Perl 学习笔记

廖海仁

2010 年 5 月

目　录

1. Perl 简介

Perl 一般被认为是 Practical Extraction and Report Language(实用获取与报表语言)的缩写，是由 C 以及 sed、awk、Unix shell 及其它语言演化而来的一种语言。它由语言学家 Larry Wall 最初发明及实现。Perl 是一种为扫描任意的文本文件，从这些文本文件中获取信息，基于这些信息打印报表而优化的语言。它也很适合于完成许多系统管理的任务。Perl 是一种粘合性语言，旨在实用（易用、高效、完整）而不是漂亮（优美、小巧）。其吉祥物是骆驼，取其虽并不漂亮却任劳任怨、能干活之特点。

Perl 不随意限制数据的大小，只要你有充足的内存。递归的深度也不受限制。关联数组使用可以根据需要扩展以避免性能衰退。Perl 能利用复杂的模式匹配技巧来快速扫描大量数据。尽管善于处理文本，Perl 也能处理二进制数据。

Perl 5 增加了模块化处理、面向对象编程、引进引用以处理多维数组等复杂的数据结构、Unicode 支持、多线程支持等功能，使得 Perl 成为一种更加完备强大的语言。

Perl 语言的座右铭是：There's More Than One Way To Do It (TMTOWTDI，有多种方法可以完成一件事，或者“条条道路通北京”)。Perl 是一种自然和智能语言，它能根据上下文环境解释执行，同时有大量省略写法。

如果你通常想使用 sed、awk 或 sh 来解决的问题，但是却发现它们的能力不够，或者想运行得更快一点，却不想傻傻地用 C 来写，Perl 将是很好的选择。Perl 擅长于文本处理和系统管理，不适合于实时嵌入式系统编程、操作系统底层开发（比如驱动程序开发）、复杂的多线性共享内存应用以及极度大的应用。

Perl 语言的长处：

- 强大的正则表达式和模式匹配功能(接近理想的正则表达式语言)
- 复杂灵活的数据结构(Array of Array, Array of Hash，Hash of Array, Hash of Hash 等)
- Unicode 支持（相对 AWK 与 C）

其不足是：

- 动态类型语言，不是很可靠
- 自然语言，是优点也是缺陷，使得 Perl 语言代码可能晦涩难懂；
- 智能语言，是优点也产生不足：根据上下文解释编程者之意，可能产生臆断；也因此产生种种编程陷阱。
- 不是很优美。多种方法做事，有时会让编程者无所适从。
- 性能问题。Perl 灵活的数据结构和处理性能通常不是很高（相对 C/C++），若需要高性能的处理有时需要使用其它语言重写。

Perl 语言的学习曲线浅而长。只要看看《Learning Perl》，就可以编写简单的 Perl 程序了，然而要深入掌握 Perl，要使用 Perl 的复杂数据结构，进行面向对象编程、多线程编程则不是那么容易的事，需要很长的时间学习和实践。

2. 数据类型

2.1 概览

Perl 有以下几种数据类型：标量（Scalars）、 数组(Arrays)、关联数组(Associative Arrays,或称 Hash)、子程序(Subroutine)和 Typeglob（*标识，指所有以上几种类型）。这些类型都有不同的符号标识。这些类型的说明如下：

Type	Character	Example	Is a Name for
Scalar	$	$cents	An individual value(number or string)
Array	@	@large	A list of values, keyed by number
Hash	%	%interest	A group of values, keyed by string
Subroutine	&	&how	A callable chunk of Perl code
Typeglob	*	*struck	Everything named stuck

(以上引自《Programming Perl》)

关于 Perl 的类型可以做如下说明：

- 不同的数据类型不同的命名空间，所以三种的标识符可以相同但互不干扰，即$array, @array，%array 是完全不同的三个变量。
- Perl 的数据类型是大小写敏感的；直接使用，不用声明（这一点可能成为缺陷，可以使用 use strict 语句强制必须声明）
- 引用（references）是 Perl 5 引进的一种特殊的标量。
- 变量的命名以下划线或字母开头，可以是任意长度（1-251）。
- 文件句柄(Filehandle)指给一个文件、设备、Socket 或管道的名称。
- 子程序相对其它类型是动词（其余为名词），较为特别，将用特别的一章详细说明

2.2 命名空间(Namespaces)

Perl 有两种命名空间，分别是符号表（symbol tables，也称为包(package)）和词汇范围（lexical scopes，可以理解为局部空间）。符号表是存储全局变量的全局关联数组（包括存储其它关联数组），词汇范围是无名的空间，不存在于任何符号表中，而是与你程序中的一段代码块相关联。它们包含只对该代码块可见的变量。以 our 定义的变量存在符号表中，又称为全局变量或包变量(它们对包来说名义上是私有的,但是由于包本身是全局的,所以又是全局的)，以 my 定义的变量存储在词汇范围中，也称为局部变量。

在任一命名空间中，任一变量类型都有其子命名空间（由其前面的字符决定），用户可以给标量、数组、关联数组、文件句柄、子程序名、标号起相同的名字。Perl 的保留字也不会与变量名冲突。

Perl 做名字查找的次序如下：

- 查找最小一级的包含的程序块，看变量是否在同一块中声明（my 或者 our）
- 查找更高一级的包含程序块进行查找
- 查找整个编译单元看是否有声明
- 如果没有找到，Perl 将假定变量是包变量
- 如果没有包定义，Perl 将在无名的最高层包（main）中查找，$::bert 等同于$main::bert

除了用 my 与 our 声明变量外，Perl 中还有一种 local 的声明方式，注意 local 不是声明一个局部变量，而是使全局变量局部话，如果声明时未赋值，则所有的标量被初始化为 undef，所有的数组与关联数组被初始化为()。其使用方式比如：

```
if ($sw eq '-v') {
    local @ARGV = @ARGV;
    unshift @ARGV, 'echo';
    system @ARGV;
}
```

又如当需要输出 CSV 格式的文件时可以使用 local $, = ","; local $/="\n"; 免得每次 print 都得写许多","和最后的"\n";

2.3 标量(Scalars)

标量是一个字符串、数值或者指向某类型的引用。

Perl 的数值常量可以如下使用：

```
$x = 12345;              #整数
$x = 12345.67;           #浮点数
$x = 6.02e23;            #科学计数法
$x = 4_294_967_296;      #因为,是分隔符，所以 4,294,967,296 不能成立，Perl 用_代替
$x = 0377;               #八进制
$x = 0xffff;             #十六进制
$x = 0b1100_0000;        #二进制
```

Perl 的字符串常量通常由单引号或双引号括起来。双引号支持变量和转义字符序列替换；而单引号只支持两个转义字符 \'和\"。双引号支持的转义字符如下：

Code	Meaning
\n	Newline (usually LF)
\r	Carriage return (usually CR)
\t	Horizontal tab

\f	Form feed
\b	Backspace
\a	Alert(bell)
\e	ESC character
\033	ESC in octal
\x7f	DEL in hexadecimal
\cC	Control-C
\x{263a}	Unicode (smiley)
\N{NAME}	Named character (需要使用 use charnames)
\u	Force next character to uppercase
\l	Force next character to lowercase
\U	Force all following characters to uppercase
\L	Force all following characters to lowercase
\Q	Backslash all following nonalphanumeric characters
\E	End \U, \L, or \Q

Perl 除了使用单引号和双引号来引，还可以使用 q//与 qq//的更一般格式，Perl 的引用结构如下：

Customary	**Generic**	**Meaning**	**Interpolates**
‘’	q//	Literal String	NO
“”	qq//	Literal String	YES
``	qx//	Command execution	YES
()	qw//	Word list	NO
//	m//	Pattern Match	YES
s///	s/abc/123/	Pattern substitution	YES
y///	tr/abc/123/	Character translation	NO
“”	qr//	Regular expression	YES

（引自《Programming Perl》）

Perl 的标量相关函数有：

函数	**用法**	**作用**
chop	chop VARIABLE chop(LIST) chop 例子： @lines = `cat myfile`; chop @lines; chop($answer=<STDIN>);	删除标量的最后一个字符,返回所删除的字符,不带参数相当于 chop $_ 得到字符串的最后字符外的子字符串使用 substr($string, 0, -1);

chomp	chomp VARIABLE
	chomp(LIST)
	chomp
	例子：
	while (<PASSWD>) {
	chomp;
	@array = split /:/;
	…
	}
		删除变量最后的换行符
		与 chop 不同的是如果字符串不含有换行符将不起作用
		它返回删除字符的个数
		如果是 LIST,需要用()，比如使用 chomp($a, $b)而不是 chomp $a, $b;
		事实上，chomp 删除$/定义的任意值，而不只是最后的字符。
		如果$/是””(段落模式),chomp 将删除所有最后的换行符
		chomp %hash; 将删除所有%hash 值后的换行符

length	length EXPR length 例子： $blen = do {use bytes; length $string; };	返回表达式的长度 不要用它来求数组或 Hash 的元素个数，这时，使用 scalar @array; scalar keys %hash;
substr	substr EXPR, OFFSET, LENGTH, REPLACEMENT substr EXPR, OFFSET, LENGTH substr EXPR, OFFSET 例子： substr($var, 0, 0) = “Larry”; $oldstr = substr($var, -1, 1, “Curly”);	获取子字符串 如果 OFFSET 是负数，从字符串尾部开始计算；如果 LENGH 是负数，直到反向|LENGTH|长度的位置
uc	uc EXPR	转换为大写
ucfirst	ucfirst EXPR	首字母大写
lc	lc EXPR	转换为小写
lcfirst	lcfirst EXPR	首字母小写
index	index STR, SUBSTR, OFFSET index STR, SUBSTR	寻找子字符串的首个位置，未找到返回-1
rindex	rindex STR, SUBSTR, OFFSET rindex STR, SUBSTR	与 index 相同，只是返回最后一个匹配的位置
defined	defined EXPR defined	测试变量是否定义

undef	undef EXPR undef 例如： undef *xyz ($a, $b, undef, $c) = split;	Undefine 一个标量值、整个数组，整个 Hash,一个函数，或者 Typeglob **比较 delete $hash{$key}与 undef $hash{$key}的不同：delete 后，exists 将返回 false;undef 后，exists 继续返回 true,但是 defined 返回 false.**

Perl 的标量使用注意事项如下：

- 标量字符串可以是任何大小，以内存为限。
- Perl 由字符串向数值的自动转换不能识别 0，0b,0x，必须使用 oct 函数来做这种转换工作。
- C/C++中的 char、int、long、double、string 在 Perl 中全部抽象为标量。它可以是字符串、数或引用。.
- **Perl 根据上下文来对待 Scalars，认定其是字符串还是数值,即可以自动进行字符串与数值的互换(这一点从 AWK 继承而来)。**
- 在 Boolean 上下文中，一个标量值是 false 如果这个标量是空字符串””，数值 0 或者字符串”0”或者未定义(undef)。
- Perl 将整数放在放在计算机的浮点寄存器中，所以整数被当作浮点数看待。

2.4 数组(Arrays)

Perl 的数组是在 C 语言基础上的扩展而来，它的使用更加自由、灵活。Perl 数组的使用说明与注意事项如下：

- 它可以包含任意多个元素：最小的数组可以不含元素，而最大的数组可以占满全部可用内存；
- 数组下标默认从 0 开始，可以通过$[改变下标，最后一个下标是$#array_name，所以下列等式总是成立的 scalar(@array_name) == $#array_name - $[+1；
- 若访问不存在的数组元素，结果为 null (而不会产生错误)；
- 若给超过数组大小的元素赋值，则数组自动增长，原来没有的元素值为 null；
- 数组的元素不必是相同类型的，它可以是数值、字符串或引用的任意组合；
- **数组下标可以为负值，表示从数组末尾算起的数组元素(-1 相当于$#arary_nam)，但是应知道，使用下标的绝对值不能超过数组大小（若数组@array 有 10 个元素，不能给**

$array[-11]赋值，若只是访问，则为未定义）。

- 数组的初始化方式：

  @array = (0, 23, “point”, 3);

  @array = 1..100;

  @array = qw/a b c d/;

对于数组的常见操作是排序，Perl 数组的排序用 sort 函数来实现，但是应记住，即使对于数值数组，sort 也是按照字符顺序排序的，要实现按数值排序，应使用：sort { $a <=> $b} @array; 反向排序，使用 reverse sort @array; 反向数值排序，使用 sort {$b <=> $a} @array; 不分大小写排序，使用 sort {lc($a) <=> lc($b)} @array; 这里$a, $b 的使用很特别，它们都是默认的包全局变量，所以在程序中一般不要使用变量$a 和$b; 这种排序方式是 Perl 实用而难看的体现之一。数组相关函数如下：

函数	用法	作用
sort	sort USERSUB LIST sort BLOCK LIST sort LIST	对 LIST 排序，USERSUB 或 BLOCK 定义排序的方式
reverse	reverse LIST 例子： for(reverse 1..10) {} print reverse <>; #line tac undef $/; print scalar reverse <>; %barfoo = reverse %foobar; #Hash key/value 互换	反转函数（注意标量环境与 LIST 环境的差别）
chop	见标量函数节	去尾函数，对每一元素都去尾
chomp	见标量函数节	
split	split /PATTERN/, EXPR, LIMIT split /PATTERN/, EXPR split /PATTERN/ split 例子： @chars = split //, $word; @fields = split /:/, $line; @words = split “ “, $paragraph; @lines = split /^/, $buffer;	切分函数 **如果 PATTERN 省略或者是空格” “,函数将在/\s+/上切分，忽略所有前面的空格。** **注意：split(“ “)将模拟 AWK 的默认行为，但是 split(/ /)将允许有初始列，即真正对空格切分；** **LIMIT 是切分数，如果不提**

	print join ':', split / */, 'hi there'; ($login, $passwd, $remainder) = split /:/, $_, 3; split /([-,])/, "1-10,20"; = (1,'-',10,',', 20)	**供，Perl 默认 LIMIT=（列表变量数+1） /PATTERN/后可以加 i,x 参数 如果 PATTERN 含有括号，返回的 LIST 将包含分隔符**
join	join EXPR, LIST 例子： $string = join "\t", @array; $record = join ":", $login,$passwd,$uid,$gid,$gcos;	连接函数，把列表或数组中的元素合成一个字符串 **注意：与 split 函数不同的是,join 不接受模式作为参数**

push	push ARRAY, LIST
	例如：
	for (;;) {
	push @array, shift @array;
	}
		将 ARRAY 当成一个栈，将 LIST 的值压到 ARRAY 的末尾，相当于
		foreach $value (LIST) {
		$array[++$#array] = $value;
		}
		splice @array, @array, 0,LIST;

pop	pop ARRAY pop （默认在主程序中 pop @ARGV 在子程序中 pop @_）	将 ARRAY 当成一个栈，从中取出一个元素,相当于 $ARRAY[$#ARRAY--]

shift	shift ARRAY
	shift
	例如：
	sub marine {
	my $fathoms = shift; # depth
	my $fishes = shift; # number of fish
	…
	}
		从目标数组前端删除一个元素

unshift	unshift ARRAY, LIST 例如： unshift @ARGV, '-e', $cmd unless $ARGV[0] =~ /^-/;	向目标数组前端加入 LIST
splice	splice ARRAY, OFFSET, LENGTH, LIST splice ARRAY, OFFSET, LENGTH splice ARRAY, OFFSET splice ARRAY	删除数组从起始位置起固定个数的元素，同时返回被删除的数组元素

2.5 关联数组(Hashes)

虽然数组很有用，但它们有一个显著的缺陷，即很难记住哪个元素存储什么内容。这个问题产生的原因是数组元素要通过数字下标访问。关联数组是 Perl 语言最具特色的地方，也是 Perl 能够成为 CGI 程序首选的重要条件。关联数组是以任意字符串作为下标的数组。

关联数组的定义方式为 %hash = (key1, value1, key2, value2, …)，Perl5 允许使用"=>"来分隔下标与值。用"=>"可读性更好。比如： %salary = ("Tom"=>1000, "Jack"=>1298, "Roses"=>1892)关联数组总是随即存储的，它们不会以创建的顺序出现。

创建一个关联数组元素的最好方法是赋值，如$salary{"Jane"} =223，删除用 delete。如 delete ($salary{"Tom"})。一定要使用 delete 函数来删除关联数组的元素，这几乎是唯一的方法。一定不对关联数组使用内部函数 push, pop, shift, unshift 及 splice，因为其元素位置是随机的。

%hash 在标量环境中使用，将返回使用的桶与分配的桶的比值，要找到关联数组中键的数量，使用 scalar(keys %hash); 注意$hash{$x, $y, $z}与@hash{$x, $y, $z}的区别：$hash{$x,$y, $z}是一个值，相当于$hash{ join $; => $a, $b, $c }; @hash{$x, $y, $z}是关联数组的切片，是三个值。

列出关联数组的键与值， keys (%hash)操作符可以生成由关联数组%hash 中由关键字组成的列表，但是返回的元素不以任何固定顺序出现！可以用 sort()函数对 keys()返回值进行排序.

排序方式	例 子

对键值字符串排序	foreach $key (sort keys %ENV) {
	print $key, '=', $ENV{$key}, "\n";
	}

对 Hash 的键用数值排序	foreach $key (sort {$a <=> $b} keys %hash) { .. }

根据 Hash 的值从大到小排序	foreach $key (sort { $hash{$b} <=> $hash{$a} } keys %hash) {
	printf "%4d %s\n", $hash{$key}, $key;
	}

根据 Hash 的值字符串排序	foreach $key (sort { $hash{$a} cmp $hash{$b} } keys %hash) {
	printf "%4d %s\n", $hash{$key}, $key;
	}

values %hash 将生成值的列表，得到的顺序与 keys %hash 相同。Perl5 允许对值直接进行修改：

for (@hash{keys %hash}) {s/foo/bar/g} #老方式

for (values %hash) { s/foo/bar/g} #新方式

each %hash 将遍历关联数组的 key/value 对，一次一对。在使用 DBM 时，最好使用 each

而不是 keys 和 values，否则将一次将所有数据读入内存。each 得到的(key, value)顺序与 keys 和 values 相同。另外，应注意不要在遍历时删除关联数组的值，但是例外是在删除 ecch() 刚刚返回的项总是安全的，就是说，以下代码可以正常工作：

```
while (($key, $value) = each %hash) {
    print $key, "\n";
    delete $hash{$key};     #This is safe!
}
```

测试%hash 中某 key 是否存在，应注意分别 defined $hash{$key}与 exists $hash{$key}的分别。

2.6 引用(References)

2.6.1 Perl 引用简介

Perl5 的重要新特性之一是能够处理多维数组与嵌套关联数组等复杂的数据结构。这一特性是由于引进引用后获得的，使用引用是 Perl 处理复杂数据结构的秘诀。这里说的引用现在叫硬引用(Hard Reference)，相对于 Perl4 就存在的符号引用(Symbolic Reference, 也叫 Soft Reference)。硬引用是实在的引用。在 Perl4 中（与其借鉴的 AWK 类似），数组与关联数组的值必须是标量，为了在不改变最初设计的基础仍能使用复杂的数据结构，其解决方案是使用引用。引用是指向一个数组或关联数组或其它东西的标量。

Perl 引用的使用是简单的。只有一个原则：Perl 不会自动引用或反引。当标量是一个引用时，它总是像一个普通标量一样。它不会自动神奇地变成一个数组、关联数组或者子函数；用户必须显式地告诉 Perl 这样做，通过反引。

2.6.2 创建引用

创建引用的一种方式是使用反斜杠(\)，这一操作符类似于 C 的取址操作符(&)。比如：

```
$scalarref = \$foo;
$constref = \186_282.42;
$arrayref = \@ARGV;
$hashref = \%ENV;
$coderef = \&handler;
$globref = \*STDOUT;
```

创建引用的另一方法是创建无名数组与关联数组：创建无名数组使用[ITEMS]方式，创建无名关联数组使用{ ITEMS }。无名子程序使用不带子程序名称的 sub 语句。例如：

```
$aref =  [ 1, “foo”, undef, 13 ];
$href =  { APR => 4, AUG => 8 };
$aref = [ 1, 2, 3 ]; #相当于@array = (1, 2, 3);  $aref = \@array;
$coderef = sub { print “Boink!\n” };
```

还有一种创建引用的方式是使用 *foo(THING)的方式：

```
$scalarref  =  *foo{SCALAR};
$arrayref   =  *ARGV{SCALAR};
$hashref    =   *ENV{HASH};
$coderef    =  *handler{CODE};
$ioref      =  *STDIN{IO};
$globref    =  *foo{GLOB}
$formatref  = *foo{FORMAT}
```

说明：

- 引用一旦创建，就可以如其它标量一样使用；
- 无名数组与关联数组的创建方式可以组合使用，生成 Aarray of Arrays, Array of Hashes, Hash of Arrays， Hash of Hashes 等复杂的数据结构，下一章专门用来讨论这些数据结构。
- 适当类型的引用在反引假定它存在时可以自动生成(Autovivification)
- @list = (\$a, \@b, \%c)等价于 @list = \($a, @b, %c)
- \(@foo)不同于\@foo，前者是对@foo 内容的引用，而不是@foo 本身
- 子程序可以返回引用，但是需要小心，因为花括号{ }已经有其它使用方式。

  ```
  sub hashem {      { @_ }   } # 错误！实际返回@_，要返回引用，请使用下面的方式：
  sub hashem {      +{ @_ } } #OK
  sub hashem { return { @_ } } #OK
  ```

- 对象的构造函数也通常返回引用：

  ```
  $objref = Doggie::->new(Tail => ‘short’, Ears => ‘long’);
  $objref = new Doggie:: Tail => ‘short’, Ears => ‘long’;
  ```

2.6.3 使用引用

规则一：使用数组和关联数组引用时，总是可以在原来使用数组或关联数组名的地方，使用花括号{}内的数组引用或关联数组引用。例如：

数组或关联数组用法	引用用法	作用
@a	@{$aref}	数组
reverse @a	reverse @{$aref}	反转数组

$a[3]	${$aref}[3]	数组元素
$a[3] = 17	${$aref}[3] = 17	给数组元素赋值
%h	%{$href}	关联数组
keys %h	keys %{$href}	得到关联数组的键
$h{‘red’}	${$href}{‘red’}	关联数组的一个元素
$h{‘red’} = 17	${$href}{‘red’} = 17	给关联数组的一个元素赋值

规则一可以处理任何需求，但是实际用起来可能比较繁琐。对于常见的获取某一个元素的操作，可以如下更方便的方式：

规则二： ${$aref}[3] 可以写为$aref->[3]; ${$href}{red}可写为$href->{red}。

规则三（箭头规则）：在两个下标之间，箭头是可选的。

所以， $a[1]->[2]可以写为$a[1][2]；有了这一个规则，三维数组元素可以用易读的方式$x[2][3][5]，而不是${${$x[2]}[3]}[5]。

使用引用的注意事项如下：

- **$aref->[3]与$aref[3]是完全不同的：$aref[3]是与@{$aref}完全不同的数组@aref 的第四个元素。**
- **$href->{red}与$href{red}也完全不同：$href{red}是关联数组%href 的某一元素。**
- **在使用规则一时，若花括号内是原子引用，可以省略花括号，如@$aref 等价于@{$aref}; $$aref[1]等价于${$aref}[1](可是使用$aref->[1]也是一种选择)**
- **将引用赋值给另一引用并不拷贝底层数据结构：$aref2 = $aref1;将得到一个数组的两个引用。要拷贝数组，使用 $aref2 = [@{ $aref1 }]; 要拷贝关联数组，使用 $href2 = { %{ $href1 } };**
- **要看一个变量是否是一个引用，使用”ref”函数。如果它的参数是引用，将返回真值。它返回”HASH”如果是关联数组引用，返回”ARRAY”如果是数组引用。**
- **如果试图像字符串一样使用引用，将得到 ARRAY(0x80f5dec)或者 HASH(0x826afc0)之类的字符串，如果看到这样的字符串，一般说明错误地输出了引用。不过这一特征可以让我们使用 eq(使用==会更迅速)来看两个引用是否指向同一样东西。**

2.6.4 符号引用

如果引用已经定义，而不是硬引用，将被看做是符号引用。符号引用的使用方式如下：

```
$name = "foo";
    $$name = 1;                 # Sets $foo
    ${$name} = 2;               # Sets $foo
    ${$name x 2} = 3;           # Sets $foofoo
    $name->[0] = 4;             # Sets $foo[0]
```

```
@$name = ();                # Clears @foo
&$name();                   # Calls &foo() (as in Perl 4)
$pack = "THAT";
${"${pack}::$name"} = 5;    # Sets $THAT::foo without eval
```

如果只想使用硬引用，使用 use strict ‘refs’; 如果要允许使用符号引用，使用 no strict ‘refs’; 语句。

符号引用的使用注意事项：

- 只有全局变量对符号引用是可见的，所以以下代码将输出 10 而不是 20

```
local $value = 10;
    $ref = "value";
    {
        my $value = 20;
        print $$ref;
    }
```

- 注意以下两行代码的区别：

  ${identifier}　　#等同于$identifier

  ${“identifier”}　#符号引用，在 use strict ‘refs’下将报错。

- Perl 继承了 Shell 对变量的使用方式，所以，

  ```
  $push = "pop on ";
  print ${push}over;
  ```

  将输出 pop on over

  可以用$hash{ aaa }{ bbb }{ ccc }来代替${ "aaa" }{ "bbb" }{ "ccc" }。

  若要使用$hash{ shift }可以用$hash{ shift() }或者$hash{ +shift }或者$hash{ shift @_ };

2.6.5 垃圾回收与弱引用

Perl 使用一个基于引用的垃圾回收器。如果使用循环引用，那么普通的垃圾回收机制将难以回收对象。比如{ my ($a, $b); $a=\$b; $b=\$a; }。解决这一问题的方法是使用弱引用(Weak references)，这需要 CPAN 的 WeakRef 包配合。不过弱引用只是实验特性。

2.7 数据结构

Perl5 之前的 Perl 只能处理处理简单的数据结构（标量，一维 Array、一维 Hash），它可以像 AWK 模拟多维结构。Perl5 引进了引用的概念，Perl 所能处理的数据结构一下子复杂了起来。

但是应记住 Perl 的 Array 和 Hash 本质上都是一维的，它们只能装标量值（字符串、数值或引用），它们不能直接装其它 Array 或 Hash。Array of Arrays 实际上是 Array of references to arrays，Array of Hashes 实际上是 Array of references to hashes，余此类推。最常见的复杂数据结构有：Array of Arrays、Hash of Arrays、Array of Hashes、Hash of Hashes、Hash of Subroutines。尽管还可以构建更复杂的结构，但是一般不建议使用（过于复杂）。

2.7.1 Arrays of Arrays

Array of Arrays 又称为二维数组或矩阵。定义二维数组的方法：

```
@AoA = (
            [“fred”,    “barney” ],
            [“george”, “jane”, “elroy”],
            [“homer”, “marge”, “bart” ],
          );
 print $AoA[2][1];

 $ref_to_AoA = [
                    [“fred”,    “barney” ],
                    [“george”, “jane”, “elroy”],
                    [“homer”, “marge”, “bart” ],
                  ];
   print $ref_to_AoA->[2][1];
```

产生 Array of Arrays 的方法：

```
# 从文件读取
while ( <> ) {
      push @AoA, [ split ];
}

# 使用引用
while ( <> ) {
      push @$ref_to_AoA, [ split ];
}

# 调用函数
```

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

# 加入已存在的行
push @{ $AoA[0] }, “Wilma”, “betty”;
```

使用多维数组的方法：

```
$AoA[0][0] = “Fred”;                #使用一个元素
$AoA[1][1] =~ s/(\w)/\u$1/;         #另一个元素

# 使用引用打印所有元素
for $aref (@AoA ) {
      print “\t [ @$aref ], \n”;
}

# 使用指标打印
for $i    ( 0 .. $#AoA ) {
      print “\t [ @{$AoA[$i] } ], \n”;
 }

 # 使用双指标打印
for $i    ( 0 .. $#AoA ) {
      for $j ( 0 .. $#{ $AoA[$i] } ) {
            print “elt $i $j is $AoA[$i][$j]\n”;
      }
 }
```

使用注意事项：

- **不要企图用 print “@AoA”的方式来打印整个多维数组。**
- **以下创建多维数组的方式是错误的：**

```
for $i ( 0 .. 10) {
      @array = somefunc($i);
      $AoA[$i] = @array;          # 错误！等价于$AoA[$i] = scalar @array;
                                  # 使用 $AoA[$i] = \@array; 同样错误
```

```
}
```

- **正确的方式有如下几种：**

```
$AoA[$i] = [ @array ];          # 推荐方式

for $i ( 0 .. 10) {             # 与上面错误方式只有微妙差别，却是正确的！
    my @array = somefunc($i);
    $AoA[$i] = @array;
}

@{ $AoA[$i] } = @array;                                  # 正确，却有点难懂
```

2.7.2 Hash of Arrays

定义与生成 Hash of Arrays 的方式与使用多维数组类似，其主要差别在于使用上，因为使用 Hash of Arrays 存在输出排序的问题。

定义 Hash of Arrays 的方式：

```
%HoA = (
    flintstones => [ “fred”, “barney” ],
    jetsons => [“george”, “jane”, “elroy” ],
    Simpsons => [“homer”, “marge”, “bart” ],
);
```

增加另一数组到 Hash 中：

```
$HoA{teletubbies} = [“tinky winky”, “dipsy”, “laa-laa”, “poo” ];
```

增加一个新元素到已存在的数组中：

基本思想： $HoA{$s} = [@array];

```
# 从文件读入
# flintstones: fred  barney  wilma  dino
push @{ $HoA{flintstones} }, “wilma”, “pebbles”;
```

产生 Hash of Arrays 方法：

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

```
}

# 读入文件格式同上
while ( $line = <> ) {
    ($who, $rest) = split /:\s*/, $line, 2;
    @fields = split ' ', $rest;
    $HoA{$who} = [ @fields ];
}

# 调用函数
for $group    ( "Simpsons", "jetsons", "flintstones" ) {
      $HoA{$group} = [ get_family($group) ];
}
```

使用 Hash of Arrays:

```
# 使用一个元素
 $HoA{flintstones}[0] = "Fred";
 # 另一个元素
 $HoA{simpsons}[1] =~ s/(\w)/\u$1/;
 # 打印所有
 foreach $family ( keys %HoA ) {
     print "$family: @{ $HoA{$family} }\n"
 }
 # 带指标打印所有
 foreach $family ( keys %HoA ) {
     print "family: ";
     foreach $i ( 0 .. $#{ $HoA{$family} } ) {
         print " $i = $HoA{$family}[$i]";
     }
     print "\n";
 }
 # 按数组元素的个数排序进行打印
 foreach $family ( sort { @{$HoA{$b}} <=> @{$HoA{$a}} } keys %HoA ) {
     print "$family: @{ $HoA{$family} }\n"
 }
```

```
# 按打印数组元素的个数及名字排序进行打印
foreach $family ( sort {
                        @{$HoA{$b}} <=> @{$HoA{$a}}
                                    ||
                                $a cmp $b
        } keys %HoA )
{
    print "$family: ", join(", ", sort @{ $HoA{$family} }), "\n";
}
```

2.7.3 Arrays of Hashes

Arrays of Hashes 相比其它几种数据结构不是特别常用。它在需要访问一系列记录而每一条记录包含 key/value 对时适用。

Arrays of Hashes 的形成：

```
@AoH = (
        {
            Husband => “barney”,
            Wife    => “betty”,
            Son     => “bam bam”,
         },
         {
            Husband => “george”,
            Wife    => “jane”,
            Son     => “elroy”,
         },
      );
```

增加另一个元素：

```
push @AoH, { Husband => “fred”, wife => “wilma”,   Son => “pebbles” };

# 从文件读取
# format: Lead=fred FRIEND=barney
while ( <> ) {
    $rec = {};
    for $field ( split ) {
```

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

# no temp
while ( <> ) {
        push @AoH, { split /[\s=]+
  };

#  增加 key/value 到一个元素
$AoH[0]{pet} = “dino”;

访问 Arrays of Hashes 元素：
$AoH[0]{Husband} = “fred”;
$AoH[1]{Husband} =~ s/(\w)/\u$1/;
#  打印所有数据
for $href ( @AoH ) {
        print “{ “;
        for $role (keys %$href ) {
            print “$role=$href->{$role} “;
        }
        print “}\n”;
}

#  带指标打印
for $i ( 0 .. $#AoH ) {
        print “$i is { “;
        for $role ( keys %{ $AoH[$i] } ) {
            print “$role=$AoH[$i]{$role} “;
        }
        print “}\n”;
}
```

2.7.4 Hashes of Hashes

多维的 Hash 是 Perl 嵌套结构中最灵活的，应记住 Hash 的键不会以已知顺序存储。若需要输出，应使用 sort 函数。

产生 Hash of Hashes :

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

# 从文件读入
# flintstones: lead=fred pal=barney wife=Wilma pet=dina
while ( <> ) {
    next unless s/^(.*?):\s*//;
    $who = $1;
    for $field ( split ) {
        ($key, $value) = split /=/, $field;
        $HoH{$who}{$key} = $value;
     }
}

# calling a function  that returns a key,value hash
for $group ( "simpsons", "jetsons", "flintstones" ) {
```

```
    $HoH{$group} = { get_family($group) };
}
```

使用和打印 Hash of Hashes:

```
# one element
$HoH{flintstones}{wife} = "wilma";
# another element
$HoH{simpsons}{lead} =~ s/(\w)/\u$1/;

# print the whole thing
foreach $family ( keys %HoH ) {
    print "$family: { ";
    for $role ( keys %{ $HoH{$family} } ) {
        print "$role=$HoH{$family}{$role} ";
    }
    print "}\n";
}

# print the whole thing  somewhat sorted
foreach $family ( sort keys %HoH ) {
    print "$family: { ";
    for $role ( sort keys %{ $HoH{$family} } ) {
        print "$role=$HoH{$family}{$role} ";
    }
    print "}\n";
}

# print the whole thing sorted by number of members
foreach $family ( sort { keys %{$HoH{$b}} <=> keys %{$HoH{$a}} } keys %HoH ) {
    print "$family: { ";
    for $role ( sort keys %{ $HoH{$family} } ) {
        print "$role=$HoH{$family}{$role} ";
    }
    print "}\n";
}
```

```
# establish a sort order (rank) for each role
$i = 0;
for ( qw(lead wife son daughter pal pet) ) { $rank{$_} = ++$i }
# now print the whole thing sorted by number of members
foreach $family ( sort { keys %{ $HoH{$b} } <=> keys %{ $HoH{$a} } } keys %HoH ) {
    print "$family: { ";
    # and print these according to rank order
    for $role ( sort { $rank{$a} <=> $rank{$b} }  keys %{ $HoH{$family} } ) {
        print "$role=$HoH{$family}{$role} ";
    }
    print "}\n";
}
```

2.7.5 Hashes of Functions

有时候 Hashes of Functions 是有用的，例子如下：

```
%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 $cmd}->() }
else { warn “Uknown command: `$cmd`; Try `help` next time\n” }
```

3 操作符（Operators）

3.1 概述

从数学角度来看，操作符是特殊形式的普通函数；从语言学的角度来看，操作符是不规则动词，而各种数据类型则是名词。

根据带参数的多少，操作符可分为一元操作符（unary）、二元操作符(binary)、三元操作符(ternary)以及多元操作符(主要指 list operators，可跟任何多参数)。
操作符的性质包括元数（arity）、优先级（precedence）和附着性（associativity）。附着性包括左附、右附和不附着。

3.2 Perl 操作符一览

按照优先级从高到低，Perl 的操作符列表如下：

Name	**Associativity**	**Arity**	**Precedence Class**
Terms, and list operators(leftward) (项与左附列表运算符)	None	0	Term 指变量、引号运算符、括号运算符(（）[] {})等
The Arrow Operator (箭头操作符)	Left	2	->
Autoincrement and Autodecrement (自增自减)	None	1	++ --
Exponentiation (乘方)	**Right**	**2**	**
Ideographic Unary Operators (表意一元操作符)	Right	1	! ~ \ and unary + and -
Binding Operators (捆绑操作符)	Left	2	=~ !~
Multiplicative Operators (乘操作符)	Left	2	* / % x
Additive Operators (加操作符)	Left	2	+ - .
Shift Operators (移位操作符)	Left	2	<< >>
Name Unary and File Test Operators (有名一元和文件测试操作符)	**Right**	**0,1**	**-f my do rmdir return 等**

Relational Operators (关系运算符)	None	2	< > <= >= lt gt le ge
Equality Operators (等号操作符)	None	2	== != <=> eq ne cmp
Bitwise Operator (位操作符)	Left	2	&
Bitwise Operators (位操作符)	Left	2	| ^
Logical Operator (逻辑操作符)	Left	2	&&
Logical Operator (逻辑操作符)	Left	2	||
Range Operators (范围操作符)	None	2	.. …
Conditional Operator (条件操作符)	Right	3	?:
Assignment Operators (赋值操作符)	Right	2	= += -= *= and so on
Comma Operators (逗号操作符)	Left	2	, =>
List Operators(Rightward) (右附列表操作符)	Left	0+	List operators (rightward)
Logical not (逻辑非)	Right	1	not
Logical and (逻辑与)	Left	2	and
Logical or xor (逻辑或与异或)	Left	2	or xor

3.3 各种操作符使用说明

3.3.1 项与左赋列表操作符

项在 Perl 的优先级最高。项包括变量、引号操作符、在各种括号内的表达式和参数用括号括起来的函数。注意 chdir($foo) *20 表示 （chdir $foo）* 20， chdir +($foo) * 20 表示 chdir

($foo * 20)。另外要注意括号的附着性，比如 print $foo, exit 可能不是你所需要的,但是 print($foo), exit 却达到目的！另外： print ($foo*255)+1, “\n”也不能达到打印($foo*255)+1 的目的！所以一定要小心！如果有疑惑，可以加括号消除歧义。

3.3.2 箭头操作符

像 C/C++一样，Perl 的箭头操作符（->）指反引（dereference）：
右边是[…]时，左边必须是数组引用；右边是{…}时，左边必须是 Hash 引用；左边是（…）时，左边必须是子程序引用或者一个对象（a bless reference）或类名(a package name)。比如,

```
$aref->[42]                    #an array dereference
$href->{“corned beef”}         #a hash dereferenceu
$sref->(1,2,3)                 #a subroutine dereference
$yogi = Bear->new(“Yogi”);     #a class method call
$yogi->swipe($picnic);         #an object method call
```

3.3.3 自增自减

Perl 的++和--操作符与 C 的操作一致。但是 Perl 的++却有一个特殊的功能：当它对/^[a-zA-Z]*[0-9]*$/形式的字符串操作时，它是带进位的字符串自增，比如：

```
print ++($foo = ‘99’);      #prints ‘100’
print ++($foo = ‘a9’);      #prints ‘b0’
print ++($foo = ‘Az’);      #prints ‘Ba’
print ++($foo = ‘zz’);      #prints ‘aaa’
```

3.3.4 乘方

C 中没有乘方运算符，Perl 的乘方运算符（**）是从 FORTRAN 中借鉴来的，使用 C 的 pow(3)函数来实现。注意乘方运算符是右附的，所以-2**4 是指-（2**4）而不是(-2)**4。

3.3.5 表意一元操作符

这些操作符基本都与求反相关：
！是逻辑求反，相当于 not， 但 not 的优先级更低
- 如果操作数是数值则是算术负。**注意如果操作数是一个标识符，将返回一个负号与标识符的连接值！（如 – “abc” 是”-abc”, - “-abc”是”+abc”）**

~ 位求反。注意此操作是不可移植的。 ~123 在 32 位机器与 64 位机器上结果是不同的！

+ 正号无论对数值与字符串都不起作用。**但是正号有特别的作用是防止括号与函数名连接。如 print +($foo*255)+1, “\n”;**

\ 对任何跟随的操作数生成一个引用

3.3.6 捆绑操作符

这些操作将一个字符表达式与模式匹配、替换或转换捆绑，否则这些操作将作用在$_变量上。注意$_ =~ $pat 等价于$_ =~ /$pat/，$string !~ /pattern/等价于 not $string =~ /pattern/。如果操作作用在替换上，将返回替换的次数，一般返回是否匹配成功。

3.3.7 乘操作符

*（乘）、/(除)、%(求模)的基本行为与 C 一致。关于此类操作符需说明以下几点：

- **/默认是进行浮点计算，除非 use integer（这一点与 C 不同！）。即$i=23; print $i/2 不是打印 12，而是 11.5。**
- **%将在求模前将操作数转换为整数。**
- **x 是 Perl 特有的重复操作符！非常实用！如 print ‘-‘ x 80；将打印一行-；@ones =(5) x @one 将所有@ones 的元素设为 5。**

3.3.8 加操作符

注意事项：

- **Perl 的+（加）-（减）操作符若作用在字符串上，会将字符串转化为数值！**
- **Perl 有另外的.操作符进行字符串连接。Perl 不会在字符串之间加空格。如果有多个字符串需要连接，可以使用 join 函数。**
- **Perl 的字符串连接操作符类似 AWK 的空格。其+-将字符串转化为数值也是从 AWK 借鉴来的。**

3.3.9 移位操作符

左移（<<）右移（>>）操作符行为与 C 一致，操作数必须为整数，注意结果依赖于机器表示整数的位数。

3.3.9 有名一元和文件测试操作符

Perl 的有些函数实际是一元操作符，这些函数包括：

-X (file tests)	gethostbyname	localtime	return
alarm	getnetbyname	lock	rmdir
caller	getpgrp	log	scalar
chdir	getprotobyname	lstat	sin
chroot	glob	my	sleep
cos	gmtime	oct	sqrt
defined	goto	ord	srand
delete	hex	guotemeta	stat
do	int	rand	uc
eval	lc	readlink	ucfirst
exists	lcfirst	ref	umast
exit	length	require	undef

文件测试操作符及其意义如下：

Operator	**Meaning**
-r	File is readable by effective UID/GID
-w	File is writable by effective UID/GID
-x	File is executable by effective UID/GID
-o	File is owned by effective UID
-R	File is readable by real UID/GID
-W	File is writable by real UID/GID
-X	File is executable by real UID/GID
-O	File is owned by real UID
-e	File exists
-z	File has zero size
-s	File has nonzero size (returns size)
-f	File is a plain file
-d	File is a directory
-l	File is a symbolic link
-p	File is a named pipe(FIFO)
-S	File is a socket

-b	File is block special file
-c	File is a character special file
-t	Filehandle is opened to a tty
-u	File has setuid bit set
-g	File has setgid bit set
-k	File has sticky bit set
-T	File is a text file
-B	File is binary file (opposite of –T)
-M	Age of file (at startup) in days since modification
-A	Age of file (at startup) in days since last access
-C	Age of file (at startup) in days since inode change

使用注意事项：

- 一元有名操作符的默认参数一般为$_，next if length < 80 必须用 next if length() < 80；
- 一元有名操作符的优先级比一些二元操作符高，所以注意 sleep 4 | 3 的意义；
- 文件测试时_表示延续上一测试（包括 stat 函数）的文件，比如 print “Can do.\n” if –r $a || -w _ || -x _;

3.3.10 关系操作符

应注意 Perl 有两套比较操作符，针对数值与针对字符串的，两者不可混用！这是 Perl 相对于 C 特别之处！Perl 的关系操作符如下：

Numeric	String	Meaning
>	gt	Greater than
>=	ge	Greater than or equal to
<	lt	Less than
<=	le	Less than or equal to
==	eq	Equal to
!=	ne	Not equal to
<=>	cmp	Comparison, with signed result

另外需注意：后三套操作符的优先级比前四套低；还有<=>和 cmp 是 Perl 特有的关系操作符，如果左操作数比右操作数小它返回-1，如果相等返回 0，如果左操作数大于右操作数返回+1。<=>操作符被称为 “ 宇宙飞船 ” 操作符（spaceship operator）。

3.3.11 位操作符

Perl 的位操作符&(AND)、|（OR）、^(XOR)针对数值与字符串时有不同操作：**如果任一操作数是数，两个操作数都被转化为整数；如果两个操作数都是字符串，对这个字符串的每一位做位操作。如：**

print “12345**”** & “2345**6”** **#prints 02044**

print 12345 & 23456; #prints 4128

3.3.12 C 风格逻辑操作符

Perl 提供与 C 类似的&&（logical AND）和||(logical OR)操作符。但是应注意：**Perl 的&&与||的返回值的方式与 C 不同：它不是返回 0 或 1，而是返回最后一个计算的值！**

所以可以通过如下方式赋值：

$home = $ENV{HOME} || $ENV{LOGDIR} || (getpwuid($<))[7] || die “You’are homeless!\n”;

$and = “abc” && “def” ; #$and 值为 def

$or = “abc” || “def”; #$or 值为 abc

3.3.13 范围操作符

范围操作符..实际是两个完全不同的操作符，针对不同环境意义不同：

- **在标量环境中，返回一个 Boolean 值，它模拟 sed 和 awk 的逗号操作符：if (2..10) {print;}**
- 在标量环境，（3..n）返回 1、2、…n-2E0,最后一个量附加了额外的 “ E0”。
- ..与…的差别是…操作符在比较左值为真后不对右值进行比较：所以 if(3..3) {print:}将不打印任何行，而 if(3…3){print;}将打印第 3 行之后的所有行。
- **在列表(List)环境中，返回从左值到右值的一个列表，注意左右值可以是数值也可以是字符串，如(‘aa’..’zz’)或（2..10）。**

3.3.14 条件操作符

条件操作符的格式是：COND ? THEN : ELSE

比如，以下语句是很常见的：

printf “I have %d camel%s.\n” $n, $==1 ? ”” : “s”;

另外，注意条件操作符的优先级。下面语句可以判别某一年是否闰年：

$leapyear =

$year % 4 == 0

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

3.3.14 赋值操作符

Perl 提供 C 的所有的赋值操作符，同时增加了一些自己特有的。有很多赋值操作符，如

**= x= %= &&= ||= += -= ^=

TARGET OP= EXPR 等价于 TARGET = TARGET OP EXPR;

注意事项：

- TARGET 只求值一次，所以$var[$a++] += $value 中$a 只增一次
- **与 C 不同的是，Perl 的赋值表达式产生一个有效的左值(lvalue)。**

比如： （$a += 2） *= 3;等价于$a += 2; $a *= 3;

这个语句也很常见： （$new = $old） =~ s/foo/bar/g;

3.3.15 逗号操作符

注在标量环境中，它计算左边的所有值，扔掉，返回最右边的值。注意以下语句结果的不同：

```
$a = (1, 3);        #$a=3
($a) = (1,3)        #$a=1
@a=(1,3)            #@a=qw/1 3/;
```

=>大多数时候只是逗号操作符的同义操作符。列表操作符的优先级比逗号操作符低。

3.3.16 逻辑 and, or, not 和 xor 操作符

这四个逻辑操作符比相应 C 操作符优先级低。所以

unlink “alpha”, “beta”, “gamma” or gripe(), next LINE; 与

unlink(“alpha”, “beta”, “gamma”) || (gripe(), next LINE);等价。

但应注意 or 与||不总能互换，比如：

$xyz = $x || $y || $z; 与 $xyz = $x or $y or $z；意义是不同的！

xor 操作在 C 与 Perl 中都没有对应，因为左右两边都会计算，不会 short-circuit。

3.4 与 C 操作符的比较

3.4.1 Perl 操作符的特别之处

- 乘方（**）、比较（<=> ）、模式匹配（=~ !~）、字符串连接（.）、区域操作（ .. ...）C 没有；
- Perl 对数值和字符串有两套不同的关系（比较）操作符，应注意区分。

3.4.1 C 有 Perl 没有的操作符

Perl 没有 C 的如下操作符：取址操作符(&)、指针操作符（*，用来 dereference）、类型转换操作符((TYPE))。

说明：Perl 没有地址，所以不需要 dereference 一个地址，它的确有引用，它用$、@、%、&来反引。Perl 也有*符号，也用表示一个 typeglob。

4. 语句

Perl 程序由一系列声明(Declaration)与语句(Statement)构成。声明主要在编译期间起作用。Perl 不要求变量显式声明，它在初次使用时自动生成，如果未被赋值，在数值环境中会被作为 0，在字符串环境中会被当成""，在作为逻辑变量时将作为 false。但是 Perl 一般使用 use strict; 防止使用未定义变量。

语句按复杂程度可分为简单语句、复合语句，按逻辑关系可分为条件语句、循环语句等。

4.1 简单语句

简单语句必须以分号结束，除非它是一个块的最后语句。一个简单语句可以加以下修饰符：

if EXPR

unless EXPR

while EXPR

until EXPR

foreach LIST

比如：

$trash->take('out') if $you_love_me;

shutup() unless $you_want_me_to_leave;

kiss('me') until $I_die;

$expression++ while –e "$file$expression";

s/java/perl/ for @resumes;

print "field: $_\n" foreach split /:/, $dataline;

注意事项：

- Perl 的 if、unless 语句正常使用时后面必须是 BLOCK，哪怕只有一句话，也必须加{ }，这对于 C 程序员会觉得很别扭，但是 Perl 也提供了替代方案,就是将 if、unless 作为后修饰符。
- Perl 的 if 与 unless，while 与 until 完全起相反的功能：unless EXPR 等价于 if （! EXPR)，until EXPR 等价于 while (!EXPR)。
- Perl 的 until 语句与 C 的 do{ }until 语句的意义是不同的！Perl 的 until 是与 while 相反的。until (EXPR)相当于 while (! EXPR)。先测试后执行，而不是无条件执行。以后两个例子的输出是完全不同的：

  $i = 0; do{ print "abc"} until ($i == 0); # 会输出 abc

  $i= 0; print "abc" until($==0); # 不会输出 abc

  即 until 在没有 do 时在最开始也会对条件进行判断，以决定是否执行前面的语句。

4.2 复合语句

在一个范围内一系列语句称为一个块(BLOCK)，复合语句是由表达式与 BLOCKs 组成，表达式由项与操作符组成。复合语句又可以分为复合条件语句与复合循环语句。

4.2.1 条件语句(if/unless 语句)

条件语句的语句如下：

if (EXPR) BLOCK

if (EXPR) BLOCK else BLOCK

if (EXPR) BLOCK elsif (EXPR) BLOCK

if (EXPR) BLOCK elsif (EXPR) BLOCK … else BLOCK

unless (EXPR) BLOCK

unless (EXPR) BLOCK else BLOCK

unless (EXPR) BLOCK elsif (EXPR) BLOCK

unless (EXPR) BLOCK elsif (EXPR) BLOCK … else BLOCK

说明：

- if 与 unless 是互补的，unless ($x == 1) 等价于 if（$x != 1）或者 if (! ($x == 1)) 。
- **if/unless 语句后面都是跟 BLOCK，即必须有{ }，哪怕只有一条语句！这是 Perl 很特别的地方。但是有不用括号的方法，即使用简单 if/unless 语句。**
- 注意 Perl 使用 elsif， 试比较 C 之 else if ，好像是缺了一个字母似地！
- 变量声明的范围从声明开始之处到所有本条件语句之内，包括 elsif 和 else 在内。如：

```
if ((my $color = <STDIN> ) =~ /red/i) {
        $value = 0xff0000;
}
elsif ($color =~ /green/i) {
        $value = 0x00ff00;
}
else{
        warn “Unknown RGB component\n”;
        $value = 0x000000;
}
```

4.2.2 循环语句(while/until/for/foreach 语句)

循环语句的语句如下：

LABEL while (EXPR) BLOCK

LABEL while (EXPR) BLOCK continue BLOCK

LABEL until (EXPR) BLOCK

LABEL until (EXPR) BLOCK continue BLOCK

LABEL for (EXPR: EXPR: EXPR) BLOCK

LABEL foreach (LIST) BLOCK

LABEL foreach VAR (LIST) BLOCK

LABEL foreach VAR (LIST) BLOCK continue BLOCK

例如:

```
LINE: while (<STDIN>) {
        next LINE if /^#/;              # skip comments
        next LINE if /^$/;              # skip blank lines
        …
} continue {
        $count++;
}

LINE: while (defined($line = <ARGV>)) {
        chomp($line);
        if ($line =~ s/\\$//) {
                $line .= <ARGV>;
                redo LINE unless eof(ARGV);
        }
        # now process $line
}
```

说明：

- while/until 的进行的判断是互补的，until 也是先判断再决定是否执行语句。
- LABEL 是可选的，但是在 Perl 实践中还是经常使用，特别是循环内有 next/last/redo 语句的时候。
- 注意 for 和 foreach 关键字在实践上可以互换！但两者概念上是不同的。但是系统可以分别是 for 还是 foreach 操作。
- **foreach 是直接对数组元素操作的，变量是真正数组元素的别名，这一点一定要清楚。比如@arr=(1,2,3,4,5); foreach $i(@arr){ $i++} @arr 就变成了（2，3，4，5，6）。**
- 循环控制操作符如下：

  last　LABEL　(相当于 C 的 break)

  next　LABEL　(相当于 C 的 continue)

  redo　LABEL

  LABEL 是可选的，如果省略，表示最内层循环。

  应明白： redo/last 后不执行 continue BLOCK。
- **在结构化编程的最初阶段，有些人坚持循环与子程序只能有一个入口和一个出口。**

One-entry 是一个好的思想，但是 one-exit 通常是不太现实的，所以 Perl 建议按照需要退出循环。

- last/redo/next 可以用于 BLOCK，但是 eval{}，sub{}，do{}却不属于循环 BLOCK，同样 if/unless 中也不能直接使用。要使用的话，必须再加一层{}。例如：

```
if (/pattern/) {{
        last if /alpha/;
        last if /beta/;
        last if /gamma/;
        # do something her only if still in if()
}}

do {{
        next if    $x == $y;
        # do something here
}} until $x++ > $z;

{
        do {
                last if $x = $y ** 2;
                # do something here
        } while $x++ <= $z;
}
```

注意：如果仍然用 do {{ last if $x = $y ** 2 }} where $x++ <= $z; 那么 last 起不到作用！后面的 while 继续运行。

4.2.3 分支语句

Perl 没有正式的 switch 和 case 语句。这是因为 Perl 并不需要，可以用以下方式达到同样目的：

```
SWITCH: {
        if    (/^abc/) { $abc = 1; last SWITCH; }
        if    (/^def/) { $def = 1; last SWITCH; }
        if    (/^xyz/) { $xyz = 1; last SWITCH; }
        $nothing = 1;
}
```

```
SWITCH: {
        /^abc/ && do { $abc = 1; last SWITCH; }
        /^def/ && do { $def = 1; last SWITCH; }
        /^xyz/ && do { $xyz = 1; last SWITCH; }
        $nothing = 1;
}
```

5. 子程序（函数）

5.1 子程序简介

子程序又称为函数。在 Perl 里子程序与函数是同一个概念。Perl 允许用户自定义子程序，它们可以在主程序的任何位置，从其它文件中通过 do/require/use 关键字引入，或者使用 eval 在运行时生成，或者通过无名子程序的引用使用。

Perl 的输入、输出模型非常简单：所有的输入的参数都是转化为一个标量的 List，所有的函数都返回一个 List 列表。Perl 中，数组变量@_是一个特殊的系统变量。如果函数调用时后面跟着一个用括号括起来的列表，则在函数调用期间该列表将被自动分配给一个以@_命名的特殊变量。@_是一个局部变量，它的值只在它出现的函数中有定义。return 用来返回参数，如果 return 后面不带参数，在 List 环境中将返回空列表，在标量环境中将返回 undef，在 Boolean 环境中将返回 void。

声明子程序：

```
sub NAME;                   # A "forward" declaration.
sub NAME(PROTO);            #  ditto, but with prototypes
sub NAME : ATTRS;           #  with attributes
sub NAME(PROTO) : ATTRS;    #  with attributes and prototypes
```

定义子程序：

```
sub NAME BLOCK              # A declaration and a definition.
sub NAME(PROTO) BLOCK       #  ditto, but with prototypes
sub NAME : ATTRS BLOCK      #  with attributes
sub NAME(PROTO) : ATTRS BLOCK #  with prototypes and attributes
```

定义无名子程序：

```
$subref = sub BLOCK;                # no proto
$subref = sub (PROTO) BLOCK;        # with proto
$subref = sub : ATTRS BLOCK;        # with attributes
$subref = sub (PROTO) : ATTRS BLOCK; # with proto and attributes
```

从模块中引入子程序：

```
use MODULE qw(NAME1 NAME2 NAME3);
```

使用子程序：

```
NAME(LIST);    # & is optional with parentheses.
NAME LIST;     # Parentheses optional if predeclared/imported.
&NAME(LIST);   # Circumvent prototypes.
&NAME;         # Makes current @_ visible to called subroutine.

&$subref(LIST)
$subref->(LIST)
&$subref
```

注意：有带&与不带&的使用方式，Perl5 后建议不用&。但是若 NAME 不带参数不带任何参数时，及需要将通过函数引用使用函数时（ &$subref() or &{$subref}()）则不可省略（但是也可以用$subref->()代替)。

例如：

```
sub max {
        my $max = shift(@_);
        foreach $foo (@_) {
            $max = $foo if $max < $foo;
        }
        return $max;
    }

$bestday = max($mon,$tue,$wed,$thu,$fri);
```

```
@common = inter( \%foo, \%bar, \%joe );
sub inter {
    my %seen;
    for my $href    (@_) {
```

```
        while (my $k = each %$href ) {
            $seen{$k}++;
        }
    }
    return grep { $seen{$_} == @_ } keys %seen;
}
```

5.2 函数原型与属性

Perl 有限地允许使用函数原型，函数原型的使用说明如下：

- 函数原型只有在&字符忽略的时候影响函数的解释；
- 反斜杠(\)原型字符代表一个实际的参数；
- 分号(;)分割必选参数与可选参数；
- *允许子程序接受文件句柄作为参数

使用函数原型的例子如下：

定　义	使　用
sub mypush (\@@)	mypush(@array, $v)
sub mylink ($$)	mylink $old, $new
sub myreverse (@)	myreverse $a, $b, $c
sub myjoin ($@)	mypop @array
sub mysplice (\@$$@)	mysplice @array, @array, 0, @pushme
sub mykeys (\%)	mykeys %{$hashref}
sub myindex ($$;$)	myindex &getstring, “substr” myindex &getstring, “substr”, $start
sub mysyswrite (*$;$$)	mysyswrite OUTF, $buf mysyswrite OUTF, $buf, length($buf)-off, $off
sub myopen (*;$@)	myopen HANDLE myopen HANDLE, $name myopen HANDLE, “-|”, @cmd
sub mygrep (&@) sub myrand ($) sub mytime ()	mygrep { /foo/ } $a, $b, $c myrand 42 mytime

Perl 函数也可以定义属性，三个标准属性如下：

- locked : 只允许一个进程进入
- method: 表明函数是一个方法。等价于 use attributes __PACKAGE__, \&foo, 'method'; #foo 为函

数名

- lvalue:　允许函数作为左值使用

5.3 Perl 内部函数

按函数的分类，Perl 提供的主要内部函数如下：

类别	函数
标量处理	chomp, chop, chr, crypt, hex, index, lc, lcfirst, length, oct, ord, pack, q//, qq//, reverse, rindex, sprintf, substr, tr//, uc, ucfirst, y///
正则表达式与模式匹配	m//, pos, qr//, quotemeta, s///, split, study
数值函数	abs, atan2, cos, exp, hex , int, log, oct, rand, sin, sqrt, srand
数组(Array)处理	pop, push, shift, splice, unshift
列表(List)处理	grep, join, map, qw//, reverse, sort, unpack
关联数组(Hash)处理	delete, each, exists, keys, values
输入输出	binmode, close, closedir, dbmclose, dbmopen , die, eof, fileno, flock, format, getc, print, printf, read, readdir, readpipe, rewinddir, seek, seekdir, select(ready file descriptors), syscall, sysread, sysseek, syswrite, tell, telldir, truncate, warn, write
固定长度数据与记录	pack, read, syscall, sysread, sysseek, syswrite, unpack, vec
文件句柄、文件与目录	chdir, chmod, chown, chroot, fcntl, glob, ioctl, link, lstat, mkdir, open, opendir, readlink, rename, rmdir, select(ready file descriptors), select(output filehandles), stat, symlink, sysopen, umask, unlink, utime
程序控制流	caller, continue, die, do, dump, eval, exit, goto, last, next, redo, return, sub, wantarray
范围	caller, import, local, my, no, our, package, use
杂项	defined, dump, eval, formline, lock, prototype, reset, scalar, undef, wantarray
进程与进程组	alarm, exec, fork, getpgrp, getppid, getpriority, kill, pipe, qx//, setpgrp, setpriority, sleep, system, times, wait, waitpid
库模块	do, import, no, package, require, use
类与对象	bless, dbmclose, dbmopen, package, ref, tie, tied, untie, use
低层 Socket 访问	accept, bind, connect, getpeername, getsockname, getsockopt, listen, recv, send, setsockopt, shutdown, socket, socketpair
System V 进程间通信	msgctl, msgget, msgrcv, msgsnd, setmctl, semget, semop, shmctl,

	shmget, shmread, shmwrite
获取用户和组信息	dengrent, endhosten, endnetent, endpwent, getgrent, getgrgid, getgrnam, getlogin, getpwent, getpwnam, getpwuid, getgrent, setpwent
获取网络信息	endprotoent, endservent, gethostbyaddr, gethostbyname, gethostent, getnetbyaddr, getnetbyname, getnetent, getprotobyname, getprotobynumber, getprotoent, getservbyname, getservbyport, getservent, sethostent, setnetent, setprotoent, setservent
时间	gmtime, localtime, time, times

6. 文件、目录与 I/O

句柄(Handle)是 Perl 得到数据的一种方法。Perl 提供两种句柄：文件句柄（File Handles）与目录句柄(Directory Handles)，另外提供三个默认的句柄：标准输入(STDIN)、标准输出(STDOUT)与标准错误(STDERR).

6.1 文件操作

文件的打开的格式为：

open *FILEHANDLE*, *MODE*, *LIST*

open *FILEHANDLE*, *EXPR*

open *FILEHANDLE*

实例：

```
open(INFO,      "datafile")  || die("can't open datafile: $!");
open(INFO,    "< datafile")  || die("can't open datafile: $!");
open(RESULTS, "> runstats")  || die("can't open runstats: $!");
open(LOG,    ">> logfile ")  || die("can't open logfile:  $!");
open INFO,      "datafile"   or die "can't open datafile: $!";
open INFO,    "< datafile"   or die "can't open datafile: $!";
open RESULTS, "> runstats"   or die "can't open runstats: $!";
open LOG,    ">> logfile "   or die "can't open logfile:  $!";
open(INPUT,  "-" ) or die;     # re-open standard input for reading
open(INPUT,  "<-") or die;     # same thing, but explicit
open(OUTPUT, ">-") or die;     # re-open standard output for writing
open(PRINTER, "| lpr -Plp1")    or die "can't fork: $!";
print PRINTER "stuff\n";
```

```
close(PRINTER)                  or die "lpr/close failed: $?/$!";
```

文件的打开模式的意义如下：

Mode	Read Access	Write Access	Append Only	Create Nonexisting	Clobber Existing	C mode
< *PATH*	Y	N	N	N	N	r
> *PATH*	N	Y	N	Y	Y	w
>> *PATH*	N	Y	Y	Y	N	a
+< *PATH*	Y	Y	N	N	N	r+
+> *PATH*	Y	Y	N	Y	Y	w+
+>> *PATH*	Y	Y	Y	Y	N	a+
| *COMMAND*	N	Y	n/a	n/a	n/a	
COMMAND |	Y	N	n/a	n/a	n/a	

文件的关闭：

close FILEHANDLE

文件的删除：

unlink 可以从目录中删除文件

如： unlink <*.c>; unlink glob "*.o";

文件的重命名:

rename oldname newname

文本文件的读写：

open(LOG, “/log/logfile”)

while (<LOG>) {

… chomp;

……

}

6.2 目录操作

Perl 的目录句柄与文件句柄有不同的命名空间，改变目录的方式为：

```
  chdir "/etc"   or die "cannot chdir to /etc: $!";
```

读取目录文件有几种方式：使用 glob,<>或者 readdir：

```
my @all_files = glob "*";
my @pm_files = glob "*.pm";
my @files = <FRED/*>;  ## a glob
opendir(THISDIR, " . ")            or die " Cannot open current directory $! ";
@allfiles = readdir THISDIR;
@allfiles = grep -T, readdir THISDIR;
@allfiles = grep { $_ ne ' . ' And $_ ne ' .. ' } readdir THISDIR
close THISDIR;
print " @allfiles\n ";
```

6.3 print/printf

Perl 有两个主要的输出函数：print 与 printf,其使用格式为：

print *FILEHANDLE LIST*

print *LIST*

print

printf *FILEHANDLE FORMAT, LIST*

printf *FORMAT, LIST*

*(相当于*print *FILEHANDLE* sprintf(*FORMAT, LIST)*

例如：

```
print { $OK ? "STDOUT" : "STDERR" } "stuff\n";
print { $iohandle[$i] } "stuff\n";
print {$fh}  @array1, @array2, "\n";
```

6.4 注意事项

- 有很多种打开文件的方式：

  open FH, "<$filename" or die "Cannot open filename $filename $!"; (1)

  open(FH, "<$filename") || die "Cannot open filename $filename $!"; (2)

  open(FH, '<', $filename) || die "Cannot open filename $filename $!"; (3)

  open(my $fh, '<', $filename) || die "Cannot open filename $filename $!"; (4)

  open my $fh, '<', $filename or die "Cannot open filename $filename $!"; (5)

  open(my $fh, '<', $filename) or die "Cannot open filename $filename $!"; (6)

建议使用方式（4)、(5）或(6)。注意||和 or 与前面的 open 语句最好有相当大的间隔，以表明此语句是与 open 分离的。裸词存在于当前包的符号表中，若此前该文件句柄已经存在，Perl 将无声地代替以前的文件句柄（很危险）;而两个参数的打开方式不能很好地处理以’<”>’开头的文件名。唯一需要两个参数的打开方式的时候是打开标准 I/O 流(三参数方式此时无效)：

```
open my $stdin, ‘<-‘          or croak “Cannot open stdin; $OS_ERROR”;
open my $stdout, ‘>-‘         or croak “Cannot open stdout; $OS_ERROR”;
```

- open FH, “$filename”等价于 open FH, “<$filename”。
- **以下打开方式是错误的：**

  open my $fh, ‘<’, $filename || die “Cannot open filename $filename $!”;

  因为||的优先级比 List 分隔符(,)高，所以||会附在$filename 上。以上方式(6)中的括号虽然是多余的，但是对于初学者更有可读性。
- 使用引用来作为句柄，print 时应养成加括号的习惯，即 print {$fh} LIST，这样可以避免将$fh 作为输出 LIST 一部分的错误。
- **输出函数 print 默认在 LIST 之间不会加空格或其它符号，在最后也不会自动加回车符。这与 AWK 与很大的不同（默认 OFS 为空格，ORS 为回车），有时候会觉得使用不方便（如需要输出 CSV 格式文件时），这是可以通过以下语句设置 LIST 和记录分割符：**

  local $, = “,”;

  local $\ = “\n”;

  或者可读性更好的方式：

  use English;

  local $OFS = “,”;

  local $ORS = “\n”;

7. 模式匹配

Perl 语言的最大特点，也是 Perl 作为 CGI 首选语言的最大特点，是它的模式匹配操作符。Perl 语言的强大的文本处理能力正是通过其内嵌的对模式匹配的支持体现的。模式通过创建正则表达式实现。Perl 的正则表达式与模式匹配的特点一是内嵌于语言之中，而不是通过库或函数来实现，因此使用更简便；二是比一般的正则表达式与模式匹配功能强大。

7.1 模式匹配操作符简介

操作符	意义	实例
=~	匹配(包含)	

!~	不匹配（不包含）	
m//	匹配	$haystack =~ m/needle/ $haystack =~ /needle/
s///	替换	$italiano =~ s/butter/olive oil/
tr///(y///)	转换	$rotate13 =~ tr/a-zA-Z/n-za-mN-ZA-M/
qr//	正则表达式	

使用说明：

- 注意区别记忆 Perl 的绑定操作符(=~)与 AWK 的相应操作符（AWK 的绑定匹配操作符是 ~），Perl 与 AWK 的否定匹配操作符相同(都是!~)
- 没有绑定操作符时，默认是对$_进行绑定：

  /new life/ and /new civilizations/ （对$_进行两次查找）

  s/suger/aspartame/ (对$_进行替换)

  tr/ATCG/TAGC/ (对$_进行转换)
- m//操作符前面的 m 可以省略，但是不省略可读性更好，建议不省略。
- 如果有绑定操作符=~，m//都省略也表示匹配：

  print “matches” if $somestring =~ $somepattern; 等价于

  print “matches” if $somestring =~ m/$somepattern/;
- m//, s///, tr///, qr//操作符是引用操作符，你可以选择自己的分割符（与 q//, qq//, qw//一样）：

```
$path =~ s#/tmp#/var/tmp/scratch#
if    ($dir =~ m[/bin]) {
      print “No binary directories please. \n”;
}
```

- 一个括号可与其它括号配合使用，可以用空格分开：

```
s(egg)<larva>
s(larva){pupa};
s[pupa]/imago/;
s (egg) <larva>;
```

- 如果一个模式成功匹配上，$`, $&, $’将被设置，分别表示匹配左边、匹配、匹配右边的字符串：

```
“hot cross buns” =~ /cross/;
print “Matched: <$`> $& <$’>\n”;     # Matched <hot > cross < buns>
```

- 模式模式后设置的特殊变量如下：

变量	含义
$`	匹配文本之前的文本
$&	匹配文本

$’	匹配文本之后的文本
$1、$2、$3	对应第 1、2、3 组捕获括号匹配的文本
$+	编号最大的括号匹配的文本
$^N	最后结束的括号匹配的文本
@-	目标文本中各匹配开始位置的偏移值数组
@+	目标文本中各匹配结束位置的偏移值数组
$^R	最后执行的嵌入代码的结果，如果嵌入代码结构作为条件语句的 if 部分，则不设定$^R

7.2 模式修饰符

m//, s///和 qr//都接受以下修饰符：

修饰符	意　　义
/i	进行忽略字母大小的匹配
/s	单行模式(让.号匹配换行符并且忽略过时的$*变量，点号通配模式)
/m	多行模式（让^和$匹配内含的换行符(\n)的之后与之前。如果目标字符串中没有“\n”字符或者模式中没有 ^ 或 $，则设定此修饰符没有任何效果)。（增强的行锚点模式）
/x	宽松排列和注释模式（忽略空白符（除转义空白符之外）并且允许模式中的注释）
/o	仅编译一次模式，防止运行时重编译

例如：

```
m/\w+:(\s+\w+)\s*\d+/;                # A word, colon, space, word, space, digits

m/\w+: (\s+    \w+) \s* \d+/x;        # A word, colon, space, word, space, digits

m{
    \w+;                              # Match a word and a column
    (                                 # (begin group)
        \s+                           # Match one or more spaces.
        \w+                           # Match another word
    )                                 # (end group)
    \s*                               # Match zero or more spaces
    \d+                               # Match some digits
```

```
 }x;

$/ = "";        # "paragrep" mode

while (<>) {
    while ( m{
            \b              # start at a word boundary
            (\w\S+)         # find a wordish chunk
            (
                \s+         # separated by some whitespace
                \1          # and that chunk again
            ) +             # repeat ad lib
            \b              # until another word word boundary
        }xig
    )  {
        print "dup word '$1' at paragraph $. \n";
    }
}
```

7.3 模式匹配操作符详解

7.3.1 m//操作符(匹配)

EXPR =~ m/PATTERN/cgimosx
EXPR =~ /PATTERN/cgimosx
EXPR =~ ?PATTERN?cgimosx
m/PATTERN/cgimosx
/PATTERN/cgimosx
?PATTERN?cgimosx

说明：

- 如果 PATTERN 是空字符串，最后成功执行的正则表达式将被代替使用。
- m//特殊修饰符如下：

修饰符	意　　义
/g	查找所有的匹配
/cg	在/g 匹配失败后允许继续搜索

- 在 LIST 上下文中 m//g 返回所有匹配

```
if （@perls = $paragraph =~ /perl/gi) {
    printf “Perl mentioned %d times.\n”, scalar @perls;
}
```

- ??分隔符表示一次性匹配， ‘’分隔符压制变量替换和\U 等六个转换

```
open DICT, "/usr/share/dict/words"   or die "Cannot open words: $!\n";
while (<DICT>) {
    $first = $1 if ?(^love.*)?;
    $last   = $1 if /(^love.*)/;
}
print $first, "\n";
print $last, "\n";
```

7.3.2 s///操作符(替换)

LVALUE =~ s/PATTERN/REPLACEMENT/egimosx
s/PATTERN/REPLACEMENT/egimosx
说明：

- 该操作符在字符串中查找 PATTERN，如果查找到，用 REPLACEMENT 代替匹配的子串。返回值是成功替换的次数（加/g 修饰符可能大于 1）。若失败，返回””(0)。

```
if ($lotr =~ s/Bilbo/Frodo/) { print “Successfully wrote sequel. “ }
$change_count = $lotr =~ s/Bilbo/Frodo/g;
```

- 替换部分作为双引字符串，可以使用动态生成的模式变量（$`，$&, $’, $1, $2 等）：

```
s/revision/version/release/\u$&/g;
s/version ([0-9.]+)/the $Names{$1} release/g;
```

- 如果 PATTERN 是空字符串，最后成功执行的正则表达式将被代替使用。PATTERN 和 REPLACEMENT 都需进行变量替换，但是 PATTERN 在 s///作为一个整体处理的时候替换，而 REPLACEMENT 在每次模式匹配到时替换。
- s///特殊修饰符如下：

修饰符	意　　义
/g	替换所有的匹配
/e	将右边部分作为一个 Perl 表达式（代码）而不是字符串

/e 修饰符的实例：

```
s/[0-9]+/sprintf(“%#x”, $1)/ge
s{
    version
    \s+
    (
      [0-9.]+
    )
}{
```

```
    $Names{$1}
        ? “the $Names{$1} release”
        : $&
}xge;
```

- 不替换原字符串的方式：

```
$lotr = $hobbit;
$lotr =~ s/Bilbo/Frodo/g;
($lotr = $hobbit) =~ s/Bilbo/Frodo/g;
```

- 替换数组中的每一元素：

```
for (@chapters) { s/Bilbo/Frodo/g }
s/Bilbo/Frodo/g for @chapters;
```

- 对某一字符串进行多次替换：

```
for ($string) {
        s/^\s+//;
        s/\s+$//;
        s/\s+/ /g
}
for ($newshow = $oldshow) {
        s/Fred/Homer/g;
        s/Wilma/Marge/g;
        s/Pebbles/Lisa/g;
        s/Dino/Bart/g;
}
```

- 当一次全局替换不够的时的替换：

```
# put comma in the right places in an integer
1 while s/(\d)(\d\d\d)(?!\d)/$1,$2/;
# expand tabs to 8-column spacing
1 while s/\t+/' ' x (length($&)*8 – length($`)%8)/e;
# remove (nested (even deeply nested (like this))) remarks
1 while s/\([^()]*\)//g;
# remove duplicate words (and triplicate ( and quadruplicate…))
1 while s/\b(\w+) \1\b/$1/gi;
```

7.3.3 tr///操作符(字译)

LVALUE =~ tr/SEARCHLIST/REPLACELIST/cds

tr/SEARCHLIST/REPLACELIST/cds

使用说明：

- tr///的修饰符如下：

修饰符	意　　义
/c	补替换　（Complement SEARCHLIST）
/d	删除找到未替换的字符串（在 SEARCHLIST 中存在在 REPLACELIST 中不存在的字符）
/s	将重复替换的字符变成一个

- 如果使用了/d 修饰符，REPLACEMENTLIST 总是解释为明白写出的字符串，否则，如果 REPLACEMENTLIST 比 SEARCHLIST 短，最后的字符将被复制直到足够长，如果 REPLACEMENTLIST 为空，等价于 SEARCHLIST，这种用法在想对字符进行统计而不改变时有用，在用/s 修饰符压扁字符时有用。

```
tr/aeiou/!/;              # change any vowel into !
tr{/\\\r\n\b\f. }{_};    # change strange chars into an underscore
tr/A-Z/a-z/ for @ARGV;      # canonicalize to lowercase ASCII
$count = ($para =~ tr/\n//);
$count = tr/0-9//;
$word =~ tr/a-zA-Z//s;         # bookkeeper -> bokeper
tr/@$%*//d;                    # delete any of those
tr#A-Za-z0-9+/##cd;           # remove non-base64 chars
# change en passant
($HOST = $host) =~ tr/a-z/A-Z/;
$pathname =~ tr/a-zA-Z/_/cs;    # change non-(ASCII) alphas to single underbar
```

7.4 元字符

Perl 元字符有：

\ | ( ) [{ ^ $ * + ?

正则表达式元字符的意义如下：

Symbol	Atomic	Meaning
\...	Varies	转义
...|...	No	选择

(...)	Yes	集群（作为一个单位）
[...]	Yes	字符集合
^	No	字符串开始
.	Yes	匹配一个字符（一般除换行符外）
$	No	字符串结尾(或者换行符之前)

* + ？是数量元字符，Perl 数量相关元字符意义如下：

Quantifier	Atomic	Meaning
*	No	匹配 0 或多次(最大匹配)，相当于{0,}
+	No	匹配 1 或多次(最大匹配)，相当于{1,}
?	No	匹配 1 或 0 次(最大匹配)，相当于{0,1}
{*COUNT*}	No	匹配精确 *COUNT* 次
{*MIN*,}	No	匹配最少 *MIN* 次（最大匹配)
{*MIN*,*MAX*}	No	匹配最小 *MIN* 最大 *MAX* 次(最大匹配)
*?	No	匹配 0 或多次(最小匹配)
+?	No	匹配 1 或多次(最小匹配)
??	No	匹配 1 或 0 次(最小匹配)
{*MIN*,}?	No	匹配最少 *MIN* 次（最小匹配)
{*MIN*,*MAX*}?	No	匹配最小 *MIN* 最大 *MAX* 次(最小匹配)

扩展正则表达式序列如下：

Extension	Atomic	Meaning
(?#...)	No	Comment, discard.
(?:...)	Yes	Cluster-only parentheses, no capturing.
(?imsx-imsx)	No	Enable/disable pattern modifiers.
(?imsx-imsx:...)	Yes	Cluster-only parentheses plus modifiers.
(?=...)	No	True if lookahead assertion succeeds.
(?!...)	No	True if lookahead assertion fails.
(?<=...)	No	True if lookbehind assertion succeeds.
(?<!...)	No	True if lookbehind assertion fails.
(?>...)	Yes	Match nonbacktracking subpattern.
(?{...})	No	Execute embedded Perl code.
(??{...})	Yes	Match regex from embedded Perl code.
(?(...)...|...)	Yes	Match with if-then-else pattern.
(?(...)...)	Yes	Match with if-then pattern.

说明：以上定义了向前查找(?=PATTERN)，负向前查找(?!PATTERN)，向后查找(?<=PATTERN)，负向后查找(?<!PATTERN)，条件查找等较为高级的正则表达式匹配功能，需要使用时请查阅相关资料。

字母顺序元字符意义：

Symbol	Atomic	Meaning
\0	Yes	Match the null character (ASCII NUL).
NNN	Yes	Match the character given in octal, up to \377.
n	Yes	Match *n*th previously captured string (decimal).
\a	Yes	Match the alarm character (BEL).
\A	No	True at the beginning of a string.
\b	Yes	Match the backspace character (BS).
\b	No	True at word boundary.
\B	No	True when not at word boundary.
\c*X*	Yes	Match the control character Control-*X* (\cZ, \c[, etc.).
\C	Yes	Match one byte (C char) even in utf8 (dangerous).
\d	Yes	Match any digit character.
\D	Yes	Match any nondigit character.
\e	Yes	Match the escape character (ASCII ESC, not backslash).
\E	--	End case (\L, \U) or metaquote (\Q) translation.
\f	Yes	Match the form feed character (FF).
\G	No	True at end-of-match position of prior m//g.
\l	--	Lowercase the next character only.
\L	--	Lowercase till \E.
\n	Yes	Match the newline character (usually NL, but CR on Macs).
\N{*NAME*}	Yes	Match the named char (\N{greek:Sigma}).
\p{*PROP*}	Yes	Match any character with the named property.
\P{*PROP*}	Yes	Match any character without the named property.
\Q	--	Quote (de-meta) metacharacters till \E.
\r	Yes	Match the return character (usually CR, but NL on Macs).
\s	Yes	Match any whitespace character.
\S	Yes	Match any nonwhitespace character.
\t	Yes	Match the tab character (HT).
\u	--	Titlecase next character only.
\U	--	Uppercase (not titlecase) till \E.

\w	Yes	Match any "word" character (alphanumerics plus "_").
\W	Yes	Match any nonword character.
\x{*abcd*}	Yes	Match the character given in hexadecimal.
\X	Yes	Match Unicode "combining character sequence" string.
\z	No	True at end of string only.
\Z	No	True at end of string or before optional newline.

（以上均直接 Copy 自《Programming Perl》，下面未翻译者同）

其中应注意以下经典的字符集合：

Symbol	Meaning	As Bytes	As utf8
\d	Digit	[0-9]	\p{IsDigit}
\D	Nondigit	[^0-9]	\P{IsDigit}
\s	Whitespace	[\t\n\r\f]	\p{IsSpace}
\S	Nonwhitespace	[^ \t\n\r\f]	\P{IsSpace}
\w	Word character	[a-zA-Z0-9_]	\p{IsWord}
\W	Non-(word character)	[^a-zA-Z0-9_]	\P{IsWord}

POSIX 风格的字符类如下：

Class	Meaning
alnum	Any alphanumeric, that is, an alpha or a digit.
alpha	Any letter. (That's a lot more letters than you think, unless you're thinking Unicode, in which case it's still a lot.)
ascii	Any character with an ordinal value between 0 and 127.
cntrl	Any control character. Usually characters that don't produce output as such, but instead control the terminal somehow; for example, newline, form feed, and backspace are all control characters. Characters with an ord value less than 32 are most often classified as control characters.
digit	A character representing a decimal digit, such as 0 to 9. (Includes other characters under Unicode.) Equivalent to \d.
graph	Any alphanumeric or punctuation character.
lower	A lowercase letter.
print	Any alphanumeric or punctuation character or space.
punct	Any punctuation character.
space	Any space character. Includes tab, newline, form feed, and carriage return (and a lot more under Unicode.) Equivalent to \s.
upper	Any uppercase (or titlecase) letter.
word	Any identifier character, either an alnum or underline.

xdigit	Any hexadecimal digit. Though this may seem silly ([0-9a-fA-F] works just fine), it is included for completeness.

注意：POSIX 风格字符类的使用方法,

42 =~ /^[[:digit:]]+$/ (正确)

42 =~ /^[:digit:]$/ （错误）

这里使用的模式以[[开头，以]]结束，这是使用 POSIX 字符类的正确使用方法。我们使用的字符类是[:digit:]。外层的[]用来定义一个字符集合，内层的[]字符是 POSIX 字符类的组成部分。

7.5 常见问题的正则解决方案

IP 地址：

(((\d{1,2}）|(1\d{2})|(2[0-4]\d)|(25[0-5]))\.){3}((\d{1,2}）|(1\d{2})|(2[0-4]\d)|(25[0-5]))

邮件地址：

(\w+\.)*\w+@(\w+\.)+[A-Za-z]+

(以上邮件地址正则表达式并非严格的，但是可以匹配绝大多数普通的邮件地址。

HTTP URL:

{http://([^/:]+)(:(\d+))?(/.*)?$}i

https?://(\w*:\w*@)?[-\w.]+(:\d+)?(/([\w/_.]*(\?\S+)?)?)?

C 语言注释：

/*.*?*/

8. 面向对象编程

在 Perl 中，类、包、模块是相关的，一个模块只是以同样文件名（带.pm 后缀）的一个包；一个类就是一个包；一个对象是一个引用；一个方法就是一个子程序。这里只说明其最简单的使用方法。

8.1 模块使用

以下是一个模块(Bestiary.pm)的编写方式，可以作为写一般模块的参考。

```
package      Bestiary;
require      Exporter;

our @ISA       = qw(Exporter);
our @EXPORT    = qw(camel);    # Symbols to be exported by default
our @EXPORT_OK = qw($weight);  # Symbols to be exported on request
our $VERSION   = 1.00;         # Version number
```

```
### Include your variables and functions here

sub camel { print "One-hump dromedary" }

$weight = 1024;

1;
```

（引自《Programming Perl》）

8.2 对象使用

以下例子用来构建一个 Ipregion 对象，可以使用该对象的 get_area_isp_id 方法查找一个 IP 的地区与运营商。本例可以作为写一般对象的参考。

```
package Ipregion;
use strict;

my ($DEFAULT_AREA_ID, $DEFAULT_ISP_ID) = (999999, 9);
my ($START_IP, $END_IP, $AREA_ID, $ISP_ID) = (0 .. 3);
sub new {
        my $invocant = shift;
        my $ip_region_file = shift;
        my $class = ref($invocant) || $invocant;
        my $self = [ ];                                 # $self is an reference of array of arrays

        # Read into ip region data from file
        open my $fh_ip_region, '<', $ip_region_file
                or    die "Cannot open $ip_region_file to load ip region data $!";

        my $i = 0;
        while (<$fh_ip_region>) {
              chomp;
              my ($start_ip, $end_ip, $area_id, $isp_id) = split;
              $self->[$i++] = [ $start_ip, $end_ip, $area_id, $isp_id ];
        }

        bless($self, $class);
        return $self;
}

sub get_area_isp_id {
      my $self  = shift;
      my $ip            = shift;
      my $area_id = $DEFAULT_AREA_ID;
```

```
    my $isp_id      = $DEFAULT_ISP_ID;

    # Check if a ip address is in the table using binary search method.
    my $left   = 0;
    my $right= @$self - 1;                          # Get max array index
    my $middle;

    while ($left <= $right) {
        $middle = int( ($left + $right) / 2 );
        if ( ($self->[$middle][$START_IP] <= $ip) && ($ip <= $self->[$middle][$END_IP]) )
        {
            $area_id = $self->[$middle][$AREA_ID];
            $isp_id    = $self->[$middle][$ISP_ID];
            last;
        }
        elsif ($ip < $self->[$middle][$START_IP]) {
            $right = $middle - 1;
        }
        else {
            $left = $middle + 1;
        }
    }

    return ($area_id, $isp_id);
}
```

该对象的使用方法是：

```
use Ipregion;
my $ip_region = Ipregion->new("new_ip_region.dat");
my @search_result = $ip_region->get_area_isp_id(974173694);
```

9．Perl 特殊变量

变量符号（名）	意　义
$a	**sort 函数使用存储第一个将比较的值**
$b	**sort 函数使用存储第二个将比较的值**
$_ ($ARG)	**默认的输入或模式搜索空间**
@_ (@ARG)	**子程序中默认存储传入参数**
ARGV	The special filehandle that iterates over command-line filenames in @ARGV

$ARGV	Contains the name of the current file when reading from ARGV filehandle
@ARGV	**The array containing the command-line arguments intended for script**
$^T ($BASETIME)	The time at which the script began running, in seconds since the epoch
$? ($CHILD_ERROR)	The status returned by the last pipe close, backtick(``)command, or wait, waitpid, or system functions.
DATA	**This special filehandle refers to anything following the __END__ or the __DATA__ token in the current file**
$) ($EGID, $EFFECTIVE_GROUP_ID)	The effective GID of this process
$> ($EUID, $EFFECTIVE_USER_ID)	The effective UID of this process as returned by the geteuid(2) syscall
%ENV	**The hash containing your current environment variables**
$@ ($EVAL_ERROR)	**The currently raised exception or the Perl syntax error message from the last eval operation**
@EXPORT	**Exporter 模块 import 方法使用**
@EXPORT_OK	**Exporter 模块 import 方法使用**
%EXPORT_TAGS	**Exporter 模块 import 方法使用**
%INC	**The hash containing entries for the filename of each Perl file loaded via do FILE, require or use**
@INC	**The array containing the list of directories where Perl module may be found by do FILE, require or use**
$. ($NR, $INPUT_LINE_NUMBER)	**The current record number (usually line numberZ) for the last filehandle you read from.**
$/ ($RS, $INPUT_RECORD_SEPARATOR)	**The input record separator, newline by default, which is consulted by the readline function, the <FH> operator, and the chomp function.** **$/=””将使得记录分割符为空白行，不同于”\n\n”** **undef $/; 文件剩余所有行将全部一次读入** **$/=\$number 将一次读入$number 字节**
@ISA	**This array contains names of other packages to look through when a method call cannot be found in the**

	current package
@+ @- $` $' $& $1 $2 $3	匹配相关变量
$^ $~ $|	Filehandle 相关
$" ($LIST_SEPARATOR)	**When an array or slice is interpolated into a double-quoted string, this variable specifies the string to put between individual elements. Default is space.**
$^O ($OSNAME)	**存储平台名**
$! ($ERRNO, $OS_ERROR)	**数值上下文：最近一次调用的返回值** **字符串上下文：响应系统错误信息**
$, ($OFS, $OUTPUT_FIELD_SEPARATOR)	**print 的字段分割符(默认为空)**
$\($ORS, $OUTPUT_RECORD_SEPARATOR)	**print 的记录分割符(默认为空，设为"\n"是很好的选择)**
$$ ($PID)	The process number
$0 ($PROGRAM_NAME)	程序名
$(($GID, $PEAL_GROUP_ID)	进程的真正 GID
$< ($UID, $PEAL_USER_ID)	
%SIG	The hash used to set signal handlers for various signals
STDERR	标准错误 Filehandle
STDIN	标准输入 Filehandle
STDOUT	标准输出 Filehandle
$; $SUBSEP $SUBSCRIPT_SEPARATOR	The subscript sesparator for multidimensional hash emulation $foo{$a,$b,$c}=$foo{join($;,$a,$b,$c)}

说明：若需要使用长文件名，必须使用 use English;

10 Perl 程序文档(POD)

POD(Plain Old Documentation)，它是一种简单而易用的标记型语言（置标语言），用于 perl 程序和模块中的文档书写。POD 中用段分可以分为三种，普通段落，字面段落（Verbatim Paragraph）和命令段落。三者的区分非常简单，以=pod|head1|cut|over 等指示字开始的段落为命令段落，以空格或制表符（\t）等缩进开始的段落为字面段落，其余的就是普通段落。POD 中有其独特的格式代码来表现粗体，斜体，超链接等。[1]

POD 使得 Perl 语言的文档编写易于完成，程序说明文档与程序源代码同时存在。可以用以

[1] http://fayland.org/journal/POD.html

下解释器解释 POD: pod2text、pod2man (pod2man File.pm |nroff –man |more)、pod2html、pod2latex。

一般建议对源代码包括以下部分的 POD 文档：

```
=head1 NAME

        The name of your program or module.

=head1 SYNOPSIS

        A one-line description of what your program or module does (purportedly).

=head1 DESCRIPTION

        The bulk of your documentation. (Bulk is good in this context.)

=head1 AUTHOR

        Who you are. (Or an alias, if you are ashamed of your program.)

=head1 BUGS

        What you did wrong (and why it wasn't really your fault).

=head1 SEE ALSO

        Where people can find related information (so they can work around your bugs).

=head1 COPYRIGHT

        The copyright statement. If you wish to assert an explicit copyright, you
        should say something like:

              Copyright 2013, Randy Waterhouse.  All Rights Reserved.
        Many modules also add:
        This program is free software.  You may copy or
        redistribute it under the same terms as Perl itself.
```

11. Perl 编程风格

为了使程序易于阅读、理解和维护，建议使用以下编程风格(以下建议均为 Larry Wall 在 perlstyle 文档中所写，其实许多条对于其它语言编程也适用)：

- 多行 BLOCK 的收尾括号应该跟结构开始的关键字对齐；
- 4 列的缩进；

- 开始括号与关键字同一行，如果可能的话，否则，与关键字对齐；
- 在多行 BLOCK 的开始括号之前留空格；
- 一行的 BLOCK 可以写在一行中，包括括号；
- 在分号前不留空格；
- 分号在短的一行 BLOCK 中省略；
- 在大多数操作符两边留空格；
- 在复杂的下标两边加空格（在方括号内）；
- 在做不同事情的代码段中留空行；
- 不连接在一起的 else；
- 在函数名和开始的括号之间不加空格；
- 在每一逗号后加空格；
- 长行在操作符后断开（除了 and 和 or 外）
- 在本行匹配的最后的括号后加空格
- 相应项竖直对齐
- 只要清晰性不受损害省略冗余的标点符号
- 你能怎样做不意味着你应该怎样做。Perl 设计来给予你做任何事的多种方法，所以选择最易读的方法。

  open(FOO, $foo) || die “Cannot open $foo: $!”;

  比 die “Cann’t open $foo: $!” u;nless open(FOO, $foo)
- 不要害怕使用循环标号，它们可以增强可读性并且允许多层循环跳出。
- 选择易于理解的标示符。如果你不能记住你的记忆法，你将碰到问题
- 在使用长标示符时用下划线分隔单词
- 在使用复杂的正则表达式时使用/x
- 使用 here 文档而不是重复的 print()语句
- 垂直对齐相应的东西，特别是在一行不能容下时
- 总是检查系统调用的返回值。好的错误信息应该输出的标准错误输出，包括那一程序造成了问题的信息。
- 对齐转换的信息：

  tr [abc]

  [xyz]
- 考虑复用性，一般化你的代码，考虑写一个模块或者类。
- 使用 POD 文档化你的代码
- 保持一致
- 保持友好

12. 参考文献

【1】Larry Wall, Tom Christiansen & Jon Orwant. Programming Perl. Third Edition (The Camel Book)

【2】Randal L. Schwartz, Tom Phoenix, and brian d foy. Learning Perl (The Llama Book)

【3】Damian Conway. Perl Best Practices(Perl 最佳实践—影印版). O'reilly. 东南大学出版社. 2006.4

【4】Ben Forta 著.杨涛等译. 正则表达式必知必会.北京:人民邮电出版社. 2007.12

【5】Jeffrey E.F. Friedl 著. 余晟译. 精通正则表达式（第 3 版）.北京 :电子工业出版社. 2007.7

【6】The perldoc mainpage

13. 申明