

第九章，数据结构

- ↓ 第九章，数据结构
 - ↓ 9.1 数组的数组
 - ↓ 9.1.1 创建和访问一个两维数组
 - ↓ 9.1.2 自行生长
 - ↓ 9.1.3 访问和打印
 - ↓ 9.1.4 片段
 - ↓ 9.1.5 常见错误
 - ↓ 9.2 数组的散列
 - ↓ 9.2.1 数组的散列的组成
 - ↓ 9.2.2 生成数组的散列
 - ↓ 9.2.3 访问和打印数组的散列
 - ↓ 9.3 散列的数组
 - ↓ 9.3.1 组成一个散列的数组
 - ↓ 9.3.2 生成散列的数组
 - ↓ 9.3.3 访问和打印散列的数组
 - ↓ 9.4 散列的散列
 - ↓ 9.4.1 构成一个散列的散列
 - ↓ 9.4.2 生成散列的散列
 - ↓ 9.4.3 访问和打印散列的散列
 - ↓ 9.5 函数的散列
 - ↓ 9.6 更灵活的记录
 - ↓ 9.6.1 更灵活的记录的组合，访问和打印
 - ↓ 9.6.2 甚至更灵活的记录的组合，访问和打印
 - ↓ 9.6.3 复杂记录散列的生成
 - ↓ 9.7 保存数据结构

Perl 免费提供许多数据结构，这些数据结构在其他编程语言里是需要你自己制作的。比如那些计算机科学的新芽们都需要学习的堆栈和队列在 Perl 里都只是数组。在你 `push` 和 `pop`（或者 `shift` 和 `unshift`）一个数组的时候，它就是一个堆栈；在你 `push` 和 `shift`（或者 `unshift` 和 `pop`）一个数组的时候，它就是一个队列。并且世界上有许多树结构的作用只是为了给一些概念上是平面的搜索表文件提供快速动态的访问。当然，散列是内建于 Perl 的，可以给概念上是平面的搜索表提供快速动态的访问，只有对编号不敏感的递归数据结构才会被那些脑袋已经相当程度编了号的人称为美人。

但是有时候你需要嵌套的数据结构，因为这样的数据结构可以更自然地给你要解决的问题建模。因为 Perl 允许你组合和嵌套数组和散列以创建任意复杂的数据结构。经过合理地组合，它们可以用来创建链表，二叉树，堆，B-tree（平衡树），集，图和任何你设计的东西。参阅 *Mastering Algorithms with Perl* (O'Reilly, 1999)，*Perl Cookbook* (O'Reilly 1998)，或者 CPAN——所有这些模块的中心仓库。不过，你需要的所有东西可能就是简单地组合数组和散列，所以我们就在本章介绍这些内容。

9.1 数组的数组

有许多种类型的嵌套数据结构。最容易做的是制作一个数组的数组，也叫做两维数组或者矩阵。（明显的总结是：一个数组的数组的数组就是一个三维数组，对于更高的维数以此类推。）多维数组比较容易理解，并且几乎所有它适用的东西都适用于我们在随后各节里将要讲述的其他更奇特的数据结构。

9.1.1 创建和访问一个两维数组

下面是如何把一个两维数组放在一起的方法：

```
# 给一个数组赋予一个数组引用列表。  
@Aoa = (
```

```

    ["fred", "barney" ],
    ["george", "jane", "elroy" ],
    ["homer", "marge", "bart" ],
);

print $AoA[2][1];      # 打印 "marge"

```

整个列表都封装在圆括弧里，而不是花括弧里，因为你在给一个列表赋值而不是给引用赋值。如果你想要一个指向数组的引用，那么你要使用方括弧：

```

# 创建一个指向一个数组的数组的引用。
$ref_to_AoA = [
[ "fred", "barney", "pebbles", "bamm bamm", "dino", ],
[ "homer", "bart", "marge", "maggie", ],
[ "george", "jane", "elroy", "judy", ],
];

print $ref_to_AoA->[2][3];      # 打印 "judy"

```

请记住在每一对相邻的花括弧或方括弧之间有一个隐含的 `->`。因此下面两行：

```

$AoA[2][3]
$ref_to_AoA->[2][3]

```

等效于下面两行：

```

$AoA[2]->[3]
$ref_to_AoA->[2][3]

```

不过，在第一对方括弧前面没有隐含的 `->`，这也是为什么 `$ref_to_AoA` 的解引用要求开头的 `->`。还有就是记住你可以用负数索引从一个数组后面面向前面计数，因此：

```

$AoA[0][-2]

```

是第一行的倒数第二个元素。

9.1.2 自行生长

大的列表赋值是创建大的固定数据结构的好方法，但是如果你想运行时计算每个元素，或者是一块一块地制作这些结构的时候该怎么办呢？

让我们从一个文件里读入一个数据结构。我们将假设它是一个简单平面文本文件，它的每一行是结构的一个行，并且每行包含由空白分隔的元素。下面是处理的方法：（注：在这里和其他章节一样，我们忽略那些通常你要放进去的 `my` 声明。在这个例子里，你通常写 `my @tmp = split`。）

```

while (<>) {
    @tmp = split;          # 把元素分裂成一个数组
    push @AoA, [ @tmp ];   # 向 @AoA 中增加一个匿名数组引用
}

```

当然，你不必命名那个临时的数组，因此你还可以说：

```

while(<>){
    push @AoA, [ split ];
}

```

如果你想要一个指向一个数组的数组的引用，你可以这么做：

```
while (<>){
    push @ref_to_AoA, [ split ];
}
```

这些例子都向数组的数组增加新的行。那么如何增加新的列呢？如果你只是对付二维数组，通常更简单的方法是使用简单的赋值：（注：和前面的临时赋值一样，我们在这里已经简化了；在这一章的循环在实际的代码中应该写做 `my $x`。）

```
for $x (0..9){
    # 对每一行...
    for $y (0..9) {
        # 对每一列...
        $AoA[$x][$y] = func($x, $y); # ...设置调用
    }
}

for $x (0..9) {
    # 对每一行...
    $ref_to_AoA->[$x][3] = func2($x); # ...设置第四行
}
```

至于你给元素赋值的顺序如何则没有什么关系，而且 `@AoA` 的脚标元素是否存在也没有什么关系；Perl 会很开心地为你创建它们，并且把中间的元素根据需要设置为未定义值。（如果有必要，Perl 甚至会在 `$ref_to_AoA` 中创建最初的引用。）如果你只是想附加一行，你就得做得更奇妙一些：

```
# 向一个已经存在的行中附加一个新列
push @{$AoA[0]}, "wilma", "betty";
```

请注意下面这些无法运转：

```
push $AoA[0], "wilma", "betty"; # 错误！
```

上面的东西甚至连编译都过不了，因为给 `push` 的参数必须是一个真正的数组，而不只是一个指向一个数组的引用。因此，第一个参数绝对必须以 `@` 字符开头。而跟在 `@` 后面的东西则可以忽略一些。

9.1.3 访问和打印

现在把数据结构打印出来。如果你只想要一个元素，下面的就足够了：

```
print $AoA[3][2];
```

但是如果你想打印所有的东西，那你不能这么写：

```
print @AoA; # 错误！
```

这么做是错误的，因为它会打印出字串化的引用，而不是你的数据。Perl 从来不会自动给你解引用。你必须自己用一两个循环遍历你的数据。下面的代码打印整个结构，循环遍历 `@AoA` 的元素并且在 `print` 语句里对每个元素进行解引用：

```
for $row (@AoA) {
    print "@$row\n";
}
```

如果你想追踪脚标，你可以这么做：

```
for $i (0..$#AoA) {
    print "row $i is: @{$AoA[$i]}\n";
}
```

或者甚至可以是下面这样：

```
for $i (0..$#AoA){
    for $j (0..${$AoA[$i]}){
        print "element $i $j is $AoA[$i][$j]\n";
    }
}
```

就象你看到的那样，这里的程序有点复杂。这就是为什么很多时候你用一个临时变量事情会变得更简单：

```
for $i (0..$#AoA){
    $row = $AoA[$i];
    for $j (0..${$row}){
        print "element $i $j is $row->[$j]\n";
    }
}
```

9.1.4 片段

如果你想访问一个多维数组的某个片段（一行的一部分），你就是在准备做一些奇特的脚标处理。指针箭头赋予我们一种访问单一变量的极好的方法，但是对于片段而言却没有这么好的便利方法。当然，你总是可以用一个循环把你的片段一个一个地取出来：

```
@part = ();
for ($y = 7; $y < 13; $y++) {
    push @part, $AoA[4][$y];
}
```

这个循环可以用一个数组片段代替：

```
@part = @{$AoA[4]} [7..12];
```

如果你想要一个两维的片段，比如 **\$x** 在 4..8 而 **\$y** 是 7..12，下面是实现的一些方法：

```
@newAoA = ();
for ($startx = $x = 4; $x <= 8; $x++) {
    for ($starty = $y=7; $y <= 12; $y++) {
        $newAoA[$x - $startx][$y - $starty] = $AoA[$x][$y];
    }
}
```

在这个例子里，我们的两维数组 **@newAoA** 里的每个独立的数值都是一个一个地从一个两维数组 **@AoA** 中取出来赋值的。另外一个方法是创建一个匿名数组，每个由一个 **@AoA** 中我们要的子数组组成，然后然后把指向这些匿名数组的引用放到 **@newAoA** 中。然后我们就可以把引用写到 **@newAoA**（也是脚标，只是这么说而已），而不用把一个子数组值写到两维数组 **@newAoA** 中。这个这个方法消除了内层的循环：

```
for ($x = 4; $x <= 9; $x++) {
    push @newAoA, [ @{$AoA[$x]} [ 7..12] ];
}
```

当然，如果你经常这么做，那么你可能就应该写一个类似 **extract_rectangle** 这样的子过程。而如果你经常对大的多维数组做这样的处理，那么你可能要使用 **PDL**（Perl Data Language）模块，你可以在 **CPAN** 找到。

9.1.5 常见错误

正如我们早先提到过的那样，Perl 数组和散列都是一维的。在 Perl 里，甚至“多维”数组实际上都是一维的，但是该维的数值实际上是其他数组的引用，这样就把许多元素压缩成了一个。

如果你不首先解引用就把这些打印出来，那么你看到的就是字串化的引用而不是你需要的数字。比如，下面两行：

```
@AoA = ([2, 3], [4, 5, 7], [0] );
print "@AoA";
```

结果是象下面这样的东西：

```
ARRAY(0x83c38) ARRAY(0x8b194) ARRAY(0x8b1d0)
```

但是，下面这行显示 7：

```
print $AoA[1][2];
```

在构造一个数组的数组的时候，要记得为子数组构造新的引用。否则，你就只创建了一个包含子数组元素计数的数组，象这样：

```
for $i (1..10) {
    @array = somefunc($i);
    $AoA = @array;          # 错误!
}
```

在这里 `@array` 是在一个标量环境里访问的，因此生成它的元素的计数，然后这个计数被忠实地赋予 `$AoA[$i]`。赋予引用的正确方法我们将在稍后介绍。

在产生前面的错误之后，人们认识到他们需要赋一个引用值，因此人们随后很自然会犯的的错误包括把引用不停地放到同一片内存位置：

```
for $i (1..10) {
    @array = somefunc($i);
    $AoA[$i] = \@array;      # 又错了!
}
```

每个 `for` 循环的第二行生成的引用都是一样的，也就是说，一个指向同一个数组 `@array` 的引用。的确，这个数组在循环的每个回合中都会变化，但是当所有的话都说完了，所有的事都做完了之后，`$AoA` 就包含 10 个指向同一数组的引用，这个时候它保存给它的最后一次赋值的数值。 `print @{$AoA[1]}` 将检索和 `print @{$AoA[2]}` 一样的数值。

下面是更成功的方法：

```
for $i (1..10) {
    @array = somefunc($i);
    $AoA[$i] = [ @array ];    # 正确!
}
```

在 `@array` 周围的方括弧创建一个新的匿名数组，`@array` 里的元素都将拷贝到这里。然后我们就把一个指向它的引用放到这个新的数组里。

一个类似的结果——不过更难读一些——可以是：

```
for $i (1..10) { @array = somefunc($i); @{$AoA[$i]} = @array; }
```

因为 `$AoA` 必须是一个新引用，所以该引用自动生成。然后前导的 `@` 把这个新引用解引用，结果是 `@array` 的数值赋予了（在列表环境中）`$AoA[$i]` 引用的数组。出于程序清晰性的考虑，你可以避免这种写法。

但是有一种情况下你可能要用这种构造。假设 `$AoA` 已经是一个指向数组的引用的数组。也就是说你要做类似下面这样的赋值：

```
= $AoA[3] = \@original_array;=
```

然后我们再假设你要修改 `@original_array`（也就是要修改 `$AoA` 的第四行）这样它就指向 `@array` 的元素。那么下面的代码可以用：

```
= @{$AoA[3]} = @array;=
```

在这个例子里，引用本身并不变化，但是被引用数组的元素会变化。这样就覆盖了 `@original_array` 的数值。

最后，下面的这些看起来很危险的代码将跑得很好：

```
for $i (1..10) {
    my @array = somefunc($i);
    $AoA[$i] = \@array;
}
```

这是因为在循环的每个回合中，词法范围的 `my @array` 都会重新创建。因此即使看起来好象你每次存储的都是相同的变量的引用，但实际上却不是。这里的区别是非常微小的，但是这样的技巧却可以生成更高效的代码，付出的代价是可能有误导稍微差一些的程序员。（更高效是因为它没有最后赋值中的拷贝。）另一方面，如果你必须拷贝这些数值（也就是循环中第一个赋值干的事情），那么你也可以使用方括号造成的隐含拷贝，因而省略临时变量：

```
for $i (1..10) {
    $AoA[$i] = [ somefunc($i) ];
}
```

概括来说：

```
$AoA[$i] = [ @array ];      # 最安全，有时候最快
$AoA[$i] = \@array;        # 快速但危险，取决于数组的自有性
@{ $AoA[$i] } = @array;    # 有点危险
```

一旦你掌握了数组的数组，你就可以处理更复杂的数据结构。如果你需要 **C** 结构或者 **Pascal** 记录，那你在 **Perl** 里找不到任何特殊的保留字为你设置这些东西。**Perl** 给你的是更灵活的系统。如果你对记录结构的观念比这样的系统的灵活性差，或者你宁愿给你的用户一些更不透明的，更僵化的东西，那么你可以使用在第十二章，对象，里详细描述的面向对象的特性。

Perl 只有两种组织数据的方式：以排序的列表存储在数组里按位置访问，或者以未排序的键字/数值对存储在散列里按照名字访问。在 **Perl** 里代表一条记录的最好的方法是用一个散列引用，但是你所选择的组织这样的记录的方法是可以变化的。你可能想要保存一个有序的记录列表以便按照编号来访问，这种情况下你不得不用一个散列数组来保存记录。或者，你可能希望按照名字来寻找记录，这种情况下你就要维护一个散列的散列。你甚至可以两个同时用，这时候就是伪散列。

在随后的各节里，你会看到详细地讲述如何构造（从零开始），生成（从其他数据源），访问，和显示各种不同的数据结构的代码。我们首先演示三种直截了当的数组和散列的组合，然后跟着一个散列函数和更不规则的数据结构。最后我们以一个如何保存这些数据结构的例子结束。我们在讲这些例子之前假设你已经熟悉了我们在本章中前面已经设置的解释集。

9.2 数组的散列

如果你想用一个特定的字串找出每个数组，而不是用一个索引数字找出它们来，那么你就需要用数组的散列。在我们电视角色的例子里，我们不是用第零个，第一个等等这样的方法查找该名字列表，而是设置成我们可以通过给出角名字找出演员列表的方法。

因为我们外层的数据结构是一个散列，因此我们无法对其内容进行排序，但是我们可以使用 `sort` 函数声明一个特定的输出顺序。

9.2.1 数组的散列的组成

你可以用下面的方法创建一个匿名数组的散列：

```
# 如果键字是标识符，我们通常省略引号
%HoA = (
    flintstones => [ "fred", "barney" ],
    jetsons    => [ "george", "jane", "elroy" ],
    simpsons   => [ "homer", "marge", "bart" ],
);
```

要向散列增加另外一个数组，你可以简单地说：

```
$HoA{teletubbies} = [ "tinky winky", "dipsy", "laa-laa", "po" ];
```

9.2.2 生成数组的散列

下面是填充一个数组的散列的技巧。从下面格式的文件中读取出来：

```
flintstones:  fred barney wilma dino
jetsons:      george jane elroy
simpsons:     homer marge bart
```

你可以用下列两个循环之一：

```
while( <> ) {
    next unless s/^(.*?):\S*//;
    $HoA{$1} = [ split ];
}

while ( $line = <> ) {
    ($who, $rest) = split /\:\S*/, $line, 2;
    @fields = split ' ', $rest;
    $HoA{$who} = [ @fields ];
}
```

如果你有一个子过程叫 `get_family`，它返回一个数组，那么你可以用下面两个循环之一填充 `%HoA`：

```
for $group ( "simpsons", "jetsons", "flintstones" ) {
    $HoA{$group} = [ get_family($group) ];
}

for $group ( "simpsons", "jetsons", "flintstones" ) {
    @members = get_family($group);
    $HoA{$group} = [ @members ];
}
```


你可以用下面的方法向一个已存在的数组追加新成员：

```
push @{ $HoA{flintstones}}, "wilma", "pebbles";
```

9.2.3 访问和打印数组的散列

你可以用下面的方法设置某一数组的第一个元素：

```
$HoA{flintstones}[0] = "Fred";
```

要让第二个 **Simpson** 变成大写，那么对合适的数组元素进行一次替换：

```
$HoA{simpsons}[1] =~ s/(\w)/\u$1/;
```

你可以打印所有这些家族，方法是遍历该散列的所有键字：

```
for $family ( keys %HoA ) {
    print "$family: @{ $HoA{$family} }\n";
}
```

我们稍微多做一些努力，你就可以一样追加数组索引：

```
for $family ( keys %HoA ) {
    print "$family: ";
    for $i ( 0 .. ${ $HoA{$family} }) {
        print " $i = $HoA{$family}[$i]";
    }
    print "\n";
}
```

或者通过以数组拥有的元素个数对它们排序：

```
for $family ( sort { @{$HoA{$b}} <=> @{$HoA{$a}} } keys %HoA ) {
    print "$family: @{ $HoA{$family}}\n";
}
```

或者甚至可以是元素的个数对数组排序然后以元素的 **ASCII** 码顺序进行排序（准确地说是 **utf8** 的顺序）：

打印以成员个数和名字排序的所有内容

```
for $family ( sort { @{$HoA{$b}} <=> @{$HoA{$a}} } keys %HoA ) {
    print "$family: ", join(", ", sort @{$HoA{$family}}), "\n";
}
```

9.3 散列的数组

如果你有一堆记录，你想顺序访问它们，并且每条记录本身包含一个键字/数值对，那么散列的数组就很有用。在本章中，散列的数组比其他结构用得少一些。

9.3.1 组成一个散列的数组

你可以用下面方法创建一个匿名散列的数组：

```
@AoH = (
{
    husband => "barney",
    wife    => "betty",
```



```

        son      => "bamm bamm",
    },
    {
        husband => "george",
        wife     => "jane",
        son      => "elroy",
    },
    {
        husband => "homer",
        wife     => "marge",
        son      => "bart",
    },
);

```

要向数组中增加另外一个散列，你可以简单地说：

```
push @AoH, { husband => "fred", wife => "wilma", daughter => "pebbles" };
```

9.3.2 生成散列的数组

下面是一些填充散列数组的技巧。要从一个文件中读取下面的格式：

```
husband=fred friend=barney
```

你可以使用下面两个循环之一：

```

while (<>) {
    $rec = {};
    for $field ( split ) {
        ($key, $value) = split /=/, $field;
        $rec->{$key} = $value;
    }
    push @AoH, $rec;
}

while (<>) {
    push @AoH, { split /\s=/+ };
}

```

如果你有一个子过程 `get_next_pair` 返回一个键字/数值对，那么你可以用它利用下面两个循环之一来填充 `@AoH`：

```

while ( @fields = get_next_pair() ) {
    push @AoH, {@fields};
}

while (<>) {
    push @AoH, { get_next_pair($_) };
}

```

你可以象下面这样向一个现存的散列附加新的成员：

```

$AoH[0]{pet} = "dino";
$AoH[2]{pet} = "santa's little helper";

```

9.3.3 访问和打印散列的数组

你可以用下面的方法设置一个特定散列的数值/键字对：

```
$AoH[0]{husband} = "fred";
```

要把第二个数组的丈夫（husband）变成大写，用一个替换：

```
$AoH[1]{husband} =~ s/(\w)/\u$1/;
```

你可以用下面的方法打印所有的数据：

```
for $href ( @AoH ) {
    print "{ ";
    for $role ( keys %$href ) {
        print "$role=$href->{$role} ";
    }
    print "}\n";
}
```

以及带着引用打印：

```
for $i ( 0 .. $#AoH ) {
    print "$i is { ";
    for $role ( keys %{ $AoH[$i] } ) {
        print "$role=$AoH[$i]{$role} ";
    }
    print "}\n";
}
```

9.4 散列的散列

多维的散列是 Perl 里面最灵活的嵌套结构。它就好像绑定一个记录，该记录本身包含其他记录。在每个层次上，你都用一个字串（必要时引起）做该散列的索引。不过，你要记住散列里的键字/数值对不会以任何特定的顺序出现；你可以使用 **sort** 函数以你喜欢的任何顺序检索这些配对。

9.4.1 构成一个散列的散列

你可以用下面方法创建一个匿名散列的散列：

```
%HoH = (
    flintstones => {
        husband => "fred",
        pal      => "barney",
    },
    jetsons => {
        husband => "george",
        wife     => "jane",
        "his boy" => "elroy",      # 键字需要引号
    },
    simpsons => {
        husband => "homer",
        wife     => "marge",
        kid      => "bart",
    },
);
```

要向 %HoH 中增加另外一个匿名散列，你可以简单地说：

```
$HoH{ mash } = {
    captain => "pierce",
    major   => "burns",
    corporal=> "radar",
}
```

9.4.2 生成散列的散列

下面是一些填充一个散列的散列的技巧。要从一个下面格式的文件里读取数据：

flintstones

husband=fred pal=barney wife=wilma pet=dino

你可以使用下面两个循环之一：

```
while( <> ){
    next unless s/^(.*?):\S*//;
    $who = $1;
    for $field ( split ) {
        ($key, $value) = split /=/, $field;
        $HoH{$who}{$key} = $value;
    }
}
```

```
while( <> ){
    next unless s/^(.*?):\S*//;
    $who = $1;
    $rec = {};
    $HoH{$who} = $rec;
    for $field ( split ) {
        ($key, $value) = split /=/, $field;
        $rec->{$key} = $value;
    }
}
```

如果你有一个子过程 `get_family` 返回一个键字/数值列表对，那么你可以拿下面三种方法的任何一种，用它填充 %HoH：

```
for $group ("simpsons", "jetsons", "flintstones" ) {
    $HoH{$group} = {get_family($group)};
}

for $group ( "simpsons", "jetsons", "flintstones" ) {
    @members = get_family($group);
    $HoH{$group} = {@members};
}

sub hash_families {
    my @ret;
    for $group (@_) {
        push @ret, $group, {get_family($group)};
    }
    return @ret;
}
```

```
}
```

```
%HoH = hash_families( "simpsons", "jetsons", "flintstones" );
```

你可以用下面的方法向一个现有的散列附加新的成员：

```
%new_floks = (
wife => "wilma",
pet  => "dino",
);

for $what (keys %new_floks) {
    $HoH{flintstones}{$what} = $new_floks{$what};
}
```

9.4.3 访问和打印散列的散列

你可以用下面的方法设置键字/数值对：

```
$HoH{flintstones}{wife} = "wilma";
```

要把某个键字/数值对变成大写，对该元素应用一个替换：

```
$HoH{jetsons}{'his boy'} =~ s/(\w)/\u$1/;
```

你可以用先后遍历内外层散列键字的方法打印所有家族：

```
for $family ( keys %HoH ) {
    print "$family: ";
    for $role ( keys %{ $HoH{$family} } ) {
        print "$role=$person ";
    }
    print "\n";
}
```

在非常大的散列里，可能用 **each** 同时把键字和数值都检索出来会略微快一些（这样做可以避免排序）：

```
while ( ($family, $roles) = each %HoH ) {
    print "$family: ";
    while ( ($role, $person) = each %$roles ) {
        print "$role=$person";
    }
    print "\n";
}
```

（糟糕的是，需要存储的是那个大的散列，否则你在打印输出里就永远找不到你要的东西。）你可以用下面的方法先对家族排序然后再对脚色排序：

```
for $family ( sort keys %HoH ) {
    print "$family: ";
    for $role ( sort keys %{ $HoH{$family} } ) {
        print "$role=$HoH{$family}{$role} ";
    }
    print "\n";
}
```

要按照家族的编号排序（而不是 **ASCII** 码（或者 **utf8** 码）），你可以在一个标量环境里使用

keys:

```
for $family ( sort { keys %{$HoH{$a}} <=> keys %{$HoH{$b}}} keys %HoH ) {
    print "$family: ";
    for $role ( sort keys %{$HoH{$family}} ) {
        print "$role=$HoH{$family}{$role}";
    }
    print "\n";
}
```

要以某种固定的顺序对一个家族进行排序，你可以给每个成员赋予一个等级来实现：

```
$i = 0;
for ( qw(husband wife son daughter pal pet) ) { $rank{$_} = ++$i }

for $family ( sort { keys %{$HoH{$a}} <=> keys %{$HoH{$b}}} keys %HoH ) {
    print "$family: ";
    for $role ( sort { $rank{$a} <=> $rank{$b} } keys %{$HoH{$family}} ) {
        print "$role=$HoH{$family}{$role} ";
    }
    print "\n";
}
```

9.5 函数的散列

在使用 Perl 书写一个复杂的应用或者网络服务的时候，你可能需要给你的用户制作一大堆命令供他们使用。这样的程序可能有象下面这样的代码来检查用户的选择，然后采取相应的动作：

```
if ($cmd =~ /^exit$/i) { exit }
elsif ($cmd =~ /^help$/i) { show_help() }
elsif ($cmd =~ /^watch$/i) { $watch = 1 }
elsif ($cmd =~ /^mail$/i) { mail_msg($msg) }
elsif ($cmd =~ /^edit$/i) { $edited++; editmsg($msg); }
elsif ($cmd =~ /^delete$/i) { confirm_kill() }
else {
    warn "Unknown command: '$cmd'; Try 'help' next time\n";
}
```

你还可以在你的数据结构里保存指向函数的引用，就象你可以存储指向数组或者散列的引用一样：

```
%HoF = (
    exit    => sub { exit },
    help    => \&show_help,
    watch   => sub { $watch = 1 },
    mail    => sub { mail_msg($msg) },
    edit    => sub { $edited++; editmsg($msg); },
    delete => \&confirm_kill,
);

if ($HoF{lc $cmd}) { $HoF{lc $cmd}->() } # Call function
else { warn "Unknown command: '$cmd'; Try 'help' next time\n" }
```

在倒数第二行里，我们检查了声明的命令名字（小写）是否在我们的“遣送表”%HoF 里存在。如果是，我们调用响应的命令，方法是把散列值当作一个函数进行解引用并且给该函数传递一个空的参数列表。我们也可以用 `&{ $HoF{lc $cmd} }()` 对散列值进行解引用，或者，在 Perl 5.6 里，可以简单地是 `$HoF{lc $cmd}()`。

9.6 更灵活的记录

到目前为止，我们在本章看到的都是简单的，两层的，同质的数据结构：每个元素包含同样类型的引用，同时所有其他元素都在该层。数据结构当然可以不是这样的。任何元素都可以保存任意类型的标量，这就意味着它可以是一个字串，一个数字，或者指向任何东西的引用。这个引用可以是一个数组或者散列引用，或者一个伪散列，或者是一个指向命名或者匿名函数的引用，或者一个对象。你唯一不能干的事情就是向一个标量里填充多个引用物。如果你发现自己在做这种尝试，那就表示着你需要一个数组或者散列引用把多个数值压缩成一个。

在随后的节里，你将看到一些代码的例子，这些代码设计成可以演示许多你想存储在一个记录里的许多可能类型的数据，我们将用散列引用来实现它们。这些键字都是大写字串，这是我们时常使用的一个习惯（有时候也不用这个习惯，但只是偶然不用）——如果该散列被用做一个特定的记录类型。

9.6.1 更灵活的记录的组合，访问和打印

下面是一个带有六种完全不同的域的记录：

```
$rec = {
    TEXT      => $string,
    SEQUENCE  => [ @old_values ],
    LOOKUP    => { %some_table },
    THATCODE  => sub { $_[0] ** $_[1] },
    HANDLE    => \*STDOUT,
};
```

TEXT 域是一个简单的字串。因此你可以简单的打印它：

```
print $rec->{TEXT};
```

SEQUENCE 和 LOOKUP 都是普通的数组和散列引用：

```
print $rec->{SEQUENCE}[0];
$last = pop @{$rec->{SEQUENCE}};

print $rec->{LOOKUP}{"key"};
($first_k, $first_v) = each %{$rec->{LOOKUP}};
```

THATCODE 是一个命名子过程而 THISCODE 是一个匿名子过程，但是它们的调用是一样的：

```
$that_answer = $rec->{THATCODE}->($arg1, $arg2);
$this_answer = $rec->{THISCODE}->($arg1, $arg2);
```

再加上一对花括弧，你可以把 \$rec->{HANDLE} 看作一个间接的对象：

```
print { $rec->{HANDLE} } "a string \n";
```

如果你在使用 [FileHandle²](#) 模块，你甚至可以把该句柄看作一个普通的对象：

```
use FileHandle;
$rec->{HANDLE}->autoflush(1);
$rec->{HANDLE}->print("a string\n");
```

9.6.2 甚至更灵活的记录的组合，访问和打印

自然，你的数据结构的域本身也可以是任意复杂的数据结构：

```

%TV = (
flintstones => {
    series    => "flintstones",
    nights    => [ "monday", "thursday", "friday" ],
    members   => [
        { name => "fred",    role => "husband", age => 36, },
        { name => "wilma",   role => "wife",    age => 31, },
        { name => "pebbles", role => "kid",     age => 4,  },
    ],
},

jetsons      => {
    series    => "jetsons",
    nights    => [ "wednesday", "saturday" ],
    members   => [
        { name => "george",  role => "husband", age => 41, },
        { name => "jane",    role => "wife",    age => 39, },
        { name => "elroy",   role => "kid",     age => 9,  },
    ],
},

simpsons     => {
    series    => "simpsons",
    nights    => [ "monday" ],
    members   => [
        { name => "homer",  role => "husband", age => 34, },
        { name => "marge",  role => "wife",    age => 37, },
        { name => "bart",   role => "kid",     age => 11, },
    ],
},
);

```

9.6.3 复杂记录散列的生成

因为 Perl 分析复杂数据结构相当不错，因此你可以把你的数据声明作为 Perl 代码放到一个独立的文件里，然后用 `do` 或者 `require` 等内建的函数把它们装载进来。另外一种流行的方法是使用 CPAN 模块（比如 `XML::Parser`）装载那些用其他语言（比如 XML）表示的任意数据结构。

你可以分片地制作数据结构：

```

$rec = {};
$rec->{series} = "flintstones";
$rec->{nights} = [ find_days()];

```

或者从文件里把它们读取进来（在这里,我们假设文件的格式是 `field=value` 语法）：

```

@members = ();
while (<>) {
    %fields = split /[\\s=]+/;
    push @members, {%fields};
}
$rec->{members} = [ @members ];

```

然后以一个子域为键字，把它们堆积到更大的数据结构里：


```
$TV{ $rec->{series} } = $rec;
```

你可以使用额外的指针域来避免数据的复制。比如，你可能需要在一个人的记录里包含一个“kids”（孩子）数据域，这个域可能是一个数组，该数组包含着指向这个孩子自己记录的引用。通过把你的数据结构的一部分指向其他的部分，你可以避免因为在一个地方更新数据而没有在其他地方更新数据造成的数据倾斜：

```
for $family (keys %TV) {
    my $rec = $TV{$family};      # 临时指针
    @kids = ();
    for $person ( @{$rec->{members}} ) {
        if ($person->{role} =~ /kid|son|daughter/) {
            push @kids, $person;
        }
    }
    # $rec 和 $TV{$family} 指向相同的数据！
    $rec->{kids} = [@kids];
}
```

这里的 `$rec->{kids} = [@kids]` 赋值拷贝数组内容——但它们只是简单的引用，而没有拷贝数据。这就意味着如果你给 Bart 赋予下面这样的年龄：

```
$TV{simpsons}{kids}[0]{age}++; # 增加到 12
```

那么你就会看到下面的结果，因为 `$TV{simpsons}{kids}[0]` 和 `$TV{simpsons}{members}[2]` 都指向相同的下层匿名散列表：

```
print $TV{simpsons}{members}[2]{age}; # 也打印 12
```

现在你打印整个 `%TV` 结构：

```
for $family ( keys %TV ) {
    print "the $family";
    print " is on ", join ( " and ", @{$TV{$family}{nights}} ), "\n";
    print "its members are:\n";
    for $who ( @{$TV{$family}{members}} ) {
        print " $who->{name} ($who->{role}), age $who->{age}\n";
    }
    print "children: ";
    print join ( ", ", map { $_->{name} } @{$TV{$family}{kids}} );
    print "\n\n";
}
```

9.7 保存数据结构

如果你想保存你的数据结构以便以后用于其他程序，那么你有很多方法可以用。最简单的方法就是使用 Perl 的 `Data::Dumper` 模块，它把一个（可能是自参考的）数据结构变成一个字符串，你可以把这个字符串保存在程序外部，以后用 `eval` 或者 `do` 重新组成：

```
use Data::Dumper;
$Data::Dumper::Purity = 1;      # 因为 %TV 是自参考的
open (FILE, "> tvinfo.perldata") or die "can't open tvinfo: $!";
print FILE Data::Dumper->Dump([\%TV], ['*TV']);
close FILE                      or die "can't close tvinfo: $!";
```

其他的程序（或者同一个程序）可以稍后从文件里把它读回来：

```
open (FILE, "< tvinfo.perldata") or die "can't open tvinfo: $!";
undef $/;                # 一次把整个文件读取进来
eval ;                    # 重新创建 %TV
die "can't recreate tv data from tvinfo.perldata: $" if $@;
close FILE                or die "can't close tvinfo: $!";
print $TV{simpsons}{members}[2]{age};
```

或者简单的是：

```
do "tvinfo.perldata"      or die "can't recreate tvinfo: $! $@";
print $TV{simpsons}{members}[2]{age};
```

还有许多其他的解决方法可以用，它们的存储格式的范围从打包的二进制（非常快）到 XML（互换性非常好）。检查一下靠近你的 CPAN 镜象！

Revision: r1.4 - 26 Aug 2005 - 16:50 - [TingYu](#)

[Perl](#) > [PerlProgramming3](#) > [DataStructures](#)

版权 © 1999-2006 归这里所有作者. [PostgreSQL](#) 的中文文档版权归何伟平所有.
向为这里贡献想法,文章的人致敬 [PostgreSQL](#) 中文网
[反馈意见](#)