perldoc -u -f 查相关命令帮助

use utf8；

use warnings;开启警告功能。或者在编译时用perl -w \*.pl

use diagnostics;这个编译命令报告更为详尽的问题描述。或者在编译时用perl -Mdiagnostics \*.pl

Perl 尽可能遵循”无内置限制“的原则

不用my操作符修饰的变量，都将是全局变量！！无论是在函数外定义还是在函数内部定义。Perl版本v5.20.2

$0表示程序的名称

标量数据

perl用标量来称呼单个事物。标量是perl里最简单的一种数据类型。

标量可由数字或者字符组成的序列。

可以用操作符对标量进行操作，比如加减或者字符串连接，产生的结果通常也是一个标量。标量可以存储在标量变量里，也可以从文件和设备读取或者写入这些为位置。

数字：

Perl内部总是按“双精度浮点数”的要求来保存数字并进行运算的。也就是说Perl内部并不存在整数值--程序中用到的整型常量会被转换成等效的浮点数值。

浮点数直接量：

直接量是指某个数字在Perl源代码中的写法。直接量并非运算结果，也不是I/O操作的结果。它只是直接键入程序源代码中的数据。

如：1.25 255.000

整数直接量：

0 2001 -40

非十进制的整数直接量：

八进制直接量以o开头。十六进制直接量以0x开头。二进制直接量以0b开头。

非十进制直接量长度超过4个字符时，读起来可能有些困难，Perl允许在这里写直接量中使用下划线。

如：0x1377-0b77 0x50\_60\_72\_7c

数字操作：

数字操作运算符：加 减 乘 除 取模%

字符串：

空字符串：不包含任何字符,是最短的字符串

最长的字符串没有限制。

单引号内的字符串直接量：

如：‘fred’

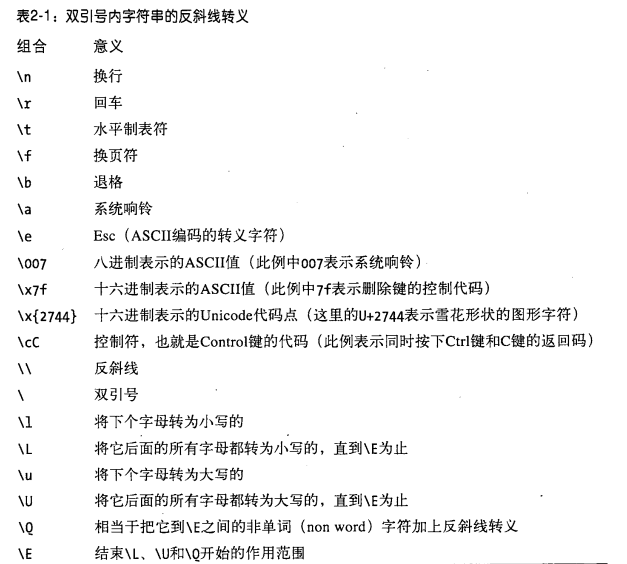
单引号的字符串直接量前后两个单引号并不属于字符串的内容。\也是表示转义字符。如：”Don\`t let ......“

注意：单引号内部的\n（即`\n`）并不是换行符，而是表示字面上的两个字符：反斜杠和字母n。

双引号内的字符直接量：

双引号内的字符直接量同样也是字符串序列，只不过这次换成双引号表示首位。双引号中的反斜杠更为强大，可以转义许多控制字符，或是用八进制或者十六进制写法来表示的任何字符。如：

”barner“ ”Hello World\n“



双引号内字符串的另一种特性为变量内插，这是指在使用字符串是，将字符串内的变量名称替换为改变量当前的值。

字符串操作：

字符串可以用【.】操作符连接起来，两边的字符串都不会因为操作而被修改。

如：“Hello”.“World” “Hello”.‘ ’. “World”(两个‘’之间是空格结果是“Hello World”) ‘Hello World’ . “\n”(结果“Hello World\n”)

需注意：连接运算必须显式使用连接操作符。

字符串重复操作符：x(小写x字母)

此操作符会将其左边的操作数(也就是要重复的字符串)与它本身重复连接，重复次数则由右边的操作数指定。

如：“fred” x 3 结果“fredfredfred”

5 x 4.8 结果：“5555”

重复操作符左操作数必须是字符串类型，所以数字5再进行重复操作前，先被转换为单字符串“5”，然后重复4次。  
重复次数会在使用前先取整(4.8变成4)，重复次数小于1时，会生成长度为零的字符串。

数字与字符串之间的自动转换：

Perl通常根据需要自动在数字和字符串之间进行类型转换。如何转换取决于操作符。如+需要的是数字，Perl会将操作数视为数字。.操作符需要字符串，Perl将操作数视为字符串。

如：“12”\*“3” =36 “12fred34”\*“3” =36 “fred31”\*5 =0

开头非数字的字符串会被转换为0。

自动转换总是按照十进制数字来处理的。如：

0377：十进制数字255的八进制写法。“0377”会被转换成十进制数字377而非255。

“z”.5\*7:则为“z35”。有关优先级和括号。

标量变量：

标量变量，就是单单存储一个值的变量。变量的名称在整个程序中保持不变，但它的值是可以在程序运行时不断修改变化的。

标量变量的名称以美元符号开头。

Prel通过变量标识符前的魔符来区分它是什么类型的变量。

标量赋值：

$fred=17; $barney=‘Hello’;$barney=$fred+3;$barney=$barney\*2;

双目赋值操作符：

如$fred=$fred+5;可以写为$fred+=5;

$str=$str.“ ”;可以写为$str.=“ ”

print输出结果：

print可以接受标量值作为参数，然后不经修饰地将它送到标注输出。

字符串中的标量变量内插：

$barney=“fred ate a $meal”

$barney=‘fred ate a’.$meal;

‘’单引号中的变量取值符$不会起作用，只是普通的$符号，双引号中的$符则起到取变量值的作用。

借助代码点创建字符：

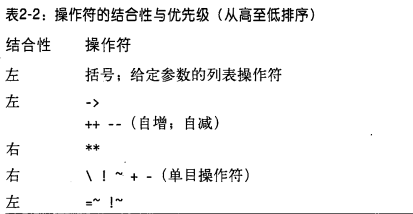
如$alef=chr( 0x5D0);0x5D0为某字符的代码点，chr()函数转换这些代码点为对应的字符。

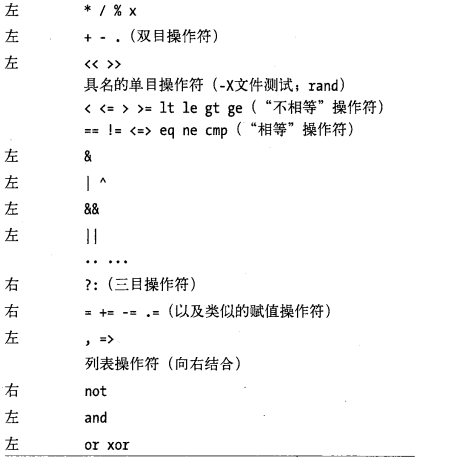
$code\_point=ord(‘?’);通过ord()函数将字符转换为代码点。

如果不先创建变量，也可以直接在双引号内用\x{}的形式表示。如：

“\x{03B1}\x{03C9}”

操作符的优先级与结合性：





所有在Perl和C中同时出现的操作符，他们的优先级和结合性都是相同的。

如：4\*\*3\*\*2为：4\*\*(3\*\*2)即：4\*\*9向右结合。

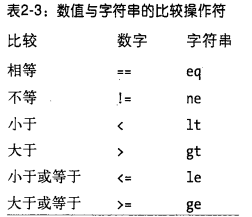
72/12/3为：(72/12)/3 向左结合。

36/6\*3 为：(36/6)\*3

比较操作符：

对数值进行比较时，Perl的比较操作符类似于代数系统：<、<=、==、>=、>、!=。这些操作符的返回值要么是真(true)，要么是假(false)。

比较字符串时，Perl的比较操作符：lt、le、eq、ge、gt、ne。



布尔值：

Perl没有boolean数据类型，它靠一些简单的规则来判断：

如果是数字，0为假；所有其他数字都为真。

如果是字符串，空字符串(‘ ’)为假；所有其他的字符串都为真。

如果既不是数字也不是字符串，拿就先转化成数字或者字符串再进行比较。undef表示假，而且所有的引用都是真。(要查资料)

字符串‘0’跟数字0是同一个标量值，所以Perl会将他们一视同仁，也就是说，字符‘0’是唯一被当成假的非空字符串。

要取得任何布尔值的相反值，可以用！这个单目取反操作符。

获取用户输入：

行输入：<STDIN>。将<STDIN>放在程序中希望返回标量的位置上，Perl就会从标准输入读取一行文本，直到遇到换行符为止。

<STDIN>在标量上下文中会返回输入数据的下一行，在列表上下文中，则会返回所有剩下的行，直到文件结尾为止。返回的每一行都会成为列表中的元素。

可以用chomp()操作符去掉末尾换行符。

如：$text=<STDIN> #输入“a line of text\n”

chomp($text) #去掉行末的换行符。

还可以：

chomp($text=<STDIN>);读入文字，略过最后的换行符。

chomp()本质是函数，它有自己的返回值，chomp()函数返回值是实际移除的字符数。

udef值：

在对某个变量首次赋值前，变量的初始值就是特殊的undef(未定义)值。如果当成数字使用，它就表现的像零，如果当成字符串使用，它就会表现的像空字符串。但undef既不是数字也不是字符串，它完全是另一种类型的标量值。

许多操作符在参数越界或者不合理时会返回undef。

defined()函数：

行输入操作符<STDIN>有时会返回undef。在一般情况下，它会返回一行文本。但若没有更多输入，比如读到文件的结尾时，它就会返回undef来表示这个状况。要判断某个字符串是undef而不是空字符串，可以用defined函数。如果是undef，该函数返回假，否则返回真。

创建undef值，可以直接使用同名的undef操作符。

如：$madonna = undef;

列表与数组：

列表指的是标量的有序集合。

数组表示存储列表的变量。

在Perl中，列表和数组常常混用，但更精确地说：列表是数据，而数组指的是变量。列表的值不一定要放在数组里，但每个数组变量一定包含一个列表。(即便是不含任何元素的空列表)。

因为每个元素都是独立不相关的标量值，所以列表或者数组可能包含数字、字符串、undef值或者不同类型标量值的混合。

访问数组中的元素：

如$fred[0] = “yabba”;

$fred[1] = “dabba”;

$fred[2] = “doo”;

数组(fred)中的名字空间和标量名字空间是完全分开的，可以在同一个程序中再取一个名为$fred的标量变量。

任何求值能得到数字的表达式都可以用作下标。假如他不是整数，则会自动舍去小数，无论正负：如

$number = 2.71828;

print $fred[$number - 1];结果与print $fred[1]相同。

假如下标超出数组的尾端，则对应的值将会是undef，这和一般的标量相同，如果从没有对标量变量进行赋值，他的值就是undef。

特殊的数组索引：

如果对超过数组尾端的元素进行赋值，数组将会根据需要自动扩大。

找出数组里最后一个元素的索引值：$#数组名。如数组名为rocky的最后一个元素的索引：$number = $#rocky。

从数组尾端往回计数：”负数数组索引值“。

如：

$fred[0] = “yabba”;

$fred[1] = “dabba”;

$fred[2] = “doo”;

print $fred[-1];结果为yabba，即fred数组中最后一个元素。

print $fred[-2];为fred数组中中间的一个元素，也就是倒数第二个元素。依次类推。

列表直接量：

列表直接量也就是在程序代码中表示一列数据的写法，可以有圆括号内用逗号隔开的一串数据表示，而这些数据就成为列表元素。列表可以包含任何标量值，例如：

(1,2,3) 包含1,2,3这三个数字的列表

(1,2,3,)与上面相同的三个数字，末尾的逗号会被忽略。

(“fred”,4.5) 两个元素，“fred”和4.5

( ) 空列表，0个元素。

(1..100) 100个整数构成的列表。

.. 范围操作符。

..范围操作符会从左边的数字计数到右边，每次加1，以产生一连串的数字。

qw简写：

在Perl程序里，经常要简历简单的单词列表，这是只需使用qw简写，就可以省略许多无谓的引号的麻烦。Perl都会将其当成单引号内的字符串来处理(所以，在qw构建的列表中，不能想双引号内的字符串一样使用\n或者$fred)。其中的空白符(如空格、制表符以及换行符)会被抛弃，然后剩下的就是列表的元素。因为qw算是一种引用的形式，所以不能将注释放在qw列表中。如：

qw(fred barney betty wilma dino);

Perl允许用任何标点符号作为定界符：如

qw！[/或者#都行]fred barney betty wilma dino ![/或者#都行]

前后两个定界符也可以不同，如() {} [] <>等，左右要对应。

如果你要在被圈印的字符串中使用定界符，可以通过反斜线转义来引入这个字符。如：

qw！ yahoo \! google ask msn ! 将yahoo！作为一个元素包含进来。

和单引号内的字符串一样，两个连续的反斜线表示一个实际的反斜线：

如：qw(This as a \\ real backslash);

列表的赋值：

如：($fred,$barney,$dino) = (“flintstone”,“rubble”,undef)

左侧列表中的三个变量会依次被赋予右侧列表中对应的值。列表是在赋值运算开始之前建立的，所以在Perl里互换两个变量的值可以这样：($fred,$barney) = ($barney,$fred)

列表在等号左边的变量的个数不等于给定的列表值(等号右边的值的个数)时，如果右边多出来，则会被忽略；如果右边少，那么左边多出来的会被设置为undef。

@符表示引用整个数组。如：

@rocky = qw(1 2 3 4 5 6 8)

print @rocky

@tiny = ();空列表

@giant = 1..1e5; 包含100000个元素的列表

@stuff = (@giant,undef,@giant);包含200001个元素的列表。

$dino = “granite”;

@quarry = (@rocks,”crushed rock”,@tiny,$dino);5个元素的列表。因为@tiny是空列表，贡献0个元素，并不会有undef被赋值到列表中。如果需要undef我们可以显示写明。

数组无法成为列表里的元素，因为数组只能包含标量。被赋值前，数组变量的值是空列表，即()。就像标量的初始值是undef一样，新的或者空的数组的初始值是空列表。

将某个数组复制到另一个数组时，仍然算是列表赋值运算，只不过这些列表是存储在数组里。如

@copy = @quarry; 将一个数组中的列表复制到另一个数组。

pop和push操作符：

pop()操作符负责取出数组中最后的一个元素并将其作为返回值返回。

如：@array = 5..9；

$fred = pop(@array); $fred 变成9；@array现在是(5,6,7,8)

$barney = pop @array; $barney变成8，@array现在是(5,6,7)

pop @array #这一行是空上下文中使用pop操作符。所谓“空上下文”只不过是表示返回值无处可去的一种说辞。可以用来删除数组中的最后一个元素。

如果数组是空的，pop什么也不做，直接返回undef。

push()操作符用于添加一个元素(或者一串元素)到数组的尾端。如：

push(@array,0); #现在@array是(5,6,0)

push @array,8; #@array现在是(5,6,0,8)

push @array,1..10;#@array得到了10个新元素。

@others = qw/ 9 0 2 1 0 /；

push @array，@others；#@array又得到了5个新元素。(共19个)

push的第一个 参数或者pop唯一的参数都必须是要操作的数组变量。

shift和unshift操作符：

shift和unshift操作符是对数组开头的进行相应处理。如

@array = qw# dino fred barney #;

$m = shift(@array); #$m变成“dino”，@array现在是(“fred”,”barney”)

$n = shift @array;

shift @array; #现在@array为空

$o = shift @array； #$o变成undef，@array还是空的。

unshift(@array,5);#@array现在仅包含只有一个元素列表(5)

unshift @array,4; #@array现在是(4,5)

@others = 1..3;

unshift @array,@others;#@array又变成了(1,2,3,4,5)

与pop类似，对于一个空的数组变量，shift会返回undef。

splice操作符：

splice可以在数组中间添加或者移除某些元素。它最多可以接受4个参数，最后连个是可选参数。第一个参数是要操作的目标数组，第二个参数是要操作的一组元素的开始位置。如果仅仅给出这两项参数，Perl会把从给定的位置开始一直到末尾的全部元素取出来并返回。第三个参数表示要操作的元素的个数，亦即长度。第四个参数是要替换的列表，这个参数可以将新的列表添加到原来的数组中。

如：

@array = qw(pebbles dino fred barney betty );

@remove = splice @array,2;#在原来的数组中删除掉fred及其后的元素。@remove变成qw(fred barney betty)，而原来的@array则变成qw(pebbles dino)。

@array = qw( pebbles dino fred barney betty );

@remove = splice @array,1,2,qw(wilma);#删除dino和fred

@remove变成qw(dino fred);

@array变成qw(pebbles wilma barney betty)

如果只想添加元素而不需要删除元素，则可以把表示操作元素个数的第三个参数设置为0。

如：@array = qw(pebbles dino fred barney betty);

@removed = splice @array,1,0,qw(wilma);

什么元素也不删除，@remove变为qw();@array变成qw(pebbles wilma dino fred barney betty)

注意：wilma出现在dino之前的位置。Perl从索引位置为1的地方插入新列表，然后顺移原来的元素。

字符串中的数组内插：

数组的内容可以被插入到双引号串中。内插时，会在数组的各个元素之间自动添加分隔用的空格。：

@rocks = qw(flintstone slate rebble);

print “quzrtz @rocks limestone\n”;

数组被内插后，它的首尾不会添加额外空格。print “quzrtz(@rocks)limestone\n”; 则为quzrtz(flintstone slate rebble)limestone。

如果要在某个标量变量后面接着写左方括号，需要现将这个括号隔开，他才不会被识别为数组引用的一部分，如：

@fred = qw(eating rocks is wrong);

$fred = “right”;

print “this is $fred[3]\n”;用到了$fred[3],打印“wrong”

print “this is ${fred}[3]\n”;打印“right”(用花括号避免歧义)

print “this is $fred”.”[3]\n”;

print “this is $fred\[3]\n”;打印right，用反斜线转义避免歧义。

for循环：

控制变量需要添加$

实例：

#!/bin/perl

chomp(@lines = <STDIN>);

for ($i = 0;$i < 5;$i++)

{

print “$lines[$i]”;

}

foreach控制结构：

foreach(循环)能逐项遍历列表中的值，依次迭代。

控制变量并不是列表元素的复制，实际上，他就是列表元素的本身。如果在循环中修改了控制变量的值，也就是同时修改了这个列表元素。如：

#!/usr/bin/perl

@rocks = qw (bedrock slate lava);

foreach $rock (@rocks){

$rock = “\t$rock”;

$rock .= “\n”;

}

print “The rocks are:\n”,@rocks;

在循环结束后控制变量仍然是循环执行前的值，如此例中循环控制变量rock在执行循环之前是undef的，则执行循环之后仍然是undef，循环中是列表rocks中的某一个元素。

Perl最喜欢用的默认变量$\_:

如果在foreach循环开头省略控制变量，Perl就会用它最喜欢的默认变量$\_。如：

foreach (1..10){

pint “I can count to $\_!\n”;

}

reverse操作符：

reverse操作符会读取列表的值(也可能来自数组)，并按相反的次序返回该列表。如：

@fred = 6..10;

@barney = reverse(@fred);得 10， 9 ，8 ，7 ，6

@wilma = reverse 6..10;同上

@fred = reverse @fred;将逆序后的结果放回原来的那个数组。

Perl总是会先计算(等号右边)要赋的值，再实际进行赋值操作。

reverse会返回次序相反的列表，但它并不会修改传进来的参数。

sort操作符：

sort操作符会读取列表的值(也可能来自数组)，而且会根据内部的字符编码顺序对它们进行排序。sort操作符的参数应该是某个输入列表，返回值为排序后的列表。

@rocks = qw (bedroc slate rubble granite);

@sorted = sort(@rocks);得bedroc,granite,rubble,slate

@back=reverse sort @rocks;得slate,rubble,granite,bedroc

each操作符：

Perl从5.12版开始，each可用于数组。此前版本只能用于提取哈希的键-值对。

每次对数组调用each，会返回数组中下一个元素所对应的两个值：该元素的索引以及该元素的值。如：

my @rocks = qw(bedrock slate rubble granite);

while(my($index,$value)=each @rocks){

print “$index:$value\n”

}

标量上下文与列表上下文：

同一个表达式出现在不同的地方会有不同的意义。

所谓上下文，指的是你如何使用表达式。

如：以数组的名称为例，在列表上下文中，它会返回元素的列表；在标量上下文中，则返回数组中的元素个数。

@people = qw(fred barney betty);

@sorted = sort @people;列表上下文：barney，betty，fred

$number = 42 + @people;标量上下文：42 + 3

@list = @people；得到barney，betty，fred

$n = @people;得到列表元素的个数3

但并不是完全这样。实际上能够概括的规则也就是：哪种上下文更有意义，就应该是哪种上下文。

在标量上下文中使用产生列表的表达式：

sort在标量上下文中使用返回undef。

reverse在标量上下文中使用，则返回逆序后的字符串(先将列表中的所有字符串全部链接到一起，再对结果中的每一个字符作逆序处理)。如：

@backwards = reverse qw(yabba dabba doo);

得到doo,dabba,yabba

$backwards = reverse qw(yabba dabba doo);

得到oodbbbadabbay

($dino) = something; 列表上下文。($dino)是列表。

在列表上下文中使用产生标量的表达式：

如果表达式求值结果为标量值，则自动产生一个仅含此标量的列表，如：

@fred = 6\*7;得到仅有单个元素的列表(42)

@wilma = undef;得到一个列表，而且仅有的一个元素为未定义的(undef)，并不会将数组清空。

@betty = ()；这样操作才是清空数组的正确方法。

因为undef是标量值，所以将undef赋值给数组并不会清空数组。若要清空数组，直接赋予一个空列表就可以了。

强制指定标量上下文：

如果要在列表上下文的地方强制引入标量上下文，可以使用伪函数scalar。scalar不是真正的函数，只是告诉Perl在这里要切换到标量上下文。如：

@rocks = qw(talc quatrz jade obsidian);

print “How many rocks do you hava?\n”;

print “I have ”,@rocks,”rocks!\n”;错误，这会打印各种石头的名。

print “I have ”,scalar @rocks,”rocks!\n”;打印石头的种类数。

没有相应的函数可以用来强行切入列表上下文。

列表上下文中的<STDIN>:

chomp()参数可以是数组，依次把数组中的每一行中的换行符去掉。

@lines = <STDIN>;

chomp(@lines);这样更好：chomp(@lines<STDIN>);

子程序：

子程序名属于独立的名字空间，这样Perl就不会将同一段代码中的子程序&fred和标量$fred搞混。

定义子程序：

定义子程序可以用关键字sub、子程序名(不包含与号)以及用花括号封闭起来的代码块。如：

sub marine {  
 $n +=1;#全局变量 $n

print “Hello,sailor number $n!\n”;

}

子程序不需要进行事先声明，子程序的定义是全局的。如果定义了两个重名的子程序，后面一个将会覆盖前面一个。

调用子程序：

&marine；

返回值：

Perl任何子程序都有返回值。可以不写return，在子程序的执行过程中，最后一次运算的结果(不管是什么)都会被自动当成子程序的返回值。

sub marine {  
 $n +=1;#全局变量 $n

print “Hello,sailor number $n!\n”;

}

这个执行成功后将返回1，表示执行成功。

参数：

要传递参数列表到子程序里，只要在子程序调用的后面加上被括号圈引的列表表达式即可。

sub max {  
 if ($\_[0] > $\_[1]){

$\_[0];

}else{

$\_[1];

}

}

如$n = &max(10,15);

Perl在子程序中会先将这个列表存在某处，自动将参数列表化名为特殊的数组变量@\_，该变量在子程序执行期间有效。

表示子程序第一个参数存储于$\_[0]，第二个参数存储于$\_[1]，依次类推。但这些变量和$\_毫无关联。

max子程序并没有说明只接受两个参数，所以

$n = &max(10,11,12);

也没错，多余的参数将会被忽略。参数如果不足也会被忽略--如果用到超出@\_数组边界的参数，只会得到undef。

@\_是子程序的私有变量。如果已经有了全局变量@\_，则该变量在子程序调用前会被存起来，并在子程序返回时恢复原来的值。

子程序中的私有变量：

my操作符用来创建私有变量。私有变量称之为词法变量。

sub max {

my($m,$n) = @\_;  
 if ($m > $n){

$m;

}else{

$n;

}

}

变长参数列表：

检查@\_数组长度：if (@\_ != 2) #(此处用的是标量上下文)

改进&max子程序，使其能接受任意数目的参数：

$maximum = &max(3,5,10,4,6);

sub max{  
 my($max\_so\_far) = shift @\_;

foreach (@\_){

if($\_ > max\_so\_far){

$max\_so\_far = $\_;

}

}

$max\_so\_far;

}

my($num) = @\_ 是列表上下文。

my $num = @\_ 是标量上下文。

在my操作符不加括号时，只能用来声明单个词法变量。my也可以用来创建私有数组：my @phones

use strict编译命令：

return操作符：

省略函数调用的&号：

1. 将参数列表放进括号，他就是函数调用，如：

#!/usr/bin/perl

my $num = max(5,6);

print $num."\n";

sub max{

if ($\_[0] > $\_[1]){

¦ return $\_[0];

}

return $\_[1];

}

1. 将函数定义写在函数调用之前，这种情况参数列表可以不写()。如：

#!/usr/bin/perl

sub max {

if ($\_[0] > $\_[1]){

¦ return $\_[0];

}

return $\_[1];

}

$num = max 5,6;

print $num."\n";

如果自定义函数与Perl内置函数名相同，则调用时必须加上&符号。

非标量返回值：

持久性私有变量：

在创建私有变量时，用state操作符来声明变量，在子程序的多次调用期间保留之前的值，并将变量作用域局限在子程序内部。

使用state时需要在程序前添加 use feature ‘state’

#!/usr/bin/perl

use feature 'state';

sub marine {

state $n = 0; //这一句只会在第一次调用marine函数时执行一次

$n += 1;

print "Hello,sailor number $n!\n";

}

&marine;//结果为1

print $n; //$n是空的。

&marine;//结果为2

state用在数组和哈希类型的state变量时，有限制！！！如：

state @array = qw(a b c)；这是错误的。

输入与输出：

读取标准输入：

如：

while(defined($line = <STDIN>)){...}

while(<STDIN>){print “I saw $\_”;} 这两例while循环中条件都是标量。输入一个打印一个。此处如果单一行写一个<STDIN>，则是不会自动写进变量$\_的。唯独while循环的条件表达式里只有行输入操作符的前提下，这个简写才起作用。

foreach (<STDIN>){print “I saw $\_”};#<STDIN>处应该是数组，所以直到你输入结束，他才开始打印。

来自钻石操作符的输入：

程序调用参数：

钻石操作符：<>。他能让程序在处理调用参数时，提供类似于标准Unix工具程序的功能。如下：

perl my\_program fred barney betty #fred barney betty等都是程序调用参数。

钻石操作符是行输入操作符的特例。如：

while (defined($line = <>)){

chomp($line);

print “It was $line that I saw!\n”

}

也可以写成下面的：

while(<>){

chomp;

print “It was $line that I saw!\n”

}

连字符 -

如果在执行程序时没有程序调用参数，则钻石操作符<>将从标准输入读取。如果在多个程序调用参数之间使用连字符-，则会先读取连字符之前的程序调用参数，然后从标准输入中读取，然后读取连字符之后的程序调用参数。

注意：在一个程序中，若是出现多个钻石操作符，则第一个就会将所有的程序调用参数处理完，第二个以及后面所有的都不会再处理程序调用参数，只能等待从标准输入读取。如果在以后的钻石操作符执行之前更改@ARGV，则紧接着的那个钻石操作符还将处理@ARGV中的文件。

调用参数：

钻石操作符不会检查命令行参数，他的参数来自@ARGV数组，这个数组是由Perl解释器事先简历的特殊数组，他的内容就是命令行参数组成的列表。可以像普通的数组一样使用@ARGV。可以在未使用钻石操作符之前<>，可以更改@ARGV数组，指定操作什么文件。

输出到标准输出：

print @array;#把数组的元素打印出来，中间没有空格。

print @array.”\n”;#这个将得到@array中元素个数3，而不是@array中的每一个元素。因为.操作符是用来操作标量的。

print “@array”;#把数组的元素打印出来，每个元素之间添加一个空格(数组元素内插的结果)

如果@array每个元素都是以换行符结尾的，则print “@array”将会分成三行输出。

用printf格式化输出：

%d：按照十进制正式，无条件舍去小数点后的数字。

%s：代表字符串格式，它的功能就是字符串内插，还能设定字符的宽度。

%g：将按照需要自动选择浮点数、整数、甚至是指数形式。

%-5d %-5s %-5g 等都是左对齐，%5d %5s %5g 等都是右对齐

%f转换浮点数格式会按照四舍五入，还可以指定小数点之后的输出位数：

如printf “%12.3f\n”,6\*7+2/3;#``````42.667 。`代表空格，12.3中0.3代表小数后面3位，而12代表右对齐12个字符。

要输出%，则应该用%%。

数组和printf：

数组可以包含任意数目的元素，格式化字符串只会用到固定数目的元素:格式化字符串里有多少个转换格式，那就必须有多少个元素可供使用。

可以在程序运行时动态产生格式化字符串，如：

my @items = qw (wilma dino pebbles);

my $format = “The items are:\n” . (“%10s\n” x @items);

printf $format,@items;

文件句柄：

文件句柄就是在程序里代表Perl进程与外界之间的I/O联系的名称。它是“这种联系”的名称，不是文件名称。

尽量用全大写字母来命名文件句柄。

6个特殊的文件句柄名是Perl保留的：STDIN、STDOUT、STDERR、DATA、ARGV、ARGVOUT。

打开文件句柄：

open操作符：如：

open CONFIG, ‘dino’； ：打开一个名为CONFIG的文件句柄，让他指向文件dino，也就是说它会打开文件dino，文件中的任何内容都能从文件句柄CONFIG被读到我们的程序中来。

open CONFIG,‘<dino’；和上面所做的事情一样，只是用小于号来声明“此文件只是用来读取的，而非写入”。打开文件句柄的默认模式就是读取数据。

open BEDROCK ,’>fred’;打开一个文件句柄BEDROCK，创建一个新文件并输出到新文件fred。如果已经存在fred文件，则会清空原来文件内容并用新的内容取代。

open LOG,’>>logfile’;追加。不存在就创建，存在就追加。

Perl 版本5.6以后可以这样写：

open CONFIG,’>>’,’dino’;

指定数据打开的编码方式：

open CONFIG,’<:encoding(UTF-8)’,’dino’;

open CONFIG,’<:utf8’,’dino’;这两种写法的区别是：第一种会检测编码是否正确，第二种则不管拿来的是什么，直接当成UTF-8编码使用，可能会出现问题。

open BEDROCK, ‘>:encoding(UTF-8)’,$file\_name;

使用encoding()的形式，还可以指定其他类型的编码，可以通过：

% perl -MEncode -le “print for Encode->encodings(‘:all’)”

打印出所有Perl能理解和处理的字符编码清单。

open CONFIG,’+<’,’file’;以读写方式打开文件。

实例：

#!/bin/perl

open CONFIG,”test.pl”;

while(<CONFIG>)

{

print “$\_”;

}

以二进制方式读写文件句柄：

binmode 可以关闭文件句柄中的换行符相关处理。如：

binmode STDOUT;

binmode STDOUT,’:encoding(UTF-8)’;

有问题的文件句柄：

Perl自身无法打开系统中的文件，只能要求操作系统打开。如果权限不足、文件名错误等，操作系统会拒绝打开文件，那么文件句柄就会有问题。从有问题的文件句柄读取数据会立刻读到文件尾。(“文件结尾在”在标量上下文中是undef，在列表上下文则是空列表)。

关闭文件句柄：

close操作符用来关闭文件句柄。

close BEDROCK;

当重新打开某个文件句柄时(在新的open命令中重用之前的文件句柄名)，Perl会自动关闭原来的文件句柄，在程序结束时，Perl也会自动关闭文件句柄。

关闭文件句柄会刷新输出并释放文件上的锁。

用die处理致命错误：  
当Perl遇到致命错误时(如：除以零，使用不合法的正则表达式，或调用未定义的子程序等等)，程序应该立刻终止，并发出错误信息告知原因。可以用die函数来实现。

die函数会输出你指定的信息到专为这类信息准备的标准错误流中，并且让你的程序立刻终止并返回不为零的退出码。

如：

#!/usr/bin/perl

if (!open LOG,'>>','/log'){

die "Cannot create logfile:$!\n";

}

如果以普通用户去执行这个程序，open失败，die会终止程序的运行，并打印错误信息。$!表示可读的系统错误信息，比如这个就是”权限不够“。如果die函数显示程序中不属于系统服务请求的错误，这时不要用$!。$!只有在系统服务请求失败后的瞬间，他的值才有用，如果操作成功，$!里不会有任何有用信息。

用warn送出警告信息：

warn不会终止程序的运行。

自动检测致命错误：

use autodie;

使用文件句柄：

#!/usr/bin/perl

if (!open PASSWD,'<','/etc/passwd'){

die "How did you get logged in? ($!)";

}

while (<PASSWD>){

print $\_."\n";

}

以写入或者添加模式打开的文件句柄可以在print和printf函数中使用，使用时，直接将文件句柄放在函数名之后，参数列表之前。如：

print LOG “Caption’s log,stardate 3.14159\n”;

printf STDERR “%d percent complete.\n”,$dne/$total \* 100;

改变默认的文件输出句柄：

可以使用select操作符来改变默认的文件句柄。如：

select BEDROCK;

print “I hope Mr. Slate doesn’t find out about this.\n”;

pirnt “Wilma!\n”;

一旦使用了select默认文件句柄，程序就会一直往那里输出。将数据输出到文件句柄时，默认情况下都会经过缓冲处理，只要将特殊变量$|设定为1，就会使当前的默认文件句柄在每次进行输出操作后立刻刷新缓冲区。这样可以立刻刷新缓冲区，也可以防止上一个文件句柄的内容跑到下一个文件句柄中。

select LOG；

$| = 1;

select STDOUT;

print LOG “This gets written to the LOG at once!\n”;

用say来输出：

say的功能和print函数差不多。但在打印每行内容时会自动添加换行符。

标量变量中的文件句柄：

将文件句柄放到标量中，文件句柄就可以作为子程序的参数传递，或者放在数组、哈希中排序，或者严格控制它的作用域。

**裸字**——没有引号的单词——来作为字符串

my $rocks\_fh;

open $rocks\_fh,’<’,’rocks.txt’

or die “Could not open rocks.txt：$!”;

**哈希：**

哈希是一种数据结构，它和数组的相似之处在于可以容纳任意多的值并按需取用，而它和数组的不同在于索引方式，数组是以数字来索引，哈希则是以名字来索引。哈希的索引值称为键，并不是数字，而是任意唯一的字符串。

可以用任意字符传表达式作为哈希键。哈希里没有顺序，也没有所谓的第一个元素。

哈希可能是任意大小的，从没有任何键-值对的空哈希到填满内存的巨大哈希。

虽然键是唯一的，但他们对应的值可以是重复的。哈希的值可以是数字、字符串、undef、或是这些类型的组合。但哈希的键必须全部是唯一的字符串。

哈希是从键到值的单行道，无法在哈希中查询值并反推出其相应的键。

访问哈希元素：

语法：$hash{some\_key}#使用花括号来表示索引值(哈希键。)

如：

$family\_name{‘fred’} = ‘flintstone’;

$family\_name{‘barney’} = ‘rubble’;

可以这样写：

foreach my $person (qw <barney fred>){

print “I’ve heard of $person $family\_name{$person}.\n”;

}

哈希元素会因赋值而诞生：

$family\_name{‘wilma’} = ‘flintstone’;#增加一个新的键-值对

$family\_name{‘betty’} .= $family\_name{‘barney’};#在需要的时候动态创建该元素。

访问整个哈希：

要指代整个哈希，可以用百分号(%)作为前缀。

哈希可以转换为列表。列表也可转换为哈希。对哈希赋值等同于在列表上下文文中赋值，列表中的元素应该为键-值对。如：

%some\_hash = (‘foo’,35,’bar’,12.4,2.5,’hello’,’wilma’,172e30,’betty’,”bye\n”);

在列表上下文中，哈希的值就是简单的键-值对列表：

@any\_array = %some\_hash;

这个变换叫做展开哈希。将它变成键-值对列表。

哈希赋值：

哈希可以用一般的赋值语法来复制:

my %new\_hash = %old\_hash;

反序哈希：键-值互换。

my %inverse\_hash = reverse %any\_hash;

如原来哈希(key,value,key,value,key,value)，上述操作后%inverse\_hash变成(value,key,value,key,value,key)。这样只能用在哈希值不重复的情况。

胖箭头：

只是用于区分在哈希赋值时将“，”替换为“=>”。如：

my %last\_name = (

‘fred’ => ‘flintstone’,

‘dino’ => undef;

‘barney’ => ‘rubble’;

‘betty’ => ‘rubble’;

)

哈希函数：

keys和values函数：

keys函数可以返回哈希的键列表，而values函数可以返回对应的值列表。如果哈希没有任何成员，则两个函数都返回空列表。如：

my %hash = (‘a’ => 1,’b’ => 2, ‘c’=> 3);

my @k = keys %hash;

my %v = values %hash;

my $count = key $hash; #结果为3，标量上下文。

each函数：

each函数用来迭代整个哈希，它以包含两个元素的列表形式返回键-值对。每次对同一个哈希调用此函数，它就会返回下一组键-值对，直到所有的元素都被访问过。在没有任何新的键-值对，each返回空列表。

在while循环中使用，如：

while(($key,$value) = each %hash){

print “$key => $value\n”;

}

exists函数：

exists函数用来检查哈希中是否存在某个键，他能返回真或假，分别表示键存在与否，和键对应的值无关。如：

if (exists $books{“dino”}){

.......

}

delete 函数：

delete函数能从哈希中删除指定的键及其对应的值。如果没有这样的键，则会直接结束。

将undef存入哈希元素，exists会的到1，即键存在。但delete之后，键将不存在。

哈希元素的内插：

即将单一哈希元素内插到双引号引起的字符串中。如：

foreach $person (sort keys %books){

if ($books{$person}){

print “$persion has $books{$persion} items\n”;

}

}

这种方式不支持内插整个哈希。

%ENV哈希：

正则表达式：

正则表达式在Perl中也叫模式

grep ‘flint.\*stone’ chapter\*.txt

使用简单模式：

若匹配的对象是$\_的内容，只要把模式写在一对斜线(/)中就可以。而模式本身就是一串简单的字符序列：

$\_ = “yabba dabba doo”;

if(/abba/){

print “It matched!\n”;

}

表达式/abba/默认会在$\_中寻找

所有在双引号引起的字符串中能使用的技巧(尤其是反斜线转义)都可以在模式串里使用。因此，/cake\tsprite/这个模式会匹配cake、一个制表符和sprite这11个字符。

Unicode属性：

如：

$\_ = “Hello 14161 World”;

匹配空格：

if (/\p{Space}/){

print “The string has some whitespace.\n”;

}

匹配数字：

if (/\p{Digit}/){

print “The string has a digit.\n”;

}

匹配十六进制数字的字符集合[0-9A-F a-f]:

if (/\p{Hex}\p{Hex}/){

print “The string has a pair of hex digits.\n”;

}

匹配不包含特定属性的字符，只需将小写p改写成大写。匹配指定属性以外的字符；

if(/\P{Space}/){

print “The string has one or more non-whitespace characters.\n”;

}

关于元字符：

点号(.)能匹配任意一个字符的通配符，换行符除外。只能用来匹配一个字符。

如果用来匹配点号(.)本身，则需添加一个\进行转义。

反斜线(\)：匹配反斜线，需要用两个反斜线。一个是转义符。

简单的量词：

星号(\*)用来匹配前面的条目零次或者多次。如：

/fred\t\*barney/能够匹配fred和barney之间任意数量的制表符。但不能有其他的字符出现。

.\*：可以匹配任意字符零次到无限次。如：

/fred.\*barney/任意一行只要提到fred并且在其后提到barney，就可以匹配此模式。

加号(+)：匹配前一个条目一次以上。如：

/fred+barney/ 会匹配在fred与barney之间用空格隔开而且只用空格隔开的字符串。

问号(?)：表示前面一个条目出现一次或者不出现。如：

/bamm-?bamm/ 能且只能匹配两种情况：bamm-bamm或bammbamm

这三个量词指定了前一个条目出现的次数，所以它们都必须接在某个东西之后。

模式分组：

在正则表达式中，圆括号()的作用是对字符串分组。所以圆括号也是元字符。如：

模式/(fred)+/会匹配像fredfredfred这样的字符串。模式/(fred)\*/因为星号代表要匹配重复出现零次以及零次以上的fred，所以任何字符串都匹配这个模式。

反向引用：写法在反斜线后面接上数字编号，如\1、\2。相应的数字表示对应顺序的捕获组。

捕获组：用反向引用来引用圆括号中的模式所匹配的文字。

使用圆括号包围的点号可以匹配任意非换行字符。我们可以用反向引用\1来再次匹配刚刚在圆括号中匹配到的任意字符。如：

$\_ = “abba”;

if(/(.)\1/){#匹配 ‘bb’

print “It matched same character next to itself!\n”;

}

(.)\1表明需要匹配连续出现的两个同样字符。(.)首先会匹配a，但是在查看反向引用的时候就会发现下一个字符不是a，导致匹配失败，Perl会跳过这个字符，用(.)来匹配下一个字符b，在查看反向引用的时候会发现下一个字符也是b，于是成功匹配。

$\_ = “yabba dabba doo”;

if(/y(....) d\1/){#匹配y后面的4个连续的非换行符，并用\1反向引用表示匹配d后面也出现这4个字符(要和y后面的4个相同)的情况。

print “It matched the same after y and d!\n”;

}

$\_ = “yabba dabba doo”;

if(/y(.)(.)\2\1/){#匹配‘abba’

print “It matched after the y!\n”;

}

如何区分哪个括号是第几组：依次点算左括号(包括嵌套的括号)的序号就可以了。如下：\3是b,\2是a。

$\_ = “yabba dabba doo”;

if(/y((.)(.)\3\2) d\1/){

print “It matched!\n”

}

新的反向引用写法：\g{N},可以用负数。如:

$\_ = “aa11bb”;

if(/(.)\111/){

print “It matched!\n”;

}

\111是\1还是\11还是\111呢？？Perl尽可能创建最多数量的反向引用，所以Perl认为这个是\111。用新出的写法\g{1}，就能消除反向引用与模式的直接量部分的二义性。如：

use 5.0.10；

$\_ = “aa11bb”;

if(/(.)\g{1}11/){

print “It matched!\n”;

}

$\_ = “aa11bb”;

if(/(.)\g{-1}11/){

print “It matched!\n”

}

$\_ = “xaa11bb”;

if(/(.)(.)\g{-1}11/){

print “It matched!\n”;

}

\g{N}，我的理解：当N为正时，从左边括号算起。当N为负数时从右边括号算起。

择一匹配：

竖线(|)：或。如果左边的模式匹配失败了，可以用右边的再试试。如：

/fred|barney|betty/能匹配到任何包含有fred或者barne或betty的字符串。

/fred(\t)+barney/加号表示重复一次或更多。

/fred(and|or)barney/用来匹配任何含有fredandbarney或者fredorbarney的字符串。

字符集：

字符集是指一组可能出现的字符，通过写在方括号[]内表示。它只能匹配单个字符，但可以是字符集列出的任何一个。如：

字符集[a b c w x y z]，它可以匹配这7个字符中的任意一个。

字符集[a-c w-z] - 连字符。如：

$\_ = “The HAL-9000 requires authorization to continue.”;

if (/HAL-[0-9]+/){#+加号匹配9一次或多次

print “The string mentions some model of HAL computer.\n”;

}

脱字符：^，表示某些字符除外。如：[^def]会匹配def之外的任何字符。

[^n\-z]会匹配n、连字符、z以外的任何字符。

字符集的简写：

\d：表示任意一个数字的字符集简写。[0-9]

/a：写在正则表达式末尾，表示按照ASCII的语义展开。如：

$\_ = “The HAL-9000 requires authorization to continue.”;

if (/HAL-[\d]+/a){

say “The string mentions some model of HAL computer.”;

}

\s：简写，能匹配任意空白符。大抵等同于Unicode属性\p{Space}。

空白符：换页符、水平制表符(tab)、换行符、回车符、空格符。如：

if(/\s/a){  
 say “The string matched ASCII whitespace.”;

}

范围小的空白字符集：\h 只匹配水平空白符。\v 只匹配垂直空白符。

\h和\v 并起来，就成了\p{Space}。

反义简写：

\D \W \S

用m//进行匹配：

m//：模式匹配操作符。可以改为m(fred) m<fred> m{fred}等，和qw差不多。如果用//则可以省略m。

模式匹配修饰符：

模式匹配修饰符：也叫标志，它们是一些追加在模式表达式末尾定界符后面的字母，用来改变默认的匹配行为。如之前的/a。

用/i进行大小写无关的匹配：

要实现大小写无关的模式匹配，比如同的匹配FRED、fred和Fred，可以用/i匹配。如：

print　“Would you like to play a game?”;

chomp($\_ = <STDIN>);

if (/yes/i){

print “In that case,I recommend that you go bowling.\n”;

}

用/s匹配任意字符：

默认情况，点号(.)无法匹配换行符。

/s修饰符会将模式中的每一个点号换成按字符集[\d\D]来处理，也就是会匹配任意字符，包括换行符。如：

$\_ = “I saw Barney\ndown at the bowling alley\nwith Fred\nlast night.\n”;

if(/Barney.\*Fred/s){

print “That string mentions Fred after Barney!\n”;

}

/m修饰符表示对多行内容进行匹配

用/x加入空白符：

/x修饰符：在模式中随意添加空白符。如：

/-?[0-9]+\.?[0-9]\*/

/ -? [0-9]+ \.? [0-9]\* /x

加上/x后，原来表示空格和制表符本身的空白符失去意义。在空格前添加反斜线\或者使用\t，常用的是\s或者\s\*或者\s+，表示匹配空白符。

组合修饰符：

如：

if(/barnery.\*fred/is){#同时使用/i和/s

print “That string mentions Fred after Barney!\n”;

}

选择一种字符解释方式：

三种字符解释方式：ASCII、Unicode和locale。

/a：ASCII

/u：Unicode

/l：表示遵从本地化语言的设定，按照相应的字符集编码做相应的处理。

锚位：

通过锚位，可以让模式仅在字符串指定位置匹配。

\A：锚位匹配字符串的绝对开头，如果开头位置上不匹配，是不会顺移到下一个位置继续尝试匹配的。如：

m{\Ahttps?://}i

如果要匹配字符串的绝对末尾，可以用\z锚位。如：

m{\.png\z}i

\z后面再无任何其他东西。

\Z行末锚位，它允许后面出现换行符。如：

while (<STDIN>) {

print if(/\.png\Z/);

}

同时使用行首锚位和行末锚位，以确保模式能匹配给定字符串的全部，如：

/\A\s\*\Z/ 匹配一个空行。

行首锚位：^

行末锚位：$

如：$\_ = ‘This is a wilma line

barney is on another line

but this ends in fred

and a final dino line’;

匹配末行出现fred的字符串：/fred$/m。

匹配以barney开头的字符串：/^barney/m。

单词锚位：

\b：单词边界锚位，它匹配任何单词的首尾。如：

/\bfred\b/可以匹配fred，但无法匹配frederick、alfred

绑定操作符：

默认情况下模式匹配操作对象是$\_,绑定操作符(=~)，告诉Perl，用右边的模式来匹配左边的字符串，而不是匹配$\_。如：

my $some\_other = “I dream of betty rubble.”;

if($some\_other =～ /\brub/){

print “Aye,there’s the rub.\n”

}

模式中的内插：

正则表达式内部可以进行双引号形式的内插。如：

my $what = “larry”;

while (<>){

if(/\A($what)/){

print “We saw $what in beginning of $\_”;

}

}

捕获变量：

模式匹配中捕获到的变量存储在$1，$2 .....

实例：

$\_ = “Hello there, neighbor”;

if (/\s(\[a-zA-Z]+),/){ #捕获空白符和逗号之间的单词。

print “the word was $1\n”;#打印the word was there

}

也可以一次捕获多个字符：

$\_ = “Hello there, neighbor”;

if (/(\S+) (\S+), (\S+)/) {

print “words were $1 $2 $3\n”;

}

结果是words were Hello there neigbor。输出结果没有逗号，是因为模式里的逗号放在圆括号外面，所以第二个捕获中不会有逗号。

这些变量通常能存活到下次成功匹配为止。失败的匹配不会改动上次成功匹配时捕获的内容，而成功的匹配会将他们的值重置。可以将捕获到变量放到普通变量当中。

不捕获模式：

不捕获圆括号需要在左括号后面加上问号和冒号(?:)，告诉Perl这一对括号完全是为了分组而存在的。

如：

if(/(?:bronto)?saurus (steak|burger)/){ #不会匹配bronto，并且$1是steak或者burger。

print “Fred wants a $1\n”;

}

择一匹配：

我们可以利用圆括号的捕获功能提取特定字符串并保存到诸如$1、$2这样的变量中。不过就算是较为简单的模式，要维护数字变量和圆括号之间的对应关系也是比较困难的事情。如下例子：

use 5.010;

my $names = ‘Fred or Barney’;

if ( $names =~ m/(\w+) and (\w+)/ ) { #不会匹配成功

say “I saw $1 and $2”;

}

实际看不到say的输出，因为模式字符串中期望的是and，而实际变量中给出的是or，所以正则表达式中加入择一匹配，不管是and还是or都没关系，作为择一匹配的部分必须加上圆括号表示候选列表范围：

use 5.010;

my $names = ‘Fred or Barney’;

if ( $names =~ m/(\w+) (and|or) (\w+)/ ) { #不会匹配成功

say “I saw $1 and $2”;

}

命名捕获：

为了避免记忆$1之类的数字变量，Perl 5.10增加了对捕获内容直接命名的写法，最终捕获到的内容会保存在特殊哈希%+里面：其中的键就是捕获时用的特殊标签，对应的值则是被捕获的字符串。具体写法是(?<LABEL>PATTERN)，其中LABEL可以自行命名。

实例：下面把第一个捕获标签定为name1，第二个标签定为name2，所以，提取捕获内容时需要访问的变量变成了$+{name2}和$+{name2}:

use 5.010;

my $names = “Fred or Barney”;

if ( $names =~ m/(?<name1>\w+) (and|or) (?<name2>\w+)/ ){

say “I saw $+{name1} and $+{name2}”;

}

在使用捕获标签后，反向引用的用法也随之有所变化。之前我们用\1或者\g{1}这样的写法，现在我们可以使用\g{label}这样的写法：

use 5.010;

my $names = ‘Fred Flintstone and Wilma Flintstone’;

if ( $names =~ m/(?<last\_name>\w+) and \w \g{last\_name}/ ) {

say “I saw $+{last\_name}”;

}

我们也可以使用另一种语法表示反向引用。\k<label>等效于\g{label}.

use 5.010;

my $names = ‘Fred Flintstone and Wilma Flintstone’;

if ( $names =~ m/(?<last\_name>\w+) and \w \k{last\_name}/ ) {

say “I saw $+{last\_name}”;

}

自动捕获变量：

有三个变量就算不加圆括号也可以使用，他们是$&、$`、$’。当字符串里实际匹配模式的部分就会被自动存进$&里：

if (“Hello there, neighbor” =~ /\s(\w+),/){

print “That actually matched ‘$&’.\n”;

}

当上面的程序运行时，会告诉我们字符串里匹配的部分是“there,”（一个空格、一个单词以及一个逗号）。第一个捕获内容存在$1中，是具有5个字母的单词there，但是$&里保存的是整个匹配区段。

匹配区段之前的内容会保存到$`里，二匹配区段之后的内容则会保存到$’里。

if (“Hello there, neighbor” =~ /\s(\w+),/){

print “\$& is $&.\n”;

print “\$1 is $1.\n”

print “\$` is $`\n.”;

print “\$’ is $.’\n”;

}

这三个自动捕获变量也肯能是空字符串。

使用这三个变量，其他的正则表达式的运行速度将会变慢。尽量不使用。

在perl 5.10或以上版本，修饰符/p只会针对特定的正则表达式开启类似的自动捕获变量，但是他们的名字不再是$`、$&和$’，而是用${^PREMATCH}、${^MATCH}和${^POSTMATCH}表示。之前的例子可以写成：

use 5.010;

if (“Hello there, neighbor” =~ /\s(\w+),/p){

print “That actually matched ‘${^MATCH}’.\n”;

}

if (“Hello there, neighbor” =~ /\s(w+),/p){

print “That was (${$PREMATCH}) (${^MATCH}) (${^POSTMATCH}).\n”;

}

用户自定义的变量是不可以使用^开头的。

通用量词：

模式中的量词代编前面条目的重复次数。目前我们已经见过三个量词：\*、+以及？。如果这三个量词都不符合需要，则可以使用或括号{}的形式指定具体的重复次数范围。比如：

模式/a{5,15}/表示可匹配重复出现5到15次字母a。如果a出现20次，则只有前15个会匹配。模式/a{5,}表示匹配至少5次无上限次数的a。模式/\w{8}/会匹配正好有8个单词。

这三个量词都是简写，星号和量词{0,1}相同，表示零次或多次。加号相当于{1,}，表示一次以上。问号也可以写成{0,1}。

优先级：

用正则表达式处理文本:

用s///进行替换：只进行一次替换

实例：

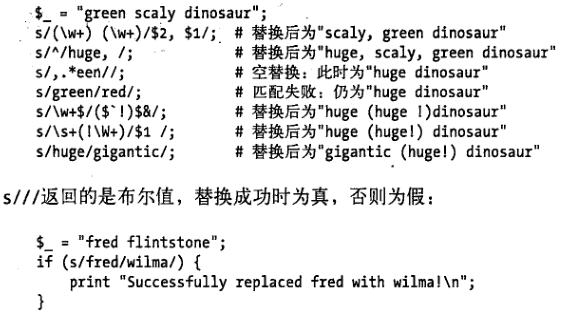
$\_ = “He’s out bowling with Barney tonight.”;

s/Barney/Fred/; 把Barney替换为Fred。

print “$\_\n”;

s/with (\w+)/against $1’s team/;

print “$\_\n”; #打印“He’s out bowling against Fred’s team tonight.”



用/g进行全局替换：

s///默认只会进行一次替换。/g修饰符可以让s///进行所有可能的替换：

实例：

$\_ = “home,sweet home!”;

s/home/cave/g;

print “$\_\n”; 打印“cave，sweet cave！”

将连续多个空格替换为单一空格：

$\_ = “Input data\t may have extra whitespace.”;

s/\s+/ /g; #现在它变成了“Input data may have extra whitespace.”

如何删除开头和结尾的空格：

s/^\s+//; #将开头的空白替换为空字符串。

s/\s+$//; #将结尾的空白替换成空字符串。

s/^\s+|\s+$//g; #去除开头和结尾的空白符。

绑定操作符：

无损替换：

同时保留原始字符串和替换后的字符串。

可以先用临时变量保留原始字符串，然后进行替换：

my $original = ‘Fred ate 1 rib’;

my $copy = $originam;

$copy =~ S/\d+ ribs?/10 ribs/;

也可以把后两步并作一步，先做赋值运算，然后针对运算结果进行替换：

(my $copy = $original) =~ s/\d+ ribs?/10 ribs/;

Perl 5.14添加修饰符/r，会保留原来的字符串变量中的值不变，而把替换结果作为替换操作的返回值返回：

use 5.014;

my $copy = $original =~ s/d+ robs?/10 ribs/r;

大小写转换：

\U转义符会将其后的所有字符转换成大写的：

实例：

$\_ = “I saw Barney with Fred.”;

s/(fred|barney)/\U$1/gi; #$\_现在成了“I saw BARNEY with FRED.”

类似地，\L转义符会将它后面的所有字符转换成小写的。

实例：

s/(fred|barney)/\L$1/gi; #$\_现在变成了”I saw barney with fred.”；

默认情况下，它们会影响之后全部的（替换）字符串。可以使用\E关闭大小写转换的功能：

s/(\w+) with (\w+)/\U$2\E with $1/I; #$\_ 替换后为 “I saw FRED with barney.”

使用小写形式(\l与\u)时，它们只会影响紧跟其后的第一个字符：

split：

根据给定模式拆分字符串。对于使用制表符、冒号、空白或任意符号分割不同字段数据的字符串来说，用这个操作符分解提取字段相当方便。用法如下：

my @fields = split /:/,”abc:def:g:h”; #得到(“abc”,”def”,”g”,”h”);

如果两个分隔符连在一起，就会产生空字段：

my @fields = split /:/, “abc:def::g:h”; #得到（”abc“,”def”,””,”g”,”h”）

split会保留开头处的空字段，却会舍弃结尾处的空字段。

实例：

my @fields = split /:/, “:::a:b:c:::”; #得到(“”,””,””,”a”,”b”,”c”);

join函数：

join函数不会使用模式，他的功能与split恰好相反：split会将字符串分解为若干片段，而join则会把这些片段合成一个字符串。用法如下：

my $result = join $glue, @pieces;

$glue表示分割符。

@value = (4,6,8,10,12);

my $z = join “-”,@values; #$z为 “4-6-8-10-12”

列表上下文中的正则：

在列表上下文中使用模式匹配操作符（m//）时，如果模式匹配成功，那么返回的是所有捕获变量的列表；如果匹配失败，则返回空列表：

实例：

$\_ = “Hello there,neighbor!”;

my($first,$second,$third) = /(\S+) (\S+), (\S+)/;

print “$second is my $third\n”;

之前在s///的例子中可以看到的/g修饰符同样也可以用在m//操作符上，其效果就是让模式能够匹配到字符串中的多个地方。

实例：

my $text = “Fred dropped a 5 ton granite block on Mr. Slate”;

my @words = ($text =~ /[a-z]/ig);

print “Result: @words\n”; #打印：Fred dropped a ton granite block on Mr Slate

如果模式中有多组圆括号，那么每次匹配就能捕获多个字符串。假设我们想把一个字符串变成哈希，就可以这样：

my $data = “Barney Rubble Fred Flintstone Wilma Flintstone”;

my %last\_name = ($data =~ /(\w+)\s+(\w+)/g);

每次模式匹配成功，就会返回一对被捕获的值，而这一对值正好成为新哈希的键-值对。

贪婪量词：

\*  .    +   ?四个。所谓贪婪，就是尽可能多的匹配字符。

非贪婪量词：

在贪婪量词\*  .    +   ?后面加上“？”号，则那么贪婪量词就变成了非贪婪量词，他会尽可能少的匹配字符。

跨行的匹配模式：

^和$都是代表整个字符串的开头和结尾的锚位。但当模式加上/m修饰符之后，就可以用它们匹配字符串内的每一行。

实例：

$\_ = “I’m much better \nthan Barney is \nat bowling,\nWilma.\n”;

print “Found ‘wilma’ at start of line\n” if /^Wilma\b/im;

变量$^I：

$^I当为空值时，不会对程序造成任何影响，但如果将其赋值为某个字符串，钻石操作符就会具有比平常更多魔力。当$^I这个变量是个只字符串时，这个字符串就会变成备份文件的扩展名：钻石操作符会先打开原文件，然后把文件名称改为\*.$^I，接着钻石操作符会打开一个新的文件并取名为原文件名，然后再对新文件进行读写操作。

从命令行直接编辑：

实例：

$ perl –p –i.bak –w –e ‘s/Randall/Randal/g’ fred\*.dat

-p参数可以让perl自动生成如下一段小程序：

while (<>) {

print;

}

添加-n参数可以将自动执行的print去掉。

-i.bak参数，其实就是在程序开始运行之前把$^I设为.bak，如果不需要备份，直接写-i，不加参数即可。

-w开启警告功能。

-e参数用来告诉perl后面跟着的是可执行的程序代码，

其他控制结构：

unless控制结构：

在if控制结构中，只有当条件表达式为真时，才执行某块代码，在代码块为假条件时执行，则使用unless：

实例：

unless ($fred =~ /\A[A-Z\_]\w\*\z/i){

print “……….”

}

伴随unless的else子句

实例：

unless ($mon =~ /\AFeb/){

print “………..”;

}else{

print “……”;

}

until控制结构：

实例：

until ($j > $i) {

$j \*= 2;

}

这个循环会一直执行，直到条件为真。与while的差别：until会在条件为假时重复执行，而while则是在条件为真时执行。因为条件判断发生在循环第一次迭代之前，所以他仍旧是一个执行零次以上的循环，和while循环一样。

表达式修饰符：

为了进一步简化代码书写，表达式后面可以接一个用于控制他行为的修饰符。

实例：

print “$n is a negative number.\n” if $n < 0;

这个if修饰符，他实际的作用相当于一个if语句块。

诸如其他：

&error(“Invalid input”) unless &valid($input);

$i \*= 2 until $i > $j;

print “ ”, ($n += 2) while $n < 10;

&greet($\_) foreach @person;

裸块控制结构：

所谓裸块，就是没有关键字或条件表达式的代码块。

实例：  
{

print “Please enter a number:”;

chomp(my $n = <STDIN>);

my $root = sqrt $n;

print “The square root of $n is $root.\n”;

}

elsif子句：

自增与自减：

自增操作符：++

实例：

my $bedrock = 42;

$bedrock++;

和其他将变量加1的方法一样，若有需要标量将会被自动创建：

my @people = qw{ fred barney fred Wilma dino barney fred pebbles };

my %count;

$count{$\_}++ foreach @people;

自减操作符：--

自增的值：

操作符：

last操作符能立即中止循环的执行，当看到last，循环就会结束。类似break

实例：

while (<STDIN>) {

if (/\_END\_/) {

last;

}elsif (/fred/) {

print;

}

}

在Perl中共有5种循环块，分别是for、foreach、while、until以及裸块。last操作符对整个循环块起作用。

last操作符只对当前正在运行中的最内层循环块发挥作用。

next操作符：

next操作符立刻结束当前这次循环迭代，跳到当前循环块的底端。类似continue

实例：

while (<>) {

foreach (split) {

$total++;

next if /\W/;

$valid++;

$count{$\_}++;

}

}

print “total things = $total, valid words = $valid\n”;

foreach $word (sort keys %count) {

print “$word was seen $count{$word} times.\n”;

}

redo操作符：

redo操作符能将控制返回到当前循环块的顶端，而不经过任何条件测试，也不会进入下一次循环迭代。

实例：

my @words = qw{ fred barney pebbles dino Wilma betty };

my $errors = 0;

foreach (@words) {

#redo指令会跳到这里。

print “Type the word ‘$\_’:”;

chomp(my $try = <STDIN>);

if ($try ne $\_) {

print “Sorry – That’s not right.\n\nn”;

$errors++;

redo;

}

}

redo可以在5中循环块里使用，并且在循环块嵌套的情况下只对最内层的循环起作用。

next和redo的区别：

next会正常进行下一次迭代，而redo则会重新执行这次迭代。

实例：

foreach (1..10) {

print “Iteration number $\_.\n\n”;

print “Please choose: last,next,redo,or none of the above?”;

chomp(my $choice = <STDIN>);

print “\n”;

last if $choice =~ /last/i;

next if $choice =~ /next/i;

redo if $choice =~ /redo/I;

print “That wasn’t any of the choices … onward!\n\n”;

}

print “That’s all,folks\n”;

带标签的块：

标签可以从内层循环对外层的循环块进行控制。

要对某个循环块加上标签，通常只要将标签及一个冒号放在循环前面就行。在之后的循环里，若有需要就可以在last、next或redo的后面加上这个标签：

实例：

LINE:wile (<>) {

foreach (split) {

last LINE fi /\_END\_/;

…

}

}

条件操作符：?:

实例：

my $location = $is\_weekend($day) ? “home” : “work”;

逻辑操作符：

AND(&&)和OR(||)

逻辑与操作符的左边表达式为假时，将不再检查右边的表达式。同理或操作符左边为真，则右边不在计算。

实例：  
if ( (9 <= $hour) && ($hour < 17) ) {

print “……………….”;

}

短路操作符：

+

定义或操作符：//

在发现左边的值属于已定义时进行短路操作，而不管该值属于逻辑真还是假。

实例：

use 5.010;

foreach my $try (0,undef,’0’,1,25) {

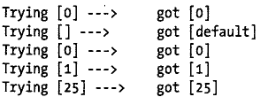
print “Trying [$try] -🡪”;

my $value = $try // ‘default’;

say “\tgoy [$value]”;

}

输出结果：只有在$try是undef的时候才会收到default字符串。



Perl模块

使用某模块：

实例：

use File::Basename;

仅选用模块中的部分函数：

在use声明中加入导入列表：

实例：

use File::Basename qw/ basename /;

什么函数也不导入，可以这样写：

use File::Basename qw / /;或者use File::Basename ();

不导入这些函数，但可以通过去全名的方式来调用相应的函数：

实例：

use File::Basename qw / /;

my $betty = &dirname($wilma);

my $name = “/usr/local/bin/perl”;

my $dirname = File::Basename::dirname $name;

其中模块里的dirname函数全名是File::Basename::dirname。

模块默认导入列表。

OO模块不会导入任何符号。调用方法时的必须功过全名方式来调用方法：

实例：

use File::Spec;

use Fiel::Basename;

print “Please enter a filename:”;

chomp(my $old\_name = <STDIN>);

my $dirname = dirname $old\_name;

my $basename = basename $old\_name;

$basename =~ s/^/not/;

my $new\_name = File::Spec->catfile($dirname,$basename);

rename($old\_name,$new\_name)

or warn “Can’t rename ‘$old\_name’ to ‘$new\_name’:$!”;

文件测试操作符：

所有文件测试符都是同一种形式：连字符加上一个字母，字母表示测试的意义，后面跟上要测试的文件名或文件句柄。大多数文件测试操作符返回的都是布尔真假值，少数返回的是表示特别意义的数据。

-e文件测试符：测试文件是否存在。

实例：

die “Oops! A file called ‘$filename’ already exists.\n”

if –e $filename;

-M测试符：返回文件最后一次修改时间到当前程序启动时刻之间的天数。

实例：

warn “Config file is looking pretty old!\n”

if –M CONFIG > 28;

文件测试操作符及其意义：

|  |  |
| --- | --- |
| 文件测试操作符 | 意义 |
| -r | 文件或目录，对目前(有效的)用户或组来说是可读的。 |
| -w | 文件或目录，对当前(有效的)用户或组来说是可写的。 |
| -x | 文件或目录，对目前(有效的)用户或组来说是可执行的。 |
| -o | 文件或目录，由目前(有效的)用户拥有。 |
| -R | 文件或目录，对实际的用户或组来说是可读的。 |
| -W | 可写。 |
| -X | 可执行。 |
| -O | 文件或目录，由实际的用户拥有。 |
| -e | 文件或目录，是存在的。 |
| -z | 文件存在而且没有内容(对目录来说永远为假) |
| -s | 文件或目录存在而且有内容(返回值是以字节为单位的文件大小) |
| -f | 是普通文件 |
| -d | 是目录 |
| -l | 是符号链接 |
| -S | 是socket类型文件 |
| -p | 是命令管道，也就是先入先出(fifo)队列 |
| -b | 是块设备文件(比如是某个可挂载的磁盘) |
| -c | 是字符设备文件(比如某个I/O设备) |
| -u | 文件或目录设置了setuid位 |
| -g | 文件或目录设置了setgid位 |
| -k | 文件或目录设置了sticky位 |
| -t | 文件句柄是TTY设备(类似系统函数isatty()的测试，不能对文件名进行此测试) |
| -T | 看起来像文本文件 |
| -B | 看起来像二进制文件 |
| -M | 最后一次修改后至今的天数 |
| -A | 最后一次被访问后至今的天数 |
| -C | 最后一次文件节点编号(inode)被变更后之间的天数 |

实例：

my @original\_files = qw/ fred barney betty Wilma pebbles dino bamm-bamm /;

my @big\_old\_files;

foreach my $filename (@original\_files) {

push @big\_old\_files, $filename

if –s $filename > 100\_000 and –A $filename > 90;

}

如果文件测试操作符后面没写文件名或文件句柄，那么默认的操作数就是$\_里面的文件名。但是文件测试操作符-t是个例外，因为此项测试对文件名(不可能是TTY)而言无用武之地，所以默认情况下它测试的是STDIN。

实例：测试一连串的 文件名来找出哪些是可读的

foreach (@lots\_of\_filenames) {

print “$\_ is readable\n” if –r;

}

测试同一文件的多项属性：

虚拟文件句柄：\_ 。作用，用来告知Perl用上次查询过的文件信息来做当前测试。

实例：

if (-r $file and –w \_) {

……..  
}

并非只能在一条语句中连续使用\_：

实例：

if (-r $file) {

print “The file is readable!\n”;

}

if (-w \_) {

print “The file is writable!\n”;

}

栈式文件测试操作符：

在Perl之后，可以使用，实例如下：

use 5.010;

if (-w -r $file) {

print “The file is both readable and writable!\n”;

}

使用栈式写法时，靠近文件名的测试会先执行，次序为从右往左。

stat和lstat函数：

stat函数能返回和同名的UNIX系统调用stat近乎一样的结果。

函数stat的参数可以是文件句柄(包括虚拟文件句柄\_)，或是某个会返回文件名的表达式。如果stat函数执行失败(通常是因为无效的文件名或是文件不存在)，它会返回空列表，要不然就返回一个含13个元素的数字列表，具体意义见下面由标量构成的列表：

my($dev,$ino,$mode,$nlink,$uid,$gid,$rdev,

$size,$atime,$mtime,$ctime,$blksize,$blocks)

= stat($filename);

对符号链接名调用stat函数将会返回符号链接指向对象的信息，而非符号链接本身的信息(除非该链接指向的对象目前无法访问)。

lstat用来使用在符号链接上。

就像文件测试操作符一样，stat和lstat的默认操作数是$\_。

localtime函数：

可以从stat函数中获取时间，但返回的是类似1180630098这样的形式，可以使用localtime函数将它转换为比较容易阅读的形式，比如”Thu May 31 09:48:18 2007”这样的字符串。

实例：

my $timestamp = 1180630098;

my $date = localtime $timestamp;

在列表上下文中，localtime会返回一个数字元素组成的列表，但其中某些元素并不是我们所需要的：

my($sec, $min, $hour, $day, $mon, $year, $wday, $yday, $isdst)

= locatetime $timestamp;

gmtime函数和localtime函数一样，只是gmtime返回的是世界标准时间(格林威治标准时间)。如果需要从系统时钟取得当前的时间戳，可以使用time函数。不提供参数的情况下，不论localtime或是gmtime函数，默认情况下都使用当前的time返回的时间值：

my $now = gmtime;

按位操作符：

&：按位与

|：按位或

^：异或。相同是0，不同是1

<<：按位左移。以0来调补最低有效位。

>>：按位右移。丢弃移出的最喜有效位。

~：按位取反。

实例：可以使用这些操作符对stat函数返回的&mode信息进行位操作

$mode是从配置文件CONFIG的stat信息中取出的状态值

warn ”Hey, the configuration file is world-writable!\n“

if $mode & 0002; #0002表示任意人都可以写该文件

my $classical\_mode = 0777 & $mode; #屏蔽额外的高位

my $u\_plus\_x = $classical\_mode | 0100; #将一个位设置为1

my $go\_minus\_r = $classical\_mode & (~ 0044); #将两个位都设为0

使用位字符串：

目录操作：

在目录树中移动，chdir操作符。

实例：

chdir ‘/etc’ or die “cannot chdir to /etc:$!”;

如果chdir不加参数，则默认进入用户主目录。这是少数不以$\_作为默认参数的情形之一。

文件名通配：

echo \*.pm

这里的echo命令其实并不知道如何展开\*.pm，因为shell会先把\*.pm展开成一些符合条件的文件名，然后再交给它处理。这对perl程序来说也是一样。

实例：

foreach $arg (@ARGV) {

print “one arg is $arg\n”;

}

运行时：$perl show-args \*.pm

show-args完全不必知道如何进行文件名通配处理，因为shell已经价格展开好的名称放在了@ARGV数组里面了。

在程序内部使用\*.pm之类的模式时，使用glob操作符：

实例：

my @all\_files = glob ‘\*’;

my @pm\_files = glob ‘\*.pm’;

其中，@all\_files会取得当前目录中的所有文件并按字母顺序排列，但不包含以点号开头的文件，这和shell中的做法完全不同。

文件名通配的另一种语法：

<\*>：尖括号语法。这种写法和my @all\_files = glob “\*”;完全一致

readline操作符：

目录句柄：

使用opendir操作符来打开目录句柄。读取内容使用readdir，关闭使用closedir，读取到的是目录里面的文件名，不是文件的内容。

实例：

my $dir\_to\_process = ‘/etc’;

opendir my $dh, $dir\_to\_process or die “Cannot open $dir\_to\_process:$!”;

foreach $file (readdir $dh) {

print “one file in $dir\_to\_process is $file\n”;

}

closedir $dh;

目录文件句柄返回的名称列表并未按照任何特定的顺序排列，列表中将包含所有的文件，包括以点号开头的文件，. 和 ..也在其中。

实例：处理以pm结尾的文件，可以在循环内使用一个跳过函数

while ($name = readdir $dh) {

next unless $name =~ /\.pm$/;

……

}

readdir操作符返回的文件名并不包含路径名，它们只是目录里的文件名而已。

实例：加上路径名得到文件的全名

opendir my $somedir, $dirname or die “Cannot open $dirname：$!”;

while (my $name = readdir $somedir) {

next if $name =~ /^\./; #跳过名称以点号开头的文件

$name = “$dirname/$name”;

next unless –f $name and –r $name; #只需要可读的文件

…

}

为了让程序更具有可移植性，可以使用File::Spec::Functions模块构造用于本地系统的合适文件名。

若是没有接上路径，文件测试符会在当前目录下查找文件，而不是在$dirname指定的目录下。

递归访问目录：

使用File::Find模块

删除文件：

在perl中，可以使用unlink操作符，并指定要删除的文件列表：

unlink ‘slate’, ‘bedrock’, ‘lave’;

unlink qw(slate bedrock lava);

unlink可以和glob函数一起使用：  
 unlink glob ‘\*.o’;

unlink的返回值代表成功删除的文件数目。

unlink不能删除目录

删除文件的权限跟文件本身的权限位无关，他取决于文件所在目录的权限。

重命名文件：

rename函数可以为现有文件更改名字，可以将文件移动到其他目录。

实例：

rename ‘over\_there/some/place/some\_fiel’, ‘some\_file’;

可以使用胖箭头表示改名先后：

rename ‘over\_there/some/place/some\_file’ => ‘some\_file’;

实例：批量把名称以.old结尾的文件改名为以.new结尾

foreach my $file (glob “\*.old”) {

my $newfile = $file;

$newfile =~ s/\.old$/.new/;

if (-e $newfile) {

warn “can’t rename $file to $newfile: $newfile exists\n”;

} elsif (rename $file => $newfile) {

#

} else {

warn “rename $file to $newfile failed:$!\n”;

}

}

链接与文件：

link硬链接，操作符不能跨设备

symlink符号链接，软连接，可以跨设备。符号链接不会增加inode的连接数。

readlink函数可以取得符号链接指向的位置，返回符号链接的指向位置，如果参数不是符号链接，则返回undef。

这两种链接都可以用unlink移除，unlink只是从目录里溢出该文件的链接条目，并将他们的连接数递减，必要时在释放inode。

创建和删除目录：

mkdir函数创建新的目录：

实例：

mkdir ‘fred’, 0755 or warn “Cannot make fred directory:$!”;

当在用户键入权限值当作命令行参数时，权限位需要用otc()函数转换为八进制。

实例：

my ($name, $perm) = @ARGV;

mkdir $name, otc($perm) or die “cannot create $name: $!”;

移除空目录可以用rmdir函数，每次只能删除一个空目录。

实例：

foreach my $dir (qw(fred barney betty)) {

rmdir $dir or warn “cannot rmdir $dir:$!\n”;

}

非空目录需要先删除里面的东西，然后再删除目录，使用unlink删除目录中的文件。

实例：

my $temp\_dir = “/tmp/scratch\_$$”;

mkdir $temp\_dir, 0700 or die “cannot create $temp\_dir :$!”;

….

unlink glob “$temp\_dir/\* $temp\_dir/.\*”;

rmdir $temp\_dir;

修改权限：

chmod函数，返回值为成功更改的条目数量。参数不可以是wrx。

实例：

chmod 0755, ‘fred’, ‘barney’;

修改隶属关系：

chown函数修改一系列文件的拥有者以及其所属组。在指定是必须给出数字形式的用户标识符以及组标识符。

实例：

my $user = 1004;

my $group = 100;

chown $user, $group, glob ‘\*.o’;

如果要处理的不是数字，而是类似于dange这样的字符串，需要使用getpwnam函数将用户名转换成用户编号，在用相应的getgrnam函数把用户组转换成组编号：

实例：

defined(my $user = getpwnam ‘merlyn’) or die ‘bad user’;

defined(my $group = getgrnam ‘users’) or die ‘bad group’;

chown $user, $group, glob ‘/home/merlyn/\*’;

这里使用defined函数来检查返回值不是undef，如果指定的用户或组不存在就会返回undef。

修改时间戳：

utime函数来改变某个文件最近更改或访问时间。前两个参数是新的访问时间和更改时间，其余参数就是要修改时间戳的文件名列表。

实例：

my $now = time;

my $ago = $now -24 \* 60 \*60;

utime $new, $ago, glob ‘\*’;

在文件有任何改动时，第三个时间戳(ctime值)一定会被设置成“now(当前时刻)”，所以utime函数无法修改它。

字符串与排序：

用index查找子字符串：

查找子字符串其实就是找出它在主字符串中的相对位置。

实例：

$where = index($big, $small);

perl会在$big字符串中寻找$small字符串首次出现的地方，并返回一个整数表示第一个字符的匹配位置，返回的字符位置是从零算起的。如果找不到子串，返回-1。

实例：

my $stuff = “Howdy world!”;

my $where = index($stuff, “wor”);

index函数每次都会返回首次出现子字符串的位置。可以添加可选的第三个参数来指定开始搜索的位置，这样index就不会从字符串的最开始寻找，而是从该参数指定的位置开始寻找子字符串。

实例：

my $stuff = “Howdy world!”;

my $where1 = index($stuff, “w”);

my $where2 = index($stuff, “w”, $where1 + 1);

my $where3 = index($stuff, “w”, $where2 + 1);

rindex函数可以从字符串末尾的地方开始找起，它和index返回的结果是一样的。

实例：

my $last\_slash = rindex(“/etc/passwd”, “/”);

rindex函数也有可选的第三个参数，但这里是用来限定返回值的上限。

my $fred = “Yabba dabba doo!”;

my $where1 = rindex($fred, “abba”); #7

my $where2 = rindex($fred, “abba”, $where1 - 1); #1

my $where3 = rindex($fred, “abba”, $where2 – 1); #-1

用substr操作子字符串

substr函数只处理较长字符串中的一小部分内容，大致用法如下：

my $patrt = substr($string ,$initial\_position, $length);

三个参数：一个原始字符串、一个葱岭起算的其实位置(类似index的返回值)、一个子字符串的长度，找到的子字符串将会被返回。

实例：

my $mineral = substr(“Fred J.Flintstone”, 8, 5); #返回“Flint”

my $rock = substr “Fred J. Flintstone”, 13, 1000; #返回“stone”

如果一直取到字符串结尾，则可以省略第三个参数(子字符串的长度)就行。

实例：

my $pebble = substr “Fred J. Flintstone”, 13; #返回“stone”

一个较大字符串的子字符串的起始位置可以为负值，表示从字符串结尾开始倒数。

实例：

my $out = substr(“some very long string”, -3, 2); #out为“in”

index与substr可以紧密合作。在下面的例子里，我们会取出以字符l的位置开头的子字符串：

实例：

my $long = “some very very long string”;

my $right = substr($long, index(long, “l”));

当原始字符串放在变量里面，就可以修改字符串被选取得部分内容。

实例：

my $string = “Hello, world!”;

substr($string, 0, 5) = ”Goodbye”;#$string 现在的值为”Goodbye, world!”

用来取代子字符串的长度并不一定要与被取代的字符串的长度相同。

用sprintf格式化字符串：

sprint函数与printf函数有相同的参数(可选的文件句柄参数除外)，但他返回的是所请求的字符串，而不会直接打印出来，可以将格式化之后的字符串存在变量里以便稍后使用。

实例：

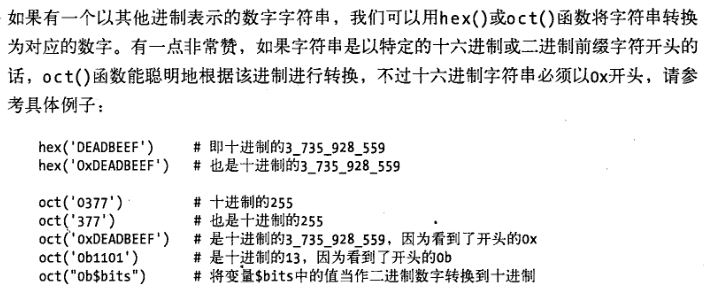
my $date\_tag = sprint

“%4d/%02d/%02d %2d:%02d:%02d”,

$yr, $mo, $da, $h, $m, $s;

$data\_tag会得到类似”2038/01/19 3:00:08“的结果。

非十进制数字字符串的转换：



高级排序：

sort操作符以ASCII码序队列表排序。

按哈希值排序：

实例：按照分数从高到低排序

sub by\_score { $score{$b} <=> $score{$a} };

my %score = (“barney” => 195, “fred” = > 205, “dino” => 30);

my @winners = sort by\_score keys %score;

<=>：飞船操作符。

实例：

my @arr = qw/5 9 3 8 2 1 0 7 92 5 6/;

my @result = sort {$a <=> $b} @arr;

print @result . “\n”;

按多个键排序：

实例：

my %score = (

“barney” => 195, “fred” => 205,

“dino” => 30, “bamm-bamm” => 195,

)

其中两个人的数据一样，

my @winners = sort by\_score\_and\_name keys $score;

sub by\_score\_and\_name {

$core{$b} <=> $score{a} #根据分数降序排列

or

$a cmp $b #分数相同的在按照名字的ASCII码序排列。

}

当<=>两边都为真时，这里优先级别低的短路操作符or就会被表达式的其余部分跳过，并返回我们所要的比较结果。但是当<=>两边相同的分数时，他会返回0，此时or后面的cmp操作符就会获得执行的机会。

也可以多级排序：

实例：此排序根据每个读者的未缴罚金(用子程序&fines计算，在此未列出)、目前他们借阅的本数(取自%items)、他们的姓名(先按姓排，后按名排，两者都取自哈希)，最后是顾客的ID编号。

@patron\_IDs = sort {

&fines($b) <=> &fines($a) or

$items{$b} <=> $items{$a} or

$family\_name{$a} cmp $family\_name{$b} or

$personal\_name{$a} cmp $family\_name{$b} or

$a <=> $b

} @patron\_IDs;

智能匹配与given-when结构：

智能匹配操作符：~~。他会根据两边的操作数的数据类型自动判断该用何种方式进行比较或匹配。

实例：在哈希%names查找任何匹配Fred的键，如果找到就打印一条信息，无法使用exists判定，因为需要给定确切的键，可以使用foreach遍历每个键，尝试用正则表达式匹配，跳过那些不匹配的，直到发现要找的键，保存到变量$flag中，然后用last跳出循环

my $flag = 0;

foreach my $key ( key %names ) {

next unless $key =~ /Fred/;

$flag = $key;

last;

}

使用智能匹配操作符，只要把哈希写在左侧，把正则表达式写在右侧：

use 5.010001;

say “I found a key matching ‘Fred’” if %names ~~ /Fred/;

比较两个数组：

实例：为了简单起见，暂时只考虑相同长度数组之间的比较，可以按数组索引依次遍历，取出相同位置的两个元素来比较。如果比较下来两者相等，则令计数器$equal自增1.循环结束后，如果$equal的数字和数组@names1的长度一样，就说明这两个数组完全相同。

my $equal = 0;

foreach my $index ( 0 .. $#name1 ) { # $#表示数组的最后一个index。

last unless $names1[$index] eq $names2[$index];

$equal++;

}

print “The arrays have the same elements!\n” if $equal == @names1;

使用智能匹配操作符可以这样写：

use 5.010001;

say “The arrays have the same elements!” if @names1 ~~ @names2;

实例：判断一个值是否在一个数组中

my @nums = qw( 1 2 3 27 42 );

my $result = max( @nums );

my $flag = 0;

foreach my $num ( @nums ) {

next unless result == $num;

$flag =1;

last;

}

print “The result is one of the input values\n” if $flag;

使用智能匹配操作符可以这样写：

use 5.0.10001;

my @nums = qw( 0 1 2 3 27 42 );

my $result = max( @nums );

say “The result [$result] is one of the input values (@nums)” if @nums ~~ $result;

智能匹配操作符对两边操作数的顺序一般没有要求，倒过来写也是可以的。

given语句：

given-when控制结构能够根据given后面的参数执行某个条件对应的语句块。这是perl用来应对C语言的switch语句的等效物。

实例：从命令行取第一个参数，$ARGV[0]，然后依次走一遍when条件测试，看看是否可以找到Fred。每个when语句块都对应于不同的处理方式。

use 5.010001;

given ( $ARGV[0] ) {

when ( ‘Fred’ ) { say ‘Name is Fred’ }

whein ( /fred/i ) { say ‘Name has fred in it’ }

when ( /\AFred/ ) { say ‘Name starts with Fred’ }

default { say “I don’t see a Fred” }

}

given会将参数化名为$\_，每个when条件都尝试用智能匹配对$\_作测试。为了测试可以用显式智能匹配的方式改写上面的例子：

use 5.010001;

give( $ARGV[0] ) {

when ( $ ~~ ‘Fred’ ) { say ‘Name is Fred’ }

when ( $\_ ~~ /\AFred/ ) { say ‘Name starts with Fred’ }

when ( $\_ ~~ /fred/I ) { say ‘Name has fred in it’ }

default { say “I don’t see a Fred” }

}

与if-elsif-else不同，given-when可以在满足某个条件的基础上继续测试其他条件，而if-elsif-else一旦满足了某个条件，就只能执行对应的那个语句块。

除非明确指定，否则在when语句块后面都好像有一句break，它告诉Perl现在就跳出given-when结构，继而执行后面的程序。

如果在when语句块的末尾使用continue，Perl就会尝试执行后续的when语句了。当另一个when的条件满足时，Perl会执行对应语句块(同样，默认退出整个控制结构，除非写清楚)

。在每个when语句块的末尾写上continue，就意味着所有的条件测试都会执行：

实例：

use 5.010001;

given ( $ARGV[0] ) {  
 when ( $\_ ~~ ‘Fred’ ) { say ‘Name is Fred’;continue }

when ( $\_ ~~ /Fred/I ) { say ‘Name has fred in it’; continue }

when ( $\_ ~~ /\AFred/ ) { say ‘Name starts with Fred’} #此处不写continue，不然默认

#条件总是会执行

default { say “I don’t see a Fred”}

}

笨拙匹配：

笨拙匹配指的是之前已经讲过的正则表达式匹配方式。

实例：

use 5.010001;

given ( $ARGV[0] ) {

when ( ‘Fred’ ) { #智能匹配

say ‘Name is Fred’; continue }

when ( $\_ =~ /\AFred/ ) { #笨拙匹配

say ‘Name starts with Fred’; continue }

when ( /Fred/I ) { #智能匹配

say ‘Name has fred in it’; }

default { say “I don’t see a Fred” }

}

智能匹配操作符用于判断事物是否相同(或者差不多是相同)，所以在西药比较大小时，就不能用智能匹配了。

某些特定的情况下，Perl会自动使用笨拙匹配方式。若在when里调用某个子程序，Perl会根据返回值的真假作判断：

use 5.010001;

given ( $ARGV[0] ) {

when ( name\_has\_fred( $\_ ) ) { #笨拙模式

say ‘Name has fred in it’; continue }

}

内置的函数，当返回真假值是同样不会用智能匹配。

否定的表达式，包括否定的正则表达式，都不会使用智能匹配方式。

实例：

use 5.010001;

given( $ARGV[0] ) {

when( ! $boolean ) { #笨拙匹配

say ‘Name has fred in it’ }

when( ! /fred/i ) { #笨拙匹配

say ‘Doed not match Fred’ }

}

多个条目的when匹配：

因为given只能一次接受一个参数，当需要遍历许多条目的时候，可以将given放到foreach里面进行循环。

实例：遍历@names，可以依次将当前的元素赋值到$name，然后再用given：

use 5.010001;

foreach my $name ( @names ) {

given( $name ) {

……

}

}

简单写法：

use 5.010001;

foreach ( @names ) {

when ( /fred/i ) { say ‘Name has fred in it’; continue }

when ( /\AFred/ ) { say ‘Name starts with Fred’; continue }

when ( ‘Fred’ ) { say ‘Name is Fred’; }

default { say “I don’t see a Fred” }

}

进程管理：perl直接运行其他程序并管理这些孩子

system函数：

实例：从perl调用UNIX的date命令

system ‘date’;

由外部命令date输出的日期与时间字符串会立即传送到当前Perl程序的STDOUT句柄所指向的地方。

通常提供给system函数的参数就是那些一般在shell中键入的命令，当用ls –l $HOME之类的复杂命令时，只要把它全部放进参数里就行了：

实例：

system ‘ls –l $HOME’;

这里的$HOME是shell的环境变量，所以用的不是双引号，而是单引号，否则，因为美元符号同时也是通知Perl的标量符号，所以放在双引号会进行变量内插，而内插之后shell看到的这个变量就不是环境变量了。还可以这样写：

实例：

system “ls –l \$HOME”;

当运行的外币命令是交互式命令时，程序将从标准输入(继承自perl的STDIN)等待回应。子程序正在运行时，Perl会等待其结束。

实例：

#!/usr/bin/perl

system 'mysql -uroot -p';

print "test!!!!!!!!!!!!!!!!!!!!!!!\n";

system 'sudo mysql -uroot -pnfsnfs';

print "test...............................\n";

system 'mysql -uroot -p';

print "test\*\*\*\*\*\*\*\*\*\*\*\*\*\n";

这样的shell程序：

system “long\_running\_command with parameters &”;

shell将该命令设置为后台进程并退出继续执行perl。perl会注意到shell已经退出，继续进行其他的事情。perl无法管理long\_running\_command进程， 并且也不知道他的存在。

可以在system的参数中写入简短的shell脚本：

实例：

system ‘for i in \*;do echo == $i ==;cat $i ;done’;

避免使用shell

system操作符也可以用一个以上的参数来调用，如此一来，不管你给的文本有多复杂，都不会用到shell。

实例：

my $tarfile = ‘something\*wicked.tar’;

my @dirs = qw( fred|flintstone <barney&rubble> betty );

system ‘tar’, ‘cvf’, $tarfile, @dirs;

system操作符的返回值是根据子进程的结束状态来决定的。

实例：

unless ( system ‘date’ ) {

print “We gave you a date, OK!\n”;

}

环境变量：

在Perl中，环境变量可以通过特殊的%ENV哈希取得，其中每个键都代表一个环境变量。在程序开始运行时，%ENV会保留从父进程(通常为shell)继承而来的设定值。修改此哈希就能改变环境变量，他会被新的进程继承，也可以有Perl本身来使用。

实例：需要运行make程序，并且在私有目录中寻找命令的首选位置，还要禁止IFS环境变量

$ENV{ ‘PATH’ } = “/home/rootbeer/bin:$ENV{ ‘PATH’ }”;

delete $ENV{ ‘IFS’ };

my $make\_result = system ‘make’;

**exec函数：**

system函数的语法适用于exec函数，但是system函数会创建子进程，子进程会在Perl睡眠期间执行任务。而exec函数会在Perl调用外部命令之后，Perl进程退出。写在exec调用之后的任何代码都不会运行，除非exec调用出错。

实例：运行/tmp目录下的bedrock命令并带上-o args1以及程序本身所调用的参数，可以这样写：

chdir ‘/tmp’ or die “Cannot chdir /tmp:$!”;

exec ‘bedrock’, ‘-o’, ‘args1’, @ARGV;

**用反引号捕获输出结果：**

无论是system还是exec，所执行的命令的输出结果都是送往Perl的标准输出。

若要提取该输出信息，只要用反引号代替单引号或者双引号就可以了。

实例：

my $now = `date`;

print “The time is now $now”;

反引号里面的内容就相当于单个参数形式的system函数调用并会以双引号内的字符串的方式进行解释。换句话说，我们可以在里面使用反斜线转义和变量，他们会被适当的展开。

实例：取得一系列Perl的说明文档，可以重复执行perldoc命令，每次使用不同的参数

my @functions = qw{ int rand sleep length hex eof not exit sqrt umask };

my %about;

foreach ( @functions ) {

$about{$\_} = `perldoc -t -f $\_`;

}

**qx操作符：引起操作符**

实例：

foreach (@functions) {

$about{$\_} = qx(perldoc -t -f $\_);

}

选用这种写法可以避免转义被引起内容中出现的分隔符。如果要执行的命令中本身就需要包含反引号，那就可以使用qx()避免频繁转义带来的干扰。选用单引号作为分隔符的话，可以禁止变量内插。qx()以及``(反单引号)以及qx``(反单引号)会进行变量内插，而qx''(单引号)不会进行变量内插，

实例：比如希望选用shell的进程编号ID变量而不是Perl的时候，就可以使用qx''(单引号)避免perl内插该变量。

实例：

my $output = qx'echo $$';#shell的进程ID

my $output = qx(echo $$);#Perl的进程ID

**在列表上下文中使用反引号**

如果命令会输出很多行，那么在标量上下文中使用反引号会得到一个很长的字符串，其中包含换行符。不过，如果是在列表上下文使用同样的反引号，则会返回输出字符串按行拆分的列表。

**用IPC::System::Simple执行外部进程**

可以直接使用IPC::System::Simple提供的同名函数取代内置的System函数。

实例：

use IPC::System::Simple;

my $tarfile = 'something\*wicked.tar';

my @dirs = qw( fred|flintstone <barney&rubble> betty );

system 'tar', 'cvf', $tarfile, @dirs;

另外一个systemx函数，在执行外部命令时不会通过shell调用，所以不会碰到shell导致的意外状况：

systemx 'tar', 'cvf', $tarfile, @dirs;

如果要捕获外部命令的输出，只需要把system或systemx改成capture或capturex就可以了。

实例：

my @output = capturex 'tar', 'cvf', $tarfile, @dirs;

**通过文件句柄执行外部进程**

以上的方法都是由Perl同步控制子进程：启动一个命令，然后等待它结束，然后也许还会捕获其输出。Perl还可以启动一个异步运行的子进程，并和他保持通信，直到子进程结束为止。

要启动并发运行的子进程，需要将命令放在open调用的文件名部分，并在它前面或后面加上竖线，也就是管道符。在两个参数的形式中，管道符号安放在要执行的命令的开头或者结尾。

实例：

open DATE, 'date|' or die "cannot pipe from date:$!";

open MAIL, '|mail merlyn' or die "cannot pipe to mail:$!";

第一个例子里的竖线在命令的右边，表示该命令执行时它的标准输出会连接到供程序读取的文件句柄DATE，也就是date的输出输入到了文件句柄DATE中。第二个例子里，竖线在左边，所以该命令的标准输入会连接到供程序写入的文件句柄MAIL，也就是。。。。（没写出例子，这个不是很懂）。

三个参数的形式看起来有点奇怪，因为就读取用的文件句柄而言，管道符号写在了命令“占位符”的后面。其实这里定义的是文件句柄的打开模式，如果需要读取用的文件句柄，就用-|，如果需要写入用的文件句柄，就用|-，-的位置就好比是要执行的命令在管道传递中的位置。

实例：

open my $date\_fh, '-|', 'date' or die "cannot pipe from date:$!";

open my $mail\_fh, '|-', 'mail merlyn'

or die "cannot pipe to mail:$!";

使用管道方式的open操作还可以使用三个以上的参数。第四个及其后的所有参数都将作为要执行的外部命令的参数，所以上面的写法可以拆分成下面的形式：

实例：  
 open my $mail\_fh, '|-', 'mail', 'merlyn'

or die "cannot pipe to mail:$!";

这个文件句柄是连接到进程而非文件，所以如果要从以读取模式打开的文件句柄读取数据，只要采用正常的读取方式即可：  
实例：

my $now = <$date\_fh>;

要想发送数据到mail进程(它此时正在等待从标准输入读取发送给merlyn的邮件正文)，只要利用输出到文件句柄的print就能完成：

实例：

print $mail\_fh "The time is now $now";

关闭连接至进程的文件句柄会让Perl等待该进程结束以取得它的结束状态。结束状态会存入$?变量，它的值就与system函数返回的数值一样：零表示成功，非零值表示失败。

如果子进程不时有数据要送给父进程，必须使用管道。

实例：find命令依照文件属性寻找文件。虽然可以将find命令放到反引号内，但如果在每找到一个文件时就立刻取得他的名称，使用管道是比较好的做法。

open my $find\_fh, '-|',

'find', qw( / -atime +90 -size +1000 -print )

or die "fork:$!";

while ( <$find\_fh> ) {

chomp;

printf "%s size %dK last accessed %.2f days ago\n",

$\_, (1023 + -s $\_)/1024, -A $\_;

}

这里的find命令是查找那些90内未被访问过的占用空间超过1000个块的大文件。在find工作时，Perl会等待。每找到一个文件，Perl会对每个传进来的文件名做出响应并进一步显示文件的相关信息供分析。如果使用反引号来获取外部程序输出信息时，就得等到find彻底搜索网才能看到信息的输出。

**用fork进行深入和复杂的工作**

实例：

system 'date';

这是高级调用，它的低级系统调用，大致可以写成这样：

defined( my $pid = fork ) or die "Cannot fork:$!";

unless ($pid) { #当判断条件是假时，unless才执行，和if刚好相反。

exec 'date'; #子进程

die "cannot exec date:$!";

}

waitpid($pid, 0);#父进程

fork返回子进程的PID，为0。

**发送及接收信号**

实例：给进程ID 4201发送 2这个信号

kill 2, 4201 or die "Cannot signal 4201 with SIGINT:$!";

也可以使用字符串"INT"代替2发送信号：

kill 'INT', 4201 or die "Cannot signal 4201 with SIGINT:$!";

如果进程早已退出，就会收到表示失败的返回值。可以发送编号为0的特殊信号来判断某个进程是否依然存活。编号为0的特殊信号表示“看我能否像某个进程发送信号，但目前并不是真的要发送，所以请不要打扰它。”

实例：

unless (kill 0,$pid) {

warn "$pid has gone away!";

}

接收信号

实例：在接收到Control+C后，进行某些东西的清理

my $temp\_directory = "/tmp/myprog.$$";

mkdir $temp\_directory, 0700 or die "Cannot create $temp\_directory:$!";

sub clean\_up {

unlink glob "$temp\_directory/\*";

rmdir $temp\_directory

}

sub my\_int\_handler {

&clean\_up();

die "interrupted, exitting...\n";

}

$SIG{'INT'} = 'my\_int\_handler';

.... #临时文件被创建，然后按下了Control+C

&clean\_up();#这个是正常执行流程的结尾部分，并不是按下Control+C之后执行的。

其中，对特殊的哈希%SIG赋值会启动信号处理程序(直到撤销为止)，哈希键是信号的名称(不用写固定的SIG前缀)，哈希值是子程序名称(不用写&)。当收到SIGINT信号，Perl就会暂停当前工作并立刻执行信号处理子程序。

当信号处理子程序没有结束进程而是直接返回，那么程序就会从先前中断的地方继续运行，如果该信号只是要中断某些事情，而不是停止整个程序的话，这种时候就该用返回而非退出。

实例：

my $int\_count = 0;

sub my\_int\_handler { $int\_count++ }

$SIG{'INT'} = 'my\_int\_handler';

...;

wile ( <SOMEFILE> ) {

...;

if ($int\_count) {

print "[processing interrupterd...\n]";

last;

}

}

**高级Perl技巧**

**切片**

实例：存放信息为读者姓名，借书证编号，地址，家庭电话，工作电话和借阅数量

fred flintstone:2168:301 Cobblestone Way:555-1212:555-2121:3

barney rubble:709918:3128 Granite Blvd:555-333:555-3438:0

从中取出借书证编号和借阅数量：

while (<$fh>) {

chomp;

my @items = split /:/;

my($care\_num,$count) = ($items[1],$items[5]);

....

}

此处的@items数组不会有其他的用处，是一种浪费，可以使用一组标量来容纳split的结果，但同样有浪费：

my($name, $card\_num, $addr, $home, $work, $count) = split /:/;

可以使用undef写法，如果被赋值的列表中含有undef的话，就直接忽略源列表的相应元素：

实例：

my(undef, $card\_num, undef, undef, undef, $count) = split /:/;

这样可以避免引入不需要的，但需要弄清楚undef的数量。若是使用列表元素数量稍微多些就麻烦了，比如stat的结果，如果写错undef的数量，获得的值可能就会出错。

更好的办法是Perl可以把列表当作数组，用索引取得里面的值，这既是所谓的列表切片。

实例：获取stat返回的列表的9号元素mtime

my $mtime = (stat $some\_file)[9];

这里stat周围的括号是必须的，因为需要他们产生列表上下文。

列表切片必须在一对圆括号引起的列表后面有一个由方括号括起的下标表达式。

实例：

my $card\_num = (split /:/)[1];

my $count = (split /:/)[5];

还可以这样写：

my($card\_num, $count) = (splig /:/)[1,5];

切片是从列表中读取少量数据的最简单方法。

实例：从列表中那个取出第一个和最后一个元素，借助索引-1,代表最后一个元素

my($first, $last) = (sort @names)[0, -1];

切片的下标可以是任意顺序的，也可以是重复的。

实例：

my @names = qw{ zero one two three four five six seven eitght nine };

my @numbers = ( @names )[ 9, 0, 2, 1, 0];

print "Bedrock @numbers\n";

**数组切片**

从数组(而不是从列表)切出元素时，圆括号并非必须的。

实例：

my @numbers = @names[ 9, 0, 2, 1, 0 ];

切片可以被直接内插到字符串中：

实例：

my @names = qw{ zero one two three four five six seven eight nine };

print "Bedrock @names[ 9, 0, 2, 1, 0 ]\n";

**哈希切片**

实例：

**捕获错误：**

用eval

将代码包裹在eval块里：

实例：

eval { $barney = $fred / $dino };

即使$dino是零，这一行代码也不至于让程序崩溃。只要eval发现在它的监察范围内出现致命错误，就会立刻停止运行整个块，退出后继续运行其余的代码。eval块的末尾有一个分号，因为eval只是一个表达式，而不是类似于while或foreach那样的控制结构。

eval的返回值就是语句块中最后一条表达式的执行结果，如果eval捕获到了错误，那么整个语句块返回undef，所以可以利用定义或操作符(//)对最终的变量设定默认值。

实例：

use 5.010;

my $barney = eval { $fred / $dino } // 'NaN';

当运行eval块的期间出现致命错误时，停下来的只是这个语句块，整个程序不会崩溃。

如果eval捕获到错误，eval返回undef，并且在特殊的变量$@中设置错误消息。如果没有捕获到错误，则$@内容就是空的。

实例：

use 5.010;

my $varney = eval { $fred / $dino } // 'NaN';

print "I couldn't divide by \$dino:$@" if $@;;

也可以检查返回值来判断：  
实例：

unless( eval { $fred / $dino } ) {

print "I couldn't divide by \$dino:$@" if $@;

}

有时候测试的部分即使成功了也并没有什么有意义的返回值，所以得另外构造一个有返回值的代码块，如果eval捕获到了错误，就不会执行最后一条语句，下面的例子中也就是1：

实例：

unless( eval { some\_sub(); 1 } ) {

print “I couldn’t divide by \$dino:$@” if $@;

}

在列表上下文中，捕获到错误的eval会返回空列表。

实例：当eval失败的话，@averages最终只会得到两个元素，因为eval返回的是空列表，等于不存在。

my @averages = ( 2/3, eval { $fred / $dino }, 22 / 7 );

eval块和其他语句块一样，所以可以设定词法变量(my)的新作用域，并且块内的语句数目不限。

实例：

foreach my $person (qw/ fred Wilma betty barney dino pebbles /) {

eval {

open my $fh, ‘<’, $person

or die “Can’t open file ‘$person’: $!”;

my($total, $count);

while (<$fh>) {

$total += $\_;

$count++;

}

my $average = eval { $total/$count } // ‘NaN’;

print “Average for file $person was $average\n”;

&do\_something($person, $average);

};

if ($@) {

print “An error occurred ($@), continuing\n”;

}

}

一共有四种类型的错误eval是无法捕获的：

1. 源代码中的语法错误
2. 让perl解释器本身崩溃的错误，比如内存溢出或者收到无法接管的信号。因为perl解释器已经退出，所以也就无法运行，eval自然无法捕获。
3. eval块无法捕获警告，不管是由用户发出的(通过warn函数)，还是Perl自己内部发出的(通过打开-w这个命令行选项，或者使用use warnings编译指令)。
4. exit操作符会立即终止程序运行，就算是从eval块内的子程序来调用它。

eval字符串：

实例：这种形式的eval会把拿来的字符串直接当作Perl源代码编译，然后执行，就好比你手工在程序里敲上这段代码。

my $operator = ‘unlink’ ;

eval “$operator \@files;”

更为高级的错误处理：

在Perl里面最基本的做法是，用die抛出异常，然后用eval捕获异常，通过识别保存在$@里面的错误消息来判断问题所在

实例：

eval {

…;

die “An unexpected exception message” if $undexpected;

die “Bad denominator” if $dino == 0;

$barney = $fred / $dino;

}

if ( $@ =~ /unexpected/ ) {

…;

}

elsif( $@ =~ /denominator/ ) {

…;

}

这样的代码有弊端，最明显的就是$@变量的动态作用域问题了。由于$@是一个特殊变量，而eval可能会嵌套在另一个eval中，那么就需要确保这里出现的错误不干扰高层出现的错误。

实例：

{

local $@; #不干扰高层错误。

eval {

…;

die “An unexpected exception message” if $undexpected;

die “Bad denominator” if $dino == 0;

$barney = $fred / $dino;

}

if ( $@ =~ /unexpected/ ) {

…;

}

elsif( $@ =~ /denominator/ ) {

…;

}

}