

第 4 章 对程序进行逐条跟踪

前面我们讲过，发现程序中错误的最好方法是执行程序。在程序执行过程中，通过我们的眼睛或者利用断言和子系统一致性检查这些自动的测试工具来发现错误。然而，虽然断言和子系统检查都很有用，但是如果程序员事先没有想到应该对某些问题进行检查也不能保证程序不会遇到这些问题。这就好比家庭安全检查系统一样。

如果只在门和窗户上安装了警报线，那么当窃贼从天窗或地下室的入口进入家中时，就不会引起警报。如果在录像机、立体声音响或者其它一些窃贼可能盗取的物品上安装了干扰传感器，而窃贼却偷取了你的 Barry Manilow 组合音响，那么他很可能不会被发现地逃走。这就是许多安全检查系统的通病。因此，唯一保证家中物品不被偷走的办法是在窃贼有可能光顾的期间内呆在家里。防止错误进入程序的办法也是这样，在最有可能出现错误的时候，必须密切注视。

那么什么时候错误最有时能出现呢？是在编写或修改程序的时候吗？确实是这样。虽然现在程序员都知道这一点，但他们却并不总能认识到这一点的重要性，并不总能认识到编写无错代码的最好办法是在编译时对其进行详尽的测试。

在这一章中，我们不谈为什么在编写程序时对程序进行测试非常重要，只讲在编写程序时对程序进行有效测试的方法。

增加对程序的置信

最近，我一直为 Microsoft 的内部 Macintosh 开发系统编写某个功能。但当我对所编代码进行测试时，发现了一个错误。经过跟踪，确定这个错误是出在另一个程序员新编的代码中。使我迷惑不解的是，这部分代码对其他程序员的所编代码非常重要，我想不出他这部分代码怎么还能工作。我来到他的办公室，以问究竟

“我想，在你最近完成的代码中我发现了一个错误”。我说道。“你能抽空看一下吗？”他把相应的代码装入编辑程序，我指给他看我认为的问题所在。当他看到那部分代码时不禁大吃一惊。

“你是对的，这部分代码确实有错。可是我的测试程序为什么没有查出这个错误呢？”

我也对此感到奇怪。“你到底用什么方法测试的这部分代码？”，我问道。

他向我解释了他的测试方法，听起来似乎它应该能够查出这个错误。我们都感到很费解。

“让我们在该函数上设置一个断点对其进行逐条跟踪，看看实际的情况到底怎样”，我提议道。

我们给该函数设置了一个断点。但当我们按下运行键之后，相应的测试程序却运行结束了，它根本就没有碰上我们所设置的断点。没过多久，我们就发现了测试程序没有执行该函数的原因——在该函数所在调用链上几层，一个函数的优化功能使这个函数在某种情况下跳过了不必要的工作。

读者还记得我在第 1 章中所说的黑箱测试问题吗？测试者给程序提供大量的输入，然后

通过检查其对应的输出来判断该程序是否有问题。如果测试者认为相应的输出结果没有问题,那么相应的程序就被认为没有问题。但这种方法的问题是除了提供输入和接受输出之外,测试者再没有别的办法可以发现程序中的问题。上述程序员漏掉错误的原因是他采用了黑箱方法对其代码进行测试,他给了一些输入,得到了正确的输出,就认为该代码是正确的。他没有利用程序员可用的其他工具对其代码进行测试。

同大多数的测试者不同,程序员可以在代码中设置断点,一步一步地跟踪代码的运行,观察输入变为输出的过程。尽管如此,但奇怪的是很少有程序员在进行代码测试时习惯于对其代码进行逐条的跟踪。许多程序员甚至不耐烦在代码中设置一个断点,以确定相应代码是否被执行到了。

还是让我们回到这一章开始所谈论的问题上。捕捉错误的最好办法是在编写或修改程序时进行相应的检查。那么,程序员测试其程序的最好办法是什么呢?是对其进行逐条的跟踪,对中间的结果进行认真的查看。对于能够始终如一地编写出没有错误程序的程序员,我并不认识许多。但我所认识的几个全都有对其程序进行逐条跟踪的习惯。这就好比你在家时夜贼光临了——除非此时你睡着了,否则就不会不知道麻烦来了。

作为一个项目负责人,我总是教导许多程序员在进行代码测试时,要对其代码进行遍查,而他们总是会吃惊地看着我。这倒不是他们不同意我的看法,而是因为进行代码遍查听起来太费时间了。他们好不容易才能赶得上进度,又哪有时间对其代码进行逐条的跟踪呢?幸好这一直观的感受是错误的。是的,对代码进行逐条的跟踪确实需要时间,但它同编写代码相比,只是其一小部分。要知道,当实现一个新函数时,你必须为其设计出函数的外部界面,勾画出相应的算法并把源程序全部输入到计算机中。与此相比,在你第一次运行相应的程序时,为其设置一个断点,按下“步进”键检查每行的代码又能多花多少时间呢?并不太多,尤其是在习惯成自然之后。这就好比学习驾驶一辆手扳变速器的轿车,一开始好象不可能,但练习了几天以后,当需要变速时你甚至可以无意识地完成。同样,一旦逐条地跟踪代码成为习惯之后,我们也会不加思索地设置断点并对整个过程进行跟踪。可以很自然地完成这一过程,并最后检查出错误。

不要等到出了错误再对程序进行逐条的跟踪

代码中的分支

当然有些技术可以使我们更加有效地对代码进行逐条的跟踪。但是如果我们只对部分而不是全部的代码进行逐条跟踪,那么也不会取得特别好的效果。例如,所有的程序员都知道错误处理代码常常有错,其原因是这部分代码极少被测试到,而且除非你专门对这部分代码进行测试,否则这些错误就不会被发现。为了发现错误处理程序中的错误,我们可以建立使错误情况发生的测试用例,或者在对代码进行逐条跟踪时可以对错误的情况进行模拟。后一种方法通常费时较少。例如,考虑下面的代码中断:

```
pbBlock = (byte*)malloc(32);
```

```
if( pbBlock == NULL )
{
    处理相应的错误情况;
    .....
}
.....
```

通常在逐条跟踪这段代码时，malloc 会分配一个 32 字节的内存块，并返回一个非 NULL 的指针值使其中的错误处理代码被绕过。但为了对该错误处理代码进行测试，可以再次逐条跟踪这段代码并在执行完下行语句之后，立即用跟踪程序命令将 pbBlock 置为 NULL 指针值：

```
pbBlock = (byte*) malloc(32);
```

虽然 malloc 可能分配成功，但将 pbBlock 置为 NULL 指针就相当于 malloc 产生了分配失败，从而使我们可以步进到相应的错误处理部分。（注意：在改变了 pbBlock 的值之后，malloc 刚分配的内存块即被丢失，但不要忘了这只是在做测试！）除了要对错误情况进行逐条的跟踪之外，对程序中每一条可能的路径都应该进行逐条的跟踪。程序中具有多条代码路径的明显情况是 if 和 switch 语句，但还有一些其它的情况：&&，|| 和 ? : 运算符，它们每个都有两条路径。

为了验证程序的正确性，至少要对程序中的每条指令逐条跟踪一遍。在做完了这件事之后，我们对程序中含错误就有了更高的置信。至少我们知道对于某些输入，相应的程序肯定没错。如果测试用例选择得好，代码的逐条跟踪会使我们受益非浅。

对每一条代码路径进行逐条的跟踪

大的变动怎么样？

过去程序员问过这样的问题：“如果我增加的功能与许多地方的代码都有关系怎么办？那对所有增加的新代码进行逐条的跟踪不是太费时间了吗？”假如你是这么想的，那么我不妨问你另一个问题：“如果你做了这么大的变动，在进行这些改动时可能不引进任何的问题吗？”

习惯于对代码进行逐条跟踪会产生一个有趣的负反馈回路。例如，对代码进行逐条跟踪的程序员很快就会学会编写较小的容易测试的函数，因为对于大函数进行逐条的跟踪非常痛苦。（测试一个 10 页长的函数比测试 10 个一页长的函数要难得多）程序员还会花更多的时间去考虑如何使必需做的大变动局部化，以便能够更容易地进行相应的测试。这些不正是我们所期望的吗？没有一个项目的负责人喜欢程序员做大的变动，它们会使整个项目太不稳定。也没有一个项目负责人喜欢大的、不好管理的函数，因为它们常常不好维护。

如果发现必须做大的变动，那么要检查相应的改变并进行判断。同时要记住，在大多数情况下，对代码进行逐条跟踪所花的时间要比实现相应代码所花的时间少得多。

数据流 —— 程序的命脉

在我编写的第 2 章中介绍的快速 `memset` 函数之前，该函数的形式如下（不含断言）：

```
void* memset( void *pv, byte b, size _tsize )
{
    byte pb=(byte*)pv;
    if( size >= sizeThreshold )
    {
        unsigned long l;
        /* 用 4 个字节拼成一个长字 */
        l = (b<<24) | (b<<16) | (b<<8) | b;
        pb = (byte*)longfill( (long*)pb, 1, size/4 );
        size = size % 4;
    }
    while( size-- > 0 )
        *pb++ = b;
    return(pv);
}
```

这段代码看起来好象正确，其实有个小错误。在我编完了上述代码之后，我把它用到了一个现成的应用程序中，结果没有问题，该函数工作得很好。但为了确信该函数确实起作用了，我在该函数上设置了一个断点并重新运行该应用程序。在进入代码跟踪程序得到了控制之后我检查了该函数的参数：其指针参数值看起来没问题，大小参数亦如此，字节参数值为零。这时我感到使用字节值 0 来测试这个函数真是太不应该，因为它使我很难观察到许多类型的错误，所以我立即把字节参数的值改成了比较奇怪的 0x4E。

我首先测试了 `size` 小于 `sizeThreshold` 的情况，那条路径没有问题。随后我测试了 `size` 大于或等于 `sizeThreshold` 的情况，本来我想也不会有什么問題。但当我执行了下条语句之后：

```
l = (b<<24) | (b<<16) | (b<<8) | b;
```

我发现 `l` 被置成了 0x00004E4E，而不是我所期望的值 0x4E4E4E4E。在对该函数进行汇编语言的快速转储之后，我发现了这一错误，并且知道了为什么在有这个错误的情况下该应用程序仍能工作。

我用来编译该函数的编译程序将整数处理为 16 位。在整数为 16 位的情况下，`b<<24` 会产生什么样的结果呢？结果会是 0。同样 `b<<16` 所产生的结果也会是 0。虽然这个程序在逻辑上并没有什么错误，但其具体的实现却是错的。之所以该函数在相应应用程序中能够工作，是因为该应用程序使用 `memset` 来把内存块填写为 0，而 `0<<24` 则仍是 0，所以结果正确。

我几乎立即就发现了这个错误，因为在把它搁置在一边继续往下走查之前，我又多花了一点时间逐条跟踪了这部分代码。确实，这个错误很严重，最终一定会被发现。但要记住，

我们的目标是尽可能早地查出错误。对代码进行逐条跟踪可以帮助我们达到这个目标。

对代码进行逐条跟踪的真正作用是它可以使我们观察到数据在函数中的流动。如果在对代码进行逐条跟踪时密切地注视数据流，就会帮助你查出下面这么多的错误：

- 上溢和下溢错误；
- 数据转换错误；
- 差 1 错误；
- NULL 指针错误；
- 使用废料内存单元错误（0xA3 类错误）；
- 用 = 代替 == 的赋值错误；
- 运算优先级错误；
- 逻辑错误。

如果不注重数据流，我们能发现所有这些错误吗？注重数据流的价值在于它可以使你以另一种非常不同的观点看待你的代码。你也许没能够注意到下面程序中的赋值错误：

```
if( ch = '\t' )  
    ExpandTab();
```

但当你对其进行逐条跟踪，密切注视其数据流时，很容易就会发现 ch 的内容被破坏了。

当对代码进行逐条跟踪时，要密切注视数据流

为什么编译程序没有对上述错误发出警告？

在我用来测试本书中程序的五个编译程序中尽管每个编译程序的警告级别都被设置到最大，但仍没有一个编译程序对于 `b<<24` 这个错误发生警告。这一代码虽然是合法的 ANSI C，但我想象不出在什么情况下这一代码实际能够完成程序员的意图。既然如此，为什么不给出警告呢？

当你遇到这种错误，要告诉相应编译程序的制造商，以使该编译程序的新版本可以对这种错误送出警告。不要低估作为一个花了钱的顾客你手中的权利。

你遗漏了什么东西吗？

使用源级调试程序的一个问题是在执行一行代码时可能会漏掉某些重要的细节。例如，假定在下面的代码中错误地将 `&&` 输入了 `&`：

```
/* 如果该符号存在并且它有对应的正文名字，  
 * 那么就释放这个名字  
 */  
if( psym != NULL & psym->strName != NULL )  
{  
    FreeMemory( psym->strName );
```

```
    psym->strName = NULL;
}
```

这段程序虽然合法但却是错误的。if 语句的使用目的是避免使用 NULL 指针 psym 去引用结构 symbol 的成员 strName，但上面的代码做的却并不是这件事情。相反，不管 psym 的值是否为 NULL 这段程序总会引用 strName 域。

如果使用源级调试程序对代码进行逐条跟踪，并在到达该 if 语句时，按了“步进”键，那么调试程序将把整个 if 语句当做一个操作来执行。如果发现了这个错误，你就会注意到即使在其表达式的左边是 FALSE 的情况下，表达式的右边仍会被执行。（或者，如果你很幸运，当程序间接引用了 NULL 指针时系统会出现错误。但并没有许多的台式机这样做，至少在目前它们不这样做。）

记得我们以前说过：&，|| 和 ?: 运算符都有两条路径，因此要查出错误就必须对每条路径进行逐条的跟踪。源级调试程序的问题是用一个单步就越过了 &&、|| 和 ?: 的两条路径。有两个实用的方法可以解决这一问题。

第一个方法，只要步进到使用 && 和 || 运算符的复合条件语句，就扫描相应的一些条件，验证这些条件拼写无误然后使用调试程序命令显示条件中每个比较的结果。这样做可以帮助我们查出在某些情况下虽然整个表达式的计算结果正确，但该表达式中确实有错误这一情况。例如，如果你认为在这种情况下 || 表达式的第一部分应该是 TRUE，第二部分应该是 FALSE，但其结果恰恰相反。此时虽然整个表达式的计算结果虽然正确，但表达式中却有错误。观察表达式的各个部分可以发现这类问题。

第二个，也是更彻底的方法是在汇编语言级步进到复合条件语句和 ?: 运算符的内部。是的，这要花费更多的工夫，但对于关键的代码，为了观看到中间的计算结果而对其内部的代码实际地走上一遍是很重要的。这同对 C 语句进行逐条的跟踪一样，一旦你习惯之后。对汇编语言指令进行逐条地跟踪也很快，只不过需要经过练习而已。

源级调试程序可能会隐瞒执行的细节 对关键部分的代码要进行汇编指令级的逐条跟踪

关掉优化？

如果所用的编译程序优化功能很强，那么对代码进行逐条的跟踪可能会是一个十分有趣的练习。因为编译程序在生成优化的代码时，可能会把相邻源语句对应的机器代码混在一块。对于这种编译程序，一条“单步”命令跳过三行代码并非不常见；同样，利用“单步”指令执行完一行将数据从一处送到另一处的源语句之后却发现相应的数据尚未传送过去的情况也很常见。

为了对代码进行逐条跟踪容易一些，在编译调试版本时可以考虑关掉不必要的编译程序优化。这些优化除了扰乱所生成的机器代码之外，毫无用处。我听到过某些程序员反对关掉编译程序的优化功能他们认为这会在程序的调试版本和交付版本之间产生不必要的差别从

而带来风险。如果担心编译程序会产生代码生成错误的话，这种观点还有点道理。但同时我们还应该想到，我们建立调试版本的目的是要查出程序中的错误，既然如此，如果关掉编译的优化功能可以帮助我们做到这点，那么就值得考虑。

最好的办法是对优化过的代码进行逐条的跟踪，先看看这样做的困难有多大，然后为了有效地对代码进行逐条跟踪，只关闭那些你认为必须关闭的编译程序优化功能。

小结

我希望我知道一种能够说服程序员对其代码进行逐条跟踪的方法，或者至少能够使他们尝试一个月。但是我发现，程序员一般说来都克服不了“那太费时间”这一想法。作为项目负责人的一个好处是对于这种事情你可以霸道一些，直到程序员认识到这样做并不费很多时间，并且觉得很值得这样做，因为出错率显著的下降了。

如果你还没有对你的程序进行逐条的跟踪，你会开始这样做吗？只有你自己才知道这个问题的答案。但我猜想当你拿起这本书并开始阅读的时候，准是因为你正被减少你或你领导的程序员的代码中的错误所困扰。这自然就归结为如下的问题：你是宁愿花少量的时间，通过对代码进行逐条的跟踪来验证它；还是宁愿让错误溜进原版源代码中，希望测试者能够注意到这些错误以便你日后对其进行修改。选择在你。

要点：

- 代码中不会自己生出错误来，错误是程序员编写新代码或者修改现有代码的产物。如果你想发现代码中的错误，没有哪个办法比在对代码进行编译时对其进行逐条跟踪更好。
- 虽然直观上你可能认为对代码进行走查会花费大量的时间，但这是不对的。刚开始进行代码的走查确实要多花一点时间，但当这一切习惯成自然之后并不会多花多少时间，你可以很快地走查一遍。
- 一定要对每一条代码路径进行逐条的跟踪，至少要跟踪一遍，尤其是对代码中的错误处理部分。不要忘记 `&&`、`||` 和 `?:` 这些运算符，它们每个都有两条代码路径需要进行测试。
- 在某些情况下也许需要在汇编语言级对代码进行逐条的跟踪。尽管不必经常这样做，但在必要的时候不要回避这种做法。

课题：

如果看看第一章中的练习，你就会发现它们所涉及的都是编译程序能够自动为你检查出来的常见错误。重新考查一遍这些练习，这次问问自己：如果使用调试程序对相应的代码进行逐条跟踪，你会漏掉那些错误吗？

课题：

看着六个月以来对你的程序报告出来的错误，确定假如你在编写程序时对其进行了逐条跟踪的话，你会抓住多少个错误。