

第三章 单目和双目操作符

- ↓ 第三章 单目和双目操作符
 - ↓ 3.1 项和列表操作符（左向）
 - ↓ 3.2 箭头操作符
 - ↓ 3.3 自增和自减操作符
 - ↓ 3.4 指数运算
 - ↓ 3.5 表意单目操作符
 - ↓ 3.6 绑定操作符
 - ↓ 3.7 乘号操作符
 - ↓ 3.8 附加操作符
 - ↓ 3.9 移位操作符
 - ↓ 3.10 命名单目操作符和文件测试操作符
 - ↓ 3.11 关系操作符
 - ↓ 3.12 相等操作符
 - ↓ 3.13 位操作符
 - ↓ 3.14 C 风格的逻辑（短路）操作符
 - ↓ 3.15 范围操作符
 - ↓ 3.16 条件操作符
 - ↓ 3.16 赋值操作符
 - ↓ 3.18 逗号操作符
 - ↓ 3.19 列表操作符（右向）
 - ↓ 3.20 逻辑与，或，非和异或
 - ↓ 3.21 Perl 里没有的 C 操作符

在上面一章里，我们讲了各种你可能在表达式里用到的项，不过老实说，把项隔离出来让人觉得有点无聊。因为许多项都是群居动物。它们相互之间有某种关系。年轻的项急于以各种方式表现自己并影响其它项，而且还存在不同类型的社会关系和许多不同层次的义务。在 Perl 里，这种关系是用操作符来表现的。

社会学必须对某些事物有利。

从数学的角度来看，操作符只是带着特殊语法的普通函数。从语言学的角度来说，操作符只是不规则动词。不过，几乎任何语言都会告诉你，在一种语言里的不规则动词很可能是你最常用的语素。而从信息理论的角度来看，这一点非常重要，因为不规则动词不管是在使用中还是识别上都比较短而且更有效。

从实用角度出发，操作符非常易用。

根据操作符的元数（它们操作数的个数）的不同，它们的优先级（它们从周围的操作符手中夺取操作数的难易）的不同，它们的结合性（当与同优先级的操作符相联时，它们是从左到右处理还是从右到左处理。）的不同，操作符可以分成各种各样类型。

Perl 的操作符有三种元数：单目，双目和三目。单目操作符总是前缀操作符（除自增和自减操作符以外）。（注：你当然可以认为各种各样的引号和括弧是项与项之间分隔的环缀操作符。）其他的都是中缀操作符——除非你把列表操作符也算进来，它可以做任意数量参数的前缀。不过大多数人认为列表操作符只是一种普通的函数，只不过你可以不为它写括弧而已。下面是一些例子：

```
! $x          # 一个单目操作符
$x * $y       # 一个双目操作符
$x ? $y : $z   # 一个三目操作符
print $x, $y, $z # 一个列表操作符
```

操作符的优先级控制它绑定的松紧度。高优先级的操作符先于低优先级的操作符攫取它们周围的参

数。优先级的原理可以直接在基本数学里面找到，在数学里，乘法比加法优先级高：

1. `+ 3 * 4` # 生成14而不是20

两个同等优先级的操作符在一起的时候，它们的执行顺序取决于它们的结合性。这些规则在某种程度上仍然遵循数学习惯：

```
2 * 3 * 4      # 意味着 (2*3)*4，左结合
2 ** 3 ** 4     # 意味着 2**(3**4)，右结合
2 != 3 != 4     # 非法，不能结合
```

表 3-1 列出了从高优先级到低优先级的 Perl 操作符，以及它们的结合性和元数。

表 3-1。操作符优先级

结合性	元数	优先级表
无	0	项，和列表操作符（左侧）
左	2	<code>-></code>
无	1	<code>++ --</code>
右	2	<code>**</code>
右	1	<code>!~&gt;</code> 和单目 <code>+</code> 和 <code>-</code>
左	2	<code>=~ !~</code>
左	2	<code>* / % x</code>
左	2	<code>+ - .</code>
左	2	<code><< > ></code>
右	0,1	命名单目操作符
无	2	<code>< > <= >= lt gt le ge</code>
无	2	<code>= <=> eq ne cmp</code>
左	2	<code>&</code>
左	2	<code> </code>
左	2	<code>&&</code>
左	2	<code> </code>
无	2	<code>.. ...</code>
右	3	<code>?:</code>
右	2	<code>+ += -+ *=</code> 等等
左	2	<code>, =></code>
右	0+	列表操作符（右侧）
右	1	<code>not</code>
左	2	<code>and</code>
左	2	<code>or xor</code>

看起来好象要记很多的优先级级别。不错，的确很多。幸运的是，有两件事情可以帮助你。首先，这里定义的优先级级别通常遵循你的直觉（前提是你没得精神病）。第二，如果你得了精神病，那你总还是可以放上额外的圆括弧以减轻你的疑虑。

另外一个可以帮助你的线索是，任何从 C 里借来的操作符相互之间仍然保留相同的优先级关系，尽管 C 的优先级有点怪。（这就让那些 C 和 C++ 的爱好者，甚至还包括 JAVA 的爱好者学习起 Perl 来会更容易些。）

随后的各节按照优先级顺序讲述这些操作符。只有极少数例外，所有这样的操作符都只处理标量值，而不处理列表值。我们会在轮到它们出现的时候提到这一点。

尽管引用是标量值，但是在引用上使用大多数操作符没有什么意义，因为一个数字值的引用只是在 Perl 内部才有意义。当然，如果一个引用指向一个允许重载的类里的一个对象，你就可以在这样的对象上调用这些操作符，并且如果该类为那种特定操作符定义了重载，那它也会定义那个操作符应该如何处理该对象。比如，复数在 Perl 里就是这么实现的。有关重载的更多内容，请参考第十三章，重载。

3.1 项和列表操作符（左向）

在 Perl 里，项的优先级最高。项包括变量，引起和类似引起的操作符、大多数圆括弧（或者方括弧或大括弧）内的表达式，以及所有其参数被圆括弧包围的函数。实际上，Perl 里没有这种意义上的函数，只有列表操作符和单目操作符会表现得象函数——当你在它们的参数周围放上圆括弧的时候。不管怎样，第二十九章的名称是函数。

现在请注意听了。下面有几条非常重要的规则，它们能大大简化事情的处理，但是如果你粗心地话，偶尔会产生不那么直观的结果。如果有哪个列表操作符（如 `print`）或者哪个命名单目操作符（比如 `chdir`）后面跟着左圆括弧做为下一个记号（忽略空白），那么该操作符和它的用圆括弧包围的参数就获得最高优先级，就好像它是一个普通的函数调用一样。规则是：如果看上去象函数调用，它就是函数调用。你可以把它做得不象函数——在圆括弧前面加一个单目加号操作符即可，（从语意上来说，这个加号什么都没干，它甚至连参数是否数字都不关心）。

例如，因为 `||` 比 `chdir` 的优先级低，我们有：

```
chdir $foo || die;    # (chdir $foo) || die
chdir ($foo) || die;  # (chdir $foo) || die
chdir ($foo) || die;  # (chdir $foo) || die
chdir +($foo) || die; # (chdir $foo) || die
```

不过，因为 `*` 的优先级比 `chdir` 高，我们有：

```
chdir $foo * 20;    # chdir ($foo * 20)
chdir ($foo) * 20;  # (chdir $foo) * 20
chdir ($foo) * 20;  # (chdir $foo) * 20
chdir +($foo) * 20  # chdir ($foo * 20 )
```

这种情况对任何命名单目操作符的数字操作符同样有效，比如 `rand`：

```
rand 10 * 20;          # rand (10 * 20)
rand(10) * 20;         # (rand 10) * 20
rand (10) * 20;        # (rand 10) * 20
rand +(10) * 20;       # rand (10 * 20)
```

当缺少圆括弧时，象 `print`、`sort` 或 `chmod` 这样的列表操作符的优先级要么非常高，要么非常低——取决于你是向操作符左边还是右边看。（这也是为什么我们在本节标题上有“左向”字样的原因。）比如，在：

```
@ary = (1, 3, sort 4, 2);
print @ary;    # 打印1324
```

在 `sort` 右边的逗号先于 `sort` 计算，而在其左边的后其计算。换句话说，一个列表操作符试图收集它后面所有的参数，然后当做一个简单的项和它前面的表达式放在一起。但你还是要注意圆括弧的使用：

```
# 这些在进行print前退出
print($foo, exit); # 显然不是你想要的。
print $foo, exit; # 也不是这个

# 这些在退出前打印:
(print $foo), exit; # 这个是你想要的。
print ($foo), exit; # 或者这个。
print ($foo), exit; # 这个也行。
```

最容易出错的地方是你用圆括弧把数学参数组合起来的时候，但是你却忘记了圆括弧同时用于组合函数参数：

```
print ($foo & 255) + 1, "\n"; # 打印($foo & 255)
(译注：这里 print ($foo & 255) 构成函数，函数是一个项，项的优先级最高，因而先
执行。)
```

这句话可能和你一开始想的结果不同。好在这样的错误通常会生成类似 "Useless use of addition in a void context" 这样的警告——如果你打开了警告。

同样当作项分析的构造还有 `do {}` 和 `eval {}` 以及子过程和方法调用，匿名数组和散列构造符 `[]` 和 `{}`，还有匿名子过程构造符 `{}`。

3.2 箭头操作符

和 C 和 C++ 类似，双目操作符 `->` 是一个中缀解引用操作符。如果右边是一个 `[...]` 数组下标、一个 `{...}` 散列下标、或者一个 `(...)` 子过程参数列表，那么左边必须是一个对应的数组、散列、或者子过程的应用（硬引用或符号引用都行）。在一个左值（可赋值）环境里，如果左边不是一个引用，那它必须是一个能够保存硬引用的位置，这种情况下这种引用会为你自动激活。有关这方面的更多的信息（以及关于故意自激活的一些警告信息），请参阅第八章，引用。

```
$aref->[42] # 一个数组解引用
$href->{"corned beff"} # 一个散列解引用
$sref->(1,2,3) # 一个子过程解引用
```

要不然，它就是某种类型的方法调用。右边必须是一个方法名（或者一个包含该方法名的简单标量变量），而且左边必须得出一个对象名（一个已赐福引用）或者一个类的名字（也就是说，一个包名字）：

```
$yogi = Bear->new("Yogi"); # 一个类方法调用
$yogi->swipe($picnic); # 一个对象方法调用
```

方法名可以用一个包名修饰以标明在哪个包里开始搜索该方法，或者带着特殊包名字， `SUPER::`，以表示搜索应该从父类开始。参阅第十二章，对象。

3.3 自增和自减操作符

`++` 和 `--` 操作符的功能和 C 里面一样。就是说，当把它们放在一个变量前面时，它们在返回变量值之前增加或者减少变量值，当放在变量后面时，它们在返回变量值后再对其加一或减一。比如，

`$a++` 把标量变量 `$a` 的值加一，在它执行增加之前返回它的值。类似地， `--$b{!(\w+)}[0]` 把散列 `%b` 里用缺省的搜索变量 (`$_`) 里的第一个“单词”索引的元素先减一，然后返回。（注：哦，这儿可能有点不公平，因为好多东西你还不知道。我们只是想让你专心。该表达式的工作过程是这样的：首先，模式匹配用表达式 `\w+` 在 `$_` 里找第一个单词。它周围的圆括弧确保此单词作为单元元素列表值返回，因为该模式匹配是在列表环境里进行的。这个列表环境是由列表片段操作符， `(...)[0]` 提供的，它返回列表的第一个（也是唯一一个）元素。该值用做散列的键字，然后散列记录

（值）被判断并返回。通常，如果碰到一个复杂的表达式，你可以从内向外地分析它并找出事情发生的顺序。）

自增操作符有一点额外的内建处理。如果你增加的变量是一个数字，或者该变量在一个数字环境里使用，你得到正常自增的功能。不过，如果该变量从来都是在字串环境里使用，而且值为非空，还匹配模式/`^[a-zA-z]*[0-9]*$/`，这时自增是以字串方式进行的，每个字符都保留在其范围之内，同时还会进位：

```
print ++($foo = '99');    # 打印'100'
print ++($foo = 'a0');    # 打印'a1'
print ++($foo = 'Az');    # 打印'Ba'
print ++($foo = 'zz');    # 打印'aaa'
```

在我们写这些的时候，自增的额外处理还没有扩展到 **Unicode** 字符和数字，不过将来也许会的。

不过自减操作符没有额外处理，我们也没有准备给它增加这个处理。

3.4 指数运算

双目 ****** 是指数操作符。请注意它甚至比单目操作符的绑定更严格，所以 `-2**4` 是 `-(2**4)`，不是 `(-2)**4`。这个操作符是用 **C** 的 `pow(3)` 函数实现的，该函数在内部以浮点数模式运转。它用对数运算进行计算，这就意味着它可以处理小数指数，不过有时候你得到的结果不如直接用乘法得出的准确。

3.5 表意单目操作符

大多数单目操作符只有名字（参阅本章稍后的“命名的单目和文件测试操作符”），不过，有些操作符被认为比较重要，所以赋予它们自己的特殊符号。所有这类操作符好象都和否定操作有关。骂数学家去。

单目 **!** 执行逻辑否，就是说，“**not**”。参阅 **not** 看看一个在优先级中级别较低的逻辑否。如果操作数为假（数字零，字串“0”，空字串或未定义），则对操作数取否，值为真（**1**），若操作数为真，则值为假（“”）。

如果操作数是数字，单目 **-** 执行数学取负。如果操作数是一个标识，则返回一个由负号和标识符连接在一起的字串。否则，如果字串以正号或负号开头，则返回以相反符号开头的字串。这些规则的一个效果是 `-bareword` 等于 `"-bareword"`。这个东西对 **Tk** 程序员很有用。

单目 **~** 操作符进行按位求反，也就是 **1** 的补数。从定义上来看，这个是有有点不可移植的东西，因为它受限于你的机器。比如，在一台 **32** 位机器上，`~123` 是 `4294967172`，而在一台 **64** 位的机器上，它是 `18446744073709551493`。不过你早就知道这个了。

你可能还不知道的是，如果 **~** 的参数是字串而不是数字，则返回等长字串，但是字串的所有位都是互补的。这是同时翻转所有位的最快的方法，而且它还是可移植的翻转位的方法，因为它不依靠你的机器的字大小。稍后我们将谈到按位逻辑操作符，它也有一个面向字串的变体。

单目 **+** 没有任何语义效果，即使对字串也一样。它在语法上用于把函数名和一个圆括弧表达式分隔开，否则它们会被解释成一个一体的函数参数。（参阅“项和列表操作符”的例子。）如果你向它的一边进行考虑，**+** 取消了圆括弧把前缀操作符变成函数的作用。

单目操作符 **** 给它后面的东西创建一个引用。在一个列表上使用，它创建一系列引用。参阅第八章中的“反斜杠操作符”获取详细信息。不要把这个性质和字串里的反斜杠的作用混淆了，虽然两者都有防止下一个东西被转换的模糊的含义。当然这个相似也并不是完全偶然的。

3.6 绑定操作符

双目 `=~` 把一个字串和一个模式匹配、替换或者转换绑定在一起。要不然这些操作会搜索或修改包含在 `$_`（缺省变量）里面的字串。你想绑定的字串放在左边，而操作符本身放在右边。返回值标识右边的操作符的成功或者失败，因为绑定操作符本身实际上不做任何事情。

如果右边的参数是一个表达式而不是模式匹配、子过程或者转换，那运行时该表达式会被解释成一个搜索模式。也就是说，`$_ =~ $pat` 等效于 `$_ =~ /$pat/`。这样做要比明确搜索效率低，因为每次计算完表达式后都必须检查模式以及可能还要重新编译模式。你可以通过使用 `qr//`（引起正则表达式）操作符预编译最初的模式的方法来避免重新编译。

双目 `!~` 类似 `=~` 操作符，只是返回值是 `=~` 的对应返回值的逻辑非。下面的表达式功能上是完全一样的：

```
$string !~ /pattern/
not $string =~ /pattern/
```

我们说返回值标识成功，但是有许多种成功。替换返回成功替换的数量，转换也一样。（实际上，转换操作符常用于字符计数。）因为任何非零值都是真，所以所有的都对。最吸引人的真值类型是模式的列表赋值：在列表环境下，模式匹配可以返回和模式里圆括弧相匹配的子字符串。不过，根据列表赋值的规则，如果有任何东西匹配并且赋了值，列表赋值本身将返回真，否则返回假。因此，有时候你会看到这样的东西：

```
if( ($k, $v) = $string =~ m/(\w+)= (\w*)/) {
    print "KEY $k VALUE $v\n";
}
```

让我们分解这个例子。`=~` 的优先级比 `=` 高，因此首先计算 `=~`。`=~` 把字符串 `$string` 绑定与右边的模式进行匹配，右边是扫描你的字符串里看起来象 `KEY=VALUES` 这样的东西。这是在列表环境里，因为它是在一个列表赋值的右边。如果匹配了模式，它返回一个列表并赋值给 `$k` 和 `$v`。列表赋值本身是在标量环境，所以它返回 `2`——赋值语句右边的数值的个数。而 `2` 正好又是真——因为标量环境也是一个布尔环境。当匹配失败，没有赋值发生，则返回零，是假。

关于模式规则的更多内容，参阅第五章，模式匹配。

3.7 乘号操作符

Perl 提供类似 C 的操作符（乘）、/（除）、和 %（模除）。和 / 的运行和你预料的一样，对两个操作数进行乘或除。除法是以浮点数进行的，除非你用了 `integer` 用法模块。

% 操作符在用整数除法计算余数前，把它的操作数转换为整数。（不过，如果必要，它会以浮点进行除法，这样你的操作数在大多数 32 位机器上最多可以有（以浮点）15 位。）假设你的两个操作数叫 `$b` 和 `$a`。如果 `$b` 是正数，那么 `$a % $b` 的结果是 `$a` 减去 `$b` 不大于 `$a` 的最大倍数（也就意味着结果总是在范围 `0 .. $b-1` 之间）。如果 `$b` 是负数，那么 `$a % $b` 的结果是 `$a` 减去 `$b` 不小于 `$a` 的最小倍数（意味着结果介于 `$b+1 .. 0` 之间）。

当 `use integer` 在范围里时，% 直接给你由你的 C 编译器实现的模除操作符。这个操作符对负数定义得不是很好，但是执行得更快。

双目 `x` 是复制操作符。实际上，它是两个操作符，在标量环境里，它返回一个由左操作数重复右操作数的次数连接起来的字符串。（为了向下兼容，如果左操作数没有位于圆括弧中，那么它在列表环境里也这样处理。）

```
print '-' x 80;           # 打印一行划线
print "\t" x ($tab/8), ' ' x ($tab%8); # 跳过
```

在列表环境里，如果左操作数是在圆括弧中的列表，`x` 的作用是一个列表复制器，而不是字符串复制器。这个功能对初始化一个长度不定的数组的所有值为同一值时很有用：

```
@ones = (1) x 80;        # 一个80个1的列表
@ones = (5) x @ones;     # 把所有元素设置为5
```

类似，你还可以用 `x` 初始化数组和散列片段：

```
@keys = qw(perls before swine);
@hash{@keys} = (" ") x @keys;
```

如果这些让你迷惑，注意 `@keys` 被同时当做一个列表在赋值左边使用和当做一个标量值（返回数组长度）在赋值语句右边。前面的例子在 `%hash` 上有相同的作用：

```
$hash{perls} = "";
$hash{before} = "";
$hash{swine} = "";
```

3.8 附加操作符

很奇怪的是，Perl 还有惯用的 `+`（加法）和 `-`（减法）操作符。两种操作符都在必要的时候把它们的参数从字符串转换为数字值并且返回一个数字值。

另外，Perl 提供 `.` 操作符，它做字符串连接处理。比如：

```
$almost = "Fred" . "Flitstone"; # 返回 FredFlitstone2
```

请注意 Perl 并不在连接的字串中间放置空白。如果你需要空白，或者你要连接的字串多于两个，你可以使用 `join` 操作符，在第二十九章，函数，中介绍。更常用的是人们在一个双引号引起的字符串里做隐含的字符串连接：

```
$fullname = "$firstname $lastname";
```

3.9 移位操作符

按移位操作符（`<<` 和 `>>`）返回左参数向左（`<<`）或向右（`>>`）移动由右参数声明位（是 `bit`）数的值。参数应该是整数。比如：

1. `<< 4;` # 返回16
2. `>> 4;` # 返回2

3.10 命名单目操作符和文件测试操作符

在第二十九章里描述的一些“函数”实际上都是单目操作符。表 3-2 列出所有命名的单目操作符。

表 3-2 命名单目操作符

-X (file tests)	gethostbyname	localtime	return
alarm	getnetbyname	lock	rmdir
caller	getpgrp	log	scalar
chdir	getprotobyname	stat	sin

chroot	glob	my	sleep
cos	gmtime	oct	sqrt
defined	goto	ord	srand
delete	hex	quotemeta	stat
do	int	rand	uc
eval	lc	readlink	ucfirst
exists	lcfirst	ref	umask
exit	length	require	undef

单目操作符比某些双目操作符的优先级高。比如：

```
sleep 4 | 3;
```

并不是睡 7 秒钟；它先睡 4 秒钟然后把 `sleep` 的返回值（典型的是零）与 3 进行按位或操作，就好像该操作符带这样的圆括弧：

```
(sleep 4) | 3;
```

与下面相比：

```
print 4 | 3;
```

上面这句先拿 4 和 3 进行或操作，然后再打印之（本例中是 7），就好像是下面这样写的一样：

```
print (4 | 3);
```

这是因为 `print` 是一个列表操作符，而不是一个简单的单目操作符。一旦你知道了哪个操作符是列表操作符，你再把单目操作符和列表操作符区分开就不再困难了。当你觉得有问题时，你总是可以用圆括弧把一个命名的单目操作符转换成函数。记住：如果看起来象函数，那它就是函数。

有关命名单目操作符的另一个趣事是，它们中的许多在你没有提供参数时，缺省使用 `$_`。不过，如果你省略了参数，而跟在名单目操作符后面的记号看起来又象一个参数开头的话，那 Perl 就傻了，因为它期望的是一个项。如果 Perl 的记号是列在表 3-3 中的一个字符，那么该记号会根据自己是期待一个项还是操作符转成不同的记号类型。

表3-3 模糊字符

字符	操作符	项
+	加法	单目正号
-	减法	单目负号
*	乘法	*类型团
/	除法	/模式/
<	小于号，左移	,<<END
.	连接	.3333
?	?:	?模式?
%	模除	%assoc
&	&, &&	&subroutine（子过程）

所以，典型的错误是：

```
next if length < 80;
```


在这里，< 在分析器眼里看着象 <> 输入符号（一个项）的开始，而不是你想要的“小于”（操作符）。我们实在是没办法修补这个问题同时还令 Perl 没有毛病。如果你实在懒得连 \$ _ 这两个字符都不愿意敲，那么用下面的代替：

```
next if length() <80;
next if (length) < 80;
next if 80 > length;
next unless length >== 80;
```

当（分析器）期望一个项时，一个负号加一个字母总是被解释成一个文件测试操作符。文件测试操作符是接受一个参数的单目操作符，其参数是文件名或者文件句柄，然后测试该相关的文件，看看某些东西是否为真。如果忽略参数，它测试 \$ _，但除了 -t 之外，-t 是测试 STDIN。除非另有文档，它测试为真时返回 1，为假时返回 ""，或者如果文件不存在或无法访问时返回未定义。目前已实现的文件测试操作符列在表 3-4。

表3-4 文件测试操作符

操作符	含义
-r	文件可以被有效的UID/GID读取。
-w	文件可以被有效的UID/GID写入。
-x	文件可以被有效的UID/GID执行。
-o	文件被有效UID所有
-R	文件可以被真实的UID/GID读取。
-W	文件可以被真实的UID/GID写入。
-X	文件可以被真实的UID/GID执行。
-O	文件被真实的UID所有
-e	文件存在
-z	文件大小为零
-s	文件大小不为零（返回大小）
-f	文件是简单文件
-d	文件是目录
-l	文件是符号连接
-p	文件是命名管道（FIFO）。
-S	文件是套接字
-b	文件是特殊块文件
-c	文件是特殊字符文件
-t	文件句柄为一个tty打开了
-u	文件设置了setuid位
-g	文件设置了setgid位
-k	文件设置了sticky位
-T	文件是文本文件
-B	文件是一个二进制文件（与-T对应）
-M	自从修改以来的文件以天记的年龄（从开始起）
-A	自从上次访问以来的文件以天记的年龄（从开始起）
-C	自从inode修改以来的文件以天记的年龄（从开始起）

请注意 -s/a/b/ 并不是做一次反向替换。不过，说 -exp(\$foo) 仍然会和你预期的那样运行，因为

只有跟在负号后面的单个字符才解释成文件测试。

文件权限操作符 `-r` , `-R` , `-w` , `-W` , `-x` 和 `-X` 的解释各自基于文件和用户的用户ID 和组 ID。可能还有其他原因让你无法真正读, 写或执行该文件, 比如 **Andrew File System(AFS)** 的访问控制列表(注: 不过, 你可以用 `use filetest` 用法覆盖内建的语义。参阅第三十一章, 用法模块)。还要注意的, 对于超级用户而言, `-r` , `-R` , `-w` 和 `-W` 总是返回 `1` , 并且如果文件模式里设置了执行位, `-x`和`-X`也返回 `1`。因此, 由超级用户执行的脚本可能需要做一次 `stat` 来检测文件的真实模式, 或者暂时把 `UID` 设置为其他的什么东西。

其他文件测试操作符不关心你是谁。任何人都可以用这些操作符来测试"普通"文件:

```
while (<>) {
    chomp;
    next unless -f $_;      #忽略“特殊”文件
    ...
}
```

`-T` 和 `-B` 开关按照下面描述的方法运转。检查文件的第一块的内容, 查找是否有类似控制字符或者设置了第八位的字符(这样看起来就不象 **UTF-8**)。如果有超过三分之一的字符看起来比较特殊, 它就是二进制文件; 否则, 就是文本文件。而且, 任何在第一块里包含 **ASCII NUL** (`\0`) 的文件都会被认为是二进制文件。如果对文件句柄使用 `-T` 或 `-B`, 则检测当前输入(标准 **I/O** 或者“**stdio**”)缓冲区, 而不是文件的第一块。`-T` 和 `-B` 对空文件都返回真, 或者测试一个文件句柄时读到 **EOF** (文件结束)时也返回真。因为 **Perl** 需要读文件才能进行 `-T` 测试, 所以你大概不想在某些特殊文件上用 `-T` 把系统搞得挂起来, 或者是发生其他让你痛苦的事情吧。所以, 大多数情况下, 你会希望先用 `-f` 测试, 比如:

```
next unless -f $file && -T $file;
```

如果给任何文件测试(操作符)(或者是 `stat` 或 `lstat` 操作符)的特殊文件句柄只包含单独一个下划线, 则使用前一个文件测试的 `stat` 结构, 这样就省了一次系统调用。(对 `-t` 无效, 而且你还要记住 `lstat` 和 `-l` 会在 `stat` 结构里保存符号连接而不是真实文件的数值。类似地, 在一个正常的 `stat` 的后面的 `-l _` 总是会为假。)

下面是几个例子:

```
print "Can do.\n" if -r $a || -w _ || -x _;

stat($filename);
print "Readable\n" if -r _;
print "Writable\n" if -w _;
print "Executable\n" if -x _;
print "Setuid\n" if -u _;
print "Setgid\n" if -g _;

print "Sticky\n" if -k _;
print "Text\n" if -T _;
print "Binary\n" if -B _;
```

`-M` , `-A` 和 `-C` 返回脚本开始运行以来一天(包括分数日子)计的文件年龄。这个时间是保存在特殊变量 `$^T` (`$BASETIME`) 里面的。因此, 如果文件在脚本启动后做了改变, 你就会得到一个负数时间。请注意, 大多数时间值(概率为 **86400** 分之 **86399**)都是分数, 所以如果不用 `int` 函数就拿它和一个整数进行相等性测试, 通常都会失败。比如:

```
next unless -M $file > .5;    # 文件长于 12 小时
```

```
&newfile if -M $file < 0;    # 文件比进程新
&mailwarning if int(-A) == 90;    # 文件 ($_) 是 90 十天前访问的
```

要把脚本的开始时间重新设置为当前时间，这样：

```
$^T = time;
```

3.11 关系操作符

Perl 有两类关系操作符。一类操作符操作数字值，另一类操作字符串值，在表 3-5 中列出。

表3-5 关系操作符

数字	字符串	含义
>	gt	大于
>=	ge	大于或等于
<	lt	小于
<=	le	小于或等于

这些操作符在真时返回 1 而为假时返回 ""。请注意关系操作符不能结合，这就意味着 `$a < $b < $c` 是语法错误。

如果没有区域声明，字符串的比较基于 ASCII/Unicode 的顺序比较，而且和一些计算机语言不同的是，在比较中，尾部的空白也计入比较中。如果有区域声明，比较顺序以所声明区域的字符集顺序为基础。（以区域字符集为基础的比较机制可能可以也可能不能和目前正在开发的 Unicode 比较机制很好地交互。）

3.12 相等操作符

相等操作符在表 3-6 里面列出，它们和关系操作符很象。

表3-6 相等操作符

数字	字符串	含义
==	eq	等于
=	ne	不等于
<=>	cmp	比较，带符号结果

等于和不等操作符为真时返回 1，为假时返回 ""（和关系操作符一样）。`<=>` 和 `cmp` 操作符在左操作数小于右操作数时返回 -1，相等时返回 0，而大于时返回 1。尽管相等操作符和关系操作符很象，但是它们的优先级比较低，因此 `$a < $b <=> $c < $d` 语法上是合法的。

因为很多人看过“星球大战”，`<=>`操作符也被称为“飞船”操作符。

3.13 位操作符

和 C 类似，Perl 也有位操作符 AND，OR，和 XOR（异或）：`&`，`|` 和 `^`。在本章开始的时候，你辛辛苦苦地检查表格，发现按位 AND（与）操作符比其他的优先级高，但我们那时候是骗你的，在这里我们会一并讨论一下。

这些操作符对数字值和对字符串值的处理不同。（这是少数几个 Perl 关心的区别。）如果两个操作数都是数字（或者被当作数字使用），那么两个操作数都被转换成整数然后在两个整数之间进行位操

作。我们保证这些整数是至少 32 位长，不过在某些机器上可以是 64 位长。主要是要知道有一个由机器的体系所施加的限制。

如果两个操作数都是字串（而且自从它们被设置以来还没有当作数字使用过），那么该操作符用两个字串里面来的位做位操作。这种情况下，没有任何字长限制，因为字串本身没有尺寸限制。如果一个字串比另一个长，Perl 就认为短的那个在尾部有足够的 0 以弥补区别。

比如，如果你 AND 两个字串：

```
"123.45" & "234.56"
```

你得到另外一个字串：

```
"020.44"
```

不过，如果你拿一个字串和一个数字 AND：

```
"123.45" & 234.56
```

那字串先转换成数字，得到：

```
1. 45 & 234.56
```

然后数字转换成整数：

```
1. & 234
```

最后得到值为 106。请注意所有位字串都是真（除非它们结果是字串“0”）。这意味着如果你想看看任意字节是否为非零，你不能这么干：

```
if ( "fred" & "\1\2\3\4" ) { ... }
```

你得这么干：

```
if ( )"fred" & "\1\2\3\4" ) =~ /[^\0]/ ) { ... }
```

3.14 C 风格的逻辑（短路）操作符

和 C 类似，Perl 提供 &&（逻辑 AND）和 ||（逻辑 OR）操作符。它们从左向右计算（&& 比 || 的优先级稍稍高一点点），测试语句的真假。这些操作符被认为是短路操作符，因为它们是通过计算尽可能少的操作数来判断语句的真假。例如，如果一个 && 操作符的左操作数是假，那么它永远不会计算右操作数，因为操作符的结果就是假，不管右操作数的值是什么。

例子	名称	结果
<code>\$a && \$b</code>	And	如果\$a为假则为\$a，否则\$b
<code>\$a \$b</code>	Or	如果\$a为真则为\$a，否则\$b

这样的短路不仅节约时间，而且还常常用于控制计算的流向。比如，一个经常出现的 Perl 程序的俗语是：

```
open(FILE, "somefile") || die "Can't open somefile: $!\n";
```

在这个例子里，Perl 首先计算 open 函数，如果值是真（somefile 被成功打开），die 函数的执行就不必要了，因此忽略。你可以这么读这句文本“打开文件，要不然就去死！”。

&& 和 **||** 操作符和 **C** 不同的是，它们不返回 **0** 或 **1**，而是返回最后计算的值。如果是 **||**，这个特性好就好在你可以从一系列标量数值中选出第一个为真的值。所以，一个移植性相当好的寻找用户的家目录的方法可能是：

```
$home = $ENV{HOME}
|| $ENV{LOGDIR}
|| (getpwuid($<)) [7]
|| die "You're homeless!\n";
```

另一方面，因为左参数总是在标量环境里计算，所以你不能把 **||** 用于在两个集群之间选择其一进行赋值：

```
@a = @b || @c;      # 这样可不对
@a = scalar(@b) || @c; # 上面那句实际上是这个意思，@a 里只有 @b 最后的元素
@a = @b ? @b : @c;   # 这个是对的
```

Perl 还提供优先级比较低的 **and** 和 **or** 操作符，这样程序的可读性更好而且不会强迫你在列表操作符上使用圆括弧。它们也是短路的。参阅表 1-1 获取完整列表。

3.15 范围操作符

范围操作符 **..** 根据环境的不同实际上是两种不同的操作符。

在标量环境里，**..** 返回一个布尔值。该操作符是双稳定的，类似一个电子开关，并且它仿真 **sed**，**awk**，和各种编辑器的行范围（逗号）操作符。每个 **..** 操作符都维护自身的状态。只要它的左操作数为假就一直为假。一旦左操作数为真，该范围操作符就保持真的状态直到右操作数为真，右操作数为真之后该范围操作符再次为假。该操作符在下次计算之前不会变成假。它可以测试右操作数并且在右操作数变真后在同一次计算中变成假（**awk** 的范围操作符的特性），不过它还是会返回一次真。如果你不想拖到下一次计算中才测试右操作数（也是 **sed** 的范围操作符的工作方式），只需要用三个点（**...**）代替两个点（**..**）。对于 **..** 和 **...**，当操作符处于假状态后就不再测试右操作数，而当操作符处于真状态后就不再测试左操作数。

返回的值要么是代表假的空字符串或者是代表真的一个序列数（从 **1** 开始）。该序列数每次碰到新范围时重置。在一个范围里的最后序列数后面附加了字符串“**E0**”，这个字符串不影响它的数字值，只是给你一些东西让你可以搜索，这样你可以把终点排除在外。你也可以通过等待 **1** 的序列数的方法把起始点排除在外。如果标量 **..** 的某个操作数是数字文本，那么该操作数隐含地与 **\$.变量** 对比，**\$.** 里包含你的输入文件的当前行号。比如：

```
if(101 .. 200) {print;}      # 打印第二个一百行
next line if( 1.. /$^/);     # 忽略开头行
s/^/> / if (/^$/ .. eof());   # 引起体
```

在列表环境里，**..** 返回一系列从左值到右值计数（以一）的数值。这样对书写 **(1 .. 10)** 循环和数组片段的操作很有帮助：

```
for (101 .. 200) {print;}    # 打印 101102 ... 199200
@foo = @foo[0 .. $#foo];     # 一个昂贵的空操作
@foo = @foo[-5 .. -1];       # 最后5个元素的片段
```

如果左边的值大于右边的值，则返回一个空列表。（要产生一系列反序的列表，参阅 **reverse** 操作符。）

如果操作数是字符串，范围操作符利用早先讨论过的自增处理。（注：如果在所声明的终值不是自增处理中产生的序列中的数，那么该序列将继续增加直到下一个值比声明的终值长为止。）因此你可以

说:

```
@alphabet = ('A' .. 'Z');
```

以获得所有英文字母, 或者:

```
$hexdigit = (0 .. 9, 'a' .. 'f')[$num & 15];
```

获得一个十六进制位, 或者:

```
@z2 = ('01' .. '31'); print $z2[$mday];
```

获得带有前导零的日期。你还可以说:

```
@combos = ('aa' .. 'zz');
```

获取所有两个小写字符的组合。不过, 用下面的语句要小心:

```
@bigcombos = ('aaaaaa' .. 'zzzzzz');
```

因为这条语句要消耗很多内存。准确地说, 它需要存储 308,915,776 个标量的空间。希望你分配了一个非常大的交换分区。可能你会考虑用循环代替它。

3.16 条件操作符

和 C 里一样, ?: 是唯一的一个三目操作符。它通常被成为条件操作符, 因为它运转起来非常象一个 if-then-else, 而且, 因为它是一个表达式而不是一个语句, 所以它可以非常容易地嵌入其他表达式和函数调用中。作为三目操作符, 它的两个部分分隔了三个表达式:

COND ? THEN : ELSE

如果条件 COND 为真, 那么只计算 THEN 表达式, 并且其值成为整个表达式的值。否则, 只计算 ELSE 表达式的值, 并且其值成为整个表达式的值。

不管选择了哪个参数, 标量或者列表环境都传播到该参数。(第一个参数总是处于标量环境, 因为它是一个条件。)

```
$a = $ok ? $b : $c;    # 得到一个标量
@a = $ok ? @b : @c;    # 得到一个数组
$a = $ok ? @b : @C;    # 得到一个数组元素的计数
```

你会经常看到条件操作符嵌入在一列数值中以格式化 printf, 因为没人愿意只是为了在两个相关的值之间切换而复制整条语句。

```
printf "I have $d camel$.\\n",
    $n,    $n == 1 ? "" : "s";
```

?: 的优先级比逗号高, 但是比你可能用到的其他大多数操作符 (比如本例中的 ==) 都低, 因此通常你用不着用圆括弧括任何东西。不过如果你愿意, 你可以用圆括弧让语句更清晰。对于嵌套在其他 THEN 部分的其他条件操作符, 我们建议你在它们中间放入断行和缩进, 就好象它们是普通 if 语句一样:

```
$leapyear =
    $year % 4 == 0
    ? $year % 100 == 0
    ? $year % 400 == 0
    ? 1
    : 0
```



```

        :1
    :0;

```

类似地，对于嵌套在 **ELSE** 部分的更早的条件，你也可以这样处理：

```

$leapyear =
    $year % 4
    ? 0
    : $year % 100
    ? 1
    : $year % 400
    ? 0
    : 1;

```

不过通常最好把所有 **COND** 和 **THEN** 部分垂直排列：

```

$leapyear =
    $year % 4 ? 0 :
    $year % 100 ? 1 :
    $year % 400 ? 0 : 1

```

把问号和冒号对齐可以让你在非常混乱的结构中找到感觉：

```

printf "Yes, I like my %s book!\n",
    $i18n eq "french"    ? "chameau"    :
    $i18n eq "german"    ? "Kamel"      :
    $i18n eq "japanese"  ? "\x{99F1}\x{99DD}" :
    "camel"

```

如果第二个和第三个参数都是合法的左值（也就是说你可以给他们赋值），而且同时为标量或者列表（否则，**Perl** 就不知道应该给赋值的右边什么环境），那你就可以给它赋值（注：这样无法保证你的程序有很好的可读性。但是这种格式可以用于创建一些很酷的东西。）：

```

($a_or_b ? $a : $b) = $c;      # 把 $a 或 $b 置为 $c 的值

```

请注意，条件操作符比各种各样赋值操作符绑定得更紧。通常这是你所希望的（比如说上面 **\$leapyear** 赋值），但如果你不用圆括弧的话，你就没法让它反过来。不带圆括弧（在条件操作符）中使用嵌入的赋值会给你带来麻烦，并且这时还不会有分析错误，因为条件操作符左值分析。比如说，你可能写下面这些：

```

$a % 2 ? $a += 10 : $a += 2      # 错

```

上面这个会分析为：

```

(($a % 2) ? ($a += 10) : $a) += 2

```

3.16 赋值操作符

Perl 可以识别 **C** 的赋值操作符，还提供了几个自己的。这里是一些：

```

=      **=      +=      *=      &=      <<=      &&=
      -=      /=      |=      >>=      ||=
      .=      %=      ^=
      x=

```

每个操作符需要一个目标左值（典型值是一个变量或数组元素）在左边以及一个表达式在右边。对于简单赋值操作符：

```
TARGET = EXPR
```

EXPR 的值被存储到 TARGET 指明的变量或者位置里面。对其他操作符而言，Perl 计算表达式：

```
TARGET OP= EXPR
```

就好像它是这么写的一样：

```
TARGET = TARGET OP EXPR
```

这是有利于简便的规则，不过它在两个方面会误导我们。首先，赋值操作符总是以普通赋值的优先级进行分析的，不管 OP 本身的优先级是什么。第二，TARGET 只计算一次。通常这样做没有什么问题，除非存在副作用，比如一个自增：

```
$var[$a++] += $value;           # $a增加了一次
$var[$a++] = $var[$a++] + $value; # $a增加了两次
```

不象 C 里，赋值操作符生成一个有效的左值。更改一个赋值等效于先赋值然后把变量修改为赋的值。这招对修改一些东西的拷贝很管用，象这样：

```
($tmp = $global) += $constant;
```

等效于：

```
$tmp = $global + $constant;
```

类似：

```
($a += 2) *= 3;
```

等效于：

```
$a += 2;
$a *= 3;
```

本招并非绝招，不过下面是你经常看到的习惯用语：

```
($new = $old) =~ s/foo/bar/g;
```

无论什么情况，赋值的值是该变量的新值。因为赋值操作符从右向左结合，所以这一点可以用于给多个变量赋同一个值，比如：

```
$a = $b = $c = 0;
```

把 0 赋予 \$c，其结果（还是 0）给 \$b，其结果（依然为 0）再给 \$a。

列表赋值可能只能在列表环境里用简单赋值操作符，=，赋值。列表赋值返回该列新值，就象标量赋值一样。在标量环境里，列表赋值返回赋值右边可获得的值的数目，就象我们在第二章，集腋成裘，里谈到的一样。我们可以利用这个特性测试一个失败（或者不再成功）时返回空列表的函数，比如：

```
while (($key, $value) = each %gloss) { ... }

next unless ($dev, $ino, $mode) = stat $file;
```

3.18 逗号操作符

双目“,”是逗号操作符。在标量环境里它先在空环境里计算它的左参数，把那个值抛弃，然后在标量

环境里计算它的右参数并且返回之。就象 C 的逗号操作符一样。比如：

```
$a = (1,3);
```

把 3 赋予 `$a`。不要混淆标量环境用法和列表环境用法。在列表环境里，逗号只是列表参数的分隔符，而且把它的两个参数都插入到列表中。并不抛弃任何数值。

比如，如果你把前面例子改为：

```
@a = (1,3);
```

你是在构造一个两元素的列表，而：

```
atan2(1,3);
```

是用两个参数调用函数 `atan2`。

连字符 `=>` 大多数时候就是逗号操作符的同义词。对那些成对出现的文档参数很有用。它还强制把它紧左边的标识符解释为一个字串。

3.19 列表操作符（右向）

列表操作符的右边是所有列表操作符的参数（用逗号分隔的），所以如果你往右看的话，列表操作符的优先级比逗号低，一旦列表操作符开始啃那些逗号分隔的参数，你能让它停下来的唯一办法就是停止整个表达式的记号（比如分号或语句修改器），或者停止当前子表达式的记号（比如右圆括弧或花括弧），或者是我们下面要讲的低优先级的逻辑操作符。

3.20 逻辑与，或，非和异或

作为 `&&`，`||` 和 `!` 的低优先级候补，Perl 提供了 `and`，`or`，和 `not` 操作符。这些操作符的性质和它们对应的短路操作符是一样的，尤其是 `and` 和 `or`，这样它们不仅可以用于逻辑表达式也可以用于流控制。

因为这些操作符的优先级远远比从 C 借来的那些低，所以你可以很放心地把它用在一个列表操作符后面而不用加圆括弧：

```
unlink "alpha", "beta", "gamma"
  or gripe(), next LINE;
```

而对 C 风格的操作符你就得这么写：

```
unlink("alpha", "beat", "gamma") || (girpe(), next LINE);
```

不过你不能简单地把所有 `||` 替换为 `or`。假设你把：

```
$xyz = $x || $y || $z;
```

改成：

```
$xyz = $x or $y or $z;    # 错
```

这两句是完全不同的！赋值的优先级比 `or` 高但是比 `||` 低，所以它总是会先给 `$xyz` 赋值 `$x`，然后再进行 `or`。要获得和 `||` 一样的效果，你就得写：

```
$xyz = ( $x or $y or $z );
```

问题的实质是你仍然必须学习优先级（或使用圆括弧），不管你用的是哪种逻辑操作符。

还有一个逻辑 `xor` 操作符在 `C` 或 `Perl` 里面没有很准确的对应，因为另外一个异或操作符（`^`）按位操作。`xor` 操作符不能短路，因为两边都必须计算。`$a xor $b` 的最好的等效可能是 `!$a = !$b`。当然你也可以写 `!$a ^ !$b`，甚至可以是 `$a ? !$b : !$b`。要点是 `$a` 和 `$b` 必须在布尔环境里计算出真或假，而现有的位操作符在没有帮助的前提下并不提供布尔环境。

3.21 Perl 里没有的 C 操作符

下面是 `Perl` 里没有的 `C` 操作符：

单目 `&` 取址操作符。不过，`Perl` 的 `\` 操作符（用于使用引用）填补了这个生态空白：

```
$ref_to_var = \ $var;
```

不过 `Perl` 的引用要远比 `C` 的指针更安全。

单目 `*` 解取址操作符。因为 `Perl` 没有地址，所以它不需要解取址。它有引用，因此 `Perl` 的变量前缀字符用做截取址操作符，并且还标明类型：`$`，`@`，`%`，和 `&`。不过，有趣的是实际上有一个 `*` 解引用操作符，但因为 `*` 是表示一个类型团的趣味字符，所以无法将其用于同一目的。（类型）类型转换操作符。没人喜欢类型转换。

Revision: r1.2 - 14 Oct 2005 - 04:42 - [TingYu](#)

[Perl](#) > [PerlProgramming3](#) > [P3GoryDetail](#) > [PerlUnaryandBinaryOperators](#)

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](#)