漂亮的调试

——Andreas Zeller

*“我叫Andreas，我曾有过调试的经历。”*欢迎来到Debuggers Anonymous。

在这里，你可以讲述自己的调试故事，并从其他人的故事中找到安慰……你是不是又没有在家里睡上一觉？还好你只是在调试器前度过了一晚上。你还是无法告诉你的经历何时才能改好这个程序？让我们多往好的方面想想吧。隔壁工作间的同事吹嘘连续花了36小时的时间查找一个bug？这确实令人难忘！

*……不，调试并没有什么可炫耀的。*它是我们工作中的丑小鸭；是一个还远未完善的任务；是一种最不可预测或无法解释的行为。如果说程序中的缺陷是一种犯罪，调试则是相应的惩罚。

漂亮的调试

假设我们已经竭尽全力以防止错误的发生，但有时仍发现调试的必要。与所有其他工作一样.....

我们需要以最专业和最漂亮的方式来处理调试。

那么，调试中是否存在着漂亮性呢？

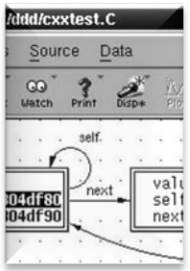
——Andreas Zeller

漂亮的调试

答案当然是肯定的。

数学科学学院
高艺00801016

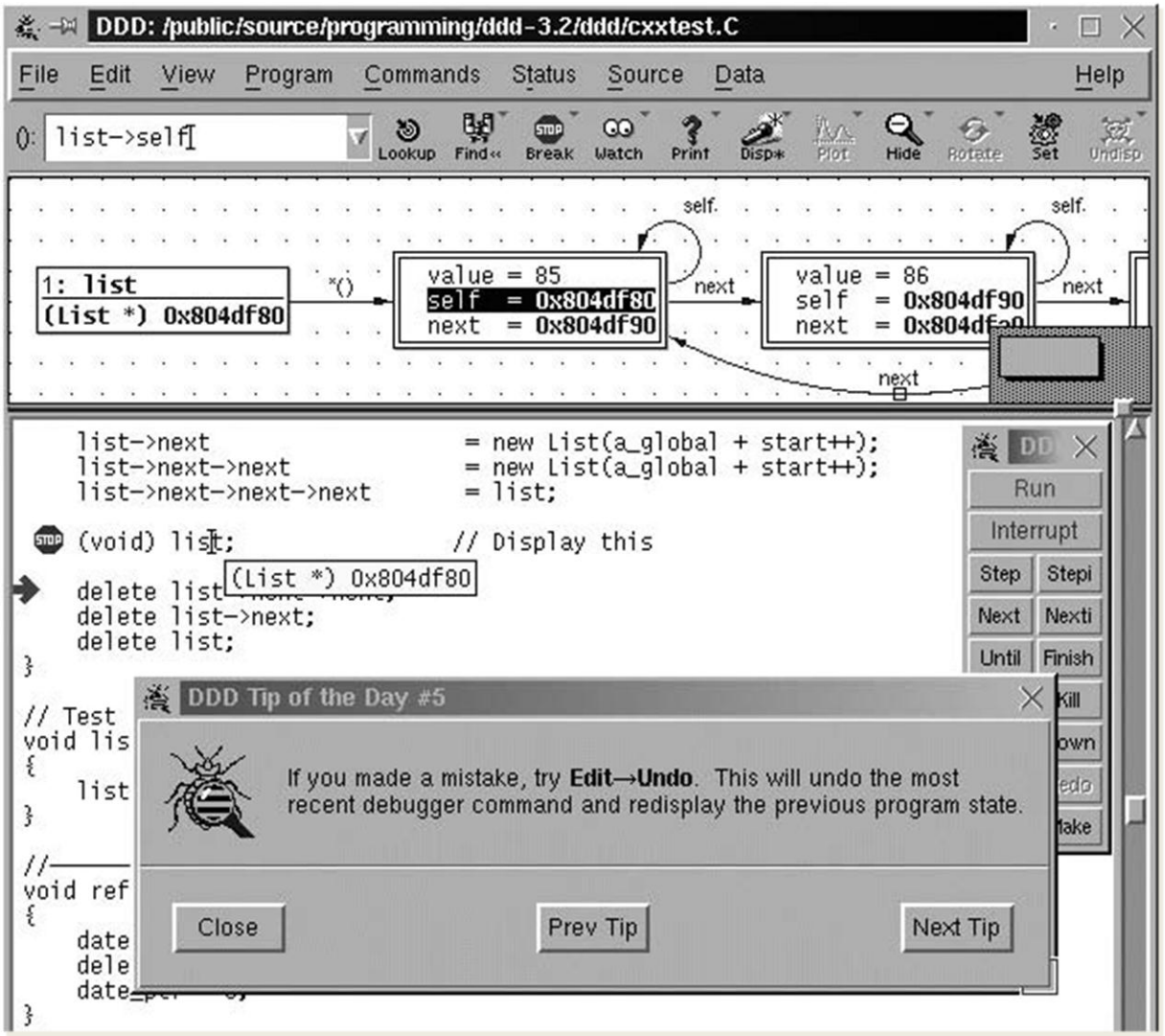
D · D · D



1994年，Dorothea Lutkehaus在她的硕士论文中构建了一个可视化的调试器界面，用来为数据结构提供规范的视图。这个软件叫做Data Display Debugger，简称*ddd*。

这个调试器令人印象深刻：可以在几秒钟内解析复杂的数据，并以直观易懂的方式展示出来，且仅通过鼠标就可以浏览和操纵这些数据。

具体而言，ddd对强大却难以使用的gnu调试器gdb进行了包装。由于当时图形化的编程工具很罕见，因此ddd可以算是个小小的革命。后来，在本文作者，也就是Dorothea的导师和她的努力下，ddd成为十分漂亮的调试器界面，并且最终成为了GNU系统的一部分。



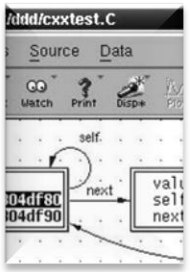
调试中的漂亮性其一：DDD

DDD实用截图，展示了一个循环链表视图

虽然用ddd来进行调试比用命令行工具通常更加有趣，但他无法使你成为一个更高效的调试人员。因此.....

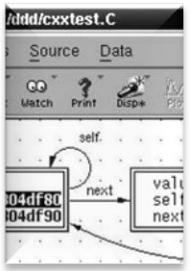
D · D · D

调试过程远比调试工具重要。



ddd虽使调试变得漂亮，但流于表层，带给编程者更多的是直观性与趣味。在本质上与最基础底层的调试没有区别。

D · D · D



那么，何谓真正的调试漂亮性？

所有的事情起源于1998年7月31号一个关于ddd的bug报告……（原作者废话有点多）

bug产生的原因是gdb从4.16至4.17的版本更新，ddd作为一个前台程序与新版本的gdb产生了“兼容性”问题。说白了，两个版本间的代码差异是关键。

作者在diff中运行了两个版本的代码库，结果显示：存在差异的代码大概有178200行，这是非常庞大的：gdb的总代码行数也就是大约600000行。开发人员至少在8721个位置上对源代码进行了改动。这些差异对于一个次要发布版本来说太多了。以一己之力手动查找与处理遭遇困难……

数学科学学院
高艺00801016

增量调试

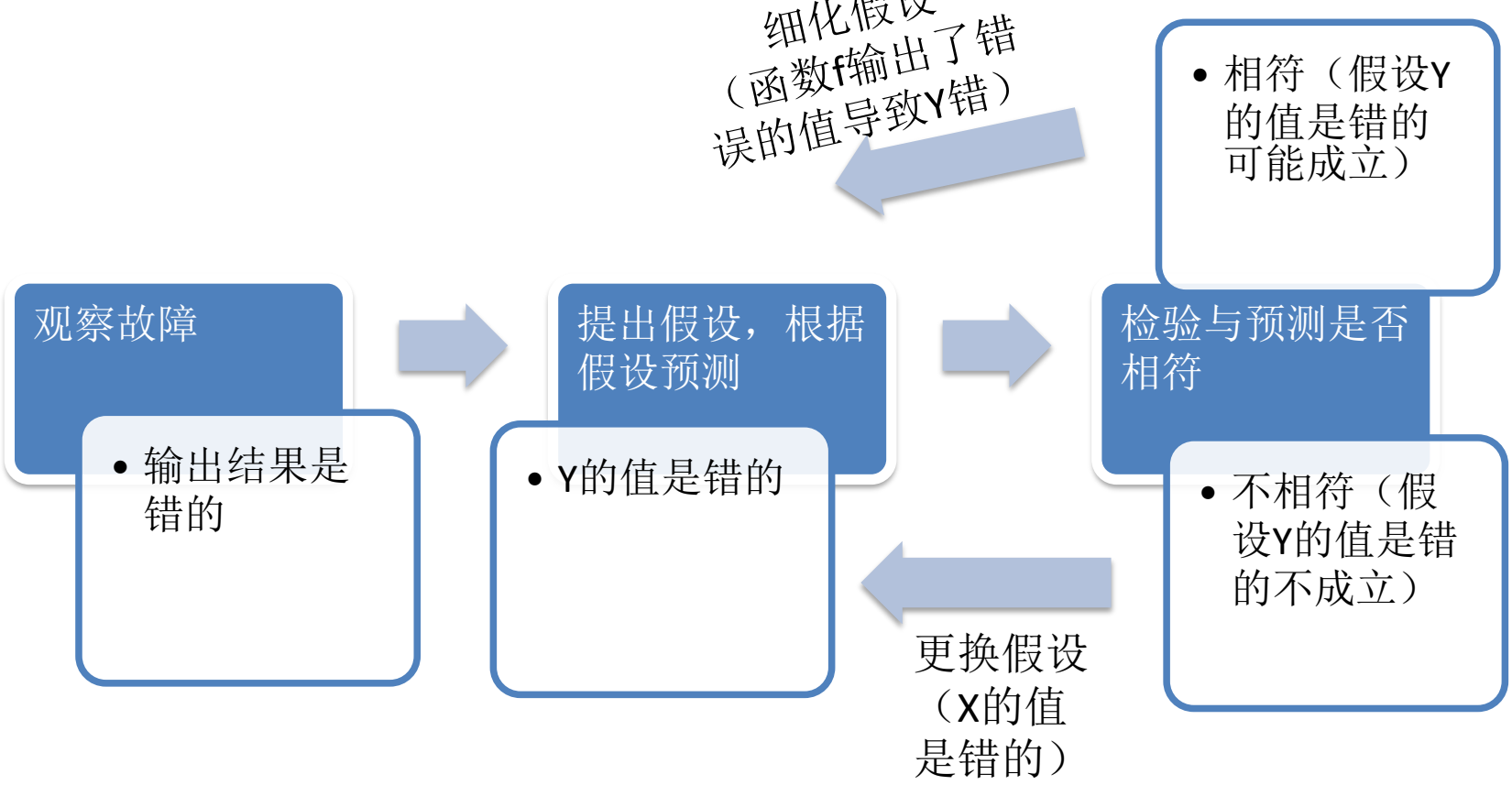
那么，首先，一般的调试过程是怎样的？

程序员调试程序时，*会查找导致问题的起因*（可能存在于代码中，输入数据或运行环境中）。只有消除了所有导致问题的起因后，程序才可能正确运行。查找起因的一般过程叫做科学方法，其工作方式如下：

1.观察程序故障。
2.对于观察结果一致的故障起因做一个假设。
3.通过假设来进行预测。
4.通过实验堆预测进行判断并作进一步观察。
  a.如果实验和观察满足预测，则对假设进行细化。
  b.如不满足，更换假设。
5.重复3&4直至假设不能被细化。

最后一个被验证的假设即明确指出问题所在。

数学科学学院
高艺00801016



坚持使用科学方法是成为调试大师的关键因素。

增量调试

回到之前的问题。对于那8721处改动，有没有自动化的、也就是真正漂亮的方法来找出问题所在？

所谓增量调试，就是试图将大的改动分解为小的“单位”改动，并从初始状态逐步赋以改动的“增量”，找到问题所在。

但如果仅仅是逐个应用每个改动并测试（很像是模拟了4.16至4.17的开发过程），这其中就有一个问题：我不知道应用改动的次序，而这个次序十分重要，因为每个改动都可能依赖于其他改动。例如……（书中例子）

在考虑顺序、逐个应用的情况下，最坏需要8721+8720+…+2+1次测试。当然可以先一次应用较多的改动，在得不到一致构建的情况下再二分缩小增量——此即作者的想法。

数学科学学院
高艺00801016

```
n = 2  # Number of subsets
while 1:
    delta = listminus(c_fail, c_pass)
    if n > len(delta):
        # No further minimizing
        return (delta, c_pass, c_fail)
    deltas = split(delta, n)
    offset = 0
    j = 0
    while j < n:
        i = (j + offset) % n
        next_c_pass = listunion(c_pass, deltas[i])
        next_c_fail = listminus(c_fail, deltas[i])
        if test(next_c_fail) == FAIL and n == 2:
            c_fail = next_c_fail
            n = 2; offset = 0; break
        elif test(next_c_fail) == PASS:
            c_pass = next_c_fail
            n = 2; offset = 0; break
        elif test(next_c_pass) == FAIL:
            c_fail = next_c_pass
            n = 2; offset = 0; break
        elif test(next_c_fail) == FAIL:
            c_fail = next_c_fail
            n = max(n - 1, 2); offset = i; break
        elif test(next_c_pass) == PASS:
            c_pass = next_c_pass
            n = max(n - 1, 2); offset = i; break
        else:
            j = j + 1
    if j >= n:
        if n >= len(delta):
            return (delta, c_pass, c_fail)
        else:
            n = min(len(delta), n * 2)
```

增量调试

增量调试（或其他形式的科学方法的自动化）中很有意思的地方是，它是非常普遍的方法。它不仅可以在一组代码改动中查找导致故障的原因，还可以应用于其他搜索空间中。譬如：程序员可以很容易地通过增量调试分离处导致故障的两个数据间的差异：“导致浏览器崩溃的原因是在第40行的<SELECT>标签”。

很容易修改这个算法，使其返回“最小输入”：“要使浏览器崩溃，只需输入包含<SELECT>的即可。”在最小输入中，每个字符都与故障相关。

对于调试器来说，最小输入是非常有价值的，因为他们使事情变得简单：我们只需分析更短的程序执行和更少的状态。另一个重要、且漂亮的作用是，他们找出了故障的本质（见原文例）。

增量调试

一. 我们希望不仅能够在程序输入或代码*改动*中找出故障起因，还能在实际的源代码中直接找出故障起因。可以利用“程序状态”...

二. 为何不公开、推广此技术？

www.whyprogramsfail.com
www.st.cs.uni-sb.de/dd （增量调试）

漂亮的调试

原文《代码之美》第28章，《漂亮的调试》
作者 Andreas Zeller

数学科学学院
高艺00801016