The Pragmatic Programmers

TURING 图灵程序设计丛书

Debug It!

Find, Repair, and Prevent Bugs in Your Code

软件调试修炼之道

[美] Paul Butcher 著
曹玉琳 译

- 亚马逊全五星畅销图书
- 资深专家经验总结
- 问题重现→问题诊断→缺陷修复→反思

人民邮电出版社
POSTS & TELECOM PRESS

TURING 图灵程序设计丛书

Debug It!

Find, Repair, and Prevent Bugs in Your Code

软件调试修炼之道

[美] Paul Butcher 著
曹玉琳 译

人民邮电出版社
北京

版 权 声 明

前　言

我一直想不明白，为什么关于调试的书这么少。其他软件工程方面的书可谓汗牛充栋，设计、代码结构、需求获取、方法学，林林总总，然而有关调试方面的书却既没有什么人写，也没有什么人出，我希望这本书有助于改善这种情况。

如果写代码，那么在某些时候肯定要调试代码（也许很快就需要调试）。调试比其他任何过程都更需要动脑，它不发生在调试器或代码中，而是形成于大脑之中。找到并理解问题的根源，才能进行其他的工作。

多年来，我有幸在多个软件领域与许多出色的团队共事。我曾做过位片处理机微编码的所有抽象层次上的工作，涉猎过设备驱动程序、嵌入式代码、主流台式机软件和网络应用程序。我希望能够通过这本书总结一些我从同事那里得到的经验教训。

关于本书

本书分为三部分，每一部分都详细阐述了调试的某一方面。

第一部分　问题的核心

这部分介绍了实证方法，即借助软件特有的功能向我们展示这是怎么回事儿，以及建立在实证方法之上的核心调试方法（问题重现、问题诊断、缺陷修复、反思）。

第二部分　从大局看调试

我们怎样发现存在需要修复的问题？又怎样将调试融入到更广泛的软件开发过程中去？

第三部分　深入调试技术

这部分将关注一些高级的话题。

- 尽管本书前面讨论的方法适用于所有缺陷，但某些类型的缺陷还是需要特别对待。
- 调试应该早早启动，不要等到受困的客户愤怒地打电话来才开始调试。我们能够提前采取什么措施和流程来解决客户可能提出的问题呢？
- 最后讨论如何避免一些常见的缺陷。

致谢

直到开始写这本属于自己的书时，我才认识到致谢这一节多么重要。我的名字得以出现在封面上，但是没有很多人给我的帮助和对我的宽容，这本书是不可能完成的。

感谢参加本书电子邮件讨论列表并提供灵感、批评和鼓励的每一个人：Andrew Eacott、Daniel Winterstein、Freeland Abbott、Gary Swanson、Jorge Hernandez、Manuel Castro、Mike Smith、Paul McKibbin 和 Sam Halliday。要特别感谢 Dave Strauss、Dominic Binks、Marcus Grber、Sean Ellis、Vandy Massey、Matthew Jacobs、Bill Karwin 和 Jeremy Sydik，他们允许我将他们的轶事和见解与你分享。还要感谢 Allan McLeod、Ben Coppin、Miguel Oliveira、Neil Eccles、Nick Heudecker、Ron Green、Craig Riecke、Fred Daoud、Ian Dees、Evan Dickinson、Lyle Johnson、Bill Karwin 和 Jeremy Sydik，感谢他们花费时间为本书做技术审查。

感谢我的编辑 Jackie Carter，他耐心地指导我，让我这个第一次写书的人摸到了写书的门道。感谢 Dave 和 Andy 给了我这次机会。

我要向工作在 Texperts 的同事们致歉，他们不得不忍受我喋喋不休一味地谈论这本书的内容（不要担心——我很快就会有一辆新的赛车，那时你们又必须忍受我喋喋不休地谈论它了）。还要向我的家人致歉，为我在漫长的夜晚和周末不能和家人在一起而道歉，感谢他们对我的支持。

最后，感谢所有我能够荣幸地与之共事的人。软件开发生涯中最好的一面就是软件人的器量，在一个好的团队中工作是一件特别的幸事。

Paul Butcher
2009 年 8 月
paul@paulbutcher.com

目　录

第一部分　问题的核心

Part 1 第一部分

问题的核心

本部分内容

第 1 章

山重水复疑无路

如果你的软件不能正常工作，该怎么做呢？

一些开发人员似乎有窍门能够准确无误地找出发生缺陷的根本原因，而另一些开发人员似乎总在漫无目的地寻找原因却得不到确切的结果。是什么使他们之间存在着这样的差异呢？

在这一章中，我们将仔细探究一种调试方法，这种方法在专业的软件开发中已经被反复证实。它绝非灵丹妙药，它仍然依赖于调试者的智慧、直觉、探查缺陷的技巧，甚至一点儿运气。但是，它会使你的努力更加有效，避免做无用功，并且能够尽快地找到问题的核心。

具体来说，我们将介绍以下内容：

- 调试与排除缺陷的区别；
- 实证方法——借助软件本身来告诉你现在的运行状态；
- 核心调试过程（问题重现，问题诊断，缺陷修复，反思）；
- 做最先应该做的事——在深入调试之前应该先考虑的事情。

1.1 调试不仅是排除缺陷

试问一个没有经验的程序员什么是调试，他可能回答调试就是“找到一种修复缺陷的方法”。事实上，这仅仅是调试诸多目标中的一个，甚至不是最重要的目标。

有效的调试需要采取以下步骤。

(1) 弄清楚软件为什么会运行失常。
(2) 修复这一问题。
(3) 避免破坏其他部分。
(4) 保持或者提高代码的总体质量（可读性、架构、测试覆盖率、性能等）。
(5) 确保同样的问题不会在其他地方发生，也不会再次发生。

其中，目前最重要的是首先查明问题的根本原因，这是一切事情的基础。

理解万岁

一些没有经验的开发人员（遗憾的是，有时就是我们中本应知道得更多的那些人）经常完全忽略问题诊断这一过程，而是往往立即采取他们认为能够修复程序的措施。如果他们走运，只是修改了的程序运行不了，他们所做的一切是在浪费时间而已。真正的危险是修改了的程序可以运行，或者看来可以运行，因为这时开发人员在一知半解的情况下改变了源程序。他或许修复了缺陷，但是实际上很可能掩盖了潜在的根本原因。更糟糕的是，这种改变可能会导致新问题——破坏一些曾经可以正确运行的程序。

浪费时间和精力

几年前，我在一个拥有很多具有丰富经验的天才开发人员的团队中工作。他们的经验大部分来自 UNIX 操作系统，但是当我加入这个团队时，他们正处于将软件移植到 Windows 操作系统的后期阶段。

在移植的过程中，他们发现了一个缺陷，就是同时运行多个线程时出现了性能问题。某些线程突然死掉，而另外的线程在正常运行。

所有的程序在 UNIX 操作系统下运行正常，问题很显然是 Windows 操作系统破坏了线程，因此他们决定定制一个线程调度系统，以避免使用操作系统所提供的调度系统。显然这是一项很繁重的工作，但是我们的团队有能力完成这项工作。

我加入这个团队时，他们正在以各种方式实现这项工作，果然，线程不再突然死掉。但是，线程调度是难以捉摸的，即使变化引起了很多问题（比如使整个系统运行变慢），这些线程仍然能够继续工作。

我对这个缺陷很好奇，因为在以前的 Windows 线程编程中，我从来没碰到过这样的问题。略作调查后我们发现，性能问题事实上是由于 Windows 实现了*动态线程优先*。这个缺陷其实通过禁用一行代码就可以修复（即调用 `SetThreadPriorityBoost()` 函数）。

这件事情告诉了我们什么？该团队没有深入调查所看到的现象，就下结论，认为 Windows 线程被破坏。在某种程度上，这可能是文化问题，Windows 在 UNIX 黑客中声誉不佳。不过，如果他们花点儿时间找出问题的根源，就能省去大量的工作，避免引入使系统效率低下并易产生错误的复杂因素。

如果不首先弄清楚缺陷的真正根源，我们就没有遵循软件工程的原则，而是一头扎进*巫毒编程*（voodoo programming）[①]或者*巧合编程*（programming by coincidence）[②]里了。

1.2 实证方法

可以采用很多不同方法去实现你所探查的目标。只要你选择的方法让你更接近目标，它就达到了目的。

话虽如此，在大多数情况下，一种特别的方法——实证方法，是迄今为止最有效的方法。

> **构建实验，观察结果。**
> *Construct experiments,and observe the results.*

实证依赖的是观察和经验，而不是理论和纯逻辑推理。就调试而言，这意味着直接观察软件的行为。是的，你可以阅读全部的源代码，并用纯理论去了解软件的运行状况（有时你可能没有其他选择），但是这样做通常没什么效率，而且非常危险。通过仔细地构建实验环境并观察软件的运行状况，你可以更有效地找到问题。这样做不仅仅更快捷，而且通过观察可以使你重新审视自己关于软件运行还有哪些错误的假设。软件本身就是你的工具箱中最强有力的工具——就让它来告诉你运行状况是什么样的吧。

软件的本质

软件是非常了不起的产品。有时，或许是因为我们一直与它一起工作，所以才会忘记它是多么地了不起。

人类的经验中有一小部分具有可塑性，允许我们毫无限制地自由发挥聪明才智和创造力。当然，软件是确定性的（有少数例外，后面会提到）——下一个状态完全取决于当前的状态；更为关键的是，只要需要，我们就可以进入各个状态。

相对于传统的工程学，我们太幸运了。你认为一名 F1 工程师会瞬间停止每分钟 19 000 转的引擎来检查它的每一个细节吗？例如，他能查看每个组件在压力下的精确状态，动态记录点火时燃烧室内前方火焰的位置和形状吗？

而我们却能够凭借这种技巧检查我们的软件，这就是为什么在调试时实证方法特别强大的原因。

① “在晦涩难懂、令人发指的系统、程序或算法中，人们不能真正地理解，使用的是一种猜测或手册方式。其含义是，该技术可能无法工作，如果它没工作，人们永远不知道是为什么。”摘自 *The Jargon File*[ray]。

② 见 *The Pragmatic Programmer*[HTOO]。

下一节中描述的方法将会利用实证法去提供一个结构化的手段来集中找出缺陷。

1.3 核心调试过程

调试过程的核心包括以下四步。

问题重现 找一个可靠并简洁的方式来按需求重现问题。

问题诊断 提出假设，并通过实验来测试它们，直到找出引起缺陷的潜在原因。

缺陷修复 设计和进行一些修改来修复问题，不要引入回归问题，保持和提高软件的整体质量。

反思 吸取教训。哪里出了问题？是否还有其他类似的问题需要修复？怎样做才能确保同样的问题不再发生？

调试是一个反复的过程。
Debugging is an iterative process.

如图 1-1 所示。一般来说，这些步骤是按顺序执行的，但这并不是严格的“瀑布”模型。虽说你肯定不希望直到重现了问题或者设计了修复计划后才启动诊断，而后才明白所发生的问题，但这是一个迭代的过程。在问题诊断中学到的知识可能会告诉你如何提高重现问题的水平，或者在软件修复中学到的知识可能会使你重新考虑你的诊断结果。

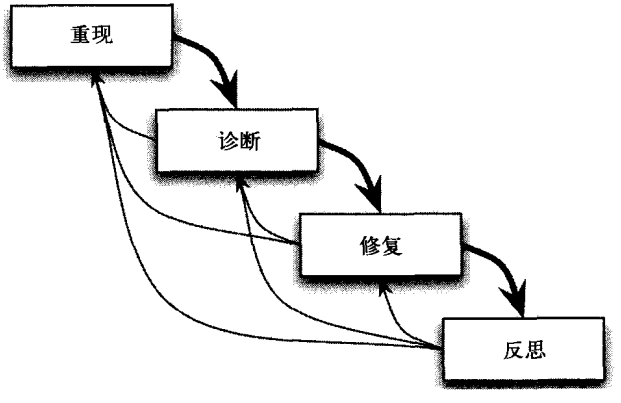

图 1-1 核心调试方法

在下面的章节中，我们将详细研究这些步骤。而在此之前先来了解一些基本原则。

1.4 先澄清几个问题

径直深入讨论可能更诱人，然而还是值得花点儿时间来说说一些基本原则，以确保之后的讲解能够顺利进行。

1.4.1 你知道要找的是什么吗

发生了什么？应该发生什么？
What is happening, and what should?

在开始重现问题或者推测问题产生的原因之前，你需要明确地了解到底发生了什么。同样重要的是，你还要知道正常情况下应该发生什么。假如你是根据一份正式的缺陷报告来工作，那么这个报告应该包含了所有你需要的信息。（我们将在第6章详细讲解缺陷报告。）花点儿时间仔细阅读一下，确保你充分理解它。

假如你没有正式的缺陷报告（可能你正在解决一个备感困惑的缺陷或者这个缺陷是在不经意间提交的），那么更应该先暂停你的工作，要确保在继续工作之前真正地了解整个程序。

请记住，缺陷报告本身的错误不会比其他文档更少。仅仅是因为缺陷报告说应该这样运行而不应该那样运行，就能真正地符合软件的规格说明书吗？如果它没有明确地说明应该怎样运行，那么在你彻底弄清楚之前不要做任何改变——否则有可能会把正确的改为不正确的，仅仅听信缺陷报告是不会有帮助的。

不要轻信缺陷报告

曾经有一次，我修改一个非常简单的缺陷——生成的报告说程序没有考虑夏令时间，因此当时钟变化时出现了错误。我快速地修复了此缺陷，然后转向解决下一个问题。

然而稍后出现了另一个缺陷，缺陷是账目不能收支平衡。报告生成的数量与我们从供应商那里得到的发票的数额不一致。

果然，事实证明，这些票据没有考虑夏令时间，从而导致了差异的产生。再查以前的记录发现，我们一年前就已经遇见了这个问题，那时我们故意忽略了夏令时间。[①]

显然，这个问题不是软件没有听我们指挥，而是我们自己不知道想让软件做什么。因为报告是在不同的环境下使用的，在某些情况下，夏令时间是需要考虑的，而另外一些情况下，夏令时间是不需要考虑的。正确的解决办法是在报告中增加选项，允许用户选择。

① 顺便说一句，最开始对源程序做出改变的开发人员如果能够加上简单的代码注释，说明为什么在这种情况下忽略夏令时间，让我们明白是有意忽略夏令时间的，就会减少很多麻烦。

1.4.2 一次一个问题

有时，面对几个问题时往往会并行处理，这似乎挺带劲。如果缺陷都发生在同一区域，这种做法尤其常见。但是请不要这样做。

调试一个缺陷已经很困难了，使情况更为复杂是没有必要的。无论怎么小心，你为了查找一个缺陷所进行的实验很有可能会以某种方式诱发另外的缺陷。这样很难弄清楚究竟发生了什么。此外（正如我们4.5节所讲），你最终要签入修复程序时，会希望每做一次逻辑修改就签入一次，而如果你同时修改多个缺陷，那么这就很难实现。

有时，你会发现你认为一个缺陷是由某个原因引起的，而事实上它却是由多个原因引起的。通常情况下，当你处于思维迟钝的状态时，这点更明显——奇怪的是这种现象似乎没有明确的解释。要进一步探讨，你可以参阅3.4节。

1.4.3 先检查简单的事情

很多缺陷是由于简单的疏漏而引起的。因此，有时你会面对非常微妙的事情，但请千万不要忽略简单的事。

由于某些原因，开发者似乎有一种感觉——不得不亲自做一切事情。在NIH（Not-Invented-Here，非我发明症）中表现得最为明显，这种感觉使得我们想要自己实现程序，但其实在其他地方已经存在一个很好的解决方法。这种错误的观念在调试方面的表现就是你必须亲自调试你遇到的每一个问题。

问问团队中的其他成员，是否以前也遇到过类似的问题，这样做可以大大降低成本，而且很有可能避免浪费大量的精力。如果你工作在一个并不熟悉的领域，这样做尤其重要。

Subversion之困惑

肖恩·埃利斯

这周，我的一个新伙伴正在受SVN export问题的困扰。这是一个致命的打击——服务器及其工作站拥有相同版本的SVN，但是状态却不相同，存在很多潜在的问题。

终于，他崩溃了。他问我，用这个特有的命令是否有问题，并剪切粘贴了SVN中的命令行给我。

“是的，有问题。”我说。Apache库中有这样的缺陷，在路径中用../../..标记是错误的。两秒钟后，我们确定这就是真正的问题所在。几分钟后我们确定了服务器端有

不同版本的 Apache 运行时 DLL。

当然，几个月前很多潜在的问题就相继发生，而这是第一次发现这个缺陷。

因此，沟通总是重要的——不沟通容易陷入奇怪、微妙、令人生厌且难以描述的方式中。古语说得好："站起来，问问是否有人以前见到过此类问题。"

下一章，我们将详细讲解这个过程的第一步——重现问题。

1.5 付诸行动

- 一定要做到以下几点。
  - 找到软件运行异常的原因。
  - 修复问题。
  - 避免破坏其他程序。
  - 保持或提高软件整体质量。
  - 确保不在其他地方发生同样的问题，确保这种问题不重复发生。
- 利用软件自身来告诉你发生了什么。
- 每次只解决一个问题。
- 确保你知道自己要找的是什么。
  - 正在发生什么？
  - 应该发生什么？
- 先检查简单的事情。

第 2 章

重现问题

正如上一章所述，运用实证方法进行调试可以充分利用软件的独特能力，来告诉你软件运行的状态，而发挥这种能力的关键是找到能够重现问题的方法。

在这一章中，我们将介绍以下内容：

- 为什么重现问题是如此重要；
- 如何对你的软件施加必要的控制，找到一个重现问题的方法；
- 怎样才能做好问题重现，以及如何通过反复优化来达到这个理想目标。

2.1 重现第一，提问第二

为什么重现问题如此重要？因为如果你不能重现问题，那就几乎不可能取得进展。原因如下。

- 实证过程依赖于我们观察存在缺陷的软件执行的能力。我们应该首先设法使软件表现出它的缺陷，如果不能，那么这就如同我们丢失了军火库中最强大的武器。
- 即使你设法提出一个为什么软件可能出现缺陷的理论，但是如果你不能重现该缺陷，那么又如何证明你的理论呢？
- 如果你认为你已经实施了软件修复，那你又如何证明确实解决了问题呢？

不仅重现问题很关键，而且更为关键的还在于这是你应该做的第一件事情。如果未能成功地重现该问题就开始修改源代码，那么你所做的修改可能会掩盖一些问题或者带来其他的问题。[①]

那么，究竟如何开始这个调试的关键阶段呢？

① 这类似于测试优先开发中的原则，没有一个失败的测试，你就不应该写任何新的代码。在这种里，你的“失败的测试”就是被重现的缺陷。