第三十六章 数组

In the last chapter, we looked at how the shell can manipulate strings and numbers. The data types we have looked at so far are known in computer science circles as scalar variables; that is, variables that contain a single value.

在上一章中，我们查看了 shell 怎样操作字符串和数字的。目前我们所见到的数据类型在计算机科学圈里被 成为标量变量；也就是说，只能包含一个值的变量。

In this chapter, we will look at another kind of data structure called an array, which holds multiple values. Arrays are a feature of virtually every programming language. The shell supports them, too, though in a rather limited fashion. Even so, they can be very useful for solving programming problems.

在本章中，我们将看看另一种数据结构叫做数组，数组能存放多个值。数组几乎是所有编程语言的一个特性。 shell 也支持它们，尽管以一个相当有限的形式。即便如此，为解决编程问题，它们是非常有用的。

### 什么是数组？

Arrays are variables that hold more than one value at a time. Arrays are organized like a table. Let’s consider a spreadsheet as an example. A spreadsheet acts like a two-dimensional array. It has both rows and columns, and an individual cell in the spreadsheet can be located according to its row and column address. An array behaves the same way. An array has cells, which are called elements, and each element contains data. An individual array element is accessed using an address called an index or subscript.

数组是一次能存放多个数据的变量。数组的组织结构就像一张表。我们拿电子表格举例。一张电子表格就像是一个 二维数组。它既有行也有列，并且电子表格中的一个单元格，可以通过单元格所在的行和列的地址定位它的位置。 数组行为也是如此。数组有单元格，被称为元素，而且每个元素会包含数据。 使用一个称为索引或下标的地址可以访问一个单独的数组元素。

Most programming languages support multidimensional arrays. A spreadsheet is an example of a multidimensional array with two dimensions, width and height. Many languages support arrays with an arbitrary number of dimensions, though two- and three-dimensional arrays are probably the most commonly used.

大多数编程语言支持多维数组。一个电子表格就是一个多维数组的例子，它有两个维度，宽度和高度。 许多语言支持任意维度的数组，虽然二维和三维数组可能是最常用的。

Arrays in bash are limited to a single dimension. We can think of them as a spreadsheet with a single column. Even with this limitation, there are many applications for them. Array support first appeared in bash version 2. The original Unix shell program, sh, did not support arrays at all.

Bash 中的数组仅限制为单一维度。我们可以把它们看作是只有一列的电子表格。尽管有这种局限，但是有许多应用使用它们。 对数组的支持第一次出现在 bash 版本2中。原来的 Unix shell 程序，sh，根本就不支持数组。

### 创建一个数组

Array variables are named just like other bash variables, and are created automatically when they are accessed. Here is an example:

数组变量就像其它 bash 变量一样命名，当被访问的时候，它们会被自动地创建。这里是一个例子：

[me@linuxbox ~]$ a[1]=foo

[me@linuxbox ~]$ echo ${a[1]}

foo

Here we see an example of both the assignment and access of an array element. With the first command, element 1 of array a is assigned the value “foo”. The second command displays the stored value of element 1. The use of braces in the second command is re- quired to prevent the shell from attempting pathname expansion on the name of the array element.

这里我们看到一个赋值并访问数组元素的例子。通过第一个命令，把数组 a 的元素1赋值为 “foo”。 第二个命令显示存储在元素1中的值。在第二个命令中使用花括号是必需的， 以便防止 shell 试图对数组元素名执行路径名展开操作。

An array can also be created with the declare command:

也可以用 declare 命令创建一个数组：

[me@linuxbox ~]$ declare -a a

Using the -a option, this example of declare creates the array a.

使用 -a 选项，declare 命令的这个例子创建了数组 a。

### 数组赋值

Values may be assigned in one of two ways. Single values may be assigned using the fol- lowing syntax:

有两种方式可以给数组赋值。单个值赋值使用以下语法：

name[subscript]=value

where name is the name of the array and subscript is an integer (or arithmetic expression) greater than or equal to zero. Note that the first element of an array is subscript zero, not one. value is a string or integer assigned to the array element.

这里的 name 是数组的名字，subscript 是一个大于或等于零的整数（或算术表达式）。注意数组第一个元素的下标是0， 而不是1。数组元素的值可以是一个字符串或整数。

Multiple values may be assigned using the following syntax:

多个值赋值使用下面的语法：

name=(value1 value2 ...)

where name is the name of the array and value… are values assigned sequentially to elements of the array, starting with element zero. For example, if we wanted to assign abbreviated days of the week to the array days, we could do this:

这里的 name 是数组的名字，value… 是要按照顺序赋给数组的值，从元素0开始。例如，如果我们希望 把星期几的英文简写赋值给数组 days，我们可以这样做：

[me@linuxbox ~]$ days=(Sun Mon Tue Wed Thu Fri Sat)

It is also possible to assign values to a specific element by specifying a subscript for each value:

还可以通过指定下标，把值赋给数组中的特定元素：

[me@linuxbox ~]$ days=([0]=Sun [1]=Mon [2]=Tue [3]=Wed [4]=Thu [5]=Fri [6]=Sat)

### 访问数组元素

So what are arrays good for? Just as many data-management tasks can be performed with a spreadsheet program, many programming tasks can be performed with arrays.

那么数组对什么有好处呢？ 就像许多数据管理任务一样，可以用电子表格程序来完成，许多编程任务则可以用数组完成。

Let’s consider a simple data-gathering and presentation example. We will construct a script that examines the modification times of the files in a specified directory. From this data, our script will output a table showing at what hour of the day the files were last modified. Such a script could be used to determine when a system is most active. This script, called hours, produces this result:

让我们考虑一个简单的数据收集和展示的例子。我们将构建一个脚本，用来检查一个特定目录中文件的修改次数。 从这些数据中，我们的脚本将输出一张表，显示这些文件最后是在一天中的哪个小时被修改的。这样一个脚本 可以被用来确定什么时段一个系统最活跃。这个脚本，称为 hours，输出这样的结果：

[me@linuxbox ~]$ hours .

Hour Files Hour Files

---- ----- ---- ----

00 0 12 11

01 1 13 7

02 0 14 1

03 0 15 7

04 1 16 6

04 1 17 5

06 6 18 4

07 3 19 4

08 1 20 1

09 14 21 0

10 2 22 0

11 5 23 0

Total files = 80

We execute the hours program, specifying the current directory as the target. It produces a table showing, for each hour of the day (0-23), how many files were last modified. The code to produce this is as follows:

当执行该 hours 程序时，指定当前目录作为目标目录。它打印出一张表显示一天（0-23小时）每小时内， 有多少文件做了最后修改。程序代码如下所示：

#!/bin/bash

# hours : script to count files by modification time

usage () {

echo "usage: $(basename $0) directory" >&2

}

# Check that argument is a directory

if [[ ! -d $1 ]]; then

usage

exit 1

fi

# Initialize array

for i in {0..23}; do hours[i]=0; done

# Collect data

for i in $(stat -c %y "$1"/\* | cut -c 12-13); do

j=${i/#0}

((++hours[j]))

((++count))

done

# Display data

echo -e "Hour\tFiles\tHour\tFiles"

echo -e "----\t-----\t----\t-----"

for i in {0..11}; do

j=$((i + 12))

printf "%02d\t%d\t%02d\t%d\n" $i ${hours[i]} $j ${hours[j]}

done

printf "\nTotal files = %d\n" $count

The script consists of one function (usage) and a main body with four sections. In the first section, we check that there is a command line argument and that it is a directory. If it is not, we display the usage message and exit.

这个脚本由一个函数（名为 usage），和一个分为四个区块的主体组成。在第一部分，我们检查是否有一个命令行参数， 且该参数为目录。如果不是目录，会显示脚本使用信息并退出。

The second section initializes the array hours. It does this by assigning each element a value of zero. There is no special requirement to prepare arrays prior to use, but our script needs to ensure that no element is empty. Note the interesting way the loop is constructed. By employing brace expansion ({0..23}), we are able to easily generate a sequence of words for the for command.

第二部分初始化一个名为 hours 的数组。给每一个数组元素赋值一个0。虽然没有特殊需要在使用之前准备数组，但是 我们的脚本需要确保没有元素是空值。注意这个循环构建方式很有趣。通过使用花括号展开（{0..23}），我们能 很容易为 for 命令产生一系列的数据（words）。

The next section gathers the data by running the stat program on each file in the directory. We use cut to extract the two-digit hour from the result. Inside the loop, we need to remove leading zeros from the hour field, since the shell will try (and ultimately fail) to interpret values “00” through “09” as octal numbers (see Table 35-1). Next, we increment the value of the array element corresponding with the hour of the day. Finally, we increment a counter (count) to track the total number of files in the directory.

接下来的一部分收集数据，对目录中的每一个文件运行 stat 程序。我们使用 cut 命令从结果中抽取两位数字的小时字段。 在循环里面，我们需要把小时字段开头的零清除掉，因为 shell 将试图（最终会失败）把从 “00” 到 “09” 的数值解释为八进制（见表35-1）。 下一步，我们以小时为数组索引，来增加其对应的数组元素的值。最后，我们增加一个计数器的值（count），记录目录中总共的文件数目。

The last section of the script displays the contents of the array. We first output a couple of header lines and then enter a loop that produces two columns of output. Lastly, we output the final tally of files.

脚本的最后一部分显示数组中的内容。我们首先输出两行标题，然后进入一个循环产生两栏输出。最后，输出总共的文件数目。

### 数组操作

There are many common array operations. Such things as deleting arrays, determining their size, sorting, etc. have many applications in scripting.

有许多常见的数组操作。比方说删除数组，确定数组大小，排序，等等。有许多脚本应用程序。

#### 输出整个数组的内容

The subscripts \* and @ can be used to access every element in an array. As with positional parameters, the @ notation is the more useful of the two. Here is a demonstration:

下标 \* 和 @ 可以被用来访问数组中的每一个元素。与位置参数一样，@ 表示法在两者之中更有用处。 这里是一个演示：

[me@linuxbox ~]$ animals=("a dog" "a cat" "a fish")

[me@linuxbox ~]$ for i in ${animals[\*]}; do echo $i; done

a

dog

a

cat

a

fish

[me@linuxbox ~]$ for i in ${animals[@]}; do echo $i; done

a

dog

a

cat

a

fish

[me@linuxbox ~]$ for i in "${animals[\*]}"; do echo $i; done

a dog a cat a fish

[me@linuxbox ~]$ for i in "${animals[@]}"; do echo $i; done

a dog

a cat

a fish

We create the array animals and assign it three two-word strings. We then execute four loops to see the affect of word-splitting on the array contents. The behavior of notations $ {animals[\*]} and ${animals[@]} is identical until they are quoted. The \* notation results in a single word containing the array’s contents, while the @ notation results in three words, which matches the arrays “real” contents.

我们创建了数组 animals，并把三个含有两个字的字符串赋值给数组。然后我们执行四个循环看一下对数组内容进行分词的效果。 表示法 ${animals[\*]} 和 ${animals[@]}的行为是一致的直到它们被用引号引起来。

#### 确定数组元素个数

Using parameter expansion, we can determine the number of elements in an array in much the same way as finding the length of a string. Here is an example:

使用参数展开，我们能够确定数组元素的个数，与计算字符串长度的方式几乎相同。这里是一个例子:

[me@linuxbox ~]$ a[100]=foo

[me@linuxbox ~]$ echo ${#a[@]} # number of array elements

1

[me@linuxbox ~]$ echo ${#a[100]} # length of element 100

3

We create array a and assign the string “foo” to element 100. Next, we use parameter ex- pansion to examine the length of the array, using the @ notation. Finally, we look at the length of element 100 which contains the string “foo”. It is interesting to note that while we assigned our string to element 100, bash only reports one element in the array. This differs from the behavior of some other languages in which the unused elements of the array (elements 0-99) would be initialized with empty values and counted.

我们创建了数组 a，并把字符串 “foo” 赋值给数组元素100。下一步，我们使用参数展开来检查数组的长度，使用 @ 表示法。 最后，我们查看了包含字符串 “foo” 的数组元素 100 的长度。有趣的是，尽管我们把字符串赋值给数组元素100， bash 仅仅报告数组中有一个元素。这不同于一些其它语言的行为，数组中未使用的元素（元素0-99）会初始化为空值， 并把它们计入数组长度。

#### 找到数组使用的下标

As bash allows arrays to contain “gaps” in the assignment of subscripts, it is sometimes useful to determine which elements actually exist. This can be done with a parameter ex- pansion using the following forms:

因为 bash 允许赋值的数组下标包含 “间隔”，有时候确定哪个元素真正存在是很有用的。为做到这一点， 可以使用以下形式的参数展开：

**${!array[\*]}**

**${!array[@]}**

where array is the name of an array variable. Like the other expansions that use \* and @, the @ form enclosed in quotes is the most useful, as it expands into separate words:

这里的 array 是一个数组变量的名字。和其它使用符号 \* 和 @ 的展开一样，用引号引起来的 @ 格式是最有用的， 因为它能展开成分离的词。

[me@linuxbox ~]$ foo=([2]=a [4]=b [6]=c)

[me@linuxbox ~]$ for i in "${foo[@]}"; do echo $i; done

a

b

c

[me@linuxbox ~]$ for i in "${!foo[@]}"; do echo $i; done

2

4

6

#### 在数组末尾添加元素

Knowing the number of elements in an array is no help if we need to append values to the end of an array, since the values returned by the \* and @ notations do not tell us the maxi- mum array index in use. Fortunately, the shell provides us with a solution. By using the += assignment operator, we can automatically append values to the end of an array. Here, we assign three values to the array foo, and then append three more.

如果我们需要在数组末尾附加数据，那么知道数组中元素的个数是没用的，因为通过 \* 和 @ 表示法返回的数值不能 告诉我们使用的最大数组索引。幸运地是，shell 为我们提供了一种解决方案。通过使用 += 赋值运算符， 我们能够自动地把值附加到数组末尾。这里，我们把三个值赋给数组 foo，然后附加另外三个。

[me@linuxbox~]$ foo=(a b c)

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

a b c

[me@linuxbox~]$ foo+=(d e f)

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

a b c d e f

#### 数组排序

Just as with spreadsheets, it is often necessary to sort the values in a column of data. The shell has no direct way of doing this, but it’s not hard to do with a little coding:

就像电子表格，经常有必要对一列数据进行排序。Shell 没有这样做的直接方法，但是通过一点儿代码，并不难实现。

#!/bin/bash

# array-sort : Sort an array

a=(f e d c b a)

echo "Original array: ${a[@]}"

a\_sorted=($(for i in "${a[@]}"; do echo $i; done | sort))

echo "Sorted array: ${a\_sorted[@]}"

When executed, the script produces this:

当执行之后，脚本产生这样的结果：

[me@linuxbox ~]$ array-sort

Original array: f e d c b a

Sorted array:

a b c d e f

The script operates by copying the contents of the original array (a) into a second array (a\_sorted) with a tricky piece of command substitution. This basic technique can be used to perform many kinds of operations on the array by changing the design of the pipeline.

脚本运行成功，通过使用一个复杂的命令替换把原来的数组（a）中的内容复制到第二个数组（a\_sorted）中。 通过修改管道线的设计，这个基本技巧可以用来对数组执行各种各样的操作。

#### 删除数组

To delete an array, use the unset command:

删除一个数组，使用 unset 命令：

[me@linuxbox ~]$ foo=(a b c d e f)

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

a b c d e f

[me@linuxbox ~]$ unset foo

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

[me@linuxbox ~]$

unset may also be used to delete single array elements:

也可以使用 unset 命令删除单个的数组元素：

[me@linuxbox~]$ foo=(a b c d e f)

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

a b c d e f

[me@linuxbox~]$ unset 'foo[2]'

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

a b d e f

In this example, we delete the third element of the array, subscript 2. Remember, arrays start with subscript zero, not one! Notice also that the array element must be quoted to prevent the shell from performing pathname expansion.

在这个例子中，我们删除了数组中的第三个元素，下标为2。记住，数组下标开始于0，而不是1！也要注意数组元素必须 用引号引起来为的是防止 shell 执行路径名展开操作。

Interestingly, the assignment of an empty value to an array does not empty its contents:

有趣地是，给一个数组赋空值不会清空数组内容：

[me@linuxbox ~]$ foo=(a b c d e f)

[me@linuxbox ~]$ foo=

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

b c d e f

Any reference to an array variable without a subscript refers to element zero of the array:

任何引用一个不带下标的数组变量，则指的是数组元素0：

[me@linuxbox~]$ foo=(a b c d e f)

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

a b c d e f

[me@linuxbox~]$ foo=A

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

A b c d e f

### 关联数组

Recent versions of bash now support associative arrays. Associative arrays use strings rather than integers as array indexes. This capability allow interesting new approaches to managing data. For example, we can create an array called “colors” and use color names as indexes:

现在最新的 bash 版本支持关联数组了。关联数组使用字符串而不是整数作为数组索引。 这种功能给出了一种有趣的新方法来管理数据。例如，我们可以创建一个叫做 “colors” 的数组，并用颜色名字作为索引。

declare -A colors

colors["red"]="#ff0000"

colors["green"]="#00ff00"

colors["blue"]="#0000ff"

Unlike integer indexed arrays, which are created by merely referencing them, associative arrays must be created with the declare command using the new -A option.

不同于整数索引的数组，仅仅引用它们就能创建数组，关联数组必须用带有 -A 选项的 declare 命令创建。

Associative array elements are accessed in much the same way as integer indexed arrays:

访问关联数组元素的方式几乎与整数索引数组相同：

echo ${colors["blue"]}

In the next chapter, we will look at a script that makes good use of associative arrays to produce an interesting report.

在下一章中，我们将看一个脚本，很好地利用关联数组，生产出了一个有意思的报告。

### 总结

If we search the bash man page for the word “array,” we find many instances of where bash makes use of array variables. Most of these are rather obscure, but they may provide occasional utility in some special circumstances.In fact, the entire topic of arrays is rather under-utilized in shell programming owing largely to the fact that the traditional Unix shell programs (such as sh) lacked any support for arrays. This lack of popularity is unfortunate because arrays are widely used in other programming languages and provide a powerful tool for solving many kinds of programming problems.

如果我们在 bash 手册页中搜索单词 “array”的话，我们能找到许多 bash 在哪里会使用数组变量的实例。其中大部分相当晦涩难懂， 但是它们可能在一些特殊场合提供临时的工具。事实上，在 shell 编程中，整套数组规则利用率相当低，很大程度上归咎于这样的事实， 传统 Unix shell 程序（比如说 sh）缺乏对数组的支持。这样缺乏人气是不幸的，因为数组广泛应用于其它编程语言， 并为解决各种各样的编程问题，提供了一个强大的工具。

Arrays and loops have a natural affinity and are often used together. The

数组和循环有一种天然的姻亲关系，它们经常被一起使用。该

for ((expr; expr; expr))

form of loop is particularly well-suited to calculating array subscripts.

形式的循环尤其适合计算数组下标。

### 拓展阅读

* A couple of Wikipedia articles about the data structures found in this chapter:
* Wikipedia 上面有两篇关于在本章提到的数据结构的文章：

<http://en.wikipedia.org/wiki/Scalar_(computing)>

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