第三十五章 字符串和数字

Computer programs are all about working with data. In past chapters, we have focused on processing data at the file level. However, many programming problems need to be solved using smaller units of data such as strings and numbers.

所有的计算机程序都是用来和数据打交道的。在过去的章节中，我们专注于处理文件级别的数据。 然而，许多程序问题需要使用更小的数据单位来解决，比方说字符串和数字。

In this chapter, we will look at several shell features that are used to manipulate strings and numbers. The shell provides a variety of parameter expansions that perform string operations. In addition to arithmetic expansion (which we touched upon in Chapter 7), there is a common command line program called bc, which performs higher level math.

在这一章中，我们将查看几个用来操作字符串和数字的 shell 功能。shell 提供了各种执行字符串操作的参数展开功能。 除了算术展开（在第七章中接触过），还有一个常见的命令行程序叫做 bc，能执行更高级别的数学运算。

**参数展开**

Though parameter expansion came up in Chapter 7, we did not cover it in detail because most parameter expansions are used in scripts rather than on the command line. We have already worked with some forms of parameter expansion; for example, shell variables. The shell provides many more.

尽管参数展开在第七章中出现过，但我们并没有详尽地介绍它，因为大多数的参数展开会用在脚本中，而不是命令行中。 我们已经使用了一些形式的参数展开；例如，shell 变量。shell 提供了更多方式。

**基本参数**

The simplest form of parameter expansion is reflected in the ordinary use of variables.

最简单的参数展开形式反映在平常使用的变量上。

For example:

例如：

**$a**

when expanded, becomes whatever the variable a contains. Simple parameters may also be surrounded by braces:

当 $a 展开后，会变成变量 a 所包含的值。简单参数也可能用花括号引起来：

**${a}**

This has no effect on the expansion, but is required if the variable is adjacent to other text, which may confuse the shell. In this example, we attempt to create a filename by ap- pending the string “\_file” to the contents of the variable a.

虽然这对展开没有影响，但若该变量 a 与其它的文本相邻，可能会把 shell 搞糊涂了。在这个例子中，我们试图 创建一个文件名，通过把字符串 “\_file” 附加到变量 a 的值的后面。

[me@linuxbox ~]$ a="foo"

[me@linuxbox ~]$ echo "$a\_file"

If we perform this sequence, the result will be nothing, because the shell will try to ex- pand a variable named a\_file rather than a. This problem can be solved by adding braces:

如果我们执行这个序列，没有任何输出结果，因为 shell 会试着展开一个称为 a\_file 的变量，而不是 a。通过 添加花括号可以解决这个问题：

[me@linuxbox ~]$ echo "${a}\_file"

foo\_file

We have also seen that positional parameters greater than 9 can be accessed by surround- ing the number in braces. For example, to access the eleventh positional parameter, we can do this:

我们已经知道通过把数字包裹在花括号中，可以访问大于9的位置参数。例如，访问第十一个位置参数，我们可以这样做：

**${11}**

**管理空变量的展开**

Several parameter expansions deal with nonexistent and empty variables. These expan- sions are handy for handling missing positional parameters and assigning default values to parameters.

几种用来处理不存在和空变量的参数展开形式。这些展开形式对于解决丢失的位置参数和给参数指定默认值的情况很方便。

**${parameter:-word}**

If parameter is unset (i.e., does not exist) or is empty, this expansion results in the value of word. If parameter is not empty, the expansion results in the value of parameter.

若 parameter 没有设置（例如，不存在）或者为空，展开结果是 word 的值。若 parameter 不为空，则展开结果是 parameter 的值。

[me@linuxbox ~]$ foo=

[me@linuxbox ~]$ echo ${foo:-"substitute value if unset"}

if unset

substitute value

[me@linuxbox ~]$ echo $foo

[me@linuxbox ~]$ foo=bar

[me@linuxbox ~]$ echo ${foo:-"substitute value if unset"}

bar

[me@linuxbox ~]$ echo $foo

bar

**${parameter:=word}**

If parameter is unset or empty, this expansion results in the value of word. In addition, the value of word is assigned to parameter. If parameter is not empty, the expansion re- sults in the value of parameter.

若 parameter 没有设置或为空，展开结果是 word 的值。另外，word 的值会赋值给 parameter。 若 parameter 不为空，展开结果是 parameter 的值。

[me@linuxbox ~]$ foo=

[me@linuxbox ~]$ echo ${foo:="default value if unset"}

default value if unset

[me@linuxbox ~]$ echo $foo

default value if unset

[me@linuxbox ~]$ foo=bar

[me@linuxbox ~]$ echo ${foo:="default value if unset"}

bar

[me@linuxbox ~]$ echo $foo

bar

Note: Positional and other special parameters cannot be assigned this way.

注意： 位置参数或其它的特殊参数不能以这种方式赋值。

**${parameter:?word}**

If parameter is unset or empty, this expansion causes the script to exit with an error, and the contents of word are sent to standard error. If parameter is not empty, the expansion results in the value of parameter.

若 parameter 没有设置或为空，这种展开导致脚本带有错误退出，并且 word 的内容会发送到标准错误。若 parameter 不为空， 展开结果是 parameter 的值。

[me@linuxbox ~]$ foo=

[me@linuxbox ~]$ echo ${foo:?"parameter is empty"}

bash: foo: parameter is empty

[me@linuxbox ~]$ echo $?

1

[me@linuxbox ~]$ foo=bar

[me@linuxbox ~]$ echo ${foo:?"parameter is empty"}

bar

[me@linuxbox ~]$ echo $?

0

**${parameter:+word}**

If parameter is unset or empty, the expansion results in nothing. If parameter is not empty, the value of word is substituted for parameter; however, the value of parameter is not changed.

若 parameter 没有设置或为空，展开结果为空。若 parameter 不为空， 展开结果是 word 的值会替换掉 parameter 的值；然而，parameter 的值不会改变。

[me@linuxbox ~]$ foo=

[me@linuxbox ~]$ echo ${foo:+"substitute value if set"}

[me@linuxbox ~]$ foo=bar

[me@linuxbox ~]$ echo ${foo:+"substitute value if set"}

substitute value if set

**返回变量名的参数展开**

The shell has the ability to return the names of variables. This is used in some rather exotic situations.

shell 具有返回变量名的能力。这会用在一些相当独特的情况下。

**${!prefix\*}**

**${!prefix@}**

This expansion returns the names of existing variables with names beginning with prefix. According to the bash documentation, both forms of the expansion perform identically. Here, we list all the variables in the environment with names that begin with BASH:

这种展开会返回以 prefix 开头的已有变量名。根据 bash 文档，这两种展开形式的执行结果相同。 这里，我们列出了所有以 BASH 开头的环境变量名：

[me@linuxbox ~]$ echo ${!BASH\*}

BASH BASH\_ARGC BASH\_ARGV BASH\_COMMAND BASH\_COMPLETION

BASH\_COMPLETION\_DIR BASH\_LINENO BASH\_SOURCE BASH\_SUBSHELL

BASH\_VERSINFO BASH\_VERSION

**字符串展开**

There is a large set of expansions that can be used to operate on strings. Many of these expansions are particularly well suited for operations on pathnames.

有大量的展开形式可用于操作字符串。其中许多展开形式尤其适用于路径名的展开。

**${#parameter}**

expands into the length of the string contained by parameter. Normally, parameter is a string; however, if parameter is either @ or \*, then the expansion results in the number of positional parameters.

展开成由 parameter 所包含的字符串的长度。通常，parameter 是一个字符串；然而，如果 parameter 是 @ 或者是 \* 的话， 则展开结果是位置参数的个数。

[me@linuxbox ~]$ foo="This string is long."

[me@linuxbox ~]$ echo "'$foo' is ${#foo} characters long."

'This string is long.' is 20 characters long.

**${parameter:offset}**

**${parameter:offset:length}**

These expansions are used to extract a portion of the string contained in parameter. The extraction begins at offset characters from the beginning of the string and continues until the end of the string, unless the length is specified.

这些展开用来从 parameter 所包含的字符串中提取一部分字符。提取的字符始于 第 offset 个字符（从字符串开头算起）直到字符串的末尾，除非指定提取的长度。

[me@linuxbox ~]$ foo="This string is long."

[me@linuxbox ~]$ echo ${foo:5}

string is long.

[me@linuxbox ~]$ echo ${foo:5:6}

string

If the value of offset is negative, it is taken to mean it starts from the end of the string rather than the beginning. Note that negative values must be preceded by a space to pre- vent confusion with the ${parameter:-word} expansion. length, if present, must not be less than zero.

若 offset 的值为负数，则认为 offset 值是从字符串的末尾开始算起，而不是从开头。注意负数前面必须有一个空格， 为防止与 ${parameter:-word} 展开形式混淆。length，若出现，则必须不能小于零。

If parameter is @, the result of the expansion is length positional parameters, starting at offset.

如果 parameter 是 @，展开结果是 length 个位置参数，从第 offset 个位置参数开始。

[me@linuxbox ~]$ foo="This string is long."

[me@linuxbox ~]$ echo ${foo: -5}

long.

[me@linuxbox ~]$ echo ${foo: -5:2}

lo

**${parameter#pattern}**

**${parameter##pattern}**

These expansions remove a leading portion of the string contained in parameter defined by pattern. pattern is a wildcard pattern like those used in pathname expansion. The dif- ference in the two forms is that the # form removes the shortest match, while the ## form removes the longest match.

这些展开会从 paramter 所包含的字符串中清除开头一部分文本，这些字符要匹配定义的 patten。pattern 是 通配符模式，就如那些用在路径名展开中的模式。这两种形式的差异之处是该 # 形式清除最短的匹配结果， 而该 ## 模式清除最长的匹配结果。

[me@linuxbox ~]$ foo=file.txt.zip

[me@linuxbox ~]$ echo ${foo#\*.}

txt.zip

[me@linuxbox ~]$ echo ${foo##\*.}

zip

**${parameter%pattern}**

**${parameter%%pattern}**

These expansions are the same as the # and ## expansions above, except they remove text from the end of the string contained in parameter rather than from the beginning.

这些展开和上面的 # 和 ## 展开一样，除了它们清除的文本从 parameter 所包含字符串的末尾开始，而不是开头。

[me@linuxbox ~]$ foo=file.txt.zip

[me@linuxbox ~]$ echo ${foo%.\*}

file.txt

[me@linuxbox ~]$ echo ${foo%%.\*}

file

**${parameter/pattern/string}**

**${parameter//pattern/string}**

**${parameter/#pattern/string}**

**${parameter/%pattern/string}**

This expansion performs a search-and-replace upon the contents of parameter. If text is found matching wildcard pattern, it is replaced with the contents of string. In the normal form, only the first occurrence of pattern is replaced. In the // form, all occurrences are replaced. The /# form requires that the match occur at the beginning of the string, and the /% form requires the match to occur at the end of the string. /string may be omitted, which causes the text matched by pattern to be deleted.

这种形式的展开对 parameter 的内容执行查找和替换操作。如果找到了匹配通配符 pattern 的文本， 则用 string 的内容替换它。在正常形式下，只有第一个匹配项会被替换掉。在该 // 形式下，所有的匹配项都会被替换掉。 该 /# 要求匹配项出现在字符串的开头，而 /% 要求匹配项出现在字符串的末尾。/string 可能会省略掉，这样会 导致删除匹配的文本。

[me@linuxbox~]$ foo=JPG.JPG

[me@linuxbox ~]$ echo ${foo/JPG/jpg}

jpg.JPG

[me@linuxbox~]$ echo ${foo//JPG/jpg}

jpg.jpg

[me@linuxbox~]$ echo ${foo/#JPG/jpg}

jpg.JPG

[me@linuxbox~]$ echo ${foo/%JPG/jpg}

JPG.jpg

Parameter expansion is a good thing to know. The string manipulation expansions can be used as substitutes for other common commands such as sed and cut. Expansions improve the efficiency of scripts by eliminating the use of external programs. As an example, we will modify the longest-word program discussed in the previous chapter to use the parameter expansion ${#j} in place of the command substitution **$(echo $j | wc -c)** and its resulting subshell, like so:

知道参数展开是件很好的事情。字符串操作展开可以用来替换其它常见命令比方说 sed 和 cut。 通过减少使用外部程序，展开提高了脚本的效率。举例说明，我们将修改在之前章节中讨论的 longest-word 程序， 用参数展开 ${#j} 取代命令 $(echo $j | wc -c) 及其 subshell ，像这样：

#!/bin/bash

# longest-word3 : find longest string in a file

for i; do

if [[ -r $i ]]; then

max\_word=

max\_len=

for j in $(strings $i); do

len=${#j}

if (( len > max\_len )); then

max\_len=$len

max\_word=$j

fi

done

echo "$i: '$max\_word' ($max\_len characters)"

fi

shift

done

Next, we will compare the efficiency of the two versions by using the time command:

下一步，我们将使用 time 命令来比较这两个脚本版本的效率：

[me@linuxbox ~]$ time longest-word2 dirlist-usr-bin.txt

dirlist-usr-bin.txt: 'scrollkeeper-get-extended-content-list' (38

characters)

real 0m3.618s

user 0m1.544s

sys 0m1.768s

[me@linuxbox ~]$ time longest-word3 dirlist-usr-bin.txt

dirlist-usr-bin.txt: 'scrollkeeper-get-extended-content-list' (38

characters)

real 0m0.060s

user 0m0.056s

sys 0m0.008s

The original version of the script takes 3.618 seconds to scan the text file, while the new version, using parameter expansion, takes only 0.06 seconds — a very significant improvement.

原来的脚本扫描整个文本文件需耗时3.168秒，而该新版本，使用参数展开，仅仅花费了0.06秒 —— 一个非常巨大的提高。

**大小写转换**

Recent versions of bash have support for upper/lowercase conversion of strings. bash has four parameter expansions and two options to the declare command to support it.

最新的 bash 版本已经支持字符串的大小写转换了。bash 有四个参数展开和 declare 命令的两个选项来支持大小写转换。

So what is case conversion good for? Aside from the obvious aesthetic value, it has an important role in programming. Let’s consider the case of a database look-up. Imagine that a user has entered a string into a data input field that we want to look up in a database. It’s possible the user will enter the value in all uppercase letters or lowercase letters or a combination of both. We certainly don’t want to populate our database with every possible permutation of upper and lower case spellings. What to do?

那么大小写转换对什么有好处呢？ 除了明显的审美价值，它在编程领域还有一个重要的角色。 让我们考虑一个数据库查询的案例。假设一个用户已经敲写了一个字符串到数据输入框中， 而我们想要在一个数据库中查找这个字符串。该用户输入的字符串有可能全是大写字母或全是小写或是两者的结合。 我们当然不希望把每个可能的大小写拼写排列填充到我们的数据库中。那怎么办？

A common approach to this problem is to normalize the user’s input. That is, convert it into a standardized form before we attempt the database look-up. We can do this by converting all of the characters in the user’s input to either lower or uppercase and ensure that the database entries are normalized the same way.

解决这个问题的常见方法是规范化用户输入。也就是，在我们试图查询数据库之前，把用户的输入转换成标准化。 我们能做到这一点，通过把用户输入的字符全部转换成小写字母或大写字母，并且确保数据库中的条目 按同样的方式规范化。

The declare command can be used to normalize strings to either upper or lowercase. Using declare, we can force a variable to always contain the desired format no matter what is assigned to it:

这个 declare 命令可以用来把字符串规范成大写或小写字符。使用 declare 命令，我们能强制一个 变量总是包含所需的格式，无论如何赋值给它。

#!/bin/bash

# ul-declare: demonstrate case conversion via declare

declare -u upper

declare -l lower

if [[ $1 ]]; then

upper="$1"

lower="$1"

echo $upper

echo $lower

fi

In the above script, we use declare to create two variables, upper and lower. We assign the value of the first command line argument (positional parameter 1) to each of the variables and then display them on the screen:

在上面的脚本中，我们使用 declare 命令来创建两个变量，upper 和 lower。我们把第一个命令行参数的值（位置参数1）赋给 每一个变量，然后把变量值在屏幕上显示出来：

[me@linuxbox ~]$ ul-declare aBc

ABC

abc

As we can see, the command line argument (“aBc”) has been normalized.

正如我们所看到的，命令行参数（“aBc”）已经规范化了。

There are four parameter expansions that perform upper/lowercase conversion:

有四个参数展开，可以执行大小写转换操作：

|  |  |
| --- | --- |
| *Table 35-1: Case Conversion Parameter Expansions* | |
| Format | Result |
| ${parameter,,} | Expand the value of parameter into all lowercase. |
| ${parameter,} | Expand the value of parameter changing only the first character to lowercase. |
| ${parameter^^} | Expand the value of parameter into all uppercase letters. |
| ${parameter^} | Expand the value of parameter changing only the first character to uppercase (capitalization). |

|  |  |
| --- | --- |
| *表 35-1: 大小写转换参数展开* | |
| 格式 | 结果 |
| ${parameter,,} | 把 parameter 的值全部展开成小写字母。 |
| ${parameter,} | 仅仅把 parameter 的第一个字符展开成小写字母。 |
| ${parameter^^} | 把 parameter 的值全部转换成大写字母。 |
| ${parameter^} | 仅仅把 parameter 的第一个字符转换成大写字母（首字母大写）。 |

Here is a script that demonstrates these expansions:

这里是一个脚本，演示了这些展开格式：

#!/bin/bash

# ul-param - demonstrate case conversion via parameter expansion

if [[ $1 ]]; then

echo ${1,,}

echo ${1,}

echo ${1^^}

echo ${1^}

fi

Here is the script in action:

这里是脚本运行后的结果：

[me@linuxbox ~]$ ul-param aBc

abc

aBc

ABC

ABc

Again, we process the first command line argument and output the four variations supported by the parameter expansions. While this script uses the first positional parameter, parameter my be any string, variable, or string expression.

再次，我们处理了第一个命令行参数，输出了由参数展开支持的四种变体。尽管这个脚本使用了第一个位置参数， 但参数可以是任意字符串，变量，或字符串表达式。

**算术求值和展开**

We looked at arithmetic expansion in Chapter 7. It is used to perform various arithmetic operations on integers. Its basic form is:

我们在第七章中已经接触过算术展开了。它被用来对整数执行各种算术运算。它的基本格式是：

$((expression))

where expression is a valid arithmetic expression.

这里的 expression 是一个有效的算术表达式。

This is related to the compound command (( )) used for arithmetic evaluation (truth tests) we encountered in Chapter 27.

这个与复合命令 (( )) 有关，此命令用做算术求值（真测试），我们在第27章中遇到过。

In previous chapters, we saw some of the common types of expressions and operators. Here, we will look at a more complete list.

在之前的章节中，我们看到过一些类型的表达式和运算符。这里，我们将看到一个更完整的列表。

**数基**

Back in Chapter 9, we got a look at octal (base 8) and hexadecimal (base 16) numbers. In arithmetic expressions, the shell supports integer constants in any base.

回到第9章，我们看过八进制（以8为底）和十六进制（以16为底）的数字。在算术表达式中，shell 支持任意进制的整形常量。

|  |  |
| --- | --- |
| *Table 35-2: Specifying Different Number Bases* | |
| Notation | Description |
| number | By default, numbers without any notation are treated as decimal (base 10) integers. |
| 0number | In arithmetic expressions, numbers with a leading zero are considered octal. |
| 0xnumber | Hexadecimal notation |
| base#number | number is in base |

|  |  |
| --- | --- |
| *表 35-2: 指定不同的数基* | |
| 表示法 | 描述 |
| number | 默认情况下，没有任何表示法的数字被看做是十进制数（以10为底）。 |
| 0number | 在算术表达式中，以零开头的数字被认为是八进制数。 |
| 0xnumber | 十六进制表示法 |
| base#number | number 以 base 为底 |

Some examples:

一些例子：

[me@linuxbox ~]$ echo $((0xff))

255

[me@linuxbox ~]$ echo $((2#11111111))

255

In the examples above, we print the value of the hexadecimal number ff (the largest two-digit number) and the largest eight-digit binary (base 2) number.

在上面的示例中，我们打印出十六进制数 ff（最大的两位数）的值和最大的八位二进制数（以2为底）。

**一元运算符**

There are two unary operators, the + and -, which are used to indicate if a number is pos- itive or negative, respectively. For example, -5.

有两个二元运算符，+ 和 -，它们被分别用来表示一个数字是正数还是负数。例如，-5。

**简单算术**

The ordinary arithmetic operators are listed in the table below:

下表中列出了普通算术运算符：

|  |  |
| --- | --- |
| *Table 35-3: Arithmetic Operators* | |
| Operator | Description |
| + | Addition |
| - | Subtraction |
| \* | Multiplication |
| / | Integer division |
| \*\* | Exponentiation |
| % | Modulo (remainder) |

|  |  |
| --- | --- |
| *表 35-3: 算术运算符* | |
| 运算符 | 描述 |
| + | 加 |
| - | 减 |
| \* | 乘 |
| / | 整除 |
| \*\* | 乘方 |
| % | 取模（余数） |

Most of these are self-explanatory, but integer division and modulo require further discussion.

其中大部分运算符是不言自明的，但是整除和取模运算符需要进一步解释一下。

Since the shell’s arithmetic only operates on integers, the results of division are always whole numbers:

因为 shell 算术只操作整形，所以除法运算的结果总是整数：

[me@linuxbox ~]$ echo $(( 5 / 2 ))

2

This makes the determination of a remainder in a division operation more important:

这使得确定除法运算的余数更为重要：

[me@linuxbox ~]$ echo $(( 5 % 2 ))

1

By using the division and modulo operators, we can determine that 5 divided by 2 results in 2, with a remainder of 1.

通过使用除法和取模运算符，我们能够确定5除以2得数是2，余数是1。

Calculating the remainder is useful in loops. It allows an operation to be performed at specified intervals during the loop’s execution. In the example below, we display a line of numbers, highlighting each multiple of 5:

在循环中计算余数是很有用处的。在循环执行期间，它允许某一个操作在指定的间隔内执行。在下面的例子中， 我们显示一行数字，并高亮显示5的倍数：

#!/bin/bash

# modulo : demonstrate the modulo operator

for ((i = 0; i <= 20; i = i + 1)); do

remainder=$((i % 5))

if (( remainder == 0 )); then

printf "<%d> " $i

else

printf "%d " $i

fi

done

printf "\n"

When executed, the results look like this:

当脚本执行后，输出结果看起来像这样：

[me@linuxbox ~]$ modulo

<0> 1 2 3 4 <5> 6 7 8 9 <10> 11 12 13 14 <15> 16 17 18 19 <20>

**赋值运算符**

Although its uses may not be immediately apparent, arithmetic expressions may perform assignment. We have performed assignment many times, though in a different context. Each time we give a variable a value, we are performing assignment. We can also do it within arithmetic expressions:

尽管它的使用不是那么明显，算术表达式可能执行赋值运算。虽然在不同的上下文中，我们已经执行了许多次赋值运算。 每次我们给变量一个值，我们就执行了一次赋值运算。我们也能在算术表达式中执行赋值运算：

[me@linuxbox ~]$ foo=

[me@linuxbox ~]$ echo $foo

[me@linuxbox ~]$ if (( foo = 5 ));then echo "It is true."; fi

It is true.

[me@linuxbox ~]$ echo $foo

5

In the example above, we first assign an empty value to the variable foo and verify that it is indeed empty. Next, we perform an if with the compound command (( foo = 5 )). This process does two interesting things: 1) it assigns the value of 5 to the variable foo, and 2) it evaluates to true because foo was assigned a nonzero value.

在上面的例子中，首先我们给变量 foo 赋了一个空值，然后验证 foo 的确为空。下一步，我们执行一个 if 复合命令 (( foo = 5 ))。 这个过程完成两件有意思的事情：1）它把5赋值给变量 foo，2）它计算测试条件为真，因为 foo 的值非零。

Note: It is important to remember the exact meaning of the = in the expression above. A single = performs assignment. foo = 5 says “make foo equal to 5,” while == evaluates equivalence. foo == 5 says “does foo equal 5?” This can be very confusing because the test command accepts a single = for string equiva- lence. This is yet another reason to use the more modern [[ ]] and (( )) com- pound commands in place of test.

注意： 记住上面表达式中 = 符号的真正含义非常重要。单个 = 运算符执行赋值运算。foo = 5 是说“使得 foo 等于5”， 而 == 运算符计算等价性。foo == 5 是说“是否 foo 等于5？”。这会让人感到非常迷惑，因为 test 命令接受单个 = 运算符 来测试字符串等价性。这也是使用更现代的 [[ ]] 和 (( )) 复合命令来代替 test 命令的另一个原因。

In addition to the =, the shell also provides notations that perform some very useful as- signments:

除了 = 运算符，shell 也提供了其它一些表示法，来执行一些非常有用的赋值运算：

|  |  |
| --- | --- |
| *Table 35-4: Assignment Operators* | |
| Notation | Description |
| parameter = value | Simple assignment. Assigns value to parameter. |
| parameter += value | Addition. Equivalent to parameter = parameter + value. |
| parameter -= value | Subtraction. Equivalent to parameter = parameter - value. |
| parameter \*= value | Multiplication. Equivalent to parameter = parameter \* value. |
| parameter /= value | Integer division. Equivalent to parameter = parameter / value. |
| parameter %= value | Modulo. Equivalent to parameter = parameter % value. |
| parameter++ | Variable post-increment. Equivalent to parameter = parameter + 1 (however, see discussion below). |
| parameter-- | Variable post-decrement. Equivalent to parameter = parameter - 1. |
| ++parameter | Variable pre-increment. Equivalent to parameter = parameter + 1. |
| --parameter | Variable pre-decrement. Equivalent to parameter = parameter - 1. |

|  |  |
| --- | --- |
| *表35-4: 赋值运算符* | |
| 表示法 | 描述 |
| parameter = value | 简单赋值。给 parameter 赋值。 |
| parameter += value | 加。等价于 parameter = parameter + value。 |
| parameter -= value | 减。等价于 parameter = parameter – value。 |
| parameter \*= value | 乘。等价于 parameter = parameter \* value。 |
| parameter /= value | 整除。等价于 parameter = parameter / value。 |
| parameter %= value | 取模。等价于 parameter = parameter % value。 |
| parameter++ | 后缀自增变量。等价于 parameter = parameter + 1 (但，要看下面的讨论)。 |
| parameter-- | 后缀自减变量。等价于 parameter = parameter - 1。 |
| ++parameter | 前缀自增变量。等价于 parameter = parameter + 1。 |
| --parameter | 前缀自减变量。等价于 parameter = parameter - 1。 |

These assignment operators provide a convenient shorthand for many common arithmetic tasks. Of special interest are the increment (++) and decrement (--) operators, which increase or decrease the value of their parameters by one. This style of notation is taken from the C programming language and has been incorporated by several other programming languages, including bash.

这些赋值运算符为许多常见算术任务提供了快捷方式。特别关注一下自增（++）和自减（--）运算符，它们会把它们的参数值加1或减1。 这种风格的表示法取自C 编程语言并且被其它几种编程语言吸收，包括 bash。

The operators may appear either at the front of a parameter or at the end. While they both either increment or decrement the parameter by one, the two placements have a subtle difference. If placed at the front of the parameter, the parameter is incremented (or decre- mented) before the parameter is returned. If placed after, the operation is performed after the parameter is returned. This is rather strange, but it is the intended behavior. Here is a demonstration:

自增和自减运算符可能会出现在参数的前面或者后面。然而它们都是把参数值加1或减1，这两个位置有个微小的差异。 若运算符放置在参数的前面，参数值会在参数返回之前增加（或减少）。若放置在后面，则运算会在参数返回之后执行。 这相当奇怪，但这是它预期的行为。这里是个演示的例子：

[me@linuxbox ~]$ foo=1

[me@linuxbox ~]$ echo $((foo++))

1

[me@linuxbox ~]$ echo $foo

2

If we assign the value of one to the variable foo and then increment it with the ++ operator placed after the parameter name, foo is returned with the value of one. However, if we look at the value of the variable a second time, we see the incremented value. If we place the ++ operator in front of the parameter, we get the more expected behavior:

如果我们把1赋值给变量 foo，然后通过把自增运算符 ++ 放到参数名 foo 之后来增加它，foo 返回1。 然而，如果我们第二次查看变量 foo 的值，我们看到它的值增加了1。若我们把 ++ 运算符放到参数 foo 之前， 我们得到更期望的行为：

[me@linuxbox ~]$ foo=1

[me@linuxbox ~]$ echo $((++foo))

2

[me@linuxbox ~]$ echo $foo

2

For most shell applications, prefixing the operator will be the most useful.

对于大多数 shell 应用来说，前缀运算符最有用。

The ++ and -- operators are often used in conjunction with loops. We will make some improvements to our modulo script to tighten it up a bit:

自增 ++ 和 自减 -- 运算符经常和循环操作结合使用。我们将改进我们的 modulo 脚本，让代码更紧凑些：

#!/bin/bash

# modulo2 : demonstrate the modulo operator

for ((i = 0; i <= 20; ++i )); do

if (((i % 5) == 0 )); then

printf "<%d> " $i

else

printf "%d " $i

fi

done

printf "\n"

**位运算符**

One class of operators manipulates numbers in an unusual way. These operators work at the bit level. They are used for certain kinds of low level tasks, often involving setting or reading bit-flags.

位运算符是一类以不寻常的方式操作数字的运算符。这些运算符工作在位级别的数字。它们被用在某类底层的任务中， 经常涉及到设置或读取位标志。

|  |  |
| --- | --- |
| *Table 35-5: Bit Operators* | |
| Operator | Description |
| ~ | Bitwise negation. Negate all the bits in a number. |
| << | Left bitwise shift. Shift all the bits in a number to the left. |
| >> | Right bitwise shift. Shift all the bits in a number to the right. |
| & | Bitwise AND. Perform an AND operation on all the bits in two numbers. |
| | | Bitwise OR. Perform an OR operation on all the bits in two numbers. |
| ^ | Bitwise XOR. Perform an exclusive OR operation on all the bits in two numbers. |

|  |  |
| --- | --- |
| *表35-5: 位运算符* | |
| 运算符 | 描述 |
| ~ | 按位取反。对一个数字所有位取反。 |
| << | 位左移. 把一个数字的所有位向左移动。 |
| >> | 位右移. 把一个数字的所有位向右移动。 |
| & | 位与。对两个数字的所有位执行一个 AND 操作。 |
| | | 位或。对两个数字的所有位执行一个 OR 操作。 |
| ^ | 位异或。对两个数字的所有位执行一个异或操作。 |

Note that there are also corresponding assignment operators (for example, <<=) for all but bitwise negation.

注意除了按位取反运算符之外，其它所有位运算符都有相对应的赋值运算符（例如，<<=）。

Here we will demonstrate producing a list of powers of 2, using the left bitwise shift operator:

这里我们将演示产生2的幂列表的操作，使用位左移运算符：

[me@linuxbox ~]$ for ((i=0;i<8;++i)); do echo $((1<<i)); done

1

2

4

8

16

32

64

128

**逻辑运算符**

As we discovered in Chapter 27, the (( )) compound command supports a variety of comparison operators. There are a few more that can be used to evaluate logic. Here is the complete list:

正如我们在第27章中所看到的，复合命令 (( )) 支持各种各样的比较运算符。还有一些可以用来计算逻辑运算。 这里是比较运算符的完整列表：

|  |  |
| --- | --- |
| *Table 35-6: Comparison Operators* | |
| Operator | Description |
| <= | Less than or equal to |
| >= | great than or equal to |
| < | less than |
| > | greater than |
| == | Equal to |
| != | Not equal to |
| && | Logical AND |
| || | Logical OR |
| expr1?expr2:expr3 | Comparison (ternary) operator. If expression expr1 evaluates to be non-zero (arithmetic true) then expr2, else expr3. |

|  |  |
| --- | --- |
| *表35-6: 比较运算符* | |
| 运算符 | 描述 |
| <= | 小于或相等 |
| >= | 大于或相等 |
| < | 小于 |
| > | 大于 |
| == | 相等 |
| != | 不相等 |
| && | 逻辑与 |
| || | 逻辑或 |
| expr1?expr2:expr3 | 条件（三元）运算符。若表达式 expr1 的计算结果为非零值（算术真），则 执行表达式 expr2，否则执行表达式 expr3。 |

When used for logical operations, expressions follow the rules of arithmetic logic; that is, expressions that evaluate as zero are considered false, while non-zero expressions are considered true. The (( )) compound command maps the results into the shell’s normal exit codes:

当表达式用于逻辑运算时，表达式遵循算术逻辑规则；也就是，表达式的计算结果是零，则认为假，而非零表达式认为真。 该 (( )) 复合命令把结果映射成 shell 正常的退出码：

[me@linuxbox ~]$ if ((1)); then echo "true"; else echo "false"; fi

true

[me@linuxbox ~]$ if ((0)); then echo "true"; else echo "false"; fi

false

The strangest of the logical operators is the ternary operator. This operator (which is modeled after the one in the C programming language) performs a standalone logical test. It can be used as a kind of if/then/else statement. It acts on three arithmetic expressions (strings won’t work), and if the first expression is true (or non-zero) the second expression is performed. Otherwise, the third expression is performed. We can try this on the command line:

最陌生的逻辑运算符就是这个三元运算符了。这个运算符（仿照于 C 编程语言里的三元运算符）执行一个单独的逻辑测试。 它用起来类似于 if/then/else 语句。它操作三个算术表达式（字符串不会起作用），并且若第一个表达式为真（或非零）， 则执行第二个表达式。否则，执行第三个表达式。我们可以在命令行中实验一下：

[me@linuxbox~]$ a=0

[me@linuxbox~]$ ((a<1?++a:--a))

[me@linuxbox~]$ echo $a

1

[me@linuxbox~]$ ((a<1?++a:--a))

[me@linuxbox~]$ echo $a

0

Here we see a ternary operator in action. This example implements a toggle. Each time the operator is performed, the value of the variable a switches from zero to one or vice versa.

这里我们看到一个实际使用的三元运算符。这个例子实现了一个切换。每次运算符执行的时候，变量 a 的值从零变为1，或反之亦然。

Please note that performing assignment within the expressions is not straightforward.

请注意在表达式内执行赋值却并非易事。

When attempted, bash will declare an error:

当企图这样做时，bash 会声明一个错误：

[me@linuxbox ~]$ a=0

[me@linuxbox ~]$ ((a<1?a+=1:a-=1))

bash: ((: a<1?a+=1:a-=1: attempted assignment to non-variable (error token is "-=1")

This problem can be mitigated by surrounding the assignment expression with parentheses:

通过把赋值表达式用括号括起来，可以解决这个错误：

[me@linuxbox ~]$ ((a<1?(a+=1):(a-=1)))

Next, we see a more complete example of using arithmetic operators in a script that produces a simple table of numbers:

下一步，我们看一个使用算术运算符更完备的例子，该示例产生一个简单的数字表格：

#!/bin/bash

# arith-loop: script to demonstrate arithmetic operators

finished=0

a=0

printf "a\ta\*\*2\ta\*\*3\n"

printf "=\t====\t====\n"

until ((finished)); do

b=$((a\*\*2))

c=$((a\*\*3))

printf "%d\t%d\t%d\n" $a $b $c

((a<10?++a:(finished=1)))

done

In this script, we implement an until loop based on the value of the finished variable. Initially, the variable is set to zero (arithmetic false) and we continue the loop until it becomes non-zero. Within the loop, we calculate the square and cube of the counter variable **a**. At the end of the loop, the value of the counter variable is evaluated. If it is less than 10 (the maximum number of iterations), it is incremented by one, else the variable **finished** is given the value of one, making finished arithmetically true, thereby terminating the loop. Running the script gives this result:

在这个脚本中，我们基于变量 finished 的值实现了一个 until 循环。首先，把变量 finished 的值设为零（算术假）， 继续执行循环之道它的值变为非零。在循环体内，我们计算计数器 a 的平方和立方。在循环末尾，计算计数器变量 a 的值。 若它小于10（最大迭代次数），则 a 的值加1，否则给变量 finished 赋值为1，使得变量 finished 算术为真， 从而终止循环。运行该脚本得到这样的结果：

[me@linuxbox ~]$ arith-loop

a a\*\*2 a\*\*3

= ==== ====

0 0 0

1 1 1

2 4 8

3 9 27

4 16 64

5 25 125

6 36 216

7 49 343

8 64 512

9 81 729

10 100 1000

**bc - 一种高精度计算器语言**

We have seen how the shell can handle all types of integer arithmetic, but what if we need to perform higher math or even just use floating point numbers? The answer is, we can’t. At least not directly with the shell. To do this, we need to use an external program. There are several approaches we can take. Embedding Perl or AWK programs is one possible solution, but unfortunately, outside the scope of this book. Another approach is to use a specialized calculator program. One such program found on most Linux systems is called bc.

我们已经看到 shell 是可以处理所有类型的整形算术的，但是如果我们需要执行更高级的数学运算或仅使用浮点数，该怎么办？ 答案是，我们不能这样做。至少不能直接用 shell 完成此类运算。为此，我们需要使用外部程序。 有几种途径可供我们采用。嵌入的 Perl 或者 AWK 程序是一种可能的方案，但是不幸的是，超出了本书的内容大纲。 另一种方式就是使用一种专业的计算器程序。这样一个程序叫做 bc，在大多数 Linux 系统中都可以找到。

The bc program reads a file written in its own C-like language and executes it. A bc script may be a separate file or it may be read from standard input. The bc language supports quite a few features including variables, loops, and programmer-defined functions. We won’t cover bc entirely here, just enough to get a taste. bc is well documented by its man page.

该 bc 程序读取一个用它自己的类似于 C 语言的语法编写的脚本文件。一个 bc 脚本可能是一个分离的文件或者是读取 标准输入。bc 语言支持相当少的功能，包括变量，循环和程序员定义的函数。这里我们不会讨论整个 bc 语言， 仅仅体验一下。查看 bc 的手册页，其文档整理非常好。

Let’s start with a simple example. We’ll write a bc script to add 2 plus 2:

让我们从一个简单的例子开始。我们将编写一个 bc 脚本来执行2加2运算：

/\* A very simple bc script \*/

2 + 2

The first line of the script is a comment. bc uses the same syntax for comments as the C programming language. Comments, which may span multiple lines, begin with /\* and end with \*/.

脚本的第一行是一行注释。bc 使用和 C 编程语言一样的注释语法。注释，可能会跨越多行，开始于 /\* 结束于 \*/。

**使用 bc**

If we save the bc script above as foo.bc, we can run it this way:

如果我们把上面的 bc 脚本保存为 foo.bc，然后我们就能这样运行它：

[me@linuxbox ~]$ bc foo.bc

bc 1.06.94

Copyright 1991-1994, 1997, 1998, 2000, 2004, 2006 Free Software

Foundation, Inc.

This is free software with ABSOLUTELY NO WARRANTY.

For details type `warranty'.

4

If we look carefully, we can see the result at the very bottom, after the copyright message. This message can be suppressed with the -q (quiet) option. bc can also be used interactively:

如果我们仔细观察，我们看到算术结果在最底部，版权信息之后。可以通过 -q（quiet）选项禁止这些版权信息。 bc 也能够交互使用：

[me@linuxbox ~]$ bc -q

2 + 2

4

quit

When using bc interactively, we simply type the calculations we wish to perform, and the results are immediately displayed. The bc command quit ends the interactive session.

当使用 bc 交互模式时，我们简单地输入我们希望执行的运算，结果就立即显示出来。bc 的 quit 命令结束交互会话。

It is also possible to pass a script to bc via standard input:

也可能通过标准输入把一个脚本传递给 bc 程序：

[me@linuxbox ~]$ bc < foo.bc

4

The ability to take standard input means that we can use here documents, here strings, and pipes to pass scripts. This is a here string example:

这种接受标准输入的能力，意味着我们可以使用 here 文档，here字符串，和管道来传递脚本。这里是一个使用 here 字符串的例子：

[me@linuxbox ~]$ bc <<< "2+2"

4

**一个脚本实例**

As a real-world example, we will construct a script that performs a common calculation, monthly loan payments. In the script below, we use a here document to pass a script to bc:

作为一个真实世界的例子，我们将构建一个脚本，用于计算每月的还贷金额。在下面的脚本中， 我们使用了 here 文档把一个脚本传递给 bc：

#!/bin/bash

# loan-calc : script to calculate monthly loan payments

PROGNAME=$(basename $0)

usage () {

cat <<- EOF

Usage: $PROGNAME PRINCIPAL INTEREST MONTHS

Where:

PRINCIPAL is the amount of the loan.

INTEREST is the APR as a number (7% = 0.07).

MONTHS is the length of the loan's term.

EOF

}

if (($# != 3)); then

usage

exit 1

fi

principal=$1

interest=$2

months=$3

bc <<- EOF

scale = 10

i = $interest / 12

p = $principal

n = $months

a = p \* ((i \* ((1 + i) ^ n)) / (((1 + i) ^ n) - 1))

print a, "\n"

EOF

When executed, the results look like this:

当脚本执行后，输出结果像这样：

[me@linuxbox ~]$ loan-calc 135000 0.0775 180

475

1270.7222490000

This example calculates the monthly payment for a $135,000 loan at 7.75% APR for 180 months (15 years). Notice the precision of the answer. This is determined by the value given to the special **scale**variable in the bc script. A full description of the bc scripting language is provided by the bc man page. While its mathematical notation is slightly different from that of the shell (bc more closely resembles C), most of it will be quite familiar, based on what we have learned so far.

若贷款 135,000 美金，年利率为 7.75%，借贷180个月（15年），这个例子计算出每月需要还贷的金额。 注意这个答案的精确度。这是由脚本中变量 scale 的值决定的。bc 的手册页提供了对 bc 脚本语言的详尽描述。 虽然 bc 的数学符号与 shell 的略有差异（bc 与 C 更相近），但是基于目前我们所学的内容， 大多数符号是我们相当熟悉的。

**总结**

In this chapter, we have learned about many of the little things that can be used to get the “real work” done in scripts. As our experience with scripting grows, the ability to effectively manipulate strings and numbers will prove extremely valuable. Our loan-calc script demonstrates that even simple scripts can be created to do some really useful things.

在这一章中，我们学习了很多小东西，在脚本中这些小零碎可以完成“真正的工作”。随着我们编写脚本经验的增加， 能够有效地操作字符串和数字的能力将具有极为重要的价值。我们的 loan-calc 脚本表明， 甚至可以创建简单的脚本来完成一些真正有用的事情。

**额外加分**

While the basic functionality of the loan-calc script is in place, the script is far from complete. For extra credit, try improving the loan-calc script with the following features:

虽然该 loan-calc 脚本的基本功能已经很到位了，但脚本还远远不够完善。为了额外加分，试着 给脚本 loan-calc 添加以下功能：

* Full verification of the command line arguments
* A command line option to implement an “interactive” mode that will prompt the user to input the principal, interest rate, and term of the loan.
* A better format for the output.
* 完整的命令行参数验证
* 用一个命令行选项来实现“交互”模式，提示用户输入本金、利率和贷款期限
* 输出格式美化

**拓展阅读**

* The Bash Hackers Wiki has a good discussion of parameter expansion:
* 《Bash Hackers Wiki》对参数展开有一个很好的论述：

<http://wiki.bash-hackers.org/syntax/pe>

* The Bash Reference Manual covers this, too:
* 《Bash 参考手册》也介绍了这个：

<http://www.gnu.org/software/bash/manual/bashref.html#Shell-Parameter-Expansion>

* The Wikipedia has a good article describing bit operations:
* Wikipedia 上面有一篇很好的文章描述了位运算：

<http://en.wikipedia.org/wiki/Bit_operation>

* and an article on ternary operations:
* 和一篇关于三元运算的文章：

<http://en.wikipedia.org/wiki/Ternary_operation>

* as well as a description of the formula for calculating loan payments used in our loan-calc script:
* 还有一个对计算还贷金额公式的描述，我们的 loan-calc 脚本中用到了这个公式：

<http://en.wikipedia.org/wiki/Amortization_calculator>