顺序来讨论调试方法，从高级别的调试技术，减少代码的绝对数量来查找 bug；到低级别的调试技术，以发现实质问题。

7.1.1 测试组

测试组用于确保程序按照期望的方式工作。在许多实例中，仅仅通过检查测试实例，就可以得知应该去哪些代码区寻找问题，而不需要看另外哪些区域。如果 bug 直接与测试实例相矛盾，那么要么该 bug 是应用程序的误解部分，要么测试实例没有成功运行。无论哪一种情况，都要检查代码对该特定的测试实例所做出的反应方式，看看它是如何工作的。这会很快且容易地告诉你是测试实例出错了还是 bug 出错了。如果是 bug 出错了，就可以向用户解释该问题，双方在代码应如何工作问题上（根据用户）达成一致理解，或者可以改进应用程序。如果是代码错误，测试实例不正确，就可以根据适当的运行要求修改测试实例，然后修改程序代码让测试实例正确运行，因此，测试实例是 bug 的最容易重现的实例。根据这个思想更进一步考虑，可以利用每一个 bug 报告作为一个新的测试实例。这可用于检验 bug 是否真正地被修复了，还可以认为应用程序的任何变化不会重建一个已知的 bug。如同我们在前一章中讨论的概念一样，可以使用这样的测试组来检验 bug 是否真正地被修复了；而不只是检查症状，改动一下代码，然后就此认为已经修复了 bug。

理解测试实例如何形成，并且如何在测试和调试周期内使用，是很重要的。理想的测试

实例是根据要求和设计规范写成的。编码之后，一个测试实例有 6 种可能的结果状态：

- ◆ 测试实例正确地测试了需求设计的条目，代码执行正确，产生了由测试实例指定的输出。这是最简单、最直接的情况。我们称这种情况为代码执行正确，测试验证了这个事实。
- ◆ 测试实例正确地测试了需求设计的条目，代码执行不正确且没有产生由测试实例指定的输出。这是系统中 bug 的最简单的情况。
- ◆ 测试实例正确地测试了需求设计的条目，代码执行正确，但是，没有产生由测试实例指定的输出。这种情况的出现是有许多原因的，但最基本的是编码系统中的接口错误。也有可能是测试实例书写不正确。
- ◆ 测试实例没有正确地测试需求设计的条目，代码正确地满足测试实例，没有正确完成想得到的需求设计条目。这是进一步验证代码、需求和设计正确性的方法。
- ◆ 测试实例没有正确测试需求设计的条目，代码没有正确地满足测试实例，没有正确地完成想得到的需求设计的条目。这可能是所有现象中最让人不快的。系统设计中编码正确，但系统设计却没有充分反映应用需求。

7.1.2 测试套

组成应用程序的代码，其绝对大小是在大规模应用中测试假说的最大的问题之一。为了检查代码中的核心方法函数或进程，可能必须建立一些高层进程，输入大量的信息，演练到被怀疑包含问题的那一层。当然，有许多方法可避免这个问题。最简单的方法之一就是测试套，也称为骨架代码（或应用程序模板）。

提示：建立一个骨架应用程序，在写子系统时用于测试每个子系统。

骨架代码用于测试小型的代码段，比如实用函数或核心库例程。可以把外部的骨架代码认为是用于测试的伪代码，那自然最好，这样就容易用作测试套。如果公司或部门不需要它，就应当为添加到应用程序中的任何子系统制定一条规则以提供测试套，并同源代码和文档说明在一起。如果不能让开发者做到这一点，就该注意使用应用程序调试阶段的初始化调查层，为每一个你有机会使用的子系统写骨架代码。

7.1.3 过时的 bug 组

要维持一个现存的和过时的 bug 组用来测试程序，看起来好像是多余的观察。然而使用其他项目组来测试当前应用程序并不是很明显。而且，因为程序是趋于创建相似的 bug，所以保留以前工程和现在工程的 bug 组，与应用程序作对比是很有用的。这可与其他项目的症状作对比，它们往往比较相似，从而可以找出相同类别的 bug。如果项目之中重用了其他项目的代码，这将更有必要了。

重用项目之间的代码和库，甚至是需求，其最大的好处之一就是无需大量的劳力即可完成大量的工作；能够重复利用代码只不过是冰山上之一角。当在两个项目之间的需求级上重复利用一个组件时，不但要获得需求信息和完成目前项目要求任务的代码，还要获取能够进

入该子系统的测试实例，例如为子系统而写成的测试套。简言之，需要获得一个可以添加到自己项目中的完整的系统片断。可能还需获得一些关于该子系统的文档说明。

提示：如果重用了应用程序之间的需求条件，还可以重用测试实例、代码和调试实用程序。

7.1.4 日志

在上一章中，我们简单讨论了日志及其在应用程序中的可能用法。现在，让我们来看看如何使用日志来调试程序，以及不宜使用的时机。

如果想知道应用程序如何工作，以及程序各段如何组织在一起时，使用日志是一个很好的办法。通过回头看看各选择分支的日志，你就可得知日志可显示每种方法的输入、不同方法的调用条件，这是关于日志的好消息。现在，让我们来看看坏消息。

在试图调试一个具体问题，有一个最大的优点也是最大的缺点。问题表现出的信息交错复杂，但是还要通过详查这些信息之后，才能推断出代码中真正发生了什么事。在程序产生的输出的困境中，会很容易地错过个别问题。缓和此类问题的一种方法就是：只在框架代码中使用日志，来测试具体的应用程序段。如果已经得到了包含问题的逻辑子系统，这样来处理问题会得到满意的效果。然而，如果把程序作为一个整体来看待，就会发现自己被大量的信息搞得昏头转向。

那么当程序需要作为一个整体搜索时，如何使用日志求助呢？最简单的方法就是建立灵活的日志对象，例如，可以在C中使用如下函数：

```
void LogMessage( int lvl, char *fmt, ... )
{
    // Only log information above a certain priority level
    if ( lvl > gLoggingLevel )
    {
        vfprintf(fOutFile, fmt);
    }
}
```

这个简单函数使用了两个全局变量（回忆一下我们曾讲过的全局变量的概念），来跟踪输出日志信息的“层”，跟踪数据输出的文件指针。使用这种方法，可以更好地控制记录下来的信息，不至于昏头转向。也可以使用日志范围进一步约束数据输出。日志范围的简单定义就是程序中数据来源的地方。例如，可以为程序的每个主要子系统定义一个日志范围，这样在程序运行时，就可以通过开关不同的日志范围，来获取想要的信息。

注意：有时候可能需要写分析程序来总结和显示日志数据，以判断哪里可能出了错误。写这样的分析程序需要大量的工作，但往往在调试工具库中是不可缺少的。

总之，日志是用于粗略估计问题所在的一个好方法，一旦知道了问题的所在，也就获得了产生问题的信息。然而，要想有效地使用日志，就需要在任何给定的时间简化所碰到的数据。

7.1.5 跟踪

同日志一样，我们已在前一章中以正面使用的观点介绍了跟踪。现在我们来看看跟踪作为一个纯粹的调试工具使用所做的工作。同日志一样，跟踪也有输出大量调试信息的趋势，大部分信息又与想要跟踪的区域无关。还是同日志一样，跟踪在想找出出错问题时，常常输出大量的不需要的信息。一旦找到了代码采取的路径，获得了你怀疑可能产生 bug 的代码区；或者至少限定了受影响程序的范围，就将可能关闭跟踪程序事件或者停止查看日志。

一旦选择了想要查看的程序段，可能就不想检查跟踪信息了，其中原因之一就是太容易混乱了。在试图建立问题产生原因时，你会发现自己在函数与函数之间、程序区与程序区之间或同对象与对象之间周旋，这没有什么价值。另外，还应该寻求一些其他科学方法，来判断产生问题的真正的可能原因。当只是用直观感觉去修改大部分代码时，提出一个有效的假说是困难的。

提示：发现可能产生问题的子系统时，就从调试程序中删除跟踪代码。

跟踪是一个很有价值的工具，用于判断程序如何通过代码从一个状态转到另一个状态。它对于推断代码完成某项功能任务所遵循的路径来说相当有用。但是，对于给定调试过程，判断可能的确切问题时，它却是个相当高层次的调试方法。当我们在寻找一些方法，弄清楚应用程序的行为时，跟踪还是我们考虑的最后一个高层次的调试技术。

7.2 中级调试技术

在已经拿着粗齿锯从应用程序找出最可能产生问题的程序区域之后，就需要进行更小的工作，更细地检查问题。大多数的中级调试技术都是假定已经检查过了问题的症状，找到了产生问题的可能子系统，并且限制了问题原因的可能性。这时就可以寻找代码中产生那些观察到的症状的原因了。对于任何给定的一系列症状，都会有许多可能的原因。所发现的症状越多，能够发现出问题代码的机会就越多。让我们来讨论可用于发现问题的中级工具。

7.2.1 内存漏洞检测工具

如果使用的编程语言允许分配内存，就将很有可能碰到内存漏洞问题。这个问题很一般的形式就是资源漏洞问题，也就是同一回事。发现应用程序中的内存漏洞问题，常常是内存块的分配与释放的匹配问题。在本小节中我们学习一些寻找匹配对的方法。

如果在调试环境中工作，从而可以自动地跟踪内存分配与释放，那么就占了优势。这时可以使用编程环境中的嵌入式工具，发现和修复内存分配问题。如果编程环境没有这种跟踪功能，就需要或者使用第三方内存分配跟踪程序，或者自己来编写跟踪程序。市场上有很多可完成此项功能的第三方工具，但是如果选择自己来写（或许因为所使用的专业操作系统或特殊的语言），就需要掌握以下信息：

- ◆ 完成分析的文件和/或函数
- ◆ 为什么分配
- ◆ 完成释放的文件和/或函数
- ◆ 分配或释放内存大小

Visual Basic 没有真正分配的内存, Java 有一定内嵌工作集, 好像排除了分配/释放函数需求。但是如果使用这类语言, 会很奇怪地发现内存漏洞仍会发生。

看一个简单例子, 考虑下面一段 Visual Basic 应用程序代码:

```
Private Type NCB
; Private members deleted for space
End Type
Private Type NAME_BUFFER
End Type
Private Type ASTAT
End Type
Private Declare Function HeapAlloc Lib "kernel32" _
    (ByVal hHeap As Long, ByVal dwFlags As Long, _
    ByVal dwBytes As Long) As Long
Private Declare Function HeapFree Lib "kernel32" (ByVal hHeap As Long, _
    ByVal dwFlags As Long, lpMem As Any) As Long

Private Function EthAddr(LanaNumber As Long) As String
; Setup code removed
lngASTAT = HeapAlloc(GetProcessHeap(), HEAP_GENERATE_EXCEPTIONS _
    Or HEAP_ZERO_MEMORY, udtNCB.ncb_length)
; Modification code removed.
HeapFree GetProcessHeap(), 0, lngASTAT
End If
EthAddr = strOut
End Function
```

可以看出, Visual Basic 有很强的能力, 调用 Windows API 函数 HeapAlloc 和 HeapFree, 它们分别完成分配和释放内存的功能。使用这些函数, VB 不能自动地释放内存。更糟的是, 由此而产生的漏洞很难修补。如果在“帮助系统”例程中包含这些函数, 跟踪他们会容易一些。

注意: 帮助系统例程是对函数或函数集的封装, 可以跟踪一些比如 alloc/free 的函数集对被调用的次数。如果存在全局变量中的次数为奇数, 则意味着是一个漏洞; 如果为负数, 则意味着释放了两次。

同样地, 在 Java 中, 也会出现内存漏洞。这看起来好像不可能, 但事实上很容易发生。让我们来看一个很典型的例子。下面的代码中包含有一丢失了引用对象的侦听对象:

```
public class PrinterObject
{
    static PrintService single;
    PrintTarget target;
```

```
public PrintService getPrintService()
{
    return single;
}
public void setTarget(PrintTarget p)
{
    target = p;
}

public void doPrint()
{
    // Initialize
    // Print the object
}
}
```

这个对象产生了一个内存漏洞，不是很明显。但是，当设置了目标点，例程 `doPrint` 被调用时，如果理解会发生什么事，那么就变得明显了。因为类 `PrinterObject` 包含一个单元素、（这种类型无论在什么时候只有一个活跃变量），所以在目标成员变量中，`PrinterObject` 对象包含一个关于该对象的引用。一旦调用了 `doPrint` 方法，重新赋予该目标为一个新的打印对象，就会出现一个悬空的对该对象的引用，也不能被收集为垃圾。实际上，在 Java 中，比起在 C、C++ 或 Pascal 中，内存漏洞还是比较少的，但是它们的效果却是一样的。类似这样的错误也很难消除。在此需要一个可用于任何对象的 `Assign` 方法，可以赋值为一个成员变量，然后 `Assign` 就可以跟踪变量记录的内容了。

内存漏洞和资源漏洞都是软件开发中最古老的问题。如果在应用程序中不是无差别地使用资源，而是非常认真地跟踪资源，就能够很容易地解决资源漏洞问题。

7.2.2 交叉索引及工具用法

在调试工具库中另一个非常有用的第三方工具是交叉索引（cross-indexing）。这个工具可有助于看清楚不同的代号（全局变量、方法或函数）用在了应用程序的什么地方。如果拥有这个工具，并且消除了一个给定函数中的给定 bug，就可以判断出在应用程序中谁调用了这个函数。这个工具的最大优点就是可以很容易地迅速地自下而上跟踪应用程序而不是自上而下。

当今市场上有很多交叉索引编辑器。在试图判断给定的问题如何会产生 bug 时，我强烈推荐交叉索引。这种工具的另一个有用的特征就是：它能很容易地说明“死代码”，即那些在应用程序中从来不被调用的代码。一些语言开发环境拥有这个嵌入式功能。

如果代码中包含有全局变量，在写代码或者调试应用时，交叉索引是必需的。如果不能确切知道什么地方修改了全局变量，就要花费宝贵的时间努力去判断为什么在调试运行中变量突然变化了。

交叉索引的最后一个用法是在优化过程中。如果知道哪些函数在很多地方都被调用，就可以确保所做的工作可使给定的函数运行更快或更有效，而不会由于应用程序从来不使用这个函数而浪费工作。

7.2.3 调试器

在关于调试的书籍中，不可避免地要围绕着调试器进行讨论。我们一直没有谈及这一点的原因之一就是：几乎每本关于调试的书，都是讨论如何最好地使用调试器检测 bug 的。如何使用调试器并不是本书的主题。每种调试器都包括使用指南，当然也可以找到很多关于如何在不同平台上使用不同调试器工具的书籍。

一方面，在试图发现和修复应用程序中的问题时，调试器是可用的一个强大工具。它们提供了查看实时情况的方法。调试器可允许在任何时刻停止程序，监视代码中的变量，甚至对一些调度器还可以向后单步执行代码行，看看在代码执行前发生了什么。调试器可使人们在解决问题方面变得简单明了。

然而调试器黑暗的一面也是程序员和调试员生活的一部分。我们并不关心如何使用调试器，或者在什么环境下使用。另外，我们将探讨一些实例，在这些实例中不需要调试器，也不使用调试器。

7.2.4 Heisenberg 的不确定原理

Heisenberg 曾说过观察测试可以改变结果。这尤其体现在使用调试器调试过程中。在调试程序中最不如意的事情之一，就是在运行调试器程序时，问题却不发生了。发生这种情况的原因有很多。许多调试程序修改了被调试程序运行时所占用的内存。在调入应用程序之前将许多零填入了内存，这就隐藏了指针丢失问题和变量未初始化问题。更糟的是，调试器可在调试模式下向应用程序中添加代码，这样而产生的问题与应用程序的最后版本表现不一样了。这就是许多测试者即使测试一个软件的非最终版本也犹豫不决的一个原因。

调试器的另一个问题就是，它们会在调试程序时给开发者一种不正确的自信。开发人员使用调试器检查在开发阶段的问题。调试人员很少使用调试器，因为他们所使用的软件的版本必须与用户应用条件相匹配。例如，不可以在产品环境中使用调试器，因为这样做会拒绝用户访问应用程序。同样地，不能总是在一个“干净”系统中调用调试器来查找问题，该机器的环境是惟一的。在这台机器上调用任何其他软件，都会使问题改变形式或转移位置。

由此可以看出，调试器并非总是专业 bug 修复人员的朋友。如果你知道不要求外部干涉的技巧和技术，就会发现在产品质量代码中发现和修复问题也很容易。另外，也可以避免在产品版本中出现的 bug 在代码调试版本中不出现的困境。这并不是说调试人员无需在应用程序中调试了，他们还必须那样做。但是，他们应该局限于单用户产品中，在那里程序运行的环境不会是产生问题的原因。

7.2.5 嵌入式诊断

在上一章中，我们简要地以嵌入式调试、隐藏诊断屏、日志代码等等形式讨论了应用程序中的嵌入式诊断代码。在这一小节中，我们讨论如何运用关于程序工作方式的知识，来嵌入可用于发现 bug 的诊断信息。关于表现诊断的实际形式，可以是日志消息诊断显示、日志文件或其他形式的输出。

检测那些值得检测的条件的类别是很重要的。在这种情况下，需要考虑 PIPE（预条件、输入、后条件和错误处理）方法的益处。让我们来逐一看看这些元素，看看它们代表的意义。

预条件（pre-conditions）就是在调用一个方法、函数或进程之前所做出的假设。任何需要处理的初始化，任何需要调用的方法或函数，任何不能首先出现的操作，都是所要处理的方法或函数的预条件。通过对感兴趣的函数或方法的测试驱动程序或框架代码的观察，一般会发现预条件。

输入（input）也就是对给定函数或方法的输入处理。输入可以是传递给函数的参数，需要设置的全局变量或者是用于方法或函数的外部资源。

后条件（post-conditions）是在方法或函数正常结束后产生的状态、输出参数、全局变量以及其他设置。后条件也可以是被给定的方法或函数在对象实例中修改了的对象的状态。后条件可用于描述方法或函数中期望的（或至少说处理的）错误的情况条件。

错误处理（error handling）指函数或方法返回的错误状态，或者指调用了方法或函数之后存在的条件。例如，如果正在使用 Unix 操作系统中外壳脚本执行的进程，可能要说明该进程运行进程中生成的错误日志文件的错误状态。只检查返回的简单的错误状态往往不够。至少，必须证实进程说明的所有存在状态都完成了，这也是进程说明的一部分。

副作用也可以认为是一个后条件或错误状态。副作用可能包括设置一个全局变量，或者修改一个输入变量来说明一个状态。也许对象中带有的方法或函数可以执行一种操作，比如保存。一旦这个对象在保存操作中被写到了磁盘上，它就会标记上一个“脏”标志来说明它不再被写了。这个标志在被存对象的内部，只能由保存方法（透明）使用。由于这个原因，必须小心说明和注释在函数成功退出时设置这个标志，这也是最能确保该方法或函数成功地保存了对象的一个指示符。在方法或函数退出时检查这个标志，是一种安全后条件反应，以确保该方法确实起了作用。在这种情况下，副作用是那个脏标志。当一个对象输出到磁盘时（即是说，写到永久性存储介质上时），一般是不再改变了。如果这个副作用对象在 Cache 中，管理器不知道其变化，那么修改这个对象会产生问题。

使用保存方法作为一个好例子，要假设拥有可以保存对象列表的方法或函数。函数的返回说明由于被保存的对象之一不能写出来而产生了错误。如果只观察错误状态，就会认为保存工作并没有做。其实并非如此，相反，某个对象状态可能不对，从而导致后来在程序中数据丢失。现在，假定函数执行的错误解释只在不能写所有对象时，才返回同一个错误状态。在这种情况下，就要看看后面的条件，考虑一下需要检查的列表，看看是不是所有标志都设

置了，在日志报告一个错误指示一个或多个对象没有保存。

通过检查函数、方法或进程的预条件和后条件来使用嵌入式诊断，是你所拥有的证实程序是否工作正常的最重要的方法之一。消除那些代码与预条件集或后条件集不匹配的问题，比从四处抓瞎开始要容易得多。类似的，如果检查函数所有的输入以及所有的错误返回，就会提前知道是否存在问题。发现问题所在并知道结果，往往是调试工作的主要部分。当程序运行的错误日志中的信息都掌握了，也就基本上达到目的了。

7.2.6 断言的弊端

尽管书中某些地方提到了这个，但是作为一个诊断分类中的例子使用 C 断言语句还是值得的，你可能不想在程序中使用作为其他诊断类型的替代。这是因为一个只调试诊断不能有助于消除产品代码中的问题。更糟的是，在开发中使用这样的诊断，你就会得到一个对安全的错误理解，因为你提供给应用程序的数据不能触发该断言。如果问题很重要，足以断定其不可能为真，就值得加入一些附加代码，如果问题发生，就检查处理它。

在第 12 章中，我们会更多地讨论如何在代码中防止错误意外发生。现在，接受“只调试检查”不好就可以了。

7.2.7 同用户一起工作

调试应用程序时最强有力的工具之一就是坐下来，同真正的“用户一起工作”来观察他们在做什么，看看他们是如何产生出所需修复的问题的。很少有调试员同真正的用户一起工作的，他们宁肯接收问题和测试实例报告，他们用这些报告重建问题，再修复它。

和用户一起工作的好处之一，就是可以发现他们使用应用程序时试图完成的任务。你常常会发现并不需要修复所谓的“bug”。相反，如果可使用户更容易地完成手边的任务，不使用产生 bug 的方法，用户会更加满意。

在存盘时，关于文件冲突的 bug 就是一个很好的例子。例如，可以想象 bug 这样显示：“当我们的格式保存当前存在的项目时，文件冲突了，导致该项目不能载入。”这可能是写新文件的方法存在问题。也许真正的问题在于新的文件格式导入之前，该区域的文件没有清除掉，搞得数据一团糟。然而，根本没有必要修复这个问题。从用户的观点来考虑这个问题，用户真正想做的就是更新文件格式到更新的设置。要做到这一点，用户用旧名字以新的格式保存该文件。然而，这样做会导致许多问题，如果日后用户需要对它再做转化，用户此时就失去了现存项目的备份，这是最起码的一个问题。

在文件更新例子中，解决这个问题的“正确”方法就是首先对现有项目文件重命名为扩展名为“old”，然后创建以新格式存储的项目的新文件。这样做是可能的，这样可以修复 bug，因为新文件没有包含旧数据，同时创建一个新版本，保留一个旧版本，用户会很满意。通过简单地观察用户，询问用户试图完成什么任务，很容易地会得出这个结论。因为修复了 bug，用户也会很高兴。实际上，因为不但修复了 bug，而且还提供前一版本中没有的特征，用户

会更加满意。

在调试程序中决不可小看同用户一起工作的威力，用户还拥有关于问题本质的大量信息。例如，用户可能观察到一个程序失效只发生在程序某一特定运行中使用某特定的特征时。什么时候使用这个特征无关紧要，只要使用了这个特征，就立刻会发生程序失效。但是，通过听取用户意见，你可能会得出这样的假说，在该特定特征中的什么东西造成了内存冲突，或者产生了一些其他问题，这些问题只在使用那些内存块时表现出来。如果总是依赖于 bug 报告，你将永远不知道这点重要信息。

提示：决不可小看同用户一起工作的威力。

7.2.8 bug 跟踪

把跟踪其他 bug 当作可用于修复现存 bug 的工具，好像有点儿奇怪。但是，跟踪过去问题和 bug 的结果往往是非常好的工具，可用于修复现在的 bug。而且，通过考察以前是如何“修复” bug 的，在第二次碰到该 bug 并修复它时，很快决定跳过哪些步骤。

统计数字表明大约一半以上的 bug “修复”决不会只出现一次。这就是说，第一次修复可能被认为已经修复了错误，其实修复要么不彻底，要么不正确。另外有些情况存在有许多要对待的症状，只有一个修复了，可能会产生更多问题。在所有这些情况中，如果首先知道了处理步骤，可有助于理解在前面的调试员认为修复了问题时在想什么。在决定 bug 真正所在时，这可使你避免犯同样的错误。

在系统中跟踪以前的 bug 还有另外一个原因。bug 趋于成簇，这意味着当发现与以前的 bug 相似的新 bug 时，会自然地怀疑在过去包含有大量 bug 的模块。另外，还可以查看 bug 数据库中描述的症状，看看问题是如何修复的。最后，通过检查所有的开着的 bug，并与关闭的或部分关闭的 bug 进行比较，可以发现这样一个简单的解决方法，千篇一律地修复它们。对调试员来讲，这是目前所有可能方法中最好的。

7.2.9 代码覆盖范围分析

一般地，代码覆盖范围工具用于检查是否处理了所有的测试实例，在系统中是否存在“死”代码。然而，对调试员来讲，代码覆盖范围分析可以展示出以前没有考虑过的问题。例如，考虑下面一段 Java 代码：

```
if ( x == 1 )
{
}
if ( x == 2 )
{
}
if ( x != 1 && x != 2 )
{
    return;
```

```
    }  
    // Do some stuff  
    if ( x == 3 ) // 1  
    {  
    }  
}
```

这段代码可能与前面的代码衔接不明显，但是无论如何测试，包含有 x 等于 3 的情况决不会执行（以 `//1` 标记的代码）。让我们迅速浏览这段代码，来阐明这个语句。如果 x 为 1，第一条 `if` 语句执行。如果 x 为 2，第二 `if` 语句执行。如果 x 为 3，会发生什么情况呢？因为 x 不等于 1，且 x 不等于 2，所以第三条 `if` 语句执行。一个好的代码覆盖范围工具将检测这种情况。因为第三条 `if` 语句下面的代码永远不会被执行，所以这个事件会出现在日志中。

提示：注意应用程序中不使用的代码。

尽管清理代码并不是调试人员的权限，但是如果删除了无关代码并只保留真正使用的部分，会使调试变得十分容易。最后，如果删除这些死代码，就可以释放一些内存，这些内存存在修复内存受限问题时是非常有用的。但是要注意那些特意放置的死代码。比如 `//` 内容，是作为注释的。这段代码在版本 3.3 中将重新被激活。当暂时抛弃或废掉某种特征来满足交付截止日期时，就会使用到死代码。

7.2.10 编译器

等一下！编译器是不是调试人员的工具？事实上，是的，它往往是最有价值的工具，用于找出导致需要修复的症状。让我们看看编译器用于帮助调试程序的一些方法。

注意：对于非编译语言，比如 Visual Basic 和 Java，解释器中有等价的设置，与我们在这一小节中讨论的事件种类一样。

使用最高的警告级别

发现问题的最好机会，就是检查所有可能不完全正确的代码。检查代码的最好的工具，就是在所工作的编译器上使用最高警告级别，需要检查的代码可能不遵守所使用的语言标准，或者引发一些令人惊讶的副作用。这项特征往往警告提示出一些可能从未没有考虑过的问题。例如在我知道的一个工作室中，调试员使用专门的编译器设置可把警告当作错误来处理。这就迫使写代码的人在检查代码之前检查所有的语句；因为主源码控制树中检查代码时，错误被认为不可原谅的违规现象。

使用高级警告级别的另一个大的优点，就是编译器作者通过阅读代码来检查那些很难验证的情况。看下面一段 C++ 代码：

```
int x ;  
call_func (x);
```

显然这段代码调用了函数，然后函数返回一个值，实际上，这个函数可能完成了它的使命。然而，考虑一下该函数碰到错误没有给 x 设置任何值的情况。当函数返回时 x 会是什么呢？好问题，这个问题不必检查语言规则就可以知道。一般情况下，答案就是 x 没有定义。你肯定不想在程序运行时出现无定义的值。如果在 C++ 编译器中使用了最高警告级别，

就会得到这样一条警告：“警告：变量 `x` 在赋值之前可能被使用了。”这样就可在以后使用 `x` 时，避免在代码中不得不消除一个很麻烦的问题。

检验消息

编译代码时，还要检查所有的来自编译器结果的诊断消息。编译器具有诊断问题的能力，这一点你也许并不怀疑；并且，编译器还能够鉴别从来不使用的代码（死代码），以及已经赋值却从来不使用的变量。检查所有消息，并确定你能够理解这些消息。例如，如果在一行代码中对一个变量进行强制类型转换，那么重要的是必须理解这种强制是否起作用。检查每条来自编译器的消息，以确认理解代码在这里所起的作用。

观察生成代码

观察生成代码并不是一种“普通”的调试技术，但是，在不能明白代码所起作用的原因时，倒是一个好办法。大部分编译器要么生成一套组合代码表示，要么生成正在编写的高级别代码 P-code（伪代码）补充。很少会在编译器本身上出问题，这种问题会使生成的程序运行不正确。像这样的问题是很难诊断的，因为我们一般认为编译程序时，程序是按预定方式工作的。

验证来自编译器的代码输出的最容易的方法，就是提取那些可疑的代码段，放入一个小测试驱动程序中。测试驱动程序简单说就是框架应用程序，它们是硬编码的，用来练习某一特定的功能集，比如，单个类的方法或函数。如果测试驱动程序工作出乎意料，就可以怀疑编译器了。使用编译器选项可以查看实际生成的代码，看看它的真面目。如果和期望的不一样，就等于发现了一个编译器 bug。

注意：编译器 bug 相对是比较少见的。只有当排除了所有其他的可能之后，再来检查这种情况。

如果发现了一个编译器 bug，首先要重写一个可以工作的简单实例，然后把转化后的代码移到主程序中去，如果这个工作正常，就通知编译器出售商，在已改动的代码上面加上主要的注释，说明修改的原因。要随时准备同事们可能提出的问题。发现编译器 bug，并不只是一个“修复 bug”类型的事件，而是一个“下赌注”类型的事件。但是这种事件总会发生的。

语言特征

使用编译器的最后一种考虑，就是由编译器指定的语言的特征，可以帮助调试工作，或者妨碍调试工作。一些编译器制造商加入了语言扩展，帮助开发人员使他们开发的程序鲁棒性更好，或者更容易编码。有时，这些语言特征也会带来严重的问题。

这种语言扩展的一个具体例子就是 C 编译器中的 `program` 语句。这条语句本身并不是一个扩展，它存在于大多数 C 编译器中。然而，它可以很大程度地在编译器和编译器之间使用。例如，某一特定的编译器允许使用 `program` 语句在代码中禁止某些警告消息。通过编译器的命令行接口的一系列标志的设置，也可以完成同样的功能。但是，无论哪种情况，在观察程序输出时，编译器都会屏蔽一些症状。如果不知道这种 `program` 语句扩展，就搞不懂为

什么在编译器输出中看不到期望看到的诊断语句。这可能会令人失望。

在想看明白应用程序的状态时，只要有可能，就要找出代码或编译器中所有编译器所指定的语言扩展。最终会得到更多的可迁移代码，它们对调试很有用，且还会得到一套完整的诊断集。

7.3 小结

在这一章中，我们讨论了在调试工作中可用的可能的调试工具。有商业工具比如调试器和代码管理器，也有日常工具如日志对象、嵌入式诊断工具等。你可能已想到了，在不同的环境中，调试不同类型的应用程序，可以采用不同的方法。

显而易见，调试一个运行于单开发系统上的单机应用程序，与调试一个运行于多个不同类型操作系统上的产品系统，是完全不一样的。同样的，用于调试这些系统的工具，和在这些系统中使用的语言一样，也是不相同的。作为一个调试员，你的工作就是：在调试时决定对用户产生最少可能影响的最好的可能工具。

一个木匠绝不会在一个木板上用锤子钉上一个螺丝钉，调试员也必须同样地小心，确保所使用的工具与项目的规模和应用程序的级别相适应。如果应用程序要运行在产品 Web 服务器上，就不能只使用会停止程序流的调试器。如果对所做工作使用正确的工具，就会发现调试工作很顺利，用户也会很满意。

7.4 bug 问题

让我们看一个例子，使用我们这一章中讨论的工具。这是一个调试产品程序的例子，该程序运行在我们无法访问的服务器上。一个 Web 页调用了该应用程序，出现了新的用户界面的介绍之后，程序失效了。没有办法调入任何软件到产品机器上来检测这个问题。请给出要找到这个问题所采取的步骤，以及在这个调查中哪些工具能够使用，哪些工具不能使用。在这个例子中哪些工具最可能有帮助？哪些工具最不可能有帮助？

第 8 章 调试的一般过程

尽管我们用了本书的大部分篇幅来讨论什么是调试，什么不是，以及用什么工具来进行调试，我们却没有花时间来讨论调试一个问题的实际过程。本章主要讨论发现、修正和验证一个应用程序或过程中的 bug 所应采取的一般步骤。尽管并非每次都要使用所有的步骤，但是可以从这个表里挑出需要的步骤。我们的目标是修正一个 bug，并确保它不会复发。发现 bug 的过程，包含一系列可能的技巧。这些技巧的目的，是使你尽可能地像一个职业的调试员。当然，剩下的就靠自己了。

在本书的前面部分我们已经在一定程度上讨论了调试的一般过程。它由四个步骤组成。第一，识别 bug。第二，搜集有关 bug 的信息。第三，提出关于 bug 发生的原因以及如何排除的猜测。第四，测试猜测，把它和现象进行匹配，看它是否符合。如果猜测确实是正确的，就可以修正这个 bug 并继续下去。在本章，我们将看看这四步到底是什么意思，以及如何让它们操作得更简单更有效率。

8.1 识别问题

在任何过程中的第一步，都是看看问题是否会重复出现，而真正的问题是什么。这并不是要去诊断在底层代码中的什么导致了这个 bug，而是试图从用户的角度来看并理解如何使问题重现的，从而就能检测出到底是什么问题。

在调试过程的第一阶段，根本不必看代码。所需要的是报告的错误和程序的需求集。要想确定一个 bug 的真实情况，就需要扮演侦探的角色。侦探通常会问很多问题，一个调试“侦探”也是一样。所以戴上你的 Sherlock Holmes 帽子，叼上烟嘴，开始问问题吧。

8.1.1 这是 bug 吗

当接到用户的问题陈述时，第一个问题是问自己“它确实有 bug 吗？”如果你是一个典型的开发者，如果在把这些问题解释后，用户还对这段代码说三道四的话，你通常会很快说“不”。应该杜绝这种现象的出现。重要的是这到底是不是 bug。

有三种情况的问题不是 bug。首先，该程序需求描述说明程序应该那样工作。在这种情况下，通过定义，出现的问题不是 bug。第二，如果程序是模仿另一个程序，而且设计成与这个程序以同样的方式运行，那么这个问题不是 bug。第三，如果关于 bug 的报告是以“如果怎样就会真的很好”开头的，你就能满足该要求。由于变化、加强、新特性和喜好的要求出现的问题不是 bug。它们仍需改进，但它们并不是 bug，如果这些情况无一适用，那么这

个问题就非常可能是 bug 并需要修正。

8.1.2 为什么它是一个 bug

要理解程序里发生了些什么，就需要知道为什么把 bug 称为 bug。这通常并不像它表现的那样简单。最简单的例子是当需求描述说应用程序应该这样工作，而代码说它应该那样工作。另一个简单的例子是程序执行了一些各方都不希望发生的功能，例如重新启动系统和重新格式化硬盘。可以假设这是个 bug，除非程序被特别设计成要毁坏机器。

“为什么”问题的灰色区域是，你仅仅知道程序不做想要它做的事情，而不能十分确定它在做些什么。这样的 bug 的一个例子是，要一个程序打印但它什么都没打印。有可能是程序遇到了一个阻止它打印到打印机上的情况，例如机器没有连接到打印机。这种情况肯定会使用户打印起来很困难。然而，缺少给用户的反馈会表示这个动作成功了，但用户因为在打印机上什么都找不到而迷惑。这种情况下，该问题就是一个 bug，因为程序不做用户想要它做的事情。

警告：在这个阶段，不要开始深入挖掘代码来确定程序在做什么，这是十分重要的。应该仅仅从观察到的现象着手，因此不必知道代码说些什么。如果从合理的角度来说程序应该打印而程序没有打印，那么这就是一个 bug，因为这不合理。

让我们看一些表现得像 bug 的例子，以及为什么诊断是错误的。一个好例子是一个崩溃的程序。假设用户给了我们一个完整的步骤表来重现问题。第一步是打开一个已经在机器上存在的文件。第二步是通过给它加一个新的入口来修改这个文件。第三步是把这个文件以一种新的格式保存到机器上。这样，在用户的机器上，该程序总是会崩溃。

在这种情况下，开始可能会假设问题出现在保存例程里。通过观察用户机器上的问题，可以看到问题确实像他说的那样存在。这个假设有意义吗？有的，因为它考虑到了所有可得到的观察。现在是否应该深入到代码中看这种文件格式的保存例程是否出了问题呢？在此之前，考虑一下这个问题。有处理这个特殊的保存变量的测试用例吗？如果有，该程序是否不必通过该测试用例？在另一台机器上试试这个问题。假设仍然有用吗？我们假定，在这种情况下，存在一个程序在测试中通过了测试用例，而且问题在另一台机器上不会重现。假设仍有用吗？在这种情况下，答案是不。如果假设正确，就至少有两种情况不应该存在：程序通过了测试用例，问题不在第二台机器上出现。可能有其他的原因来解释为什么假设会失败，比如拙劣的测试员或者软件版本的不同，但假设本身有可能是错误的。

这个例子说明为什么不必一开始就深入代码。观察到这个问题并不出现在其他机器上之后，就该回到观察阶段了。两台机器有什么不同？进一步的观察可能会发现第一台机器的磁盘空间严重不足。事实上，磁盘空间是如此不足，以至于在将它以新的格式存储之前没有足够的空间来保存备份版本。程序发现它不能以新格式存储，不能继续运行，并向自己报错。稍后，结果可能是“bug”，是一个未处理的错误。这是 bug 吗？绝对是。当程序遇到不好的情况时，它不应该崩溃。为什么是 bug？这个问题比较复杂。它是 bug，那是因为程序应该

更加优美地退出，而不是因为它崩溃了。中止不只是程序在这种情况下惟一可以做的事情。

8.1.3 程序应该在做什么

一旦接受了报告的问题是一个 bug 这个事实，而且已经确定为什么它是 bug，下一步就应该确定问题发生时程序实际上应该在做什么。这并不是说你要知道代码在做什么，而是这个时候应用程序从逻辑上来说应该在做什么。只有通过理解它应该在做什么，才能发现它为什么没做它应该做的事情，以及如何使它去做要它做的事情。

如果 bug 是需求未被正确处理的结果，那么逻辑结果就是：它应该在做的事情，就是需求说它应该在做的事情。不幸的是，这种问题很少是你试图确定的，需求错误通常导致其他更复杂的编码问题。

我们把这个论题联系到前面的那个例子上去，磁盘空间不够的那个例子。如果磁盘空间不足是 bug，问题是在此种情况下该怎么办呢？有几种可能的答案。第一，程序显示错误消息然后退出。第二，程序拒绝保存文件，然后退出。第三，程序删除直到有足够的空间来存储备份文件和新格式的文件。在各种可能性中，正确的做法是向用户显示错误信息然后退出。程序不应该继续运行因为这些变化不会被存储起来。另一个可能的做法是通知用户不能存储文件，并允许用户选择释放磁盘空间。

这时是否看代码，取决于 bug 的性质。例如，可能需要去看看内部数据的生成，并理解应该正在发生什么。到知道正确答案之前，都无法修正 bug，能够做的是深入到看起来在工作的一些东西中去。当不理解问题时，深入的解决办法几乎肯定会重现并常常打扰你。

我们再来看一个例子——一个有外部症状的问题，但如果不能知道代码如何工作就不能真正理解它。在这个例子中，我们在一个支持 MPEG 视频的系统上工作，把视频缓冲到磁盘上，并允许对视频进行重绕和重放，然后把它输出到一个输出设备上。正常情况下，数据流入、被缓存、流出，因此在接收机上看到的图像是很好的。但是在某些特定的异常情况下，接收机屏幕会出现空白，因为它没有从系统得到输入。

这是 bug 吗？当然。系统被要求提供稳定的电视视频流到输出设备上，但它没有做到这一点。为什么这是一个 bug？因为“用户过程”被中断了，而且视频没有如期待的那样显示出来。它应该在做些什么呢？好的，在这一点上我们确实必须了解一点技术细节。从接收机的角度来看，内在的技术问题是，当系统需要视频数据时没有准备好以进行处理。其结果是它会重置，从而导致暂时的屏幕空白。更技术性地说，在任何需要数据的时候，要做的就是提供数据。还有，由于某些原因，数据并不是总能在需要的时候被提供。问题不是数据不存在，它很可能是这样。我们能够通过回放缓冲区包含的数据，来说明所有流入的数据被保存在回放缓冲区里（这种情况下是硬盘）。那么，为什么不是需要数据的时候总有数据提供？它将我们带到了最后的问题：我们知道代码假定在做什么，但代码实际上在做什么呢？

8.1.4 程序到底在做什么

现在需要去了解为什么程序没有做它应该做的事情。我们仅仅能够通过检查代码来确定答案，所以现在要暂停观察工作，而开始进行对代码本身的侦探工作。在这种情况下，我们会继续用电视显示的例子来看看到底发生了什么，为什么代码没有按照期望的那样运行。

因为我们引用的例子是一个现实的例子，所以我们不能提供私人拥有的代码。但是，检查代码所实现的算法当然是可能的。在这种特殊的情况下，代码分析表明算法被正确地编码实现了。所以，可以假设问题出在算法和实现本身上而不是一个简单的编码错误。

接下来就是一个系统的概要。有三种基本“线程”在系统内运行。第一，输入线程运行用以从系统的输入端收集数据。第二，磁盘线程用以将数据缓存到磁盘内以及在需要时将数据从磁盘内读出。第三，最后一个线程从磁盘线程维护的缓冲区获取数据并把数据输出到系统的输出端，允许它们被回放给用户观看。

深入剖析问题可以看到我们应该尽量多了解处理过程一些。输入线程填充缓冲区，缓冲区由磁盘线程来写到磁盘去。类似地，输出线程从缓冲区读数据，缓冲区由磁盘线程从磁盘读取。有一个数目固定的缓冲区池。用来读或者写的缓冲区数目却是随时变化的。磁盘线程不会把一个未满的缓冲区写到磁盘上去；在一个缓冲区满并且不再被写入前，不会把它发送到输出线程。

问题的结果是输出线程总是在等待数据，尽管输入线程不断地将数据送给磁盘线程，为输出进行缓冲。看到问题了吗？要断定所有提供的事实需要一段时间。但是，一旦这些事实十分明显，问题就很快变得清楚了。如果不能很快发现问题也不要难过，我们必须深入代码两个月来找出问题。

我们先来看看为发现问题所采取的步骤，这样就能在行动上对调试过程有一个深入的理解。首先，我们提出对问题的一个假设。我们最初的假设是磁盘跟不上输入应用程序的数据流，而且这会导致输入数据错误。但是，验证这个假设，通过把数据“记录”到磁盘，然后在隔离的事件中回放（而不是同时录放），没有观察到数据有丢失。这使我们知道不可能是因为处理的输入端有错。类似地，通过简单地以普通速度回放来自文件的数据，我们发现输出端出错的可能性也在降低。如果数据被正确回放，那么输出端就是工作正常的。还剩下两种可能性。第一，可能有什么东西干扰了处理，比如另外的线程或者其他的一些偶然在线程内的“饿死”输出处理的东西。第二，可能是输入与输出之间的接口有错。关闭盒子里的其他事件的试验，结束了第一个假设。通过检测输入与输出流的速率来测试第二个假定，说明这是一个需要考虑的区域。

记住，在缓冲区满之前，磁盘线程是不会向输出线程传数据的。输入线程一直在填充缓冲区。但是，数据流速率不是稳定的。在某些情况下，有许多数据流入，而在一些情况下，数据流速率十分低。数据流速率十分低的时候，就会看到输入缓冲区填充得十分缓慢，正如想象的那样。但是，输出流速率确是恒定的。每当中断发生时，它就需要数据。这就导致一

个问题——由于数据流速率慢，输入线程不能填满缓冲区以供发送，所以缓冲区没有准备好由磁盘线程发送到输出线程。

Bug 导致当屏幕变黑，用户过程被打断。这是 bug 吗？当然。为什么呢？之所以这是 bug，是由于用户需求说用户经验必须是无缝的。程序做了些什么？在这种情况下，程序输出将会饿死，因为线程机制初始时没有设计好。由于系统没有传送数据到输出系统，用户过程不是完整的。最后一个问题——程序到底在做什么，是一个谜。系统在按照被设计成的样子运转。能被修正的代码内没有“bug”。也不能执行一些改动来修正系统，使得系统按照它需要的那样运转。

和我们在本书前面部分所讨论的那样，bugs 能在任何层次发生。在过程的早期 bug 进入系统，不从系统中清除的话，修复起来就更加难了。在这个例子中，有两种选择来修复“黑屏”bug。在第一个情况下，知道了关于数据率的问题后，就可以重新设计和重新实现系统。这无疑是更好的一个选择，因为这样就能够拥有一个稳定的、良好备案的、良好实现的系统。不幸的是，在现实世界的调试过程中会发现，这并不高明，因为时间太久。管理人员要求的速度，要高于我们重新设计和实现的速度。这样，我们就被逼进了系统的“hack”中。解决问题的方法是在与输入设备的初始接触中缓存更多的数据。这里意味着要延迟输入的显示，直到有足够的数据保证输出线程都能读到一个充满数据的缓存。这不是一个完美的方法。它是使用户感觉更好和在可以接受的时间内提供产品之间的折衷。调试常常需要折衷。

这个例子的重点在于即使代码做的事情不正确，也可以导致 bug。要认识到有时候设计选择是可以影响最终的产品，从而导致用户报告 bug。这些 bug 的修复往往不需要修改系统的基本设计。但是当这些 bug 发生时，做文档和追究原因同样重要，因为这样在重新设计导致问题的系统组件时就会纠正。

提示：常常记录设计缺陷，这样有时间的话就可以重新设计组件。

在这个阶段，我们已经识别了问题并发现了潜在的原因。我们并不确信是否明白其真正原因，但是的确知道问题，并且知道程序应该在做什么。我们也知道程序到底在做什么，所以就能迅速知道它为什么不按照期望去做。下一步是收集关于系统和应用程序运行的信息，这样就可以真正定位缺陷了。

8.2 收集信息

收集问题的信息通常是过程中最难的部分。信息收集步骤的目标是产生一个可以赋值的例子，以判断应用程序中为什么会出现 bug。有相当多的一些技术用于信息收集。现在看看哪些在过程中的某些时候有所帮助。

8.2.1 用户对问题的描述

在正常的环境下，会有用户对问题的一个描述。可能是用户直接递交的，或由客户支持

人员或非技术人员收集的。无论如何，对数据应该首先怀疑，直到得到了关于真正发生的事情的完整描述。问题的第一手描述通常是重要的，所以要准确地记录下所听到的。将关于同一问题的几个描述进行比较，看有没有相同点。例如，考虑如下关于某个系统所报告的 bug 的描述：

- ◆ “我开始时想建立一个收藏列表。先回到主页。选择了右边的收藏菜单，滚动到第三项并敲入了回车键。然后到子菜单的第四项，点击。最后输入 Irving 作为我的名字，点击 OK，然后程序终止了。”

- ◆ “我来到编程菜单，选择‘新建’菜单选项。点击‘创建对象’菜单选项并且输入 HouseObject 作为对象的名字域。点击 OK，程序终止了。”

- ◆ “我选择了‘新建对象’菜单选项，点击了看起来像是齿轮的图标，输入 Rudolph 作为对象名，点击 OK，程序却终止了。”

这些当然不是一个 bug 的症状，是不是？看起来它们和程序的三个部分相关，描述了三种不同的情况。但是如果停下来真正检查每个用户的报告，就会发现潜在的问题。在此必须做一个小小的解释，这在现实世界里是很正常的。人们经常跳过一些步骤，并曲解他们正在做的事情。在报告时，多数人过于注重自己对过程中重要步骤的想法。要经常过滤掉不重要的东西，并寻找其中的共同点。

在上述所给的例子中，有一些共同点。第一，用户点击 OK 结束，然后程序就终止。所以看起来程序中的 OK 处理可能有致命的缺陷。一些实验会做出证明。在此，我们假定这不是问题，而且程序不共享任何与 OK 按钮处理相同或粘贴来的代码。

第二，在我们检查报告时，注意到每人都在菜单上工作。由于许多程序在提供一些功能时都提供一些菜单，这个代码应该在系统的不同部分共享。而且，仔细检查语句之后每个人都是既用鼠标又用键盘，在菜单上选择点击 OK 后导致程序终止。我们快速看一下每个人对菜单的描述：

- ◆ “我选择了右边的收藏菜单，滚动到第三项并敲入了回车键。然后到子菜单的第四项，点击鼠标……”

- ◆ “……我选择‘新建菜单’选项。点击‘创建对象’菜单选项……”

- ◆ “我选择了‘新建对象’菜单选项，点击了图标……”

注意在第一种和第三种情况中，用户明确地使用键盘从菜单选择选项。在第二种情况中，用户提到选择一个菜单选项并点击一个菜单项。这样就可以推断这个用户既使用键盘又使用鼠标。那么，下一个应该看的逻辑位置是菜单处理器，看它对键盘和鼠标入口的处理有何不同。最后的线索是第三个的入口，用户没有用鼠标从中选择任何东西，但点击了图标。查看代码时，我想弄清楚程序在处理键盘和鼠标的组合时发生了什么。简单的测试实例应该是先使用键盘后使用鼠标来选择一些东西，然后反之。给定了用户的讨论之后，就能发现这个领域的问题的根源了。当然，发现这么一个 bug 之后，一次修复即可解决整个问题。这就是调试者的梦想。

我们在处理用户描述时给出了一个警告。这个警告应用于 bug 数据库中，记载 bug 的测试者的输入。因为用户不知道什么对调试人员最重要，他们会提供太多的信息。或者他们只提供信息中的一小段，以使得调试更加方便。不要让报告者猜测问题。如果让他们猜测，他们的信息会倾向于他们对问题的想法。因为问题常常在于代码中一个完全不同的地段，要保证用户描述了他们所能回忆起的所作的事情。

关于用户报告还有最后一个问题。用户常常将问题报告成“在操作<X>系统一段时间之后，系统崩溃”，这种报告没什么用处。常常会有问题，但没有信息。操作<X>系统之前在做什么？到底怎么操作的？大量使用某个函数了吗？操作的顺序是什么？这个例子中，调试人员如果能够坐在用户后面观察他们的动作或者自己动手就最好了。如果能够重现问题，要记住当时所做的动作。也许问题的避免是和步骤有关的。也许问题在于进行某个动作的次数。注意原来的用户何时进行的动作以及何时自己重现了用户的动作。写下它发生的每一个步骤。问题发生一次之后，判断相同的动作集是否产生相同的结果。如果是，就有了一个可重现的例子，用来发现问题的根本原因。

8.2.2 日志文件

日志文件是进行观察的好材料。日志文件可以包含查明所发生的问题的信息。当然，问题是先保证日志文件包含用来识别问题的信息，而且要判断日志中哪些信息说明了问题。

让我们看一个假设的日志文件，产生于一个有问题的系统。注意真正能写出产生这种日志文件的人几乎是不存在的。

```
Starting up...
Running first command...
Running second command
Second command failed!!
Problem with file
Error in function call parameters.
Bad script file.
Error! Danger Will Robinson!
GrepMe: I'm here.
```

乍一看，程序就有严重问题。所有的东西都失败，有无效的脚本文件和函数调用错误。看完代码后，它将告诉你一些所谓的错误，而实际上这是程序的正常处理。毕竟，如果程序正在产生这里显示的错误，那么程序将永远不会运行。由于程序实际上正常，看来这个层次上没产生什么错误，那么这段代码怎么了？

这个文件中显示了多数日志文件中的问题（可能有点夸张）。多数日志文件包含程序员认为在跟踪问题时很重要的信息。不幸地，这些文件往往缺乏一些所需要的关键标准。一个日志文件至少要包含以下的可用信息：

- ◆ 事件的日期和时间
- ◆ 系统产生事件的函数、方法和过程的名字

- ◆ 事件的严重性水平
- ◆ 被记录问题的描述
- ◆ 事件发生时相关变量的值

我们来看看每一条信息，并判断它的含义以及为什么重要。时间和日期的重要性有一些原因。如果知道事件发生的时间，就可以快速查看数据发现应用中当时发生了什么。这会节省很多时间，避免在判断哪个错误与哪个程序的运行相匹配时产生误解。更重要的是它提供了实际的执行序列以及所导致的错误。

产生事件的组件名字的重要性有两个原因。第一，它避免搜索项目中所有的代码。这很重要，尤其是当描述中包含连接的串时。如果能够迅速查出相应的进程，就可以迅速发现问题。第二，代码的同一描述中可能有多个入口。例如假设描述是“无法打开文件”。一般的程序中这个串会在代码中出现许多次。但是如果输出是“SomeObject::OpenFile - 无法打开文件”，就能迅速查出代码中的问题了。如果它能提供打不开的文件名字就更好了。用一个生成的面向对象文件处理器例程，可能就会看到长长的“……无法打开文件”的信息列表。

将严重性水平记录进日志有几个原因。例如，如果为日志文件的每个入口分配了严重性水平，就能迅速发现严重错误而忽略小的错误。在输出日志入口时正确地分配严重性水平，是最重要的。我建议使用四个层次的信息语句：致命的、严重的、警告、提示。显然它们是从最重要到最不重要。例如提示消息会提供某人正在执行什么用户函数。这将有助于从用户所遇到的问题追踪用户的动作。另外，致命的错误会指出什么时候发生了异常事件，如磁盘或内存溢出。

入口 1:

```
Entering OpenFile
Checking FileName .. Ok
Checking Filestatus.. Ok
Checking File Exists .. Not Ok
Creating File.. Ok
File Created.. assigning pointer
Loading file contents..
Done with Open File
```

入口 2:

```
OpenFile : Creating File
OpenFile: Blank New File
```

两个入口都说明了一个事情。在两个例子中，OpenFile 函数被调用，它试图打开一个已经存在的文件却失败了，就产生了这些信息。函数没有特别的问题，它运行完成后程序正常继续。那么我们为什么要输出这八行告诉我们这些呢？假设这个函数调用了无数次，无效的数据将湮没整个文件。

提示：日志入口的一个简单的规则：详细的错误，简明的一般事件。

8.2.3 新自观察

了解所发生的问题的最可行方法，是亲自去观察，这样就能避免观察者的个人偏见。作为一个调试员，观察应该尽可能没有偏见。这表示不应考虑代码，也不应考虑错误发生在哪个子系统。为了正确地观察，应该看着它发生，记录自己的所见所为。这些记录在后来对问题做假设时是很有价值的。

亲自观察的最好的方式是，从一个关于输出的列表开始。列表中可能包含系统环境、安装的硬件以及应用运行时的选项。而且要记录所有的动作。调试人员根本不知道在试图发现问题时什么是有用的线索，所以要记录所有东西。有疑问时，记录下观察：尽量有更多的数据。

调试的时候要一直带着纸和笔。这样就能随时记下每个动作和结果。例如打开了一个文件，并记下文件名比应用中显示时所分配的区域短，这个简单的笔记对于当时可能没什么用处，只是可能需要标题显示区域的尺寸。但是接着可能根据这个观察发现不仅标题显示区域太小，而且标题所分配的内部缓存也小。结果缓冲区溢出，可能会导致一些后来在程序运行时发现的问题。

个人观察应该一直与知道里面究竟在发生什么事情这个事实相协调。因为我们的确知道代码在做什么（或者至少认为它在做什么），这样就有忽略潜在问题的趋势。很难简单地抛弃对代码的知识，但至少应该保证，在做完一些动作后，实际上是在观察而不是假定它工作正常，只因为它按照预期的去做了。观察整个程序，检查内存的使用情况，并看占用的总的磁盘空间。最后，检查结果的输出文件。可能就会发现一个看起来是加载函数中的问题，实际上是在保存函数中，它写出了错误的信息，使得加载函数无法处理。因为确实知道代码在后台做什么，就应该对潜在的被使用资源或者产生的输出有更好的了解。在假设问题的本质时，要检查每个环节。

8.2.4 症状

调试一个应用系统时，多数开发人员和调试人员会忽略而且不跟踪观察的信息。有时候症状就是实际上报告的 bug。其他时候，症状说明了表面之下正在发生的事情。看一下现实世界中的例子。

在一个实现卫星电视记录系统的项目中，程序的行为很奇怪。内部硬盘存储的程序会突然消失。其他时候，系统不能够正确地记录，显示黑屏而不是显示记录的东西。而且那些计划在未来记录程序的记录事件，根本就不产生或者从盒子的显示中消失。

看起来似乎就是报告中描述的那样，所有这些事件都是独立的问题。但在这个例子中，所有报告的“bug”都是更基本的内在问题的表现症状。原来维护硬盘文件目录的代码有错误，使得目录结构被破坏。当系统通过一个内部验证机制检测到目录被破坏时，程序通过删除表示那些被破坏入口的目录节点来进行恢复。

卫星系统中的目录节点，与正常磁盘操作系统中文件入口是相同的。所以，当一个节点从目录树中删除时，它所代表的文件将消失并且系统无法访问。对于剩下的程序，这个文件就好像从来不存在，所输出的文件不同，结果就不同。如果删除的文件包含记录的程序列表，那么盒子上所有的程序对用户来说就消失了。同样，如果删除的文件包含未来计划的事件，这些事件则会从盒子上的显示中消失。如果破坏严重，目录将不能产生新的文件。这将导致记录机制失败，使得任何数据都不能显示在用户屏幕上。指针的 bug 是无法从外部的观察检测的。

一般地，跟踪症状时的最好的事情是，建立一个受被观察行为所影响的相关子系统的树。如果存在几个 bug 而且它们在某个给定的点重叠，就应该看看各个子系统是如何影响这些观测的症状的。如果单个子系统导致所有在给定报告的 bug 集中观察的症状，而且一个问题可以导致另外一些问题，那么修复一个问题就可以消除所有的 bug。

8.2.5 失败的测试实例

虽然看起来明显，测试集的失败实例也能很好地说明问题。不明显的是失败的测试实例也可能会显示出底层的问题。判断其失败的原因，是发现应用程序的给定部分如何工作的好方法。这与读者所想的可能恰恰相反。

测试实例的编写有一些原因。一般地，它们或者直接来自需求，或者由于观察现实中用户如何使用程序。所以，它们常常显示一些用户运行应用程序时的经历，而不是一些特别的用法。而且测试实例一般比 bug 报告简单得多。通常只包含很少的步骤。bug 报告中可以包含很多的步骤，许多都是冗余的和没有必要的。所以通过测试实例来发现问题更加快速和容易，无论是重现还是发现和修复。在第 13 章将更多地讨论测试，以及它与调试的关系。现在，重要的是要记住查看所有测试实例的结果，特别是自动的测试实例。例如衰减测试实例，会检测出最近的修改是否会导致新的问题。如果跟踪测试实例并且相互比较，就能迅速发现什么在什么时候发生了变化。一旦知道是什么导致了问题，修复就变得容易了。

8.2.6 相似的问题

检查相似的问题是另一个应该比较明显的方法，但是许多程序员在用自己的想法修复手头的 bug 时，却常常忽略它。通过查看过去和现在的 bug 列表，往往能发现相似的 bug。因为相同的开发人员会开发应用的许多部分，通过观察什么问题导致从前的 bug，他们就能明白目前问题的原因。

除了历史 bug 列表之外，也可以制作目前应用程序中的 bug 列表。在许多例子中会发现它们非常相似。以下是在曾经工作过的一个项目中报告的系统 bug：

- ◆ bug 1——在数据入口屏幕，有一个域说明我应该输入从 1 到 10 的数，我输入了 11，程序终止。
- ◆ bug 2——在数据入口屏幕，有一个域说明我应该输入从 1 到 10 的数，我输入了 12，

程序终止。

◆ bug 3——在数据入口屏幕，有一个域说明我应该输入从 1 到 10 的数，我输入了 13，程序终止。

这个列表一直持续到 99（该域只允许两位数字入口）。显然这是一个极端例子和一个天真的测试员。但是我相信大家会在自己的应用中，发现这些分散在列表中的相似入口的种类。而且，给定应用程序如果有多个测试员，他们会报告相似的 bug。这是一个理想的情况，因为例子中的测试员也会报告与别人不同的症状和步骤。这就需要使用不同的测试员提供的信息来考察问题。例如考虑如下两个不同调试员提供的 bug：

◆ bug 1——我修改应用程序上的频率锁并给机器断电，机器重启后频率锁丢失。但是如果机器允许继续进行，频率锁在当前的频道就工作正常。

◆ bug 2——我修改应用程序上的频率锁并给机器断电，机器重启后频率锁根本没变。但是所显示的程序即使在锁正常时也会继续运行。改变频道后问题消失。

这个问题也是来自卫星盒子软件系统。“频率锁”用于阻止小孩子看到不该看的东西。每个节目都有一个频率。而且对于任何有比允许频率更大频率的节目，只在解锁之后才能看。这样，关闭盒子时将保存锁，但重启时却没有保存。当然，你知道盒子在正常“运行”时就好了，即使电源指示灯不亮而且屏幕是黑的。只有断电才能使盒子重启。这些就是我们从两个报告中收集的信息。而且可以断定频率锁在当前停留的频道工作正常，但正常地重启后就不行了。最后一点是有趣的，因为它说明问题不在盒子的频率一边，而是应用频率锁时的一些控制。在前面的例子中，当单元关闭时盒子在保存某些东西。如果它掉电或者重启，它只会把信息放在内存中，然后丢失。回头看 bug 列表，发现有许多类似的 bug。数据在运行时和正常关闭时会正确，在掉电时会错误甚至丢失。进一步检查发现对以前的不同数据的 bug 的处理，是在发生改变后立即存盘。对频率锁用同样的方法也解决了问题。所以可以发现，在这个例子中，查看以前的 bug 报告和相应代码的修改，就能成功了。

8.2.7 近期变化

当一个应用程序中在新的编译和构造之后出现问题时，软件工程界的一个老调是“我真的没有改什么东西”。虽然新的问题确实可能因为外部变化和新的输入数据而出现，最可能的解释是代码中的变化导致了问题，而且最可能是软件最近的变化。

在考虑这些时，最近的变化应该是测试最不充分的变化。早期的变化很可能早就已经经过了后来的程序员和 QA 部门的测试。只有近期的代码变化可能没经过什么测试。而且发布之前最后的变化，可能是为系统添加原来没有的特征，或者修复正式版之前发现的问题。这些就是新的软件版本中会导致问题改变的种类。所以，在一个以前没有 bug 的代码突然出现 bug 时，应该考虑这些变化。

为了判断哪一个变化是最近，应该能够查看发生变化的代码和模块。因为实际上现在每个公司都使用一些形式的源代码控制，这通常只是对最近变化的报告。如果源代码控制系统

不能做到这一点，就要认真考虑换一个新的系统。源代码控制的一个最有用形式是：能够在不可分的登记层次控制变化。系统应该能够识别一组应用于给定的特征或 bug 修复的变化。这就可以只去看那些可能导致问题的变化。

有几种方式使用变化信息来识别问题。一个是检查每个变化，看看它是否能导致问题。检查一组变化是一个很难的过程，因为实际的 bug 可能是由于不同的变化中所做的修改组合而导致的。清除每个变化，可能会看出问题的修改，但也可能会把问题移到其他地方。我们需要理解每个变化及其原因，在真正了解产生问题的原因之前，不要管它是如何做的。

发现和修复问题的一个更好的方法，是慢慢地恢复变化，直到发现哪次恢复导致问题消失。然后慢慢地一点点返回变化，或是最少的代码增加量，直到问题重现。通过简单的消除，就能够发现哪些变化破坏了代码。

现在，就有了几个选择。可以继续下面的关于变化的调试过程，或者用关于问题地点的知识中止这个过程并开始修复。这个选择依赖于是否理解为什么代码的变化导致问题。如果能够立即看出导致的问题并知道如何修复，就可以简单地修改变化并继续标准地测试。如果变化看起来无法解释问题，可能会对代码段做出标记以备调查并继续下去。

最后，必须认识到，如果不能理解问题为什么由那个可能是原因的变化引起的，那个变化可能就没有问题。例如考虑一个使用全局缓冲区保存自己数据的函数。看起来一个对函数最近的修改引起软件终止。但是检查对函数修改的代码后没有发现什么问题。这里的 bug 可能是在代码的其他地方，而且只是在这个函数使用时表现出来。早期的编程中，往往加上一个“printf”（向控制台输出消息）就能使 bug 消失。而且更坏的是这样的“方法”经常存在于产品代码中。那个时代已经过去了。你当然不会这样做了；只有别人可能这样。

8.2.8 运行的环境信息

在工作一段时间之后，就会学到一些业务技巧。跟踪环境和机器的其他外部因素就是这种技巧之一。如果这些因素变化了，它们对程序就会有影响。许多开发人员的一个不好的假定就是：他们的程序是一个孤立的岛屿。但是，存储器变化、磁盘空间变化以及其他运行着的程序，都可以在应用程序中轻易地导致问题。

看下面这个沮丧的故事，这是现实世界中遇到的环境导致的问题的例子。在我曾经工作过的一个项目中，系统自从建立起以来已经在实时环境下运行了一年多了。程序中没有加上不好的东西，也没有做什么修改。一天，程序停止运行了。我们正要对它做一些修改，所以我们对修改还没有加入系统就失败很震惊。什么出问题了？重启也是一样。程序会运行一会，然后突然死掉。没有错误消息、日志文件入口，应用程序的内部调试器也没有发现致命的错误。总之，很神秘。

一旦我们验证了程序自身没有修改，就开始看外部原因。我们发现盒子的内存足够。磁盘空间也足够（一般情况下会首先检查这一点，但程序会对此做一个内部检查而且会显示错误信息的）。我们最后断定在程序失败期间数据操作组对机器做了三个改变。第一，他们把

操作系统升级到新的版本。第二，他们安装了一些防病毒软件，捕捉盒子上的恶意程序。最后，他们在盒子上安装了新的看门狗程序，追踪可用的内存和磁盘空间量。最后这个程序最被动，但也与网络上的另一个服务器通信以使自己拥有机器最新的状态。

如果你和我一样，可能看了变化表后就会叫。之后，就会怀疑是操作系统的问题。我们当时就是这么怀疑的。我们来到另一台运行同样软件的机器上也升级操作系统。这里，我们是在不知道问题的具体位置时每次消除一个怀疑。程序在这里运行良好，让数据操作人员把我们笑话了一通。

然后我们检查了内部编写的看门狗程序。虽然我们无法描述它如何导致问题，我们能断定它是新的，值得怀疑。但我们又浪费了时间，遭到了数据操作人员的笑话。现在剩下防病毒软件，这是我们认为最不可能的原因。它的工作就是监视那些类似于病毒的操作并向用户发送警告，说明某程序可能有病毒。这是我们对它的工作的认识。但这些机器一般没人管理所以没有人查看过警告。所以安装软件的人告诉软件运行在无人管理的模式。最后看软件的手册（优秀的程序员最后才看手册）上说在无人模式下，防病毒软件会自动终止那些导致问题的程序并做出内部日志。

防病毒软件杀死我们的应用程序的基本原理很简单。我们在系统目录中建立一个日志文件。因为我们正在修改我们的应用程序中像是系统文件的东西，防病毒软件认为我们的应用程序是病毒，并将其终止，程序为什么会运行一会才终止呢？因为软件要产生一些日志信息，而且日志入口是用块的方式缓存和写出的，所以它要花大约 5 分钟的时间来产生数据，修改日志文件。

一旦我们发现发生了什么事情，我们就修改了防病毒软件，这样它就“知道”了我们的应用程序，而且此后一切正常了。注意这里没有对有“bug”的应用程序做任何修改。这纯粹是一个外部问题。

所要跟踪的信息种类，依赖于所生成的应用程序而不同。例如，如果正在公司中心部门的服务器上构建一个客户/服务器应用程序，你可能要一直管理可用的磁盘空间、应用程序所使用的内存以及可用的内存、盒子上的其他应用程序、应用程序所使用的总的 CPU 时间。而且应该有一个应用程序（如数据库、通信程序、Web 服务器、等等），能够在任何时候显示操作系统版本以及任何补丁、任何使用它们的第三方程序的版本，以及应用软件的版本。

另外，如果你正在写一个单机软件产品，方法就不一样了。应用程序中包含的工具（或者与产品一起销售的附加的工具），应该可以跟踪程序所使用的最大和最小内存、程序运行时的可用磁盘空间总量与应用程序同时运行的其他应用程序、任何已知与系统相关的外部设备。这个工具或者屏幕应该是易用的，这样当用户报告问题时，所有的信息都可以通过电话或邮件告知相关的人员。

理想情况下，bug 报告还应该保留一些空间，留给在应用程序运行中收集的所有信息。而且如果程序因为一个已知的内部原因终止（如缺少磁盘空间），那么在需要描述出用户的动作时，要是应用程序能打印出诊断信息就更好了。

8.3 形成假设

现在你认识到有了一个问题，以及用户观点中这个问题的本质。你从各个方面收集了所有能得到的信息，为浏览代码准备一切东西。现在应该看看应用程序中的代码到底在干什么了。

如何依靠调试经验的水平，和对应用程序源代码的经验水平，形成对 bug 可能的原因的假设呢？如果以前从未和源代码打过交道，作为一个项目的新的开发人员，你就应该采取另外的方法来猜想 bug 的原因。

如果一开始就在系统上工作或至少花了相当多的时间，那么就应该大致知道出问题代码的位置。如果有一个 bug 出现在应用程序中的“Foobar 屏幕”中，就需要知道这个屏幕的形式、代码和调用的过程。这就提供了一个起点，研究源代码判断问题是否在代码中。为了做到这一点，可能会需要使用前面所做的观察，并将其应用于目前的代码。

如果对项目陌生的话，通过查看代码来寻找问题的根源是很难的。可以使用一些步骤来查找代码中正确的开始位置，以形成关于资源树的哪一部分是问题的假设。一个步骤是使用跟踪技术观察程序的输出。通过在每个函数、方法和过程的首部都放置一个跟踪入口，就可以看到每一步并看到出现错误时到底在发生什么。当然，如果错误只是机器上同时发生的其他东西，那么我们就不会按正确的次序看见正确的跟踪入口。这是程序中所发生的事情的一个线索。

最后，如果从来没有调试过大型的应用程序，你可能会剪裁代码，直到能够找到一个足够小的域开始工作。这通常是顶层的代码，其中事件是以宏的方式发生的。你可以先看代码的总图，然后到最高层次的包含错误的代码中。如果代码包含菜单选择，就需要找到程序所有屏幕的菜单处理器，然后到菜单处理器中寻找 bug 报告的起始点。在那里将发现一些以函数、过程或对象形式发生的事件列表。检查每一个（通过调试器或者日志），直到发现导致问题的那一个。这时就从新的层次开始了。

调试往往是辛苦和迂回的工作。你会发现自己在钻研代码寻找问题，而且慢慢地备份资源树寻找问题发生的原因。在任何事件中，都要创建关于问题为什么会发生的假设，这样才能钻研到更低的层次。

在已经做了一段时间的调试工作之后，就会发现问题都属于某些广义的种类。假设通常是从这个种类列表开始，以及排除那些被认为不是原因的种类之后的聚集。一般地会发现 bug 属于以下种类：

- ◆ 内存错误
- ◆ 编码错误
- ◆ 逻辑错误
- ◆ 需求错误

- ◆ 没有检查错误条件
- ◆ 定时/多线程错误
- ◆ 外部产生的错误

当然，并非所有的 bug 都能准确地归入这些分类。有些可能不只满足一个，有些可能哪个都不满足。但是从 bug 可以分为这些种类这个想法开始，会对工作有所帮助。让我们看一下如何能最好地将问题归为一类，以使得工作容易下手。

存储区错误的原因有一些。可能是过量的分配用尽了内存，可能是没有释放程序中分配的内存，或者某块被分配的内存被重新分配但原来的指针没有清除。另外更一般的一种情况是：指针没有被正确初始化，或者初始化错误。在将代码添加到应用程序时，出现在过程输入中的错误的指针、错误的分配或错误的已破坏数据将会在内存中活动。如果对程序添加新的语句之后发现问题转移了，就有可能遇到了内存错误。对这些情况的跟踪，通常是通过使用某种工具或寻找程序中内存的所有释放来检查内存，并保证那些指向内存块的指针被清除了。处理内存错误的两个明显标志是：看到应用程序中所使用的内存量持续下降，以及看到（C++中）所使用的对象（this 指针）成为 NULL。

编码错误是目前最有可能遇到的 bug。编码错误表示没有在代码中把一些东西做正确。可能是调用函数、过程、方法时使用了无效的参数或向将要加载的外部文件发送了错误的数据。编码错误跟踪起来很讨厌，但修复起来容易。为了寻找编码错误，你需要理解代码的每一步在做什么以及能不能做到。最简单的编码错误是未初始化的变量、没有检查空指针、向函数发送无效参数、没有检查如除 0 之类的数学错误。这些发现起来通常很简单，特别是在对程序的崩溃用一个栈跟踪时。有疑问时，就应该考虑编码错误。

让我们看一个很简单的（愚蠢的）在 C 函数中可能发生的错误。我们编写一个函数，初始化一个数组里的所有入口，给它们赋值。

程序中代码如下：

```
void InitializeArray( int *array, int size, int value )
{
    for ( int i=0; i<size; ++i ) ;
    array[i] = value;
}
```

看起来代码能做到期望的事情。循环中没有减 1 错误，数组也不会因为太多的值而溢出。因为我们有一个整数数组并且传给它整数值，是不会溢出这个数据类型的。但是外部程序调用这个函数以后，就会发现这个数组根本没有初始化。这么简单一个程序会出现错误？

再看 for 循环。在 C 语言中，句子的结尾是“;”。for 循环操作“;”之前的语句或者块。这里 for 循环之后的“;”使得这成为一个循环并多次运行。for 循环之后执行的语句是空的，并非我们所期望的数组赋值。真笨，是不是？但这种错误常常发生。

逻辑错误是特殊形式的错误，发现和修复时需要大量对程序的理解。逻辑错误只是实际程序逻辑中的错误，而不是代码本身的错误。例如一个算法是找出比输入的数大的奇数，你

可能会写出如下的 Pascal 程序：

```
{
    if ( odd(inputNumber) ) then
        FindNextOdd : inputNumber + 2;
    else
        FindNextOdd : inputNumber + 1;
}
```

如果你还不知道的话，你会猜想 odd()函数在得到奇数后返回 true，否则返回 false。这个程序看起来任何时候都正常，但有一个逻辑错误。你能猜出来吗？注意错误不在这段代码中，而在于代码所使用的逻辑。看一下算法，看能不能得到关于为什么这段函数会在给定的条件下失败的假设。

看到了？如果没有看到，就先看看表 8.1 的简单的输入和输出，看能否发现错误。

表 8.1 函数输出

输入	输出
1	3
2	3
3	5
-1	0
-2	-1

现在发现了？程序被设计成只能输入正数或无符号数。但代码中根本没有检查负数输入。“-1”之后的奇数单调递增的结果应该是 1 或 3。正常情况下会认为“-1”的下一个奇数递减应该是“-3”。如果不做一些后台工作理解函数如何工作就很可能犯这样的错误。

需求错误可能是软件工程师在调试时所遇到的最坏的错误。这里的问题不在于代码或代码中的逻辑。实际上，它是系统的需求文档中的基本错误。这可能是个疏忽、误解或对问题本身缺少基本的理解。并不能真正地调试一个需求错误。你必须重新编写代码的错误的核心部分，或者对代码做出纠正，以使得程序按照用户的期望去做。

考虑需求错误的一个简单例子。当编写一个处理实时数据的应用程序时，系统中的需求为每秒钟必须能处理 1000 个并行的请求。编程人员认真地检查了所有不同可能的实现来满足需求，并开始做一个看起来每秒钟能处理 2500 个需求的应用程序。他们知道实验室结果和现实世界中的结果常常不相同，他们满足于这个算法，指出现实的性能是需求的 2.5 倍。实际中实现的算法很棒，每秒钟满足 2250 个请求，远远超出了需求。有问题吗？

系统在被请求湮没之后崩溃了。由于需求分析和硬件配置部门的误解，只有少量的机器用于系统中。所以每个机器接受的需求达原来的 5 到 10 倍。软件工作正常，但是它不能处理每秒钟 1000 个需求了。这是 bug 吗？从用户和管理人员的角度讲，应该是。软件工程师们接到了修复 bug 的要求。问题是它无法修复，除非配置更多的设备（昂贵的方法）或者重

写算法处理 5 倍的需求。无论怎样，抱怨都会落到错误的人身上，方法不好。

软件开发中一个更一般的问题是：不检查错误条件和返回值。这个问题常常与错误不在同一位置。实际上，它常常令人意外地露出程序的表面并导致 bug。某些方法和函数返回错误的代码，那是有理由的。不检查这些返回代码，只会导致难以跟踪的、甚至是难以重现的问题。

C 程序中最容易发生的问题也许是文件打开失败。因为 C 或者给文件对象返回一个指针，或者返回一个可以被不同的文件处理例程使用的句柄。如果文件处理中不保证文件是打开的，这是很危险的。可以轻易地将其称为底层的操作函数 bug，但是却无法修复运行时的库。结果最终的 bug 归罪于写程序的软件工程师。

我们看一个包含致命缺陷的 BASIC 程序。这个缺陷不会一直发生，但可以在任何时候发生。仔细阅读代码，想想问题为什么会发生。

```
Function GetValue() As Long
    If GLOBAL_X < 100 Then
        GetValue = GLOBAL_X
    Else
        GetValue = -1
    End If
End Function

Sub Recurse(FileNumber As Long)
    If FileNumber <> 0 Then
        Recurse (FileNumber - 1)
    Else
        Print "File Number is Now: " + FileNumber
    End If
End Sub

Sub CalcRecurse(FileNumber As Long)
    Recurse (GetValue())
End Sub
```

显然，这个例子是人为的，但它想表明不注意返回错误会如何导致问题。Recurse 函数取一个数并递归调用自己将数数到 0。GetValue 函数想返回一个小于 100 的全局变量，超过 100 则返回错误码-1。一旦全局变量超过 100，-1 就传到 Recurse 函数。这个函数检查它是否为 0，如果不是，则减 1，并自调用。由于值已经小于 0，函数会一直递归直到栈充满，程序崩溃。任何结果看起来都可能不会导致应用程序的终止。

开发人员一般在出错时返回错误的代码。但对函数和方法的用户来说，更重要的是没有得到期望的结果。例如考虑一个要分配一块内存代表一个矩阵的函数。函数接受一个参数，是被返回的已分配内存块的指针。它返回一个错误代码表示分配是否成功。如果失败，错误码为非 0 值，输入指针不变。这时，调用者必须检查错误码。如果不检查，矩阵内存指针可能永远也不会初始化为任何值。这是一个典型的内存错误。忽略返回码和对它的检查，会导

致跟踪错误的各种不同的困难。

返回码问题的一个扩展是异常的处理问题。一些语言如 C++ 和 Java, 解决未检查的返回值时抛出“异常(exception)”。异常是一个错误对象, 从栈里产生, 直到得到处理。处理异常的典型伪码如下所示:

```
try
{
    call_a_function_that_throws_exceptions()
}
catch ( an exception )
{
    do_something_with_error();
}
catch ( another exception )
{
    do_something_else()
}
```

语言实际上如何实现异常处理, 对我们并不重要, 就像实际对异常处理时的语法一样。重要的是, 它就像返回码问题一样, 有些方面更加糟。

异常在某些方面比返回码好些; 它们必须被处理, 否则导致程序崩溃并显示异常。但是对于最终用户来说是一样的。程序非正常终止, 用户不得不处理结果。同样, 程序员可以看出异常的本质, 但并不能一直看出其原因。而且, 虽然可以用某种方法捕捉异常, 但可能使得问题更加难于跟踪。异常可能会发生, 但代码只是忽略它。这是和忽视返回码相同的问题。

基本假设中最后的错误类型是定时和多线程错误。这个错误类型的症状, 在第5章中已经描述了, 但这里仍然要提。如果程序在访问全局变量或共享的内存时发生崩溃, 就有可能是遇到了多线程问题。同样如果程序只是在对实时系统添加了一些输出语句后才崩溃, 就有可能遇到了定时问题。两者都很难跟踪, 所以在决定问题是多线程或定时的原因之前, 必须排除所有其他的可能性。

在任何事件中, 一旦对假设分了类, 就要判断问题到底出在哪里了。给定了假设的分类之后, 就应该能够把问题限制在代码的特定段。如果不能把问题限制到这个程度, 你就不是在假设, 而是在乱摸索。那样不会解决问题, 注重的只是症状而不是问题的实质。

现在来看一个为问题做假设的例子。我们使用的是先前讨论的“黑屏”问题。这个问题发生在实时电视显示系统中, 当数据不能从系统的输入流到输出时, 屏幕就经常变黑。

我们收集关于问题的信息, 发现虽然数据流入了系统, 但没有流出, 有几种可能的原因。第一个可能的假设是在从系统的输入端向系统的存储端传输数据时发生了丢失, 导致不能传到输出端。第二个假设是没有满足系统的输出端或者输入端的要求。第三个假设是正确地将数据传到了系统、存储, 但没有及时送到输出端。

现在, 我们有了几个假设, 目前的问题是哪一个假设可以测试, 哪一个解释了我们所收集的所有症状和观察, 哪一个是最可能的。注意: 做假设时, 不要因为感觉另一个更像而取

消这个。只要事实支持理论，它就是一个有效的选择。把它加到列表去检查，把列表中你认为差不多的分为一级。

提示：做假设时，不要因为感觉另一个更像而取消这一个。

8.4 测试假设

对问题识别的最后步骤，是测试对问题的原因的假设。为了充分测试问题，要做到三件事。第一，保证你认为正在发生的事情确实在发生。第二，给自己证明假设的基础是问题的根本原因。第三，给定假设和一组不同的数据后，预测将会发生什么。一个假设就如同对问题的数学证明，遵循标准的证明规则。先证明它，再假设，再证明。

注意：软件工程界最大的问题是我们有了工程师的纪律，但从不遵循工程师的规则。工程师只是假定某些事情会奏效。他们不设想问题的症状是根本原因。工程师从不通过处理症状或者在不理解问题的真正原因时而修复问题。我们无权将自己称为工程师，直到遵循工程师的规则。我们也许叫软件技术师更加恰当。

说明如何建立和测试假设的最容易的方法是使用例子。这是因为每个假设因其项目和问题而不同。虽然一般的调试过程适用于任何产业和任何问题，但解决问题的细节却截然不同。

8.5 崩溃的 Web 服务器示例

所有真正值得记住的例子都不是空泛的工程文本，而更像是神秘的小说。我们看现实中的一个例子，其所有元素都神秘：犯罪、疑问、阴谋和参数。对不起，这里没有谋杀，下次可能有。这些调试观察可能很吸引人。

问题发生在几年以前，我开始为一个小的网络公司工作。公司想为用户做一个交互的网站，允许他们在网上查找、浏览和查看文档。开发时间表很紧张，我们想把网站建成并运行起来。在计划对网站代码进行最后测试的两周前，出现了一个严重的问题。网站随即莫名其妙地崩溃。因为网站本身是多用户系统，很难跟踪出是谁或是什么导致了问题。那些开发了网站所使用的所有 COM 对象的内部组件的人员，与编写了网站的所有脚本和 HTML 页的 Web 开发人员之间，相互指责。自然都说对方造成了问题，而自己的工作没有错误。自然地，由于双方相互指责而不是寻找原因，最后什么也没修复。

在最后发布 beta 测试版前的那几天，人们开始冷静下来。问题被提上来，人们开始严肃地考虑。收集观察到的信息，提出假设，开始严肃地调试。

开始时寻找问题的步骤是识别什么发生了崩溃。我们选择在网页的 ASP 代码中实现一个日志对象。因为网络服务器崩溃了，也许我们能判断出它首先在哪里崩溃。这将有助于排除系统的一部分问题，直到能发现寻找真正问题的更好措施。使用来自用户和程序员的观察信息，我们向人们报告看到崩溃以前所使用的脚本中添加了日志语句。这就允许忽略大量的

系统，可以假设崩溃时没有谁在那个领域而且那里的代码不会导致问题。这在任何线性或类似线性的系统中都是有效的假设，网络服务器就是一例。我们研究网络服务器日志（服务器自己产生的，而不是我们的内部日志程序），看哪些页在崩溃期间处于服务中。

一天内，我们把问题跟踪到了服务器的某个页。通过测试人员、网络开发人员和服务器开发人员之间的合作，我们得出了可重现的测试实例，使得问题能够按要求发生。这时候，我们知道了这个页面使用了哪些组件，所以可以产生合理的假设并测试它。

导致问题的页包含三个不同的元素。第一，有用来产生用户看到的 HTML 文本的 ASP 脚本代码。第二，有内部服务器组件组开发的 COM 组件，为数据库提供接口（面向对象数据库，不是 SQL）。第三，有几行 Perl 代码检查本地文件是否在 Web 服务器上。Web 和服务组件组都被命令用有效的测试实例测试他们各自的组件，以判断崩溃是否因为这些原因。

第一个假设是 Web 组提出的，说 COM 组件返回了错误代码，ASP 页的 VBScript 无法理解。所以服务器上的 COM 组件就出现了错误，使得服务器崩溃。由于我们以前有过类似的问题，这个假设看起来合理。我们看了页面上所有的 COM 组件调用，检查了 COM 组件代码看会返回什么错误。建立了一个测试组件，向 ASP 脚本返回可能的错误看会发生什么。但这都没有问题。

第二个假设是服务器组件组提出的，说是 Perl 脚本的错误。争论的是 COM 组件工作起来与在其他脚本中没什么两样，在每个测试实例里都没有奇怪的错误报告，而且这是惟一包含 Perl 脚本段的页。为了测试，Perl 脚本从原来的脚本里取出并被放置在单独的脚本文件里。测试实例是为那些存在或不存在的文件而写的。脚本看起来正常，没有问题。

这时，我们有两个可能的假设，都好像是正确的。不同的开发人员也没有其他观点。开发人员决定退回一步以看看问题的原因。我们只是看看能收集到什么症状和观察信息，而不是试图理解问题的原因。

为了跟踪问题，我们决定把调查限制在系统上的单个用户，用不同的事件使用那个页。我们在脚本的每一行都加入了日志，以查看页面在哪些代码处失败。我们试验了不同的输入看能否导致问题。观察一整天之后，我们得到了最后的结论。服务器组是正确的。问题应该在于 Perl 代码中，这是由页面在单个用户时有多个不同输入下判断的。我们也认为问题就在于检查文件的存在性的例程里。那只是 Perl 代码中的一行，而且通常情况下工作正常。在系统中其他地方也有这样的代码。那么有何区别呢？

这把我们带到了最有趣的调试阶段。一旦问题语句定位但没有识别，下一步就是用已经掌握的知识生成一组新的假设，去寻找问题真正的根本原因。

我们又看了一次神秘的 Perl 脚本问题，提出了几个假设来解释我们的所见以及问题原因。因为问题中的函数检查文件的存在性，我们估计文件里的某些东西可能出错了，也许是文件的属性。例如一个只读文件就会导致这样的问题。我们建立了一组文件，包含文件属性集所有可能的组合，用来测试假设。这要求在每个文件目录入口设置属性位。每个文件都在

Perl 代码中测试，都正常。

然后我们测试了过长文件名导致问题的假设。这个 bug 在我们在操作系统中使用了过长的文件名时出现，所以假设看起来有道理。由于并非所有我们使用的文件都有过长的文件名，这可以解释问题的间断性。而且由于长文件名可以导致内部存储机制中类似于缓冲区溢出的问题，这个解释更加有意义了。我们用从空文件名（无字符）到很长的文件名对函数进行了测试，函数都成功了。又一个假设被事实否定。

现在，应该看我们的过程了。给定了所收集的观察之后，我们想集中所知来合理地解释为什么程序会崩溃。当每一组观察都被加入假设时，建立了一组测试实例来验证这些假设。如果测试实例证明问题与假设无关，假设就被抛弃，而考虑另一个。这时最重要的两点是不要因为失败而灰心和不要回到开始。两者都会让你离题，失去线索。

我要解释问题到底是怎么回事和如何修复的，而不是我们所讨论的各种假设。必须明白无论问题多简单，过程都是漫长而讨厌的。一旦发现原因，你往往会奇怪当初为什么没有发现。这是正常的。理解问题需要时间。研究问题和观察需要时间。找到观察与问题之间的关系也需要时间。

问题的答案当然不会自己摆在桌面上。实际上问题是由于来自数据库的文件名和路径常常无效。数据库旧的入口包含了无效的驱动器和路径。Perl 解释器中的 Perl 函数不会正确地处理错误的驱动器和路径，会导致操作系统出现异常。这使得解释器中的进程，即 Web 服务器的崩溃。

我们如何发现这个问题？一个假设提到了送到 Perl 函数的数据库域。这迫使我们实地观察了传到函数的数据。我们的确注意到路径和驱动器无效，但以为函数的文档是对的而且问题会正确解决。犯了最重要的错误：不要认为函数就像所描述的那样工作，要经常检查。毕竟如果都像所说的那样工作，就不会有问题了。一旦使用无效的文件路径来测试，程序就崩溃了。那么解决方法就很简单。我们用另一种语言重写了函数，并保证它在所有的测试实例下都正常工作，问题得以解决。

测试假设首先是将假设与开始的观察相匹配，看假设能否解释所有的观察。如果能，就应该整理自己对问题原因的理解。如果这和那些为假设做的观察数据相匹配，就可以开始修复了。

8.6 重复直到某个假设被证实

在提出了假设并测试之后，可能发现要放弃它。这在调试过程中会多次发生。我经常发现自己看到后认为“知道”问题是什么。最后往往发现其原因大不相同。重要的是不要太认定一个想法，你可能会忽视那些证明假设错误的观察现象。而且必须想到没有关于正确答案的时间表。管理人员要看下面这个提示：不要问调试员多久才能修复问题，直到他们判断了问题的根本原因，并验证了假设。在此之前所给的时间表可能过于乐观或者是错误的。

提示：无法精确估计修复 bug 的时间，直到证明了关于它的一个假设。

注意例子中有另一个问题——即使给定一个假设，也往往需要反复。虽然知道问题是网络服务器上特定的页引起的，我们仍然需要查明这个页上是什么导致的问题。这个缩小潜在问题范围的过程就是反复。一旦发现了 bug 的宏指令，就要去掉所有无关的代码，直到找到导致问题的行。这里是通过为每一行做日志，直到停下来得到日志信息。根据定义，在这种问题中，上一条日志信息后面的一行可能就是导致问题的行。找到行后，又要开始假设了，要猜出问题为什么会发生在那一行。一旦跟踪问题到了某一代码行，就可以把这个语句处理的信息加入日志，并用这个信息对代码进行 bench 测试，看问题到底是什么。但就像从例子中看到的那样，如果没有给定有 bug 的函数的代码或方法的细节，将常常做一些更多的测试工作。

有时候你会发现对问题的原因相当自信，因为所有的观察都支持结论。这时就应该修复错误了。

8.7 提出解决方法

最后，你解开了谜，发现了问题的原因。现在，你往往长吁一口气。问题是，你还没有解决问题呢，只是识别了它而已。管理人员不会只满足于解释，他们需要修复。调试过程的下一个逻辑步骤是提出一个方法来解决问题。

当然，提出方法的实际行为依赖于 bug 本身及代码的结构。讨论问题的正确方法要求了解问题中的应用程序和 bug。为每一种问题都提出一个方法，应该是很难的。我们能做的是检查一些可以加进方法中的因素。

在理想世界里，只需修复在假设证明阶段识别出的问题。不幸的是现实世界的要求不一样，往往需要发现对修复方法所能做的最小范围的修改。作为例子，我们考虑那个神秘崩溃的网络服务器。假设最后假设的问题是因网页中包含 Perl 函数引起的。虽然这是一个有效的情况并且被观察所支持，它不是一个好的最终答案，因为整个网站有许多使用 Perl 函数的文件。最终的修改可能会导致比他们所要解决的还多的问题。而且修改所有的文件所需要的时间会相当多，以至无法按时完成。这里，管理人员当然不会允许把所有的 Perl 函数都从网页中删除。你将不得不重新考虑这个假设，判断到底是哪一个函数导致了问题。

作为职业调试员，你第一个要学到的规则是对想要的结果做最小的修改。如果问题是由于进入某函数的数据引起的，最简单最安全的方法是在函数的入口检查数据，对调用者返回合理的错误代码。如果问题是由于在一个对话框上多次点击一个按钮引起的，方法是在它被点击一下后立即将它设置为“disabled”，然后在安全之后再设置回去。

另一个规则是在提出方法之前，看这个方法需要什么。你可能假定了一些根本没有提供的东西。这将导致人们不高兴，而且你在公司的信誉也下降。两者都会使你在项目中受苦。

在提出一个方法之前，特别是没有参与系统的初始实现时，应该与开发这段代码的人员

交流。这会避免提出一个已经试验过的或者导致其他问题的方法。也应该检查 bug 跟踪系统，看以前是否试验过类似的方法，而且因为导致问题而被删除了。试验那些已经用过的失败的方法，会得到同事和管理人员的嘲笑。

你提出的方法应该覆盖三个领域。第一，它能解释为什么能够修复问题。第二，如果它要替换原来的代码，它应该指出应用程序中受影响的区域。第三，它应该保证自己不比问题本身严重，意味着它不会导致另外的问题。

8.8 测试解决方法

在提出了解决方法并把它实现之后，下一步就是在最终产品中测试解决方法。这个过程分两步。第一步是执行能够使问题重现的步骤，看看 bug 是否仍然存在。第二步是试试其他导致此问题的方法，看看问题是否得到重现，这些方法可以确信不会导致同样的问题。

使问题修复的惟一正确办法，是准确知道问题是如何造成的。如果不能找到造成问题的步骤，就不会真正地了解这个问题，也不可能确定问题是否已修复。你所做的是聪明而有效地解决问题，这个问题可能在将来会以某种变异的形式表现出来。我的建议是别这样做，否则会在这个漫长的过程中使自己发疯。生成一个可重现的测试用例，确认解决方法修复了这个问题。

提出导致问题的其他的方法很困难，但这是绝对值得去做的。如果知道多种方法使问题出现，就能很容易地确定：解决办法不是只修复了初始测试用例的表面工作。如果方法解决了所有产生这个问题的可能途径，这个方法就是一个合格的修复，从而可以认为这个问题修复了。

8.9 重复直到某个解决方法被证实

你常常会发现猜测是正确的，但是所选择的解决办法却没有作用。在这种情况下，就需要重新看问题，找到其他可能的解决方法，并测试之。你需要继续这个工作直到办法起作用，充分地解决问题。这时就可以说 bug 解决了，并把它移交给 QA 部门做回归测试。

这时另一个要执行的关键任务，是寻找可能会有同样问题的相似的代码片断。在一个范围内运行重现问题的等价测试用例，通常能在相关代码的其他范围发现问题。这一点对于从系统的一处粘贴到另一处的代码尤其重要。

需要提醒注意的重要一点是，如果一个解决方案没有解决问题，在转移到其他方案之前，它就应该被彻底清除。如果没有这样做，就会受困于两个问题。一是可能引起一个掩盖了老问题的新问题，使得一个有效的方案看起来失效了。第二个问题是可能会以在系统更广的主题里毫无意义的孤立代码结束。你也将不再能测试到真正的问题了。先痛快地将所有代码删除。首先理解对代码进行的所有修改的整体特性是十分重要的。在寻找 bug 前，需要理解如

何把代码恢复到它以前的状态。

8.10 回归测试

任何调试任务的最后一步，是做一次系统的全回归测试，以保证在修复问题时没有造成其他问题。要在修复了一个问题后立即做，这一点十分重要。理想情况下，如果软件支持，应该有一段自动回归测试的脚本。那样的话，就可以立即确认没有造成新的问题。

到目前为止，最让开发者和用户感到灰心的事情，可能就是软件版本虽然修复了一个或者更多的问题，却产生了一组新的 bug。即使新 bug 比已修复的 bug 的危害要轻一些，发布版里的 bug 也将影响软件的可信度。愿意忍受那些已存在的、他们知道并了解的 bug 的用户，不会乐于见到新的、他们不知道如何遇到或者避免的 bug。

回归测试应采取两种形式。第一，应该有自动程序来测试最通常的方面。第二，整个系统应该确认因修补 bug 所作的改变没有产生新的 bug。一个执行精确测试的好办法是把测试用例归到被鉴别或者相互关联的应用程序的范围之内。如果能够事先确定应用程序的哪些范围可能被变化所影响，就能通过仅在那个范围内运行测试用例，做一个迅速的“脏”测试。这并不能替代完全的回归测试，但它应该提供一个温暖但不准确的感觉——并没有破坏任何新东西。

运行了一个新的回归测试之后，由于问题修复了而且没有引入新的问题到系统里，从而使你自己和 QA 部门感到满意，然后就可以将这个 bug 标记为已修复。

8.11 小结

作为总结，我们浏览一下应该采取的步骤，以解决系统中的问题。这个过程应该对软件 bug、处理 bug、甚至是需求 bug 都有效。每一步实际的内部过程可能因不同类型的 bug 而不同，但处理过程仍有如下的相同部分：

- ◆ 识别问题。确定所要修复的是什么，这个问题是否一个值得修复的 bug、一个需要加强的地方，或是需要修改的文档错误，或者是用户的一个误会。
- ◆ 一旦确认问题确实存在，系统需要修改，就要收集关于问题的信息。确认问题是如何发生的，问题的症状是什么，以及用户按照指定的步骤操作时希望发生什么。
- ◆ 在了解问题并搜集了足够的、用以猜测问题的根源的信息以后，就可以提出假设了。问题的根源可能与症状有关，也可能完全无关。所提出的假设应该能够解释症状和应用程序的观察结果。
- ◆ 如果有了一个假设，就需要对它进行检测，以确信它能解释所有可能的现象和症状。这时，应该采取下一个逻辑的步骤，看看通过假设能预测什么其他的信息。测试阶段要确信预测满足应用程序的实际行为。

◆ 下一阶段是反复进行假设和证明，直到提出一个通过了测试、并能解释所有观察到的问题的解决办法。这时，就有了一个有效的、似乎准确的假设。

◆ 假如发现的假设通过了所有的测试，下一步就是提供一个对给定现象和假设造成的问题的解决办法。显然，提议的解决办法要修复问题，并必须考虑合理折衷，比如时间和可利用资源等。

◆ 这时，实现这个办法并对它进行测试。如果问题解决了，过程是完全的，然后准备测试。如果问题仍然存在，解决办法就是不合格的，需要被清除。然后要提出一个新的办法并实现它。重复这个过程，直到解决办法合格并解决了问题。

◆ 调试过程的最后阶段是将系统进行完全的回归测试，来确认对系统的改动没有产生新的问题。

8.12 bug 问题

当你为一个 web 站点设计数据库的时候，生成了三个表。一个表是关于用户信息、姓名、地址、电话号码等。第二个表是通过邮政编码确定税率。第三个表包含邮政编码需要的税利信息，这样这个电子商务系统就能准确地向政府纳税了。三个表的设计要素如下：

表：用户数据

Name: String

Address: String

City: String

State: String

Zip Code: Long Integer

表：由邮政编码确定的税收

Zip Code: Long Integer

Tax Rate: Float

ExemptFlag: Boolean

表：已付税收

Zip Code: Long Integer

Total Taxable Amount: Long Integer

当系统被实现时，系统会报告两个严重的 bug。已付税收表的总数与从电子商务中实际交易的报告的总数不相符，而且在系统的某些部分一些邮政编码也没有准确地报道。

仅仅看表，能够预想为什么会发生这些问题吗？如何跟踪这些问题？如何修复它们呢？

第9章 调试技术

前面，我们已经讨论了很多关于调试的理论、情况和例子。但从另一些方面来看，我们确实需要考虑在代码出现、问题和 bug 时该如何来解决这些问题。在工作中，必须应用一些相关的调试技术。在本章，我们将介绍几种调试的技术，并且告诉你如何在实际的工作中应用这些技术。

在计算机科学方面的书中，最令人烦恼的事情之一就是：它们往往把自己限制在一个“理想”的状态中。从一个干净的平台开始：工作时具备了所有的完美条件：有无限的时间去处理所有可能出现的情况，而这些情况在现实世界中都是不可能发生的。资源的缺乏、条件的局限和代码的删除是常常发生的事。如果以上这些情况不会发生，当然是最好不过了；但这显然是不可能的。在试图去解决问题时，必须考虑到这些因素。由于以上这些原因，我们在讨论调试技术时，更多的是在“现实的”环境中进行的，而不是在“理想的”条件下进行讨论。

为了使专业的调试人员能从本章中得到更多的帮助，我们把本章分成三节。第一节讨论一些经常使用的调试技术，你可以在任何情况下——不论是否找到 bug——使用它。第二节讨论的调试技术是“现实的”机器调试技术，所谓的“现实的”机器也就是不能完全相信的工作环境。第三节讨论的调试技术只有在工作环境是“理想”的时候才能使用。

在开始讨论调试技术和它们的相关描述之前，让我们先熟悉一下调试技术的一些属性。这样，你就可以在适当的时机应用所给的技术。在选择一种技术取解决系统的问题时，必须清楚地认识到你所使用技术的一些属性，比如它所包含的内容，它的灵活性，以及使用它可能产生的危险。

最后要讨论的是，哪些技术有助于工作，哪些对工作没用。有一点是至关重要的，那就是在进行调试之前，应该找到一种最适合问题的调试方法。如果使用的方法适合解决问题，那是最好不过的；如果不是这样，就应当尝试不同的技术，直到找到一个适合问题的方法。使用一个好的定义过程可以在很大的程度上帮助发现和解决 bug。与使用一个特殊的技术相比，从一些现有的技术中找一个最适合的工作也许更为重要。不论什么时候调试 bug，在发现有效的方法之前，都要考虑从未使用过的技术，并把它当成一个候选方法。要强调的一点是：不要害怕使用新技术。

9.1 插入式调试和非插入式调试

插入式调试技术一般应用在那些在运行时需要修改的系统或者环境。非插入式的调试技

术应用在运行时不对源代码进行修改的系统或者环境。总而言之，除非插入不会影响到运行环境下程序的运行，或者没有其他可行的方法，否则，不能使用插入式调试技术。就像我们前面讨论过的一样，有时“现实”的环境要求我们必须使用我们不愿用的插入式调试技术，如果没有其他更好的替代方法的话。

一个简单的插入式调试技术是在应用系统中使用调试器。调试器会修改源代码，因为当把调试器链接到原始系统时，会把调试信息包含到符号表中去。另外，在系统的运行的情况下，调试器还会改变系统的条件。这样，在有调试器情况下运行的程序就不能认为是在“正常”的情况下运行，因为断点和条件有可能改变变量。

非插入式调试技术适用于那些在运行时完全独立于系统，比如观察系统和监视器系统。通常，这些程序能让你监视系统运行时的状态，而不会改变系统或者应用程序。

虽然在大部分的应用中你可能会优先选择非插入式调试技术。但是，在决定使用那种调试技术之前，应当权衡一下问题的紧迫性、bug 的严重性和调试系统的时间。如果问题很严重以至于无法利用客户/服务器应用程序，那么使用复杂的插入式调试技术去快速地发现问题是值得的，尽管结果可能会影响到更多的人使用系统。

应当从以下几个方面，来考虑使用插入式调试技术还是非插入式调试技术：

- ◆ 调试技术要改变源代码是否不会出现？
- ◆ 调试技术改变了程序的运行环境是否对程序的运行没有影响？
- ◆ 在正常的系统操作时，指令调度、特殊错误屏蔽或者不正常中断等情况是否不会发生？

如果以上问题的答案都是“是”的话，那么就可以使用插入调试技术；否则，就只能使用非插入调试技术。包含插入调试和非插入调试的调试技术通常称为“使用仪器”，这个术语是从科学试验和硬件系统测试中借用的。硬件系统试验和测试是利用硬件仪器来监视操作和收集数据，利用所收集来的数据对被测系统进行鉴别和量化。软件调试试验和测试和它没什么区别，也要用“使用仪器”来监视、收集和被调试问题有关的数据。硬件调试也能像使用硬件设备一样使用软件“使用仪器”工具。同样，软件调试有时也要使用硬件工具来确定问题的 bug。

9.2 短期调试和长期调试技术

当使用调试方法时，还应当考虑它是长期调试技术还是短期调试技术。长期调试技术是指试图发现系统正常运行超过公平的时间时出现的错误的调试方法；相反，短期调试技术是指快速发现问题。如果运行程序是为了插获问题，并且是你要插获的惟一问题，那么你可以试图使用短期调试技术；但是你如果要插获的是细节上的错误和记录特殊的信息，或者是在系统出现错误或条件改变时详细的错误屏蔽情况，那么就要使用长期调试技术。

总的来说，短期调试技术比长期调试技术要求更多的插入。例如，当你怀疑有个问题是

由存储器溢出引起的，但是你还不能确定。这时，如果假设正确的话，你就可能会通过电子邮件或一个特定的弹出式屏幕插入代码消息。这个消息包括对程序员有用的信息，这样一旦假设被证实，程序员就能解决问题。如果消息没有被接受，但是问题又出现，这就说明假设可能是错误的。

在调试器中运行一个程序就是短期调试技术。既不能在成品环境中的调试器运行程序，也不能解除只能在调试器下运行的软件。不过，调试器是短期调试技术的基本条件。例如，在 Web 服务器脚本的每一行后面加入日志记录，这些记录不仅会使处理的速度减慢到不可接受的程度，而且会填满日志文件或者掩盖了数据屏蔽。显然，除非在系统设计时考虑到这个情况，否则长期调试技术将不利于用户，这样的记录只有在短期调试技术中才能正常运行。

应当从以下这个方面来考虑使用长期调试技术还是短期调试技术：适当地应用相应的调试技术，系统是否还能正常工作？

如果答案是肯定的，那么就应用长期调试技术；否则，使用短期调试技术。只有在开发软件的过程中进行调试时，才使用短期调试技术，短期调试技术不适用于成品软件的调试中。另外，当软件问题严重影响到用户对系统的使用时，这时就要使用短期调试技术。在这种情况下，我们最有可能接受使用短期调试技术，即使是在成品软件的调试中。

9.3 成品软件调试的折衷办法

在客户/服务器模型或者在单机应用模式下，由于调试器的影响，会出现独特的问题。例如，质量保证（Quality Assurance, QA）部门不允许在成品软件或 QA 系统中安装软件监视器或专门的调试软件。这是因为，如果在服务被解除时出现故障的系统不能找到引起故障的真正原因，那么就不能对它进行准确的测试，这种不稳定测试对软件的调试没用任何用处。

对付成品软件的调试有很多折衷的办法，当然，安装一个调试器是不行的。但是，我们可以通过原有软件的基础上增加事件记录或错误屏蔽的办法来调试。其实这都是应用了插入调试的技术，在问题比较严重的时候，这种办法也是可行的。如果问题严重到了用户没办法运行软件时，管理部门和 QA 部门就会放弃所有对调试的限制，这是一个经常使用的办法。不过，从管理上来看，这不是一个好办法。

以下是一些对成品软件调试的折衷办法：

- ◆ 你可能不能直接安装你的软件，但可以让操作人员帮你安装。
- ◆ 可以让操作人员帮你监视软件的运行，提供事件日志文件。
- ◆ 如果是运行在客户/服务器模式下，就不能在与其他用户隔离的情况下调试，而必须和其他在线者共同进行调试。
- ◆ 在单机模式下，一个折衷的办法是对出现问题的用户进行专门的调试，这样就能得到较好的调试结果，而且不会影响到产品在市场的销售。

9.4 调试技术介绍

在下面几节，我们将详细讨论各种具体的调试技术，给出每种技术的一些主要属性，包括：

- ◆ 技术的名称。
- ◆ 技术的详细描述。
- ◆ 该技术属于插入式调试还是非插入式调试。
- ◆ 该技术属于短期调试还是长期调试。
- ◆ 该技术最适用于哪种类型（单机、客户/服务器、成品软件）。
- ◆ 技术的具体应用（如果该技术是可应用的）。

9.4.1 面对实际用户

在软件开发初期，程序员很难想到将来会是谁来使用这个软件。如果仅仅是开发者使用这个系统，那么对可能存在和出现的问题就存在偏见。因为系统的开发人员对系统很熟悉，他们总是按照系统最能支持工作的方式来应用系统，而不是按照实际用户的方法来应用系统。所以，一个最好的发现和解决问题的办法是让用户来使用软件。在软件开发中，经常让一批测试人员使用我们开发的系统来帮助我们确定是否存在 bug。遗憾的是，在开发软件过程中，我们很少有这么一批测试人员，让他们每天对软件进行测试来发现新的 bug。在软件开发的过程中，调试每时每刻都在进行，我们也应该每时每刻进行测试。

- ◆ 调试类型：非插入式。
- ◆ 调试时间：长期调试。
- ◆ 使用层次：任意。

9.4.2 查看观察日志

在监视一个程序的运行过程，一个最好的追踪事件的办法是记录程序运行的日志。日志不仅可以帮助收集到所有的信息，发现程序的问题；而且，对后续的开发人员，日志也是很用价值的文档。对代码的观察包括事件发生的地点、不同子系统的行为和系统的假设条件；对系统运行的观察包括出现的奇异现象或者是不正确的运行结果。在运行程序前，首先应当明确输入和相应的输出，特别是在结果出错时更应该这样。记录正确的操作和结果（执行程序）和记录不正确的操作和结果是一样重要的。在系统中发现一个 bug 时，可以通过对观察日志的检查来消除 bug。查看观察日志是一个简单易行的查看系统运行过程的方法，也是为工程后续开发人员提供调试资料的有效途径。

如果在系统的观察日志上已经有了正确的执行结果，那么就不必在这一个已经有了正确结果的地方进行开发（或者重新开发）和测试，这些正确的执行记录同时也为回归测试和问

题重定义认证测试提供了快速执行顺序。在调试过程中，虽然没有改变任何东西，但 bug 还是突然消失了（这是最令人苦恼的事）。以前正确的程序执行记录对这种情况是很有价值的。

- ◆ 调试类型：非插入式。
- ◆ 调试时间：长期调试。
- ◆ 使用层次：任意。

9.4.3 记录调试代码和过程

对于专业的开发者和调试者，最好的长期调试技术是记录解决问题所写的代码和系统的使用过程。虽然很多开发者记录了他们的内部代码，但很少有系统的外部特性的记录。因为开发者很熟悉系统的设计和开发，所以他们在写代码时往往会掩盖了这个假设。当完成了最初开发后立即进行调试时，这时也许可以用自己的观点来处理相关的文档，因为这时对系统还是很熟悉。但几年之后，因为开发过程的故障，系统中出现了很多的 bug。这时，开发人员就无法测试代码段，或者无法做出合适的修改，或者不能保证修改不会影响到代码的其他部分。如果是从团队的角度来正确地记录系统的使用过程，就能解决以上这个问题。如果对工作的工作和假设作了相应的文档，就能防止一些问题的出现，这些问题正是因为没用注意到假设而引起的。

怎样才能做好文档呢？在我看来，每个开发人员在开发前应当有个手册。手册里有对软件安装和版本的详细指导；有程序的源代码和文件树；有系统编译、安装和调试的文档等内容。对系统管理过程的改变和测试过程也应当被包含在手册中。

- ◆ 调试类型：非插入式。
- ◆ 调试时间：长期调试。
- ◆ 使用层次：内部。

9.4.4 模拟代码和问题

使用这个调试技术，首先应当找到过去出现的类似问题或者过去出现问题的代码。这个方法的前提是软件在开发过程中已经保持了系统出现的历史记录和它们当前的状态，此外，还要求记录了修改文档和解决 bug 之间的关系。如果同伴没有保持这样的记录，就很难使用这个技术。也可以在代码中加上诸如“Fixed a problem with <xxx>”这样的注释来代替记录。寻找这样的记录能完成对 bug 的追踪，虽然它不是一个很有效的办法。

使用这个方法的基本原理是：相似的问题由相似的原因引起，相似的代码产生相似的问题。虽然很难正确证明这个原理，但这个原理是适用的。

- ◆ 调试类型：非插入式/插入式（使用注释）。
- ◆ 调试时间：长期调试（追踪问题）/短期调试（发现问题）。
- ◆ 使用层次：内部。
- ◆ 具体应用：假设已经有了记录 bug 的数据库，用 bug 描述的关键词在数据库中查找

相似的 bug。在这里，使用一个能充分反映 bug 的关键词是很重要的。例如，在一个子系统中有关于 MPEG 压缩，那么就应当把 MPEG 当成 bug 描述的关键词。这样，就会很容易查找到所期待的结果和信息，并可以适当地应用这些信息。另外，如果 bug 数据库支持产生问题的子系统，那么只要在子系统中查找类似的信息就可以了。

使用这个技术的另外一个方法是：在产生问题的局部代码中查找类似的代码，并且确定代码是否被修改过，这将能发现和调整在程序中一个地方被调整的代码。

此外，如果知道出现问题代码的编写者，那么也就可以发现他所编写的其他代码中的 bug。因为人们经常会犯相同的错误。如果能确定一个开发者的普遍错误，也就可以发现他的 bug，这将有助于其他 bug 的调试。

9.4.5 简化重现性

当出现一个比较独特的复杂问题时，经常会采用一系列的步骤来重现这个问题。总体上，首先要为一个甚至一群用户来重现这个问题。为了解决问题，把一个要重现的问题分解成几个最小的解决步骤是很重要的，这样的简化有两个好处。第一，每一次能用小的步骤去重新测试，使出现错误的机会减小。第二，也是更为重要的一点，在测试过程的每一个步骤中只需要检查系统中有限的一些症状就能发现引起问题的原因。检查的因素和因素之间相互影响越多，那么调试问题的时间就越长，因为每个人都希望花费的时间越少越好，所以把问题分解简单的几个步骤是必要的。

- ◆ 调试类型：非插入式。
- ◆ 调试时间：短期调试。
- ◆ 应用层次：任意。
- ◆ 具体应用：用一系列的步骤来重现 bug。如果发现某个步骤不影响到 bug 的调试，就把该步骤删除；否则，就把该步骤加入到调试步骤集合中。实际上，需要两个列表，一个是测试元素列表，另一个是有用测试列表。如果某个步骤间接地和调试有关系，那么一旦问题被确定就把该步骤加到集合中。重复几次这样的步骤，删除那些不重要的步骤，最后，只保留对问题有影响的最小集合。

9.4.6 把问题简化成最简单的元素

调试软件的 bug，应排除所有的无关信息，发现引起问题的真正原因。通常情况下，当软件工程师遇到 bug 时，这个问题通常代表一个类型的错误。现在我们用一个简单的例子来说明这个问题。其实，真正的问题和 bug 没关系或者关系很小。出于简单的考虑，现在看以下这个例子：“使用系统的某一功能部件，输入起始日期为 02/30/2000，处理后的结果完全错误，点击 Continue 键后，系统崩溃了。”有以下几个现象和这个结果相关关系。第一，结果显示“messed up”（专业术语，意思是结果不正确）；第二，系统崩溃（非专业术语，意思是系统崩溃）。经过对以上现象仔细的研究，真正的问题并不是以上的两个结果。输入的日

期是 02/30/2000，即 2 月 30 日，但是 2 月没有 30 天。分析这个简单的例子，问题出在日期检查例程没有被调用或者出错。

- ◆ 调试类型：非插入式。
- ◆ 调试时间：短期调试。
- ◆ 最适用层次：任意。

◆ 具体应用：最简单的办法是把错误报告分解成单个的句子，捕获详细的错误报告并把它们记在不同的卡片上或不同行上，接着仔细阅读每一个错误报告来发现引起问题的真正原因。使用这个方法，就能把混乱的显示和系统的崩溃联系起来。所有的可能情况，显示混乱必然会引起系统崩溃，因为内部处理系统已经被破坏了。“using <x> feature of the system”的描述和其他描述没有关系，它总是单独出现的。输入日期 02/30/2000 和被选择的功能部件有关系，按 Continue 键与输入日期和所选择的功能部件都有关系。结果，有两个可能的检查方面。观察每个方面，制定一个最简单的例子。通过对错误描述的检查，发现了非法的日期，非法日期的输入在系统中和其他问题独立，所以它可能是引起问题的原因。如果输入合法的日期不会引起相同的问题，那么就能把问题归结到非法的日期上。

当然，如果假设不正确，就必须重新对假设进行评估。例如，如果把问题归结于非法的日期，修改日期重新运行，但仍然出现类似的问题，那么日期就不是引起 bug 的根本原因。这时，就必须考虑其他的方面（按 Continue 键），或者可以说是没有足够的解决问题的信息，如果需要更多的信息，就必须回到调查阶段。

9.4.7 代码消去法

当对代码有适当的理解时，代码消去法是一个好的发现问题的办法。在这种方法中，从系统中消去一段代码直到发现问题，就能确定引起问题的代码（函数或方法）。代码消去法最适合线性代码，也可以应用在任何系统调试中。这种方法要求对代码有足够的理解，知道哪些代码能删除而哪些不能删除。例如，如果删除了初始化的代码，虽然程序不会再出现问题，但系统也崩溃。显然，这不是调试所希望的结果。

代码消去法不能应用在没有严格折衷的用户的生产系统中。当对代码完全理解，并且编译整个系统所用的时间不长时，就可以使用这个方法。如果知道怎样重现问题，但不知道代码的哪部分会引起问题，代码消去法是很有效的。在这种情况下，通过探测，用逻辑的办法解决问题，消除每个潜在的疑点。惟一的问题是，在典型的软件开发中，疑点包含在成千上万条代码中。为了发现疑点，代码消去法经常和简化法一起使用。

代码消去法要注意以下几个重要的细节。第一，要非常明确解决问题的强制途径。这种办法特别适用于小的系统中，例如命令行实用程序。第二，在使用这个方法之前，要有绝对有效的可重现条件。如果不这样，就不能准确地确定引起问题的代码行，将花费很多的时间。第三，保留功能相同的源代码。为了解决原始问题，必须删除整个子系统。这时，问题可能被解决了；也有可能根本得不到解决，问题得不到解决可能是当子系统调用被删除之后，系

统产生新的路径。路径分析对于证实程序在每次运行期间走相同的路径是至关重要的。如果发现是路径引起问题，就可以在分支代码上使用代码消去法，这样，就可能发现引起问题的原因。

使用代码消去法，应当承认对程序的运行没有任何线索。如果选择使用这个调试方法，应当知道是什么问题，接着使用更普通的探测技术来发现引起问题的代码。

- ◆ 调试类型：插入式。
- ◆ 调试时间：短期调试。
- ◆ 最适用层次：单机、客户/服务器非成品软件。

◆ 具体应用：在代码中应用这个调试方法，必须熟悉系统的源代码。如果对源代码不熟悉，这个方法就不能使用。用 9.4.5 节和 9.4.6 节介绍的方法找到好的起始点。在这点上，把程序分成两部分。前一部分由用户接口和逻辑组成，后一部分是实现系统的功能部分。找到起始点后，一直在功能块中删除系统的前半部分代码，直到发现引起问题的代码。当引起问题的代码确定后，一步步地替换系统中的代码，来确定该行代码是否会引起其他问题。注意，在增加代码后删除问题代码或替换问题代码，可能会出现迷失点问题（stray pointer problem），这时就必须考虑到针对 bug 类型的技术选择。

9.4.8 简化法

对于软件工程师来说，最困难的事就是在系统代码中突然发现所在子系统的问题。例如，当怀疑问题出在底层磁盘操作例程中，并且这个例程仅在很长的代码的后面才被调用。这样，为了调试这个问题，每次都必须重新编译，你将会感到烦扰、灰心。在这种情况下，最好的调试方法是简化法。当我们怀疑问题出在底层子系统，就从主程序中删除这个子程序，替换成别的外部输入例程。写一个简单的测试驱动来测试所怀疑的子系统。

这个方法有一个前提。对于紧密结合的系统，就不能从主程序中把被怀疑的子程序分离出来，用测试驱动来测试，也就不能用这个方法。同时也要求像原系统一样，为底层代码提供相同层次的输入。如果忽略了一些条件，像全局变量、函数的副作用等，就不能生成初始化代码，测试也就没有意义。

简化法可以概括如下：如果有关于问题原因的理论，就用一些最简单的例子来证明它。当对代码的错误有很好的了解时，简化法就很有用；相反地，当不知道怎样发现问题时，简化法就没有作用了。在单机应用系统或用户/服务器的 QA 系统中很容易使用这个方法；在成品软件中，即使有非常详细的、非常好的测试用例，这些测试用例也可以应用在这里，简化法也是很难使用的。简化法的好处是可以简化一个复杂的问题，比如多用户系统。简化法的目标是发现一个引起问题的最小功能部件集合，这个功能部件含有最小的输入集。

- ◆ 调试类型：插入式。
- ◆ 调试时间：短期调试。
- ◆ 最适用层次：单机、客户/服务器非成品软件。

◆ 具体应用：最好的应用简化法的方法是为可能出现问题的子系统写一个测试驱动程序，接着，模拟访问问题子系统的系统其他部分。简单的测试驱动程序包括子系统、问题输入和替换外部联系，观察是否还会出现问题。如果问题没有解决，可以使用包含测试驱动的标准调试技术，而不是去调试整个产品的、多用户的、多线程的应用系统。

如果用测试驱动不能解决问题，那么另一个好的调试办法是用一个模仿问题的小数据集来解决问题，可以通过去掉系统用户接口代码和提供底层的输入来实现。当然，这是建立在用户接口层代码能很容易从系统的逻辑代码中删除的假设基础上，如果这个假设不成立，就要考虑其他的办法。

提示：为了测试底层的功能，一般要保持删除用户接口代码段的能力。

例如，当编写一个客户/服务器的应用系统，这个系统包含了传输信息给服务器应用程序的客户端图形用户接口（GUI），服务器不经常崩溃，但出现的问题很讨厌。为了确定引起程序崩溃的原因，可以使用简化法。首先，必须知道哪些代码可能引起问题。所以，使用了第一级调试技术，例如检查服务器的日志来发现可能引起问题的不正常终端。在检查日志的时候，发现了问题可能是由客户端的非法输入引起的，不过，这个非法输入每一次都不是完全相同的。还有，这个非法的输入只有在另一个客户端通过当前数据交换确定数据值时才出现。此时的假设是初始数据值引起传递数据给服务器的数据缓存区错误。

为了检验这个假设，可以建立两个不同的客户端来发现它们如何发送请求的数据，接着来测试；也可以去掉客户端，编写一个简单的测试驱动来发送特殊的消息给服务器。如果已经有了这样的测试程序，就能容易道过发送消息来检验假设，启动系统，发送第二个消息。如果假设是正确的话，这个简单的测试程序将很容易证明它。

9.4.9 使用调试器

用调试器调试程序看起来是理所当然的事，但这个方法的使用范围是有限的。能使用调试器的条件比起不能使用调试器的调试更受限制。让我们观察一下它们的条件集。

可以在以下状态下使用调试器：

- ◆ 单用户的单机应用系统。
- ◆ 控制台应用系统，例如通用网关接口（Common Gateway Interface, CGI）或者其他真正的多用户但可以通过测试驱动来运行的应用系统。
- ◆ 只有单用户出现 bug 的多用户系统。

以上这些是真正适合使用完整的调试器进行调试的状态。特殊的调试器，例如硬件调试器，也能应用于设备驱动、系统维护和其他运行在操作系统下的组件。如果扩展了“调试器”的含义，也可以在调试脚本或者其他解释系统中使用调试器。

不可以以下状态下使用调试器：

- ◆ 任何要求多用户产生问题的多用户系统。
- ◆ 任何运行在原始机器上的成品系统。

- ◆ 任何和时间相关的系统（例如实时数据采集）。
- ◆ 任何能改变通过系统数据速度或数据流的系统。在这种环境下，使用调试器是不可能的，因为这将影响系统的运行，并且可能引起系统的崩溃。

在使用调试器调试时，确实很难发现某一类型的 bug，而且也会引起问题。两个最明显的例子是偏移指针问题和非初始化问题，因为当调试器正常初始化存储器整个模块时，通常赋给程序变量一个固定的值（一般为 0）。这样，就会使非初始化变量出现问题，因为所有的变量都被初始化了。同样，在大部分的程序中，当指针被赋值 0 或 NULL 时，它就不能指向任何地址了，所以它隐藏了问题而不被发现。在包含这些类型的 bug 中，不能使用调试器，而应当使用其他调试技术。

- ◆ 调试类型：插入式。

为什么使用调试器法是插入式调试方法？因为必须使用不同的选项编译系统，生成一个可调试的应用程序，在程序运行时改变这些选项条件。

- ◆ 调试时间：短期调试。
- ◆ 应用层次：单机。

9.4.10 跳跃

我们前面提到，跳跃调试技术经常和其他调试技术（例如检查代码法、代码消去法）结合起来使用。跳跃法建立在对系统代码了解很多的假设之上，这种方法最适合系统的开发者，因为他们对代码了如指掌。在这种情况下，看到一个故障就会下意识地产生一个分析所熟悉代码的过程，并选择一个和故障可能相关的代码范围。跳跃法是一种系统消去调试技术，如果知道所给的子系统不会引起特定的错误，就可以把该子系统排除在考虑范围之外。通过排除了大量的子系统，要考虑的范围就变得很小。人们通常按照代码的复杂性（越复杂的子系统总是越有可能出错）、所遇到问题的个数和每个子系统的故障、子系统的任务对系统进行排列。

例如，如果出现的故障是数据文件被破坏，那么就应当立即排除调制解调器控制、图形用户接口和存储分配等子系统，而文件操作子系统和对象高速缓存子系统可能是引起问题的原因。下意识地看到对象高速缓存子系统的输出没有出现错误（也就是没有看到任何不正常的现象，但也没有证明它是正确的），那么就可以把问题归结到文件操作子系统。

一些对跳跃法应用的告诫令人惊骇，甚至很多的软件工程师在查找 bug 时使用该技术时也很惊讶。第一，当没有真正掌握代码的工作过程，了解代码的作用时，跳跃技术几乎无法工作；第二，如果使用了跳跃法，但对问题的假设不正确，那么将完全浪费时间在证明假设上。不像其他的调试技术，跳跃法通常不会有助于对问题的进一步理解；第三，跳跃可能是错误的，但它却表现为正确。这是因为这种调试技术不是按照“正常”的调试步骤进行的，在没有发现根本的原因时也能很容易地排除故障。如果试图使用这种调试办法，必须重新考虑问题，并确认所解决的问题就是系统的真正问题，不是去对付故障的现象，而是要对付问

题的本质。

- ◆ 调试类型：非插入式。
- ◆ 调试时间：短期调试。
- ◆ 应用层次：任意。

因为该调试技术要么适用，要么不适用，而不管所使用的系统的类型；同时，因为在使用该技术时对问题做出了不确定的假设，所以该调试技术既能应用在成品的系统中也可以应用在开发系统过程中。

9.4.11 分解法

可以通过多种途径使用分解法。可以把整个系统分解成子系统，然后去掉那些对问题没有影响的子系统。同样地，可以根据故障现象把问题本身分解成几个子问题，由多个开发者各自对付每个可能的故障，一旦某个开发者发现了故障的可能原因，就可以检查它们各自的原因来发现哪个是引起问题的真正原因。如果应用适当，分解法是一个特别有效的调试技术，它可以方便地和其他类似的技术结合起来使用，例如代码消去法、简化法、使用调试器法和简化问题发现独立故障原因法。

虽然调试系统的目的是发现引起所观察到故障的根本原因，但发现引起每个故障的表面原因通常会更容易。一旦知道了引起每个问题的可能原因，就可以通过追踪问题逻辑，甚至系统的要求来发现问题的共同点。这就更容易得出一个假设，因为只需控制较少的变量。当用这种观点考虑调试时，调试就像解决数学方程组。但是，我们通常会碰到 n 个方程、 $n+1$ 个变量的情况，这就很难去解方程组了。事实上，这个数学方程组使我们知道我们没法解决这样的系统，只能提出问题的可能解集，而找出正确的答案是很困难的事。

分解法的实现就像二叉树搜索。从可能包含问题的一组代码出发，把这组代码分成几组功能相关的代码，删除和当前代码相关的每个问题，重复以上过程。举一个例子，开始时，把所有问题的子系统分成图形用户接口（GUI）块和底层逻辑块，从系统中删除所有的 GUI 元素，同时为底层子系统提供相同的输入。如果问题仍然存在，就可以排除所有的 GUI 元素；相反，如果问题解决了，就可以假设底层逻辑是正确的，问题可能出在 GUI 元素上。当然，如果 GUI 元素也没有发现问题，就必须返回重新考虑逻辑子系统。值得注意的是，按照以上这种划分系统的方法，系统是被分成三部分，而不是两部分。这三部分是 GUI 部分、逻辑部分和它们之间的接口部分。问题可能出现在任何一个部分，如果 GUI 部分和逻辑部分都工作正常，那么，就有必要在接口部分中查找问题。

注意：分解法和代码消去法不同在于分解法不需要删除任何代码。通过把系统划分成几个组成部分，一步步鉴别它们之间的关系，然后用二叉树搜索法确定哪些部分和 bug 有关系。这个方法通常要求对代码有较高的熟悉程度。

- ◆ 调试类型：非插入式。
- ◆ 调试时间：短期调试。

◆ 适用层次：一般情况下，这个方法可以应用在任何的应用系统调试中。不过，如果怀疑问题可能出在系统的内部逻辑而不是出在需求或 GUI 逻辑上时，这个调试方法就不能较好地应用在成品软件系统中，因为在成品软件系统中不能发现程序运行的过程。

9.4.12 种植错误法

种植错误法的思想来自 QA 领域。它的思想是在代码中播种一些已知的错误，这样就能提高测试的有效性。这种方法看起来好像和调试技术没关系，但是这个概念的引入是为了消除 bug，而不是引入 bug。考虑到不是总能在代码中捕获到适当的错误条件，而这个错误通过在系统中的传播会引起完全不同类型的问题，很难检测故障现象。种植错误法就是故意触发这样的问题来观察它们所引起的结果，这和简化法很类似，因为开始时就知道问题的存在。不过，种植错误法还可以发现底层的问题。

例如，你追踪一个奇特的问题，但是毫无结果，并且无法解决 bug，但你已经注意到一个底层函数，这个函数不能正确地处理错误条件。这时，你可能想到引起错误条件的可能原因和错误条件在整个系统运行时引起的各种可能故障现象。通过有选择地在代码中引起错误，就有可能发现错误的整体影响。当然，这个方法要求能正确地把特定的错误引入到代码中，除了成品软件系统，其他系统调试中都可以使用这个调试技术。

经常会听到这么一句话：“除非发现了一个没有把柄的错误，否则是不会不去解决它的”。虽然发现这样的错误时会去解决它，但在解决问题之前必须知道所要解决的问题是什么。令开发者最灰心的一件事是从 bug 数据库中得到一个 bug，但是既不能重现它，又不能从源树处为它找到一个修复。在这种情况下，第一步就必须确定要解决什么问题，然后再修改、测试来证明是否要解决该问题，最后再看这样修复问题是否还需要什么。在这种方法下，最后一步是最难的。例如，从几千个 bug 中发现所检测的错误是否是存储器溢出，从而到处都是回归测试。

◆ 调试类型：插入式。

◆ 调试时间：短期调试/长期调试。

◆ 应用层次：单机系统、开发服务程序、QA 服务程序。

◆ 具体应用：种植错误法通常在问题已经存在的情况下使用。也可以在调试过程中的发现问题阶段使用，这时仅用来证明在碰到底层错误时对程序动作的假设。种植错误法不仅是测试方法，也是一个很有用的探测技术。

在应用系统调试中使用种植错误法的最简单方法是提供一个能被调用的内部方法，这个方法将直接调用要测试的函数链，还提供了要测试的数据值。例如，如果有了一个方法，这个方法执行打开文件操作和读取数据值，你提供的高层输入是一个空的文件名、一个无效的文件名或者是一个只读文件的文件名。因为高层函数在调用底层功能前就被调用了，此时如果在向上（或向下）的路径中发现错误，那么肯定也会遇到错误检测。另一方面，如果代码依靠检查 GUI 来保证用户的输入，但在代码中调用底层的函数，这样就不会有错误检验，

那么以上的错误仍可能出现。

9.4.13 编译检测法

用的最多的且最有效的调试技术，是使用最大的警告集去简化编译程序。你可能会对这种发现问题的办法提出怀疑。其实，可以通过这种方法来发现一些简单的问题，例如强制数据类型转换、函数(方法)参数错误或遗漏、未初始化变量和函数中未使用的变量。Brocklehurst 的一项研究表明，未完全使用变量的函数或方法，比起完全使用所定义变量的函数，会产生更多的问题。另外，如果编译器没有最优化未使用的变量，就可以释放一些存储空间。还有，如果删除了未使用的变量，代码就变得清楚、易读。看以下的 Visual Basic 代码，如果首先没有读懂代码就去运行，就会对结果产生怀疑：

```
START_MONTH = 1 ; Default to January
IF START__MONTH < 3 THEN
    DO_FIRST_TWO
ELSE

    DO_LAST_TEN
END IF
```

无可否认，前面提到调试程序比编写程序更困难。在正常情况下，是很难去发现错误的。上面代码的问题在于第二个变量中有两条下划线，而第一个变量只有一条下划线。可能不会马上明白 IF 语句执行条件不成立的分支。不过，当使用一个好的编译器进行编译时，将要求对所有的变量进行定义，这样就会立刻显示出第二个错误（只要在程序中没有另一个名字为 START__MONTH 的变量）。

利用编译器进行错误检查是一个简单的办法，特别是应用在源代码语法正确性的检查中。在任何情况下，这种办法都不会损坏程序，并能为追踪源代码进行查找错误节省时间，因为代码在编译阶段就进行了正确性检查。

- ◆ 调试类型：非插入式。
- ◆ 调试时间：长期调试。
- ◆ 应用层次：单机。
- ◆ 具体应用：把编译器设定成“最多错误检查”，把所有的警告都当成错误对待，确定所有的文件都编译正确，接着核对编译器的所有输出。如果可能的话，使用“lint-like”检查源代码来监视所有可自动检测的错误。

9.4.14 整体考虑法

就像我们前面章节所提到的，整体考虑法就是从整体上来考虑程序，而不是把程序看成是行的集合、代码的集合或子系统的集合。整体考虑法要求把源代码的 bug 放在整个完整的系统中来考虑，而不是只考虑模块或一行代码中的单个问题。整体考虑法的优点是可以预见到一个模块的修改对整个系统的影响。

虽然整体考虑法不是真正意义上的调试技术，但我们把它当成一种调试技术，那是因为它和其他的调试技术结合起来使用是很有效的。例如，当使用代码消去法追踪一个特定的奇异问题，就要考虑到消去的代码对系统的影响，只有这样，才不会对消去代码后系统出现的完全不同的结果感到惊讶。整体考虑法要求掌握系统的整体，并且要认识到引起问题的原因。知道所给状态将引起的结果对于发现问题是十分有用的。

整体考虑法没有一个固定的应用过程，必须了解系统所有的组成部分，理解这些不同部件的接口。整体考虑法还能发现“标准”的调试技术不能发现的出现在两个子系统之间的问题。这些问题（称为接口错误）经常发生是由于两个不同的人用两个不同的思想开发两套不同的子系统。接口错误通常是很难追踪的，因为它们不是在接口的一边产生的。而且，这个问题总是被假设发生在接口的一边，并且不会被传到接口的另一边。

存储器溢出是一个简单的接口问题例子。假设在模块 A 中，有一个过程分配到一个存储块，并把它放在列表中传给模块 B，模块 B 认为列表中的存储块是静态的（不是分配的），或者要注意到列表中存储块的消除过程。但是，模块 A 假设一旦为列表分配了存储块，它就属于分配列表的函数调用者，这就引起了系统存储器溢出。这个 bug 是由模块 A 分配了存储空间而没有释放空间引起的吗？不是的，因为模块 A 认为存储器是属于调用者的。那么 bug 是由模块 B 没有释放列表中的存储空间引起的吗？也不是，因为模块假设了释放了列表就释放了列表中分配的所有存储空间。对这个 bug 的惟一解释是列表例程本身，它应当释放所包含的所有存储空间。不过，这刚好和列表的原理相反，它只是像一个容器一样接收放进来的元素。

在问题上使用整体考虑法，必须追踪每个模块的输入和输出、系统运行的过程和调用模块的结果。在这些条件下，观察模块 A 分配列表后系统存储空间的减量，接着观察到模块 B 释放列表后系统的存储空间是不是增加相同的数量。通过对整个系统的观察，就可以知道问题出在列表的存储器分配上。那么就可以让模块 A 和模块 B 的开发者通过“商议释放存储器功能的实现”来解决这个问题。

- ◆ 调试类型：非插入式。
- ◆ 调试时间：长期调试。
- ◆ 应用层次：任意。

只要对系统的每一层都充分理解，这个方法就无所谓应用在系统的哪一层上。

9.4.15 使用不同操作系统上的另一个编译器

有时，一些本应当能够被编译器发现的错误却没有捕获到，不能捕获到错误是因为编译能力的不足，或者是因为在编译阶段编译器被作上了特殊的标志。如果代码在不同的操作系统中是可移植的，那么就可以通过试图使用其他系统的编译器来查找没有被原来编译器发现的错误。当然，这有个前提，不能在系统中使用特殊编译器代码，并且所使用文件（或其他配置脚本）能够很容易地不同系统中进行移植。

为什么要使用其他编译器？有以下几个必须考虑的原因。第一个原因（可能是最不是理由的理由），那就是怀疑所使用的系统编译器或代码产生器出现了问题。如果出现这种情况，那么用其他编译器检查问题就能找到错误，否则，就没法对错误进行跟踪，因为少数的编译器错误所产生的 bug 能被跟踪。使用另一种编译器是不得已的办法。第二，在两个不同的操作系统中查找可移植的问题，在这种情况下，系统可以在一个系统上正常工作，在另一个系统上却不能正常工作，使用不同的编译器可能会发现某些处理结构上的不同点。这种不同可能是由代码或数据在内存中排列不同引起的，可能是由某些代码的副作用引起的。在这种情况下，使用不同的编译器能发现问题。

在另一个操作系统上，同样会出现不同的问题。例如，某些操作系统能忽略内存错误，尤其是在较老的操作系统中，像 CP/M 和 MS-DOS。一个在 MS-DOS 上运行完好的程序（由于不同的硬件配置，出现了奇偶校验错误）在 UNIX 系统上运行就崩溃。简单地把代码移植到另一个操作系统，就能立刻查找出问题，节省了大量的调试时间。

在什么时候要考虑使用不同的操作系统或编译器法？如果正在编写一个可移植的程序，而且已经在多个操作系统上运行。在这种情况下，就没有必要使用另一个编译器去检验警告和错误。因为使用这种方法不仅不能发现其他有用的信息，也不能从特别的编译问题中摆脱出来。如果是在多操作系统平台上编程，当以下条件成立时，就应该考虑这个方法：

- ◆ 能把问题简化成简单的代码行或函数。
- ◆ 已经检查了引起问题的代码，并且没有异议。
- ◆ 怀疑编程人员没有使用正确的编码技术，需要用不同的编译器来刷新。
- ◆ 需要测试函数或方法的可移植性。

如果没有以上这些条件，使用这个调试技术就可能没有多大的用处。

- ◆ 调试类型：非插入式。
- ◆ 调试时间：长期调试。
- ◆ 应用层次：仅适用于内部。

9.4.16 每次改变一个变量

为了尽快发现和修改 bug，我们经常会把注意力集中在我们认为有问题的地方，以至忘记了基本的工程性原则。这就是在一个时刻只能改变一个元素。但一个很普遍的现象，在发现并解决一个重要问题时，每次改变一个变量法经常被忽略。

这个方法为什么这么重要？最根本的原因是因为不知道通过改变哪些因素可以解决问题。另一个原因是通过同时改变多个元素可能会产生新的问题或增加现存问题的复杂性，这就使你忽略了改变条件引起的问题，也使你不得不继续发现已经存在的问题。所以，在进行修改时必须每次只改变一个条件。

在改变变量时，还有一个重要的因素是在进一步的修改前要还原被修改的变量，这点和每次改变一个变量一样重要。因为已经通过了代码中的一个修改，可能会产出一个新的问题。

另外，当进行库中源代码的最后检查时，可能还会在系统中发现很多没用的代码。所以在进行进一步的修改时，要取消上次的修改，回到代码的起始状态。如果源代码控制系统支持这样做，就可以简单地通过还原校对文件来实现，然后对原始代码进行重新检查。

每次改变一个变量，可以使人们在给定的时间集中注意力来验证他们的假设，既不用担心问题的其他因素，也不用担心会改变所要调试的问题，或者会产生新的问题。不过，一个主要要考虑的问题是：一旦作出修改，就要测试它，抛弃它，也要抛弃它的推理。使用和第一次相同的推理对程序作出第二次修改来发现问题，而没有发现第二次修改有它和第一次修改完全不同的原因，这是没有道理的。

- ◆ 调试类型：插入式。
- ◆ 调试时间：短期调试。
- ◆ 应用层次：非成品软件。

9.4.17 数字命理学和边界条件

虽然数字命理学听起来很陌生，事实上它就是研究数字的科学。在数字研究中一个有趣的现象是某些数字不能在一起很好地工作，这就引入了数值“边界条件”这个概念。举一个简单的例子来说明我们的讨论会比引入低层的数学概念来说明更容易点。

看下面一个计算正方形面积的例子。这个公式是：

$\text{Area} = \text{length} \times \text{width}$

这就意味了在计算一个长方形（一个特殊的正方形）面积时，就把它一边的长度乘以另一边的长度。面积有一个特性，它必须是一个正数，而不应当是负数。也就是说，正方形的面积不会比它的边包含的空间小。不过，当我们在实际计算面积的过程中就会产生问题。例如，考虑表 9.1 所列长方形各点的坐标 (x,y)。

表 9.1 示例 x, y 对

x	y
-1	-1
-3	-1
-1	-4
-3	-4

表 9.1 是正常坐标轴上的负数点，如果要计算边的长度，按照正常的计算方法，将会发现有一边的计算结果是负数，这就使面积的计算结果是负数。我们已经讨论了面积的结果不可能是负数，为什么会得到一个负数的结果？显然，当用计算机程序进行长方形面积计算时，计算公式应当是：

$\text{Area} = \text{abs}(\text{length}) \times \text{abs}(\text{width})$

在这种例子中，等式中的 **abs** 表示绝对值，总是返回一个正数或 0。那么总是返回一个

正数值，因为是两个正数值得乘积。

怎样在调试中使用这个方法？当然有好多途径。首先要知道所处理数据的值域，如果程序中有一个交通灯的变量，那么这个变量值域就应当是颜色的集合{red, yellow, green}。如果值是 pink,那么就说明结果是错误的。当我们读程序的源代码时，应当清楚所处理变量的数学或逻辑的值域。

在调试一个应用系统时，边界条件是另一个要考虑的重要因素。边界条件可能引起程序在某种输入集下不能正常工作。在前一章，我讨论过一个卫星电视接收系统，这个系统在硬件设备上存储程序数据。它的工作过程如下：从卫星上接收数据，存入固定大小的存储器缓冲区中，然后写入硬盘。另一边，数据从硬盘中读出，放到存储器的缓冲区中，传给输出设备，通常是电视。大部分的时间，系统的工作是正常的，数据流出，存到硬盘，然后传给输出设备，电视的工作是不间断的。不过，在有些时候，电视图像在短时间内会变得断断续续的。研究的结果表明这个空白是由于缺乏输出数据流而引起的。进入系统的数据传输率总是不为零的，所以总有数据进入系统并传给输出设备。那为什么会出现断图的现象呢？

通过对问题的研究，我发现这个过程的边界条件是输入端的缓冲区在填满时才传送数据给硬盘。设计者没有考虑到输出缓冲区由于和输入端没关系而没有填满。这个特别的条件是在高速输入之后可能有非常低速的输入，引起输出缓冲区变满而输入还没填满。输出过程就有可能读空输出缓冲区，但输入缓冲区还没准备输出。这就引起输出过程暂时的停顿，出现空白现象。

很难从代码中发现数值和边界条件问题。为了发现这些问题，必须从原始的需求和算法中发现可能存在的条件。通过代码中可能引起边界问题知识的帮助，可以检查系统中的实际源代码，观察条件是否适当地被处理。如果没有，就可以用源代码中的特殊条件来证实对问题的假设，可以根据系统和问题的严重性出发。经常，对数值和边界条件的真正解决办法是重新设计和编写系统。但这个方法被排除，因为经常不得不砍掉处理额外条件的代码。这确实不是最好的解决问题的方法，你必须对所删除的代码作详细的记录，只有这样才能最后解决问题。对大多数的边界条件，记录长期调试中的问题是首选的解决问题办法。

- ◆ 调试类型：非插入式。
- ◆ 调试时间：长期调试。
- ◆ 应用层次：任意。

9.4.18 检查最近的修改

如果正在调试一个现成的系统，这个系统在测试前能正常工作，在测试时突然出现问题。该怎么调试呢？

- ◆ 假设这个问题总是存在，并查找引起问题的原因。
- ◆ 假设测试条件对问题没有影响。
- ◆ 假设最近的修改使系统出错，导致 bug 突然出现。

如果是第一个现象，问题可能可以被发现正确，但要花费很长的一段时间发现根本问题；如果是第二个现象，就要寻找一个全新的从未使用过的方法；如果是第三个现象，就要对以下的调试技术有所了解。修改，特别是最近的修改，通常会突然使原来工作正常的代码出现问题。当向编程人员询问有关修改，得到的答案几乎都是“我修改了<x>，但没有任何作用”。

如果用有一个修改控制系统，就不难发现因为修改引起系统突然崩溃的原因。如果没有这样的系统，工作起来就有一些难度，在这种情况下，可以试试以下的办法：

- ◆ 在源代码中查看所有文件的时间和数据，和最近的进行比较。
- ◆ 使用文件比较工具来比较，发现有可能出现问题文件的两个不同版本。
- ◆ 向项目组中所有的成员询问他们最近对文件的修改情况。

可以发现，系统自动保持最近修改的足迹，并且可以看到对这些修改的各种报告，这些信息比用户手册更有用。如果没有这样的系统，在调试过程中就要考虑更多的因素。

不仅仅考虑系统中源代码的修改，还要考虑系统运行环境的改变。如果没有对源代码作任何修改，但还是在代码中突然出现问题，可以考虑系统运行环境的改变。在使用这个方法时，必须知道对环境做了哪些修改。当安装一个系统时，要注意当前系统的环境（包括操作系统的版本和应用软件的版本）。如果有记录这方面信息的工具，也可以用这个工具测量当前的环境来发现变化。通常情况下，网络的变化、操作系统补丁、第三方工具新版本和机器的硬件的改变都会引起问题。为了测试系统，通过跟踪记录中的所有的信息，就可以证实对系统的修改引起的系统的失败，然后开始寻找可能的影响。

如果问题是发生在外部系统，这并不意味着源代码没有问题。相反，恰恰说明问题很可能就出现在源代码中。在这样的情况下，应当用变量修改的知识重新回到调试的开始步骤来发现 bug。

- ◆ 调试类型：非插入式。
- ◆ 调试时间：长期调试。
- ◆ 应用层次：任意。

9.4.19 清除系统中的“死代码”

在调试系统时，一件最讨厌的事是在代码块中企图发现为什么代码不能按编程者的意图执行，而且能发现系统几乎不调用代码。看以下一段 C 语言代码：

```
for ( i = 0; i<10; ++i )
{
    DoSomething();
    if ( i == 8 )
#ifdef DEBUG
        DoDebugSomething();
#endif
#ifdef MIPS_CHIP
        DoMipsStuff();

```

```

#endif
#ifdef INTEL_CHIP
    DoIntelChipStuff();
#endif
}

```

不管这段代码的可读性，你能知道这段代码在运行时的执行情况吗？可能不能，尤其当你不是非常清楚 MIPS_CHIP、DEBUG 和 INTEL_CHIP 的意思时。它们可能是过程类型和调试模式，也可能和这些内容毫无关系。这段代码可能包含 bug，也可能没有 bug。这都很难说，即使你知道 DoMipsStuff 函数包含一些错误，但它可能没被调用。

如果你认为前面这段代码不是很混乱的话，请看下面一段代码（来自实际系统）：

```

if ( theCell != nil ) {
    long row = theCell->GetRow();
    long col = theCell->GetColumn();

    #if !defined(UPDATECHECKMARKUSESCRIPT)
        ulong bitNum = (col * kTVChannelTableRowCount) + row;

    if ( fMultiSelect )
    {
        if ( fIsCheckedBits & (1U << bitNum) )
            fIsCheckedBits &= ~(1U << bitNum);
        else
            fIsCheckedBits |= (1U << bitNum);
    }
    else
    {
        if ( fIsCheckedBits & (1U << bitNum) )
            fIsCheckedBits = 0;
        else
            fIsCheckedBits = (1U << bitNum);
    }
}
#endif

theCell->SetDisabled( false );
theView->ExecuteSelectable( theCell );
theCell->SetDisabled( true );

#ifdef UPDATECHECKMARKUSESCRIPT
long channel = fChannelTable[row][col];
UpdateIsChecked(channel, col, row);
#endif

```

这段代码的问题在于设置或清除标志。虽然不能从详细的注释中知道，但这是个基本意图。注意到这两个不同的代码块，一个完全嵌入工作，另一个调用完成相同工作的函数。当要调试这段代码时，我首先花大量的时间检查函数条件，因为我不知道定义符号 UPDATECHECKMARKUSESCRIPT 的地方。后来，在包含的定义文件中找到，它被包含在

工程的 MAKE 文件中。现在，你知道我想在这个环境下说什么了，我在嵌入代码中发现了问题。

怎样才能发现这样的问题？最简单的办法是使用像 C 语言这样的预处理程序生成代码，显示在最终系统运行的代码中，也可以在工作平台上用调试器查看代码。这样，就可以简单、容易发现问题。如果所使用的编程语言没有预处理程序，那么可能就要手工进行代码预处理或者写一个程序来解决这个问题。把代码量减到必须查看的部分，就会增加在代码中发现问题的机会。

有时，你可能会想：“是的，但是我不用有预编译器的语言，我就不会出现这样的问题”。可是，类似的问题会出现在很多语言中，而不管编译器是否实施。看下面一段 Visual Basic 代码：

```
; Uncomment the correct value for your machine
const Machine=1          ; Intel
REM const Machine=2      ; MIPS
REM ;const Machine=3; // Alpha
If ( Machine = 1 ) THEN
BEGIN
    Do_Something
END IF
If ( Machine = 2 ) THEN
BEGIN
    Do_Something_For_Mips
END IF
IF ( Machine = 3 ) THEN
BEGIN
    Do_Something_For_Alpha
END IF
```

在任何操作系统的任何语言中，都会出现相同的问题，在 C 和 C++ 语言中更经常出现。这是因为编译器允许直接条件编译，并且 C 语言在代码说明部分经常使用 #ifdef 机制。就像：

```
#if 0
    // Some code to comment out
#endif
```

在功能上等同于：

```
// Some code to comment out
/* Code */
```

- ◆ 调试类型：非插入式。
- ◆ 调试时间：长期调试。
- ◆ 应用层次：任意。

9.4.20 问题假设法

程序员在测试他自己编写的代码时往往很困难，一个原因是他总是被程序开发阶段的基

本假设所限制。在调试代码时，特别是自己写的代码，为了能查找出真正的问题，就应该去掉这些假设。假设可以简化问题，例如“这段代码的调用和 if 语句有关”；假设也可以使问题复杂化，例如“进入系统的最高数据传输率是<x>bytes 每小时”。如果假设不正确，建立在这个假设上的代码将都不正确，这就引起系统的 bug。

在调试代码时，我们如何避免作出假设？为了解决这个问题，必须雇用没有参加编程的人来调试。即使对所给子系统、模块或函数的工作很了解，你必须确定它们的功能和你想的一样。不要轻易相信任何一段代码。例如，考虑以下一段 Delphi 代码：

```
X := 1;
Y := 2;
PrintOutXAndY();
X := X + 3;
Y := Y + 4;
```

如果你是一个 Delphi 程序员，可能一看到这段代码执行后，就认为 X 的值是 4，Y 的值是 6。毕竟，函数 PrintOutXAndY 不会改变这些值，真是这样的吗？这是一个假设，很可能是错误的。例如，如果 PrintOutXAndY 函数像以下这样：

```
Function PrintOutXAndY() : Integer
Begin
    X := X + 1; (* Adjust for offset in file *)
    Y := Y - 1; (* Adjust for line count *)
    PrintLn X;
    PrintLn Y;
End;
```

在这个情况下，很明显，函数 PrintOutXAndY 执行后，X 和 Y 的值就会改变。因为外部原因，它们在函数中作了调整。很难说为什么程序员在函数中改变系统变量的函数，但这样的副作用并不是像你想象的那么少。在调试时，不要假设任何情况，除非已经证明了假设。一旦开始一个例程，并且证实了这个函数或方法对所观察的数据没有影响之后，就可以放心忽略它。

另一个可能引起问题的情况是使用了一个“这不会引起问题”的假设。例如，看以下一段 C 语言代码：

```
int vars[MaxVars];
char theString[20];
GetFormTitle( FormID, theString );
for ( int I=0; I<MaxVars+1; ++I )
    vars[I] = 2 << I;
Form.Title = theString;
```

我们调试的 bug 是：窗口的标题总是空白。GetFormTitle 方法设置的变量是正确的，但不在窗口中显示。如果要做假设，首先要假设窗口标题没有正确存储在数据库中。另外，也可能假设如果窗口标题显示装载正确，而后面的代码把它擦掉。毕竟，在这一部分代码中不会出现这种情况。

这些假设是正确的。事实上，GetFormTitle 例程返回正确值。作出这些假设是因为我们

在后面的代码中没有改变字符串，它肯定是正确的。如果是在单机状态，并且使用调试器，就能很容易测试这个假设。在程序中增加 `theString` 变量来观察，一步步运行代码，肯定能在某点上发现变量的改变，为什么？这有点复杂，但是这段代码确实显示了要去解决的问题的所有信息。

当然，还需要一点经验和一点 C 语言编程经验。注意到 `vars` 数组被定义成固定大小 (`MaxVars`)，也注意到循环进行了 `MaxVars` 次。在 C 语言中，数组的范围是 0 到下标-1。这就意味着在最后一次，循环超出了数组范围。在 C 语言条件下，超出的指针指向下一个变量（通常是这样的，会因为编译器不同而不同），这正是 `theString` 变量。这就意味着一旦访问 `theString` 变量的地址，就消除了该地址的内容。结果，当分配 `theString` 给窗口标题时，就引起问题。调试这样的问题的惟一办法是一步步运行代码，检查所有有问题的变量。如果假设这个代码段不会改变变量，在一开始时就不可能发现问题。

- ◆ 调试类型：非插入式。
- ◆ 调试时间：短期调试/长期调试。
- ◆ 应用类型：任意。

9.4.21 检查未测试的代码

在任何大规模系统中，总会有一些代码没有被测试。可能是因为没有测试工具，也可能是因为代码太简单，不必要在测试平台中进行测试。未被测试的代码经常是引起问题的主要部分，因为没有人真正测试它来看是否有问题发生。未测试代码的一个好的例子是还没有公开给用户。

例如，假如你在函数库中查找系统所给的一个特征。库的开发人员尽可能地提供完整的包，增加了好多功能，这些功能在第一版中没有公布。在下一个版本中使用这些代码，来执行开发者所期望的功能。不过，在下一个版本测试时，这些新的功能有几个 bug，并且不容易在增加到系统中的新代码中发现。

在这种情况下，在系统中出现问题通常会假设是增加到系统中的新代码引起的问题，这就和“检查最近修改”技术相关。不过，并不总是这样的。在这个例子中，这个问题已经在代码中出现很长时间了。然而，这一部分的代码没有被测试。如果知道这些代码从来没有被测试过，就可以在这些地方查找问题。当然，问题在于你怎么知道哪些代码经过测试，哪些没有被测试。

有不同的方法解决这个问题。第一个，也是最难的一个方法是查看所有的测试条件，用手工进行所有函数、子系统代码分支测试。这是一个值得努力的方法，但我从来不这样做，我倒是支持别人做所有这些工作。我的方法是使用系统的测试驱动和代码覆盖工具哪些能使用，哪些不能使用。如果没有测试工具来测子系统，可以用另一个工具——交叉索引 (`cross-index`)。这类工具能显示代码调用函数的情况。如果在前一版本中的某一部分没被使用，但在后一个版本中使用，就可以在这一点上开始测试。

提示：使用代码覆盖法和交叉索引工具，来决定哪些代码被使用。

对这个技术的一个讨论是对我们使用代码的理解程度。虽然用来确定这个信息的工具已经出现很长一段时间了，但很少使用在探测工作中。相反地，代码分析工具比如交叉索引，被用来发现能被安全从系统中删除的“死代码”。这些工具的另一个用途是发现系统最经常使用的部分，这样就可以对这些部分进行最优化。虽然这些目标很有价值，但这些工具很少在实际调试中使用，因为会占用我们大量的时间。如果在调试中提前使用这一方法，就会得到一个需要较少时间解决问题的系统。

- ◆ 调试类型：非插入式。
- ◆ 调试时间：短期调试/长期调试。
- ◆ 应用类型：任意。

9.4.22 不变式法

不变式是系统在任何时候都必须正遵循的状态。也就是说，不变式是系统运行的原则。当应用程序处理总是时，错误条件不会也被进行适当的处理。当不变式规则被破坏，系统就会崩溃。

在任何系统中，不变式的有以下这些情况：足够的装载程序的存储空间、足够的保存程序运行状态的磁盘空间以及那些记录进入系统、修改系统数据的硬件设备。为了使程序在开始时就运行，这些规则必须工作。不过，在一个给定类型的系统中，可能不用某些不变式规则。例如一个记账系统。在这个系统中，要求系统在任何时候都要保持平衡。也就是说，在任何时候，贷出的款项加上库存量必须和存款总数相等。对这各系统的另一种描述方法是，对于记账系统，资产必须和债务相等。如果这个条件被破款了，那么系统就会出现严重的问题。

不变式是系统在技术上的要求，例如硬件设备的有效性，就像一个连接在计算机上的“dongle”。不变式也是交易的规则，就像一个用户在登录系统时必须给出一个标识。另外，不变性也是程序赖以运行的基础，所以它必须存在并且正确。例如，考虑一个交互电视系统列表条件。这个系统允许查找男演员、女演员和他们的头衔等。当然，这个系统要求有一些列表的数据，这样你才能进行查找。所以，列表信息的有效性就是系统的不变式。

现在，已经知道了不变式，那么怎样在调试工作中使用这个知识？当我们发现系统的严重问题是不变式的错误引起时，解决问题的最好办法是确保不变式保持不变。为了达到目的，所收集到的数据在系统中必须是正确的，然后对他们进行简单的诊断。这个诊断例程是从代码的安全部分中调用出来的。安全部分就是被有规则调用的代码，但不会影响任何产品级操作。注意，如果在成品中放置这样的函数，那么就要求处理适度的错误，而不是简单地退出系统。

一个错误处理的可能是通知管理者（在客户/服务器环境中）或用户（在单机系统中）系统已经出现了错误，那么他们必须关闭系统或立刻解决问题。问题能不能解决有赖于不变

式能不能被解决。如果系统要求有一个文件，但文件又不在系统中，那么就可以解决；如果在一个结构中的“magic cookie”要求包含特殊的值（例如校验和），但这个值不存在，用户就不可能解决问题。解决方法可能不同，但要小心出错。

在调试过程中，不变式是发现系统严重问题的好方法。如果能发现一个不变式被修改，就能回溯系统并且发现引起这个改变的原因。知道引起问题的根本原因是发现引起改变和正确解决问题的关键。

- ◆ 调试类型：非插入式。
- ◆ 调试时间：短期调试/长期调试。
- ◆ 应用层次：任意。

9.4.23 存储器使用情况

当调试一个出现奇怪问题的系统时，两个最有可能引起问题的原因是指针和存储器错误。在这两个原因中，存储器错误更容易被发现，因为系统整个可利用的存储器数量是知道的。虽然发现指针错误是较困难的工作，但发现存储器错误是相当容易和直接的。

如果追踪系统使用存储器数量，就很容易发现存储器错误。从这一点上来说，发现存储器溢出的位置会更困难一些。为了这样做，必须知道存储器什么时候被分配和释放。有几个途径可以解决这个问题，这要根据所使用的语言、编译器和操作系统来决定。在很多情况下，不必要做额外的工作，在语言或操作系统的标准库中存储器分配释放例程能用调试版本的相应例程来代替。

最后一个调试存储器的技巧是在程序运行时监视被分配的存储器的总数。一个特殊的诊断组合图会在系统运行时显示存储器使用值的窗口。在调试前，可能就记下程序日志中自由存储器的数量，以保证能按你的意图来做。

不过，记住程序“吃掉”存储器并不包含存储器溢出。有时，系统缺少存储器是设计的错误。例如，考虑你分配一点存储器来跟踪用户输入系统的字符，这样就可以记录和取消这些命令。在你保持的用户列表中没有保持一个数目的上限，就经常有可能超出系统存储器的范围。在这个情况下，不是存储器的漏洞，因为你一开始就没有释放存储器。在这种情况下，扫描存储器分配例程是发现和解决设计错误的最好办法。

像前面提到的，不要以为这个问题简单，因为所使用的语言有时没有直接分配存储器，像 Visual Basic，或没有自己的回收程序，例如 Java，那么你将和存储器分配没有关系。即使这些语言也有分配存储器的方法。另外，支持存储器分配的任何语言会出现无限的存储器分配设计问题。

- ◆ 调试类型：非插入式。
- ◆ 调试时间：短期调试/长期调试。
- ◆ 应用层次：任意。

9.4.24 互斥

mutex 来自“Mutually Exclusive”的缩写形式，它表示系统中的一个看守目标。虽然每个实体有互斥机制，其他等待正确访问互斥的实体必须等到实体操作的完成。互斥对程序的构造特别有用，特别是在多线程的环境下。它和软件的调试有什么关系？

在调试一个多用户或多线程的环境时，经常会发现问题不容易被解决：因为一些代码每当从两个不同地方同时使用时，就不能正常工作。例如，考虑以下一段代码：

```
int func()
{
    static int index = 0;
    extern char gGlobalArray[ 100 ];

    for ( int i=index; i<100; ++i )
    {
        call_a_function( gGlobalArray[i] );
    }
    index ++;
}
```

这个特定的函数有几个问题。让我们观察两个较严重的问题。第一，这里使用了一个外部全局数组。第二，包含了静态变量。如果不是用 C 语言编程，一个静态变量在调用函数中保留它自己的值。不过，正常情况下，这没有严重的错误。如果一个函数在不同的地方被多次调用，内部状态寄存器将自加 1。不过，当这个函数出现多个重入的问题，重入是指在函数最后的调用完成之前就被另一个线程或例程调用。这时不会出现问题因为对局部变量的复制只出现在第二个调用进程自己的空间内，静态变量只出现在全局存储空间内，而且不被复制。同样，在这个函数中的全局数组存在于这个函数的外部程序空间里。但是当出现函数的第二个调用时，就出现了问题，因为程序的两个不同线程的两段代码在同一时刻使用相同的空间。最好的情况，程序结束时得到不正确的结果。最坏的情况下，可能出现因为存储器错误而导致程序崩溃。如果感到好奇，就修改这个来自实际系统中的特定代码。

互斥有助于定义状态的类型。根据编程的语言和操作系统，互斥可以在系统层上执行。在任何情况下，如果怀疑前面函数的问题是出于多个线程在同一个时刻访问函数，那么就考虑这种情况将出现的现象。我们在函数中添加一个互斥对象。这个互斥能屏蔽任何第一个调用在操作过程中出现的任何其他调用。这样，这个程序运行效率很慢，但不会出现崩溃。

互斥能用来解决问题，或者能发现问题的存在。而真正的解决办法要依靠于软件工程师。如果能在多线程的环境中生成一个单线程函数，那么互斥是一个正确的调试方法，它可以用来发现问题产生的地方，也可以正确解决问题。

- ◆ 调试类型：插入式。
- ◆ 调试时间：短期调试/长期调试。
- ◆ 应用层次：多线程成品系统。

9.4.25 显示系统的运行情况

在调试一个出现 bug 的系统时，在检查源代码的同时显示系统的运行状态，是一个好的测试源代码的技术。同样，也可以在监视系统运行时查看源代码的行为。在任一种情况下，都可以看到系统工作的情况。就像我们在书中强调的，当对系统工作的情况和可能工作的情况不能直接了解时，就不可能充分调试系统。

怎样从源代码中观察运行的系统？为了这样做，必须把源代码的高层子系统映射到运行系统的深一层功能性代码上。这个映射让我们看到了个别被调用的构件，这些构件的每个功能都被用户调用。让我们来看两个简单的例子，来观察系统中的一些行为。

首先，看一个运行在单机环境下的例子。当用户从主菜单中选择 File | Open 时，程序就调用菜单处理例程来调用适当的函数来响应菜单项。这个菜单处理例程显示打开文件对话框，让用户选择要打开的文件。这个例程真正在物理上实现了打开文件，并装载被调用文件的信息。为了讨论的方便，我们把这个例程的执行分成以下几个步骤：

1. 物理上打开文件。
2. 装载文件头，并证实它是正确的文件版本。
3. 如果文件头不正确，显示错误并停止程序。
4. 如果文件头正确，就把独立的文件构件装载到存储器中，并生成一个新的显示。

当在观察系统打开一个文件时，就必须考虑以上这些操作步骤。这对你来说很重要，有以下几个原因。第一，当系统在所给条件下不能正常工作时，你就知道出现了什么问题。第二，你也会知道下一步会发生什么事，这样就能在出现严重问题（例如系统崩溃）前注意到问题。第三，通过知道系统的行为，你就知道将会遇到哪一类问题，出现什么错误。在前面这个例子中，我们就知道在打开文件时引起程序崩溃的原因是不正确的文件格式。在所有的可能中，知道了系统执行的阶段，如果第 2 阶段不正确定义文件是有效的，这个问题要么出现在第 2 阶段，要么潜在于第 3 阶段。

另一个观察程序的运行是显示逐步调试源代码的情况。例如，你可能在源代码中注意到，物理上打开文件后，状态条显示的信息告诉用户文件正在装载。接着，你看到系统显示了“等待”，接着文件显示在屏幕上。知道了事件的序列，就可以观察在文件内容显示到出现让用户等待信息这段时间内系统的动作。这就要求识别执行代码的所有步骤。例如，如果知道程序在某点上崩溃，就可以把它和相应阶段的代码联系起来。

知道了系统运行状态和代码的内部状态之间的关系对发现和解决代码中的问题是很关键的。但是，即使正确地知道将发生的事，也不可能解决错误，除非知道代码出现问题的地方。同样，也不能确定对代码的修改能解决问题，除非确实知道修改源代码对系统的影响。

- ◆ 调试类型：非插入式。
- ◆ 调试时间：长期调试。
- ◆ 应用层次：任意。

9.4.26 和工作的系统进行代码比较

在修改源代码时，出现的最普遍的问题是程序突然停止工作或突然出现奇怪的行为。如果仅仅是在系统中增加或修改源代码，这些代码就最有可能成为检查的对象（参照“检查最近的修改”）。不过，有时完全不清楚为什么所做的修改会影响到整个系统。在这种情况下，将出现问题的系统和正常工作的系统进行比较是很有用的。可以和前一个稳定版本的代码进行比较，也可以和函数或其他执行相同算法的正常工作的代码进行比较。

例如，你要重新执行一个代码子系统来优化它，并整理子系统的全部源代码。当执行完代码之后，你发现只能在特定的条件下才能正确执行。在这个情况下，你已经有了一个工作正常的代码版本，并且知道在修改代码之前系统工作正常。让前一个版本的系统工作，就可以通过这个版本代码的运行来发现这个版本代码的动作。例如，可能发现在特定的状态下有一个特殊条件处理。你可能发现这个边界条件对于所选择执行的算法不起作用，或者是对输入数据的类型不正确地执行。在任何一个情况下，都要检查源代码来寻找解决问题的方法，这样才能使系统像原来一样正确执行。

另一个可能是现在执行的代码确实存在没有考虑的副作用。有时，在代码中经常出现这种情况，像“spaghetti effect”来发现已经被详细描述的模式。软件调试者要发现在模块间的清楚边界，可能要求一些技巧去处理某些事件，这些事件是开发者没有考虑到的，或者是没有设计到的，因为其他的特征在这个时间没有出现。一个系统在开始设计时在模块之间耦合度很低，就成为相互链接的老鼠窝。这个作用，就是软件平均信息量，这就是经常重写软件的原因。

如果有一个现存的算法或子系统的执行来比较代码，你通常可以发现问题。另一方面，解决一个问题要求不仅要理解所执行的算法，还要知道这段代码怎样嵌入到整个系统中。当一个新的算法不能正常工作时，要考虑以下几个方面：

- ◆ 代码的边界条件。
- ◆ 两个模块之间的耦合度。
- ◆ 不同的输入、输出变量类型。
- ◆ 函数的不同返回类型。
- ◆ 代码的静态变量或状态框架。
- ◆ 调试类型：非插入式。
- ◆ 调试时间：长期调试。
- ◆ 应用层次：任意。

9.4.27 理解算法

很多软件系统在源代码中有各种算法。这些算法经常符合软件使用领域的“工业标准”。如果不知道软件执行的算法，就很难进行调试代码。

例如，让我们来看一个在 POS (point-of-sale) 系统中计算税收的算法。如果你和我一样，在第一次运行它时，你就会被搞糊涂了。你想，消费税根本不是比例，有一系列的记录需要填写。每个记录都有下界（这个记录的开始数目）和上界（这个记录的结束数目）。基本上，它这样工作：

1. 计算整个美元，然后计算相应的税率。
2. 计算部分的美元，加上每个记录总数，这些记录的开始值少于保留的美分，结束值大于总的美分的数目。
3. 相加的结果就是税。

我想这已经很混乱了，但我还是不能确定。这是怎样在实际环境下工作的：人们的税率是根据他们所选择的美分决定的。这就意味了购买 5 美分到 50 美分的东西时的税率是 8%。这当然不很清楚，但很多地方都这样。在其他地方，税率是固定的，也就是你所买东西的价钱乘以税率，然后四舍五入得到结果。

如果知道了这个例子的算法，在大多地方计算消费税的结果就清楚了。如果不知道这个算法，即使系统没有问题，你也可能认为有 bug 存在。同样，也可能认为没有问题，虽然系统有一系列问题存在。在系统中，算法通常体现了交易的格式。虽然源代码有可能对或者是错，但交易的规则必须是正确的，以保证系统的正确性。例如，如果你发现在实际环境中的所有 POS (point-of-sale) 系统所收的税太高，那么系统就不会得到广泛使用。而且，当政府机构发现这个软件问题时，你将要承担责任，并要受到处罚。

当在系统中遇到一个算法时，对付它的最好方法是阅读它的文档并用其他资源证实它（例如公司的交易分析）。另外，如果在系统中使用了一个来自外部资源的算法（例如从一本书上得到排序或查找算法），不仅要知道算法的工作，也要把它和原技术联系起来。来自工程外领域的其他人员能读懂原文档并发现是否存在问题。

- ◆ 调试类型：非插入式。
- ◆ 调试时间：长期调试。
- ◆ 应用层次：任意。

9.4.28 检查连通性

虽然连通性是很明显的问题，很多客户/服务器类型的 bug 是由客户和服务器的链接或者是服务器和末端服务器（像数据库）之间的链接的失败引起的。显然，在系统工作之前要对这些东西进行测试，但一些链接还经常会断，并且得不到验证。程序员经常认为程序正常工作，并且没有理由要对像数据库的有效性这样的事进行检查。连接错误很普遍，所以在查看代码前就要进行检查。

考虑下面一段代码，想一想会出现什么问题。在 Web 服务器中，或者是一群服务器中的一个，用户有时被拒绝访问，并给出“用户名密码错误，请重试”的错误消息。不论输入怎样的用户名和密码，结果都是一样。这就产生了 bug，奇怪，在机器重启后，错误就消失

了。这就是系统的现象。

所给信息不多，查看 Web 服务器上的代码，看下面一段伪代码（实际的代码没有给出，因为 Web Server 代码难读并且复杂）：

```
On Error GOTO HandleUserError
Extract User Name from HTML edit box
Extract Password from HTML edit box
Connect to UserDatabase
Check User Name in UserDatabase; Note that error will go to HandleUserError
for record not found
    Check Password in Record returned
    If Password <> Record.Password goto Bad User
    Go to Login Successful Page
HandleUserError:
    Display "User Name or Password is Incorrect, Please Try Again"
    End
Bad User:
    Display "Password does not match, please try again"
    End
```

在上面一段伪代码中，我们可以知道用户看到显示消息的原因，因为没有在数据库中查找用户名。这就产生了“记录未找到”错误，并跳到错误处理程序上。这条记录找不到，为什么？有很多原因。例如，存储的用户名可能是大写的，也可能在查找语句中插入了空格。我们可以在其他相同的语言集中工作，或者可以跟踪到记录的结尾。有几百种可能。

如果让我来调试这个错误，我首先不管错误操作，而是查找出确切的错误。例如，在程序连接到数据库阶段，我将跳到不同的错误处理。我将发现问题是计算机和数据库失去了连接。这确实是和其他错误一样严重，但和其他错误有很多不同。在调试过程的这个点上，我将开始查找权限问题、网络问题和其他类似的问题，例如条件转换错误或填充问题。

在客户/服务器系统中，比起其他系统而言，连接错误是出现最经常的错误。在出现的点上检查这些错误将为你节省很多花在跟踪奇怪问题上的时间和精力。

- ◆ 调试类型：非插入式。
- ◆ 调试时间：长期调试。
- ◆ 应用层次：客户/服务器，成品软件。

9.4.29 核心文件

在很多现代的操作系统中，一旦程序崩溃，就有一个“存储文件”记录了程序在崩溃点上的状态信息。在 Unix 操作系统中，这个文件称为“核心存储器”。在 Windows 操作系统中，他们通常是一些 Dr. Watson 文件格式。在任何情况下，他们包含了可以帮助你发现引起程序崩溃原因的信息。当你得到一个“核心存储器”文件或 Dr. Watson 格式的文件，都应该对这些文件进行认真地检查。

在我开发的一个嵌入式系统中，我发现一个好的特征是：核心文件能通过内部调试器在

系统中产生，并上载到监测公司的主要站点上。这些核心文件的信息包括引起系统崩溃的函数、系统崩溃的类型（坏的堆栈格式、空指针的分配等等）和崩溃时系统的状态等。虽然这些文件对发现问题和检查错误只有一点帮助，但它们对跟踪系统遇到的“奇怪”的问题是至关重要的。

在自己的应用系统中，也能生成相同的核心文件。看下面一段 C 语言代码：

```
char *x = (char *)malloc( someSize );
if ( x == null )
    assert( "Memory Allocation Error!" );
```

不用我断言，从这一点上可以看出，这些代码确实没有任何用处。如果出现了一个存储器分配错误，你能得到的信息就是简单诊断信息告诉你这个错误，并停止程序的运行。既不能得到被调用函数的状态，也不知道程序要分配的存储器的大小。相反，如果程序的显示如下：

```
Memory Allocation Error in function foo File: Bar.c Line: 200
Input: FileName="fred.dat" someSize=-3
Global Variables:
<gX=1>
<gY=2>
```

我敢保证，你就能从这样的显示中得到大量的有用信息，而不仅仅是“存储器分配错误”。在第一种情况下，你可能会认为是程序的运行超出了存储器的边界，并开始查找存储器溢出。事实上，从第二个显示可以知道，根本不是程序的运行超出存储器的边界。相反，是我们试图分配的存储数量是负数。如果这样做，标准函数 `malloc` 将返回 `NULL`。所以，是存储器分配错误。

为了执行这样的诊断，就必须保持每个函数的“调试”目标，这个调试目标必须包含函数的状态，也要包含函数的输入、函数调用的上下文环境，和程序视图的错误。在函数开始时，这些全局目标加到全局调试堆栈中，当函数执行结束后，这些信息从堆栈中退出。这就使你能特别精确地知道错误。这个函数可以被调用声明激发，但打印堆栈的内容就像没有异常处理的 Java 应用程序。

充分考虑执行一些像系统中的这种事。生成自己的核心代码和操作系统生产的核心代码是同样有用的。

- ◆ 调试类型：插入式/非插入式（根据你自己的执行情况）。
- ◆ 调试时间：长期调试。
- ◆ 应用层次：任意。

9.4.30 增加跟踪

在本书中，我们已经对跟踪技术做了深入的讨论，如果我们仅仅读了技术部分或者重新读这一部分，就很有必要重新提起这个概念。跟踪提供了对程序源代码的程序流的跟踪。对每个被调用的函数，就产生足迹（`trace`）来指派函数的调用和函数在调用点上的输入。一个

足迹的显示如下:

```
Start User Feature x
Function 1 (Input x=1, y =2, z=3)
Function 2 (Input z=4, a=5, b=6)
Function 3 (No input)
End User Feature x
```

通过对足迹显示的观察,可以知道,当试图从用户的角度调用用户特征<x>时,系统内部就调用了函数 1、2 和 3。你可以通过这些知道系统的内部工作情况。和可视化系统内部工作情况技术相结合,就可以知道更多的系统运行情况,尤其是在先前对工程不理解的情况下。足迹对那些从一开始就接触系统的人也是有帮助的,可以使你知道系统的实际工作运行情况而不是你所设计的运行情况,也可以使你捕获不好的程序流,知道为什么会出现结果不同的情况。

因为这个技术也在其他章节讨论,我们在本章不作更多的讨论。需要说明的一点是,这个方法在实时系统中很少应用。跟踪技术经常有问题,因为它要求一些地方来放置数据。

- ◆ 调试类型:插入式。
- ◆ 调试时间:长期调试。
- ◆ 应用层次:任意。

9.4.31 数据关系检查

在三种情况下,可以认为程序有数据依赖关系。第一,为了正常工作,代码要求数据有依赖关系。例如,在数据库中的配置文件和数据关系,被程序装载的目录信息。第二,某一类型的数据使系统崩溃也会存在数据依赖关系。这种类型数据不是程序所需要的或者超出了输入数据的合法边界。例如,程序可能希望输入的值为 1 至 5,如果用户输入 8 或者是输入的不是数字,例如 A,那么程序可能崩溃。在这个情况下,程序就和外部正确的数据有关系。第三种数据依赖关系是希望特定的数据比例和总数所出现的位置。在上一章的例子中,出现了卫星电视系统中数据暂停的现象,我们需要一个固定的数据输入速率。

怎样确定程序是否和数据有关系,并且是一种什么关系?首先,必须研究和系统有关的所有配置文件,可以认为配置文件不是系统执行文件的格式。认真列出这些文件,并从源代码中发现它们被应用的地方。会经常发现,编写系统的程序员依靠这些文件并且没有在读文件时提供足够的缺省设置。一个简单的测试方法是修改文件名,然后运行程序。如果程序在怀疑有 bug 的地方崩溃,就可以跟踪到配置文件的数据依赖关系。

第二种数据依赖关系的形式可以通过研究算法和允许的输入类型来发现。可以依靠算法的文档,或者是要发现的问题。看下面的一个等式:

$$\text{Ratio} = (\text{Total} - \text{Used}) / \text{Total}$$

这是一个清楚的例子。有两个参数:Total 和 Used,它们用来计算 Ratio。这个简单的公式,哪些数据会产生问题?显然,如果 Total 是 0,这个公式就会出错。有其他引起错误

的情况吗？有一个不明显的情况，当 Total 和 Used 相等时，就会出现结果为 0。虽然这是正确的，并且在大多数的程序中也是可以接受的，但我们不知道 Ratio 有什么用。有可能是用来计算它本身，例如：

```
Calc=Cost/Ratio
```

显然，如果 Ratio 为 0，就会出现被 0 除的相同问题。现在，看下面一个等式：

```
Calc=Cost/(Total-Ratio)*100
```

如果 Total=Ratio，问题是很显然的，会使系统崩溃。不过，Total 是在开始计算 Ratio 时的一个参数。既然如此，你可能要想，Total 不可能和 Ratio 相等。在数学上，这是有可能发生的。例如当 Total=1 并且 Used=0。在这个条件下，我们得到：

```
Ratio=(1-0)/1
```

计算的结果是 1。因为 Total 和 Ratio 相等，那么 Calc 的计算就出现了除 0 错误。任何包含潜在数据依赖关系的代码都要检查。例如，如果有除法，就需要检查被 0 除；如果有乘法并且要求正数结果，那么就需要检查负数结果。通过观察代码并检查类似这样的问题，将会节省很多时间。

- ◆ 调试类型：非插入式。
- ◆ 调试时间：长期调试。
- ◆ 应用层次：任意。

9.4.32 重放能力（记录动作）

增加记录动作的能力并重放它们，将对跟踪问题并解决它们有帮助。如果系统没用宏能力（保存和重放按键和动作的能力），就要特别考虑增加这个功能。

宏能力对调试系统有什么帮助呢？可以跟踪动作，可以为测试者运行它来记录它们的动作，检查宏中的活动记录来发现系统崩溃时的动作。这对于开发者很有用，不再需要调试者实时观察操作者的动作，测试者也不需要为了查找问题的原因，而记录他们的每一步动作。测试者应当为系统的宏能力感到惊讶，但实际上没有。把错过的东西指出来，他们将成为朋友。

一旦有了稳固的重放能力，宏记录也可以定义一个在测试时产生的按键输入集合。这样就可以不管程序的实际工作，只需要把注意力集中在试图发现和证实的假设上。当然，这是假设已经有了重放的能力。如果没有这个能力，就仍然需要发现引起问题的原因。另外，如果有了一个可编辑的宏，就可以对每个命令修改参数。例如，假设有一个打开文件查找字符串的宏。通过改变宏参数就能发现不同的字符串，并发现程序内部所发生的事。而这个过程的其他部分将保持一样。

因为这些原因，应该考虑在系统中增加宏能力。即使不允许外部用户使用它，这也会对调试有所帮助。使用宏会使系统变得很复杂吗？事实上，一点也不复杂。可以在很多发送按键信息给测试程序中比较类似的函数。通过简单的运行记录的能力，至少能发现用户在程序

中的每一步执行。对于客户/服务器系统或者基于 Web 的系统，这个简单的能力确实能发现问题产生的原因。如果不知道用户的动作，又怎么能发现用户系统的错误呢？

- ◆ 调试类型：插入式。
- ◆ 调试时间：长期调试。
- ◆ 应用层次：任意。

9.4.33 生成系统的副本

一个理想的客户/服务器软件开发系统包含四个复杂的系统。第一，开发系统允许调试者使用他们熟悉的调试工具进行调试。第二，有标准的 QA 工具，使测试者在受控而清楚的环境下测试新的开发功能。第三，允许用户访问成品系统。不管怎样，这个系统是有效的。第四，生成系统的副本，使调试者能对实际的数据和实际的软件进行调试。这并不包含调试工具或其他包含在开发系统上的修改。相反，它是成品软件系统的副本，能用来调试。可以在没有 QA 的允许下在这个系统使用新的软件。可以修改软件打印出变量或其他运行时的跟踪信息，而不用担心用户的经历。另外，你可以在任何时候停止系统，而不用担心影响到其他人，除了其他的调试者。

如果公司还没有生成副本的机制给开发者使用，就要考虑应当添加一个这样的机制。并不要求完全一样地复制成品系统。如果资金很紧张，也可以考虑降低系统的规模。例如，如果系统包含 20 台服务器和 5 个后端数据库，就可以去掉一部分，比如 4 台服务器和 3 个后端数据库，只要这个比例合理的话。你不能只用单个服务器或后端数据库，这不符合实际系统的比例。

在公司里添加这样的系统是有好处的，那就是在系统崩溃时，这个系统作为一个紧急情况下的备份。另外，也可以对开发时出现新的概念进行测试，可以完全改变系统，可以也可以测试提高系统性能的新理论。要记住，调试工具并不完全用于调试，有时它们对整个系统长期的开发也是有用的。

- ◆ 调试类型：非插入式。
- ◆ 调试时间：长期调试。
- ◆ 应用类型：客户/服务器或 Web 应用程序。

9.5 小结

这几十年来，调试者发展了无数的调试技术。通过阅读这些技术，理解和使用技术，就可以成为一个优秀的调试者。也就是说，在调试过程中，要记住每个调试的技术，不论它是使用在什么环境中。取决于调试的情形和环境，不同的技术将有不同的结果。要理解调试技术使用的环境和在这种环境下使用这种技术的原因，就能对不同的调试技术做较好的选择。

9.6 bug 问题

一个 Web 浏览器的登录页有三个要求。第一，验证试图登录个人的用户名和密码。第二，确保用户只在一个地方登录。第三，确保登录系统的总人数不超过系统的最大允许量。这个系统的伪代码如下：

```
Increment Number of Users by One
See if Too Many Users
If Too Many Users
    Exit Login
End If
Check User Name in Database
If User Name Not Found
    Tell User Bad User Name
    Exit Login
End If
Check Password in Database for User Name
If Password Doesn't Match
    Tell User Bad Password
    Exit Login
End If
See if User is already logging in
If User Already Logged in
    Tell User Only One Login Per Account
    Exit Login
End If
Add User To Login List
Exit Login
```

这个伪代码出现了一个致命的错误和两个容易引起 bug 的安全隐患。请将它们查找出来。

第 10 章 不同应用系统的调试

如果一个有经验的软件工程师知道调试，那么一个明确的规模就一点也不适用了。用来调试 10 行代码的控制程序的调试方法，和用来调试 10,000,000 行应用在宇宙飞船的系统的方法，是大不一样的。不仅在方法上有区别，在用来解决问题的技术上也是不同的。在宇宙飞船控制系统调试中，将在程序的驱动和系统的主要流程上花费大量的时间；但在命令行代码调试中，几乎一眼就可以决定整个系统的范围，包括系统的每个功能。

在成品服务程序中调试一个大规模的系统，和在开发工作站上调试一个小规模的单机系统相比，方法上也有很大的差别。在这一章，我们将分析应用在不同规模问题上的不同调试方法，使用这些方法将使调试起来更容易，包括怎样把问题缩减到可以用所选择的技术来解决的能力。你将知道哪些技术对 Web 服务器是不适用的，哪些技术对简单的应用系统是不适用的。通常情况下，调试的第一步是确定是否可以成功地解决问题，但在对不同的应用系统进行调试时，第一步的工作是要确定故障的不同点和一个主要的调试方法。

在这一章，我们首先从一个小规模的单机系统开始，它是一个完全封闭的系统；然后讨论大规模的客户/服务器系统，这个系统可能不仅包含三部分的库和构件，也包含了中间件、不同类型的环境和操作系统、实时系统。另外，我们还讨论了嵌入式系统的调试，基于 Web 系统的调试，以及运行在其他操作系统上的模拟系统的调试。

到此为止，我们已经把调试科学看成普通的学科。我们已经接触到了关于 bug 的一些著名的例子，我们知道了各种类型的 bug 和各种调试技术。不过，我们不仅要知道如何把这些技术应用在不同规模的系统中，还要使用一些从长期调试工作中得到的经验。

10.1 小规模单机系统

我们所指的小规模系统，并不是指那些代码量小或者运行时要求内存容量小的系统，而是指那些由小团体（1 至 6 名开发者）在较短的时间内（一年以内）就完成的工程。这样小的系统，一个开发者就能完全理解。此外，小规模单机系统对运行在封闭的单元内是有好处的，这并不意味着它和外部系统没有接口（例如网络或数据库系统），而是说系统本身就是一个独立的程序。

多数的开发者在他们职业生涯的头几年就进行这类系统的调试。单机系统通常应用在不同的环境下，像压缩软件、企业的内部应用系统等等。总之，单机系统就是只能在一台机器上运行，更进一步也只是能访问其他机器的信息，它们之间不是客户/服务器关系，也不是在这种环境下进行过程之间的相互通信。

10.1.1 成为系统的用户

为了真正了解并调试系统，就需要像“实际”用户一样来使用系统。这就意味着要像用户一样来使用系统，而不是像调试者或软件工程师一样来使用。亲自使用系统，是理解怎样从一个领域到另一个领域的最好办法，这样也可以知道系统运行的结果。

为什么要像实际用户一样使用系统？有以下两个原因。第一，可以更好地知道系统需要哪些条件才能正常工作，并且在系统通过测试和投入使用之前就能发现系统的问题所在。第二，通过使用系统，将会更好地知道哪些部分需要改善，知道哪些 bug 需要重点调试。用户喜欢那些了解自己开发的系统的人。

前些日子，我在一个公司工作，这个公司把网络当成了图书馆，为每个用户在网上提供了电子文档。这就需要有一个“check out”机制来保证在一个时刻只有一个用户在使用某个电子文档。在公司工作期间，我发现这些电子图书总是不能正确工作，表总是显得杂乱无章，经常掉页。我不是等用户来埋怨我们的系统，而是在这之前就直接找编辑人员，让他们改正。

编辑人员对我们遇到的问题感到十分惊讶，这是因为他们从来没有在站点上浏览过电子书刊，而是用内部的工具浏览他们编辑的结果。因为没有在站点上看过，就不知道内部工具不能很好地适用于最终的结果。事实上，他们简直就是玩忽职守。编辑并不是系统的用户，他们的工作只是组装每一行的文字，他们不会发现在 Web 系统上出现的问题，因为从来没有使用过。

10.1.2 复制环境

在单机环境下，另一个重要的问题是复制用户系统的能力。单机环境下出现的很多问题，是因为开发者没有在最终环境中检查系统。考虑以下简单的规则。

如果所开发的系统是在一般的客户机上使用，就不要在高性能的机器上进行开发和测试。有很多类似的埋怨：用户经常说系统在 Pentium 133 的机器上运行太慢。这是事实，因为系统是在 Pentium III 700 的环境下开发的。除非系统是在用户最可能使用的环境下进行测试，否则在其他环境下对系统的测试都是不合理的。当然，开发者需要高性能的环境，而不是一般性能的环境，但必须确保是在标准的配置下有规则地对系统进行测试。

对开发者调试的工作要特别小心。开发者所装载的构件经常会改变程序运行的环境。调试动态链接库（Dynamic Link Library, DLL）时出现编译器错误，进行监视存储器的第三方工具总是代表软件开发者的工具。但是，这些工具所带来的问题总是出现在用户环境中，而不会出现在开发者环境中。这就是我们经常在测试和调试实验室听到“可是它在我的机器上能够工作呀”的原因。

必须确保运行相同的软件版本。例如，如果希望系统运行在 Windows 98 的环境下，那么最好还是在 Windows 98 的环境下进行调试和测试，而不是在 Windows 2000 的环境下进行。的确，有时在更稳定的操作系统下能做更多的检测，却得不到和用户相同的结果。只有

在相同版本的操作系统（Linux、Mac、Windows 等等）下才会出现特定的问题。没有相同的问题环境，就不能确定问题是否已经被解决。也就是说，应当在程序运行的所有可能操作系统版本下进行测试，确保程序在这些系统中都不会出现问题。我们将在第 13 章对这个问题进行详细的讨论。

专业的调试工程师经常在附加的条件下运行系统，或者解决还未涉及到的一些 bug，以便使系统适合进一步的需要。在调试领域中，最难学的课程是在调试一个新的问题时，总是要从“干净”的源代码开始。如果不这样做，就经常没有办法发现问题，因为已经修改的代码对问题做了进一步的屏蔽。之所以困难是因为要在开发环境中从两个完全相同的源代码开始，这就意味着要调试两个代码，在调试一个问题时，已经解决的另一个 bug 有可能又出现了。如果不按这个方法进行，调试到最后你就会后悔。这本来是个矛盾，但这个矛盾可以避免很多的麻烦。版本控制和源代码转移一样重要，有必要成为调试的一个武器。

提示：在调试时，总是要从干净的源代码副本开始。

10.1.3 提防 DLL Hell

虽然 DLL Hell 这个术语的本意已经在计算机开发历史上慢慢消失了，但这个术语本身却和 Microsoft 的 Windows 操作系统紧密联系在一起。我第一次是在 Microsoft 公司 Rick Anderson 的论文里遇到这个术语。简单地说，DLL Hell 就是一组运用在系统中不同的外部构件版本。

虽然这个术语只在 Windows 操作系统中使用，但问题会出现在任何的操作系统中。当从各种系统库中使用共享的代码（例如内核库函数、数据库函数或在操作系统中编译通信例程）时，对所安装系统库版本的不同就无能为力了。

对于使用像 DLL 这样的外部构件，一个要特别注意的问题是：要注意所使用构件的版本号。一旦发现两个版本不匹配，就应该提出这个事实警告用户和你自己。这将会节省大量的时间和精力，因为这些时间和精力是花费在追踪由于另一个系统安装了老的或不兼容版本的系统库所引起的问题上。

提示：必须考虑到所有外部构件的版本！

10.1.4 输入/输出错误

几乎所有的小规模单机系统的错误都是输入和输出的不同。在处理时，程序允许非法的输入引起问题，程序也会去掉引起显示错误的输出。

对大部分的单机应用系统，最好限定每个系统能处理的输入类型范围，然后利用输入控制或过滤器构件来确保进入系统的输入在规定范围之内。如果在系统添加了这个过滤层，将会带来很多的好处，也就不需要去检查每一功能性层次的值。如果整理了输入的数据，那么处理过程就变得更快、更有效，因为已经知道进入系统的输入是正确的。当然，你必须在代码中做好注释，以防别人把这些代码使用在另一个环境中。输出错误是由以下几个原因引起的。

例如，当还没有完全处理好输出计算过程的某些条件时，此时因为一些条件从这一串值中选取一个值输出，那么就不能处理其他异常的条件。像这样的没有默认条件的选择语句所出现的 bug，变换这种类型的 bug 要求变换选择语句的条件。这可能是系统中的一个错误，或是系统中的一个漏洞。

提示：总要假设不能在系统中使用变换。

另一个引起输出错误的原因是格式化，格式化的错误使输出看起来好像不正确：输出错误的类型，或者输出的格式不是用户所期望看到的。格式化错误是由不好的需求或设计引起的，也可能是由编程或逻辑的错误引起的。不论是哪种情况，都会引起用户的不满，因为所得的结果不是他们所期望的。

为了调试一个格式化的错误，例如不好的显示，必须知道显示的结果。最好是对系统每个输入和输出的格式做好记录，把它和显示结果以及修改代码的日志一起保存起来。这样，不仅有利于调试一个陌生的系统，对试图改变和评估代码改变层次的设计者和规划人员也是有价值的。

提示：把所有显示结果的日志和产生这些结果的代码一起保留起来。

10.2 中规模单机系统

为了讨论的方便，首先应当明确小规模单机系统和规模单机系统的不同。它们的不同点在于中规模系统通常是从其他系统中接收数据和其他输入的，而不是从本身得到这些数据。也就是说，一个单机系统可能要处理它们自己的数据或者其他系统中的一些静态数据，但中规模的系统需要处理来自其他系统或设备的不断改变的数据。

一旦对输入系统的数据的格式和内容失去控制，如果不能适当地为它们设计和编码，那么就会出现一系列的 bug。大部分系统的 bug 是由外部数据引起的，因为在系统设计时总是假设任何处理都是正确的，输入数据的格式是正确的。但在编程时，输入的数据流不可能是完全正确的，由于系统和硬件设备之间的接口原因，硬件设备提供的数据经常会丢失或被破坏。如果没有考虑到处理被破坏的数据，那么就肯定会产生被破坏的数据。在程序中一定要考虑到这个问题。

提示：不要假设到达的数据没有错误。

因为大部分的错误和数据相关，那么在调试这样的系统时，就要考虑到和数据相关的问题。第一，必须对发现的问题做好记录。第二，必须在代码中保证所输入的数据都是所期望的数据，要不然的话，输入的数据可能完全被拒绝，错误将会被外部的文件设备记录。第三，必须保证事件发生的顺序是正确的。在很多系统中，数据被读进来，被保存成固定（或内部）的格式，然后被核对。但是，在很多时候，在核对新数据之前，转换成新数据的原始数据就被删除了。当新数据核对不正确时，这就有可能出现错误，并引起 bug。

举一个简单的例子，看一个 point-of-sale 设备（通常被称为 cash 寄存器）。价格和纳税

信息经常存放在寄存器中，保证能查找到每个公共项目交易的发生。如果一个项目的输入数据被破坏，那么价格和纳税信息就被混淆了。在我曾经开发过的一个系统中，由于在项目数据库中破坏了一个数据库实体，产生了一个可怕的 bug。这个被破坏的数据发送到寄存器中，就清除了 Cache 中的内容，用新的数据更新它。当这个被破坏的数据加入到 Cache 中，不仅弄乱了项目条款，也引起在 Cache 中的其他条款产生错误。当然，系统的开发者是不会检查到数据有效性的，这些数据被认为是正确的，并且被认为仅仅是只读的。

因为这个错误经常出现，我最后终于在进入系统的被破坏数据上找到了原因。我发现系统没有明确地检查价格。因为价格出现负数是不可能的，毕竟商店不会既给你商品又给你钱（我猜想，除了网上购物）。同样，纳税信息也是一个代表那个层次税率（当地、省、联邦、消费税等）的字段，也是应用在所给的项目中的。在这里也应该有一个简单的检查，如果有“当地税”的标志，那么不能有“消费税”的标志。如果数据被破坏，这两个标志就会同时出现，这就引起系统中其他逻辑的错误，就会出现异常的接收和异常的税收。当商店发现系统的错误而引起损失，当然会不答应。

我们解决问题的办法是：对每个进入系统的项目进行检查。如果项目是非法的，就拒绝它并发出另一个数据请求。如果两个连续的数据都是非法的，那么就把这个项目做上标志并观察寄存器的显示结果。如果结果显示项目是非法的，那么就要通知检查主数据库（在我们控制范围之外的系统）。同样，如果第一次显示是\$10.00 的交易，而第二次显示是\$99.99 的交易，那么这个项目也要做上标志并拒绝它。通过简单的检查，我们就排除了每年几千美元的损失，不用说，人家都喜欢我们的系统。

10.3 中规模客户/服务器系统

什么是中规模客户/服务器系统？过去，中规模客户/服务器系统通常是指那些公司局域网的服务器系统，这样的系统的用户通常是公司的雇员，一二百个雇员同时使用这样的系统。同时使用系统意味着高的系统负载，但负载通常是在预定的范围内。在设计这样的系统时，通常已经知道每一年有多少人使用这个系统，因为已经知道了公司当前的人数和还要招聘的人数。总之，不需要怀疑中规模客户/服务器系统负载能力。

现在的中规模客户/服务器系统和过去的有些不同。现在的系统是少客户系统，使用了固定的存储容量，这些存储系统通常是一些类型的数据库。很多系统具有相同形式的登录过程，用户都通过某种方法注册，并提供密码登录。另外，这些系统还为用户提供了一些个性化的服务，所以用户能更有效更容易完成任务。对于调试者，能立刻知道确切的 bug 类型。

当用户在提供密码时输入非法的字符或者不知什么原因改变了用户名，那么就不能进入系统。为了使用户登录，就产生了登录问题。当创建一个新用户时，就出现了一个普遍的问题，系统经常要求输入密码，然后通过要求再次输入来证实密码。但是，系统只是简单地从一个字段复制密码到另一个字段。很明显，如果在第一个字段中出现了一个非法的字符，那

么这个非法的字符也会出现在第二个字段。通过简单地扫描非法的字符（或者警告用户出现了非法的字符），就能解决问题了。

作为一个调试者，必须定时扫描数据库，以发现字段中奇怪的字符或者其他非法实体。这个过程就叫做 bug 预防（pre-bugging），在成品系统中进行调试之前就能发现问题并解决错误。另一个提示是检查大小写字母的结合。在很多条件下，用户不注意这样的结合，以为输入是正确的，其实不然。然而，在很多系统中经常使用这样的结合。所以，注意用户注册时输入的字母大小写组合和登录时输入的不同组合是很重要的。另外，这还会使用户无法登录系统。这些问题也会被当成是 bug。

提示：注意用户名和密码的大小写、连续的空格数和非法的字符。

怎样才能更容易地调试中规模客户/服务器系统？下面，将介绍两个较好的调试方法。

10.3.1 生成后端数据库的一个简单描述

调试一个中规模客户/服务器系统时，通常要测试底层工作信息。虽然可以通过在表中输入随机的值来生成一个简单测试数据库进行测试，但通常倾向于使用成品数据库的简单描述来测试。

使用简单描述有以下几个优点。第一，可以清楚地知道包含所有问题的真实数据。第二，如果问题确实是由数据库的具体用户产生的，这时，使用真实数据将会使调试过程变得特别容易。因为对待具体用户问题的调试方法是去调试系统本身，使用简单描述法当然是好的方法。

为了充分地证明假设或修改，就需要一个和成品系统的数据库完全相同的数据库副本，但并不需要整个系统都相同。例如，如果系统包含了一打的服务程序和数据库，就只需要对系统中的两三个对应的服务程序或数据库进行建模。模型越靠近实际，就越容易发现和调试系统的 bug。当然，所有这些要有一个前提，那就是系统已经被开发并被使用。

10.3.2 保留数据的使用情况来发现哪些数据经常被使用

基本的使用数据包括系统中每个用户的执行情况和所调用函数的参数。对一个账目系统来说，可能先有一个记录显示用户的第一次存款，接着是取款和一些税收。了解了用户企图，就能更容易找出错误的地方。例如，因为多个用户操作同一个帐户引发的问题，只简单地询问一个人将不能得到解决 bug 的充分信息。为了真正地解决问题，必须知道另一个用户同时改变的数据。

使用数据必须最少包括用户标志（一般是用户名）、动作的时间和数据、动作本身和其他与动作相关的信息。最好的情况是使用的数据能完全重现用户的所有动作。事实上，在我曾经工作过的一个公司，因为一个运行部门的操作错误引起数据库的突然崩溃，我们就能仅仅使用数据记录即可完全恢复数据库。

10.4 大规模系统

对于所有的系统，大规模系统的错误是最难想象和调试的。一个大规模系统是一个开放的模型，也就是说，在理论上对使用系统的人数是没有限制的，Internet Web sites 和 Internet Service Provider 系统就是大规模系统的例子。在这两个系统中，可以猜想到有多少人可能同时使用这个系统，但仅仅是猜想。另外，用户是可变的，使用系统的用户个数是由市场行为、产品输出或者网络上邮寄等条件决定的。

调试大规模系统的困难是显而易见的。对同时使用系统的用户个数一无所知；因为系统对每个人都是开放的，就无法推测哪些特征是系统经常使用的。大规模系统的问题通常很难解决，时间限制显得很重要。如果总是希望系统能 24 小时不停地运行，就没更多的时间来调试和维护系统。当像 e-commerce Web 站点这样的大系统崩溃了，即使是很有能力的人也不可能立刻解决。为了向公司的副总裁解释崩溃的原因，可能要花几天的时间去研究，但最后可能还不能成功。

下面介绍一些能简单、有效调试大规模系统的调试技术。

10.4.1 为测试安装一个“后门”

虽然为系统留一个备份是很好的办法，但有些错误不能通过简单的对这种备份的观察就能得到解决。好的办法是了解问题负载和多个用户之间的相互作用，在这些条件下，就需要观察在真实负载下的实际系统的运行情况。在这种情况下，在系统中留一个“后门”（back door）是好的办法。

“后门”为你提供了一个特殊的登录名和密码，这将允许你观察实际系统运行状态下的调试输出情况。当然，这要预先假定在系统中已经建立了一些故障诊断（第 9 章提到的）。如果没有这样，这正是去了解这些诊断并知道如何把它应用在系统中的时候。

最好提供一些对“后门”登录追踪的信息，一系列的记录准确地显示系统的运行情况和代码的执行路径。此外，如果问题产生了死锁条件，也能准确看到引起问题的语句和在问题点上的用户。

我曾经开发过一个 Web 服务器系统，出现了几个用户账目看不到他们用户的显示结果。我们不知道问题的原因，我们有规律地从备份的系统中观察数据，但没发现问题。为了进一步查找问题，我们建立了一个特别的登录，这个登录允许“假扮”成系统中的任何其他用户。这样有两个好处：第一，允许我们像实际用户一样观察系统；第二，系统只向我们显示用户所能看到的，而不能访问用户密码，从而保护用户的保密性；第三，这个特殊的登录仅仅允许用只读方式来访问用户的账户，我们将不会改变用户的任何设置和参数。这个特殊的登录也为我们提供了途径以查看代码内调试的信息，利用这个特别的调试，我们就可以在用户许可的基础上关闭或打开数据显示。

在 Web 站点, 我们通过查看用户的信息就能发现问题, 并且知道了没有显示的原因。这是一个简单的逻辑问题, 使得参与系统的用户发送错误的页面。如果不用特殊的逻辑来追踪这种问题, 将要花费较长的时间去解决问题。

10.4.2 观察外部条件的改变

大部分的大规模系统没有为用户提供所有的条件。例如, 很多基于条件的 Web 服务器被外部资源保护, 有时外部资源是一个计划外的分公司, 有时外部资源是另一个公司或收集数据的 Web 站点。

在调试领域中, 所听到的一个最有趣的注释是: 为了避免更多的问题, 我们标准化了相互改变的格式。在调试系统时, 将会发现一个标准化的相互改变的格式其实是非标准化的。考虑 XML (eXtended Markup Language), 一种新的数据转换语言, XML 是一个基本的数据格式标识, 包括标识名和标识值, 无论如何也不会想到这样的系统会出现错误, 尤其是在标准而有效的格式下。

不过, XML 依赖正确的标识名。如果取得一个 XML 文件并在系统中分析信息, 从而可以确定文件的格式是固有的, 就可以分析所有的标识名和标识值。不过, 不能确信的是标识名是否是你所需要的。这就没有一个标准, 而仅仅是发布者和接收者之间的协调。如果发布者突然改变他所发送内容的某些方面, 接收者的程序就可能崩溃或者得到错误的结果。

10.5 实时系统

所谓实时系统, 就是指那些在非常短的时间内对输入作出相应, 并给出相应输出的系统。股票交易公司就是一个实时系统, 它必须在很短的时间内根据股票价格作出买股或抛股的决定。另一个例子就是卫星电视系统, 它必须立刻处理输入的信号并提供给电视。

实时系统自身有很多问题。调试这些系统并不容易, 因为不能停止系统来观察它的状态。在很多条件下, 在调试实时系统时, 遇到问题之前就要分析所提供的数据来发现系统的动作, 然后根据所掌握的调试信息猜测引起问题的可能原因。

在调试实时系统时, 有很多有益的告诫和一些特别好的技术。以下就对这些技术和告诫进行讨论。

10.5.1 注意添加调试语句

总体上说, 在实时系统中很难添加太多的调试语句来显示调试信息。这是因为系统要在很短的时间内就要发生相应事件, 每次添加语句都会使系统改变语句执行的频率。

特别的, 几乎不可能通过各种的中断例程来得到各种有意义的输出。事实上, 格式化数据是很困难的, 因为需要很多机器周期来执行简单的格式化命令。因为这个原因, 必须特别

注意调试中断例程的记录。在这样的系统中，所输出的数据是一种抽象的格式，通常是标识数据和标识值。数据被缓存在存储器中，并在时间允许时输出，通常是一个独立的线程。这不仅使中断例程能够无阻地继续执行下去，还能输出调试中断例程所需的信息。

10.5.2 监视硬件软件的冲突

当系统的两个不同部分抢用系统的一个资源时，实时系统就出现了危险，不论这个资源是硬件设备还是软件子系统。这样的问题通常很难调试，这是因为它和时间、数据联系太密切了。

我曾经开发过一个系统，在系统中，实时数据从一个输入设备得到，但由三个不同的线程处理。很奇怪，系统会莫名其妙地崩溃。我没有任何可利用的数据，只能观察输入的数据流，但没有为排除系统错误发现任何有用的信息。总之，几乎没有办法进行调试。因为是个实时系统，跟踪和记录整个系统是不可能的。我们只好通过跟踪代码来发现引起问题的可能原因。

最后，我们使用了惟一的调试方法——分解法。也就是把系统分成小部分，直至问题消失，然后再慢慢增加系统直到问题出现。结果发现是系统独立的两个部分调度一个缓存区引起的。这不是一个真正意义上的问题，因为是一个独立的缓存区分配给每个输入流。不过，一个简单的例程用来追踪整个系统可利用的缓存区总数，而这个例程不是可重入式的，有时会在一个时刻调用两个不同的线程。这就引起了内部访问冲突，引起系统崩溃和我们观察到的 bug。

在调试实时系统时，要注意系统中两个或多个独立子系统使用软硬件资源的冲突。一旦发现这样的冲突，就应当想到设计时的同步问题和这种子系统的执行情况。

10.5.3 时间问题

时间问题是实时系统编程和调试的难点，发生在系统生存周期内在某些时间段上的问题是最难调试的 bug。例如，有一个在凌晨 3 点下载系统信息的应用系统，为了跟踪问题，就必须在凌晨 3 点观察系统的运行状态。当然，如果错过了第一个下载事件，就必须在第二个晚上重新来观察。这当然不是个令人满意的调试方法。

其他类型的时间问题也是一样的。例如，如果一个事件要在另一个事件发生的几毫秒（或更少）内发生，就像一个消息的应答事件，有时就在一小段时间内垄断了 CPU 的资源，就会出现时间错误。发现时间问题不仅是一个检测的工作，更是一个查找的经验。

在调试实时系统时，一个重要的要决定的因素是哪些是系统的时间要求性高的地方。根据你的系统，可能有不同优先级的线程被激活来完成不同的任务。最令人讨厌的是，因为部分开发者的忽视，低优先级的监视线程可能打断高优先级的数据查找线程。一个几乎要求立刻反应时间的任务应当给予最高的优先缓和尽量少的代码。

10.6 嵌入式系统

嵌入式系统是在另一个设备中运行的特殊芯片或平台。小到电子表，大到宇宙飞船都有嵌入式系统。应当认识到调试嵌入式系统会受到环境的约束。除非提供了嵌入式系统的接口，否则几乎不可能调试嵌入式系统。很多嵌入式系统是通过模拟器来调试的。

10.6.1 模拟器问题

在嵌入式系统中，很多工作，像单元测试、调试和执行代码都是通过模拟器进行的，所以出现问题也是正常的。在模拟器中，很多工作是正常的——模拟系统运行的环境——但在实际环境下却很难正常工作，因为模拟器只是模拟运行的环境，却不能发现在实际环境下工作的嵌入式系统产生的奇怪问题。

在模拟器中几乎没有电压毛刺、错码引起的奇怪消息和关键构件失败的现象。在软件开发领域中有个普遍的观点，我们只关心正确的并如何利用它，而不管系统中可能的错误。你能在模拟器中捕获到逻辑错误、缓冲区溢出和其他程序上的错误，但是不会捕获到没有发生的、也不想让它发生的问题。当然，如果要捕获到这些问题，首先必须为它们编写代码。

10.6.2 抑制中断

首先，我们讨论一点关于中断例程如何知道工作在实时系统中的中断时间问题，这样的问题也同样存在于嵌入式系统中，尽管嵌入式系统和实时系统有不同之处。当一个子系统关闭中断例程时，就发生了中断抑制，这时如果中断产生，就会有不安全的因素存在。一个明显的地方就是中断处理程序本身。

在中断处理程序中，一个存在的危险是在还没有处理完一个中断时，又产生了另一个中断，并调用了中断例程。由于这个原因，通常在中断处理时关闭中断系统，在处理完毕后重新启动。在很多时候，能正常工作，能继续运行中断过程。但是有两种情况，系统不能正常工作。

第一，在一个中断处理中关闭所有中断而没有充分考虑这对系统其他部分的影响。通常情况下在中规模、大规模嵌入式系统中有不止一个的中断处理。例如，可能有数据的输入、输出处理。如果关闭了整个中断系统，输出中断处理就不会被激活，就饿死了其他需要输出数据的系统。这就会引起系统的问题，但这样的 bug 可以最后通过代码来解决。

第二个可能引起问题的情况是不能成功启动中断系统。如果中断处理程序出现错误，中断例程不存在了，就有可能出现这种情况。如果不能启动中断程序，系统就要暂停。

10.6.3 协议错误

当使用的硬件发生改变，而和它对应的软件没有变化时就会产生协议错误。这种情况经

常发生在固件环境中，在这种环境下，协议是通过可下载的软件来实现的。当用来执行协议的固件下载后发生变化时，另一边的软件也需要变化。

协议错误经常出现，因为用软件完成的应用程序接口（API）不会和固件同时发生变化。在很多情况下，API 不会真正变化，只发生开发者不知道的单方面的改变。在这种情况下，软件总是认为是正确的，即使固件已经发现了问题。很多时候，一个应答信息从固件发送给软件，但却被忽略，因为软件不能理解所定义的消息。

10.6.4 看门狗时钟

在很多嵌入式系统中，看门狗时钟是一个独立的过程。它通过监视一些机制来发现软件是否在运行。如果软件出现一个灾难性的错误，系统被迫停止，看门狗时钟就重启整个系统。

看门狗时钟复位时，不能说明它本身存在 bug。相反，它是其他问题的标志。如果看门狗时钟没有记录系统复位的原因，那么就应当尽可能地在系统中添加这个功能。当然，这也就要求添加看门狗时钟知道系统崩溃原因的能力。如果系统在复位时没有表现出明显的原因，那么就很难找出嵌入式系统的问题。

10.6.5 调试嵌入式系统

有许多调试嵌入式系统的办法。如果问题出在硬件上，可以逐行分析来检查系统的输入输出。如果出现错误是系统的连接或电压设置，可以通过量电压法来发现问题。如果是软件原因，就只有两个调试系统的办法。

当软件控制的硬件不能为硬件系统提供正确的输出，就要用逻辑分析法。逻辑分析仪是一个放在两个硬件之间的设备，来解码两个设备之间的通信信息。如果知道两个硬件之间的通信协议，那么逻辑分析仪就能显示应用的协议不正确或者是时间不同步，这样就有很多的信息来发现和解决问题。

仿真器是调试嵌入式系统的另一个好途径。使用一个仿真器，就能发现软件本身的问题。当然，必须注意到使用仿真器和模拟设备引起的问题。

10.7 分布式系统

分布式系统是在不同的构件上运行的系统，也可能是在不同的机器上运行的系统。显然，除了调试系统上的一般性问题，分布式系统增加了问题的复杂程度。分布式系统具有不确定性，因为有很多进入和退出系统的途径。另外，分布式系统经常使用多种语言、主机和操作系统，使用中间设备来连接这些节点。这就使调试更加困难，因为只有掌握系统上的不同技术才能发现问题。

另外，大部分分布式系统是由 COTS（commercial off the shelf）构件组成的。这些构件几乎不公布源代码，这就增加了调试问题的难度。

10.7.1 中间设备错误

在几乎所有的分布式系统的调试中，中间系统的错误是主要原因。在你自己的系统或不正确调用中间系统的构件中的实际代码中产生了 bug，就会出现中间设备错误。在后者的调试中，通常使用分解法来发现引起问题的构件，然后重新查看构件的指导和说明书，直到发现错误。

在中间设备构件中的 bug 是很可怕的。因为几乎没有构件的源代码，只有两种调试的办法。第一，可以去掉这个构件，用完成相同功能的构件来代替它。这通常需要重写大量的代码并改变系统的体系结构。第二，可以围绕问题，去掉那些引起问题的构件的某些方法。在这个情况下，就要通过删除代码来测试系统的功能是否和原来一样。

避免中间设备构件 bug 最好的办法，就是在系统使用构件之前，对构件的每个方法进行全面的测试。

10.7.2 预测错误

预测就是在给定的条件下推测将会发生的情况。不过在分布式系统中，通常很难在短时间内知道将要发生的事。改变路由、失的机器和不同的构件加载时间都会改变整个系统事件的顺序。

当开发者事先推测部分系统的加载和运行情况，并且写下了依赖顺序的代码就会出现预测错误。如果依赖顺序改变或者不像事先预测那样，就会出现很难调试的错误。分布式环境中一个普遍的错误是在使用构件时，却没法加载构件。在更新构件时也可能出现错误。通过一个记录设备，验证所有运行在构件上的服务都被更新，进一步验证构件在更新前已经和系统的其他部分结合起来测试。如果一个方面没有被测试，就要在没有测试的部分查找问题。

10.7.3 连接错误

连接错误是分布式系统中最普遍的问题。在一个构件中使用另一个分布构件时，这类问题就有可能产生。不能检测和处理突然出现在分布式系统中连接错误的 bug。在这种情况下，错误会在系统中衍生，而且很难被发现。如果没有在出现点上发现连接错误，调试系统就很困难了。不过，在很多情况下，连接错误会在任何构件的任何功能调用时出现。大部分的程序没有检查这样的错误，因为简单的 get 和 set 方法不会产生这样的错误。

看以下的代码，这类代码可能出现在典型的 Web 服务器系统中的 VBScript 页面中。

```
Dim X as DBObj
X.SetName myName
X.SetAddress myAddress
On Error Go to HandleError
REM Do some lookups using the information stored.
```

在上面代码片段中，程序在设定一些查找的数值之后就会出现查找错误。不过，毫无疑

问, DbObj 的 SetName 和 SetAddress 方法很容易返回连接错误。如果 DbObj 是在另一台机器上运行的 COM (Common Object Model) 构件, 并且 Web 服务器失去了远程机器的连接, 那么 SerName 方法就会返回错误。这个错误不会被处理并且一直上传到最后的用户。不久以后, 就会引起操作人员的注意, 操作人员就会告诉调试人员, 这个问题的调试可能要几天的时间。

10.7.4 安全错误

当安全策略被破坏或者构件试图使用没有授权的资源, 就会出现安全错误。在很多情况下, 安全错误是部分的错误操作引起的, 这将拒绝访问需要资源未被授权的系统。

当数据库用户改变时, 通常会产生安全问题。例如, Web 页面上的用户名和密码可能被拒绝进行写访问, 因为数据库管理者认为这是不必要的。在我曾经开发的 Web 站点中, 一个上线的新数据库管理者, 他的第一个动作是清除用户访问数据库, 问题在于个别的用户账单被系统使用, 系统已装入的数据和清除的数据被 Web 服务器使用。引起的直接结果是使用的失败, 没有正确更新数据库。最后, 问题传到终端用户, 他将看到非法的数据。可以想象, 调试这样的问题要花一段时间。

安全问题还包括目录和文件的访问。例如, 在 Unix 操作系统中, 访问权限问题会引起某些系统不能运行, 除非他们运行在特权状态下。如果发现一个原来正常运行的程序突然停止运行, 就应该从系统的安全方面开始调试系统。

10.7.5 信息数据库

当在分布式网络的多机系统上运行构件时, 就会碰到信息数据库问题。这个问题的最普遍形式是 Windows 的注册表问题。在这种情况下, 不同的机器有不同的机器或软件配置, 这就引起不同系统上构件有不同的行为。

对于专业的调试人员, 外部的注册表问题是最难调试的问题。因为我们习惯发现系统代码的问题, 外部问题是一个全新的挑战。更坏的是, 因为设置是外部的, 甚至是系统的配置信息, 那么就不得不追踪这个系统来发现引起问题的原因。

在分布式环境中, Windows 环境下的 DCOM (Distributed Common Object Model) 是这类问题的一个简单例子。如果系统使用了 DCOM 构件, 就会发现系统在设置 DCOM 时会严重影响系统。这个依赖外部系统的影响将会引起有些机器系统的崩溃, 而对有些机器没有影响。

对付信息数据库的最好办法是用你自己的程序记录来显示这些数据库的相关信息。你也可以通过比较这些能工作和不能工作机器的记录, 来发现设置的不同点。和 DDL 问题类似, 设置问题是一个检测过程, 以此来决定机器之间的不同。如果能尽量把这个工作做好, 会发现这个过程很好。

10.7.6 记录的事后调查分析

在一个分布式系统中工作，当系统崩溃时，经常不可能正确地预测系统的行为，这是由分布式系统不可预测的性质决定的。在很多情况下，和时间、安全或者简单接口相关的问题崩溃一次就不会再崩溃了。

一个简单的例子，假设有两个构件试图锁定数据库中的相同记录。在某些情况下，第一个构件首先访问到记录并试图获得锁。这时，第二个构件就失败了。在另一些情况下，第二个构件锁成功了，而第一个构件失败。因为系统的时序问题，要发现哪个问题的错误是一件复杂的事情。

那么，怎么知道问题是锁定记录事件，而不是安全或时序问题？这个问题需要由系统崩溃后的事后调查分析来解决，解决这类问题的最好资源是系统日志。对于分布式系统，绝对需要大量的日志系统和记录收集机制。如果系统中没有这样的日志子系统，那么将要在机房里花费大量的时间来查找资料。

10.8 模拟系统

在很多情况下，特别是在硬件或操作系统还没有完成的情况下，开发者为了开发和测试系统就必须进行模拟。模拟系统有它自己的问题和独特的调试方法。例如，你总是当心所碰到的问题是否仅发生在模拟器中，是否会在实际的产品中出现；或者是相反的情况，出现在实际产品中的问题是否也会出现在模拟器中？

如果进行模拟系统的设计考虑以下几个方法，就有助于调试。下面来看看可能碰到的问题以及可以用来有效调试的相应办法。

10.8.1 封装硬件接口

在系统中处理各种硬件接口，把这些接口封装成一个代码段是很重要的。在模拟环境中，除非发生重大的偏移，模拟器一般能够很好地反映实际的系统。当硬件错误时，模拟器不会返回错误，因为对硬件的模拟很少出现错误。

如果建立一个硬件接口的封装，就可以用函数来替换模拟硬件 API，当全局标识或一些固定数据的重复操作出现时，就会随机返回错误值。例如，我们来考虑进入模拟器的串口输入。有不同的执行方法，最普遍的方法是从磁盘播放现存的文件到应用程序。在这种条件下，就很难发现字符丢失、连线错误和校验错误等。这些错误没有出现在模拟器中，怎样检验这些错误呢？这个简单的问题只能由你自己解决。

10.8.2 把模拟调用封装成预测错误返回

和硬件模拟器类似，模拟操作系统调用和返回错误的思想，对调试系统也是有价值的。

如果能在模拟器中正常工作，却不能在实际系统中工作，可以试着改变一些不能在实际系统中工作的软硬件接口函数。这是因为不好的模拟器应用，或者是从实际系统到软件有捕获不到的错误。

不久前，我开发一个 POS (point-of-sale) 系统。系统的硬件和最后接口的 API 代码在我们开发系统软件时还没有完成。结果，我们通过系统的模拟器进行工作，并在模拟器中编译代码。我们试图考虑在硬件和软件失败下的每个可能方案，并在随机的情况下把它放到模拟器中。在运行的情况下，硬件出现了随机的错误并向用户显示错误。同样，软件也会显示我们能在系统文档中发现的错误代码，并提示模拟器用户已经有错误出现了。

当系统最后完成，并安装在硬件中，系统从一开始就正常运行。模拟器已经考虑了可能的问题并用 bug 预防技术提前处理了问题。这样，因为几乎处理了所有这样的问题，我们就能用模拟器来生成一个问题进行调试，接着通过跟踪代码来查看系统的结果。

10.8.3 把实际系统模拟器的的问题简化成最简单的形式

在处理一个复杂系统时，一个特别困难的问题是如何测试底层部分的代码的修改和出现的问题，而这些代码只能被复杂系统中大部分的活动访问。用模拟器就能很容易地解决这个问题。

给出一个要调试的特定子系统的小的框架，就可以用一个模拟器来测试可能出问题的地方。接着就能够测试这个框架，把它放在真实的系统中来进一步检测要查找的、但模拟器没有发现的问题。因为这个原因，模拟器对于调试嵌入式系统是一个很有价值的调试方法。

10.9 小结

应用在不同环境中的调试技术，是由这些系统特征决定的。在这一章，我们举出了很多最普遍类型的系统，分析了这些系统中出现的不同问题和所应用的调试技术。

和调试领域中其他问题一样，调试某个特定的系统时没有绝对正确的方法。你的调试经验将决定了发现和修改问题的最好和最容易的方法。不过，通过检查环境和问题域，就能较好地决定调试特定问题的最好办法。

10.10 bug 问题

有些问题特别复杂，要求充分理解应用系统、问题域和程序开发语言。而另一些问题，像这个例子所描述的，只要求考虑和检查源代码。给你一个线索，这个问题出现在 C 语言程序中，但语言本身不会有问题。

下面两个小程序段是跟踪 Web 站点的信息。第一个程序接收一些输入参数和输出一些状态信息给文件。第二个程序读这个文件，并把信息显示给操作人员。不过，在这个程序运

行时，不会输出有用的状态信息，为什么？

程序 1 的源代码

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
int main(int argc, char **argv )
{
    if ( argc < 3 )
        return -1;
    int numVisitors = atoi(argv[1]);
    int numDownSeconds = atoi(argv[2]);
    // Open the file for append.
    FILE *fp = fopen("TODAYSDATA.DAT", "a");
    if ( fp != NULL )
        fprintf("%d %d", numVisitors, numDownSeconds );
    fclose(fp);
    return 0;
}
```

程序 2 的源代码

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
int main(int argc, char **argv )
{
    int numVisitors, numDownSeconds;

    // Open the file for input.
    FILE *fp = fopen("TODAYSDATA.DAT", "r");
    if ( fp != NULL )
        fscanf(fp, "%d %d", &numVisitors, &numDownSeconds );
    fclose(fp);
    printf("Number of Visitors: %d, Number of Downtime Seconds: %d",
        numVisitors, numDownSeconds );
    return 0;
}
```

第 11 章 调试之后

在调试完成一个 bug 之后，该做些什么？我们将安装这个修改后的系统并对修改进行测试，接着在配置管理系统中运行这个修改，然后通知测试者让他们也可以测试你的修改。过了一段时间他们将返回信息，告诉你修改看起来很好，并且把这个修改传给成品环境，然后你就可以轻松了。当你看着香烟的烟雾围绕着自己，你靠着椅子，尽情地享受着调试成功的软件。过了一会儿，在屏幕上出现了“你有邮件”。立刻，从 bug 追踪系统发来邮件，通知你下一个的调试任务。你熄灭了手上的雪茄，斜视着，努力地读着模糊的屏幕。读完电子邮件后，一阵傻笑，你想着：“这个 bug 听起来有意思。”然后你把剩余的雪茄扔到一边，并把烟头从嘴里吐出。仍然注视着屏幕，在脑子里想着调试的策略和制定计划。不用想，你将燃烧的烟头摁在鼠标垫板上，压下它，要开始一个新的挑战。

我们大家开始一个新的调试过程也和上面描述的一样。我们解决一个 bug，测试它，接着下一个 bug。在两个 bug 之间几乎没有时间休息。

提交修改并通过回归测试，不是调试过程的结束，一旦已经向配置管理系统提交修改，我们从来不会多想到进行下一个 bug 和下一个工程。毕竟已经解决了 bug，还有什么好担心的？不过，一个好的调试人员能预见到 bug，而不是等着对 bug 做出反应。在发现和解决 bug 时，为什么等着 bug 的出现？为什么要在一个不同的文件再次解决相同的 bug？为什么在解决一个 bug 的几个星期之后又引入动态跟踪打印语句？一个调试过程的结束是开始下一个调试过程的最好时间，也是从错误中反省和学习的好时机。

在准备下一个调试时，应当向自己提出以下五个问题。

- ◆ 我是不是在其他地方犯同样的错误？如果你是因为不能理解一个特殊语言结构而犯了错误，那么就有可能在其他相同代码的地方犯相同的错误。这个常识可以作为预防 bug 的出现。

- ◆ bug 背后隐含了什么？经常，一个 bug 会掩盖了系统的另一个 bug。你剥掉一层，只能发现这层下面的 bug。必须认真观察你的修改对系统的影响。

- ◆ 怎样更容易发现类似的 bug？在下一次调试中，要考虑怎样才能更容易发现 bug。

- ◆ 怎样预防这类 bug？考虑预防措施能帮助你在未来的调试中避免出现相同的错误。当你最先没有将故障引入程序中这就更容易调试程序。

- ◆ 我是否有所进步？像软件开发者一样来看看你的进步。

11.1 是不是在其他地方犯同样的错误

当错误的发生是因为对问题的某些方面或工具使用缺乏理解时，就可能在其他地方犯同样的错误。如果能解决几个类似的 bug（因为每个 bug 都是我们发现和解决的），我们就能像调试器一样熟练了。这个技巧是针对潜在的问题和不断出现的 bug。

例如，bug 的产生是因为开发者没有把空字符考虑在字符串的长度之内（C 或 C++）。这个错误的潜在原因是什么？可能是开发者输入的错误，也可能是开发者不知道 C 语言字符串的定义，也不知道字符串的停止，还可能是开发者做了相同类型指针的操作，犯了同样一个错误。

如果怀疑开发者不熟悉特定的语言，那么就要特别检查同一个开发者写的其他代码，查找相同类型的错误。如果是开发者不熟悉 C 语言的字符串，就可以预测到使用字符串函数引起的其他问题，例如 `strcpy()` 函数。事实上，最有可能是同一个开发者使用了存储器和指针操作。由于理解语言结构引起的错误，那么提出被开发者错误理解的概念就很有用。把错误理解的概念扩展到正确的概念，并观察错误是否也关联到其他概念。例如，如果开发者不知道 C 语言中空结束符字符串的概念，那么在使用 `strcpy()` 函数时就可能出错。

另一方面，如果假设引起错误的原因是输入或是其他一个简单的错误，那么就不必要在其他地方检查类似的错误。但一个优秀的调试者会坚持查找并在任意地方进行仔细检查。

这些技巧是建立在假设你知道产生错误开发者习惯的基础上。不过，作为一个调试者，你可能不清楚开发者知道什么和不知道什么。那么，怎样才能知道错误是不是由概念的不正确而引起的呢？你在开始查找 bug 时，可以试图发现这个 bug 和特定的错误概念有无关系。为此，必须对编程语言和概念都有所了解，也需要过去的调试经验和这个领域的知识。也就是说，你需要比产生错误的开发者具有更多的经验。在发现一个潜在的错误概念之后，就可以发现代码其他错误概念所表现出的现象。

用 C 语言的字符 bug 作为例子，让我们看下面的代码：

```
1. char *string;
2. string = (char *) malloc( sizeof(char)*11 );
3. if ( string != 0 )
4.   strcpy( string, "Hello World" );
```

作为一个调试者，你可以准确发现故障并知道正确的修改：把第 2 行的内容由 11 改为 12。开发者忘了考虑作为结束字符的空字符。这是一个简单的修改。但更重要的问题是，这个错误是否会出现代码的其他部分？从经验来看，我们知道在 C 语言字符串中空结束字符不是个直觉的错误。在编写 C 语言程序时，我们也有同样的错误概念并犯同样的错误。一个优秀的调试者，会在代码的其他部分进行查找，来发现系统的错误。如果所有其他的 C 语言字符串都正确，就可以假设这个错误是由错误的输入而引起的。但是，在代码的其他部

分继续出现相同的错误时，就可以假设开发者有了错误的概念，那么就有必要检查相关的错误概念。一个与对 C 语言空字符错误理解相关的例子，是不正确地使用 `strlen()` 函数。犯了前面错误的开发者，很容易犯下面的错误：

```
1. char *string1;  
2. char *string2 = "Test String";  
3. string1 = (char *) malloc( sizeof(char)*strlen(string2) );  
4. if ( string1 != 0 )  
5.     strcpy( string1, string2 );
```

在这个错误中，开发者没有注意到 `strlen()` 函数没有在字符串结束时考虑空字符，所以 `string1` 变量的值为 1。

显然，查找相同错误法对查找标准系统函数或 API 也是可以使用的，也可以使用在库函数和代码中。如果对特定 API 函数的错误理解的普遍现象是明显的，那么说明 API 函数很难被使用。对于 API 函数，可以通过再次访问 API 函数来确定是否有可能减少使用它们的错误，使它们能被更安全使用。

11.2 bug 背后隐含了什么

在解决了一个 bug 之后，你可能已经揭开了先前隐藏的 bug。被解决的 bug 掩盖了其他一些问题。这个 bug 使程序只运行一个代码的路径，解决这个 bug 使程序运行了代码的其他部分，这些部分以前从来没有测试或执行过。这个 bug 可能把一些变量设置成某些值，解决这个 bug 可能改变了变量的值并引起代码其他部分的相互作用。优秀的调试者不是仅仅解决手头上的 bug，他们也试图通过了解所调试的 bug 对软件的影响，从而预防引入新的 bug。优秀的调试者会向自己提出以下问题：

- ◆ 修改影响了哪些变量？如果改变了变量的分配或者控制的程序执行的路径，这就将会影响到代码中的其他变量。在软件中，变量的值决定了系统的状态。当变量被修改了，也就改变了系统的状态，这将影响到软件的正确性。

- ◆ 如何影响变量的使用？如果变量的改变是因为你的调试，就有可能打破一些先前在变量合法范围中未被测试的假设。例如，如果被改变的变量是除法操作的分母，并且代码没有检查它是否为 0，你估计就要进行修改，确保这个假设仍然成立。

- ◆ 因为变量改变了控制路径，会出现哪些新的值？经常，变量值是用来决定分支条件或者是软件执行循环体的时间。通过修改变量值，可能会改变程序的行为。有时，这个改变是合理的（解决 bug），但要注意预料不到的修改。

这些问题假设你的修改不会影响到和包含修改的软件模块相关的前提条件和后置条件。如果不是这样的，就要估计这些被修改的模块怎样在系统中被使用。

11.3 怎样预防同类 bug

在调试结束的时候，是考虑怎样预防 bug 重新出现的最好时候。在这个 bug 仍然在你的脑海中时，花费几个小时观察引起 bug 的原因，花费几个小时作出和测试修改。在这个时间来发现是否能预防类似的 bug 出现。

11.3.1 理解原因

如果你是引起 bug 的程序开发者，就要回想当时在进行特定代码开发时的事。做了些什么？想了些什么？如果是忘记检查错误条件，怎样才能预防自己再次忘记？是否对在设计时的较好定义有帮助？如果错误条件的处理程序是特殊的需求，是否对检查特殊文档有帮助？别人检查代码是否对发现错误有所帮助？进行编译器设置对警告的问题是否有帮助？改变软件的设计，对不再出现 bug 是否有帮助？

如果你不是引起 bug 程序的开发者，自然就不必进行反省。不过，你仍然要查看 bug 并考虑产生它的原因。是不是因为开发者没有理解语言的结构？是不是开发者不理解问题域？是不是设计太复杂？即使你不能得到有理由的探索，仍然可以确定途径，预防其他人犯相同的错误。设计讨论和代码讨论能帮助预防一些缓和的形式。较好的训练能帮助预防其他问题。可能是工具的改变，可能是客户和开发者之间的交流出了问题。有很多的原因，但真正的原因是组织里的人和过程。

11.3.2 保留测试用例

预防 bug 进入成品软件的一个办法，是在测试时就发现它。为了完成这个任务，除了查看特定的 bug 之外，没有其他更好的办法，这可以通过保留 bug 的测试用例来实现。测试用例可以成为回归测试组的一部分，有利于将来的测试。

11.3.3 利用 bug 为将来的设计

有些 bug 能在设计期间被预防。在第 2 章，我们讨论了存储器溢出 bug。如果在设计系统时避免使用静态缓存器，就能避免缓存器溢出；如果避免使用不安全的语言如 C 或 C++，也能避免缓存器溢出。但在实际工作中，这两个方法不是切实可行的。不过，如果我们提供一个检查缓存器长度的机制，并且分配给它的优先级比其他可能引起缓存器溢出的执行程序优先级高，那么就仍然能避免缓存器溢出的问题。怎样做呢？可以提供对一个对不安全函数调用的安全包装。例如，C 语言最容易混乱的一个特点是字符串的空结束符。因为开发者忘记了这一点，从而产生了很多 bug。下面的 CopyString() 函数是包装 strcpy() 的例子。

```
1. BOOLEAN CopyString( char *buffer, /* The buffer to set the content to */  
2.                     int size, /* The size of the buffer */  
3.                     char *from ) {
```



```
4.  if ( (strlen( from ) + 1) > size ) /* Don't forget the null */
5.      return FALSE;
6.  if ( buffer == NULL )
7.      return FALSE;
8.  strcpy( buffer, from );
9.  return TRUE;
10.)
```

显然，在 `CopyString()` 函数中仍然存在一些危险。`CopyString()` 函数仅仅是一个让代码更具有安全性的例子。如果程序员很懒惰，他们可能不会检查 `CopyString()` 函数调用的返回值，仍然像正确分配存储器一样使用缓存器，即使 `CopyString()` 函数返回 `FALSE`。而且，可能存在开发者得到的大小参数不是缓存器的大小。但这个函数提供了一个错误警告，犯这个错误主要是因为开发者暂时的思维偏差。

考虑另一个小例子：在电子商务中，很多地方要用到钱。毕竟，钱代表了在网上进行交易货物的价格和价值。你发现了多少能使用浮点数值或者双精度浮点数值简单代表钱的系统？好，不是很正确的，除了钱，作为一个概念，比起系统中执行语言定义的浮点数类型，有更多的约束条件和规则，为什么不在系统中只定义钱的数据类型，并用钱的数据类型代替普通的浮点值？我们可以使用更加强有力的类型来预防更多的 bug。这个概念将在第 12 章作进一步的讨论。

11.4 怎样更容易发现类似的 bug

如果不能预防所有的 bug，那么至少应当让 bug 更容易被发现。可以在测试和开发过程中使用工具来帮助我们，可以把 bug 编成文档提供给其他的开发者和调试者，可以重用以前生成的用来发现 bug 的工具和代码。

11.4.1 创造工具

创造工具可以帮助可视化，测试代码能使调试更有效。在很多开发系统中，编辑人员为类提供了可视化的方法，为方法提供了标记图。在这种环境下进行编辑能有效地预防愚蠢的错误和 bug。像屏蔽文件、配置管理系统和调试器等标准的配套工具使我们调试起来更容易。

但是工具不仅仅是开发系统，开发的特殊应用工具将对系统的调试有帮助。例如，在进行一个类库的调试时，要处理三维空间的几何结构图，那么，写一个简单的显示和操作这个三维几何图的程序，将有助于调试。创造一个调试和诊断工具的另一个优点是，这个工具能用来进行操作人员和用户的交流，成为技术支持的第一级。例如，我为公司的部分产品和服务开发一个 XML (Extended Markup Language) 接口。在使用 XML 接口之前，公司提出了解决方法，为客户设计了 Web 站点。这就使我们的顾客服务部门更容易通过 Web 站点观察系统是否正常运行，并验证顾客的埋怨。但是如果使用 XML 接口，就没有图形的解决方案。当出现问题，顾客服务部门就不容易发现埋怨。为了模仿顾客的用途，要用 XML 接口生成

一个 Web 页面，为顾客服务部门提供使用的工具，使我们测试 XML 接口更为容易。

设计、创建、开发和测试等工具有时可能是、有时可能不是正常的调试和开发的使用工具，但它们都很值得研究。

11.4.2 记录 bug

文档是软件开发的祸害，我们都很害怕它，但文档是很有价值的。如果要从错误中学习，我们就需要将引起问题的原因和解决办法做成文档。文档也使其他人从我们的错误得到启发，并理解我们的修改的基本原理。关于 bug 的文档信息能给调试者一些关于系统相互关系的启示，并使调试者更容易发现问题。文档能做成嵌入式的，就像代码的注释；也可以做成外部文档或者是综合调试信息的一部分。

更新现存文档

设计和 API 文档通常是脱离数据的，因为在创建时没有考虑这些东西如何保持。当我们修改一个 bug 时，如果这个修改是和执行相关的，一些外部文档就需要修改。不过，如果这个修改和设计相关，或者修改了模块的前提条件或后期条件，或者引入了副作用，这些设计相关的修改就需要被存档，并传播到现存的文档中。这就是为什么嵌入式文档是更受欢迎的文档策略，它更容易和代码一起进行修改，而不是要逐个去修改每个文档。Java doc 是一个较好的工具，使调试者和开发者更容易为他们的代码制作文档，并使文档比较好看而且不需要额外的工作。

外部文档制作通常比较困难并且需要时间去修改，所以，在调试阶段通常很少有修改的必要，这就使它越来越成为脱离数据的文档。我们通常使用外部文档来记录领域层次的知识。领域层次的知识，一旦被理解就不会经常被改变。这就意味着外部文档仅包含一些相对稳定的领域层次的知识，不会经常被修改。

记录 bug 出现的原因

在前面几节，我们讨论了解释 bug 产生的原因。这些理解能帮助我们发现和预防类似的 bug。如果我们记录这些原因，就能帮助其他人学习经验。

记录怎样发现 bug

调试经常被认为是一个神奇的过程。“你是怎样发现 bug 的？”“哦，我通过查看商店的 cart 类，想起了价目表 catalog 类存储的产品的价格是浮点数，所以我想起了我要检查商店是怎样计算总的价格的。我就发现了商店类把它当成了双精度。”这是文档的作用吗？当然是，因为如果没有观察设计和代码，就不会注意到软件的很多方面。例如，①商店的 cart 类和价目表 catalog 类有关系；②价目表 catalog 类产品的价格存储成一个形式，而相同产品的价格在 cart 中却存储成另一个形式。通过对调试过程的听和看，就可以知道软件的信息，而不用要求真正查看软件。这些信息为我们提供了一些线索，而这个线索是很难通过检查和读源代码得到的。我们可能知道商店的 cart 类中存储价格是双精度型，也知道 catalog 类中存储的价格是浮点型，但很容易忘掉这个细节。我们可能错过它们之间的关系，这个调试的

过程提出了正确的关系。这个故事提供了一个可接受的途径，阐述了类之间的依赖关系。另外，这个故事也为我们提供了发现其他关于 bug 的工具和技术的新思想。

很少有人真正记录了他们怎样发现 bug。但我们知道每个人都喜欢谈论它。所以，这对调试者遇见特定困难的 bug，并使他们能从中学到经验是很有教育意义。

11.4.3 预留调试框架

当我们还是编程的初学者时，我们通过在代码中插入打印语句来帮助调试。在下面一段 C 语言代码例子中，要检查计算商品的销售价格函数是否正确，因此在第 8 行和第 13 行引入了打印语句。

```
1. float calculateTotal( int numberOfItems, float *itemPrices,
2.                      StateCode state )
3. {
4.     int count;
5.     float total = 0.0;
6.     float tax = 0.0;
7.     for ( count = 0; count < numberOfItems; count++ ) {
8.         printf( "Count:%d, price:%f, total:%f\n",
9.               count, itemPrices[count], total );
10.        total += itemPrices[count];
11.    }
12.    tax += StateTax[state]*total; /* StateTax is some static global array.
*/
13.    printf( "Tax for %s = %f%% - %f", /* StateCodeString also. */
14.          StateCodeString[state], StateTax[state], tax );
15.    total += tax;
16.    return total;
17. }
```

在发现 bug 后，我们通常删除打印语句，因为打印语句使代码难读。通常，删除打印语句不是问题，直到我们发现另一个 bug，并需要相同的函数。这时就需要重新增加打印语句，可能只是不同的格式和不同的内容。这次，我们吸取了教训。在 bug 解决以后，我们就不再删除打印语句，因为可能还要用到它。相反地，我们仅仅把它注释出来。不过，如果使用了像 C 或 C++ 这样的语言，可能还可以使用预处理的能力注释出打印语句，就像下面的代码：

```
1. float calculateTotal( int numberOfItems, float *itemPrices,
2.                      StateCode state )
3. {
4.     int count;
5.     float total = 0.0;
6.     float tax = 0.0;
7.     for ( count = 0; count < numberOfItems; count++ ) {
8. #ifdef DEBUG
9.         printf( "Count:%d, price:%f, total:%f\n",
10.              count, itemPrices[count], total );
11. #endif
```

```

12.     total += itemPrices[count];
13. }
14.     tax += StateTax[state]*total; /* StateTax is some static global array.*/
15. #ifdef DEBUG
16.     printf( "Tax for %s = %f%% = %f", /* StateCodeString also.*/
17.           StateCodeString[state], StateTax[state], tax );
18. #endif
19.     total += tax;
20.     return total;
21. }

```

使用`#ifdef`和`#endif`机制的优点是注释出代码，这样就能改变代码的行为而不要修改代码。为了得到打印bug信息的函数，只需在编译时修改编译的条件。这个技术在预定义软件中的很多地方用到。注释出调试语句可以使代码更容易读，并且运行更快一些。而且，这些调试语句对客户是不可见的。

但是有时使用预定义指示或注释掉调试代码不是一个理想的解决方法。这个困难在于只有在代码重新编译之后调试信息才可见。在产品和调试之间切换代码的版本是很麻烦的，并且要配置管理错误。解决的办法是使用命令行操作或配置文件，来打开或关闭调试。所以可以将代码改变如下：

```

1. float calculateTotal( int numberOfItems, float *itemPrices,
2.                      StateCode state )
3. {
4.     int count;
5.     float total = 0.0;
6.     float tax = 0.0;
7.     for ( count = 0; count < numberOfItems; count++ ) {
8.         if ( debug ) { /* debug is some global variable */
9.             printf( "Count:%d, price:%f, total:%f\n",
10.                  count, itemPrices[count], total );
11.         }
12.         total += itemPrices[count];
13.     }
14.     tax += StateTax[state]*total; /* StateTax is some static global array.*/
15.     if ( debug ) { /* debug is some global variable */
16.         printf( "Tax for %s = %f%% = %f", /* StateCodeString is */
17.               /* also some static global array.*/
18.               StateCodeString[state], StateTax[state], tax );
19.     }
20.     total += tax;
21.     return total;
22. }

```

通过这个途径，就不再要编译两个版本的代码。打开调试器，只需要重启程序即可。但在实时环境下，根本不可能随意地关闭软件。所以我们要通过它来查找一个解决办法。为什么要完全地停止软件的运行？如果能够使用所有这些特别的功能，去建立一个调试支持的结

构，就有能力重用它并加入一些有用的特征，一个调试特征的例子是能够控制所有调试输出的细节。这可以通过使用数字值来完成，这样就只有在特殊值下设计的调试语句才会输出。另一个特征就是能只有在特定的代码块中才打印调试信息。

下面一段代码说明了 `calculateTotal()` 函数，使用这样的调试基本构造和 C 语言的调试构造函数声明：

```

1. extern void debugEnterFunction( char *functionName );
2. extern void debugExitFunction( char *functionName );
3. extern void debugSetLevel( int maximumDebuggingLevel );
4. extern void debugLogging( int level, char *format, ... );
5.
6. float calculateTotal( int numberOfItems, float *itemPrices,
7.                      StateCode state )
8. {
9.     int count;
10.    float total = 0.0;
11.    float tax = 0.0;
12.    debugEnterFunction( "calculateTotal" );
13.    for ( count = 0; count < numberOfItems; count++ ) {
14.        debugLogging( 3, "Count:%d, price:%f, total:%f\n",
15.                     count, itemPrices[count], total );
16.        total + itemPrices[count];
17.    }
18.    tax += StateTax[state]*total; /* StateTax is some static global array.
*/
19.    debugLogging( 3, "Tax for %s = %f%% - %f", /* StateCodeString also. */
20.                StateCodeString[state], StateTax[state], tax );
21.    total + tax;
22.    debugExitFunction( "calculateTotal" );
23.    return total;
24. }
```

预留调试框架的思想使我们想起了跟踪调试技术（已经在第 9 章讨论过）。有很多执行调试框架的方法，在这一节中我们已经列举了几种方法。用一个标准和一致性来建立调试支持，是发现 bug 的一个重要技术。我们将在第 12 章讨论这些思想。

11.5 我是否有所进步

怎样知道自己是否有所发展？身体上，可以通过体重和身高来测量；在学校里，可以通过考试来评价；在体育比赛中，可以通过 100 米的成绩或者所跳的高度来测量。不过，在软件工程中，就不好定义个人发展的测量标准。这个问题应当追溯到面试的过程。在面试中，我们试图从不出色的人中找出出色的人，我们希望雇佣高生产率的工程师。有时，我们在一个产品上花费好几年的时间；而另一些时候，花费的时间相对较少。经常，我们在特定语言的经历是相同的，优秀的应试者能回答深奥的语言问题。

不过，测量的标准不仅仅和生产力有关。从生意的角度来看，一个效率高的软件工程师每天的代码量是最高的，而引入 bug 和错误是最少的。这就是所有可测量的数量。我们可以简单通过测量花费在开发特定功能代码的时间，和在子序列中可接受的测试下 bug 的数量。显然，其他对产品的测量，例如可升级和可扩展设计，也是软件工程师的重要素质。对于调试者，这样的评价可能会出现误导。因为一个调试者的工作不是写代码，那不是他们工作的内容。对于调试者，像前面这样的测量是很重要的。全面值（over-under value）是解决的 bug 数目时所引入的 bug 数目。一个优秀的调试者的全面值应当很低。

一个人在软件开发过程中可以通过分析和理解他们的开发过程来提高效率，降低错误率。通过记下他们的过程并对他们的错误进行回溯，开发者能改正引起错误的过程。而且，开发者能通过对错误率历史值的观察来评价自己是否提高。

11.5.1 bug 量度

测量就是数量上的度量。物理上，可以用长度单位测量我们的身高；在学校里，可以用等级或数字值；在运动场上，时间单位、长度单位、重量单位和点数是经常使用的量度。在软件工程方面，可以通过错误率来测量。典型的，错误率是在一定数目的代码中发现的错误或 bug 的数目，这是比较容易测量的。在不同的时间看我们的错误率，就可以知道自己是否有所提高。

仅仅通过看错误率并不能帮助我们提高。最多只能告诉我们是否提高了。我们还需要其他信息来指出我们的习惯和错误。既然这样，bug 量度需要更多的信息。以下一些项目可以用作 bug 的量度：

- ◆ 故障现象的描述。
- ◆ bug 的根本原因。
- ◆ 补救措施；解决 bug 的步骤。
- ◆ bug 的定位；出现错误的文档、函数、模块、功能部件、屏幕、类或者方法名字。
- ◆ 发现或解决 bug 的经验。

11.5.2 bug 跟踪

从管理者的角度，bug 跟踪系统能帮助控制 bug，理解系统的稳定性，并保持跟踪严重的问题。管理者也需要解决 bug 的附加信息。以下列出一些可能需要的信息。不过，因为环境的不同，对信息的需求量也不一样。

◆ bug 标识符——可以用来对 bug 进行分类的一些标识符。这些标识符可以用来跟踪因为 bug 而修改的代码或文件。

◆ 报告数据——关于 bug 的数据和输入系统的一些数据。

◆ bug 状态——标识 bug 在系统中运行的状态。一个 bug 的报告需要一个新的状态。一个分配给调试者的 bug 可能有一个“指派”的状态。同样，“测试”状态可以用来标识出

bug 的测试。“解决”状态用来标识被解决和证实的 bug。

- ◆ bug 分配——标识被分配解决 bug 的开发者。
- ◆ 开始数据——标识调试者开始调试过程的数据。
- ◆ 严重性——标识第 5 章中描述的 bug 的严重性。
- ◆ 工程/应用程序名——标识发现 bug 的工程名。
- ◆ 报告者——标识报告 bug 的人。
- ◆ 系统名——标识出现 bug 的应用程序运行的软、硬件系统。
- ◆ 构件名——标识出现 bug 的应用程序的软硬件构件。
- ◆ 文件修改——标识为了解决 bug 而进行修改的文件。
- ◆ 重现——标志在可重现 bug 情况下，需要重现 bug 的步骤。这些步骤将成为测试用例，可以作为回归测试程序的一部分。
- ◆ 环境——标识硬件平台、操作系统和其他相关环境参数。
- ◆ 描述解决——标识用来解决问题的方法，应当用自然语言来书写。
- ◆ 解决数据——标识用来解决 bug 的数据。
- ◆ 测试者——标识验证解决的测试人员。
- ◆ 测试数据——标识用来验证解决的测试数据。
- ◆ 测试结果——标识验证的结果。
- ◆ 回归测试需求——标识回归测试所需要的一系列需求。
- ◆ 负责回归测试的测试人员——标识负责回归测试的测试者。
- ◆ 回归测试结果——标识回归测试的结果。

11.5.3 怎样处理数据

bug 度量的目标不是为了追究责任，也不是为了评价一个开发者的能力。相反，bug 度量的目标是让开发者从他们的错误中吸取教训，并提高自己从而成为一个工程师。但度量也可以通过监视 bug 的出现率和解决率，来帮助管理者确定工程的故障和软件系统的状态。

一般来说，调试者和开发者同时学习 bug 的原因和作用。当他们被要求解决 bug 时，他们首先查看 bug 的现象，并用自己的方式从代码中发现 bug 的根本原因，然后解决 bug。这个特定的 bug 变成了一个有用的知识，并成为他们能力的一部分。随着经验和知识的增加，他们变成了更有能力的调试者。这是一个缓慢的过程，和个人的能力有关。一个调试者学到的经验不是很容易就能传授给另一个调试者的。不过，bug 的现象和原因的收集可以提高他们的工作效率。开发者和调试者可以查看以前的 bug 并从中学习。他们可以吸收其他人的经验，并把它作为己有。

开发者通过观察相同原因的 bug，可以确定相同类型 bug 的方法，并保证不再发生同样的错误。这和前面一节“怎样预防 bug”中提到的预防 bug 的方法有些不同。在前一节中，我们讨论的是对特定 bug，确定 bug 预防技术的方法。不过，使用 bug 度量方法，开发者可

以查看所有的 bug 并使用普遍的 bug 预防方法。这是一个查看系统错误的技术，这是 bug 预防概念的一部分，将在第 12 章中讨论。

通常，bug 是不正确过程的结果。工程管理者可以用 bug 度量决定是否可以通过过程的修改来帮助预防 bug，或者使调试更有效。另外，预先的 bug 度量描述可以使项目管理者确定调试的历史代价和 bug 的发生率和解决率。这些数字可以帮助项目管理者确定软件维护的代价。关于和调试相关的软件维护将在第 14 章中提到。

11.6 小结

在本章中，我们知道了在 bug 解决之后应当而且必须做的一些事。这些工作的目的是把调试过程由被动变为主动。我们要求自己用以下五个问题来预防 bug，而不是等着 bug 的出现。

- ◆ 我是否在其他地方犯了相同的错误？提出这个问题，可以帮助我们解决在代码的其他地方的相同 bug。
- ◆ bug 后面隐含了什么？提出这个问题，可以帮助我们预防因为修改而引入新的 bug。
- ◆ 怎样预防 bug？提出这个问题，可以使我们从错误中吸取教训，这样我们就不会犯相同的错误。
- ◆ 怎样较早发现这类错误？提出这个问题，可以使下一次的调试更容易。我们试图理解发现 bug 的过程，并试图寻找使这个过程更好、更有效的方法。
- ◆ 我是否有所提高？提出这个问题，是为了测量我们技巧和所开发软件的进步。
- ◆ 对这些问题的回答，可以帮助预防 bug 并使 bug 更容易发现。在第 12 章中，将讨论这些问题的答案。

11.7 bug 问题

在这一章中的例子大多数是 C 语言字符串函数的例子，因此可以把它当成是在错误使用 C 语言字符串函数时产生的一个 bug 难题。考虑以下一段代码：

```
1. #include <string.h>
2. int main(int argc, char *argv[])
3. {
4.     char firstname[20];
5.     char *lastname;
6.     char *ptr;
7.     char *username = NULL;
8.     if ( argc != 2 ) {
9.         printf("Usage: %s username\n", argv[0]);
10.        exit(1);
11.    }
```



```
12. username = argv[1];
13. printf( "username:<%s>\n", username );
14. /* Find the space character */
15. ptr = index( username, ' ' );
16. *ptr = NULL;
17. /* copy the first part of the user name */
18. strncpy( firstname, username, strlen(username) );
19. ptr++;
20. /* copy the last name of the user name */
21. lastname = (char *) malloc( sizeof(char)*strlen(ptr) );
22. strcpy( lastname, ptr );
23. /* See what it looks like */
24. printf( "first name:<%s>\n", firstname );
25. printf( "last name:<%s>\n", lastname );
26. return 0;
27. }
```

这个程序的目的是查找用户名字的名和姓。用户名字通过命令行参数输入。我们假设用户的姓和名被空格分开。你能够发现代码中所有的错误吗？

第 12 章 bug 预防

确实，我们当中的大多数人对计算机的信息战都很有经验。你可以和同事分享不同战争的经验，包括连续几个晚上的通宵、渐增的软饮料和一个个空的比萨饼盒子（根据个人的口味，也有可能是中国菜）。在很多情况下，你的努力会有好的报答：你发现了 bug，并且解决了它；管理部门很着迷，第二天就通过产品；在你得到物质和精神上的奖励前，你就会被当成真正的英雄看待。当我们从学校毕业，第一个工作就是软件工程师，我们将会为有整个晚上去发现一个很重要但又很困难的 bug 的能力而感到自豪。我们把 bug 看成是需要征服的挑战。同事们可能投来称赞和怀疑的眼光，他们的评论是“你连续工作了几个晚上？”“喔，你解决了问题？”“你怎么发现它的？”。听到这些评论你当然很高兴。

几年之后，虽然这些评论仍然存在，我们已经不再有激情进行调试的探索。我们首先要看一看 bug 的价值，我们观察相同错误重复的次数。系统类型的 bug 出现，而不管计算机的语言、操作系统、行业或环境，这些都过时了。我们不想去解决相同的 bug，无论有没有必要；我们也日益成熟了，并有了专家的感觉，我们一个星期只要工作 40 个小时，并且不会有在早晨 3 点从网络上接到紧急处理通知的情况。我们喜欢事情进展如此好，以至于公司里没有人注意到我们的存在。这就是软件的瓦尔哈拉殿堂，这是从无数的战争中得到的报酬。怎样才能达到这个状态呢？我们确保代码的安全，试图通过提出一些正确的问题来理解所要解决的问题，在实际环境下学习交替使用，从自己的错误中学习，从而能够快速而简单地发现、定位和修改 bug。实际上，我们所进行的这些过程就是 bug 预防。

12.1 什么是 bug 预防

bug 预防就是减少 bug 的过程。bug 预防的目标是减少奇怪的错误和在我们犯错误时能建立一个机制，用以快速而简单地发现、确定和排除问题。第 2 章、第 3 章和第 4 章是描述 bug 的本质和产生的原因。在这些章节中，我们对这些概念的理解有了进展，对在软件中排除和预防 bug 也有了进展。

在本章，我们列出和讨论一些 bug 预防技术。一些技术在软件开发生命周期中的每个阶段都使用，而另一些技术主要是应用在特殊的阶段；一些技术应当在组织层次上进行考虑，而另一些技术只要由个别开发者考虑。我们对有些技术进行详细讨论，但对另一些技术只给出概念。对于那些概念性的技术，只需要理解它的概念和怎样把它应用在特定的环境下，适当地，我们提供了一些例子。不过，这些例子很简单，只是为了说明我们的观点。

bug 预防的观念

软件的产品质量的首要标准是由宇航公司提出的：无 bug 软件。在这个期望下，会让人感到谦卑。只要是人，那么就会犯错误。谦卑的同时，要开动脑筋查找问题，然后很困难地完成任务。如果像我们鼓吹的那样，这就可能是我们的原因。质量是良心的判决，不会引起事故。

假设，你是为了查找问题而来阅读这本书。你就已经开始了第一步。如果要制造高质量的软件，并且已经意识到自己的局限性，就有两个较基本的 bug 预防心理能帮助达到生产质量软件的目标：总是很执着，总是很忘我。

执着

在我看来，最重要的 bug 预防观念是知道自己肯定会犯错误。我们必须假设我们开发的软件存在了很多的 bug。我们要变得执着，不要认为已经从 bug 中解脱出来了。

在假设事情很不好的情况设计、执行软件系统，将需要一个完全不同的思考方法，这就迫使你提出很多的问题，例如“如果 Y 发生了，要我怎样处理 X？”“Y 会发生吧？”“Y 的发生会引起什么事件发生？”和“如果 Y 发生，我该怎么办？”。如果很执着，你就会认为需要检查变量值和考虑失败结尾、容错和冗余等等。不仅仅是停止在这上面，你还会想到其他开发者不正确使用的接口引起你代码的错误或者他们代码的错误。所以你要细致地定义使用接口的办法，写一个简单的使用接口的例子，并写相关的文档说明引起的副作用和所需的输入输出。通过使他们不可能不正确使用你的代码，来帮助他们在代码中预防 bug。

执着意味着要两次检查所使用接口的输入输出情况，并且不是假设，特别是在对接口不熟悉的情况下。向自己提出问题，例如“参数顺序是不是正确？”“这个方法是否为释放的目标提供存储器分配，或者仅仅把指针传递给所管理的目标？”

如果对软件中的 bug 报执着的态度，就会很少碰到系统输出和软件行为的矛盾。当遇到无法解释的行为时，你的任务是发现系统为什么会出现这种行为。即使不存在 bug，你也会对软件有更多的了解。这些理解将有助于你发现 bug，并解释以后发生的行为。

执着意味着在每次人们提到特定的软件行为时，你就要求和他们一起去发现所有的细节。接着，你在代码或系统上理解系统是否可能出现这种情况，以及为什么会出现这种情况。不要放过 bug 报告，即使问题确实是由用户产生的。一个用户错误意味着用户接口的错误或混乱，这是错误使用系统引起的。这些不正确的使用会引起系统环境的问题，甚至会激发软件中的一些模糊代码路径的 bug。你应当假设用户不会犯错误。如果由于一个用户的错误而引起系统的崩溃，说明系统太脆弱，必须进行修改。

不过，执着不是意味着你必须把任何事分析到最后的细节，麻痹的分析不能帮助你解决问题或预防 bug。

忘我

不久前，在一个有关软件设计的会议上，一个高级的开发人员表达了他设计的思想，大

部分的开发者同意他的观点。但存在一个根本上的不同观点，两个组之间既不同意折衷，也不同意交换意见。后来，成为了一个严重的争议。这两组发生了冲突，在会议结束时也没达成一致的观点。

当在工作中很自私，那么就失去了客观性。在出现这种情况时，你就不能作出评语，就不能相信 bug 报告，就会变得抵触。这些情况都会导致 bug。记住，当别人指出你设计或执行软件的错误时，他们就把这些错误记在纸上，他们不是讨论个人的错误或埋怨，这仅仅是工作，只是为了让它更专业。

12.2 一般技术

质量软件的意义就是质量软件几乎没有 bug。但在开始写第一行代码之前，就可以开始 bug 预防。在第 3 章和第 4 章，我们已经讨论了软件修改会引起 bug。这就意味着随时都有可能，我们可以把现成的技术当成通过减少软件修改来预防 bug 的方法。我们讨论了人的错误，对需求和设计的错误理解，缺乏交流和对源代码 bug 的不恰当测试。这些技术能帮助我们发现和捕获 bug，对减小成品环境中奇怪的 bug 有很大的价值。复杂性是引起 bug 和错误的主要原因。既然这样，如果能减少和控制复杂性，我们就有希望减少 bug 的数目。

在这一节讨论的一般技术，如果执行正确，就会对错误给予打击。这些全局的预防措施能在个人和组织上应用。

12.2.1 预防 bug 的人

人制造了软件，既是人在软件中产生了 bug，也是人修改软件中的 bug。人是和 bug 作战的第一道线，代表性的人是管理上的问题。作为一个软件开发者，问题是我们通常要和其他人一起工作。有时，我们有机会选择我们合作的人。有时，我们会影响管理者的想法和决定。

雇佣优秀的人

面试不是一个科学的过程，仅仅通过简历和几个小时的面谈，很难把优秀的工程师和一般的工程师分开。这并不是说他们知道哪种开发工具，也不是说他们的工作时间，而是无用的工程师比起优秀的工程师会产生更多的 bug。无数的研究表明，雇佣优秀的人是划算的。关于这个问题和技巧有很多的论文，可以用这些论文来指导面试应聘者，我们就不在这本书上重复了。不过，我们希望雇佣问题的解决者，我们发现优秀的工程师就是优秀的问题解决者。他们可以解决很多语言、工具和问题域的问题。优秀的问题解决者同时也是优秀的调试者。

对于一个高级软件工程师，我们可以从他身上看到以下几点。提以下几个问题，也可以帮助确定他们的特点：

- ◆ 是否能够告诉我们一些我们不懂的事？是否对问题或解决方法有新的见解？即使我

们不同意他们的观点，我们也宁愿要这些有合理观点的应聘者。

- ◆ 对概念是否有独特的看法？优秀的高级工程师对问题有独特的见解和自己的解决办法。优秀的 C++ 工程师应当对抽象类和方法的概念有独特的见解，并用自己的语言和类似的方法表达出来，而不仅仅是说出抽象类和方法的定义。

- ◆ 能不能为我们权衡方案？软件工程师就是用来权衡方案的。优秀的软件工程师知道所设计方案的优点和缺点。他们根据手头上的信息制定方案，有时是正确的、有时是错误的。但他们至少知道自己的缺点。

- ◆ 给出一个未完成和不明确的问题表述，优秀的软件工程师就会避免假设并询问问题，来帮助理解我们出现的问题。而差的工程师就会根据他们对问题的理解而给出答案。

保持连贯性

因为很多组织受缺乏文档的影响，这就有必要在一个工程中保持连贯性。理论上，连贯性由技术个人来提供；但为了节省开支，这个角色由其他人来担任。这个角色的人就是提供工程的背景信息，并成为软件开发资源的新人员。此外，这个人还要熟悉软件，这样，在解决 bug 和预防 bug 时就更有效。

最好的保持连贯性的方法是保留住项目的原来开发团队，直到项目的生命周期结束。这显然是有困难的，因为如果一个地方没有新的挑战，他们就会跳槽，感觉不到兴趣的人就离开了公司。所以，在知道他们能提供系统形式的技术迁移的情况下，或者通过文档、或者通过直接的指导，应当允许团队成员到不同的项目去。不管怎样，这些知识都会传递下来。

使用备份

保持连贯性的另一个办法是使用备份。每个项目成员都应当有备份，这样就能和原来的开发者一样有效地工作，这样也能预防经常跳槽综合症。对那些不熟悉这个综合症的人，这里有个对综合症危害的描述。如果这里有一个软件的关键部分，只有一个高级的软件工程师知道它。一天，这个软件工程师跳槽了。现在，在公司里就没有人知道这一部分关键的代码，并且也没有相关的文档。在这种情况下，没有做备份带来的代价是多大啊？创建立刻备份方法就是使用极端编程方法学提到的成对编程技术。当有两个人做同一个项目，他们之间就要进行交流，并且学习对方的任务。结果，知识就传播开了。

在环境中使用正确的人

软件工程协会（Software Engineering Institute, SEI）定义了能力完备模型（Capability Maturity Model, CMM），用来测量软件开发组织的完备性。SEI 定义了五个 CMM 层次，第一层次水平的组织是混乱的，没有过程的；第二层次水平的组织的特征是过程的重复，如果你的公司工作在 Internet 上，并且不能提供第 2 层的功能，那么就不要欺骗自己，雇佣一个在软件工程过程权威的人。不管目的多好，你将没有时间执行和设计这个过程。我们中有多少骨干说不要高级的管理？雇佣一个黑客加入，他们将会高兴，他们不会注意，他们将进行工作。一个优秀的黑客可以用最少的资料用技巧对系统进行诊断。他们喜欢投入，他们能快速发现你日常系统的改变。另一方面，如果你雇佣了面向过程的工程师，他们将指责和抱怨

缺乏的过程和文档。如果在工程的中间提出需求改变，他们将要花费三个月来分析这个改变。结果，每个人都讨厌其他人，他们在一起就不高兴。

如果你的公司达到 CMM 三级，并且你雇佣了黑客，这个黑客就能在几个星期内解决问题。如果你管理一组“Dirty-Dozen”工人（雇员都是个人工作的，就像 1967 年电影中的战士一样不守规矩和武断，可能他们认为他们自己完成类似于自杀的任务），那么任何试图引进有格式的过程将会失败。为了形式化产品，你就要用能处理过程的人来替代他们。如果你对您的环境真实评价，并且所雇佣的人员在工作，那么就能为自己节省不幸。

修改剧或主角

如果工作在一个黑客的环境中，就注定了不能工作在 CMM 第一层的环境中。黑客也有一个过程，攻击不是无意识的也是记录和可重复的过程攻击。为了获得你在项目中管理者或 CTO 的位置，你必须计划好如何使用人员，使他们最大限度发挥潜能。并且要找一个能把他们放在合适的可重复过程中的方法。体育运动就是一个类似的现象。作为一个足球教练，你是否问过你的四分卫，谁的准确度在 20 码之内，但力量不大，整场都能踢出 40 码远？这个谜在你的脑海中浮现。如果进攻线脆弱，为了坚持超过 3 秒，你是否把四分卫调到这个位置上来？对方的防卫线可能要四分卫的头。在篮球中，如果你有个高大的队员，你是否都进行三分球的投篮？当然不，作为一个教练，你将要根据当时的情况制定比赛规则，用你的强点对付对手的弱点。管理一个软件开发队伍也一样，软件开发过程不是一条真正的组装线，不像排列成面包的切割机的形状。必须制定出一个对人员的安排，通过对目标和每一个步骤的目的理解就可以制定出这个使用计划，并在你的环境中达到总的目标。

当然，另一个选择是改变开发人员。可以利用过程来解雇任何不合群的开发者，这是很简单的一件事。但你首先最好要有一个替代这些被解雇人工作的策略。

发展而不是革命

有人认为没必要在一天之内改变世界。发展是生物学的系统，也可以应用在软件开发过程中。在发展的方法上，不要大规模改变团队工作的方式。相反，在每个时刻只进行一个改变。你引入的改变有赖于环境和你所一起工作的团队。以下几点是需要考虑的几个技术：

- ◆ 孤立混乱的元素。在大多数的软件开发环境中，最混乱的元素是开发的团队。但这并不意味着测试的过程也是混乱的。测试团队可以进行 bug 追踪、生成测试计划并记录测试结果，这和开发者的工作是完全独立的。开发团队可能从来就没有检查 bug 数据库，但存在的 bug 要求管理者把它分配给黑客，也要求测试者追踪哪些被解决了哪些没有被解决。也许这对整个开发过程不是有效的，但这是必须做的第一步。而且，通过孤立混乱的元素，就可以围绕这些因素生成结构、组织和过程，这些元素将替代和改变任何适当的混乱元素。

- ◆ 在一个时候引入过程元素。开发团队的执行是过程因素，因为他们认为这是重要的，或者可以使用一个人，这个人的工作是在目前混乱的环境和将来过程化的环境中建立桥梁。例如，很多开发者和黑客认为配置管理环境是有用的，并且愿意承担起使用的不便之处。他们相信没有配置管理过程的代价大于被迫使用这一过程的代价，这就引出了下一点。

◆ 预防动机。这有两个动机：正面动机和反面动机，你可以使用任意一个。正面动机经常被使用，所以我们主要讨论反面动机。一个过程有一系列乏味的任务，这些任务并不都是黑客愿意做的，原因不仅仅是讨厌。但过程带来的痛苦是很大的。例如，没有人愿意凌晨 4 点工作，也没有人愿意在周末工作。所以，当灾难出现或者开发者遭受这种灾难的折磨时，就可以引入文档和培训的概念。例如，可以提醒他们“如果写一个培训向导，那么操作人员就能处理他们试图处理的问题”或者“如果你有一个助手，那么就可以把这个人叫来代替你。”当然，这里有个危险就是可能开发人员只是说说而已。在这个情况下，就有必要让另一个人来代替他的工作。

◆ 安排替换。如果开发人员坚决反对过程形式，那么就要替换他。在混乱的环境下，替换一个开发者很困难，这是因为只有这个开发者知道他的工作。所以，就不能仅仅解雇这个开发者了，还要知道他所知道的知识。这显然是最后的手段。在当前软件工业中，不管公司环境怎样，安排替换是必要的策略。因为开发人员来去自由，你可以用来对付的惟一办法是扣住雇员的抵押品。有无数的技术可以用来执行替换计划。文档、向导和培训仅仅是一些例子。

12.2.2 问题原因分析

一个聪明的人会说我们应当从失败中吸取教训。bug 当然是错误，但它不会让我们从中吸取教训，即使是在我们已经解决后。在开发阶段，我们写代码，测试它，发现 bug，修改它，接着我们处理下一段代码或下一个 bug。我们从来没有记录和分析 bug。当我们在测试阶段，bug 可能会被报告和记录，但是这些 bug 很少被收集和分析，用来发现可能出现问题的模块和确定那些能从中学习和预防的系统的错误。通过一些错误分析手段可以成功地预防错误。1981 年，NEC 开发了一套错误原因分析仪，可以追踪和监视发生错误的概率和观察在 10 年应用中错误逐渐减小的趋势。IBM 和计算机科学公司在他们的软件开发过程中也开发了一些错误原因分析仪，在两年时间内，错误率减少了 50%。

分析错误的原因可以在企业和个人两个层次上完成。在企业层上，错误分析必须和项目、会议和格式结合成软件开发过程的一部分。在个人层次上，这个过程可以是一定的形式和机会。不管错误分析的形式怎样，目标是相同的：发现错误的原因，寻找预防这些错误发生的方法。在这一章中讨论的一些观点是分析我们自己的错误和所看到的其他人的错误的结果，其他一些观点是从经验研究中精选出的。

系统错误

错误原因分析对预防 bug 的主要贡献是识别系统错误。这些系统的错误占典型软件工程中 so 发现错误的绝大部分。如果能预防这些系统错误，就能提高软件的质量并减小错误的出现。

一个系统中的错误如果经常发生，它是有相同原因的。如果错误经常在特定的时间和地方出现，它们也是系统性的。在第 3 章中，讨论了对软件的修改时引入错误的主要原因。所

以, 软件修改引起的错误是系统的错误, 可以使用技术来减小修改的总数达到减少错误的目的, 也可以减小修改对软件质量的影响。对系统和程序的理解是修改 bug 的前提, 所以必须发现不同的技术来帮助维护者和开发者理解系统和程序。如果是在系统的接口处出现 bug, 你就应当对所设计的接口的方法进行评价使出现的错误较少。

发现系统错误的关键是观察模型。有时, 这个错误具有迷惑性和教育性, 可以帮助我们排除大部分的错误。我们需要一个能帮助发现 2000 英尺之外错误的办法。

鉴别系统错误

为了观察我们可能出现错误的类型, 有必要记录先前所犯错误的历史。应该记录已经发现并修改错误的哪些内容呢? 至少, 一个错误记录应当包含以下几个问题:

- ◆ 是什么错误? 这是对错误的描述。可以使用这个描述来分类和收集类似的错误, 用于模型分析。

- ◆ 错误出现在什么地方? 我们对在不同层次上的错误出现的地方感兴趣。我们希望知道哪些软件模块包含错误。我们希望知道错误出现在模块的哪些地方, 是在接口处还是在初始化的地方? 是在数据结构定义还是在计算过程中? 是出现在数据传输过程中还是在条件逻辑中?

- ◆ 什么时候引入错误? 我们要知道在过程的哪一个阶段引入了错误。这个错误出现在需求阶段、设计阶段还是在实现阶段? 当错误出现时, 我们做了些什么? 这些问题的答案可以帮助我们理解我们要改进哪个阶段。

- ◆ 为什么会引入错误? 一个人可能会想起引入错误的精确环境。这些原因可能是投机和经验的猜想。不管是哪种情况, 这些原因能帮助我们找到预防错误再次发生的途径。

一旦有了错误的记录, 就能根据错误出现的时间和地点对错误进行分类。有了这些分类, 就可以试图发现在这个环境下发生错误的根本原因, 并且设计模块和过程来排除错误的原因。

个人错误原因分析

在没有组织帮助你较好理解所试图修改的错误的情况下, 也可以进行错误原因分析。一个方法是, 准备一个工程手册, 在上面记下你的想法和你修改的 bug。另一个方法是, 在解决了一个 bug 之后, 就在代码中直接留下你的修改文档。之后, 当它们再次出现, 就可以收集这些文档来回顾你的错误。下面一个例子, 就是说明保留一个文档对更容易重新找到和分析 bug 的作用。

```
1. //some codes
2. /* ## 12/25/2000 orderEntry.cc getOrderItems() ych
3.    ## I read the design incorrectly. I was supposed to
4.    ## return a list even if there was no item in the order,
5.    ## but I returned a null when there was no item in the order,
6.    ## and that broke the orderManager.
7. */
8. // code with defect that was fixed.
```


`##`是注释的符号，这样就可以容易扩展与错误相关的注释。注释中的时间、文件名和函数名让你知道错误被发现和解决的时间和地点。这个描述告诉为什么犯了错误和怎样修改这个错误。用一个像 `grep` 这样的工具，就可以很容易收集所有的错误并提供错误的概述。

代码中错误注释的另一个好处是让维护者容易读代码并知道所作修改之间的关系，让维护者知道为什么要进行这样的修改。

另外一个可以记录错误信息的地方是大部分的版本控制系统的注释部分。大部分的版本注释系统允许在库中检查文件时记下对文件的修改。相同的信息也应当在注释中作记录，以备利用后来的时间进行分析。

12.2.3 检测错误

因为 bug 是软件错误的表现（在第 3 章定义），这就是为什么要使错误在运行时变成 bug 之前，发现错误、修改错误来预防 bug，特别是在产品的环境中。有两个发现软件错误的途径。第一个途径是检查，第二个途径是在我们的控制下把错误变成 bug，也就是测试。检查是一个静态的技术，相反，测试被认为是一个动态的技术，将在第 13 章中讨论。

检查

检查就是要注意需求、设计和实现，以此来发现任何可能被错过的错误。在正式的软件开发方法中，在评审阶段就出现了检查：需求评审、设计评审和代码评审。有很多关于检查作用和检查过程的研究。在软件开发领域中，我们对 bug 的观点是，检查能使开发软件产品的时间减少 10 到 100 倍，测试时间减少到 85%。

怎样才能不用去检查

我曾经在不同的公司进行一些正式的设计评审和代码评审。典型的，工程管理人员和主管工程师需要一些评审部分。这些评审人员将收到一些材料，有时在一个星期，但通常是评审会开始前的一两天。在设计阶段到来的时候，我和我的随从评审人员集中在会议室里，设计者和开发者递交他们的材料接受评审。检查有什么用吗？实在地说，不是像我们做的那样。问题也出现在其他地方：公司要进行一个检查过程，但是公司没有人确实有检查的线索。需求没有被评审，这就意味着即使设计和实现很好，但它们还是要被修改的，所以最好在需求中发现错误。代码中主要的注释评审结果违反了代码的规则。设计代码的关系很少被讨论和提起，主要是因为评审人员在评审过程中很少有机会理解设计的系统。设计评审变成了设计人员讨论的地方。很难对设计作出评论，因为评审人员没有机会设计和理解设计是否满足需求。

怎样去检查

检查过程的关键是要有质量检查人员。这些人对问题和解决方法很熟悉。可以通过为他们准备资料和学习资料的时间来帮助创建一个知名的团队，可以通过使用更多的检查人员提高团队的知识。下面四点是建立成功检查机构的关键：

- ◆ 熟悉——检查人员对被检查的问题、产生、解决方法和执行过程要非常熟悉。在检

查代码时，检查人员必须知道软件的功能，知道软件怎样和系统的其他部分协调工作。检查人员还应当熟悉编写代码的语言，知道执行不同代码分支潜在的危险。在检查设计时，检查人员必须熟悉系统的功能和性能需求，还应当知道不同设计元素的交替使用。检查者对问题和解决方法越熟悉，他们在检查过程发挥的作用就越大。

◆ **准备**——帮助检查人员理解问题的一个方法是为他们准备检查。准备意味着开发人员必须尽可能地为检查人员提供文档和资料，也意味着检查人员要有充足的时间来理解问题和解决问题的办法。

◆ **更多的眼睛**——使用更多的检查者检查相同的资料将有助于提高发现错误的效率。毕竟，单个人对问题和解决办法的理解能力是有限的，他会变得疲乏和讨厌。在检查团队里增加第二个检查者有助于提高整体知识，也能避免其他检查者的弱点。同样，增加第三和第四个检查者对检查过程的效率是有好处的。不过，我们做同样的假设，太多的检查者会有副作用，将会出现太多的声音。建立合适数目的检查者机构的一个方法将在下一部分“检查技术”中提到。

◆ **跟踪检查数据**——使用检查团队作出的检查结果数据有助于进行错误原因分析的团队的工作。这些信息将帮助他们更有效地修改他们的过程。而且，跟踪检查数据也能帮助管理团队看到开发过程的进展情况以便作出决定。

检查什么

每一件事都要检查，当在最小的情况下，我想至少要检查以下内容：

- ◆ 详细的需求分析
- ◆ 详细的设计
- ◆ 代码
- ◆ 测试方案

这些项目必须进行内部的正确性和复杂性测试。另外，项目间的关系也要进行检查。如果设计没有满足需求的要求，这个设计是错误的，要进行修改。

检查技术

总的来说，有三种检查技术：特别技术、核对表技术和基于部分技术。在特别技术中，所有的评审人员所得到的责任是相同的，就是在文档或代码中寻找错误。在核对表技术中，给评审者一个列表，它和特别技术在其他方面是相似的：所有的评审在整个文档和代码中查找错误。在基于部分的技术中，每个评审人员的问题集中在特定的部分上，这些部分可以是需求特征集合、用户情况集合或者是功能集合。研究表明，整个需求检查的性能可以提高基于部分技术的使用。令人惊讶（或者是不令人惊讶）的是，相同的研究也表明评审对整体的错误率的减少没有任何贡献。

在基于部分的技术上，个别的评审人员使用系统的程序标志一个小数目的特定讨论。这个技术能应用在需求、设计、编码和测试计划的评审上。不是每个评审人员在整个系统中作相同的评审工作，而是每个评审人员负责不同的模块。例如，在电子商务系统中，一个评审

人员负责购买装载模块，另一个评审人员负责订货模块，第三个评审人员可能负责所有模块之间的接口。

基于部分的检查技术的一个优点是负责需求评审中的特定部分评审人员能对所给部分在设计、实现和测试计划中提出特定的问题。因为评审人员已经对部分的需求做出评审，我们就能假设他对需求很熟悉了。所以，评审者就能很好地判断软件开发周期最后阶段产品的正确性。

不过，基于部分检查的缺点是需要太多的资源，花费太大。因为一个评审人员只负责一小部分，那么整个系统就需要很多的评审人员。

代替正式的检查

检查对预防 bug 是很有效的（如果你能提供这个形式）。检查需要在管理者的控制和具体的约束下进行。不过，如果没有那么多的时间和资源，那么我们怎么才能在没有正式的检查过程下取得检查的目标？在我们回答这个问题之前，让我们真正理解一下检查怎样帮助我们预防 bug。

检查并非源于软件工业。我们在校的时候，我们的作业就被老师检查。当我们重复检查自己的数学作业，我们就是在检查自己的工作。一个检查人员就是一个校对，检查的整个思想就是一个人（检查者）检查另一个人的工作。如果检查人员熟悉问题和解决的办法，那么检查者就应当能评价工作正确性并指出缺点。

检查自己的工作

在高校和大学里，当我写完作文时，我的英语老师总是要我校对自己的作文。他说，最好把作文先放在一边，过一会再来读。很明显，这将更容易发现错误。

这个建议也同样可以应用在软件开发过程中。当你集中精力写需求文档或开发程序时，你尽量避免出现错误。但一旦错误发生了，你总是希望能通过读文档和程序来捕获错误。检查自己工作的主要问题在于当你刚刚完成你的工作就要去发现错误，这时，你可能还被你自己的想法所包围，并在遇到问题时总是认为它们是对的。

发现自己问题的最有效方法是先把问题放在一边，然后再回头看它。最后，你可能很明显就发现错误。如果你没有时间，就需要使用一些方法进行这个工作。例如，在开发一个项目时，你负责了多个部分。你可以完成了一部分后，尽可能检查它，然后再进行另一部分。当你在开发第二或第三部分时，你可以返回执行第一部分并检查第一部分。

考虑另一个检查自己工作的方法：很多开发者喜欢在实现和测试都做好的情况下写文档。这是进行检查的好时候。在你为接口部分写文档时，可以利用这个时间再次访问和评审代码。这样，既可以写好文档，也可以有机会在自己的代码中进行测试，发现 bug。

如果时间太短，把工作放在一边可能有困难。到最后，你可能没有机会检查你的成果。还有一个检查自己工作的缺点是如果你对第一次的问题、过程、工具或解决方法不理解。第二次，你可能认识不到你的错误。

同类和异类检查

如果时间太短，就必须在同一时间进行开发和检查。这可以通过应用复制程序的方法来完成。在极端编程方法中，复制程序就产生了两个人在同一时间处理同一问题的思想。

复制程序也能应用在检查和软件开发生命周期的其他阶段，包括需求分析阶段、测试计划阶段和设计阶段。一个人单独工作是没有理由的。另一方面，两个需求分析人员进行一个需求分析是没有必要的，同样，在两个测试人员进行一个测试工作也是没有必要的。使用异类分析是很有用的。

在异类检查团队中，两个不同角色的人在一起进行相同工作。这个方法对于完成检查有很多优点。第一，两个不同角色的人工作在一起能相互学习对方的任务和责任，并且知道怎样通过满足对方的需要使对方的工作更容易。第二，公司的核心作用增加，因为雇员学习了不同的工作方面。第三，不同角色的人可能在他们的工作上有很大的进展，并提前想到他们的任务。

例如，一个需求分析者和主要设计者可以在需求阶段一起工作。这样的好处是设计者能对需求提出理解不到的地方和技术上不能满足的地方。而且设计者也能提前知道任务。技术人员工作在需求阶段的一个缺点是太早做出了技术上的决定。

看另一个例子：有一个需求分析人员和测试者一起进行测试，这样就结合了两个方面的技巧生成测试计划，不仅能根据需求文档来测试软件，而且也测试了需求的初衷。毕竟，其他人比需求分析者更有资格回答有关需求的问题。而且，通过观测测试者如何使用需求来生成测试计划，需求分析人员在生成需求时就更具有可测试性。

异类测试的主要困难是人员的选择。这个技术的集合体有时候是合适的，有时候却不是这样的。需求分析者和代码开发人员一起工作可能是困难和不可理解的。另外，有些人不愿意学习和理解不同的工作，有些人没有能力学习不同的工作。有些人被强迫在不熟悉的规则下工作，这将对检查过程没有好处。这些人将不能成为异类检查团体的一部分。

生产者—消费者关系

双向检查的一个替代方法是在项目成员之间建立生产者—消费者关系，让消费者检查产品的工作。因为消费者完全使用产品工作，双方都有自己的评估和检查兴趣，作为传送和接受的标准。这个传送和接受的标准可以是正式的，也可以是非正式的，这由你公司的氛围决定。

在软件开发过程的不同阶段都会出现生产者—消费者关系。设计者进行了需求者的需求分析，开发者使用设计者提供的设计文档，测试者消费了需求和实现。在这些情况下，消费者为了执行他们的任务，必须用到制造者的结果，那么，消费者首先要确保他们所使用的产品的质量。

同样，在软件开发周期的相同阶段，也有可能建立生产者—消费者关系。例如，在交易层，一个开发者能开发一个模块，另一个开发者在表达层上使用图形用户接口（GUI）。因为图形用户接口开发者需要从交易层使用应用程序接口（API），他就是一个消费者的角色。

作为他工作的一部分，GUI 开发者就要评审交易层的实现，确保交易层产品的正确，并理解交易层的设计和使用。

检查的风险

好的检查结果会产生假的信任度。有可能出现开发过程正确执行，但检查过程有很多错误的情况。这样，项目最后还是被取消，因为在需求中没有解决需求中的基本缺点。这个项目的基本要求没有在项目中正确定义和合理化。另一个研究表明，项目的检查数据显示项目在实现阶段做得很好。但是，因为需求的不断修改这个项目没有成功使用。在这种情况下，需求和设计没有被检查，仅对代码的检查不能确保项目的成功。

这个教训很简单：检查有利于预防 bug，但不能确保项目成功开发。

12.2.4 重用

关于软件重用的一般观念是有利于用可重用的构件建立系统。它的优点是能提高效率和软件的产品化开发过程，减小维护费用，并提高软件的可靠性。重用研究的经验表明了这些看法。例如，在一个研究中，重用没有修改的代码在每个模块中只导致 0.02 个错误，而写一个新的代码在每个模块中会引起 1.28 个错误。

重用的阻力

既然软件重用有这样的优点，为什么我们不尽可能地进行软件重用？实际上，我们使用的重用超过了大多数人的想象。为了临时的使用，我们没有写操作系统；我们重用现有的文本和代码编辑器；我们购买和使用现有的数据库服务器和 Web 服务器。为了临时的使用，我们没有写编译器。C++ 语言编程者重用 STL；Java 编程者重用大量的 JavaAPI。我们调用这些构件工具，并且很容易重用它们。然而，当用它们来构造我们的软件时，仍然要为这种的临时使用编写代码。Lynex 和 Layzell 进行的一项调查，彩访了 19 个组织来发现阻止重用的原因。他们发现了 9 个反对意见，跨越了管理、过程、经济和技术等领域。

从技术上考虑，很难生成可重用的构件和软件模块。可重用的构件要求简单的接口，并且可重用的构件要做文档。而且，对于工程师来说，用一个新的解决方法比使用构件来组合更有意思。另外，当出现 bug 时，很难从别人的代码发现 bug。

从经济上考虑，开发可重用的构件和模块要额外的投资。分类、发布、宣传和记录可重用的构件要花费钱和资源。在短期运行中，维护可重用构件的收集也不便宜。

现有的过程很难修改，在 Internet 网上工作的时代意味了当前的问题有最高的优先级。为什么要修改能工作的事情？这个惯性使公司很难修改更多的有希望的过程。

基本上，重用不是技术上的问题。重用的困难是怎样把它和业务融为一体。使用重用技术，公司不能看到投资的立刻效益，它要求组织和管理承诺来保证运行。这就意味了公司组织结构需要重新工作，也意味了管理需要看到重用的利益。重用需要从上至下的委托。

不过，Java 编程提供了从上到下重用委托的异常处理。从一开始，Java 程序员就要通过下载可用的 SDK 和 API 访问 Java Web 站点，并且他们也提前编译了 API。当我不得不写

一个 Java 程序时,我要做的第一件事是到 Java 站点去查找有没有这样的 API。我通过键盘查找需要解决方法的内容。接着,通过适当新组中的档案文件查找需要的重用软件。Java 也提供容易做文档的工具。得到这些正确的环境和正确的工具,开发者从一开始就被鼓励使用重用软件。

没有修改的重用

我们怎样使用重用技术来预防和减少 bug? 在软件模块化研究中,有一个在大的通信系统中重用一个大到一百三十万行代码的模块的例子,研究结果表明:

- ◆ 当重用没有修改软件模块时,错误率大约是 0.52 个错误每个模块。
- ◆ 当重用修改过的软件模块时,错误率大约是 1.75 个错误每个模块。
- ◆ 开发相同功能的软件模块,错误率大约是 2.29 个错误每个模块。

另一个关于软件错误的研究表明同样的结果。不过,对 30K 到 100K 行代码量的研究结果表明:

- ◆ 在一个新的软件构件中,平均错误率是 1.28 个错误每个构件,每个错误平均花费 2.1 个小时去解决。

- ◆ 如果软件构件是经过大修改重用,平均错误率是 1.18 个错误每个构件,每个错误平均花费 2.3 个小时去解决。在这个研究中,大修改是指软件中的修改大于 25%。

- ◆ 如果软件构件是经过小修改重用,平均错误率是 0.58 个错误每个构件,每个错误平均花费 1.1 个小时去解决。在这个研究中,小修改是指软件中的修改小于 25%。

- ◆ 如果软件构件没有修改地被重用,平均错误率是 0.02 个错误每个构件,每个错误平均花费 0.02 个小时去解决。

这个研究没有显示任何使人惊讶的结果,相反,证实了我们的经历和常识。被重用了模块,比起新模块,已经经过更多的测试和使用,所以,就出现更少的 bug。修改被重用的模块否定一些优点并引起奇怪 bug 现象的增多。第二个研究的有趣结果发现,对现存的软件模块的完全修改生成软件中的 bug 最难被发现。结论是简单的:在重用软件时,应当重用每个没有被修改和没有定型的模块。不需要去修改代码。即插即用的软件构件是预防 bug 的关键。

可重用构件

提高没有修改软件的重用的方法是重用商业脱架 (Commercial Off-The Shelf, COTS) 构件,比如数据库服务器、Web 服务器、企业资源安排系统等等。另一个方法是建立自己的可重用构件库,EJB (Enterprise Java Beans)、COM、CORBA (Common Object Request Broker Architecture) 包含了一些为可重用构件执行提供接口的最新尝试。不过,Unix 操作系统为可重用执行提供了一个最好的思想。不像 COM 和 CORBA 那样需要复杂的接口和安装构件,Unix 构件是一些通信管道或文件的小程序,这些程序能通过管道连接起来,执行复杂有序的任务顺序。为了执行没有顺序的任务,可以通过简单的循环和条件命令组成的内核脚本来完成。这些 Unix 程序和思想已经被其他操作系统和某些大量简单的任务所利用。

例如,我要在登录模式下运行成品服务程序。这样,当某些事务出错时,就可以立刻观

察到 (a) 服务程序仍在运行, (b) 服务程序在这一刻做了些什么工作。登录文件包含了很多信息, 但在这些信息中没有我需要的答案。把太多的工作花在写一个程序上, 以至不能解析和重现数据, 因为直到问题发生前, 我不知道我该注意什么。这个问题的解决办法是使用 “grep”、“sed”、“awk”、“tail” 和 “more” 等组合命令来自动减小要查找的数据量。马上, 我就可以查找到有助于解决问题的信息。我所遇到的 bug 是对程序发布了一个错误的命令。但因为我没有编译任何程序, 我将知道我是否犯了错误。这个做法简单而有效。

在大学进行项目研究期间, 我和我的团队如此喜欢可重用程序的思想, 以至我们甚至利用简单构件写了完整的系统, 这个系统能被 “make” 命令控制。这个 “make” 程序执行了 “Makefile” 的内容。典型的, make 程序的描述文件被用来编译二进制库和可执行文件。它的最简单形式, make 程序的描述文件是一个基于规则的系统, 由下列规则组成:

```
if ( A is older than B ) then execute Some Command(s)
```

A 和 B 在计算机中是相同类型的文件, 可以是数据文件或者是可执行文件。B 可以是简单的文件, 也可以是文件的集合。代表了 make 程序的描述文件的相同语法规则, 这个规则如下:

```
A: B
    Some command
```

为更多的编译使用 make 程序的描述文件不是一个新颖的观念。make 程序的描述文件也被使用在其他的应用系统中, 例如应用系统的回归测试和自我测试。但是, 怎样使用 make 程序的描述文件和可重用程序构件一个系统? 假如系统包含一系列的任务, 每个任务由一个程序执行, 并且一个程序的输出作为另一个程序的输入。程序的输出保存在磁盘中作为一个基本部分或者是存档的结果。所以一个简单的 make 程序的描述文件如下:

```
output_of_A: A
    run A
output_of_B: B, output_of_A
    run B with output_of_A
output_of_C: C, output_of_A
    run C with output_of_A
output_of_D: output_of_C, output_of_B
    run D with output_of_B and output_of_C
```

如果 make 程序的描述文件第一次运行, 将产生 A、B、C 和 D 的输出, 因为它们先前都不存在。如果没有修改, 再次运行 make 程序的描述文件没有任何作用。但如果因为 bug 的解决或功能的改进而修改了 C, 运行 make 程序的描述文件将自动重新运行 C (因为 C 的输出是旧版本的)。这将调用重新运行 D 的命令。这些不是没用的, 只有那些需要重新计算的数据才会被计算。可以看到任务每一段的输出, 这将有助于调试系统。最大的优点是不必要写一个程序来管理 make 程序的描述文件提供的所有复杂性, 这就意味着可以集中精力解决 A、B、C 和 D 的问题而不用关心这四个任务的外部构件。总之, 可以不必当心和任务捆绑在一起的 bug。如果需要添加任务 F, 只需要花费一点的时间把它加到系统中。

在软件开发过程早期的重用

重用不应当被限制在执行阶段。事实上,在软件开发周期中,应当尽可能早地开始重用。较早的重用有很多优点。我们看来,在开发早期进行重用的最引人注目的原因是子系列生命周期产品的早期重用能派生出早期阶段的产品。一项研究表明,在软件开发周期的概念、分析、规格说明、构造和详细设计阶段将近 100 个类型的产品被识别和分析。这个分析发现:

- ◆ 在问题解决中所就定义问题范围和方法的产品具有最高的重用价值。这些产品通常是在系统的分析、规格说明和工程概念阶段产生的。

- ◆ 产品具有最高的潜在的重用价值是 artifact。这就意味着较难自动分类和分析这些产品的 artifact。

- ◆ 在软件开发生命周期早期阶段的产品是由交易管理、工程管理和系统分析产生的。

这个分析表明,在软件开发周期的初始阶段通过管理和系统分析产生的产品有最大的重用价值。实际上,我们已经知道重用需求是需求模板的一个形式。不过,这些重用只是停在为不同的客户剪切和粘贴现有的需求文档的层次上。这使项目管理者有时更容易生成新客户的特殊需求;不过,对开发者重用软件模块或者生成无 bug 软件是没有任何帮助的。

如果我们不能使用设计的元素和需求的软件模块,需求重用的价值将会减小。最理想的情况,当项目管理部门重用一系列需求,执行这些需求的软件模块能在最终执行中被重用,这引入了需求模块的概念。毕竟,很难使一个简单的需求语句和任何软件模块发生联系。典型的,一个软件模块由一系列的相关联的需求组成,它定义了特定的功能或者软件模块的特征。所以,我们可以说需求模块定义了一个特征,也就是说,设计可重用的需求和软件构件就是生成基于特征的软件模块。

重用的代价

重用的代价是昂贵的。通常情况下,开发一个可重用的构件需要比开发不可重用的构件多出三到四倍的资源。一个对基于工业框架的构件体系结构实现的研究表明重用的代价是很大的。很明显,重要和复杂的开发、维护和设计过程是一个组织所要求的,是为了认识到构件重用的好处。从维护的观点出发,重用软件会产生维护的灾难。当从 COTS 构件厂商得到一个软件补丁,如果系统运行完好,是否会使用这个补丁?怎样估计所使用补丁的稳定性?理想的,对系统的任何修改必须由质量论证系统(QA)来完成,只有这样,修改才能进行回归测试和模拟系统的使用来观察修改的效果。如果 COTS 构件存在 bug 并引起系统崩溃,这不是一个通常的过程来缩小 COTS 构件的 bug 和使厂商确信他们的产品中存在 bug。

如果在工作室里开发一个可重用的构件,较早地进行维护和较早地进行 bug 跟踪将会有好处。不过,如果一个构件是成功的,它将会有很多的客户。随着时间的消逝,所有的客户将会对构件提出要求。如果所有的客户都有新的功能需求,那么这个构件是不是要发展并修改它的需求来满足客户的需要?如果是这样的,怎样才能确保为一个客户所做的修改不会对其他的用户产生负面的影响?怎样保护构件的特征?

设计可重用构件

一个为预防 bug 而重用的途径是生成简单的构件，并保护产生“做每件事”的构件。当一个构件很简单，就使使用者容易理解构件的使用和功能，也容易使构件的开发者和维护者设计代码、编写安全代码和理解代码是如何工作的。当构件很简单，就可以达到无 bug 的构件。

我们要阐明另一个工业领域的类似观点，在这个领域中使用构件是很普遍的。在电力工业，电力工程师使用电阻来调整系统中的电压和电流。在我们看到的很多电路中，很多被使用的电阻只提供一个层次的阻值。一个定值 50Ω 的电阻在特定的偏差范围内提供 50Ω 的电阻。在这样固定的阻值下，就无法调节阻值。为了得到不同的电阻值，电力工程师有一个他们可以使用的固定阻值的电阻库，这些固定阻值的电阻有不同的离散的电阻值。当电力工程师需要使用的阻值在他们可利用的电阻阻值之间时，他们可以通过电阻的串、并联得到所需要的阻值。

也存在可以通过调节器来改变阻值的可调电阻。不过，你会发现在用户期望调节阻值的电路中使用的可调电阻，并不是使用在所有的电路中。它们不被使用有以下几个原因：

- ◆ 固定电阻比较便宜。
- ◆ 可调电阻有较多的移动部分和较多的故障。
- ◆ 在不需要改变电阻的部分电路中使用可调电阻会因为环境的改变而引入错误（例如，当含有这样电路的产品掉到地板上）。
- ◆ 不使用可调电阻会减小用户的错误。如果用户不能调节阻值，他们就不会错误地调节它。

使可重用构件简单化

在我们看来，一个可重用软件构件就像一个固定阻值的电阻。它只做一件事并把这件事做好。如果需要其他功能，就生成另一个构件。一个可重用构件应当简单并且要和固定电阻一样有相同的特征。它们必须能被便宜地构建起来，并且不复杂，这就能保证有较少的 bug。目前，软件工程师宁愿构件像可调电阻一样，想让构件有尽可能多的功能。我们会不正确地认为如果构件能完成每一件事，就更能被重用。事实上，我们可能改变可调电阻，使它成为开关交换电容的功能。当然，这样的构件有较大的重用功能，更多的软件可以重用它，能被使用在更多的地方。但是，这些功能的实现是以设计者和使用者较高的复杂度为代价的，如同改变构件的功能需求的代价一样。

如果一个可重用软件构件只有少数的功能和特征，就容易被确定，这也使需求简单、清晰。一个简单的需求因为它的简单性鼓励了需求的重用。当需求能不用修改就被重用，从需求派生出的设计和实现也能够不被修改地重用。构件描述的简单性使它容易被存储、存档和查找。简单性也使它容易被其他人理解。如果我们不喜欢写构件的方法，那么把它删去并重写一个新的构件的代价也不大。

有些人认为为基本的数据结构写一个简单的构件是容易的，例如树或链表。但是，交易

层次上的构件就变得复杂。确实，交易的逻辑是复杂的，但我们也使交易层上的构件变得复杂，因为我们试图在模块或系统中开发解决所有交易问题的软件模块。这是没有必要的，交易逻辑通常能被分成一系列的工作流。使用这个办法，一个简单的构件就能执行一个工作流。一个构件集合就能执行一个工作流的集合，进而解决了整个交易问题。

我们提倡简单的构件，并不是意味着构件的功能或特征也是简单的。这里的简单是指构件容易被使用，容易被理解，并有一个简单的接口。一个构件的功能可以复杂到解决语音识别问题，但如果我们能仅通过向构件输入 WAV 文件就可以得到代表语音的文本文件，我们就能说这个构件是简单的。另一方面，如果构件要求我们安装配置文件，为了初始化和使用构件，需要 15 个参数，并且需要调用一个特定的 API 函数序列，那可以说这个构件是很复杂的。

简单性检查可以考虑以下基本准则：用一个构件产生一个过程或程序，把这个程序给一个可以信任的人。当这个人成功地使用你的构件，数一数使用程序的步骤。如果只用一步，就说明这个构件是简单的。

鼓励重用

考虑对能灵活完成每件事构件的重用和用一系列简单构件重用对比实验，它们执行相同的功能。一般灵活的构件的文档有 50 页长，相反，每个独立的简单构件的文档只有几页长。更容易理解哪一个？你是否愿意重用你没有完成理解的构件呢？

如果没有人愿意使用构件，那么就没有必要生成可重用构件。为了达到重用的目的，团队成员必须知道可重用构件的存在，他们也必须能发现适当的构件并能检索到它们。此外，团队成员必须相信这些可重用构件的质量，只有这样才能确信使用它们。

在大的组织中取得重用是一个在技术上使人畏惧的任务。所以分类、存储和检索可重用构件而生成易处理的有效过程是有价值的。一个对存储和检索技术的调查可以在“A Survey of Software Reuse Libraries”一书中找到。不过，在一个中小规模的开发群体中，鼓励重用的最好方法是使团队的成员认识到这些软件构件和这些软件构件的所有权。下面列出一些中小规模开发群体使用可重用构件的思想：

- ◆ 创建一个构件仓库。一个构件仓库可以建立在标准的配置管理库中，也可以建立在独立的配置管理环境中。

- ◆ 发布新的可重用构件。经常，我们能看到当一个开发者创建一个新的函数时，这个新的文件很快就在配置管理中登记，没有其他人认识到这些增加和修改。为了重用这些构件，人们必须认识到新的函数。即使其他开发者不需要这些新功能，他们至少要知道这些新函数的存在，以后可能要调用一个特定的构件。有很多发布新构件或新特征的方法。可以使用电子邮件或新闻组，这是一个最容易、最有效的方法。

- ◆ 建立组的所有权。如果人们知道已经开发可重用软件，它们就会考虑重用。人们更喜欢重用他们熟悉的构件。怎样建立组的所有权？要求人们加入构件库，也要求他们保持部分的构件库。在雇佣一个新成员时，这是一个好的做法。在我以前的工作中，所有的新成员

都要花费一些时间在软件模块上，或者是添加新构件，或者是调试 bug。这个过程适合基于代码的共享，但也适合组的所有权。一旦开发者习惯在构件库中查找，他们可能会得到更多。

重用的风险

重用并不能完全预防 bug。在第 2 章中，我们所研究的一些著名的 bug 就是重用软件模块直接引起的。Therac-25 重用了多 bug 的软件模块，Ariane 5 重用不适合新系统的软件。要记住的一个重要的观点是：当你重用一個软件模块时，你不仅仅重用了这个模块提供的特征和功能，也重用了这个模块所包含的错误和 bug。

重用代码最安全的方法，是重用需求模块时要同时重用设计和实现模块。为了重用代码的安全性，使用要重用构件的文档是很重要的。不过，很多的机构很少有这种基础机构和文档。“现实环境的文档”一节将提供一些可选择文档技术的思想，它将帮助你知悉一些知识。但也要求可读代码，有一个清晰简单的设计和实现使理解软件任务尽可能的简单也是很重要的。

12.2.5 减小复杂度和管理复杂度

我们知道复杂度是引起系统 bug 和错误的主要原因。当一个系统很复杂，我们会失去关键的足迹，我们无法考虑路径，我们对相互联系的变量毫无办法。减小复杂度就是减小 bug 和错误，但是要解决一个稍微困难的问题时就会遇到无法满足用户需求的危险。更好地管理复杂度，但这并不总是可行的。

使用过程

在电影 Apollo 13 中，我们看到 NASA 工程师灵活战胜危险的一系列片断。安全地发射载人宇宙飞船不是一个简单的任务，有儿百万个活动的部件和几十亿事物都可能出错。这是一个复杂的问题，NASA 用过程来管理复杂度。有一个特定的过程决定飞船是否要发射；有一些过程决定任务的完成；有一些过程处理可以想到的错误。对于这些复杂度，在一个太空设备管理中沒有足够的时间来解决。

我们被 NASA 工程师解决 Apollo 13 飞船问题的办法所打动，并不是因为他们在宇宙飞船上仅使用一个元素来实现工作系统，而是因为他们为宇航员创造了一个过程，这样宇航员能在 10,000 英尺之外建立他们自己的空气过滤器。NASA 工程师生成一个复杂的问题和复杂解决问题的办法。如果 NASA 工程师是软件工程师，他们就会给 Apollo 13 宇航员一系列项目（对象）的描述和这些项目的使用方法（接口），就假定宇航员已经知道怎样把所有的事项放在一起。我们不要担心在软件工业中有这么多的问题。

过程和方法把复杂度变成可管理的步骤，也使我们减小相互联系的变量的数目，因为其他开发者已经知道这些变量的相互作用。他们已经把这些依赖性提取为我们可以遵循的一系列步骤。虽然理想的步骤是一个组织所要关心的，但这对执行个人的过程是有好处的。The Software Engineering Institute 在 1995 年引入了个人软件过程。

可以选择哪些过程和方法？坦白地说，我们不用考虑它的真正问题。重要的是要有过程

和方法，并能满足需要。每个工作有一个过程，这个过程可能是非正式的，可能没有被记下，但人们做一件事确实要有一个方法。即使是在 CMM1 的环境中，第一步可能也要列出这些非正式的步骤，接着按这些步骤执行。

不过，过程和程序都有它们自己的缺点。其中一个缺点是创造性受限制。我们大家都听到那些关于人们陷入巨大官僚团体过程的空洞故事。这样，有了方法，我们就会产生条件反射。在遇到问题时，我们可能只选择一个过程来使用它，而没有完全理解我们所遇到的问题。但作为一个工业，软件工程师从混乱无序的环境偏离太远，以至不能立刻被过程所伤害。

不要使问题太复杂

我们已经知道在软件工程中很多的复杂性是自己造成的。我们倾向使用简洁性的技术来提供一般的解决方法，在这个过程中，我们使问题和解决方法变得很复杂。

为什么必须解决一般的问题？为什么必须使用最简洁的方法？通常，在提出这些问题时，我们所给的原因是软件工程神圣的东西：可扩充性和可维护性。但我们是否理解问题域足够好以至能产生一个普遍的解决方法能经受得起时间的测试。如果不能，我们将要对所有的普遍解决方法作修改，我们将不得不对现有的结构提出新的需求。有时这是容易做到的，但大多数的情况是很难的，没有提出新的抽象并去掉现有的大多数解决方法几乎是不可能的。所以我们通常删掉一些代码。我们的特别解决方法是没用的，而普遍的解决方法除了在少数一两个地方不行外，在其他地方是自行的。结果，在短时间运行中很难利用这个方法，因为很难得到好的普遍解决方法。同样，我们在长时间的运行中也很难利用这个方法，因为我们删除了代码并使解决方法难以被理解。那么，什么时候才能最后听到复杂的系统能容易地被维护？

软件工业是以持续的改变为标志的：修改是技术，修改才有市场，修改是客户的要求。这就意味了我们的问题和解决方法也要经常修改。创建一个稳定的、普遍的和特殊的解决方法在短期项目和长期项目中并不具有最高的性能价格比。

正确的分块方法

软件开发和调试就是解决问题。在调试时，我们的工作观察故障现象，理解系统的当前模块并根据故障现象假设问题原因。在开发软件时，我们试图阐明一个模块，这个模块解决了我们试图解决的问题，这就包含了对影响问题变量的理解。从人们解决问题过程的研究中，我们可以学到很多知识，并且可以把它们利用到 bug 预防科学中。

在认识心理学中，假定我们通过分块解决复杂的问题。事实上，我们通过把复杂问题分成一块块变成更容易管理的部分。在解决软件工程问题时，我们也采用分块的方法。在面向对象方法中，问题被分解成一个个对象和对象间的接口。在大学里，创建一个学生注册系统，我们创建了类，例如 Student 类、Course 类和 Schedule 类。这三个类之间的相互作用解决了最初的问题。

不过，面向对象的分块方法对分解一个复杂的问题可能不是必须的正确机制。一个问题之所以复杂是因为它有许多相互依赖的变量。图 12.1 表明了它的复杂性，用图的形式给出

了复杂性的表示法。节点代表变量，节点之间的边表示相关变量的相互关系。

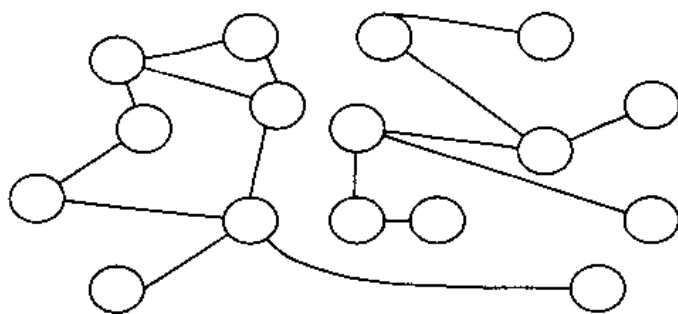


图 12.1 问题复杂性的图示

在这个图中，16个节点代表问题的16个变量，节点间的连线表示两个变量之间的关系，也就是，改变一个变量的值将影响到和它相连接的变量的值。在我们分块时，可以直观地把几个变量分在一起，如图12.2所示。

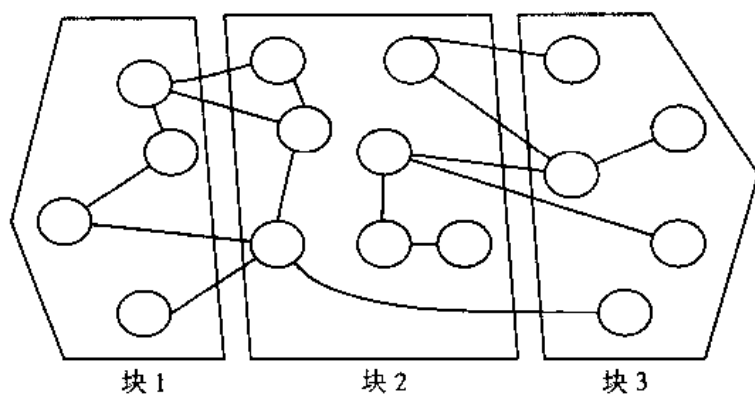


图 12.2 随意分块的结果

在图12.2中，黑线框代表了变量随意分块的结果。这个问题比较容易，因为我们首先可以用小块包含变量，接着考虑每个块之间的相互关系。在块1和块2之间，由四条线连接着，表示块1和块2有四种不同的相互作用。同样，块2和块3之间有五种不同的作用方法。但如果再分割这些变量，可能就会根据对这些变量相互作用的理解，和减少块间的相互作用来分解变量。需要处理块间相互作用的变量数目是12，超过原来相互作用变量数目的一半。

图12.3表明分块的结果，它是根据我们对相互作用变量的理解来划分变量的。仍然是3个块。每个块的大小不同，但块间的相互作用变少了。这就意味着在解决了块1的问题之后，就只需要考虑其他块对块1的一个影响，它们是孤立的两个变量。总之，只有两个变量包含在相关块的相互作用中。

在软件工程中，可以把每个变量映射到一个需求、一个类、一个变量或一个软件模块，也可以把一个块映射到一个需求、一个类、一个变量或者一个软件模块。细节的层次和映射是完全任意的。关键是问题有很多种分法，结构类型、函数类型或面向对象，但有些分解比

其他分块方法更适合一个问题域。根据问题选择正确的分解体现了简单问题和复杂问题之间的不同点。

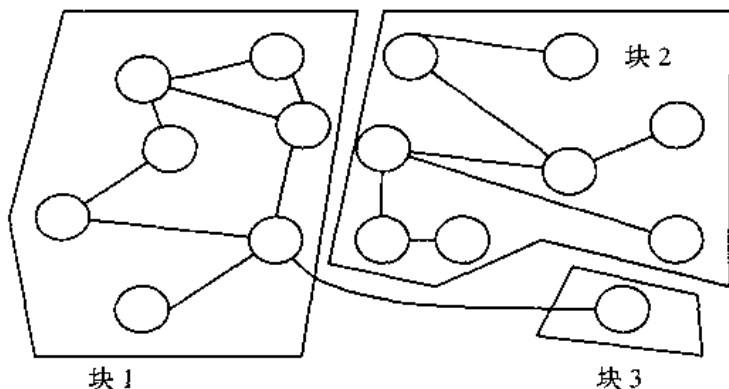


图 12.3 更好的分块方法

不能出现特征变化

“当然，为什么我想不到这个？”你可能会自言自语地说。仅仅不去做任何的变化，而不去试图理解变化的原因将会失去主张。作为一个软件工程师，不能和项目管理者一起战胜特征的改变；同样，作为一个项目管理者，在很多情况下对自己的下一执行层说不就相当于政治上的自杀。

限制特征变化的一个方法是取得正确的需求。取得正确的需求意味着我们必须理解问题域并且要避免需求的错误。不过，在很多时候，我们使客户有太多的想象力和概念。因为客户没有技术上的倾向，很难为他们具体化他们的需求。既然这样，让团队的技术人员和顾客一起来挖掘需求是很有好处的。一些对问题理解的讨论和提示在“需求的 bug 预防”中将会提到。另一个提示是检查需求文档，我们在这之前已经讨论过。

不过，即使需求是正确的，特征变化也会发生。作为专业技术人员，我们通常用技术的手段来减小特征变化的影响。我们通常使我们的解决方法具有一般性和可扩展性从而尽可能调整特征变化。但确实需要技巧，特征通常从不同的方向漂移，使得一般性解决方法失效。有很多减小和减轻特征变化带来影响的方法。看以下几个例子：

- ◆ 技巧——如果以前解决过一个特定的问题，那么对客户可能要干什么就会有一个相当好的思想，但没有明确地表达。你该怎么办呢？很好，因为只有少数的几个初始问题出现，很多业务和技术的问题已经被一些人解决了。这个技巧就是发现一个先前解决类似问题的人，让他们确定潜在的特征变化方向，接着设计出防止改变的措施。

- ◆ 市场分析——市场分析不是技术和管理人员想要做的事。但是，通过观察同伴所做的事和客户的要求，就可以知道上级管理部门将要做什么。这就意味这要看交易记录，观察市场，和客户交流，向同事学习。

- ◆ 区分优先级——一个相关的建议是简单为特征和他们的调度安排优先级。如果能为现有的特征建立调度，也许就可以发现附加的特征怎样对调度产生负面的影响，那么也就可

以对这些特征进行优先级管理。

◆ 提供可替代的解决方法和特征——有时，所要求的特征不可能得到时间的分配。不过，这个要求的特征可以被当成交易问题来解决。如果理解这个最初的交易问题，就可以提出一个替代解决方法的建议，这可能会更可行。

不要摆手

当我们看到一些人在表达一个解决方法摆手时，我们知道这些人没有把这个方法想通。这就告诉我们一部分隐藏的问题复杂性被去掉了。如果解决方法是清楚的，就能简单清晰地交流，也就没必要摆手了。

摆手的主要问题是原来摆手的问题又出现了。因此，问题的困难部分被拖延，直到没有时间去想它，接着就作出了决定，而对复杂性没有做出应有的决定。谁赢了？当然是 bug 获胜。

12.2.6 现实世界的文档

每一本软件开发的书都说到了文档的重要性。每个软件工业的人都同意文档的重要性，但没有人真正这样做。为什么？我们相信这个原因不仅是因为文档单调乏味，还因为认为文档是没必要的努力。覆盖墙壁的 100 张用 9 号字体做成的类图，是否对任何人都有帮助，除了实现和理解系统的初始设计人员之外？对系统 API 的 500 页描述，是否对你在系统中使用 API 函数有帮助？对这两个问题的回答都是否定的。另外，对维护任务的研究表明维护人员在他们不熟悉软件时，他们只花费一点时间来看文档。一旦熟悉了系统，他们很少重新看文档，他们更宁愿读代码。

文档通过让我们理解系统，来帮助我们预防 bug。文档的目的不是生成一本落满灰尘的书，也不是砍掉一大片的树来造纸。文档的目的是从问题、解决方法和实现角度来更好地理解系统。哪些需要做成文档？任何能帮助我们理解系统的方面，都应当做成文档。关键是鼓励理解。除了增加文档，也要增加使系统更加容易理解的其他方法。

分离实现者和设计者的任务

一个减少文档的办法是把设计者和开发者的任务分为两个不同的人来实现。如果设计者不允许写代码，那么他们就必须把他们的设计和开发者进行交流，这个交流通常是靠一些设计文档的形式进行。

这个方法特别适应初级的开发者，因为设计者必须更清楚表达他们的设计，设计者必须生成详细的文档和过程，使开发者能理解和遵循。高级的开发者能理解和掌握设计者的想法，能填补设计的缺陷而不管简陋的文档，结果使文档很少有用。

当我们设计一个面向对象的应用系统时，我们规定了类和每个类的方法，我们规定了类之间的接口，我们定义了每个方法的前后条件，我们依靠脑海中的说明书来看代码。惟一未完成的是进行编码、实现设计。就像我们所关心的，编写代码与画类图、顺序图是一样的。对于一个高级的开发者，写设计文档和编写代码没有任何的差别。惟一的差别是用来生成产

品的介质和工具不同。但对于初级的开发者来说,设计者需要考虑设计的复杂度。如果设计太复杂,那么初级的开发者实现起来就有困难,就可能不能按时完成工程。

我们已经有一些机会用多种方法执行这个技术。在任何情况下,我们生成了详细的文档,包括需求文档、结构文档、设计文档和测试计划。我们编写文档并观察使用文档的其他人,很快就知道哪些需要被记录,哪些不要记录,就像哪些有帮助哪些没有帮助一样。在下面几节中将要谈到另一些教训。

提供概述

至少,文档应当使它的读者对产品和问题的解决方法有一个概述。作为一个专业的软件开发者,我们总是认为我们的开发人员能读懂代码,并且知道代码的作用。但在软件系统中,源代码不能描述代码的上下文关系。读一个 100 行的软件模块,我们还可以知道这个模块怎样装配到程序中和系统模块的基本结构。如果整个系统的大小只有几千行代码,我们能够读它。但当系统有几十万行的代码,读它就很难了。更难的是,源代码不能告诉我们怎样解决软件设计的问题。如果我们不懂软件问题的解决办法,就很难知道软件是否正确解决了问题。

软件项目文档最重要的作用之一,是它可以给出问题和解决方法的概述。这个概述帮助新的开发者、测试者或者项目管理人员从哪里开始。需求文档的概述告诉读者问题是什么并提供一系列需求的高层分类,这个文档回答了系统的基本结构的问题,并提供了所有主要软件模块和这些模块之间的相互作用的基本描述。

利用图片

“一图胜千言”。图片文档比起纯粹的文本或文档来说,更能帮助维护人员更好地理解系统。图片能减少书写文字的复杂度,并通过代表观念的形状和空间关系帮助理解文档的观念。我们发现在高层的文档中,代表问题和解决方法的图形是帮助理解的无价工具。

什么情况下才需要在系统中使用图形文档?如果图形文档能帮助理解高层的概念,并且你和读者能很好使用这个工具。不过,如果图形文档是设计说明书的一部分,那么格式化语言,例如统一建模语言(Unified Modeling Language, UML)或者实体关系图就更适合于这个文档。

FAQ 格式文档

FAQ(Frequently Asked Questions, 常问问题)开始于 20 世纪 80 年代中期的网络新闻组。FAQ 的目的是记录下一系列标准问题和答案去重复回答新闻组中每个新成员提出的相同的旧问题。我们发现 FAQ 成为所提话题的简洁概要。最近,我们发现越来越多的 FAQ 格式文档伴随产品 API 出现,并且也发现这些 FAQ 对提供软件产品的高层概述是很有帮助的。

FAQ 的问题和答案格式让读者更容易发现他们要查找的问题。在阅读文档时,我们通常会产生很多问题。“程序要干什么?”“我怎样处理 XYZ?”“这个对象和另一个对象有什么关系?”FAQ 的问题特别适合这个途径。FAQ 的答案可以很详细,也可以是另一个文档的索引。

问题和答案格式也使开发者更容易写文档。给一张空白纸，这会使开发者不知道从哪里开始，写到多具体才合适，写些什么，怎样把内容正确介绍给读者。但是给出一个问题，那么开发者就特别容易回答这个问题，因为这看起来不像在写文档。你需要的大部分文档可以完全从开发者的问题中得到。他们对问题的回答或多或少是你所需要的文档。

FAQ 中的问题是从任何对软件或系统有疑问的人中得到的。问题和答案可以用来建立知识基础。

工程历史

你曾经多少次在表面上无法解释的代码上犹豫过？这些代码显得完全没用，但是当你删除这些代码时，系统又不能正确执行。另外，在每个系统中，总是有一些无法解释的、看起来很低能的奇怪设计。但你应当知道初始开发者是有经验和专业的。他们这样设计系统是否有好处？

了解软件的发展过程对维护者是有好处的，因为他们不会对执行和解决 bug 进行相反的修改。工程的历史对设计者和开发者理解问题也是同样有效的。当系统被重新设计时，他们就不必要再跟踪相同的路径。工程历史同样可以使项目管理者通过观察修改、问题和已经出现的讨论来从质量上评估软件开发过程。

以下列出一些怎样捕获工程历史信息的建议：

- ◆ 会议记录——会议上出现的问题、答案以及讨论对软件的设计具有很大的价值。捕获这些讨论能帮助我们理解软件的发展历史。

- ◆ 局部私有新闻组——建立一些内部网络新闻组进行项目讨论。这些新闻组的记录有助于理解将来项目的开发。

- ◆ 保存相互交换的电子邮件——在电子邮箱中建立具体项目文件夹，并保存所收到的和项目相关的每个邮件。同样，保存每个发出去的和项目相关的每个邮件。使这些旧的邮件顺手可得。当某些人想知道一些事情的原因时，你就可以提供一个导致最后参考执行的决定过程的概要，用它可以多次回答被多次提出的相同问题，同样也可以再次看到所作的错误决定。这些电子邮件可以作为学习的经验，也可以让你想起过去的错误。

- ◆ 版本控制的历史日志——包含在版本控制或配置管理系统中的注释和历史日志是有价值的信息源。这些日志包含了软件的所有修改，并且能被回顾。

- ◆ 代码的注释——当对代码进行修改时，直接在代码中做注释，说明为什么要进行修改，而不只是说明做了什么修改是有用的。我们可以用任何的配置管理工具读所有的代码，就很容易发现做了什么修改，但较难发现为什么代码被修改。

包含在代码中的文档

因为编程语言是我们所用的最精确的语言，在文档和交流中充当着杠杆的作用。而且，很多实验研究表明作为一个开发者，我们要读很多代码，这样才能完全理解我们所看到的软件。这个见识使我们的代码尽可能的易于理解。

包含在代码中的文档不仅仅有明确的函数和变量名，也有系统和接口的设计。我们将在

“设计中的 bug 预防”一节中讨论这个问题。

测试程序

一个最好的学习 API 的方法是使用指南。指南中有很多例子，使用 API 的例子能帮助 API 的初学者理解如何使用 API。

创建一个 API 时，必须对它进行测试。这通常需要生成测试程序，这些测试程序帮助我们检查软件的功能和特征。

既然这样，就很自然要保存测试程序作为设计文档的一部分。一方面，文档也需要例子；另一方面，我们需要测试软件的程序来检查它的功能。为什么不用这些测试程序作为文档的例子呢？这些测试程序能作为回归测试程序组的一部分。这样，如果 API 的说明书改变，那么测试程序也需要改变。当测试程序改变，使用在文档中的例子也就自动跟着改变。这就是跟着文档更新数据的好办法。

在内部网上，一个基于 HTML 的组织文档系统是适当的代码例子。HTML 页面提供了代码例子的链接。当链接被访问，最新的测试程序就显示出来。不管例子怎样保留，保留在什么地方，仍然可以被所有的开发者访问。

12.2.7 内部基础结构支持

软件 bug 就像蟑螂一样。不论我们怎样测试软件，总是会有预想不到的错误出现。在一个成品软件中，bug 可能在任何时候出现。当它们出现时，我们就可以尽可能发现这些 bug。所以，我们需要能帮助我们观察原因和后果的工具。基础结构使我们在需要解决 bug 之前发现和重现 bug。

瞬态图

在很多操作系统中，通过把堆栈和存储器的内容转储到能被监视的文件中来生成软件的状态。在 Windows 操作系统中，是 Watson 文件；在 Unix 操作系统中，是内核文件（或者是内核转储）。这些文件给出了错误发生时系统的瞬态图。在 Watson 转储文件中，可以看到错误发生的函数。在内核文件中，也有变量的内容和调用的栈，也就是导致系统崩溃的函数列表。

瞬态图的作用是有限的。用瞬态图作为发现 bug 和帮助调试的一个缺点是在系统崩溃时，通常只能得到惟一的信息，但是 bug 不是引起系统崩溃的惟一原因。另一个缺点是如果 bug 不是由调用栈引起的，这就很难解决 bug。例如，如果 bug 是因为在代码分支处预取了 100 条指令引起的存储器冲突引起的，可能会看到错误的变量内容，但不知道为什么出现内容冲突。不知道为什么发生存储器冲突，就无法解决 bug。

典型的，当有一个系统的瞬态图，我们回到错误点发现引起问题的原因。但这个问题没有出现在瞬态图中，我们就只能假设，然后来证明我们的假设。

跟踪日志文件

使用系统瞬态图最困难的地方是因为它是一个静态的技术，它只能在特定点上及时出现

故障现象。不过，计算机系统需要动态的显示。程序从一个状态到下一个状态，导致系统故障的根本原因的线索不会包含在出现故障时的系统状态中。帮助我们观察程序的执行的一个办法是利用系统在不同时刻的瞬态图，也可依靠创建日志文件来完成。这个跟踪日志文件可以简单到只需要生成一个程序进入或退出函数的状态，或者是一个主程序功能块；也可以复杂到记录一些主要变量的内容。

跟踪日志文件的主要优点是可以帮助我们观察到程序在出现错误前的一些动态行为，我们可以知道程序在出现问题前的动作。很多调试的书能告诉我们在调试期间使用跟踪信息，但这些书也告诉我们在产品环境下注意跟踪信息。应用它的主要原因通常要考虑到性能，登录跟踪信息不会降低性能。我们不能否定这是正确的，但在很多环境下，性能会降低，这是因为记录并不能够根据将要发生什么而自动调整。同样，在成品环境中使用日志文件通常很难再生成。另外，通过注释代码，就意味着如果我们要在成品环境中打开记录，我们必须重新编译程序，并且可能还会因为配置管理的修改而引入问题。

理论上，使用跟踪记录有以下一些特征：

- ◆ 记录器在运行时必须能被控制，这样就可以在运行时打开或关闭它。
- ◆ 记录器应当能记录错误的信息，并捕获到能帮助调试的数据。
- ◆ 记录器应当能向文件写入错误信息和调试信息。
- ◆ 记录器应当能产生不同细节层次的信息。当事情变得平稳，只需要高层的信息；当事情出错时，我们就要系统所做的每一件事。

在跟踪日志文件中需要哪些信息？

- ◆ 数据和时间——它们给出了程序执行的历史。在分布式计算机系统中，它们也可以帮助我们知道一台机器对另一台机器的事件。
- ◆ 线程/进程标识符——在多线程系统中，可以用来跟踪不同线程的信息。
- ◆ 定位日志登录的地方——可以是一个函数名、一个文件或者是一行数字。
- ◆ 调试信息——可以是任何你认为对调试有帮助的信息。

建立跟踪日志包有很多好处。一个好的跟踪日志包可以让机构中的其他成员使用它，这就意味着可以一致使用跟踪日志设备。机构中的每个人都应当知道怎样读和解释记录中的信息。这样，单个程序员就不再会使用他们自己的风格来生成记录。

使用重现 bug 的环境

当系统的配置和最终产品配置不同时，你怎样才能证实正确解决了 bug？你怎样重现 bug？在很多情况下，可能没有必要重现 bug，因为它通常和系统配置没有关系。不过，你隔离出 bug 的能力将和你重现生成软件产品的能力有关系。

大约两年前，我在一个网络公司工作。这个公司是应用服务提供商（Application Service Provider, ASP）并为客户提供服务。它是一系列的 Web 服务器、数据库服务和公司服务的主机。我和我同事的工作是为客户创建一个项目。我们已经通过了两次测试，系统能承受住客户的所有登录，并且我们也能观测到。系统运行了 15 分钟就崩溃了，重启之后又崩溃了。

这样一直发生了将近 5 个小时。最后我们决定关闭系统进行调查。

和所有问题一样，我们无法在开发环境中重现问题。问题好像出在登录上，但是我们不能使系统让超过系统最大承受能力的人数登录。经过查找问题，最后发现问题可能出在操作系统的补丁上，这个补丁是最近才应用在系统中的，对软件有预想不到的副作用。为了证明这个假设，我们不得不从产品中去掉这个机制，这样才能进行正确的实验。我们不能改变开发系统因为它有很多复杂的、不同的系统和软件。

理论上，应当有一个成品软件在添加补丁前的副本。这个环境就是重现 bug 环境。重现 bug 环境的目的是建立一个机器，在这个类似产品环境的机器中可以进行实验，来重现 bug、隔离 bug、并证实 bug 的解决。很明显，在成品环境中进行这个实验是很危险的。但开发环境和 QA 环境可能还有成品环境中的附加配置和软件，使实验的结果无效。

理论上，一个重现 bug 的环境有以下一些能力：

- ◆ 可以重现输入。这就可以让我们得到出现错误时系统的输入，然后把它们作为重现系统的输入来重现 bug。
- ◆ 和成品软件系统一样的系统配置和软件版本。
- ◆ 一次修改一个元素的能力和重新修改的能力。

12.3 需求中的 bug 预防

必须尽可能早地捕获错误和 bug，因为解决 bug 的代价和时间是成指数关系的。在测试阶段解决需求阶段引入的 bug 的代价是在需求阶段解决系统 bug 代价的 500 倍，这个观点是众所周知的，并已经被很多研究证实了。对这方面研究最多的是 Barry Boehm 的《Software Engineering Economics》一书。

什么是需求，它对 bug 的引入有什么影响？对于需求有很多定义，但它们都是简单从软件的行为开始的。

需求文档是软件产品的客户和开发团体之间的第一个文档。它帮助客户同意软件的正确行为。如果一个动作和需求阶段的要求不同，那么这个行为就是一个 bug，因为客户和开发团体已经同意了所期待的软件行为。这个 bug 更可能和设计与实现时的 bug 相联系。另一方面，如果软件产品的一个行为既没有被需求肯定也没有被否定，那么也可能是一个需求中的 bug，因为这个需求是不完全的。如果软件产品的行为和需求一样，但是这个行为不是客户所要求的，那么需求也有可能不明确或不正确。

一个好的需求的关键是可测试的，在很多情况下，一个好的需求文档也是一个好的测试计划。一个不可测试的需求可能是对用来创建需求的自然语言的错误判断；一个不可测试的需求是客户和开发团队之间产生分歧的原因，因为任何一方都不同意这个行为是 bug。

两个关于需求的 bug 预防概念是：理解要解决的问题，然后再把这个问题放在正确的位置。

12.3.1 理解问题

产生 bug 的一个主要原因是缺乏对问题的理解。一个典型的软件开发过程包括需求收集阶段、设计阶段、开发阶段和测试阶段。那么，开发者和测试者什么时候去理解问题呢？显然，开发者和测试者不需要去理解问题。他们只需要根据需求分析的要求进行。如果他们每个人都能按照需求文档进行编码和测试，那么就不会出现 bug。不过，好的需求并不多见。没有一个好的需求，软件开发者在他们开发时就不得不去发现需求的不足，及时在开发过程中修改软件和需求。

从概念上说，一个好的需求是完全的、正确的和没有歧义的。既然如此，它就包含了开发者和测试者需要知道的所有问题。既然一个理论上完美的需求是存在的，我们认为需求文档不是开发者查找问题的地方。

如果项目的设计者和体系结构设计者不理解问题，这使他们很难进行正确的抽象。需求文档定义了系统该做什么，但没有必要包括系统要解决什么问题。而且，很难把单条的需求语句构成一个整体。例如，考虑以下简单的需求：

```
R1.0 The vehicle shall have 4 rubber wheels.  
R1.1 The vehicle shall have an internal combustion engine.  
R1.2 The vehicle shall have a steering wheel.
```

要求我们建立什么呢？大部分人都会说是小汽车。但这个需求也可以是一个特定的动力剪草机。所以这个需求是不完整的！当然也是不完全的。小汽车完全的需求集有几千条独立的需求语句。你想你可以读完所有这些需求和简单的语句，然后就知道小汽车像什么，就知道我们怎样解决问题？

提出问题，而不是解决方法

对于我们来说，真正的问题是软件工程师不仅仅是一个程序员，他们还是体系结构设计师、设计者、开发者和问题的解决人员。如果对问题不理解就很难提出好的解决方法。同样，如果测试者不知道软件要解决的问题、软件的使用者和使用环境，也很难测试软件的真正用途。

以下的一些疑问常常发生在开发者和项目管理人员身上：

项目管理者：“你可以在两个星期之内把 XYZ 的特征添加到系统中吗？”

开发人员：“不行，这几乎是不可能的。”

项目管理者：“但你必须这样，公司的发展全靠它了。”

[开发人员没日没夜地干，最后还是没完成任务。]

交换一下角色，会出现什么问题？从开发者的角度，项目管理者的问题“你可以在两个星期之内完成 XYZ 吗？”那么开发者可能会回答“你可以在 3 分钟之内跑 1 英里吗？”项目管理者不想作出任何回答，同样对于开发者也是不可能达到的。他们陷入了僵局，没有赢的状态。不过，如果项目管理者问“我要你在 3 分钟之内达到 B 点，B 点离这里一英里。

你能做到吗？”回答可能是“没问题。”开发者走进汽车并到达了 B 点。在第一种情况下，项目管理者的问题包含了开发者解决问题的方法（必须是跑）。在第二种情况下，项目管理者的问题明确定义了问题（用 3 分钟从 A 点到达 B 点）并允许开发者使用适当的方法（可以驾驶汽车）。

说老实话，我们从来没有进行过单独依赖需求的项目。如果我们进行的每个项目总是有各种各样的会议，在会议上项目管理者总是描述概念上的系统并向设计者提出附加的需求。而且，对于问题本身，客户和项目管理人员总是有不断的讨论。

总体设计师和设计者在需求分析的早期介入

从我们的经验看来，我们已经发现对问题的理解通常是伴随着项目生命周期非正式地进行，我们也发现我们花费很多时间在会议试图理解我们要解决的问题。项目管理者 and 需求分析人员摆出需求，我们来提问。一天之后，我们返回并提问更多的问题。但除了在开发过程的需求阶段向项目管理者 and 需求分析人员提问外，为什么不让设计者在需求阶段直接向客户提问？为什么要有一个中间人？

让技术人员在开发过程的早期阶段介入的一个办法是执行不同类的检查技术，就像这一章的“检查错误”一节中描述的一样。极端编程方法学中提到的 on-site customer 技术是另一个帮助开发团体理解问题的方法。

循环方法

另一个达到问题理解的方法是执行一些循环软件开发方法，例如 Boehm's spiral。在一些循环方法中，工程团体不仅一次地重复项目的每个阶段。项目团体在第一个循环中进行了分析、设计、实现和测试。在第一个循环结束时，另一个分析、设计、实现和测试的循环又开始，使用了第一个循环的知识和结果。这个过程一直进行，直到达到要求。

循环方法有助于项目团队通过多次进行项目工作来理解问题和它的解决方法，使团队通过实验和错误来学习。不过，如果你认为循环方法不适合你，你仍然可以通过实现这个方法学习问题和它的解决方法。你可以在需求阶段建立一个原型，或者一个系统概念的证明。

原型法

理解问题的最好方法是真正地去努力解决它，一次、两次，或者尽可能多次。在软件工业中，我们总是不喜欢完全灵活的工具。用我们的工具，我们能够快速地作出修改。我们能用一个原型系统来展示我们解决问题的方法。如果方法不正确，我们就要修改它，并用另一个方法来替代。我们可以任意改变方法，直到找到一个我们喜欢的方法。但这个做法有危险，如果乱用这个方法，我们最后将得到一个不稳定的、多 bug 的软件。

原型系统有一个危险，这个危险就是管理者或者客户要使用原型，好像它是一个最后的产品。当我们用原型来解决问题，我们的目标是要很好理解问题域和潜在的解决方法，这个目标必须在管理者和客户之间很好地交流。我们不能把原型和最终产品相混淆。如果开发时间比较紧，就通常要去掉我们已经完全充分理解的模块，这些模块可以为一个成品环境设计和实现。

需求中的选项

有时,可能很难达到正确的需求,因为客户对软件的功能也不是很清楚。在这种情况下,对客户施加压力是没有作用的,因为在客户有机会使用软件之前,客户也不知道什么是正确什么是错误的。在不清楚系统该干什么前就定下最终需求的危险是需求可能改变。而没有需求的危险是客户和开发团队之间没有清楚达成一致,也就不可能产生一个客户能完全接受软件的标准。

解决这个难题的一些办法是使用循环法或原型法。如果这两个途径都不可行,另一个方法是使用多个需求“选项”来帮助定义特定的功能。使用这个方法,所有的选项都被执行,客户在查看并评估这些选项后作出一个选择。当双方都同意潜在的解决方法时,这个方法很有用,但没有一方会同意哪一个是最合适的。开发团体使用这个方法的优点是减少修改需求的危险,团队也可以设计出多种解决问题的办法。

12.3.2 正确获得需求

生成一个好的需求是不容易的。一个好的需求是完全的、不含糊的、正确的。为了生成一个好的需求,就必须理解问题域。问题域就是问题所处的环境、人员和过程,解决方法必须把问题定义在特定环境、人员和过程的上下文中。软件可能是围绕着整个的解决方法,也可能是解决方法的一部分。软件解决方法和问题域之间的相互作用定义了软件的行为,这就可以当成需求。捕获并记录这些期望的行为通常是在软件开发周期的需求阶段完成。开发团体的一些成员,特别是项目管理者 and 需求分析人员,面见客户是为了完成需求。他们的工作是生成一个好的需求。因为创建一个好的需求比较困难,以下几节将提供一些有用的思想。

标准的提示

在很多书和论文上,有很多关于如何完成需求的提示。这些书包含了怎样收集需求、怎样写需求、怎样修改不明确的需求。我们列出一些主要的思想:

- ◆ 避免不明确、主观的、比较意义上的和含糊不清的术语。这些术语通常是定性的,例如:好,好快?一个健壮的系统是否是 24/7?什么是用户友好系统?也有可能是定量术语。这样,就要面见客户来决定他们的想法并尽可能逐字记录下来。

- ◆ 使需求尽可能简单易懂。避免使用“和”或“或”去结合需求,使每个需求尽可能是原子单位。

- ◆ 进行需求检查和回顾。需求检查和回顾可以用来尽早地解决需求错误。和客户一起进行需求回顾特别有用。

- ◆ 使用循环法和原型法帮助开发团体生成好的需求。

让测试者参加

前面,我们定义了需求是软件的可测试行为。我们也知道一个好的需求和测试计划没什么不同。既然这样,就意味了测试者特别适合协助和评估需求文档。这个建议在“检查”一小节中的“不同类人检查技术”和“生产者-消费者技术”中提到。

测试者试图发现 bug，并且测试不希望相互影响。例如，如果有一个允许用户访问日期的接口，一个优秀的测试者会输入一个合法的日期和一串非法的日期，例如 13/0/2000、-1/-1/-1 或者是 12/43/1900。测试者也需要知道系统对一个非法输入的反应。如果在需求中对非法的输入没有做详细的规定，那么测试者就应当通过错误方案使需求更完整。在这种情况下，测试者应当充当一个检查者的角色，依靠他们的经验和知识使需求更具体。

同时进行测试计划

与收集需求同时进行测试计划可以避免含糊，因为测试者使用含糊的需求不能创建测试计划。如果需求一开始就是：

“系统必须很快。”

一个优秀的测试人员就会摇摇手说：“我不知道怎样测试这个系统！”“很快”是一个主观的术语，是无法进行测试的。这个需求必须从下面这句话开始：

“在系统平均每秒有 20 个请求时，系统对每个请求要在 500 毫秒内做出反应。”

这个需求是可测试的，测试者可以在这个需求上进行工作。利用这个需求，可以进行多次实验并测出平均的响应时间。如果在每秒 20 个请求下，平均响应时间大于 500 微秒，那么测试者就可以说系统不符合需求。

事实上，一个优秀的测试人员可以立刻知道不能测试的需求，并把它们给需求分析人员，让分析人员检查后重新定义。

12.4 设计中的 bug 预防

软件系统的设计就是对问题的解决，这个解决方法必须满足需求。如果不是这样，设计就有 bug，也就是我们所说的设计不完全或不正确。这种设计阶段的 bug，有时通常是出自于较早设计缺陷。可测试特征，例如 GUI 的相互作用和需求功能特征，是相同的例子。不过，和性能需求相关的设计 bug 很难在早期发现。有时直到直接执行来测试它才可能知道确实是一个设计的 bug。在前面一节，已经提到了一般的技術，例如原型法和检查法，有助于这种类型设计 bug 的调试。

不过，当我们谈论软件工业中不好的设计时，我们通常没有谈论不满足需求的设计。另外，我们把不好的设计等同于不灵活的设计。这种设计很难被重用，很难被修改，也很难被扩展。面向对象的设计原则是就是为了解决这些问题。设计模式是对问题的另一个潜在的解决方法。另外，一个设计可能增加生成 bug 的机会。这些类型的设计问题很少被讨论，我们在这一节将重点讨论它。

12.4.1 不同的设计产生不同的 bug 类型

典型的，在设计会议中，关于设计优点的讨论围绕着可扩展性、可维护性、技术简洁性和可执行性。很少人在设计会议中讨论 bug 问题。不过，的经验告诉我们不同的设计方法选

择对 bug 的类型有直接的影响。

在第 4 章中,我们已经引入了设计替换法,包括使用回调。你也可能从自己的经验中得到一些其他设计元素和例子。在这一节中,我们引入一些设计问题的例子和普遍的 bug 类型。这一节的四个例子只是为了帮助你理解这些设计的概念。我们希望这些例子帮助你考虑你现在正确使用的設計方法存在哪些危险。通过了解不同设计方法的优缺点,就可以合理使用替换,并观察在设计中和与设计相关的实现中可能存在的 bug。

排序的风险

在第 101 期《Computer Science》中,讨论了不同的排序算法:冒泡排序法、插入排序法、二叉树排序法和快速排序法。你对它进行命名,我们来介绍它并实现它。不过,从大学毕业后,我就忘了怎样实现排序算法。因为 quicksort()在标准的 C 语言库中可以得到,并且 C 语言在那时是流行的语言,所以我记得 quicksort()函数。

排序是一个普遍的问题,有很多著名的解决方法,它有什么问题呢?

当考虑使用排序算法,我们很少考虑以下一些条件:要求排序的元素的个数超过硬件存储器的能力,或者对排序算法提出不切实际的性能要求。机器能处理一百万条记录的排序吗?如果系统有确定的性能要求,能在性能要求之内完成一万条记录的排序吗?是否需要设计不同的系统,使问题的规模 and 性能可以不用讨论?

在很多情况下,这是一个需求问题。但是作为一个设计者,当遇到排序或其他很好理解的算法时,我们经常不对规模和性能提出疑问,我们也会因为排序是一个已经被解决的问题,而经常不注意什么时候丢失性能。当一个客户说他有数据需要排序,我们总觉得没有问题。我们知道这样排序,所以我们不会多想这个问题。

分类的风险

另一个有风险的设计例子是分类算法的设计。因为 XML 是流行的消息格式,我们将用 XML 作为分类例子。XML 已经成为流行的设计语言,人们用它作为网络通信和其他业务的标准消息格式。在很多工业部门有很多不同的标准试图为他们解决的问题定义 XML 规格。一个这样的标准是产品分类。在这个标准中,XML 对一个产品定义了以下信息:名字、描述、图像、价格和其他一些需要的属性。

在我的一个工作中,一个公司用 XML 标准定义服务分类的信息。一天,一个客户要我们把他的产品装载在我们的数据库中。我们说没问题,接着我们把客户的数据转化成 XML 格式,这样我们就能用我们的 XML 接口装载数据。不过,在我们装载数据之后,系统就死机了,甚至服务器也崩溃。这个问题的原因是 XML 数据太长以至不能被机器安全处理。在内部,接口对这个 XML 消息进行分类和存储,并在处理前就包含在系统中。这个系统的设计从来就没有把消息的大小当成潜在的风险来考虑。

递归的风险

对于解决某一类型的问题,递归是有用的数学模型。几乎所有的参考书都用计算 Fibonacci 系列作为递归的例子。如果函数能调用自己,函数就被执行,就像下列所示:

```
1. void func_A( int some_argument ) {  
2.   if ( some_argument <= 0 ) return;  
3.   func_A( some_argument-1 );  
4. }
```

`func_A()`不具有激发性，它没有提供任何有用的计算，但它是可执行的。如果调用函数 `func_A()` 的参数是 3，那么 `func_A()` 在返回执行之前，它将会用参数 2、1、0 连续执行三次。第 2 行的条件语句是防止无限制调用自己的唯一元素。没有第 2 行语句，`func_A()` 将调用 `func_A()` 直到出现堆栈溢出错误。

什么是堆栈溢出错误？当一个函数被调用，那么函数的参数和局部变量将被置入栈，当函数退出时这些参数和局部变量也从栈中移出。堆栈存储的容量通常是有限制的，当堆栈满时，因为没有地方存储局部变量就会出现堆栈溢出错误。

递归的一个显然风险是堆栈溢出错误。当出现堆栈溢出时，程序很可能崩溃。一个导致堆栈溢出的原因是设计和实现不正确的终止条件。例如，如果在前面的例子中，第 2 行代码丢失了，`func_A()` 就不会停止，就会出现堆栈溢出。另一个导致堆栈溢出的原因是在第 3 行中不正确执行递归调用。如果调用函数 `func_A()` 的参数不是 `some_argument-1`，而是 `some_argument+1`，那么 `func_A()` 也会出现堆栈溢出错误。

即使递归正确执行，如果递归设计不正确也有可能出现堆栈溢出的危险。例如，用机器的最大有效整数值调用 `func_A()` 就很有可能引起堆栈溢出错误，因为堆栈的大小不能存储所有的变量。这是因为算法要求进一步的递归调用。

面向对象设计的风险

因为在软件工业中面向对象设计和语言比较流行，出现了很多和面向对象设计原则相关的软件错误和缺点的研究。面向对象的优点，例如继承和多态，是众所周知的。虽然继承和多态可以提供很多面向对象的优点，但研究同样表明这些概念的使用也会引起很多困难。

对面向对象设计错误倾向的研究表明类和方法的连接会引起软件模块的错误。实际上，作者认为方法的调用、重要连接是错误倾向的强大、稳定的预兆。在这个研究中，方法调用测量了一个方法包含至少一次另一个方法的出现。在研究多态的作用中，发现会增加面向对象软件错误的概率。不过，这个研究也表明多态大多出现在有大量方法和较深继承层次类中，这通常是一个比较复杂的系统。这些研究结果没有给人惊奇。绝大部分的 bug 出现软件模块的接口或者理解不清楚、含糊和有错误接口设计的类中。很难知道面向对象设计有没有用。具有动态性质的多态使它很难被理解，也使它更容易引起错误。

面向对象的设计提供了想象问题和分割问题的概念性方法，不过，在软件维护阶段出现了困难。我们将在第 14 章讨论面向对象设计对软件维护的影响。面向对象设计的主要危险是人们对替代不熟悉，把面向对象设计看成好方法。只有你正确使用它，它才能帮助你获得软件的思想。

12.4.2 设计接口

很多研究经验表明,大部分的 bug 和软件模块之间的连接有关系。简单地说,两个模块之间的连接就是模块的接口。如果一个模块通过它的接口使用另一个模块,那么这两个模块就是有连接。显然,一个模块的设计会影响到接口的设计,并会影响接口层上的 bug。同样,接口的灵活性和模糊性也会影响接口层上的 bug。

设计模块

像我们所做的大部分解决问题的的工作一样,我们要把一个软件工程的问题变成一些小问题的集合然后我们来解决这些小问题。事实上,对这些小问题的解决就是我们所谓的软件模块。在结构化程序中,一个软件模块可以认为是一系列相关的函数;在面向对象中,一个模块就是一些类的集合。

常识告诉我们,大的集成模块维护和调试起来比较困难。但是,模块太小也是一个问题。在大的集成模块中,模块的内部逻辑存在复杂性。因为大的模块不可能被分成小的任务,在模块中的函数和类要求处理整个模块要解决的无数任务。当大模块被分解成小模块时,这个复杂性就从模块内部的复杂性变成模块间的复杂性。当模块变得太小,系统的复杂性就主要存在于模块之间的接口上,因为为了执行任务,大多数的模块需要相互通信。

正确决定系统模块的大小是一个难点,解决这个问题基本规则是“是否感觉到合适?”我们对这个问题的经验是:模块的大小不是个问题,对模块概念性的依赖决定了模块的大小。当一个模块被设计完成太多的事情,它就会变大并且承受着低模块化的问题。但当模块的功能太少并且太细,那么模块就遇到高模块化的问题,那么模块就会变小。我们决定软件模块的方法是直接基于功能的,在这里,用自上而下的方法按等级划分功能。

我们将使用电子商务作为一个例子。在这个例子中,我们的工作是一个电子商务系统。这个系统为客户提供一个 Web 页面,使用户可以浏览产品,选择商品,并购买商品。在这个例子中,系统必须有三个特征:分类浏览、商店库存查询和订购产品。我们接着确定这三个必须提供的主要特征。使用订购产品作为一个例子,发现我们必须提供信用卡确认,交货选择和发货等功能。我们可以继续把这些功能分解成更详细的细节。但在每一层上,我们将评估每个特征的复杂性。而且,为了决定哪些功能需要进一步分解,还要评估每个功能和其他功能的接口。

另外,软件基于功能的分解使模块更具有内聚性。这是因为所有支持功能的任务总是被包含在模块内。这就减小了模块之间的相互作用,使它们更独立,这将有利于减小系统的错误。

像软件工程中的每一件事,模块化是一个重复的过程。在设计中,如果你不喜欢问题的分解方法,你可以重新开始。你可以在最后合成高模块的软件模块,也可以分开低模块的软件模块。

限制灵活性

设计一个 GUI 时，开发者通常选择一个用户能使用整个信息的构件。这个构件的选择影响了软件的复杂性和出现 bug 的危险。

例如，假如你要创建一个 Web 页面，在这个页面上，用户可以输入他的住所状态。作为一个 GUI 设计者，你有两个选择。你可以给用户一个文本框让他们输入他们住所状态的名字；你也可以为他们提供一系列的状态让他们选择。在这两个方法之间，要使用哪个方法呢？对于第一个情况，用文本框输入为用户和设计者提供了灵活性。因为输入是一个开放的文本框，用户可以输入任何信息。如果你要做一个国际的页面，在表示层上不需要修改任何代码。另一方面，用一系列的选择意味着你需要为不同国家的州提供附加的实体。

但一个开放的文本框潜在地意味着需要在后台做更多的错误检查。用户可以输入一个标准的缩写或者一个不标准的缩写，用户可以选择输入州的完整名字，他们还可能出现拼写错误。你需要检查和预料这些潜在的错误，如果你没有这样做，那么就在软件上留下了潜在的 bug。使用列表选择就可以减小这个问题，因为用户的灵活度受到了限制，他们只能输入列表中存在的值。

在 API 设计中，也使用了相同的概念。当你给用户很多使用 API 的灵活性时，那么工作更加困难了。这有很多必须检查的条件，很多代码的路径需要考虑，并且存在更多的潜在 bug 和错误。考虑以下四个方法的声明：

```
1. void doSomething_A();  
2. void doSomething_B( boolean a );  
3. void doSomething_C( boolean a, boolean b );  
4. void doSomething_D( int c );
```

方法 doSomething_A() 没有参数。实现方法 doSomething_A() 的流程是自上而下的，没有分支和条件检查，它不会更容易。方法 doSomething_B() 有一个布尔类型的参数。这就意味着方法 doSomething_B() 必须检查参数值，并潜在有两个代码路径。方法 doSomething_C() 有两个布尔类型的参数，在方法中潜在有四个分支。方法 doSomething_D() 有一个整型参数，这就意味着必须考虑所使用特定操作系统的所有整数的可能值，而且还要处理上溢和下溢的条件，这就有可能出现更多的错误。

相同的概念可以使用在其他设计任务中。例如，XML 是以 Internet 标准的消息格式。一个 XML 消息的数据说明构件被称为文档类型定义 (Document Type Definition, DTD)，它定义了消息的有效格式和内容。给出一个 DTD，一个 XML 的确认分析能用来证实 XML 消息本身的内容。如果消息是无效的，这个分析通常返回错误。所以，为什么不规定 DTD，这样很多的确认工作就可以由分析来完成，而不是代码来完成？使用这个方法，代码能被执行来处理尽可能少的错误，你也可以从 XML 证实分析中重用尽可能多的代码。

提示：不要考虑给用户很多不必要的灵活性。用户只是想解决问题，他们可能不关心灵活性，更不喜欢 bug。

减小接口的不明确性

接口在很多情况下会变得含糊不清,例如,在 C 或 C++ 语言中使用 `varargs`。在 Java 语言中,不明确接口的一个例子是用基本类 `java.lang.Object` 作为参数。一旦这样做,当不正确接受它时,就引入了错误。在面向对象设计中,在类层次中使用了多重基础继承时就会出现不明确接口,并混淆了子类中将使用哪些成员函数。总的来说,当接口的用户不能容易确定如何使用接口时,并且接口没有提供任何使用它的线索时,就可以认为接口是不明确的。

以下代码片断给出了一个不明确接口的很少讨论的例子:

```
101. // Some codes.
102. UnsafeAddress addr    new UnsafeAddress( "100 Main Street",
103.                                         "AnyCity",
104.                                         "AnyState",
105.                                         "00001",
106.                                         "USA",
107.                                         "555-555-5555",
108.                                         "test@test.com" );
109. // some other codes.
```

如果告诉你在第三行代码中存在 bug,你能发现它吗?如果我们不是根据 `UnsafeAddress` 类的定义,我们将不能发现这其中存在的 bug。这个 `UnsafeAddress` 类的定义是:

```
1. public class UnsafeAddress {
2.     private String street;
3.     private String city;
4.     private String state;
5.     private String zipcode;
6.     private String country;
7.     private String phoneNumber;
8.     private String email;
9.
10.    public UnsafeAddress( String _street, String _city,
11.                          String _zipcode, String _state,
12.                          String _phoneNumber, String _email,
13.                          String _country ) {
14.        street = _street;
15.        city = _city;
16.        state = _state;
17.        zipcode = _zipcode;
18.        country = _country;
19.        phoneNumber = _phoneNumber;
20.        email = _email;
21.    }
22.    // other accessors and methods.
23. }
```

你能发现 bug 吗?

如果不能,解决方法如下。`UnsafeAddress` 结构在 102~108 行中的参数没有构造函数的

参数。表 12.1 通过列出构造函数的参数和 102~108 行中使用的参数来证明这个错误。

表 12.1 期待的参数与第 102 行参数比较

期待的参数	第 102 行的参数
street	100 Main Street
city	AnyCity
zipcode	AnyState
state	00001
phoneNumber	USA
email	555-555-555
country	test@test.com

任何人仅仅通过看以上 7 行代码是不可能发现 bug 的。这个问题的难点是问题的出现看起来很合理，任何人都没有理由来怀疑这是个问题。即使有这个类的定义在旁边，如果人们第一眼错过了错误也就不会怀疑我们，因为构造函数的编写者使用了一个非传统的办法给构造函数添上参数。就像在第 4 章中讨论的一样，人们总是倾向发现他们想得到的东西。构造函数的接口是不明显的，存在两个潜在的危险：

- ◆ 构造函数的参数和所列的变量不同。另外，参数表中参数的顺序不是以传统的顺序排列的。这就容易引起另一个开发者像 102~108 行那样编写代码。开发者不是按定义那样编写代码，而是按他们自己的想法来编写代码。

- ◆ 构造函数接口接受 7 个字符串就表明有不明确的接口。编译器不能帮助错误的发现，因为这些参数都是相同的类型。开发者和代码检查者不能仅仅通过观察代码来执行任何语义上的证实。

不过，如果只提供缺省的构造函数，不包括任何参数，并使开发者用集合的方法把目标的内容定义成集合，就像以下的代码段，那么错误就可以立刻被发现：

```
101a. // some codes
102a. SafeAddress addr = new SafeAddress();
103a. addr.setStreet( "100 Main Street" );
104a. addr.setCity( "AnyCity" );
105a. addr.setZipcode( "AnyState" ); // an error
106a. ...
107a. // some other code.

1a. public class SafeAddress {
2a.     private String street;
3a.     private String city;
4a.     private String state;
5a.     private String zipcode;
6a.     private String country;
```

```

7a.  private String phoneNumber;
8a.  private String email;
9a.
10a. public SafeAddress() {
11a.     street = null;
12a.     city = null;
13a.     state = null;
14a.     zipcode = null;
15a.     country = null;
16a.     phoneNumber = null;
17a.     email = null;
18a. }
19a. // other accessors and methods.
20a. }

```

不用查看类的定义，通过查看 105a 的代码就可以发现错误。这就是我们所谓的减小接口的不明确性。这个设计使开发者几乎不可能犯愚蠢的错误，也容易使其他人查看代码和问题，而不需要了解任何 `SafeAddress` 类的知识。这种方法就是包含文档代码，因为代码本身提供了对潜在错误的提示。

不过，使用缺省的构造函数并不是就可以保证你减小接口的不确定性。假如有一个 `Circle` 类，定义了像圆一样的几何图形。在二维世界里，通过圆心 (x 和 y) 和半径定义一个圆。是否像以下一样定义 `Circle` 类？

```

1. public class Circle {
2.     private double x;
3.     private double y;
4.     private double radius;
5.     public Circle() {
6.         x = 0.0;
7.         y = 0.0;
8.         radius = 0.0;
9.     }
10.    public void setX( double _x ) { x = _x; }
11.    public void setY( double _y ) { y = _y; }
12.    public void setRadius( double r ) { radius = r; }
13.    ...
14. }

```

这样定义 `Circle` 类的一个危险是潜在有一个不好的定义。如果在系统中没有问题，那么这个设计可能不会引起任何问题。不过，如果系统仅仅期望“合法”的圆，那么这个设计就会使一些人因为忘了设置圆的所有权而引入 bug。一个 `Circle` 类的安全设计方法是：

```

1. public class BetterCircle {
2.     private double x;
3.     private double y;
4.     private double radius;
5.     public BetterCircle( double _x, double _y, double _r ) {
6.         x = _x;

```

```
7.     y = _y;  
8.     radius = _r;  
9. }  
10. --  
11. }
```

这个设计防止我们有一个不好定义的圆，因为没有详细的参数我们无法生成圆。但类的使用者仍然可能把半径设成 0，这对系统可能是不合理、不明确的。在这个设计中，仍然有和 `UnsafeAddress` 类相类似的问题。也就是，容易混淆构造函数的参数，例如，把参数 `x` 和参数 `y` 混淆起来。怎样解决这个问题？

```
1. public class SafeCircle {  
2.     private double x;  
3.     private double y;  
4.     private double radius;  
5.     public SafeCircle( TwoDLocation loc, PositiveDouble r ) {  
6.         x = loc.getX();  
7.         y = loc.getY();  
8.         radius = r.getDouble();  
9.     }  
10. --  
11. }
```

`TwoDLocation` 和 `PositiveDouble` 是我们创建的两个新类，这将使编译器为你做尽可能多的语义检查。`TwoDLocation` 定义了二维空间的类，存储了二维笛卡儿坐标。`PositiveDouble` 仅仅是一个包含值大于 0 的对象。

用这个方法替代是显然的。通过使用这两个附加的类，我们增加了系统的复杂性，但我们也使类更安全使用，使类更容易读，也使接口更明确。像我们在这本书上提过的其他建议一样，在你决定使用哪个方法之前，你必须考虑系统和环境。

关于异常处理的看法

对于那些支持异常概念的语言，上面的替换方法通过去掉异常来处理不明确圆的定义和不好的参数。但是，使用异常的危险是使用户不容易使用 API。我们已经在第 4 章讨论过，错误条件的处理对软件开发者来说是最困难的任务。对于我们，使用异常处理是懒散的开发者的表现。不是使软件模块更安全、容易使用，而是通过选择对 API 用户的抵抗，使他们担心怎样处理这些错误和异常。为什么不设计一个软件使软件几乎不可能有错误条件？为什么不设计一个让使用者几乎不可能出现错误的软件？为什么要给用户一根上吊的绳子？

12.4.3 设计包含文档的代码

在很多书中，包含文档的代码是指变量、函数和方法的含义明确。但这仅仅是包含文档代码的一方面。包含文档代码的另一方面是可以使人们在一个地方读出关于代码的所有信息。如果可能，我们可以使代码在一个地方告诉我们它做什么和它要做什么。在包含文档的代码中，我们让编译器为我们做尽可能多的检查。在前一节中的 `SafeAddress` 和 `SafeCircle`

的设计就是一个包含文档代码的例子。

在 `SafeAddress` 中, `setStreet()` 方法是一个自我文档, 因为我们大多数人都同意 `street` 的语义意思, 它可能会检查值的内容, 并且还会检查是否会把一个不是 `street` 的值传给这个方法。在 `SafeCircle` 中, 类 `PositiveDouble` 防止用户用 0 或负数为半径创建一个圆。这个类的名字很直观, `SafeCircle` 类没有做任何的错误检查, 因为我们已经排除了创建一个非法圆的可能。

12.5 实现中的 bug 预防

代码实现使 bug 成为可能, 代码实现使需求说明和设计文档成为现实。总体上来看, 我们在实现时产生了 bug, 因为我们不理解用来生成软件的工具, 或者是因为太懒惰写下了包含很多 bug 的代码, 使用了不安全的结构。

12.5.1 理解开发工具

很多开发者缺乏优秀、安全的编程者经过的正确的培训。他们关于设计原则、语言特征和使用、软件开发过程的知识存在漏洞。他们缺乏解决问题的技巧, 他们会比优秀的开发者产生更多的 bug。

为什么必须注意其他人的错误? 软件开发不是孤立的工作。一个人在地下室里进行代码开发的时代已经过去了。今天, 我们是进行团队工作的, 我们依赖相互的软件模块构成一个工作的系统。如果你和一个没有经过培训的人一起工作, 他们的 bug 就是你的 bug。我们不喜欢发现和解决我们自己的 bug, 我们更不喜欢发现和解决其他人的 bug。一个防止不得不去发现其他人的 bug 的方法是确保他们和我们一样是个好的编程者。这使我的工作更容易了。

另一方面, 如果系统的联接很弱, 那么对我们来说最好是提高自己。这就很难成为替罪羊。

理解隐藏的语言和原则

在汇编语言的时代, 有很多隐含在设计语言背后的原则。程序从一条指令执行到下一条指令, 你只是围绕着二进制代码。编写汇编程序意味着必须考虑和跟踪寄存器和存储器中的值。而对于 Pascal 语言和 C 语言, 就要求注意抽象的数据类型和程序的分解。虽然程序还是有序地运行, 但问题是在更抽象的层次上解决。Pascal 语言和 C 语言的程序组织和结构映射了这些抽象。C++ 语言和 Java 语言把面向对象的原理引入编程的主流, 要求一个全新的编程方法。面向对象语言引入了继承、多态和抽象接口的概念。虽然正确使用这些概念是很有用的, 但使用不当的破坏性也是很大的。我们可以把面向对象编程法的使用和核能的使用进行一个比较。当和平使用核能时, 可以为许多人提供能量; 如果错误使用它, 同样也可以很容易伤害很多人。

你见过几次乱用语言的特征？人们乱用语言的特征只有一个原因，那就是人们不知道在乱用它。（另一个原因是人们知道语言很好，以至可以舒服地使用它。这个乱用通常是使用语言的缺点，并引入了这个自身的问题。主要是因为把其他人搞糊涂了。）当语言在程序中被不恰当使用，程序就变得很难理解、很难维护、很容易出错。如果不知道指针是如何工作的，就很难使用 C 或 C++ 语言编程。使用 Java 的 Swing 工具箱意味着你需要知道 Model-View-Controller 结构如何工作。如果试图使用 MFC（Microsoft Foundation Class，Microsoft 基本类），你就要知道它是围绕 Win32 的 API 开发的。知道模板工作原理是理解 STL 的关键。但是，这些东西的理解很难在一个学习或训练中得到。学习可以为你提供特征的概述，但是没有解释特征的含义，也没有说明它们和问题或解决方法的关系。

指导

有很多知识我们不能从课堂学习中得到。最近，我和一些刚毕业的人一起工作。我作为一个指导者，他们作为学生，我们在很强的时间限制下一起发布了软件系统。在这个工程中，我看到他们取得了很好的发展和进步。他们编写代码的速度更快了；他们知道为什么必须按特定的途径来做一件事；他们对所使用的工具和语言更加熟练了。这个项目也使我受益匪浅。这些毕业生帮助我了解了软件工程的困难，给我解决问题的灵感；他们教我交流，教我怎样把技术上的想法说给其他人，而不管他们的技术水平。

指导可以是一个正式的过程，也可以是一个非正式的过程。在正式的情况下，公司可以给学生指派一个导师，可以是永久的，也可以是按工程的。在非正式的情况下，指导仅仅是向团队中的专家学习，这位专家对领域中的知识知道的更多。这个做法就是在会议上列出专家，查看他们的代码和设计，向他们提出问题。这对提高技巧有帮助。

12.5.2 防御编码

在有关文献中提到防御编码的技术。像避免太深的嵌套循环，使用适当的缩进，使用匈牙利符号，使用好的层次技术，使用有意思的变量和函数名，避免全局变量，避免跳转语句等这样的防御编码技术经常被讨论。

assert 语句的危害

当我们鉴定关于调试和 bug 预防技术的书和论文时，一个在这些书和论文中都提到的技术是鼓励我们使用 assert 语句。坦白地说，我们没有发现 assert 语句对提高软件的质量有什么价值。事实上，我们相信使用 assert 语句是懒惰的程序员的表现，像我们在第 4 章提到的一样。我们认为 assert 语句会增加 bug 信息的奇怪性和更多引入 bug，所以应当避免使用 assert 语句。

让我们观察 assert() 做了些什么工作。典型的 assert 语句如下：

```
assert(condition,message);
```

当程序执行到 assert 语句时，条件就被计算。如果条件是假的，那么这个消息就在屏幕上显示，程序终止。另外，assert 语句只在调试模式下才有效。编译器有个选项，使编译器

不会在成品环境下编译 `assert` 语句。结果，`assert()` 成为调试模式的一部分代码。在产品模式下，`assert()` 不存在也不被执行。在产品中去掉 `assert` 语句的基本原理是为了增加性能。毕竟，如果你确认条件不会在调试模式下出现，那么也不会在产品模式下出现。

为什么要使用 `assert()`？在很多应用环境中，这个典型使用是为了确保这个条件有效，并且这个代码不会和非法的前提条件联系起来。下面一段 C 语言代码段是一个使用 `assert()` 的典型例子：

```
1. void ABadFunction( int parameter )
2. {
3.     int *array    NULL;
4.     assert( parameter > 5 );
5.     ...
6.     array = (int *) malloc( sizeof(int) * parameter );
7.     assert( array );
8.     ...
9.     assert( Foo( array, parameter, 5 ) );
10.    ...
11. }
```

在第 4 行，`assert()` 用来确保输入参数比 5 大；在第 7 行，`assert()` 确保能成功分配数组；在第 9 行，`assert()` 用来确保函数调用 `Foo()` 的结果正确。在调试模式下，如果任何一个 `assert` 语句失败了，程序将会停止并给出错误的信息。大概，程序员能使用这些错误信息来调试程序。不过，在产品模式下，当这些 `assert` 语句被编译，将会出现什么情况呢？

在产品模式下，失去了 `assert` 语句提供的错误检查能力。现在，这个函数可以用小于或等于 5 的参数来调用，这就引起一些函数无法处理的 bug。在第 6 行的存储分配就会出错，结果得到一个空数组。在第 9 行中使用的 `assert` 语句在产品模式下会引起更多的问题，因为函数 `Foo()` 根本就不会被调用。如果 `Foo()` 仅仅是一个检查函数，也就是说，它不会修改程序执行的状态，那么我们就失去了第 9 行的有效的检查能力。同样，如果 `Foo()` 是一个活动的函数，它的计算结果将被后面的程序使用，那么我们就失去了计算结果，程序将不能正确执行。

在这个例子中，我们指出了两个使用 `assert` 语句的主要危险：

- ◆ 仅仅因为 `assert()` 没有在调试模式下出错，并不意味着它也不会成品模式下出错。使用 `assert()` 是因为有可能出现假条件。如果你认为这个假条件可以在开发和调试环境下出现，那么有什么理由认为它不会在成品环境下出现？通常认为，在单元测试、综合测试和系统测试中，`assert` 语句可以用来发现所有对接口的非法使用。在前面一个例子中，`ABadFunction()` 要求参数的值大于 5，但我们无法知道函数的调用是否满足这个约束，无法知道测试过程是否会产生所有可能调用 `ABadFunction()` 的参数值。既然这样，就要确认参数将满足函数的前提条件。仅仅是因为它不会出现在测试和调试阶段，不能说明它也不会发生在产品中。

- ◆ 在调试和产品环境下运行不同的代码版本。因为 `assert` 语句可以不被编译，使用 `assert`

语句意味着你在产品环境中使用的版本和在调试环境中使用的版本可以不同。在产品环境中使用前面的例子，如果因为没有了第 9 行而引起 bug。然后你在调试环境下调试代码，这个 bug 就会消失，因为第 9 行又出现在调试模式中。但是，要花多长时间你才能意识到这点呢？

在很多鼓吹使用 `assert()` 的书上，作者意识到不正确使用 `assert` 语句的危险，让读者重新定义 `assert` 函数，避免使用 `assert()` 带来的危害。他们写了好多关于什么时候使用 `assert()`，什么时候不使用 `assert()` 的警告，但是这些警告太复杂、太不安全。除非我们能确认这些 `assert` 条件不会有效，否则编译器在成品环境中排除 `assert` 语句总会存在危险。很容易，也很安全在你能使用 `assert` 的地方使用明确的错误检查，这使代码量变多，但也变得更安全。这也要求你理解不同的错误条件，并且用特定的方法处理不同原因引起的错误。在很多环境中，你处理非法的参数和处理存储器溢出的方法不同。任何对没有很好定义的条件错误的处理机制都应当回到需求层处理，这将有助于重定义需求，并使需求更完整、更明确。

下面代码段是前面例子的一个较好的实现。通过添加了所有的错误处理使代码增加了复杂度。我们看来，是值得这样做的。

```
1. int ABetterFunction( int parameter )
2. {
3.     int *array = NULL;
4.     int FooResult = 0;
5.     if ( parameter < 5 ) return ERROR_CODE_1;
6.     ...
7.     array = (int *) malloc( sizeof(int) * parameter );
8.     if ( array == NULL ) {
9.         graceful_exit(0); /* might save the data...etc */
10.    }
11.    ...
12.    FooResult = Foo( array, parameter, 5 );
13.    if ( FooResult == 0 )
14.        return ERROR_CODE_FOO;
15.    ...
16. }
```

综合直接错误检查和日志跟踪

在 `ABetterFunction()` 中，我们已经排除了使用 `assert` 语句，并返回了所有可能的错误条件。但像在第 4 章提到的，错误条件处理是编写软件最困难的一点。很难定义正确的错误处理，并且容易忘记检查错误条件。在 `ABetterFunction()` 中，如果调用者在它的代码上存在 bug 并用非法的参数调用函数，函数中的 `assert()` 将会使程序停止运行并通知问题的开发者。开发者为了发现错误就必须检查 `ABetterFunction()` 返回的错误。如果他们不这样做，他们就要在这个 bug 上花费很长的时间，因为这个 bug 是隐含在设计中的。

一个解决这个问题的方法是把日志跟踪和直接的错误检查综合起来，就像下面的 `ABetterFunction2()` 函数一样：

```
1. int ABetterFunction2( int parameter )
2. {
```

```

3.   int *array = NULL;
4.   int FooResult = 0;
5.   if ( parameter <= 5 ) {
6.       logger( ERROR_MESSAGE, "AbetterFunction2",
7.           "Parameter must be greater than 5" );
8.       return ERROR_CODE_1;
9.   }
10.  ...
11.  array = (int *) malloc( sizeof(int) * parameter );
12.  if ( array == NULL ) {
13.      logger( ERROR_MESSAGE, "AbetterFunction2", "Cannot allocate array" );
14.      graceful_exit(0); /* might save the data_etc */
15.  }
16.  ...
17.  FooResult = Foo( array, parameter, 5 );
18.  if ( FooResult != 0 )
19.      logger( ERROR_MESSAGE, "AbetterFunction2", "FOO Failed" );
20.  return ERROR_CODE_FOO;
21.  }
22.  ...
23. )

```

记录器的功能是记录严重的错误代码、函数的名字和消息。记录器的功能就像“跟踪日志文件”一节中提到的一样，它可以在软件开发期间帮助开发者，也可以帮助调试者和维护者发现问题。

总之，使用 `assert()` 有三个基本的危险。第一个使用 `assert()` 的危险是由于使用有差异的软件不同版本——一个版本是包含了 `assert()` 的调试模式，另一个是不包含 `assert()` 的产品模式；第二个使用 `assert()` 的危险是开发者有一个安全的感觉，没有证实如果在测试时成功使用了 `assert()`，在产品环境下也能成功使用 `assert()`；第三个危险是在产品环境下函数 `assert()` 也可能有效，当产品环境中存在 `assert()` 时，在 `assert` 失败时，就有可能出现预想不到的终止程序现象。

尽可能使用编译器

在我们编译代码时，我们通常打开编译器提供的每个警告。我们的目标是产生一个没有错误、没有警告的编辑输出。这通常很难忽略错误信息，因为在有错误的情况下编译器不会生成一个可执行的程序。不过，特别容易忽略警告，尤其是在你没有意识到这些警告的时候。但这些警告可以帮助你避免愚蠢的错误，使你规划代码，让代码变得更明确。一些 C++ 编译器警告你从一个对象类型中抛弃另一个对象；一些编译器也警告你潜在的不明确计算。一定要慎重对待编译器的警告。看以下一行代码：

```
if(a==b && c!=0 || d>5)
```

可能会引起编译错误，我们把它改写成以下这个形式：

```
if((a==b) && ((c!=0) || (d>5)))
```

编译器就能接受上面这个语句。这个结构的另一个优点是使我们不用再计算它的值。我

们必须使代码更清楚，从而更容易理解。

有时，我们会关闭一些警告，因为我们知道它们不会提供有用的信息，相反，只会混乱其他一些潜在的有用信息。一个关掉警告的最好例子是 Microsoft 的 Visual C++，它必须截断超过 255 个字符的名字。这通常发生在模板实例化期间，它会使屏幕的输出混乱，以至不可能发现有用的警告。

但是除了打开警告，还可以做其他更多的事情。可以在设计和实现代码时，使编译器做尽可能多的确认。可以通过尽可能的强制类型转换，来增加编译器的功能。如果编程语言能保证不发生类型错误，那么这个语言就是强制类型的。Java 语言不是强制类型的，因为在运行期间它可能出现类的异常错误。C 和 C++ 语言也和它一样，允许程序员用一个类型的指针指向另一个类型。

在“减小接口的不明确性”一节中，我们讨论了使用特殊的结构和类来实现函数接口，使接口参数的语义明确，减小必须的错误检查。不是在圆的半径使用一个双精度类型，而是创建一个类仅允许使用正数的双精度值。这样，Circle 类就不用当心半径的合法性。为了使接口参数明确化，我们也可以使用编译器来帮助我们，因为我们创建了一个半径的强制类型。

12.5.3 实现选择

实现设计的方法也会影响到产生哪种类型的 bug。如果使用 strcpy() 和静态存储器，就有可能出现缓存溢出的错误；如果是动态分配存储器，就要处理存储器溢出的错误。选择实现方法的简单规则是使所用的方法尽可能简单和清楚。

避免使用模糊的语言特征

我记得在很多年前，我面试一个 C++ 程序员的应聘者。在面试快结束时，我问他对自己的 C++ 能力有什么问题。立刻，他提出了一些 C++ 语言在处理全局变量的模糊之处。说实话，我不知道他在说什么，我也不记得他的话，我从来没有听其他人提出过这个问题。我记得我问他碰到这个问题怎么办？他的回答是把这个问题和已经解决的问题联系起来。显然，我立刻想到追踪问题将要花费他几个星期的时间，为什么？什么工作要求他使用这样一个模糊的语言特征？他是否可以使用另一个设计软件的办法，这样就可以不使用这个模糊的特征？

关于模糊语言特征的问题是这些特征仅仅是模糊的，除了语言的开发者，没有多少人知道这一点，没有多少人注意到这一点，除了语言的开发者。更有可能的是，不是所有的编译器都支持这些特征。

指针问题

指针问题显然不会出现在没有使用指针、简明的存储器分配和释放的语言中。在 Java 中不会出现指针问题，因为无用的数据单元和 Java 中的每个对象实质上都是指针。在 C 语言中也没有这个问题，因为每个存储单元的申請和释放都是明确的。不过，在 C++ 语言中，指针是一个麻烦的问题。最基本的问题是由谁来负责类成员变量中指针的分配和回收问题。

我们将从以下这个例子来讨论使用指针的困难:

```
1. class B {  
2.     ~  
3. };  
4. class C : public B {  
5.     ~  
6. };  
7. class A {  
8.     public:  
9.         A();  
10.        A( const A& src );  
11.        virtual ~A();  
12.     private:  
13.         B *pointer;  
14. }
```

在这个例子中, 当类 A 被实例化后, 指针变量也同时被分配和产生了。当类 A 的一个对象超出了范围, 那么解除程序就负责回收指针成员变量。这就意味了解除程序的副本也要明确地分配指针成员变量和从源对象中复制指针成员变量的内容。

不过, 我们使用指针的一个原因是想使代码更有效地执行, 所以我们不用在任何时候都复制和生成对象。另一个原因是我们要在代码中使用继承和多态的概念。此外, 在类 A 中, 我们想让指针变量也能存储类 B 和类 C 的对象。我们需要生成一系列的方法来调用类生成对象, 像以下的代码段:

```
7a. class A {  
8a.     public:  
9a.         A();  
10a.        A( const A& src );  
11a.        virtual ~A();  
12a.        void setPointer( B *data );  
13a.     private:  
14a.         B *pointer;  
15a.     };
```

用 setPointer(), 我们可以用指针调用类 B 和类 A 的对象, 并满足我们的需要。但我们也面临了进退两难的处境, 这个困难是: 谁拥有指针成员变量? 如果是类 A 拥有指针成员变量, 就意味了每次都要调用复制构造程序, 指针变量也需要被复制, 保护机构也需要删除指针成员变量; 如果类 A 不是指针成员变量的所有者, 那么类的使用者就要注意指针变量的存储分配。此外, setPointer()是不明确的。类的使用者没有读文档就不知道谁是数据的所有者。

可以通过删除 setPointer()方法, 使类 A 变得更明确:

```
7b. class A {  
8b.     public:  
9b.         A();  
10b.        A( const A& src );
```

```
11b.    virtual ~A();  
12b.    void setB( const B &b_src );  
13b.    void setC( const C &c_src );  
14b.    private:  
15b.    B *pointer;  
16b.    );
```

setB()和 setC()的明确接口使调用者根本不用考虑存储器问题, 因为类 A 将处理指针变量的存储问题。事实上, 调用者甚至不知道基本的表示法是什么。一个明显的错误是, 如果从类 B 继承地创建一个类 D, 并希望通过类 D 调用类 A, 为了满足需要, 就必须修改类 A。

12.5.4 逐步测试

在实现阶段, 每个人都有自己的行为。通常有两个阶段: 实现和测试。当我们进行一个代码工程, 我们通常把这个工程分解成小的函数块。这个工程可以像软件模块一样大小, 也可以是一个类库。在类库中函数块可能是一个个独立的类。这个工程也可以是类, 那么函数块可能就是主要的方法和方法之间的接口。工程也可能是单个方法或函数, 对于这个工程, 函数块就是方法内的每个步骤。

一旦我们完成了函数块的实现, 我们就要测试它。当函数块被测试, 我们就继续下一个部分的工作。在完成了第二部分的实现时, 我们就同时测试第一部分和第二部分。我们重复这个过程直到完成了整个工程。

如果函数块很小, 这个过程就要花费很多时间。但是逐步测试过程的主要好处是你总是进行代码工作。如果你在开发一个新部分时出现了错误, 就最有可能出在刚刚写的代码中。出现 bug 的原因可能在已测试过的部分中, 但是是新的代码的使用触发了这个 bug。按这种过程工作就可以帮助你在较少的代码中发现 bug。

自下而上/自上而下的逐步测试方法

有很多实现逐步测试的方法。一个方法是用自下而上的方法实现功能块。在自下而上方法中, 先实现继承底层的函数和类。所以, 当你实现一个功能块时, 所有你需要实现的函数和类都必须已经实现并且可以使用。因为在自下而上方法中需要实现和测试期望的行为和特征, 这就确保了这些行为和特征的正确实现。

在自上而下方法中, 首先实现上层函数。当需要其他类和函数, 不是先去实现它们, 而是先实现一个内核, 并让内核返回错误条件。首先实现上层函数的代码就可以处理错误条件。接着实现底层类和函数的行为。

实际上, 我们可以根据开发环境、时间和软件的最困难部分来联合使用了自下而上和自上而下的一步测试方法。如果我们认为最难的部分是正确获得特征, 那么就使用自下而上的方法; 同样, 如果我们认为最难的部分是错误处理, 那么就应该采用自上而下的方法。

不要忽略 bug

在实现一些代码时, 可以使用任何的过程和方法。在实现时, 每一件事都是正确的。但

当测试实现时，程序崩溃了。你到处查找原因。你乱搓着手，直到发现程序崩溃的原因是因为找不到一个必须的文件，或者是因为外部文件所需的入口出现错误。你该怎么做呢？

要么解决了文件问题并继续测试，要么解决了程序，这样就不会因为不好的文件而使系统崩溃，然后接着测试文件。

我们发现很多程序员仅仅解决了文件然后接着测试。由于某些原因，他们认为程序的不正确行为不是因为不好的环境依赖关系。在这种情况下，我们使用不好的文件作为例子。可能问题是因为数据库访问冲突、断开的网络连接或者是硬盘错误。不管怎样，这些事件很少发生，当遇到这些事时，应当利用状态来实现代码，去正确处理这些不经常发生的事件。在实际使用环境中，测试这些代码是一个机会。而在测试和维护时重现这些条件是非常困难的。

事实上，应当解决程序，这样就不会引起不好的文件的崩溃，然后再解决文件。解决文件接着继续测试仅有的故障现象，并让这些错误留在程序中。在碰到错误文件时，如果不知道程序的正确行为应该是怎样的，就应当使用需求文档或者向需求分析人员询问。不要把错误留在程序中让别人最后来解决。

12.6 小结

我们知道开发步骤和过程关系到软件的质量。可以说，如果有一个可靠的、完全的、清楚的、可达到的合乎测试的需求，并且这个需求被大家认可，我们就能创造出高质量的软件；如果做好了安排，我们就有时间考虑设计，有时间收集需求，有时间测试，我们就能创造出大量的软件。容易理解我们需要交流和使用正确的工具。通常情况下，事实经常和我们的想法背道而驰。

我们在这一章要解决的问题是给出一些在实际环境中有利于实现一些步骤和过程的建议。我们也试图给出一些不需要太多时间的讨论，并提供了这些问题的解决方法。工程需要评估风险和使用可替代的解决方法。关于 bug 预防的讨论将帮助你考虑问题并给出在选择方法过程中对所选择方法的考虑。我们也试图提供一些技术，这些技术可以发现在第4章所提及的软件系统中的 bug 的原因。

在第4章，讨论了引起 bug 主要原因的复杂性，还提出了很多处理和管理复杂度的建议。总的来说，有一个过程能帮助我们处理这个复杂性。如果真正理解了问题并且在一开始就得到了正确的需求，我们就能处理这个复杂性。通过正确分解问题、创建正确的模块并使接口简单明确就可以容易解决问题。如果不是试图使用最一流的解决方法，就可以使我们的解决方法变得简单。

真实性是形成 bug 的另一个主要原因。时间和资源的缺乏是主要因素，较短的时间就要求雇佣较多的人，我们就需要对资源进行管理。这一章的很多建议考虑了时间和资源的问题。不变的修改是不会很快就消失的，我们已经建议使用捕获需求和重用技术来减少修改的危险。另外，我们介绍了创建现实世界的文档和使用错误原因分析技术集合整体技术，并发现

错误源。

第4章提到，人类的缺点是引起 bug 的另一个主要原因。我们在每个原因上都犯了错误。检查和不同的防御代码技术能帮助减少因为人们简单错误所引起的 bug。

最后一点：这一章的思想和技术是一个起始点，它们建立在我们的经验和习惯上。每个人都是不同的，并且我们工作在不同的环境中，对我有用的东西可能不适合你。但还是要尝试这些思想并实验它们。如果对你有帮助，那就太好了！如果没有，就要查找这个思想的失败点，这样你就可能会得到其他一些更好的东西。为了得到更多的思想，我们建议读者阅读一些附录 B 上的论文和书籍。读其他人的想法并考虑如何在你和你的环境中应用这些技术是很重要的。产生一个没有 bug 的代码要求很多训练、约定和自我测量。这些事不是一朝一夕就会发生的，也是不容易做到的。

12.7 bug 问题

在这一章的前面部分，我们讨论了重用的危险。以下这个问题就是不加思考地重用可能引起的问题。

在这个情况下，我们有一个客户和服务环境。客户端可以是任何东西，最基本的，它是一个显示一些信息的 GUI。为了简单起见，我们假设 GUI 显示了公司的小配件列表。在显示器的左边，是一列小配件的清单。你可以点击每个清单，GUI 将在右边显示清单上的产品。另外，下拉式菜单也包含了 GUI 中的一系列清单。GUI 用户使用了一些标准的 API 和服务通信，服务器给出客户需要的内容。

客户对 GUI 客户端的执行不满意，你的工作是提高系统的速度。在你挖掘代码和日志文件时，你发现服务器对客户的请求先后响应两次。这个任务是在数据库中检索清单。为什么日志文件会显示这个行为？你怎样去发现这个问题？

我们列出一系列潜在和这个行为相关的问题：

- ◆ 服务器对请求的响应是否存在 bug？
- ◆ 服务器软件是否存在 bug？
- ◆ API 是否存在 bug？
- ◆ 客户端软件是否存在 bug？

我们假设每个代码段都是正确的，并且假设软件中不存在 bug。（提示：考虑 API 中必须有什么？一个懒惰的程序员怎样使用 API。）

第 13 章 测试

在调试的书中如果没有测试这一章，这本书就不能结束。测试和调试是并存的。没有测试，就只能解决很少一些 bug；没有调试，测试者就发现不了系统性能的提高，也没有新的测试内容。测试是发现 bug 的过程，相反，调试是解决 bug 的过程。两者缺一不可。

因为这不是一本关于测试的书，在这一章后面列出了很多关于测试的优秀书籍。这是一本关于调试的书，所以当我们讨论到测试时，我们是从调试者的角度出发的。更重要的是，是从 bug 的角度出发的。毕竟，这是一本以 bug 为中心的书。

从一个专业调试者的角度出发，有三个不同类型的测试：单元测试、验证测试和 QA 测试。前两个测试包含在调试者的领域中，第三个测试是一个外部种类的测试。接下来我们讨论每个类型的测试和它们各自包含的内容。

13.1 单元测试

单元测试在单个模块中由模块的初始开发者完成。单元测试是向开发者证明单元是按设计工作的。总的来说，单元测试和任何其他软件模块都没关系。单元测试只要求模块包含了被测试的真实模块。

13.1.1 测试什么

最低程度，单元测试测试了模块中所有的路径。模块的接口也要被测试。为了确保模块被完整地测试，必须用所有进入模块的合法数据形式来测试接口。另外，还要用尽可能多的非法数据格式来测试模块，确保模块能正确处理非法的数据。除了要产生正确的模块输出，模块的输出还要正确地打印出来。只有这样，才能证明模块确实按你的设计进行工作。

开发者制定单元测试存在着的一个普遍问题是忘记了可能通过不同的方法取得所使用的数据类型。例如，如果使用脚本语言，比如 JavaScript，就可能会自动把不同的数据类型转变成所期待的数据类型。在这种情况下，证明所得的数据是正确的，这一点很重要。

13.1.2 单元测试为什么重要

从调试者角度看，单元测试是重要的，因为它可以让你不用关心标准的模块使用是否正常，而把注意力集中在单元功能的其他方面。另外，可以用单元测试证明：对模块的修改不会影响现有的功能或者改变模块的行为。

单元测试重要性的另一个原因，不像前一个原因那么明显，那就是单元测试提供了一个

对所给模块文档的前期检查。例如，单元测试可以作为对使用模块的一个例子。经常地，其他开发者把单元测试作为他们自己开发的基础，因为他们可以从测试结果中知道哪些对模块的接口有影响，而哪些对模块的接口没有影响。

13.1.3 怎样应用到调试中

有几个办法把单元测试应用到调试中。首先要提的是，可以使用单元测试来证明：修改不会对任何现存的代码行为产生影响。另外，可以用测试条件来查找对模块的最后一次修改可能带来的问题。如果能通过单元测试快速而简单地发现问题，就不要通过研究整个系统的代码来发现问题，而只要把调试作用限制在一个模块中。

13.1.4 单元测试有什么重要问题

在写单元测试时，要考虑三个问题，特别是对不是自己写的代码的测试。在项目第一次发布之后，调试者的最好介入时机是项目的提高和维护阶段。

单元测试对理解如何使用代码是很重要的。从文档和现有源代码的使用和内部的研究，应当能发现函数和方法应当在其他函数和方法之前被调用。单元测试不仅仅是用来证明代码确实在没有遵循适当的调用顺序时正确地报告错误，也用来作为一个文档的例子，解释最后的开发者如何适当地使用代码。

单元测试的另一个重要特征是研究所测试的函数或方法的接口，并证实你不仅选择了合法的输入类型，也清楚使用了非法的数据类型。例如，代码希望用一个指针指向一个目标，像很多 C 或 C++ 函数那样，你可能会测试所有的可能值：空指针、指向局部变量的指针和指向全局变量的指针。同样，如果代码希望一个整数值，那么你可能会用浮点数、负数和 0 来测试它。这些简单的测试条件可以容易地发现所测试函数的问题。

单元测试最主要的功能，是证明已经检查代码中的每个分支。通过执行代码的所有分支，确保对代码进行了全面的检查。这一点很重要，因为很多 bug 只有在特定的代码片断第一次执行时才会出现。在很多情况下，这些分支包含异常的条件和一系列的错误，这些错误是开发者从来没有意识到的，也没有适当考虑到的。

13.2 验证测试

验证测试是测试的最基本形式，开发者都应当执行它，不论他们是否对自己所写的代码做了其他的测试。简单来看，验证测试就是验证代码是否做了应当做的事。注意，这并不意味着代码应当完成需求文档要求的每一个功能。对于一个给定的模块的接口，验证测试只是简单地证明使用正确的输入，模块将会产生正确的输出。

在很多情况下，验证测试是单元测试的第一步。通过用正确值来测试模块，就容易发现哪些值是非法的。另外，只通过检查模块的功能，就可以发现一系列可能的问题。验证测试

并不是模块质量的充分测试，但它是发现是否选择合适的实现方法和编码的好方法。

此外，像单元测试一样，验证测试为调试者、维护者和模块的重用者提供了一个生成文档的好工具。从调试者的角度看，验证测试使调试者只注意到要测试的模块，并从中提炼出不同的情况。如果标准发生了改变，一个完全有效的输入很有可能会在最后的版本中导致错误。通过把系统简化成最小的版本，就可以把注意力集中在可能出现问题的模块，就可以直接给出现问题的模块输入可能引起问题的数据。

13.3 质量保证测试

通常，当我们讨论软件的测试时，我们指的是质量保证（Quality Assurance, QA）测试。QA 是由 QA 人员执行的结构测试。QA 测试是为了发现软件是否能够工作，也为了证实软件不会做不支持的工作。提到 QA 测试很奇怪，但也是必须的。程序员当心软件会不会像说明书那样执行，但是 QA 人员更关心的是用从来就不希望的方法使用软件。

现在，让我们考虑一个计算数的平方的函数。当然，数的平方就是这个数乘以它本身。通常，这个函数如下：

```
double square( double in )
{
    return in * in;
}
```

这个函数看起来足够简单了。可以使用不同的输入，例如 1、2 甚至可以是 0 来测试。如果通过了以上的测试，还可以用 -1 来测试。所有的这些输入对测试这个函数都是有效的。对于 C 语言，这些也是合理的测试数据。还可以用小数来测试，例如 0.5；也可以用非常大的数来测试，例如 1,000,000。在每种情况下，你都会发现函数能正常工作。

现在，让我们看一个复杂的例子。假如屏幕允许你输入 1 到 10 之间的数。接着，这些数被用在另一个屏幕上。这个处理函数如下：

```
void choose( const char *input )
{
    switch ( atoi( input ) )
    {
        case 0:
            DoScreen();
            Break;
        case 1:
            DoScreen0();
            Break;
        // Remainder of cases to 10
    }
}
```

同样，这个函数也是很简单的。作为开发者，应当通过调用以下的输入来测试这个函数：
choose("1");

```
choose("2");  
choose("3");  
choose("9");
```

不过，作为一个 QA 人员，就应当用不同的输入集来测试。为了发现是否正确处理屏幕上的输入，QA 工程师将输入“ABC”、“-11”和“J”。所有这些合法的键盘输入，而且在设计时你是从来都没有考虑过这些输入的。事实上，大部分的程序员都没有考虑到不输入任何东西就通知屏幕进行选择的情况。

测试、可维护性和复杂性

测试和可维护性的关系是一个复杂的关系。不过，确定它们之间的部分关系是比较容易的。例如，用来测试特征、模块或函数的测试用例的数目是这些元素复杂程度的表现。复杂度是可维护性的表现，所以你可以通过测试特征或模块的测试用例的数目来预测这些模块和特征将来的可维护性。

复杂性也是重要因素，我们已经提到，模块中 bug 的数目和模块的复杂度有直接的关系。因为测试用例的数目是复杂度的体现，那么它也是对功能块可能出现 bug 数目的一个好的预测。既然如此，把功能块分解成尽可能低的层次，来预测将要发现 bug 的数目是很重要的。因为这个发现 bug 的做法是测试者的基本工作，可以发现调试和测试几乎是同时进行的。

你怎样估计所给模块的测试用例的数目？一个好的方法是查看功能块或模块的接口。无边界类型，像整数和浮点数，比起用户自定义类型有更大的出错可能，比起简单的 1 到 12（或 0 到 11）就存在较少的错误。

13.4 测试方法

在很多情况，测试就像城市街道的垃圾。每个人都喜欢谈论它，但没有人愿意来做它。实际上，这个问题的原因是很少有开发者和调试者真正知道怎样来测试。在调试时，最好的理解测试的方法是了解可以使用的技术，并选择使用最适合的技术。

和调试不同的是，测试对问题没有简单的答案。在调试时，可以寻找解决方法并解决所出现的问题（bug）。但在测试中，却无法通过运行简单的测试就能发现问题，并知道测试是合理的。相反，测试是使你对自己证实信服的手段，集中精力在某些可能出现问题的地方，并确定已经完全解决了这些问题。测试系统的真正问题在于：随着系统复杂性的增加，用来测试系统的测试用例也随着增加。所以不管调度和管理是多合理，都不可能运行无限的测试用例，那么就需要使用最有效的技术。

在下面几节，我们将讨论这些测试技术。我们描述了每个技术、这个技术最适合的环境和系统、并用代码或程序段给出怎样使用每个技术的例子。

13.4.1 路径测试

路径测试就是使用最少的测试用例，测试系统中的每条路径，确保每条路径都正确执行。

这就意味着每个条件、每个循环和代码中的每条路径都必须执行，保证它们的正确性。

让我们看一个用 Visual Basic 写的测试路径的简单例子。

```
Sub Foo(X As Integer)
    If X < 1 Or X > 10 Then
        Exit Function
    End If

    Dim Flag as Boolean

    Flag = False
    For I = 1 to X
        Select Case I
            Case 1
                Do_Something
            Case Else
                Flag = True
                Do_Something_Else
        End Select
    End For
    GNotFound = False
    If ( Flag = False ) Then
        GNotFound = True
    End Sub
```

需要使用多少个测试用例，才能保证这个子程序的所有可能路径都会被执行？为了回答这个问题，让我们先看一看代码中的条件语句。代码中有 3 个条件语句：两个 If 语句和一个 Select 语句。Select 语句有两个条件，需要两个测试用例使所有的路径都执行。另外，还有一个简单的循环在子程序内部执行。

例如，让我们看一看子程序的第一个条件语句。这个语句是：

```
If X<1 Or X>10 Then
```

在这个语句中，注意到 X 是程序中的一个整数值。因为我们不用计算它，我们可以通过给函数一个简单的 X 的值来测试这个语句。为了最小地测试这个语句，要使用多少个 X 的值来测试？最少需要 5 个值。需要一个上限（10）、一个下限（1）、一个中值（5）、两个极值（0 和 11）。如果要更完全地测试，也可以用整数类型的最大值和最小值（-32,767 和 32,767）来确定不会发生某些严重的问题。仅仅这个语句就需要 7 个测试用例。

下面接着看代码中的最后一个 If 语句，它是基于函数其他地方的一个布尔值。为了充分测试这个条件，我们需要考虑这个值的设定。查看代码，我们知道设定布尔值的条件和选择选项是有关系的。

如果根据这一节提过的设计每个模块的单元测试的过程，就可以确保有一个好的测试流程测试模块。当你知道代码中有很多 bug，而这些 bug 在新的代码路径执行前没被发现，那么就应当考虑使用路径覆盖，也就是路径测试。

13.4.2 事务处理测试

事务处理是一系列可能发生也可能不发生的事件。在事务处理中，重要的是事件什么时候失败，系统在事务处理之前能否恢复到它的原来状态。另外，事务处理可以看成是原子事务。也就是，不论系统同时发生了多少事务处理，在另一个事务处理访问实体前，对这个实体有影响的事务处理都已经完成。

测试事务处理是系统测试中最困难的一点。如果事务处理中每个事件有 3 个失败的可能，每个事务处理中有 5 个事件，你将发现测试用例的数目太多以至无法处理。你不仅要测试每个可能的故障方法，还要测试其他可能成功的事件。这就导致了测试用例的个数按几何级数增长。因此，不可能用所有的测试用例对系统进行测试，那么在最终的系统中就可能存在 bug。

为了更合适地检查事务处理测试，让我们看下面一段银行交易的伪代码。在这里，A 写了一个账单给 B，要求账单兑现。

```
Person A writes Check  
Person B presents check to bank  
Check is debited from the account of Person A  
Check is credited to the account of Person B  
Person B receives confirmation of the deposit  
Person A receives confirmation of the check clearing
```

我们可以合理地假设：填写账单和把账单给银行这两个事件不存在 bug。事实上，正如前面所讨论的，任何的过程都可能存在 bug，但我们在这个例子中不考虑这个问题。

我们仅仅从银行向 A 支付账单开始考虑出现错误。首先，这个账单为 B 储蓄了，但没有从 A 得到借款；其次，这个账单从 A 得到借款，但没有得到 B 的确认。所有这些确定都不能通过证实。在任何的情况下，惟一解决方法是在填写账单之前对异常事务处理并恢复到原来状态。失败的处理将出现可怕的结果。

我们怎样测试这个过程，来发现哪些地方出现事务处理问题呢？最明显原因是借方或贷方失败。测试用例可能包含锁定存单，这样就可以使记录不工作。不过，对其他存单的操作还是可以继续工作的。在事务处理失败时，这个测试将检查最后的结果和起始值是不是相同。

如果确认出现错误将会出现什么情况？A 和 B 的账单是否会恢复到原来的状态？在每个账单上是否有正确的存款记录？这些都是要在交易系统中充分考虑的事。这个系统要么完全成功执行，要么就要像初始状态一样，什么事都没有发生。当然，如果事务处理失败，希望有一系列错误的日志和存款的审查，或者是账单记录的其他信息。如果不能做到这些，将会导致系统不平衡，或者不能审查存款。

13.4.3 输入验证

如果程序没有接收任何的输入，那么系统中的 bug 不仅会大大地减小，这些 bug 也会更容易被发现和解决。因为有很多不同的输入，也无法预料到用户的输入，输入是程序问题最

主要来源。输入验证包含两个阶段。第一，正确的输入必须产生正确的输出、执行正确的过程。这是最简单的输入验证，也是开发者对系统和模块测试的一个组成部分。第二，非法的输入必须不会引起问题。非法的输入应当被及时发现并报告。

输入有很多不同的来源。在命令行参数、程序入口或文件块中，输入可以直接从用户那里得到；输入也可以是间接从别的文件和其他程序中得到。另外，输入也可以从机械设备中得到，例如显示器、附属硬件设备或其他系统的外设。为了理解怎样正确测试系统或模块的输入，你应当理解进入系统的输入。

让我们看一个简单的模块，理解可能出现的这种输入：

```
int main( int argc, char **argv)
{
    if ( argc < 2 )
    {
        printf("Usage: filter <filename>\n");
        exit(1);
    }
    FILE *fp = fopen(argv[1], "r");
    while ( !feof(fp) )
    {
        if ( !fgets(szBuffer, kBufferSize, fp) )
            break;
        FilterLine( szBuffer );
        printf("Filtered line: <%s>\n", szBuffer );
    }
    fclose(fp);
    printf("Completed. Hit any key to continue");
    getch();
    return (0);
}
```

如果你了解 C 语言，就会发现不是一两个输入，而是有三种不同的模块输入。为了确保正常工作，需要验证每个输入。第一，有一些程序的命令行参数。它们从主程序的 `argc` 和 `argv` 参数得到。`argc` 变量是函数参数的个数（从 1 开始，第 0 个参数是 C 语言的程序名），`argv` 数组包含了真正的字符串参数。因为我们显式地使用数组，并假设数组包含了数据，程序通过检查 `argc` 是否足够大以至能包含文件名来验证输入。

一旦从命令行参数读到文件名，那么这个文件名就打开同名的文件。这个文件被一行行读进来，这是文件过程的输入。最后，文件关闭，用户按键来选择继续执行还是退出程序。这样就可以避免一批文件中的某个文件的消失。

在这里，我们是从开发者的角度来检查系统。现在，让我们从测试者的角度来检查测试的可能性。

第一，没有参数的程序测试是可以通过的。在这种情况下，`argc` 值为 1（函数名），`argv` 数组将不包含任何有效的文件名。这个条件被程序的第一个条件语句捕获，程序在打印错误

消息后退出。第二个测试将输入多于一个的文件名。像它实现的那样，系统将忽略任何输入的附加文件名。这可能是一个 bug，也可能不是，要由系统的需求来决定。作为测试者，你可能会觉得这是一个 bug，因为你觉得程序或者是仅处理单个文件，或者是处理列出的所有文件。

在处理了文件名之后，下一步就是打开文件。如果输入一个非法的文件名，将会出现什么情况？像程序实现的那样，系统将崩溃。显然，这就存在一个 bug。结果，我们发现了输入过程的第一步——必须检查全部文件名的有效性和访问权限。为什么要检查访问权限？如果只简单检查文件是否存在就去访问没有许可的文件，这个程序将会出现问题。为了确保能打开文件，就必须检查访问权限。

提示：检查所有要打开文件的有效性和访问权限。

因为存在不同类型的文件，在操作一个文件时仍然会出现问题。文件类型有文本文件、二进制文件以及这两种文件的混合。为了正确测试系统，我们就必须测试所有我们可能运行在真正环境下的文件类型。在一些系统中，文件可能是数据库文件、文本文件和二进制文件。要记得去测试这些所有的可能类型，而不是开发者在设计系统时考虑到的简单的一两种文件。

我们已经讨论了代码的三个不同部分。最后一部分（文件打开错误）将导致程序的崩溃。如果使用了错误类型的文件，另一部分（处理过程）可能引起严重的问题。这是两个部分，关闭文件并让用户按键选择继续。让我们检查最后的两个可能性。

如果文件在前面没有被打开，关闭文件将导致问题。因为我们已经对这个问题进行了处理，我们可以把这个问题和第一个 bug 报告给开发者，而不用担心关闭文件的问题。按键使程序继续运行也会出现问题。是不是每个键都能正确工作？在键盘上有很多可能的按键。除了标准的字符键，还有 Shift 键、功能键、Ctrl 键和 Alt 键，以及这些键的排列组合。不过，在这种情况下，这不是大问题。最有可能的测试是检查输入键（一些键盘上）和任何字符键是否正常工作。如果不正常工作，那么程序出现的 bug 就应当被修改，以满足用户所有可能的按键。我们没有检查按键和它们的有效性，所以，在这种情况下，要进行进一步的测试。

很多程序接受的另一种输入是图形形式，这是从用户那里收集到的输入。在这种情况下，最好的方法是把程序一边的输入变成可以工作的惟一有效数据。作为测试者，应当测试所发现不能工作的事，例如向数字记录输入字符或其他非法数据（例如 02/30/2000）。对一个输入类型可能出现无数的错误数据，所以你最好在要求数字的地方用字符测试。另外，还应当使用不同的修改键，例如 Shift、Alt 和 Ctrl 键。功能键也是一个好的选择，这样你就会经常发现那些不想让用户发现的功能。

可以知道，在软件开发中，程序的输入会引起很多的问题。和测试人员一同检查哪些输入是有效的，开发人员就可以发现所有的非法输入。虽然不是最好的实现系统方法，但这个方法可以使非法的数据进入系统，这是用户首先需要的。

13.4.4 算法测试

几乎每个程序都实现一个算法。一些算法非常简单——收集数据，并把它们相加。另一些算法却非常复杂，例如解码视频流。不过，在所有情况下，实现算法需要一个过程。如果这个过程和系统文档的要求相同，那么测试这个算法就是测试系统的一个主要部分。当然，不是测试真正意义上的算法，而是测试代码中算法的实现。

有时，作为一个测试者，你会发现检查系统的实现中的算法要使用一个特定测试用例。例如，一个执行二叉树查找的算法，你就要在没有可查找项的前提下测试：如果是某个排序算法，你就要在所有项都相同的情况下测试。

为了了解算法的测试，让我们看一个简单的算法，接着看这个算法的代码实现。我们将根据代码，而不是算法本身设计测试用例。

我们使用了一个简单的算法，但这个算法经常使用在计算机学科的课堂上。这个算法是计算阶乘的。阶乘的定义是把所有小于所给数的数相乘。也就是说，给定任何一个数 n ，它的阶乘是：

$$1*2*3*\dots*(n-1)*n$$

一个简单的例子，4 的阶乘是 $1*2*3*4=24$ 。

我们选择阶乘作为例子的原因是：在计算机科学中，阶乘是一个最好的递归的例子。为了实现阶乘，我们使用以下的算法：

```
If Number = 1
    Return 1
Else
    Return Number * Factorial(Number-1)
```

因为这个算法是递归的（也就是自己调用自己），它通过数字倒推，直到发现开始数字（1），然后每个数字相乘，直到达到起始点。

```
Function Factorial ( Number : Real ) : Real
Begin
    If Number <> 1 Then
        Return Number * Factorial ( Number-1 );
    Else
        Return 1;
End;
```

注意：以上这两个代码段都是伪代码段，而不是有效的 Pascal 语言。

作为开发者，读完代码之后，可能发现这是个完美的实现。但是在整个代码中有两个主要的地方。第一，我们要决定数字是否是 1；第二，我们把数和比它小 1 的数的阶乘相乘。可能会出现什么错误？

作为一个测试者，我们意识到这段代码和算法有很多问题。例如，代码中没有错误检查。当是负数时，会出现什么结果？在这个情况下，让我们来检查这个算法。例如，在计算 -4 的阶乘时，我们的计算方法如下：

$$-4*-5*-6*-7*\dots$$

可以看到，最后的结果不会出现 1，因为数字的计算方向不对。显然，这个测试用例会使程序出错，这个错误可能是堆栈溢出。计算 0 的阶乘也会出现相同的问题。在计算大数的阶乘时会出现什么问题？例如，可能要计算 10,000 的阶乘，将会发现在很多的计算机上执行都会出现系统崩溃，因为每个递归将会使堆栈的层次增多。结果，将会用完堆栈空间，程序崩溃。

可以发现，你不用为这些测试用例来查看这些代码。相反，只要查看代码的算法本身。这个算法即使在代码中绝对正确执行，也包含了等式本身固有的问题。通过理解算法的执行，我们知道代码怎样工作。这就使我们可以产生测试用例和执行结果，甚至可以使我们不用通过查看代码就可以预测程序的问题。

13.4.5 决策表

决策表是结果和决策的一个简单的交叉引用，如表 13.1 所示。

表 13.1 决策表的例子

	规则 1	规则 2	规则 3	规则 4
决策 1	True	True	False	False
决策 2	False	False	True	True
决策 3	True	False	True	False
决策 4	False	True	False	True
动作 1	Yes	Yes	No	No
动作 2	No	No	Yes	Yes

可以发现，这是一个矩形的决策表。在确定了条件的正确和/或错误之后，就可以知道某个动作是否会发生。使用决策表的好处是可以直接引出程序代码和测试条件。因为可以把代码写成一系列的条件语句，那么就可以随意从决策表中自动生成代码。同样，因为测试条件是正确和错误的，那么就可以从简单的矩阵中生成完全的测试程序组。

让我们看另一个简单的决策表例子，表 13.2，看怎样转换程序代码和测试用例。假如编写了一个自动汽车的程序。汽车在红灯时停下，绿灯时行走，黄灯时减速。另外，还必须有左右转弯的信号。我们几个导致决策的条件，和一些可以执行的结果。

可以看到，我们可以通过决策表直接编写代码。例如，第一行（红灯停）的代码如下：

```
if(LightIsRed())  
    StopCar();
```

同样的，这个条件的测试用例是：如果红灯亮，汽车是否停下？期待的结果是汽车将停下。

13.4.6 状态机分析

状态机的工作原理是通过机器内部给定的条件，机器从一个特定的状态转移到另一个特定状态。在很多系统中，整个大的系统和单个的功能特征可以很完备地映射到状态机上。

个简单的例子，考虑众所周知的 ISP (Internet Service Provider, Internet 服务提供者) 登录系统。首先显示一个对话框，然后连接网络，最后登录到系统中。登录的每个过程都是一个状态。在系统中，每个状态只能向两个方向移动。如果满足了状态转换的要求，那么就移向下一个状态。如果遇到状态转换条件失败，那么就回到初始状态或者到某一已知的错误状态。

表 13.2 汽车行驶的决策表

	红灯停	绿灯行	黄灯减速	行驶
问题				
是红灯?	True	False	False	False
是绿灯?	False	True	False	False
是黄灯?	False	False	True	False
汽车准备转弯?	False	False	False	True
动作				
停车	True	False	False	False
开车	False	True	True	True
减速	False	False	True	False
转弯	False	False	False	True

怎样理解机器的状态可用来帮助我们进行测试和调试? 像逻辑表一样，在某些情况下，把测试用例和程序直接转换成机器状态。如果知道了进入一个状态的前提条件和离开这个状态的后置条件，就可以通过简单设置这些条件或部分条件来测试程序的行为是否正确，并观察在给定的条件下程序是否正确运行。

让我们看一个状态改变的例子，来更好理解怎样测试引起状态改变的信息。在卫星电视系统中，用户经过远程输入新的数字来改变频道。如果这个输入频道是有效的，那么电视就自动跳到这个频道上，你就可以观看这个节目。不过，在显示器背后要考虑很多复杂的逻辑过程。

- ◆ 状态 1: 观看初始频道的节目。如果输入队列中有输入，到状态 2。
- ◆ 状态 2: 输入是否来自用户? 如果是，到状态 3; 否则，回到状态 1。
- ◆ 状态 3: 用户是否要求改变频道? 如果不是，到状态 4; 否则到状态 7。
- ◆ 状态 4: 用户是否要求增加音量? 如果是，到状态 5; 否则回到状态 1。
- ◆ 状态 5: 用户要求的频道是否为有效频道? 如果是，到状态 6; 否则回到状态 1。
- ◆ 状态 6: 用户要求的频道是否已被授权? 如果是，到状态 7; 否则回到状态 1。
- ◆ 状态 7: 转到用户要求的频道上。如果不能完成，显示错误信息，回到初始频道，回到状态 1。
- ◆ 状态 8: 转到新频道。回到状态 1。

一个状态改变到另一个状态都是有原因的。在这个例子中，有 7 个离散的状态和 7 个状态转化的原因。此外，通过读状态转变列表，我们就可以用简单的测试来验证系统是否正常工作。例如，我们可以输入合法和非法的频道；可以输入授权和非授权的频道；可以在输入频道中间按下音量控制来观察系统的动作。

状态机技术是理解程序如何从一个模式转变到另一个模式的有效技术，这一技术对有不同特定状态的程序特别有效，例如不同的显示或不同的输入模式。如果系统有一系列的显示允许用户完成一个任务，那么这一系列的显示就是状态机。通过理解从一个显示转变到另一个显示的信息，就容易设计出满足或不满足这些条件的测试程序组验证代码的功能。更为重要的是，还可以设计测试程序来验证代码不会出现预料之外的功能。毕竟，这个预防错误的思想是进行测试的理论基础。

13.4.7 综合测试

作为一个开发者，通常会在完成模块的编写、调试、单元测试、文档和对它进行配置管理之后就认为工作已经完成了。这个想法是不正确的，在模块单元测试完成之后开始的综合测试，像它的名字一样，综合测试是对模块的接口进行的测试，也测试了因为不正确的通信或错误的测试引起的模块间的缝隙。

综合测试是如何进行的？让我们假设有两个模块，A 和 B，这两个模块执行了两个功能。假设模块 A 从用户得到了一定格式的输入。模块 B 负责在现有的存储设备中保存输入，例如数据库。程序员负责每个模块的充分测试。模块 A 收集正确的数据，不允许出现错误的数据，并确保每个都被填充。模块 B 接受包含数据的输入对象，并把它写到数据库中。模块 B 已经被测试保证了在正确的数据库行中正确写入字段，在没有捕获和检查的数据库写操作中也不会出现错误，所有这些可能的错误转变成模块可接受的规范的格式。

以上这些描述就能说明这两个模块正确工作吗？一旦模块在配置管理系统中被检查，并被完全编译，我们就可以认为这些模块完全正确吗？当然不能，在两个模块的连接处可能出现很多的问题。例如，假设模块 B 确认从模块 A 得到的数据格式，如果模块 A 假设模块 B 不会返回错误的代码，但会出现一个异常，情况会怎样呢？所有这些可能的问题出现在两个模块接口处。

通过观察一个现实世界的例子比给一个代码的例子更容易理解综合测试。一个最基本的综合问题的例子是 1999 年火星气象卫星的坠毁。

火星气象卫星在 1998 年 12 月 11 日离开地球，9 个月后到达了火星。遗憾的是，它在火星表面消失了，从这之后就再没有有关的消息。事故的事后分析（在第 2 章中描述）表明不是软件的故障，而是两个系统间的一个简单的错误通信引起的。在检测软件模块和任务控制软件之间的接口除了一个小细节外，其他都正常工作。控制软件用英语单元发送测量信息，例如速度和距离。相反，检测软件假设接受到的信息是度量单位。据 NASA 报道，这个接口的问题引起太空飞船在到达火星表面时速度时快时慢，最后导致崩溃。

这也是一个 bug 吗？很多人都认为太空飞船的崩溃是一个严重的问题。因为这个事故是由软件引起的，你可能会疑问为什么软件的单元测试没有发现这个可能的问题。答案很简单，地球上没有发现这个问题的单元测试。问题的原因是这个问题不出在被测试的特定的单元模块，而是出现在两个模块之间的接口。如果进行了综合测试，比较两个模块的单元输出，这个问题就能被发现和解决，就避免了几十亿美元的损失。不用说，这样的接口测试可以作为工程项目故障原因的事后评估。

13.4.8 自上向下测试和自下向上测试

虽然不是真正的测试技术，自上向下测试和自下向上测试的问题经常在软件生命周期的开发、调试/测试阶段出现。自下向上测试的思想是：从系统的底层模块开始测试，直到验证了每一层都能正确工作。另一方面，自上向下测试的思想是：从系统的顶层模块开始，通过适当的测试用例，一层层向底层模块进行测试工作。这两个测试方法各有优缺点。

在自下向上测试中，先开始系统基础模块的测试，然后对使用这些基本模块的模块进行测试。在这一方面，你可以发现很多的 bug，因为你测试的部分已经开始了工作。自下向上测试的问题是它不是从用户的观点进行测试的。既然这样，虽然可以使用自下向上测试法发现更多的底层问题，你也会错过从用户角度出发的更重要的错误。这主要是因为测试的时间有限，因为从底层开始意味着在顶层的测试时间就较少。

在自上向下测试中，可能会发现用户遇到的问题，因为你是从用户角度工作的。自上向下测试用例通常是直接从说明书和用户角度编写的，这样我们就很容易理解预测输出。自上向下测试的问题是发现不了系统中底层的危险 bug。这些 bug 通常会引发很多最终用户系统问题。

当然，理论上可以混合使用这两个测试技术，以保证最大范围覆盖系统，并保证能发现用户可能遇到的主要问题。遗憾的是，事实通常陷入这样的解决方法。你用于测试工作的时间是有限的，这些时间由于要在系统中增加新的特征和没有发现的 bug 而缩短。因为为了解决一个 bug 而做代码的修改会使得测试过程重新开始，所发现的每个 bug 减少了可以测试的新 bug 的数目，所以要求时间来解决发现的 bug。因为这个原因，最好的方法不是这两个测试方法各占一半，而是要有所侧重。

测试系统的一个最好的建议是：使用自上向下测试和自下向上测试方法来测试系统。因为应当为系统的每个模块编写单元测试程序，就可以使用这些测试用例，在完成用户接口前对系统的底层部分进行测试。这就可以使你在“准备测试”前的很长一段时间就可以真正开始测试系统。另外，可以使测试者更好地理解系统的实现和局限。

一旦系统的接口层已经完成，并提供了它应有的性能，那么就可以开始自上向下测试。虽然这两个测试方法不会在程序的中间部分接上，但你可以确保在这两个范围的结尾处有很多的测试。这个方法将会捕获用户可能遇到的大量的问题。在 Ed Yourdan 提到的“足够好的软件”的建议中，如果我们在保证用户大部分时间是幸福的前提下而让他们痛苦一点时间，

那么我们的产品将是成功的。

记住，测试不会发现所有的 bug，不论是从理论的角度，还是从实际的角度都不可能做到这一点。另外，我们试图发现的 bug 会受所用户的使用经验干扰，只有这些 bug 在产品中变得产品完全不行时，才会去解决它们。

13.4.9 配置调试

不同的配置是开发者和测试者遇到的共同障碍。不同的操作系统、不同的 DLL 版本、不同的程序运行使配置系统看起来好像用户不可能完成。测试配置的工作在开发阶段落在测试者身上。由于这个原因，在寻找 bug 时，就应当保留所使用的配置信息。

几年前，我在一个为 CD-ROM 工业开发软件的公司工作。这个软件本身运行在 MS-DOS、Windows 3.1，Windows 95 和一些变型的 PC-Unix 上。当一个 bug 出现在我们的开发中时，通常不知道这个 bug 是出现在哪个操作系统的版本上。这就导致了开发者不好的感觉，当他们在错误的操作系统上进行测试时，他们无法再发现 bug。

最后，我们创建一个简单的 bug 报告形式。在运行期间，我们定义一个新的、独一无二的可能影响我们系统不同点。例如，在 MS-DOS 操作系统中，打印是一个严重的问题，因为在 MS-DOS 环境中没有一个标准的和打印机通信的方法，不同的打印机会导致不同的能力和不同的问题。

在测试一个系统时，应当记录运行系统的配置。这包括操作系统、计算机类型（厂商和类型，例如是便携式电脑或台式电脑）、所使用的系统 DLL 的版本和其他任何可能在所使用程序运行的应用系统。所有这些信息对那些试图发现为什么能在他们系统运行完好，而在其他系统不能工作的系统的问题的调试者是至关重要的。另外，还要注意系统安装的驱动器和目录以及安装在应用系统之后的其他应用程序。安装问题的其他例子包括特定的 DLL 版本、系统的配置文件、注册文件或 INI 文件的修改，或在 Unix 环境下账户的访问特权。

你可能会认为观察信息并记录它是几乎不可能的事，因为在安装时很难知道系统发生了哪些改变。不过，这是不正确的。通过安装程序或系统本身可以知道需要运行什么，这将是基本系统文档的一部分。另外，系统的信息通常可以自动检索。例如，在 Unix 中，可以通过 where 命令来检查系统文件的日期和数据。在 Microsoft 的 Windows 操作系统中，可以通过操作系统提供的标准的 msconfig 程序来发现这些信息。你的安装日志必须包含修改内容的信息。另外，像第 9 章提到的，你的系统必须能够显示所使用的不同系统文件的位置和版本。

关于配置测试的最后一个问题是程序设置。不像系统设置，程序设置对特定的程序本身是可选的配置信息。和系统的设置一样的是，应当能够显示在任何给定的时刻的所有设置信息。信息的副本应当连接所有的 bug 报告，这些信息使程序者更容易发现系统是否有故障或者是否是在设置领域中做了你要求做的事。

在下一节将看到一个设置问题的简单例子。在这个例子中，我们将使用一个初始化文件

来存储程序的信息。

13.4.10 恢复崩溃和掉电测试

在掉电时系统会出现什么情况？VCR 在 12:00 时开始闪烁，报警时钟失去了时间并重设了唤醒时间。指示灯熄灭并重新点亮。但在断电时，你的程序会出现什么问题呢？如果程序允许用户创建数据，在系统出现异常掉电时，数据是否会丢失？在系统运行时，如果设计被修改，是否会得到保存？你必须在开始测试系统前就要知道这些问题。

在测试一个恢复崩溃的系统前，要知道设计什么功能。例如，如果系统只是简单处理在掉电时存储器数据的丢失，那么测试这个系统将变得容易。拔开插座，观察系统的行为。遗憾的是，只有极少数的系统保留了整块的存储器的数据。有些系统把数据保存在磁盘上，也可能把现场状态信息保存在这些磁盘上。像这样的系统更需要进行恢复崩溃和掉电测试。

另一个极端是把程序设计成恢复和重启掉电时的状态。这些类型系统的例子有嵌入式软件、操作系统和监视软件。在每个情况下，你必须知道系统的每一个不同的状态，还要知道怎样对掉电进行测试。让我们看下面一段伪代码和我们可以用来验证在掉电情况下系统仍正常工作的测试用例。

```
Startup: Check for power failure.  
If failure detected (no shut down), reload  
current state.  
Initialization: Clear memory, load machine list  
into memory from disk.  
Monitor Phase: For each machine in machine list,  
check current status. Write status to status  
registers for that machine. Save state.  
Error Phase: If a machine reports an error, print  
out status message to console. Update current  
machine state. Save state.  
Save State: Write all machine registers to disk.  
Add Machine: Add a new machine to the memory list.  
Delete Machine: Remove a machine from the memory list  
if it exists.  
Shut Down: Save state.
```

为了进行恢复崩溃和掉电测试，要求知道机器所处的状态，每个状态定义了一个特定的信息段并为系统保存起来。必须知道系统保持的信息和信息保留的地方，在磁盘上还是在存储器中。从以上这个例子中，可以立即看到以下一些测试用例：

- ◆ 在重启系统时立刻创建掉电条件。
- ◆ 在初始化后立刻创建掉电条件。
- ◆ 在监视阶段开始时创建掉电条件。
- ◆ 在监视阶段中间时刻创建掉电条件。
- ◆ 在监视阶段完成之后立刻创建掉电条件。

- ◆ 在错误条件报告前立刻创建掉电条件。
- ◆ 在错误条件报告前立刻创建掉电条件。

有更多的测试用例，以上这个列表给出了一个好的开始。在每个测试用例中，我们创建了一个掉电条件，接着重启系统。在系统重启时我们会发现什么？

1. 系统正确运行。
2. 系统仍然有序，不会出现不知道的状态。
3. 数据被安全处理，也就是，对于每个所给过程的数据都是可用的。
4. 增加、删除和修改被保存，并在系统重启时重新载入内存。

从上面的一段伪代码中可以看到，出现问题的系统很可能不满足条件 4。在进行增加和删除操作时，数据不会立即写入磁盘。这不是一个严重的问题，但它仍然满足条件 3 的安全处理，不过，这可能是一个 bug。

提示：总是要对系统进行掉电测试，确保它正常启动。

在计算机领域中，掉电是常见的问题。即使使用了不间断电源（Universal Power Supply, UPS）和存储器的后备电池，也将会看到计算机经常会经常出现意外掉电现象。如果系统没有进行这种测试，就可能会给用户带来严重的产品问题。

13.4.11 安全性测试

安全性测试意味着只有授权的人在需要的时候才能进入系统，而非授权的人因为不被允许就不能进入系统。虽然安全可能只包括登录名和密码，也可以有使用的机器名、IP 地址、机器“dongles”和其他一些特征。这些安全的机制，对开发者实现某些系统的安全特征是很重要的。那么，对测试者验证系统的安全性有什么重要性吗？

测试安全性是一个很重要的、很不被注意的系统测试。安全性对系统的用户有十分重要的作用，然而开发者和设计者通常很少注意到它。虽然系统可能有“安全”登录和密码设置，但如果这些信息以文本的格式储存在硬盘上有什么好处呢？如果能够通过重启系统并获得这些安全信息，系统是否是安全的呢？所有这些问题必须由安全测试来回答。

安全测试有两个形式。第一，测试者通过挫败安全系统或者通过使用一些自动工具随机生成密码来登录系统，以达到进入系统的目的。第二种方法是使用后门（backdoor）或程序错误（例如缓存区溢出）来进入系统。安全的缺点可以定义为成功地获得访问权或错误的消息包含了太多系统的信息。例如，看下面一段登录系统的描述：

```
System: Enter user name:
Tester: fred
System: User name "fred" not found in database. Oracle error 0x87654
```

从安全性测试的观点来看，我们从这一段描述中可以发现什么问题呢？我们不管句了语法上的问题，而只关心安全问题。为什么系统告诉我们“fred”不是一个合法的用户？这样做，就为黑客攻击系统提供了方便。不用试图猜测用户名和密码，你很容易使他们知道用户

名是非法的，那么他们就可以修改用户名并使用另一个登录名。如果在给出错误信息前就要求用户名和密码，而没有给出哪一条信息是不正确的错误信息，那么就可以增大一些人破坏系统的难度。

既然给出了用户名部分不正确，还有其他错误吗？下一个逻辑问题是它告诉了我 Oracle 的错误信息，这有两个不好的地方。第一，通过知道所使用的数据库系统，就给黑客一个发现数据库系统后门的机会。很多数据库为系统的管理者提供了一个缺省的登录名和密码，你可以使用没有修改过的缺省设置来登录系统。因为我们可以从提示中得到数据库系统信息，就为黑客提供了一个容易进入系统的方法。第二个原因不很明显。通过给出所使用的用户的详细的错误信息，就存在了显示系统的更多信息的危险。例如，如果错误信息显示了数据库的内容，那么黑客就有足够的信息绕过安全系统并得到没有适当进行错误处理的页面。在这一点上，黑客就可以完全获得进入系统的权力。

在完成安全性测试的讨论之前，要强调的是大部分安全问题是系统超载，或者是给过程发送缓存太大以至系统不能处理等原因引起的。可以通过任何的客户端/服务器系统的实验，来发现系统中的这些问题。有很多标准的工具，用给系统发送不适当或非法的消息，来观察系统是否正确处理。

13.4.12 第三方测试

试着思考一下，如何测试完全没有文档和没有系统工作原理说明的系统。总的来说，这个系统的测试要求使用第三方工具。不仅仅可以发现自己系统中的所有 bug，也可以发现其他系统的所有 bug，就像在调试一个没有任何用户手册的操作系统一样。测试第三方工具对任何测试者来说都是最难的工作。

有两个基本的第三方工具测试方法。第一个方法是为工具创建一个特定的测试程序，并在你要创建和测试的系统的上下文之外测试它。在这种情况下，你验证了工具是否按照希望的方法工作，不会出现任何的意外。第二个方法是把这个工具作为系统完整的一小部分，然后编写测试用例以便在你的系统中最好地检查这个工具。在这种情况下，例如，你可能会担心这个工具怎样被你的代码调用，而不是它怎样处理问题。在第一种情况下，我们把第三方工具当成系统中一个独立的过程；在第二种情况下，我们把第三方工具看成是主系统中的一个子系统。

很难提供一个比较好的测试第三方工具的建议使用在系统中，除非你需要这样做。如果不是这样的话，开发者将会责备这个第三方工具，不管这个工具是否真正有问题。更糟的是，管理部门也将不相信这个工具（因为工具不被任何人相信），将用另一个构件来替代它而不去发现真正的问题。

在调试时要提到的是，如果你怀疑第三方工具存在问题，你应当编写一个简单的测试用例来说明这个问题。如果这个测试用例不能发现问题，那么就要假设问题出在你自己的代码中。一般来说，第三方工具出现的问题并不比自己编写的代码中的问题少，你就要注意这一

容易解决问题的部分。如果问题存在你自己的代码中，你就可以解决它；如果问题出在第三方工具中，那么直到这个工具的下一版本公布你才能有望更好解决它。然而有些软件，可以让你在第三方工具中解决问题，这是有困难的。如果你在工具中解决问题，这一个解决的问题可能还会出现在下一版本中，如果下一个版本中的这个问题没有解决，你就不得不重新来解决这个问题。

13.4.13 多用户测试

如果系统在同一时刻支持多用户，这就要求你在同一时刻有多用户来测试系统。虽然这看起来很顺理成章，可是一个人在同一时刻怎样来测试多个系统呢？

在考虑多用户测试时，必须运行几个重要的测试。第一，必须考虑在几乎相同的时间里用户做了几乎相同的事；第二，必须考虑在几乎相同的时间里用户做了几乎相反的事；第三，考虑在一段间隔里，两个用户做了相同的事。你能够而且应该在超过两个用户时，测试所有这些情况。事实上，还应当测试期望用户负载的系统。在讨论性能测试时，应当考虑到这么一个事实，那就是要在系统用户数超载的情况下对系统进行测试。

比起单模式或批处理模式的系统来，多用户系统更可能出现一些问题。例如再登录问题、数据库问题和资源共享问题。所以测试用例不仅要考虑到这些系统中的“正常”问题，还要考虑到可能发生的多用户问题。让我们看下面一个简单的例子。

假设在企业网上运行一个简单的记账系统，这个记账系统使用数据库来记录账目和配置文件，来跟踪需要考虑的账目。登录和密码信息也保存在数据库中。这个系统为不同的账户提供了借款和存款的输入，它允许多用户在同一时刻访问相同的账户。修改账目应当在每个使用这个账目的账户出现时通知他。

我们要对多用户系统的不同需求进行哪些类型的测试？我们列出以下一些内容。

- ◆ 用户1向账目1提交一个存款，用户2向账目1提交一个存款。出现的结果是账目1包含两个存款。

- ◆ 用户1向账目1提交一个存款，用户2向账目1提交一个取款。出现的结果是账目1的取款应当小于存款加余款。

- ◆ 用户1向账目1提交一个存款，用户1向账目1提交一个取款，用户1向账目1提交另一个存款。用户2向账目1提交一个存款，用户2向账目1提交一个取款，用户2向账目1提交另一个存款。出现的结果是余款加上所有的存款大于所有的取款。

- ◆ 用户1向账目1提交一个存款，用户2向账目1提交一个存款。用户1向账目1提交一个取款。在这种情况下，出现的结果是两个存款小于存款加余款。

我们所看的这些情况是完整的交易。因为多个人可以在相同的时间访问相同的账目，我们要保证账目保持正确的交易。不过，这还不够，我们还要测试再登录问题。我们怎样做呢？最简单的办法是让两个人向一个账目提交存款，并在同一时刻按回车键，提交这个请求。如果能正常工作，我们就接着用三个人测试，四个人测试。最后用多少人才足够呢？应当是你

所希望能同时在系统工作的人数。当然，用户并不一定就是人，如果运行的测试提前被理解了，那么就可以用自动的过程来测试系统。

13.4.14 负载和性能测试

当你的管理者要求你测试一个产品系统，通常意味着要测试负载和性能。虽然这些测试方法在决定系统将来是否能被成功使用特别有用，但不是验证系统适当工作的最好方法。负载测试用来发现系统在完全崩溃前能处理多少的用户和过程。性能测试是测试系统性能随着时间和使用的变化情况。

当要对系统进行负载测试时，就要确定系统的瓶颈。例如，如果要对单用户排序系统的负载性能进行测试，那么系统的瓶颈可能是可使用的存储器容量、可使用的磁盘空间和总的参加排序的数据。另一方面，如果要对一个 Web 应用程序进行负载测试，系统的瓶颈是连接到站点的用户数。不知道系统性能的瓶颈，就无法进行正确的负载测试。

负载测试的基本方法是从一个合理的负载数字开始，然后加倍直到系统死机。接着，减小这个负载数目，直到系统重新开始工作。这个结果就是系统在这一性能上的最高负载。在每个性能上使用这个技术，直到知道了所有可能的系统性能瓶颈。遗憾的是，你不会这样做，你只会知道单个问题。下一步就要开始综合考虑这些功能，观察瓶颈是否改变。因为瓶颈的改变，你就可以知道这两个方面是有联系的。所有这些相关联的方面可以当成一个集合。注意这些方面可能会受磁盘空间、存储器大小和 CPU 的速度影响。一旦你知道了这个范围，下一步就要开始增加系统的参数（存储器、磁盘空间等等）并确定哪些因素可能会影响到瓶颈。如果系统参数对瓶颈没有影响，那么就没有必要增加参数。

负载测试的结果数据应当包含以下一些信息：可以运行系统期望的用户数的，影响系统性能的最小系统配置和最大系统配置。例如，把存储器容量增加到 256MB 会提高系统速度，但增加到 257MB 就不会提高系统速度，那么当存储器容量达到 256MB 时，就没有必要再增加了，这就是系统的最大存储器容量。你还要发现最理想的系统大小，这就是使用最合理的软硬件配置达到最可以接受的系统速度和性能。你还应当知道系统的最大限制和系统的瓶颈是一样的。如果工作在 Internet Web 系统中，并且后端数据库只能同时处理 100 个用户，那么你就无法增加这个限制。除非增加了另一个数据库系统，否则系统能同时处理的用户数最多只能是 100。

性能测试和负载测试有点不同。在性能测试中，是从系统固定的配置开始的，然后通过测量来观察系统的性能的好坏。你使用的标准会因为系统的不同而不同，和你系统的重要性有关系。不过，总的来说，你要注意执行不同动作的时间、在这些动作中存储器的使用情况和执行这些动作要求的总的存储空间和可能保留的结果。

让我们看一个单机系统的性能测试的简单例子。在这个例子中，我们将考虑一个执行打字过程的程序。对这样系统的性能测量，我们要考虑哪些方面呢？显然，系统对按键的反应速度是要重点考虑的方面。如果这个过程执行了后台拼写检查，那么要考虑这个函数所占用

的系统时间。系统运行时所消耗的存储容量也是要重点考虑的因素。而且，包含被处理文档的文件大小也是要重点考虑的因素，这样用户就不会因为整个大的文件而消耗完他们的硬盘空间。

在很多项目的开发结束时都要进行性能测试。遗憾的是，如果测试在软件中发现了问题，那么就不能做到这一点了。更可以接受的方法是在开发过程中和单元测试一起进行性能测试。同样，在系统的每个模块中较容易做到负载测试来发现系统的瓶颈。一旦知道了系统中这些模块的公共特征瓶颈，那么为了提高速度或者系统同时工作的用户数，就知道要改变系统哪些条件。

提示：在开发过程中进行性能和负载测试，而不要在项目结束时进行。

最后要提到的是，不要认为性能和负载测试可以替代系统的测试和验证。性能测试为系统的修改作出计划并提高关键的部分，不是用来对系统进行任何功能测试。负载测试是用来发现系统功能的。虽然负载测试能发现系统关键性能需求的 bug，但并不是真正意义上的测试。

13.4.15 测量和统计

当我们讨论 bug、测试或者其他软件质量的测量时，测量所使用的标准和统计问题总是被忽略。度量标准是我们在测试软件时取得的数字。可以测量在软件中发现 bug 的数目、软件中模块的数目、甚至每个模块的代码行数。可以测量每个模块的复杂性、每个函数和方法的参数个数和在源代码中注释的行数。并不是所有这些数字直接反应软件中的 bug 概率，但和出现 bug 的概率有很多联系。

在这本书中认为，软件中 bug 数目和软件的复杂性有直接的联系。测量和统计是证明这个联系的最好方法。以下列出的一些测量是你在开发应用系统应当考虑的问题。

- ◆ bug——发现的时间和数据、bug 发生的模块、发现 bug 的人、负责解决 bug 的人、解决时间和解决 bug 的人。
- ◆ 模块——复杂度、代码的行数、创建时间、创建者。
- ◆ 屏幕——屏幕上发现 bug 的数目、屏蔽上的对象个数、支持这个屏蔽的代码模块个数。
- ◆ 数据库——行数、列数、每天增加的记录、错误发现数据的数目。
- ◆ 性能——用户数、每个请求的平均处理时间、请求数目。

显然，这些标准对你的系统来说是不够的。不过，记住这些标准并使用它们是有用的。如果你跟踪每一个标准，但从来不使用这些信息，就会发现这些标准的数据毫无价值，提供这些信息也是没有意义的。

13.5 小结

在这一章中，总体上知道了测试对系统开发和对调试者的重要性。通过理解测试技术的

基本原理,就可以使自己成为了一个较好的调试者和较好的开发者。另外,通过理解测试者在测试一个系统的过程,你对如何发现一个报告给你的问题有了较好的认识。

像调试一样,测试不是科学,而是一门艺术。通过对你使用的技术的理解,并在测试中使用更系统、更可测量的方法,就可以把测试应用在科学领域中。在调试工作中使用测试技术,同样也可以把调试技术应用到科学领域中。

13.6 bug 问题

C 语言的预处理是 C 语言程序者和调试者的一个障碍。所以很多由预处理引起的 bug 几乎可以用一本书来说明。让我们通过解释预处理如何工作来看一个著名的预处理 bug 问题。

假设你写了一个简单的产生定义在一定范围内随机数的宏。C 语言程序库提供了一个名字为 `random()` 的函数产生随机数。这个函数的问题是产生的随机数不在你想要的范围内。所以,作为一个聪明的程序者,写一个简单的宏来解决这个问题:

```
#define Random(n) random()%n
```

这个宏使函数产生的随机数的结果在 0 到 $n-1$ 的范围内。现在,当你需要一个特定范围的随机数时,就可以使用 `Random` 替代 `random` 来产生。这使用起来就比较简单了。可以写一个简单的语句如下:

```
int ran=Random(j-i+1);
```

那么不管 j 和 i 是什么数都会产生一个在 0 到 $j-i+1$ 之间的随机数。很多程序员,即使是 C 语言的程序员,读了这一行代码之后就知道你要实现什么功能。对那些不理解的人,就需要我们去解释。假设在一个循环中, i 等于 10, j 等于 20, 这个语句将产生一个 0 到 11 之间的随机数。

让我们花点时间来回顾一下这个预处理语句的真正工作过程。当你编译 C 语言源代码时,这个预处理语句是在最后产品的代码中扩展的预定义常量,包括文件和像你所写的宏的解释。例如,当你编写如下的常量定义语句:

```
#define kMaxValues 10
```

然后在某些代码中使用这个常量定义:

```
for (int I=0; I<kMaxValues; ++I)
    do_something(I);
```

这个预处理将在代码中运行,并扩展成一系列的符号。在这个例子中,扩展的结果是:

```
for (int I=0; I<10; ++I)
    do_something(I);
```

预处理语句不是非常“聪明”的,它只是挨个字符地进行替代。所以,如果你出现了一个错误,例如:

```
#define kMaxValues "10"
for (int I=0; I<kMaxValues; ++I)
```

```
do_something(i);
```

那么预处理语句扩展的结果就不正确了，编译器（在预处理之后运行）就会碰到这个问题。有时，因为符号替换过程使编译器不够清楚。不过，结果总是按照用户的要求进行文字符号的替换。所以，当使用 **Random** 宏进行编程时，你就会在最后的代码中得到以下的替换：

```
int ran=rand () % j-i+1;
```

遗憾的是，这个宏不能工作。你的任务是发现这个问题的原因，并解决它。

第 14 章 维护

在软件开发中，维护是一个很讨厌的名词。开发者通常离开公司，因为这个公司的开发已经进入了维护阶段，没有新的开发工作。而且，维护在软件开发过程中是一个很昂贵的阶段，估计要花费公司的 60% 到 80% 的软件开发费用，而且维护是软件产品开发的主要工作。

维护怎样影响到 bug？在金融领域的软件维护调查中发现只有 35.1% 的因为维护所作的修改不会产生新的错误。在这个研究中发现，平均有 16.2% 的维护任务引入了超过 15 个的错误。这并不奇怪，因为维护要修改系统，在第 3 章中提到，在要修改系统时会引入大部分的 bug。

14.1 什么是软件维护

IEEE 定义的软件维护是：在软件递交（给用户）后对软件产品的修改，以改正错误、提高性能或其他属性，或者使软件产品适应可修改的环境。

维护任务通常被分成以下三类：纠正、适应和完善。

- ◆ 纠正——解决 bug，这就是纠正任务的名字意思。维护者的责任是确定 bug 的原因并解决它。

- ◆ 适应——适应任务的目标就是为不同的系统替换系统的功能模块。维护者可能需要在不同的硬件平台、不同的操作系统或不同的编程语言上使用软件。

- ◆ 完善——完善任务的目标就是要提高系统的性能。维护者的任务是增加一个新的特征、提高效率、改进功能、改进系统设计并实现其他和改进相关的任务。

一些维护任务的分类也包括预防维护。预防维护就是要预防软件系统中可预见的错误和问题。不过，我们认为预防维护包含在完善维护或纠正维护的分类中。为预防 Y2K bug 所做的工作是预防维护最明显的例子。

根据我们的观察，一个公司维护工作的典型工作是在维护阶段达到系统的完备性。在维护的早期阶段，完成纠正维护工作，这是因为软件还没有被完全地使用。慢慢地，到软件的成熟阶段，开始完善维护任务。在这个阶段中，发现了很多的 bug，公司开始接受需求改变并注意到系统性能的提高，这样系统就随着时间发展。另外，如果系统变得成熟，并成为公司的核心部分，这样，会使用在另一个环境中。这样，使用适应维护这一最后的技术来实现这个系统，或者使用适应维护把系统移植到更流行的软件平台，运行在最近的操作系统上。

14.1.1 维护工作

典型的，一个完整的维护工作包含了四个阶段：需求、理解、修改和验证。

步骤 1：需求

维护任务通常是从修改的需求开始的。这个需求可以来自于任何地方，在很多情况下，真正的问题并不是需求来自于什么地方。如果维护的需求是出现 bug，我们就要重现 bug；如果是为了增加一个新的特征，我们就要理解需求的本质并获得需求。理解需求本质的过程可能是几分钟时间（因为很容易重现 bug），也可能是几天（获得和理解需求），甚至几个星期（当不可能重现 bug 时）。

步骤 2：理解

一旦我们对需求有了理解，我们接着就应当确定怎样实现需求。这意味了我们需要确定我们要在软件哪些地方做出修改，这些修改对软件的其他部分有什么影响。我们也要理解这个程序。这个步骤要花费多少时间是由程序的大小、理解的容易程度和是否有文档或初始开发者来回答一些问题决定的。维护的大部分时间是花费在这个阶段上。一般情况下，程序员大约花费 47% 到 62% 的时间在理解文档和程序的逻辑结构上。

步骤 3：修改

如果我们知道怎样进行修改，我们就可以进行软件的实际修改。这是一个最容易的步骤，因为在这点上，维护者已经清楚他们要做什么，所做的工作是将要做什么修改并确定这些修改对软件的影响。如果你不能确信修改的作用，就要重复步骤 2。

步骤 4：验证

接下来我们需要确认我们所做的修改是否达到预期的功能，我们也需要确认不会影响到修改之外的其他部分，这是维护任务的最重要一部分。

14.1.2 维护的挑战

根据我们的经验，维护是任何软件工作中最难的一部分，但维护者是软件工程中未被称赞的英雄。一方面，维护要求对系统进行修改；另一方面，又不允许会对系统其他部分产生影响，因为维护通常是在产品环境中进行的。维护人员通常不是软件的初始开发者，他们是在软件制造之后才被雇佣的。他们对存在的错误、不好的接口或对不灵活和不可扩展的设计一无所知。然而，维护者经常会因为系统的 bug 受到管理者和用户的责备。这是一个没有好处的

工作。引起维护工作的困难主要有以下五个原因。第一，在软件工业中，人员的流动率很高，这就意味着经常有新的开发人员参加他们从来都不熟悉的软件开发。第二，软件系统会变得复杂，这就引起软件理解的困难，特别是在软件文档丢失或出现错误的情况下。即使这些文档存在，也通常不起作用，因为它没有回答真正的问题。此外，因为软件理解的困难，维护者确定所有需要修改的地方和这些修改的作用通常是比较困难的。同时，成品环境下的软件

使维护更加困难，因为很难做不相容的修改。显然，不好的软件设计也会使软件的维护变得困难。

14.1.3 软件的退化需要软件维护

维护的困难意味着维护者在进行软件修改时会引起错误。这些错误可分成以下三类：

- ◆ 明显错误
- ◆ 潜在错误
- ◆ 概念错误

有些错误是明显的，这些错误在单元测试和回归测试就会显示出来。其他一些错误不明显，这就是潜在错误。在第 3 章中给出了潜在错误的发现方法。例如，程序中的一个不正确的错误条件处理就是程序的错误。虽然这些类型的错误不会影响到程序的执行，但它们有可能变成 bug。一个关于维护的研究表明 34% 的程序修改包含了错误，5/8 的相同修改至少包含一个潜在的错误。

最后一种错误类型更可怕。概念错误不会终止程序，它们也不是错误。这些错误每次都会引起软件的设计的退化。这些类型层次是轻微的修改，并且这些类型不再是可重用或可扩展的。随着软件变得越来越严格和脆弱，在结构和函数中就添加新的依赖关系。随着软件使用时间的增长，系统的初始设计出现退化，并且出现漏洞。所有原来是好的设计慢慢退化了，我们也就无法进行系统维护。

14.2 创建一个可维护的软件系统

什么是可维护的软件系统？从我们的观点来看，维护是在成品环境中修改软件的一个简单的过程，是为了解决 bug，增加新的特征，或者提高软件的功能，维护就是修改。如果维护就是修改，那么一个可维护软件系统就是不论在怎样的情况下都能被容易修改。如果发现 bug，那么就应当定位并修改这个错误。如果需求被修改，那么就on应该能够直接在现有的系统中增加和修改特征。但什么样的软件系统才能容易做到这些修改呢？

让我们回到前面，回想一下前面一节中提到的维护工作的四个过程。步骤 1 要求理解修改需求并不会关系到软件系统。步骤 2 包含了对软件的理解和修改。所以，软件需要容易理解和容易修改。步骤 3 实现了修改并通常不会影响到软件的维护性，因为编译器和源代码编辑器是用来实现修改的基本工具。步骤 4 验证修改并应当包含回归测试。如果回归测试是完全并有效的，就应当有更多的修改，这使维护的任务验证更容易。

14.2.1 创建可理解的软件

因为维护者花费维护的大部分时间来理解软件，那么就可以说如果软件容易理解，软件也就更容易维护。怎样才能使软件更容易理解呢？显然，文档是一个重要部分；容易读源代

码是另一个方面。但在我们确定一个可理解软件目标前，让我们看一下维护者怎样学习他们维护的软件。

程序可理解策略

假如你是一个刚参加软件开发团队的开发者。在工作的第一天，项目管理者就要你修改一个 bug。你带着这个问题到配置管理库中，这里只有极为有限的文档，并且系统的初始开发者都已经不在了。你怎样学习你将要进行修改的软件呢？

一个方法可能是从文件的源代码中开始查找。这个方法就是系统策略。系统策略的目标就是理解原始开发者的设计。为了使系统策略运行起来，你要读并理解代码的主要部分。如果软件的基本代码太大，这个方法可能不可行。所以不是去理解这个基本代码，而只要理解需要你修改的模块或部分代码。这就是按需要方法。在这个方法中，最小化了需要读的代码。你只要理解需要维护的那部分代码，就可以满足维护的需求。

如果你需要读整个代码或者代码的一个有限子集，你应当从哪里开始？我们大多数是从顶部开始。如果文档存在，我们看文档：在面向对象软件或成品软件的函数声明中，我们读类的层次图，类的层次图和函数声明帮助我们高的层次上理解所有程序段和这些程序段之间的联系。这就是所谓的自上而下理解方法。有时，我们认为我们对需要了解的代码比较清楚了，我们就使用自下而上的理解方法。在自下而上的理解方法中，我们读源代码的目的是为了理解软件的详细过程流。当我们不清楚要解决的问题时我们也使用自下而上的方法。所以，不是从头文件来理解软件相互连接程序块的总体结构，而是从主程序和跟踪指令流开始读源代码。

文档和代码

是否每个人都读了我写的文档？文档是否只是个项目管理者列表元素？我总是在想这些问题，你可能也在想这些问题。在一个研究中，程序员试图学习程序上记录的和面向对象程序的记录行为。在一个研究周期和两个维护任务中记录了这个行为。研究结果是：程序员减少了访问文档的次数，因为他们了解程序。在执行第二次维护工作时，使用文档的次数几乎为 0。这就意味着程序员已经熟悉了程序，对抽象文档信息的需求减小了。所以他们不用读文档，只要读代码。

文档有作用吗？这个问题的答案和文档的质量有关系，经验表明文档可以使维护工作变得更快，并减少错误。一个测量文档作用的实验表明经常访问文档就可以花较少的时间来理解系统并作出高质量的修改。相同的研究表明程序员编程的经验影响了维护使用文档的性能，因为程序员不得不花费大量的时间来理解程序。另一方面，没有使用文档的一组花费了更多的时间来理解软件。因此，这一组人员没用受益于他们的编程技巧，所作的修改也是低质量的。这意味着文档的可用性肯定会影响维护任务的质量。

查找源代码

即使使用了文档，我们也需要查看代码。这一部分将告诉我们如何根据现有的动机查看源代码。

在理解程序阶段，当我们试图理解系统的结构，我们就要查看函数和变量的定义，就像查看这些函数和变量如何在代码中使用一样。当我们阐明一个模块和修改的思想时，我们也要查看代码来观察所作修改对系统的影响。为了对这个影响进行分析，我们经常查看特定变量或函数的使用情况，确保我们理解这些元素的依赖关系。当我们查找一个 bug 时，我们就要确定出现错误的函数并跟踪函数内部和外部的变量。

一个对维护者如何查找代码的研究表明当我们查找代码中的信息时，就包含了我们查找的控制类型的信息：

- ◆ 函数定义
- ◆ 所有函数的使用
- ◆ 所有变量的使用
- ◆ 变量的定义

理解的困难（面向对象设计的教训）

因为最近几年面向对象设计的流行，出现了很多面向对象设计对程序维护和理解的作用的研究。这些研究的结果表明面向对象设计对程序的理解有一点副作用：

◆ 问题分解——很明显，面向对象设计使读者难于理解软件的功能。面向对象设计给读者提供了一系列的对象和这些对象的接口，而没有对如何使用一个对象做出任何的提示。这就好像给人一箱子的难题，而没有提供图片来说明如何解决这些难题。我们可能知道单个对象的作用，但很难知道程序的功能和程序段怎样解决问题。

◆ 关于局部化——当分析所需的信息在很多地方分布时，维护者就很难发现和收集所有的相关程序段。在软件工程中，代码在概念上发生联系而在物理上没有联系就发生局部化。很明显，继承会产生局部化，因为为了理解代码中的某一行，维护者需要遍历类的继承层次。另外，维护者可能没有正确理解程序并在信息发布时对信息做出不正确的修改。

◆ 概念上的消失——一个研究表明，类的层次加深时，开发者和维护者就很难在继承层次上达成共识。另一个关于概念上的消失和维护工作的关系研究表明：在面向对象软件中，对具有三层继承关系软件维护的时间比没有继承关系的软件的维护快。不过，当类的继承层次达到五层时，基层就没有这样的优点。很明显，维护者对深层次继承的软件理解确实比较困难。

◆ 动态联编和多态——动态联编和多态因为它们的特性会引起问题。直到在运行期间，多态方法被真正调用时，多态才会被系统发觉。这就意味了当我们要理解代码的工作时，我们必须查看所有执行多态方法的类。这就特别容易引起继承层次较深的类的问题。

◆ 多重继承——和多重继承相关联的问题是好的文档。多重继承的主要困难是确定基类的哪些方法被调用。

虽然这五个引起程序理解困难的问题是由面向对象设计元素引起的，但它们都不是面向对象设计所特有的问题。当维护者不知道问题怎样被分解时，如果代码是使用面向对象方法或自上而下方法设计的，那么就没有问题了。当维护者为了理解代码出现的错误——这是是否

由错误的包设计引起的、文件的错误放置、继承或多态——而需要在代码的很多地方查看，他们就很难理解代码。当维护者和代码的开发者之间存在观念差异时，维护者也将不得不花费大量的时间来理解开发者的设计和意图。有一个经验：使它简单、使它可访问。

可理解软件的特征

可理解软件应当通过提供问题和解决方法的高层知识来帮助维护者理解自上而下风格的软件，可理解软件也应当通过帮助他们生成软件的程序化模块和容易发现相关的信息来帮助维护者理解软件。可编程软件不仅仅是代码，也是和代码相关的所有文档和其他配件，这些东西可以用来帮助维护者理解软件。

根据我们已经提到的知识，以下几点帮助我们生成更容易理解的软件：

- ◆ 提供问题域知识——维护者和最初设计者、总体设计者对问题域应当有相同的理解。否则，就很难意识到已经产生的解决方法。这个领域的知识应当包括问题和问题的分解，也应当定义和问题域相关的元素和符号。另外，与问题相关的需求和用户方案也有助于维护者理解系统的使用。

- ◆ 提供高层解决方案——高层文档为维护者提供了一个指导，帮助维护者使用自上而下的理解策略。高层文档概括了软件模块和构件之间的关系，模块中软件的控制流和数据流。另外，文档还提供模块和构件中类或文件的关系。

- ◆ 提供注释——无数的研究表明，代码中的注释对提高维护者对软件的理解能力有重要的作用。当然，与代码重复的注释是没用的。注释必须提供代码中没有出现的信息，例如代码段的目的或算法的高层描述。

- ◆ 提供动态追踪机制——动态追踪机制就是运行时调试追踪，可以使用调试器或者使用跟踪日志文件来完成。这个机制帮助维护者知道程序的控制流和数据流，在自下而上的理解方法中是有帮助的。在第 12 章的“跟踪日志文件”部分有对动态追踪机制的讨论。

- ◆ 提供函数定义——在“实现”一节中就提到了提供函数定义。在 C 语言中，就要求在包含文件中定义函数原型；在 C++ 语言中，函数的定义和类的定义在一起；在 Java 语言中，Javadoc 能用来提供函数定义的集合，而不用在源文件中包含所有代码。这将帮助维护者执行自上而下的代码查找。

- ◆ 提供一致的名字约定——一致名字约定不仅仅是代码约定或使用匈牙利表示法。匈牙利表示法定义了详细的变量和函数名字的准则，在 Microsoft 的 Windows 程序中经常用到。不过，提供一致变量约定意味着必须把变量名和它表达的概念联系起来。例如，如果在很多地方都使用到客户名的系统，那么使用一个变量名将会有帮助，例如使用 `customerName` 来保存客户名，这样维护者就很容易在软件中查找所有的客户的名字。

- ◆ 记录功能开始点——在一个有上百上千行代码支持上百种功能的软件中，很难知道从哪个地方查找特定的功能。例如，如果有人找你解决一个实现“undo”功能的 bug，你知道从哪里开始查找这个功能特征吗？如果有一个文档告诉你该从哪一部分的代码或文件开始查找，不是很好吗？提供更多开始点的文档能帮助维护者实现需要方法的程序理解。也能帮

助维护者减小他们需要读的代码部分。

- ◆ 设计线性代码——很难理解多态性是因为维护者必须考虑所有可能的选项。这些选项产生了复杂性。为了使事情变得简单，我们可以减小变量选项的个数，这就意味着可理解的代码应当是线性和顺序的。

- ◆ 限制面向对象功能——如果不是用继承使类的层次变浅了，就很容易使维护者理解面向对象系统。另外，也应当避免使用维护者很难跟踪的动态调用和多态。面向对象的支持者可能会反对“但是，这些特征使面向对象设计的软件容易维护。”是的，如果维护者是开发者的话，当你在理解类的层次时就可以自由从一个抽象结构到另一个抽象结构，这些结构不会对程序的理解产生阻碍。不过，如果维护者不是程序的开发者，他们就要花一段时间来理解巧妙的设计概念。他们通常是没有时间来理解这些的。他们只希望知道他们要在哪些地方做修改和这些修改的影响。因此，应当明智和适当地使用面向对象特征。

- ◆ 限制全局变量的使用——全局变量的危害是很容易理解的。全局变量对程序理解的影响是要求维护者查找和理解软件中全局变量的所有使用情况，而不是处理使用全局变量的一部分代码，维护者必须理解整个程序。

- ◆ 减少不定位——减少不定位意味着我们应当保持相关信息在一起。相关信息就是影响一个特定功能或特征的信息。例如，如果一个功能要求装载一些参数，初始化这些参数的代码和装载这些参数的代码应当在一起。这样，维护者就不用查找所有的代码。全局变量的相关信息就是变量的内容。因此，使用全局变量增加了不定位性，因为你需要在很多地方发现这些变量怎样在代码中使用和设定。

14.2.2 可维护设计

在几分钟之内理解软件是没用的，在没有重写一半代码的情况下，你不能做出任何安全的修改。软件需要进行总体结构设计、设计，并且一开始就要考虑可维护编写。如果软件容易修改，那么它就是可维护的。但怎样才使软件容易修改？

可维护设计的特征

可维护设计使维护者容易修改软件。当维护者要解决 bug 时，他们必须理解 bug 的本质，并准确定位解决 bug 的地方。当维护者要添加一个新的功能时，他们就必须确定修改的地方。在这两个情况下，为支持作出的修改的次数将会影响到修改的容易程度。通常，所作的修改越多，修改就越难。

也就是说，一个可维护的设计就是软件模块之间有最少的依赖关系。如果有太多的依赖，那么软件就很难修改。一个软件模块可以是一个构件、一个类，或者是一个函数，这些是由要考虑修改的范围决定的。依赖的问题是如果你要修改一个模块，就必须考虑对与这个需要修改模块有关系的其他模块的影响。如果你不得不对有相互依赖关系的模块作出修改，那么就要对修改的结果作进一步分析。修改的结果影响越深，就越难修改模块。

一个对软件的复杂度和软件的维护研究表明，如果软件模块的耦合度很大，那么软件就

难维护。不过，问题不是模块间的现存的相互依赖关系，而是不正确的依赖关系。

例如，让我们来看一个电子商务的例子。简单起见，让我们假设有一个为两类用户提供的模块：买者和卖者。卖者卖东西给买者。而且，在原始需求的系统中没有卖者，它是系统的买者。所以在系统中，出现了卖者和买者之间的一对多的关系。一天，一个客户进来，这个客户对你的系统感兴趣，但他的交易模式和系统有些不同。他的模式是：客户是一个发行商，有多于一个的卖者。不是一个卖者对应很多的买者，你现在必须改变设计来满足多个卖者多个买者的模式。系统修改的容易程度有赖于系统对一个卖者对多个买者模式的依赖程度。例如，如果数据库已经嵌入了这个模式假设，那么就很难修改现有表来满足新的模块。如果账目模块中买者和卖者的关系不大，那么账目模块就需要重新注册。

设计框架而不是解决方法

一个避免错误的依赖关系的办法是为问题创建一个框架，并把这个框架一步步变成解决方法。当我们面对问题时，我们总是急着作出解决问题的方法。使用功能分解法，我们确定了解决方法的高层和用一系列函数分解问题的过程。在面向对象设计中，我们确定了问题中的所有对象和它们的相互作用。这些以问题为中心的方法是我们要实现所面对问题的特定设计。当问题改变时，我们发现不容易修改设计来满足新的问题。

除了考虑以问题为中心的设计中的问题外，我们还要考虑解决问题的工具。另外，除了提供解决问题的方法，还要提供框架。什么是框架？在这种情况下，框架就是构件功能块的集合，这些功能块能用来构造一个解决方法。构建功能块很简单并有一定格式。在电子工程中，构建功能块是用电阻、电容和导线。在建筑领域中，构建功能块是用铁钉、螺丝和组合板。在软件工程中，是用简单的代表一个且只有一个结构的对象和函数来构建。标准的 C++ 语言模块库就是构建块的集合，Java Swing toolkit 是另一个构建块集合。在面向对象设计中，可以把构建块看成是一个能用来解决问题的抽象接口的结合。当你要解决一个问题时，可以为问题域创建一系列的构建块。

在前一节提到的电子商务问题中，框架就是一个用户集合和他们之间的关系集合。数据结构代表所包含的用户，像卖者和买者，它们没有包含用户之间的关系。关系数据结构维护了关系，如果新的交易模块打破了关系，那么就比较容易创建一个新的数据关系结构来修改用户类。

过程可维护性

可能会出现异常，但复制时没出现问题或创建 one-off 的解决方法。有时，无数个 one-off 解决方法比起修改一个普通的，解决多于一件事的解决方法更具有可维护性。考虑实现一个确定交易模块的电子商务系统。假设这个电子商务系统已经做成产品并且稳定地运行。在设计阶段，你试图把这个交易模块设计成尽可能的可扩展，以便满足和更多交易领域的需要。不过，来了一个想买这个系统的顾客，你发现这个系统不满足现在顾客的需求。那么就不得不作出修改来满足新的需求。当然，客户希望马上解决这个问题。你该怎么办？

◆ 方法 1——你确定新的需求是否适合现有的设计，不适合就去掉。然后把经过删除

后的部分作为标准的、适合新用户和老用户要求的一部分。

◆ 方法 2——带着新的需求回到设计图中，根据新的需求重新设计新的交易模块。这个新的实现和现有产品联系起来，新用户和老用户都得到同样的升级。

◆ 方法 3——在现有系统中去掉现有交易模块，为新用户设计并创建一个特定的交易模块并给新用户一个不同于现有系统的系统。老用户不能从这个新版本中受益。

从我们的经验来看，每个人都讨论第 2 个方法，但很少实现它。这是因为我们通常没有时间，另一个原因是确实很难创建适合处理事情的所有可能方法。所以，我们经常使用方法 1。方法 1 可能是最危险的操作，因为破坏了系统概念的完整性并导致混乱的代码，也使代码的进一步修改更困难。在方法 1 和方法 2 中，都对现有的系统作出了修改。我们在第 3 章中知道，当对系统进行修改时，bug 就会出现。所以通过选择方法 1 和 2，就动摇了系统并存在着向系统引入了 bug 的危险。

方法 3 好像不同于任何我们在软件工程中所学的那样，大多数现有的文献都对 one-off 系统的不可维护性提出了警告。这个惯例说明 one-off 系统是不可维护的系统，因为每当你作出修改，你就不得不查找所有相同修改的地方。如果你没有这样，那么你就要维护多个系统。一个关于 GUI 的设计规则是从应用层中分离出表示层，为什么？这样可以更容易修改不同用户的表示。这不就是一个 one-off 系统吗？我们不就可以认为对不同的客户有不同的 GUI？我们对这样可维护的设计有疑问吗？当然没有，因为我们接受了不同用户需要不同接口的事实，从应用层中分离出表示层对修改这两层都很容易。

one-off 系统的不可维护性不是因为系统的 one-offness。它的不可维护性是因为我们没有在软件设计过程中设计维护过程。在设计阶段，我们没有考虑到：“如果有人需要不同的应用模型，我怎样进行对应用模块的修改？”它的不可维护性是因为我们没有把系统设计成 one-off 系统。这要求软件被设计成模块形式，这样就可以把现有的模块替换成新的应用模块。也要求我们考虑接口和其他在前一节讨论的问题。但可维护的设计仅仅是任务的一半，另一半是和软件一起进行可维护过程的设计。这个可维护过程需要清楚定义在哪些地方做了修改和哪些模块需要调动。这个可维护过程还需要说明一个系统到另一个系统的修改。没有可维护过程，那么就很难维护和管理所有不同的 one-off 系统，也就产生对 one-off 系统优点的争议。使用了可维护过程，即使软件不是模块化的，也可以对 one-off 系统进行维护。

如果操作正确，一个 one-off 系统有很多优点。第一，创建一个新的系统不会影响到现有系统的稳定性。第二，一个 one-off 系统可以潜在创建很多可重用的构件并使我们接近了即插即用的软件结构。使用方法 3 创建的新的应用模式容易被删除和被第三方应用模块替换。例如，如果需要的话，我们在另一个产品中可以重用三个应用模块。创建一个 one-off 方法意味着不用考虑你的修改怎样影响到现有的解决方法。这就可以更容易为用户测试和设计恰当的解决方法。

14.2.3 提供一个回归测试环境

总体上,可维护软件系统的特征是可以容易地评估修改的测试环境。使用回归测试环境可以容易重复修改和测试周期。理想上,回归测试环境应当是自动的,这样维护者就可以快速进行修改测试,而不需要附加的工作。如果一个工程项目很大,在完全回归测试中的测试用例个数也将很大,用来运行这些测试用例的总时间也会很长,也不合实际。所以,回归测试环境可以为维护者提供测试用例的一个子集,维护者可以在合理的时间内进行测试。当然,一个完全的回归测试需要测试者来运行,而不仅仅是进行修改。

14.2.4 创建一个可维护的环境

软件开发者使用工具、方法和开发环境使他们的工作变得容易,维护者也同样是这样的。事实上,因为维护是软件开发中最长、消耗资源最多的阶段,那么让维护者有一个好的环境,提高他们的工作效率是很有必要的。一个好的维护环境包括一系列支持维护任务的过程、能使维护者学习系统、一系列理解和修改程序的工具、能使维护者立刻知道修改的正确性。下面五个思想能帮助公司创建一个好的维护环境。

- ◆ 设计公司的维护过程——维护过程不仅仅是形式上的 bug 报告和发现过程,维护过程对需求修改和添加新的特征也需要一个管理过程。在公司里,需要定义这些过程,所有的风险承担者(例如销售者和市场、客户服务和开发团体)都注意了维护需求的影响和组织。这个过程也允许风险承担者能对不同的需求排出优先级,以便满足市场的需要。

- ◆ 设计技术维护过程——对特定软件解决方案的维护过程需要在软件设计时就被定义。这就意味着当设计者设计一个软件系统时,他们需要认识到维护的需求,也意味着在设计软件时需要回答以下几个问题:①维护者怎样调试系统?②维护者怎样添加一个新的功能?对这些问题的回答应当作记录并转换成一系列过程,这些相联系技术解决方案应当在设计中考虑。

- ◆ 提供学习系统——当你可以和运行的系统一起工作时,就容易学习系统。理想的情况,每个维护者应当能和模拟产品系统的系统一起工作,即使没有执行有用的维护工作。

- ◆ 提供维护环境——一个维护环境包括帮助程序员查找代码和观察软件的工具。在 Unix 机制中,有很多帮助查找代码和读代码的工具(我们通常使用 emacs,grep,find,gdb 等工具)。在 Windows 平台中,不同的商业开发环境提供很多查找和可视化能力。通过这些工具,可以发现通过包含一系列可查找源代码的窗口,立刻可以看到每一件事,这可能有利于解决局部问题。帮助你查看软件模块之间关系的工具(例如系统的流程)也是很有用的。这些工具包含了调试跟踪构件,像可以帮助你查看变量和函数的使用情况的可视化开发环境一样,这些构件可以让你容易跟踪软件的控制流和数据流。

- ◆ 创建回归测试环境——和学习系统一样,使用回归测试环境也是有用的。当维护者作出修改时,他可以立刻用测试程序来确定他所作的修改是否对系统有副作用。

14.3 维护现有的软件

在任何时候，都要注意到系统的可维护性。但我们大多都要工作在已有的系统上，这个系统通常是在混乱环境下的没有文档的系统。所以我们怎样维护现有的系统是非常重要的。但即使我们的系统很容易维护，也要求我们要非常认真地修改，否则我们就有可能在不可维护的情况下把危险引到可维护的软件中。

14.3.1 修改现有软件系统

当我们回答“非常仔细！”，这不是玩笑。当我们不得不修改现有软件时，这是一个好的观点和方法。我们尽我们的所能理解软件的功能和我们被要求修改的软件部分。我们花费大量的时间在查找修改部分，然后对它进行修改，接着测试它们。当我们的修改不起作用时，我们又返回，在初始阶段学习代码并重新开始。当然我们也要对我们的修改做出文档并对现有的文档进行更新。

理解软件

在做出真正的修改之前，我们需要找出修改的地方和修改的副作用。这就要求我们理解我们所要修改的软件，如果我们对软件一点都不了解，要做到这一点将很困难。以下几点可以帮助我们理解系统。

- ◆ 读文档——显然，如果有文档，就必须读它。在很多情况下，这可以让你对软件有高层的理解。但不要太相信它，因为文档可能不正确。可以用它来做向导，但不能认为它是真正的代码。

- ◆ 读代码——我通常喜欢从代码的顶层开始，一步步往下。所以我通常从外层开始来发现每一层的功能以及它与其他包的关系。接着读模块层和可能的类层，仅仅是为了得到所有这些程序段的流程。在这时我还没有准备进行任何的修改，但我已经知道应该从哪里进行修改。

- ◆ 打开跟踪——如果软件有跟踪日志的能力，就使用并观察它。跟踪日志是快速理解自下而上程序的好办法，而不需要去读代码。

- ◆ 画图——如果程序很复杂，并且代码很大，那么就不容易跟踪所有的依赖关系和程序流。当发生这种情况时，画出流程图对帮助你观察调用时发生的事件是有帮助的。在画流程图时，你可以发明任何代表你的目的的图形。你不需要为画图做准备并使用 UML 来表示程序。这些工具可能会降低你的工作效率。但一旦你有了系统工作流程图，就可以考虑用这些图为他人做文档。

- ◆ 运行/跟踪测试程序——初始的开发者很少使用测试程序。测试程序能帮助你理解软件的使用，并给你一个自下而上读代码的切入点。

- ◆ 和测试者交流——测试者需要测试软件，所以他们必须知道软件的功能。因为为了

测试软件，就必须使用软件，他们将使用工具，例如测试脚本、应用程序和预想的输出来运行和验证软件。这些工具将帮助你理解软件的使用和功能。

◆ **相信代码**——如果你的文档和代码有冲突，你知道是文档的错误。如果有人告诉你软件的工作和代码的工作不一样，那么这个人就错了。你必须根据现有的代码进行修改。当然，你可以去查找别人给你错误信息的原因。是不是代码做了修改？是不是需求做了修改？你也可以更正别人的错误理解，防止他们在别的地方犯同样的错误。

只能留下真实的修改

维护者有一个对代码做出“测试性”修改的坏习惯，这些修改仅仅是为了观察它们的作用。如果这个“测试”代码没有起作用，他们就会删除这些没用的代码并添加别的测试代码。为了调试一个复杂的 bug 或特征，这些试探和错误的方法会出现几个循环，并且代码会变得非常大。在配置管理系统中检查这些修改之前就删除这些没有起作用的代码，这个方法就很好了。但当你把所有的试探和错误的代码当成真正的修改，这就会把试图发现所有这些无代码的维护者搞糊涂了。

你必须跟踪每个修改，这样你就可以在必要的时候删除修改。很多源代码控制工具允许你删除修改并使代码回到前一个版本。这是跟踪系统改变的有效工具。

记录修改

遗憾的是，维护者没有作文档。你可以用以下的方法记录你的修改：

◆ **在代码中**——代码中的注释对维护者很有帮助。既然这样，作为维护者，你最好在其他维护者之前做好这个工作。在代码中作上你为什么做出修改的注释，谁做的修改，什么时候做的修改。这样，其他人如果对注释有疑问时就知道问谁。事实上，对每一行的修改都应当做出注释或标志。这样，每个读代码的人都知道这行代码被修改了。

◆ **在修改日志中**——修改日志应当保留每个维护任务对软件的修改。这个修改日志可用作测量的尺度帮助确定软件的可维护性。修改日志中的很多信息和它的使用方法在“进行修改跟踪”一节中提到。

◆ **在现有的文档中**——维护者最大的障碍是现有的信息在文档中没有更新。作为维护者，喜欢使用文档的原因是文档给出了软件的高层信息。但如果文档中的一段信息和实际系统不相同，那么你对整个文档就有疑问，也将不愿意读文档。

◆ **发布修改**——让每个人知道已经在程序中做了修改。可以使用代码中的注释或修改日志。在任何情况下，发布修改让每个人都知道修改。在一些情况下，初始开发者可能注意到修改但可能没认识到修改的副作用。

14.3.2 围绕现有的需求和设计 bug 展开工作

可以总是围绕着现有的需求和设计 bug 展开工作，在软件工业中总是这样做。实际的问题不是怎样围绕着 bug 工作，而是怎样最小化这个工作的副作用，防止软件性能的衰退以及变得不可维护。

这个问题有赖于你的资源、优先权和条件、机会。产品中设计或需求 bug 的本质决定了不是出现在实现中。如果你很幸运碰到一个局部的 bug，可以通过一个简单的设计修改来解决 bug。不过，我们通常没有这么幸运。通常情况下，一个工作区是指产品做了本不应该实现的功能。例如，这个产品可以是一个类，不是给出一个新的抽象，你可以移植一个新的方法、添加一个新的成员变量使它完成你的需求。（即使你使用继承来完成这个功能，你仍然要执行一个工作区。如果这个继承类在概念上不同于你修改的类，那么最后可能会影响到你。）这个产品也可以是一个程序流。例如，你添加了分支或一些延迟，这段程序完成了你想要的功能，即使你的修改在函数中引入了副作用。

为了避免重设计的缺点，你最好理解工作区的交替使用并作记录。如果系统一开始就很好设计了，工作区就会很明显。它们是从统一体中分离出的一些代码。但如果软件没有很好设计或者在维护模式下很长一段时间，那么就很难发现工作区。这个软件将缺少概念上的完整性，每一件事都像一个大的黑洞。不管怎样，对不协调情况做出清楚的注释将帮助其他维护者理解代码并防止他们破坏你的工作区。

14.3.3 彻底的回归测试

在这一章中多次提到，回归测试在维护阶段是一个最重要的元素，特别是在你对系统没有完全理解的情况下。回归测试意味着在维护工作之前就有了测试用例，你在新的系统下运行这些测试用例。当系统已经做成产品了，通过定义就可以维护修改，最后你可能会打破一些工作的系统。如果 bug 太严重，那么整体的关系都可能被破坏。在原有的修改上进行修改可以迅速取得软件的可维护性。

提示：当有疑问时，就进行测试。没有疑问时，再进行测试。当你完成这个测试，再进行别的测试。

14.3.4 保持修改跟踪

跟踪修改不仅仅是在通知配置管理库修改时做注释，还要跟踪修改需求、修改的人员、被修改的文件和实现这个需求的每个需求函数。但在没有人查看这些信息时就没有必要跟踪修改。可以收集和使用这些测量来帮助你分析软件的进展和成熟，也可以用来确定和预告需要审查或者重新设计的可能出现 bug 的模块，还可以帮助你相信你的系统已经变了不可维护并需要重新设计。以下提到的一些知识对测量很有用，你可以从修改跟踪系统中收集到，帮助你理解软件的发展过程。

- ◆ 确定修改需求的来源。如果大量的修改来源于销售商和市场，就要正确查出他们卖的东西。可能他们对产品的概念不同于开发团体对产品的理解，对销售和市场的讨论是合适的。不是继续对现有系统进行修改，而是要开发一个新的产品来满足市场改变的需要。

- ◆ 确定出现太多修改的模块。模块出现太多修改有以下几个原因：①模块可能隐含着 bug。这可以通过和对模块做出有影响修改需求的人交流来确定。如果模块确实有 bug

倾向，那么就要对模块的设计和实现进行谨慎的再评估。②这个模块是万能的模块。根据你的系统，这可能是设计和体系结构的错误。③这个模块是系统的“集线器”，也就是很多模块依赖这个方法，不论什么时候有人需要修改系统的其他部分，他们都要修改这部分模块。这可能是一个设计的失误，和这个模块相关关系的系统设计需要进行再评估。至少，这要求你不论什么时候对有修改倾向的模块进行修改时，你必须十分小心，因为这个修改会影响到大部分系统。

- ◆ 跟踪修改范围。一个修改范围可以简单定义成文件的数目、函数、类或每个修改需求的代码行。如果范围很小，这就意味着每个修改需求是局部的，或者软件模块是松散耦合的，在软件模块中没有太多的依赖关系。不过，如果范围很大，你就要提醒自己了。这意味着系统不是做了它应当做的事，或者系统变得不可维护（例如，它有很多的模块内部的依赖关系）。跟踪修改范围的历史可以为软件系统的可维护性提供有价值的线索。

- ◆ 跟踪修改的效果。修改的效果是测量维护者在理解修改需求、理解代码、做出修改、测试修改和提交修改花费的总时间。修改的效果帮助跟踪维护的代价。通过和修改范围的连接，可以告诉我们程序或模块是否真的不可维护。例如，如果维护者在修改一行代码上花费大量的时间，那么就要对软件的可理解性提出怀疑。

14.4 什么时候要放弃

很难放弃一个软件系统。今天还在使用 20 年前就开发应用的 Cobol 主框架系统。为了在 PC 时代使用，C 语言中间层软件定义了一个标准的应用程序接口集合（API）来隐藏原始系统的复杂性。这个中间软件用特定的消息协议向主框架发送一系列需求。也可以使用一些 API 格式，或屏幕扫描来实现。在网络时代，主框架中的数据满足了 Web 许可。所以一系列的 Java Servlet 被创建使用 EJB 来和 C 中间软件进行通信，和 COBOL 主框架系统对话（可以用 ASP 和 Windows 环境下的 COM 替换 servlets 和 EJB）。我们不是不要旧系统，而是经过一层层的包装来实现我们需要的行为。

从管理的角度出发，这是个好的思想。我们已经知道主框架的工作，所以当我们在主框架上加一层包装时，我们只要当心包装的正确性。因而为了创建一个新系统，我们不得当心整个系统的正确性，我们也不得不把现有的数据移植到新的系统中。此外，我们需要跟踪新的操作来处理新系统中的错误。创建一个新的系统的费用比维护一个系统要多得多。

当我们现有的系统出现一些客观的影响时，我们只能放弃这个系统。有哪些因素使我们放弃旧的系统？

- ◆ 没有可替代部件——在计算机上运行软件，而计算机变得陈旧了。当我们这样做，我们不能发现可替代部件，那么就要考虑把这个软件移植到另一个平台。

- ◆ 没有维护者——在 Y2K 时期，我们听说 Cobol 的老程序员退休了，并给了很多钱来帮助解决 Y2K 中的 bug。这是因为新一代的软件开发对 Cobol 或主框架环境不熟悉。

在后来的 10 年内, Cobol 学者的人数降到了几千人, 那么谁来维护这些系统呢?

- ◆ 第三方软件不存在了——如果你的系统依赖第三方软件, 但提供这个软件的厂商不再存在, 无法提供这个软件, 那么就要修改和重新设计你的系统。

- ◆ 新人员——当上层管理或技术专家改变时, 那么就经常要改变整个系统。雇佣的新人对软件的构造有完全不同的思想。他们可能在现有的设计中发现很多的不可接受的错误或危险, 也可能是其他原因引起软件的修改。通常情况下, 当公司的上层发生改变时, 那么“不可维护”的系统就会被抛弃。

- ◆ 不可能对旧的软件系统作修改——一个软件系统可能变得很固定了, 以至不可能作出任何的修改, 除非打破系统, 或者不可能经过简单的修改软件满足新的需求。

最后, 是维护一个旧系统还是创建一个新系统需要由管理者来决定。不过, 作为一个专业技术人员, 我们可以为管理者提供一些测量尺度帮助他们理解可维护系统和不可维护系统的代价。决策者没有同意一个新的开发项目仅仅是因为害怕现有的系统。但你可以通过给他们一些和维护代价相关的数据和尺度, 来帮助他们作出决定。例如引入 bug 的概率、bug 解决率和调试效果等数据可以让管理者知道对旧系统的维护已经没有价值了。当 bug 的出现率大于 bug 的解决率、并且调试效果随时间成指数增长时, 这些数据就特别有用。为实现一个新的特征而引入一个新的功能特征的比例, 也是一个令人信服的数据, 可以用来说服别人。

14.5 小结

我们在这一章提到, 创建可维护软件不是一个简单的技术问题。从技术本质上看, 可维护设计只是可以帮助你更好完成工作。因为软件是“软”的, 所以可以容易被修改来满足你的需求。这些灵活性是我们的毁灭。对我们来说, 创建可维护软件就是在软件开发和发展下创建一个可维护环境。这个可维护环境是技术、过程和考虑问题方法的集合。我们已经提到的思想可以使你的软件更具有可理解性, 我们已经描述了关于可维护结构和设计的设计思想, 我们也提议在软件开发和发展中创建维护过程。这一章的思想可以帮助你创建更多的易管理的维护任务。

不过, 和调试一样, 维护是软件工程中不好理解的对象。和调试一样, 维护也花费很多的时间。我们在创建软件时很少考虑到维护, 并且没有进行软件维护的流行方法。我们为将来留下了遗物, 如果我们的儿子或者孙子嘲笑我们的工具并试图创建和支持软件这是不足为怪的。

14.6 bug 问题

你在一个 API 公司工作。基本上, 你的公司在网上为客户生成应用系统。你已经在这

个公司工作了两年，每一件事都很好。这个公司发展起来，你有更多的用户，你要添加新的硬件来处理系统中增加的通信量。开发团队从 10 个人增加到 30 人。所有这些说明你的公司变大了！不过，有一个不好的情况，有什么东西出问题了？（提示：什么原因使开发团队从 10 人变到 30 人？还有，他们在干什么？）

第 15 章 专业调试

通过这本书，我们已经讨论了运用于软件工程中的开发、测试、方法的调试。在这一章中，我们将从专业角度来研究调试的概念。

所谓专业调试者是一个很可能被多数自诩为纯化论者（purist）的软件开发所嘲笑的角色。调试是作为开发过程中的一个方面或是软件公布后的软件维护中所进行的一些工作。像调试者一样，调试是用于解决由于软件的性能差而遇到的一些问题的一套工具和技术。

即使已经通篇看完全书，也不可能完全掌握调试技术。它不是临时的技术，它是一门科学，一种应该用于软件开发过程中所有阶段，从开始的需求分析阶段到最后的配置和维护阶段的技术。在本章中，我们将作为一种专业来研究调试。至于对体系结构设计者、需求分析专家、软件工程师、测试人员，调试可能是一项全职工作。我们将研究这种职位的要求、公司的利益，以及最能描述从事这种职业的人的技术和品质。

15.1 学习成为一名调试员

当然，当你把调试作为一种职业讨论时，第一个要考虑的问题是你如何学习成为一名调试员。毕竟，调试没有什么等级可言，也没有什么可参加的考试来证明你是否知道如何调试。也很少能在报纸上看到关于该职业的广告。“诚聘：为大型软件开发公司工作的调试员，必须了解调试的基本原则，并且能够在一个星期内找出 bug。”虽然这可能是个有趣的广告，但值得怀疑的是你能否获得许多这样的申请。你该如何学习成为一名专业调试员呢？

15.1.1 在软件维护和改进组中工作

学习系统调试的一种最好的、最容易的方法是工作在一个调试工作多于源代码编程的环境中，正如在整本书中所看到的，调试所需要的技术在编写代码中难以得到提高。这一点很遗憾，但它仅是生活的一方面。因为这个原因，对于学习成为一个调试员的最好的建议是工作在存在很多 bug 的项目中。

有趣的是当你告诉人们你希望接手一个有很多 bug 的项目时他们的脸部表情。找出 bug 并给予解决被视为简单的工作，只属于公司里报酬最少的员工干的活。新手经常被用来发现 bug，因为他们不知道一些建立软件项目的历史。这个原因很简单。第一，有能力的人、长期开发的人和公司的创始者开始一个新的项目。通常情况，因为他们有相当的权威并且知道他们自己的主要任务，所以他们的开发任务是在关键层次上的开发。这些项目有不同的需求，相对比较难安排，开发成功的希望也较少。不过，也有可能使一个难的项目开发成功。通常

的想法，这些项目的开发者经常一开始做就走了。在这一点上，你要在开发的世界中考虑所有可能的情节。一个项目很少或没有预先计划、很少测试和安排，那么就无法考虑将来的扩展性、可维护性或真正满足用户的需要。

被成功开发的商业工程是专业调试者理想的培育园地。不过，我们中的大多数人不可能进入这个领域。所以，我们怎样才能较好地为一个人成为调试者做准备，而不会使他们从这个领域消失呢？这个问题出在维护和改进团队中。任何成功的项目（也就是，一个被出售或被使用的项目）都需要维护。在这本书上，我们认识到几乎不可能有没有 bug 的软件。由于这个原因，任何产品质量要求的软件肯定包含了一定比例的 bug，即使在最好的软件中也存在这一点。在“最好体现”的项目中，最好的调试者是天生的。因为软件有需求，项目被较好地实现，那么调试工作就是一个科学的努力，而不是一个没有希望的工作。

你可能会想如果你是一个较好的调试者，你就应当在你自己的项目中查找并解决 bug。解决你自己的代码不同于解决别人的代码。首先是因为你知道系统中的每一件事，其次是因为你理解系统要解决的问题和系统怎样实现这些任务。这就不理解系统的整个代码和过程。事实上，你可能会犯一个较严重的错误，因为你没有去看系统。从别人的代码开始并从这一点上开始你的理解工作是相当好的。

你怎样在你自己的公司中分配维护和改进项目？除非你是在一个非常不寻常的企业中工作，否则你就可能碰到这个问题。事实上，从这样一个工程中进入的一个最好的方法是表达这样做的目的。因为其他人不愿意担任这个职位，你就经常要完成维护的工作。如果你是一个管理者，分配人员来完成这样的工作通常会遇到阻力。这是因为在很多公司中都认为这个工作是个没有希望的工作。能进行较好调试系统的人通常会无法从这个系统脱身。为了公司里的其他雇员，你的任务是看谁从其他领域到调试领域的工作比较好。这样，在你的公司里，就可以鼓励发展真正好的调试者。在这一章的后面一节“什么地方需要专业调试者”中可以看到，一个有专业调试者的公司会使工作更容易，并能开发出质量比较高的软件。

提示：将最好的调试员从一个项目调配到另一个项目，从而提高她（他）的调试技能并且避免厌倦的情绪。

一旦你成为维护组中的一员，如何才能提高你的调试技能？当然，最好的方法是调试源代码。采用我们在这本书前面部分所探讨的方法和技术，当遇见问题时则运用相应的方法，并寻求最适合你的方法和技术。不是每个人都会选择相同的方法，而且任意两个不同调试人员在解决一个问题时也不会采用完全一致的方法。这并不意味着调试作为一门科学比任何科学，例如宇宙学，逊色多少。对任意三个研究宇宙学的科学家，你将发现他们的表达方法有轻微的不同，例如宇宙是如何膨胀的或如何收缩的。尽管我们知道，他们中没有人会是错误的，虽然他们确实完全不一样。当你看一道数学证明题时，你不会检查它顺序是否正确。相反，你将检查每一步是否正确，每一步之间是否符合逻辑推理。类似的是，调试并不是特别的顺序或特别的技术，而是一种逻辑方法，关于哪种技术适用于哪种问题。

为了成为一名真正的专业调试人员，你需要调试许多不同的应用程序。为不同的环境，

和不同的部门编写的应用程序以及由不同研究开发小组编写的应用程序。调试由同样的开发小组在同一领域编写的应用程序的问题时你只能获得比较单一领域的知识而对于另一领域一无所知。当你来到一个新环境，你将发现，原先想当然的事情并不存在。虽然，由单独一家公司开发的大多数应用程序容易在同一领域遭到失败的命运，你将惊奇于同一领域内的巨大差异。一些系统是为终端用户所写，其他系统则是为客户提供服务的人所写的。也有一类应用程序只适合公司内整体操作。所有的这些应用程序都可能存在 bug。为了发挥你在这本书中所学到的知识，就要了解掌握各种环境中最有可能会出现 bug，如果你能事先预测将会遇到哪些问题的话，你将获得一个好的开头。

事实上，一个专业的调试员能够在获知应用程序的类型，以及是由谁开发这些应用程序的情况下，预测可能会出现各种问题。此外，如果知道一个应用程序的开发进度，通常就能够预测你在不同程度上会遇到哪些问题。这使得专业调试与软件开发项目的管理同样重要，不可或缺。不要因为你调试程序而认为调试简单，要不然你注定被视作白痴。事实上，我们很容易看到，将来在某个时候，调试将成为通向项目管理的自然途径。

提示：好的调试员能够预测可能遭遇到的 bug 的数量和类型。

15.1.2 学习编写源代码的一个好办法

假如你不想成为一个全职的专业调试员，那么你为什么自愿参加调试任务，最有可能的情况是解决别人的问题。专业调试员可视为一个过渡期的角色。一个能够使你获得更多更好机会的角色。一些人把这称作：公平交换。对于缺乏可能的开发经验的人而言，一个最大的缺点是刚刚从学校出来的新手常常被陷入到在新的应用领域开发新的源代码的困境中。因为这些新手在这一领域没有呆过，他们缺乏工程项目获得成功所需的关键技术。

学习如何开发源代码最好的方法是从别人的错误学起。假如你曾经看过别人所犯的某种错误，你就能努力避免犯这种错误。这毕竟对你写代码有帮助，而且在整个开发过程中都应该坚持这一点。想象一下，假如你发现当你传递给对象一个空指针，而对象的方法无效。然而，需求分析里也没有要求一定要传递非空指针，当你在写自己的代码时，如果一个方法需接受一个指针值时，你当然会记得检查其是否正确？一旦你几次都看见相同的错误，你将永远记得。因为这个原因，调试员除了在调试方面很出色，在帮助设计和实现系统方面同样很优秀。回顾以前所犯的错误，你就能确保今后的工作中不会出现类似的错误。当然，这可能会导致出现新的错误，但至少你能消除今后工作中可能出现的大量错误。

15.2 什么地方需要专业调试员

假如你是一名专业调试员，你需要养家糊口。不是所有的公司都有项目刚好处在需要调试的阶段，所以他们为什么要雇佣你呢？在公司需要销减开销时，这可能是额外的费用。一开始，公司没有自己的产品，正处在开发阶段，在雇佣开发项目的其他人员的同时雇佣调

试员是否明智呢?我们认为,这不仅是明智之举,而且专业调试对任一个项目,无论是新的还是旧的,都是不可或缺的。

15.2.1 在设计阶段

设计一个系统时为什么需要调试员?这就好像是当你建筑一栋新公寓时,雇佣一个打杂的。这看上去好像没有任何意义。不过,如果你考虑了这个问题,在这个时候你是不能少了调试者的。第一,在设计系统时调试者可以用来发现问题,因为他不得不跟踪和解决 bug。第二,调试者知道是什么使系统变得不可维护。你当然不会在一开始设计时就使系统不可维护,而在将来来提高系统。

有以下几个在设计阶段使用调试者的原因:他们习惯用他们的方式进行工作,并发现事情的工作原理。为系统提出一个设计计划,他们可以用他们的方式从系统的 A 点工作到 B 点并观察这种方法遇到的漏洞。在实现阶段遇到的一个最大问题是发现了在设计阶段没有捕获的元素,这要求实现团体作出他们自己的设计选择。这是一个你要在设计阶段使用维护者的原因,不是吗?所以,如果你排除了在设计阶段遇到问题的可能性,你就排除了 bug 的整体漏洞。

调试者知道问题的原因和结果。在设计阶段某一设计不仅仅影响了实现,也经常影响系统的性能。调试者对性能问题特别有经验。在很多问题下,如果你在性能问题变成程序问题之前捕获到问题,就更容易解决。

软件工程博学家说解决在设计阶段捕获的 bug 的费用比起解决在产品种捕获的 bug 的费用少 500 倍。因为调试者有丰富的测试和调试经验,就能很好地捕获设计 bug 类型,这就要求在设计阶段使用调试者。

15.2.2 在需求阶段

带着一个需求到软件开发公司,你听到“我们对它进行需求分析,在需求分析阶段不需要任何开发者参加。”很难相信可以得到进一步的发展。毕竟,如果需求分析不完全,或者不一致,那么项目就不可能成功。缺乏高质量的需求分析是项目失败的最普遍原因。问题在于在需求分析阶段要求哪些开发团队参加呢?当然是调试者。

需求收集是两个交替的过程。第一,你需要理解用户要实现什么,这样你就可以实现这个方案。这是一个我们已经讨论的问题“总体图”,对调试很重要。谁比调试者更好发现完整的问题?需求阶段的第二个过程是确保你的需求完全、一致。虽然很难知道需求是否完全,当你把所有的问题放在一起,完全性检查就相当容易了。再次强调,调试者做的最好。调试者检查系统的所有不同部分,并把他们集成为一个流。他们可以通过观察这些流来发现一个过程结束之后是否到达另一个过程。如果这个流不完全,那么需求也是不完全的。和设计中出现的问题一样,解决需求问题的代价也很高。为什么没有人来处理这样的问题呢?另一方面,一致性需求是最容易产生 bug 的地方,如果需求 A 要求做一件事而需求 B 要求做另一

件完全相反的事，那么最后的系统将会出现问题。

在一些情况下，调试者在需求阶段是一个顾问，调试者能很好发现和解决问题。不过，不是很明显，调试者也使其他人理解问题。当你需求解决一个 bug 问题时，你就要查看出现问题的源代码并在问题出现前对他（她）的行为提出问题。在有其他系统运行或使用特定的硬件配置的情况下，你可以发现答案。在需求收集阶段，这些信息都是同样重要的。比起对软件开发过程的其他角色来说，这个过程可能对调试者更重要。为什么没有把这个技巧使用在最好的适配环境中？在需求阶段，要确保有调试者参加。

15.2.3 在代码审查阶段

你是否进行代码审查？如果没有，你确实应当考虑进行代码审查。在软件成为真正的产品之前，代码审查是发现问题的一个最有效、性能价格比最高的方法。代码审查就是通过一组没有偏见的人对代码进行正式审评。这个思想就是让尽可能多的人同时进行读代码并指出可能看到的潜在的问题。代码审查的开始点是工程源代码部分，代码审查的结束点是写原始代码的开发者应当注意的一系列注释、评价和注意点。这一点可能很明显说明了你为什么在开发阶段要有调试者，让我们来看看另一些不明显的原因。

代码审查中的一个最严重的问题是写新代码的人要做的事。因为这个原因，很少提出可维护的讨论和什么时候进行可维护讨论，通常是这么一个问题“我不想那样做，我要这样做，”没有真正的原因。调试者害怕处理复杂的代码并知道什么使代码能或者不能进行可维护。他们能对在维护阶段是否可以理解并容易修改代码给出一个正确的、没有偏见的观点。另外，调试者可以提出新代码开发者从来没有发现的问题。例如，你可能用 API 函数写了一个从数据库中获取数据的函数，一个新的开发者看了这个函数但没有发现任何问题。不过，调试者认识到公司在 5 年之内已经对数据库进行了三次的修改，并坚决要求用一些普通的方法为数据库修改这个 API 函数，这样就可以在再次修改数据库时，减少系统的工作量。

另一个代码审查的问题是“自我”的。当你写了一段代码，你就希望对你的工作有最大的回报。如果有其他人指出你的代码的问题，你会感到很痛苦，并且会经常抱怨他们。在很多情况下，确实是这样。因为每个开发者喜欢用自己的方法来实现一些系统，他们不认为你的方法是最好的方法。毕竟，他们不会想到你的方法。当代码审查时带有个人的偏见，就不可能会有好的结果，只会出现一些失败。当然，最终的失败是项目的失败，它将不会有做好的代码来实现它。当你考虑到这些问题时，这是问题的关键。调试者不会带着个人的偏见进行代码审查，因为他们不会进行新代码的开发。另外，他们是从为解决问题的观点来查看代码的。他们所关心的是代码尽可能的清楚并且没有 bug。

代码审查的另一个问题是竞争。当你有一组开发人员一起工作时，他们之间总是有某种形式的竞争。工作最好的人总是最有可能成为每个人都想参加的新项目的候选者。由于这个原因，总是要贬低别人的工作，即使他们不想这样做。这也是“自我”的另一个表现，但不是那么明显，而且是完全无意识的。不过调试者不会和其他开发者进行竞争。毕竟，如果有

一个新的项目，他们也是要参加的。所以，为什么要担心自己是否会不会参加这个项目呢？在调试者调试一个系统时，就会放弃很多的个人考虑。

你可以知道为什么在代码审查阶段使用调试者。他们可以发现新开发人员无法发现的问题，因为他们只习惯在用户真正使用系统之前把代码调试成符合需求分析的要求。他们在出现“自我”的情况下担任仲裁者的角色。另外，他们已经遇到很多很多的编码风格和技术，并且知道在长期的运行中哪些部分最可能出现问题的，哪些部分可能最容易维护。

15.2.4 在代码移交阶段

很多的开发者、程序员和管理者都没有考虑到代码的流通。在一些地方，开发人员把他们的编辑器和开发日志包装起来并去开发另一个项目。不要指望开发人员用他们的一生来开发一个简单的项目，这是不现实的。要么给他们新的工作，要么他们到另一个公司去。代码更新是否是一个清楚的，好的定义的事件，或者是否是一个经常被项目人员发现的混乱的故障，并且他们是否理解这个过程。在项目的初始开发者不在开发这个项目并把这个项目移交给其他的开发者或者维护者时，就会出现代码的更新。

在代码更新阶段有两个使用调试者真正可行的方法。第一，可以把调试者当成中间人从原始开发团队中转移到维护或者开发团体中。第二，可以让调试者评估“移交文档”，系统的代码和文档发现新的团体是否理解了系统。让我们看一下这两个方面来理解这两个方法怎样使用在你的团队和工程模块中。

很多的公司都把调试者当成中间人，从项目的一个阶段到另一个阶段。调试者是一个理想的传输人员，这是因为他们可以在工程的每个方面中出现，从开发和文档到测试与安装。因为调试者对工程的所有方面都知道一些，他们就可以得到这些方面的好的原始信息。另外，因为调试者经常被要求挖掘一件事来观察它们的工作，他们因为知道从哪里开始查找就可以对项目进行好的侦察。当你考虑到这些时，作为中间人的另一个人选是开发者和测试者。开发者不是好的人选，主要是因为当其他人进行新的开发对他们有负面的影响。测试者也不是真正好的人选，因为他们没有深入理解系统的经验。测试者对系统如何工作和人们如何使用系统有好的领会，但他们确实不能真正理解系统内部的联系以及对其他特征的影响。

把调试者作为文档移交的评估者是一个新的概念，但有很多的优点。开发团队面临的最大的问题是缺乏对任何人都有价值的文档。例如，一个文档告诉你 C++ 方法或 Visual Basic 类，你的典型文档包含类中的一系列方法，这些类定义了各种类型的信息，并且当类在程序中被调用时这些信息都要被使用。如果你是文档的书写者，你认为这些文档是有用的。在你需要知道时，文档会告诉你需要知道的知识。但这些文档怎样帮助那些确实需要使用这些类而又不知道怎样使用这些类的人呢？怎样帮助那些需要类来完成一些功能，但不知道怎样调用这些类的人？作为一个调试者，很熟悉解决这些问题。这样，你可以对代码指出“用户手册”的需要而不是参考设计。毕竟，如果你知道怎样使用这些类，你总是可以通过查看代码本身知道怎样调用这些类。在这种情况下，转让代码的文档（技术手册）在两个地方复制更

多的信息。另外，因为文档通常是固定的，相反代码是多变的，那么文档能反映现在的信息而不是写文档时的信息就很重要了。

很少有公司真正把代码移交看成软件的一个阶段。作为一个专业的调试者，这是一个向公司展示你的价值的最好机会。遗憾的是，调试者的一生很少有好的回报，所以任何证明你存在的机会你都应当放弃。

15.2.5 项目阶段总结

你可以看到，调试者通常要没日没夜地进行他们的工作。未来的专业调试者将被包含在项目开发的每个阶段，从最初的概念设计到最后的安装。公司应当充分考虑已有的人员和检查他们成为专业调试者的技术。像我们前面提到的，这些技术可以使用在公司工作的任何阶段，并可以替代其他两到三个人的工作。这就节省了费用、时间并产生公司的新一代人才。所有这些都是好事。

当然，下一个问题是：一个好的调试者需要哪些技术？这是我们在这一章中要提到的下一个主题。知道一个优秀的调试者的技术就可以确定你是否可以担任这个工作。在这种情况下，我们检查一个真正优秀调试者的特点并看他是否适合这个位置。

15.3 优秀专业调试者的特点

当你考虑一名优秀的调试者，你应当想到他们的哪些特点呢？对引起问题的原因作出快速决定的能力？在工程要发布前一天能工作很长一段时间的能力？在调试者进行一个特定复杂的问题解决时，可以很好应付的能力？在过去，我们考虑一个优秀的调试者就会遇到这些问题。不过，他们确实需要经验而不是规则。优秀的调试者需要有长段代码调试的能力、保持代码的可维护性、当他们进行系统的修改时要保证不会引入新的问题。导致这些能力的个人特点是什么？通过对众多成为专业调试者的长时间的观察，我们从以下几个方面定义了一名优秀调试者的特点。

15.3.1 老练

调试你自己的代码要求你认识到你自己犯了错误或者不理解问题。调试别人的代码要求你告诉其他人（用各种形式）他们犯了错误。当人们意识到自己有错误就可以更容易解决问题。当我们由于某些问题被责备，我们就变得有抵制情绪并试图打破它。在调试中处理个人的问题需要一定程度的老练。

我的前任管理者告诉我，在你确实感觉到被压的喘不过气来的时候老练是对别人有礼貌的技巧。不要责备或者让别人自我感觉到他们自己的错误，并简化工作，创造出最好的软件是一种能力。老练的调试者可以解决他们的问题，而不用要求创建 bug 的人帮助并对他们大喊大叫。另外，老练不会使软件出现无法发现和不能解决 bug，因为这只是工作的一个简单

部分。

虽然老练是在计算机工业中我们要保持的一个基本特点（和这个问题的其他特点），但对调试者来说是特别重要的。如果你让一个人从专业水平考虑问题，把所有的问题当成代码中的问题而不是程序员的问题，你就创造了一个没有压力的工作环境。使用不恰当的调度时间和不断地修改需求，我们在工作中就有很大的压力。没有理由添加压力。努力用最大程序的老练和尊重来工作，你会发现可以很快地解决问题。

15.3.2 耐性

对于优秀的专业调试人员，耐性是一个最重要的特点。在调试过程中，耐性的作用体现在各种不同的情况下。你要耐心等待直到看到 bug 的出现，要耐心等待直到在作出解决方案之前知道问题的原因。如果你有耐性，你就不会有人多的问题，这一点在调试中很重要。

开发者一个很普遍的问题是在没有充分测试和调试代码的情况下就接着下一个有趣的问题，这就是在代码没有准备好之前就发布的一个原因。测试、调试、文档和分析代码的性能不是一件有趣的事，也不会很迷人。人们不会总是把没有意思的事情做得非常好，这对聪明的人类是条真理。因为很多程序员都是高智商的，他们总是不想做没有意思的事。如果你有耐性，并在整个过程中都是这样的，那么就不会出现严重的问题。另外，如果你有耐性并完成好了工作，那么就会得到鼓励并可以到一个新的、有趣的软件系统中工作，不会被旧的系统拖累。耐性是一个经常受称赞的美德。

在调试过程中，耐性是很重要的。在你解决问题之前你要确保你知道问题，你要在发布系统之前确保已经解决了问题、你要确保你的修改不会影响到系统的其他部分。所有这些步骤需要不折不扣的耐性。对于那些“认为知道他们所作的工作”的人来说，调试不是一个好的工作。如果你要调试，你必须完全按照过程，否则你确实会引起更多的问题。如果你有耐性，而且在每时每刻根据过程进行，你就可以较容易调试并有一群人围绕着你。

在讨论调试过程，耐性也是很重要的。人们通常会忽略描述 bug 的细节，仅仅是因为他们没有看到这些细节的重要性。一些小事，像在 bug 出现之前做的一些事，或者我们试图在 Macintosh 上运行 MS-Windows 系统的事实。可是，这些小事就是发生了。通常需要二到三次来告诉 bug 的观察者。如果你没有耐性来听用户说，并根据你的第一个影响做出结论，你会发现你自己变得毫无目的并且浪费很多的时间和精力。

15.3.3 探测技巧

通过读前面一章的内容，我们知道了调试过程中探测技巧的重要性。问题是什么是探测技巧？想一下历史上的众多探测者：Sherlock Holmes、Charlie Chan 和 Scooby Doo 等等，他们有什么共同点？他们注意了其他人忽略的细节，在别人以为没有办法时他们坚持了，他们试验了不可思议的事。当你在读探秘小说时，你就看到真正的调试精英的工作。

在调试时，你应当把自己当成是一个神秘的探测者。在你观察 bug 的故障现象时，应当

确保对遇到的每个现象仔细观察，因为每个现象都是有价值的线索。试图去理解犯这个错误的人的动机，也就是创建 bug 的人的动机。很多 bug 的产生是因为缺乏耐性或者缺乏对问题的理解。确保你对问题的理解，如果你这样做了，你可以预想到如何解决这个问题；如果这样做，就可以知道程序员为什么犯了这样的错误。如果你仅仅注意 bug 的表面现象，那么就会丢失一些有价值的线索。

在大量神秘的谋杀事件中，你总是能发现其中的证据；在软件中的 bug 上，你发现的 bug 故障现象经常是覆盖真实问题的正面现象。确保你在捕获 bug 之前挖掘深度的问题并保留所有的证据。

探测的另一个问题是在错误现场做了哪些修改。有些很令人困惑的错误被解决，因为人们注意到有些事是不同寻常的。Sherlock Holmes 狗不叫就是这个问题的典型例子。同样，如果一个系统突然出现错误，而有这个系统的输出日志，你就应当检查这个日志来发现是否有东西发生了变化。甚至很多微小的不同也是解决神秘 bug 的重要线索。

15.3.4 处理压力的能力

如今的世界，每一个修改需求和收入指数使压力成为软件工程师一天生活的重要部分。对于调试者，这个压力更重，因为一个 bug 在引起其他更多的问题前就要被立刻解决。贸易杂志和每日新闻有大量的关于公司 Web 站点崩溃和他们的软件出现严重漏洞的故事。这就要求有人来解决这些问题，这些人就是项目的调试者。调试一个软件需要 6 个月，另一个实时系统的调试需要每天 24 小时不停地进行。如果你工作在这样的环境中，你就不要考虑什么是压力或者对开发团队有什么影响。

压力对开发者和调试者的两个影响需要提出来。第一，压力使他们变得急躁。这个联系是有害的，因为这样会减少他们的耐性，这样的结果是错过了重要的线索。第二，压力使他们对整个工程没有了兴趣，而对摆脱压力的来源更感兴趣。这个问题的原因是因为人们不想工作在一个没有鼓励的项目中，并期望完成好的工作。如果你不关心项目的收入，你就不可能做得最好。

在有时间限制的情况下，如何减少解决问题模式下的压力？一个办法是暂时离开问题一段时间。你可以解决另一个问题，或者让自己休息一段时间。慢跑练习是缓解压力的最好办法。有很多不同的让你放松的方法，并可以使你下意识地解决问题。

你对自己和你的能力越自信，你就越没有压力；你越能控制项目的进度和需求的漂移，就会产生越少的压力。如果你是一个管理者，你应当知道工程项目的动机。这些鼓励，例如奖金和额外的假期，对项目的开发人员是次要的动机。压力来自项目的结束时，但仅仅是因为现在的开发工作。在理想的环境下，我们要在最后期限的两个星期前完成项目，并有额外的时间对下一个发布进行测试和计划。如果你安排你的人员一天工作 12 个小时，那么同时工作几天或几个星期就可以完成。人们在长期没有压力的情况下不可能有高的效率。一旦压力存在，就很难避免了。

15.3.5 工程的/科学的方法

对调试者来说，工程的方法是很重要的。在你一开始观察你要解释的现象时就使用了这一方法。一旦你观察事件，你就应当使用理论解释所有这些你所观察到的现象，使他们不会出现在其他的产品中。接着应当测试这个理论并验证结果。如果测试结果体现了这个理论，你就要进一步测试并预测另一些输入的输。如果这个输出也被证实，你就可以假设你的理论可以用来解释你所看到的事件。

对专业的调试者，这意味了什么？我们已经多次提到，在我们调试一个系统时，你必须根据这个过程。这个过程就是一个科学的方法。首先，有一个假设来测试你的假设，得出一个解决方法，实现这一方法，接着测试这个方法。这不是一个快速发展的科学，但事实上它是一个快速发展科学的基础。没有办法在短时间内循环这一个系统，并快速得到结论；也没有办法避免观察或者作出假设。任何想跳过某一步骤的专业调试者对他的选择都感到困难，并被他的结论所吓倒。

我们在这本书上强调的一点是过程是重要的。过程越好，越可靠地执行，结果就越好。当然，过程在定义时要考虑新的知识。总是会问到以下这个问题：你首先在哪里开始这个过程？对于调试者，这个问题的答案就是科学的方法。开始这个过程，并在过程中增加步骤。例如，你可能从收集系统特定的输入数据来测试你的假设。如果你是在一个标准的，面向用户的系统中，这个方法就很好。不过，当你的工作对象是一个实时的嵌入式操作系统，又会怎样呢？你可以在模拟器中进行分支测试，但这不是一个理想的解决方法。科学的方法要求用最实际的设置来观察问题。由于这个原因，你可能修改这个过程，用一些特定的硬件产生真实的数据，你在目标系统之前就作出了这个修改。接着，测试新的输入数据形式是用不同方法修改数据的一个简单问题。这样，你就创建了一个可以用来假设测试的过程。

如果你想成为世界级的专业测试者，你就必须根据以下规则。不用说，在你的职业生涯中，你可能不会按照某些步骤进行，你也根本没有正式想过它们。人们严格按照规则是很普遍的事，但在这个过程中会忽略很多步骤。当然，也只有好的过程中，人们才会严格执行这些步骤。例如，当你在处理其他部分时，可能会遇到已经出现的 bug，总是不关心这个故障现象可能会引起其他的问题。作出明智的假设并通过增加代码到系统中来验证你认为会发生的事是否真的发生。这是对优秀的调试者和成功的调试的一个基本要求。当一个科学的方法变得根深蒂固，以至成为习惯，你就会发现你自己已经成为一名真正意义上的专业调试者。

15.3.6 忘我

如果某人没有缺点，最有可能受自我的影响。自我使你不愿意听别人的话，自我无法进行交流因为你觉得你知道的更好，没必要听别人的话。所有这些，自我使我们看不到我们的错误，我们会因为自己的问题去责备别人，我们会因为“不重要”或“用户不会看到这些问题”而错过了 bug 和错误。所有的 bug 都是严重的，有些人在有些地方因为自我，就会很苦

恼地去查找 bug，并对系统作出不好的评价。

在工作中怎样避免自我？最好的避免自我的方法是保持你严格的职业态度，这一点对很多人来说都是很困难的。有些职业，例如程序员，他们希望专心在他们的工作上，很难有什么成就和情绪的成分。不过，只能把工作当成工作。它不是你的生活，不能这样对待它。如果你不能理解这一点，你就要注定对你的工作失望。

你怎样才意识到你有自我的情绪？一个好的办法是后退一步，像别人看你一样看自己。你是否对逻辑的建立和解决过程争论过？如果没有，可能就是你不喜欢争论，因为你认为别人的想法没有你的好。这就是自我的表现。如果你没有使用一些个人的攻击或无礼的评价，例如“这从来就不能工作”，就不能驳倒一些人的论点，那么你就是自我的。当然，这样做是很自然的事，因为自我仅仅是你自我保护的一部分。问题是当你由于自我允许自己退步，就会因为你失去目标而损害了你的工作质量。不时停下来看看自己的工作，你是否做了一些令自己满意的事？你是否做了一些破坏别人的事？你是否做了一些你真正相信正确的事，并做出了逻辑考虑的结论？直到你能对最后一个问题做出肯定的回答，对前两个问题做出否定的回答，你才可以确认自己没有自我的想法。

15.3.7 坚定不移

在很多情况下，坚定不移和耐性是相同的，都要求坚持一些事并把它做好。但坚定不移意味了要坚持一些事，直到你认为它不能被解决或者其他人告诉你只需要一个固定的简单解决方法；坚定不移意味了要有合理的最好解决方法。某种意义上，顽强就是最好的解决方法，不用考虑其他方面的事，例如需花费的代价和资源。

有时，坚定不移会给你带来麻烦。我曾经在一个公司工作过，这个公司为机顶盒制造卫星电视接收软件，我们在调试一个 bug。这个 bug 相当严重并影响了用户的观看。对这个特定的工作，没有比破坏电视节目观看更严重的问题。我在这个问题上花了一个半月的时间，在这段时间内，我担任了一个新的项目领导者，我的作用是联系团队和高层领导者。这个项目领导者急于向管理阶层表现自己，并跳过某些问题来“帮助”解决问题。遗憾的是，他选择的一个解决方案远不是最好的方法。我尽自己最大的可能反对这个决定，但最后还是不得不按他的要求做出修改。最后，这个问题变得严重了，又出现了其他的问题，每个人都不高兴。如果我坚持我的观点并排除我的自私，我就会让别人知道这个解决方法是不正确的。

坚定不移带来的好处可能不会立刻显示出来。如果你坚持自己的观点，而不答应别人对问题的不好或错误的解决方法，你就可以避免“我告诉你这样”的问题的讨论，在你谈论某些事并且这个替代方法不能工作时就会出现这个问题。当这个失败出现，那么一个被强迫改变他的想法的人就会说“我告诉你这样”。当然，这是自己防卫，因为所有这些使原来的人觉得这个方法不好，强迫他进入一个防卫的模式。

当你有了耐性、执着和无私的品质，你就会注意你的行为。人们也会相信你的行为，因为他们知道这些行为是经过合理考虑和深刻的分析，而不是一时的期望。你的工作将会是最

好的，因为你从总体上注意了项目，而不是为了你个人的进步。因为你的项目将会成功，你仍然可以提升的，虽然你没有这样想。有时，好人会有好报的。

15.4 专业调试者的一天

在这一章结束之前，看一看一个专业调试者一天的生活。这不是理想的一天，但确实是相当标准的一天生活。你在工作任何的一星期中，可能有过这样的生活经历。调试者的生活不像你在电视中看到的那样华丽、固定。很少有任何大的变化，也没有任何可怕的事情要我们加夜班来完成这些事情。不是深夜的调试工作不会发生，而是因为这不是个标准。一个典型的调试者的一天工作应当如下：

8:00：开始工作，收邮件，查找新的问题。在我的邮箱里有两个新的 bug。一个比较严重，另一个不很严重。我继续阅读我的邮件，过一会来调试 bug。

8:15：我读完邮件，调出运行了一个晚上的程序，检查调试日志来验证是否出现问题。接着扫描日志来确定我昨天晚上的程序是否有问题。所有这些都显示正常。

8:30：我倒了一杯水，接着阅读给我的对系统提出的新的功能需求。我在可能出现问题的地方做了一些记录，接着编辑电子文档，把这些注意增加到记录中。我把这些文档发送给在一个星期前提出需求的人。

8:45：我开始了今天的工作，查看我昨天遇到的 bug。我已经通过跟踪代码来查找错误分支的原因，这个分支错误是在用户正常的操作下产生的。我们的系统，一个机顶盒的卫星电视接收系统，可以让用户配置一系列他们想看的电视频道。这个特征，调用一个 Favorites 列表，允许用户自定义频道的显示，这样他们喜欢的频道就在最上面。遗憾的是，虽然这个列表在现有的安装特征上可以工作，但在保存列表时就会出现错误。我查看了几个调试日志，但不知道为什么程序会出现这种状态。我在代码中增加了新的调试语句并重新编译系统。

9:15：我为系统下载了一个新的软件包，这花费了我一些时间，所以我去找了一个质量保证人员，和他讨论几天前听到的另一个问题。我觉得我在这个问题上得到的信息足够多，我可以使用这些信息进行 bug 的解决，我讨论重现问题的几个步骤和结果。我同意测试者的说法，这是一个 bug，鼓励他写下他给我的所有细节。

9:45：重新进行 bug 调试，我重新运行测试来重现列表问题。在这种情况下，我增加了跟踪代码来发现系统运行的步骤，因为我不能从所写的代码中发现任何的错误。回顾调试的输出，我发现一个我所假设为真值的布尔变量，实际上它的值为假。重新回到代码中看，我进一步发现这个变量在非常严重的错误条件为真的情况下，它就被设成假。因为我不知道为什么会设置这个错误条件，我跟踪代码观察可能设置这个标志地方。

10:15：我修改代码来解决问题。引起这个错误的原因是错误理解 C++ 类的调用。我假定这个实现数据和时间包装的类在初始化阶段没有输入时的缺省值是“今天”。结果证明这是个错误的假设。这个类根本没有缺省值，这就导致了检查数据的程序以为时间是在之前发

生的，而不是现在发生的。这个错误标志就被设置，用户就得到了错误的显示。我正确地初始化了这个变量，重新编译代码，安装系统，并运行一系列的回归测试验证这个问题的解决。

10:45~11:30：我运行一系列的测试验证问题已经被解决，验证我的修改没有影响到系统的其他部分。代码工作正常，并且没有修改必要，我用对我修改的解释检查了代码。接着我更新 bug 数据库说明这个 bug 已经被解决了，指出所做的修改，并说明 bug 的解决对其他部分的影响（我们的下一个版本）。好了，在继续调试前，我上了一会儿网。每个调试者都需要回报的。

12:00：我参加了一个关于系统状态的讨论，我们必须在这个星期之内解决这个问题。总的来说，这是一个令人讨厌的问题。我们没有发现任何新的问题，所以这个会议基本上和上个星期的会议相同。有时，我觉得会议是没有必要的，因为会议的安排在商务活动中是一个最大的时间浪费。

13:00~15:00：在 bug 数据库中我回顾了剩余部分的 bug，我选择了一个新的 bug，这个问题是包含了一组相关的记录事件，检查他们是否有重叠的部分。我首先设定了两个记录，每隔 30 分钟，观察它们是否重叠。结果表明没有重叠发生。我接着设定另两个相互关联的事件，仍然没有发生任何的重叠现象。这样，我就在 bug 记录上表明这类 bug 是不会再出现的，并把它从未调试列表中删除。下一个 bug 的情况是这个 bug 只在星期五发生。因为今天是星期四，就不可能让 bug 出现。最后，我解决一个简单的被可用性报告的 bug。它在页面上显示一个时钟，和系统的其他页面一样。这样的 bug 有什么困难吗？

16:00：由于代码缺少文档和例子，我放弃了调试以上这个 bug。我花了几分钟的时间回顾了我今天的工作，并为明天的工作做了准备。保存好我今天的工作并在服务器上做了备份，我整理了我的桌子，收拾了公文包回家。

16:30：离开办公室。

这就是一个专业调试者一天的生活，他们的生活不是很迷人的，不是吗？听起来很动人，像那种可以年复一年的工作吗？如果你喜欢做正常的开发者，这些问题的答案可能是不。不过，如果你喜欢一些和我一样的冒进工作，当你发现 bug 的根本原因并解决它，那么一个专业调试者的生活可能就是你所想要的生活方式。

这是典型的一天的生活，不是特别的一天。有时专业调试者的一天工作可能都是失败的——新的 bug，旧 bug 都没有被解决，并且过去解决的 bug 可能引起新的问题。有时，情况又会变好。代码中存在几个月的 bug 被解决了，客户很高兴，写了一份电子邮件来祝贺你的成功。不过，这种情况很少发生。调试者的生活和有规律的程序员生活一样，找出问题，开会，喝咖啡。

在这一章中和这本书中，你能得到的最重要的事实就是，调试不是一个黑色的艺术。调试不是天生的技巧，也不是你能简单地学会的。调试是一门科学。既然是科学，就应当被看成是不可完全相信的，在这个领域中出现新的技术和发现时，就会有畏惧感。和生物学或地理学一样，不可能解释怎样解决任何或所有的 bug，在编写代码之前，没有万能的方法来预测代码中的 bug。调试是一门年轻的科学，在确定最好的解决 bug 方法时，可能会犯错误。

15.5 小结

调试者的一天生活和其他的计算机专业人员的生活没有什么区别。如果把调试者看成是真正的专业人员，而不是开发者被迫要完成的简单工作，那么我们在软件开发过程中就有一个好的过程。软件开发的每个过程都应当有调试者参加，因为他们的经验对软件开发的每个阶段都有价值。从最初的项目概念设计到最后的数据交互使用，专业调试者在软件开发过程中起了不同的作用。

15.6 bug 问题

这一章的问题是因为缺乏对问题的考虑。这是回答问题的惟一线索。

ADW (Acme DotCom Works) 公司为人们提供了一个特殊的服务。人们在 ADW 服务器的数据库上存下重要的信息，并可以通过网络在世界上的任何一个地方访问这些信息。也就是，这个公司提供电子记事本。

不要求用户在登录时提供惟一的用户名和密码，ADW 公司提出了一个较好的想法。他们考虑到人们可能会忘记神秘的用户名，特别是要求惟一的用户名的情况下。毕竟，在世界上有成千个叫 Smiths 的人。安排了一个为解决这个问题的讨论会。

项目的高级程序员建议使用社会安全号码 (Social Security Number, SSN) 解决这个问题，几个理由否定了这个建议，有个原因就是人们不喜欢分享他们的 SSN。他们回到会议室进一步讨论问题。经过几个小时的讨论，喝了无数的咖啡和茶水，一个开发者提出了使用电话号码当登录名。几乎每个人都有电话，他们觉得合理。毕竟，电话号码是惟一的。所以，ADW 就采用了电话号码作为登录系统的号。客户也比较能接受使用电话号码登录系统。

在系统使用的前几个月能完好运行，人们不会忘记他们的用户名，对少数忘记密码的人可以回忆起来或者通过电子邮件来告诉他们。公司的生意越来越好。晴天霹雳，问题出现了。

一个客户代表 (Customer Service Representative, CSR) 从银行接到一个电话，好像有个客户不能登录系统来接收他的信息。另一个电话从相同的银行打来，接着很多很多的电话。没有一个客户可以访问他们的数据。

首先，CSR 认为问题的原因是这个银行执行了防火墙，但结果表明不是这个原因。另一个郊区银行打电话来说，这是一个保险的设施。接着，公司的很多用户不能接收他们存下的信息。管理者对这个问题很恐慌。开发者对这个问题也很恐慌，因为他们不知道问题的原因。这个问题只能交给作为调试者的你。

为了解决这个神秘的无法访问记录问题，你必须从公司设计者的最初始的假设开始思考问题。把问题考虑一遍，向自己提出一些他们回答的问题，并看你的答案是否和他们的答案相同。这个问题主要测试系统设计的问题，在这个例子中没有程序的错误。

附录 A bug 问题的答案

在调试领域，发现一个问题的答案是整个过程的目标。但到达这个目标是很痛苦而且有时是不可能的过程。有时候你永远也不知道是什么导致了一个 bug，你只是在证明你所修复的到底是什么东西。到现在，你明白这不是好事，而且你要总知道你所要修复的 bug 是什么原因导致的。

对每一章的问题，你都会得到答案或一些可能从中得到的教训。

第 1 章

问题的答案在于 Web 服务器和存储股市信息的数据库之间的连接。当程序员发现数据库连接的每个问题时，由于错误形式的请求或语法错误，他们面临着操作人员重新启动数据库以重新建立连接的问题。并非 Web 服务器在丢失其连接，只是它们自己“挂起”以处理数据库查询的错误。

你怎么知道这一点呢？将两个组一起研究，而不是分开来，就会发现程序员正在寻找语法错误。同时，操作人员正观察到从数据库到 Web 服务器之间的给定连接正等待请求并超时。组合这两个因素，你就能够发现问题。

问题的答案？两组之间更好地通信，以及一个告诉每一方对方正在做什么。

第 2 章

bug 的问题描述和系统的体系结构描述为你推导这个问题的潜在的原因提供了充分的线索。

“页面被正确加载 34 次，错误 12 次”，这个描述的暗示如下：

- ◆ 显示页面的数据的产生源不正确，因为如果产生源不正确，页面就会一直被错误地加载。观察将数据库原因排除了。
- ◆ 如果数据库的内容正确，那么肯定是应用服务器提供了错误的数据。所有的应用服务器都产生错误的数据？不像，因为我们的观察表明只有四分之一的页面不正确（每三个正确的就有一个不正确的）。这个观察表明有一个应用服务器发生了错误，因为系统中有四台应用服务器。

我们如何证明有一个应用服务器产生了不正确的页面这个假设呢？如果你有相关的底层结构，这将没有什么问题。你只需直接向应用服务器发送请求，看它的结果。但是，没有直接询问应用服务器的能力，就很难直接证明这个假设。但我们可以假定这个假设正确，这样就能证明部分应用服务器出错了。

按照这个想法，bug 存在于应用服务器的代码中吗？对应用中的程序的版本的检查证明版本都相同。所以，应该不是版本问题。那么到底是什么问题呢？

我们回顾问题的描述，寻找第二个线索。“页面的内容已经过期”说明内容是基于时间的。而且由于应用服务器缓存了数据，可能是缓存的数据不正确。如果你有从应用服务器取出缓存的能力，这个问题就可以通过直接看缓存来证明。假设你证明了缓存的内容不正确，你可以通过删除 cache 解决这个短期的问题，而且可以让应用服务器从数据库中重新加载数据。但这只是解决表面问题：真正的原因是什么？代码中有 bug 吗？环境中 bug 吗？

我们一般首先相信最简单的原因，而且在这个例子中我们检查了计算机系统的时间。发现出现错误的应用服务器上的系统时间晚了几个小时。由于缓存是基于时间的，有错误时间的应用服务器认为缓存中的数据还很新，所以不需要加载新的数据。

第 3 章

对快速消息传递的观察，明显地指出一些内存破坏问题。不同的特征都是在点击 URL 之后的内存内容导致的结果。系统使用的内存已经被破坏了。基于哪个应用程序被影响，会产生不同的行为。

内存破坏的原因很难判断。但这里最可能的情况是缓冲区溢出。这是因为已经观察到接收 URL 并没有什么不好的影响。还没有点击 URL 的时候就已经发生了问题。点击之后会怎么样呢？按照程序，快速消息传递必须从屏幕上提取 URL 并保存定义 URL 的链。快速消息传递就把这个链向浏览器发送。如果用来保存这个链的缓冲区太小，URL 会重写缓冲区之外的内存，从而破坏了机器上的存储器的内容。这就是我们对 bug 的原因的最初假设。

第 4 章

聪明点的读者可能会立即想到问题和网络有关，因为我们给出了系统的网络布局（这里重新画出，如图 A.1 所示）。但是潜在的硬件问题的另一个线索可以从试验的结果中发现。这个线索是响应时间的变化，因为软件的缺陷是一致的、可预测的。但你仍然要排除潜在软件的原因并识别特定的硬件错误。怎么做呢？我们用于分离问题的步骤如下。

第一，可以排除后台服务器的任何问题，因为后台服务器的时钟与产品和开发环境中的都是一致的。如果问题在于软件，那么肯定是在 Web 服务器上的软件中。

第二，要保证性能与系统的负载无关。这样，定时实验就运行在空闲的系统中。

第三，要排除 Web 服务器和执行 JSP 的 Servlet 引擎的潜在的影响。这意味着我们必须使用执行着与 JSP 相同的请求的简单的 Java 应用程序进行相同的实验。为了完成这些，我们在产品 Web 服务器上和产品后台服务器上运行这个 Java 应用程序。我们也在开发 Web 服务器上 and 开发后台服务器上运行这个 Java 应用。实验的结果不变：所以，问题应该在于网络。

最后，我们还要排除客户方 API 和 Java 虚拟机的可能问题。这要求使用与上面描述相同的 Java 应用程序。我们在产品 Web 服务器上运行这个 Java 应用程序，与开发后台服务器通信。这个实验的结果与在开发 Web 服务器上运行这个 Java 应用并与开发后台服务器通信的结果一致（即性能一样快）。这告诉我们客户方 API 和 Java 虚拟机没什么问题。而且实验证明产品 Web 服务器和产品后台服务器之间的通信有问题。

网络操作人员做了一些检测工作之后，最终发现产品后台服务器上使用的开关与防火墙之间的电缆有问题。它使得报文在机器间消失导致响应时间的延长甚至无法预测。更换电缆后，产品站点的性能与开发站点的性能就一样了。

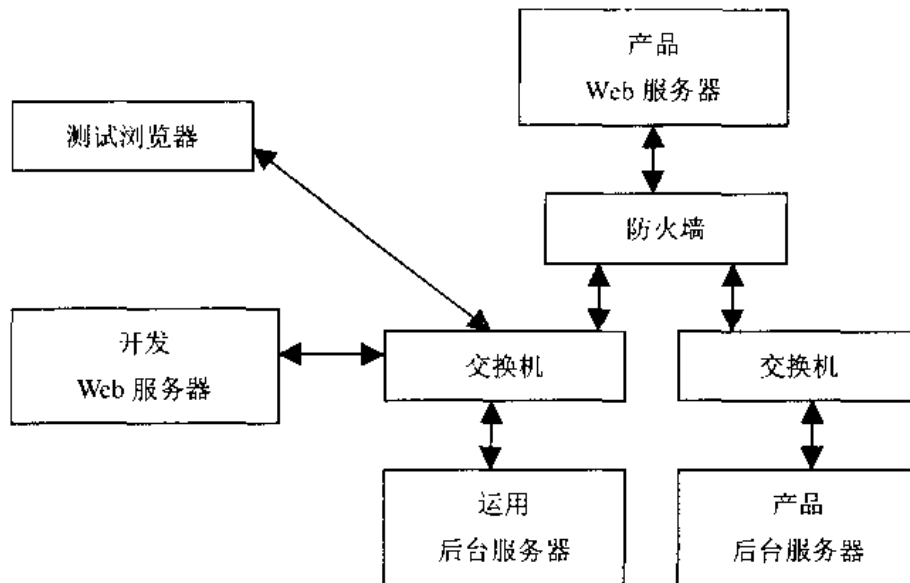


图 A.1 第 4 章中的 bug 问题中所描述的系统概念的、体系结构的和网络的布局

第 5 章

答案实际上在问题描述中已经给出了。问题很简单，装置在填充分区，导致其重启和删除那个分区。剩余空间足够时为什么还会这样呢？很简单。这个正在填充的分区并不是包含视频信息的那个。实际上，问题在于单索引分区。当这个分区填满时，装置试图为下一段写更多的信息，或修改当前这一段的索引，但发现没有空间，所以导致错误。

这里的问题是选择固定的索引分区的尺寸时没有考虑到潜在的问题。假设每个视频分区都有自己的索引，有一个主索引能定位每个视频段的起始索引（这可能只是一个小文件，不会填满分区但谨慎的人会去扩展），那么当初就不会发生问题了。

问题的修复有许多方法。建立一个新的索引分区或将原来的加倍并复制原来的所有数据——这样会解决问题，却不能避免再次发生。选择这个方法的原因在于其简单。我们要尽快将一些东西送到用户手里，同时要维护他们现有的装置和数据。方法被记载下来，同时记载了所设计的假设和这个问题可能再次发生的注释。代码中加上了附加的调试信息说明这个事情，并警告程序员可能会再次发生。这不是一个真正的“调试”，但它比起一个完美的方法来说对于现实世界更加真实。

第 6 章

问题是由于备份改变引起的。程序在备份给定的资源时锁定了它，而如果用户同时使用这个程序和资源，这个程序将因两个进程间的争夺而使应用系统崩溃。

识别这样的错误通常是一个排除过程。你可以发现正在运行的进程之间的所有共同点，

然后检查每一个以看出到底是什么发生的问题。一旦发现了那个出现问题的进程，你将依次排查进程中的每一个步骤直到发现失败的步骤。发现有问题的步骤之后，你就可以使用标准的调试过程来发现问题。

在这个例子中，你可以通过在不同进程的日志中查看每个步骤找出所使用的资源来发现问题。知道了所使用的资源后，你就可以采取步骤修改进程，或者在其他时候运行它以进行修复。

第 7 章

不可能将一个调试器或编译器加载到成品机器中，所以这些工具没什么用处。测试一致性只在你知道问题的根源时有用。是对网页的错误调用吗？这只要对输入和期望输出用 PIPE 分析就可以检测到。如果是这个问题，你就会很快发现的。否则，可能是输入的边缘条件的原因。测试新的网页，看看通过了什么数据，然后你就可以诊断日志信息以发现错误之处。最后，如果都没用，你可以跟踪程序发现在何处崩溃。通过对跟踪和对函数、过程、方法的输入进行日志之间的组合，跟踪这个 bug 应该没有什么困难。

另一个可能性是软件的新的加载暴露了内存缺陷。使用一个内存使用跟踪系统查看是否有新的内存缺陷，是否用尽了内存。最后，你可能要去询问网络开发人员（你的应用程序的用户）是否发现了一些能对你的调试有所帮助的东西。

使用这些方法，你早晚会发现 bug。如果你尝试对机器上的软件用 bench 进行测试，那么如果它是实时集的数据导致的话，你就可能无法发现这个 bug。第 10 章讨论过产品应用的调试。

第 8 章

这里有几个可能的问题。首先，由于 ZIP 代码在两个不同的表中重复，可能一个 ZIP 代码被加到了用户表或者在里面升级了，而且可能没有加进税务表或者税务支付表。既然设计没有可参考的完整性，就不可能判断这是对的还是错的。简单地查看数据库，看有没有在一个表中而不在另一个表中的，就能发现了。

这里的其他问题可能更加基本和复杂。ZIP 代码的确不是数字表。它们看起来像是，因为目前它们只包含数字。当你真正看这些数字时问题就来了。ZIP 代码可以、而且确实是从 0 开始的，当一个 ZIP 代码从一个串转换为数值时，0 被省略。为返回到 ZIP 代码的初始形式，你应该能将数值再格式化为软件中的串版本。这是一个常常被忽视的问题。

最后，关于对程序设计界的美式中心的观点会引起 bug 是没什么意义的。在 ZIP 代码的例子中，我们只考虑 US ZIP 代码，里面只有数字。其他国家，如加拿大，ZIP 代码里既有数字又有字母。一个采取纯粹数字的 ZIP 代码卖到另一个国家当然会出现问题。

第 9 章

证明的缺陷是在任何对用户登录的检查完成之前用户的数量已经增加了。如果登录失败，那么仍然要增加一个，这样总共实际登录的用户数量就成了 0。

安全风险 1：如果你告诉用户一个用户名或密码无效，你就告诉黑客什么是有效什么是

无效的危险。最好不要告诉他们无效的用户名或者密码。

安全风险 2: 通过查看多个登录, 如果不检查他们是否从同一机器或地址登录的, 你可能面临一个用户登录后丢失自己的设置而自动退出的问题。最终导致的问题与风险 1 相同。

第 10 章

这个问题很简单。文件的名字在负责写的程序和负责读的程序中是不同的。文件永远也没有读, 变量永远也没有被设置。所以负责读的程序中的变元没有被初始化, 结果不可预测。

注意, 在写的程序中: 各字是 TODAYSDATA.DAT

在读的程序中: 名字是 TODAYSDATA.DAT

第一个名字中有 0, 而第二个中有 O。

第 11 章

代码中有些缺陷, 而且它们可能会对程序带来严重的危害。这些缺陷是:

- ◆ 在第 12 行, 用户名分配的指针与 `argv[1]` 变元相同。第 16 行, 我们改变了 `username` 变元所在的存储区的内容。但由于 `username` 与 `argv[1]` 有相同的指针, 我们就不可避免地修改了 `argv[1]` 的内容。这个改变很不妙。

- ◆ 第 4 行定义的变元 `firstname` 的长度是 20。但第 18 行 `strncpy()` 函数没有检查文件名和用户名的长度是否小于 20, 这将导致潜在的 `firstname` 变元缓冲区溢出。

- ◆ 在第 16 行, `index()` 函数的返回值用 `NULL` 来检查。可能程序的用户忘记在开始的名字和最后的名字之间加上一个空格。这时, 程序会将 `NULL` 赋值给一个空指针。

- ◆ 在第 18 行, `strncpy()` 没有将 `null` 字符加入 `firstname` 字符串中, 这样后面的 `firstname` 变元的内容就不正确了。

- ◆ 在第 19 行, `ptr` 指针的增加没有考虑有效性。例如, 如果用户在命令行参数中输入这样的名字: `Joe`, 这个名字会使指针增加到一个没有意义的内存部分去。

- ◆ 在第 21 行, 最后的名字应该分配一个特殊的字符而不是 `strlen(ptr)` 的返回值。

- ◆ 在第 22 行, 使用 `malloc` 动态分配后的 `lastname` 变元的有效性没有检查过。如果 `malloc()` 失败, 后面的赋值就会导致 bug。

- ◆ `lastname` 变元不是很自由的。我们知道一旦程序退出, 分配给 `lastname` 的存储区将被释放。但这个习惯不好, 为什么还要坚持呢?

还有其他的缺陷。例如, 程序不去掉开始和结尾的空格字符, 但程序的需求描述中可能并没有要求处理首尾的空格字符, 那就很难责怪程序满足了需求。

第 12 章

可以预测, 在客户代码中使用了一个函数或方法从服务器得到种类列表。这个取得种类列表的函数可以是客户 API 的一部分。在 GUI 上有两种方法显示种类列表。GUI 只是按照需要调用 API, 而不是用函数一次性取出数据并用数据做出左边式的分类列表和下拉式菜单。这导致两个对服务器一样的请求。

表面上, 这不是一个严重的问题。为什么要担心两个一样的请求呢? 它是低效的, 还有

什么？

假设客户是一个 Web 浏览器，每次加载网页时，都对服务器发送两个请求，而实际只需要一个。在互联网上，会有成千上万个请求，而且用户可能会同时点击你的服务器。从系统的角度讲，低效性将系统的能力减半了。通过更好地设计客户方，用相同数量的机器就能处理两倍的负载。

第 13 章

在这个例子中，方法实际上在问题描述中已经给出了。当宏被扩充时，结果会是一个与开发人员所想象的不同解析的表达式。

预期的结果是：

```
random()% ( j - i + 1);
```

实际产生的等价于：

```
(random()% j ) - i + 1;
```

解决办法就是只按我们的要求去做。只需将宏改成一个有保证的顺序：

```
#define Random(n) random() % (n)
```

有时候，这是非常不错的方法。

第 14 章

这里的症结在于开发组的成长的原因。添加新的开发人员有两个原因：产生新的应用系统和维护现有的应用系统。如果你的公司雇佣 20 个开发人员开发新的应用系统，那么你的抱怨就没有理由。因为你需要人维护现有的应用系统，必须有另外的开发人员来开发新的应用系统。

另一方面，如果所有的 30 个开发人员都去做维护工作，那么对他们所做的维护工作的类型的理解就显得很重要了。在第 14 章我们已经把维护工作分为三种，让我们看看每一种的含义：

- ◆ 他们做的大多是适应性的维护工作。就是说他们在调整应用系统。这和让新的工作人员工作于新的应用系统类似，这可能没有问题。

- ◆ 他们做的人多是完善性的工作。后来的开发人员在添加新的功能并改变设计以使得系统更快更稳定。这些并不是坏事。

- ◆ 他们做的大多是校正性的工作。这时候你应该会抱怨，而且事情那么好了，当需要新的开发人员来修复 bug 时，说明你们产生 bug 的速度超过了他们修复的速度。Bug 产生率的提高有许多原因。可以是因为客户增加或者你的应用系统的体积增加。客户和业务量增加后，就更加容易暴露使程序崩溃的缺陷了。也可以是因为对应用程序的维护性的改变。你添加一个特征，就增加了 bug 宿主。你修复一个 bug，可能会导致另一个失败。所以你落后了，为了保持应用系统的稳定性，添加了许多东西以发现和修复 bug。但开发人员越多，由通信、概念和处理问题导致的错误就越多。这是一个内在的不稳定的不可衡量的情况，也是软件不可维护的特征。关键性错误的出现以及系统的崩溃只是时间问题。而且从业务角度讲，不可

维护的系统减少了公司的潜在的利益——考虑 10 个人与 30 个人维护之间的比较。

第 15 章

当你考虑这个问题时，可能会错过真正的问题。推理的缺陷很简单。电话号码根本不是一致的。我们只是那样想起它们。在多数人的公司里，有一个主号码，以及一个向不同的分支发送呼叫的 **PBX** 系统。但每个分支都有基本的号码。

当大公司申请了这个服务之后，他们将公司的主号码作为用户名。该公司的所有员工都是这样的。当每个人使用相同的号码时，系统只是用新的入口代替旧的。为什么呢？因为电话号码不能根据程序的逻辑复制。由于没有副本，程序员就在用户再次添加了新的号码时强制升级，指出他们忘记了他们的密码并且再次创建账户。

附录 B 附加的阅读资料

我们已经提及了与 bug 和调试相关的许多不同的题目。由于各题目的复杂性和本书的范围，我们不得不忽略许多重要的主题和思想。在这个附录里，我们列举了一些书和杂志，它们可能会对你提供一些帮助，使你更加理解本书中所讨论的一些主题。

Bug 相关的事件

Neumann, Peter G. *Computer Related Risks*. Addison-Wesley, New York, New York, 1995.

Peterson, Ivars. *Fatal Defect: Chasing Killer Computer Bugs*. Vintage Press, New York, New York, 1995.

设计

Brown, William H., et al. *AntiPatterns*. John Wiley and Sons, New York, New York, 1998.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns*. Addison-Wesley, Reading, Massachusetts, 1995.

编程

Kernighan, Brian W., and Pike, Rob. *The Practice of Programming*. Addison-Wesley, Reading, Massachusetts, 1999.

Maguire, S. *Writing Solid Code*. Microsoft Press, Redmond, Washington, 1993.

McConnell, Steve. *Code Complete*. Microsoft Press, Redmond, Washington, 1993.

Meyers, S. *Effective C++: 50 Specific Ways to Improve Your Programs and Designs*. Addison-Wesley, Reading, Massachusetts, 1998.

调查

Gilb, T., and Graham, D. *Software Inspection*. Addison-Wesley, Reading, MA, 1993.

Wheeler, D. A., ed., Brykeczynski, B., ed., and Meeson, R. N., ed. *Software Inspection: An Industry Best Practice*. IEEE Computer Society, 1996.

维护

IEEE Standard 1219-1992. *Software Maintenance Standard*. Published by IEEE Standards Office, P.O. Box 1331, Piscataway, New Jersey, 08855-1331.

Pigoski, T. M. *Practical Software Maintenance: Best Practices for Managing Your Software Investment*. John Wiley and Sons, New York, New York, 1996.

Smith, D. S. *Designing Maintainable Software*. Springer-Verlag, 1999.

方法学

Beck, Kent. *Extreme Programming Explained*. Addison-Wesley, Upper Saddle River, New Jersey, 2000.

Meyer, Bertrand. *Object-Oriented Software Construction*., 2d ed. Prentice-Hall PTR, Upper Saddle River, New Jersey, 1997.

Rumbaugh, J., et al. *Object-Oriented Modeling and Design*. Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

需求工程

Robertson, Suzanne, and Robertson, James. *Mastering the Requirement Process*. Addison-Wesley, Reading, Massachusetts, 2000.

Weigers, K. E. *Software Requirements*. Microsoft Press, Redmond, Washington, 1999.

重用

Jacobson, I., Griss, M. and Jonsson, P. *Software Reuse: Architecture Process and Organization for Business Success*. Addison-Wesley, Reading, Massachusetts, 1997.

Karlsson, Even-Andre, ed. *Software Reuse: A Holistic Approach*. John Wiley and Sons, New York, New York, 1995.

Szyperski, C. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, Reading, Massachusetts, 1999.

软件工程

Arthur, L. J. *Rapid Evolutionary Development: Requirements, Prototyping & Software*

Creation. John Wiley and Sons, New York, New York, 1992.

Boehm, B. *Software Engineering Economics*. Prentice Hall, Englewood Cliffs, New Jersey, 1981.

Florac, W. A., and Carleton, A. D. *Measuring the Software Process*, Addison-Wesley, Reading MA, 1999.

Gilb, T. *Principles of Software Engineering Management*. The Bath Press, Avon, Great Britain, 1988.

Jones, Capers. *Assessment and Control of Software Risks*. (Yourdon Press Computing). Prentice Hall., Upper Saddle River, NJ., 1994.

Kan, Stephen H. *Metrics and Models in Software Quality Engineering*. Addison-Wesley, Reading, Massachusetts, 1995.

Putnam, Lawrence H., and Myers, Ware. *Measures for Excellence: Reliable Software on Time, within Budget*. Prentice-Hall, Englewood Cliffs, New Jersey, 1992.

测试

Beizer, Boris. *Software System Testing and Quality Assurance*. Van Nostrand Reinhold Electrical/Computer Science and Engineering Series, Pennsauken, New Jersey, 1984.

Kaner, Cem, Nguyen Hung Quoc, and Jack Falk. *Testng Computer Software*. 2d ed. John Wiley and Sons, New York, New York, 1999.

Maguire, Steve. *Writing Solid Code*. Microsoft Press, Redmond, Washington, 1993.

软件处理

Florac, William A. and Carleton. Anitas D., *Measuring the Software Process*. Addison-Wesley, Reading, Massachusetts, 1999.

Humphrey, Watts S. *Managing the Software Process*. Addison-Wesley, Reading, Massachusetts, 1989.