

第15学时 了解程序的运行性能

编写Perl程序，用于查找文件中的数据，或者与用户进行交互操作，这是非常有用的。但是，当程序运行结束时，将会发生什么情况呢？它的运行结果消失了，你没有得到任何东西来展示你的程序的性能，你会感到怅然若失，一无所获，就像什么也没有发生一样。

数据库能够解决这个问题。数据库可以用于存储数据，供以后使用。设计良好的数据库可以被任何种类的程序使用，以便进行数据的查询、报告和输入。若要设计数据库，必须认真考虑你想存储何种数据，以及如何对它进行存储。另外还要考虑如何访问数据，是每次由一个人访问，还是许多用户同时访问。

在本学时中，我们将要介绍两种方法，以便存储数据供以后检索。

在本学时中，你将要学习下面的内容：

- 创建DBM文件并将数据存储在该文件中。
- 将普通文本文件作为数据库来使用。
- 从文件中的随机位置读取数据和将数据存入文件中的随机位置。
- 为同时访问而锁定文件。

15.1 DBM文件

若要使你的程序能够以非常有条理的方式来存储数据，最简单的方法之一是使用 DBM文件。DBM文件是已经与一个Perl的哈希结构连接起来的文件。若要读取和写入 DBM文件，只需对一个哈希结构进行操作即可，就像从第7学时以来进行的操作那样。

若要将哈希结构与DBM文件连接起来，可以使用Perl函数dbmopen，如下所示：

```
dbmopen(hash, filename, mode)
```

dbmopen函数将hash与一个DBM文件连接起来。你提供的filename实际上在硬盘上创建两个不同的文件，即filename.pag和filename.dir。Perl使用这两个文件来存储哈希结构。这些文件不是文本文件，不应该对它们使用编辑器。另外，如果这两个文件中的一个空的，或者与文件中的数据量相比似乎非常大，请不必对此担心，这是正常的。

mode是指对Perl创建的两个DBM文件的访问许可权。如果是UNIX系统，可以使用一组明确的访问许可权，它们用于控制谁能够访问你的 DBM文件。例如，0666允许每个人拥有对DBM的读和写访问权。mode 0644允许你读和写这些文件，但其他人只能读这些文件。如果是Windows，只使用0666，因为你不必担心是否拥有对任何文件系统的访问许可权。

如果成功地将哈希结构键入DBM文件，那么dbmopen函数返回真，否则返回假。请看下面这个例子：

```
dbmopen(%hash, "dbmfile", 0644) || die "Cannot open DBM dbmfile: $!";
```

上面这个语句执行后，哈希结构 %hash就与称为dbmfile的DBM文件相连接。Perl在你的磁盘上创建一对文件，称为dbmfile.pag和dbmfile.dir，以便存放该哈希结构。如果你将一个值赋予该哈希结构，如下所示，Perl就用该信息更新DBM文件：

```
$hash{feline}="cat";  
$hash{canine}="dog";
```

如果要取出信息，Perl就从DBM文件中检索关键字和数据，如下所示：

```
print $hash{canine};
```

若要使哈希结构与DBM文件断开连接，请像下面这样使用带有哈希结构名字的 dbmclose 函数：

```
dbmclose(%hash);
```

当切断哈希结构与DBM文件的连接后，存放在它里面的项目 feline和canine仍将位于该DBM文件中，这是DBM文件的重要特点。在两次调用Perl程序之间，存放在与DBM文件相连接的哈希结构中的项目均保留不变。

通常对哈希结构执行的函数，也可以对与DBM文件连接的哈希结构执行。哈希结构的函数keys，values和delete按通常情况运行。可以清空哈希结构和DBM文件，方法是将哈希结构赋予一个空列表，比如 %hash=()。也可以对哈希结构进行初始化，方法是在用 dbmopen将哈希结构与DBM文件连接起来之后，将哈希结构赋予一个列表。

15.1.1 需要了解的重点

当你将哈希结构与DBM文件连接起来时，必须了解下列要点：

- 关键字和数据的长度现在是受限制的。通常情况下，哈希结构的关键字和数据长度是不受限制的，与DBM文件相连接的哈希结构拥有一个有限的单个关键字和一个数据，合起来的长度通常为1024字符左右，这是对DBM文件的一个限制。可以存储的关键字和值的总数没有变更，它只受文件系统的限制。
- 运行dbmopen以前的哈希结构中的值均被丢失，最好只使用新的哈希结构。请看下例：

```
%h=();  
$h{dromedary}="camel";  
dbmopen(%h, "database", 0644) || die "Cannot open: $!";  
print $h{dromedary}; # Likely will print nothing at all  
dbmclose(%h);
```

在上面这个代码段中，当执行dbmopen时，哈希结构%h中的值，即dromedare的关键字将会丢失。

- dbmclose函数执行后，与DBM文件相连接时哈希结构中的值将会消失：

```
dbmopen(%h, "database", 0644) || die "Cannot open: $!";  
$h{bovine}="cow";  
dbmclose(%h);  
print $h{bovine}; # Likely will print nothing
```

哈希结构与DBM文件连接时哈希结构中的值将保留在DBM文件中，此后，哈希结构本身将是空的。

15.1.2 遍历与DBM文件相连接的哈希结构

让我们观察一下尚未与DBM文件相连接的哈希结构。如果你编写一个Perl程序，用于将约会、电话号码和其他信息存放在哈希结构中。过一段时间后，该哈希结构就会变大。由于在两次运行你的Perl程序之间，哈希结构的值均被保留，因此它们决不会跑到别的地方去，除非故意将它们删除。

如果你的DBM文件（称为records）搜集了许多信息，那么下面这个代码段就会出现一些问题：

```
dbmopen(%recs, "records", 0644) || die "Cannot open records: $!";
foreach my $key (keys %recs) {
    print " $key = $recs{$key}\n";
}
dbmclose(%recs);
```

这个代码不存在什么问题。哈希结构首先与一个DBM文件相连接，然后使用keys %recs从哈希结构中取出关键字。用foreach my \$key对关键字列表进行迭代操作，然后输出每个关键字和值。

如果%recs中的关键字列表很大，那么语句keys %recs的执行需要花费一定的时间。Perl还有另一个函数，可以允许你对哈希结构进行迭代操作，每次取出一个关键字。这个函数称为each。each的句法如下：

```
($key, $val)=each(%hash);
```

each函数返回由两个元素组成的列表，这两个元素中一个是哈希结构的关键字，另一个是它的值。对each进行的每个连续的调用，可以返回哈希结构的下一个关键字值对。当关键字用完后，each就返回一个空表。对较大的哈希结构进行迭代操作时，更好的方法是使用下面的代码：

```
dbmopen(%recs, "records", 0644) || die "Cannot open records: $!";
while( ($key, $value)=each %recs) {
    print " $key = $value\n";
}
dbmclose(%recs);
```



你不一定必须将each函数用于与DBM文件相连接的哈希结构，可以将each用于任何哈希结构。

15.2 练习：一种自由格式备忘录板

既然你拥有一种将数据存放在磁盘上的简易方法，那么现在可以很好地运用这种方法。下面这个练习展示了一种自由格式的备忘录板。程序清单 15-2显示了该程序（memopad）的清单。它用于按关键字存储信息，并使你可以使用简单的查询方法来搜索和检索信息。程序清单15-1显示了使用memopad的会话示例。

若要查询memopad程序，只需键入一个问题的名字，后随一个问号。若要用新的方式进行编程，键入“X is Y”形式的短语，其中X是问题，Y是与该问题相关的信息。如果键入“like pattern？”（其中的pattern是在该问题中要搜索的一个正则表达式），你就可以搜索数据库，寻找相似性。符合该正则表达式的所有问题均被输出。若要退出该程序，请在提示符处键入quit。

程序清单 15-1 使用memopad的会话示例

```
Your question: perl?
I don't know about "perl"
Your question: perl is a programming language
Ok, I'll remember "perl" as "a programming language"
Your question: perl's homepage is at http://www.perl.org
```

```
Ok, I'll remember "perl's homepage" as "at http://www.perl.org"
Your question: perl?
perl is a programming language
Your question: like perl?
perl is like perl
perl's homepage is like perl
Your question: quit
```

每当该程序运行时，赋予 memopad 程序的所有信息均被存储，因为数据均存放在与一个 DBM 文件相连接的哈希结构中。

程序清单 15-2 memopad 的完整清单

```
1:  #!/usr/bin/perl -w
2:  use strict;
3:
4:  my(%answers, $subject, $info, $pattern);
5:
6:  dbmopen(%answers, "answers", 0666) || die "Cannot open answer DBM: $!";
7:  while(1) {
8:      print "Your question ('quit' to quit): ";
9:      chomp($_=lc(<STDIN>));
10:     last if (/^quit$/);
11:     if (/like\s+(.*)\?/) {
12:         $pattern=$1;
13:         while( ($subject,$info)=each(%answers) ) {
14:             if ($subject =~ /$pattern/) {
15:                 print "$subject is like $pattern\n";
16:             }
17:         }
18:     } elsif (/(.*)\?/) {
19:         $subject=$1;
20:         if ($answers{$subject}) {
21:             print "$subject is $answers{$subject}\n";
22:         } else {
23:             print qq[I don't know about "$subject"\n];
24:         }
25:     } elsif (/(.*)\sis\s(.*)/) {
26:         $subject=$1;
27:         $info=$2;
28:         $answers{$subject}=$info;
29:         print qq[Ok, I'll remember "$subject" as "$info"\n];
30:     } else {
31:         print "I'm sorry, I don't understand.\n";
32:     }
33: }
34: dbmclose(%answers);
```

第1~2行：这两行是Perl程序通常的起始行，#!行带有-w，这意味着警告特性是激活的。另外，use strict命令可以防止你犯不应该犯的错误。

第6行：使用dbmopen，哈希结构%answers与DBM文件answers相连接。在磁盘上创建两个文件，即answers.pag和answers.dir。

第7行：while(1)执行该永久循环。该循环中的某个位置是个last或exit命令，用于退出该循环。

第9行：这一行代码看上去容易使人混淆，因为它会立即产生若干情况。lc将它的标量参数改为小写字母。由于<STDIN>用在标量上下文中，因此从STDIN中读入一行输入，然后转

换成小写字母，其结果被赋予 \$_.chomp 用于删除结尾处的换行符。

第10行：如果输入行只包含单词 quit，则退出 while 循环。

第11行：如果输入行（现在位于 \$_ 中）与单词 like 相匹配，然后与某个文本相匹配，再与 ? 相匹配，那么文本被保存在 \$1 中，在匹配的模式中使用括号。

第12行：第11行的匹配模式中的字符串保存在 \$pattern 中。

第13~17行：逐个关键字对哈希结构 %answers 进行搜索，寻找与 \$pattern 中的字符串相匹配的关键字。当找到每个关键字时，便将它输出。

第18行：（这一行是第11行上开始的 if 语句的继续。）否则，如果输入行以问号结尾，那么问号前面的所有数据（不包含问号）将被存放在带括号的 \$1 中。

第19行：\$1 中的模式保存到 \$subject 中。

第20~24行：如果关键字 \$subject 是在哈希结构 %answers 中，则输出该关键字和相关的数据。否则，该程序发出应答消息 I don't know（我不知道）。

第25~27行：（这一行是第11行上开始的 if 语句的继续。）否则，如果输入行采用 X is Y 的形式，则第一部分（X）存入 \$subject 中，最后一部分存入 \$info 中。

第28行：\$info 中的信息作为 \$subject 存放在哈希结构 %answers 中。

第34行：DBM 文件与 %answers 断开连接。

15.3 将文本文件用作数据库

许多情况下，数据库是一种较小和较简单的结构，比如小型系统上的一个用户列表，小型网络上的本地主机，常用的 Web 站点的列表，或者个人地址文件等，它们都属于简单的数据库形式。而对于简单数据库来说，使用普通文本文件就可以了。但是首先必须考虑到这种数据库存在的某些不足。

将文本文件用作数据库，比使用复杂文件（如 DBM 文件）或大型数据库（如 Oracle 或 Sybase）有着一些明显的优点。下面列举其中的一些优点：

- 文本文件数据库可以移植。它们可以在各种不同的系统之间进行移动，不会遇到太大的麻烦。
- 文本文件数据库可以使用文本编辑器进行编辑，不必使用特殊工具就能在纸张上打印。
- 文本文件数据库开始时的创建工作很简单。
- 文本文件数据库可以输入到其他程序中，如电子表格、文字处理程序和其他数据库中，没有什么太大的麻烦。凡是可输入数据的程序，几乎都允许你输入文本。

但是将文本文件用作数据库也有它的不足。若要全面了解它的不足，你应该知道文本文件通常的结构。文本文件数据库的传统结构方式是：文本文件中的每一行都是一个记录，每一行中的各个列称为域。但是，对于你的系统来说，文本文件是个字符流。因此，下面的文本文件

Bob 555-1212

Maury 555-0912

Paul 555-0012

Ann-Marie 555-1190

实际上可以作为下面这个连续字符串来存储：

```
Bob[space]555-1212[newline]Maury[space]555-0912[newline]Paul[space]...
```

其中[space]代表一个空格字符，[newline]代表你的操作系统中的一个换行符，有时它是个回车符，有时是个回车符和换行符。每个记录和每个域的字符均封装在一个很长的字符流中，出色的列行显示格式是人能阅读的编辑器、打印机和 Perl 展示的数据格式。

知道这种结构的特点后，再让我们来看一看文本文件数据库的缺点：

- 不能将数据插入文本文件，只能部分或全部改写文本文件。除了将数据附加在文件的结尾处，如果将数据插入文件中的任何位置，就会拷贝文件中新插入的数据后面的所有数据。

新数据：Susan 555-6613 被插入 “Bob” 的后面

```
Bob[space]555-1212[newline]Maury[space]555-0912[newline]Paul[space]...
```

所有这些数据必须拷贝到：

```
Bob[space]555-1212[newline]Susan[space]555-6613[newline]Maury[space]...
```

在文件中以这种方式拷贝数据很容易出错，并且速度很慢。

- 相反的情况也一样。从文本文件的中间位置删除数据是很难的。被删除的这部分数据后面的全部数据都必须拷贝到间隔位置。如果要从原始文本数据库中删除 Maury，你必须这样操作：

通过删除下面的数据

```
Bob[space]555-1212[newline]Maury[space]555-0912[newline]Paul[space]...
```

所有数据必须拷贝到：

```
Bob[space]555-1212[newline] Paul[space]555-0012[newline]Ann-Marie[space]...
```

- 若要在文本文件数据库中查找某个记录，你必须顺序搜索该文件，通常从上向下进行搜索。在 DBM 文件中，搜索一个记录就像在哈希结构中查找这个记录一样容易，而文本文件则必须查看其每一行，以确定它是个正确的记录。这个过程很慢，随着数据库变得越来越大，搜索过程也越来越慢。

将数据插入文本文件或从文本文件中删除数据

尽管文本文件数据库存在上述缺点，但是它并非一无是处。如果你的文本文件数据库比较小，你可以将文本文件当作一个数组来处理，这样将数据插入数据库或者从数据库中删除数据将是非常容易的。例如，如果下面这个数据库

```
Bob 555-1212
Maury 555-0912
Paul 555-0012
Ann-Marie 555-1190
```

被保存到一个称为 phone.txt 的文件中，那么用一个较短的 Perl 程序就可以将该数据库读入一个数组，如下所示：

```
#!/usr/bin/perl -w
use strict;

sub readdata {
    open(PH, "phone.txt") || die "Cannot open phone.txt: $!";
```



```

my(@DATA)=<PH>;
chomp @DATA;
close(PH);
return(@DATA);
}

```

这里的readdata()函数读取文件phone.txt，并将数据放入@DATA中，它不带换行符，并返回该数组。如果增加另一个函数Writedata()，就可以像下面这样读写该数据库：

```

sub writedata {
    my(@DATA)=@_;    # Accept new contents
    open(PH, ">phone.txt") || die "Cannot open phone.txt: $!";
    foreach(@DATA) {
        print PH "$_\n";
    }
    close(PH);
}

```

这时，若要将记录插入该数据库，只要使用readdata()函数将数据读入一个数组。使用push、unshift或splice函数，将记录插入该数组，然后用writedata()函数像下面这样再次写出该数组：

```

@PHONELIST=readdata();    # Put all of the records in @PHONELIST
push(@PHONELIST, "April 555-1314");
writedata(@PHONELIST);    # Write them out again.

```

若要从文本文件数据库中删除文本，你可以在重新写入数组之前，对数组@PHONELIST使用Splice，pop或shift函数。也可以使用一个循环，人工编辑该数组，比如使用下面这个grep循环：

```

@PHONELIST=readdata();    # Read all records into @PHONELIST
# Remove everyone named "Ann" (or Annie, Annette, etc..)
@PHONELIST=grep(! /Ann/, @PHONELIST);
writedata(@PHONELIST);

```

在上面的代码段中，各个记录从readdata()拷贝到@PHONELIST。grep对@PHONELIST数组进行迭代操作，测试每个元素，以确定它是否与Ann不相匹配。凡是不匹配的元素均被再次赋予@PHONELIST。然后数组@PHONELIST被送回给writedata()，以便进行写入操作。

15.4 随机访问文件

如果你具有冒险精神，可以像前面提到的那样在文件中进行随机读写操作。下面各节将简单介绍几种工具，可以用来进行随机读写操作，不过我们不准详细介这些工具，因为你并不经常需要使用它们。

15.4.1 打开文件进行读写操作

到现在为止，你已经了解到打开文件的3种方法。文件可以被打开以便进行阅读、写入、以及将数据附加到它的结尾处。文件还可以打开以便同时进行阅读和写入操作。表15-1列出了打开文件的不同操作方式。

两项说明：

- 设定“附加”的方式是很麻烦的。在某些系统上，比如在UNIX上，写入文件的数据总是写到文件的结尾处，无论读取文件的指针位于何处。（后面我们很快还要说明这一点。）
- 决不应该使用+>。一旦文件打开，它的内容就会被删除。

表15-1 打开文件的不同方式

open命令	读	写	附加	如果语句不存在是否创建	是否截断现有数据
open (F , "<file ") 或者open(F: file ")	是	否	否	否	否
open(F , ">file ")	否	是	否	是	是
open(F , ">>file ")	否	是	是	是	否
open(F , "+<file ")	是	是	否	否	否
open(F , "+>file ")	是	是	否	是	是
open(F , "+>>file ")	是	是	是	是	是

15.4.2 在读写文件中移动

当文件打开时，操作系统始终跟踪你在文件中所处的位置。这个指针称为读指针。例如，当文件初次打开以便进行阅读时，读指针位于文件的开始处，如下所示：

读指针



当你读完整个文件后，读指针位于文件的结尾处，如下所示：

读指针



若要将指针移到文件中的某个位置，你可以使用 seek函数。seek函数带有两个参数。第一个参数是个打开的文件句柄，第二个参数是文件中你想寻找到的位移。第二个参数是该位移处于什么位置。0是文件的开始处；1是文件中的当前位置；2是文件的结尾。下面是在文件中进行搜索的一些示例代码：

```
# open existing file for reading and writing
open(F, "+<file.txt") || die "file.txt error: $!";
seek(F, 0, 2);           # Seek to the end of the file
print F "On the end";    # Appended to the end of the file
seek(F, 0, 0);           # Seek back to the beginning of the file
print F "This is at the beginning";
```

tell函数返回文件中的当前读指针的位置。例如，当运行上面的代码段后，tell(F)便返回24，即“ This is at the beginning ”的长度，因为文件指针就位于该文本的后面。



这一节只是简单提到seek、tell和open等命令。关于这些命令的详细说明，请参见在线文档。seek、tell和open函数在perlfunc手册页中作了说明，可以在命令提示符处键入perldoc perlfunc，访问perlfunc手册页。另外，在perlopentut一节中，对open命令进行了更加详细的介绍，可以在命令提示符处键入perldoc perlopentut，查看该文档。

15.5 锁定文件


假设你编写了一个非常出色的Perl程序，并且全世界的人都想使用它。如果你使用UNIX

或Windows NT计算机，或者使用Windows 95或Windows 98计算机，那么可能有许多人同时运行你的程序。你也可以将程序放到Web服务器上，它运行得如此频繁，以致于你的程序的许多实例互相重叠了。

现在假定你的程序将一个数据库用于它的工作，例如使用前面刚刚介绍的文本文件数据库，不过下面介绍的情况适用于任何一种数据库。请看下面这个代码，它使用上一节介绍的一些函数：

```
chomp($newrecord=<STDIN>);      # Get a new record from the user
@PHONEL=readdata();              # Read data into @PHONEL
push(@PHONEL, $newrecord);       # Put the record into the array
writedata(@PHONEL);              # Write out the array
```

这个代码看上去没有任何问题。如果两个人几乎同时运行你的程序，并且试图添加不同的记录，这很可能带来一些问题，它存在相当多的错误。在下面这个插图中，这一组特定的Perl语句几乎是同时在同一个系统上由两个人来运行的（第二个人运行程序的时间比第一个人稍晚一些）。请认真观察。

时间	序号	第1个人	第2个人
	1	<code>\$newrecord = "David 555-1212";</code>	
	2	<code>@PHONEL = readdata ();</code>	<code>\$newrecord = "Joy 555-6611";</code>
	3	<code>push (@PHONEL, \$newrecord);</code>	<code>@PHONEL = readdata ();</code>
	4	<code>writedata (@PHONEL);</code>	<code>push (@PHONEL, \$newrecord);</code>
	5		<code>writedata (@PHONEL);</code>

从第1个人的角度来看，数据是在第2步上读取的，新记录（“David”）在第3步上被添加给@PHONEL，并在第4步上进行写入操作。

从第2个人的角度来看，数据是在第3步上读取的，新记录（“Joy”）在第4步上被添加给@PHONEL，并在第5步上进行写入操作。

下面是它存在的错误：第2个人在第3步上读取的数据并不包含记录“David”。它尚未由第1个人写入。因此第2个人将“Joy”添加给数组@PHONEL，它并不包含“David”。与此同时，第1个人将@PHONEL的拷贝写入数据库，该记录包含“David”。

当第2个人的程序实例最终运行到第五步时，它将改写第1个人写入的数据。该数据库最终包含记录“Joy”而不是“David”。这显然是个错误。



这个问题实际上比你上面看到的还要严重。上面介绍的情况过分简单化了。更加使人头痛的是，writedata()函数看上去是一次性打开和写入数据的，但是实际上并非如此。多重处理的操作系统实际上能够在写入数据的中间停止程序的运行，并暂时转入另一个程序的运行，然后过几个毫秒又恢复第一个程序的运行。这两个程序都能够同时将不同的数据写入同一个文件。这会使你的数据文件遭到破坏，或者被删除。

这种类型的问题有一个正式的名字，称为竞态条件。程序中的竞态条件很难发现，因为竞态条件的出现和消失取决于一个程序有多少个实例正在同时运行。与竞态条件相关联的错误往往并不明显。

让多个程序同时更新相同的数据是很困难的，不过使用称为锁的机制就能解决这个问题。文件锁可以用于防止一个程序的多个实例同时更改一个文件。

锁定文件会带来两个问题，不过最大的问题是不同的操作系统和不同的文件系统需要使用不同类型的锁定机制。下面两节将介绍如何锁定文件以防止出现上面所说的问题。

15.5.1 锁定UNIX和NT下的文件

若要锁定UNIX和Windows NT下的文件，可以使用Perl的flock函数。flock函数提供了一个“诱导式”锁定机制。这意味着你编写的程序如果需要访问文件，那么它就必须使用 flock，以确保没有其他人在同一时间将数据写入该文件。但是，其他程序如果想要修改文件，则仍然可以修改文件，这就是为什么它称为“诱导式”锁定而不是强制锁定。

你可能已经熟悉一种类型的诱导式锁定，即交通信号灯。这种信号灯是为了防止许多车辆同时进入十字路口的相同区域。但是，只有人人都按照信号灯的指示来行车，信号灯才能正常发挥作用。文件锁的情况也一样。可能同时访问一个文件的每个程序必须使用 flock函数防止撞车。诱导式锁定并不能防止其他进程访问数据，它只能防止其他进程被锁定。

flock函数带有两个参数，一个是文件句柄，另一个是锁的类型，请看下面的语句：

```
use Fcntl qw(:flock);

flock(FILEHANDLE, lock_type);
```

如果锁定成功，那么 flock函数返回真，否则返回假。有时，调用 flock会导致你的程序暂停运行，以等待其他锁被打开。下面将很快要说明这个问题。如果使用 use Fcntl qw(:flock)，那么你将使用符号名作为 lock_type，而不是使用比较难以记住的数字。

文件锁有两种类型，一种是公用锁，一种是专用锁。通常情况下，当想要读取文件时，可以使用公用锁，而当将数据写入文件时，可以使用专用锁。如果一个进程拥有对文件的专用锁，那么这是那里存在的唯一的锁，其他进程则根本不拥有锁。但是，只要不存在专用锁，那么许多进程可以同时拥有公用锁。这是因为只要没有人写入文件，那么许多进程就可以安全地同时读取文件。

下面是lock_type可以使用的一些值：

- lock_SH 这个值要求在文件上设置公用锁。如果另一个进程拥有该文件的专用锁，那么flock函数就会暂停运行，直到专用锁被清除，然后再取出该文件的公用锁。
- Lock_EX 这个值要求对已经打开而用于写入的文件设置一个专用锁。如果其他进程拥有一个锁（公用锁或专用锁均可），那么flock就暂停运行，直到这些锁被清除。
- Lock_UN 这个值用于释放一个锁。但是，很少需要这样做。你只要关闭文件，就可以写出所有未写的的数据并释放文件锁。如果释放仍然打开的文件上的锁，就会导致数据被破坏。



当你关闭文件或者当你的程序退出时，即使退出时存在一个错误，用 flock设置的锁也会被释放。

在试图读写的文件上加锁是很复杂的。由于打开文件句柄和锁定文件至少需要两个步骤的进程，因此设置文件锁就会带来一些问题，首先必须打开文件，然后才能给文件加锁。如

果用 `open (FH, ">filename")`, 然后用 `flock` 函数给文件加了锁, 那么在你获得该锁之前, 你已经修改了该文件 (用 `>` 对文件截尾了)。通过截尾你可能修改了该文件, 而其他进程则对该文件设置了锁。

若要解决这个问题, 就需要某种称为信标文件的东西。信标文件是个牺牲性文件, 它没有什么重要的内容, 凡是对该文件拥有锁的人, 均能处理该文件。

若要使用信标文件, 你只需要有一个可以用作信标的文件名和两个函数, 用于将信标文件锁定和解锁。如程序清单 15-3 所示。这不是完整的程序, 不过它可以作为其他程序的组成部分。

程序清单 15-3 通用锁函数

```
1: use Fcntl qw(:flock);
2: # Any file name will do for semaphore.
3: my $semaphore_file="/tmp/sample.sem";
4:
5: # Function to lock (waits indefinitely)
6: sub get_lock {
7:     open(SEM, ">$semaphore_file")
8:     || die "Cannot create semaphore: $!";
9:     flock(SEM, LOCK_EX) || die "Lock failed: $!";
10: }
11:
12: # Function to unlock
13: sub release_lock {
14:     close(SEM);
15: }
```

这些锁函数可以将你当前不想运行的任何代码括起来, 即使这些代码与读写文件毫无关系。例如, 下面这个代码段 (即使同时被若干个进程运行) 只允许每次有一个进程输出一条消息:

```
get_lock();      # waits for a lock.
print "Hello, World!\n";
release_lock();  # Let someone else print now...
```

上面代码中的 `get_lock()` 和 `release_lock()` 函数将在需要给文件加锁时用于锁定文件。



拿着文件锁又等待用户输入 (或者其他速度较慢的事件), 这不是个好主意。需要该锁的所有其他程序都会停止运行, 以等待该锁被释放。你应该获取你的锁, 运行你锁定的敏感代码, 然后释放文件锁。

15.5.2 在加锁情况下进行读写操作

下面我们介绍的是文本文件数据库的 `readdata()` 和 `writedata()` 函数与文件锁一道运行的情况。若要进行这项操作, 需要一个信标文件和上一节介绍的 `get_lock()` 和 `release_lock()` 子例程。

程序清单 15-4 的第一部分是上一节中的锁代码。

程序清单 15-4 在加锁情况下文本文件的输入 / 输出

```
1: #!/usr/bin/perl -w
2: use strict;
3: use Fcntl qw(:flock);
```

```
4:
5: my $semaphore_file="/tmp/list154.sem";
6:
7: # Function to lock (waits indefinitely)
8: sub get_lock {
9:     open(SEM, ">$semaphore_file")
10:    || die "Cannot create semaphore: $!";
11:    flock(SEM, LOCK_EX) || die "Lock failed: $!";
12: }
13:
14: # Function to unlock
15: sub release_lock {
16:     close(SEM);
17: }
18:
19: sub readdata {
20:     open(PH, "phone.txt") || die "Cannot open phone.txt $!";
21:     my(@DATA)=<PH>;
22:     chomp(@DATA);
23:     close(PH);
24:     return(@DATA);
25: }
26: sub writedata {
27:     my(@DATA)=@_;
28:     open(PH, ">phone.txt") || die "Cannot open phone.txt $!";
29:     foreach(@DATA) {
30:         print PH "$_\n";
31:     }
32:     close(PH); # Releases the lock, too
33: }
34: my @PHONEL;
35:
36: get_lock();
37: @PHONEL=readdata();
38: push(@PHONEL, "Calvin 555-1012");
39: writedata(@PHONEL);
40: release_lock();
```

程序清单 15-4中的大部分代码你已经见过。本学时的前面部分中介绍过 `get_lock()`、`release_lock()`、`readdata()`和`writedata()`等函数。

这个程序的关键部分从第 3 4 行开始。在这部分代码中，用 `get_lock()`函数设置了一个文件锁。这时用`readdata()`函数将文件读入`@PHONEL`，并对数据进行操作，然后用`writedata()`函数将数据重新写入同一个文件。当所有这些操作完成时，如果其他程序等待释放该锁的话，`release_lock()`函数将锁释放。

15.5.3 Windows 95和Windows 98下的加锁问题

情况表明，Windows 95和Windows 98不支持文件锁定。为何不支持呢？因为在这些操作系统下，每次只有一个程序能够打开文件进行写入操作，因此文件加锁就没有必要。如果在Windows 95或Windows 98系统上使用`flock`函数，就会看到下面这个出错消息：

```
flock() unimplemented on this platform at line...
```

不过，幸好这些操作系统通常每次只支持一个用户进行文件操作。



本书中的程序清单都涉及到使用前面介绍的 `get_lock()`和`release_lock()`函数进行文件锁定的问题。在 Windows 95或Windows 98下使用这些函数就会出错，因为在这两个操作系统下无法实现 `flock`函数。这些操作系统的程序清单中可以省略这些函数。程序清单中将配有相应的说明来提醒你。

15.5.4 在其他地方使用文件锁的问题

在某些情况下，你可能同时有多个程序在读写文件。由于某个原因，你无法使用 `flock`函数。即使在能够使用 `flock`函数的平台上，该函数也不是在所有情况下都适用的。例如，在 UNIX系统下，对网络文件系统（NFS）上的文件使用`flock`函数是不可靠的。也可能你使用的是UNIX服务器与Windows NT客户机相混合的操作环境，在这种环境中，UNIX通常支持`flock`函数，但是基本的文件系统都不支持 `flock`。

在Perl的常见问题（FAQ）列表的第五部分“文件与格式”中，你会看到如何不使用 `flock`而对文件进行锁定的说明。若要阅读该文档，请查看该文档中的 `perlfaq 5`这一节。

15.6 课时小结

在本学时中，我们介绍了在两次调用 Perl程序之间存储数据的几种方法。首先讲述了DBM文件以及如何使用DBM文件将哈希结构与你的硬盘连接起来，接着介绍了文本文件如何将它们用作简单的数据库，最后，为了防止同时访问文件带来的问题，介绍了如何锁定文件和保证数据安全的方法。

15.7 课外作业

15.7.1 专家答疑

问题：我能将第13学时中的数据结构存储在DBM文件或文本文件中吗？

解答：简单的回答是不行，或者说并不容易。而详细的回答是行，不过首先你必须将这种“结构”转换成代表数据的字符串和包含该数据的结构，然后必须将它用作与DBM相连接的哈希结构中的一个值。进行这项操作的一个模块是 `Data::Dumper`。

问题：如何锁定DBM文件？

解答：DBM文件可以使用前面介绍的信标锁定系统来锁定。你只需使用程序清单 15-3中的`get_lock()`和`release_lock()`函数，将它们放在DBM的`open`和`close`函数的前后：

```
get_lock();
dbmopen(%hash, "foo", 0644) || die "dbmopen: $!";
$hash{newkey}="Value";
dbmclose(%hash);
release_lock();
```

问题：我是否能够以某种方式查看 `flock`函数准备暂停但实际上并不让它暂停运行？

解答：是的，可以。一个值可以传递给 `flock`函数，使它不暂停运行（称为非锁定 `flock`）。若要查看`flock`是否暂停运行，请像下面这样在锁类型的后面加上 `|Lock_NB`：

```
use Fcntl qw(:flock);
# Attempt to get an exclusive lock, but don't wait for it.
if (not flock(LF, LOCK_EX|LOCK_NB)) {
    print "Could not get the lock: $!";
}
```

你甚至可以等待一会儿，看看是否给文件加了锁，如果你最终没有给文件加上锁，就会输出一条消息：

```
use Fcntl qw(:flock);
$lock_attempts=3;
while (not flock(LF, LOCK_EX|LOCK_NB)) {
    sleep 5; # Wait 5 seconds
    $lock_attempts--;
    die "Could not get lock!" if (not $attempts);
}
```

15.7.2 思考题

- 1) 与DBM文件相连接的哈希结构中的关键字能够存储长度不限的关键字，是吗？
 - a. 是
 - b. 否
- 2) 为什么将数据插入普通文件非常难？
 - a. 周围的数据必须移动以便放置插入的数据。
 - b. 普通文件不能打开后同时进行读和写操作。
 - c. 文件被编辑时，必须锁定。
- 3) 关于锁定文件的说明是在FAQ的哪一节中介绍的？

15.7.3 解答

- 1) 答案是b。按照默认设置，DBM文件中关键字与值的合计长度是1024字符。
- 2) 答案是a。数据在文件中不能随意“向上”和“向下”移动，因此移动周围的数据非常困难。如果选择c也是可以的，不过只有在多个程序同时使用文件时才能成立。
- 3) FAQ的第5节“文件与格式”。

15.7.4 实习

- 编写一段小程序，用来更新文件中的计数器，使每次运行程序时计数器递增 1。当程序的多个实例同时运行时，记住要使用文件锁定特性。