China-pub.com
下载

第1章 风 格

人们看到最好的作家有时并不理会修辞学的规则。还好，当他们这样做虽然付出了违反常规的代价，读者还经常能从句子中发现某些具有补偿性的价值。除非作者自己也明确其做法的意思，否则最好还是按规矩做。

William Strunk和E. B. White，《风格的要素》

下面这段代码取自一个许多年前写的大程序：

```
if ( (country == SING) || (country == BRNI) ||
     (country == POL) || (country == ITALY) )
{
        /*
         * If the country is Singapore, Brunei or Poland
         * then the current time is the answer time
         * rather than the off hook time.
         * Reset answer time and set day of week.
         */
        ...
```

这段代码写得很仔细，具有很好的格式。它所在的程序也工作得很好。写这个系统的程序员会对他们的工作感到骄傲。但是这段摘录却会把细心的读者搞糊涂：新加坡、文莱、波兰和意大利之间有什么关系？为什么在注释里没有提到意大利？由于注释与代码不同，其中必然有一个有错，也可能两个都不对。这段代码经过了执行和测试，所以它可能没有问题。注释中对提到的三个国家间的关系没有讲清楚，如果你要维护这些代码，就必须知道更多的东西。

上面这几行实际代码是非常典型的：大致上写得不错，但也还存在许多应该改进的地方。

本书关心的是程序设计实践，关心怎样写出实际的程序。我们的目的是帮助读者写出这样的软件，它至少像上面的代码所在的程序那样工作得非常好，而同时又能避免那些污点和弱点。我们将讨论如何从一开始就写出更好的代码，以及如何在代码的发展过程中进一步改进它。

我们将从一个很平凡的地方入手，首先讨论程序设计的风格问题。风格的作用主要就是使代码容易读，无论是对程序员本人，还是对其他人。好的风格对于好的程序设计具有关键性作用。我们希望最先谈论风格，也是为了使读者在阅读本书其余部分时能特别注意这个问题。

写好一个程序，当然需要使它符合语法规则、修正其中的错误和使它运行得足够快，但是实际应该做的远比这多得多。程序不仅需要给计算机读，也要给程序员读。一个写得好的程序比那些写得差的程序更容易读、更容易修改。经过了如何写好程序的训练，生产的代码更可能是正确的。幸运的是，这种训练并不太困难。

程序设计风格的原则根源于由实际经验中得到的常识，它不是随意的规则或者处方。代码应该是清楚的和简单的 —— 具有直截了当的逻辑、自然的表达式、通行的语言使用方式、

China-pub.com 下载

有意义的名字和有帮助作用的注释等，应该避免耍小聪明的花招，不使用非正规的结构。一致性是非常重要的东西，如果大家都坚持同样的风格，其他人就会发现你的代码很容易读，你也容易读懂其他人的。风格的细节可以通过一些局部规定，或管理性的公告，或者通过程序来处理。如果没有这类东西，那么最好就是遵循大众广泛采纳的规矩。我们在这里将遵循《C程序设计语言》(The C Programming Language)一书中所使用的风格，在处理Java和C++ 程序时做一些小的调整。

我们一般将用一些好的和不好的小程序设计例子来说明与风格有关的规则，因为对处理同样事物的两种方式做比较常常很有启发性。这些例子不是人为臆造的，不好的一个都来自实际代码，由那些在太多工作负担和太少时间的压力下工作的普通程序员 (偶然就是我们自己) 写出来。为了简单，这里对有些代码做了些精练，但并没有对它们做任何错误的解释。在看到这些代码之后，我们将重写它们，说明如何对它们做些改进。由于这里使用的都是真实代码，所以代码中可能存在多方面问题。要指出代码里的所有缺点，有时可能会使我们远离讨论的主题。因此，在有的好代码例子里也会遗留下一些未加指明的缺陷。

为了指明一段代码是不好的，在本书中，我们将在有问题的代码段的前面标出一些问号，就像下面这段：

```
?   #define ONE 1
?   #define TEN 10
?   #define TWENTY 20
```

为什么这些#define有问题？请想一想，如果某个具有 TWENTY个元素的数组需要修改得更大一点，情况将会怎么样。至少这里的每个名字都应该换一下，改成能说明这些特殊值在程序中所起作用的东西。

```
#define INPUT_MODE 1
#define INPUT_BUFSIZE 10
#define OUTPUT_BUFSIZE 20
```

1.1 名字

什么是名字？一个变量或函数的名字标识这个对象，带着说明其用途的一些信息。一个名字应该是非形式的、简练的、容易记忆的，如果可能的话，最好是能够拼读的。许多信息来自上下文和作用范围(作用域)。一个变量的作用域越大，它的名字所携带的信息就应该越多。

全局变量使用具有说明性的名字，局部变量用短名字。根据定义，全局变量可以出现在整个程序中的任何地方，因此它们的名字应该足够长，具有足够的说明性，以便使读者能够记得它们是干什么用的。给每个全局变量声明附一个简短注释也非常有帮助：

```
int npending = 0;  // current length of input queue
```

全局函数、类和结构也都应该有说明性的名字，以表明它们在程序里扮演的角色。

相反，对局部变量使用短名字就够了。在函数里， n可能就足够了，npoints也还可以，用numberOfPoints就太过分了。

按常规方式使用的局部变量可以采用极短的名字。例如用 i、j作为循环变量，p、q作为指针，s、t表示字符串等。这些东西使用得如此普遍，采用更长的名字不会有什么益处或收获，可能反而有害。比较：

```
?   for (theElementIndex = 0; theElementIndex < numberOfElements;
?           theElementIndex++)
```

China-pub.com
下载


```
?       elementArray[theElementIndex] = theElementIndex;
```

和

```
        for (i = 0; i < nelems; i++)
            elem[i] = i;
```

人们常常鼓励程序员使用长的变量名，而不管用在什么地方。这种认识完全是错误的，清晰性经常是随着简洁而来的。

现实中存在许多命名约定或者本地习惯。常见的比如：指针采用以 p结尾的变量名，例如 nodep；全局变量用大写开头的变量名，例如 Global；常量用完全由大写字母拼写的变量名，如CONSTANTS等。有些程序设计工场采用的规则更加彻底，他们要求把变量的类型和用途等都编排进变量名字中。例如用 pch说明这是一个字符指针，用 strTo和strFrom表示它们分别是将要被读或者被写的字符串等。至于名字本身的拼写形式，是使用 npending或numPending还是num_pending，这些不过是个人的喜好问题，与始终如一地坚持一种切合实际的约定相比，这些特殊规矩并不那么重要。

命名约定能使自己的代码更容易理解，对别人写的代码也是一样。这些约定也使人在写代码时更容易决定事物的命名。对于长的程序，选择那些好的、具有说明性的、系统化的名字就更加重要。

C++ 的名字空间和Java的包为管理各种名字的作用域提供了方法，能帮助我们保持名字的意义清晰，又能避免过长的名字。

保持一致性。相关的东西应给以相关的名字，以说明它们的关系和差异。

除了太长之外，下面这个Java类中各成员的名字一致性也很差：

```
?   class UserQueue {
?       int noOfItemsInQ, frontOfTheQueue, queueCapacity;
?       public int noOfUsersInQueue() {...}
?   }
```

这里同一个词“队列(queue)”在名字里被分别写为 Q、Queue或queue。由于只能在类型UserQueue里访问，类成员的名字中完全不必提到队列，因为存在上下文。所以：

```
?   queue.queueCapacity
```

完全是多余的。下面的写法更好：

```
class UserQueue {
    int nitems, front, capacity;
    public int nusers() {...}
}
```

因为这时可以如此写：

```
queue.capacity++;
n = queue.nusers();
```

这样做在清晰性方面没有任何损失。在这里还有可做的事情。例如 items和users实际是同一种东西，同样东西应该使用一个概念。

函数采用动作性的名字。函数名应当用动作性的动词，后面可以跟着名词：

```
now = date.getTime();
putchar('\n');
```

对返回布尔类型值(真或者假)的函数命名，应该清楚地反映其返回值情况。下面这样的语句

```
?   if (checkoctal(c)) ...
```

China-pub.com 下载

是不好的，因为它没有指明什么时候返回真，什么时候返回假。而：

```
if (isoctal(c)) ...
```

就把事情说清楚了：如果参数是八进制数字则返回真，否则为假。

要准确。名字不仅是个标记，它还携带着给读程序人的信息。误用的名字可能引起奇怪的程序错误。

本书作者之一写过一个名为isoctal的宏，并且发布使用多年，而实际上它的实现是错误的：

```
?   #define isoctal(c) ((c) >= '0' && (c) <= '8')
```

正确的应该是：

```
#define isoctal(c) ((c) >= '0' && (c) <= '7')
```

这是另外一种情况：名字具有正确的含义，而对应的实现却是错的，一个合情合理的名字掩盖了一个害人的实现。

下面是另一个例子，其中的名字和实现完全是矛盾的：

```
?   public boolean inTable(Object obj) {
?       int j = this.getIndex(obj);
?       return (j == nTable);
?   }
```

函数getIndex如果找到了有关对象，就返回 0到nTable-1之间的一个值；否则返回nTable值。而这里inTable返回的布尔值却正好与它名字所说的相反。在写这段代码时，这种写法未必会引起什么问题。但如果后来修改这个程序，很可能是由别的程序员来做，这个名字肯定会把人弄糊涂。

练习1-1 评论下面代码中名字和值的选择：

```
?   #define TRUE 0
?   #define FALSE 1
?
?   if ((ch = getchar()) == EOF)
?       not_eof = FALSE;
```

练习1-2 改进下面的函数：

```
?   int smaller(char *s, char *t) {
?       if (strcmp(s, t) < 1)
?           return 1;
?       else
?           return 0;
?   }
```

练习1-3 大声读出下面的代码：

```
?   if ((falloc(SMRHSHSCRTCH, S_IFEXT|0644, MAXRODDHSH)) < 0)
?       ...
```

1.2 表达式和语句

名字的合理选择可以帮助读者理解程序，同样，我们也应该以尽可能一目了然的形式写好表达式和语句。应该写最清晰的代码，通过给运算符两边加空格的方式说明分组情况，更一般的是通过格式化的方式来帮助阅读。这些都是很琐碎的事情，但却又是非常有价值的，就像保持书桌整洁能使你容易找到东西一样。与你的书桌不同的是，你的程序代码很可能还会被别人使用。

China-pub.com
下载

用缩行显示程序的结构。采用一种一致的缩行风格，是使程序呈现出结构清晰的最省力的方法。下面这个例子的格式太糟糕了：

```
?    for(n++;n<100;field[n++]='\0');
?    *i = '\0'; return('\n');
```

重新调整格式，可以改得好一点：

```
?    for (n++; n < 100; field[n++] = '\0')
?        ;
?    *i = '\0';
?    return('\n');
```

更好的是把赋值作为循环体，增量运算单独写。这样循环的格式更普通也更容易理解：

```
for (n++; n < 100; n++)
    field[n] = '\0';
*i = '\0';
return '\n';
```

使用表达式的自然形式。表达式应该写得你能大声念出来。含有否定运算的条件表达式比较难理解：

```
?    if (!(block_id < actblks) || !(block_id >= unblocks))
?        ...
```

在两个测试中都用到否定，而它们都不是必要的。应该改变关系运算符的方向，使测试变成肯定的：

```
if ((block_id >= actblks) || (block_id < unblocks))
    ...
```

现在代码读起来就自然多了。

用加括号的方式排除二义性。括号表示分组，即使有时并不必要，加了括号也可能把意图表示得更清楚。在上面的例子里，内层括号就不是必需的，但加上它们没有坏处。熟练的程序员会忽略它们，因为关系运算符(< <= == != >=)比逻辑运算符(&&和||)的优先级更高。

在混合使用互相无关的运算符时，多写几个括号是个好主意。C语言以及与之相关的语言存在很险恶的优先级问题，在这里很容易犯错误。例如，由于逻辑运算符的约束力比赋值运算符强，在大部分混合使用它们的表达式中，括号都是必需的。

```
while ((c = getchar()) != EOF)
    ...
```

字位运算符(&和|)的优先级低于关系运算符(比如==)，不管出现在哪里：

```
?    if (x&MASK == BITS)
?        ...
```

实际上都意味着

```
?    if (x & (MASK==BITS))
?        ...
```

这个表达式所表达的肯定不会是程序员的本意。在这里混合使用了字位运算和关系运算符号，表达式里必须加上括号：

```
if ((x&MASK) == BITS)
    ...
```

如果一个表达式的分组情况不是一目了然的话，加上括号也可能有些帮助，虽然这种括

China-pub.com 下载

号可能不是必需的。下面的代码本来不必加括号：

```
?    leap_year = y % 4 == 0 && y % 100 != 0 || y % 400 == 0;
```

但加上括号，代码将变得更容易理解了：

```
leap_year = ((y%4 == 0) && (y%100 != 0)) || (y%400 == 0);
```

这里还去掉了几个空格：使优先级高的运算符与运算对象连在一起，帮助读者更快地看清表达式的结构。

分解复杂的表达式。 C、C++和Java语言都有很丰富的表达式语法结构和很丰富的运算符。因此，在这些语言里就很容易把一大堆东西塞进一个结构中。下面这样的表达式虽然很紧凑，但是它塞进一个语句里的东西确实太多了：

```
?    *x += (*xp=(2*k < (n-m) ? c[k+1] : d[k--]));
```

把它分解成几个部分，意思更容易把握：

```
if (2*k < n-m)
    *xp = c[k+1];
else
    *xp = d[k--];
*x += *xp;
```

要清晰。 程序员有时把自己无穷尽的创造力用到了写最简短的代码上，或者用在寻求得到结果的最巧妙方法上。有时这种技能是用错了地方，因为我们的目标应该是写出最清晰的代码，而不是最巧妙的代码。

下面这个难懂的计算到底想做什么？

```
?    subkey = subkey >> (bitoff - ((bitoff >> 3) << 3));
```

最内层表达式把bitoff右移3位，结果又被重新移回来。这样做实际上是把变量的最低3位设置为0。从bitoff的原值里面减掉这个结果，得到的将是bitoff的最低3位。最后用这3位的值确定subkey的右移位数。

上面的表达式与下面这个等价：

```
subkey = subkey >> (bitoff & 0x7);
```

要弄清前一个版本的意思简直像猜谜语，而后面这个则又短又清楚。经验丰富的程序员会把它写得更短，换一个赋值运算符：

```
subkey >>= bitoff & 0x7;
```

有些结构似乎总是要引诱人们去滥用它们。运算符?:大概属于这一类：

```
?    child=(!LC&&!RC)?0:(!LC?RC:LC);
```

如果不仔细地追踪这个表达式的每条路径，就几乎没办法弄清它到底是在做什么。下面的形式虽然长了一点，但却更容易理解，其中的每条路径都非常明显：

```
if (LC == 0 && RC == 0)
    child = 0;
else if (LC == 0)
    child = RC;
else
    child = LC;
```

运算符?:适用于短的表达式，这时它可以把4行的if-else程序变成1行。例如这样：

```
max = (a > b) ? a : b;
```

或者下面这样：

```
printf("The list has %d item%s\n", n, n==1 ? "" : "s");
```

China-pub.com
下载

但是它不应该作为条件语句的一般性替换。

清晰性并不等同于简短。短的代码常常更清楚，例如上面移字位的例子。不过有时代码长一点可能更好，如上面把条件表达式改成条件语句的例子。在这里最重要的评价标准是易于理解。

当心副作用。像 ++ 这一类运算符具有副作用，它们除了返回一个值外，还将隐含地改变变量的值。副作用有时用起来很方便，但有时也会成为问题，因为变量的取值操作和更新操作可能不是同时发生。C和C++ 对与副作用有关的执行顺序并没有明确定义，因此下面的多次赋值语句很可能将产生错误结果：

```
?     str[i++] = str[i++] = ' ';
```

这样写的意图是给str中随后的两个位置赋空格值，但实际效果却要依赖于i的更新时刻，很可能把str里的一个位置跳过去，也可能导致只对i实际更新一次。这里应该把它分成两个语句：

```
str[i++] = ' ';
str[i++] = ' ';
```

下面的赋值语句虽然只包含一个增量操作，但也可能给出不同的结果：

```
?     array[i++] = i;
```

如果初始时i的值是3，那么数组元素有可能被设置成3或者4。

不仅增量和减量操作有副作用，I/O也是一种附带地下活动的操作。下面的例子希望从标准输入读入两个互相有关的数：

```
?     scanf("%d %d", &yr, &profit[yr]);
```

这样做很有问题，因为在这个表达式里的一个地方修改了 yr，而在另一个地方又使用它。这样，除非yr的新取值与原来的值相同，否则profit[yr]就不可能是正确的。你可能认为事情依赖于参数的求值顺序，实际情况并不是这样。这里的问题是：scanf的所有参数都在函数被真正调用前已经求好值了，所以&profit[yr]实际使用的总是yr原来的值。这种问题可能在任何语言里发生。纠正的方法就是把语句分解为两个：

```
scanf("%d", &yr);
scanf("%d", &profit[yr]);
```

下面的练习里列举了各种具有副作用的表达式。

练习1-4 改进下面各个程序片段：

```
?     if ( !(c == 'y' || c == 'Y') )
?         return;

?     length = (length < BUFSIZE) ? length : BUFSIZE;

?     flag = flag ? 0 : 1;

?     quote = (*line == '"') ? 1 : 0;

?     if (val & 1)
?         bit = 1;
?     else
?         bit = 0;
```

练习1-5 下面的例子里有什么错？

China-pub.com 下载

```
?    int read(int *ip) {
?        scanf("%d", ip);
?        return *ip;
?    }
?        ...
?    insert(&graph[vert], read(&val), read(&ch));
```

练习1-6 列出下面代码片段在各种求值顺序下可能产生的所有不同的输出：

```
?    n = 1;
?    printf("%d %d\n", n++, n++);
```

在尽可能多的编译系统中试验，看看实际中会发生什么情况。

1.3 一致性和习惯用法

一致性带来的将是更好的程序。如果程序中的格式很随意，例如对数组做循环，一会儿采用下标变量从下到上的方式，一会儿又用从上到下的方式；对字符串一会儿用 strcpy做复制，一会儿又用 for循环做复制；等等。这些变化就会使人很难看清实际上到底是怎么回事。而如果相同计算的每次出现总是采用同样方式，任何变化就预示着是经过了深思熟虑，要求读程序的人注意。

使用一致的缩排和加括号风格。缩排可以显示出程序的结构，那么什么样的缩排风格最好呢？是把花括号放在 if的同一行，还是放在下面一行？程序员们常常就程序的这些编排形式争论不休。实际上，特定风格远没有一致地使用它们重要。不要在这里浪费时间。应该取一种风格，当然作者希望是他们所采用的风格，然后一致地使用。

应该在那些并不必须用花括号的地方都加上它们吗？与一般的圆括号一样，花括号也可以用来消除歧义，但是在使代码更清晰方面的作用却不那么大。为了保持某种一致性，许多程序员总在循环或 if的体外面加花括号。当这里只有一个语句时，加花括号就不是必要的，所以作者倾向于去掉它们。如果你赞成我们的方法，那么就要注意在必需的时候不要忽略了它们，例如，在程序里需要解决“悬空的else(dangling else)”问题时。下面是这方面的一个例子：

```
?    if (month == FEB) {
?        if (year%4 == 0)
?            if (day > 29)
?                legal = FALSE;
?        else
?            if (day > 28)
?                legal = FALSE;
?    }
```

这里的缩排方式把人搞糊涂了，实际上 else隶属于行：

```
?            if (day > 29)
```

代码本身也是错的。如果一个 if紧接在另一个之后，那么请一定加上花括号：

```
?    if (month == FEB) {
?        if (year%4 == 0) {
?            if (day > 29)
?                legal = FALSE;
?        } else {
?            if (day > 28)
?                legal = FALSE;
```

China-pub.com
下载


```
?           }
?     }
```

语法驱动的编辑工具可以帮助避免这类错误。

虽然上面程序里的错误已经修正，但这个结果代码还是很难懂。如果我们用一个变量保存二月的天数，计算过程就很容易看明白了：

```
?     if (month == FEB) {
?         int nday;
?
?         nday = 28;
?         if (year%4 == 0)
?             nday = 29;
?         if (day > nday)
?             legal = FALSE;
?     }
```

这段代码实际上还是错的——2000年是闰年，而1900和2100都不是。要把现在这个结构改正确是非常容易的。

此外，如果你工作在一个不是自己写的程序上，请注意保留程序原有的风格。当你需要做修改时，不要使用你自己的风格，即使你特别喜欢它。程序的一致性比你本人的习惯更重要，因为这将使随你之后的其他人生活得更容易些。

为了一致性，使用习惯用法。 和自然语言一样，程序设计语言也有许多惯用法，也就是那些经验丰富的程序员写常见代码片段的习惯方式。在学习一个语言的过程中，一个中心问题就是逐渐熟悉它的习惯用法。

常见习惯用法之一是循环的形式。考虑在C、C++和Java中逐个处理n元数组中各个元素的代码，例如要对这些元素做初始化。有人可能写出下面的循环：

```
?     i = 0;
?     while (i <= n-1)
?         array[i++] = 1.0;
```

或者是这样的：

```
?     for (i = 0; i < n; )
?         array[i++] = 1.0;
```

也可能是：

```
?     for (i = n; --i >= 0; )
?         array[i] = 1.0;
```

所有这些都正确，而习惯用法的形式却是：

```
for (i = 0; i < n; i++)
    array[i] = 1.0;
```

这并不是一种随意的选择：这段代码要求访问n元数组里的每个元素，下标从0到n-1。在这里所有循环控制都被放在一个for里，以递增顺序运行，并使用++的习惯形式做循环变量的更新。这样做还保证循环结束时下标变量的值是一个已知值，它刚刚超出数组里最后元素的位置。熟悉C语言的人不用琢磨就能理解它，不加思考就能正确地写出它来。

C++或Java里常见的另一种形式是把循环变量的声明也包括在内：

```
for (int i = 0; i < n; i++)
    array[i] = 1.0;
```

下面是在C语言里扫描一个链表的标准循环：

```
for (p = list; p != NULL; p = p->next)
    ...
```

China-pub.com 下载

同样，所有的控制都放在一个for里面。

对于无穷循环，我们喜欢用：

```
for (;;)
    ...
```

但

```
while (1)
    ...
```

也很流行。请不要使用其他形式。

缩排也应该采用习惯形式。下面这种垂直排列会妨碍人的阅读，它更像三个语句而不像一个循环：

```
?    for (
?        ap = arr;
?        ap < arr + 128;
?        *ap++ = 0
?        )
?    {
?      ;
?    }
```

写成标准的循环形式，读起来就容易多了：

```
for (ap = arr; ap < arr+128; ap++)
    *ap = 0;
```

这种故意拉长的格式还会使代码摊到更多的页或显示屏去，进一步妨碍人的阅读。

常见的另一个惯用法是把一个赋值放进循环条件里：

```
while ((c = getchar()) != EOF)
    putchar(c);
```

do-while循环远比for和while循环用得少，因为它将至少执行循环体一次，在代码的最后而不是开始执行条件测试。这种执行方式在许多情况下是不正确的，例如下面这段重写的使用getchar的循环：

```
?    do {
?        c = getchar();
?        putchar(c);
?    } while (c != EOF);
```

在这里测试被放在对putchar的调用之后，将使这个代码段无端地多写出一个字符。只有在某个循环体总是必须至少执行一次的情况下，使用 do-while循环才是正确的。后面会看到这种例子。

一致地使用习惯用法还有另一个优点，那就是使非标准的循环很容易被注意到，这种情况常常预示着有什么问题：

```
?    int i, *iArray, nmemb;
?
?    iArray = malloc(nmemb * sizeof(int));
?    for (i = 0; i <= nmemb; i++)
?        iArray[i] = i;
```

在这里分配了nmemb个项的空间，从iArray[0]到iArray[nmemb-1]。但由于采用的是 <=

China-pub.com
下载

做循环测试，程序执行将超出数组尾部，覆盖掉存储区中位于数组后面的内容。不幸的是，有许多像这样的错误没能及时查出来，直到造成了很大的损害。

C和C++ 中也有为字符串分配空间及操作它们的习惯写法。不采用这种做法的代码常常就隐藏着程序错误：

```
?     char *p, buf[256];
?
?     gets(buf);
?     p = malloc(strlen(buf));
?     strcpy(p, buf);
```

绝不要使用函数gets，因为你没办法限制它由输入那儿读入内容的数量。这常常会导致一个安全性问题。第6章还会再来讨论这个问题，那里要说明选择 fgets总是更好的。上面代码段里还有另一个问题：strlen求出的值没有计入串结尾的‘\0’字符，而strcpy却将复制它。所以这里分配的空间实际上是不够的，这将使 strcpy的写入超过所分配空间的界限。习惯写法是：

```
p = malloc(strlen(buf)+1);
strcpy(p, buf);
```

或在C++里：

```
p = new char[strlen(buf)+1];
strcpy(p, buf);
```

如果你在这里没有看见 +1，就要当心。

在Java里不会遇到这个特殊问题，那里的字符串不是用零结尾的数组表示，数组的下标也将受到检查。这就使Java不会出现超出数组界限访问的问题。

许多C和C++ 环境中提供了另一个库函数 strdup，它通过调用malloc和strcpy建立字符串的拷贝。有了这个函数，要避免上述错误就变得更简单了。可惜， strdup不是ANSI C标准中的内容。

还有一点，无论是上面的原始代码，还是其修正版本里都没有检查 malloc的返回值。我们忽略这个改进，是为了集中精力处理这里的主要问题。在实际程序中，对于 malloc、realloc、strdup及任何牵涉到存储分配的函数，它们的返回值都必须做检查。

用else-if表达多路选择。多路选择的习惯表示法是采用一系列的if ... else if ... else，形式如下：

```
if (condition₁)
    statement₁
else if (condition₂)
    statement₂
...
else if (conditionₙ)
    statementₙ
else
    default-statement
```

这里的条件(condition)从上向下读，遇到第一个能够满足的条件，就执行对应的语句，而随后的结构都跳过去。在这里，各个语句部分可以只是单个语句，也可以是由花括号括起的一组语句。最后一个else处理默认情况，或说是处理没有选中其他部分时的情况。如果不存在默认动作，尾随的else部分就可以没有。另一个更好的办法是利用它给出一个错误信息，

China-pub.com 下载

以帮助捕捉“不可能发生”的情况。

在这里应该把所有的else垂直对齐，而不是分别让每个else与对应的if对齐。采用垂直对齐能够强调所有测试都是顺序进行的，而且能防止语句不断退向页的右边缘。

一系列嵌套的if语句通常是说明了一段粗劣笨拙的代码，或许就是真正的错误：

```
?   if (argc == 3)
?       if ((fin = fopen(argv[1], "r")) != NULL)
?           if ((fout = fopen(argv[2], "w")) != NULL) {
?               while ((c = getc(fin)) != EOF)
?                   putc(c, fout);
?               fclose(fin); fclose(fout);
?           } else
?               printf("Can't open output file %s\n", argv[2]);
?       else
?           printf("Can't open input file %s\n", argv[1]);
?   else
?       printf("Usage: cp inputfile outputfile\n");
```

这些if要求读它的人在头脑里维持一个下推堆栈，不断记住前面做了什么测试，读到某个地方能把记住的内容弹出来，直到确定了对应动作(如果还记得的话)。由于这里最多就是做一个动作，改变测试顺序完全可以得到一个更清晰的版本。这里我们还纠正了原版本中的资源流失问题[⊖]。

```
if (argc != 3)
    printf("Usage: cp inputfile outputfile\n");
else if ((fin = fopen(argv[1], "r")) == NULL)
    printf("Can't open input file %s\n", argv[1]);
else if ((fout = fopen(argv[2], "w")) == NULL) {
    printf("Can't open output file %s\n", argv[2]);
    fclose(fin);
} else {
    while ((c = getc(fin)) != EOF)
        putc(c, fout);
    fclose(fin);
    fclose(fout);
}
```

从开始向下读，直到遇到第一个值为真的测试，转而去做对应的动作，然后从最后的 else后面继续下去。这里要遵守的规则是：一个判断应该尽可能接近它所对应的动作。也就是说，一旦做过了一个测试，马上就应该去做某些事情。

人们往往企图重复使用某段代码，结果常写出一段紧紧纠缠在一起的程序：

```
?   switch (c) {
?   case '-':  sign = -1;
?   case '+':  c = getchar();
?   case '.':  break;
?   default:   if (!isdigit(c))
?                  return 0;
?   }
```

在这个开关语句里，写了一个狡猾的从上面掉下[⊜]的语句序列，目的不过是避免重写那仅有一行的代码。这样做也不符合习惯写法：case语句最后都应该写一个break，如果偶尔有例外，

⊖ 在源代码中，某些情况下程序执行中将不会释放文件缓冲区，从而造成未释放资源的流失，使它们可能无法重新投入使用。作者的话就是指出原来存在这个错误。—— 译者

⊜ 指在前一个case语句序列最后不写break，使执行直接“掉”到下面的case语句序列中去。—— 译者

China-pub.com
下载

在那里一定要加上注释。按传统编排方式和结构写出来的内容很容易读，虽然稍微长了一点：

```
?    switch (c) {
?    case '-':
?        sign = -1;
?        /* fall through */
?    case '+':
?        c = getchar();
?        break;
?    case '.':
?        break;
?    default:
?        if (!isdigit(c))
?            return 0;
?        break;
?    }
```

增加长度也不符合提高清晰性的要求。还好，对于这种不常见的结构，用一系列 else-if语句写可能更清楚些：

```
if (c == '-') {
    sign = -1;
    c = getchar();
} else if (c == '+') {
    c = getchar();
} else if (c != '.' && !isdigit(c)) {
    return 0;
}
```

围在每个单行语句块外面的花括号强调了它们是平行结构。

“从上面掉下”的方式在一种情况下是可以接受的，那就是几个 case使用共同的代码段。常规的编排形式是：

```
case '0':
case '1':
case '2':
    ...
    break;
```

这里不需要任何注释。

练习1-7 把下面的C/C++程序例子改得更清晰些：

```
?    if (istty(stdin))  ;
?    else if (istty(stdout))  ;
?         else if (istty(stderr))  ;
?              else return(0);
```

```
?    if (retval != SUCCESS)
?    {
?        return (retval);
?    }
?    /* All went well! */
?    return SUCCESS;
```

```
?    for (k = 0; k++ < 5; x += dx)
?        scanf("%lf", &dx);
```

China-pub.com 下载

练习1-8 确定下面的Java程序段中的错误，并把它改写为一个符合习惯的循环。

```
?    int count = 0;
?    while (count < total) {
?        count++;
?        if (this.getName(count) == nametable.userName()) {
?            return (true);
?        }
?    }
```

1.4 函数宏

老的C语言程序员中有一种倾向，就是把很短的执行频繁的计算写成宏，而不是定义为函数。完成I/O的getchar，做字符测试的isdigit都是得到官方认可的例子。人们这样做最根本的理由就是执行效率：宏可以避免函数调用的开销。实际上，即使是在 C语言刚诞生时(那时的机器非常慢，函数调用的开销也特别大)，这个论据也是很脆弱的，到今天它就更无足轻重了。有了新型的机器和编译程序，函数宏的缺点就远远超过它能带来的好处。

避免函数宏。在C++ 里，在线函数更削减了函数宏的用武之地，在 Java里根本就没有宏这种东西。即使是在C语言里，它们带来的麻烦也比解决的问题更多。

函数宏最常见的一个严重问题是：如果一个参数在定义中出现多次，它就可能被多次求值。如果调用时的实际参数带有副作用，结果就会产生一个难以捉摸的错误。下面的代码段来自某个<ctype.h>，其意图是实现一种字符测试：

```
?    #define isupper(c) ((c) >= 'A' && (c) <= 'Z')
```

请注意，参数c在宏的体里出现了两次。如果isupper在下面的上下文中调用：

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

那么，每当遇到一个大于等于 A的字符，程序就会将它丢掉，而下一个字符将被读入并去与 Z做比较。C语言标准是仔细写出的，它允许将 isupper及类似函数定义为宏，但要求保证它们的参数只求值一次。因此，上面的实现是错误的。

直接使用ctype提供的函数总比自己实现它们更好。如果希望更安全些，那么就一定不要嵌套地使用像getchar这种带有副作用的函数。我们重写上面的测试，把一个表达式改成两个，这里还为捕捉文件结束留下机会：

```
while ((c = getchar()) != EOF && isupper(c))
    ...
```

有时多次求值带来的是执行效率问题，而不是真正的错误。考虑下面这个例子：

```
?    #define ROUND_TO_INT(x) ((int) ((x)+(((x)>0)?0.5:-0.5)))
?        ...
?    size = ROUND_TO_INT(sqrt(dx*dx + dy*dy));
```

这种写法使平方根函数的计算次数比实际需要多了一倍。甚至对于很简单的实际参数，像ROUND_TO_INT体这样的复杂表达式也会转换成许多指令。这里确实应该把它改成一个函数，在需要时调用。宏将在它每次被调用的地方进行实例化，结果会导致被编译的程序变大 (C++的在线函数也存在这个缺点)。

给宏的体和参数都加上括号。如果你真的要使用函数宏，那么请特别小心。宏是通过文本替换方式实现的：定义体里的参数被调用的实际参数替换，得到的结果再作为文本去替换原来

China-pub.com
下载

的调用段。这种做法与函数不同，常给人带来一些麻烦。假如 square是个函数，表达式：

```
1 / square(x)
```

的工作将很正常。而如果它的定义如下：

```
?    #define square(x)   (x) * (x)
```

上面表达式将被展开成一个错误的内容：

```
?    1 / (x) * (x)
```

这个宏应该定义为：

```
#define square(x)   ((x) * (x))
```

这里所有的括号都是必需的。即使是在宏定义里完全加上括号，也不可能解决前面所说的多次求值问题。所以，如果一个操作比较复杂，或者它很具一般性，值得包装起来，那么还是应该使用函数。

C++ 提供的在线函数既避免了语法方面的麻烦，而且又可得到宏能够提供的执行效率，很适合用来定义那些设置或者提取一个值的短小函数。

练习1-9 确定下面的宏定义中的问题：

```
?    #define ISDIGIT(c) ((c >= '0') && (c <= '9')) ? 1 : 0
```

1.5 神秘的数

神秘的数包括各种常数、数组的大小、字符位置、变换因子以及程序中出现的其他以文字形式写出的数值。

(1) 给神秘的数起个名字。作为一个指导原则，除了 0和1之外，程序里出现的任何数大概都可以算是神秘的数，它们应该有自己的名字。在程序源代码里，一个具有原本形式的数对其本身的重要性或作用没提供任何指示性信息，它们也导致程序难以理解和修改。下面的片段摘自一个在24×80的终端屏幕上打印字母频率的直方图程序，由于其中存在一些神秘的数，程序的意义变得很不清楚：

```
?        fac = lim / 20;     /* set scale factor */
?        if (fac < 1)
?            fac = 1;
?                            /* generate histogram */
?        for (i = 0, col = 0; i < 27; i++, j++) {
?            col += 3;
?            k = 21 - (let[i] / fac);
?            star = (let[i] == 0) ? ' ' : '*';
?            for (j = k; j < 22; j++)
?                draw(j, col, star);
?        }
?        draw(23, 2, ' '); /* label x axis */
?        for (i = 'A'; i <= 'Z'; i++)
?            printf("%c  ", i);
```

在这段代码里包含许多数值，如 20、21、22、23、27等等。它们应该是互相有关系的……但是……它们确实有关系吗？实际上，在这个程序里应该只有三个数是重要的： 24是屏幕的行数；80是列数；还有26，它是字母表中的字母个数。但这些数在代码中都没出现，这就使上面那些数显得更神秘了。

China-pub.com 下载

通过给上面的计算中出现的各个数命名，我们可以把代码弄得更清楚些。我们发现，例如数字3是由(80 - 1)/26得到的，而let应该有26个项，而不是27个(这个超一(off-by-one)错误可能是由于写程序的人把屏幕坐标当作从1开始而引起的)。通过一些简化后，我们得到的结果代码是：

```
enum {
    MINROW    = 1,                /* top edge */
    MINCOL    = 1,                /* left edge */
    MAXROW    = 24,               /* bottom edge (<=) */
    MAXCOL    = 80,               /* right edge (<=) */
    LABELROW  = 1,                /* position of labels */
    NLET      = 26,               /* size of alphabet */
    HEIGHT    = MAXROW - 4,       /* height of bars */
    WIDTH     = (MAXCOL-1)/NLET   /* width of bars */
};
    ...
    fac = (lim + HEIGHT-1) / HEIGHT;    /* set scale factor */
    if (fac < 1)
        fac = 1;
    for (i = 0; i < NLET; i++) {    /* generate histogram */
        if (let[i] == 0)
            continue;
        for (j = HEIGHT - let[i]/fac; j < HEIGHT; j++)
            draw(j+1 + LABELROW, (i+1)*WIDTH, '*');
    }
    draw(MAXROW-1, MINCOL+1, ' ');  /* label x axis */
    for (i = 'A'; i <= 'Z'; i++)
        printf("%c  ", i);
```

现在，主循环到底做什么已经很清楚了：它是一个熟悉的从 0到NLET的循环，是一个对数据(数组)元素操作的循环。程序里对draw的调用也同样容易理解，因为像MAXROW和MINCOL这样的词提醒我们实际参数的顺序。更重要的是，现在我们已经很容易把这个程序修改为能够对付其他的屏幕大小或不同的数据了。数被揭掉了神秘的面纱，代码的意义也随之一目了然了。

把数定义为常数，不要定义为宏。 C程序员的传统方式是用#define行来对付神秘的数值。C语言预处理程序是一个强有力的工具，但是它又有些鲁莽。使用宏进行编程是一种很危险的方式，因为宏会在背地里改变程序的词法结构。我们应该让语言去做正确的工作[⊖]。在C和C++ 里，整数常数可以用枚举语句声明，就像上面的例子里所做的那样。在 C++里任何类型都可使用const声明的常数：

```
const int MAXROW = 24, MAXCOL = 80;
```

在Java中可以用final声明：

```
static final int MAXROW = 24, MAXCOL = 80;
```

C语言里也有const值，但它们不能用作数组的界。这样， enum就是C中惟一可用的选择了。

使用字符形式的常量，不要用整数。 人们常用在<ctype.h>里的函数，或者用与它们等价的内容检测字符的性质。有一个测试写成这样：

```
?    if (c >= 65 && c <= 90)
?        ...
```

⊖ 注意，C预处理命令不是C语言本身的组成部分，而是一组辅助成分。这里说“让语言……”，也就是说不要用预处理命令做。——译者

China-pub.com
下载

这种写法将完全依赖于特殊的字符表示方式。写成下面这样更好一些：

```
if (c >= 'A' && c <= 'Z')
    ...
```

但是，如果在某个编码字符集里的字母编码不是连续的，或者其中还夹有其他字母，那么这种描述的效果就是错的。最好是直接使用库函数，在C和C++ 里写：

```
if (isupper(c))
    ...
```

或在Java里面：

```
if (Character.isUpperCase(c))
    ...
```

与此类似的还有另一个问题，那就是程序里许多上下文中经常出现的 0。虽然编译系统会把它转换为适当类型，但是，如果我们把每个 0的类型写得更明确更清楚，对读程序的人理解其作用是很有帮助的。例如，用 (void*)0或NULL表示C里的空指针值，用'\0'而不是0表示字符串结尾的空字节。也就是说，不要写：

```
?    str = 0;
?    name[i] = 0;
?    x = 0;
```

应该写成：

```
str = NULL;
name[i] = '\0';
x = 0.0;
```

我们赞成使用不同形式的显式常数，而把 0仅留做整数常量。采用这些形式实际上指明了有关值的用途，能起一点文档作用。可惜的是，在 C++ 里人们都已接受了用 0(而不是NULL)表示空指针。Java为解决这个问题采用了一种更好的方法，它定义了一个关键字 null，用来表示一个对象引用实际上并没有引用任何东西。

利用语言去计算对象的大小。不要对任何数据类型使用显式写出来的大小。例如，我们应该用sizeof(int) 而不是 2或者 4。基于同样原因，写 sizeof(array[0]) 可能比sizeof(int) 更好，因为即使是数组的类型改变了，也没有什么东西需要改变。

利用运算符sizeof常常可以很方便地避免为数组大小引进新名字。例如，写：

```
char buf[1024];

fgets(buf, sizeof(buf), stdin);
```

这里的缓冲区大小仍然是个神秘数。但是它只在这个声明中出现了一次。为局部数组的大小引进一个新名字价值并不大，而写出的代码能在数据大小或类型改变的情况下不需要任何改动，这一点肯定是有价值的。

Java语言中的数组有一个length域，它给出数组的元素个数：

```
char buf[] = new char[1024];

for (int i = 0; i < buf.length; i++)
    ...
```

在C和C++里没有与 .length对应的内容。但是，对于那些可以看清楚的数组 (不是指针)，下面的宏定义能计算出数组的元素个数：

```
#define NELEMS(array) (sizeof(array) / sizeof(array[0]))
```

China-pub.com 下载

```
double dbuf[100];

for (i = 0; i < NELEMS(dbuf); i++)
    ...
```

在这里，数组大小只在一个地方设置。如果数组的大小改变，其余代码都不必改动。对函数参数的多次求值在这里也不会出问题，因为它不会出现任何副作用，事实上，这个计算在程序编译时就已经做完了。这是宏的一个恰当使用，因为它做了某种函数无法完成的工作，从数组声明计算出它的大小。

练习1-10　如何重写下面定义，使出错的可能性降到最小？

```
?   #define FT2METER    0.3048
?   #define METER2FT    3.28084
?   #define MI2FT       5280.0
?   #define MI2KM       1.609344
?   #define SQMI2SQKM   2.589988
```

1.6 注释

注释是帮助程序读者的一种手段。但是，如果在注释中只说明代码本身已经讲明的事情，或者与代码矛盾，或是以精心编排的形式干扰读者，那么它们就是帮了倒忙。最好的注释是简洁地点明程序的突出特征，或是提供一种概观，帮助别人理解程序。

不要大谈明显的东西。注释不要去说明明白白的事，比如 i++能够将i值加1等等。下面是我们认为最没有价值的一些注释：

```
?   /*
?    * default
?    */
?   default:
?       break;

?   /* return SUCCESS */
?   return SUCCESS;

?   zerocount++;    /* Increment zero entry counter */

?   /* Initialize "total" to "number_received" */
?   node->total = node->number_received;
```

所有这些都该删掉，它们不过是一些无谓的喧嚣。

注释应该提供那些不能一下子从代码中看到的东西，或者把那些散布在许多代码里的信息收集到一起。当某些难以捉摸的事情出现时，注释可以帮助澄清情况。如果操作本身非常明了，重复谈论它们就是画蛇添足了：

```
?   while ((c = getchar()) != EOF && isspace(c))
?       ;                       /* skip white space */
?   if (c == EOF)               /* end of file */
?       type = endoffile;
?   else if (c == '(')          /* left paren */
?       type = leftparen;
?   else if (c == ')')          /* right paren */
?       type = rightparen;
?   else if (c == ';')          /* semicolon */
?       type = semicolon;
```

China-pub.com
下载


```
?     else if (is_op(c))            /* operator */
?         type = operator;
?     else if (isdigit(c))          /* number */
?         ...
```

这些注释也都应该删除，因为仔细选择的名字已经携带着有关信息。

给函数和全局数据加注释。注释当然可以有价值。对于函数、全局变量、常数定义、结构和类的域等，以及任何其他加上简短说明就能够帮助理解的内容，我们都应该为之提供注释。

全局变量常被分散使用在整个程序中的各个地方，写一个注释可以帮人记住它的意义，也可以作为参考。下面是从本书第3章取来的一个例子：

```
struct State {  /* prefix + suffix list */
    char    *pref[NPREF];   /* prefix words */
    Suffix  *suf;           /* list of suffixes */
    State   *next;          /* next in hash table */
};
```

放在每个函数前面的注释可以成为帮人读懂程序的台阶。如果函数代码不太长，在这里写一行注释就足够了。

```
// random: return an integer in the range [0..r-1].
int random(int r)
{
    return (int)(Math.floor(Math.random()*r));
}
```

有些代码原本非常复杂，可能是因为算法本身很复杂，或者是因为数据结构非常复杂。在这些情况下，用一段注释指明有关文献对读者也很有帮助。此外，说明做出某种决定的理由也很有价值。下面程序的注释介绍了逆离散余弦变换（inverse discrete cosine transform, DCT)的一个特别高效的实现，它用在一个JPEG图像解码器里：

```
/*
 * idct: Scaled integer implementation of
 * Inverse two dimensional 8x8 Discrete Cosine Transform,
 * Chen-Wang algorithm (IEEE ASSP-32, pp 803-816, Aug 1984)
 *
 * 32-bit integer arithmetic (8-bit coefficients)
 * 11 multiplies, 29 adds per DCT
 *
 * Coefficients extended to 12 bits for
 * IEEE 1180-1990 compliance
 */

static void idct(int b[8*8])
{
    ...
}
```

这个很有帮助的注释点明了参考文献，简短地描述了所使用的数据，说明了算法的执行情况，还说明为什么原来的代码应该修改，以及做了怎样的修改等等。

不要注释差的代码，重写它。应该注释所有不寻常的或者可能迷惑人的内容。但是如果注释的长度超过了代码本身，可能就说明这个代码应该修改了。下面的例子是一个长而混乱的注释和一个条件编译的查错打印语句，它们都是为了解释一个语句：

China-pub.com 下载

```
?     /* If "result" is 0 a match was found so return true (non-zero).
?        Otherwise, "result" is non-zero so return false (zero). */
?
?     #ifdef DEBUG
?     printf("*** isword returns !result = %d\n", !result);
?     fflush(stdout);
?     #endif
?
?     return(!result);
```

否定性的东西很不好理解，应该尽量避免。在这里，部分问题来自一个毫无信息的变量名字result。改用另一个更具说明性的名字matchfound之后，注释就再没有存在的必要，打印语句也变得清楚了：

```
#ifdef DEBUG
printf("*** isword returns matchfound = %d\n", matchfound);
fflush(stdout);
#endif

return matchfound;
```

不要与代码矛盾。许多注释在写的时候与代码是一致的。但是后来由于修正错误，程序改变了，可是注释常常还保持着原来的样子，从而导致注释与代码的脱节。这很可能是本章开始的那个例子的合理解释。

无论产生脱节的原因何在，注释与代码矛盾总会使人感到困惑。由于误把错误注释当真，常常使许多实际查错工作耽误了大量时间。所以，当你改变代码时，一定要注意保证其中的注释是准确的。

注释不仅需要与代码保持一致，更应该能够支持它。下面的例子里的注释是正确的，它正确地解释了后面两行的用途。但细看又会发现它与代码矛盾，注释中谈的是换行，而代码中说的则是空格：

```
?     time(&now);
?     strcpy(date, ctime(&now));
?     /* get rid of trailing newline character copied from ctime */
?     i = 0;
?     while(date[i] >= ' ') i++;
?     date[i] = 0;
```

一个可能的改进是采用惯用法重写代码：

```
?     time(&now);
?     strcpy(date, ctime(&now));
?     /* get rid of trailing newline character copied from ctime */
?     for (i = 0; date[i] != '\n'; i++)
?         ;
?     date[i] = '\0';
```

现在注释和代码一致了。但是这两者都还可以进一步改进，应该写得更直截了当些。这里要解决的问题就是删除函数ctime返回时放在字符串最后的换行字符。注释里应该说明这个情况，代码也应该是做这件事：

```
time(&now);
strcpy(date, ctime(&now));
/* ctime() puts newline at end of string; delete it */
date[strlen(date)-1] = '\0';
```

最后这个表达式是在C语言里截去字符串最后字符的习惯写法。现在代码变短了，注释也支持它，解释了为什么这个语句需要做。

China-pub.com
下载

澄清情况，不要添乱。注释应该在困难的地方尽量帮助读者，而不是给他们设置障碍。下面的例子中遵循了我们为函数写注释的建议，解释函数不寻常的特征。但是这里的函数是strcmp，这是个标准的东西，具有人们熟悉的界面，它的不寻常特性对手头工作来说根本不重要：

```
?    int strcmp(char *s1, char *s2)
?    /* string comparison routine returns -1 if s1 is */
?    /* above s2 in an ascending order list, 0 if equal */
?    /* 1 if s1 below s2 */
?    {
?        while(*s1==*s2)  {
?            if(*s1=='\0')  return(0);
?            s1++;
?            s2++;
?        }
?        if(*s1>*s2) return(1);
?        return(-1);
?    }
```

如果真的需要用许多话来解释出了什么事，这可能就说明了有关代码应该重写。上面的代码可以改进，但其中真正的问题在注释，它几乎有代码本身那么长，而且也非常含糊（例如，什么是这里的above？)。我们很难说这段代码难以理解，但是，由于它实现的是一个标准函数，它的注释就应该在这方面有所帮助，应该概述其行为，告诉我们哪里有它原来的定义。下面的注释是合适的：

```
/* strcmp: return < 0 if s1<s2, > 0 if s1>s2, 0 if equal */
/*         ANSI C, section 4.11.4.2 */
int strcmp(const char *s1, const char *s2)
{
    ...
}
```

学生常被告之应该注释所有的内容。职业程序员也常被要求注释他们的所有代码。但是，应该看到，盲目遵守这些规则的结果却可能是丢掉了注释的真谛。注释是一种工具，它的作用就是帮助读者理解程序中的某些部分，而这些部分的意义不容易通过代码本身直接看到。我们应该尽可能地把代码写得容易理解。在这方面你做得越好，需要写的注释就越少。好的代码需要的注释远远少于差的代码。

练习1-11 评论下面的注释：

```
?    void dict::insert(string& w)
?    // returns 1 if w in dictionary, otherwise returns 0


?    if (n > MAX || n % 2 > 0) // test for even number


?    // Write a message
?    // Add to line counter for each line written
?
?    void write_message()
?    {
?        // increment line counter
?        line_number = line_number + 1;
?        fprintf(fout, "%d  %s\n%d  %s\n%d  %s\n",
?            line_number, HEADER,
```

China-pub.com 下载

```
?             line_number + 1, BODY,
?             line_number + 2, TRAILER);
?         // increment line counter
?         line_number = line_number + 2;
?     }
```

1.7 为何对此费心

在这一章里，我们谈论的主要问题是程序设计的风格：具有说明性的名字、清晰的表达式、直截了当的控制流、可读的代码和注释，以及在追求这些内容时一致地使用某些规则和惯用法的重要性。没人会争辩说这些是不好的。

但是，为什么要为风格而煞费苦心？只要程序能运行，谁管它看起来是什么样子？把它弄得漂亮点是不是花费了太多时间？这些规则难道没有随意性吗？

我们的回答是：书写良好的代码更容易阅读和理解，几乎可以保证其中的错误更少。进一步说，它们通常比那些马马虎虎地堆起来的、没有仔细推敲过的代码更短小。在这个拼命要把代码送出门、去赶上最后期限的时代，人们很容易把风格丢在一旁，让将来去管它们吧。但是，这很可能是一个代价非常昂贵的决定。本章的一些例子说明了，如果对好风格问题重视不够，程序中哪些方面可能出毛病。草率的代码是很坏的代码，它不仅难看、难读，而且经常崩溃。

这里最关键的结论是：好风格应该成为一种习惯。如果你在开始写代码时就关心风格问题，如果你花时间去审视和改进它，你将会逐渐养成一种好的编程习惯。一旦这种习惯变成自动的东西，你的潜意识就会帮你照料许多细节问题，甚至你在工作压力下写出的代码也会更好。

补充阅读

就像我们在本章开始时说的，写出好的代码与书写好的英文有许多共同之处。 Strunk和White的《风格的要素》(The Elements of Style， Allyn & Bacon)仍然是关于如何写好英文的最好的简短的书。

本章采用了Brian Kernighan和P. J. Plauger的《程序设计风格的要素》 (The Elements of Programming Style， McGraw-Hill， 1978)中的方式。 Steve Maguire的《写可靠的代码》(Writing Solid Code， Microsoft Press， 1993)是有关程序设计各方面的忠告的一本佳作。 Steve McConnell的《完整编程》(Code Complete， Microsoft Press， 1993)和Peter van der Linden的《熟练的C程序设计：深入C的奥密》(Expert C Programming: Deep C Secret， Prentice Hall， 1994)中都有一些关于程序风格的有益讨论。

China-pub.com
下载

第2章　算法与数据结构

总而言之，只有熟悉了这个领域的工具和技术才能对特殊的问题提供正确解答，只有丰富的经验才能提供坚实的专业性结果。

Raymond Fielding，《特殊效果立体电影的技术》

算法和数据结构的研究是计算机科学的重要基石。这是一个富集优雅技术和复杂数学分析结果的领域。这个领域并不是理论嗜好者们的乐园和游戏的场所：一个好的算法或数据结构可能使某个原来需要用成年累月才能完成的问题在分秒之中得到解决。

在某些特殊领域，例如图形学、数据库、语法分析、数值分析和模拟等等，解决问题的能力几乎完全依赖于最新的算法和数据结构。如果你正要进入一个新领域去开发程序，那么首先需要弄清楚在这里已经有了些什么，以免无谓地把时间浪费在别人早已做好的东西上。

每个程序都要依靠算法与数据结构，但很少有程序依赖于必须发明一批全新的东西。即使是很复杂的程序，比如在编译器或者网络浏览器里，主要的数据结构也是数组、表、树和散列表等等。如果在一个程序里要求某些更精巧的东西，它多半也是基于这些简单东西构造起来的。因此，对大部分程序员而言，所需要的是知道有哪些合适的、可用的算法和数据结构，知道如何在各种可以互相替代的东西之中做出选择。

这里要讲的是一个核桃壳里的故事。实际上基本算法只有屈指可数的几个，它们几乎出现在每个程序中，可能已经被包含在程序库里，这就是基本检索和排序。与此类似，几乎所有的数据结构都是从几个基本东西中产生出来的。这样，本章包含的材料对于每个程序员都是熟悉的。我们写了些能够实际工作的程序，以使下面的讨论更具体。如果需要，你可以抄录这些代码，但要事先检查用的是哪种语言，你是否有它所需要的库。

2.1　检索

要存储静态的表格式数据当然应该用数组。由于可以做编译时的初始化，构建这种数组非常容易(Java的初始化在运行中进行。只要数组不是很大，这就是一个无关紧要的实现细节)。要检查学生练习中是否对某些废话用得太多，在程序里可能会定义：

```
char *flab[] = {
    "actually",
    "just",
    "quite",
    "really",
    NULL
};
```

检索程序必须知道数组里有多少元素。对这个问题的一种处理方法是传递一个数组长度参数。这里采用的是另一种方法，在数组最后放一个NULL作为结束标志。

```
/* lookup: sequential search for word in array */
int lookup(char *word, char *array[])
```

China-pub.com 下载

```
{
    int i;

    for (i = 0; array[i] != NULL; i++)
        if (strcmp(word, array[i]) == 0)
            return i;
    return -1;
}
```

在C和C++ 里，字符串数组参数可以说明为char *array[或者char **array。虽然这两种形式是等价的，但前一种形式能把参数的使用方式更清楚地表现出来。

这里采用的检索算法称为顺序检索，它逐个查看每个数据元素是不是要找的那一个。如果数据的数目不多，顺序检索就足够快了。标准库提供了一些函数，它们可以处理某些特定类型的顺序检索问题。例如，函数strchr和strstr能在C或C++ 字符串里检索给定的字符或子串，Java的String类里有一个indexOf方法，C++的类属算法find几乎能用于任何数据类型。如果对某个数据类型有这种函数，我们就应该直接用它。

顺序检索非常简单，但是它的工作量与被检索数据的数目成正比。如果要找的数据并不存在，数据量加倍也会使检索的工作量加倍。这是一种线性关系——运行时间是数据规模的线性函数，因此这种检索也被称为线性检索。

下面的程序段来自一个分析 HTML的程序，这里有一个具有实际规模的数组，其中为成百个独立的字符定义了文字名：

```
typedef struct Nameval Nameval;
struct Nameval {
    char    *name;
    int     value;
};

/* HTML characters, e.g. AElig is ligature of A and E. */
/* Values are Unicode/ISO10646 encoding. */

Nameval htmlchars[] = {
    "AElig",    0x00c6,
    "Aacute",   0x00c1,
    "Acirc",    0x00c2,
    /* ... */
    "zeta",     0x03b6,
};
```

对这样的大数组，使用**二分检索**方法效率将更高些。二分检索是人在字典里查单词时采用的一种有条理性的方法：先看位于中间的元素，如果那里的值比想要找的值更大，那么就去查前边的一半；否则就去查后边一半。反复这样做，直到要找的东西被发现，或者确定它根本不存在为止。

为了能做二分检索，表格本身必须是排好序的，就像上面做的那样 (无论如何，这样做都是一种好的风格，人们在排序的表格里找东西也更快一些)。另一方面，程序还必须知道表格的长度，第1章定义的NELEMS宏可以用在这里：

```
printf("The HTML table has %d words\n", NELEMS(htmlchars));
```

对这种表格的二分检索函数可以写成下面的样子：

```
/* lookup: binary search for name in tab; return index */
int lookup(char *name, Nameval tab[], int ntab)
{
```

China-pub.com 下载


```
    int low, high, mid, cmp;

    low = 0;
    high = ntab - 1;
    while (low <= high) {
        mid = (low + high) / 2;
        cmp = strcmp(name, tab[mid].name);
        if (cmp < 0)
            high = mid - 1;
        else if (cmp > 0)
            low = mid + 1;
        else    /* found match */
            return mid;
    }
    return -1;  /* no match */
}
```

把这些放在一起，检索htmlchars的操作应写为：

```
half = lookup("frac12", htmlchars, NELEMS(htmlchars));
```

这里要查找的是字符数组下标的½。

二分检索在每个工作步骤中丢掉一半数据。这样，完成检索需要做的步数相当于对表的长度n反复除以2，直至最后剩下一个元素时所做的除法次数。忽略舍入后得到的是 $\log_2 n$。如果被检索的数据有1000个，采用线性检索就要做1000步，而做二分检索大约只要10步。如果被检索的数据项有1百万个，二分检索只需要做20步。可见，项目越多，二分检索的优势也就越明显。超过某个界限后(这个界限因实现不同可能有差别)二分检索一定会比线性检索更快。

2.2 排序

二分检索只能用在元素已经排好序的数组上。如果需要对某个数据集反复进行检索，先把它排序，然后再用二分检索就会是很值得的。如果数据集事先已经知道，写程序时就可以直接将数据排好序，利用编译时的初始化构建数组。如果事先不知道被处理的数据，那么就必须在程序运行中做排序。

最好的排序算法之一是快速排序(quicksort)，这个算法是1960年由C. A. R. Hoare发明的。快速排序是尽量避免额外计算的一个极好例子，其工作方式就是在数组中划分出小的和大的元素：

从数组中取出一个元素(基准值)。

把其他元素分为两组：

“小的”是那些小于基准值的元素；

“大的”是那些大于基准值的元素。

递归地对这两个组做排序。

当这个过程结束时，整个数组已经有序了。快速排序非常快，原因是：一旦知道了某个元素比基准值小，它就不必再与那些大的元素进行比较了。同样，大的元素也不必再与小的做比较。这个性质使快速排序远比简单排序算法(如插入排序和起泡排序)快得多。因为在简单排序算法中，每个元素都需要与所有其他元素进行比较。

快速排序是一种实用而且高效的算法。人们对它做了深入研究，提出了许多变形。在这里要展示的大概是其中最简单的一种实现，但它肯定不是最快的。

下面的quicksort函数做整数数组的排序：

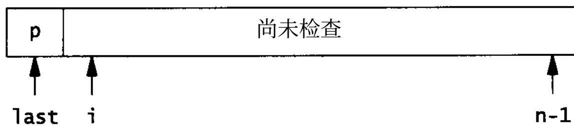
China-pub.com 下载

```
/* quicksort: sort v[0]..v[n-1] into increasing order */
void quicksort(int v[], int n)
{
    int i, last;

    if (n <= 1)  /* nothing to do */
        return;
    swap(v, 0, rand() % n);    /* move pivot elem to v[0] */
    last = 0;
    for (i = 1; i < n; i++)    /* partition */
        if (v[i] < v[0])
            swap(v, ++last, i);
    swap(v, 0, last);          /* restore pivot */
    quicksort(v, last);        /* recursively sort */
    quicksort(v+last+1, n-last-1); /*   each part */
}
```

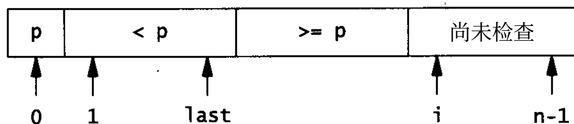
函数里的swap操作交换两个元素的值。swap在quicksort里出现三次，所以最好定义为一个单独的函数：

```
/* swap:  interchange v[i] and v[j] */
void swap(int v[], int i, int j)
{
    int temp;

    temp = v[i];
    v[i] = v[j];
    v[j] = temp;
}
```

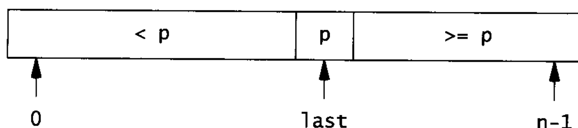
划分动作需要选一个随机元素作为基准值，并将它临时性地交换到最前边。此后，程序扫描其他元素，把小于基准值的元素 (小的)向前面交换(到位置last)，大的元素向后面交换(到位置i)。在这个过程开始时，基准值被放到数组的最前端，last=0且从i=1到n-1的元素都还没有检查，这时的状态是：



在for循环体的开始处，从1到last的元素严格地小于基准值，从last+1到i-1的元素大于或等于基准值，而从i到n-1的元素至今还没有检查过。当v[i]>=v[0]时，算法有可能交换v[i]和它自己，这会浪费了一点时间，但是没什么大影响。



在所有元素都划分完毕后，接着交换位置 0的元素与last处的元素，把基准值放到它的最终位置，这样就维护了正确的顺序。现在数组的样子变成了：



China-pub.com
下载

再把同样的过程用到左边和右边的部分数组上。当这一切都结束时，整个数组的排序就完成了。

快速排序有多快？在最好的情况下：

- 第一遍划分把n个元素分成各有$n/2$个元素的两组。
- 第二层把两个组，每个大约$n/2$个元素，划分为每组大约$n/4$个元素的4组。
- 下一层把有大约$n/4$个元素的4组划分为每组大约$n/8$个元素的8组。
- 如此下去。

这个过程将经历大约$\log_2 n$层。也就是说，在最好的情况下，快速排序算法的总工作量正比于$n+2\times n/2+4\times n/4+8\times n/8\cdots(\log_2 n$项)，也就是$n\log_2 n$。在平均情况下，它花的时间将稍微多一点。计算机领域的人都习惯于使用以2为底的对数，因此我们说排序算法花费的时间正比于$n\log n$。

快速排序的这个实现看起来很清楚。但是它也有另一面，有一个致命的弱点。如果每次对基准值的选择都能将元素划分为数目差不多相等的两组，上面的分析就是正确的。但是，如果执行中经常出现不平均的划分，算法的运行时间就可能接近于按n^2增长。我们在实现中采用随机选取元素作为基准值的方法，减少不正常的输入数据造成过多不平均划分的情况[⊖]。但如果数组里所有的值都一样，这个实现每次都只能切下一个元素，这就会使算法的运行时间达到与n^2成比例。

有些算法的行为对输入数据有很强的依赖性。如果遇到反常的、不好的输入数据，一个平常工作得很好的算法就可能运行极长时间，或者耗费极多存储。就快速排序而言，虽然上面的简单实现有时候确实可能运行得很慢，但采用另外一些更复杂的实现方式，就能把这种病态行为减小到几乎等于0。

2.3 库

C和C++ 的标准库里已经包含了排序函数。它应该能够对付恶劣的输入数据，并运行得尽可能快。

这里的库函数能用于对任何数据类型的排序工作，但是，与此同时，我们写出的程序也必须适应它的界面，这个界面比上面定义的函数要复杂很多。 C函数库中排序函数的名字是qsort，在调用qsort时必须为它提供一个比较函数，因为在排序中需要比较两个值。由于这里的值可以是任何类型的，比较函数的参数是两个指向被比较数据的 void*指针，在函数里应该把指针强制转换到适当类型，然后提取数据值加以比较，并返回结果（根据比较中的第一个值小于、等于或者大于第二个值，分别返回负数、零或者正数）。

首先考虑如何实现对字符串数组的排序，这是程序中经常需要做的。下面定义函数 scmp，它首先对两个参数做强制转换，然后调用 strcmp做比较。

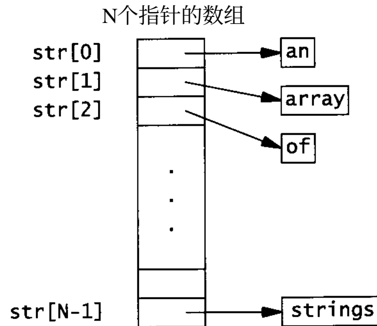
```
/* scmp: string compare of *p1 and *p2 */
int scmp(const void *p1, const void *p2)
{
    char *v1, *v2;

    v1 = *(char **) p1;
    v2 = *(char **) p2;
    return strcmp(v1, v2);
}
```

⊖ 这个说法是错误的，从概率上看，采用随机选取方法并不能减少不平均划分的可能性。——译者

China-pub.com 下载

我们也可以写一个一行的函数，增加临时变量是为了使代码更容易读。

我们不能直接用strcmp作为比较函数，因为qsort传递的是数组里元素的地址，也就是说，是&str[i](类型为char**)而不是str[i](类型为char*)，如下图所示：



如果需要对字符串数组的元素str[0]到str[N-1]排序，那么就应该以这个数组、数组的长度、被排序元素(数组元素)的大小以及比较函数作为参数，调用qsort：

```
char *str[N];

qsort(str, N, sizeof(str[0]), scmp);
```

下面是一个用于比较整数的类似函数icmp：

```
/* icmp: integer compare of *p1 and *p2 */
int icmp(const void *p1, const void *p2)
{
    int v1, v2;

    v1 = *(int *) p1;
    v2 = *(int *) p2;
    if (v1 < v2)
        return -1;
    else if (v1 == v2)
        return 0;
    else
        return 1;
}
```

我们或许想写：

```
?   return v1-v2;
```

但在这种情况下，如果v1是很大的正数而v2是大负数，或者相反，这个计算结果就可能溢出，并由此产生不正确的回答。采用直接比较写起来虽然长一点，但是却更安全。

调用qsort同样要用数组、数组长度、被排序元素的大小以及比较函数做为参数：

```
int arr[N];

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

ANSI C也定义了一个二分检索函数bsearch。与qsort类似，bsearch也要求一个指向比较函数的指针(常用与qsort同样的函数)。bsearch返回一个指针，指向检索到的那个元素；如果没找到有关元素，bsearch返回NULL。下面是用bsearch重写的HTML查询函数：

```
/* lookup: use bsearch to find name in tab, return index */
int lookup(char *name, Nameval tab[], int ntab)
{
```

China-pub.com
下载


```
    Nameval key, *np;

    key.name = name;
    key.value = 0;  /* unused; anything will do */
    np = (Nameval *) bsearch(&key, tab, ntab,
                         sizeof(tab[0]), nvcmp);
    if (np == NULL)
        return -1;
    else
        return np-tab;
}
```

与qsort的情况相同，比较函数得到的也是被比较项的地址，所以，这里的 key也必须具有项的类型。在这个例子里，我们必须先专门造出一个 Nameval项，将它传给比较函数。函数nvcmp比较两个Nameval项，方法是对它们的字符串部分调用函数 strcmp，值部分在这里完全不看。

```
/* nvcmp: compare two Nameval names */
int nvcmp(const void *va, const void *vb)
{
    const Nameval *a, *b;

    a = (Nameval *) va;
    b = (Nameval *) vb;
    return strcmp(a->name, b->name);
}
```

这个函数与scmp类似，不同点是这里的字符串是结构的成员。

为bsearch提供一个key就这么费劲，这实际上意味着它的杠杆作用比不上 qsort。写一个好的通用排序程序大概需要一两页代码，而二分检索程序的长度并不比为 bsearch做接口所需要的代码长多少。即使这样，使用 bsearch而不是自己另外写仍然是个好主意。多年的历史证明，程序员能把二分检索程序写正确也是很不容易的。

标准C++库里有一个名字为sort的类属算法，它保证 $O(n\log n)$的执行性质。使用它的代码非常简单，因为这个函数不需要强制转换，也不需要知道元素的大小。对于已知排序关系的类型，它甚至也不要求显式的比较函数。

```
int arr[N];

sort(arr, arr+N);
```

C++库里也有一个类属的二分检索函数，使用情况也类似。

练习2-1 快速排序用递归方式写起来非常自然。请你用循环把它写出来，并比较这两个版本(Hoare 描述了用循环写快速排序是如何困难，进而发现用递归做快速排序实在是太方便了)。

2.4 一个 Java 快速排序

Java语言的情况有所不同，其早期的发布中没有标准的排序函数，所以在那里就必须自己写。Java更新的版本已经提供了一个 sort函数，但是它只能对实现 Comparable界面的类使用。但不管怎么说，我们可以用库函数做排序了。由于这里牵涉到的技术在其他地方也很有用，因此，在本节里我们将仔细研究在 Java里实现快速排序的细节。

很容易对quicksort函数做一些改造，使它能处理我们需要排序的类型，但是，更有意义的是写一个类属的排序函数，它具有类似于前面 qsort的界面，而且可以用于任何种类的

China-pub.com 下载

对象。

Java与C、C++ 有一个重要差别：在这里不能把比较函数传递给另一个函数，因为这里没有函数指针。作为其替代物，在这里需要建立一个界面，其中仅有的内容就是一个函数，它完成两个Object的比较。对每个需要排序的数据类型，我们也需要建立一个类，其内容只包括一个成员函数，实现针对这个数据类型的界面。我们将把这个类的一个实例传递给排序函数，排序函数里用这个类中定义的比较函数做元素比较。

我们从定义这个界面开始，将它命名为Cmp，它的惟一成员就是比较函数cmp，这个函数做两个Object的比较：

```
interface Cmp {
    int cmp(Object x, Object y);
}
```

此后就可以写实现这个界面的各种具体比较函数了。例如，下面的类里定义的是对 Integer类型进行比较的函数：

```
// Icmp: Integer comparison
class Icmp implements Cmp {
    public int cmp(Object o1, Object o2)
    {
        int i1 = ((Integer) o1).intValue();
        int i2 = ((Integer) o2).intValue();
        if (i1 < i2)
            return -1;
        else if (i1 == i2)
            return 0;
        else
            return 1;
    }
}
```

下面的类定义了字符串的比较：

```
// Scmp: String comparison
class Scmp implements Cmp {
    public int cmp(Object o1, Object o2)
    {
        String s1 = (String) o1;
        String s2 = (String) o2;
        return s1.compareTo(s2);
    }
}
```

这种方法有一个限制，只能对所有由Object导出的并带有上面这种机制的类型做排序工作，因此我们定义的排序函数将无法用到各种基本类型，如int或double上。这就是为什么我们选择对Integer做排序，而不对int做的原因。

有了上面这些东西之后，现在可以把C语言的quicksort翻译到Java了，这里需要在作为参数的Cmp对象中调用比较函数。另一个最重要的改造是改用下标 left和right，因为Java不允许有指向数组的指针。

```
// Quicksort.sort: quicksort v[left]..v[right]
static void sort(Object[] v, int left, int right, Cmp cmp)
{
    int i, last;
```

China-pub.com
下载


```
        if (left >= right)  // nothing to do
            return;
        swap(v, left, rand(left,right));  // move pivot elem
        last = left;                      //    to v[left]
        for (i = left+1; i <= right; i++) // partition
            if (cmp.cmp(v[i], v[left]) < 0)
                swap(v, ++last, i);
        swap(v, left, last);              // restore pivot elem
        sort(v, left, last-1, cmp);       // recursively sort
        sort(v, last+1, right, cmp);      //    each part
    }
```

Quicksort.sort用cmp比较两个对象，像前面一样调用swap交换对象的位置。

```
    // Quicksort.swap: swap v[i] and v[j]
    static void swap(Object[] v, int i, int j)
    {
        Object temp;

        temp = v[i];
        v[i] = v[j];
        v[j] = temp;
    }
```

随机数由一个函数生成，它产生范围在left到right之间(包括两端)的随机数：

```
    static Random rgen = new Random();

    // Quicksort.rand: return random integer in [left,right]
    static int rand(int left, int right)
    {
        return left + Math.abs(rgen.nextInt())%(right-left+1);
    }
```

这里需要用Math.abs计算绝对值，因为Java随机数生成器的返回值可正可负。

上面这些函数，包括sort、swap和rand，以及生成器对象rgen都是类Quicksort的成员。

最后，如果要调用Quicksort.sort对一个String数组做排序，就应该写：

```
    String[] sarr = new String[n];

    // fill n elements of sarr...

    Quicksort.sort(sarr, 0, sarr.length-1, new Scmp());
```

这里用一个当时建立起来的字符串比较对象调用sort。

练习2-2　我们的Java快速排序要做许多类型转换工作，把对象从原来类型（例如Integer)转换为Object，而后又转回。试验Quicksort.sort的另一个版本，它只对某个特定类型做排序，估计由于类型转换带来的性能下降情况。

2.5　大O 记法

我们常需要描述特定算法相对于 n(输入元素的个数)需要做的工作量。在一组未排序的数据中检索，所需的时间与 n成正比；如果是对排序数据用二分检索，花费的时间正比于 $\log n$。排序时间可能正比于n^2或者$n\log n$。

我们需要有一种方式，用它能把这种说法弄得更精确，同时又能排除掉其中的一些具体

China-pub.com 下载

细节，如CPU速度，编译系统(以及程序员)的质量等。我们希望能够比较算法的运行时间和空间要求，并使这种比较能与程序设计语言、编译系统、机器结构、处理器的速度及系统的负载等等复杂因素无关。

为了这个目的，人们提出了一种标准的记法，称为“大 O记法”。在这种描述中使用的基本参数是n，即问题实例的规模，把复杂性或运行时间表达为 n的函数⊖。这里的“O”表示量级(order)，比如说“二分检索是$O(\log n)$的”，也就是说它需要“通过 $\log n$量级的步骤去检索一个规模为n的数组”。记法$O(f(n))$表示当n增大时，运行时间至多将以正比于 $f(n)$的速度增长。$O(n^2)$和$O(n\log n)$都是具体的例子。这种渐进估计对算法的理论分析和大致比较是非常有价值的，但在实践中细节也可能造成差异。例如，一个低附加代价的 $O(n^2)$算法在n较小的情况下可能比一个高附加代价的 $O(n\log n)$算法运行得更快。当然，随着 n足够大以后，具有较慢上升函数的算法必然工作得更快。

我们还应该区分算法的最坏情况的行为和期望行为。要定义好“期望”的意义非常困难，因为它实际上还依赖于对可能出现的输入有什么假定。在另一方面，我们通常能够比较精确地了解最坏情况，虽然有时它会造成误解。前面讲过，快速排序的最坏情况运行时间是 $O(n^2)$，但期望时间是$O(n\log n)$。通过每次都仔细地选择基准值，我们有可能把平方情况 (即$O(n^2)$情况)的概率减小到几乎等于0。在实际中，精心实现的快速排序一般都能以 ($O(n\log n)$时间运行。

下表是一些最重要的情况：

记　法	名　字	例　子
$O(1)$	常数	下标数组访问
$O(\log n)$	对数	二分检索
$O(n)$	线性	字符串比较
$O(n\log n)$	$n\log n$	快速排序
$O(n^2)$	平方	简单排序算法
$O(n^3)$	立方	矩阵乘法
$O(2^n)$	指数	集合划分

访问数组中的元素是常数时间操作，或说 $O(1)$操作。一个算法如果能在每个步骤去掉一半数据元素，如二分检索，通常它就取 $O(\log n)$时间。用`strcmp`比较两个具有n个字符的串需要$O(n)$时间。常规的矩阵乘算法是$O(n^3)$，因为算出每个元素都需要将n对元素相乘并加到一起，所有元素的个数是n^2。

指数时间算法通常来源于需要求出所有可能结果。例如，n个元素的集合共有 2^n个子集，所以要求出所有子集的算法将是 $O(2^n)$的。指数算法一般说来是太昂贵了，除非 n的值非常小，因为，在这里问题中增加一个元素就导致运行时间加倍。不幸的是，确实有许多问题 (如著名的“巡回售货员问题”)，到目前为止找到的算法都是指数的。如果我们真的遇到这种情况，通常应该用寻找近似最佳结果的算法替代之。

练习2-3　什么样的输入序列可能导致快速排序算法产生最坏情况的行为？设法找出一些情况，使你所用的库提供的排序函数运行得非常慢。设法将这个过程自动化，以便你能很容易描述并完成大量的试验。

练习2-4　设计和实现一个算法，它尽可能慢地对 n个整数的数组做排序。你的算法必须合乎

⊖ 这里的复杂性应该看作是算法的时间或空间开销的一种抽象，并不就是运行时间本身。——译者

China-pub.com
下载

情理，也就是说：算法必须不断前进并能最终结束，实现中不能有欺诈，如浪费时间的循环等。算法的复杂性(作为n的函数)是什么？

2.6 可增长数组

前面几节使用的数组都是静态的，它们的大小和内容都在编译时确定了。如果前面用的废话词表或者HTML字符表需要在程序运行中改变，散列表可能就是更合适的结构。向一个 n 元排序数组中逐个插入 n 个元素，使其逐渐增大，这是个 $O(2^n)$的操作。当n比较大时这种操作应该尽量避免。

我们常常需要记录一个包含某些东西的变量的变化情况，在这里数组仍然是一种可能选择。为减小重新分配的代价，数组应当以成块方式重新确定大小。为了清楚起见，维护数组所需要的信息必须与数组放在一起。在C++和Java的标准库里可以找到具有这种性质的类。在C里我们可以通过`struct`实现类似的东西。

下面的代码定义了一个元素为`Nameval`类型的可增长数组，新元素将被加到有关数组的最后。在必要时数组将自动增大以提供新的空间。任何元素都可以通过下标在常数时间里访问。这种东西类似于Java和C++库中的向量类。

```
typedef struct Nameval Nameval;
struct Nameval {
    char    *name;
    int     value;
};

struct NVtab {
    int     nval;          /* current number of values */
    int     max;           /* allocated number of values */
    Nameval *nameval;      /* array of name-value pairs */
} nvtab;

enum { NVINIT = 1, NVGROW = 2 };

/* addname: add new name and value to nvtab */
int addname(Nameval newname)
{
    Nameval *nvp;

    if (nvtab.nameval == NULL) { /* first time */
        nvtab.nameval =
            (Nameval *) malloc(NVINIT * sizeof(Nameval));
        if (nvtab.nameval == NULL)
            return -1;
        nvtab.max = NVINIT;
        nvtab.nval = 0;
    } else if (nvtab.nval >= nvtab.max) { /* grow */
        nvp = (Nameval *) realloc(nvtab.nameval,
                (NVGROW*nvtab.max) * sizeof(Nameval));
        if (nvp == NULL)
            return -1;
        nvtab.max *= NVGROW;
        nvtab.nameval = nvp;
    }
    nvtab.nameval[nvtab.nval] = newname;
    return nvtab.nval++;
}
```

China-pub.com 下载

函数addname返回刚加入数组的项的下标，出错的时候返回 - 1。

对realloc调用将把数组增长到一个新的规模，并保持已有的元素不变。这个函数返回指向新数组的指针；当存储不够时返回 NULL。这里采用每次调用 realloc时将数组规模加倍的方式，是为了保证复制数组元素的期望代价仍然是常数。如果数组在调用 realloc时一次增长一个元素，那么执行代价就会变成 $O(2n^2)$。由于在重新分配后数组的位置可能改变，程序其他部分对数组元素的访问必须通过下标进行，而不能通过指针。注意，在上面的代码中没采用下面这种方式：

```
?    nvtab.nameval = (Nameval *) realloc(nvtab.nameval,
?            (NVGROW*nvtab.max) * sizeof(Nameval));
```

如果采用这种形式，当重新分配失败时原来的数组就会丢失。

我们开始用一个非常小的初值(NVINIT=1)确定数组规模。这迫使程序一定要增长其数组，保证这段程序能够被执行。如果这段代码放在产品里使用，初始值可以改得大一些。当然，由小的初始值引起的代价也是微不足道的。

按说realloc的返回值不必强制到最后类型，因为 C能自动完成对 void*的提升。而C++ 中就不同，在那里必须做类型强制。有人在这里会争辩，究竟是应该做强制 (这样做清晰而认真)还是不做强制(因为强制实际上可能掩盖真正的错误)。我们选择写强制，是因为这种写法能同时适用于C和C++，付出的代价是减少了C编译器检出错误的可能性。通过允许使用两种编译器，我们也得到了一点补偿。

删除名字需要一点诀窍，因为必须决定怎样填补删除后数组中留下的空隙。如果元素的顺序并不重要，最简单的方法就是把位于数组的最后元素复制到这里。如果还必须保持原有顺序，我们就只能把空洞后面的所有元素前移一个位置：

```
/* delname: remove first matching nameval from nvtab */
int delname(char *name)
{
    int i;

    for (i = 0; i < nvtab.nval; i++)
        if (strcmp(nvtab.nameval[i].name, name) == 0) {
            memmove(nvtab.nameval+i, nvtab.nameval+i+1,
                (nvtab.nval-(i+1)) * sizeof(Nameval));
            nvtab.nval--;
            return 1;
        }
    return 0;
}
```

这个程序里调用memmove，通过把元素向下移一个位置的方法将数组缩短。 memmove是标准库函数，用于复制任意大小的存储区块。

在ANSI C的标准库里定义了两个相关的函数： memcpy的速度快，但是如果源位置和目标位置重叠，它有可能覆盖掉存储区中的某些部分； memmove函数的速度可能慢些，但总能保证复制的正确完成。选择正确性而不是速度不应该是程序员的责任，对这种情况实际上应该只提供一个函数。姑且认为这里只有一个，并总是使用 memmove。

也可以用下面的循环代替程序里对memmove的调用：

```
int j;
for (j = i; j < nvtab.nval-1; j++)
    nvtab.nameval[j] = nvtab.nameval[j+1];
```

China-pub.com 下载

我们喜欢用memmove，因为这样可以避免人很容易犯的复制顺序错误。例如，如果这里要做的是插入而不是删除，那么循环的顺序就必须反过来，以避免覆盖掉数组元素。通过调用memmove完成工作就不必为这些事操心了。

也可以采用其他方法，不做数组元素移动。例如把被删除数组元素标记为未用的。随后再要插入元素时，首先检索是否存在无用位置，只有在找不到空位时才做数组增长。对于这里的例子，可以采用把名字域置为NULL的方式表示无用的情况。

数组是组织数据的最简单方式。所以大部分语言都提供了有效而方便的下标数组，字符串也用字符的数组表示。数组的使用非常简单，它提供对元素的 $O(1)$访问，又能很好地使用二分检索和快速排序，空间开销也比较小。对于固定规模的数据集合，数组甚至可以在编译时建构起来。所以，如果能保证数据不太多，数组就是最佳选择。但事情也有另一方面，在数组里维护一组不断变化的数据代价很高。所以，如果元素数量无法预计，或者可能会非常多，选择其他数据结构可能就更合适些。

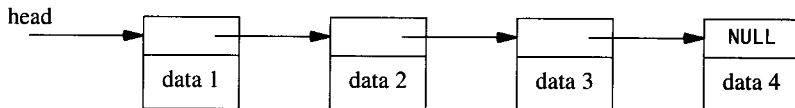
练习2-5 在上面的代码里，delname没有通过调用realloc去返回删除后释放的存储。那样做值得吗？你怎么决定是做这件事还是不做？

练习2-6 对addname和delname做适当修改，通过把删除项标记为未用的来删除元素。如何使程序的其余部分同这种修改隔离开来？

2.7 表

除了数组之外，表(链表)是典型程序里使用最多的数据结构。许多程序设计语言里有系统内部定义的表，有些语言——如Lisp——就完全是基于这种结构构造起来的。虽然在 C++和Java里表已经由程序库实现了，我们还是需要知道如何使用以及何时使用它。而在 C语言里我们就必须自己实现。这一节我们准备讨论 C的表，从中学到的东西可以用到更广泛的地方去。

一个**单链表**包含一组项，每个项都包含了有关数据和指向下一个项的指针。表的头就是一个指针，它指向第一个项，而表的结束则用空指针表示。下面是一个包含 4个元素的表：



数组和表之间有一些很重要的差别。首先，数组具有固定的大小，而表则永远具有恰好能容纳其所有内容的大小，在这里每个项目都需要一个指针的附加存储开销。第二，通过修改几个指针，表里的各种情况很容易重新进行安排，与数组里需要做大面积的元素移动相比，修改几个指针的代价要小得多。最后，当有某些项被插入或者删除时，其他的项都不必移动。如果把指向一些项的指针存入其他数据结构的元素中，表的修改也不会使这些指针变为非法的[⊖]。

这些情况说明，如果一个数据集合里的项经常变化，特别是如果项的数目无法预计时，表是一种可行的存储它们的方式。经过这些比较，我们容易看到，数组更适合存储相对静态的数据。

对表有一些基本操作：在表的最前面或最后加入一个新项，检索一个特定项，在某个指定项的前面或后面加入一个新项，可能还有删除一个项等等。表的简单性使我们很容易根据需要给它增加一些其他操作。

⊖ 这点不能一概而论，如果被指向的表项(表元素)随后被删除，有关指针就将变为非法的。——译者

China-pub.com 下载

在C里通常不是直接定义表的类型List，而是从某种元素类型开始，例如HTML的元素Nameval，给它加一个指针，以便能链接到下一个元素：

```
typedef struct Nameval Nameval;
struct Nameval {
    char    *name;
    int     value;
    Nameval *next;  /* in list */
};
```

我们无法在编译时初始化一个非空的表，这点也与数组不同。表应该完全是动态构造起来的。首先我们需要有方法来构造一个项。最直接的方式是定义适当的函数，完成有关的分配工作，这里称它为newitem：

```
/* newitem: create new item from name and value */
Nameval *newitem(char *name, int value)
{
    Nameval *newp;

    newp = (Nameval *) emalloc(sizeof(Nameval));
    newp->name = name;
    newp->value = value;
    newp->next = NULL;
    return newp;
}
```

这本书里的许多地方都要使用函数emalloc，这个函数将调用malloc，如果分配失败，它报告一个错误并结束程序的执行。函数的定义代码在第4章给出，现在只要认定emalloc是个存储分配函数，它绝不会以失败的情况返回。

构造表的最简单而又快速的方法就是把每个元素都加在表的最前面：

```
/* addfront: add newp to front of listp */
Nameval *addfront(Nameval *listp, Nameval *newp)
{
    newp->next = listp;
    return newp;
}
```

当一个表被修改时，结果可能得到另一个首元素，就像addfront里的情况。对表做更新操作的函数必须返回指向新首元素的指针，这个指针将被存入保持着这个表的变量里。函数addfront和这组函数里的其他函数都返回指向表中首元素的指针，以此作为它们的返回值。函数的典型使用方式是：

```
nvlist = addfront(nvlist, newitem("smiley", 0x263A));
```

这种设计对于原表为空的情况同样可以用，函数也可以方便地用在表达式里。另一种可能的方式是传递一个指针给保存表头的指针。我们这里的方法更自然些。

如果要在表末尾加一个元素，这就是一个$O(n)$操作，因为函数必须穿越整个表，直到找到了表的末端：

```
/* addend: add newp to end of listp */
Nameval *addend(Nameval *listp, Nameval *newp)
{
    Nameval *p;

    if (listp == NULL)
        return newp;
    for (p = listp; p->next != NULL; p = p->next)
        ;
```

China-pub.com
下载


```
        p->next = newp;
        return listp;
    }
```

如果想把addend做成一个$O(1)$的操作，那么就必须维持另一个独立的指向表尾的指针。除了总需要费心维护表尾指针外，这种做法还有另一个缺点：表再不是由一个指针变量表示的东西了。下面我们将坚持最简单的风格。

要检索具有某个特定名字的项，应该沿着next指针走下去：

```
/* lookup: sequential search for name in listp */
Nameval *lookup(Nameval *listp, char *name)
{
    for ( ; listp != NULL; listp = listp->next)
        if (strcmp(name, listp->name) == 0)
            return listp;
    return NULL;    /* no match */
}
```

这也需要$O(n)$时间，不存在能改进这个限度的一般性方法。即使一个表是排序的，我们也必须沿着表前进，以便找到特定的元素。二分检索完全不能适用于表。

如果要打印表里的所有元素，可以直接写一个函数，它穿越整个表并打印每个元素；要计算表的长度，可以写一个函数穿越整个表，其中使用一个计数器；如此等等。这里再提出另一种解决问题的方式，那就是写一个名为apply的函数，它穿越表并对表的每个元素调用另一个函数。我们可以把apply做得更具一般性，为它提供一个参数，把它传递给apply对表元素调用的那个函数。这样，apply就有了三个参数：一个表、一个将要被作用于表里每个元素的函数以及提供给该函数使用的一个参数：

```
/* apply: execute fn for each element of listp */
void apply(Nameval *listp,
        void (*fn)(Nameval*, void*), void *arg)
{
    for ( ; listp != NULL; listp = listp->next)
        (*fn)(listp, arg);  /* call the function */
}
```

apply的第二个参数是一个函数指针，这个函数有两个参数，返回类型是void。这种标准描述方式在语法上很难看：

```
void (*fn)(Nameval*, void*)
```

这说明了fn是一个指向void函数的指针。也就是说，fn本身是个变量，它将以一个返回值为void的函数的地址作为值。被fn指向的函数应该有两个参数，一个参数的类型是Nameval*，即表的元素类型；另一个是void*，是个通用指针，它将作为fn所指函数的一个参数。

要使用apply，例如打印一个表的元素，我们可以写一个简单的函数，其参数包括一个格式描述串：

```
/* printnv: print name and value using format in arg */
void printnv(Nameval *p, void *arg)
{
    char *fmt;

    fmt = (char *) arg;
    printf(fmt, p->name, p->value);
}
```

China-pub.com 下载

它的调用形式是：

```
apply(nvlist, printnv, "%s: %x\n");
```

为统计表中的元素个数也需要定义一个函数，其特殊参数是一个指向整数的指针，该整数被用作计数器：

```
/* inccounter: increment counter *arg */
void inccounter(Nameval *p, void *arg)
{
    int *ip;

    /* p is unused */
    ip = (int *) arg;
    (*ip)++;
}
```

对这个函数的调用可以是：

```
int n;

n = 0;
apply(nvlist, inccounter, &n);
printf("%d elements in nvlist\n", n);
```

并不是每个表操作都能很方便地以这种方式实现。要销毁一个表就必须特别小心：

```
/* freeall: free all elements of listp */
void freeall(Nameval *listp)
{
    Nameval *next;

    for ( ; listp != NULL; listp = next) {
        next = listp->next;
        /* assumes name is freed elsewhere */
        free(listp);
    }
}
```

当一块存储区被释放之后，程序里就不能再使用它了。因此，在释放 listp所指向的元素之前，必须首先把listp->next的值保存到一个局部变量(next)里。如果把上面的循环写成下面的样子：

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

由于listp->next原来的值完全可能被free复写掉，这个代码有可能会失败。

注意，函数freeall并没有释放listp->name，它实际上假定每个Nameval的name域会在程序中其他地方释放掉，或者它们根本就没有分配过。要保证在项的分配和释放方面的一致性，要求newitem和freeall之间有一种相互协调。既要保证所有存储都能被释放，又能弄清楚哪些东西没有释放并应该释放，在这两方面之间有一个平衡问题。如果不能正确处理就可能导致程序错误。在另一些语言里，例如在 Java里，有一个废料收集程序，它能帮人做这些事。在第4章讨论资源管理时还要回到这个问题。

如果想从表里删除一个元素，需要做的事情更多一些：

```
/* delitem: delete first "name" from listp */
Nameval *delitem(Nameval *listp, char *name)
{
    Nameval *p, *prev;
```

```
        prev = NULL;
        for (p = listp; p != NULL; p = p->next) {
            if (strcmp(name, p->name) == 0) {
                if (prev == NULL)
                    listp = p->next;
                else
                    prev->next = p->next;
                free(p);
                return listp;
            }
            prev = p;
        }
        eprintf("delitem: %s not in list", name);
        return NULL;    /* can't get here */
    }
```

与freeall里的处理方式一样，delitem也不释放name域。

函数eprintf显示一条错误信息并终止程序执行，这至多是一种笨办法。想从错误中得体地恢复程序执行是很困难的，需要一段很长的讨论。我们把这个讨论推迟到第4章，那里还要给出eprintf的实现。

上面是关于基本表结构和有关基本操作的讨论，这里也介绍了在写普通程序时可能遇到的许多重要应用。在这个方面也有许多可能的变化。有些程序库，例如C++的标准模板库(STL)支持双链表，其中的每个元素包含两个指针，一个指向其后继元素，另一个指向其前驱。双链表需要更多的存储开销，但它也使找最后元素和删除当前元素都变为 $O(1)$操作。有的实现方式为表指针另行分配位置，与被它们链接在一起的数据分开放。这样用起来稍微麻烦一点，但却能允许一个项同时隶属于多个不同的表。

表特别适合那些需要在中间插入和删除的情况，也适用于管理一批规模经常变动的无顺序数据，特别是当数据的访问方式接近后进先出(LIFO)的情况时(类似于栈的情况)。如果程序里存在多个互相独立地增长和收缩的栈，采用表比用数组能更有效地利用存储。当信息之间有一种内在顺序，就像一些事先不知道长短的链，例如文本中顺序的一系列单词，用表实现也非常合适。如果同时还必须对付频繁的更新和随机访问，那么最好就是使用某些非线性的数据结构，例如树或者散列表等。

练习2-7 实现其他一些表的操作，例如复制、归并、分裂以及在特定元素前面或者后面做插入等。两种不同插入操作在实现的困难性方面有什么差异？前面写的程序代码你可以用多少？哪些操作必须自己来做？

练习2-8 写一个递归的和一个非递归的reverse，它们能把一个表翻转过来。操作中不要建立新的表项，只用已经有的项。

练习2-9 在C里写一个类属的List类型。最简单的方法是令每个表项包含一个void*指针，它指向有关数据。在C++里利用模板，在Java里通过定义一个包含Object类型的表的方式重做这件事。对于这个工作，各种语言有哪些有利条件，又存在哪些弱点？

练习2-10 设计和实现一组测试，验证你写的各种表操作是正确的。第6章将讨论各种测试策略。

2.8 树

树是一种分层性数据结构。在一棵树里存储着一组项，每个项保存一个值，它可以有指

China-pub.com 下载

针指向0个或多个元素，但只能被另一个项所指。树根是其中惟一的例外，没有其他项的指针指向它。

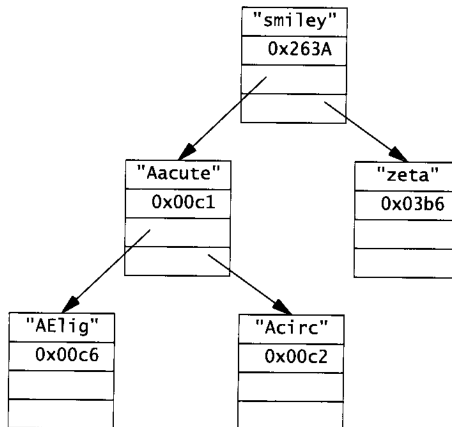
实际存在许多不同种类的树，它们表示各种复杂的结构。例如，语法树表示句子或程序的语法，家族树描述了人们互相间的关系。下面用二分检索树来说明树的原理，这种树最容易实现，也能很好地表现树的一般性质。二分检索树的每个结点带有两个子结点指针，left和right，它们分别指向对应的子结点。子结点指针可能取空值，当实际子结点少于两个时就会出现这种情况。在二分检索树里，结点中存储的值定义了树结构：对于一个特定结点，其左子树中存储着较小的值，而右子树里存储着较大的值。由于有这个性质，我们可以采用二分检索的一种变形在这种树里检索特定的值，或确定其存在性。

定义Nameval的树结点形式也很方便：

```
typedef struct Nameval Nameval;
struct Nameval {
    char     *name;
    int      value;
    Nameval *left;  /* lesser */
    Nameval *right; /* greater */
};
```

这里的lesser和greater注释指明了有关链接的性质：左子树里存储着较小的值，而右子树里存储较大的值。

作为一个实际例子，下面的图中显示的是一个Nameval二分检索树，其中存储着字符名字表的一部分，按照名字的ASCII编码排序。



由于树的许多结点包含多个指向其他元素的指针，所以，很多在表或者数组结构中需要$O(n)$时间的操作，在树中只需要$O(\log n)$时间。结点存在多个指针能减少为寻找一个结点所需要访问的结点个数，从而降低操作的复杂性。

二分检索树(在本节下面将直接称它为“树”)的建构方式是：在树里递归地向下，根据情况确定向左或向右，直到找到了链接新结点的正确位置。结点应该正确地初始化为Nameval类型的对象，它包含一个名字、一个值和两个空指针。新结点总是作为叶子加进去的，它没有子结点。

```
/* insert: insert newp in treep, return treep */
Nameval *insert(Nameval *treep, Nameval *newp)
{
```

China-pub.com
下载


```
    int cmp;

    if (treep == NULL)
        return newp;
    cmp = strcmp(newp->name, treep->name);
    if (cmp == 0)
        weprintf("insert: duplicate entry %s ignored",
            newp->name);
    else if (cmp < 0)
        treep->left = insert(treep->left, newp);
    else
        treep->right = insert(treep->right, newp);
    return treep;
}
```

至此我们一直没提重复项的问题。在上面这个insert函数里，对企图向树中加入重复项的情况(cmp==0)将输出一个“抱怨”信息。在表插入函数里没有这样做，因为如果要做这件事，就必须检索整个表，这就把插入由一个 $O(1)$操作变成了 $O(n)$操作。对树而言，这个测试完全不需要额外开销。但是另一方面，在出现重复时应该怎么办，数据结构本身并没有清楚的定义。对具体应用，有时可能应该接受重复情况，有时最合理的处理是完全忽略它。

函数weprintf是eprintf的一个变形，它打印错误信息，在输出信息的前面加上前缀词warning。这个函数并不终止程序执行，这一点与eprintf不同。

在一棵树里，如果从根到叶的每条路径长度大致都相同，这棵树被称为是平衡的㊀。在平衡树里做项检索是个 $O(\log n)$操作，因为不同可能性的数目在每一步减少一半，就像在二分检索中那样。

如果项是按照出现的顺序插入一棵树中，那么这棵树就可能成为不平衡的，实际上它完全可能变成极不平衡的。例如元素以排好序的方式出现，上面的代码就会总通过树的一个分支下降，这样产生出来的可能是一个由right链接起来的表。这样的树将与表结构具有同样的性能问题。如果元素以随机的顺序到达，那么就很难出现这种情况，产生出的树或多或少是平衡的。

要实现某种永远能够保证平衡的树是很复杂的，这也是为什么实际中存在很多不同种类的树的一个原因。对于我们的目的而言，还是先把这个问题放到一边，假定数据本身具有足够的随机性，生成的树是足够平衡的。

对树的lookup操作与insert很像：

```
/* lookup: look up name in tree treep */
Nameval *lookup(Nameval *treep, char *name)
{
    int cmp;

    if (treep == NULL)
        return NULL;
    cmp = strcmp(name, treep->name);
    if (cmp == 0)
        return treep;
    else if (cmp < 0)
```

㊀ 作者希望把“平衡树”概念弄得既不那么难懂，又具有一般性。但是这个说法实际上是错误的。对于只有一个分支的树，所有结点都在该分支上，只有一个叶结点。显然这个树满足作者的定义，但它绝不是一般数据结构意义下的平衡树。可见正文中的定义必须附加其他条件，或改用其他方式。这里提出平衡概念的目的，就是希望由树根到所有叶结点的最长距离与树全部结点个数成对数关系。——译者

China-pub.com 下载

```
        return lookup(treep->left, name);
    else
        return lookup(treep->right, name);
}
```

对于lookup和insert，还有几个情况应该注意。首先，这两个算法看起来非常像本章开始时给出的二分检索算法。这实际上并不奇怪，因为它们的基本思想与二分检索相同，都是“分而治之”，这是产生对数时间性能的根源。

第二，这些程序是递归的。如果将其改写成循环算法，它们就更像二分检索。实际上，从lookup的递归形式通过一个优雅的变换，就可以直接构造出它的循环形式。除了已经找到对应项的情况，lookup的最后一个动作是返回对它自己递归调用的结果，这种情况称为**尾递归**。尾递归可以直接转化为循环，方法是修补有关参数，并重新开始这个过程。最直接的方法是用一个goto语句，更清楚的方法是用一个while循环：

```
/* nrlookup: non-recursively look up name in tree treep */
Nameval *nrlookup(Nameval *treep, char *name)
{
    int cmp;

    while (treep != NULL) {
        cmp = strcmp(name, treep->name);
        if (cmp == 0)
            return treep;
        else if (cmp < 0)
            treep = treep->left;
        else
            treep = treep->right;
    }
    return NULL;
}
```

一旦知道如何在树上穿越，其他的常用操作都很自然了。我们可以采用管理表的某些技术，写一个一般的树遍历器，它对树的每个结点调用另一个函数。但是这次还需要做一些选择，确定什么时候对这个项进行操作，什么时候处理树的其余部分。相应的回答依赖于我们用树表示的是什么东西。如果这里表示的是排序的数据(例如二分排序树)，那么就该先访问左边一半，然后是右边。有时树的结构反映了数据的某种内在顺序(例如一棵家族树)，这时对树叶的访问顺序应该根据树所表示的关系确定。

在**中序**遍历中对数据项的操作在访问了左子树之后，并在访问右子树之前：

```
/* applyinorder: inorder application of fn to treep */
void applyinorder(Nameval *treep,
        void (*fn)(Nameval*, void*), void *arg)
{
    if (treep == NULL)
        return;
    applyinorder(treep->left, fn, arg);
    (*fn)(treep, arg);
    applyinorder(treep->right, fn, arg);
}
```

当结点需要按照排序方式顺序处理时，就应该使用这种方式。例如要按序打印树中数据，可以这样做：

```
applyinorder(treep, printnv, "%s: %x\n");
```

China-pub.com
下载

这实际上也提出了一种合理的排序方法：首先把数据项插入一棵树，接着分配一个正确大小的数组，然后用中序遍历方式顺序地把数据存入数组中。

后序遍历在访问了子树之后才对当前结点进行操作：

```
/* applypostorder: postorder application of fn to treep */
void applypostorder(Nameval *treep,
        void (*fn)(Nameval*, void*), void *arg)
{
    if (treep == NULL)
        return;
    applypostorder(treep->left, fn, arg);
    applypostorder(treep->right, fn, arg);
    (*fn)(treep, arg);
}
```

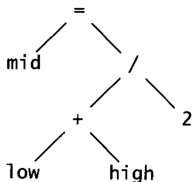
如果对各个结点的操作依赖于对它下面的子树的操作，那么就应该采用后序遍历。在所举的例子中，包含计算一棵树的高度 (取两棵子树的高度中大的一个，再加上 1)，用绘图程序包编排一棵树的图形(在页面上为每个子树安排空间，并将它们组合起来构成本结点的空间)，计算总的存储量，等等。

第三种选择称为**前序**，实际中很少使用，所以这里不讨论了。

在现实中，二分检索树的使用并不很多。另一种 B树有非常多的分支，它常被用在二级存储的数据组织中。在日常的程序设计里，树的常见用途之一是用于表示语句或表达式结构。例如，语句：

```
mid = (low + high) / 2;
```

可以表示为下面的语法分析树。要对这棵树做求值，只要对它做一次后序遍历，并在每个结点做适当的操作。



在第9章我们还有一大段关于语法分析树的讨论。

练习2-11　比较`lookup`和`nrlookup`的执行性能。递归与循环相比耗费高多少？

练习2-12　利用中序遍历方式建立一个排序过程。它的复杂性怎样？在什么条件下它的性能会很差？它的性能与快速排序或库函数比较起来又怎么样？

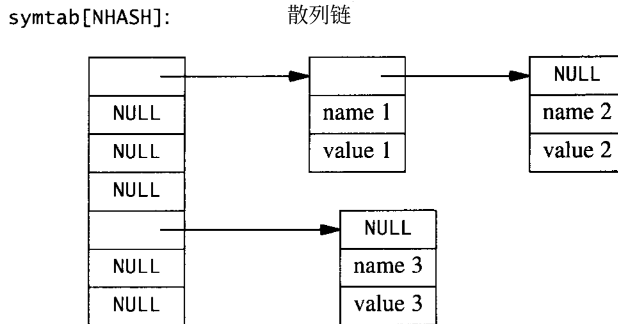
练习2-13　设计并实现一组测试，用它们验证各种树操作函数的正确性。

2.9　散列表

散列表是计算机科学里的一个伟大发明，它是由数组、表和一些数学方法相结合，构造起来的一种能够有效支持动态数据的存储和提取的结构。散列表的一个典型应用是**符号表**，在一些值(数据)与动态的字符串(关键码)集合的成员间建立一种关联。你最喜欢用的编译系统十之八九是使用了散列表，用于管理你的程序里各个变量的信息。你的网络浏览器可能也很好地使用了一个散列表来维持最近使用的页面踪迹。你与 Internet的连接可能也用到一个散列表，缓存最近使用的域名和它们的IP地址。

China-pub.com 下载

散列表的思想就是把关键码送给一个散列函数，产生出一个散列值，这种值平均分布在一个适当的整数区间中。散列值被用作存储信息的表的下标。Java提供了散列表的标准界面。在C和C++ 里，常见做法是为每一个散列值(或称“桶”)关联一个项的链表，这些项共有这同一个散列值，如下图所示：



在实践中，散列函数应该预先定义好，事先分配好一个适当大小的数组，这些通常在编译时完成。数组的每个元素是一个链表，链接起具有该散列值的所有数据项。换句话说，一个具有n个项的散列表是个数组，其元素是一组平均长度为 n/(数组大小) 的链表。这里提取一个项是$O(1)$操作，条件是选择了好的散列函数，而且表并不太长[⊖]。

由于散列表是链接表的数组，其基本元素类型与链接表相同：

```
typedef struct Nameval Nameval;
struct Nameval {
    char    *name;
    int     value;
    Nameval *next;      /* in chain */
};

Nameval *symtab[NHASH]; /* a symbol table */
```

在2.7节已经讨论过的链接表技术可以用于维护这里的表。一旦有了一个好的散列函数，随后的事就轻而易举了：直接取得散列桶(链接表)位置，然后穿越这个链表寻找正确的匹配。下面是为在散列表中做查询/插入的代码。如果找到了有关项目，函数将它返回。如果找不到，而且放置了create标志，lookup在表中加入一个新项目。与以前一样，这里也不为对应的名字建立新拷贝，假定调用的程序已经建立了安全的拷贝。

```
/* lookup: find name in symtab, with optional create */
Nameval* lookup(char *name, int create, int value)
{
    int h;
    Nameval *sym;

    h = hash(name);
    for (sym = symtab[h]; sym != NULL; sym = sym->next)
        if (strcmp(name, sym->name) == 0)
            return sym;
    if (create) {
        sym = (Nameval *) emalloc(sizeof(Nameval));
```

⊖ 文中说在这些条件下数据提取是一个 $O(1)$操作，在理论上看这种说法是不对的。但作者的本意是，在这些条件下操作效率很高，这个意思是很清楚的。——译者

China-pub.com
下载


```
        sym->name = name; /* assumed allocated elsewhere */
        sym->value = value;
        sym->next = symtab[h];
        symtab[h] = sym;
    }
    return sym;
}
```

把查询和可能的插入操作组合在一起，这也是很常见的情况。如果不这样做，常常会出现许多重复性工作。例如：

```
if (lookup("name") == NULL)
    additem(newitem("name", value));
```

在这里散列就计算了两次。

数组到底应该取多大？普遍的想法是要求它足够大，使每个链表至多只有几个元素，以保证查询能够是 $O(1)$操作。例如，一个编译程序用的数组可能有数千项，因为一个大的源文件可能有几千行，估计每行代码中不会有多于一个的新标识符。

我们现在必须考虑散列函数hash应该计算出什么东西。这个函数必须是确定性的，应该能算得很快，应该把数据均匀地散布到数组里。对于字符串，最常见的散列算法之一就是：逐个把字节加到已经构造的部分散列值的一个倍数上。乘法能把新字节在已有的值中散开来。这样，最后结果将是所有输入字节的一种彻底混合。根据经验，在对 ASCII串的散列函数中，选择31和37作为乘数是很好的。

```
enum { MULTIPLIER = 31 };

/* hash: compute hash value of string */
unsigned int hash(char *str)
{
    unsigned int h;
    unsigned char *p;

    h = 0;
    for (p = (unsigned char *) str; *p != '\0'; p++)
        h = MULTIPLIER * h + *p;
    return h % NHASH;
}
```

在这个计算中用到了无符号字符。这样做的原因是， C和C++对于char是不是有符号数据没有给出明确定义。而我们需要散列函数总返回正值。

散列函数的返回值已经根据数组的大小取模。如果散列函数能把关键码均匀地散开，那么有关数组到底有多大在这里就不必特别关心了。但是，实际上我们很难确定散列函数具有真正的独立性。进一步说，即使是最好的函数在遇到某些输入集合时也会有麻烦。考虑到这些情况，用素数作为数组的大小是比较明智的，因为这样能保证在数组大小、散列的乘数和可能的数据值之间不存在公因子。

经验表明，对于范围广泛的字符串，要想构造出一个真正能比上面给出的这个更好的散列函数，那是非常困难的。但是要给出一个比它差的就容易多了。 Java的一个早期版本里有一个字符串散列函数，它对长字符串的效率比较高。该函数节约时间的方法是：对那些长于16个字符的串，它只从开头起按照固定间隔检查 8个或9个字符。不幸的是，虽然这个散列函数稍微快一点，它的统计性质却比较差，完全抵消了性能上的优点。由于在散列中要跳过一

China-pub.com 下载

些片段，该函数常常忽略掉字符串中仅有的某些特征部分。实际上文件名字常有很长的共同前缀——目录名等，有的互相间只有最后几个字符不同（如.java和.class等）。URL的开头通常都是http://www.，结束都是.html，它们的差异只是在中间。如果一个散列函数经常只检查到名字里那些不变化的部分，结果就会造成很长的散列链，并使检索速度变慢。 Java的这个问题后来解决了，用的就是与我们上面给出的散列函数等价的函数（用乘数37），操作中还是检查字符串里的每个字符。

对一类输入集合(例如短的变量名字)工作得非常好的散列函数，也可能对另一类输入集合(例如URL)工作得很差。所以，对一个散列函数，应该用各种各样的典型输入集合做一些测试。例如，它对于短小的字符串散列得好不好？对于长的串如何？对于长度相同但有微小差别的串怎么样？

字符串并不是能被散列的惟一东西。我们也可以散列在物理模拟中表示粒子位置的三个坐标，以减少对存储需求，用一个线性的表（O(粒子数目))代替一个三维的数组（其大小是O(xsize×ysize×zsize)。

散列技术有一个很值得一提的例子，那就是Gerand Holzman为分析通信规程和并发系统而开发的Supertrace程序。Supertrace取得被分析系统每个可能状态的全部信息，将这种信息散列到存储器中单个二进制位的地址上。如果一个位已置位，那就表示对应状态已经检查过，否则就是没检查过。Supertrace使用一个几兆字节的散列表，而其中的每个桶只有一位长。这里没有链，如果两个状态发生冲突（它们具有相同的散列值），程序根本就是视而不见。Supertrace的工作依赖于冲突的概率很低(不必是0，因为Supertrace采用的是概率模型，而不是精确模型)。因此，这里用的散列函数必须特别小心。Supertrace采用一种循环冗余检查方式，函数从数据出发做了彻底的混合。

要实现符号表，采用散列表结构是最好的，因为它对元素访问提供了一个$O(1)$的期望性能。散列表也有一些缺点。如果散列函数不好，或者所用的数组太小，其中的链接表就可能变得很长。由于这些链接表没有排序，得到的将是$O(n)$操作。即使元素排了序也无法直接访问它们。但是这后一个问题比较容易对付：可以分配一个数组，在里面存储指向各个元素的指针，然后对它做排序。总之，散列表如果使用得当，常数时间的检索、插入和删除操作是任何其他技术都望尘莫及的。

练习2-14 我们的散列函数对字符串而言是个极好的散列函数，但是，某些特殊数据也可能使它表现欠佳。请设法构造出能使这个散列函数性能极差的数据集。对于NHASH的不同值，要找到这样的坏数据集合很容易吗？

练习2-15 写一个函数，以某种顺序访问散列表里所有的元素。

练习2-16 修改lookup函数，使得当链表的平均长度大于某个x值时，数组将按照y的某个因子扩大，并重新构造整个散列表。

练习2-17 设计一个散列函数，用它存储二维点的坐标。如果要改变坐标的类型，你的函数很容易修改吗？例如从整数坐标改为浮点数坐标，从笛卡儿坐标改为极坐标，或者从二维改到高维？

2.10 小结

选择算法有几个步骤。首先，应参考所有可能的算法和数据结构，考虑程序将要处理的

China-pub.com
下载

数据大概有多少。如果被处理数据的量不大，那么就选择最简单的技术。如果数据可能增长，请删掉那些不能对付大数据集合的东西。然后，如果有库或者语言本身的特征可以使用，就应该使用。如果没有，那么就写或者借用一个短的、简单的和容易理解的实现。如果实际测试说明它太慢，那么就需要改用某种更高级的技术。

虽然实际存在许多不同的数据结构，某些结构在一些特殊环境中对于达到较高性能是至关重要的，但是一般程序员主要使用的仍然是数组、链表、树和散列表。这些结构中的每一个都支持一组基本操作，通常包括建立新元素，寻找一个元素，在某处增加一个元素，或许还有删除一个元素，以及把某个操作作用于所有的元素，等等。

各种操作都具有一定的期望计算时间，这常常决定了一个数据结构(一个实现)是否适于某种特定应用。数组支持在常数时间里访问任何元素，但不能方便地增长或缩短。链表对于插入、删除可以很好地适应，但对它做随机元素访问要求 $O(n)$的时间。树与散列表提供了好的折衷，既支持对特定元素的快速访问，又支持方便的增长，条件是只要能维持好某种平衡性。

对于某些特殊问题，还存在一些更复杂的数据结构。但是，上面这个基本集合对构造绝大部分软件而言，已经完全够用了。

补充阅读

Bob Sedgewick的一套“算法”丛书(Algorithms，Addison-Wesley)是查找有用算法的绝好去处，其处理也很容易理解。《C++算法》(Algorithms in C++)的第3版有一段关于散列函数和表大小的讨论非常有意思。Don Knuth的《计算机程序设计技巧》(The Art of Computer Programming，Addison-Wesley)永远是对许多算法的严格分析的信息源，它的第3卷(第2版，1998)专门讨论排序和检索。

Gerard Holzman的《计算机规程的设计与验证》(Design and Validation of Computer Protocols，Prentice Hall，1991)里讨论了Supertrace的问题。

Jon Bentley和Doug McIlroy在文章“Engineering a sort function”(Software——Practice and Experience, **23**, 1, pp.1249~1265, 1993)里讨论了如何构造出一个高速的、强壮的快速排序程序。

China-pub.com
下载

第3章　设计与实现

给我看你的流程图而藏起你的表，我将仍然是莫名其妙。如果给我你的表，那么我将不再要你的流程图，因为它们太明显了。

—— Frederick P. Brooks, Jr.,《神秘的人月》

以上从Brooks的经典书中摘录的内容想说的是，数据结构设计是程序构造过程的中心环节。一旦数据结构安排好了，算法就像是瓜熟蒂落，编码也比较容易。

这种观点虽然有点过于简单化，但也不是在哄骗人。在前一章里我们考察了各种基本数据结构，它们是许多程序的基本构件。在这一章中，我们将组合这些结构，要完成的工作是设计和实现一个中等规模的程序。我们将说明被处理的问题将如何影响数据结构，从这里还可以看到，一旦数据结构安排好之后，代码将会如何自然地随之而来。

我们的观点的另一个方面是：程序设计语言的选择在整个设计过程中，相对而言，并不是那么重要。我们将抽象地设计这个程序，然后用 C、C++、Awk和Perl把它写出来。由不同实现之间的比较，可以看出语言在这里能有什么帮助或者妨碍，以及它们并不那么重要的各种情况。程序的设计当然可以通过语言来装饰，但是通常不会为语言所左右。

我们要选择的问题并不是很常见的，但它在基本形式上又是非常典型的：一些数据进去，另一些数据出来，其处理过程并不依赖于多少独创性。

我们准备做的就是产生一些随机的可以读的英语文本。如果随便扔出来一些随机字母或随机的词，结果当然是毫无意义的。例如，一个随机选取字母 (以及空格，用作词之间的分隔) 的程序可能产生：

```
xptmxgn xusaja  afqnzgxl  lhidlwcd rjdjuvpydrlwnjy
```

没人会觉得这有什么意思。如果以字母在英语里出现的频率作为它们的权重，我们可能得到下面这样的内容：

```
idtefoae tcs trder jcii ofdslnqetacp t ola
```

这个也好不到哪儿去。采用从字典里随机选择词的方式也弄不出多少意思来：

```
polydactyl equatorial splashily jowl verandah circumscribe
```

为了得到更好一些的结果，我们需要一个带有更多内在结构，例如包含着各短语出现频率的统计模型。但是，我们怎么才能得到这种统计呢？

我们当然可以抓来一大堆英语材料，仔细地研究。但是，实际上有一种更简单也更有意思的方法。这里有一个关键性的认识：用任何一个现成的某种语言的文本，可以构造出由这个文本中的语言使用情况而形成的统计模型。通过该模型生成的随机文本将具有与原文本类似的统计性质。

3.1　马尔可夫链算法

完成这种处理有一种非常漂亮的方法，那就是使用一种称为**马尔可夫链算法**的技术。我

China-pub.com 下载

们可以把输入想像成由一些互相重叠的短语构成的序列，而该算法把每个短语分割为两个部分：一部分是由多个词构成的前缀，另一部分是只包含一个词的后缀。马尔可夫链算法能够生成输出短语的序列，其方法是依据(在我们的情况下)原文本的统计性质，随机性地选择跟在前缀后面的特定后缀。采用三个词的短语就能够工作得很好——利用连续两个词构成的前缀来选择作为后缀的一个词：

设置w_1和w_2为文本的前两个词
输出w_1和w_2
循环：
　　随机地选出w_3，它是文本中w_1w_2的后缀中的一个
　　打印w_3
　　把w_1和w_2分别换成w_2和w_3
　　重复循环

为了说明问题，假设我们要基于本章开头的引语里的几个句子生成一些随机文本。这里采用的是两词前缀：

```
Show your flowcharts and conceal your tables and I will be
mystified.  Show your tables and your flowcharts will be
obvious.  (end)
```

下面是一些输入的词对和跟随它们之后的词：

输入前缀	跟随的后缀词
Show your	flowcharts tables
your flowcharts	and will
flowcharts and	conceal
flowcharts will	be
your tables	and and
will be	mystified. obvious.
be mystified.	Show
be obvious.	(*end*)

处理这个文本的马尔可夫算法将首先打印出Show your，然后随机取出flowcharts或table。如果选中了前者，那么现在前缀就变成your flowcharts，而下一个词应该是and或will。如果它选取tables，下一个词就应该是and。这样继续下去，直到产生出足够多的输出，或者在找后缀时遇到了结束标志。

我们的程序将读入一段英语文本，并利用马尔可夫链算法，基于文本中固定长度的短语的出现频率，产生一段新文本。前缀中词的数目是个参数，上面用的是2。如果将前缀缩短，产生出来的东西将趋向于无聊词语，更加缺乏内聚力；如果加长前缀，则趋向于产生原始输入的逐字拷贝。对于英语文本而言，用两个词的前缀选择第三个是一个很好的折衷方式。看起来它既能重现输入的风味，又能加上程序的古怪润饰。

什么是一个词？最明显的回答是字母表字符的一个序列。这里我们更愿意把标点符号也附着在词后，把“words”和“words.”看成是不同的词，这样做将有利于改进闲话生成的质量。加上标点符号，以及(间接的)语法将影响词的选择，虽然这种做法也可能会产生不配对的引语和括号。我们将把“词”定义为任何实际位于空格之间的内容，这对输入语言并没有造成任何限制，但却将标点符号附到了词上。许多语言里都有把文本分割成“空白界定的词”的功能，这个功能也很容易自己实现。

China-pub.com 下载

根据这里所采用的方法，输出中所有的词、所有的两词短语以及所有三个词的短语都必然是原来输入中出现过的，但是，也会有许多四个词或更多个词的短语将被组合产生出来。下面几个句子是由我们在本章里将要开发的程序生成的，给它提供的文本是艾尼思特·海明威的《太阳也升起来》的第4章：

> As I started up the undershirt onto his chest black, and big stomach muscles bulging under the light. "You see them?" Below the line where his ribs stopped were two raised white welts. "See on the forehead." "Oh, Brett, I love you." "Let's not talk. Talking's all bilge. I'm going away tomorrow." "Tomorrow?" "Yes. Didn't I say so? I am." "Let's have a drink, then."

我们很幸运，在这里标点符号没出问题。实际中却未必总能这样。

3.2 数据结构的选择

有多少输入需要处理？程序应该运行得多快？要求程序读完一整本书并不是不合理的，因此我们需要准备对付输入规模 n=100 000个词甚至更多的情况。输出将包括几百甚至几千个词，而程序的运行应该在若干秒内完成，而不是几分钟。假定输入文本有 100 000个词，n已经相当大了，因此，如果还要求程序运行得足够快，这个算法就不会太简单。

马尔可夫算法必须在看到了所有输入之后才能开始产生输出。所以它必须以某种形式存储整个输入。一个可能的方式是读完整个输入，将它存为一个长长的字符串。情况的另一方面也很明显，输入必须被分解成词。如果另用一个指向文本中各词的指针数组，输出的生成将很简单：产生一个词，扫描输入中的词，看看刚输出的前缀有哪些可能的后缀，然后随机选取一个。但是，这个方法意味着生成每个词都需要扫描 100 000个输入词。1000个输出就意味着上亿次字符串比较。这样做肯定快不了。

另一种可能性是存储单个的词，给每个词关联一个链表，指出该词在文本中的位置。这样就可以对词进行快速定位。在这里可以使用第 2章提出的散列表。但是，这种方式并没有直接触及到马尔可夫算法的需要。在这里最需要的是能够由前缀出发快速地确定对应的后缀。

我们需要有一种数据结构，它能较好地表示前缀以及与之相关联的后缀。程序将分两部分，第一部分是输入，它构造表示短语的整个数据结构；第二部分是随后的输出，它使用这个数据结构，生成随机的输出。这两部分都需要 (快速地)查询前缀：输入过程中需要更新与前缀相关的后缀；输出时需要对可能后缀做随机选择。这些分析提醒我们使用一种散列结构，其关键码是前缀，其值是对应于该前缀的所有可能后缀的集合。

为了描述的方便，我们将假定采用二词前缀，在这种情况下，每个输出词都是根据它前面的一对词得到的。前缀中词的个数对设计本身并没有影响，程序应该能对付任意的前缀长度，但给定一个数能使下面的讨论更具体些。我们把一个前缀和它所有可能后缀的集合放在一起，称其为一个状态，这是马尔可夫算法的标准术语。

对于一个特定前缀，我们需要存储所有能跟随它的后缀，以便将来取用。这些后缀是无序的，一次一个地加进去。我们不知道后缀将会有多少，因此，需要一种能容易且高效地增长的数据结构，例如链表或者动态数组。在产生输出的时候，我们要能从关联于特定前缀的后缀集合中随机地选出一个后缀。还有，数据项绝不会被删除。

如果一个短语出现多次，那么又该怎么办？例如，短语“ might appear twice ”可能在文

本里出现两次，而“ might appear once ”只出现了一次。这个情况有两种可能的表示方式：或者在“ might appear ”的后缀表里放两个“ twice ”；或者是只放一个，但还要给它附带一个计数值为2的计数器。我们对用或不用计数器的方式都做过试验。不用计数器的情况处理起来比较简单，因为在加入后缀时不必检查它是否已经存在。试验说明这两种方式在执行时间上的差别是微不足道的。

总结一下：每个状态由一个前缀和一个后缀链表组成。所有这些信息存在一个散列表里，以前缀作为关键码。每个前缀是一个固定大小的词集合。如果一个后缀在给定前缀下的出现多于一次，则每个出现都单独包含在有关链表里。

下面的问题是如何表示词本身。最简单的方法是把它们存储为独立的字符串。在一般文本里总是有许多反复出现的词，如果为单词另外建一个散列表有可能节约存储，因为在这种情况下每个词只需要存储一次。此外，这样做还能加快前缀散列的速度，因为这时每个词都只有一个惟一地址，可以直接比较指针而不是比较词里的各个字符。我们把这种做法留做练习。下面采用每个词都分开存放的方式。

3.3 在C中构造数据结构

现在开始考虑C语言中的实现。首先是定义一些常数：

```
enum {
    NPREF   = 2,     /* number of prefix words */
    NHASH   = 4093, /* size of state hash table array */
    MAXGEN  = 10000 /* maximum words generated */
};
```

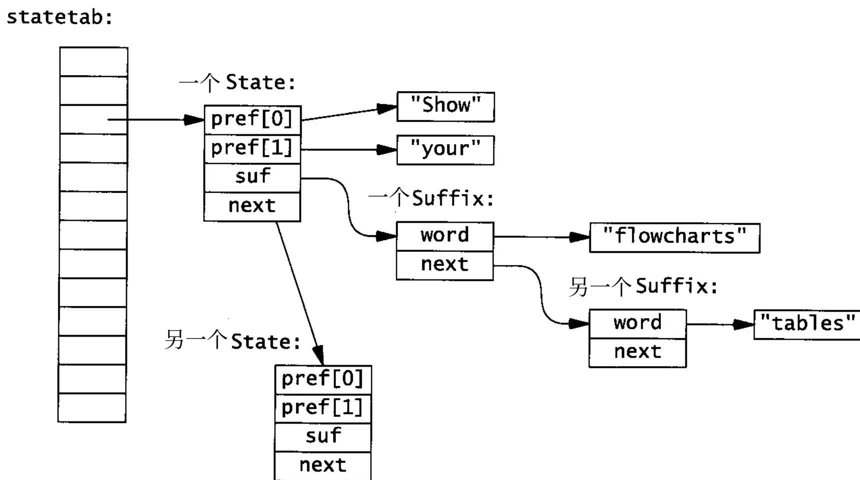
这个声明定义了前缀中词的个数 (NPREF)，散列表数组的大小 (NHASH)，生成词数的上界(MAXGEN)。如果NPREF是个编译时的常数而不是运行时的变量，程序里的存储管理将会更简单些。数组的规模设得相当大，因为我们预计程序可能处理很大的输入文件，或许是整本书。选择NHASH=4093，这样，即使输入里有10 000个不同前缀(词对)，平均链长仍然会很短，大约两个到三个前缀。数组越大，链的期望长度越短，查询进行得也越快。实际上，这个程序还仍然是个摆设，因此其性能并不那么关键。另一方面，如果选用的数组太小，程序将无法在合理时间里处理完可能的输入。而如果它太大，又可能无法放进计算机的存储器中。

前缀可以用词的数组的方式存储。散列表的元素用 State(状态)数据类型表示，它是前缀与Suffix(后缀)链表的关联：

```
typedef struct State State;
typedef struct Suffix Suffix;
struct State {  /* prefix + suffix list */
    char    *pref[NPREF];   /* prefix words */
    Suffix  *suf;           /* list of suffixes */
    State   *next;          /* next in hash table */
};

struct Suffix { /* list of suffixes */
    char    *word;          /* suffix */
    Suffix  *next;          /* next in list of suffixes */
};

State   *statetab[NHASH];   /* hash table of states */
```

现在看一看图示，整个数据结构将具有下面的样子：

China-pub.com 下载



我们需要一个作用于前缀的散列函数，前缀的形式是字符串数组，显然不难对第 2章的字符串散列函数做一点修改，使之可用于字符串的数组。下面的函数对数组里所有字符串的拼接做散列：

```
/* hash: compute hash value for array of NPREF strings */
unsigned int hash(char *s[NPREF])
{
    unsigned int h;
    unsigned char *p;
    int i;

    h = 0;
    for (i = 0; i < NPREF; i++)
        for (p = (unsigned char *) s[i]; *p != '\0'; p++)
            h = MULTIPLIER * h + *p;
    return h % NHASH;
}
```

再加上对检索函数的简单修改，散列表的实现就完成了：

```
/* lookup: search for prefix; create if requested. */
/*  returns pointer if present or created; NULL if not. */
/*  creation doesn't strdup so strings mustn't change later. */
State* lookup(char *prefix[NPREF], int create)
{
    int i, h;
    State *sp;

    h = hash(prefix);
    for (sp = statetab[h]; sp != NULL; sp = sp->next) {
        for (i = 0; i < NPREF; i++)
            if (strcmp(prefix[i], sp->pref[i]) != 0)
                break;
        if (i == NPREF)      /* found it */
            return sp;
    }
    if (create) {
        sp = (State *) emalloc(sizeof(State));
        for (i = 0; i < NPREF; i++)
            sp->pref[i] = prefix[i];
        sp->suf = NULL;
        sp->next = statetab[h];
```

China-pub.com
下载


```
        statetab[h] = sp;
    }
    return sp;
}
```

注意，lookup在建立新状态时并不做输入字符串的拷贝。它只是向 sp->pref[]里存入一个指针。这实际上要求调用lookup的程序必须保证这些数据不会被覆盖。例如，如果字符串原来存放在I/O缓冲区里，那么在调用lookup前必须先做一个拷贝。否则后面的输入就会把散列表指针所指的数据覆盖掉。对于跨越某个界面的共享资源，常常需要确定它的拥有者到底是谁。在下一章里有对这个问题的详尽讨论。

作为下一步，我们需要在读入文件的同时构造散列表：

```
/* build: read input, build prefix table */
void build(char *prefix[NPREF], FILE *f)
{
    char buf[100], fmt[10];

    /* create a format string; %s could overflow buf */
    sprintf(fmt, "%%%ds", sizeof(buf)-1);
    while (fscanf(f, fmt, buf) != EOF)
        add(prefix, estrdup(buf));
}
```

对sprintf的调用有些奇怪，这完全是为了避免 fscanf的一个非常烦人的问题，而从其他方面看，使用fscanf都是很合适的。如果以%s作为格式符调用fscanf，那就是要求把文件里的下一个由空白界定的词读入缓冲区。但是，假如在这种情况下没有长度限制，特别长的词就可能导致输入缓冲区溢出，从而酿成大祸。假设缓冲区的长度为 100个字节(这远远超出正常文本中可能出现的词的长度)，我们可以用%99s(留一个字节给串的结束符 '\0')，这是告诉fscanf读到99个字符就结束。这样做有可能把过长的词分成段，虽然是不幸的，但却是安全的。我们可以声明：

```
?   enum { BUFSIZE = 100 };
?   char    fmt[] = "%99s"; /* BUFSIZE-1 */
```

但是这里又出现了由一个带随意性的决定(缓冲区大小)导出的两个常数，并要求维护它们之间的关系。这个问题可以一下子解决：只要利用 sprintf动态地建立格式串，也就是上面程序里采用的方式。

函数build有两个参数，一个是prefix数组，用于保存前面的NPREF个输入词；另一个是个FILE指针。函数把prefix和读入词的一个拷贝送给 add，该函数在散列表里加入一个新项，并更新前缀数组：

```
/* add: add word to suffix list, update prefix */
void add(char *prefix[NPREF], char *suffix)
{
    State *sp;

    sp = lookup(prefix, 1);  /* create if not found */
    addsuffix(sp, suffix);
    /* move the words down the prefix */
    memmove(prefix, prefix+1, (NPREF-1)*sizeof(prefix[0]));
    prefix[NPREF-1] = suffix;
}
```

对memmove的调用是在数组里做删除的一个惯用法。该操作把前缀数组里从 1到NPREF-1的

China-pub.com 下载

元素向下搬，移到从0到NPREF-2的位置。这也就删去了第一个前缀词，并为新来的一个在后面腾出了位置。

函数addsuffix把一个新后缀加进去：

```
/* addsuffix: add to state. suffix must not change later */
void addsuffix(State *sp, char *suffix)
{
    Suffix *suf;

    suf = (Suffix *) emalloc(sizeof(Suffix));
    suf->word = suffix;
    suf->next = sp->suf;
    sp->suf = suf;
}
```

这里的状态更新操作分由两个函数实现：add完成给有关前缀加入一个后缀的一般性工作，addsuffix做的是由特定实现方式决定的动作，把一个词具体地加进后缀链表里。函数 add 由build调用，而addsuffix只在add内部使用，因为这里牵涉到的是一个实现细节，这个细节今后也可能会变化。所以，虽然该操作只在这一个地方用，最好也将它建立为一个独立的函数。

3.4 生成输出

数据结构构造好之后，下一步就是产生输出。这里的基本思想与前面类似：给定一个前缀，随机地选出它的某个后缀，打印输出并更新前缀。当然，这里说的是处理过程的稳定状态，还需要弄清算法应该如何开始和结束。如果我们已经记录了文中第一个前缀，操作就非常简单了：直接从它们起头。结束也容易。我们需要一个标志字来结束算法，在所有正常输入完成之后，我们可以加进一个结束符，一个保证不会在任何输入里出现的“词”：

```
build(prefix, stdin);
add(prefix, NONWORD);
```

NONWORD应该是某个不可能在正规输入里遇到的值。由于输入词是由空白界定的，一个空白的“词”总能扮演这个角色，比如用一个换行符号：

```
char NONWORD[] = "\n";  /* cannot appear as real word */
```

还有一件事也需要考虑：如果输入根本就不够启动程序，那么又该怎么办呢？处理这类问题有两种常见方式：或是在遇到输入不足时立即退出执行；或是通过安排使得输入总是足够的，从而就完全不必再理会这个问题了。对这里的程序而言，采用后一种方式能够做得很好。

我们可以用一个伪造的前缀来初始化数据结构构造和输出生成过程，这样就能保证程序的输入总是足够的。在做循环准备时，我们把前缀数组装满 NONWORD词。这样做有一个非常好的效果：输入文件里的第一个词将总是这个伪造前缀的第一个后缀[⊖]。这样，生成循环要打印的全都是它自己生成的后缀。

如果输出非常长，我们可以在产生了一定数目的词之后终止程序；另一种情况是程序遇到了后缀NONWORD。最终看哪个情况先出现。

在数据的最后加上几个 NONWORD，可以大大简化程序的主处理循环。这是一种常用技术的又一个实例：给数据加上哨卫，用以标记数据的界限。

⊖ 实际上也是惟一的一个。——译者

China-pub.com
下载

作为编程的一个规则，我们总应该设法处理数据中各种可能的非正常情况、意外情况和特殊情况。编出正确代码很不容易，因此应该尽量使控制流简单和规范。

函数generate采用的就是前面已经给出了轮廓的算法。它产生每行一个词的输出，用文字处理程序可以把它们汇成长的行。第 9章将给出一个能完成这个工作的简单格式化程序fmt。

借助于数据开始和结束的NONWORD串，generate也很容易开始和结束：

```
/* generate: produce output, one word per line */
void generate(int nwords)
{
	State *sp;
	Suffix *suf;
	char *prefix[NPREF], *w;
	int i, nmatch;

	for (i = 0; i < NPREF; i++) /* reset initial prefix */
		prefix[i] = NONWORD;

	for (i = 0; i < nwords; i++) {
		sp = lookup(prefix, 0);
		nmatch = 0;
		for (suf = sp->suf; suf != NULL; suf = suf->next)
			if (rand() % ++nmatch == 0) /* prob = 1/nmatch */
				w = suf->word;
		if (strcmp(w, NONWORD) == 0)
			break;
		printf("%s\n", w);
		memmove(prefix, prefix+1, (NPREF-1)*sizeof(prefix[0]));
		prefix[NPREF-1] = w;
	}
}
```

注意，算法需要在不知道到底有多少个项的情况下随机地选取一个项。变量 nmatch用于在扫描后缀表的过程中记录匹配的个数。表达式：

```
rand() % ++nmatch == 0
```

增加nmatch的值，而且使它为真的概率总是 1/nmatch。这样，第一个匹配的项被选中的概率为1，第二个将有1/2的概率取代它，第3个将以1/3的概率取代前面选择后留下的项，依此类推。在任何时刻，前面匹配的*k*个项中的每一个都有1/*k*的被选概率。

在算法开始时，prefix已经被设为初始值，可以保证散列表中一定有它。这样得到的第一个suffix值就是文件里的第一个词，因为它是跟随初始前缀的惟一后缀。在此之后，算法随机地选择后缀：循环中调用 lookup，查出当前prefix对应的散列表项，然后随机地选出一个对应后缀，打印它并更新前缀。

如果选出的后缀是NONWORD，则工作完成，因为已经选到了对应输入结束的状态。如果后缀不是NONWORD，则打印它，然后调用memmove丢掉前缀的第一个词，把选出的后缀升格为前缀的最后一个词，并继续循环下去。

现在，我们可以把所有东西放到一起，装进一个 main函数里，它从标准输入流读入，生成至多有指定个数的词序列：

```
/* markov main: markov-chain random text generation */
int main(void)
{
```

China-pub.com 下载

```
    int i, nwords = MAXGEN;
    char *prefix[NPREF];        /* current input prefix */
    for (i = 0; i < NPREF; i++) /* set up initial prefix */
        prefix[i] = NONWORD;
    build(prefix, stdin);
    add(prefix, NONWORD);
    generate(nwords);
    return 0;
}
```

这就完成了我们的C实现。本章最后对不同语言的实现做比较时我们还要回到这里。 C语言最强的地方就是给了程序员对实现方式的完全控制，用它写出的程序趋向于快速。但是这也有代价，这就是C程序员必须做更多的工作，包括分配和释放存储，建立散列表和链接表，以及其他许多类似的事项。C语言像一把飞快的剃刀，使用它，你可以创造优雅和高效的程序，或者弄出些一塌糊涂的东西来。

练习3-1 算法从未知长度的链表里随机地选择项，这依赖于一个好的随机数生成器。设计并执行一些试验，确定这个方法在实践中到底工作得怎么样。

练习3-2 如果把所有输入词都存入另一个散列表，每个词只保存一个，那么可以节约许多存储。对一些文本做测试，估计可以节省多少。采用这种组织方式，我们对前缀的散列链表可以用比较指针而不是比较字符串的方式，这应该运行得更快。请实现这种方式，估计速度和存储消耗的改变情况。

练习3-3 去掉在数据开头和结束设置的作为哨卫的`NONWORD`的语句，修改`generate`，使程序在没有它们的情况下还能正常开始和结束。请确认程序能对 0、1、2、3和4个词的输入正确工作。将这个实现方式和使用哨卫的实现方式做些比较。

3.5 Java

我们在Java里做第二个马尔可夫链算法的实现。 Java是面向对象的语言，这种语言将促使人们对程序里各组件之间的界面给予特别关注。在这里，具有独立性的数据项和与之相关的函数(称为方法)被封装一起，形成称为对象或类的程序部件。

Java的库比C语言丰富得多，其中包含了一组能以各种方式把已有数据汇集起来的容器类。`Vector`是容器类的一个例子，它提供一种动态增长的数组，可以存储任何`Object`类型的东西。另一个例子是`Hashtable`类，它允许以任何类型的对象作为关键码，存储或提取某种类型的值。

在这个应用里，字符串的`Vector`是保存前缀和后缀的自然选择。用一个散列表，以前缀向量作为关键码，后缀向量作为值。按照术语，这种结构是从前缀到后缀的一个映射。在Java语言里不需要显式定义的`State`类型，因为`Hashtable`结构能将前缀隐含地连接(映射)到后缀。这种设计与前面C版本的情况不同，那里建立了`State`结构，其中包含前缀和后缀链表，通过对前缀的散列计算得到的是一个完整的`State`。

`Hashtable`类提供了一个`put`方法，用于存储关键码和值的对；另外还提供了一个 `get`方法，通过它可以从关键码出发得到对应的值：

```
Hashtable h = new Hashtable();
h.put(key, value);
Sometype v = (Sometype) h.get(key);
```

China-pub.com
下载

我们的实现总共使用了三个类。第一个类Prefix保存前缀向量的词：

```
class Prefix {
    public Vector pref; // NPREF adjacent words from input
    ...
```

第二个类是Chain，它读取输入、构造散列表并产生输出。下面是它的类定义：

```
class Chain {
    static final int NPREF = 2; // size of prefix
    static final String NONWORD = "\n";
                    // "word" that can't appear
    Hashtable statetab = new Hashtable();
                    // key = Prefix, value = suffix Vector
    Prefix prefix = new Prefix(NPREF, NONWORD);
                    // initial prefix
    Random rand = new Random();
    ...
```

第三个类是公共界面，其中包含一个main函数和一个Chain实例：

```
class Markov {
    static final int MAXGEN = 10000; // maximum words generated
    public static void main(String[] args) throws IOException
    {
        Chain chain = new Chain();
        int nwords = MAXGEN;

        chain.build(System.in);
        chain.generate(nwords);
    }
}
```

Chain类实例在创建时构造起一个散列表，其中的前缀数组用NPREF个NONWORD设置了初始值。函数build利用库函数StreamTokenizer进行输入剖析，将输入分解为空白符分隔的一系列单词。进入循环前有三个函数调用，这是为了对完成单词分解的库函数做一些设置，使它的工作方式能符合我们关于“词”的定义。

```
// Chain build: build State table from input stream
void build(InputStream in) throws IOException
{
    StreamTokenizer st = new StreamTokenizer(in);

    st.resetSyntax();                        // remove default rules
    st.wordChars(0, Character.MAX_VALUE); // turn on all chars
    st.whitespaceChars(0, ' ');          // except up to blank
    while (st.nextToken() != st.TT_EOF)
        add(st.sval);
    add(NONWORD);
}
```

函数add根据给定的前缀参数从散列表里提取对应的后缀向量，找不到(向量为空)时就创建一个新向量，并把这个新向量和新前缀一起存入散列表。在任何情况下，它都向后缀向量里加入一个新词，然后更新前缀，丢掉第一个词并在最后加一个新词。

```
// Chain add: add word to suffix list, update prefix
void add(String word)
{
    Vector suf = (Vector) statetab.get(prefix);
    if (suf == null) {
        suf = new Vector();
```

China-pub.com 下载

```
            statetab.put(new Prefix(prefix), suf);
        }
        suf.addElement(word);
        prefix.pref.removeElementAt(0);
        prefix.pref.addElement(word);
    }
```

请注意，如果suf为空，add将在散列表里建立一个新的Prefix，而不是直接把prefix本身存进去。这里必须采用这种做法，因为Hashtable类是以引用方式存储数据项的，如果不明确地建立新拷贝，随后的操作就会把已经存入表里的数据覆盖掉。在前面写C程序时，我们也特别注意到对这个问题的处理。

生成函数与C程序里对应的函数差不多，只是稍微紧凑一些。在这里可以通过下标随机地直接访问向量元素，不必写一个循环去扫描整个链表。

```
    // Chain generate: generate output words
    void generate(int nwords)
    {
        prefix = new Prefix(NPREF, NONWORD);
        for (int i = 0; i < nwords; i++) {
            Vector s = (Vector) statetab.get(prefix);
            int r = Math.abs(rand.nextInt()) % s.size();
            String suf = (String) s.elementAt(r);
            if (suf.equals(NONWORD))
                break;
            System.out.println(suf);
            prefix.pref.removeElementAt(0);
            prefix.pref.addElement(suf);
        }
    }
```

Prefix有两个建构函数，它们根据由参数提供的数据创建新实例。第一个建构函数复制已有的Prefix，第二个函数建立一个新前缀，其中存放某字符串的 n个拷贝。在程序初始化时，我们要用第二个建构函数建立包含NPREF个NONWORD的前缀：

```
    // Prefix constructor: duplicate existing prefix
    Prefix(Prefix p)
    {
        pref = (Vector) p.pref.clone();
    }

    // Prefix constructor: n copies of str
    Prefix(int n, String str)
    {
        pref = new Vector();
        for (int i = 0; i < n; i++)
            pref.addElement(str);
    }
```

Prefix类里也有两个方法，hashCode和equals，它们在Hashtable的实现中隐含地被调用，产生下标和对表进行检索。Hashtable类要求必须为这两个方法专门建立一个类，这迫使我们把Prefix建成一个完整的新类，而不是像后缀那样只建立一个 Vector。

hashCode对向量各元素使用hashCode方法，将得到的值组合成一个散列值：

```
    static final int MULTIPLIER = 31;   // for hashCode()

    // Prefix hashCode: generate hash from all prefix words
    public int hashCode()
```

China-pub.com
下载


```
    {
        int h = 0;
        for (int i = 0; i < pref.size(); i++)
            h = MULTIPLIER * h + pref.elementAt(i).hashCode();
        return h;
    }
```

函数equals对两个前缀做比较，采用逐个比较元素的方式。

```
    // Prefix equals: compare two prefixes for equal words
    public boolean equals(Object o)
    {
        Prefix p = (Prefix) o;
        for (int i = 0; i < pref.size(); i++)
            if (!pref.elementAt(i).equals(p.pref.elementAt(i)))
                return false;
        return true;
    }
```

这个Java程序比前面的C程序小了不少，并照顾到更多的细节，Vector和Hashtable是其中最明显的例子。一般地说，这里的存储管理比较简单，因为向量本身能够自动增长，废料收集将管理回收那些不再引用的存储。但是，为了能使用 Hashtable类，我们还必须自己写函数hashCode和equals。可见Java并没有照顾好一切细节。

对C和Java程序里针对相同基本数据结构的表示和操作做一个比较，可以看到，在 Java程序里的功能划分做得更好。例如，在这里想把 Vector换成数组是非常简单的。在C程序里就不同，每个东西都知道其他东西在做什么。例如：散列表在数组上进行操作，因此，在许多地方它都要做数组的维护工作；lookup知道State和Suffix的内部结构；任何东西都知道前缀数组的大小。

```
% java Markov <jr_chemistry.txt | fmt
Wash the blackboard.  Watch it dry.  The water goes
into the air.  When water goes into the air it
evaporates.  Tie a damp cloth to one end of a solid or
liquid.  Look around.  What are the solid things?
Chemical changes take place when something burns.  If
the burning material has liquids, they are stable and
the sponge rise.  It looked like dough, but it is
burning.  Break up the lump of sugar into small pieces
and put them together again in the bottom of a liquid.
```

练习3-4 重写Java的markov程序，用数组(而不是Vector)表示Prefix类里的前缀。

3.6 C++

我们的第三个实现将在C++ 中完成。因为C++ 语言几乎是C的一个超集，它也能以C的形式使用，只要注意某些写法规则。实际上，前面 C版本的markov也是一个完全合法的C++ 程序。对C++而言，更合适的用法应该是定义一些类，建立起程序中需要的各种对象，或多或少像我们写Java程序时所做的那样，这样可以隐蔽起许多实现细节。我们在这里希望更前进一步，使用C++的Standard Template Library(标准模板库)，即STL。因为STL提供了许多内部机制，能完成我们需要做的许多事情。ISO的C++ 标准已经把STL作为语言定义的一部分。

STL提供了许多容器类，例如向量、链表、集合等，还包括了许多检索、排序、插入和删

China-pub.com 下载

除的基本算法。通过利用 C++ 的模板特性，每个STL算法都能用到很多不同的容器类上，容器类的元素可以是用户定义类型的或者是内部类型的 (如整数)。这里的容器都被描述为 C++ 模板，可以对特定类型进行实例化。例如，STL里有一个vector容器类，由它可以导出各种具体类型，比如vector<int>或vector<string>等。所有的vector操作，包括排序的标准算法等，都可以直接用于这些数据类型。

在STL里，除了有vector容器(它与Java的vector类差不多)，还提供了一个deque容器类。deque(念为deck)是一种双端队列，它正好能符合我们对前缀操作的需要：可以用它存放NPREF个元素，丢掉最前面的元素并在后面添一个新的，这都是 $O(1)$操作。实际上，STL的deque比我们需要的东西更一般，它允许在两端进行压入和弹出，而执行性能方面的保证是我们选择它的原因。

STL还提供了一个map容器，其内部实现基于平衡的树。在map中可以存储关键码—值对。map的实现方式保证，从任何关键码出发提取相关值的操作都是 $O(\log n)$。虽然map可能不如 $O(1)$的散列表效率高，但是，直接使用它就可以不必写任何代码，这样也很不错 (某些非标准的C++库提供了hash或hash_map容器，它们的性能可能更好些)。

我们也可以使用内部提供的比较函数，用于对前缀中各字符串做比较。

手头有了这些组件，有关代码可以流畅地做出来了。下面是有关声明：

```
typedef deque<string> Prefix;
map<Prefix, vector<string> > statetab; // prefix -> suffixes
```

STL提供了deque的模板，记法deque<string>将它指定为以字符串为元素的 deque。由于这个类型将在程序里多次出现，在这里用一个 typedef声明，将它另外命名为 Prefix。映射类型中将存储前缀和后缀，因为它在程序里只出现一次，我们就没有给它命名。这里声明了一个map类型的变量statetab，它是从前缀到后缀向量的映射。在这里工作比在 C或Java中更方便，根本不需要提供散列函数或者equal方法。

main函数对前缀做初始化，读输入 (对于标准输入，调用C++ iostream里的cin)，在输入最后加一个尾巴，然后产生输出。和前面各个版本完全一样。

```
// markov main: markov-chain random text generation
int main(void)
{
    int nwords = MAXGEN;
    Prefix prefix;                  // current input prefix

    for (int i = 0; i < NPREF; i++) // set up initial prefix
        add(prefix, NONWORD);
    build(prefix, cin);
    add(prefix, NONWORD);
    generate(nwords);
    return 0;
}
```

函数build使用iostream库，一次读入一个词：

```
// build: read input words, build state table
void build(Prefix& prefix, istream& in)
{
    string buf;

    while (in >> buf)
```

China-pub.com 下载


```
        add(prefix, buf);
}
```

字符串buf能够根据输入词的长度需要自动增长。

函数add能够进一步说明使用STL 的优越性：

```
// add: add word to suffix list, update prefix
void add(Prefix& prefix, const string& s)
{
    if (prefix.size() == NPREF) {
        statetab[prefix].push_back(s);
        prefix.pop_front();
    }
    prefix.push_back(s);
}
```

这几个非常简单的语句确实做了不少事情。 map容器类重载了下标运算符([]运算符)，使它在这里成为一种查询运算。表达式 statetab[prefix]在statetab里完成一次查询，以prefix作为查询的关键码，返回对于所找到的项的一个引用。如果对应的向量不存在，这个操作将建立一个新向量。 vector和deque类的push_back函数分别把一个新字符串加到向量或deque的最后；pop_front从deque里弹出头一个元素。

输出生成与前面的版本类似：

```
// generate: produce output, one word per line
void generate(int nwords)
{
    Prefix prefix;
    int i;

    for (i = 0; i < NPREF; i++) // reset initial prefix
        add(prefix, NONWORD);

    for (i = 0; i < nwords; i++) {
        vector<string>& suf = statetab[prefix];
        const string& w = suf[rand() % suf.size()];
        if (w == NONWORD)
            break;
        cout << w << "\n";
        prefix.pop_front();      // advance
        prefix.push_back(w);
    }
}
```

总的来说，这个版本看起来特别清楚和优雅 —— 代码很紧凑，数据结构清晰，算法完全一目了然。可惜的是，在这里也要付出一些代价：这个版本比原来的 C版本慢得多，虽然它还不是最慢的。不久我们将回头来讨论性能测试问题。

练习3-5 STL的强项是很容易在其中做不同数据结构的试验。修改这里C++ 版本的马尔可夫程序，用不同的结构表示前缀、后缀表以及状态表。对于不同结构，程序的执行性能有什么变化？

练习3-6 另写一个C++ 程序，只使用类和string数据类型，不使用其他高级的库功能。从风格和速度方面将它与采用STL的版本做各种比较。

3.7 Awk和Perl

为使这个演习比较圆满，我们也用两个应用广泛的脚本语言 (Awk和Perl)写了这个程序。

China-pub.com 下载

这些语言都提供了本应用所需要的各种特征，包括关联数组和字符串处理等。

关联数组实际上是散列表的一种包装，使用起来非常方便。它看起来像数组，但可以用任意的字符串或数，或者字符串和数的由逗号分隔的表作为下标。关联数组也是一种从一个数据类型到另一类型的映射。在Awk里，所有数组都是关联数组；Perl则不同，这里既有以整数作为下标的普通数组，也有关联数组——称为"散列"，这实际上也说明了它们的实现方法。

在程序的Awk和Perl实现里，前缀长度都固定为2。

```
# markov.awk: markov chain algorithm for 2-word prefixes
BEGIN { MAXGEN = 10000; NONWORD = "\n"; w1 = w2 = NONWORD }
{   for (i = 1; i <= NF; i++) {     # read all words
        statetab[w1,w2,++nsuffix[w1,w2]] = $i
        w1 = w2
        w2 = $i
    }
}
END {
    statetab[w1,w2,++nsuffix[w1,w2]] = NONWORD  # add tail
    w1 = w2 = NONWORD
    for (i = 0; i < MAXGEN; i++) {  # generate
        r = int(rand()*nsuffix[w1,w2]) + 1  # nsuffix >= 1
        p = statetab[w1,w2,r]
        if (p == NONWORD)
            exit
        print p
        w1 = w2         # advance chain
        w2 = p
    }
}
```

Awk是一个模式—操作对的语言：输入总以一次一行的方式读入，每个读入行都拿来与程序里的模式做匹配，与此同时，对各个成功匹配执行有关动作。这里存在着两个特殊的模式，BEGIN和END，它们分别能在输入的第一行之前和最后一行之后匹配成功。

动作是由花括号括起的一个语句块。在上面的Awk版本的马尔可夫程序里，BEGIN动作块对前缀和若干其他变量做初始化。

随后的一个语句块没有模式部分，这是一种默认方式，意味着这个块将对每个输入行执行一次。Awk自动把每个读入的行分割成一些域(由空白分隔的词)，它们将分别成为$1到$NF。变量NF的值是域的个数。语句：

```
statetab[w1,w2,++nsuffix[w1,w2]] = $i
```

建立从前缀到后缀的映射。数组nsuffix记录后缀个数，其元素nsuffix[w1,w2]记录与前缀对应的后缀的个数。而后缀本身则被做为数组 statetab的元素，如statetab[w1, w2, 1]，statetab[w1, w2, ,2 等等。

当END块执行时，所有内容都已经输入完毕。到这个时刻，对于每个前缀，nsuffix里都有一个元素记录着它对应的后缀个数，而在statetab里则存有相应个数的后缀。

用Perl语言写出的东西与此类似，但是需要另外用一个匿名数组(而不是用第三个下标变量)来保存后缀的有关情况。这里还需要用一个多重赋值完成对前缀的更新。Perl采用特殊字符表示变量的类型，$表示标量，@表示有下标的数组。这里的[ ]用于下标数组，而{ }表示散列。

China-pub.com 下载


```
# markov.pl: markov chain algorithm for 2-word prefixes

$MAXGEN = 10000;
$NONWORD = "\n";
$w1 = $w2 = $NONWORD;            # initial state
while (<>) {                     # read each line of input
    foreach (split) {
        push(@{$statetab{$w1}{$w2}}, $_);
        ($w1, $w2) = ($w2, $_); # multiple assignment
    }
}
push(@{$statetab{$w1}{$w2}}, $NONWORD);      # add tail

$w1 = $w2 = $NONWORD;
for ($i = 0; $i < $MAXGEN; $i++) {
    $suf = $statetab{$w1}{$w2}; # array reference
    $r = int(rand @$suf);        # @$suf is number of elems
    exit if (($t = $suf->[$r]) eq $NONWORD);
    print "$t\n";
    ($w1, $w2) = ($w2, $t);      # advance chain
}
```

和前面程序一样，映射本身保存在变量statetab里。程序中最关键的行是：

```
push(@{$statetab{$w1}{$w2}}, $_);
```

它把一个新后缀加入到存储在statetab{$w1}{$w2}的匿名数组的最后。在随后的生成阶段里，$statetab{$w1}{$w2}是对后缀数组的一个引用，而$suf->[$r]指向其中的第r个后缀。

与前三个程序相比，用Awk和Perl写的程序都短得多，但要把它们修改为能处理前缀词不是两个的情况却很困难。采用C++ 的STL实现，其核心部分(函数add和generate)看起来长一点，但是却更清晰。无论如何，脚本语言对有些情况是很好的选择，例如做试验性的程序设计，构造系统原型，以及做那些对运行时间要求不高的实际产品。

练习3-7 修改上面的Awk和Perl程序，设法使它们能处理任意长度的前缀。通过试验确定这种修改对性能的影响。

3.8 性能

我们对上面的几个实现做了些比较。对程序做计时用的是《圣经·雅各书》中的《诗篇》，其中共有42 685个词(5 238个不同的词，22 482个前缀)。这个文本里有足够多的重复出现的短语(如“Blessed is the ...”)，其中有一个后缀包含多于400个元素，另外还有几百个链包含几十个后缀。所以这是个很好的测试数据集。

> Blessed is the man of the net. Turn thee unto me, and raise me up, that I may tell all my fears. They looked unto him, he heard. My praise shall be blessed. Wealth and riches shall be saved. Thou hast dealt well with thy hid treasure: they are cast into a standing water, the flint into a standing water, and dry ground into watersprings.

下面的表格列出的是生成10 000个输出词所用的秒数。这里用的机器一台是 250 MHz的MIPS R10000，运行Irix 6.4系统，另一台是400 MHz的Pentium II，带有128 M内存，运行Windows NT系统。运行时间几乎完全由输入数据的规模决定，相对而言，输出生成是非常快的。在表格里，还以源程序行数的方式给出了程序的大致规模。

China-pub.com 下载

	250MHz R10000	400MHz Pentium II	源代码行数
C	0.36 sec	0.30 sec	150
Java	4.9	9.2	105
C++/STL/deque	2.6	11.2	70
C++/STL/list	1.7	1.5	70
Awk	2.2	2.1	20
Perl	1.8	1.0	18

C和C++ 程序都用带优化的编译器完成编译， Java程序运行时打开了即时编译功能。 Irix的C和C++编译是从三个不同编译器里选出的最快的一个，由 Sun SPARC和DEC Alpha机器得到的数据也差不多。C版本的程序是最快的，比别的程序都快得多。 Perl程序的速度次之。表格里的时间是我们试验的一个剪影，用的是特定的编译器和库。在你自己的环境里做，得到的结果也可能与此有很大差别。

Windows上的STL deque版本肯定存在什么问题。试验说明表示前缀的 deque用掉了大部分的运行时间，虽然在它里面从来都不超过两个元素。按说作为中心数据结构的映射所消耗的时间应该是最多的。把deque改成链表(在STL里是双链表)能使程序性质大大改善。另一方面，在Irix环境中把映射改为 (非标准的)hash容器则并没有产生多大影响。在我们的Windows机器上没有散列可以用。要完成上面说的这些修改，需要做的只是把 deque换成list，或者把map换成hash，然后重新做一下编译，这也是对 STL基本设计思想的有效性的一个认证。我们也认为， STL作为C++ 的一个新部分，仍然受到不成熟实现的损害。在使用STL的不同实现或使用不同数据结构时，导致的性能变化是不可预测的。Java也存在这个问题，那里的实现也变得很快。

对于测试一个有意产生大量随机输出的程序，实际上存在着一些具有挑战性的问题。我们怎么能知道它确实是能工作的？怎样知道它在所有情况下都能工作？在第 6章讨论测试时将提出一些建议，并描述如何测试马尔可夫程序。

3.9 经验教训

马尔可夫程序已经有了很长历史。其第一个版本是 Don P. Mitchell写的，后来由 Bruce Ellis做了些修改，它在80年代被大量应用到各种文献分析 (deconstructionist)活动中。这个程序一直潜伏在那里，直到我们想到把它用在一个大学课程中，作为讨论程序设计的一个例子。我们并不是简单地捡起已有的东西，而是用 C语言重新把它写出来，通过遇到的各种问题重新唤醒我们的记忆。而后我们又用几种不同语言重新写它，用各语言独特的习惯用法表述同样的基本想法。在这个课程之后，我们又多次重写这个程序，设法改进其清晰性和表现形式。

在这整个过程中，基本的设计并没有改变。最早的版本使用的也正是我们在这里给出的方式，虽然其中确实用了第二个散列表来表示各个词。如果我们还要重写它，基本情况大概也不会有多大变动。一个程序的设计根源于它的数据的形式。数据结构并没有定义所有的细节，但它们确实规定了整个解的基本构造。

有些数据结构选择造成的差异不太大，例如，是用链接表还是用可增长数组。有些实现方式比别的方式更具普遍性——例如，很容易把Perl和Awk程序改造成使用一个词或三个词前缀的程序，但要想使这个选择能够参数化，就会遇到很多麻烦。面向对象的语言有它们的优

China-pub.com
下载

越之处，对C++或Java的实现做点小修改，就可能使它们的数据结构适合英语之外的其他文本，例如程序(在那里空格可能是重要的东西)，或者乐谱，甚至产生测试序列的鼠标点击和菜单选择。

当然，即使数据结构本身差不多，在程序的一般表现方面，在源代码的大小方面，在程序执行性能方面也可以存在很大差异。粗略地说，使用较高级的语言比更低级的语言写出的程序速度更慢，但这种说法只是定性的，把它随意推广也是不明智的。大型构件，如 C++的STL或脚本语言里的关联数组、字符串处理，能使代码更紧凑，开发时间也更短。当然这些东西不是没有代价的，但是，性能方面的损失对于大部分程序而言可能并不那么重要。比如马尔可夫这样的程序，它不过只运行几秒钟。

当系统内部提供的代码太多时，人们将无法知道程序在其表面之下到底做了什么。我们应该如何评价这种对控制和洞察力的丧失，这是更不清楚的事情。这也就是 STL版本中遇到的情况，它的性能无法预料，也没有很容易的办法去解决问题。我们使用过一个很不成熟的实现，必须对它做一些修正后才能够运行我们的程序。但是，很少有人能有资源和精力去弄清其中的问题并且修复它们。

目前存在着一种对软件的广泛的不断增长的关注：当程序库、界面和工具变得越来越复杂时，它们也变得更难以理解和控制了。当所有东西都正常运转时，功能丰富的程序设计环境可以是非常有生产效率的，但是如果它们出了毛病，那就没什么东西可以依靠了。如果问题牵涉到的是性能或者某些难于捉摸的逻辑错误时，我们很可能根本没有意识到有什么东西出了毛病。

在这个程序的设计和实现过程中，我们看到了许多东西，这些对于更大的程序是很有教益的。首先是选择简单算法和数据结构的重要性，应该选择那些能在合理时间内解决具有预期规模的问题的最简单的东西。如果有人已经把它写好并放在库里，那是最好了。我们的C++ 程序由此获益匪浅。

按照Brooks的建议，我们发现最好是从数据结构开始，在关于可以使用哪些算法的知识的指导下进行详细设计。当数据结构安置好后，代码就比较容易组织了。

要想先把一个程序完全设计好，然后再构造它，这是非常困难的。构造现实的程序总需要重复和试验。构造过程逼迫人们去把前面粗略做出的决定弄清楚。这正是我们在写这些程序中遇到的情况。这个程序经历了许多细节方面的变化。应该尽可能从简单的东西开始，根据获得的经验逐步发展。如果我们的目标只是为个人兴趣而写一个马尔可夫链算法程序，很可能就在Awk或者Perl里写一个，可能也会不像这里这样仔细地打磨它，而是随其自然。

做产品代码要花费的精力比做原型多得多。例如可以把这里给出的程序看作是产品代码(因为它们已经被仔细打磨过，并经过了彻底的测试)。产品质量要求我们付出的努力要比个人使用的程序高一两个数量级。

练习3-8 我们已经看到马尔可夫程序在许多语言里的版本，包括 Scheme、Tcl、Prolog、Python、Generic Java、ML以及Haskell。从其中每一个，都可以看到其特殊的挑战性和优越性。在你最喜欢的语言里实现这个程序，并考察它的一般风格和性能。

补充阅读

标准模板库在许多书籍里都有介绍，包括 Matthew Austern的《类属程序设计与 STL》

China-pub.com 下载

(Generic Programming and the STL，Addison-Wesley，1998)。对C++ 语言本身的参考文献当然是Bjarne Stroustrup的《C++程序设计语言》(C++ Programming Language第3版，Addison-Wesley，1997)。对于Java，我们参考了Ken Arnold和James Gosling的《Java程序设计语言》(The Java Programming Language第2版，Addison-Wesley，1998)，对Perl语言的最好描述是Larry Wall、Tom Christiansen和Randal Schwartz的《Perl程序设计》(Programming Perl 第2版，O'Reilly，1996)。

隐藏在设计模式后面的基本思想是：大部分程序所采用的不过是很少几种不同的设计结构，与此类似，实际上也只有不多的几种基本数据结构。说的远一点，这与我们在第 1章讨论过的编码习惯用法也是很相像的。这方面的经典参考文献是 Erich Gamma、Richard Helm、Ralph Johnson和John Vlissides的《设计模式：可重用面向对象软件的要素》(Design Pattern: Elements of Reusable Object-Oriented Software，Addison-Wesley，1995)。

`Markov`程序原来称为`shaney`，其传奇经历刊登在1989年6月号《科学美国人》杂志的“趣味计算”专栏里，该文重载于A. K. Dewdney的《神奇的机器》(The Magic Machine，W. H. Freeman，1990)。

China-pub.com
下载

第4章 界 面

在造墙之前，我必须设法弄清
该把什么放在墙里，什么放在墙外，
最需要防御的又是什么。
确实有些东西不喜欢墙，
总希望它倒下来。

—— Robert Frost，《修墙》

设计的真谛，就是在一些互相冲突的需求和约束条件之间寻找平衡点。如果要写的是一个自给自足的小程序，那么常常可以找到许多折衷方式，所做出的特定选择将产生一些后果，会遗留在系统里，但其影响还只限于写程序的个人。如果写出的代码是为了别人使用的，这些选择决定就会产生更广泛的影响。

在进行设计的时候，必须考虑的问题包括：

- 界面：应提供哪些服务和访问？界面在效能上实际成为服务的提供者和使用者之间的一个约定。在这里要做的是提供一种统一而方便的服务，使用方便，有足够丰富的功能，而又不过多过滥以至无法控制。
- 信息隐藏：哪些信息应该是可见的，哪些应该是私有的？一个界面必须提供对有关部件的方便访问方式，而同时又隐蔽其实现的细节。这样，部件的修改才不会影响到使用者。
- 资源管理：谁负责管理内存或者其他有限的资源？这里的主要问题是存储的分配和释放，以及管理共享信息的拷贝等。
- 错误处理：谁检查错误？谁报告？如何报告？如果检查中发现了错误，那么应该设法做哪些恢复性操作？

在第2章里，我们关注的主要是孤立的程序片段 —— 数据结构，它们是构造各种系统的基础。在第3章我们考察了怎样把它们组合成一个小程序。现在我们的论题将转到部件之间的界面，这些部件的来源又可能不同。在这一章里，我们将针对一个日常任务构造出一个函数库和一些数据结构，通过这个工作展示界面设计中的问题。其间我们还要提出一些设计原则。在设计界面时，我们需要做出大量的决定，而其中绝大部分常常又是在无意识中做出的。如果在这个过程中不遵循某些原则，产生出来的可能就是某种非常随意的界面，它们将会妨害甚至挫败我们日常的程序设计工作。

4.1 逗号分隔的值

逗号分隔的值(comma-separated value)，或CSV，是个术语，指的是一种用于表示表格数据的自然形式，使用很广泛。这里表格的每行是个正文行，行中不同的数据域由逗号分隔。看前面一章最后的那个表格，其开始的一段用 CSV格式表示大概是：

China-pub.com 下载

```
,"250MHz","400MHz","Lines of"
,"R10000","Pentium II","source code"
C,0.36 sec,0.30 sec,150
Java,4.9,9.2,105
```

这种格式通常由某些程序(如电子报表程序等)进行读写。自然，这种形式也会出现在Web页里，作为一种服务项目，例如显示股票价格行情表等。一个 Web页上典型的股票行情表显示出来可能是这种样子：

Symbol	Last Trade		Change		Volume
LU	2:19PM	86-1/4	+4-1/16	+4.94%	5,804,800
T	2:19PM	60-11/16	−1-3/16	−1.92%	2,468,000
MSFT	2:24PM	106-9/16	+1-3/8	+1.31%	11,474,900

下载电子表格格式

以交互方式，用一个Web浏览器提取数据是可行的，但是又很耗费时间，令人生厌：启动浏览器、等待、观看狂泻而来的广告，输入一个股票表、等待、等待、等待，观看又一轮倾泻，最后才得到几个数。要处理这些数还要做许多交互式操作：选择“下载电子表格格式”链接，取得一个包含着差不多同样数据的文件，其中有些像下面这样的CSV数据行(经过编辑，以便放在这里)：

```
"LU",86.25,"11/4/1998","2:19PM",+4.0625,
        83.9375,86.875,83.625,5804800
"T",60.6875,"11/4/1998","2:19PM",-1.1875,
        62.375,62.625,60.4375,2468000
"MSFT",106.5625,"11/4/1998","2:24PM",+1.375,
        105.8125,107.3125,105.5625,11474900
```

在这个过程中，明显欠缺的是让机器帮助做的原理。浏览器使你的计算机可以访问位于远程服务器的数据，但更方便的是在无须人工交互的方式下提取数据。在所有按钮动作的后面实际上是一个纯粹的文本处理过程——浏览器读HTML，你输入一些文本，浏览器把它送到服务器，再读一些回来。如果有合适的工具和语言，自动提取数据是不难做到的。下面写的是一段Tcl语言的程序，它能访问有关股票行情的Web站点，提取上面说的那种形式的CSV数据，数据中带有几个前导行：

```
# getquotes.tcl: stock prices for Lucent, AT&T, Microsoft

set so [socket quote.yahoo.com 80]   ;# connect to server
set q "/d/quotes.csv?s=LU+T+MSFT&f=sl1d1t1c1ohgv"

puts $so "GET $q HTTP/1.0\r\n\r\n"   ;# send request
flush $so
puts [read $so]                      ;# read & print reply
```

跟在&符号后面那段神秘的序列f=…是一段不作为文档的控制串，其作用就像printf的第一个参数，用来确定被提取数据值是什么。经过试验，我们确定其中 s表示股票符号，l1是最终价格，c1是昨天以来的变化，如此等等。这里重要的并不是细节，因为它们都很可能会改变，最重要的是自动化：在没有人工干预的情况下提取所需数据，并把它们转换为人们需要的形式。我们应该让机器做这些事。

在典型情况下，运行getquotes程序只需要几分之一秒的时间，远快于用浏览器来做。一旦有了数据，我们将希望能对它做进一步处理。如果能有一个方便的库，完成数据格式的

China-pub.com
下载

转入转出，像CSV这样格式的数据是很容易使用的。库里最好还带有另一些辅助处理功能，例如数值转换等等。但是，我们不知道哪里有处理CSV数据的库，因此只能自己写一个。

在下面几节里，我们要写出这个库的三个版本，其功能是读CSV数据，将它转换为内部表示。库是需要与其他软件一起工作的软件，在这个工作中，我们要讨论一些在设计这类软件时经常遇到的问题。例如，由于没有CSV的标准定义，它的设计不能是基于精确规范进行的，这也是界面设计时经常面临的情况。

4.2 一个原型库

我们不大可能在第一次设计函数库或者界面时就做得很好。正如Fred Brooks有一次写的"要计划扔掉一个，你总会这样做，无论是以什么方式"。Brooks的话是针对大系统说的，但是，这种说法实际上适用于任何实质性的软件片段。事情往往是这样，只有在你已经构造和使用了程序的一个版本之后，才能对如何把系统设计正确有足够的认识。

基于这种理解，我们构造CSV库时准备采用的途径就是：先搞出一个将要丢掉的，搞出一个原型。我们的第一个版本将忽略许多完备的工程库应该牵涉的难点，但却又必须足够完整和有用，以便能帮助我们熟悉问题。

我们的出发点是一个名为csvgetline的函数，它由文件读入一个CSV数据行，将它放入缓冲区，在一个数组里把该行分解为一些数据域，删除引号，最后返回数据域的个数。以前我们已经在自己熟悉的几乎所有语言里写过类似代码，所以这是一个熟悉的工作。下面是用C语言写的原型版本，这里用问号做标记，是因为它仅仅是个原型：

```
?   char buf[200];       /* input line buffer */
?   char *field[20];     /* fields */
?
?   /* csvgetline: read and parse line, return field count */
?   /* sample input: "LU",86.25,"11/4/1998","2:19PM",+4.0625 */
?   int csvgetline(FILE *fin)
?   {
?       int nfield;
?       char *p, *q;
?
?       if (fgets(buf, sizeof(buf), fin) == NULL)
?           return -1;
?       nfield = 0;
?       for (q = buf; (p=strtok(q, ",\n\r")) != NULL; q = NULL)
?           field[nfield++] = unquote(p);
?       return nfield;
?   }
```

函数前面的注释包含了本程序能接受的输入行的一个例子，对于理解那些剖析较复杂输入的程序而言，这种形式的注释是非常有帮助的。

由于CSV数据太复杂，不可能简单地用函数scanf做输入剖析，我们使用了C标准库函数strtok。对strtok(p, s)的调用将返回p中的一个标识符的指针，标识符完全由不在s中的字符构成。strtok将原串里跟在这个标识符之后的字符用空字符覆盖掉，用这种方式表示标识符的结束。在第一次调用时，strtok的第一个参数应该是原来的字符串，随后的调用都应该用NULL作为第一个参数，指明这次扫描应该从前次调用结束的地方继续下去。这是一个很糟糕的界面，在函数的不同调用之间，strtok需要在某个隐秘处所存放一个变量。这样，同

China-pub.com 下载

时激活的调用序列就只能有一个，如果有多个无关的调用交替进行，它们之间必定会互相干扰。

函数unquote的功能是去除像前面例子的数据行里那些表示开头和结束的引号。它并不处理嵌套引号的问题，对于原型而言这样做已足够了。当然这种做法还不够一般。

```
?    /* unquote: remove leading and trailing quote */
?    char *unquote(char *p)
?    {
?        if (p[0] == '"') {
?            if (p[strlen(p)-1] == '"')
?                p[strlen(p)-1] = '\0';
?            p++;
?        }
?        return p;
?    }
```

下面的简单测试程序可以帮我们确认csvgetline能够工作：

```
?    /* csvtest main: test csvgetline function */
?    int main(void)
?    {
?        int i, nf;
?
?        while ((nf = csvgetline(stdin)) != -1)
?            for (i = 0; i < nf; i++)
?                printf("field[%d] = '%s'\n", i, field[i]);
?        return 0;
?    }
```

在printf里用一对单引号括起数据域，这起着划清界限的作用，还能帮助我们发现空格处理不正确一类的错误。

我们可以用这个函数处理getquotes.tcl生成的输出：

```
% getquotes.tcl | csvtest
...
field[0] = 'LU'
field[1] = '86.375'
field[2] = '11/5/1998'
field[3] = '1:01PM'
field[4] = '-0.125'
field[5] = '86'
field[6] = '86.375'
field[7] = '85.0625'
field[8] = '2888600'
field[0] = 'T'
field[1] = '61.0625'
...
```

(我们已经通过编辑方式删去了HTTP头部的行。)

现在我们有了一个原型，看起来它能对付上面给出的那种数据形式。不过，再用另外一些东西试试它可能是更明智的，特别是如果打算把它提供给别人使用。我们发现了另外的Web站点，从那里下载了一些股票行情表，得到包含类似信息的文件，但文件的格式有些不同：分隔数据记录用的是回车符号(\r)而不是换行符号，文件最后也没有表示文件结束的回车字符。我们对它做了些格式编辑，以便能放在这里：

```
"Ticker","Price","Change","Open","Prev Close","Day High",
    "Day Low","52 Week High","52 Week Low","Dividend",
    "Yield","Volume","Average Volume","P/E"
"LU",86.313,-0.188,86.000,86.500,86.438,85.063,108.50,
```

China-pub.com
下载


```
    36.18,0.16,0.1,2946700,9675000,N/A
"T",61.125,0.938,60.375,60.188,61.125,60.000,68.50,
    46.50,1.32,2.1,3061000,4777000,17.0
"MSFT",107.000,1.500,105.313,105.500,107.188,105.250,
    119.62,59.00,N/A,N/A,7977300,16965000,51.0
```

对于这样的输入，我们的原型败得很惨。

我们在查看了一个数据来源之后设计了这个原型，并且只用同样来源的数据做过一些测试。因此，如果在使用其他来源的数据时发现了程序里的大错误，我们一点都不应该对此感到惊奇。长的输入行、很多的数据域以及未预料到的或者欠缺的分隔符都可能造成大麻烦。这个脆弱的原型作为个人使用而言可能还勉强，或者可以用来说明这种方法的可行性，但绝不可能有更多的意义。在着手开始下一个实现之前，我们需要重新认真地想一想，到底应该如何做这个设计。

现在这个原型里包含着我们的许多决定，有些是明显的，也有些是隐含的。下面列出的是前面做过的一些选择，对一个通用库而言，它们并不都是最好的选择。实际上，每个选择都提出了一个问题，需要进一步仔细考虑。

- 原型没有处理特别长的行、很多的域。遇到这些情况时它可能给出错误结果甚至垮台，因为它没有检查溢出，在出现错误时也没有返回某种合理的值。
- 这里假定输入是由换行字符结尾的行组成。
- 数据域由逗号分隔，数据域前后的引号将被去除，但没有考虑嵌套的引号或逗号。
- 输入行没有保留，在构造数据域的过程中将它覆盖掉了。
- 在从一行输入转到另一行时没有保留任何数据。如果需要记录什么东西，那么就必须做一个拷贝。
- 对数据域的访问是通过全局变量(数组`field`)进行的。这个数组由`csvgetline`与调用它的函数共享。这里对数据域内容或指针的访问都没有任何控制。对于超出最后一个域的访问也没有任何防御措施。
- 使用了全局变量，这就使得这个设计不能适合多线程环境，甚至也不允许两个交替进行的调用序列。
- 调用库的程序必须显式地打开和关闭文件，`csvgetline`做的只是从已经打开的文件读入数据。
- 输入和划分操作纠缠在一起：每个调用读入一行并把它切分为一些域，不管实际应用中是否真的需要后一个服务。
- 函数返回值表示一个输入行中的数据域个数，每行都被切分，以便得到这个数值。这里也没有办法把出现错误和文件结束区分开。
- 除了更改代码外，没有任何办法来改变这些特性。

以上所列的并不完全，它说明了许多可能的设计选择，各个决定都已经被编织到代码里。对于一个急迫的工作，这样做还说得过去，例如要剖析从一个已知信息源来的固定格式的东西。但是，如果格式可能有变化，例如逗号出现在引号括起的串内，或者服务器产生很长的行或很多的域，那么又会怎么样呢？

这些具体东西看起来都好对付，因为这个“库”很小，而且只是一个原型。但是，如果设想一下，当这个库出台几个月或者几年以后，它不可能变成某些大系统的一部分，而系统的规范又在随着时间的推移不断变化，`csvgetline`能怎样适应这些情况吗？如果这个程序

China-pub.com 下载

是提供给别人用的，在原始设计中的这些仓促选择引起的麻烦就可能到许多年后才浮现出来。这正是许多不良界面的历史画卷。事实确实非常令人沮丧，许多仓促而就的肮脏代码最后变成广泛使用的软件，在那里它们仍然是肮脏的，而且常常也达不到它们应有的速度。

4.3 为别人用的库

借助于从原型中学到的东西，我们现在希望能构造出一个值得普遍使用的库。一个最明显的需求是，必须使csvgetline更健壮，使它能够处理很长的行和很多的域，它也必须能更仔细地剖析数据域。

要建立一个其他人能用的界面，我们必须考虑在本章开始处列出的那些问题：界面、信息隐藏、资源管理和错误处理。它们的相互作用对设计有极强的影响。我们把这些问题分割开是有点太随意了，实际上它们是密切相关的。

界面。我们决定提供3个操作：

char *csvgetline(FILE *)：读一个新CSV行

char *csvfield(int n)：返回当前行的第n个数据域

int csvnfield(void)：返回当前行中数据域的个数

函数csvgetline的返回值应该是什么？它应该返回尽可能方便使用的有用信息。正是这种想法导致我们在原型里让它返回域的个数。而如果要这样做，就必须先计算出域的个数，即使还没有用它们。另一个可能性是令函数返回行的长度，但这又牵涉到是否把换行符计算在内的问题。经过一些试验，我们决定让 csvgetline返回指向输入行本身的指针。如果遇到文件结束就返回NULL。

我们令csvgetline在返回输入行前去掉最后的换行符，因为在必要时很容易恢复。

域的定义是很复杂的，我们试图使该定义符合见到过的各种实际的电子表格或者其他程序。一个域是0个或多个字符的序列。域由逗号分隔，开头和尾随的空格应该保留。一个域可能由一对双引号括起，在这种情况下它还可能包含逗号。一个引号括起来的数据域里可能包含双引号本身，这需要用连续两个双引号表示，例如 CSV域 "x" "y" 定义的实际上是串x"y。域可以为空，两个连续的引号 "" 表示空，连续出现的逗号也表示有一个空域。

域从0开始编号。如果用户要求一个不存在的域，例如 csvfield(-1)或者csvfield(100000)，我们又该怎么办？我们可以返回 "" (一个空串)，因为这样可以做打印或比较。这里的程序要准备处理数目不定的域，不应该让它们再肩负处理根本不存在的东西的特殊使命。不过这种选择无法区分空域和不存在。另外两种选择是输出错误消息，或者是终止执行。我们很快就会说明为什么这些方式也很不理想。最后，我们决定让函数返回NULL，这是C中表示不存在的常用方式。

信息隐藏。这个库应该对输入行长或域的个数没有限制。为了达到这个目的，或者是让调用程序为它提供存储，或者是被调用者 (库)自己需要做分配。在调用库函数 fgets时，传给它一个数组和一个最大长度，如果输入行比缓冲区长，那么就将它切分成片段。这种方式对CSV界面不太合适，我们的库在发现需要更多存储时应该能自动做存储分配。

这样就只有csvgetline知道存储管理情况，外边程序对其存储组织方式完全不能触及。提供这种隔离的最好途径是通过函数界面： csvgetline读入一行，无论它有多大；csvfiled(n)返回到当前行的第n个域那些字节的指针；csvnfield返回当前行中域的个数。

China-pub.com
下载

当程序遇到更长的行或者更多的域时，它使用的存储就必须增加。我们把如何做这件事的细节都隐藏在csv函数中，其他程序部分完全不知晓这方面的情况。例如，库函数是先使用小数组而后再扩大？还是使用了一个非常大的数组？或者用的是某种与此完全不同的办法？从界面上也看不到存储在何时释放。

如果用户只调用了csvgetline，那么就没有必要做域的切分，这件事可以随后根据命令再做。关于这种切分是立即做(读到行的时候立刻就做)，或是延后再做(直到有了对数据域或者域个数的要求时再做)，还是以更延后的方式做(只有需要用的域才做切分)，这是另一个实现细节，对用户也是隐藏的。

资源管理。必须确定谁负责共享的信息。函数csvgetline是直接返回原始数据，还是做一个拷贝？我们确定的方式是：csvgetline的返回值是到原始输入的一个指针，在读下一行时这里将被复写。数据域将在输入行的一个拷贝上构造，csvfield返回指向这个拷贝里的域指针。按照这种安排，如果需要保存或修改某个特殊的行或者域，用户就必须自己做一个拷贝。当这种拷贝不再需要时，释放存储也是用户的责任。

谁来打开和关闭文件？做文件打开的部分也应负责关闭：互相匹配的操作应该在同一个层次或位置完成。我们将假定csvgetline调用时得到的是一个打开了的文件的FILE指针，它的调用者也要负责在处理完毕后关闭文件。

在一个库与它的调用程序之间共享的、或者传递通过它们之间的界限的资源，管理起来总是很困难的，这里经常出现一些合理而又互相矛盾的道理，要求我们做这种或者那种选择。在共享资源的责任方面常常出现错误或误解，这是程序错误最常见的根源之一。

错误处理。由于确定了csvgetline返回NULL，在这里就没有办法区分文件结束和真正的错误，例如存储耗尽等情况。类似地，用户访问不存在的域也不产生错误。这里我们也可以参考ferror的方法，给界面增加一个csvgeterror函数，它报告最近发生的一个错误。为了简单起见，这个版本里将不包括这种功能。

这里有一个基本原则：在错误发生的时候，库函数绝不能简单地死掉，而是应该把错误状态返回给调用程序，以便那里能采取适当的措施。另一方面，库函数也不应该输出错误信息，或者弹出一个会话框，因为这个程序将来可能运行在某种环境里，在那里这种信息可能干扰其他东西。错误处理本身就是一个值得仔细研究的题目，本章后面还有进一步的讨论。

规范。把上面做出的这些决定汇集到一起，就形成一个规范，它说明了csvgetline能提供什么服务，应该如何使用等等。对于一个大项目，规范是在实现之前做出来的，因为写规范和做实现的通常是不同的人，他们甚至来自不同的机构。但是，在实践中这两件事常常是一起做的，规范和实现一起发展。甚至在有些时候，“规范”不过就是为了写明已经做好的代码大概是干了些什么。

最好的方式当然是及早写出规范，而后，随着从正在进行的实现中学到的新情况，对规范进行必要的修改。规范写得越精确越细心，程序工作得很好的可能性也就越大。即使对于个人使用的程序，先准备一个具有合理完整性的规范也是很有价值的，因为它促使人们考虑各种可能性，并记录自己做过的选择。

就我们的目的而言，有关规范应该包含函数的原型、函数行为的细节描述、各种责任和假设等：

域由逗号分隔。

China-pub.com 下载

一个域可能由一对双引号 "..."括起。
一个括起的域中可以包含逗号，但不能有换行。
一个括起的域中可以包含双引号 " 本身，表示为 ""。
域可以为空； "" 和一个空串都表示空的域。
引导的和尾随的空格将预留。

`char *csvgetline(FILE *f);`

从打开的输入文件f读入一行；
假定输入行的结束标志是\r、\n、\r\n、或EOF。
返回指向行的指针，行中结束符去掉；如果遇到 EOF则返回NULL[⊖]。
行可以任意长，如果超出存储的限度也返回 NULL。
行必须当作只读存储看待；
如果需要保存或修改，调用者必须自己建立拷贝。

`char *csvfield(int n);`

域从0开始编号。
返回由csvgetline最近读入的行的第n个域；
如果n<0或超出最后一个域，则返回NULL。
域由逗号分隔。
域可以用 "..." 括起来，这些引号将被去除；
在 "..." 内部的 "" 用"取代，内部的逗号不再作为分隔符。
在没有引号括起来的域里，引号当作普通字符。
允许有任意个数和任意长度的域；
如果超出存储的限度，返回NULL。
域必须当作只读存储看待；
如果需要保存或修改，调用者必须自己建立拷贝。
在调用csvgetline之前调用本函数，其行为没有定义。

`int csvnfield(void);`

返回由csvgetline最近读入的行的长度。
在调用csvgetline之前调用本函数，其行为没有定义。

这个规范还是遗留下一些未尽的问题。例如，如果 csvfield或csvnfiled在csvgetline遇到EOF之后被调用，它们的返回值是什么？具有错误形式的域将如何处理？要想把所有这些难题都解决，即使对一个很小的系统也是很困难的；对大系统而言那就是真正的挑战，当然我们还是应该试一试。通常的情况就是这样，直到实现已经在进行时，还可能发现有些遗漏的东西。

这节剩下的部分就是csvgetline的一个符合上面的规范的新实现。我们把这个库分成两个文件，头文件csv.h包含了函数声明，表示的是界面的公共部分；实现文件 csv.c是程序代码。使用者应该在他们的代码中包括这里的 csv.h，并把csv.c编译后的代码连接到他们的代码上。源代码从来都不必是可见的。

⊖ 这个说法应该准确理解。作者的意思是“直接遇到 EOF时返回NULL”。遇到一个行后跟EOF(行可能由EOF结束)时，仍然正常返回这个行。按照C语言的规定，随后再读输入将直接得到EOF。——译者

China-pub.com
下载

这里是头文件：

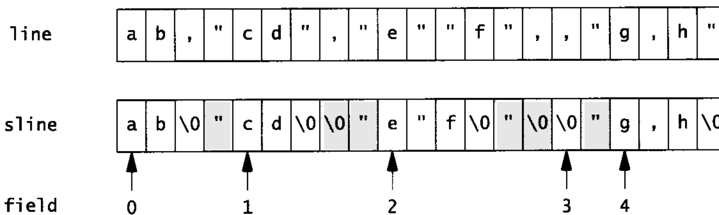
```
/* csv.h: interface for csv library */

extern char *csvgetline(FILE *f); /* read next input line */
extern char *csvfield(int n);     /* return field n */
extern int csvnfield(void);       /* return number of fields */
```

用于存储正文行的内部变量，以及split等内部函数都被声明为static，这样就使它们只在自己的定义文件里是可见的。这是C语言里最简单的信息隐蔽方法。

```
enum { NOMEM = -2 };          /* out of memory signal */

static char *line    = NULL;  /* input chars */
static char *sline   = NULL;  /* line copy used by split */
static int  maxline  = 0;     /* size of line[] and sline[] */
static char **field  = NULL;  /* field pointers */
static int  maxfield = 0;     /* size of field[] */
static int  nfield   = 0;     /* number of fields in field[] */

static char fieldsep[] = ","; /* field separator chars */
```

所有这些变量都静态地进行初始化，这些初始值将被用来检测是否需要建立或增大数组。

上述声明描述了一种很简单的数据结构。数组 line存放输入行， sline用于存放由line复制而来的字符行，并用于给每个域添加结束符号。数组 field指向sline的各个项。下面的图显示出这三个数组的状态，表示在输入行 ab,"cd","e""f",,"g,h"被处理完之后的情况。加阴影的字符不属于任何一个域。



这里是csvgetline的定义：

```
/* csvgetline:  get one line, grow as needed */
/* sample input: "LU",86.25,"11/4/1998","2:19PM",+4.0625 */
char *csvgetline(FILE *fin)
{
    int i, c;
    char *newl, *news;

    if (line == NULL) {            /* allocate on first call */
        maxline = maxfield = 1;
        line = (char *) malloc(maxline);
        sline = (char *) malloc(maxline);
        field = (char **) malloc(maxfield*sizeof(field[0]));
        if (line == NULL || sline == NULL || field == NULL) {
            reset();
            return NULL;           /* out of memory */
        }
    }
    for (i=0; (c=getc(fin))!=EOF && !endofline(fin,c); i++) {
        if (i >= maxline-1) {      /* grow line */
            maxline *= 2;          /* double current size */
            newl = (char *) realloc(line, maxline);
```

China-pub.com 下载

```
            news = (char *) realloc(sline, maxline);
            if (newl == NULL || news == NULL) {
                reset();
                return NULL;     /* out of memory */
            }
            line = newl;
            sline = news;
        }
        line[i] = c;
    }
    line[i] = '\0';
    if (split() == NOMEM) {
        reset();
        return NULL;             /* out of memory */
    }
    return (c == EOF && i == 0) ? NULL : line;
}
```

一个输入行被积累存入line，必要时将调用realloc使有关数组增大，每次增大一倍，就像第2.6节中所讲的那样。在这里还需要保持数组sline与line的大小相同。csvgetline调用split，在数组field里建立域的指针，如果需要的话，这个数组也将自动增大。

按照我们的习惯，开始时总把数组定义得很小，当需要时再让它们增大，这种方法能保证增大数组的代码总会执行到。如果分配失败，我们就调用reset把所有全局变量恢复到开始的状态，以使随后对csvgetline的调用还有成功的机会。

```
/* reset: set variables back to starting values */
static void reset(void)
{
    free(line); /* free(NULL) permitted by ANSI C */
    free(sline);
    free(field);
    line = NULL;
    sline = NULL;
    field = NULL;
    maxline = maxfield = nfield = 0;
}
```

函数endofline处理输入行的各种可能结束情况，包括回车、换行或两者同时出现，或者EOF：

```
/* endofline: check for and consume \r, \n, \r\n, or EOF */
static int endofline(FILE *fin, int c)
{
    int eol;

    eol = (c=='\r' || c=='\n');
    if (c == '\r') {
        c = getc(fin);
        if (c != '\n' && c != EOF)
            ungetc(c, fin); /* read too far; put c back */
    }
    return eol;
}
```

在这里使用一个独立函数是很有必要的，因为标准输入函数不会处理实际输入中可能遇到的各种各样乖张古怪的格式。

前面的原型里用strtok找下一标识符，其方法是查找一个分隔符，一般是个逗号。可是

China-pub.com
下载

这种做法无法处理引号内部的逗号。在split的实现里必须反映这个重要变化，虽然它的界面可以和strtok相同。考虑下面的输入行：

```
"",,""
,"",
,,
```

这里每行都包含三个空的域。为了保证split能剖析这些输入以及其他的罕见输入，这个函数将会变得更复杂一些。这又是一个例子，说明一些特殊情况和边界条件会对程序起到某种支配作用。

```
/* split: split line into fields */
static int split(void)
{
    char *p, **newf;
    char *sepp; /* pointer to temporary separator character */
    int sepc;   /* temporary separator character */

    nfield = 0;
    if (line[0] == '\0')
        return 0;
    strcpy(sline, line);
    p = sline;

    do {
        if (nfield >= maxfield) {
            maxfield *= 2;          /* double current size */
            newf = (char **) realloc(field,
                        maxfield * sizeof(field[0]));
            if (newf == NULL)
                return NOMEM;
            field = newf;
        }
        if (*p == '"')
            sepp = advquoted(++p);  /* skip initial quote */
        else
            sepp = p + strcspn(p, fieldsep);
        sepc = sepp[0];
        sepp[0] = '\0';             /* terminate field */
        field[nfield++] = p;
        p = sepp + 1;
    } while (sepc == ',');

    return nfield;
}
```

在循环中，数组可能根据需要增大，随后它调用一个或两个函数，对下一个域进行定位和处理。如果一个域由引号开头，advquoted将找出这个域，并返回指向这个域后面的分隔符的指针值。如果域不是由引号开头，程序就调用标准库函数strcspn(p,s)，找下一个逗号，该函数的功能是在串p里查找s中任意字符的下一次出现，返回查找中跳过的字符个数。

数据域里的引号由两个连续的引号表示，advquoted需要把它们缩成一个，它还将删除包围着数据域的那一对引号。由于要考虑处理某些不符合我们规范的可能输入，例如"abc"def，这又会给程序增加一些复杂性。对于这种情况，我们把所有跟在第二个引号后面的东西附在已有内容后面，把直到下一分隔符的所有东西作为这个域的内容。 Microsoft的

China-pub.com 下载

Excel使用的看来是类似算法。

```
/* advquoted: quoted field; return pointer to next separator */
static char *advquoted(char *p)
{
    int i, j;

    for (i = j = 0; p[j] != '\0'; i++, j++) {
        if (p[j] == '"' && p[++j] != '"') {
            /* copy up to next separator or \0 */
            int k = strcspn(p+j, fieldsep);
            memmove(p+i, p+j, k);
            i += k;
            j += k;
            break;
        }
        p[i] = p[j];
    }
    p[i] = '\0';
    return p + j;
}
```

由于输入行都已经切分好，csvfield和csvnfield的实现非常简单：

```
/* csvfield:  return pointer to n-th field */
char *csvfield(int n)
{
    if (n < 0 || n >= nfield)
        return NULL;
    return field[n];
}

/* csvnfield:  return number of fields */
int csvnfield(void)
{
    return nfield;
}
```

最后，我们可以对原有测试驱动程序稍微做点修改，再用它来检测这个新版本的库。由于新库保留了输入行的副本(前面原型没保留)，在这里可以先打印出原来的行，然后再打印各个域：

```
/* csvtest main: test CSV library */
int main(void)
{
    int i;
    char *line;

    while ((line = csvgetline(stdin)) != NULL) {
        printf("line = '%s'\n", line);
        for (i = 0; i < csvnfield(); i++)
            printf("field[%d] = '%s'\n", i, csvfield(i));
    }
    return 0;
}
```

这就完成了库的C语言版本，它能够处理任意长的输入，对某些格式乖张的数据也能够做出合理处理。其中也付出了一些代价：新库的大小超过第一个原型的四倍，而且包含一些很复杂的代码。在从原型转换到产品时，程序在大小和复杂性方面有所扩张是很典型的情况。

练习4-1 在数据域切分方面有多种延后再做的可能性。其中包括：一次完成所有切分，但是只在要求做的时候才做；只切分所要求的域；一直切分到所要求的域；等等。列举各种可能

China-pub.com
下载

性，评价不同做法的好处和潜在困难。然后把它们写出来，并测试其速度。

练习4-2 加上一个功能，使分隔符可以改变为：(a) 任意的字符类；(b) 不同域可以有不同分隔符；(c) 正则表达式(见第9章)。这时库的界面应该是什么样的？

练习4-3 在这里我们以C提供的静态初始化机制为基础，实现了一个仅动作一次的开关：当一个指针是NULL时就执行初始化。另一种可能性是要求用户调用一个显式的初始化函数，其中还可以包括建议数组初始大小的参数等。设法实现这个库的一个新版本，使之能同时具有两种方式的优点。在新版本中reset将起什么作用？

练习4-4 设计和实现一个建立CSV格式数据的库。最简单的方式可能是以一个字符串数组为参数，加上引号和逗号打印出来。复杂点的方式可以采用类似printf的格式串，如果想这样做，可以参看第9章关于记法的一些建议。

4.4 C++ 实现

本节打算写一个C++ 版本的CSV库，它应该能克服C版本里遗留下的某些限制。为此我们需要对原规范做一些改变，其中最重要的是把函数处理的对象由 C语言的字符数组变成 C++的字符串。使用C++的字符串功能将自动解决许多存储管理方面的问题，因为有关的库函数能帮我们管理存储。由于可以让域函数返回字符串，这就允许调用程序直接修改它们，比原来的版本更加灵活。

类Csv定义了库的公共界面，它明确地隐藏了实现中使用的一些变量和函数。由于在一个类对象里包含了所有的状态信息，建立多个 Csv的实例也不会有问题，不同实例之间是相互独立的，这就使我们的程序能够同时处理多个CSV输入流。

```
class Csv { // read and parse comma-separated values
    // sample input: "LU",86.25,"11/4/1998","2:19PM",+4.0625

  public:
    Csv(istream& fin = cin, string sep = ",") :
        fin(fin), fieldsep(sep) {}

    int getline(string&);
    string getfield(int n);
    int getnfield() const { return nfield; }

  private:
    istream& fin;            // input file pointer
    string line;             // input line
    vector<string> field;    // field strings
    int nfield;              // number of fields
    string fieldsep;         // separator characters

    int split();
    int endofline(char);
    int advplain(const string& line, string& fld, int);
    int advquoted(const string& line, string& fld, int);
};
```

类的建构函数对参数提供了默认定义，按默认方式建立的 Csv对象将直接从标准输入读数据，使用正规的域分隔符。这些都可以显式地通过实际参数重新设定。

为了管理字符串，在这里使用了标准 C++的string和vector类，没有采用C风格的字

China-pub.com 下载

符串。对于string而言，它没有“不存在”这种状态，“空”串只意味着字符串的长度为0。在这里没有NULL的等价物，我们不能用它作为文件的结束信号。所以，Csv::getline中采用的方式是通过一个引用参数返回输入行，用函数返回值处理文件结束和错误报告。

```
// getline: get one line, grow as needed
int Csv::getline(string& str)
{
    char c;

    for (line = ""; fin.get(c) && !endofline(c); )
        line += c;
    split();
    str = line;
    return !fin.eof();
}
```

运算符+=已被重载，其作用是在字符串的后面加一个字符。

函数endofline需要做一点改造。在这里程序也需要一个一个地读入字符，因为不存在能处理输入中各种变化情况的标准输入函数。

```
// endofline: check for and consume \r, \n, \r\n, or EOF
int Csv::endofline(char c)
{
    int eol;

    eol = (c=='\r' || c=='\n');
    if (c == '\r') {
        fin.get(c);
        if (!fin.eof() && c != '\n')
            fin.putback(c); // read too far
    }
    return eol;
}
```

这里是新版本的split：

```
// split: split line into fields
int Csv::split()
{
    string fld;
    int i, j;

    nfield = 0;
    if (line.length() == 0)
        return 0;
    i = 0;

    do {
        if (i < line.length() && line[i] == '"')
            j = advquoted(line, fld, ++i);  // skip quote
        else
            j = advplain(line, fld, i);
        if (nfield >= field.size())
            field.push_back(fld);
        else
            field[nfield] = fld;
        nfield++;
        i = j + 1;
    } while (j < line.length());
```

China-pub.com
下载


```
    return nfield;
}
```

由于对 C++ 的串不能用 strcspn，我们必须改造 split和advquoted。新的advquoted用C++标准函数find_first_of确定分隔符的下一个出现位置。函数调用s.find_first_of(fieldsep,j)由字符串s的第j个位置开始查找，检查fieldsep里任何字符的第一个出现。如果无法找到这种位置，该函数将返回串尾后面一个位置的指标，在最后还必须把它改回范围之内。随后的一个内层 for循环把字符逐个附到在fld里积累的域后面，直到分隔符处为止。

```
// advquoted: quoted field; return index of next separator
int Csv::advquoted(const string& s, string& fld, int i)
{
    int j;

    fld = "";
    for (j = i; j < s.length(); j++) {
        if (s[j] == '"' && s[++j] != '"') {
            int k = s.find_first_of(fieldsep, j);
            if (k > s.length()) // no separator found
                k = s.length();
            for (k -= j; k-- > 0; )
                fld += s[j++];
            break;
        }
        fld += s[j];
    }
    return j;
}
```

函数find_first_of也被用在新的advplain函数里，这个函数扫过一个简单的无引号数据域。之所以需要做这种改动，其原因和前面一样，因为 strcspn无法用在C++串上，这是一种完全不同的数据类型。

```
// advplain: unquoted field; return index of next separator
int Csv::advplain(const string& s, string& fld, int i)
{
    int j;

    j = s.find_first_of(fieldsep, i); // look for separator
    if (j > s.length())               // none found
        j = s.length();
    fld = string(s, i, j-i);
    return j;
}
```

函数Csv::getfield仍然非常简单，它可以直接写在类的定义里。

```
// getfield: return n-th field
string Csv::getfield(int n)
{
    if (n < 0 || n >= nfield)
        return "";
    else
        return field[n];
}
```

现在的测试程序和前面的差不多，只有很少的改动：

China-pub.com 下载

```
// Csvtest main: test Csv class
int main(void)
{
    string line;
    Csv csv;

    while (csv.getline(line) != 0) {
        cout << "line = '" << line <<"'\n";
        for (int i = 0; i < csv.getnfield(); i++)
            cout << "field[" << i << "] = '"
                 << csv.getfield(i) << "'\n";
    }
    return 0;
}
```

函数的使用方式与C版本有微小的不同。对一个包括30 000行，每行25个域的大输入文件，根据编译程序的不同，这个C++版本比前面的C版本慢40%到4倍。正如我们对markov程序做比较时看到的，实际上，这种变化情况主要反映出标准库本身的不成熟。C++的源程序大约短20%左右。

练习4-5 改造C++的实现，重载下标操作operator[]，使域访问可以用csv[i]形式完成。

练习4-6 写一个Java版本的CSV库，然后从清晰性、坚固性和速度方面比较这三个实现。

练习4-7 把C++版本的CSV代码重新做成一个STL的迭代器。

练习4-8 C++版本允许建立多个独立的csv实例，它们可以平行地操作而不会互相干扰。这是把一个对象的所有状态封装起来，允许多次实例化带来的收获。设法修改C版本以得到同样效果，采用的方法是把全局数据结构都装进一个结构，并用一个显式的csvnew函数做这种结构分配和初始化。

4.5 界面原则

在前面几节里，我们一直在努力做好界面的各方面细节，使它能成为在提供服务的代码和使用服务的代码间的一条清晰的分界线。界面定义了某个代码体为其用户提供的各种东西，定义了哪些功能或者数据元素可以为程序的其他部分使用。我们的CSV界面提供了三个函数：读一行、取得一个域以及得到域的个数，这些就是能使用的全部操作。

一个界面要想成功，它就必须特别适合有关的工作——必须简单、通用、规范、其行为可以预料及坚固等等，它还必须能很好地适应用户或者实现方式的变化。好的界面总是遵循着一组原则，这些原则不是互相独立的，互相之间甚至可能并不很协调，但它们能帮助我们刻画那些位于界限两边的两部分软件之间的问题。

隐藏实现细节。对于程序的其他部分而言，界面后面的实现应该是隐藏的，这样才能使它的修改不影响或破坏别的东西。人们用了许多术语来描述这种组织原则：信息隐蔽、封装、抽象和模块化，它们谈论的都是类似的思想。一个界面应该隐藏那些与界面的客户或者用户无关的实现细节。这些看不到的细节可以在不影响客户的情况下做修改。例如对界面进行扩充，提高执行效率，甚至把它的实现完全换掉。

各种各样程序设计语言的基本库是大家熟悉的例子，虽然它们并不是都设计得很好的。C语言的标准I/O库是其中最好的：几十个函数可以执行打开、关闭文件，对文件做读、写以及其他操作。文件I/O的所有实现细节都隐蔽在一个数据类型FILE*的后面，该类型的实现细节通常可以看到(因为它们通常直接写在<stdio.h>里面)，但却绝不应该使用。

如果在头文件里不包含实际结构的声明，而只有结构的名字，这种情况被称作是模糊类型，因为其特性本身无法看到，所有操作都通过指针方式进行，实际的数据对象则潜藏在指针后面。

应该避免全局变量。如果可能，最好是把所有需要引用的数据都通过函数参数传递给函数。

我们强烈反对任何形式的公有可见的数据，因为如果用户同样可以改变这些变量，要维护值的一致性就太困难了。通过函数界面，可以很容易提出要求遵守的访问规则。但是人们常常违反这个设计原则。例如，预定义的 I/O流，如sdtin和stdout，基本上都被定义为一个全局的FILE结构数组的元素：

```
extern FILE    __iob[_NFILE];
#define stdin  (&__iob[0])
#define stdout (&__iob[1])
#define stderr (&__iob[2])
```

这就使功能的实现完全暴露出来。这种方式也意味着人们无法给 sdtin、stdout或stderr重新赋值，虽然它们看起来像变量。这里写的特殊名字 —— __iob使用了ANSI C的惯例：采用由连续两个下划线开头的名字为必须暴露的私有对象命名，这种形式的名字不大可能与程序用的其他名字冲突。

C++和Java的类提供了更好的信息隐藏机制，这也是正确使用这些语言的核心。我们在第3章用过的标准模板库把这种想法更向前推进了一步，除了某些执行性能方面的保证之外，这里没有任何关于实现的信息，而库的构造者则可以使用自己喜欢的任何技术。

选择一小组正交的基本操作。一个界面应该提供外界所需要的全部功能，但是绝不要更多；函数在功能方面不应该有过度的重叠。虽然提供大量函数可能使库变得更容易使用，你需要什么就有什么。但是大的界面既难写又难维护，太大的规模也使它难以学习、难以被用好。许多“应用程序界面”(Application Program Interface)或称API有时是如此的庞大，以至没人能有指望完全把握它。

为使用方便，有些界面为做同一件事提供了多种方式，这种冗余其实是应该反对的。 C语言标准I/O库提供了至少四个不同函数，它们都能把一个字符写进一个输出流：

```
char c;
putc(c, fp);
fputc(c, fp);
fprintf(fp, "%c", c);
fwrite(&c, sizeof(char), 1, fp);
```

如果输出流是stdout，我们还可以找到另外一些方式。这样做确实很方便，但是却完全没有必要。

一般地说，窄的界面比宽的界面更受人欢迎，至少是在有了强有力的证据，说明确实需要给界面增加一些新功能之前。我们应该做一件事并且把它做好。不要因为一个界面可能做某些事就给它增加这些东西。如果是实现方面出了毛病那么就不要去修改界面。再比如，我们不应该为了速度就使用 memcpy，为了安全又去用 memmove，最好是始终用一个总是安全的，而又比较快的函数。

不要在用户背后做小动作。一个库函数不应该写某个秘密文件、修改某个秘密变量，或者改变某些全局性数据，在改变其调用者的数据时也要特别谨慎。函数 strtok在这方面有几个毛

China-pub.com 下载

病：它在输入串里面写进空字节，这常常使人感到一点意外；它用空指针作为信号，回到上次离开的位置，这意味着它在不同调用之间保留了一些秘密数据，这个情况很可能成为程序错误的原因，也排除了并行使用这个函数的可能性。一种更好的设计是提供一个独立函数从输入字符串提取标识符。由于类似的原因，我们库的第二个 C版本也不能同时用于两个流 (参看练习4-8)。

一个界面在使用时不应该强求另外的东西，如果这样做仅仅是为了设计者或实现者的某些方便。相反，我们应该使界面成为自给自足的。如果确实无法做到这一点，那么也应该把需要哪些外部服务的问题做成明显的。不这样做就会给用户增加很大的负担。这方面有一个非常明显的例子，那就是在 C和C++ 源程序中经常需要费力地管理长长的头文件列表。有些头文件可能有数千行长，它还可能包含几十个其他头文件。

在各处都用同样方式做同样的事。一致性和规范性是非常重要的。相关的事物应该具有相关的意义。例如，C函数库里的str...函数没有文档也很容易使用，因为它们的行为具有一致性：数据总是从右向左流动，与赋值语句的流向相同；它们总是返回结果串。而另一方面，在C标准I/O库里，要预期函数的参数顺序就非常困难。有些函数把 FILE*作为第一个参数，有些则放在最后；另一些函数的元素大小参数和个数参数又有各种不同顺序。 STL容器类的算法提供了非常一致的界面，这样，即使面对一个不熟悉的函数，预计应该如何使用它也会变得很容易。

外部一致性，与其他东西的行为类似也是非常重要的。例如， C函数库里的mem...函数是在str...函数之后设计的，完全借用了它们前驱的风格。如果标准 I/O函数fread和fwrite看起来更像read和write(这是它们的由来)一些，它们也会更容易记忆。Unix系统的命令行参数都由一个负号引导，但是同一个选项字母可能意味着完全不同的东西，甚至对一些相互有关的程序也常常是这样。

如果*这样的通配符(例如在*.exe里)都由命令解释器展开，其行为是一致的。但如果它们是由独立程序展开的，多半就会出现不一致的行为。 Web浏览器对一次鼠标点击的反应是跟踪一个链接，有些应用程序遇到两次鼠标点击才执行一个程序或跟踪一个链接。这种情况造成的后果是许多人每次都自动地点击两下。

要遵循这些原则，在有些环境里常常比在另一些环境里更容易。但是，无论如何，这些原则都是有效的。比如，在C语言里要隐藏实现细节就非常困难，但是，好的程序员就不会去探察这种细节，因为这种做法将牵连上界面的某些细节部分，违反了信息隐蔽原则。头文件里的注释，特殊形式的名字 (例如__iob)，以及其他一些类似东西，都是为了在不能强制规定的情况下，鼓励好行为而采用的方式。

当然，不管我们今天在设计界面时做得如何好，事情也总有一个限度。今天最好的界面最终也会变成明天的问题。但是，一个好的界面设计将把这个明天推到更远的将来。

4.6 资源管理

在设计库(或者类、包)的界面时，一个最困难的问题就是管理某些资源，这些资源是库所拥有的，而又在库和它的调用程序之间共享。这里最明显的资源是存储 —— 谁负责存储的分配和释放？其他的共享资源包括那些打开的文件以及共同关心的变量状态等。粗略地说，有关的问题大致涉及初始化、状态维护、共享和复制以及清除等等。

China-pub.com
下载

在我们的CSV原型中，采用静态初始化方式为各种指针、计数器等设置初始值。这种方式的功能很有限，因为一旦函数被调用过之后，就无法使这部分程序再从它们的初始状态重新开始。另一种方式是提供一个初始化函数，为所有内部变量正确设置初始值。这种方式能允许重新启动，但要依靠用户显式地对这个函数做调用。我们可以把第二个版本里的 reset函数提供出来，服务于这个目的。

在C++和Java语言里，建构函数专用于给类的数据成分做初始化，设计正确的建构函数能保证所有数据成员都做了初始化，不可能建立未初始化的类对象。一组建构函数能支持各种各样的初始化方式。例如，我们可以为 Csv提供一个以文件名为参数的建构函数，另一个函数则以输入流作为参数，如此等等。

如果需要拷贝由库管理的数据，例如输入行或数据域，那么又会怎么样？在我们 C语言版本的csvgetline程序里，是通过返回指针的方式提供了对输入串 (行和数据域)的直接访问。这种不加限制的访问方式有几个缺点：它使用户有可能复写有关的存储区，以至可能使其他信息变成非法的。例如用户写了下面的表达式：

```
strcpy(csvfield(1), csvfield(2));
```

就可能造成多种不同后果。最可能的是复写掉第二个数据域的开头 (当第二个域比第一个长的时候)。此外，库用户如果想把任何信息保留到下一次 csvgetline调用之后，就必须自己做一个拷贝。在下面的例子里，有关的指针值在第二次 csvgetline调用后就可能变成非法的，因为这个调用有可能导致行缓冲区存储的重新分配。

```
char *p;

csvgetline(fin);
p = csvfield(1);
csvgetline(fin);
/* p could be invalid here */
```

C++版本要安全得多，由于在那里的串都是拷贝，所以可以随意修改。

Java采用的是对象引用，对所有基本类型 (如int)之外的东西都采用这种做法。这种做法比建立拷贝更有效，但是也很容易使人误解，误把引用当作拷贝。在用 Java写markov程序时，我们在一个早期版本里就犯过这样一个的错误。另外，C语言里与字符串有关的错误层出不穷，其根源也在这个地方。Java语言的克隆方法使人可以在需要时建立拷贝。

初始化和建构的另一面是终止和析构 —— 当某些实体不再需要时，就必须对它们进行清理并回收有关资源。对于存储而言这件事特别重要，因为如果一个程序不能把不再使用的存储收回，最终它就会把存储用光。许多很时髦的软件都存在这种毛病，实在令人羞愧。类似问题还出现在打开的文件需要关闭时，如果数据被做了缓冲，在关闭文件时应该做缓冲区的刷新(以及存储回收)。C语言的标准库函数在程序正常终止时将会自动做刷新，其他情况下我们就需要自己做了。C和C++ 的标准函数atexit提供了一个在程序正常终止前取得控制的方法，界面的实现者可以利用这个机制安排一些清理工作。

释放资源与分配资源应该在同一个层次进行。控制资源分配和回收有一种基本方式，那就是令完成资源分配的同一个库、程序包或界面也负责完成它的释放工作。这种处理原则的另一种说法是：资源的分配状态在跨过界面时不应该改变。例如，我们的 CSV库从一个已经打开的文件读数据，那么它在完成工作时还应该使文件保持在打开状态，由库的调用程序关闭文件。

China-pub.com 下载

C++的建构函数和析构函数对实施这个规则很有帮助。当一个类实例离开了作用域或者被显式地清除时，析构函数就会被自动调用，它可以刷新缓冲区、恢复存储、重新设置值以及完成其他一些应该做的事情。Java没有提供与此等价的机制，虽然可以为类定义一个终结方法，但是却不能保证它一定会执行，更不用说要求在某个特定时刻做了。因此，在这里不能保证清理操作一定会发生，虽然通常人们都合理地假定它们会。

Java也确实对存储管理提供了很大的帮助，它有一个内部的废料收集系统。Java程序在运行时必定要分配存储，但是它没办法显式地释放存储。运行系统跟踪所有在用对象和无用对象，周期性地把不再有用的对象收回到可用存储池里。

废料收集有许多不同的技术。有一类模式是维持着对每个对象的使用次数，即它们的引用计数值，一旦某个对象的引用计数值变成0时就释放它。这种技术可以显式地应用到C或C++里，用来管理共享的对象。另一种算法是周期性地工作，按照某种痕迹跟踪分配池到所有被引用的对象。所有能够以这种方式找到的对象都是正在被使用的，未被引用的对象是无用的，可以回收。

自动废料收集机制的存在并不意味着设计中不再有存储管理问题了。我们仍然需要确定界面是应该返回对共享对象的引用呢，还是返回它们的拷贝，而这个决定将影响到整个程序。此外，废料收集也不是白来的——这里有维护信息和释放无用存储的代价，此外，这种收集动作发生的时间也是无法预期的。

如果一个库将被使用在有多个控制线程的环境里，其函数可能同时存在多个调用，例如用在多线程的Java程序里，上述所有问题都将变得更复杂。

为了避免出问题，我们必须把代码写成可重入的，也就是说，无论存在多少个同时的执行，它都应该能正常工作。可重入代码要求避免使用全局变量、静态局部变量以及其他可能在别的线程里改变的变量。对于好的多线程设计，最关键的就是做好各部件之间的隔离，使它们除了经过良好设计的界面外不再共享任何东西。那些随意地把变量暴露出来共享的库就破坏了这个模型(对于多线程的程序而言，strtok纯粹是个灾星，C函数库里其他采用在内部静态变量里存储值的函数也一样)。如果存在必需的共享变量，它们就必须放在某种锁定机制的保护之下，保证在每个时刻只有一个线程访问它们。类在这里有极大的帮助作用，因为它们能够成为共享和锁定模型的中心。Java的同步方法就是这类东西，它使线程可以锁定整个的类或者类的实例，防止其他线程同时进行修改。同步程序块在每个时刻只允许一个线程执行这个代码段。

多线程给程序设计增加了许多的新问题，这是一个太大的题目，我们无法在这里讨论更多的细节。

4.7 终止、重试或失败

在前面的章节里，我们用到了eprintf和estrdup等函数，用于处理结束程序执行之前的错误信息显示问题。例如，eprintf的功能与fprintf(stderr, .类似，它在出错状态下报告了错误后就结束程序。定义这个函数需要用 <stdarg.h>头文件，通过库函数vprintf打印原型中 ...表示的那些参数。使用 stdarg标准库的功能，必须先调用va_start做初始化，最后调用va_end做终止处理。在第9章里我们还要更多地使用这个界面。

China-pub.com
下载


```
#include <stdarg.h>
#include <string.h>
#include <errno.h>

/* eprintf: print error message and exit */
void eprintf(char *fmt, ...)
{
    va_list args;

    fflush(stdout);
    if (progname() != NULL)
        fprintf(stderr, "%s: ", progname());

    va_start(args, fmt);
    vfprintf(stderr, fmt, args);
    va_end(args);

    if (fmt[0] != '\0' && fmt[strlen(fmt)-1] == ':')
        fprintf(stderr, " %s", strerror(errno));
    fprintf(stderr, "\n");
    exit(2); /* conventional value for failed execution */
}
```

如果格式参数由一个冒号结束， eprintf就调用标准C函数strerror，该函数返回一个附加的系统错误信息串(如果存在的话)。我们也定义了weprintf，它的功能与eprintf类似，显示一个警告信息，但却不结束程序的执行。利用这种与 printf类似的界面构造字符串非常方便，然后可以输出或者显示在对话框里。

与此类似，函数estrdup试图构造串的拷贝，如果存储用光了，它就通过 eprintf输出一个错误信息并结束程序：

```
/* estrdup: duplicate a string, report if error */
char *estrdup(char *s)
{
    char *t;

    t = (char *) malloc(strlen(s)+1);
    if (t == NULL)
        eprintf("estrdup(\"%.20s\") failed:", s);
    strcpy(t, s);
    return t;
}
```

函数emalloc提供与调用malloc时类似的服务：

```
/* emalloc: malloc and report if error */
void *emalloc(size_t n)
{
    void *p;

    p = malloc(n);
    if (p == NULL)
        eprintf("malloc of %u bytes failed:", n);
    return p;
}
```

头文件eprintf.h里声明了这些函数：

```
/* eprintf.h: error wrapper functions */
extern  void    eprintf(char *, ...);
extern  void    weprintf(char *, ...);
extern  char    *estrdup(char *);
```

China-pub.com 下载

```
extern  void    *emalloc(size_t);
extern  void    *erealloc(void *, size_t);
extern  char    *progname(void);
extern  void    setprogname(char *);
```

这个头文件应该被包含到任何需要使用这些错误处理函数的文件里。输出的每个错误信息串中还可以包含一个程序名，这需要在调用程序里事先进行设置。完成这件事的是两个非常简单的函数setprogname和progname。它们也在头文件里声明，与eprintf在同一个源文件里定义：

```
static char *name = NULL;  /* program name for messages */

/* setprogname: set stored name of program */
void setprogname(char *str)
{
    name = estrdup(str);
}

/* progname: return stored name of program */
char *progname(void)
{
    return name;
}
```

这些函数的典型使用方式是：

```
int main(int argc, char *argv[])
{
    setprogname("markov");
    ...
    f = fopen(argv[i], "r");
    if (f == NULL)
        eprintf("can't open %s:", argv[i]);
    ...
}
```

输出的信息是这样的：

```
markov: can't open psalm.txt: No such file or directory
```

我们觉得这些包装函数给程序设计带来很大方便，因为它们能使错误得到统一处理，它们的存在也促使我们尽量去捕捉错误而不是简单地忽略错误。在我们的这种设计里并没有什么特殊的东西，但你在写自己的程序时也可能喜欢采用其他方式。

假定我们写的函数不是为了自己用，而是为别人写程序提供一个库。那么，如果库里的一个函数发现了某种不可恢复性的错误，它又该怎么办呢？在本章前面写的函数里，采取的做法是显示一段信息并令程序结束。这种方式对很多程序是可以接受的，特别是对那些小的独立工具或应用程序。而对另一些程序，终止就可能是错误的，因为这将完全排除程序其他部分进行恢复的可能性。例如，一个字处理系统必须由错误中恢复出来，这样它才能不丢掉你键入的文档内容。在某些情况下库函数甚至不应该显示信息，因为本程序段可能运行在某种特定的环境里，在这里显示信息有可能干扰其他显示数据，这种信息也可能被直接丢掉，没留下任何痕迹。在这种环境里，一种常用的方式是输出诊断情况，写入一个显式的记录文件，这个文件可以独立地进行监控和检查。

在低层检查错误，在高层处理它们。 作为一条具有普遍意义的规则，错误应该在尽可能低的层次上检测和发现，但应该在某个高一些的层次上处理。一般情况下，应该由调用程序决定

China-pub.com 下载

对错误的处理方式，而不该由被调用程序决定。库函数应该以某种得体的失败方式在这方面起作用。在前面我们让函数对不存在的域返回 NULL值，而不是立即退出执行，也就是由于这个原因。与此类似， csvgetline在遇到文件结束之后，无论再被调用多少次，也总是返回NULL值。

合适的返回值未必都是非常明显的，我们在前面讨论 csvgetline应该返回什么时就遇到了这种情况。我们总希望能尽量返回某些有用信息，而且应该是以程序其他部分最容易使用的形式。在C、C++和Java里，这就意味着要求以函数值的方式返回某种东西，与此同时，其他的值就只能通过引用 (或者指针)参数返回。许多库函数都区分了正常的值和错误值。像getchar一类的输入函数对正常数据返回一个 char数据，而对错误或者文件结束则返回某种非char值，例如EOF等。

如果函数的合法返回值能取遍所有可能的返回值，上面这种方式就行不通了。例如一个数学函数，如log，它本来就可能返回任何浮点数值。在 IEEE标准的浮点数里，有一个特殊值称为NaN(“不是数”，not a number)，就是专用于指明错误的，可以作为返回错误信号的值。

有些语言，例如Perl和Tcl等，提供了把两个或多个值组合起来形成元组的简便方法。对这类语言，我们很容易把函数值和可能的错误状态信息一起返回。 C++ 的STL提供了一种pair数据类型，也可以用在这个地方。

如果能把各种各样的异常值(如文件结束、可能的错误状态)进一步区分开，而不是用单个返回值把它们堆在一起，那当然就更好了。如果无法立刻区分出这些值，另一个可能的方法是返回一个单一的“异常”值，但是另外提供一个函数，它能返回关于最近出现的错误的更详细信息。

这也是在Unix和C标准库里采用的方法。在那里许多系统调用或库函数都在返回- 1(表示错误)的同时给名为errno的变量设一个针对特定错误的编码值，随后用 strerror就可以返回与各个错误编码关联的字符串。在我们使用的系统里，程序：

```
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <math.h>

/* errno main: test errno */
int main(void)
{
    double f;

    errno = 0;  /* clear error state */
    f = log(-1.23);
    printf("%f %d %s\n", f, errno, strerror(errno));
    return 0;
}
```

输出：

```
nan0x10000000 33 Domain error
```

如上所示， errno必须事先予以清除。此后，如果发生错误， errno就会被置上非0值。

只把异常用在异常的情况。有些语言提供了异常机制，以帮助捕捉不正常状态并设法做恢复，这种机制实际上是提供了在某些坏事情发生时的另一条控制流。异常机制不应该用于处理可

China-pub.com 下载

预期的返回值。读一个文件最终总要遇到文件结束，这个情况就应该以返回值的方式处理，而不是通过异常机制。

在Java里可以写：

```
String fname = "someFileName";
try {
    FileInputStream in = new FileInputStream(fname);
    int c;
    while ((c = in.read()) != -1)
        System.out.print((char) c);
    in.close();
} catch (FileNotFoundException e) {
    System.err.println(fname + " not found");
} catch (IOException e) {
    System.err.println("IOException: " + e);
    e.printStackTrace();
}
```

这里的循环读字符，一直读到文件结束。这是一个可预期的事件，由 read的返回值为-1表示。如果文件打不开，就会引发一个异常，而不是把输入流设置为 null，后者是C语言或C++ 中的常用做法。如果在try块执行中出现了其他I/O错误，也会引发异常，这个异常将被最下面的IOException处理段捕获。

异常机制常常被人过度使用。由于异常是对控制流的一种旁路，它们可能使结构变得非常复杂，以至成为错误的根源。文件无法打开很难说是什么异常，在这种情况下产生一个异常有点过分。异常最好是保留给那些真正无法预期的事件，例如文件系统满或者浮点错误等等。

对于C程序，存在着一对库函数setjmp和longjmp，它们提供了一种很低层的服务，借助它们可以实现一套异常处理机制。不过这些机制太神秘，我们不打算再进一步讨论它们。

在发生错误时应该如何恢复有关的资源？如果发生了错误，库函数应该设法做这种恢复吗？通常它们不做这些事，但也可以在这方面提供一些帮助：提供尽可能清楚的信息和以尽可能无害的方式退出。当然，不再使用的存储应该释放，如果有些变量还可能被使用，那么就应该把它们设置成某种明显的值。有一种错误很常见，那就是通过指针使用那些已经释放的存储。如果错误处理代码在释放存储之后把指向它们的指针都置为空，这种错误就一定能发现。CSV库的第二个版本里的reset函数就考虑了这些问题。一般说来，我们的目标应该是保证在发生了错误之后，库仍然是可用的。

4.8 用户界面

至此我们谈论的主要是一个程序里不同部件之间的界面，或者是不同程序之间的界面问题。实际中还有另一类，也是非常重要的界面，那就是程序与作为程序使用者的人之间的界面。

本书里大部分程序例子都是基于文本方式的，因此它们的用户界面非常简单。正如我们在前一节谈到的，应该对各种错误进行检查并报告，还应该在有意义的情况下设法恢复到某种常态。错误信息输出应该包含所有可用信息，在可能情况下应给出有意义的上下文信息。一个诊断信息不应该这样简单：

```
estrdup failed
```

China-pub.com
下载

如果它应该提供的信息是：

```
markov: estrdup("Derrida") failed: Memory limit reached
```

像我们在estrdup里所做的那样并没有多费什么事，但却能帮助用户发现问题，或者告诉他们如何提供合法的输入。

程序在使用方式出现错误时应该显示有关正确方式的信息，就像下面的函数所做的：

```
/* usage: print usage message and exit */
void usage(void)
{
    fprintf(stderr, "usage: %s [-d] [-n nwords]"
        " [-s seed] [files ...]\n", progname());
    exit(2);
}
```

程序名标识了信息的来源，尤其是当它实际上是一个更大过程的一部分时，这个信息更是特别重要。如果一个程序提供的错误信息仅仅是说 syntax error(语法错)或者estrdup failed(estrdup失败)，用户将根本无法知道是谁在说话。

错误信息、提示符或对话框中的文本应该对合法输入给出说明。不要简单地说一个参数太大，而应说明参数值的合法范围。如果可能的话，给出的这段文本本身最好就是一段合法的输入，比如提供一个带合适参数的完整命令行。除了指导用户如何正确使用外，这种输出还应该能导入到一个文件里，或者通过鼠标器选择拷贝，以便用于运行某个另外的程序。由此也可以看到对话框的一个弱点：它们的内容通常是不能使用的。

建立有效用户界面的另一种途径是设计一个为设置参数、控制各种操作等等而用的特殊语言。一种好的记法常常能使程序的使用更加简单，也能够帮人组织程序的实现。以语言为基础的界面问题是第9章要讨论的题目。

防御性程序设计，或者说是保证程序在遇到坏的输入时本身不会受到损害，无论是从防止用户破坏的角度看，还是从安全性的角度看都是非常重要的。第 6章还要更多地讨论这个问题，那里将讨论程序测试问题。

对于大部分人而言，图形用户界面就是他们计算机的用户界面。图形用户界面是一个巨大的题目，对此我们将只想说一点与本书切题的内容。首先，图形界面很难做“正确”，因为它们的适用性或者成功非常强烈地依赖于个人的行为和期望。其次，作为另一个实际情况，如果一个系统有一个用户界面，通常这个程序里处理用户交互的代码将比实现所完成工作的算法的代码更多。

无论如何，我们熟悉的各种原则也适用于用户界面软件的外部设计及其内部实现。从用户的观点看，风格问题，如简单性、清晰性、规范性、统一性、熟悉性和严谨性等，对于保证一个界面容易使用都是非常重要的，不具有这些性质的界面必定是令人讨厌的难对付的界面。

统一和规范是非常必要的，这方面的问题包括术语、单位、格式、排布方式、字体、颜色、大小，以及其他一切图形系统可能选择的方面，在使用上都应该有一致性。对退出一个程序或者关闭一个窗口，有多少可以使用的英文词？从“Abandon”(放弃)到“control-Z”，这种词至少有一打之多。这种不一致把说英语的人都搞糊涂了，更不用说是其他人了。

对图形程序而言，界面更加重要，因为这些系统都很大，很复杂，采用的是与文本顺序扫描很不相同的方式驱动的。对实现图形用户界面而言，面向对象的程序设计远胜于其他方

China-pub.com 下载

法，因为它提供了一种途径，能够封装各种窗口的行为和状态细节，通过继承来取得基类的相似性，通过导出类区分出相互的差异。

补充阅读

Frederick P. Brooks, Jr. 的《神秘的人月》(The Mythical Man Month，Addison-Wesley，1975，1995的周年版)包含了许多关于软件开发的真知灼见，虽然其中讨论的有些技术细节已经过时了，但在今天它仍然像当年出版的时候一样值得读一读。

对界面设计而言，几乎所有关于程序设计的书都可能有些用处。其中一本根据从艰难工作中的取胜经验写的实践性书籍是 John Lakos的《大规模C++软件设计》(Large-Scale C++ Software Design，Addison-Wesley，1996)，该书讨论如何构建和管理真正大规模的 C++程序。David Hanson的《C界面和实现》(C Interfaces and Implementation，Addison-Wesley，1997)是一本关于C程序的好书。

Steve McConnell的《快速开发》(Rapid Development，Microsoft Press，1996)是一本极好的关于怎样以软件队(组)方式构造软件的书，其中特别强调了原型的作用。

关于图形用户界面设计有许多有趣的书，它们提出了各种各样的看法。我们推荐 Kevin Mullet和Darrell Sano的《设计可视的界面：基于交流的技术》(Designing Visual Interfaces: Communication Oriented Techniques，Prentice Hall，1995)，Ben Shneiderman的《设计用户界面：有效人机交互的策略》(Designing the User Interface: Strategies for Effective Human-Computer Interaction，第3版，Addison-Wesley，1997)，Alan Cooper的《表面：用户界面设计要素》(About Face: The Essentials of User Interface Design，IDG，1995)，以及Harold Thimbleby的《用户界面设计》(User Interface Design，Addison-Wesley，1990)。

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 268156158598852001534108794260233396350\
    1936585971793218047714963795307788611480564140\
    0796821289594743537151163524101175474084764156\
    422771408323839623430144.000000
```

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。

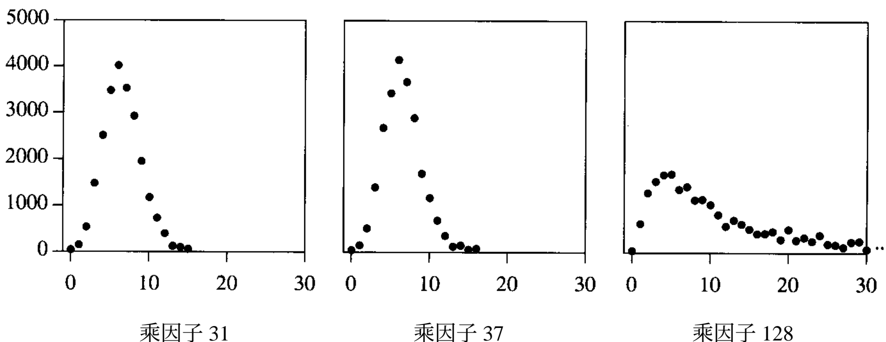
写记录文件。另一种战术是写一个记录文件，以某种固定格式写出一系列的排错输出。当程序垮台的时候，这个文件里已经记录了垮台前发生的情况。Web服务器和其他网络程序都维护着有关数据流动的记录文件，其内容相当广泛，这使它们可以监视自己和客户的活动。下面的片段来自一个本地系统(重新编排过)：

```
[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

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。

China-pub.com
下载

第6章　测　　试

> 在日常的手工或者通过办公机器的计算实践中，人们形成了一种习惯，那就是要检查计算的每个步骤。如果发现了错误，确认它的方式就是从第一次注意到错误的那个地方开始，做反向的处理。
>
> Norbert Wiener(诺伯特·维纳)，《控制论》

测试和排错常常被说成是一个阶段，实际上它们根本不是同一件事。简单地说，排错是在你已经知道程序有问题时要做的事情。而测试则是在你在认为程序能工作的情况下，为设法打败它而进行的一整套确定的系统化的试验。

Edsger Dijkstra有一个非常有名的说法：测试能够说明程序中有错误，但却不能说明其中没有错误。他的希望是，程序可以通过某种构造过程正确地做出来，这样就不再会有错误了，因此测试也就不必要了。这确实是个美好的目标，但是，对今天的实际程序而言，这仍然还只是一个理想。所以在这一章里，我们还是要集中精力讨论如何测试，才能够更快地发现程序错误，工作更有成效，效率也更高。

仔细思考代码中可能的潜在问题是个很好的开端。系统化地进行测试，从简单测试到详尽测试，能帮我们保证程序在一开始就能正确工作，在发展过程中仍然是正确的。自动化能帮助我们减少手工操作，鼓励人们做更广泛的测试。在这方面也存在大量技巧，是设计程序的人们从实践中学到的。

产生无错误代码的一个途径是用程序来生成代码。如果某些程序的工作已经彻底理解清楚了，写代码的方式很机械，那么就应该把它机械化。用某种特定语言描述后，程序可以自动生成出来，这种情况也是很常见的。典型的例子如：我们把高级语言的程序编译成汇编代码；使用正则表达式来描述正文的模式；我们使用 `SUM(A1:A50)`一类的记法，表示的是在电子表格系统中一系列单元上的运算。对这些情况，如果所用的生成器或者翻译器是正确的，给出的描述完全正确，那么作为结果的程序必然也是正确的。我们在第 9章将对这个内涵丰富的领域做更细致的讨论。在本章中，我们将只对通过某些紧凑描述来生成测试实例的方法做一点简单讨论。

6.1　在编码过程中测试

问题当然是发现得越早越好。如果你在写代码时就系统地考虑了应该写什么，那么也可以在程序构造过程中验证它的简单性质。这样做的结果是，甚至代码还没有经过编译，就已经经过了一轮测试。这样，有些错误类根本就不会出现了。

测试代码的边界情况。一项重要技术是边界条件测试：在写好一个小的代码片段，例如一个循环或一个条件分支语句之后，就应该检查条件所导致的分支是否正确，循环实际执行的次数是否正确等。这种工作称为边界条件测试，因为你的检查是在程序和数据的自然边界上。例如，

China-pub.com
下载

应该检查不存在的或者空的输入、单个的输入数据项、一个正好填满了的数组，如此等等。这里要强调的观点是：大部分错误都出现在边界上。如果一段代码出毛病，毛病最可能是出在边界上。相反，如果它在边界上都工作得很好，一般来说在别的地方也能够这样。

下面的片段模拟fgets，它读入一些字符，直到遇到一个换行或者缓冲区满了：

```
?     int i;
?     char s[MAX];
?
?     for (i = 0; (s[i] = getchar()) != '\n' && i < MAX-1; ++i)
?         ;
?     s[--i] = '\0';
```

设想你刚刚写好这个循环，现在要在你的头脑里模拟它读入一行的过程。第一个边界测试很简单，读入一个空行，如果一个行的开始就是个换行符。我们很容易看到循环在第一次重复执行时就结束了，这时 i被设置为0。随后，最后一行代码将 i的值减小到-1，并把一个空字节写入s[-1]，这位于数组的开始位置之前。通过边界条件测试，错误被发现了。

如果我们重写这个循环，采用普通的用输入字符填充数组的惯用写法，它的形式应该像下面这样：

```
?     for (i = 0; i < MAX-1; i++)
?         if ((s[i] = getchar()) == '\n')
?             break;
?     s[i] = '\0';
```

重复前面的边界测试，对只有一个换行字符的行，很容易验证程序能正确处理： i是0，第一个输入字符跳出循环，而'\0'被存入s[0]中。对于在一个或两个字符之后是换行符的输入做类似检查，使我们相信在接近边界的地方这个循环也能工作。

当然，还有另一些边界条件需要检查。如果输入中有一个很长的行，或者其中没有换行符，保证i总是小于MAX-1的检测能处理这个问题。但如果输入为空那又会怎么样，这时对getchar的第一次调用就返回了EOF。我们必须增加新条件，检查这种情况：

```
?     for (i = 0; i < MAX-1; i++)
?         if ((s[i] = getchar()) == '\n' || s[i] == EOF)
?             break;
?     s[i] = '\0';
```

边界检查可以捕捉到许多错误，但是未必是所有的错误。第 8章我们还要回到这个例子，在那里要说明，在这段代码里实际上还有一个移植性错误。

下一步应该是在另一端的边界上检查输入：检查数组接近满了、正好满了或者超过了的情况，特别是如果换行字符正好在这个时候出现。我们不准备在这里写出所有的细节，不过，这个例子是一个很好的训练。对于边界条件的思考将会提出一个问题：如果在 '\n'出现前缓冲区已满，程序应该做些什么？这是早在规范描述里就应该解决的问题，而测试边界条件能帮助我们识别它。

边界条件检查对发现"超出一个"的错误特别有效。在实践中，做这种检查已经变成了许多人的习惯。这样，大量的小错误在它们还没有真正发生前就已经被清除了。

测试前条件和后条件。防止问题发生的另一个方法，是验证在某段代码执行前所期望的或必须满足的性质(前条件)、执行后的性质(后条件)是否成立。保证输入取值在某个范围之内是前条件测试的一类常见例子，下面的函数计算一个数组里 n个元素的平均值，如果n小于或者等于0，就会有问题：

China-pub.com 下载

```
?     double avg(double a[], int n)
?     {
?         int i;
?         double sum;
?
?         sum = 0.0;
?         for (i = 0; i < n; i++)
?             sum += a[i];
?         return sum / n;
?     }
```

当n是0时avg应该返回什么？一个无元素的数组是个有意义的概念，虽然它的平均值没有意义。avg应该让系统去捕捉除零错误吗？还是终止执行？提出抱怨？或许是默默地返回某个无害的值？如果n是负数又该怎么办？这当然是无意义的但也不是不可能的。正像前面第 4章里建议的，按照我们的习惯，如果n小于等于0时或许最好返回一个0：

```
return n <= 0 ? 0.0 : sum/n;
```

当然，在这里并没有惟一的正确答案。

然而，忽略这种问题则一定是个错误的回答。在 1998年11月的《科学美国人》杂志上有一篇文章，描述了美国导弹巡洋舰约克敦号上的一起事故。一个船员错误地输入了一个 0作为数据值，结果造成了除零错，这个错误窜了出去，最后关闭了军舰的推进系统。约克敦号“死”在海上几个小时，就是因为某个程序没有对输入的合法性进行检查。

使用断言。 C和C++在<assert.h>里提供了一种断言机制，它鼓励给程序加上前 /后条件测试。断言失败将会终止程序，所以这种机制通常是保留给某些特殊情况使用的，写在这里的错误是真正不应该出现的，而且是无法恢复的。我们可能在前面程序段里的循环前面加一个断言：

```
assert(n > 0);
```

如果这个断言被违反，它就会导致程序终止，并给出一个标准的信息：

```
Assertion failed: n > 0, file avgtest.c, line 7
Abort(crash)
```

断言机制对于检验界面性质特别有用，因为它可以使人注意到调用和被调用之间的不一致性，并可以进一步指出麻烦究竟是出在哪里。如果关于 n大于0的断言在函数调用时失败，它实际上就指明了调用程序是造成麻烦的根源，问题不在 avg本身。如果一个界面做了些修改，而我们忘记对那些依赖于它的程序做更新，断言使我们有可能在错误还没有造成实际损害之前把它抓住。

防御性的程序设计。 有一种很有用的技术，那就是在程序里增加一些代码，专门处理所有“不可能”出现的情况，也就是处理那些从逻辑上讲不可能发生，但是或许 (由于其他地方的某些失误)可能出现的情况。前面在avg里加上检查数组长度是否为 0或者负数就是一个例子。作为另一个例子，一个处理学生成绩的程序应该假定不会遇到负数或者非常大的数，但是它却应该检查：

```
if (grade < 0 || grade > 100)   /* can't happen */
    letter = '?';
else if (grade >= 90)
    letter = 'A';
else
    ...
```

China-pub.com
下载

这些都是防御性程序设计的实例：设法使程序在遇到不正确使用或者非法数据时能够保护自己。空指针、下标越界、除零以及其他许多错误都可以提前检查出来，或者是提出警告，或者是使其转向。防御性程序设计(这里并不想作为双关语⊖)应该能很好地捕捉到约克敦号上的除零错误。

检查错误的返回值。一个常被忽略的防御措施是检查库函数或系统调用的返回值。对所有输入函数(例如fread或fscanf)的返回值一定要做检查，看它们是否出错。对文件打开操作(如fopen)也应该这样。如果一个读入操作或者文件打开操作失败，计算将无法正确地进行下去。

检查输出函数(例如fprintf或fwrite)的返回值也可以发现一些错误，例如，要向一个文件写入，而磁盘上已经没有空间了。检查 fclose的返回值也是有道理的，它返回 EOF说明操作过程中出了错，否则就返回0。

```
fp = fopen(outfile, "w");
while (...)                    /* write output to outfile */
    fprintf(fp, ...);
if (fclose(fp) == EOF) {       /* any errors? */
    /* some output error occurred */
}
```

输出错误也可能成为严重问题。如果当时正在写的是一个贵重文件的新版本，这个检查就可能防止你在新文件并没有成功建立的情况下删掉老的文件。

在编程的过程中测试，其花费是最小的，而回报却特别优厚。在写程序过程中考虑测试问题，得到的将是更好的代码，因为在这时你对代码应该做些什么了解得最清楚。如果不这样做，而是一直等到某种东西崩溃了，到那时你可能已经忘记了代码是怎样工作的。即使是在强大的工作压力下，你也还必须重新把它弄清楚，这又要花费许多时间。进一步说，这样做出的更正往往不会那么彻底，可能更脆弱，因为你唤回的理解可能不那么完全。

练习6-1 对下面例子的边界情况做各种检查，然后根据需要按照第 1章的风格和本章的建议对它们进行修改。

(a) 这个函数想计算阶乘：

```
?   int factorial(int n)
?   {
?       int fac;
?       fac = 1;
?       while (n--)
?           fac *= n;
?       return fac;
?   }
```

(b) 这个程序段是要把一个字符串里的字符按一行一个的方式输出：

```
?   i = 0;
?   do {
?       putchar(s[i++]);
?       putchar('\n');
?   } while (s[i] != '\0');
```

(c) 这个函数希望从src复制字符串到dest：

```
?   void strcpy(char *dest, char *src)
?   {
```

⊖ 防御性程序设计(defensive programming)。由于军舰有防御问题，programming有做方案、做规划的意思，这个词组又可以解释为防御计划，防御方案。作者这里说的是俏皮话。——译者

China-pub.com 下载

```
?         int i;
?
?         for (i = 0; src[i] != '\0'; i++)
?             dest[i] = src[i];
?     }
```

(d) 这是另一个字符串复制，要从s拷贝n个字符到t：

```
?     void strncpy(char *t, char *s, int n)
?     {
?         while (n > 0 && *s != '\0') {
?             *t = *s;
?             t++;
?             s++;
?             n--;
?         }
?     }
```

(e) 一个数值比较：

```
?     if (i > j)
?         printf("%d is greater than %d.\n", i, j);
?     else
?         printf("%d is smaller than %d.\n", i, j);
```

(f) 一个字符类检查：

```
?     if (c >= 'A' && c <= 'Z') {
?         if (c <= 'L')
?             cout << "first half of alphabet";
?         else
?             cout << "second half of alphabet";
?     }
```

练习6-2 当我们在1998年末写这本书的时候，2000年问题逐渐显现出来了，这可能是有史以来最大的边界条件问题。

(a) 你应该对哪些日期做检查，以验证一个系统是否可能在2000年里工作？假定测试的执行是非常昂贵的，你在试过2000年1月1日之后，准备以什么顺序继续你的测试？

(b) 你怎样测试标准函数ctime，它返回一个具有下面形式的表示日期的字符串：

```
Fri Dec 31 23:58:27 EST 1999\n\0
```

假定你的程序调用ctime，你怎样写你的代码，使它能够抵御一个有毛病的标准函数实现。

(c) 说明你怎样测试一个以下面形式打印输出日历的程序：

```
    January 2000
 S  M Tu  W Th  F  S
                   1
 2  3  4  5  6  7  8
 9 10 11 12 13 14 15
16 17 18 19 20 21 22
23 24 25 26 27 28 29
30 31
```

(d) 你认为在你所使用的系统里还有哪些时间边界？可能通过怎样的测试弄清楚它们是否能得到正确处理？

6.2 系统化测试

最重要的是应该系统化地对程序进行测试，这样你可以知道每一步测试的是什么，希望

China-pub.com
下载

得到什么结果。你应该有秩序地做这些工作，而没有忽略掉任何东西；应该做一个记录，以便能随时了解已经完成了哪些工作。

以递增方式做测试。测试应该与程序的构造同步进行。与逐步推进的方式相比，以“大爆炸”方式先写出整个程序，然后再一古脑地做测试，这样做困难得多，通常也要花费更长时间。写出程序的一部分并测试它，加上一些代码后再进行测试，如此下去。如果你有了两个程序包，它们已经都写好并经过了测试，把它们连接起来后就应该立即测试，看它们能否在一起工作。

例如，当我们写第4章的CSV程序时，第一步是写出刚好能读输入的那么多代码，这使我们能检查输入处理的正确性。下一步是写出把输入行在逗号处切开的代码。当这些部分都能工作后，我们再转到带引号的域，这样逐步工作下去，直到测试完所有的东西。

首先测试简单的部分。递增方式同样适用于对程序性质的测试。测试应该首先集中在程序中最简单的最经常执行的部分，只有在这些部分能正确工作之后，才应该继续下去。这样，在每个步骤中你使更多的东西经过了测试，对程序基本机制能够正确工作也建立了信心。通过容易进行的测试，发现的是容易处理的错误。在每个测试中做最少的事情去发掘出下一个潜在问题。虽然错误可能是一个比一个更难触发，但是可能并不更难纠正。

本节里我们要谈的是怎样选择有效的测试，以及按什么顺序去执行它们。在随后两节里我们将讨论如何将这种过程机械化，以使得它能高效地进行。测试的第一步，至少对于小程序或独立函数而言，是做一些扩展的边界条件测试(在前面一节里已经讨论过)：系统化地测试各种小的情况。

假设我们有一个函数，它对一个整数数组做二分检索。我们将着手做下面的测试，按复杂性递增的顺序安排：

- 检索一个无元素的数组。
- 检索一个单元素的数组，使用一般的值，它
  - 小于数组里的那个元素；
  - 等于那个元素；
  - 大于那个元素。
- 检索一个两元素的数组，使用一般的值，这时
  - 检查所有的五种可能情况。
- 检索有两个重复元素的数组，使用一般的值，它
  - 小于数组里的值；
  - 等于数组里的值；
  - 大于数组里的值。
- 像检索两个元素的数组那样检索三个元素的数组。
- 像检索两个和三个元素的数组那样检索四个元素的数组。

如果函数能够平安地通过所有这些测试，它很可能已经完美无缺了。但是还需要对它做进一步的测试。

上面这个测试集非常小，完全能以手工方式进行。但还是最好为此建立一个测试台，以使测试过程机械化。下面的驱动程序是我们按尽可能简单的方式写出的，它读入一个输入行，其中包含要检索的关键码和一个数组大小。它建立一个具有指定大小的数组，其中包含值 1，

China-pub.com 下载

3，5，···，并在这个数组里检索指定的关键码。

```
/* bintest main: scaffold for testing binsearch */
int main(void)
{
    int i, key, nelem, arr[1000];

    while (scanf("%d %d", &key, &nelem) != EOF) {
        for (i = 0; i < nelem; i++)
            arr[i] = 2*i + 1;
        printf("%d\n", binsearch(key, arr, nelem));
    }
    return 0;
}
```

这当然是过分简单化了，但它也说明一个有用的测试台不必很大。很容易对这个程序做些扩充，使之可以执行更多的测试，而且需要更少的手工干预。

弄清所期望的输出。对于所有测试，都必须知道正确的答案是什么，如果你不知道，那么你做的就是白白浪费自己的时间。这件事看起来很明显，因为对许多程序而言，了解它们能否工作是很容易的事。例如，一个文件的拷贝或者是个拷贝，或者就不是。由一个排序程序得到的输出或者是排好序的或者不是，当然它还必须是原始输入的一个重排。

但是，也有许多程序的特性更难以说清楚，例如编译程序（其输出是输入的一个正确翻译吗？），数值算法(答案是不是在可容忍的误差范围之内？)，图形系统(所有的像素都在正确的位置上吗？)等等。对于这些系统，通过把输出与已知的值进行比较，用这种方式来验证它们就特别重要。

- 测试一个编译程序，方法应该是编译并运行某些程序文件。这些程序本身应该能产生输出，这样就可以用它们的输出和已知结果进行比较。
- 测试一个数值程序，应该产生那种能揭示算法边界情况的测试实例，既有简单的也有困难的。如果可能的话，写一些代码去验证输出特性的合理性。例如，对数值积分程序的输出可以检查其连续性，验证它的结果与闭形式积分解的相符情况。
- 要测试一个图形程序，仅看它能否画一个方框是不够的。应该从屏幕上把方框读回来，检查其边界是否正好位于它应该在的位置。

如果一个程序有逆计算，那么就检查通过该逆计算能否重新得到输入。例如加密操作和解密操作是互逆的，如果你加密的什么东西不能解密，那么一定有某些东西存在毛病。与此类似，无损的压缩和展开程序之间也是互逆的。把文件捆绑起来的程序也必须能分毫不差地把它们提取出来。有时存在着完成逆操作的多种方式，应该检查它们的各种组合情况。

检验应保持不变的特征。许多程序将保持它们输入的一些特征。有些工具，如 wc(计算行、词和字符数)和sum(计算某种检验和)可用于验证输出是否与输入具有同样大小、词的数目是否相同、是否以某种顺序包含了同样的字节以及其他类似的东西。另外还有一些程序可以比较文件的相等(cmp)或报告差异(diff)等。这些程序，或其他类似的东西在许多环境里都可以直接使用，也是非常值得用的。

一个字节频度程序可以用来检查数据特征的不变性，也可以报告出现了反常情况。例如，在假定只包含文本的文件里出现了非文本字符。下面就是一个这种程序，我们称其为 freq：

```
#include <stdio.h>
#include <ctype.h>
#include <limits.h>
```

China-pub.com
下载


```
unsigned long count[UCHAR_MAX+1];

/* freq main: display byte frequency counts */
int main(void)
{
    int c;

    while ((c = getchar()) != EOF)
        count[c]++;

    for (c = 0; c <= UCHAR_MAX; c++)
        if (count[c] != 0)
            printf("%.2x  %c  %lu\n",
                c, isprint(c) ? c : '-', count[c]);
    return 0;
}
```

对于那些应该保持不变的特征，实际上也可以在程序内部进行检查。一个求数据结构的元素个数的函数就提供了一种简单的一致性检查能力。一个散列表应该有这样的性质：每个插入其中的元素都可以提取出来。这个条件很容易检查，只要有一个能把散列表内容转入一个文件或者数组里的函数。在任何时刻，插入一个数据结构的元素数目减掉删除的元素数目，必然等于结果里包含的元素数目，这是一个很容易检查的条件[⊖]。

比较相互独立的实现。 一个库或者程序的几个相互独立的实现应该产生同样的回答。例如，由两个编译程序产生的程序，在相同机器上的行为应该一样，至少在大部分情况下应该是这样。

有时一个回答可以由两条完全不同的途径得到，或许你可以写出一个程序的某种简单版本，作为一个慢的但却又是独立的参照物。如果两个相互无关的程序得到的结果完全相同，这就是它们都正确的一个很好的证据。如果得到的结果不同，那么其中至少有一个是错的。

作者之一某次和另一个人一起工作，为一种新机器做一个编译程序。在工作中他们把对编译程序产生的代码做排错的工作分成两块：一个人去写为目标机器产生指令的软件；另一个人写排错系统的反汇编程序。在这种做法之下，指令集合解释或实现中的任何错误都很难在这两个部分里重复出现。当编译程序错误地编码了一条指令时，反汇编程序立刻就能发现它。编译的所有前期输出都用反汇编程序进行处理，再用这个输出结果与编译程序自己的排错(汇编)输出进行比较。这种策略在实践中非常行之有效，很快就找出两部分中的不少错误。如果还遗留下什么障碍，那就是当两个人对机器系统结构描述中的某种歧义性正好做了同样的错误理解时，这时排错就比较困难了。

度量测试的覆盖面。 测试的一个目标是保证程序里的每个语句在一系列测试过程中都执行过，如果不能保证程序的每一行都在测试中至少经过了一次执行，那么这个测试就不能说是完全的。完全覆盖常常很难做到，即使是不考虑那些“不可能发生”的语句。设法通过正常输入使程序运行到某个特定语句有时也是很困难的。

存在一些衡量覆盖面的商用工具。轮廓程序(Profiler)通常是编译系统套装工具中的一部分，它提供了一种方法，能计算出程序里每个语句执行的次数，由此指明在特定测试中达到的覆盖面。

我们测试了第3章的马尔可夫程序，综合使用了各种技术。本章的最后一节将详细描述这些测试。

练习6-3 说明你准备如何测试`freq`。

⊖ 这个条件还依赖于插入时对重复元素的处理方式，可能有一些附加条件。——译者

China-pub.com 下载

练习6-4 设计和实现另一个freq程序，它能够统计其他类型的数据值出现的频度，如 32位整数或者浮点数等。你能够做一个这类程序，使它能够很好地处理各种类型吗？

6.3 测试自动化

以手工方式做大量测试既枯燥无味又很不可靠，因为严格意义上的测试总要涉及到大量的测试实例、大量的输入以及大量的输出比较。因此，测试应该由程序来做，因为程序不会疲劳，也不会疏忽。花点时间写一个脚本程序或者一个简单程序，用它包装起所有的测试是非常值得做的，这能使一个完整的测试集可以通过 (文字或者图形)一个按键而得以执行。测试集运行起来越容易，你运行它的次数就会越多，也越不会跳过它 (即使时间非常紧张)。我们在写这本书时，就为验证所有程序写了一个测试集，每次无论做了什么修改，我们都再次运行它，其中有些部分能在每次重新编译之后自动运行。

自动回归测试。自动化的最基本形式是回归测试，也就是说执行一系列测试，对某些东西的新版本与以前的版本做一个比较。在更正了一个错误之后，人们往往有一种自然的倾向，那就是只检查所做修改是否能行，但却经常忽略问题的另一面，所做的这个修改也可能破坏了其他东西。回归测试的作用就在这里，它要设法保证，除了有意做过的修改之外，程序的行为没有任何其他变化。

有些系统提供了很丰富的工具，以帮助实现这种自动化。脚本语言使我们能很方便地写一些短的脚本，去运行测试序列。在 Unix上，像cmp或diff这样的文件比较程序可用于做输出的比较，sort可以把共同的元素弄到一起，grep可以过滤输出，wc、sun和freq对输出做某些总结。利用所有这些很容易构造出一个专门的测试台。或许这些对于大程序还不够，但是对个人或者一个小组维护的程序则完全是适用的。

下面是一个脚本，它完成对一个名字为 ka的应用程序的回归测试。它用一大堆各种各样的测试数据文件运行程序的老版本 (old_ka)和新版本(new_ka)，对输出中每个不相同情况都给出信息。这个脚本是用Unix的shell写的，但是很容易改写为Perl或者其他脚本语言。

```
for i in ka_data.*          # loop over test data files
do
    old_ka $i >out1         # run the old version
    new_ka $i >out2         # run the new version
    if ! cmp -s out1 out2   # compare output files
    then
        echo $i: BAD        # different: print error message
    fi
done
```

测试脚本通常应该默不做声地运行，只在发生了某些未预见情况时才产生输出，就像上面所采用的方式。我们也可以在测试过程中打印每个文件名，如果什么东西出了毛病，就在文件名之后给出错误信息。这种显示工作进展情况的方式也能指出一些错误，例如无限循环，或者测试脚本没能运行正确的测试等。但是，如果测试运行得很好，喋喋不休的废话就会使人生厌。

上面用的-s参数能使cmp只报告工作状态，不产生输出。如果文件比较相等， cmp的返回状态为真， !cmp是假，这时就不打印任何东西。如果老的输出和新的不同，那么 cmp将返回假，这时文件名和警告信息都会被输出。

China-pub.com
下载

回归测试实际上有一个隐含假定，假定程序以前的版本产生的输出是正确的。这个情况必须在开始时仔细进行审查，使这些不变性质能够一丝不苟地维持下去。如果在回归测试里潜伏着错误结果，人们将很难把它查出来，所有依赖它的东西将一直都是错的。所以，在实践中应该周期性地检查回归测试本身，以保证它的可靠性。

建立自包容测试。自包容测试带着它们需要的所有输入和输出，可以作为回归测试的一种补充。我们测试Awk的经验可能有些教益。我们对特定输入运行一些小程序，以测试各种语言结构，检查它们产生的输出是否正确。下面是为验证一个复杂增量表达式而写的一大堆测试里的一部分。这个测试运行一个Awk的新版本(newawk)，让一个很短的Awk程序把输出写进文件里，用echo命令向另一个文件里写入正确输出，然后做文件比较，如果有差异就报告错误。

```
# field increment test: $i++ means ($i)++, not $(i++)

echo 3 5 | newawk '{i = 1; print $i++; print $1, i}' >out1

echo '3
4 1' >out2  # correct answer

if ! cmp -s out1 out2   # outputs are different
then
    echo 'BAD: field increment test failed'
fi
```

第一个注释是测试输入的一部分，它解释这个测试所做的到底是什么。

有时不用很多时间就可以构造出大量测试。对于简单表达式，我们可以建立一个小而特殊的语言，描述测试、输入数据以及所期望的输出。下面是一个不长的测试序列，考察在Awk里数值1可以用哪些方式表示：

```
try {if ($1 == 1) print "yes"; else print "no"}
1       yes
1.0     yes
1E0     yes
0.1E1   yes
10E-1   yes
01      yes
+1      yes
10E-2   no
10      no
```

第一行是被测试程序(所有出现在单词try后面的东西)，随后的每一行是一对数据，分别表示输入和我们所希望的输出，两者间用制表符分隔。第一个测试说明，如果第一个输入域是 1，那么输出应该是yes。前面7个测试都应该输出yes，而最后两个应该输出no。

我们用一个Awk程序(还能用其他什么东西呢？)把每个测试转换为一个完整的 Awk程序，然后再对各个输入运行它，并将实际输出与所希望的输出做比较。它只报告那些回答错误的情况。

类似机制也可以用来测试正则表达式匹配和代换命令。例如用一个写测试的小语言，可以很方便地写出许多测试情况，用程序来写程序去测试另一个程序，回报将是丰厚的 (第9章还有许多关于小语言和用程序来写程序的讨论)。

总计起来，我们为Awk设计了大约一千个测试，而这整个的测试集用一个命令就可以运行，如果所有的东西都没有问题，那么测试将不产生任何输出。每当我们增加了一个新特征，

China-pub.com 下载

或者更正了一个程序错误，我们就在测试集里增加一些测试，验证有关操作的正确性。一旦程序做了修改，哪怕是最简单的，我们也会运行整个的测试集，这只不过花费几分钟时间。有时这种测试会发现某些完全没有预料到的错误，它也使 Awk的作者们多次避免了当众出丑。

如果你发现了一个程序错误，那么又该怎么办？如果这个错误不是通过已有的测试发现的，那么你就应该建立一个能发现这个问题的新测试，并用那个崩溃的代码版本检验这个测试。一个错误实际上可能会提出一批测试，甚至是需要检查的整整一集新问题，或许是要求在程序里增加一些防御机制，使程序能在其内部发现这个错误。

不要简单地把测试丢掉，因为它能够帮你确定一个错误报告是否正确，或是说明某些东西已经更正了。应保留所有关于程序错误、修改和更正的记录，它能够帮助你识别老问题、纠正新错误。在许多产业的程序设计公司里，这种记录是必须做的。对于你个人的程序设计，这也是一种小的投资，而它们的回报则会是源源不断的。

练习6-5 为printf设计一个测试集，尽可能多地使用机器的帮助。

6.4 测试台

到目前为止，我们讨论的主要是对独立程序以完全形式进行测试时所遇到的问题。这当然不是测试自动化的惟一形式，也不是在大程序构造过程中对程序部分做测试的方式，特别是如果你是程序组的成员。对于测试将要装进某些更大程序里的小部件而言，这也不是有效的方法。

要孤立地测试一个部件，通常必须构造出某种框架或者说是测试台，它应能提供足够的支持，并提供系统其他部分的一个界面，被测试部分将在该系统里运行。我们在本章前面用测试二分检索的小例子说明了一些问题。

如果要测试的是数学函数、字符串函数、排序函数以及其他类似东西，构造一个测试台通常很简单，这种测试台的主要工作是设置输入参数、调用被测试函数，然后检查结果。如果要测试一个部分完成的程序，构造测试台可能就是个大工作了。

为了展示这方面的情况，我们将用 memset(C/C++ 标准库里的一个mem... 函数)的测试来说明有关问题。mem类函数通常是用汇编语言为特定机器写的，这是由于它们的执行性能非常重要。程序被调整得越仔细，也就越容易出错，因此也就越需要做彻底测试。

第一步应该是提供一个尽可能简单的C版本，并保证它能够工作。这既可以作为检验执行性能的基准程序，更重要的是为了考察程序的正确性。在移到一个新环境时，应该总带着简单程序版本，一直使用它们，直到经过仔细调整的东西能工作为止。

函数memset(s, c, 把存储器里从s开始的n个字节设置为c，最后返回s。如果不考虑速度，这个函数很容易写出来：

```
/* memset: set first n bytes of s to c */
void *memset(void *s, int c, size_t n)
{
    size_t i;
    char *p;

    p = (char *) s;
    for (i = 0; i < n; i++)
        p[i] = c;
    return s;
}
```

但是如果考虑速度，就必须用一些技巧，例如一次设置32位或者64位的一个字。这样做可能引入程序错误，所以必须做范围广泛的测试。

测试采用在可能出问题的点上做穷尽检查和边界条件测试相结合的方式。对于memset而言，边界情况包括*n*的一些明显的值，如0、1和2，但也应包括2的各个幂次以及与之相近的值，既有很小的值，也包括2^{16}这样的大数，它对应许多机器的一种自然边界情况，一个16位的字。2的幂值得注意，因为有一种加速memset的方法就是一次设置多个字节，这可以通过特殊的指令做；也可能采用一次存一个字的方式，而不是按字节工作。与此类似，我们还需要检查数组开始位置的各种对齐情况，因为有些错误与开始位置或者长度有关。我们将把作为目标的数组放在一个更大的数组里面，以便在数组两边各建立一个缓冲区域或安全边缘，这也使我们可以方便地试验各种对齐情况。

我们还需要对各种不同c值做检查，包括0、0x7F(最大的有符号数，假定采用的是8位字节)、0x80、0xFF(检查与有符号字符或无符号字符有关的可能错误)以及某些比一个字节大的值(以保证使用的就是一个字节)。我们必须将存储区初始化为某种已知模式，与这些字符值都不同，这样才能查清memset是不是向合法区域之外写了东西。

我们可以用最简单方式为测试提供对照的标准：分配两个数组，在数组里对n、c和偏移量在各种组合下的情况做比较：

```
big = maximum left margin + maximum n + maximum right margin
s0 = malloc(big)
s1 = malloc(big)
for each combination of test parameters n, c, and offset:
    set all of s0 and s1 to known pattern
    run slow memset(s0 + offset, c, n)
    run fast memset(s1 + offset, c, n)
    check return values
    compare all of s0 and s1 byte by byte
```

如果memset有问题，写到了数组界限之外，那么最可能受到影响的是靠近数组开始或结束位置的字节。我们预留下缓冲区域，以使被破坏的字节容易观察到，也使因程序错误复写掉程序其他部分的可能性变得小些。为检查函数是否写出了范围，我们需要比较s0和s1的所有字节，而不仅是那n个应该写入的字节。

这样，一个合理的测试集可能应该包含以下内容的所有组合：

```
offset = 10, 11, ..., 20
c = 0, 1, 0x7F, 0x80, 0xFF, 0x11223344
n = 0, 1, 2, 3, 4, 5, 7, 8, 9, 15, 16, 17,
    31, 32, 33, ..., 65535, 65536, 65537
```

对从0到16的各个*i*，这里的n值至少包括了所有的2^i-1、2^i和2^i+1。

这些值不应该写进测试台的主要部分，但是应该出现在一个用手工或者程序建立的数组里。最好是自动地产生它们，因为这样才能方便地描述出更多2的幂，或许包含更多的偏移量和更多的字符。

这些测试能给memset一个彻底表现的机会，而构造出它们，并让程序执行花费的时间并不多。对于上面的值，总共只有不到3500个测试情况。这个测试完全是可移植的，在需要时可以把它搬到新的环境里。

作为一个警告，请读者注意下面的故事。有一次我们把memset测试程序的一个拷贝给了某些人，他们正在为一个新处理器开发一个新的操作系统和一个函数库。几个月以后，我们

China-pub.com 下载

(作为原来测试程序的作者)开始使用这种机器，发现一个大应用程序在它自己的测试集上出了毛病。我们对程序做跟踪，最后把问题归结到memset的汇编语言实现中有一个与符号扩展有关的微妙错误。由于某些不清楚的原因，库的实现者修改了我们的 memset测试程序，不让它检查0x7F以上的c值。当然，当我们认识到memset非常可疑后，用原来的测试程序立刻就把这个错误揪出来了。

像memset这样的函数对于彻底测试非常敏感。它们很小，很容易证明测试集已经穷尽了代码中所有的可能执行路径，形成了一个完全覆盖。例如，要对 memmove做重叠、方向和对齐方式上所有组合的测试是完全可能的。这里所说的穷尽，并不是所有可能的复制操作，而是指所有的具有代表性的不同输入情况。

与其他所有测试方法一样，测试台方法也需要有正确答案来验证被测试的操作。这里的一种重要技术，正是我们在测试memset时使用的，就是用一个完全能相信是正确的简单版本与一个可能不正确的新版本做比较。这件事可以在测试的不同阶段进行，下面的例子也说明了这个问题。

作者之一实现过一个光栅图形程序库，其中需要提供的一个操作是从一个图像里复制一个像素块到另一个图像。根据参数不同，这个操作完成的可以是简单的存储区拷贝；或者需要把像素值从一个颜色空间转换到另一个颜色空间；或者它需要做“贴块”(tile)，也就是说重复地做输入的拷贝，铺遍一个矩形区域；还可以是这些特征和其他特征的组合。这个操作的规范很简单，而要想高效地实现，就需要写出大量处理各种情况的特殊代码。要保证所有的代码都正确，需要一种有效的测试策略。

首先，我们通过手工写出了最简单的代码，使之能正确完成对一个像素的操作。用它来测试库函数对一个像素的处理。一旦完成了这个阶段，库函数对单个像素的操作就成为可信任的了。

下一步，手工写出调用库函数的代码，每次操作一个像素，构造出一个很慢的程序，实现对一个水平行中所有像素的操作，并用它与库中效率更高的一行操作程序做比较。在这个工作完成后，我们认为库函数已经能完成对行的操作了。

这样继续下去，用行构造矩形，用矩形构造贴块，依此类推。在这个过程中发现了许多程序错误，包括测试程序本身的错误，但是这也正说明了该方法的有效性。我们同时测试了两个互相独立的实现，在此过程中建立了对两者的信任。如果一个测试失败，测试程序将打印出一个细致的分析，帮助了解什么东西可能出错，同时也可以检查测试程序本身的工作是否正确。

这个库在后来的许多年月里做过修改或移植，在这些工作中，该测试程序又反复地表现出它在发现程序错误方面的价值。

由于其一层一层进行的性质，每次使用这个测试程序都需要从零开始，逐步建立起这个程序本身对库的信任。附带地说，这个测试程序不是彻底的、而是概率性的，它生成随机的测试实例。但是，只要经过足够长时间的运行，它最终可能探查出代码中的所有错误。由于这里实际存在数目巨大的可能测试情况，与设法用手工方式构造完全的测试集相比，这种策略更容易实施，它又比做穷尽测试的效率高得多。

练习6-6 按照我们描述的方式为memset建立一个测试台。

练习6-7 为mem...函数族的其他函数建立测试台。

China-pub.com 下载

练习6-8 讨论对数值函数的测试方式，数值函数是指可以在math.h里找到的如sqrt、sin等函数。哪些输入值是有意义的？可以执行哪些独立的检查？

练习6-9 为C标准库的str... 函数族的函数，如strcmp，定义有关的测试机制。其中的一些函数比mem...族的函数复杂得多，特别是strtok和strcspn这样的单词化函数，因此需要做更复杂的测试。

6.5 应力测试

采用大量由机器生成的输入是另一种有效的测试技术。机器生成的输入对程序的压力与人写的输入有所不同。量大本身也能够破坏某些东西，因为大量的输入可能导致输入缓冲区、数组或者计数器的溢出。这对于发现某些问题，例如程序对在固定大小存储区上的操作缺乏检查等，是非常有效的。人本身常常有回避“不可能”实例的倾向，这方面的例子如空的输入，超量级、超范围的输入、不大可能建立的特别长的名字或者特别大的数据值等等。与人相反，计算机会严格按照它的程序做生成，不会回避任何东西。

为说明这个问题，下面是Microsoft Visual C++ Version5.0编译程序产生的一个输出行，是在编译马尔可夫程序的C++ STL实现时产生的。我们已经对它重新编辑了一下，以便使它能放在这里。

```
xtree(114) : warning C4786: 'std::_Tree<std::deque<std::
basic_string<char,std::char_traits<char>,std::allocator
<char>>,std::allocator<std::basic_string<char,std::
  ... “删去1420个字符”
allocator<char>>>>>::iterator' : identifier was
truncated to '255' characters in the debug information
```

这个编译系统警告我们，它生成了一个长度为1594个字符的变量名字(非常可观)，但是它只留下255个字符用在输出排错信息方面。并不是所有程序都能够保护自己，能对付这种罕见长度的字符串。

随机的输入(未必都是合法的)是另一种可能破坏程序的攻击方法。这也是“人们不会做这种事”的推理的一个逻辑延伸。例如，人们对某些商用编译程序用一些随机生成的合语法的程序做测试，这里采用的技巧是根据问题的规范——C语言标准——驱动一个程序，产生合法的但又是希奇古怪的测试数据。

通过随机输入测试，考查的主要是程序的内部检查和防御机制，因为在这种情况下一般无法验证程序产生的输出是否正确。这种测试的目标主要是设法引起程序垮台，或者让它出现“不可能发生的情况”，而不是想发现直接的错误。这也是一种检查程序里错误检查代码能否工作的好方式。如果用的都是有意思的输入，大多数错误根本就不会发生，结果处理这些错误的代码无法得到执行，这就可能使程序错误隐藏在那些角落里。用随机输入测试有时也可能得不到什么回报：例如它可能发现了某些问题，可是这些问题在现实生活中出现的可能性微乎其微，以至都没有必要去考虑它们。

有些测试是针对明显的恶意输入进行的。安全性攻击经常使用极大的或者不合法的输入，设法引起对已有数据的覆盖。检查这方面的弱点是非常明智的。有些标准库函数在这类攻击下很容易受到伤害。例如，标准库函数gets没有提供限制输入行大小的方法，因此绝不要使用它，任何时候都改用fgets(buf, sizeof(buf), stdin)。不加限制的scanf("%s", buf)

China-pub.com 下载

调用对输入行长度也没有限制，应该改用带有明显长度限制的形式，如scanf("%20s",buf)。在第3.3节里，我们说明了对于一般的缓冲区大小，应该怎样处理好这个问题。

任何可能从程序外部接收信息的例程，无论是直接的或者间接的，都应该在使用有关数据之前对它们进行检验。下面是从某本教科书里抄来的一段程序，它要求由用户键盘输入读一个整数，在整数值太大时给用户提出一个警告。书中给出这个例子的目的是说明如何克服gets的问题，但这种解决办法有时并不奏效。

```
?	#define MAXNUM 10
?
?	int main(void)
?	{
?		char num[MAXNUM];
?
?		memset(num, 0, sizeof(num));
?		printf("Type a number: ");
?		gets(num);
?		if (num[MAXNUM-1] != 0)
?			printf("Number too big.\n");
?		/* ... */
?	}
```

如果输入包含10个或者更多的数字，就会有一个非零值覆盖掉数组 num最后的0，按说在gets返回后这个问题应该能被检查出来。不幸的是，这个检查是不够的。恶意攻击者可能提供一个非常长的输入串，它将覆盖掉数组后面的某些关键性数据，甚至覆盖掉函数调用的返回地址，使程序根本不返回到后面的 if语句，而可能是去做某些穷凶极恶的事情了。这种不加检查的输入确实是潜在的安全漏洞。

不要认为这不过是个无关紧要的教科书问题。在 1998年7月，几个最主要的电子邮件程序里都发现了这种形式的程序错误。《纽约时报》报道说：

> 这个安全漏洞是由所谓的"缓冲区溢出错误"引起的。按说程序员应该在他们的软件里包括一些代码，检查进入数据是否具有安全的类型，收到的数据单元是否具有正确长度。如果一个单元过长，它就可能越出"缓冲区"——即那种被设置好，用来存放数据的存储块。在这种情况下，该电子邮件程序就会崩溃，而一个恶意的程序员就可能欺骗计算机，使它运行放在某个位置的一个预谋的程序。

这也是在1988年著名的"Internet蠕虫"事件中被攻击的内容之一。

对这种向小数组里存储非常大的字符串的攻击方式，某些剖析 HTML形式的程序也可能是不设防的：

```
?	static char query[1024];
?
?	char *read_form(void)
?	{
?		int qsize;
?
?		qsize = atoi(getenv("CONTENT_LENGTH"));
?		fread(query, qsize, 1, stdin);
?		return query;
?	}
```

这段代码假定输入绝不会超过 1024字节长。就像gets一样，它对使其缓冲区溢出的攻击也没有设防。

China-pub.com
下载

另一些我们更熟悉的溢出同样可能引起麻烦。如果整数默默地溢出，其后果也可能是个灾难。考虑下面的存储分配：

```
?     char *p;
?     p = (char *) malloc(x * y * z);
```

假设x、y和z的乘积溢出，对malloc的调用有可能产生一个具有合理大小的数组。但是p[x]却可能引用到超出被分配区域的存储。假定 int是16位，x、y和z的值都是41。这样x*y*z是68921，它在模2^{16}下就是3385。因此，对malloc的调用仅分配到3385个字节，任何超出这个值的下标引用都将越过界限。

类型间的转换是溢出的另一个原因，在这种情况下，甚至成功地捕捉到错误都不一定能解决问题。阿里亚娜五型火箭在1996年7月的初次试验中爆炸，是因为它由阿里亚娜四型那里继承来的导航系统没有经过很好的测试。新火箭飞得更快，结果导致了导航软件里某些变量具有更大的值。火箭发射后不久，有一次企图把 64位浮点数转换到16位带符号整数的操作产生了溢出，这个错误其实被捕捉到了，但是，捕捉它的代码做出的选择是关闭这个子系统。这造成火箭转出航线并且爆炸。不幸的是，在这里出毛病的代码生成了惯性参考信息，这种信息本来只在起飞前有用，如果在发射时把这个功能关掉的话，也可能就不会出问题。

还有另一类更简单的情况，有时二进制输入会导致要求文本信息的程序崩溃，特别是如果程序假定输入的是7位的ASCII字符集。对那些要求文本输入的程序，将二进制输入 (例如编译产生的程序)送给它做个试验也是有意义的，或者说是明智的。

好的测试实例经常能使用到许多不同的程序上。例如，任何读文件的程序都应该用空文件做测试；任何读文本文件的程序都应该用二进制文件做测试；任何读文本行的程序都应该用极长的行、空行和完全没有换行符号的输入做测试。在手头保存一批这种测试文件是很好的做法，这样，当你需要用它们测试任何程序时就不必重新建立了。另一种方式是写一个程序，在需要时用它生成这些文件。

当Steve Bourne写他的Unix外壳(即后来著名的Bourne外壳)时，他建立了一个目录，其中包含254个名字是一个字符的文件，每个字节值有一个对应文件，除了 '\0'和斜线符号之外(因为这两个字符不能出现在Unix文件名里)。他用这个目录做所有模式匹配和单词化方式的测试(这个目录本身当然也是利用程序构造的)。在此后的许多年里，这个目录一直是各种文件树遍历程序的灾星，经常能发现它们的毛病。

练习6-10 请设法建立一个文件，它能够击溃你最喜欢的文本编辑器、编译系统或者其他程序。

6.6 测试秘诀

有经验的测试者使用许多技术和技巧，以提高自己的工作效率。本节将介绍一些我们最喜欢的东西。

程序都应该检查数组的界限(如果语言本身不做这件事的话)，但是，如果数组本身的大小比典型输入大得多时，这种检查代码就常常测试不到。为演习这种检查，我们可以临时地把数组改为很小的值，这样做比建立大的测试实例更容易些。在第 2章和第4章的CSV库中，我们对那里的数组增长代码都使用了这种技巧。实际上我们放了一个很小的初始值，这引起的附加的初始代价是微不足道的。

China-pub.com 下载

让散列函数返回某个常数值，使所有元素都跑到同一个散列桶里。这种做法能演习链的机制，它也指明了最坏情况下的性能。

写一个你自己的存储分配函数，有意让它早早地就失败，利用它测试在出现存储器耗尽错误时设法恢复系统的那些代码。下面的函数在调用 10次后就会返回NULL：

```
/* testmalloc: returns NULL after 10 calls */
void *testmalloc(size_t n)
{
    static int count = 0;
    if (++count > 10)
        return NULL;
    else
        return malloc(n);
}
```

在你提交代码的时候，应该关掉其中所有的测试限制，因为它们会影响程序性能。我们在使用某产品编译系统时遇到了性能问题，最后追踪到一个总返回 0的散列函数，原因是测试代码还在使用中。

把数组和变量初始化为某个可辨认的值，而不是总用默认的 0。这样，如果出现越界访问，或者取到了一个未初始化的变量值，你将更容易注意到它。常数 0xDEADBEEF在排错系统里很容易辨认，存储分配程序有时用这样的值，帮助捕捉未初始化的数据。

变动你的测试实例，特别是在用手工做小测试时。总使用同样东西很容易使人陷入某种常规，很可能忽略了其他的崩溃情况。

如果已经发现有错误存在，那么就不要继续设法去实现新特征或者再去测试已有的东西，因为那些错误有可能影响测试的结果。

测试输出中应该包括所有的参数设置，这可以使人容易准确地重做同样的测试。如果你的程序使用了随机数，应当提供方法，设置和打印开始的种子值，并使程序的行为与测试中是否使用随机性无关。应当保证能准确标识测试输入和对应的输出，这样才能理解和重新生成它们。

另一个做法也是很明智的，那就是提供一种方法来控制程序运行中输出的量和种类，附加的输出常能对测试有所帮助。

在不同的机器、编译系统和操作系统上做测试，每种组合都可能揭露出一些在其他情况下不能发现的错误。例如对于字节顺序的依赖性、对空指针的处理、对回车符和换行符的处理以及各种库或者头文件的特殊属性等等。在多种机器上测试还可以发现在集成程序的各个部件，以便发运过程中弄出的错误，可以揭示程序对开发环境的一些无意的依赖性，这是我们在第8章将要讨论的。

我们将在第7章讨论性能测试问题。

6.7 谁来测试

由程序实现者或其他可以接触源代码的人做的测试有时也被称为白箱测试 (这个术语与黑箱测试类似，但用的没有那么多。叫它“透明箱测试”可能更说明问题。在黑箱测试中测试者不知道部件的实现方式)。对自己的代码做测试是非常重要的，不要指望某种测试组织或者用户可以为你发现什么。但是，人很容易用自己已经非常仔细地做过测试来欺骗自己。所以，

China-pub.com
下载

你应该试着不考虑代码本身，仔细考虑最困难的测试实例，而不是那些容易的。这里引用 Don Knuth的一段话，他描述了自己如何为TEX排版程序建立测试：“我设法使自己进入最卑劣的、极其令人讨厌的思想状态，写出自己能想到的最卑劣的测试代码，然后我再环顾四周，设法把它嵌入更加卑劣的几乎是污秽的结构之中”。做测试的原因就是要发现程序里的错误，而不是为了表明这个程序能够工作。所以，测试应该是恶毒的，如果发现了问题，那是你的方法有效的证明，根本不应该恐慌。

黑箱测试的测试者对代码的内部结构毫不知情，也无法触及。这样就可能发现另一类的错误，因为做测试的人对该在哪里做检查可能另有一些猜测。边界条件可能是开始做黑箱测试的好地方，大量的、邪恶的、非法的输入应该是随后的选择。当然你也应该测试常规的“大路货的”东西，测试程序的常规使用，验证其基本功能。

实际用户接踵而至。新用户往往能发现新错误，因为他们会以我们无法预知的方式来探测这个程序。在把一个程序发布到世界之前做这种测试是非常重要的，可惜的是许多程序在没有做好任何一种测试之前就发了货。软件的 Beta发布就是希望在它完成之前请一些真实用户来测试程序，但是， Beta发布绝不该被用做彻底测试的替代物。随着软件系统变得更大更复杂，开发进度表变得更短，不经过适当测试就发货的压力确实是在不断增加。

测试交互式程序是特别困难的，特别是如果它们还涉及到鼠标输入等。有些测试可以用脚本来做(脚本的特性依赖于语言、环境和其他类似东西)。交互式程序应该能够通过脚本控制，这样我们就可以用脚本模拟用户的行为，使测试可以通过程序完成。这方面的一种技术是捕捉真实用户的动作，并重新播放它；另一种技术是建立一个能表述事件序列和时间的脚本语言。

最后，还应该想一想如何测试所用的测试代码本身。在第 5章里，我们曾提到由表程序包的一个错误测试程序所造成的狼狈局面。如果一个回归测试集里感染了一个错误，能够造成的麻烦将是很长远的。如果一个测试集本身有毛病，由它得到的结果也就没有多少意义了。

6.8 测试马尔可夫程序

第3章的马尔可夫程序是非常复杂的，需要仔细地进行测试。这个程序要产生的是无意义的东西，因此很难分析其合法性。另外，我们还在几个语言里写了多个程序版本。在这里最麻烦的是程序输出是随机的，每次都不同。我们怎样才能把本章里学到的东西应用到这个程序的测试中呢？

第一个测试集由几个很小的文件组成，用于测试边界条件，目标是保证程序对只包含几个词的输入能正确产生输出。对于前缀长度为 2的情况，我们用了五个文件，它们分别包含(一行是一个文件)：

```
(空文件)
a
a b
a b c
a b c d
```

对于这里的每个文件，程序的输出都应该与输入完全相同。这些测试揭示出几个在表的初始化、生成程序的开始、结束等地方的“超出一个”错误。

第二项测试检验某些必须保持的特征。对于两词前缀的情况，一次运行中输出的每个词、

China-pub.com 下载

每个词对、以及每个三词序列都必然也出现在输入里。我们写了一个 Awk程序，用它把原始输入读进一个巨大的数组，构造出所有两个词和三个词的序列的数组，再把马尔可夫程序的输出读入另一个数组，然后对它们做比较。

```
# markov test:  check that all words, pairs, triples in
#   output ARGV[2] are in original input ARGV[1]
BEGIN {
    while (getline <ARGV[1] > 0)
        for (i = 1; i <= NF; i++) {
            wd[++nw] = $i    # input words
            single[$i]++
        }
    for (i = 1; i < nw; i++)
        pair[wd[i],wd[i+1]]++
    for (i = 1; i < nw-1; i++)
        triple[wd[i],wd[i+1],wd[i+2]]++

    while (getline <ARGV[2] > 0) {
        outwd[++ow] = $0     # output words
        if (!($0 in single))
            print "unexpected word", $0
    }
    for (i = 1; i < ow; i++)
        if (!((outwd[i],outwd[i+1]) in pair))
            print "unexpected pair", outwd[i], outwd[i+1]
    for (i = 1; i < ow-1; i++)
        if (!((outwd[i],outwd[i+1],outwd[i+2]) in triple))
            print "unexpected triple",
                outwd[i], outwd[i+1], outwd[i+2]
}
```

我们并不想建立一个效率很高的测试，而只是想使测试程序越简单越好。像这样把一个有10000个词的输出文件与一个42685个词的输入文件对照检查一次，需要 6~7分钟时间，并不比有些马尔可夫程序版本生成这个文件用的时间长多少。对于应保持特征的检查捕捉到我们Java实现里的一个大错误：该程序有时复写掉散列表项，因为这个程序用的是引用，而不是做前缀的拷贝。

这个测试还显示了一个原理：验证输出的某种性质比建立这个输出本身要容易得多。例如，要检查一个文件是否已经排序就比把它排序容易得多。

第三步测试是统计性的，输入由下面的序列构成：

```
a b c a b c ... a b d ...
```

这里每十个abc有一个abd。如果随机选择部分工作得很好，那么在输出中 c的数目大约应该是d的十倍。我们用freq检验这个性质，没有问题。

在Java程序的一个早期版本里为每个后缀关联了一个计数器，统计测试说明它对每个 d大约生成20个c，比合理的情况多了一倍。经过令人头疼的努力，我们终于认识到 Java的随机数生成器不但产生正数，也产生负数。这里因子 2的出现是因为随机数值的范围是我们所期望的两倍，所以取模后得零的个数也就多了一倍，这就使得链表里第一个元素占了些便宜，而它恰好就是c。改正这个毛病只需在取模之前先求绝对值。没有这个测试，我们绝不可能发现这个错误，因为当时的输出用眼睛看起来非常好。

最后，我们给马尔可夫程序普通的英文文本，看着它产生出很漂亮的无意义的费话。当然，我们在程序开发的前期也运行过这种测试。但是，在程序处理正规输入时我们并没有放

China-pub.com
下载

弃测试，因为有些难对付的实例可能会出现在实际中。对简单的实例能够正确处理是很诱人的，难的实例也必须测试。自动的、系统化的测试是避免这种陷阱的最好方法。

所有这些测试都是机械化的。用一个脚本产生必需的输入数据，运行测试并且对它们计时，打印出反常的输出。这个脚本本身是可配置的，所以同一个测试能够应用到马尔可夫程序的各种版本上，每次我们对这些程序中的一个做了更改后，就重新运行所有测试，以保证所有东西都没出问题。

6.9 小结

你把开始的代码写得越好，它出现的错误也就越少，你也就越能相信所做过的测试是彻底的。在写代码的同时测试边界条件，这是去除大量可笑的小错误的最有效方法。系统化测试以一种有序方式设法探测潜在的麻烦位置。同样，毛病最可能出现在边界，这可以通过手工的或者程序的方式检查。自动进行测试是最理想的，用得越多越好，因为机器不会犯错误、不会疲劳、不会用臆想某些实际无法工作的东西能行来欺骗自己。回归测试检查一个程序是否能产生与它们过去相同的输出。在做了小改变之后就测试是一种好技术，能帮助我们将出现问题的范围局部化，因为新问题一般就出现在新代码里面。

对于测试，惟一的、最重要的规则就是必须做。

补充阅读

学习测试的一个方法就是研究最好的、能自由使用的软件中的实际例子。 Don Knuth在《软件：实践和经验》(Software: Practice and Experience19卷第7期pp607~685, 1989)的文章“The Errors of TEX”中描述了到当时为止在TEX排版程序中发现的每一个错误，还包含了许多对他的测试方法的讨论。为 TEX构造的TRIP测试是完整测试集的一个绝佳的实例。 Perl也带有一个范围广泛的测试集，在它被编译安装到一个新系统以后，可以用来检验其正确性。这个测试集里还包括一些模块，如 MakeMaker和TestHarness等，用来帮助测试Perl的扩充。

Jon Bentley在《美国计算机协会通信》(Communications of the ACM)上写过一系列文章，后来汇编成两本书，《程序设计精萃》(Programming Pearls)和《更多的程序设计精萃》(More Programming Pearls)，由Addison-Wesley公司分别于1986和1988年出版。文中常常谈到测试问题，特别是大量测试的组织和机械化问题。

China-pub.com
下载

第7章 性 能

他过去所做的许诺，是非凡的；
而他今天的执行[⊖]，则什么也不是。

莎士比亚，《亨利八世》

很久以前，程序员花费了他们的主要精力，想方设法把程序的效率弄得高一点。因为在那个时候计算机非常慢，而且非常昂贵。今天的机器便宜多了也快多了，因此，对绝对效率的需求也就大大地减弱了。我们难道还值得去考虑执行性能吗？

当然需要。但是，只有在问题确实是非常重要，程序真正是非常慢，而且确有预期能在保持正确性、坚固性和清晰性的同时把程序弄得更快的情况下，才应该做这件事情。一个快速的程序如果给出的是错误结果，那么它一点也没有节省时间。

优化的第一要义是不做。程序是不是已经足够好了？应该了解程序将如何使用以及它将要运行于其中的环境，把它搞得更快有什么益处吗？为大学课程作业而写的程序不会再去使用，速度一般是没有关系的。对于大部分个人程序、偶尔用用的工具、测试框架、各种试验和原型程序而言，通常也都没有速度问题。而在另一方面，对一个商业产品或者其中的核心部件，例如一个图形库，性能可能就是非常关键的。因此，我们还是需要理解如何去考虑性能问题。

什么时候我们应该试图去加速一个程序？我们该如何做，又能够期望得到些什么呢？本章要讨论如何能使程序运行得更快些，或者少用些存储。速度通常是人们最关心的，因此也是我们讨论的主题。空间(主存、磁盘)出问题的情况少一点，但也可能是至关重要的，所以我们也要在这里花一些时间和空间(篇幅)。

正如我们在第2章看到的，最好的策略就是使用那些能适应工作需要的、最简单最清晰的算法和数据结构。然后再度量其性能，看是否需要做什么改变；打开编译系统的选择项，生成尽可能快的代码；考虑对程序本身做什么改变才能有最大作用；一次做一个改变并重新进行评价。在这期间始终保留最简单的版本，用它作为测试版本的对照物。

测量是改进性能过程中最关键的一环，推断和直觉都很容易受骗，所以在这里必须使用各种工具，如计时命令或轮廓文件等等。性能改进与程序测试有许多共同之处，包括许多技术，如自动化，认真保存记录，用回归测试以保证所做的改变维持了正确性，而且没有损害以前的改进等。

如果算法选择是明智的，程序也写得很仔细，那么你就可能发现完全没有进一步加速的必要性。对于写得很好的代码，很小的改变往往就能解决它的性能问题，而对那些设计拙劣的代码，经常会需要大范围地重写。

7.1 瓶颈

让我们从消除一个瓶颈的事例开始，事情出在我们局部环境里的一个关键程序上。

⊖ 这里的原文是performence，在本章题目和正文中均译为“性能”或“执行性能”。——译者

China-pub.com
下载

外来的电子邮件汇集之后通过一台称为网关的机器，这台机器将内部网络与外部的Internet连接起来。外来的电子邮件消息——对于一个有数千人的团体，每天是数以万计的。这些邮件到达网关并传送到内部网。这个分隔把我们的私有网络和公共的Internet隔离开，并使这个团体所有的人只需要公开一台机器的名字(网关的名字)。

网关有一项工作是过滤掉所有的“垃圾邮件”(spam)，即那些不请自来的邮件，它们通常宣扬一些没什么价值的服务。通过早期的成功试验之后，这个垃圾邮件过滤程序被安装好，作为提供给网关用户的一项固定服务。这之后，一个问题就出现了。该网关机器原来已经很陈旧，已经非常忙，现在则完全被压垮了。造成这种情况的原因是过滤器程序占用了太多的机器时间——比对消息的所有其他处理所需时间加在一起还要多得多，这使消息队列被填满，在系统的哽噎挣扎之中，消息发送被成小时地拖延。

这是性能真有问题的一个例子：程序对于完成指定工作而言太慢，而人则由于其工作拖延感到非常不舒服。这种程序确实必须大大地加快运行速度。

对问题稍做简化，这个垃圾邮件过滤程序的工作方式大致是这样：每个外来消息被当作一个字符串看待，一个模式匹配程序检查这种串，看它是否包含了某些已知垃圾邮件中的短语，如“Make millions in your spare time”或“XXX-rated”等。这种消息总是翻来覆去的，所以这个技术应该很有效。如果发现一个垃圾邮件没被捉住，只要向表里加入一个短语，程序在下次就能逮住它。

系统里现成的串匹配工具，如grep等，都不能同时具有合适的执行性能和包装方式，因此我们只能设法写一个特殊的垃圾邮件过滤程序。原始代码非常简单，它检查一个消息里是否包含任一指定的短语(模式)：

```
/* isspam: test mesg for occurrence of any pat */
int isspam(char *mesg)
{
    int i;

    for (i = 0; i < npat; i++)
        if (strstr(mesg, pat[i]) != NULL) {
            printf("spam: match for '%s'\n", pat[i]);
            return 1;
        }
    return 0;
}
```

这种东西还怎么可能弄得更快呢？字符串必须被检索，而来自C标准库的strstr函数是做这种检索的最好方式：它是标准的、高效的。

通过使用轮廓文件技术(将在下一节里讨论)，我们弄清问题就出在strstr的实现方式上。在用于垃圾邮件过滤程序时，它具有一些很不幸的性质。通过改变strstr的工作方式，可以使它的效率大大提高(这是针对这个特定问题而言)。

函数strstr的现有实现大致是下面的样子：

```
/* simple strstr: use strchr to look for first character */
char *strstr(const char *s1, const char *s2)
{
    int n;

    n = strlen(s2);
    for (;;) {
        s1 = strchr(s1, s2[0]);
```

China-pub.com 下载

```
            if (s1 == NULL)
                return NULL;
            if (strncmp(s1, s2, n) == 0)
                return (char *) s1;
            s1++;
        }
    }
```

这个写法已经考虑了效率问题。实际上，在典型应用中它也是非常快的，因为工作中用的是经过高度优化的库代码。函数调用strchr去发现模式中第一个字符在串里的下一个出现位置，然后调用strcmp查看串的后面部分能否与模式的剩余字符匹配。这样，在寻找模式首字符的过程中，strstr能迅速地跳过消息的绝大部分，而后又能对其余部分做快速检查。这怎么会产生很糟糕的性能呢？

确实有许多原因：首先，strncmp带有一个模式长度参数，它必须通过strlen计算出来。由于执行中的模式是固定的，因此不应该对各个消息都重新计算模式的长度。

第二，strncmp用到一个复杂的内部循环。它不只是对两个串里的各个字节做比较，在对长度值向下计数的同时还必须关注两个串的结束\0字节。由于所有字符串的长度都为已知的(虽然strncmp并不知道)，这种复杂性完全是不必要的。知道计数值就足够了，检查\0纯粹是浪费时间。

第三，strchr本身也很复杂，因为它必须在寻找字符的同时也关注作为消息结束标志的\0字节。在对isspam的每次调用中，消息都是固定不变的，所以在寻找\0上花的时间完全是白费功夫，因为我们知道消息到那儿结束。

最后，虽然strstr、strchr和strncmp各自独立地看效率都很高，但反复调用这些函数的代价，与它们执行的计算工作相比，也非常可观。完成所有这些工作，一个更有效的方法是用一个特殊的、仔细写出的strstr版本，完全避免对其他函数的调用。

这些问题都是性能方面出麻烦的常见原因——某个例程或者界面对于各种典型情况都工作得非常好，但是对某些特殊情况却很糟糕，特别是如果这些情况又恰好出现在程序所处理工作的中心位置时。现有的strstr在模式和字符串都比较短、每次调用的处理对象又都不同的情况下，都能工作得很好；对于那些长而固定的字符串，它的开销就太高了。

有了这些考虑后，我们重写了strstr，同时处理模式和消息串，在其中寻找匹配，不调用任何子程序。这个实现确实具有我们所期望的行为：在某些情况下它稍微慢一点，但是做垃圾邮件过滤器就快得多，更重要的是它不再出现极坏的情况。为了验证新实现的正确性和执行性能，我们构造了一个性能测试集。这个测试集里不仅包括简单的例子，例如在一个句子里查找一个词；还包括一些病态情况，如在有一千个e的串里查找只有单个字母x的模式，在一个字母e的串里查找有一千个x的模式等。简单实现对这两种情况的处理都非常糟糕。这种极端性例子也是性能评价的关键。

用新的strstr更新了函数库之后，垃圾邮件过滤程序的运行速度提高了大约30%。对于只是重写一个子程序而言，这已是很好的回报了。

不幸的是，现在的程序仍然太慢。

在需要解决问题的时候，最重要的是给出正确的提问。到目前为止，我们提出的问题一直是，寻找在一个字符串里查找一个正文模式的最快方法。而实际问题却是要在一种很长的变动的串里，查找一个很大的固定的正文模式集合。对于这种提问，strstr是不是正确的解

China-pub.com
下载

决办法，这件事并不是一目了然的。

要使程序运行得快，最有效的方法就是使用更好的算法。对问题有了更清楚的看法之后，现在是认真想一想什么算法有可能工作得更好的时候了。

基本循环：

```
for (i = 0; i < npat; i++)
    if (strstr(mesg, pat[i]) != NULL)
        return 1;
```

对消息做npat次独立的扫描。即使它没有发现任何匹配，也需要对消息串里的每个字节做npat次检查，总共要做strlen(mesg)*npat次比较。

一种更好的方法是把循环翻转过来，在外层循环中只对消息串扫描一次，其间在内层循环里并行地查找各个模式：

```
for (j = 0; mesg[j] != '\0'; j++)
    if (some pattern matches starting at mesg[j])
        return 1;
```

通过简单观察，就可以看出性能改善的可能性。要知道是否有模式能在位置 j与消息串匹配，我们并不需要检查所有的模式，只要检查那些开始字符与 mesg[j]相同的模式就足够了。粗略估计，由于有52个大写和小写字母，我们只需要做大约strlen(mesg)*npat/52次比较。由于字母的分布不均匀——例如，以s开头的词远多于以x开头的——我们不能期望性能会提高52倍，但总应该提高一些。为此我们构造了一个散列表，用模式串的第一个字母作为关键字。

通过预先计算，构造出一个表，其中的项是以各个字符开始的模式，此后的 isspam仍然非常短：

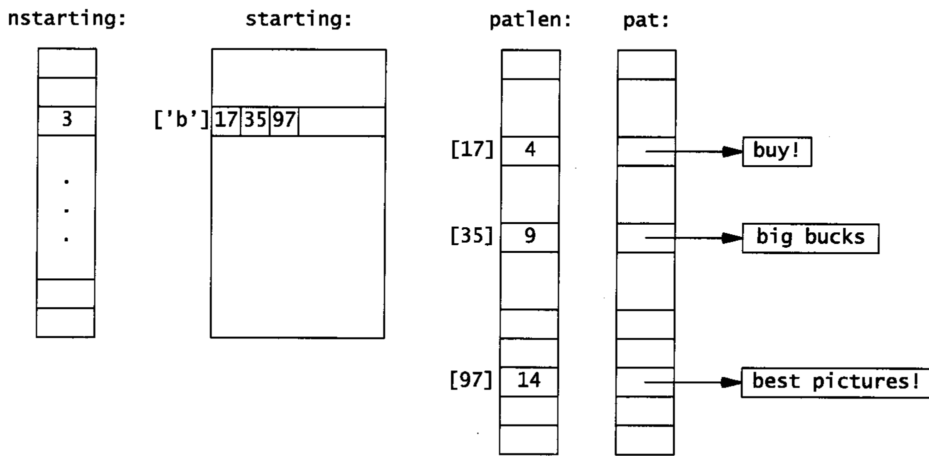
```
int patlen[NPAT];                    /* length of pattern */
int starting[UCHAR_MAX+1][NSTART]; /* pats starting with char */
int nstarting[UCHAR_MAX+1];          /* number of such patterns */
...
/* isspam: test mesg for occurrence of any pat */
int isspam(char *mesg)
{
    int i, j, k;
    unsigned char c;

    for (j = 0; (c = mesg[j]) != '\0'; j++) {
        for (i = 0; i < nstarting[c]; i++) {
            k = starting[c][i];
            if (memcmp(mesg+j, pat[k], patlen[k]) == 0) {
                printf("spam: match for '%s'\n", pat[k]);
                return 1;
            }
        }
    }
    return 0;
}
```

这里用了一个两维数组starting[c][]。对每个字符c，在数组starting[c]里存储着以c开头的所有模式的指标。在与之相关的另一个数组 nstarting[c]里，记录着以c开头的模式的个数。如果没有这些表格，内部循环就必须从 0运行到npat，大概需要上千次。现在它只要从0循环到大约20。最后，在数组patlen[k]里存储着预先算出的strlen（pat[k]）

China-pub.com 下载

的值。

下面的图中显示了这些数据结构的样子，其中只画出由字母 b开头的三个模式：



构造这些表格的代码很简单：

```
int i;
unsigned char c;

for (i = 0; i < npat; i++) {
    c = pat[i][0];
    if (nstarting[c] >= NSTART)
        eprintf("too many patterns (>=%d) begin '%c'",
            NSTART, c);
    starting[c][nstarting[c]++] = i;
    patlen[i] = strlen(pat[i]);
}
```

根据输入不同，这个垃圾邮件过滤程序比采用改进的 strstr程序快5~10倍，比初始程序快7~15倍。我们没有得到52倍的改进，其中一个原因是字母分布的不一致性，还由于新程序的循环比原来复杂许多。此外，在这里仍要做许多失败的串比较。但是垃圾邮件过滤程序已经不再是电子邮件发送的瓶颈，性能问题已经解决了。

本章的后面部分将详细讨论用于发现性能问题、隔离出过慢的代码并使其加速的各种技术。在讨论这些内容之前，重新审视垃圾邮件过滤程序，看看它给我们上了一堂什么课，也是非常有益的。在这里最重要的，就是必须弄清性能是不是一个实实在在的问题。如果垃圾邮件过滤程序不是瓶颈，那么所有努力就都毫无价值了。一旦弄清楚性能确实是问题，我们就可以用轮廓程序或者其他技术去研究程序的行为，搞清楚问题究竟出在哪里。我们必须保证确实是找准了出问题的地方。这可能需要检查整个程序，而不是仅仅把精力集中到 strstr上，虽然它是明显的，但却又是误认的嫌犯。最后，我们用一个更好的算法解决了应该解决的问题，并通过测试知道它确实快了许多。一旦程序已经足够快了，我们就停止，干嘛要没事找事呢？

练习7-1 通过一个表格，实现从一个字符到以这个字符开头的一组模式的映射，能得到大约一个数量级的性能改进。实现 isspam的另一个版本，它采用两个字符作为下标变量，这样做可能得到怎样的性能改进？这些做法都是一种称为字符树（trie)的数据结构的特例。许多这类数据结构都是用空间换时间。

China-pub.com 下载

7.2 计时和轮廓

自动计时测量。 许多系统里都有这种命令，它们可用于测量一个程序到底用了多少时间。Unix系统里有关命令的名字是time：

```
% time slowprogram

real        7.0
user        6.2
sys         0.1
%
```

这执行了一个命令，报告出三个数值，都以秒数计：“real(实际)”时间，程序从开始到完成的全部时间；“user”CPU时间，执行用户程序本身所花费的时间；以及“system”(系统)CPU时间，也就是花费在操作系统内部程序行为方面的时间。如果你的系统里有类似命令，请使用它。与用秒表计时相比，这样做得到的信息更多，更可靠，也更容易追踪。应该留下很好的记录。在对一个程序做工作，修改并测量的过程中，你可能积累了大量数据，过一两天就可能把人搞糊涂了(哪一个版本的速度快20%？)。我们在关于测试的一章里讨论过许多技术，它们都可以移植到性能的测量和改进方面来。用机器去运行并测量你的测试集，更重要的是使用回归测试来保证所做的修改并没有使程序崩溃。

如果在你的系统里没有time命令，或者你需要孤立地对一个函数进行计时，构造一个计时平台也很容易，与构造一个测试台差不多。C和C++提供了一个标准函数，clock，它报告程序到某个时刻总共消耗的CPU时间。可以在一个函数的执行前和执行后调用clock，测量CPU的使用情况：

```
#include <time.h>
#include <stdio.h>
    ...
    clock_t before;
    double elapsed;

    before = clock();
    long_running_function();
    elapsed = clock() - before;
    printf("function used %.3f seconds\n",
        elapsed/CLOCKS_PER_SEC);
```

CLOCKS_PER_SEC是个尺度项，表示由clock报告的计时器的分辨率。如果一个函数消耗的时间远远不到一秒，那么就应该把它放在一个循环里运行，但这时就需要考虑循环本身的开销，如果这个开销所占的比例很大的话：

```
before = clock();
for (i = 0; i < 1000; i++)
    short_running_function();
elapsed = (clock()-before)/(double)i;
```

在Java中，Date类里的函数给出的是挂钟时间，它是CPU时间的近似值：

```
Date before = new Date();
long_running_function();
Date after = new Date();
long elapsed = after.getTime() - before.getTime();
```

函数getTime的返回值以毫秒计。

使用轮廓程序。 除了可靠的计时方法外，在性能分析中最重要的工具就是一种能产生轮廓文

China-pub.com 下载

件的系统。轮廓文件是对程序在哪些地方消耗了时间的一种度量。在有些轮廓文件中列出了执行中调用的各个函数、各函数被调用的次数以及它们消耗的时间在整个执行中的百分比。另一些轮廓文件计算每个语句执行的次数。执行非常频繁的语句通常对总运行时间的贡献比较大，根本没执行的语句所指明的可能是些无用代码，或者是没有合理测试到的代码。

轮廓文件是一种发现程序中执行热点的有效手段，所谓热点就是那些消耗了大部分计算时间的函数或者代码段。当然，对轮廓文件的解释也应该慎重。由于编译程序本身的复杂性、缓冲存储器和主存的复杂影响，还有做程序的轮廓文件对其本身执行所造成的影响等，轮廓文件的统计信息只能看作是近似的。

在1971年引进轮廓文件这个术语的文章中， Don Knuth写到：“一个程序中少于百分之四的部分通常占了程序一半以上的执行时间。”这也指明了轮廓文件的使用方法：弄清程序中最消耗时间的部分，尽可能地改进这些部分，然后再重新测量，看看是否有新的热点浮现出来。通常在重复一次两次以后，程序里就再也没有明显的热点了。

轮廓文件常常可以通过编译系统的一个标志或选择项打开，然后运行程序，最后用一个分析工具显示结果。在Unix上，这个标志一般是-p，对应的工具是prof：

```
% cc -p spamtest.c -o spamtest
% spamtest
% prof spamtest
```

在下面的表格里，列出的是对垃圾邮件过滤程序某个特定版本产生的轮廓文件，我们建立这个文件，以便理解程序的行为。这里用的是一个固定的消息，一个也是固定的、有 217个短语的测试集合，用它与该消息匹配 10 000次。程序运行在 250MHz的MIPS R10 000上，使用strstr的原始实现并调用其他基本函数。程序输出经过编辑和重新编排，以便能够放在这里。请注意输入的大小(217个短语)和运行的次数(10 000)怎样出现在表的“调用”列里，该列中显示的是各个函数的调用次数，可以作为一种一致性检查。

12234768552: 总的执行的指令数
13961810001: 总的计算循环数
55.847: 总的执行时间(秒)
1.141: 平均每指令的循环数

秒	%	累计%	循环	指令	调用	函数
45.260	81.0%	81.0%	11314990000	9440110000	48350000	strchr
6.081	10.9%	91.9%	1520280000	1566460000	46180000	strncmp
2.592	4.6%	96.6%	648080000	854500000	2170000	strstr
1.825	3.3%	99.8%	456225559	344882213	2170435	strlen
0.088	0.2%	100.0%	21950000	28510000	10000	isspam
0.000	0.0%	100.0%	100025	100028	1	main
0.000	0.0%	100.0%	53677	70268	219	_memccpy
0.000	0.0%	100.0%	48888	46403	217	strcpy
0.000	0.0%	100.0%	17989	19894	219	fgets
0.000	0.0%	100.0%	16798	17547	230	__malloc
0.000	0.0%	100.0%	10305	10900	204	realfree
0.000	0.0%	100.0%	6293	7161	217	estrdup
0.000	0.0%	100.0%	6032	8575	231	cleanfree
0.000	0.0%	100.0%	5932	5729	1	readpat
0.000	0.0%	100.0%	5899	6339	219	getline
0.000	0.0%	100.0%	5500	5720	220	_malloc

China-pub.com
下载

很明显，strchr和strncmp(两个都是strstr调用的)完全主导了整个执行过程。Knuth的教诲是正确的，程序里很小的一部分消耗了大部分执行时间。当对某个程序首次做轮廓时，经常能看到一个运行最多的函数使用了 50%或更多的时间，就像在这里一样，因此很容易决定应该把注意力集中于何处。

集中注意热点。在重写了strstr之后，我们重新做spamtest的轮廓文件。发现当时99.8%的时间都消耗在strstr一个函数上，虽然整个程序确实得到了明显的加速。当某一个函数单独成为一个瓶颈的主导因素时，那么就只有两条路可以走了：或是采用一种更好的算法来改进这个函数；或者直接删去这个函数，重写环绕它的程序部分。

在目前情况下，我们采用了重写程序的方式。对于最后的 isspam的快速实现，下面是它的spamtest轮廓文件的前几行。在这里可以看到总的时间大大减少了，函数 memcmp变成了新热点，而isspam也消耗了时间中相当大的一部分。现在这个 isspam比直接调用strstr的那个版本复杂多了，但它的开销早已通过删去 strlen和strchr，并用memcmp取代strncmp得到了补偿，memcmp对每个字节做的工作少得多。

秒	%	累计%	循 环	指 令	调 用	函 数
3.524	56.9%	56.9%	880890000	1027590000	46180000	memcmp
2.662	43.0%	100.0%	665550000	902920000	10000	isspam
0.001	0.0%	100.0%	140304	106043	652	strlen
0.000	0.0%	100.0%	100025	100028	1	main

花一点时间，比较两个轮廓文件里机器循环计数和调用计数，也是很有教益的。注意这里对strlen的调用从数百万次下降到652，而对memcmp的调用则与原来对strncmp的调用次数相同。还应该注意 isspam，它现在包含了 strchr的功能，但占用周期比原来的strchr少得多，这是因为它在每步只检查相关的模式。通过检查这里的数字，可以看到执行情况的许多细节。

程序中的热点常常可以用比我们处理垃圾邮件过滤程序简单得多的方法除去，或者至少是将它冷却下来。很久以前，在一次回归测试中， Awk的一个轮廓文件指明有一个函数被调用了大约一百万次，下面是有关循环：

```
?       for (j = i; j < MAXFLD; j++)
?           clear(j);
```

这个循环的作用是在每次读入新行之前清理各个域，它大概耗费了整个执行时间的 50%。在这里常数MAXFLD的值为200，是一个输入行里最多允许的域的个数。但是，对于大部分 Awk应用而言，实际上域的个数常常只是 2或3。这样，大量时间被白白浪费到清理那些根本没使用过的域上了。我们修改了方式，以曾经用到的最多的域个数值取代这个常数，这就使总的执行速度提高了25%。

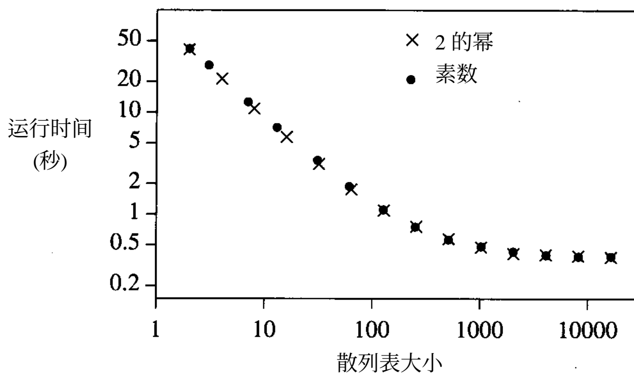
```
for (j = i; j < maxfld; j++)
    clear(j);
maxfld = i;
```

画一个图。图形特别适合用来表现性能测量的情况，它可以很好地传达信息，例如参数改变的影响、算法和数据结构的比较，有时也能指出某些没有预料到的情况。在第 5章给出过一些散列乘因子产生的链长度的图形，显示出某些乘因子比另一些好得多。

下面的图形显示的是散列数组大小对运行时间的影响，这里用的是 C版本的markov程序，输入数据是《诗篇》(共42 685个词，22 482个前缀)。我们做了两集试验，一集试验用 2的幂

China-pub.com 下载

作为数组大小，从2到16 384；另一集试验采用小于2各个幂次的最大素数作为数组大小。我们希望了解的是，采用素数大小的数组在性能方面能否产生什么可以测量到的差异。



这个图说明，当数组大小超过1000个元素之后，运行时间对数组大小的变化就不再敏感了。对于素数或者2的幂作为数组大小，我们根本看不到任何明显差异。

练习7-2 无论你的系统里有没有time命令，都请用clock或者getTime写一个为自己使用的计时程序。将它的时间与挂钟做比较，机器的其他活动对计时会有影响吗？

练习7-3 在第一个轮廓文件里可以看到，strchr被调用了48 350 000次，而strncmp只被调用了46 180 000次。请解释这个次数差。

7.3 加速策略

在改造一个程序，设法把它弄得更快之前，首先应该确定它确实太慢。然后应该通过计时工具和轮廓文件，弄清时间到底跑到哪里去了。在你知道发生了什么之后，有一些可以采用的策略。我们列出几个，按照获益递减的顺序。

使用更好的算法或数据结构。要使程序运行速度快，最重要的因素是算法与数据结构的选择，有效的算法和不那么有效的算法造成的差距是巨大的。在spam程序上，我们已经看到由于改变数据结构所产生的效率上十倍的变化。如果一个新算法降低了计算的数量级，例如从$O(n^2)$降低到$O(n\log n)$，那么速度的改善又会更大得多。第2章已经讨论过这个问题，这里就不再仔细谈它了。

应该弄清实际的复杂性正是我们所期望的，否则这里就可能隐藏着一个性能错误。下面的片段看起来是一个线性的字符串扫描算法：

```
?     for (i = 0; i < strlen(s); i++)
?         if (s[i] == c)
?             ...
```

但它实际上是平方的。如果s有n个字符，每次调用strlen都需要对这个串里的n个字符扫视一遍，而循环本身又要做n次。

让编译程序做优化。有一种毫不费力的改变就可能产生明显的加速效果，那就是打开编译系统的所有优化开关。现代编译程序已经做得非常好了，这实际上大大减小了程序员对程序做各种小改进的必要性。

在默认情况下，C和C++ 编译程序并不设法做很多优化工作，通过编译选项可以打开一个优化程序(说得更准确些，应该是“改进程序”)。按说这个选项应该是默认的，但是由于做优

China-pub.com 下载

化会妨碍源程序排错系统的工作，所以，程序员必须在确认程序已经排错完毕之后，自己来打开优化系统。

编译程序的优化通常可以在所有地方改进运行的时间性能，一般从几个百分点到两倍的样子。不过，有时优化也可能使程序变慢，所以你应该在交付产品之前重新测量一下。对垃圾邮件过滤程序的几个版本，我们比较了未优化的和优化编译的结果。对用来测试匹配算法最后版本的测试集，原来的运行时间是8.1秒，打开优化开关后时间降到5.9秒，性能改进了大约25%。另一方面，对于使用原始strstr的版本，优化后性能并没显示出什么改进，因为strstr在装入程序库之前已经进行过优化，而优化程序只对被编译的源代码工作，也不再去处理系统库。实际上，也有些编译系统能做全局优化，它们分析整个程序，寻找所有可以改进的地方。如果在你的系统里有这种编译程序，那么也不妨试试它，说不定它能够进一步挤出一些指令周期来。

还有一个问题应该引起警惕，那就是编译优化做得越多越深入，把错误引进编译结果（程序)的可能性也就越大。在打开了优化程序之后，应该重新运行回归测试集，就像你自己做了什么改动一样。

调整代码。只要数据有足够的规模，算法的正确选择就会显示它的作用。进一步说，算法方面的改进是跨机器、跨系统和跨语言的。但是，如果已经选择了正确的算法，程序的速度仍然是问题的话，下一步还能做的就是调整代码，整理循环和表达式的细节，设法使事情做得更快些。

第7.1节最后给出的isspam版本并没有经过仔细调整。这里我们要说明，通过翻转有关循环，可以进一步改进程序的性能。为帮助回忆，我们把原来的东西抄在这里：

```
for (j = 0; (c = mesg[j]) != '\0'; j++) {
    for (i = 0; i < nstarting[c]; i++) {
        k = starting[c][i];
        if (memcmp(mesg+j, pat[k], patlen[k]) == 0) {
            printf("spam: match for '%s'\n", pat[k]);
            return 1;
        }
    }
}
```

这个初始版本在经过优化编译后，运行我们的测试集需要6.6秒。在内部循环的循环条件里有一个数组下标(nstarting[c])，对于外层循环的各次重复执行，这个值都是固定的。我们可以把这个值存储在一个局部变量里，以避免反复重新计算：

```
for (j = 0; (c = mesg[j]) != '\0'; j++) {
    n = nstarting[c];
    for (i = 0; i < n; i++) {
        k = starting[c][i];
        ...
```

这使程序运行时间减少到5.9秒，大约提高了10%，这是调整代码能得到的典型加速情况。在这里还有一个可以揪出来的变量，starting[c]也是固定的，把它拉到循环之外可能也有所帮助。通过测试之后，我们发现这个改动并没有产生显著影响。这个情况在代码调整中也是很典型的，调整某些东西有作用，而调整另一些东西则没有作用，人必须自己设法确定各种情况。此外，对于不同的机器或者编译系统，情况可能又会不同。

对于垃圾邮件过滤程序，还有一个改造也是可以做的。在内层循环里，程序用整个模式

China-pub.com 下载

与字符串做比较，而算法实际上保证了第一个字符已经做过匹配。代码可以调整，让 memcmp 从一个字节之后开始做。我们试验了这个改进，得到 3%的加速。这个收获确实不大，但只需要修改程序中的三行，其中有一行在预先计算中。

不要优化那些无关紧要的东西。有时所做的代码调整毫无作用，这就是因为用到了一些不能产生差异的地方。首先应该确认你优化的代码正是那些真正耗费时间的东西。下面的故事可能是虚构的，但我们还是要说一说。有人对一个现已倒闭的公司的一种早期机器做分析，安装了一个硬件执行监控器，发现系统把 50%的时间花在执行同一个包含几条指令的序列上。工程人员构造了一条特殊指令来封装这个指令序列的功能，重新构造好系统之后，却没有看到任何变化。原来他们优化的是操作系统的空转循环。

我们到底应该在加快程序速度方面花多少时间？最重要的标准就是看所做的改造能否带来足够的效益。作为一个准则，在加快程序速度方面所花的时间不应该超过这种加速在程序的整个生存期间中获得的时间。按照这个规则，对 isspam的算法改造是值得做的：这个工作用掉大约一天时间，但在以后的每一天里都节约了 (还将继续节约)几个小时。从内层循环里去掉一个数组下标的收获没那么激动人心，但也是值得做的，因为这个程序是在为一个大机构提供一种服务。对于像垃圾邮件过滤程序或者程序库这样的公共服务程序，优化通常都是很值得的，而加快一个测试程序的速度几乎就没有什么价值了。如果有一个程序要运行一年，那么就应该从中挤压出你所能做到的一切。甚至在这个程序已经运行了一个月以后，如果你发现了一种能使它改进百分之十的方法，可能也值得从头再开始一次。

存在竞争对象的程序 —— 游戏程序、编译系统、字处理系统、电子表格系统和数据库系统等 —— 都应该纳入这个类别，因为商业上的成功常常就在这细微的差别上，至少是在公开的标准测试结果方面。

一件最重要的工作就是在做了修改之后对程序重新计时，以确认情况真正得到了改善。实际中也有这样的情况，两个修改分开来看对程序都有改进，但它们却会互相影响，对另一方的改进起负面作用。也存在这种情况，由于计时机制的误差太大，以至无法对修改程序的影响做出准确评价。即使是在单用户系统里，时间也可能有无法预计的起伏波动。如果内部计时器的变数是 10%(或至少报告给你的时间是如此)，那么对那些只产生了 10%改进的修改，我们就很难把它们与噪声区分开来。

7.4 代码调整

在发现热点之后，存在许多可以使用的能缩短运行时间的技术。这里提出一些建议，不过使用时都应该小心，使用之后应该做回归测试，确认结果代码还能工作。请记住，好的编译程序能为我们做其中的许多优化工作，而且你所做的事情有时也可能使程序变得更加复杂，从而可能妨碍编译程序的工作。无论你做了些什么，都应该通过测量，确定其作用，弄清它是否真的有所帮助。

收集公共表达式。如果一个代价昂贵的计算多次出现，那么就只在一个地方做它，并记录计算的结果。例如在第 1章里我们给出过一个宏，它在一行里用同样的值两次调用 sqrt，以这种方式计算距离。这个计算的实际作用是：

```
?    sqrt(dx*dx + dy*dy) + ((sqrt(dx*dx + dy*dy) > 0) ? ...)
```

应该只算一次平方根，而在两个地方使用得到的值。

China-pub.com
下载

如果一个计算出现在循环里，而它又不依赖任何在循环过程中改变的东西，那么就应该把它移到循环之外。就像我们把：

```
for (i = 0; i < nstarting[c]; i++) {
```

改造为：

```
n = nstarting[c];
for (i = 0; i < n; i++) {
```

用低代价操作代替高代价的。术语降低强度指的是用低代价操作代替高代价操作的那些优化。在过去，这个说法特别用来指用加法和移位操作取代乘法。今天再这样做，恐怕不会有太大收获了。除法和求余数比乘法慢得多，如果可以用乘倒数的方法来代替除法，或者在除数是 2 的幂时用掩码操作代替求余，则确实可能得到性能改进。在 C和C++ 里，用指针代替数组下标有可能提高速度，但是今天的大部分编译程序都已经能自动做这种优化了。用简单计算代替函数调用仍然是有价值的。平面上的距离由公式 sqrt(dx*dx+dy*dy)确定，因此，要确定两个点中哪个离得更远，按正规方式需要算两个平方根。但是，同样的判断也可以通过比较两个距离的平方做出来。

```
if (dx1*dx1+dy1*dy1 < dx2*dx2+dy2*dy2)
    ...
```

这样给出的结果与比较表达式的平方根完全一样。

另一个例子与文本模式匹配有关，我们在垃圾邮件过滤程序和 grep里都用到这种东西。如果模式的开始是个文字字符，我们可以用这个字符在输入文本中快速地查找下去。如果这样做也找不到匹配，更昂贵的查找机制根本就不需要调用。

铺开或者删除代码。循环的准备和运行都需要一定的开销。如果循环体本身非常小，循环次数很少，有一个更有效的方法，就是把它重写为一个重复进行的计算序列。例如，把下面的循环：

```
for (i = 0; i < 3; i++)
    a[i] = b[i] + c[i];
```

改造为：

```
a[0] = b[0] + c[0];
a[1] = b[1] + c[1];
a[2] = b[2] + c[2];
```

这样可以去掉循环的开销和特殊的分支操作，而这些都会造成执行流的中断，降低现代处理器的速度。

如果循环的次数很多，也可以做一些类似形式的变换，通过较少的重复来减少开销。例如把：

```
for (i = 0; i < 3*n; i++)
    a[i] = b[i] + c[i];
```

改为：

```
for (i = 0; i < 3*n; i += 3) {
    a[i+0] = b[i+0] + c[i+0];
    a[i+1] = b[i+1] + c[i+1];
    a[i+2] = b[i+2] + c[i+2];
}
```

China-pub.com 下载

请注意，只有在循环长度是步数大小的整倍数时，这种方式才是适用的。若不是这样，在代码最后还需要增加一些补充代码，这常常会成为潜藏错误的地方，有些情况下也可能使效率的收获重新丧失。

高速缓存频繁使用的值。以缓冲方式保存的值无须重新计算。缓存的价值来自于局部性。程序和人都有这种倾向，那就是重复使用最近访问过的值，或者是近旁的值，而对老的值和远距离的值则用得比较少。计算机硬件领域里广泛使用了缓存技术，给计算机增加缓冲存储器后，确实能大大地改进机器在速度方面的表现。对于软件，情况也是一样的。例如，Web浏览器应用缓存技术，保存页面和图形，以减少慢速的Internet数据传输。在前几年我们写的一个打印预览程序里，如½这样的非字母字符必须通过查表方式处理。我们统计了这些特殊字符的使用情况，发现大部分使用都是用这种字符排成一个行，用的是同一字符的长长的序列。将最近用过的一个字符缓存起来，就能大大提高程序对典型输入的处理速度。

最好是把缓存操作对外部隐蔽起来。这样，这种操作除了使程序运行得更快外，不会对程序其他部分有任何影响。在上面的打印预览程序的例子里，列出一个字符的函数并没有改变，它一直是：

```
drawchar(c);
```

函数drawchar原来的版本是直接调用show(lookup(c))。使用缓存技术的函数定义了一个内部的静态局部变量，用于记录前一个字符值。有关代码是：

```
if (c != lastc) { /* update cache */
    lastc = c;
    lastcode = lookup(c);
}
show(lastcode);
```

写专用的存储分配程序。经常可以看到这种情况，程序里的惟一热点就是存储分配，表现为对malloc和free的大量调用。如果程序中需要的经常是同样大小的存储块，采用一个特定用途的存储分配器取代一般的分配器，有可能使速度得到实质性提高。这种特定的存储分配器调用malloc一次，取得基本存储块的一个大数组，在随后需要时一次送出去一块，这是一个代价很低的操作。释放后的存储块接在一个自由表的最后，这使它们可以立即重新投入使用。

如果所需要的块在大小上差不多，你也可以用空间来交换时间，总是分配能够满足最大需求的块。对所有长度不超过某个特定值的字符串都使用同样大小的块，这是管理短字符串的一种有效方法。

有些算法可以采用栈方式的存储管理，先完成了一系列的存储分配，然后将整个集合一下子释放掉。在这种情况下，分配器可以先取得一个大块，而后像用一个栈似的使用它，需要的时候将分配的项目压入，结束时通过一个操作就把它们都弹出去。有些C函数库里为这种分配方式提供了一个alloca函数，但它不是标准的。这个函数用一个局部栈作为存储资源，当调用alloca的函数返回时，自动释放掉所有的项目。

对输入输出做缓冲。缓冲方式使数据传输操作以成批的方式完成，这能使频繁操作所造成的负担减到最小，并使代价昂贵的操作只在不可避免时才进行，使这种操作的代价能分散到许多数据值上。例如，当C程序调用printf操作时，有关字符被存入一个缓冲区，而不是直接传给操作系统，直到缓冲区满，或者是显式地执行刷新操作。操作系统本身也可能推迟向磁

China-pub.com
下载

盘写数据的动作。这种方式也有缺点。为了使数据变成可见的，就必须对输出缓冲区做刷新。最糟糕的情况是，如果程序垮台，驻留在缓冲区里的数据就会丢失。

特殊情况特殊处理。通过使用特殊代码去操作同样大小的对象，特殊用途的分配器可以比通用分配器节约时间和空间开销，还可能减少碎片问题。在 Inferno系统的图形库里，基本的draw函数先用最简单、最直截了当的方式写出来。在这个东西能工作之后，再把对各种各样情况的优化(通过轮廓文件来选择)以一次一个的方式加进来。这样，每次都可以用简单版本与优化版本对照测试。到最后，也只有十来个特殊情况做了优化。产生这种情况的原因是，通过分析绘图函数调用的动态情况，我们发现其中最大量的还是做字符显示，因此没必要对所有情况都写出巧妙的代码。

预先算出某些值。有时可以让程序预先计算出一些值，需要时拿起来就用，这也可能使程序运行得更快些。我们已经在垃圾邮件过滤程序里看到过这种方式，在那里首先计算出所有strlen(pat[i])的值，将它们存入数组元素patlen[i]中。如果一个图形程序需要反复计算某个数学函数，例如正弦函数，但又只需要对某个离散集合里的某些值做计算，例如对所有整数角度做计算。那么预先算出这个 360项的表(甚至是把它作为数据提供给程序)，需要时直接通过下标取用，肯定能够快很多。这又是一个用空间交换时间的例子。用数据代替代码，或者在编译时预先完成某些计算的可能性非常多，这些都可以节约时间，有时甚至还能节约空间。例如，像isdigit这一类函数，通常是预先定义一个位标志的表，然后通过下标方式实现的，并不是通过一系列的测试来实现。

使用近似值。如果精度不太重要，那么就尽量使用具有较低精度的数据类型。在老的或者小的机器上，或者在那些采用软件模拟方式实现浮点数的机器上，单精度浮点运算通常比双精度运算更快一些，所以，用 float而不是double就有可能节省时间。一些新型的图形处理器也采用了类似技巧。IEEE的浮点数标准要求“适度下溢”，把它作为可表达的数值在最低端的计算方式，但是这种计算方式是昂贵的。对于图像而言，这种特性就完全没有必要，直接将它截断为0要快得多，也是完全可以接受的。这样做，不仅能在数值出现下溢时节省时间，还可以简化整个算术运算硬件。采用整数的 sin和cos函数是使用近似值的另一个例子。

在某个低级语言里重写代码。低级语言程序的效率可能更高，不过这样做也要付出代价，那就是程序员的时间。如果把 C++或Java程序里的某些关键部分用 C语言重写，或者用某种编译语言的程序取代一个解释性脚本程序，都可能使程序加快许多。

有时，使用依赖于机器的代码也可能得到显著的加速效果。但这是最后一招，是不该轻易采用的，因为它破坏了可移植性，也使将来的维护和修改都变得更加困难。实际中也常常出现这种需要，在这种情况下，用汇编语言描述的操作应该是些相对较小的函数，而且应该嵌入一个库里，memset、memmove以及一些图形操作都是这方面的典型例子。这时应该采用的方式是，首先在某种高级语言里以尽可能清晰的方式写出代码，并通过像我们在第 6章对memset描述的那样做好测试，确定它是正确的。这是你的可移植版本，它能在任何地方工作，可惜是慢了一点。当你遇到一个新环境时，就可以从这个已知能够工作的版本开始入手。如果你写好了一个汇编语言版本，就可以利用可移植版本对新版本做彻底的测试。当测试发现错误时，不可移植的版本总是最大的疑点。有一个可做比较的实现确实是非常舒服的。

练习7-4 要使memset一类函数运行得更快，一个方法就是让它一次写一个字，而不是一次写一个字节。这样做可能与硬件匹配得更好，也能以 4或者8的因子减少循环开销。不利的一

China-pub.com 下载

面是，这时需要处理各种端点情况，目标位置可能不是开始在与字对齐的边界点，长度也可能不是字大小的整数倍等等。写一个memset版本实现这种优化。将它的性能与现存的库版本，以及直截了当的一次一个字节的版本做些比较。

练习7-5 为C语言的字符串专门写一个存储分配器smalloc，它对小字符串使用一个专用分配器，对大字符串则直接调用malloc。在这种情况下，你可能需要定义一个struct，以表示串的两种情况。你怎样确定何时从调用smalloc转到malloc？

7.5 空间效率

存储空间过去一直是最贵重的计算资源，而且总是短缺的，想从这种没有多少油水的地方榨出许多东西来，就会形成特别坏的程序设计。声名狼藉的“两千年问题”是经常被人提起的典型例子。当存储器真正稀缺的时候，甚至为存储19所需要的那两个字节也被认为是太昂贵了。无论缺少空间是否是其中真正的原因(实际上，这种代码也直接反映了人们在日常生活中写日期的习惯，其中的世纪总是被省略的)，这件事总是说明了一个短视的优化所具有的内在危险性。

无论如何，时代已经不同了，主存和二级存储器都已经便宜得令人感到惊异。这样，优化空间的第一要义应该和改善速度完全一样：别为它费心。

实际中，确实还是存在某些情况，在那里空间效率是非常重要的。如果一个程序无法放进可用的存储里，对它的各部分必须反复做页面倒换，而这又会使性能降低到令人无法容忍的程度。我们都熟悉这种情况，新版本的软件大量地挥霍着存储器。一个可悲的现实是，软件升级通常都伴随着存储器的大量销售。

使用尽可能小的数据类型以节约存储。提高空间效率的一个步骤是做些小修改，使现有存储能使用得更好，例如使用能满足工作需要的最小数据类型。比如说，如果合适的话，可以用short取代int，这是2-D图形系统常用的一种技术，因为16位一般足以处理屏幕坐标的可能取值范围。或许是用float代替double，这带来的潜在问题是精度损失，因为float一般只能保持6到7位十进制数字。

对于这些修改，或者其他类似情况，程序也必须做一些改动，值得注意的是 printf和(特别是)scanf语句的格式描述。

这种方法的逻辑延伸是将信息编码到字节里面，甚至到若干个二进制位里，如果可能的话就只使用一个位。请不要使用C或C++的位域，它们是高度不可移植的，而且倾向于产生大量的低效代码。你应该把所需要的操作都封装在函数里面，在这些操作中利用移位和掩码，取出或者设置字或字的数组里的位。下面的函数能够从一个字中间取出连续一段位值：

```
/* getbits:  get n bits from position p */
/*  bits are numbered from 0 (least significant) up */
unsigned int getbits(unsigned int x, int p, int n)
{
        return (x >> (p+1-n)) & ~(~0 << n);
}
```

如果这种函数太慢了，你可以用在本章前面描述的技术改进。 C++ 语言的操作符可以重载，这样就可以把对二进制位的访问做成正规的下标形式。

不存储容易重算的东西。与代码调整类似，这方面的修改也不太重要。最重要的改进应该是来自好的数据结构，或许还要伴随着算法的修改。这里有一个例子：许多年前一位同事来找

China-pub.com
下载

作者之一，该同事正试图做一个矩阵计算，但是这个矩阵实在太大了，他必须关掉计算机，重新装入一个简化的操作系统之后才能进行处理。这种操作方式实在是个噩梦，该同事想知道是不是有其他办法。我们问矩阵是什么样的，了解到的情况是，矩阵里存储着整数值，而其中大部分都是零。实际上，非零矩阵元素还不到5%。这立即提醒我们采用另一种表示方式，其中只存储矩阵的所有非零元素，m[i][j]一类形式的矩阵元素访问用函数调用m(i,j)取代。有许多存储这种数据的方法，最简单的就是用指针数组，一个指针对应矩阵的一行，它指向一个压缩数组，其元素是列号和对应的数据元素值。对于每个非零元素而言，采用这种方法耗费了较大空间，但就整个矩阵来说，所需要的空间却小多了。虽然每次元素访问都会慢一些，但从整体看则远快于重装操作系统。故事的结尾是，该同事采纳了这个建议，满意地离开了。

我们采用同样技术解决过另一个现代版本的同样问题。在一个无线电通信的设计系统里，需要表示一个很大的地理区域(边长100到200公里，解析度是100米)中的地形数据和无线电信号强度。存储这样巨大的矩形数组已经超出了目标机器的存储能力，必然会导致无法接受的页倒换开销。但是，在一些大片的部位中，地形和信号强度又几乎是一样的，因此我们采用一种层次性的表示方法，把具有同样值的区域结合成一个单元，这就使问题变得可以把握了。

这种问题有许许多多不同的变形，由此带来各种各样的特殊表示方式。这些方式有一个共同的基本思想：采用隐含的方式或者一种共同的形式存储公共值，在其他值上花更多的空间和时间。如果其中共性最强的值真正是共有的，这个招数就奏效了。

在组织程序时，复杂类型的特殊数据表示方式总是应该隐蔽在一个类里，或者隐藏在一集对私有数据类型操作的函数里。采取了这种预防措施，我们就能保证，当某个表示方法改变时，程序其他部分完全不会受影响。

空间效率方面的考虑有时也会表现在信息的外部表示方面，与转换和存储都有关系。一般地说，如果可能，最好以文本形式存储信息，而不要采用某种二进制表示形式。文本是可移植的，容易读，可以任由各类工具处理。二进制表示则完全没有这些优点。倾向二进制的论据通常都基于“速度”，实际上这种说法是很值得怀疑的，因为在文本形式和二进制形式之间的差异可能没那么大。

空间效率的获得往往要付出时间代价。在某个应用里，需要把大的图像从一个程序传给另一个程序。有一种简单的图像格式叫PPM，其典型大小是一兆字节左右。由此我们设想，如果把图像编码到压缩的GIF格式，其典型大小是50K字节，这样传输工作一定进行得更快。但是，完成到GIF的编码以及对应的解码要花费时间，与传送一个短文件节约的时间差不多，这样我们就不会有什么收获。处理GIF格式的代码大约有500行那么长，而处理PPM的源程序大约是10行。为了维护的方便，GIF编码方式被放弃了，在这个应用里继续使用PPM方式。当然，如果该文件是通过一个速度很慢的网络传输的，权衡结果就可能不同，GIF编码方式很可能更为有效。

7.6 估计

要预先估计一个程序能运行得多么快，通常是非常困难的，而要估计某个特殊程序语句或者机器指令的时间代价，那就是双倍的困难了。然而，为一个语言或者系统做一次代价模拟却是比较简单的，它至少能使你对各种重要操作花费的时间有一个粗略的概念。

China-pub.com 下载

对于常规的程序设计语言，可以用一个为有代表性的代码序列做计时的程序。在这里实际上存在着许多困难，比如很难取得可以重现的结果、难以消除无关开销等等。不过，这至少是一种不费多少事就能得到一些有用信息的途径。例如，我们用一个 C和C++ 代价模拟程序估计了一些独立语句的代价，采用的方法就是把它们放在循环里，运行成百万次，然后计算出平均时间。我们用一台250 MHz的MIPS 10 000产生这些数据，操作代价以纳秒计算。

```
Int Operations
  i1++                          8
  i1 = i2 + i3                 12
  i1 = i2 - i3                 12
  i1 = i2 * i3                 12
  i1 = i2 / i3                114
  i1 = i2 % i3                114

Float Operations
  f1 = f2                       8
  f1 = f2 + f3                 12
  f1 = f2 - f3                 12
  f1 = f2 * f3                 11
  f1 = f2 / f3                 28

Double Operations
  d1 = d2                       8
  d1 = d2 + d3                 12
  d1 = d2 - d3                 12
  d1 = d2 * d3                 11
  d1 = d2 / d3                 58

Numeric Conversions
  i1 = f1                       8
  f1 = i1                       8
```

整数计算是极快的，但除法和取模除外。浮点运算也很快，甚至可能更快。对于在浮点运算比整数运算昂贵得多的年代里成长起来的人们而言，看到这些一定很吃惊。

其他基本操作也都相当快，包括函数调用，至少在下面一组的最后三行里：

```
Integer Vector Operations
  v[i] = i                     49
  v[v[i]] = i                  81
  v[v[v[i]]] = i              100

Control Structures
  if (i == 5) i1++              4
  if (i != 5) i1++             12
  while (i < 0) i1++            3
  i1 = sum1(i2)                57
  i1 = sum2(i2, i3)            58
  i1 = sum3(i2, i3, i4)        54
```

但是，输入输出操作就不那么便宜了，大部分库函数也是如此：

```
Input/Output
  fputs(s, fp)                270
  fgets(s, 9, fp)             222
  fprintf(fp, "%d\n", i)     1820
  fscanf(fp, "%d", &i1)      2070

Malloc
  free(malloc(8))             342
```

China-pub.com
下载


```
String Functions
  strcpy(s, "0123456789")          157
  i1 = strcmp(s, s)                176
  i1 = strcmp(s, "a123456789")      64

String/Number Conversions
  i1 = atoi("12345")               402
  sscanf("12345", "%d", &i1)      2376
  sprintf(s, "%d", i)             1492
  f1 = atof("123.45")             4098
  sscanf("123.45", "%f", &f1)     6438
  sprintf(s, "%6.2f", 123.45)     3902
```

这里有关于malloc和free的时间，它可能对真实计算没有多少指导意义，因为刚刚分配后立即释放并不是一种典型模式。

最后是数学函数：

```
Math Functions
  i1 = rand()                      135
  f1 = log(f2)                     418
  f1 = exp(f2)                     462
  f1 = sin(f2)                     514
  f1 = sqrt(f2)                    112
```

在不同的硬件上，有关的具体数据应该是不同的。但是，作为一种大致走向，这些数据还是可以用于粗略估计某些事情可能需要多长时间，或者比较 I/O和基本操作之间的相对代价，或者确定是否有必要重写一个表达式、使用一个在线的函数，等等。

在这里也还有很多变数。一个是编译系统优化的级别。现代编译系统完全可能发现一些能难倒大部分程序员的优化。进一步的问题是，目前的 CPU极端复杂，只有好的编译程序才能充分利用CPU能力方面的优势，同时发送多条指令，将它们的执行过程流水线化，在需要之前提取出有关的指令和数据，以及完成许多其他的类似事项。

对有关执行情况难以做出预计，另一个重要的原因是计算机的体系结构。缓冲存储器极大地改变了机器速度，灵巧的硬件设计几乎能够掩盖这样的事实：主存储器实际上比缓冲存储器慢很多很多。处理器主频，例如“ 400 MHz ”，只有提示的意义，并没有讲出全部故事。我们的一台老的200 MHz奔腾机比另一台更老的100 MHz奔腾机明显慢了许多，就是因为后者有一个很大的二级缓存而前者没有。另外，不同代的处理器，即使它们的指令系统相同，为完成一个特定指令所用的时钟周期也可能是不同的。

练习7-6 为你身边的各种计算机或编译系统建立一个估计基本操作代价的测试集，并研究它们在性能方面的相似性和差异。

练习7-7 为C++ 语言的一些高级操作建立一个代价模型，有关的特征可能包括：类成员的构造、复制和删除，成员函数调用，虚函数，在线函数， iostream库，STL库。这个练习完全是开放性的，请集中于一小批有代表性的操作。

练习7-8 对Java重做上述练习。

7.7 小结

如果你已经选择了正确的算法，那么只有到了写程序的最后，才有可能要关心执行的优

China-pub.com 下载

化问题。如果你必须做这件事情，进行工作的基本循环应该是：测量，把注意力集中到若干个做一点修改就能产生最大改进的地方，验证你所做修改的正确性，然后再测量。在能停下的时候立刻停下来。在这里还应该保留那个最简单的版本，作为计时和验证正确性的基础。

当你要着手改进一个程序的速度或者空间耗费时，一个很好的做法是先建立一些基准测试和问题，这样你就很容易做出估计，并可以为自己保存性能的变化轨迹。如果对你的工作已经有某些标准的基准测试，那么就应该使用它们。如果程序是相对自足的，可以采用的一种办法是找到或者建立一集典型输入，这些也可以成为测试集的一部分。实际上，这也正是那些为编译系统、计算机或者其他类似东西而使用的基准测试集的起源。例如， Awk带着大约20个小程序，它们的全体覆盖了大部分常用的语言特征。这些程序运行时要处理很大的输入文件，以保证能够计算出同样结果，又没有引进性能缺陷。我们还有一集标准的大数据文件，它们用于做计时测试。在某些情况下，如果这些文件有特别容易辨识的特征也很有帮助，例如其大小是10或者2的幂。

对于基准测试，也可以用我们在第 6章讨论测试时提出的测试台来管理。计时测试可以自动执行，其输出中应该包含足够多的标志，使它们能够被理解和复制。还应该保存测试记录，以便于研究其中重要的趋势和变化。

当然，要想做出好的基准测试也是很困难的。大家都知道，某些公司设法调整它们的产品，使之在基准测试中表现良好。因此，对所有基准测试带着点疑问是很明智的。

补充阅读

我们关于垃圾邮件过滤程序的讨论是基于 Bob Flandrena和Ken Thompson的工作。他们的过滤程序包括正则表达式，以便于描述复杂的匹配，还包括根据与串匹配的情况自动对消息进行分类(肯定是垃圾邮件、可能是垃圾或不是垃圾)。

Knuth关于轮廓文件的文章“ FORTRAN程序的实证研究 ”(An Empirical Study of FORTRAN Programs)发表在《软件：实践和经验》(Software: Practice and Experience第1卷第2期pp105~133，1971)。该文的核心是对一些从废纸篓里和某计算机中心机器上公开可见的目录里翻出来的程序的统计分析。

在Jon Bentley的《程序设计精萃》和《更多的程序设计精萃》 (Programming Pearls, More Programming Pearls，Addison-Wesley，1986和1988)里，有一些关于算法和代码调整改进的极好实例，对使用测试台改善性能和使用轮廓文件也有很好的讨论。

Rick Booth的《内部循环》(Inner Loops，Addison-Wesley，1997)是关于调整PC程序的一本很好的参考书。但是，由于处理器更新得太快，书中的某些细节很快就过时了。

John Hennessy和David Patterson关于计算机体系结构的一套书，例如《计算机组织和设计：硬件/软件接口》，(Computer Organization and Design: The Hardware/Software Interface，Morgan Kaufman，1997)深入论述了当前计算机性能方面的许多问题。

China-pub.com
下载

第8章 可移植性

最后，标准化与常规相仿，能成为强有力秩序的另一种具体化形式。但是它又与常规不同，它已经被现代建筑学公认为是我们技术的浓缩产物，因此以其潜在的支配地位和蛮横特征而令人恐惧。

Robert Venturi，《建筑学中的复杂和矛盾》

写出能够正确而有效地运行的软件是很困难的。因此，如果某个程序能在一个环境里工作，当你需要把它移到另一个编译系统，或者处理器，或者操作系统上时，不会希望再重复做太多原来已经做过的工作。最理想的情况是什么都不用改。

这种理想就是程序的可移植性。实际上，“可移植性”常被用来指一个更弱的概念，其意思是说，与凭空写出这个程序相比，对它做些修改挪到另一个地方将更容易一些。这种修改越容易做，我们就说这个程序的可移植性越强。

你可能会奇怪，为什么我们还要为可移植性费心呢？如果软件都是准备在某些特定条件下，在一个特定环境里运行的，为什么还要在使它具有更广泛的可接受性方面白费精力呢？首先，任何成功的程序，几乎总是注定要被以原来不曾预料的方式，用到从未想到的地方去。把一个软件构造得比它的规范更一般些，结果就会是以后的更少维护和更好使用。第二，环境总是在变。当编译系统、操作系统或者硬件升级的时候，其特性可能就不同了。程序对特殊特征的依赖越少，它也就越少可能崩溃，也越容易适应改变以后的环境。最后，也是最重要的，可移植的程序总是更优秀的程序。为把程序构造得更具有可移植性的努力也会使它具有更好的设计，更好的结构，经过更彻底的测试。一般地说，可移植程序设计的技术与优良程序设计的技术是密切相关的。

当然，可移植性的程度也应当根据现实来考虑。不存在什么绝对的可移植程序，只有那种已经在足够多的环境里试验过的程序。但是，我们仍然可以把可移植性作为一个目标，力图去开发那种几乎不用修改就能够运行在任何环境上的软件。甚至当这个目的不能完全达到时，在程序构造过程中花在可移植性上的功夫也将会得到回报，例如在这个软件需要升级的时候。

我们的看法是：应该设法写这样的软件，它能工作在它必须活动于其中的各种标准、界面和环境的交集里。不要为纠正每个移植性问题写一段特殊代码，正相反，应该修改这个软件本身，使它能够在新增加的限制下工作。利用抽象和封装机制限制和控制那些无法避免的不可移植代码。通过将软件维持在各种限制的交集里面，局部化它的系统依赖性，这样你的代码在被移植后仍将更加清晰、更具通用性。

8.1 语言

盯紧标准。得到可移植代码的第一步当然是使用某种高级语言，应该按照语言标准(如果有的话)去写程序。二进制不可能很容易地移植，但是源代码可以。当然，即使这样做也还会有问

China-pub.com 下载

题，在编译系统如何将源程序翻译到机器指令的方式方面，也可能有些东西没有精确定义，对标准语言也是如此。广泛使用的语言只存在一个实现的情况是罕见的，通常都有多个编译系统提供商，对于不同操作系统，又有不同的版本，还有随着年月更替而不断出现的不同发行版本。它们对你的源代码可能做出不同解释。

为什么语言标准不是一个严格定义呢？有时标准是不完全的，对某些特性之间的相互作用没有给出定义。有时标准会有意地不对某些东西做出定义，例如，C和C++ 语言里的char类型可以是有符号的或是无符号的，而且不必正好是8位。把这些事项留给写编译系统的人去解决，有可能产生出更有效的实现，或者避免语言对它能在其上运行的硬件提出太多限制。当然，这种做法可能给程序员带来困难。政治上和技术上的相容性问题也可能导致某种妥协，使标准对某些细节不做具体定义。最后，语言都是极端复杂的，编译系统也很复杂，理解中可能出错，实现里面也可能有毛病。

有时语言根本没有经过标准化。C语言正式的ANSI/ISO标准在1988年颁布，而ISO的C++的标准直到1998年才被批准，在我们写这些的时候，还没有一个在用的编译系统支持这个正式标准。Java是更新一些的语言，与标准化的距离还有许多年。一个语言标准的开发通常总是要等到这个语言已经有了许多不同的、互相冲突的实现，有了进行统一的需求的时候；此外，它也必须已经被广泛使用，值得付出标准化的代价。在这期间，还是有许多程序需要写，有许多环境需要支持。

综上所述，虽然在给人的印象上，参考手册和标准是一种严格规范，但它们从来也不能完全地定义一个语言。这样，由不同实现给出的就可能都是合法的，但却又是互不相容的解释。有时甚至实现中还存在错误。在我们刚开始写这一章的时候，发现过一个很有意思的小问题。下面的外部说明在C或者C++ 里都是不合法的：

```
? *x[] = {"abc"};
```

我们测试了十来个编译系统，只有不多几个正确诊断出 x缺少char类型说明符；好几个系统给出类型不匹配的警告(它们明显是采用了语言的老定义，错误地推论出 x是一个整型指针的数组)，还有几个在编译这段非法代码时一点牢骚也不发。

在主流中做程序设计。某些编译系统不能辨识上面的错误，这当然很不幸，也说明了与可移植性有关的一个重要问题。任何语言都有黑暗的角落，在那里实践会出现分歧。例如 C和C++的位域，回避它们是比较稳健的做法。我们应该只使用那些在语言的定义里毫无歧义、而且又很容易理解的东西。这类特性更可能是到处都能用的，也会在任何地方都具有同样的行为方式。我们称这种东西为语言的**主流**。

要想确定哪里是主流有时也非常困难，但我们很容易辨明哪些东西是在主流之外。一些全新的东西，例如C里面的 // 注释或者complex类型；或者那些特定的与某种体系结构有关的东西，如near或者far；它们一定会带来麻烦。如果某个特性是如此地不寻常、不清楚，为了理解它，你在阅读定义时必须去咨询一个“语言律师”，一个专家，那么请不要用它。

在下面的讨论里，我们要把注意力集中在 C和C++，它们是常被人用来写可移植程序的通用程序语言。C语言标准已经有了十几年的历史，这个语言也是很稳固的。人们正在为建立一个新标准而工作，所以，也可以说是喷发在即。在另一方面，C++ 标准则是全新的，各种实现还没有时间汇合到一起。

什么是C语言的主流？这个术语常被用来指那些已经建立起来的语言使用风格，但我们最

China-pub.com
下载

好还是为将来做点准备。例如，原来的C版本并不要求函数原型，说明sqrt是一个函数的方式是写：

```
?	double sqrt();
```

这里定义了函数的返回值类型，对参数则什么也没说。ANSI C增加了函数原型，它把所有东西都刻画清楚了：

```
double sqrt(double);
```

ANSI C标准要求编译系统也要接受原来的语法，不过你无论如何也应该为自己的所有函数都写出原型。这样做能保证得到更安全的代码，保证所有的函数调用都得到完全的检查。此外，如果界面改变了，编译系统也能够捕捉到它们。如果你的代码里有调用：

```
func(7, PI);
```

如果函数func没有原型，那么编译系统就很可能不检验func调用的正确性。如果后来有关的库改变了，func改变为有了3个参数，必须修改软件这件事很可能被忽略，因为C语言的老语法关闭了对函数参数的类型检查。

C++ 是一个更庞大的语言，有最新的标准，所以它的主流就更难辨别清楚。例如，虽然我们希望STL能够变成主流，但这件事一时半会是不可能实现的。况且当前的一些实现根本就不支持它。

警惕语言的麻烦特性。我们已经提过，标准里常常有意遗留下一些东西，不给以定义或者不加以清楚的说明，通常这是为了给写编译系统的人更大的自由度。这种东西的列表实在太长了。

数据类型的大小。在C和C++ 里，基本数据类型的大小并没有明确定义，给出的仅仅是下面这些规则：

```
sizeof(char) ≤ sizeof(short) ≤ sizeof(int) ≤ sizeof(long)
sizeof(float) ≤ sizeof(double)
```

此外，还规定char至少必须有8位，short和int至少是16位，long至少应该是32位。这里有许多不加保证的性质。甚至没要求一个指针值应该能够放进一个int中。

很容易确定在一个特定编译系统里的各种类型的大小：

```
/* sizeof: display sizes of basic types */
int main(void)
{
    printf("char %d, short %d, int %d, long %d,",
        sizeof(char), sizeof(short),
        sizeof(int), sizeof(long));
    printf(" float %d, double %d, void* %d\n",
        sizeof(float), sizeof(double), sizeof(void *));
    return 0;
}
```

在我们正常使用的大部分机器上，输出都是一样的：

```
char 1, short 2, int 4, long 4, float 4, double 8, void* 4
```

但是完全可能有其他情况。例如，在某些64位机器上产生的是：

```
char 1, short 2, int 4, long 8, float 4, double 8, void* 8
```

在早期的PC机上，典型的输出是：

```
char 1, short 2, int 2, long 4, float 4, double 8, void* 2
```

China-pub.com 下载

对于早期的PC机，硬件支持多种指针。为处理这些麻烦事，人们发明了一些指针修饰符，如far和near等，它们都不是标准的，但这些魔鬼保留字仍然在纠缠着当前的编译系统。如果在你的编译系统里基本类型的大小能够改变，或者你使用几种有着不同数据类型大小的机器，那么就应该在这些不同配置之下试试你的程序。

标准头文件sdtdef.h里定义了一些类型，它们对可移植性能有些帮助。这其中最常用的是size_t，它是一个无符号的整数类型，是 sizeof运算符的返回类型。有些函数(例如strlen)返回这种类型的值；也有不少函数(如malloc)要求这种类型的参数。

根据从上面这些情况取得的经验， Java对所有基本数据类型都明确地定义了大小： byte为8位，char和short为16位，int是32位，而long是64位。

我们将忽略对浮点数计算方面各种问题的讨论，关于这个问题可以写出整本书。幸运的是，大多数现代机器都支持 IEEE的浮点硬件标准，这也使浮点算术的有关特性都合理地定义清楚了。

求值顺序。在C和C++ 语言里，有关表达式中的运算对象、副作用产生以及函数参数的求值顺序都没给出明确定义。例如，赋值语句

```
?    n = (getchar() << 8) | getchar();
```

这里的第二个getchar也有可能率先执行，表达式的书写顺序不一定就是它们的执行顺序。在语句

```
?    ptr[count] = name[++count];
```

里，count的增值可能在它被用做ptr的下标之前或者之后完成。同样，在

```
?    printf("%c %c\n", getchar(), getchar());
```

里，第一个输入字符可能被打印在后面(未必是第一个打印)。在

```
?    printf("%f %s\n", log(-1.23), strerror(errno));
```

里，errno也可能在log调用之前就求了值。

对于各种表达式如何求值，实际也有些规则。按照定义，在每个分号处，或者到一个函数被实际调用的时刻，所有的副作用或者函数调用[⊖]都必须完成。运算符 &&和 || 总是从左到右执行，而且只执行到表达式的真值能够确定时为止 (包括有关副作用，也只到此时为止)。在运算符 ?: 里，条件先被求值(包括副作用)，此后，后面两个表达式中只有一个被求值。

Java对求值的顺序有严格定义，它要求所有的表达式，包括副作用，都严格地从左向右进行。然而，有一本权威性手册中提出建议，不要写“过分”依赖这种行为的代码。这是一个合理建议，如果存在着把Java转换到C或者C++ 的可能性，情况就更是如此，因为 C、C++ 都没有如上的保证。语言间的转换是对可移植性的一种极端性测试，虽然这种东西并不太有用。

char的符号问题。char数据类型到底是有符号还是无符号的， C和C++ 并没有对此给出明确规定。在结合了char和int的代码里，这个问题就有可能造成麻烦，例如 getchar() 函数得到int值，调用它的代码就可能出问题。假设你写了：

```
?    char c;    /* should be int */
?    c = getchar();
```

如果char是无符号的， c值将在0和255之间；而如果char是有符号的，对于2补码机器上8位字符的最一般配置情况， c的值将在-128与127之间，这种情况将造成一些影响。例如

⊖ 这个函数调用指的是作为外层函数或运算符的参数的那些函数调用。——译者

China-pub.com 下载

我们用字符作为数组的下标，或者用它去与 EOF做比较(EOF通常在stdio.h里定义为-1)。在6.1节的一个例子里，我们改正了原来代码的一个边界条件，在那里遇到的就是这种代码。如果char是无符号类型，条件s[i]==EOF将总是假的：

```
?   int i;
?   char s[MAX];
?
?   for (i = 0; i < MAX-1; i++)
?       if ((s[i] = getchar()) == '\n' || s[i] == EOF)
?           break;
?   s[i] = '\0';
```

假设getchar返回EOF，存入s[i]的值将是255(即0xFF，这是把 -1转换到unsigned char所得到的结果)。如果char是无符号的，在与EOF做比较时这个值还是255，这必然导致比较的失败。

即使char是有符号的，上面的代码同样也不正确。在执行中遇到 EOF值时，这里的比较就会成功。但是，在这种情况下正常输入的字节 0xFF也会被当作EOF，从而导致这个循环不正确地结束。所以，无论char的符号情况如何，你都必须把getchar的返回值存入一个int，以便与EOF做比较。下面是按可移植方式写出的同一循环：

```
int c, i;
char s[MAX];

for (i = 0; i < MAX-1; i++) {
    if ((c = getchar()) == '\n' || c == EOF)
        break;
    s[i] = c;
}
s[i] = '\0';
```

Java没有unsigned修饰符。在这里所有的整型都是有符号的，只有 16位char类型是无符号的。

算术或者逻辑移位。在对有符号的量用运算符>>做右移时，这个移位可以是算术的(符号位将在移位的过程中复制传播)，也可以是逻辑的(移位中空出的位被自动补0)。同样，根据从C和C++学到的经验，Java把>>保留作算术右移，为逻辑右移另外提供了一个>>>。

字节顺序。在类型short、int和long里，字节的顺序并没有规定，具有最低地址的字节可能是最高位的字节，也可能是最低位的字节。这是一种依赖硬件的特性，在本章的后面部分我们将做详细讨论。

结构或类成员的对齐。在结构、类或者联合里，各个成分的对齐方式并没有规定。这里只规定各成分一定按说明的顺序排列。例如，在下面的结构里：

```
struct X {
    char c;
    int i;
};
```

成分i的地址与结构开始位置的距离可能是 2、4或者8个字节。很少有机器允许 int存储在奇数边界上，一般都要求占据 n个字节的基本数据类型存放在 n字节的边界上。例如， double一般是8个字节长，所以需要存储在 8的倍数的地址上。在这之上，写编译程序的人还可能再做进一步调整，例如可能为了执行性能做进一步的强制对齐。

你绝不能假定在一个结构里各成员占着连续的存储区。对齐限制实际上会造成结构中的

China-pub.com 下载

“空洞”，在上面的struct X里，至少存在一个字节的未用空间。空洞的存在说明了一个结构可能比它成员的大小之和更大一些，在不同的机器上又可能具有不同大小。如果你需要分配空间，用以存放上述结构，就必须去申请 sizeof(struct X) 个字节，而绝不应该是sizeof(char)+sizeof(int)个。

位域。位域对机器的依赖太强，无论如何都不应该用它。

从上面关于危险特征的长表里可以总结出下面的规则，不要使用副作用，除了在很少的几个惯用结构里，例如：

```
a[i++] = 0;
c = *p++;
*s++ = *t++;
```

不要用char与EOF做比较。总使用sizeof计算类型和对象的值。决不右移带符号的值。你所用的数据类型应该足够大，足够存储你希望放在里面的值。

用多个编译系统试验。人们很容易认为自己已经理解了可移植性，但是编译系统能看出某些你没有看到的问题。进一步说，不同编译程序有时会对你的程序有不同的看法，因此，你应该尽量利用这些帮助。打开编译程序所有的警告开关，在同一机器或者不同机器上试用多种编译系统，试用一个C++ 编译系统处理你的C程序。

由于不同编译系统在接受的语言方面可能有差异，所以，即使你的程序能用一个编译系统完成编译，你甚至都无法保证它在语法上是正确的。如果几个编译系统都能接受你的代码，那么你的胜算就大得多。对于这本书里的每个 C程序，我们都在三个互不相干的操作系统(Unix，Plan 9和Windows)上用三个C编译系统和若干C++ 编译系统处理过。这是一个清醒的试验，它确实挖出了数十个移植性错误，而这些问题通过人工的大量仔细检查也没有发现。所有这些错误的更正都是非常简单的。

当然，编译系统本身也会引起可移植性问题，因为它们可能对语言中未加规定的行为做出各自不同的选择处理。即使如此，我们的途径仍然是有希望的。我们应该努力去构造这样的软件，使它们的行为能够独立于具体的系统、环境或者编译之间的差异，而不应该去写那种以某种方式展现这些差异情况的代码。简而言之，我们应该设法回避那些很有可能变动的性质和特征。

8.2 头文件和库

头文件和库提供了许多服务，它们是基本语言的扩充。有关例子包括 C里通过stdio、C++里通过iostream、Java里通过java.io完成的输入和输出等。严格地说，这些都不是语言的组成部分，但它们又是和语言本身一起定义，并被期望作为支持有关语言的环境的一个组成部分。在另一方面，由于库通常都覆盖了范围相当广泛的一组操作，常要处理与操作系统有关的问题，因此也就很容易成为不可移植问题的避风港。

使用标准库。在这里，应该提出与核心语言同样的建议：盯紧标准，特别是其中比较成熟的、构造良好的成分。C语言定义了标准库，其中包括许多函数，它们处理输入输出、字符串操作、字符类检测、存储分配以及另外的许多工作。如果你把与操作系统的交互限制在这些函数的范围内，那么如果要从一个系统搬到另一个，你的代码很有希望还能具有同样的行为方式，执行得很好。不过你也要当心，因为存在许多标准库的实现，其中有些包含了标准里未定义的行为。

China-pub.com
下载

ANSI C没有定义串复制函数strdup，然而许多系统里都提供了它，甚至在那些声明自己完全符合标准的系统里。一个有经验的程序员可能会根据习惯去使用strdup，完全没意识到它并不在标准之中。而后，当这个系统被移植到某个未提供这个功能的系统上时，程序在编译时就会出问题。这种问题是库引起的移植麻烦中最主要的一类。要解决这类问题，只能靠严格按照标准行事，并要在多种不同环境里测试你的程序。

在头文件或者包定义里声明了标准函数的界面。与头文件有关的一个问题是它们往往非常杂乱，因为它们必须设法在一个文件里同时服侍多种不同的语言。例如，我们常常看到，像stdio.h这样的头文件需要同时为老的C语言、ANSI C和C++ 编译程序服务。在这种情况下，文件里到处都散布着#if和#ifdef一类的条件编译指示符号。由于语言预处理程序并不很灵活，这些文件常常都非常复杂，有时可能还包含着错误。

这里是从我们的某系统里摘录的一段，它比其他许多类似的东西还好一些，因为它具有很好的格式：

```
?   #ifdef _OLD_C
?      extern int fread();
?      extern int fwrite();
?   #else
?   #  if defined(__STDC__) || defined(__cplusplus)
?        extern size_t fread(void*, size_t, size_t, FILE*);
?        extern size_t fwrite(const void*, size_t, size_t, FILE*);
?   #  else /* not __STDC__ || __cplusplus */
?        extern size_t fread();
?        extern size_t fwrite();
?   #  endif /* else not __STDC__ || __cplusplus */
?   #endif
```

虽然这个例子相对而言是清晰的，它也确实印证了我们前面的说法，像这样的头文件（和程序）的结构过于复杂，很难进行维护。针对每个编译系统或者环境建立一个独立的头文件，事情可能更容易些。这样就要求维护一组文件，但其中的每一个都是自足的，适应一个特定系统。这样做也减少了像在严格的ANSI C环境里包括strdup这一类的错误。

头文件还可能“污染”名字空间，因为它里面的某个函数可能正好与程序里的函数同名。例如，我们的weprintf原来被称为wprintf，但是后来发现，在一些环境里根据新的C标准在stdio.h里定义了一个函数，用的也是这个名字。我们只好修改自己函数的名字，以便能在这种系统里完成编译，同时也是为未来做点准备。如果遇到的问题源于一个错误的实现，而不是规范的合法变化，我们就会想办法绕过它，可以采用的方法是在引入头文件时对有关名字重新做定义：

```
?   /* some versions of stdio use wprintf so define it away: */
?   #define wprintf stdio_wprintf
?   #include <stdio.h>
?   #undef wprintf
?   /* code using our wprintf() follows... */
```

这样做的效果，是把头文件里所有的wprintf都映射到stdio_wprintf，使它们不会再与我们的函数发生冲突。在此之后，我们就可以用原来的wprintf，不必再改名了。这种写法有些臃肿，而且还付出了额外代价，与程序连接的库将会调用我们的wprintf函数，而不是调用原来的那个。对于一个函数而言，这可能不必特别担心。但是，确实有些系统给出的环境是非常混乱的，我们必须尽可能地保持代码的清晰性。应该用注释说明这个结构到底

China-pub.com 下载

做了些什么，绝不能再用条件编译把它弄得更糟糕了。如果发现在有的环境里定义了wprintf，那么就应该假定所有的环境里都有这种定义。这样，这个修改就是永久性的，你完全不需要再去维护那些#ifdef语句。当然，更简单的方式是绕道而行，而不是去做斗争，这样做也更安全些。这也就是我们做的事，把函数名字改成weprintf。

即使你总能严格地按规矩办事，环境本身也非常干净，仍然很容易走出限定的范围，例如无意识地假定某些自己喜欢的性质在所有地方都对等等。这方面的例子如，ANSI C定义了6个信号，函数signal能够捕捉到它们。而在POSIX里定义了19个，大部分Unix系统支持32个或者更多的信号。如果你想要用一个非ANSI信号，这很明显就牵涉到功能和可移植性之间的权衡问题，你必须决定哪方面对自己更重要。

目前还存在许多其他标准，它们又不是程序语言定义的组成部分。这方面的例子包括操作系统和网络界面、图形界面，以及许多其他类似的东西。这其中有些东西试图跨越多个系统，例如POSIX；另一些则是为某个特定系统度身打造的，例如各种不同的Microsoft Windows API。与上面类似的建议也适用于这些方面。如果你选择广泛适用的具有良好构造的标准，如果你能盯住最核心的使用最广泛的特性，你的程序就能更具可移植性。

8.3 程序组织

达到可移植性的方式，最重要的有两种，我们将把它们称为联合的方式和取交集的方式。联合方式使用各个特殊途径的最佳特征，采用条件式的编译和安装，根据各个具体环境的特殊情况分别进行处理。这样，结果代码是所有方案的一种联合，它可以利用各系统在能力方面的优点。这种方式的缺点包括：安装过程的规模和复杂性，由代码中大量费解的编译条件造成的复杂性等等。

只使用到处都可用的特征。我们建议采用取交集的方式，即：只使用那些在所有目标系统里都存在的特性，绝不使用那些并不是到处都能用的特征。强求使用普遍可用特性也有危险性，这可能限制了目标系统的范围，或者限制了程序的功能。此外，也可能在某些系统里导致性能方面的损失。

为了比较这两种不同方式，我们来看一些使用联合方式的例子，以及采用交集方式对它们重新进行整理的情况。正如你将要看到的，联合方式的代码从设计上看根本就是不可移植的，虽然它们声称可移植性是自己的目标；而交集代码不仅是可移植的，通常也更加简单。

下面是个小例子，这里试图处理环境中因为某些原因而没有标准头文件stdlib.h的情况：

```
?    #if defined (STDC_HEADERS) || defined (_LIBC)
?    #include <stdlib.h>
?    #else
?    extern void *malloc(unsigned int);
?    extern void *realloc(void *, unsigned int);
?    #endif
```

如果偶然用用的话，这种防御式测试还是可以接受的，但频繁地这样做就很不好了。这里也提出了另一个问题：到底有多少stdlib函数最后出现在这种形式的或者其他类似形式的条件代码里。如果在程序里用到了malloc或者realloc，那么肯定也需要用其他的函数，例如free。如果unsigned int的大小与size_t(这是malloc和realloc参数的正确类型)不一样，那么又会出什么问题？进一步说，我们怎么知道STDC_HEADERS或_LIBC确实已经定

China-pub.com
下载

义了，而且定义正确？怎么保证绝不会有其他名字能在某种环境里启动这里的代换？任何像这样的条件代码都是不完全的、不可移植的，因为总会遇到某个系统不能与这里的条件协调，这时我们就必须重新编辑这些 #ifdef。如果能不通过这类条件编译解决问题，我们就能够根除这些在程序维护中最令人头疼的事情。

然而，这个例子力图解决的问题确实是存在的，那么，怎样做才能一劳永逸地解决它呢？我们认为，宁可事先假设标准头文件是存在的。如果确实没有的话，那就是其他人的问题了。而如果实际情况就是没有，那么更简单的办法是与本软件一起发送一个头文件，在其中定义函数malloc、realloc和free，与ANSI C定义它们的形式完全相同。在程序里总包含这个文件，而不是在代码中到处打上面这样的绷带。这样，我们就能知道必要的界面总是可用的。

避免条件编译。使用#ifdef和其他类似预处理指示写的条件编译是很难管理的，因为在这种情况下有关信息趋向于散布在整个源文件里。

```
?    #ifdef NATIVE
?        char *astring = "convert ASCII to native character set";
?    #else
?    #ifdef MAC
?        char *astring = "convert to Mac textfile format";
?    #else
?    #ifdef DOS
?        char *astring = "convert to DOS textfile format";
?    #else
?        char *astring = "convert to Unix textfile format";
?    #endif /* ?DOS */
?    #endif /* ?MAC */
?    #endif /* ?NATIVE */
```

在这个摘录中，最好是在各定义之后用 #endif，而不是在最后堆积一批 #endif。但是，实际问题是，无论写程序时的动机如何，这段代码都是高度不可移植的，因为它对每个系统的行为不同，对每个新环境必须再写一个新的 #ifdef。用一个串，其中使用一个一般性的词可能更方便，完全是可移植的，而且也提供了同样的信息：

```
char *astring = "convert to local text format";
```

这就不需要任何条件代码，因为它对所有系统都完全一样。

将编译时的控制流(由#ifdef语句确定的)和运行时的控制流混在一起，会使情况变得更坏，因为这种东西极其难读。

```
?    #ifndef DISKSYS
?        for (i = 1; i <= msg->dbgmsg.msg_total; i++)
?    #endif
?    #ifdef DISKSYS
?        i = dbgmsgno;
?        if (i <= msg->dbgmsg.msg_total)
?    #endif
?        {
?            ...
?            if (msg->dbgmsg.msg_total == i)
?    #ifndef DISKSYS
?                break; /* no more messages to wait for */
?            about 30 more lines, with further conditional compilation
?    #endif
?        }
```

China-pub.com 下载

对于那些明显无害的应用，条件编译常常可以用更清晰的形式取代。例如 #ifdef常被用来控制排错代码的执行：

```
?    #ifdef DEBUG
?        printf(...);
?    #endif
```

用一个带常量条件的正规的条件语句也能把事情做得同样好：

```
enum { DEBUG = 0 };
...
if (DEBUG) {
    printf(...);
}
```

如果DEBUG的值是0，大部分编译系统对这段程序不会产生任何目标代码，不过它们会检查被排除代码的语法。与此相反，#ifdef里完全可能隐藏着语法错误，而以后一旦把 #ifdef打开，有关代码就会阻碍编译的执行。

有时人们用条件编译排除掉一大段代码：

```
#ifdef notdef   /* undefined symbol */
    ...
#endif
```

或

```
#if 0
    ...
#endif
```

在编译时采用有条件地替换文件的方式，可以完全避免这种条件代码。下一节里我们还要回到这个问题。

如果需要修改一个程序去适应某个新环境，你不应该以该程序的一个新副本作为出发点。相反，你应该设法调整现存的代码。你可能需要对代码的主体做一些修改。如果采用编辑程序副本的方式，慢慢地你就会做出许多发散的版本。对于一个程序，只要可能，应该只存在惟一的一套源文件。如果你发现某些东西需要改变，以便把程序移植到某个特定的环境去，那么请设法找到一种办法，使改造后的东西在所有地方都能用。如果认为需要，也可以修改内部的界面，但应该保持代码里不出现 #ifdef。这种做法每次都将使你的代码变得更具可移植性，而不是变得更特殊。应该缩小交集，而不是放宽联合。

我们已经说了许多反对条件编译的话，也展示了由它引起的一些问题。但我们还没有提到这其中最恶劣的问题：这种代码几乎是无法测试的。一个 #ifdef实际上把单个的程序变成了两个分别编译的程序，我们很难弄清程序的每个变形是否都已经编译过、测试过。如果在一个#ifdef块里做了修改，那么在其他地方也可能需要做这种修改。要想验证有关的修改，就要求环境条件能够打开对应的 #ifdef。虽然在某些其他配置方式下也需要类似修改，这些情况也不可能检测到。此外，如果程序里需要增加一个新 #ifdef块，我们很难把这种修改孤立出来，很难确定需要满足哪些额外条件才能到达这个地方，很难确定为解决这个问题还要修改哪些地方。最后，如果代码里确有某些东西，按照条件它们将被忽略，那么编译就根本看不到它。这里完全可能是些乱七八糟的东西，而我们却根本就不知道，直到某个不幸的用户试图在某个环境里编译程序，恰巧触发了有关条件。下面的程序当 _MAC有定义时是能够编译的，如果不是这样就会出毛病：

China-pub.com
下载


```
#ifdef _MAC
    printf("This is Macintosh\r");
#else
    This will give a syntax error on other systems
#endif
```

基于上述理由，我们更喜欢只使用那些对所有目标环境都是共同的特性。这样我们就能编译和测试所有代码。如果某些东西产生可移植性问题，我们不是增加条件性代码，而是设法重写代码，设法避免这些问题。沿着这条路走下去，程序的可移植性将逐步增强，其本身也会不断得到改进，而不是变得越来越复杂。

有些大系统在发布时带有一个配置脚本，以便能根据局部环境的情况对代码做一些剪裁。在编译的时候，这个脚本将检测局部环境的各方面特性——头文件和库的位置，字的字节顺序，各种类型的大小，实现者已知可能崩溃的情况(这虽然出人意料，但却是很常见的)，如此等等，由此生成一套配置参数或者make文件，以便对有关情况做出正确配置和设置。这些脚本可能很大、很复杂，是软件发布的重要组成部分，需要不断进行维护，以保证它们能完成任务。有的时候这种技术确实是必须的。但是从另一个角度说，代码的可移植性越强，#ifdef越少，它的配置和安装也就会越简单、越可靠。

练习8-1 研究你的编译系统如何处理括起来放在条件块里面的代码，例如：

```
const int DEBUG = 0;
/* or enum { DEBUG = 0 }; */
/* or final boolean DEBUG = false; */

if (DEBUG) {
    ...
}
```

在什么情况下它确实做了语法检查？什么时候它产生实际的代码？如果你能用的编译系统不止一个，它们互相比较的情况又如何？

8.4 隔离

我们宁愿能有这样一个源程序，它能在所有系统上编译，不需要做任何修改。不过这常常是不现实的。虽然如此，任由不可移植代码散布在程序的各处仍然是不对的，而这也正是条件编译造成的问题之一。

把系统依赖性局限在独立文件里。如果不同系统需要不同的代码，应该使这种差异局限在独立的文件里，一个文件对应一个系统。例如，文本编辑器Sam能在Unix、Windows和许多其他系统上运行。这些环境的系统界面差别极大，但是Sam的绝大部分代码在各处都是一样的。这里对每个特定环境提供一个独立文件，覆盖系统的变化情况。unix.c提供到Unix系统的界面代码，而windows.c用于Windows环境。这些文件实现了一个到操作系统的可移植界面，掩盖掉它们之间的差别。Sam实际上是针对它自己的虚拟操作系统写的，这个虚拟操作系统可以移植到各种实际系统上，方法就是写出几百行C代码，利用可用的系统调用实现十来个小的无法移植的操作。

各种操作系统的图形环境几乎是互不相干的，Sam处理这个问题的办法就是为自己的图形提供一个可移植库。与直接为某个给定系统修改代码相比，建立这种库要做更多的工作(例如，关联X Window系统的界面代码大约有Sam所有其他部分的一半那么大)，但是从长远看，累计

China-pub.com 下载

起来的工作量则要小得多。作为这个工作的副产品，该图形库本身也很有价值，可以单独使用，并已经把几个其他程序弄得可移植了。

Sam是个很老的程序，今天，各种可移植的图形环境，例如 OpenGL、Tcl/Tk和Java等已经在许多不同平台上可以使用了，利用这些东西写你的代码，而不是用专有图形库，这将使你的程序具有更强的可用性。

把系统依赖性隐藏在界面后面。抽象是一种强有力的技术，应该通过它划清程序的可移植部分与不可移植部分之间的界限。大部分程序设计语言所附带的 I/O库就是一个很好的例子，它们使用可供打开/关闭、读和写的文件概念，从不提及任何物理位置或结构，为二级存储器提供了一种抽象。使用这些界面的程序将能在任何实现了它们的系统上运行。

Sam程序的实现是抽象的另一个例子。这里定义了与文件系统和图形操作相关的界面，程序里只使用界面所提供的特征，而界面本身则可以使用下层系统提供的任何功能。对不同系统而言，界面的实现可能差别很大。但是，只使用界面的程序则与这些情况完全无关，当它需要搬到另一处时根本不需要做任何修改。

Java的可移植途径也是个很好的例子，它也说明了沿着这条路上有可能走多么远。一个Java程序被翻译为一种“虚拟机器”上的一系列操作。所谓虚拟机就是一个模拟的计算机，它可以在任何现实的计算机上实现。 Java的库提供了一套一致的对下层系统特性进行访问的功能，包括图形、用户界面、网络以及其他类似的东西。库将被映射到局部系统提供的功能上。从理论上说，完全可能在任何地方，无须任何改变地运行同一个 Java程序(甚至是翻译之后的程序)。

8.5 数据交换

正文数据很容易从一个系统搬到另一个系统去，这是在不同系统间交换任意信息的最简单的方式。

用正文做数据交换。正文容易用各种工具操作，以原来未曾预计到的方式去处理。例如，如果一个程序的输出并不正好适合做另一个程序的输入，我们可以用一个 Awk或Perl脚本去矫正它；可以用grep选择或者删除其中的一些行；可以用你最喜欢的编辑器对它做各种更复杂的修改。正文文件很容易做文档，甚至可能再不需要做文档，因为人完全可以直接阅读它们。正文文件里可以写注释，指明处理这些数据需要什么版本的软件。例如在 PostScript文件里的第一行就说明了它的编码方式：

```
%!PS-Adobe-2.0
```

与此相反，处理二进制文件就需要有专门的工具，甚至在同一台机器上，这些工具常常也不能一起使用。存在许多使用广泛的工具，其作用就是把任意的二进制数据转换成正文，以便能以最不容易出毛病的形式发送出去。这其中包括 Macintosh系统里的binhex，Unix系统的uuencode和uudecode，以及各种各样的为在电子邮件里传递二进制数据而使用的MIME编码工具。在第9章，我们还要给出一组包装和解包例程，它们能用于对二进制数据进行编码，使其能够可移植地进行传递。存在这么多工具，这个情况就足以说明二进制格式存在某些问题。

正文数据交换中也存在一个不和谐音： PC系统使用一个回车符‘\r’和一个换行符‘\n’来结束一个行，而在 Unix系统里只用一个换行符。回车是一种称为电传打字机的古老设备的产

China-pub.com
下载

物，这种设备需要用一个回车(CR)操作使打字部件回到行的开始，再用另一个独立的换行操作(LF)将它推进到一个新行。

虽然今天的计算机已经根本没有什么车可以回了，大部分PC软件仍然期望在每行的最后有这种组合(习惯上称为CRLF，读为“curliff”)。如果没有回车符，一个文件就可能被当作一个极长的行，行和字符计数都可能出错，或发生不能预期的更改。有些软件能很得体地适应这些情况，但大部分软件都不行。PC并不是仅有的罪犯，由于一系列兼容性方面的考虑，某些现代的网络标准，例如HTTP，也用CRLF作为行限界符号。

我们建议只使用标准化的界面，这将在任何给定系统上一致地建立CRLF，无论是(在PC上)输入时去掉\r，到输出时再把它加回去；还是(在Unix上)什么也不做。对那些必须从这边搬到那边的文件，就需要使用能在两种格式间完成转换的程序。

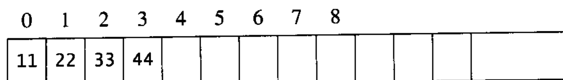
练习8-2 写一个程序从文件中去掉荒谬的回车。写第二个程序把它们加进去，方法是将每个换行符替换为一个回车和一个换行。你将如何测试这些程序？

8.6 字节序

虽然存在着上面讨论里提出的各种缺点，二进制数据有时仍然是必需的，因为它更紧凑，解码也更迅速。在计算机网络领域，二进制形式是许多工作的基础，紧凑和速度都是最根本的原因。但是，二进制数据确实存在许多移植性问题。

至少有一个情况可以确定：所有现代机器都使用8位字节。但是，不同机器在如何表示大于一个字节的对象时就有许多不同方式，特别依赖某种特定方式就是一个错误。一个短整数(通常是16位，含两个字节)的低位字节可能存储在比其高位字节低的存储地址(低尾端方式)，或者存在较高的存储地址(高尾端方式)。这方面的决定确实带有随意性，有的机器甚至同时支持两种方式。

这样一来，虽然对采用高尾端或低尾端方式的机器而言，存储器都被看成是某种顺序下的字序列，它们对一个字里各字节的解释却采用了相反的顺序。在下面的图中，从地址0开始的4个字节表示某个十六进制整数。对高尾端机器而言，它是0x11223344，而对低尾端机器就是0x44332211。



要想知道实际机器中的字节顺序问题，可以看下面的程序：

```
/* byteorder: display bytes of a long */
int main(void)
{
    unsigned long x;
    unsigned char *p;
    int i;

    /* 11 22 33 44 => big-endian */
    /* 44 33 22 11 => little-endian */
    /* x = 0x1122334455667788UL; for 64-bit long */

    x = 0x11223344UL;
    p = (unsigned char *) &x;
    for (i = 0; i < sizeof(long); i++)
        printf("%x ", *p++);
```

China-pub.com 下载

```
        printf("\n");
        return 0;
    }
```

在32位的高尾端机器上，其输出是：

```
    11 22 33 44
```

在低尾端机器上，输出就会是：

```
    44 33 22 11
```

但是，在PDP-11机器(这是某个时期的一种卓越的机器，现在还可以在许多嵌入式系统中看到)上是：

```
    22 11 44 33
```

在具有64位long类型的机器上，只要改变常数的大小，就可以看到类似现象。

这看起来像是一个无聊的问题，但是，如果我们需要把整数通过具有一个字节宽度的界面(例如通过网络连接)传输时，就必须选定首先传输哪个字节，而这个选择必然要牵涉到低尾端/高尾端的问题。换句话说，该程序应该以明显方式做某些事，而这些事在：

```
    fwrite(&x, sizeof(x), 1, stdout);
```

中是以隐含方式处理的。如果在一台计算机里写出一个 int(或者short、long)，而要另一台计算机读它，这将是件很不安全的事情。

例如，假设源计算机用下面方式写

```
    unsigned short x;
    fwrite(&x, sizeof(x), 1, stdout);
```

而接收的计算机用下面方式读

```
    unsigned short x;
    fread(&x, sizeof(x), 1, stdin);
```

如果两台机器具有不同的字节顺序， x的值将无法保持原样。如果 x在开始时是0x1000，它在到达时可能就是0x0010。

人们通常用条件编译和字节交换来解决这种问题，写的东西大概是：

```
?     short x;
?     fread(&x, sizeof(x), 1, stdin);
?     #ifdef BIG_ENDIAN
?     /* swap bytes */
?     x = ((x&0xFF) << 8) | ((x>>8) & 0xFF);
?     #endif
```

当存在许多二字节和四字节整数需要交换时，这种方式就显得非常笨拙。在实践中，在把这些字节从一个地方传到另一个地方的过程中，可能需要多次完成这类交换。

对short而言，事情已经很不妙了，对于更长的数据类型，情况当然会更糟糕，因为这时存在着更多的排列其中字节的方式。如果再加上结构成员之间的大小可能不同的填充、对齐方面的限制，以及各种老式机器千奇百怪的字节顺序，问题看起来很难对付。

数据交换时用固定的字节序。这里提出一种解决办法：写可移植的代码，总按照正规的顺序写出各个字节：

```
    unsigned short x;
    putchar(x >> 8);        /* write high-order byte */
    putchar(x & 0xFF);      /* write low-order byte */
```

China-pub.com
下载

一次一个字节地读回，把数据重装起来：

```
unsigned short x;
x = getchar() << 8;     /* read high-order byte */
x |= getchar() & 0xFF;  /* read low-order byte */
```

这个方法也可以推广到结构，你只要把结构成员按照一种规定顺序写出去，一次一个字节，丢掉填充。无论你采用什么字节顺序，所有事情都将能够一致地完成。这里惟一的要求是：对于传送中所采用的字节顺序和各种对象的大小，发送方和接收方都应该有同样的认识。在下一章里，我们要给出一对例行程序，它们的功能就是对一般性的数据进行打包和解包。

一次一个字节的处理方式看起来代价很高，但相对于需要打包和解包的 I/O而言，这个代价是微不足道的。考虑X Window系统，在其中客户总按照它自己的字节顺序去写数据，而服务器就必须设法解开客户送来的任何东西。从客户端的角度看，这样做确实能节约几条指令；而服务器就被弄得更大更复杂了，因为它必须同时处理各种可能的字节顺序——它可能要同时应付低尾端和高尾端的客户——在复杂性和代码方面付出的代价是很明显的。另一方面，这又是一个图形环境，在这里包装字节的代价会完全被浸没在它所编码的图形操作的执行之中。

X Window系统在客户方面完全忽略了字节顺序问题，而要求服务器方面能同时处理两种情况。与此相反，Plan 9操作系统则为送到文件服务器(或图形服务器)的信息定义了一种字节顺序，数据采用上面说的那种可移植代码做打包和解包。在实践中，这种操作对运行的影响根本无法察觉，与I/O操作相比，包装数据的代价完全可以忽略不计。

Java是一种比C或C++ 更高级的语言，它完全隐蔽了字节顺序的问题。Java程序库提供了一个`Serializable`界面，它定义了在交换时数据项的包装方式。

如果你在C或C++ 里工作，那么你就得自己做这件事。采用一次处理一个字节的方式，最关键的收获就是既不需要使用 `#ifdef`，又能对所有8位字节的机器解决问题。我们将在下一章里继续这方面的讨论。

当然，解决问题的最好办法常常还是把信息转换到正文形式，这种形式 (除了CRLF问题外)完全是可移植的，这里根本不存在表达的歧义性问题。当然，这并不一定就是正确的解，因为时间和空间都是极其重要的。有些数据，特别是浮点数据，在传过 `printf`和`scanf`时有可能丢失精度，主要是由于截断问题[㊀]。如果你必须完全精确地交换浮点数据，那么就必须弄清你确有很好的格式化 I/O库。这种库是存在的，不过它可能不是你工作环境的一部分。想要以可移植的方式用二进制形式表示浮点数据非常困难，另一方面，如果你足够细心，通过正文可以做好这件事。

在使用标准库函数处理二进制文件时，还有一个很微妙的可移植问题——必须以二进制方式打开文件：

```
FILE *fin;

fin = fopen(binary_file, "rb");
c = getc(fin);
```

如果忽略了这里的‘b’，在所有Unix系统上都不会出任何问题。但是在 Windows系统里，程

㊀ 实际上，更根本的问题是不同数制之间的转换，例如十进制的小数在转换到二进制表示时常常变成无限循环小数，不一定能有精确的对应物。——译者

China-pub.com 下载

序输入中遇到的第一个control-Z字节(八进制的032，十六进制的1A)将结束有关的读入动作(我们在第5章的strings程序里已经看到这种情况的发生)。在另一方面，如果按二进制模式读入正文文件，会导致\r字节被留在了输入里面，而在输出时又不会自动地产生它。

8.7 可移植性和升级

可移植性问题中还有一个最让人头疼的因素，那就是，系统软件总是在随着时间推移而不断变化，这种变化有可能发生在系统的任何层面上，导致程序的现存版本之间无缘无故地发生互不相容的情况。

如果改变规范就应改变名字。 我们最喜欢(如果可以用这个词的话)用的例子是Unix系统中echo命令的性质变化。在开始时，它的设计仅仅是想做对参数的回应：

```
% echo hello, world
hello, world
%
```

但是，到了后来，echo变成了许多shell脚本里的关键部分，希望产生格式化输出的需求变得越来越强烈，所以echo被改造为解释它的参数，从某种意义上看更像printf了：

```
% echo 'hello\nworld'
hello
world
%
```

这种新特性确实很有用，但它也使许多shell脚本产生了移植性问题，因为它们可能就依赖于echo命令除了回应之外什么也不做。而

```
% echo $PATH
```

的行为方式则依赖于现在所用的echo是什么版本的。如果不巧在变量值里包含了一个反斜线符号(在DOS或Windows系统里很容易遇到这种情况)，echo可能对它做出某种特殊解释。这个问题类似于用printf(str)或printf("%s",str)产生输出时的差别。请看看，如果在str里正好有一个百分号，会发生什么事情。

我们说的还仅仅是echo故事中的一部分，但是它已经阐明了根本的问题：对系统的修改可能产生不同版本的软件，有时我们有意地使它们在行为方面有些变化，但同时也无意地造成了许多移植性问题，而这些问题常常是很难绕过去的。如果给echo的新版本另起一个不同的名字，造成的麻烦可能要少得多。

现在再看一个更具启发性的例子。考虑Unix命令sum，它打印出一个文件的大小和它的检验和。下面一串命令是想验证一次信息传递是否完全成功：

```
% sum file
52313 2 file
%
% copy file to other machine
%
% telnet othermachine
$
$ sum file
52313 2 file
$
```

由于传递之后的检验和相同，我们可以合理地推断，新副本和老文件是一样的。

China-pub.com
下载

而后系统被改进了，版本发生了变化，有人发现检验和的算法并不完美，因此就修改了sum，使用了更好的算法。另外的什么人也做了同样研究，给出 sum的另一个更好的算法。这样发展下来，到了今天，实际中已经存在着许多不同版本的 sum，每个版本给出的回答都不同。我们把一个文件复制到附近另一台机器上，想看看 sum算出的是什么：

```
% sum file
52313 2 file
%
% copy file to machine 2
% copy file to machine 3
% telnet machine2
$
$ sum file
eaa0d468     713     file
$ telnet machine3
>
> sum file
62992    1 file
>
```

是文件破坏了吗？还是仅仅因为我们恰好用到不同版本的 sum？或许两者都是？

这样的sum就形成了一个完美的可移植性灾难：一个程序，其目的是为了帮助人们在从一台机器到另一台复制软件时做检查。但是，由于存在许多互不兼容的版本，从原来的意图看，它已经变成毫无意义的东西了。

对于原本要应付的简单工作而言，最早的 sum是很好的，其低技术的检验和算法也是适宜的。虽然“修改”它有可能造就出一个更好的程序，但从中得到的并不多，至少完全不足以抵消这种不相容的代价。问题不在于性能提升，而在于互不相容的程序又偏偏用了同样名字。这种变化带来的版本问题还会折磨我们许多年。

维护现存程序与数据的相容性。当一个软件(例如一个字处理系统)的新版本发布时，新版本通常都能读入由较早版本产生的文件。我们也可以断定，由于增加了新的原来没有的特征，文件格式也可能有变化。但是，新版本常常没有提供一种方式，使之能写出原来格式的文件。这样，新版本的用户，即使他们没有使用新版本中的任何新特征，也无法与那些使用旧版本的人们共享文件。这将迫使每个人都去升级。无论是作为一个工程观点，还是作为一种市场策略，这种设计都是特别令人遗憾的。

向后兼容是使程序符合其过去规范的一种能力。如果你打算修改一个程序，那就应该保证你没有破坏老的程序和依赖于它的数据。应该正确地修改文档，提供一些办法去恢复原来的行为方式。最重要的是，应该仔细考虑你计划做的改变是不是真正的改进，与你将引进的不可移植性的代价相比，是不是真正值得去做。

8.8 国际化

生活在美国的人很容易忘记英语并不是惟一的语言， ASCII不是惟一的字符集，$也不是仅有的钱币符号，写日期时可以把日子写在前面，时间可以采用 24小时的钟点，如此等等。所以，可移植性的另一方面，更广泛地说，是要处理程序在跨越语言和文化边界时的可移植性问题。这是一个内涵极其丰富的题目，我们只能用很有限的篇幅指出其中的一些基本考虑。

国际化是个术语，意指设法使一个程序并不对其运行的文化环境有任何假定。这方面的

China-pub.com 下载

问题非常多，从所用的字符集，到界面上图标的解释信息等。

不要假定是ASCII。在世界上许多地方，字符集都比ASCII丰富得多。在ctype.h里提供的标准字符检测函数通常能够掩盖这些差异：

```
if (isalpha(c)) ...
```

这样就能独立于字符的特定编码方式。进而，在那些字母比从a到z更多几个或者少几个的地方，如果程序在那里编译，它也应该能正常工作。当然，甚至isalpha这个名字也表达了它自己的来历，因为在有些语言里根本没有字母表。

ASCII编码只定义了直到0x7F的值(7位)，大多数欧洲国家都对此做了扩充，增加了另外一些字符，以便表示他们语言中的字母。Latin-1编码集在西欧使用很广泛，它就是ASCII的一个超集，其中把80到FF的字节值定义为一些符号和带调字母。例如编码E7表示ç。英语单词boy在ASCII(和Latin-1)里用三个字节表示，按十六进制写是62 6F 79，而法文词garçon在Latin-1里用字节67 61 72 E7 6F 6E表示。其他语言还定义了另外的符号。但是这些东西不可能全都放进ASCII没用的那128个值里。因此，现实中存在许多互相冲突的标准，它们给80到FF的字节指定了不同字符。

有些语言根本就无法放进8位的字节里。各种主要的亚洲语言都有成千上万的字符，中国、日本和韩国用的编码都采用每个字符16位的方式。由此产生的结果是，要在一台某种语言的计算机系统上阅读以另一种语言书写的文档，就会出现一个大的移植性问题。假定到来的字符本身并未受到任何损害，在一台美国计算机上读一个中文文档，至少也要涉及到特殊的软件和字型。如果我们希望能同时使用中文、英文和俄文，面临的障碍将是十分明显的。

Unicode字符集是人们希望改善这种状况的一个尝试，它为全世界所有的语言提供了一套统一编码。Unicode与ISO 10646标准的16位子集相兼容，这里对每个字符用16位做编码，其中的00FF及比它小的值对应于Latin-1。这样，法文单词garçon用16位值0067 0061 0072 00E7 006F 006E表示。西里尔字符集的编码占的位置是0401到04FF，所有表意符号语言的字符占据从3000开始的大片位置。在Unicode里包含了所有重要的语言，还有许多不那么重要的语言。因此，如果要在不同国家之间传递文件，或者需要存储多语言的文本，Unicode是一种很合适的编码选择。Unicode已经在Internet上逐渐流行起来，有些系统甚至把它作为标准加以支持，例如Java语言就用Unicode作为它字符串的基本字符集。Plan 9和Inferno操作系统都完全使用了Unicode，甚至对文件名和用户名也是这样。Microsoft Windows支持Unicode字符集，但却不是强制性的要求，大部分Windows应用仍然是在ASCII里工作得最好，不过实际情况正在迅速向Unicode方面发展。

Unicode也带来了一个问题：字符不再能放进字节里。因此，Unicode文本也会遇到字节顺序问题的滋扰。为了避免这种状况，Unicode文档在程序间或者通过网络进行传递之前，通常都先行转换为一种称为UTF-8的字节流编码形式。每个16位字符被编码为1个、2个或3个字节的一个序列，专门用于传输。ASCII字符集仍然用从00到7F的值表示，在使用UTF-8时，所有这些字符都被放在一个字节里，所以UTF-8对ASCII具有向后兼容性。位于80和7FF之间的值用两个字节表示，800以及比它更大的值用三个字节表示。单词garçon在UTF-8里表示为字节67 61 72 C3 A7 6F 6E，其中的Unicode值E7，即字符ç，在UTF-8里用两个字节C3 A7表示。

UTF-8对ASCII的向后兼容性是极其重要的。因为这就能保证，把正文当作不加解释的字节流的程序能在任何语言里对Unicode工作。我们在UTF-8编码的多种语言正文上试验了第3

China-pub.com
下载

章的Markov程序，包括俄文、希腊文、日文和中文。程序运行都没有问题。对于欧洲语言，它们的词都是由ASCII的空格、制表符或换行符分隔，程序输出的是合理的无意义句子。对于其他语言，我们就必须对词的分割规则做必要的修改，以设法得到在精神上接近于程序意图的输出。

C和C++语言支持“宽字符”，这是16位或者更大的整数。另外有几个伴随而来的函数，可用于处理Unicode或者其他大字符集的字符。宽字符串文字应写成 `L"..."`，它们也带来了进一步的移植性问题：一个用宽字符串的程序只能在那些能够使用该字符集的显示器上使用，只有这样人们才能够理解。为了在机器间可移植地进行传输，这种字符必须首先转换到字节流，例如UTF-8之类的东西，C提供了把宽字符转换到字节和反向转换的函数。但是我们应该用什么转换呢？对字符集的解释、对字节流编码的定义都隐蔽在程序库里，很难弄清楚。这种情况至少不是很令人满意的。或许在美好未来的某个时候，人们在使用的字符集上取得了一致，但是，随后出现的场面很可能又会使人回想起今天仍在折磨着我们的字节序问题。

不要假定是英语。界面的构建者必须记住，不同的语言在谈论同一事情时所使用的字符数目经常有很大差异。所以，在屏幕上和数组里都必须留有足够的空间。

错误信息该怎么办？至少说，它们必须不包含任何仅仅在某个特殊人群中才能理解的暗语和行话，用简明的语言写出它们不过是个开始。人们广泛采用的一种技术是把所有信息正文汇集到同一个地方，这样，在需要翻译到其他语言时就能方便地替换它们。

现实中存在着许多与文化密切相关的东西，例如， mm/dd/yy的日期格式表示只在北美使用。如果一个软件有一点用到其他国家去的可能性，就应该设法避免这类文化依赖性，或应设法将其减到最少。图形界面上的图标常常也带有文化依赖性，许多图标把预期使用环境中的本地人都弄得莫名其妙，更不用说具有其他文化背景的人们了。

8.9 小结

可移植代码是一个非常值得去追求的理想，因为有如此多的时间被浪费在修改程序方面，无论是把程序从一个系统移到另一个系统，还是为了它本身演化的需要，或是因为它运行的系统发生了变化，在这些情况下需要设法维持程序的继续运行。当然，可移植性不是随便就能得到的，它要求实现中的特别注意，也需要开发者具有对所有的潜在目标系统在可移植问题方面的知识。

我们已经指出了追求可移植性的两种途径，即联合和交集。联合途径相当于为在每个目标系统上工作而写一个版本，利用条件编译一类的机制，把这些代码尽可能地汇集在一起。这种途径的缺点很多，它造成过多的代码，而且常常是很多非常复杂的代码。它很难更新，也很难测试。

交集途径是设法以一种形式写出尽量多的代码，使它能在每种系统上运行而不需要做任何修改。把无法逃避的系统依赖性封装在独立的源文件里，其作用就像是程序与基础系统之间的界面。交集方法也有缺点，包括可能存在性能方面，甚至特征方面的损失。但是从长远的观点看，这种途径的利大于弊。

补充阅读

对于程序设计语言有许多描述，但是，在这之中只有很少是足够精确的，能够作为定义

China-pub.com 下载

性的参考手册。作为作者个人的偏爱，倾向于提出由 Brian Kernighan和Dennis Ritchie写的《C程序设计语言》(The C Programming Language，Prentice Hall，1988)。但它并不是语言标准的替代品。由 Sam Harbison和Guy Steele写的《C：参考手册》(C: A Reference Manual，Prentice Hall，1994)现在已经是第4版，在其中包括了许多关于可移植性的很好建议。官方的C和C++ 标准可以由ISO(国际标准化组织)得到。与Java的官方标准最接近的东西是 James Gosling、Bill Joy和Guy Steele的《Java语言规范》(Java Language Specification，Addison Wesley，1996)。

Rich Stevens的《Unix环境高级编程》[⊖](Advanced Programming in the Unix Evironment，Addison Wesley，1992)对于Unix程序员而言是绝好的资源，它提供了有关在不同 Unix变形间的可移植性问题的丰富材料。

POSIX(可移植操作系统界面，Portable Operating System Interface)是一个基于Unix的命令和程序库的国际标准定义。它提供了标准化的环境、有关应用的源代码可移植性以及一个到I/O、文件系统和进程的统一界面。IEEE出版了描述它的丛书。

术语“高尾端”可以追溯到1726年的Jonathan Swift。Danny Cohen的文章《论圣战以及对和平的祈祷》(On holy wars and a plea for peace，IEEE Computer，1981年10月)是一篇关于字节顺序的美妙寓言，它把“尾端”这个术语引进计算机界。

贝尔实验室开发的 Plan 9系统把可移植性作为其核心议题。这个系统可以从完全没有`#ifdef`的源程序编译到多种处理器，它从根本上使用了 Unicode字符集。Sam的第一个描述出现在“文本编辑器sam”(The Text Editor sam)，《软件：实践和经验》(Software：Practice and Experience)卷17，第11期，pp813~845页，其最新版本使用了Unicode，而且可以运行在许多系统上。在Rob Pike和Ken Thompson的文章“Hello World or K μ´ μ orこんにちは世界”(Proceedings of the Winter 1993 USENIX Conference，1993年冬季USENIX会议录，圣迭戈，1993，pp43~50页)里讨论了处理16位字符集，例如Unicode的问题。UTF-8编码也是第一次出现在这篇文章里。这篇文章也可以在贝尔实验室的 Plan 9网络站点找到，Sam的当前版本也保存在那里。

Inferno系统与Java很像，是基于Plan 9的经验，在其中定义了一个虚拟机器，它可以在任何实际机器上实现。它还提供了一种语言 (Limbo)，该语言可以翻译到这种虚拟机上，并使用Unicode作为其基本字符集。在这里还包括一个虚拟操作系统，提供了到一些商品系统的可移植界面。有关工作发表在《Inferno操作系统》(The Inferno Operating System)上，作者是Sean Dorward、Rob Pike、David Leo Presotto、Dennis M. Ritchie、Howard W. Trickey和Philip Winterbottom，刊于《贝尔实验室技术杂志》(Bell Labs Technical Journal)，卷2，第1期，1997年冬。

⊖ 此书已由机械工业出版社出版。——编者

China-pub.com
下载

第9章 记 法

在人的所有造物中，语言或许是最奇妙的东西了。

Giles Lytton Strachey，《词和诗》

采用正确的语言有可能使某个程序的书写变得容易许多。正是由于这种情况，在实际程序员的武器库里都有许多东西，不仅有像C这一类的通用语言，还有可编程的shell、脚本语言以及许多面向特定用途的语言。

好记法的威力不仅表现在传统的程序设计中，也体现在各种特定问题领域。正则表达式使我们能以非常紧凑的形式(或许还有点像密码)定义一个字符串类；HTML使我们能定义交互式文档的编排格式，其中还常常嵌入其他语言，如 JavaScript程序；PostScript能把整个文档——例如这本书——表示为一个格式程序。电子表格和文字处理器也常常包含某种程序语言，例如Visual Basic，用以计算其中的表达式，访问有关信息，或者做格式编排的控制等。

如果你发现自己为某些平庸的事情写了太多代码，或者你需要表述某些过程却遇到了很大麻烦，那么你正在使用的很可能是一种不适当的语言。如果合适的语言不存在，那么这很可能就是个机会，需要你自己来建立一种。发明语言并不意味着是建立某种像 Java那样复杂的东西，许多棘手的问题常常可以通过改变描述方式的办法来解决。请考虑一下 printf一类函数的格式描述串，那就可以看作是一种控制数据打印方式的、紧凑的、描述能力很强的语言。

在本章中我们要讨论怎样通过记法去解决问题。这里还要展示一些技术，这些技术可以用于实现你自己的专用语言。我们还要探索用程序来写其他程序的可能性，这是记法使用的一种极端形式，也是很常见的。实际上，这种技术并不难使用，远不像许多程序员所认为的那样。

9.1 数据格式

在我们最希望对计算机说的东西(“请解决我的问题”)与为了使一个工作能够完成而必须说的东西之间，永远存在着一条鸿沟。能把这条鸿沟填得越窄，当然就越好。好的记法使我们能更容易说出自己想说的东西，又不太容易因为不当心而说出错误的东西。确实也有这样的情况，好的记法能给人提供新的见识，帮助我们解决看起来非常困难的问题，甚至引导我们得出新的发现。

小语言是指那些针对较窄的领域而使用的特定记法，它们不仅提供了某种好界面，还能够帮助人组织实现它们的程序。printf的格式控制序列是一个很好的例子：

```
printf("%d %6.2f %-10.10s\n", i, f, s);
```

格式串里的每个%标记着一个位置，要求在这里插入 printf下一个参数的值。在一些可省缺的标志或域宽说明之后，最后一个字符指明了所要求的参数类型。这种记法非常紧凑，既直观又容易书写，实现起来也很方便。作为其替代物的 C++ 的iostream和Java的

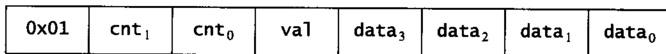
China-pub.com 下载

java.io看起来更笨拙，究其原因，虽然它们扩充到了用户定义类型，提供了类型检查，但却没能提供一种特殊的记法。

也有些非标准的printf实现，它们允许人们在内部功能之外增加自己的方式。如果你使用其他数据类型，经常要做它们的输出转换，有这种功能就非常方便。例如，一个编译程序可能想用%L输出行号和文件名；一个图形系统可能用 %P表示点，而用%R表示矩形。在第4章里我们看到为提取股票价格行情而设的字母数字的神秘序列，采用的也是类似想法，是为编排股票数据的组合而提供一种紧凑记法。

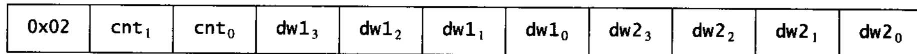
现在我们在C和C++里做些类似的例子。假设我们需要从一个系统向另一个系统传送一种包含各种数据类型的组合数据包。在第 8章我们已经了解到，最清晰的一种解法可能就是把数据包转换为正文表示。当然，标准网络规程所使用的多半是二进制格式，为的是效率或者数据规模。现在的问题是：应该如何去写处理数据包的代码，使它能够可移植、高效率，而且又很容易使用？

为使这个讨论更实在些，设想我们在系统间传递的是包含 8位、16位和32位数据项的包。ANSI C告诉我们，8位数据总可以存储在char里，16位可以放进short，而32位放进long，因此我们就用这些数据类型表示有关的值。实际中可能存在许多不同种类的包。例如，在第一种包里有一个字节的类型描述，2字节的计数值，1字节的值，以及一个4字节的数据项：



0x01	cnt_1	cnt_0	val	$data_3$	$data_2$	$data_1$	$data_0$

第二种包类型包含有一个短的和两个长的数据字：



0x02	cnt_1	cnt_0	$dw1_3$	$dw1_2$	$dw1_1$	$dw1_0$	$dw2_3$	$dw2_2$	$dw2_1$	$dw2_0$

可采用的一种方式就是为每种类型的包写一对打包和开包函数：

```
int pack_type1(unsigned char *buf, unsigned short count,
        unsigned char val, unsigned long data)
{
    unsigned char *bp;

    bp = buf;
    *bp++ = 0x01;
    *bp++ = count >> 8;
    *bp++ = count;
    *bp++ = val;
    *bp++ = data >> 24;
    *bp++ = data >> 16;
    *bp++ = data >> 8;
    *bp++ = data;
    return bp - buf;
}
```

对于实际规程，我们将需要成打的这种例程，它们都是某个东西的一种变形。利用处理基本类型(如short、long等)的宏或者函数可以简化这些例程。但是，即使是按这种方式去做，这么多重复性代码也是很容易写错的。它们既难阅读，也难进行维护。

这些代码的内在重复性实际上提醒了我们，记法可能会有所帮助。我们可以从 printf借用有关的想法，定义一个小语言，使每种包在这里都可以用一个简洁的串描述，用这个串刻画数据包的编排形式。包中连续的各个元素用编码表示，用 c表示8位字符，s表示16位短整

China-pub.com
下载

数，l表示32位的长整数。例如，上面提出的第一个包（包括它的类型描述）就可以用格式串cscl表示。这样，我们只需要写一个打包函数就可以应付所有不同类型的数据包了。要建立上面的包，需要写的不过是：

```
pack(buf, "cscl", 0x01, count, val, data);
```

这里的格式串只包含数据定义，因此不必像printf那样使用%一类的字符。

在实践中，数据包开始处的信息可以告诉接受方怎样对其他部分进行解码。我们将假定数据包的第一个字节能用于确定有关的编排格式。发送方按这种格式对数据编码并发送出来，接受方读这种包，提取其第一个字节，并依据它对包的其余部分解码。

下面是pack的一个实现，它按照格式串确定的方式，将参数的编码表示填入buf中。我们把所有数据都看成无符号的，包括位于包缓冲区里的字节数据，这样可以避免符号扩展问题。在这里还用了一些typedef，以便使声明更简短些：

```
typedef unsigned char  uchar;
typedef unsigned short ushort;
typedef unsigned long  ulong;
```

就像sprintf、strcpy及其他类似函数一样，pack假定它所用的缓冲区对于要存放的结果而言是足够大的，调用它的程序必须保证这个条件。这里也不企图去检查格式与参数表之间不匹配的情况。

```
#include <stdarg.h>

/* pack: pack binary items into buf, return length */
int pack(uchar *buf, char *fmt, ...)
{
    va_list args;
    char *p;
    uchar *bp;
    ushort s;
    ulong l;

    bp = buf;
    va_start(args, fmt);
    for (p = fmt; *p != '\0'; p++) {
        switch (*p) {
        case 'c':   /* char */
            *bp++ = va_arg(args, int);
            break;
        case 's':   /* short */
            s = va_arg(args, int);
            *bp++ = s >> 8;
            *bp++ = s;
            break;
        case 'l':   /* long */
            l = va_arg(args, ulong);
            *bp++ = l >> 24;
            *bp++ = l >> 16;
            *bp++ = l >> 8;
            *bp++ = l;
            break;
        default:    /* illegal type character */
            va_end(args);
            return -1;
        }
    }
```

China-pub.com 下载

```
    va_end(args);
    return bp - buf;
}
```

这个pack函数使用了stdarg.h头文件的功能，比第4章的eprintf用得更多。利用va_arg顺序地提取各个参数：va_arg的第一个参数是va_list类型的变量，必须调用va_start对它预先做设置；va_arg的第二个参数是函数参数的类型(这也是为什么va_arg是宏，而不是函数的根本原因)。在处理完成后，应该调用va_end。虽然‘c’和‘s’对应的参数分别是char和short值，这里必须用int方式提取它们，原因在于char和short是用函数参数表最后的...表示的，在这种情况下，C语言将自动把它们提升到int。

现在每个打包函数都只有一行了，不过是把它的参数排列在一个pack调用里：

```
/* pack_type1: pack format 1 packet */
int pack_type1(uchar *buf, ushort count, uchar val, ulong data)
{
    return pack(buf, "cscl", 0x01, count, val, data);
}
```

开启数据包的工作也可以同样处理：不是为敲开每种数据包格式写一段单独代码，而是写一个带有格式描述的unpack。这种做法把所有数据变换都集中到了一个地方：

```
/* unpack: unpack packed items from buf, return length */
int unpack(uchar *buf, char *fmt, ...)
{
    va_list args;
    char *p;
    uchar *bp, *pc;
    ushort *ps;
    ulong *pl;

    bp = buf;
    va_start(args, fmt);
    for (p = fmt; *p != '\0'; p++) {
        switch (*p) {
        case 'c':   /* char */
            pc = va_arg(args, uchar*);
            *pc = *bp++;
            break;
        case 's':   /* short */
            ps = va_arg(args, ushort*);
            *ps  = *bp++ << 8;
            *ps |= *bp++;
            break;
        case 'l':   /* long */
            pl = va_arg(args, ulong*);
            *pl  = *bp++ << 24;
            *pl |= *bp++ << 16;
            *pl |= *bp++ << 8;
            *pl |= *bp++;
            break;
        default:    /* illegal type character */
            va_end(args);
            return -1;
        }
    }
    va_end(args);
    return bp - buf;
}
```

China-pub.com
下载

像scanf一样，unpack也必须给它的调用者返回多个值，因此它的参数是那些指向准备存储结果的变量的指针。函数返回数据包的字节数，可以用于错误检查。

由于所有值都是无符号的，而且我们坚持不超出 ANSI C对各种数据类型定义的大小，上面这些代码总能以可移植的方式传递数据，甚至在那些具有不同大小的 short和long的机器之间。例如，只要使用 pack的程序不试图把32位无法表示的值当作 long传出去，传递的值就一定能正确接收。如果它这样做了，那么实际传送的将是数据的低 32位。如果真的需要传送更大的值，我们也很容易增加其他格式定义。

通过调用unpack，特定类型开包函数的定义都变得非常简单：

```
/* unpack_type2: unpack and process type 2 packet */
int unpack_type2(int n, uchar *buf)
{
    uchar c;
    ushort count;
    ulong dw1, dw2;

    if (unpack(buf, "csll", &c, &count, &dw1, &dw2) != n)
        return -1;
    assert(c == 0x02);
    return process_type2(count, dw1, dw2);
}
```

要调用unpack_type2，必须先确认遇到的正是一个类型 2的数据包，这意味着在接收程序里应该有一个下面形式的循环：

```
while ((n = readpacket(network, buf, BUFSIZ)) > 0) {
    switch (buf[0]) {
    default:
        eprintf("bad packet type 0x%x", buf[0]);
        break;
    case 1:
        unpack_type1(n, buf);
        break;
    case 2:
        unpack_type2(n, buf);
        break;
    ...
    }
}
```

以这种方式做程序设计也可能拖得很长。实际上有一种紧凑写法，那就是定义一个函数指针的表，其中各个项是所有的开包函数，以数据包的类型编号作为下标：

```
int (*unpackfn[])(int, uchar *) = {
    unpack_type0,
    unpack_type1,
    unpack_type2,
};
```

表中的每个函数处理一种数据包，检查结果，并启动对包的下一步处理过程。有了这个表之后，接收程序的工作就非常简单了：

```
/* receive: read packets from network, process them */
void receive(int network)
{
    uchar type, buf[BUFSIZ];
    int n;
```

China-pub.com 下载

```
    while ((n = readpacket(network, buf, BUFSIZ)) > 0) {
        type = buf[0];
        if (type >= NELEMS(unpackfn))
            eprintf("bad packet type 0x%x", type);
        if ((*unpackfn[type])(n, buf) < 0)
            eprintf("protocol error, type %x length %d",
                type, n);
    }
}
```

这样，每种数据包的处理代码都非常紧凑，而且只写在一个地方，很容易维护。这也使接收程序基本上独立于传输规程，同时又是清晰和快速的。

上面的例子来源于为实现某个产品网络规程而写的实际代码。当作者认识到这种方式能工作之后，数千行重复的、充满错误的代码被缩减成几百行非常容易维护的代码。记法消解混乱的力量实在太大了。

练习9-1 修改pack和unpack，使它能正确传递带符号的值，甚至在具有不同大小的short和long的机器之间。你应该如何修改格式串以描述带符号数据？你将怎样去测试这些代码，例如，怎样确定程序能正确地把-1从一台具有32位long数据的机器传送给另一台具有64位long的机器？

练习9-2 扩展pack和unpack，使它们能处理字符串。一个可能的方式是在格式串里包含有关字符串长度的描述。扩充函数，使它们能利用计数值处理重复的数据项。这与字符串的格式编码方式又会有什么相互影响？

练习9-3 上面C程序里的函数指针表是 C++ 虚函数机制的核心。在 C++ 里重写pack、unpack和receive，利用C++ 在这方面记法上的优点。

练习9-4 写一个printf的命令行版本，它按第一个参数指明的格式，打印其第二个及后面的参数。有些操作系统外壳已经提供了这种内部功能。

练习9-5 写一个程序，它应能实现电子表格程序或者 Java的DecimalFormat格式规范，能按照模式显示数值。模式里指明必须的或可选的数字、小数点和逗号的位置等等。作为说明，下面的格式：

```
##,##0.00
```

描述的是有两个小数位的数，小数点左边至少有一个数字，在千位数字后面有一逗号，这里还要求用空格填充，直到万位数字的位置。这样， 12345.67将被表示为12,345.67， 而 .4将表示为_ _ _ _0.40(这里的下划线表示的是实际空格)。完全的规范请参看DecimalFormat的定义，或者某种电子表格程序。

9.2 正则表达式

pack和unpack的格式描述串是非常简单的记法，它们可用于定义数据包的编排形式。我们的下一论题是一种稍微复杂一点，但是更具表达能力的记法，正则表达式，它能够描述正文的各种模式。我们已经在本书中许多地方零星地用过正则表达式，但还没有给它准确的定义。正则表达式是我们很熟悉的东西，不需要多少解释也能理解。虽然正则表达式在 Unix程序设计环境里随处可见，但在其他的系统里使用得却没有这么广泛，因此，在这一节里，我们想显示正则表达式的某些威力。可能你手头没有正则表达式函数库，我们也要在这里给

China-pub.com
下载

出一个初步的实现。

正则表达式有多种不同的风格，但它们在本质上都是一样的，是一种描述文字字符模式的方法，其中包括重复出现、不同选择以及为某些字符类(如数字、字母)而提供的缩写形式等等。人们最熟悉的一个例子就是所谓“通配符”，它用在命令处理器或者外壳中，用于描述被匹配文件名的模式。典型的使用方式是令*表示“字符的任意序列”，这样，下面的命令：

```
C:\> del *.exe
```

用的是一个模式，该模式将与一些文件相匹配，只要文件名是以“.exe”结尾的某个字符串。对于这种模式的作用，不同系统之间也可能有细微差别，程序与程序间也可能这样，这也不足为奇。

对于正则表达式，不同程序的处理确实存在差别，这可能使人认为它不过是某种信手拈来的东西。实际上，正则表达式也是一种语言，在语言中所有能说的东西都有严格的意义。进一步说，正则表达式的正确实现的运行速度很快。理论和工程实践的结合方式不同有可能造成很大的差异，第2章里提过这方面的例子，讲过特殊算法的作用。

一个正则表达式本身也是一个字符序列，它定义了一集能与之匹配的字符串。大部分字符只是简单地与相同字符匹配，例如正则表达式abc将匹配同样的字符序列，无论它出现在什么地方。在这里还有几个**元字符**(metacharacter)，它们分别表示重复、成组或者位置。在Unix通行的正则表达式中，字符^表示字符串开始，$表示字符串结束。这样，^x只能与位于字符串开始处的x匹配，x$只能匹配结尾的x，^x$只能匹配单个字符的串里的x，而^$只能匹配空串。

字符“.”能与任意字符匹配。所以，模式x.y能匹配xay、x2y等等，但它不能匹配xy或xaby。显然^.$能够与任何单个字符的串匹配。

写在方括号[]里的一组字符能与这组字符中的任一个相匹配。这样[0123456789]能与任何数字匹配。这个模式也可以简写为[0-9]。

这些就是基本构件，对它们可以用括号结成组，用|表示两个里面选一个，用*表示零次或者多次重复出现，+表示一次或多次出现，而?表示零或一次出现。最后，\作为一种前缀，用于引出一个元字符，关闭它的特殊意义。例如*表示的就是*本身，而\\表示的就是字符\。

最有名的正则表达式工具就是前面已经多次提到过的grep程序。这个程序是一个极好的例子，它确实显示出记法的价值。grep将一个正则表达式作用于输入的每一行，打印出所有包含匹配字符串的行。这是一个非常简单的规范，但是，借助于正则表达式的威力，它能够完成许多我们天天都能遇到的工作。在下面例子里，请注意作为grep参数的那些正则表达式，其语法形式与用于表示文件名集合的通配符差别很大，这种差异正反映出它们的不同用途。

哪些文件里用到类Regexp？

```
% grep Regexp *.java
```

这个类的实现在哪里？

```
% grep 'class.*Regexp' *.java
```

我在哪里保存着Bob发来的电子邮件？

```
% grep '^From:.* bob@' mail/*
```

在这个程序里有多少空行？

```
% grep '.' *.c++ | wc
```

China-pub.com 下载

加上各种标志开关，如打印匹配行的行号、对匹配计数、做区分大小写的匹配、在相反的意义下工作(选择那些不匹配的行)以及这些基本想法的许多其他变形，grep被使用得如此广泛，以至它已经成为基于工具的程序设计的典型实例。

不幸的是，并不是每个环境里都提供了grep或者它的等价物。有的系统提供了一个正则表达式库，通常称为regex或regexp，你可以用它写一个自己的grep版本。如果这些都没有，要实现正则表达式的一个适当子集也不是件难事。下面我们要给出正则表达式的一个实现，同时给出一个grep。为了简单起见，这里采用的元字符只包括 ^ $.和*，用*表示位于它前面的单个圆点或一个字符的重复出现。这个子集已经提供了一般正则表达式的大部分威力，但程序的复杂性却小得多。

让我们从匹配函数本身入手。这个函数的工作就是确定一个正文串的什么地方与一个正则表达式匹配：

```
/* match: search for regexp anywhere in text */
int match(char *regexp, char *text)
{
    if (regexp[0] == '^')
        return matchhere(regexp+1, text);
    do {    /* must look even if string is empty */
        if (matchhere(regexp, text))
            return 1;
    } while (*text++ != '\0');
    return 0;
}
```

如果正则表达式的开头是^，那么正文必须从起始处与表达式的其余部分匹配。否则，我们就沿着串走下去，用matchhere看正文是否能在某个位置上匹配。一旦发现了匹配，工作就完成了。注意这里do-while的使用，有些表达式能与空字符串匹配 (例如：$ 能够在字符串的末尾与空字符串匹配，.*能匹配任意个数的字符，包括0个)。所以，即使遇到了空字符串，我们也还需要调用matchhere。

递归函数matchhere完成大部分匹配工作：

```
/* matchhere: search for regexp at beginning of text */
int matchhere(char *regexp, char *text)
{
    if (regexp[0] == '\0')
        return 1;
    if (regexp[1] == '*')
        return matchstar(regexp[0], regexp+2, text);
    if (regexp[0] == '$' && regexp[1] == '\0')
        return *text == '\0';
    if (*text!='\0' && (regexp[0]=='.' || regexp[0]==*text))
        return matchhere(regexp+1, text+1);
    return 0;
}
```

如果正则表达式为空，就说明我们已经到达了它的末端，因此已经发现了一个匹配；如果表达式的最后是$，匹配成功的条件是正文也到达了末尾；如果表达式以一个圆点开始，那么它能与任何字符匹配。否则表达式一定是以某个普通字符开头，它只能与同样的字符匹配。这里把出现在正则表达式中间的^或$都当作普通字符看待，不再作为元字符。

注意，当matchhere完成模式与字符串中一个字符的匹配之后，就会递归地调用自己。

China-pub.com
下载

所以，函数递归的深度将与模式的长度相当。

在这里，惟一不容易处理的情况就是在表达式开始处遇到了带星号字符，例如 x*。这时我们调用matchstar，其第一个参数是星号的参数 (在上面的例子里是x)，随后的参数是位于星号之后的模式，以及对应的正文串。

```
/* matchstar: search for c*regexp at beginning of text */
int matchstar(int c, char *regexp, char *text)
{
    do {    /* a * matches zero or more instances */
        if (matchhere(regexp, text))
            return 1;
    } while (*text != '\0' && (*text++ == c || c == '.'));
    return 0;
}
```

在这里又遇到了do-while。之所以这样写，原因同样是x*也能与0个字符匹配。在这个循环里，检查正文能否与表达式的剩余部分匹配，只要被检查正文的第一个字符与星号的参数匹配，程序就会继续检查随后的位置。

这是一段完全可以接受的、并不太复杂的实现程序。它也说明正则表达式不需要高级的技术就可以投入使用。

我们不久将会展示一些扩充这个代码的想法。现在，让我们先写一个 grep的版本，它调用match函数。这里是主函数：

```
/* grep main: search for regexp in files */
int main(int argc, char *argv[])
{
    int i, nmatch;
    FILE *f;

    setprogname("grep");
    if (argc < 2)
        eprintf("usage: grep regexp [file ...]");
    nmatch = 0;
    if (argc == 2) {
        if (grep(argv[1], stdin, NULL))
            nmatch++;
    } else {
        for (i = 2; i < argc; i++) {
            f = fopen(argv[i], "r");
            if (f == NULL) {
                weprintf("can't open %s:", argv[i]);
                continue;
            }
            if (grep(argv[1], f, argc>3 ? argv[i] : NULL) > 0)
                nmatch++;
            fclose(f);
        }
    }
    return nmatch == 0;
}
```

令C程序在成功结束时返回一个0值，对各种失败都返回非0值，这也是一种规矩。在我们的程序grep里，就像一般的Unix版本一样，成功被定义为至少发现了一个匹配行，如果存在匹配就返回0值；找不到匹配时返回值为1；如果出现错误，程序返回2(通过eprintf)。这些返回

China-pub.com 下载

值可以在其他程序里检查，例如在操作系统外壳里。

函数grep扫描一个文件，对其中的每个行调用match：

```
/* grep: search for regexp in file */
int grep(char *regexp, FILE *f, char *name)
{
    int n, nmatch;
    char buf[BUFSIZ];

    nmatch = 0;
    while (fgets(buf, sizeof buf, f) != NULL) {
        n = strlen(buf);
        if (n > 0 && buf[n-1] == '\n')
            buf[n-1] = '\0';
        if (match(regexp, buf)) {
            nmatch++;
            if (name != NULL)
                printf("%s:", name);
            printf("%s\n", buf);
        }
    }
    return nmatch;
}
```

主程序在无法打开文件时并不直接结束。这个设计是一种选择，考虑到人们经常会做的某些事，如：

```
% grep herpolhode *.*
```

既而发现目录下的某个文件无法打开。让grep在报告了问题后继续工作下去可能是更合适的，这里不应该立即放弃，因为那样做就会迫使用户一个个地输入文件名，以回避那些出了问题的文件。另外还请注意，grep在一般情况下打印出文件名和匹配的行，但当它读标准输入文件时就不输出文件名。这种设计看起来好像很奇怪，其实它反映的是实际使用中的一种习惯性方式。在只提供一个文件名时，grep的工作常常是做选择，附加上的文件名就将成为赘物。如果用它检索许多文件，人们要做的通常是发现某些东西的所有出现，这时文件名就成为很有帮助的东西了。请比较：

```
% strings markov.exe | grep 'DOS mode'
```

和

```
% grep grammer chapter*.txt
```

这些想法已经接触到grep为什么变得如此大众化的基础，它也说明了另一个问题：好的记法只有与人们的工程经验相结合，才能产生出自然而又有效的工具。

我们的match在发现了一种匹配后就立即返回。对于grep而言，这是一种很好的默认行为方式。但是如果要实现的是文本编辑器中的那种代换操作（搜索和替换），实现**最左最长**的匹配就更是合适的。例如，给定的文本是“ aaaaa ”，模式a*能与文本开始的空字符串匹配，但是，看起来最好还是让它匹配所有的五个 a。要使match能找到最左最长的匹配串，matchstar必须重新写成最贪婪的：它不应该从左向右一个个地查看字符，而应该跳过最长的能与带星号字符匹配的串，只有在串的剩余部分不能与模式的剩余部分匹配时才往后退。换句话说，它应该从右向左工作。下面是实现最左最长匹配的 matchstar版本：

```
/* matchstar: leftmost longest search for c*regexp */
int matchstar(int c, char *regexp, char *text)
```

China-pub.com
下载


```
{
    char *t;

    for (t = text; *t != '\0' && (*t == c || c == '.'); t++)
        ;
    do {    /* * matches zero or more */
        if (matchhere(regexp, t))
            return 1;
    } while (t-- > text);
    return 0;
}
```

对于grep而言，发现的匹配是什么并不重要，因为它只需要检查匹配的存在性，并打印出相应的整个行。由于最左最长匹配需要做许多附加工作，这对 grep而言并不是必须的，但是对于实现一个替换操作而言，这种功能就是最基本的了。

如果不考虑正则表达式的形式，我们的 grep还是能和系统提供的东西相比美的。不过，这里也存在病态表达式，它们可能导致指数式的行为。例如在用模式 a*a*a*a*a*b去匹配输入串aaaaaaaaac的时候。实际上，指数式行为也出现在某些商品的实现中。在 Unix里有一个grep的变体，名字是egrep，它使用一种非常复杂的匹配算法，在部分匹配失败后它能够避免进行回溯，这样就能保证线性的性能。

怎样使match能处理所有的正则表达式？为此就必须包括：像 [a-zA-Z]那样的字符组与各个字母的匹配，能够把元字符引起来(例如需要查找正文中的一个圆点)，形成分组的括号，以及选择(abc或def)等。要做的第一步，应该是把模式翻译到某种表示形式，使它很容易查看。在与一个字符做比较时，如果每次都要检查整个字符组，是非常费时间的，用一个基于位向量的预先做出的表示形式，就能很有效地实现字符组。对于完全的正则表达式，带有括号和选择，实现起来必定更加复杂。在本章后面部分要讲到的某些技术可以用在这里。

练习9-6　如果做普通正文的查找，match的执行性能与strstr比较起来怎么样？

练习9-7　写一个非递归的matchhere版本，将它与递归的版本做性能比较。

练习9-8　给grep增加几个选项。常见的包括：-v表示颠倒匹配的含义，-i表示对字母做区分大小写的匹配，-n表示在输出中指明行号。行号应该如何输出？它们是否应该与被匹配的正文输出在同一行里？

练习9-9　扩充match，增加 +(一个或多个)和?(0个或一个)。模式a+bb?应该匹配一个或多个a后跟一个或两个b。

练习9-10　当^和$不出现在表达式的开头或结尾，或者 *不是紧跟在普通文字字符或圆点的后面时，目前的match实现都不使用它们作为元字符的特殊意义。一种更习惯的实现是在元字符前放一个反斜线符号，表示将它引起来。修改 match，使之能以这种方式处理反斜线符号。

练习9-11　给match增加字符组功能。字符组表示的是一个模式，它能与方括号里任一个字符匹配。要使字符组使用更方便，还应该再加上范围描述，例如 [a-z]能匹配所有的小写字母；翻转匹配的含义，例如[^0-9]能匹配所有不是数字的字符。

练习9-12　改造match，使用最左最长匹配的matchstar版本；再修改它，使它能返回被匹配字符串的开始和结束位置。用这个 match写一个程序gres，它与grep类似，但打印的是用一个新字符串替代了被匹配串之后的字符行。

China-pub.com 下载

```
% gres 'homoiousian' 'homoousian' mission.stmt
```

练习9-13 修改match和grep，使它们能处理Unicode字符的UTF-8串。由于UTF-8和Unicode都是ASCII的超集，这种修改是向上兼容的。除了被检索的正文，正则表达式本身也需要改造，使之能正确使用UTF-8。现在字符组又该如何实现呢？

练习9-14 写一个正则表达式的自动测试程序，它应能生成测试表达式和被检索的测试串。如果你能用某个现成的库作为实现时的参考，或许还能发现其中的程序错误。

9.3 可编程工具

许多工具都是围绕着一个专用语言构造起来的。grep程序不过是常见工具集合中的一个，这些工具使用正则表达式或者其他什么语言来解决程序设计中的问题。

这方面最早的例子是命令解释器或作业控制语言。人们很早就认识到，应该能把常用的命令序列写在一个文件里，这种文件可以由命令解释器的一个实例，或者说是一个外壳作为输入来执行。从这里出发，向前迈出一小步，就是加上参数、条件、循环、变量以及其他东西，所有这些都是从常规程序设计语言里搬来的。最主要的差别是在这里只有一种数据类型，即字符串，而外壳程序的基本运算通常都是完整的程序，它们能完成具有实质意义的计算工作。虽然外壳程序设计的辉煌时代已经过去了，它的领地也丢给了其他替代品，比如命令环境中的Perl语言，图形用户界面上的按钮。但是，它仍然是从简单片段构造出复杂操作的一种非常有效的手段。

Awk是另一种可编程工具，它带有一种很小的、特定的模式匹配语言，主要用于对输入流进行选择和变换。正如我们在第3章已经看到的，Awk程序能自动读入输入文件，把每个行分解为一些域，分别称作 $1到 $NF，这里的NF是一行中域的个数。通过对许多常见工作提供相关的默认行为方式，使人可以用Awk做出许多很有用的单行程序。例如，下面就是一个完整的Awk程序，

```
# split.awk: split input into one word per line
{ for (i = 1; i <= NF; i++) print $i }
```

它打印各个输入行里的词，一个词一行。再看看另一方向的东西。下面是fmt的一个实现，该程序用词填起每个输出行，每行不超过60个字符。空行将导致分段。

```
# fmt.awk: format into 60-character lines

/./  { for (i = 1; i <= NF; i++) addword($i) } # nonblank line
/^$/ { printline(); print "" }                 # blank line
END  { printline() }

function addword(w) {
    if (length(line) + 1 + length(w) > 60)
        printline()
    if (length(line) == 0)
        line = w
    else
        line = line " " w
}

function printline() {
    if (length(line) > 0) {
        print line
        line = ""
    }
}
```

China-pub.com
下载

我们常用fmt对电子邮件或其他短文件重新做分段整理。我们也用它对第3章Markov程序的输出做格式化。

可编程工具常常起源于一个小语言，该语言可能是为在某个较窄领域里自然地表达问题的解而设计的。Unix工具eqn是个很好的例子，它完成数学表达式的显示排版。eqn的输入语言接近数学家读数学公式的表达方式：$\frac{\pi}{2}$ 写为pi over 2。TEX采用了类似形式，它对上面公式的记法是\pi\over 2。对于你要去解决的问题，如果存在一种自然的或人们熟悉的记法，那么就应该使用它，或者是修改它，不要总是从头开始。

Awk是由另一个程序获得的灵感，该程序利用正则表达式在电话流通记录中标记出反常的数据。当然，由于Awk包含了变量、表达式和循环等等，这使它成了一个真正的程序设计语言。Perl和Tcl在开始设计时，也是为了将小语言的方便性和表达能力与大语言的威力结合到一起。它们都是真正的通用程序设计语言，虽然最常见的应用是处理正文。

这类工具的通用名称是**脚本语言**，因为它们是从早期的命令解释器发展起来的，其可编程能力受到一定限制，只能执行包装起来的程序脚本。脚本语言创造性地使用正则表达式，不仅仅是做模式匹配——即识别某种特定模式的存在——也用于标识需要做变换的正文区域。在下面的Tcl程序里，两个regsub(regular expression substitution，正则表达式代换)做的就是这种事情。这个程序是我们在第4章给出的股票行情提取程序的一个扩充，它按给定的第一个参数值取得URL。其中的第一个替换用于去掉http://(如果存在的话)；第二个替换把遇到的第一个/换成空格，这就把参数分成了两个域。lindex命令从串里取出第一个域(用下标0)。括在[]里面的正文将被作为Tcl命令执行，并用结果正文串替代；$x用变量x的值替代。

```
# geturl.tcl: retrieve document from URL
# input has form [http://]abc.def.com[/whatever...]

regsub "http://" $argv "" argv   ;# remove http:// if present
regsub "/" $argv " " argv        ;# replace leading / with blank

set so [socket [lindex $argv 0] 80] ;# make network connection
set q "/[lindex $argv 1]"

puts $so "GET $q HTTP/1.0\n\n"   ;# send request
flush $so
while {[gets $so line] >= 0 && $line != ""} {} ;# skip header
puts [read $so]                  ;# read and print entire reply
```

在典型的情况下，这个脚本将产生大量输出，其中许多是由 <和>括起来的HTML标志。用Perl做正文替换非常方便，所以我们的下一个工具是个Perl脚本，它利用正则表达式和替换，去掉文本里的HTML标志。

```
# unhtml.pl: delete HTML tags

while (<>) {              # collect all input into single string
    $str .= $_;           # by concatenating input lines
}

$str =~ s/<[^>]*>//g;     # delete <...>
$str =~ s/ / /g;     # replace   by blank
$str =~ s/\s+/\n/g;       # compress white space
print $str;
```

如果不懂Perl，这段东西看起来就像是密码。结构

```
$str =~ s/regexp/repl/g
```

在字符串str的正文里，把与正则表达式regexp匹配的(最左最长)串替换为repl，最后的g表示要全局性地做，也就是说，对串中所有匹配做这件事，而不是仅处理第一个匹配。元字

China-pub.com 下载

符序列\s是一个缩写形式，表示所有的空白字符(空格、制表符、换行一类东西)；\n表示换行字符。串“ ”是HTML里的一个字符，就像第2章说明的那些，它定义一个不允许断开的空白字符。

把所有这些东西放到一起，就构成了一个能力不强但也可以工作的网络浏览器，实现它只要写一个单行的外壳程序：

```
# web: retrieve web page and format its text, ignoring HTML
geturl.tcl $1 | unhtml.pl | fmt.awk
```

它能提取网页，丢掉其中所有的控制和格式信息，然后按照自己的规矩重新进行格式化。这是从网络上快速获取页面的一种途径。

注意，在这里我们以串联方式同时使用了多种语言：Tcl、Perl和Awk，每种语言都有特别适合的工作，在每种语言里都有正则表达式。记法的威力就在于对每个问题都可能有最合适的工具。用Tcl完成通过网络获取正文的工作十分方便；Perl和Awk适合做正文的编辑和格式化；当然，还有正则表达式，它最适合用来刻画作为查找和修改对象的正文片段。这些语言放在一起，其威力远大于它们中的任何一个。把工作分解成片段有时是很值得做的，如果这样做有可能使你得益于某些正确的记法。

9.4 解释器、编译器和虚拟机

一个程序怎样才能从它的源程序代码形式进入执行？如果这个语言足够简单，就像出现在printf里的或者是前面最简单的正则表达式那样，我们可以直接对照着源代码执行。这非常容易，可以立即就开始做。

在设置方面的时间开销和执行深度之间有一种此消彼长的关系。如果语言更复杂些，人们通常就会希望做一些转换，把源代码转换为一种对执行而言既方便又有效的内部表示形式。处理原来的源代码需要用一些时间，但这可以从随后的快速执行中得到回报。有些程序里综合了转换和执行的功能，它们能读入源代码正文，对其做转换后运行之，这种程序称作“解释器”。Awk和Perl是解释性的，许多其他脚本语言和专用语言也是这样。

第三种可能性是为特定种类的计算机(程序要在这里执行)生成指令代码，编译器做的就是这件事。这种做法事先要付出极大的努力，但能导致随后最快速的执行。

也存在着另一种组合方式，这就是我们在本节里准备研究的：把程序编译成某种假的计算机(虚拟机)的指令，而这种虚拟机可以用任何实际计算机进行模拟。虚拟机综合了普通解释和编译的许多优点。

如果语言很简单，要弄清程序的结构并把它转换为某种内部形式，并不需要做多少处理工作。当然，如果语言里存在着一些复杂性——有说明、嵌套结构、递归定义的语句或表达式、带优先级的运算符以及其他一些类似东西——要正确地剖析输入，弄清楚它的结构就麻烦多了。

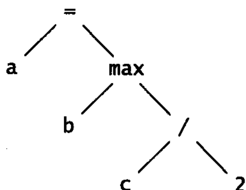
分析程序(parser)常常是借助于某个分析程序自动生成器写出来的，这种工具也被称为编译程序的编译程序，例如yacc或bison等。这种工具能从语言的描述(语言的语法)出发，转换出(典型情况是)一个C或C++程序。这个程序经过编译后，就能把该语言的语句转换到对应的内部形式。当然，由语法描述出发能生成一个剖析程序，这也是好记法威力的一个例证。

由分析程序生成的表示形式通常是树，其内部结点包含着运算符号，叶结点包含的是运算对象。下面这个语句：

China-pub.com
下载


```
a = max(b, c/2);
```

可能产生下面这个分析树(或称语法树)：



在第2章描述的许多树算法可以用来构造或者处理分析树。

一旦这种树构造好了，我们就有许多可能的方法来处理它。最直接的方法 (也是Awk所采用的)是直接在树中运动，并在这个过程中求出各个结点的值。要完成对这种整数表达式语言的求值过程，下面给出一个简化的程序版本，它采用后序遍历方式：

```
typedef struct Symbol Symbol;
typedef struct Tree Tree;

struct Symbol {
    int     value;
    char    *name;
};

struct Tree {
    int     op;          /* operation code */
    int     value;       /* value if number */
    Symbol  *symbol;     /* Symbol entry if variable */
    Tree    *left;
    Tree    *right;
};

/* eval: version 1: evaluate tree expression */
int eval(Tree *t)
{
    int left, right;

    switch (t->op) {
    case NUMBER:
        return t->value;
    case VARIABLE:
        return t->symbol->value;
    case ADD:
        return eval(t->left) + eval(t->right);
    case DIVIDE:
        left = eval(t->left);
        right = eval(t->right);
        if (right == 0)
            eprintf("divide %d by zero", left);
        return left / right;
    case MAX:
        left = eval(t->left);
        right = eval(t->right);
        return left>right ? left : right;
    case ASSIGN:
        t->left->symbol->value = eval(t->right);
        return t->left->symbol->value;
    /* ... */
    }
}
```

China-pub.com 下载

前几个分情况完成的是对最简单的表达式求值，例如那些常量和值等；随后的情况对算术表达式求值；其他的可能是做另一些特殊处理，如条件语句和循环等。要实现这些控制结构，树中还需要附加其他信息，以表示控制流。这里都没有给出来。

就像在前面处理pack和unpack那样，我们也可以用一个函数指针的表取代这里显示的开关语句。各运算符对应的函数和上面开关语句里的差不多：

```
/* addop: return sum of two tree expressions */
int addop(Tree *t)
{
    return eval(t->left) + eval(t->right);
}
```

一个函数指针的表将运算符关联到那些实际执行操作的函数：

```
enum {  /* operation codes, Tree.op */
    NUMBER,
    VARIABLE,
    ADD,
    DIVIDE,
    /* ... */
};

/* optab: operator function table */
int (*optab[])(Tree *) = {
    pushop,     /* NUMBER */
    pushsymop,  /* VARIABLE */
    addop,      /* ADD */
    divop,      /* DIVIDE */
    /* ... */
};
```

求值过程以运算符为指标，从表里得到函数指针，随之去调用正确的函数。下面的版本将递归地调用其他函数。

```
/* eval: version 2: evaluate tree from operator table */
int eval(Tree *t)
{
    return (*optab[t->op])(t);
}
```

上面这两个eval版本都是递归的。存在着一些去除递归的方法，包括一种称为**线索化代码**的巧妙技术，它能使调用栈完全瘪下来。要去掉所有递归，最巧妙的方法是把有关函数存入一个数组，然后线性地穿过这个数组，执行对应的程序。这个数组实际上已经变成了由一个小的特殊机器执行的指令序列。

我们仍然需要用一个栈来表示计算过程中部分计算出的结果。这时函数的形式也要做些改变，不过这种转换是很容易的。这样，我们已经发明了一种堆栈机器，它的指令是一些小函数，而运算对象都存在一个独立的运算对象栈里。这并不是一个真正的机器，但我们可以像使用真的机器一样为它编程，同时也很容易把它实现为一个解释器。

我们将不再需要通过在树上的运动，对树进行求值，而是生成一个执行这个程序的函数数组。数组里也应该包含指令所使用的数据值，例如常数和变量 (符号)等，因此这个数组的元素类型应该是一个联合：

```
typedef union Code Code;
```

China-pub.com 下载


```
union Code {
    void    (*op)(void); /* function if operator */
    int     value;       /* value if number */
    Symbol  *symbol;     /* Symbol entry if variable */
};
```

下面是函数generate，它生成一些函数指针，并把它们放入以这些东西为内容的数组 code里。generate的返回值不是表达式的值——这种值在所生成代码被真正执行的时候才产生——而是一个code下标，随后再生成的操作将被放在这里：

```
/* generate: generate instructions by walking tree */
int generate(int codep, Tree *t)
{

    switch (t->op) {
    case NUMBER:
        code[codep++].op = pushop;
        code[codep++].value = t->value;
        return codep;
    case VARIABLE:
        code[codep++].op = pushsymop;
        code[codep++].symbol = t->symbol;
        return codep;
    case ADD:
        codep = generate(codep, t->left);
        codep = generate(codep, t->right);
        code[codep++].op = addop;
        return codep;
    case DIVIDE:
        codep = generate(codep, t->left);
        codep = generate(codep, t->right);
        code[codep++].op = divop;
        return codep;
    case MAX:
        /* ... */
    }
}
```

处理语句a= max(b,c/2)生成的代码看起来是下面的样子：

```
pushsymop
b
pushsymop
c
pushop
2
divop
maxop
storesymop
a
```

有关运算符函数将对堆栈进行操作，弹出运算对象并压入结果。

解释器本身就是一个循环，它用一个程序计数器，在函数指针数组里移动：

```
Code code[NCODE];
int stack[NSTACK];
int stackp;
int pc; /* program counter */

/* eval: version 3: evaluate expression from generated code */
int eval(Tree *t)
```

China-pub.com 下载

```
    {
        pc = generate(0, t);
        code[pc].op = NULL;

        stackp = 0;
        pc = 0;
        while (code[pc].op != NULL)
            (*code[pc++].op)();
        return stack[0];
    }
```

在我们发明的这个堆栈机器里，上述循环实际上是以软件方式模拟着真实计算机的硬件所做的事情。下面是几个有代表性的运算函数：

```
/* pushop: push number; value is next word in code stream */
void pushop(void)
{
    stack[stackp++] = code[pc++].value;
}

/* divop: compute ratio of two expressions */
void divop(void)
{
    int left, right;

    right = stack[--stackp];
    left = stack[--stackp];
    if (right == 0)
        eprintf("divide %d by zero\n", left);
    stack[stackp++] = left / right;
}
```

注意，对除数为0的检查出现在divop中，而不放在generate里。

检查条件、分支和循环的执行等，在运算符函数里表现为直接修改程序计数器，这样做就产生了分叉，到达函数数组中的另一个地方。例如，一个goto运算符就是直接设置pc变量，而一个条件分支只在条件为真的情况下设置 pc。

code数组是解释器内部的东西。当然，我们也可以设想把生成出来的程序存入一个文件。如果直接把函数的地址写出去，得到的结果将是不可移植的和脆弱的。我们应该换一种方式，写出去一些代表函数的常数，例如用 1 000表示addop，用1001表示pushop等等。在重新读入程序准备执行时，再把这些常数翻译回函数指针。

如果我们查看由上面的过程产生的文件，它们就像是虚拟机的指令流，虚拟机的指令实现的是我们小语言的基本运算符，而 generate函数实际上就是个编译程序，它把我们的语言翻译到对应的机器。虚拟机是一个可爱的历史悠久的想法，最近，由于 Java和Java虚拟机(JVM)的出现，这种想法又变得时髦起来。要从高级语言程序产生出一种可移植并且高效的表示形式，虚拟机方法实际上指出了一条康庄大道。

9.5 写程序的程序

函数generate最值得注意的地方，或许就在于它是一个写程序的程序：它的输出是另一个(虚拟)机器可以执行的指令序列。编译程序每时每刻都在做这件事：把源代码翻译成机器指令，所以这种想法是人人皆知的。在实践里，写程序的程序可能以许多不同的形式出现。

一个最常见的例子是网页 HTML的动态生成。HTML是一种语言，其功能当然非常有限，

China-pub.com
下载

它还可以包含JavaScript代码。网页常常是由Perl或者C程序在运行中生成的，其具体内容(例如，某些查询结果或者是定向广告)根据外来需求确定。我们对这本书里的各种图形、表格、数学表达式以及索引等使用了多种特殊的语言。作为另一个例子，PostScript也是一种程序设计语言，它可以由文字处理程序、画图程序和许许多多其他程序生成。在处理的最后阶段，本书就被表示为一个大约有57 000行的PostScript程序。

一个文档是一个静态的程序，使用某个程序语言作为记法，表达各种问题领域是一种非常有力量的思想。许多年以前，程序员就梦想着有朝一日计算机能帮他们写所有的程序，当然这可能永远都只是一个梦。但是，今天的计算机已经在日复一日地为我们写程序，经常是在写那些我们原来从未想到还可能表达为程序的东西。

最常见的写程序的程序是编译器，它能把高级语言程序翻译成机器代码。然而，把代码翻译到主流程序设计语言常常也很有用。在前面的章节里，我们曾提到的分析程序生成器能把一个语言的语法定义转换为一个能分析这个语言的C程序。C常常被作为一种“高级的汇编语言”使用。有些通用的高级语言(例如Modula-3和C++)的第一个编译程序产生的就是C代码，这种代码随后用标准C编译系统编译处理。这种方式有不少优点，包括效率——因为这些程序原则上可以运行得像C程序一样快，可移植性——因为该编译系统可以搬到任何有C编译器的系统上。这大大促进了这些语言的早期传播。

作为另一个例子，Visual Basic的图形界面生成一组Visual Basic赋值语句，完成用户通过鼠标在屏幕上选择的那些对象的初始化工作。许多其他语言也有“可视化的”开发环境和“巫师(wizard)”，它们都能从鼠标的点击中综合产生出各种用户界面的代码。

尽管各种程序生成器的威力如此强大，尽管有这么多极好的例子，记法问题还是没有得到它应有的尊重，仍然很少被程序员们使用。实际中确实存在着大量的用程序生成小规模代码的机会，所以，你自己应该学会从这里获得一些益处。下面是生成C或者C++代码的几个例子。

Plan 9操作系统由一个头文件生成它的错误信息，该文件里包含一些名字和注释；这些注释被机械性地转换成带引号的字符串，放进一个数组里，而该数组由枚举值做为下标。下面这段代码显示了头文件的结构：

```
/* errors.h: standard error messages */

enum {
    Eperm,      /* Permission denied */
    Eio,        /* I/O error */
    Efile,      /* File does not exist */
    Emem,       /* Memory limit reached */
    Espace,     /* Out of file space */
    Egreg       /* It's all Greg's fault */
};
```

有了这个输入，一个简单的程序就能产生下面这段错误信息声明：

```
/* machine-generated; do not edit. */

char *errs[] = {
    "Permission denied", /* Eperm */
    "I/O error", /* Eio */
    "File does not exist", /* Efile */
    "Memory limit reached", /* Emem */
    "Out of file space", /* Espace */
    "It's all Greg's fault", /* Egreg */
};
```

China-pub.com 下载

这种方式有许多优点。首先，enum值和它们所表示的字符串之间的关系在书面上自成文档，很容易做到对自然语言的独立性。此外，信息只出现一次，只有“一个真理点”，所有其他代码都由此生成。这样，如果需要做更新，那么就只有一个地方需要考虑。如果这些信息出现在许多地方，很难避免在某些时候它们之间失去同步性。最后，我们很容易对 .c文件做出适当安排，使得在这个头文件改变时，所有有关的文件都重新建立和重新编译。当某个错误信息必须改变时，需要做的所有事情就是修改这个头文件，并重新编译整个操作系统。这样所有信息都能得到更新。

生成程序可以用任何语言来写。用Perl一类字符串处理语言写起来更容易：

```
# enum.pl: generate error strings from enum+comments

print "/* machine-generated; do not edit. */\n\n";
print "char *errs[] = {\n";

while (<>) {
    chop;                           # remove newline
    if (/^\s*(E[a-z0-9]+),?/) {     # first word is E...
        $name = $1;                 # save name
        s/.*\/\* *//;               # remove up to /*
        s/ *\*\///;                 # remove */
        print "\t\"$_\", /* $name */\n";
    }
}

print "};\n";
```

正则表达式又参与了这里的行动。所有第一个域是标识符后跟着逗号的东西都被选中[⊖]，第一个代换删掉直到注释里第一个非空字符之前的所有东西，第二个代换删去注释结尾和它前面的所有空格。

作为编译系统测试工作的一部分，Andy Koenig开发了一种写C++ 代码的方便方法，以检查编译系统能否捕捉到各种程序错误。他给所有能导致编译诊断的代码段都加上一个奇妙的注释，描述应该出现的错误信息。每个这种注释行都以///开头(以便区别于普通的注释)，还包含一个能与相关诊断信息做匹配的正则表达式。例如，下面这两个代码段都应该产生诊断信息：

```
int f() {}
    /// warning.* non-void function .* should return a value

void g() {return 1;}
    /// error.* void function may not return a value
```

如果用我们的C++ 编译系统处理到第二个测试实例，它打印出的正是我们所期望的，能与对应正则表达式匹配的信息：

```
% CC x.c
"x.c", line 1: error(321): void function may not return a value
```

每个这种代码片段都提交给编译系统，然后将产生的输出与所期望的诊断信息做比较，这个过程通过shell和Awk程序的结合来管理，出现失误时将指明是哪个测试实例使编译系统的输出与期望的不同。由于在注释里用的是正则表达式，输出可以有些自由度，根据情况不同，可以把它们写成或多或少具有某些宽容性。

使用带语义的注释不是什么新想法，PostScript里就有这种东西。在PostScript里，注释由

⊖ 实际上是以E开始的标识符，后跟逗号。——译者

China-pub.com
下载

%开头，而那些由%%开始的注释，按照习惯，带有一些附加信息，可以描述关于页号、限界盒(bounding box)、字体名以及其他类似的东西：

```
%%PageBoundingBox: 126 307 492 768
%%Pages: 14
%%DocumentFonts: Helvetica Times-Italic Times-Roman
            LucidaSans-Typewriter
```

在Java里，以/**开始*/结束的注释被用于建立跟随其后的类定义的文档。对自带文档代码做大范围的推广，就成为所谓的带文档程序设计 (literate programming)，就是说，把一个程序和它的文档集成在一起。这样，通过某种过程，其中的文档可以按自然顺序打印出来供人阅读，另一个过程则以正确顺序安排它去进行编译。

对于上面的所有实例，仔细研究其中记法的作用、语言的混合方式以及工具的使用等，都是很有价值的。这几个方面的组合能够放大各个成分的威力。

练习9-15 有一个历史很久远的有关计算的故事，要求写出一个程序，执行后恰好产生其自身原来的形式。这是用程序写程序的一个非常特殊而又很巧妙的例子。请在你自己最喜欢的语言里试一试。

9.6 用宏生成代码

现在向下降几个层次，我们同样也能写这样的宏，它们在编译时能够产生代码。在这本书里我们始终在告诫读者，应该反对使用宏和条件编译，因为它们鼓励的是一种破绽百出的程序设计风格。但是，宏也确实有它自己的位置，有时正文替换恰恰就是问题的正确解决方法。这里有一个例子，是关于如何利用C/C++ 的宏预处理装配起同样风格的重复出现的代码片段。

在第7章里我们提到过估计语言基本结构速度的程序，它就利用了 C的预处理功能，通过把具体代码装入一段框架，组装起一大批测试例子。有关测试的基本构架是封装起来的一段代码，其中有一个循环。在循环开始时设置时钟，随后将代码段运行许多次，最后停下时钟并报告结果。所有重复出现的代码段都被包裹在几个宏里面，实际被计时的代码则被作为参数传递给宏。基本的宏具有如下形式：

```
#define LOOP(CODE) {                      \
    t0 = clock();                         \
    for (i = 0; i < n; i++) { CODE; }     \
    printf("%7d ", clock() - t0);         \
}
```

写在最后的反斜线符号使宏可以延伸到多行。这个宏将被用在“语句”里，应用的形式大致是下面的样子：

```
LOOP(f1 = f2)
LOOP(f1 = f2 + f3)
LOOP(f1 = f2 - f3)
```

有时需要用另一些语句完成初始化工作，但最基本的计时部分都表示在这种一行一个的程序段里，它们展开后将形成很长的一段代码。

宏处理功能也可以用于生成产品代码。有一次， Bart Locanthi要写一个高效的两维图形操作程序。这种操作被称为bitblt或rasterop。要想把它们做得速度很快是十分困难的，因为在这里有许多参数，它们可能以很复杂的方式互相组合。经过认真的分析之后， Locanthi把

China-pub.com 下载

所有组合情况归结到一个循环，它可以独立地进行优化。在此之后，他用宏替换构造出每种情况，就像前面性能测试的例子那样，然后把各种变形安排在一个大的开关语句里面。原始的源代码只有几百行，宏处理的结果就变成了数千行。这样通过宏展开得到的代码并不是最优的，这是由于问题本身的复杂性。但是这种做法又非常实际，也很容易完成。另一方面，作为一段高性能代码，它也具有相对的可移植性。

练习9-16　练习7-7要求写一个程序去度量C++ 里各种操作的代价。使用本节提出的想法重新写这个程序。

练习9-17　练习7-8要求做一个Java的代价模型，在Java中没有宏的功能。我们可以通过另外写一个程序的方式来解决这个问题，在这里可以采用你所选定的任何语言（或几个语言）。令这个程序写出所需要的Java程序，并自动完成计时运行。

9.7　运行中编译

这几节我们一直在谈论写程序的程序。在前面的所有例子里，生成出来的程序都具有源代码的形式，它还需要经过编译或者解释才能运行。实际上也可能产生出能够立即投入执行的代码，只要我们生成的不是源代码而是机器指令。这种做法通常被称为“运行中”的编译，或者“即时”编译。前一个词出现得更早，而现在后一个词却更流行，包括该词　(Just In Time)的首字母缩写词JIT。

虽然这样编译产生代码必定是不可移植的——只能运行在一种类型的处理器上——但它却可能非常快。考虑表达式：

```
max(b, c/2)
```

计算过程中必须求出c，将它除以2，用得到的结果与b比较，选出其中较大的一个。如果用本章前面给出了轮廓的虚拟机计算这个具体表达式，我们可以删去在divop里对除零的检查，因为2不是0，有关的检查毫无意义。但是，无论确定采用什么样的设计，我们在对虚拟机的实现做安排时，都不可能去掉其中的检查，在任何除法运算的实现里都必须做除数与0的比较。

这也就是动态代码生成可以起作用的地方。如果直接从表达式出发构造代码，而不只是简单地串起预先定义的一些操作，在那些已知除数不是零的地方，是完全可以避免再做除零检查的。实际上还可以再前进一步，如果整个表达式就是常数，例如 max(3*3,4/2)，我们可以在产生代码的过程中对它求一次值，然后直接用常数9取代它。如果表达式出现在一个循环里，那么循环的每次执行都将节省时间。只要循环的次数足够多，我们就能把为弄清表达式和生成代码所花费的额外时间都节省回来。

这里的关键思想还是记法。记法给了我们一种表达问题的一般性方法，而这种记法的编译程序可以针对特定计算的细节生成专门代码。例如，在处理正则表达式的虚拟机里，我们最好有一个匹配文字型字符的运算：

```
int matchchar(int literal, char *text)
{
    return *text == literal;
}
```

当我们为特定模式生成代码时，由于literal的值是固定的，例如就是‘x’，那么我们最好能改用像下面这样的运算：

China-pub.com
下载


```
int matchx(char *text)
{
    return *text == 'x';
}
```

与此同时，我们不希望为每个文字字符值预先定义一个特殊运算，而是想把问题弄得更简单些，只有在表达式实际需要的时候才生成这种代码。把这个思想推广到整个的运算集合，就可以写出一个运行时的编译器，它把当时处理的正则表达式翻译成一段为此表达式而特别优化了的代码。

1967年，Ken Thompson在IBM7094机器上做正则表达式的实现时，所做的实际上就是这件事。他的程序对表达式里的各种操作生成一些小块的二进制 7094代码，再把它们串联在一起，最后通过调用来运行结果程序，就像调用普通函数一样。类似技术也可以用到图形系统里，创建一些屏幕更新的特定指令序列。由于在图形处理中存在太多的特殊情况，与其事先把它们都写出来，或在更一般的代码中加入大量条件测试，还不如对每种实际发生的情况动态地构造相关代码。后面这种方法更有效。

如果要展示实际的运行时编译系统的构造过程，就有可能使我们的讨论涉及到过多的特定指令集合的细节。但是，显示一下这种系统如何工作还是很值得的。在阅读本节余下的部分时，你主要应该了解其中的思想和见解，而不是具体的实现细节。

请回忆一下，前面我们把虚拟机写成具有下面的结构：

```
Code code[NCODE];
int stack[NSTACK];
int stackp;
int pc; /* program counter */
    ...
    Tree *t;

    t = parse();
    pc = generate(0, t);
    code[pc].op = NULL;
    stackp = 0;
    pc = 0;
    while (code[pc].op != NULL)
        (*code[pc++].op)();
    return stack[0];
```

要将这些代码弄成运行时编译，我们必须做一些修改。首先， code数组将不再是函数指针的数组，而应该是可执行指令的数组。到底这些指令的类型是 char或int或long，要看我们的编译所面对的处理器的情况，这里假定它是 int。在代码生成之后，我们准备像函数一样调用它。这里将不再需要虚拟的程序计数器，因为处理器本身有它的执行循环，它将为我们遍历这些代码，一旦计算完成，它就会返回，就像正常的函数一样。此外，我们还可以有些选择，是自己为机器维护一个运算对象栈，还是直接使用处理器的堆栈。这两种方法各有长短。在这里我们仍然使用一个独立的栈，以便把注意力集中到代码本身的细节方面。现在的实现大致是这个样子：

```
typedef int Code;
Code code[NCODE];
int codep;
int stack[NSTACK];
```

China-pub.com 下载

```
int stackp;
    ...
    Tree *t;
    void (*fn)(void);
    int pc;

    t = parse();
    pc = generate(0, t);
    genreturn(pc);      /* generate function return sequence */
    stackp = 0;
    flushcaches();      /* synchronize memory with processor */
    fn = (void(*)(void)) code;  /* cast array to ptr to func */
    (*fn)();            /* call function */
    return stack[0];
```

在generate结束后，genreturn将安排一些指令，使生成的代码能把控制返回到eval。

函数flushcaches要求完成一些动作，迫使处理器为执行新生成的代码做好准备，这是必须做的。现代计算机的速度非常快，部分原因就在于它们拥有为指令和数据而使用的各种缓冲存储器，而且还有内部的流水线，这使许多顺序指令的执行可以互相重叠。缓存和流水线都期望遇到的指令流是静态的。如果我们要求它执行刚刚生成的那些代码，处理器很可能会被搞糊涂。因此，在执行新生成的代码之前， CPU需要腾空其流水线，刷新其缓存。这些都是对具体机器有高度依赖性的操作，对各种特定类型的计算机， flushcaches的实现肯定是不同的。

最值得注意的表达式是(void(*)(void)) code，这是一个模子(Cast)[⊖]，它把包含着新生成的指令序列的数组地址转换成一个函数指针，通过它就可以像函数一样调用这些代码。

从技术上看，生成代码本身并没有太大困难，但要想有效地完成这个工作还有许多工程性的事项。让我们从某些基本构件开始。和前面一样，在编译过程中需要维护 code数组和它的一个下标。为了简单起见，这里把它们都定义为全局的，这也是前面一直采用的方式。此后，我们就可以写一个编排指令的函数了：

```
/* emit: append instruction to code stream */
void emit(Code inst)
{
    code[codep++] = inst;
}
```

指令本身可以通过与具体处理器相关的宏或者小函数来定义，它们填充指令字中各个域，装配起各种指令。假设我们有一个名为popreg的函数，它生成的代码是从堆栈里弹出一个值，并将它存入处理器的寄存器；另一个函数叫 pushreg，它生成的代码从一个寄存器取出值，并将这个值压进堆栈。我们修改后的 addop使用这些基本功能，它看起来大致是下面的样子，使用某些预先定义的常数，这些常数描述指令本身 (像ADDINST)或者其编排形式(定义有关格式的各种SHIFT位置)：

```
/* addop: generate ADD instruction */
void addop(void)
{
    Code inst;

    popreg(2);      /* pop stack into register 2 */
    popreg(1);      /* pop stack into register 1 */
    inst = ADDINST << INSTSHIFT;
```

⊖ cast，也就是类型强制转换运算符。——译者

China-pub.com
下载


```
        inst |= (R1) << OP1SHIFT;
        inst |= (R2) << OP2SHIFT;
        emit(inst);      /* emit ADD R1, R2 */
        pushreg(2);      /* push val of register 2 onto stack */
    }
```

这些只不过是个开始。如果要写一个实际的运行中编译器，我们还必须做许多优化方面的工作。例如，如果被加的是个常数，那么就不必先把它压入堆栈，然后再弹出来做加的操作，而是可以直接做加法。通过这类考虑可以清除大量多余的开销。当然，即使是写成上面的样子，这个addop也会比我们前面写的东西快得多，因为各种操作不是通过大量函数调用串联起来的。在这里，执行操作的所有代码都放在存储器里，形成一个指令块，真实处理器的程序计数器为我们完成所有的顺序处理。

与我们为虚拟机实现的那个generate相比，这个函数看上去要大得多。因为现在它编排的是真实的机器指令，而不是预先定义的函数指针。为了生成高效代码，还需要花很多功夫，去查看应该去掉的常量计算，以及做其他可能的优化。

在这里，对代码生成的讨论完全是走马观花式的，只是很初步地显示了一些真实编译系统所使用的技术，更多的东西完全被忽略了。我们还回避了许多由于现代 CPU的复杂性带来的问题。但是，即使如此，这些讨论仍然显示出一个程序可以如何去分析问题的描述，生成能够高效解决问题的特殊代码。你可以利用这些思想写一个快如闪电的 grep程序，或去实现你自己发明的某个小语言，或设计和实现一个能完成特定计算的虚拟机，或者甚至是 (在不多的其他东西的帮助下)为某种有趣的语言写一个编译程序。

从正则表达式到C++ 语言之间有一段很长很长的路，但是，这两者都是为了解决问题而设计出的记法。有了正确记法的辅佐，许多问题的解决将变得容易得多。而设计和实现这些记法本身也是趣味无穷的。

练习9-18 如果运行时编译能把仅仅包含常数的表达式，例如max(3*3, 4/2)用对应的值取代，它将能产生更快的代码。一旦识别出这种表达式，它怎样才能计算出有关的值？

练习9-19 你怎么测试一个运行时的编译器？

补充阅读

由Brian Kernighan和Rob Pike合写的《Unix程序设计环境》 (The Unix Programming Environment, Prentice Hall, 1984)里对于计算的基于工具途径做了广泛讨论， Unix系统对这些有很好的支持。该书第 8章给出了一个简单程序设计语言的完整实现，从 yacc语法描述直到可执行代码。

Don Knuth的《TEX: 程序》(TEX: The Program, Addison-Wesley, 1986)描述了复杂文档的格式化问题，给出了一个完整的程序，大约有 13 000行具有“带文档程序设计”风格的Pascal代码。在这些程序的正文里结合进有关文档，可以用程序对文档做格式化，或者从中提取可编译代码。Chris Fraser和David Hanson在《一个目标可重定位的C编译器：设计和实现》(A Retargetable C Compiler: Design and Implementation, Addison-Wesley, 1995)里对ANSI C做了同样的事情。

Java虚拟机在Tim Lindholm和Frank Yellin的《Java虚拟机规范》 (The Java Virtual

China-pub.com 下载

Machine Specification，第2版，Addison-Wesley，1999)中描述。

有关Ken Thompson算法(它是最早的软件专利之一)的讨论出现在《正则表达式搜索算法》(Regular Expression Search Algorithm，Communications of The ACM，卷11，第6期，pp419~422，1968)。Jeffrey E. F. Friedl的《掌握正则表达式》(Mastering Regular Expression，O'Reilly，1997)包含了对这个问题的范围非常广泛的论述。

Rob Pike、Bart Locanthi和John Reiser的文章《Blit上点阵图形的硬件 /软件权衡》(Hardware/Software Tradeoffs for Bitmap Graphics on the Blit，Software-Practice and Experience，卷15，第2期，pp131~152，1985年1月)描述了一个针对二维图形操作的运行时编译器。

China-pub.com
下载

附录：规则汇编

我发现的每个真理都变成了一条规则，它们为我以后发现其他的东西服务。

—— René Descartes，《方法论》

本书有的章节里列出了一些规则或准则，作为有关讨论的总结，我们把它收集到一起，以方便读者查阅。请注意，这些规则各有各的出处，那里解释了它们的意义和适用性。

风格

全局变量用具有描述意义的名字，局部变量用短名字。
保持一致性。
函数采用动作性的名字。
要准确。
以缩行形式显示程序结构。
使用表达式的自然形式。
利用括号排除歧义。
分解复杂的表达式。
要清晰。
当心副作用。
使用一致的缩行和加括号风格。
为了一致性，使用习惯用法。
用else-if 处理多路选择。
避免使用函数宏。
给宏的体和参数都加上括号。
给神秘的数起个名字。
把数定义为常量，不要定义为宏。
使用字符形式的常量，不要用整数。
利用语言去计算对象的大小。
不要大谈明显的东西。
给函数和全局数据加注释。
不要注释不好的代码，应该重写。
不要与代码矛盾。
澄清情况，不要添乱。

界面

隐蔽实现的细节。

China-pub.com 下载

选择一小组正交的基本操作。
不要在用户背后搞小动作。
在各处都用同样方式做同样的事。
释放资源与分配资源应该在同一层次进行。
在低层检查错误，在高层处理。
只把异常用在异常的情况。

排错

寻找熟悉的模式。
检查最近的改动。
不要两次犯同样错误。
现在排除，而不是以后。
取得堆栈轨迹。
键入前仔细读一读。
把你的代码解释给别人。
把错误弄成可以重现的。
分而治之。
研究错误的计数特性。
显示输出，使搜索局部化。
写自检测代码。
写记录文件。
画一个图。
使用工具。
保留记录。

测试

测试代码的边界情况。
测试前条件和后条件。
使用断言。
做防御性程序设计。
检查错误的返回值。
以递增方式做测试。
首先测试最简单的部分。
弄清所期望的输出。
检验那些应当保持的特征。
比较相互独立的实现。
度量测试的覆盖面。
自动回归测试。
建立自包容测试。

性能

自动计时测量。
使用轮廓程序。
集中注意热点。
画一个图。
使用更好的算法或数据结构。
让编译程序做优化。
调整代码。
不要优化无关紧要的东西。
收集公共表达式。
用低代价操作代替高代价操作。
铺开或者删除代码。
缓存频繁使用的值。
写专用的存储分配程序。
对输入输出做缓冲。
特殊情况特殊处理。
预先算出某些值。
使用近似值。
在某个低级语言里重写代码。
使用尽可能小的数据类型以节约存储。
不存储容易重算的东西。

可移植性

盯紧标准。
在主流中做程序设计。
警惕语言的麻烦特性。
用多个编译系统试验。
使用标准库。
只使用到处都能用的特征。
避免条件编译。
把系统依赖性局限到独立文件里。
把系统依赖性隐藏在界面后面。
用正文做数据交换。
数据交换时用固定的字节序。
如果改变规范就应该改变名字。
维护现存程序与数据的相容性。
不要假定是ASCII。
不要假定是英语。

China-pub.com
下载

后　　记

如果人能从历史中学习，我们将能学到多少东西啊！但是激忿和党派蒙住了我们的双眼，而经验的光亮就像船的艉灯，只能在我们背后的波涛上留下一点余辉。

Samuel Taylor Coleridge，《回忆》

计算的世界每时每刻都在变化，步伐看起来是越来越快。程序员必须不断应付新的语言、新的工具和新的系统，它们总有一些与老东西不兼容的新特性。程序越来越大，界面越来越复杂，而任务的时限也越来越短。

但是，总有某些东西是不变的，总有一些稳定点，在这种地方从过去中学到的东西和洞察力，对于未来必定能有所帮助。本书的背景就是基于这些具有持久性的概念。

简单性和清晰性是第一位的、最重要的，因为几乎所有其他东西都只能跟着它们而来。做那种最简单的能解决问题的东西，选择那些应该是足够快的最简单的算法、能够满足需要的最简单的数据结构；用整齐清楚的代码把它们组合起来。除非性能测试的结果说明需要做更多的事，我们绝不要把事情复杂化。界面应该是整齐和简单的，至少是在发现无可辩驳的证据、说明把它弄得复杂一些有极大优越性之前。

普遍性常与简单性同在，它使我们可能一次就完全解决了问题，而不是对各种情况一个个地重复去做。普遍性常常也是达到可移植的正确途径：找一个一般性的，能够在所有系统上工作的解，而不是去扩大不同系统之间的差异。

进化接踵而来。想第一次就构造出一个绝好的程序通常是不可能的。发现正确解决方法的必要洞察力只能来自思考和经验的结合；纯粹的内省不可能造就出好系统，纯粹靠玩命干也不行。由用户得来的反馈在这个地方非常重要。通过循环的方式：原型、试验、用户反馈和进一步精化，常常是最有效的。我们自己构建的程序常常进化得不够；从别人那里买来的大程序变得太快，根本没经过必要的改进。

界面是程序设计战斗中的一个大战场，界面问题出现在许多不同的地方。程序库是最明显的例子，还有不同程序之间的、程序与用户之间的界面问题。对简单性和普遍性的需求在界面设计方面表现得特别强烈。我们应该使界面具有一致性，容易学习和使用，应该一丝不苟地追求这些东西。抽象是一种有效技术：首先设想一种完美的部件，或者库，或者程序，让界面尽可能地符合这个理想。应该把实现细节都隐蔽在边界的后面，这永远是最安全的方法。

自动化一直没有得到足够重视。让计算机做你需要的事，常常比自己直接用手做的效率高得多。我们已经看到了许多这方面的例子，在程序测试、排错和性能分析方面，特别是写代码方面。在这些地方，对于合适的问题，程序可以写出人很难写出来的程序。

记法也一直被人所忽视，实际上它远不仅仅是程序员告诉计算机去做什么时采用的方式。记法提供了一种组织框架，在实现应用领域非常广泛的各种工具方面，对指导人们去构造那种写程序的程序方面都很有价值。我们对大型的通用程序设计语言都很习惯了，它们为我们

China-pub.com 下载

的大量程序设计工作服务。但有时也会发现面前的工作变得非常集中，已经理解得很清楚，写这种程序几乎完全是机械性的。这可能就预示着一个时机，说明应该建立一种能自然地表达有关工作的记法和一个实现它的语言。正则表达式是我们最喜爱的例子，实际中存在着无数的这种机会，在那里我们可以为对付特殊工作建立一个小语言。当然，这种语言绝不应该复杂到抵消掉其收益的程度。

作为程序员个人，我们很容易感到自己像是位于某种大机器上的一个小齿轮，必须使用那些强加给我们的语言、系统和工具，做那些我们必须完成的工作。但是，从长远的观点看，真正起作用的还是看我们在使用现有东西的情况下工作做得怎么样。通过应用本书里的某些思想，你将能够发现你的代码更容易用了，你的排错过程也不再那么痛苦了，你对自己的程序设计更加有自信心了。我们希望这本书能带给你一些东西，使你在自己的计算生涯中获得更多的成果和回报。