China-pub.com
下载

第5章 排　　错

bug.
b. 机器、计划或其他类似东西中的缺陷、故障或过失。源自美国。
1889年《Pall Mall报》3月11日1/1，我听说爱迪生先生前两夜都爬起来在他的留声机里寻找“ bug ”—— 这表示解决一个困难，说是有什么想像中的害虫秘密地隐藏在里面并造成了所有的麻烦。

《牛津英语词典》第2版

我们在前四章里已经给出了许多代码，而且一直假装这些代码一写好就都工作得完美无缺。当然这绝不会是真的。程序里必然有大量的错 (bug)。“ bug ”这个词并不是程序员发明的，但它现在确实是计算领域中最常见的一个词。难道软件就该这么难吗？

这里的一个原因是，程序的复杂性与各部件间可能互相作用的途径数目有关。一个软件通常由许多部分组成，其互相作用的可能途径真是数不胜数。人们提出了许多技术，以减弱软件各部件间的关联，使存在交互作用的程序片段更少一些。这方面的技术包括信息隐蔽、抽象和界面，以及各种支持它们的语言特征等等。也有些技术的目标是为了保证程序的完整性——程序证明、模型技术、需求分析和形式化验证——不过它们还只是被成功地用到一些比较小的问题上。至今还没有什么东西能够改变软件构造的方式。现实就是这样，总是存在许多程序错误，需要通过测试来发现，通过排错去纠正。

好的程序员知道他们在排错上花费的时间至少与写程序一样多，所以他们努力从自己的错误中学习。你发现的任何错误都能教导你如何防止类似错误的再次发生，以及在发生这种问题时及早识别它。

排错是非常困难的，有可能花费很长的、无法预期的时间。这里的目标应该是避免出现太多问题。对减少排错时间能有所帮助的技术包括：好的设计、好的风格、边界条件测试、代码中的断言和合理性检查、防御性程序设计、设计良好的界面、限制全局数据结构以及检查工具等。总之，早期预防胜过事后治疗。

那么语言又有什么作用呢？在程序设计语言的发展中，一个重要的努力方向就是想通过语言特征的设计帮助避免错误。有些特征使某些种类的错误很难再出现了，例如：下标范围检查、受限制的指针或完全取消指针、废料收集、字符串数据类型、带类型的 I/O以及强类型检查等等。但是，硬币也有它的另外一面，有些语言特征有引起错误的倾向： goto语句、全局变量、无限制的指针及自动类型转换等等。程序员应该知道他们所用语言中有潜在危险的那一部分，使用那些机制时必须特别当心。他们还应该打开所有的编译检查，留意所有的警告。

每个为预防某些问题而设置的语言特征都会带来它自己的代价。如果一个高级语言能自动地去掉一些简单的错误，其代价就是使得它本身很容易产生一个高级的错误。没有任何语言能够防止你犯错误。

China-pub.com 下载

虽然没有人会希望这样，但实际程序设计的大部分时间确实是花在了调试和排错上。在本章里，我们将讨论如何尽可能地缩短排错时间，提高这方面工作的效率。第 6章将讨论调试问题。

5.1 排错系统

重要语言的编译系统通常都带有一个复杂的排错系统。它常常是作为整个开发环境里的一个组成部分，在这个环境里集成了有关程序建立和源代码编辑、编译、执行和排错的各种功能。排错系统一般包括一个图形界面，使人能够以按语句或者按函数的方式分步执行程序，在某个特定源程序行或者在某个特定条件发生时停下来等等。通常还提供了按照某些指定格式显示变量值等许多功能。

在已知某程序里存在错误的情况下，可以直接启动排错系统。有的排错系统也可以在程序执行中发生某些未预料到的问题时自动取得控制。当程序死了的时候，通常很容易确定它执行到了什么位置：只要检查活动的函数序列 (追踪执行栈)，显示出局部和全局变量值。这么多信息可能已经足够标识出错误了。如果还不行，利用断点和单步执行机制，可以一步步地重新执行程序，找到某些东西出问题的第一个位置。

在一个正确的环境里，对一个有经验的使用者，好的排错系统确实能使排错工作很有成效，工作效率也很高，人们甚至一点都不觉得烦恼。有了这样强有力的工具在手边，为什么我们还要考虑在不用它们的情况下做排错工作？为什么还需要整整一章来讨论排错问题呢？

确实有一些重要原因，有些是客观的，另一些则是来自个人的体验。一些在主流之外的语言并没有排错系统，或者只有非常低级的排错功能。排错系统是依赖于具体系统的，因此，当你在另一个系统中工作时，可能就没法使用你很熟悉的排错系统。有些程序用排错系统很难处理，例如多进程的或多线程的程序、操作系统和分布式系统，这些程序通常只能通过低级的方法排错。在上述这些情况下，你只能依靠自己，除了打印语句、自己的经验和对代码的推理能力之外，无法指望能得到多少其他帮助。

作为个人的观点，我们倾向于除了为取得堆栈轨迹和一两个变量的值之外不去使用排错系统。这其中有一个重要原因：人很容易在复杂数据结构和控制流的细节中迷失方向，我们发现以单步方式遍历程序的方式，还不如努力思考，辅之以在关键位置加打印语句和检查代码。后者的效率更高。与审视认真安排的显示输出相比，通过点击经过许多语句花费的时间更长。确定在某个地方安放打印语句比以单步方式走到关键的代码段更快，即使是你已经知道要找的位置。更重要的是，用于排错的语句存在于程序之中，而排错系统的执行则是转瞬即逝的。

使用一个排错系统，盲目地东翻西找绝不可能有效率。通过排错系统帮助发现程序出毛病的状态，则常常很有帮助。而在此之后，我们就应该仔细想想问题为什么会发生。排错系统是一类神秘的和难于使用的程序，特别是对初学者而言，它们带来的困惑可能比帮助还大。如果你提出的是一个错误的问题，它通常也能给出一个回答，但你可能就在不知不觉中被引错了方向。

排错系统可以是一种无价之宝，你确实应该在自己的排错工具箱里包括这种东西，它也很可能是你打开的第一个工具。但是，如果你没有排错系统，或者你要攻克的是极端困难的问题，本章的技术将能对你有所帮助，使你的排错工作有成效，效率更高，因为它们主要是

China-pub.com
下载

关于如何对错误及其可能原因进行推理的技术。

5.2 好线索，简单错误

哎呀！事情真是糟透了。我的程序垮台了，或者打印出的东西乱七八糟，或者看起来停不下来了。现在该怎么办啊？

初学者都有一个倾向，那就是抱怨编译系统、或者程序库、或者除了他们的代码之外的其他任何东西。有经验的程序员当然也希望能这样做，但是他们知道，实际上多半是他们自己的错。

幸运的是，大部分程序错误是非常简单的，很容易通过简单技术找出来。检查错误输出中的线索，设法推断它可能如何被产生。看看程序垮台前已经有了什么样的输出，如果可能的话，通过排错系统得到堆栈轨迹。现在你知道了一些情况，有关程序里发生了什么、在哪里发生等等。停一下好好想想，这些又为什么会发生？从程序垮台的状态向回推断，设法确定导致这些情况的原因。

排错涉及到一种逆向推理，就像侦破一个杀人谜案。有些不可能的事情发生了，而仅有的信息就是它确实发生了。因此我们必须从结果出发，逆向思考，去发现原因。一旦有了一个完全的解释，我们就知道如何去更正了。在这个过程中，我们多半还会发现一些其他的原来没有预料到的东西。

寻找熟悉的模式。问问自己这是不是一个熟悉的模式。“我确实见过它”常常是理解问题以至得到整个回答的开始。常见错误都有特有的标志，例如新的 C程序员常写出：

```
?     int n;
?     scanf("%d", n);
```

而不是：

```
int n;
scanf("%d", &n);
```

在典型的情况下，这将导致当程序要读入一行时，出现超范围的存储器访问企图。教授 C语言课程的人立刻就能认出它来。

在printf或scanf中类型与转换描述不匹配，也是常见错误的一种原因：

```
?     int n = 1;
?     double d = PI;
?     printf("%d %f\n", d, n);
```

这种错误的标志是有时出现十分荒谬的不可能的值，例如特别大的整数，或者特大特小的浮点数等等。在一台 Sun SPARC上，上面程序的输出是一个很大的整数和另一个更不可思议的数(为放在这里重新编排过)：

```
1074340347 2681561585988520015341087942602333963 50\
    1936585971793218047714963795307788611480564140\
    0796821289594743537151163524101175474084764156\
    42277140832383962343014 4.000000
```

另一个常见错误是，在用 scanf读入double数据时没有用%lf，而是用了%f。有的编译系统能够俘获这种错误，它们检查 scanf和printf的参数类型与格式串是否匹配。例如GNU编译系统gcc。如果我们打开所有的检查， gcc对上面的printf将报告：

China-pub.com 下载

```
x.c:9: warning: int format, double arg (arg 2)
x.c:9: warning: double format, different type arg (arg 3)
```

忘记对局部变量进行初始化是另一类容易识别的错误，其结果常常是特别大的值，这是由以前存放在同一存储位置的内容遗留下来的垃圾造成的。有些编译系统能对这类情况提出警告，你可能也必须打开所有的编译检查，而且绝不要指望它们能够捕捉到所有情况。由存储分配器如malloc、realloc或者new返回的存储里面通常也是垃圾，一定要记得做初始化。

检查最近的改动。哪个是你的最后一个改动？如果你在程序发展中一次只改动了一个地方，那么错误很可能就在新的代码里，或者是由于这些改动而暴露出来。仔细检查最近的改动能帮助问题定位。如果在新版本里出现错误而旧版本原来没有，新代码一定是问题的一部分。这意味着你至少应该保留程序的前一个你认为是正确的版本，以便比较程序的行为。这也意味着你应该维持一个关于已经做过的修改和错误更正的记录。这样，当你设法去改正一个错误时，就不必设法去重新发现这些关键信息。源代码控制系统和其他历史记录机制在这个方面很有帮助。

不要两次犯同样的错误。当你改正了一个错误后，应该问问自己是否在程序里其他地方也犯过同样错误。下面情况发生在其中一位作者正准备写这一章的时候。这是为某同事写的一个原型程序，其中包括一些处理可选参数的常见代码：

```
?     for (i = 1; i < argc; i++) {
?         if (argv[i][0] != '-')  /* options finished */
?             break;
?         switch (argv[i][1]) {
?         case 'o':               /* output filename */
?             outname = argv[i];
?             break;
?         case 'f':
?             from = atoi(argv[i]);
?             break;
?         case 't':
?             to = atoi(argv[i]);
?             break;
?         ...
```

我们的同事刚拿去不久，就送回报告说在输出文件名的前面总附有一个前缀 -o。这个情况很令人羞愧，但很容易改正。代码应该是：

```
outname = &argv[i][2];
```

这个问题改好后，程序交了出去。马上回来的报告是程序对像 -f123这样的参数不能正确处理，转换得到的数值总是0。这实际上是同一个错误，开关语句的下一个case应该改成：

```
from = atoi(&argv[i][2]);
```

由于作者还是太着急，他没有注意到同样的疏忽还出现了两次，这使事情又重复了一遍之后，所有实质上完全一样的错误才都得到更正。

简单的代码可能因其特别熟悉，而使我们放松了警惕，因此就可能出现错误。所以，即使是那些你闭上眼睛也可以写出来的简单代码，写它的时候也绝不能打瞌睡。

现在排除，而不是以后。在急忙中需要处理的事情太多，也可能造成其他损害。在任何一次程序垮台时都不要忽视它，应该立即对它进行跟踪，因为它可能不会再现，直到一切都变得太晚了。这里有一个著名的例子，发生在“火星探路者”任务中。这个航天器在 1997年7月完

China-pub.com
下载

成了一次完美无缺的着陆，但在此之后，探路者上的计算机差不多每天都要重新启动一次，把工程师们折腾得够呛。他们通过跟踪终于发现了问题，才知道原来见到过这个毛病。在发射前测试时出现过重新启动的情况，而他们却忽略了这个问题，因为当时正在忙着搞别的与此无关的事情。这就使他们不得不现在来设法解决问题，而机器已经在亿万英里之外，改正错误的困难就大得多了。

取得堆栈轨迹。虽然排错系统可以用来检查程序，但是它们最重要的用途之一就是在程序死了之后检查其状态。失败位置的源程序行号，堆栈追踪中屡次出现的部分都是最有用的排错信息。实际中不应该出现的参数值也是重要线索，例如空指针，应该很小的整数值现在却特别大，应该是正的值现在是负的，字符串里的非字母字符等等。

下面是一个典型的例子，取自第2章关于排序的讨论。要想对一个整型数组排序，我们应该用整数比较函数icmp调用qsort：

```
int arr[N];
qsort(arr, N, sizeof(arr[0]), icmp);
```

但是，假定我们无意中把字符串比较函数scmp传递进去：

```
?     int arr[N];
?     qsort(arr, N, sizeof(arr[0]), scmp);
```

编译无法发现这里的类型不匹配，灾难就要发生了。程序在运行时将会垮台，因为它企图访问非法的存储器地址。运行dbx排错系统产生的堆栈追踪是(经过编排)：

```
0 strcmp(0x1a2, 0x1c2) ["strcmp.s":31]
1 scmp(p1 = 0x10001048, p2 = 0x1000105c) ["badqs.c":13]
2 qst(0x10001048, 0x10001074, 0x400b20, 0x4) ["qsort.c":147]
3 qsort(0x10001048, 0x1c2, 0x4, 0x400b20) ["qsort.c":63]
4 main() ["badqs.c":45]
5 __istart() ["crt1tinit.s":13]
```

这里说的是，程序死在strcmp里，可以看到送给strcmp的两个指针值都特别小，这就是出毛病的一个明确标志。堆栈追踪还给出了每个函数调用的源程序行号，在我们的测试程序badqs.c里的第13行是：

```
return strcmp(v1, v2);
```

这标明了失败的调用，矛头直指错误。

排错系统还可以用于显示局部或全局变量的值，这往往能进一步提供一些有用信息，帮助确定错误是如何出现的。

键入前仔细读一读。一个有效的但却没有受到足够重视的排错技术，那就是非常仔细地阅读代码，仔细想一段时间，但是不要急于去做修改。出错时最大的诱惑就是赶快去用键盘，立刻开始修改程序，看看错误是否马上就能烟消云散。但是，很可能你并没有弄清楚到底什么是真正的毛病，所做的修改根本不对，或许还会弄坏别的什么东西。在纸面上打印出程序的关键部分能给人一种与看屏幕大不相同的视觉效果，也能促使人们花更多的时间去思考。当然，也不要把打印程序当作例行公事。打印整个程序实际上是在浪费纸张，因为程序将摊在许多页里，很难看清结构。一旦你做了一点编辑，打印出来的东西立刻就过时了。

应该稍微休息一下。有时你看到的代码实际上是你自己的意愿，而不是你实际写出的东西。离开它一小段时间能够松弛你的误解，帮助代码显出其本来面目。

China-pub.com 下载

抵抗急于键入的诱惑，换一个方式，思考一会。

把你的代码解释给别人。另一种有效技术就是把你的代码解释给其他什么人，这常常会使你把错误也给自己解释清楚了。有时你还没有说多少字，跟着就会不好意思地说“请别介意，我看到是什么错了。很抱歉打搅了你。”这种方式非常有效，你甚至不必要找个程序员作为听众。某大学的计算中心在咨询台上放了个绒毛玩具熊，出现了奇怪错误的学生被要求首先给玩具熊做解释，然后再去找做咨询的人。

5.3 无线索，难办的错误

“我没有发现任何线索，世界上居然会发生这种事？！”如果你确实对发生了什么事情一无所知，那么看来情况不太妙。

把错误弄成可以重现的。第一步应该是设法保证你能够使错误按自己的要求重现。想驱除一个并不是每次都出现的错误困难要大得多。你应该花点时间，设法构造输入或者参数设置，使自己能可靠地再现问题。然后再做总结，把有关步骤包装起来，使你通过一个按钮或几次按键就可以运行它。如果遇到的是非常困难的错误，你必须能在追踪问题的过程中使它一次次地重现，把它弄得很容易重现将能大大节约你的时间。

如果无法把错误弄成每次都出现的，那么就应该设法弄清为什么做不到。是否在某些条件下能使它比在其他条件下出现得更频繁？即使你无法保证错误每次都出现，如果你能减少等待它出现的时间，也就能够更快地找到它。

如果一个程序提供了排错输出，那么就应该打开它。像第 3章的Markov链程序一类的模拟程序应该包括一个选项：产生排错输出，例如输出随机数生成器的种子值，以保证你能产生同样的输出。另一个可能性是允许设置种子值。许多程序有这类选择项，你应该在自己的程序中包括某些类似的机制。

分而治之。能否把导致程序失败的输入弄得更小一点，或者更集中一点？设法构造出最小的又能保证错误现身的输入，这样可以减少可能性。什么样的变化使错误不见了？设法去发现最能体现错误特征的关键性测试。每个测试的目的都应该明确，用于肯定或者否定一个关于什么可能出错的假设。

采用二分检索的方式，丢掉一半输入，看看输出是否还是错的。如果不是，回到前面状态，丢掉输入的另一半。同样的二分检索过程也可以用到程序正文本身：排除程序中某些看来与错误无关的部分，看看错误是否仍旧在那里。带有 undo功能的编辑器在这里很有用，它们能帮助缩减大的测试情况或者大程序，而又不会丢掉错误。

研究错误的计数特性。有时失败的实例具有计数特征方面的模式，这常常是很好的线索，能使我们在寻找中集中注意力。我们在本书新写的几节里发现了一些拼写错误，出现的情况是偶然有字符消失了。这确实非常奇怪。有关正文是从其他文件通过剪切和粘贴建立起来的，因此，问题很可能与正文编辑器的剪切和粘贴命令有关。但是，从哪里着手寻找问题呢？为了发现线索，我们仔细地查看数据，注意到丢失的字符看起来像是一致地分布在正文中。我们度量了距离，发现丢失字符间的距离总是 1023字节，一个可疑的而且并不随机的数。在编辑器源程序里搜索接近 1024的数，发现了几个可能性，其中有一个出现在一段新代码里，因此我们首先检查它。错误很容易就确定了：典型的截掉一个字符的错误，在一个 1024字节的缓冲区最后用空字符覆盖掉了一个字节。

China-pub.com
下载

研究与错误有关的计数模式使我们直面错误。花了多少时间？若干分钟的迷惑，五分钟检查数据并发现丢失字符的模式，一分钟的检索找到可能需要修正的位置，再就是一两分钟确定错误并排除。如果想借助排错系统来发现这种错误，那大概是没什么指望的，这个问题涉及到两个多进程的程序，它们都由鼠标器点击驱动，又通过一个文件系统进行数据交换。

显示输出，使搜索局部化。如果你不能理解程序到底在做什么，弄清楚它最简单的而又代价低廉的方法就是加一些语句，使程序显示出更多的信息。用这种语句验证你对程序的理解或者你对什么东西可能出问题的想法。例如，如果你认为执行不可能达到代码中的某个特定点，可以让程序在这里显示“ can't get here ”，此后如果你看到了这个信息，那么就可以把输出语句向前移，设法确定事情从哪里开始出了错。或者是显示“ got here ”并把它向下移，找到程序中最后的看来还能工作的位置。各个信息应该能互相区分，这样你才能知道看到的究竟是哪一个。

用某种紧凑的形式显示信息，以便它们容易用眼睛，或者用程序做扫描。例如用模式匹配工具grep(grep这样的工具对于在正文中检索真是无价之宝，第 9章给出了这种程序的一个简单实现)。如果你要显示某些变量的值，应该每次都按同样方式做格式化。在 C和C++里应该总用%x或%p以十六进制数的形式显示指针，这能帮助你看到两个指针是否具有同样的值或者是互相有关。学着读指针值，识别像的和不像的，如零、负数、奇数、小的数等。把地址的形式搞熟，在使用排错系统时也会获益匪浅。

如果输出的量非常大，只打印输出一个字符可能就足够了。例如打印 A，B，…，这可以作为说明程序走到哪里的一种紧凑显示形式。

写自检测代码。如果需要更多的信息，你可以写自己的检查函数去测试某些条件、打印出相关变量的值或者终止程序：

```
/* check: test condition, print and die */
void check(char *s)
{
    if (var1 > var2) {
        printf("%s: var1 %d var2 %d\n", s, var1, var2);
        fflush(stdout); /* make sure all output is out */
        abort();        /* signal abnormal termination */
    }
}
```

我们让check调用C语言的标准库函数abort，这个函数将导致程序以非正常方式终止，供排错系统分析。在另一些应用里，你也可能需要让 check打印了某些东西之后继续下去。

下一步，把对check的调用加到代码里可能需要它的地方：

```
check("before suspect");
/* ... suspect code ... */
check("after suspect");
```

在错误排除后，不要简单地将 check丢掉。让它留在源程序里，注释掉它或者用一个排错选项控制它。这样，如果你再遇到另一个困难问题，还可以重新启用它们。

对一些更困难的问题，可以把check发展成一种能验证和显示数据结构的东西。这个方法还可以进一步推广：写出一些例行程序，让它们对数据结构或其他信息做在线的一致性检查。对于那种采用了特别复杂的数据结构的程序，在问题出现之前就写好这种程序也是个很好的主意。可以把它们作为程序的固有部件，一旦出现了麻烦就打开，不要仅仅把它们看作排错的工

China-pub.com 下载

具，在程序开发的整个过程中都把它们安置在那里。如果它们的运行代价不大，一直打开它们也是一种很聪明的办法。大型系统，如电话交换系统等，通常都包含了大量的代码，用于“监听”子系统的情况，监视信息和设备的情况，出现错误时立即报告，甚至自动去纠正。

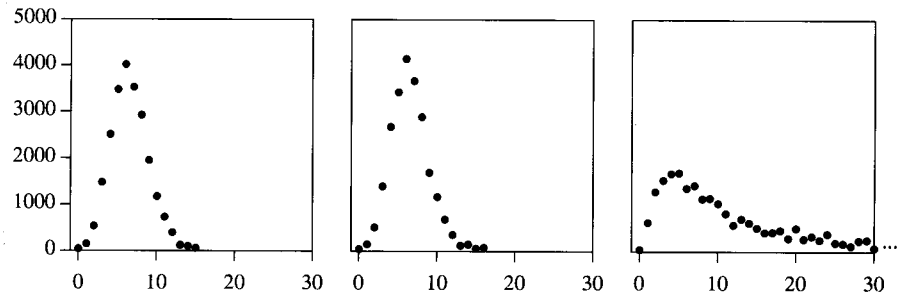
写记录文件。另一种战术是写一个记录文件，以某种固定格式写出一系列的排错输出。当程序垮台的时候，这个文件里已经记录了垮台前发生的情况。 Web服务器和其他网络程序都维护着有关数据流动的记录文件，其内容相当广泛，这使它们可以监视自己和客户的活动。下面的片段来自一个本地系统(重新编排过)：

```
[Sun Dec 27 16:19:24 1998]
HTTPd: access to /usr/local/httpd/cgi-bin/test.html
    failed for m1.cs.bell-labs.com,
    reason: client denied by server (CGI non-executable)
    from http://m2.cs.bell-labs.com/cgi-bin/test.pl
```

应该保证做了I/O缓冲区刷新，使最后的记录也能出现在记录文件里。像 `printf`一类的输出函数通常对其输出采用缓冲方式，以提高打印的效率。在程序非正常终止时，位于缓冲里的输出信息可能就被丢掉了。在 C语言里调一次函数`fflush`，就能保证把缓冲里的数据写出去， C++ 和Java语言对输出流也有与`fflush`类似的函数。如果你可以容忍这方面的负担，那么就应该对记录文件使用不加缓冲的 I/O操作，这样可以完全免除刷新问题。标准库函数`setbuf`和`setvbuf`用于做缓冲控制，调用`setbuf (fp,NULL)`将关闭对于流`fp`的缓冲。标准错误流(`stderr`、`cerr`和`System.err`)的默认方式一般都是不缓冲的。

画一个图。在测试和排错中，有时图形比文字更加有效。图形对于帮助理解数据结构特别有用，我们在第2章早已看到这一点。图形对于写图形软件当然非常重要，但是，实际上图形在各种程序设计里都可能用到。散点图形往往能比一列数值更有效地显示出某些错位的情况；数据直方图能揭示出各种反常现象，无论是考试成绩、随机数、存储分配器和散列表中筒的大小、或者其他东西。

在你无法理解程序里到底发生了什么的时候，请设法用统计方法解释其中的数据结构，并用图形方式显示其结果。下面的图形显示了第 3章中C版本的`markov`链程序的实际情况， *x*方向表示散列链的长度， *y*方向表示位于长度为 *x*的链中的总元素个数。输入数据用的是我们的标准测试，《圣经 · 雅各书》中的《诗篇》 (共42685个词， 22482个前缀)。前两个图表示的是采用好的散列乘因子 31和37的情况，第三个图表示用很糟糕的散列因子 128的情况。在前两种情况下，链的长度都不超过 15或者16，大部分元素都位于长度 5或者6的链中。而对第三种情况，点的分布要宽得多，最长的链中有 187个元素，数千个元素位于长度超过 20的链里。



乘因子 31　　乘因子 37　　乘因子 128

China-pub.com
下载

使用工具。在排错时，应该尽量利用好工作环境里的各种功能。例如，像 diff那样的文件比较程序，可用于比较成功的和失败的排错运行输出，使你能把注意力集中到出现差异的地方。如果排错的输出太长，可以用 grep进行检索或者用编辑器查看。对于把排错输出送到打印机则应该谨慎行事，对于大量的输出，利用计算机扫描比人自己做要方便得多。还可以利用外壳脚本[⊖]或其他工具，帮助我们处理由排错运行得到的输出。

写一些简单的程序去测试你的假定，或者确认你对某些东西工作方式的理解。例如，对一个空指针做释放是合法的吗？

```
int main(void)
{
    free(NULL);
    return 0;
}
```

源代码控制程序，如 RCS，能够跟踪代码的版本，使人可以看清哪些东西被修改过，还可以恢复原来版本，回到已知的状态。除了能指出哪些是程序中最近的改动外，这种程序还可以帮助标出代码中长期以来经常被修改的部分，这些部分常常是出现错误的主要位置。

保留记录。如果查找某个错误的过程花了一定时间，你可能就要开始忘记试验过的情况和已经学到的东西了。如果对已经做过的所有测试和结果都有记录，就不大容易忽略了某些东西，或者认为你已经检查了某些可能性而实际上并没有做过。写这些的过程也能帮助你记住问题，使你下次能够看到某些东西的类似性，还可以使你在给其他人解释问题时能够有所参照。

5.4 最后的手段

如果上面的建议都没有用，那么又该怎么办？这可能是使用一个好的排错系统，以步进方式遍历程序的时候了。如果你关于某些东西如何工作的思维模型根本就不对，以至你一直在完全错误的地方寻找问题，或者找的地方是对的，但却总也不能发现问题，那么排错系统将迫使你以另一种方式去思考。这种“思维模型”错误是最难发现的一类错误，在这里机械的帮助将很有价值。

有时概念性的错误实际上却非常简单：不正确的运算符优先级，或者用错了运算符，或者是程序的缩排形式与实际结构不符，或者是作用域错误，例如局部变量遮蔽了同名的全局变量，或者全局变量侵犯了一个局部作用域。例如，程序员经常忘记 &和|的优先级低于 = = 和!=，他们可能写了：

```
? if (x & 1 == 0)
?     ...
```

而怎么也看不出为什么条件老是不成立。偶然的指尖一点可能导致单个的 = 变成了两个，或者正相反：

```
? while ((c == getchar()) != EOF)
?     if (c = '\n')
?         break;
```

或者在编辑之后遗留下某些东西：

```
? for (i = 0; i < n; i++);
```

⊖ shell script，为UNIX的术语，功能远强于DOS的批处理程序文件，但机制类似。—— 译者

China-pub.com 下载

```
?         a[i++] = 0;
```

或者因为草率的输入造成了问题：

```
?     switch (c) {
?         case '<':
?             mode = LESS;
?             break;
?         case '>':
?             mode = GREATER;
?             break;
?         defualt:
?             mode = EQUAL;
?             break;
?     }
```

有时错误的原因是实际参数的次序给错了，而又恰巧出现在类型检查无法发现的地方。例如写的是：

```
?     memset(p, n, 0);     /* store n 0's in p */
```

而不是：

```
    memset(p, 0, n);     /* store n 0's in p */
```

有时候某些东西在你的背后被改变了，例如全局变量或者共享变量被修改，而你没有注意到程序的其他部分也能够修改它。

有时则是你的算法或者数据结构里存在致命缺陷，而你却始终无法看到它。在准备关于链接表的材料时，我们写了一组链接表操作函数来建立新元素、将它们链接在表的前面或者后面，等等。这些函数都出现在第2章中。当然我们也要写一个测试程序，以保证所有东西都是正确的。前几个测试都能行，而后有一个却失败得非常奇怪。下面基本上就是这个测试程序：

```
?     while (scanf("%s %d", name, &value) != EOF) {
?         p = newitem(name, value);
?         list1 = addfront(list1, p);
?         list2 = addend(list2, p);
?     }
?     for (p = list1; p != NULL; p = p->next)
?         printf("%s %d\n", p->name, p->value);
```

最后我们吃惊地发现，在第一个循环里，结点 p被同时放进两个表里，到了打印的时候，指针已经被搅得乱七八糟了。

要找到这类错误确实非常困难，因为你的思维总是领着你绕过错误。在这种情况下，排错系统将会很有帮助，它能迫使你向另一个方向走，顺着程序的实际工作流程，而不是你头脑里设想的流程。与此类似，如果问题出在错误地理解了整个程序的结构，要想看到错误到底是什么，就必须回到开始的假设去。

请注意，这里还应该顺便提一下，上面有关链接表的错误出现在测试代码里，这也使错误更难被发现。有些情况常常使人感到特别沮丧，例如花了大量时间在没有错误的地方寻找，由于测试程序本身有错，被测试的根本不是程序的正确版本，或者是忘记了在测试之前更新或者重新编译，等等。

如果你在做了大量努力后还是不能找到错误，那么就应该休息一下。清醒一下你的头脑，做一些别的事情，和一个朋友谈谈，请求帮助。问题的答案可能会突然从天而降。即使情况不是这样，在下一次再做排错时，你多半也不会再走上次的老路了。

偶然也会遇到这种情况，问题确实出在编译系统，或者库，或者操作系统，甚至是计算

China-pub.com
下载

机硬件，特别是如果在错误出现的环境里的什么东西刚刚换过。当然，你绝不应该一开始就抱怨这些东西，但是如果其他所有的可能性都已经完全排除之后，这些可能就是仅存的东西了。有一次，我们把一个大的文本格式化程序从原来运行的 Unix系统搬到PC上，程序编译没出现任何问题，但它的行为却特别奇怪：它几乎每次都丢掉输入的第二个字符。我们的第一个想法是问题可能源于这里用的是16位整数，而不是32位的，要不然就是某种字节顺序问题。通过打印进入主循环的字符，最后把问题归结到了编译厂商提供的标准头文件 ctype.h里的一个错误，在那里isprint被实现为一个函数宏：

```
? #define isprint(c) ((c) >= 040 && (c) < 0177)
```

而程序的主循环是：

```
? while (isprint(c = getchar()))
? ...
```

每次遇到的输入字符是空格(八进制40，这是写 ' ' 的一种糟糕的方式)或更大时(当然，大部分时间都会是这样)，getchar就会被调用第二次，因为上面的宏对参数求值两次，这就使第一个输入字符永远消失了。虽然原始代码并不像它应该有的那样清晰，在循环条件里写的东西太多，但厂商头文件里的错误也是无可辩解的。

今天我们仍然不难发现这类错误的例子，下面的宏来自另一个厂商目前的头文件：

```
? #define __iscsym(c) (isalnum(c) || ((c) == '_'))
```

存储器“流失”——不再使用的存储没有退还——是程序古怪行为的一个重要根源。另一个类似问题是忘记关闭文件，直到打开的文件占满了文件表，程序无法再打开新文件。带有这类漏洞的程序最终将奇怪地垮台，原因是它用光了所有资源，而特定的失败又是无法重现的。

偶然地，硬件也可能出毛病。1994年Pentium处理器的浮点错误能导致某些确定的计算得出错误结果，这是硬件设计历史上最为大众熟知的也是代价最高的错误。而一旦它被标识清楚，也就很容易重现了。我们见过的最奇怪的错误之一与一个计算器程序有关，很久以前这个程序运行在一个双处理器系统里。该程序对表达式 1/2的计算结果有时是0.5，而有时会输出另一个统一的同时又是错误的值，大概是 0.7432。在取正确值或错误值之间找不到任何模式。问题最后总算弄清楚了，追踪到一个处理器的浮点单元有毛病。由于计算器程序随机地在这个或那个处理器上执行，因此它有时给出正确结果，有时就给出乱七八糟的东西。

许多年以前，我们还用过一台机器，其内部温度可以由浮点计算中低位出错的位数来衡量。最终原因是有一块电路板松了，当机器发热时，电路板从插槽里翘出来得更多，这就使更多的数据位与底板断开了。

5.5 不可重现的错误

不能始终重现的错误是最难对付的，而且这种问题又不像硬件故障那么明显。根据这种非确定性的行为，能确认无疑的事实也就是信息本身，但这通常意味着多半不是算法本身有什么毛病，而是这些代码以某种方式使用了什么信息，而这些信息在每一次程序运行时又可能是不同的。

应该先检查所有的变量是否都正确地进行了初始化。如果没有，它就可能取到某个具有随机性的值，是以前存入同一个存储位置的。函数的局部变量和由分配得到的存储块是 C或

China-pub.com 下载

C++ 里最重要的嫌疑犯。把所有变量都用已知值设置好，如果程序里用到某个随机数的种子(正常情况下它可能会被用日期和时间设置)，现在也应该先把它设置为常数，例如设置为 0。

如果在增加排错代码之后错误的行为改变了，甚至是消失了，那么它很可能就是一个存储分配错误——某个时候你的代码在被分配的存储之外写了什么东西。排错代码的加入改变了存储安排，因此也就可能改变了错误的行为。有许多输出函数，从 printf到对话窗口，在工作过程中都要分配存储，这些都能进一步把水搅混。

如果垮台的地方看起来与任何可能出错的东西都距离很远，那么最有可能的就是错误地向某个存储位置里写进了一些东西，而又是在很久以后才用到了这个地方。有时问题出在悬空的指针，例如由于疏忽从函数里返回了一个指向局部变量的指针，而后又使用了它。返回局部变量的地址可以说是产生延迟灾难的一个秘方：

```
? char *msg(int n, char *s)
? {
?     char buf[100];
?
?     sprintf(buf, "error %d: %s\n", n, s);
?     return buf;
? }
```

当由msg返回的指针被使用时，它已经不再指向有意义的地址了。如果真需要做这类事，你必须用malloc分配存储，或者用static数组，或者要求函数的调用者提供存储。

如果在释放了一块动态分配的存储之后又去使用它，那么也可能产生类似的症状。在第 2 章里我们写freeall时曾经讲到，下面代码是错误的：

```
? for (p = listp; p != NULL; p = p->next)
?     free(p);
```

一旦存储被释放，就不应该再使用它，因为它的内容可能已经改变了。在这个例子里不能保证p->next还指向正确的位置。

对于某些malloc和free的实现，两次释放同一块存储将会破坏内部的数据结构，而这要到很久以后，直到某个后续调用踩到了前面弄出来的泥潭，才会真正造成麻烦。某些分配器带有一个排错选择项，如果设置这个选项，在每次调用时分配器都做整个内部状态的一致性检查。如果遇到了不可重现的错误，你就应该打开这个选项。如果没有这种机制，你也可以自己写一个存储分配器，让它做一些内部正确性检查，或者输出一个记录文件以便随后做分析。如果不要求运行得特别快，写一个分配器是不太难的，如果情况需要，这种做法也是可行的。市面上可以找到很好的商品工具，它们能够检查程序的存储管理，捕捉错误和漏洞。如果你找不到这种工具，通过写出自己的 malloc和free也可能得到某些与它们类似的实惠。

当一个程序对于某个人能工作，而对另一个人则不行时，一定是有某些东西依赖于程序的外部环境。这可能包括：程序需要读的文件、文件的权限、环境变量、命令的查找路径、各种默认的东西和启动文件等等。对于这些情况就很难提出建议和意见，除非你变成另一个人，复制垮台程序所在的整个环境。

练习5-1 写一套malloc和free函数，使它们能够用在与存储管理有关问题的排错工作中。一种方法是在每次malloc和free调用时检查整个工作空间；另一种方法是让它们写出一些记录信息，这种信息可以用另一个程序处理。对这两种方法，函数都应该在每个分配块的两端加上标记，以便能对超出块端写入的情况进行检查。

China-pub.com
下载

5.6 排错工具

排错系统并不是惟一能帮人检查程序错误的工具。还有许多程序也能帮我们的忙，例如从长长的输出里选出要点，发现其中的奇怪现象；或者重新安排数据，使人更容易看清情况是如何发展变化的，等等。很多这类程序都是标准工具箱的组成部分，还有的则需要我们自己写，以帮助发现一些特殊错误，或用于分析特定的程序。

在这一节里我们将描述一个很简单的程序，叫 strings，它的特殊用途是用来检索一些大部分都是非打印字符的文件，例如可执行程序或某些文字处理系统使用的神秘二进制文件等。在这种文件里常常隐藏着一些很有价值的信息，如文档正文、错误信息、没有给出文档的选择项、文件和目录的名字或者程序里需要调用的函数名字等等。

我们还发现，strings对于在其他二进制文件里找出正文信息也很有用。图像文件里经常包含有ASCII串，指明建立它们的程序名字；压缩文件和归档文件 (如zip文件)里可能包含许多文件名字。用strings也可以发现这些东西。

Unix系统已经提供了一个strings的实现，虽然它与这里要介绍的有些细微差别。那个程序能识别它的输入是否为一个程序，它对程序只检查其中的正文和数据段，而忽略其中的符号表。打开它的-a选项可以强迫它读取整个文件。

strings的功能就是从二进制文件里抽取出 ASCII文本，以便使人能阅读这种文本，或者用其他程序来处理它们。如果一个错误信息不带任何标识，要想知道它是由哪个程序产生就很不容易，更不必说是为什么产生了。在这种情况下，可以用下面的命令查找某个受怀疑的目录：

```
% strings *.exe *.dll | grep 'mystery message'
```

这样就有可能找到错误信息的来源。

函数strings读一个文件，打印出其中所有至少是连续的 MINLEN=6个可打印字符形成的字符串：

```
/* strings: extract printable strings from stream */
void strings(char *name, FILE *fin)
{
    int c, i;
    char buf[BUFSIZ];

    do {    /* once for each string */
        for (i = 0; (c = getc(fin)) != EOF; ) {
            if (!isprint(c))
                break;
            buf[i++] = c;
            if (i >= BUFSIZ)
                break;
        }
        if (i >= MINLEN) /* print if long enough */
            printf("%s:%.*s\n", name, i, buf);
    } while (c != EOF);
}
```

函数printf的格式串%.*s由下一个参数(i)取得要求打印的字符串长度，因为在参数串 (buf)里没有结尾的空字符。

这里用do-whilet循环找到每个这种串，将它们打印，遇到 EOF时结束。采用在循环最后检查文件结束的方式，可以让 getc和串处理循环共用同一个结束条件，并使一个 printf就能

China-pub.com 下载

够处理字符串结束、文件结束和串过长等多种情况。

如果采用标准形式的循环，在最前面做检测，或是用一个 getc循环带上一个复杂得多的循环体，都需要写出好几个 printf调用。我们开始就是那样写的，后来发现在它的 printf语句里有一个错误。我们更正了一个毛病但是却忘记更正另外两个（“我是不是在其他地方犯了同样错误？”）。到了此时事情已经变得很清楚了，这个程序应该重写，应该减少重复性的代码。最后得到的就是这个do-while循环。

程序strings的主函数对它的每个参数文件调用函数 strings：

```
/* strings main: find printable strings in files */
int main(int argc, char *argv[])
{
    int i;
    FILE *fin;

    setprogname("strings");
    if (argc == 1)
        eprintf("usage: strings filenames");
    else {
        for (i = 1; i < argc; i++) {
            if ((fin = fopen(argv[i], "rb")) == NULL)
                weprintf("can't open %s:", argv[i]);
            else {
                strings(argv[i], fin);
                fclose(fin);
            }
        }
    }
    return 0;
}
```

你可能会奇怪，为什么在没有给出文件名时， strings不转去从标准输入读入。原来它确实是那样做的。为了解释为什么现在改了，我们需要告诉你一个排错的故事。

对strings的一个最明显的测试项目就是让它对这个程序本身运行。在 Unix上它工作得很好，而在Windows 95上，命令：

```
C:\> strings <strings.exe
```

打印出下面这五行输出：

```
!This program cannot be run in DOS mode.
`.rdata
@.data
.idata
.reloc
```

第一行看起来像是个错误信息，我们费了点时间才认识到，实际上它正是程序里的一个字符串。程序的输出是正确的，至少在它的运行能到达的地方。类似情况也常常听到，由于误解了信息的来源，使某项排错工作出了轨。

但是，程序还应该有更多的输出，它们跑到哪里去了呢？后来的一个晚上，光明终于出现了(“我见过这种情况！”)。这实际上是一个移植问题，这种问题将在第 8章里详细讨论。我们原来写的程序是用 getchar从标准输入读数据的。但是在 Windows里，getchar在正文模式下输入时，一遇到特殊字节 0x1A或control-Z，就会返回EOF。这就是导致程序过早终止的原因。

China-pub.com
下载

这绝对是一种合法行为，但却不是我们在自己的 Unix背景下能预料到的。解决问题的办法是采用模式“ rb ”，以二进制方式打开文件。但是， sdtin是已经打开的，也没有标准的办法去改变其模式(虽然可以用某些函数如fdopen或setmode，但它们都不是C标准的部分)。最后我们面临的是一些味道都不太好的选择：或者要求用户总提供文件名，以便使程序能在Windows下正确工作，对 Unix则带来一些不方便；或者在 Windows用户想通过标准输入做处理时，就不声不响地塞给他一个错误结果；或者使用条件编译，使程序能适合不同的系统，付出的代价是削弱程序的可移植性。我们最后选择了第一条路，使同一个程序在任何地方都以同样方式工作。

练习5-2 strings程序打印的串里包含MINLEN个字符或者更多，这样有时可能产生过多无用的信息。给strings加一个可选参数，就可以定义最小的字符串长度。

练习5-3 写程序vis，它从输入向输出做复制，但是对非打印字符如退格、控制字符以及非ASCII码字符用\Xhh的形式显示，这里的hh是非打印字符的十六进制表示。与 strings的情况相反，vis对于检查那些只包含少数非打印字符的输入特别有用。

练习5-4 如果输入是\X0A，你的vis程序将输出什么？怎样才能把 vis的输出改造成无歧义性的？

练习5-5 扩展vis程序，使它能处理一系列文件、把长行在某个适当的列做折叠、完全去掉所有非打印字符等。考虑还有哪些特性与这个程序的角色协调？

5.7 其他人的程序错误

在现实中，许多程序员都已经无法享受从基本的东西出发开发全新系统的乐趣了，他们实际上把自己的大部分时间用到别人写的代码上，使用、维护和修改这些代码，或者 (无可避免地)设法排除代码中的错误。

在对别人的代码做排错时，我们前面针对为自己的代码排错所讲的全部东西都仍然是有效的。在开始工作前，你必须首先对程序原来的组织方式有所理解，还要设法理解原来的程序员是如何思考问题和写程序的。对于非常大的软件项目，人们在这里使用的术语是“ 发现 ”，这并没有什么不好的意思。当你面对着别人的代码时，那种设法在地球上发现什么的工作也会以某种形式出现在这里。

各种工具在这里都可能很有帮助。 grep一类的文本搜索程序有助于找到所有出现的名字；交叉引用程序可以帮人看清程序结构的某些思想；显示函数调用图 (如果不太大的话)也很有价值；用一个排错系统，以步进方式一个一个函数地执行程序，可以帮人看清事件发生的顺序；程序的版本历史可以给人一些线索，显示出随着时间变化人们对程序做了些什么。代码中的频繁改动是个信号，常常说明对问题的理解不够，或者表示需求发生了变化，这些经常是潜在错误的根源。

有时你可能需要在自己不需要负责的软件里，在没有源代码的程序里寻找错误。在这种情况下，要做的就是把程序错误的特征以最佳方式标识出来，以便能给出一个精确的错误报告，或许同时还应设法找到一条“ 旁路 ”，以回避这些错误。

China-pub.com 下载

如果你认为自己已经发现了别人程序里的一个错误，那么下一步要做的就是确认它确实是个真正的错误，这样才能不浪费作者的时间，也不损害你自己的信誉。

当你发现编译系统的错误时，应首先弄清错误确实是在编译系统，而不是在你自己的代码里。例如，右移运算到底是用 0填充空位(做逻辑移位)，还是做符号位的传播 (算术移位)，这一点在C和C++里并没有明确定义。新手在发现下面这类结构产生了非预期的结果时，可能认为这是个错误：

```
?     i = -1;
?     printf("%d\n", i >> 1);
```

实际上这只是个移植性问题。这个语句是合法的，但是在不同系统上可能有不同效果。请在多个不同系统里测试这个程序，以保证你确实理解了这里发生的情况；再查阅一下语言定义，进一步弄清问题。

应该确保你发现的程序错误不过时。你是否有程序的最新版本？是不是有一个错误修正表？很多软件都有多个不同的版本发布。如果你在版本 4.0b1里发现了一个错误，它可能早已经被修正，或许该软件已经被新版本 4.0b2取代了。在任何时候，除了系统的最新版本以外，程序员很少愿意耗费精力去更正其他错误。

最后，应该设身处地地为接受错误报告的人想一想。你应努力为别人提供一个做得尽可能好的测试实例。但是，如果这个错误只能用很大的输入、非常复杂的环境或者许多支持文件才能表现出来，那实际上就不会有太大帮助。应该设法剥离出一个最小的而且是自给自足的测试实例，并为它附上其他可能有关的信息，例如，程序本身的版本编号、编译系统、操作系统和硬件情况等。例如，对第 5.4节里提到的有错误的`isprint`，我们可以提供下面的程序作为一个测试程序：

```
/* test program for isprint bug */
int main(void)
{
    int c;

    while (isprint(c = getchar()) || c != EOF)
        printf("%c", c);
    return 0;
}
```

任何可打印的文本都可以作为测试实例，因为输出将只包括输入的一半：

```
% echo 1234567890 | isprint_test
24680
%
```

最好的错误报告就是那种仅需要给基本系统提供一两行输入就能说明毛病的东西，最好再带上如何更正的说明。应该送给别人那种你希望自己也能接收到的错误报告。

5.8 小结

带着一种好心情，排错也可以是件愉快的事情，就像是解一个谜题。而从另一方面看，无论你喜欢或者不喜欢，排错都是一种技艺，我们总会经常地要去实践它。如果能不出现错误，那当然是再好也没有了，所以我们应该设法避免错误，在第一次写代码时就应该努力把它写好。书写良好的代码一开始包含的错误就比较少，剩下的错误也比较容易查清楚。

一旦看到了一个程序错误，首先应该做的是努力去考虑出错的可能线索：它可能从何而

China-pub.com
下载

来？是不是什么熟悉的东西？程序里哪些东西刚刚修改过？在引起错误的输入数据里又有什么特殊的东西？在此之后，几个仔细选择的测试实例和加进代码的几个打印语句可能就足以解决问题了。

如果无法发现好的线索，努力思考仍然是最好的开始，随后应该做的就是以系统的方式缩小可能出问题位置的范围。一条途径是缩减输入数据，设法找到能导致失败的最小输入；另一个方法是切掉一些代码，减少有关的代码区域。也可以在程序里加进一些检查代码，在程序执行一些步骤后再打开它们，这样也可能把问题局部化。所有这些都是一种最一般的策略，“分而治之”的实际例子。分治法在排错过程中是非常有效的，就像它在政治和战争中一样。

也应该借助于其他力量。把你的代码解释给其他什么人 (甚至是一只玩具熊)也是很有效的方法；使用排错系统取得堆栈轨迹；使用某些商品工具检查存储流失、数组的越界操作、可疑的代码或者其他各种类似的东西。当你确信自己对代码如何工作可能有一个错误的思维图式时，可以采用步进式执行程序的方式。

还应该了解你自己，你容易犯什么样的错误。一旦你确定并更正了一个错误，就应该再想一想，你是否也清除了其他类似的错误。还应该想清楚发生的究竟是什么问题，以便不再重犯同类错误。

补充阅读

Steve Maguire的《写可靠的代码》(Writing Solid Code，Microsoft Press，1993)和Steve McConnell的《完整编程》(Code Complete，Microsoft Press，1993)中都包含了许多有关排错的好建议。