

## 第九章：关联数组/哈希表

生命科学学院



```
#!/usr/local/bin/perl
```



```
while ($inputline = <STDIN>) {
    while ($inputline =~ /\b[A-Z]\S+/g) {
        $word = $&;
        $word =~ s/[:,.-]$/; # remove punctuation
        for ($count = 1; $count <= @wordlist;
            $count++) {
            $found = 0;
            if ($wordlist[$count-1] eq $word) {
                $found = 1;
                $wordcount[$count-1] += 1;
                last;
            }
        }
        if ($found == 0) {
            $soldlength = @wordlist;
            $wordlist[$soldlength] = $word;
            $wordcount[$soldlength] = 1;
        }
    }
}

print ("Capitalized words and number of occurrences:\n");
for ($count = 1; $count <= @wordlist; $count++) {
    print ("$wordlist[$count-1]: $wordcount[$count-1]\n");
}
```

运行结果如下：

Here is a line of Input.  
This Input contains some Capitalized words.  
^D  
Capitalized words and number of occurrences:  
Here: 1  
Input: 2  
  
This: 1  
  
Capitalized: 1

这个程序每次从标准输入文件读一行文字，第四行起的循环匹配每行中首字母大写的单词，每找到一个循环一次，赋给简单变量\$word。在第六行中去掉标点后，查看该单词是否曾出现过，7~15 行中在@wordlist 中挨个元素做此检查，如果某个元素与\$word 相等，@wordcount 中相应的元素就增加一个数。如果没有出现过，即@wordlist 中没有元素与\$word 相等，16~20 行给@wordlist 和@wordcount 增加一个新元素。



正如你所看到的，使用数组元素产生了一些问题。首先，`@wordlist` 中哪个元素对应着哪个单词并不明显；更糟的是，每读进一个新单词，程序必须检查整个列表才能知道该单词是否曾经出现过，当列表变得较大时，这是很耗费时间的。这些问题产生的原因是数组元素通过数字下标访问，为了解决这类问题，Perl 定义了另一种数组，可以用任意简单变量值来访问其元素，这种数组叫做关联数组，也叫哈希表。为了区分关联数组变量与普通的数组变量，Perl 使用 `%` 作为其首字符，而数组变量以 `@` 打头。与其它变量名一样，`%` 后的第一个字符必须为字母，后续字符可以为字母、数字或下划线。





---

```

1 : #!/usr/local/bin/perl
2 :
3 : while ($inputline = ) {
4 :     while ($inputline =~ /\b[A-Z]\S+/g) {
5 :         $word = $&;
6 :         $word =~ s/[;.,:-]$//; # remove punctuation
7 :         $wordlist{$word} += 1;
8 :     }
9 : }
10: print ("Capitalized words and number of occurrences:\n");
11: foreach $capword (keys(%wordlist)) {
12:     print ("$capword: $wordlist{$capword}\n");
13: }

```

---





运行结果如下：

---

```
Here is a line of Input.
```

```
This Input contains some Capitalized words.
```

```
^D
```

```
Capitalized words and number of occurrences:
```


```
This: 1
```

```
Input: 2
```

```
Here: 1
```

```
Capitalized: 1
```

---



你可以看到，这次程序简单多了，读取输入并存储各单词数目从20行减少到了7行。

本程序用关联数组%wordlist跟踪首字母大写的单词，下标就用单词本身，元素值为该单词出现的次数。第11行使用了内嵌函数keys()。这个函数返回关联数组的下标列表，foreach语句就用此列表循环。

注：关联数组总是随机存储的，因此当你用keys()访问其所有元素时，不保证元素以任何顺序出现，特别值得一提的是，它们不会以被创建的顺序出现。

要想控制关联数组元素出现的次序，可以用sort()函数对keys()返回值进行排列，如：

```
foreach $capword (sort keys(%wordlist)) {  
    print ( "$capword: $wordlist{$capword} \n " );  
}
```





## 六、从数组变量复制到关联数组

与列表一样，也可以通过数组变量创建关联数组，当然，其元素数目应该为偶数，如：

```
@fruit = ("apples",17,"bananas",9,"oranges","none");  
% fruit = @fruit;
```

反之，可以把关联数组赋给数组变量，如：

```
%fruit = ("grapes",11,"lemons",27);  
@ fruit = %fruit;
```

注意，此语句中元素次序未定义，那么数组变量@fruit可能为("grapes",11,"lemons",27)或("lemons",27,"grapes",11)。

关联数组变量之间可以直接赋值，如：%fruit2 = %fruit1;还可以把数组变量同时赋给一些简单变量和一个关联数组变量，如：

```
($var1, $var2, %myarray) = @list;
```

此语句把@list的第一个元素赋给\$var1，第二个赋给\$var2，其余的赋给%myarray。

最后，关联数组可以通过返回值为列表的内嵌函数或用户定义的子程序来创建，下例中把split()函数的返回值--一个列表--赋给一个关联数组变量。

```
1:#!/usr /local/bin/ p e r l
2:
3:$inputline = <STDIN>;
4:$inputline =~ s/^\s+|\s+$//g;
5:%fruit = split (/ \s + / , $ i n p u t l i n e ) ;
6:print ("Number of bananas : $ fr u i t {\"bananas\"}\n" ) ;
```

运行结果如下：

```
oranges 5 apples 7 bananas 11 cherries 6
Number of bananas: 11
```



## 七、元素的增删

- ▶ 增加元素已经讲过， 可以通过给一个未出现过的元素赋值来向关联数组中增加新元素， 如`$fruit{"lime"} = 1` ;创建下标为lime、值为1 的新元素。
- ▶ 删除元素的方法是用内嵌函数delete， 如欲删除上述元素， 则：
- ▶ `delete( $fruit{"lime"})` ;  
注意：
  - 1、一定要使用delete 函数来删除关联数组的元素，这是唯一的方法。
  - 2、一定不要对关联数组使用内嵌函数push、pop、shift及splice， 因为其元素位置是随机的。

## 八、列出数组的索引和值

---

- ▶ 上面已经提到， `keys()` 函数返回关联数组下标的列表，如：
  - ▶ `%fruit = ( "apples", 9, "bananas", 23, "cherries", 11 );`
  - ▶ `@fruitsubs = keys ( %fruits );`
  - ▶ 这里，`@fruitsubs` 被赋给 `apples`、`bananas`、`cherries` 构成的列表，
  - ▶ 再次提请注意，此列表没有次序，若想按字母顺序排列，可使用 `sort()` 函数。
- 



## 八、列出数组的索引和值

- ▶ `@fruitindexes = sort keys(%fruits);`
- ▶ 这样结果为("apples", "bananas", "cherries")。类似的，内嵌函数`values()`返回关联数组值的列表，如：  
`% fruit = ( "apples" , 9 , "bananas " ,  
23 , "cherries" , 11 ) ;`
- ▶ `@fruitvalues = values ( %fruits ) ;`
- ▶ 这里，`@fruitvalues`可能的结果为(9, 23, 11)，次序可能不同。



## 九、用关联数组循环

前面已经出现过利用`keys()`函数的`foreach` 循环语句，这种循环效率比较低，因为每返回一个下标，还得再去寻找其值，如：

```
foreach $holder (keys(%records)) {  
    $record = $records{$holder};  
}
```

Perl 提供一种更有效的循环方式，使用内嵌函数`each()`，如：

```
%records = ("Maris", 61, "Aaron", 755, "Young", 511);  
while (($holder, $record) = each(%records)) {  
    # stuff goes here  
}
```

`each()` 函数每次返回一个双元素的列表，其第一个元素为下标，第二个元素为相应的值，最后返回一个空列表。

注意：千万不要在`each()` 循环中添加或删除元素，否则会产生不可预料后果。

## 十、用关联数组创建数据结构

用关联数组可以模拟在其它高级语言中常见的多种数据结构， 本节讲述如何用之实现： 链表、结构和树。

### 1 、（ 单）链表

链表是一种比较简单的数据结构， 可以按一定的次序存值。

每个元素含有两个域， 一个是值， 一个是引用（ 或称指针） ， 指向链表中下一个元素。一个特殊的头指针指向链表的第一个元素。

- ▶ 在Perl 中， 链表很容易用关联数组实现， 因为一个元素的值可以作为下一个元素的索引。下例为按字母顺序排列的单词链表：

```
% words = ("a b e l", "b a k e r",  
"b a k e r", "c h a r l i e",  
"c h a r l i e", "d e l t a",  
"d e l t a", "");  
$header = "a b e l";
```

上例中，简单变量\$header 含有链表中第一个单词，它同时也是关联数组第一个元素的下标，其值baker 又是下一个元素的下标，依此类推。



下标为delta 的最后一个元素的值为空串， 表示链表的结束。  
在将要处理的数据个数未知或其随程序运行而增长的情况下， 链表十分有用。  
下例用链表按字母次序输出一个文件中的单词。

```
1 : #!/usr/local/bin/perl
2 :
3 : # initialize list to empty
4 : $header = "";
5 : while ($line = <STDIN>) {
6 :     # remove leading and trailing spaces
7 :     $line =~ s/^\s+|\s+$//g;
8 :     @words = split(/\s+/, $line);
9 :     foreach $word (@words) {
10:         # remove closing punctuation, if any
11:         $word =~ s/[.,;:-]$//;
12:         # convert all words to lower case
13:         $word =~ tr/A-Z/a-z/;
14:         &add_word_to_list($word);
15:     }
16: }
17: &print_list;
18:
19: sub add_word_to_list {
20:     local($word) = @_;
```

```
21:     local($pointer);
22:
23:     # if list is empty, add first item
24:     if ($header eq "") {
25:         $header = $word;
26:         $wordlist{$word} = "";
27:         return;
28:     }
29:     # if word identical to first element in list,
30:     # do nothing
31:     return if ($header eq $word);
32:     # see whether word should be the new
33:     # first word in the list
34:     if ($header gt $word) {
35:         $wordlist{$word} = $header;
36:         $header = $word;
37:         return;
38:     }
39:     # find place where word belongs
40:     $pointer = $header;
41:     while ($wordlist{$pointer} ne "" &&
42:           $wordlist{$pointer} lt $word) {
43:         $pointer = $wordlist{$pointer};
44:     }
45:     # if word already seen, do nothing
46:     return if ($word eq $wordlist{$pointer});
47:     $wordlist{$word} = $wordlist{$pointer};
48:     $wordlist{$pointer} = $word;
49: }
50:
51: sub print_list {
52:     local ($pointer);
53:     print ("Words in this file:\n");
54:     $pointer = $header;
```

```
55: while ($pointer ne "") {
56:     print (" $pointer\n");
57:     $pointer = $wordlist{$pointer};
58: }
59: }
```

运行结果如下：

```
Here are some words.
Here are more words.
Here are still more words.
^D
Words in this file:
are
here
more
some
still
words
```

此程序分为三个部分：

- 主程序：读取输入并转换到相应的格式。
  - 子程序： `add_word_to_list`，建立排序单词链表。
  - 子程序： `print_list`，输出单词链表第3~17行为主程序，第4行初始化链表，将表头变量 `$header` 设为空串，第5行起的循环每次读取一行输入，第7行去掉头、尾的空格，第8行将句子分割成单词。9~15行的内循环每次处理一个单词，如果该单词的最后一个字符是标点符号，就去掉。
- 第13行把单词转换成全小写形式，第14行传递给子程 `add_word_to_list`。

子程序add\_word\_to\_list 先在第24 行处检查链表是否为空。如果是，第25 行将单词赋给\$header，26 行创建链表第一个元素，存贮在关联数组%wordlist 中。如果链表非空，37 行检查第一个元素是否与该单词相同，如果相同，就立刻返回。下一步检查这一新单词是否应该为链表第一个元素，即其按字母顺序先于\$header。如果是这样，则：

- 1、创建一个新元素，下标为该新单词，其值为原第一个单词。
- 2、该新单词赋给\$header。



如果该新单词不该为第一个元素，则40~44行利用局域变量\$pointer寻找其合适的有效位置，41~44行循环到\$wordlist{\$pointer}大于或等于\$word为止。接下来46行查看该单词是否已在链表中，如果在就返回，否则47~48行将其添加到链表中。首先47行创建新元素\$wordlist{\$word}，其值为\$wordlist{\$pointer}，这时\$wordlist{\$word}和\$wordlist{\$pointer}指向同一个单词。然后，48行将\$wordlist{\$pointer}的值赋为\$word，即将\$wordlist{\$pointer}指向刚创建的新元素\$wordlist{\$word}。最后当处理完毕后，子程序print\_list()依次输出链表，局域变量\$pointer含有正在输出的值，\$wordlist{\$pointer}为下一个要输出的值。

注：一般不需要用链表来做这些工作，用sort()和keys()在关联数组中循环就足够了，如：

```
foreach $word (sort keys(%wordlist)) {  
# print the sorted list, or whatever }
```

但是，这里涉及的指针的概念在其它数据结构中很有意义。



## 2、结构

许多编程语言可以定义结构(structure)，即一组数据的集合。

结构中的每个元素有其自己的名字，并通过该名字来访问。

Perl 不直接提供结构这种数据结构，但可以用关联数组来模拟。例如模拟C语言中如下的结构：

```

struce{
int field 1 ;
int field 2 ;
int field3; } mystructvar;

```

我们要做的是定义一个含有三个元素的关联数组，下标分别为field1、field2、field3，如：

```

%mystructvar = ( "field1", "",
"field2", "",
"field3", "", );

```

像上面C语言的定义一样，这个关联数组%mystrctvar 有三个元素，下标分别为field1、field2、field3，各元素初始值均为空串。

对各元素的访问和赋值通过指定下标来进行，如：

```

$mystructvar{"field1"} = 17;

```



## 3、树

另一个经常使用的数据结构是树。树与链表类似，但每个节点指向的元素多于一个。最简单的树是二叉树，每个节点指向另外两个元素，称为左子节点和右子节点（或称孩子），每个子节点又指向两个孙子节点，依此类推。

注：此处所说的树像上述链表一样是单向的，每个节点指向其子节点，但子节点并不指向父节点。

树的概念可以如下描述：

- 因为每个子节点均为一个树，所以左/右子节点也称为左/右子树。（有时称左/右分支）
- 第一个节点（不是任何节点的子节点的节点）称为树的根。
- 没有孩子（子节点）的节点称为叶节点。

有多种使用关联数组实现树结构的方法，最好的一种应该是：给予节点分别加上left和right以访问之。例如，

alphaleft和alpharight指向alpha的左右子节点。下面是用此方法创建二叉树并遍历的例程：

```
1 : #!/usr/local/bin/perl
2 :
3 : $rootname = "parent";
4 : %tree = ("parentleft", "child1",
5 :         "parentright", "child2",
6 :         "child1left", "grandchild1",
7 :         "child1right", "grandchild2",
8 :         "child2left", "grandchild3",
9 :         "child2right", "grandchild4");
10: # traverse tree, printing its elements
11: &print_tree($rootname);
12:
13: sub print_tree {
14:     local ($nodename) = @_;
15:     local ($leftchildname, $rightchildname);
```



```
16:
17:   $leftchildname = $nodename . "left";
18:   $rightchildname = $nodename . "right";
19:   if ($tree{$leftchildname} ne "") {
20:     &print_tree($tree{$leftchildname});
21:   }
22:   print (" $nodename\n");
23:   if ($tree{$rightchildname} ne "") {
24:     &print_tree($tree{$rightchildname});
25:   }
26: }
```



结果输出如下：

grandchild1

child1

grandchild2

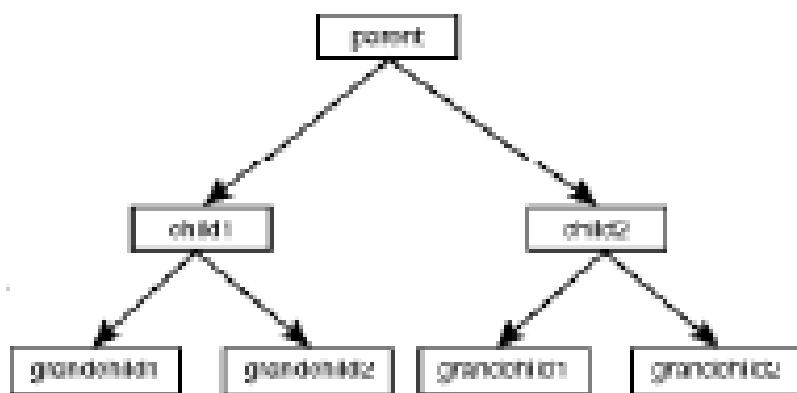
parent

grandchild3

child2

grandchild4

该程



注意函数`print_tree()`以次序“左子树、节点、右子树”来输出各节点的名字，这种遍历次序称为“左序遍历”。如果把第22行移到19行前，先输出节点明，再输出左子树、右子树，则为“中序遍历”，如果把第22行移到25行后，输出次序为左子树、右子树、节点，则为“右序遍历”。可以用同样的方法，即连接字符串构成下标，来创建其它的数据结构，如数据库等。

