1 NAME

perllol - 操作数组的数组（二维数组）

2 说明

声明和访问数组的数组

创建一个数组的数组（有时也可以叫“列表的列表”，不过不太准确）真是再简 单也不过了。它相当容易理解，并且本文中出现的每个例子都有可能在实际应用 中出现。

数组的数组就是一个普通的数组(@AoA)，不过可以接受两个下标($AoA[3][2])。 下面先定义一个这样的数组：

```
# 一个包含有“指向数组的引用”的数组
@AoA = (
    [ "fred", "barney" ],
    [ "george", "jane", "elroy" ],
    [ "homer", "marge", "bart" ],
);

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

你可能已经注意到，外面的括号是圆括号，这是因为我们想要给数组赋值，所以 需要圆括号。如果你不希望这里是 @AoA，而是一个指向它的引用，那么就得 这样：

```
# 一个指向“包含有数组引用的数组”的引用
$ref_to_AoA = [
    [ "fred", "barney", "pebbles", "bambam", "dino", ],
    [ "homer", "bart", "marge", "maggie", ],
    [ "george", "jane", "elroy", "judy", ],
];

print $ref_to_AoA->[2][2];
```

注意外面的括号现在变成了方括号，并且我们的访问语法也有所改变。这时因为 和 C 不同，在 Perl 中你不能自由地交换数组和引用（在 C 中，数组和指针在 很多地方可以互相代替使用）。$ref_to_AoA 是一个数组引用，而 @AoA 是一个 数组。同样地，$AoA[2] 也不是一个数组，而是一个数组引用。所以下面这 两行：

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

也可以用这两行来代替：

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

这是因为这里有两个相邻的括号（不管是方括号还是花括号），所以你可以随意 地省略箭头符号。但是如果 $ref_to_AoA 后面的那个箭头不能省略，因为省略 了就没法知道 $ref_to_AoA 到底是引用还是数组了 ^_^。

修改二维数组

前面的例子里我们创建了包含有固定数据的二维数组，但是如何往其中添加新元 素呢？再或者如何从零开始创建一个二维数组呢？

首先，让我们试着从一个文件中读取二维数组。首先我们演示如何一次性添加一 行。首先我们假设有这样一个文本文件：每一行代表了二维数组的行，而每一个 单词代表了二维数组的一个元素。下面的代码可以把它们储存到 @AoA：

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

你也可以用一个函数来一次读取一行：

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

或者也可以用一个临时变量来中转一下，这样看起来更清楚些：

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

注意方括号 [] 在这里非常重要。方括号实际上是数组引用的构造器。如果不 用方括号而直接写，那就犯了很严重的错误：

```
$AoA[$i] = @tmp;
```

你看，把一个数组赋值给了一个标量，那么其结果只是计算了 @tmp 数组的元素个 数，我想这肯定不是你希望的。

如果你打开了 `use strict`，那么你就得先定义一些变量然后才能避免警告：

```
use strict;
my(@AoA, @tmp);
while (<>) {
    @tmp = split;
    push @AoA, [ @tmp ];
}
```

当然，你也可以不要临时变量：

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

如果你知道想要放在什么地方的话，你也可以不要 push()，而是直接进行赋值：

```
my (@AoA, $i, $line);
for $i ( 0 .. 10 ) {
    $line = <>;
    $AoA[$i] = [ split ' ', $line ];
}
```

甚至是这样：

```
my (@AoA, $i);
for $i ( 0 .. 10 ) {
    $AoA[$i] = [ split ' ', <> ];
}
```

你可能生怕 <> 在列表上下文会出差错，所以想要明确地声明要在标量上下文中 对 <> 求值，这样可读性会更好一些：（译者注：列表上下文中，<> 返回所有的行，标量上下文中 <> 只返回一行。）

```
my (@AoA, $i);
for $i ( 0 .. 10 ) {
    $AoA[$i] = [ split ' ', scalar(<>) ];
}
```

如果你想用 $ref_to_AoA 这样的一个引用来代替数组，那你就得这么写：

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

现在你已经知道如何添加新行了。那么如何添加新列呢？如果你正在做数学中的 矩阵运算，那么要完成类似的任务：

```
for $x (1 .. 10) {
    for $y (1 .. 10) {
        $AoA[$x][$y] = func($x, $y);
    }
}

for $x ( 3, 7, 9 ) {
    $AoA[$x][20] += func2($x);
}
```

想要访问的某个元素是不是存在是无关紧要的：因为如果不存在那么 Perl 会给 你自动创建！新创建的元素的值是 undef。

如果你想添加到一行的末尾，你可以这么做：

```
# 添加新列到已存在的行
push @{ $AoA[0] }, "wilma", "betty";
```

注意我没有这么写：

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

事实上，上面这句根本就没法通过编译！为什么？因为 push() 的第一个参数必 须是一个真实的数组，不能是引用。

访问和打印

现在是打印二维数组的时候了。那么怎么打印？很简单，如果你只想打印一个元 素，那么就这么来一下：

```
print $AoA[0][0];
```

如果你想打印整个数组，那你可不能这样：

```
print @AoA;       # 错误!
```

因为你这么做只能得到一列引用，Perl 从来都不会自动地为你解引用。作为替 代，你必须得弄个循环或者是双重循环。用 shell 风格的 for() 语句就可以 打印整个二维数组：

```
for $aref ( @AoA ) {
    print "\t [ @$aref ],\n";
}
```

如果你要用下标来遍历的话，你得这么做：

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

或者这样用双重循环（注意内循环）：

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

如同你看到的一样，它有点儿复杂。这就是为什么有时候用临时变量能够看起来 更简单一些的原因：

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

哦，好像还有点复杂，那么试试这样：

```
for $i ( 0 .. $#AoA ) {
    $aref = $AoA[$i];
    $n = @$aref - 1;
    for $j ( 0 .. $n ) {
        print "elt $i $j is $AoA[$i][$j]\n";
    }
}
```

切片

切片是指数组的一部分。如果你想要得到多维数组的一个切片，那你得进行一些 下标运算。通过箭头可以方便地为单个元素解引用，但是访问切片就没有这么好 的事了。当然，我们可以通过循环来取切片。

我们先演示如何用循环来获取切片。我们假设 @AoA 变量的值和前面一样。

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

这个循环其实可以用一个切片操作来代替：

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

不过这个看上去似乎略微有些复杂。

下面再教你如何才能得到一个 二维切片，比如 $x 从 4 到 8，$y 从 7 到 12，应该怎么写？

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

也可以省略掉中间的那层循环：

```
for ($x = 4; $x <= 8; $x++) {
    push @newAoA, [ @{ $AoA[$x] } [ 7..12 ] ];
}
```

其实用 map 函数可以更加简练：

```
@newAoA = map { [ @{ $AoA[$_] } [ 7..12 ] ] } 4 .. 8;
```

虽然你的经理也许会抱怨这种难以理解的代码可能会带来安全隐患， 然而这种观点还是颇有争议的（兴许还可以更加安全也说不定 ^_^）。 换了是我，我会把它们放进一个函数中实现：

```
@newAoA = splice_2D( \@AoA, 4 => 8, 7 => 12 );
sub splice_2D {
    my $lrr = shift;       # 指向二维数组的引用
    my ($x_lo, $x_hi,
        $y_lo, $y_hi) = @_;

    return map {
        [ @{ $lrr->[$_] } [ $y_lo .. $y_hi ] ]
    } $x_lo .. $x_hi;
}
```

3 参见

perldata(1), perlref(1), perldsc(1)

4 作者

Tom Christiansen <tchrist@perl.com>
Last update: Thu Jun 4 16:16:23 MDT 1998

5 翻译者及翻译声明

本文由 flw ("flw@cpan.org") 翻译，翻译成果首次出现在 *中国 Perl 协会* http://www.perlchina.org) 的协作开发平台上。

PerlChina.org 本着“在国内推广 Perl” 的目的，组织人员翻译本文。读者可 以在遵守原作者许可协议、尊重原作者及译作者劳动成果的前提下，任意发布或 修改本文。

本文作者用一种轻松地口吻简要介绍了一下二维数组的用法，但是正因如此，文 中有些内容直译过来反而很拗口、很难懂（毕竟中西方文化不同嘛），因此译者 在翻译时，对原文有较多的修改。喜欢阅读英文原版胜过喜欢翻译版的朋友们， 可以直接看原版 ^_^，同时也希望能够理解译者的一篇苦心，而不要在背后骂我 翻译得不对就是了。

希望本文能对英文不好的朋友们有所帮助。

如果你对本文有任何意见，欢迎来信指教。本人非常欢迎与各位交流。