第11学时　系统之间的互操作性

到现在为止，我们介绍的所有Perl特性基本上都属于独立的特性。如果你想完成某项操作，那么你必须自己亲自执行这项操作，比如给数据排序，创建目录列表，插入配置信息等。问题是它的工作量很大，你必须重复进行可以在其他地方完成的工作。

关于Perl，现在有一种非常流行的说法，那就是它是一种非常出色的“胶水”语言。它的意思是说，Perl能够使用操作系统作为组件来安装的其他程序，然后将这些程序组合起来，形成一个更大的程序。它能够启动操作系统的实用程序，用它们来搜集信息，与你进行通信，然后将它们关闭。

Perl能够将这些较小的实用程序“胶合”在一起，形成一个大得多而且更加有用的实用程序。这种能力的好处是使你能够迅速编写在其他情况下需要花费很长时间来编写的代码，并且能够对代码进行调试。你应该使用对你有用的任何手段，迅速而准确地编写代码。将系统的实用程序胶合在一起，可使之具备很大的优点。

在本学时中，你将要学习：

- system()函数。
- 捕获输出。
- 代码的移植性。

本学时中的大部分代码例子都有两个版本，一个用于Windows和DOS系统，另一个用于UNIX系统。如果只有一个代码例子，那么你会在课文中找到关于如何修改该代码，以适合另一种系统的需要，这种修改通常是少量的更改。

11.1　system()函数

若要运行非Perl的命令，最简单的方法是使用system()函数。system()函数能够暂停Perl程序的运行，然后运行外部命令，接着再运行你的Perl程序。system函数的句法如下：

```
system command;
```

该语句中的command是你要运行的命令，如果一切正常，系统的返回值是0，如果出现问题，则返回非零值。请注意，这个结果与True和False这两个Perl标准返回值相反。

下面是在UNIX系统上运行system函数的一个例子：

```
system("ls -lF");         # Print a file listing
# print system's documentation
if ( system("perldoc -f system") ) {
    print "Your documentation isn't installed correctly!\n";
}
```

下面这个例子显示在DOS／Windows下运行system的情况：

```
system("dir /w");         # Print a file listing
# print system's documentation
```

```
if ( system("perldoc -f system") ) {
    print "Your documentation isn't installed correctly!\n";
}
```

总的来说，system函数在上述两种系统结构下的运行情况是相同的。应该记住的是，两种操作系统下运行的命令基本上是不同的。若要在 DOS下获得一个文件列表，可以使用 dir命令，而在UNIX下获得文件列表，要使用 ls命令。在非常少见的情况下，比如在 perldoc中，UNIX和DOS系统下的命令是相同的。

当system函数运行外部命令时，该命令的输出在屏幕上显示的情况与 Perl程序输出的情况是相同的。如果该外部命令需要输入数据，那么来自终端的输入与你的 Perl程序从终端读入的输入是相同的。system函数运行的命令继承了 STDIN和STDOUT文件描述符，因此，外部命令执行输入／输出的位置与你的 Perl程序执行输入／输出的位置是完全相同的。通过 system函数来调用完全交互式的程序是可能的。

请看下面这个在UNIX下运行的代码例子：

```
$file="myfile.txt";
system("vi $file");
```

现在再看在Windows／DOS下运行的代码例子：

```
$file="myfile.txt";
system("edit $file");
```

上面的每个例子都对 myfile.txt运行一个编辑器， UNIX的编辑器是vi，DOS的编辑器是edit。当然编辑器是全屏幕运行的，所有的标准编辑器命令均能运行。当编辑器退出时，控制权返回给Perl。

可以使用system函数运行任何程序，不只是控制台方式的程序（文本程序）。在UNIX下，下面这个例子运行一个图形时钟：

```
system("xclock -update 1");
```

在DOS／Windows下，下面这个例子用于启动一个图形文件编辑器：

```
system("notpad.exe myfile.txt");
```

基本的命令解释程序

system函数（以及本学时中介绍的大多数函数）允许你使用命令提示符向你提供的命令解释程序的特性。之所以能够这样做，是因为 Perl的system命令能够调用一个 shell（在UNIX上是/bin/sh，在DOS／Windows中是command.exe），然后将你的system命令赋予该shell。这样，你就能够在 UNIX（&）下，执行重定向（>）、管道传输（|）和后台操作等任务，并且能够使用命令解释程序提供的其他特性。

例如，若要运行一个外部命令，并且捕获它在文件中的输出，可以使用下面这个命令：

```
system("perldoc perlfaq5 > faqfile.txt");
```

上面这个命令用于运行 perldoc的perlfaq5，并且捕获它在文件中的输出，称为 faqfile.txt。这个特定语句在DOS和UNIX中均能运行。

有些特性，比如管道传输和后台操作，在 UNIX操作系统下也能按照你的期望来运行，如你在下面看到的那样：

```
# Sort the file whose name is in $f and print it
system("sort $f | lp");  # Some systems use "lpr"
```

```
# Run "xterm" and immediately return
system("xterm &");
```

在最后一个代码举例中，xterm这个程序启动运行，但是&将使UNIX的shell启动“后台”的进程。这意味着虽然该进程继续运行，但是system函数已经完成运行，并将控制权返回给Perl。Perl将不等待xterm完成运行。

在UNIX下，Perl总是将／bin/sh或类似的命令用于system函数、管道和反引号（后面将介绍）。不管你的个人shell被配置成什么，它都是这样使用的。这种用法提供了在各个UNIX系统之间的某种程序的可移植性。

本学时中使用system函数、管道和反引号的例子放到Macintosh系统上可能无法运行。详细的说明请参见MacPerl文档中的“Macintosh的特殊特性”这一节。

11.2 捕获输出

system函数有一个很小的不足，它没有提供特别好的方法，来捕获命令的输出，并将它送往Perl进行分析。如果要以迂回方式进行这项操作，你可以使用下面这个代码：

```
# 'ls' and 'dir' used for example only. opendir/readdir
# would be more efficient in most cases.
system("dir > outfile");    # Use "ls" instead of "dir" for Unix
open(OF, "outfile") || die "Cannot open output: $!";
@data=<OF>;
close(OF);
```

在上面这个代码段中，system运行的命令让它的输出转到一个称为outfile的文件中。然后该文件被打开并读入一个数组。这时数据@data包含了dir命令的输入。

这个方法很麻烦，不是一个十分聪明的办法。Perl有另外一个方法处理这个问题，即反引号。用反引号（``）括起来的任何命令均由Perl作为外部命令来运行，就像通过system运行的一样，其输出被捕获，并且作为反引号的返回值返回。请看下面这个使用反引号的代码例子：

```
$directory=`dir`;    # Unix users, use ls instead of dir
```

在上面这个代码段中，运行的是dir命令，其输出在$directory中捕获。

在反引号中，可以看到所有标准的shell处理方式：>负责重定向，|负责管道传输。在UNIX下，&负责启动后台任务。不过请记住，在后台运行的命令，或者用>重定向其输出的命令，均没有输出可以捕获。

在标量上下文中，反引号返回的命令输出是单个字符串。如果命令的输出包含许多行文本，那么字符串中出现的所有文本行均用记录分隔符分开。在列表上下文中，命令的输出被赋予该列表，列表的每一行结尾均有记录分隔符。

现在请看下面这个代码：

```
@dir=`dir`;  # Use 'ls' for Unix users
foreach(@dir) {
    # Process each line individually.
}
```

在上面这个代码段中，@dir中的输出在foreach循环中处理，每次处理一行。

Perl还有另一种方法能够起到反引号的作用，你可以使用qx{}表示法。要执行的命令放入花括号（{}）中，如下例所示：

```
$perldoc=qx{perldoc perl};
```

通过使用花括号，当反引号作为命令的组成部分出现时，可以不必在反引号的前面加上反斜杠，如下所示：

```
$complex=`sort \'grep -l 'conf' *\'`;  # Somewhat messy
```

也可以将上面的代码段改写为下面的形式：

```
$complex=qx{ sort `grep -l 'conf' *` }; # Little easier.
```

任何字符均可用来取代{}，成对的字符如< >，（）和[]等均可以使用。

避免shell中的概念混乱

Perl与命令解释程序之间的界线有时会变得比较模糊，请看下面的两个例子：

在UNIX系统中：

```
$myhome=`ls $HOME`;
```

在DOS和Windows系统中：

```
$windows=`dir %windir%`
```

在第一个例子中，$HOME究竟是Perl变量$HOME，还是shell的环境变量$HOME呢？在DOS例子中，%windir%究竟是command.com变量windir，还是Perl的哈希结构%windir后随符号%呢？

问题是$HOME被Perl进行了内插替换，也就是说，$HOME是Perl的标量变量$HOME，它可能不是你想要的。在反引号中，变量展开为它们各自的值，就像使用双引号（“ ”）一样。可能的变量名%windir并不在双引号中展开，只有标量和数组名进行了内插替换。

为了避免这种混乱，可以在不想让 Perl进行内插替换的变量前面加上一个反斜杠，如下面这两个例子所示：

```
$myhome=`ls \$HOME`;  # The \ hides $HOME
```

或者

```
$windows=`dir %windir%`
```

现在，$HOME是UNIX shell的HOME变量，%winddir%是command.com的winddir变量。

另一种方法是用qx{}表示法来代替反引号，并用单引号来限定 qx，如下面这些例子所示：

```
$myhome=qx' ls $HOME ';
```

或者

```
$windows=qx' dir %winddir% ';
```

Perl将qx” 序列视为特殊标号，并且不展开它里面的 Perl变量，因此，你可以使用反引号，并且不必用反斜杠对命令中出现的其他反引号进行转义。

11.3 管道

UNIX与DOS／Windows中的管道可用于将不同进程连接在一起，使一个进程的输出成为下一个进程序的输入。请看下面的一组命令，这些命令差不多可以在 UNIX（如果将dir改为ls）或DOS中运行：

```
dir > outfile
sort outfile > newfile
more newfile
```

dir的输出在outfile中集中，然后使用sort对outfile排序，同时sort的输出被存放在newfile中。接着more命令显示newfile的内容，每次显示一屏内容。

管道允许你执行与上面相同的命令序列，但是不带 outfile和newfile，如下所示：

```
dir | sort | more
```

dir的输出被赋予sort，然后sort对数据进行排序。 sort的输出被赋予more， 每次显示1页。它不需要重定向（>）或临时文件。

这种命令行称为管道命令行，两个命令之间的竖线称为管道。 UNIX主要依赖管道来连接它的较小的实用程序。 DOS和Windows均支持管道，但与管道一起运行的命令行实用程序要少得多。

Perl程序可以用不同方式纳入管道。首先，你可以编写一个 Perl程序，以便接收输入，对输入进行转换，然后插入管道，如下例所示：

在上面这个管道中， Totaler可以是你编写的Perl程序，以便输出目录列表的合计，也可能

```
dir /B | sort | perl Totaler | more
```

是某些统计数字，同时输出目录列表本身。如果你使用 UNIX系统，请将dir /B改为ls -1，管道就会按照你的要求来运行。程序清单 11-1包含Totaler程序。

程序清单11-1 Totaler的完整清单

```
1:  #!/usr/bin/perl
2:
3:  use strict;
4:  my($dirs,$sizes,$total);
5:
6:  while(<STDIN>) {
7:          chomp;
8:          $total++;
9:          if (-d $_) {
10:                 $dirs++;
11:                 print "$_\n";
12:                 next;
13:         }
14:         $sizes+=(stat($_))[7];
15:         print "$_\n";
16: }
17: print "$total files, $dirs directories\n";
18: print "Average file size: ", $sizes/($total-$dirs), "\n";
```

第6行：输入的每一行均从STDIN读入，再赋予$_。在管道上，一个程序的 STDIN被连接到前一个程序的STDOUT。因此，在上面的例子中， STDIN由dir ／B馈入信息。

第9~13行：如果遇到一个目录，在 $dirs中对它的号码单独相加，目录名被输出，循环再次启动运行。

第14~15行：否则，在$sizes中对文件的大小进行累加，文件名被输出。

第17~18行：输出文件的平均大小，同时输出文件和目录的总数。

Perl参与管道运行的另一种方法是将管道视为既可以读取也可以写入的文件。这是使用Perl中的open函数来实现的，如下所示：

```
# Replace "dir /B" with "ls -1" for Unix
open(RHANDLE, "dir /B| sort |") || die "Cannot open pipe for reading: $!";
```

在上面这个代码段中， open函数打开一个管道，以便从 dir/B | sort中读取数据。Perl从该

管道读取数据的这一事实是通过右边的最后一个管道（ |）来指明的。当 open函数运行时，Perl启动执行dir /B | sort命令。当文件句柄RHANDLE被读取时， sort的输出被读入Perl程序。

现在请看下面这个例子：

```
open(WHANDLE, "| more") || die "Cannot open pipe for writing: $!";
```

这个open函数打开一个管道，以便将数据写入 more命令。左边的管道符号说明Perl正在将数据写入管道。输出到WHANDLE文件句柄的所有数据均被more缓存，并每次显示1页。编写这样的函数是每次显示你的一页程序的输出的好办法。

当你完成对已向程序打开的文件句柄（如 RHANDLE和WHANDLE）的操作时，应该正确地关闭句柄，这一点非常重要，因为由 open函数打开的程序必须正确地关闭，若要关闭文件句柄，可以使用 close函数来确保它的正确关闭。当完成句柄操作后，如果不能关闭文件句柄，那么即使你的Perl程序已经终止运行，你编写的程序仍会继续运行。

当关闭了管道上打开的文件句柄后， close函数会指明管道运行是否成功。因此，你应该像下面这样认真检查close的返回值：

```
close(WHANDLE) || warn "pipe to more failed: $!";
```

open函数也许无法告诉你管道是否已经成功地启动运行，其原因与 UNIX的设计有关。当 Perl创建管道并将它启动时，它不清楚管道是否真的能够工作。如果管道组装正确，并且启动运行了，那么它认为它将能够正确地终止运行。当管道中的最后一个程序完成运行时，它应该返回一个成功退出的状态。close函数能够读取该状态，以了解是否一切运行正常，否则，就会产生一个错误。

11.4 可移植性入门

可移植性，这是Perl擅长的特性之一。无论你的 Perl代码是在VMS计算机上运行，还是在UNIX、Macintosh或MS-DOS系统下运行，都具有很强的可移植性，使你编写的 Perl代码能够在Perl支持的任何结构上天衣无缝地运行。当需要与基本的操作系统打交道时，比如当进行文件输入／输出时， Perl将设法隐藏所有不必要的细节，使你的代码能够实实在在地运行。

Perl之所以具有如此强的可移植性，第 16学时将对其中的某些原因进行详细说明。

不过， Perl能够向你隐藏的信息是要受到一定限制的。

在本学时中，有些代码例子讲明“ 这适用于 Windows和DOS，而这适用于UNIX ”，有时又说两种系统都适用，具体情况要根据你使用的系统结构而定。使你自己的程序同时适用于Windows和UNIX，这意味着每个程序必须创建两个版本，一个用于 Windows，一个用于UNIX。当你的程序运行成功并且移植到一个更加特殊的操作系统，如 MacOS 9，那么创建程序的两个版本将会带来更多的问题。

许多情况下，你为一种操作系统结构（如 Windows NT）编写了一个程序，结果却发现它是在另一种结构（比如 UNIX）中运行。由于Perl能够在那么多的不同结构中运行，因此许多人认为在Windows NT下运行Perl程序与它在UNIX下运行是相同的。 Web服务器和其他应用程

序经常在不同的操作系统之间转移，因此使你的软件具备可移植性是个非常好的思路。

为每种操作系统创建一个程序的不同版本，使程序能够在任何情况下都能够运行，这是很费时间、很浪费和低效率的。遵照一些规则，你就能够创建到处都能运行的程序，至少可以设法使之到处都能运行，并且便于维护。

下面是编写“到处均可运行的”代码时遵循的一般原则：

- 始终使警告特性处于打开状态，并使用 use strict命令。这样，就可以确保你的代码能够用不同版本的Perl来运行，并且不会出现明显的错误。
- 始终都要检查来自系统请求的返回值，例如，应该使用 open || die，而决不要只使用open。检查返回值可以在将应用程序从一个服务器移到另一个服务器（而不只是在不同的操作系统之间移动）时帮助你发现错误。
- 输出表义性强的出错消息。
- 使用Perl的内置函数，执行你要用system函数或反引号（``）来执行的操作。
- 将依赖系统执行的操作（文件I/O，终端I/O、进程控制等）封装在函数中，检查以确保这些操作受当前操作系统的支持。

前面两个原则你已经熟悉。在本书中，所有的代码例子均已检查了关键函数的退出状态，而从第8学时起，所有较大的代码例子均展示了 use strict和警告消息。

第3个原则不能忽略，因为它是输出表义性很强的出错消息。在下面这些消息中，哪个最有帮助呢？

```
(no message, or wrong output)
Died at line 15.
Cannot open Foofile.txt: No such file or directory
Cannot open Foofile.txt: No such file or directory at myscript.pl line 24
```

显然，最后一条消息最有帮助。当你安装程序后，而且几个月（或几年）后出了问题，最后一条消息能够说明哪个程序运行失败了（ myscript.pl），它想要什么（Foofile.txt），它为何运行失败（没有这个文件…），它在何处运行失败了（第 24行）。这些信息可以帮助你迅速排除故障。花费一点儿时间写一条很好的、表义性强的出错消息，总是值得的。

第4个原则意味着只要可能，你就应该使用 Perl。若要检索目录列表，最好只使用 $dir=`dir`；但是，如果该程序移植到非 Windows系统中，那么它的运行就会失败。一种好的解决方案是使用 <*>，而更好的解决方案则是只要可能，就使用 opendir/readdir/closedir函数。无论你的程序移植到什么地方，这些解决方案都能运行。

举例说明两个操作系统之间的差别

编写“到处均可运行的”代码时遵循的最后两个原则，即“将依赖系统的操作封装在函数中”以及“检查程序运行所在计算机上的操作系统”，还需要作一点补充说明和演示。

当你坐在计算机面前键入 Perl程序时，应该记住，总有一天，你的 Perl程序有可能在另一台计算机上运行。你可能建立下一个 Amazon.com web站点，它可能从你的PC移植到一台大型 Windows NT服务器上，再移植到一群 Sun 公司 1000 UNIX服务器上；或者你可能只有一些个人的CGI程序，并且更换了Web提供商，结果发现新提供商配备的是一种不同的服务器。这些情况经常会出现，必须加以考虑。

那么，你的程序究竟如何知道 Windows NT与UNIX之间的差别呢？这个问题很简单。 Perl

有一个特殊变量$^O，即美元符号，插入记号^和大写字母O，这个变量包含了程序运行时所在的操作系统结构。例如，在Windows和DOS下，它包含字符串MSWin32。在UNIX下，它包含你运行的UNIX类型，如linux，aix和solaris等。

下面是依赖你运行的操作系统而执行的一些操作任务：

- 查找关于系统配置的各种信息。
- 对磁盘和目录结构进行操作。
- 使用系统服务程序（email）。

在下面这个例子中，可以查看一个代码段，以便找出系统上的可用磁盘空间。如果有人想要将一个文件上载到一个服务器并且想要确定该文件是否适合在该服务器上运行，那么下面这个例子是有用的。若要在 Windows系统的当前目录中找出可用的磁盘空间，可以使用类似下面的代码段：

```
# The last line of 'dir' reports something like:
#       10 dir(s)     67,502,080 bytes free
# Or on Win98, "MB" instead of "bytes"
my(@dir,$free);
@dir=`dir`;
$free=$dir[$#dir];
$free=~s/.*([\d,]+) \w+ free/$1/;
$free=~s/,//g;
```

上面这个代码段取出 @dir中的目录列表的最后一行，使用正则表达式删除不包括大小（即bytes free前面的数字和逗号）在内的其他信息。最后，逗号被删除，这样， $free只包含原始的空闲磁盘空间。这种方法非常适用于 Windows系统。对于UNIX系统，尤其是Linux，则可使用下面这个代码段：

```
# Last lines of df -k . reports something like this:
# Filesystem         1024-blocks  Used    Available Capacity Mounted on
# /dev/hda1             938485        709863 180139        80%       /
# And the 4th field is the number of free 1024K disk blocks
# This format may be particular to Linux.
my(@dir, $free);
@dir=`df -k .`;
$free=(split(/\s+/, $dir[$#dir]))[3];
$free=$free*1024;
```

请注意上面这个代码段与前面那个代码段之间的差别。在 Windows下查找磁盘空间的实用程序是dir，在UNIX下，它是df -k.。df -k输出的最后一行被分割成若干部分，第4个域被放入$free中。df的输出随着UNIX系统的不同而各有差异，通常情况下，报告的域的数目是不一样的，也可能它们使用不同的顺序。你的Perl代码只需要选择一个不同的域，就很容易得到调整。

因此，现在有两种完全不同的例程可以用来确定空闲的磁盘空间。可以将它们组合起来，并让适当的例程在每个操作系统上运行，如下所示：

```
if ( $^O eq 'MSWin32') {
    # The last line of 'dir' reports something like:
    #       10 dir(s)     67,502,080 bytes free
    my(@dir,$free);
    @dir=`dir`;
    $free=$dir[$#dir];
    $free=~s/.*([\d,]+) \w+ free/$1/;
    $free=~s/,//g;
} elsif ($^O eq 'linux' ) {
    # Last line of df -k . reports something like this:
    # /dev/hda1       938485 709863 180139    80%  /
    # And the 4th field is the number of free 1024K disk blocks
```

```
    my(@dir, $free);
    @dir=`df -k .`;
    $free=(split(/\s+/, $dir[$#dir]))[3];
    $free=$free*1024;
} else {
    warn "Cannot determine free space on this machine\n";
}
```

这个示例程序现在已经扩展为同时包括DOS／Windows版本和Linux版本。如果它在任何其他操作系统的机器上运行，就会输出一条警告消息。该例程几乎已经运行结束。现在你需要做的事情是在函数中将该例程隔离出来，使得需要的变量可以声明为专用变量，并且最后的结果可以裁剪，粘贴到任何程序中，并在需要时随时使用。产生的结果代码是：

```
# Computes free space in current directory
sub freespace {
    my(@dir, $free);
    if ( $^O eq 'MSWin32') {
        # The last line of 'dir' reports something like:
        #       10 dir(s)     67,502,080 bytes free
        @dir=`dir`;
        $free=$dir[$#dir];
        $free=~s/.*([\d,]+) bytes free/$1/;
        $free=~s/,//g;
    } elsif ($^O eq 'linux' ) {
        # Last line of df -k . reports something like this:
        # /dev/hda1       938485 709863 180139   80%  /
        # And the 4th field is the number of free 1024K disk blocks
        @dir=`df -k .`;
        $free=(split(/\s+/, $dir[$#dir]))[3];
        $free=$free*1024;
    } else {
        $free=0; # A default value
        warn "Cannot determine free space on this machine\n";
    }
    return $free;
}
```

这时，每当你的程序需要了解空闲磁盘空间量时，只需要调用 freespace()函数即可，答案将被返回。如果想在没有列出的另一个操作系统上运行该函数，就会输出一条出错消息。但是，将另一个UNIX型OS添加给该函数，将是很难的，你只能添加另一个子句。

11.5 课时小结

本学时你学习了如何使用系统的实用程序来进行各项操作。使用 system函数，可以运行一个系统实用程序（或者一个管道程序）。反引号（``）能够运行一个系统实用程序，然后捕获它的输出。接着，捕获的输出存放在一个变量中，供Perl使用。open函数不仅可以打开文件，而且可以打开程序。该程序可以用 print函数写入尖括号运算符(<>)，或者从尖括号运算符中读取数据。最后，我们介绍了将这些实用程序用于许多不同类型的操作系统的方法，但是不必为每种系统编写不同的程序。

11.6 课外作业

11.6.1 专家答疑

问题： 如何打开既可以到达一个命令也可以来自一个命令的管道？例如：open (P, “| cmd |”)

这个管道似乎并不能运行。为什么？

解答：这项操作实际上相当复杂，因为从同一个进程读取和写入数据可能导致程序“死锁”。这时，你的程序期望cmd输出某些信息，并且等待带有 <P>的数据。同时，因为存在某些混乱状态，cmd实际上等待你的程序输出带有print P“...”的某些数据。事实上，如果你激活了警告特性，Perl将向你报一条消息：“Can't do bidirectional pipe（无法执行双向管道操作）”。

如果你为这种问题做好了准备，IPC::Open2模块将允许你打开一个双向开关。这些模块将在第14学时中介绍。

问题：代码$a=system（“cmd”）无法按我期望的那样捕获$a中的cmd的输出。为什么？

解答：你将system与反引号（``）的作用混淆了。system函数不能捕获cmd的输出，你需要的代码是$a=`cmd`。

问题：当我在UNIX下运行带有反引号（``）的外部程序时，没有捕获出错消息，原因何在？

解答：由于所有UNIX程序，包括Perl程序，都包含两个输出文件描述符，即STDOUT和STDERR。文件描述符STDOUT用于捕获正常程序输出。文件描述符STDERR用于捕获出错消息。反引号和带有管道的open函数只能捕获STDOUT。简单的答案是使用shell将STDOUT重定向到STDERR，然后运行下面这个命令：

```
$a=`cmd 2>&1`;   # run "cmd", capturing output and errors
```

Perl的FAQ（常见问题）详细介绍了这个命令和捕获命令的错误时使用的其他方法。键入perldoc perlfaq8，便可打开FAQ的有关内容。

11.6.2 思考题

1) 若要使你的程序生成的数据每次显示1页，应使用：

a. `perl myprog.pl | more`

b. `open(M, "| more") || die; print M "data...data...data....\n";`

c. `open(M, ">more") || die; print M "data..data...data...\n";`

2) $foo的哪个值用于$r=`dir $foo`这个语句？

a. $foo的shell的值。

b. Perl的$foo值被替换，然后运行dir。

3) 下列操作中的哪个操作随着操作系统的不同而变更？

a. 找出空闲磁盘空间的容量

b. 获取目录列表

c. 删除目录

11.6.3 解答

1) 答案是a或b。如果选择a，myprog.pl的所有输出均送入more。如果选择b，那么写入文件描述符M的任何数据均送入more，以便进行分页。

2) 答案是b。若要防止$foo被Perl展开，你可以使用qx`dix $foo`。

3) 答案只能是a。b的操作可以用glob，<*>或opendir和readdir来完成。C可以用rmdir

完成。

11.6.4 实习

- 使用第8学时中的统计函数，显示程序清单 11-1中关于文件大小的更多的统计数据。
- 如果你拥有UNIX系统，将你的UNIX的特定样式添加给freespace()函数。使用Linux作为练习的开始。