



# Perl 教程

**Perl** 是 **Practical Extraction and Report Language** 的缩写，可翻译为 "实用报表提取语言"。

Perl 是高级、通用、直译式、动态的程序语言。

Perl 最初的设计者为拉里·沃尔（Larry Wall），于1987年12月18日发表。

Perl 借用了C、sed、awk、shell脚本以及很多其他编程语言的特性。

Perl 最重要的特性是Perl内部集成了正则表达式的功能，以及巨大的第三方代码库CPAN。

## 谁适合阅读本教程？

本教程适合想从零开始学习 Perl 编程语言的开发人员。当然本教程也会对一些模块进行深入，让你更好的了解 Perl 的应用。

## 学习本教程前你需要了解

在继续本教程之前，你应该了解一些基本的计算机编程术语。如果你学习过PHP，ASP等其他编程语言，将有助于你更快的了解 Perl 编程。

## 第一个 Perl 程序

对于大多数程序语言，第一个入门编程代码便是"Hello World! "，以下代码为使用 Perl 输出"Hello World! "：

### 实例

```
#!/usr/bin/perl
print "Hello, World!\n";
```

运行实例 »

[点我分享笔记](#)

[反馈/建议](#)



# Perl 简介

Perl，一种功能丰富的计算机程序语言，运行在超过100种计算机平台上，适用广泛，从大型机到便携设备，从快速原型创建到大规模可扩展开发。

Perl 语言的应用范围很广，除CGI以外，Perl被用于图形编程、系统管理、网络编程、金融、生物以及其他领域。由于其灵活性，Perl被称为脚本语言

中的瑞士军刀。

## 什么是 Perl?

Perl是由Larry Wall设计的，并由他不断更新和维护的编程语言。

Perl具有高级语言（如C）的强大能力和灵活性。事实上，你将看到，它的许多特性是从C语言中借用来的。

Perl与脚本语言一样，Perl不需要编译器和链接器来运行代码，你要做的只是写出程序并告诉Perl来运行而已。这意味着Perl对于小的编程问题的快速解决方案和为大型事件创建原型来测试潜在的解决方案是十分理想的。

Perl提供脚本语言（如sed和awk）的所有功能，还具有它们所不具备的很多功能。Perl还支持sed到Perl及awk到Perl的翻译器。

简而言之，Perl像C一样强大，像awk、sed等脚本描述语言一样方便。

## Perl 优点

相比C、Pascal这样的"高级"语言而言，Perl语言直接提供泛型变量、动态数组、Hash表等更加便捷的编程元素。

Perl具有动态语言的强大灵活的特性，并且还从C/C++、Basic、Pascal等语言中分别借鉴了语法规则，从而提供了许多冗余语法。

在统一变量类型和掩盖运算细节方面，Perl做得比其他高级语言(如：Python)更为出色。

由于从其他语言大量借鉴了语法，使得从其他编程语言转到Perl语言的程序员可以迅速上手写程序并完成任务，这使得Perl语言是一门容易用的语言。

Perl 是可扩展的，我们可以通过CPAN（"the Comprehensive Perl Archive Network"全面的 Perl 存档网络）中心仓库找到很多我们需要的模块。

Perl 的 `mod_perl` 的模块允许 Apache web 服务器使用 Perl 解释器。

## Perl 缺点

也正是因为Perl的灵活性和"过度"的冗余语法，也因此获得了仅写（write-only）的"美誉"，因为Perl程序可以写得很随意（例如，变量不经声明就可以直接使用），但是可能少写一些字母就会得到意想不到的结果（而不报错），许多Perl程序的代码令人难以阅读，实现相同功能的程序代码长度可以相差十倍百倍，这就令程序的维护者（甚至是编写者）难以维护。

同样的，因为Perl这样随意的特点，可能会导致一些Perl程序员遗忘语法，以至于不得不经常查看Perl手册。

建议的解决方法是在程序里使用`use strict;`以及`use warnings;`，并统一代码风格，使用库，而不是自己使用"硬编码"。Perl同样可以将代码书写得像Python或Ruby等语言一样优雅。

很多时候，perl.exe进程会占用很多的内存空间，虽然只是一时，但是感觉不好。

[□ Perl 教程](#)

[Perl 环境安装](#) □

[□ 点我分享笔记](#)

[反馈/建议](#)

Copyright © 2013-2018 菜鸟教程 [runoob.com](#) All Rights Reserved. 备案号：闽ICP备15012807号-1



[首页](#) [HTML](#) [CSS](#) [JS](#) [本地书签](#)

[□ Perl 简介](#)

[Perl 基础语法](#) □

## Perl 环境安装

在我们开始学习 Perl 语言前，我们需要先安装 Perl 的执行环境。

Perl 可以在以下平台下运行：

Unix (Solaris, Linux, FreeBSD, AIX, HP/UX, SunOS, IRIX etc.)

Win 9x/NT/2000/

WinCE

Macintosh (PPC, 68K)

Solaris (x86, SPARC)

OpenVMS

Alpha (7.2 and later)

Symbian

Debian GNU/kFreeBSD

MirOS BSD

等等...

很多系统平台上已经默认安装了 **perl**，我们可以通过以下命令来查看是否已安装：

```
$ perl -v
```

```
This is perl 5, version 18, subversion 2 (v5.18.2) built for darwin-thread-multi-2level
```

```
(with 2 registered patches, see perl -V for more detail)
```

```
Copyright 1987-2013, Larry Wall
```

```
.....
```

如果输出以上信息说明已安装，如果还未安装，可以看接下来的安装指导。

## 安装 Perl

我们可以在 **Perl** 的官网下载对应平台的安装包：<https://www.perl.org/get.html>



[Unix/Linux](#)

✓ Included  
(may not be latest)



[Mac OS X](#)

✓ Included  
(may not be latest)



[Windows](#)

↓ [Strawberry Perl](#) [↗](#)  
↓ [ActiveState Perl](#) [↗](#)

## Unix 和 Linux 安装 Perl

Unix/Linux 系统上 Perl 安装步骤如下：

通过浏览器打开 <http://www.perl.org/get.html>。

下载适用于 Unix/Linux 的源码包。

下载 **perl-5.x.y.tar.gz** 文件后执行以下操作。

```
$ tar -xzf perl-5.x.y.tar.gz

$ cd perl-5.x.y

$ ./Configure -de

$ make

$ make test

$ make install
```

接下来我们如果 **perl -v** 命令查看是否安装成功。

安装成功后，Perl 的安装路径为 `/usr/local/bin`，库安装在 `/usr/local/lib/perlXX`，XX 为版本号。

## Window 安装 Perl

Perl 在 Window 平台上有 ActiveStatePerl 和 Strawberry Perl 编译器。

ActiveState Perl 和 Strawberry Perl 最大的区别是 Strawberry Perl 里面有多包含一些 CPAN 里的模块，所以 Strawberry Perl 下载的安装文件有 80 多 M，而 ActiveState Perl 只有 20M 左右。

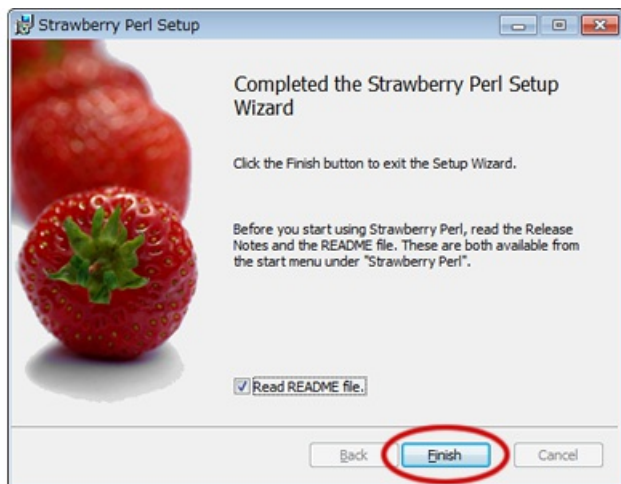
我们这里使用了 Strawberry Perl。

Window 系统上 Perl 安装步骤如下：

Strawberry 安装包链接：<http://strawberryperl.com>。

下载对应你系统的版本：32bit 或 64bit。

下载后双击打开，按安装向导一步步安装即可。



## Mac OS 安装 Perl

Mac OS 系统一般默认已经安装了 Perl，如果未安装则执行以下步骤：

通过浏览器打开 <http://www.perl.org/get.html>。

下载适用于 Mac OS 的源码包。

下载 **perl-5.x.y.tar.gz** 文件后执行以下操作。

```
$ tar -xzf perl-5.x.y.tar.gz

$ cd perl-5.x.y
```

```
$ ./Configure -de

$ make

$ make test

$ make install
```

执行成功后 Perl 的安装路径为 `/usr/local/bin`，库安装在 `/usr/local/lib/perlXX`, `XX` 为版本号。

## 运行 Perl

Perl 有不同的执行方式。

### 1、交互式

我们可以在命令行中直接执行 perl 代码，语法格式如下：

```
$perl -e <perl code>          # Unix/Linux

或

C:>perl -e <perl code>        # Windows/DOS
```

命令行参数如下所示：

选项	描述
-d[:debugger]	在调试模式下运行程序
-Idirectory	指定 @INC/#include 目录
-T	允许污染检测
-t	允许污染警告
-U	允许不安全操作
-w	允许很多有用的警告
-W	允许所有警告
-X	禁用使用警告
-e program	执行 perl 代码
file	执行 perl 脚本文件

### 2、脚本执行

我们可以将 perl 代码放在脚本文件中，通过以下命令来执行文件代码：

```
$perl script.pl          # Unix/Linux

或
```

```
C:>perl script.pl # Windows/DOS
```

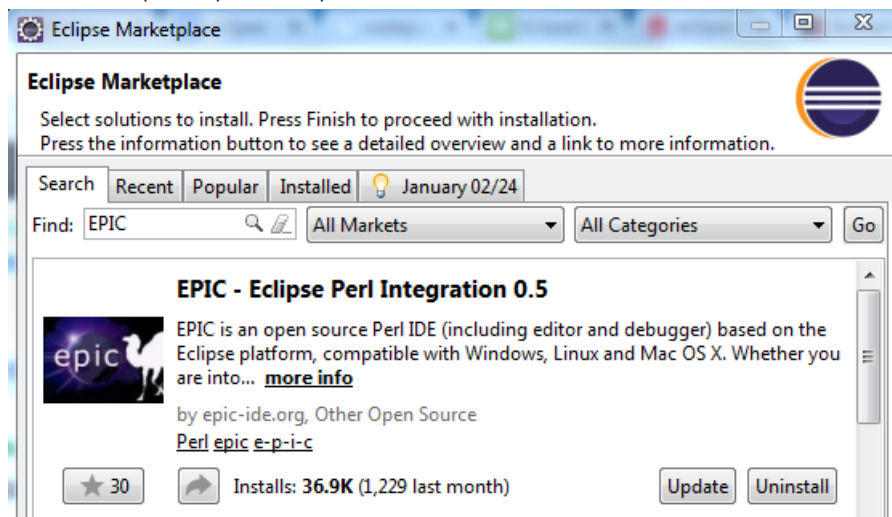
## 集成开发环境(IDE: Integrated Development Environment)

我们可以在一些图形用户界面(GUI) 环境下执行 perl 脚本。以下推荐两款常用的 Perl 集成开发环境：

**Padre**: Padre 是一个为 Perl 语言开发者提供的集成开发环境，提供了语法高亮和代码重构功能。

**EPIC**: EPIC 是 Perl Eclipse IDE 的插件，如果你熟悉 Eclipse，你可以使用它。

安装步骤：Help->Eclipse Marketplace->输入EPIC-> 选择安装并更新即可。



## Cloud Studio

**Cloud Studio** 是基于浏览器的集成式开发环境，支持绝大部分编程语言，包括 HTML5、PHP、Python、Java、Ruby、C/C++、.NET 等等，无需下载安装程序，一键切换开发环境。**Cloud Studio** 提供了完整的 Linux 环境，并且支持自定义域名指向，动态计算资源调整，可以完成各种应用的开发编译与部署。



1 篇笔记

#1

写笔记

在 perl 的路径上也可以加上 `-w`，可以显示程序中出现的一些警告，尽管可能不影响程序的执行结果，方便我们修改我们的程序，例如：

```
#!/usr/bin/perl -w
```

`#!/usr/bin/perl` 需要参考自己的 perl 路径。

jakoh1年前 (2017-08-15)

反馈/建议

# Perl 基础语法

Perl借用了C、sed、awk、shell脚本以及很多其他编程语言的特性，语法与这些语言有些类似，也有自己的特点。

Perl 程序有声明与语句组成，程序自上而下执行，包含了循环，条件控制，每个语句以分号 (;) 结束。

Perl 语言没有严格的格式规范，你可以根据自己喜欢的风格来缩进。

## 第一个 perl 程序

### 交互式编程

你可以在命令行中使用 `-e` 选项来输入语句来执行代码，实例如下：

```
$ perl -e 'print "Hello World\n"'
```

输入以上命令，回车后，输出结果为：

```
Hello World
```

### 脚本式编程

我们将以下代码放到 `hello.pl` 文件中：

实例

```
#!/usr/bin/perl
# 输出 "Hello, World"
print "Hello, world\n";
```

代码中 `/usr/bin/perl` 是 perl 解释器的路径。在执行该脚本前要先确保文件有可执行权限，我们可以先将文件权限修改为 `0755`：

```
$ chmod 0755 hello.pl

$ ./hello.pl

Hello, world                # 输出结果
```

`print` 也可以使用括号来输出字符串，以下两个语句输出相同的结果：

```
print("Hello, world\n");

print "Hello, world\n";
```

## 脚本文件

`perl` 代码可以写在一个文本文件中，以 `.pl`、`.PL` 作为后缀。

文件名可以包含数字，符号和字母，但不能包含空格，可以使用下划线（`_`）来替代空格。

一个简单的 `Perl` 文件名：

```
run_oob.pl
```

## 注释

使用注释使你的程序易读，这是好的编程习惯。

`perl` 注释的方法为在语句的开头用字符`#`，如：

```
# 这一行是 perl 中的注释
```

`perl` 也支持多行注释，最常用的方法是使用 `POD(Plain Old Documentations)` 来进行多行注释。方法如下：

### 实例

```
#!/usr/bin/perl
# 这是一个单行注释
print "Hello, world\n";
=pod 注释
这是一个多行注释
这是一个多行注释
这是一个多行注释
这是一个多行注释
=cut
```

执行以上程序，输出结果为：

```
Hello, world
```

**注意：**

`=pod`、`=cut`只能在行首。

以`=`开头，以`=cut`结尾。

`=`后面要紧接一个字符，`=cut`后面可以不用。



## Perl 中的空白

Perl 解释器不会关心有多少个空白，以下程序也能正常运行：

### 实例

```
#!/usr/bin/perl
print "Hello, world\n";
```

执行以上程序，输出结果为：

```
Hello, world
```

但是如果空格和分行出现在字符串内，他会原样输出：

### 实例

```
#!/usr/bin/perl
# 会输出分行
print "Hello
world\n";
```

执行以上程序，输出结果为：

```
Hello

world
```

所有类型的空白如：空格，**tab**，空行等如果在引号外解释器会忽略它，如果在引号内会原样输出。

## 单引号和双引号

perl 输出字符串可以使用单引号和双引号，如下所示：

### 实例

```
#!/usr/bin/perl
print "Hello, world\n"; # 双引号
print 'Hello, world\n'; # 单引号
```

输出结果如下：

```
Hello, world

Hello, world\n
```

从结果中我们可以看出，双引号 `\n` 输出了换行，而单引号没有。

Perl 双引号和单引号的区别：双引号可以正常解析一些转义字符与变量，而单引号无法解析会原样输出。

### 实例

```
#!/usr/bin/perl
$a = 10;
print "a = $a\n";
print 'a = $a\n';
```

输出结果如下：

```
a = 10

a = $a\n
```

## Here 文档

Here文档又称作heredoc、hereis、here-字串或here-脚本，是一种在命令行shell（如sh、csh、ksh、bash、PowerShell和zsh）和程序语言（像Perl、PHP、Python和Ruby）里定义一个字串的方法。

使用概述：

- 1.必须后接分号，否则编译通不过。
- 2.END可以用任意其它字符代替，只需保证结束标识与开始标识一致。
- 3.结束标识必须顶格独自占一行(即必须从行首开始，前后不能衔接任何空白和字符)。
- 4.开始标识可以不带引号或带单双引号，不带引号与带双引号效果一致，解释内嵌的变量和转义符号，带单引号则不解释内嵌的变量和转义符号。
- 5.当内容需要内嵌引号（单引号或双引号）时，不需要加转义符，本身对单双引号转义，此处相当与q和qq的用法。

## 实例

```
#!/usr/bin/perl
$a = 10;
$var = <<"EOF";
这是一个 Here 文档实例，使用双引号。
可以在这输如字符串和变量。
例如: a = $a
EOF
print "$var\n";
$var = <<'EOF';
这是一个 Here 文档实例，使用单引号。
例如: a = $a
EOF
print "$var\n";
```

执行以上程序输出结果为：

这是一个 Here 文档实例，使用双引号。

可以在这输如字符串和变量。

例如: a = 10

这是一个 Here 文档实例，使用单引号。

例如: a = \$a

## 转义字符

如果我们需要输出一个特殊的字符，可以使用反斜线（\）来转义，例如输出美元符号(\$):

## 实例

```
#!/usr/bin/perl
$result = "菜鸟教程 \"runoob\"";
print "$result\n";
print "\\$result\n";
```

执行以上程序输出结果为：

```
菜鸟教程 "runoob"$result
```

## Perl 标识符

Perl 标识符是用户编程时使用的名字，在程序中使用的变量名，常量名，函数名，语句块名等统称为标识符。

标识符组成单元：英文字母（a~z，A~Z），数字（0~9）和下划线（\_）。

标识符由英文字母或下划线开头。

标识符区分大小写，\$runoob 与 \$Runoob 表示两个不同变量。

❏

2 篇笔记

#2

❏

perl 中的单引号和双引号：

（1）双中有双，单中有单都需要 \ 转义。

（2）双中有单或单中有双均不需要转义。

（3）单引号直接了当，引号内是什么就显示什么，双引号则需要考虑转义或变量替换等。

Whale fall9个月前 (12-21)

#1

❏

EOF在这里通俗讲就是一个标记，他用来标记一段文字（一般都是多行的，省得编码麻烦，用"<<"加上一个标记就可以把一大段代码存入到一个变量中去了）\$a<<"EOF" 的意思就是说：下一行开始，直到遇见“EOF”为止，所有的字符都按照指定的格式存入变量a中。你可以用EEE，MAMA等等其他的名字都可以，就是一个标记而已。他的作用就是简化输入。

kmato3周前 (09-11)

反馈/建议



## Perl 数据类型

Perl 是一种弱类型语言，所以变量不需要指定类型，Perl 解释器会根据上下文自动选择匹配类型。

Perl 有三个基本的数据类型：标量、数组、哈希。以下是这三种数据类型的说明：

序号	类型和描述
----	-------

1	<p><b>标量</b></p> <p>标量是Perl语言中最简单的一种数据类型。这种数据类型的变量可以是数字，字符串，浮点数，不作严格的区分。在使用时在变量的名字前面加上一个"\$",表示是标量。例如：</p> <pre>\$myfirst=123;      #数字123  \$mysecond="123";   #字符串123</pre>
2	<p><b>数组</b></p> <p>数组变量以字符"@"开头，索引从0开始，如：@arr=(1,2,3)</p> <pre>@arr=(1,2,3)</pre>
3	<p><b>哈希</b></p> <p>哈希是一个无序的 key/value 对集合。可以使用键作为下标获取值。哈希变量以字符"%"开头。</p> <pre>%h=('a'=&gt;1,'b'=&gt;2);</pre>

## 数字字面量

### 一、整型

PERL实际上把整数存在你的计算机中的浮点寄存器中，所以实际上被当作浮点数看待。

在多数计算机中，浮点寄存器可以存贮约16位数字，长于此的被丢弃。整数实为浮点数的特例。

整型变量及运算：

```
$x = 12345;

if (1217 + 116 == 1333) {

    # 执行代码语句块

}
```

8进制和16进制数：8进制以0开始，16进制以0x开始。例如：

```
$var1 = 047;      # 等于十进制的39

$var2 = 0x1f;     # 等于十进制的31
```

### 二、浮点数

浮点数数据如：11.4、-0.3、.3、3、54.1e+02、5.41e03。

浮点寄存器通常不能精确地存贮浮点数，从而产生误差，在运算和比较中要特别注意。指数的范围通常为-309到+308。

## 实例

```
#!/usr/bin/perl
$value = 9.01e+21 + 0.01 - 9.01e+21;
print ("第一个值为: ", $value, "\n");
$value = 9.01e+21 - 9.01e+21 + 0.01;
print ("第二个值为:", $value, "\n");
```

执行以上程序，输出结果为：

```
第一个值为: 0

第二个值为:0.01
```

## 三、字符串

Perl中的字符串使用一个标量来表示，定义方式和c很像，但是在Perl里面字符串不是用0来表示结束的。

Perl双引号和单引号的区别: 双引号可以正常解析一些转义字符与变量，而单引号无法解析会原样输出。

但是用单引号定义可以使用多行文本，如下所示：

```
#!/usr/bin/perl

$var='这是一个使用

多行字符串文本

的例子';

print($var);
```

执行以上程序，输出结果为：

```
这是一个使用

多行字符串文本

的例子
```

Perl 语言中常用的一些转义字符如下表所示：

转义字符	含义
\\	反斜线

\'	单引号
\"	双引号
\a	系统响铃
\b	退格
\f	换页符
\n	换行
\r	回车
\t	水平制表符
\v	垂直制表符
\0nn	创建八进制格式的数字
\xnn	创建十六进制格式的数字
\cX	控制字符， <b>x</b> 可以是任何字符
\u	强制下一个字符为大写
\l	强制下一个字符为小写
\U	强制将所有字符转换为大写
\L	强制将所有的字符转换为小写
\Q	将到\E为止的非单词（ <b>non-word</b> ）字符加上反斜线
\E	结束\L、\U、\Q

## 实例

接下来让我们来具体看看单引号和双引号及转义字符的使用：

### 实例

```
#!/usr/bin/perl
# 换行 \n 位于双引号内，有效
$str = "菜鸟教程 \nwww.runoob.com";
print "$str\n";
# 换行 \n 位于单引号内，无效
$str = '菜鸟教程 \nwww.runoob.com';
print "$str\n";
# 只有 R 会转换为大写
$str = "\urunoob";
print "$str\n";
# 所有的字母都会转换为大写
$str = "\Urunoob";
print "$str\n";
# 指定部分会转换为大写
$str = "Welcome to \Urunoob\E.com!";
print "$str\n";
# 将到\E为止的非单词（non-word）字符加上反斜线
$str = "\QWelcome to runoob's family";
print "$str\n";
```

以上实例执行输出结果为：

菜鸟教程  
www.runoob.com  
菜鸟教程 \nwww.runoob.com  
Runoob  
RUNOOB  
Welcome to RUNOOB.com!  
Welcome\ to\ runoob\'s\ family

❏ Perl 基础语法

Perl 变量❏

❏ 点我分享笔记

反馈/建议

Copyright © 2013-2018 菜鸟教程 [runoob.com](#) All Rights Reserved. 备案号：闽ICP备15012807号-1



[首页](#) [HTML](#) [CSS](#) [JS](#) [本地书签](#)

❏ Perl 数据类型

Perl 标量❏

# Perl 变量

变量是存储在内存中的数据，创建一个变量即会在内存上开辟一个空间。

解释器会根据变量的类型来决定其在内存中的存储空间，因此你可以为变量分配不同的数据类型，如整型、浮点型、字符串等。

上一章节中我们已经为大家介绍了Perl的三个基本的数据类型：标量、数组、哈希。

标量 **\$** 开始，如**\$a \$b** 是两个标量。

数组 **@** 开始，如 **@a @b** 是两个数组。

哈希 **%** 开始， **%a %b** 是两个哈希。

Perl 为每个变量类型设置了独立的命令空间，所以不同类型的变量可以使用相同的名称，你不用担心会发生冲突。例如 **\$foo** 和 **@foo** 是两个不同的变量。

## 创建变量

变量不需要显式声明类型，在变量赋值后，解释器会自动分配匹配的类型空间。

变量使用等号(=)来赋值。

我们可以在程序中使用 **use strict** 语句让所有变量需要强制声明类型。

等号左边为变量，右边为值，实例如下：

```
$age = 25;           # 整型

$name = "runoob";    # 字符串

$salary = 1445.50;    # 浮点数
```

以上代码中 25, "runoob" 和 1445.50 分别赋值给 \$age, \$name 和 \$salary 变量。

接下来我们会看到数组和哈希的使用。

## 标量变量

标量是一个单一的数据单元。数据可以是整数，浮点数，字符，字符串，段落等。简单的说它可以是任何东西。以下是标量的简单应用：

### 实例

```
#!/usr/bin/perl
$age = 25; # 整型
$name = "runoob"; # 字符串
$salary = 1445.50; # 浮点数
print "Age = $age\n";
print "Name = $name\n";
print "Salary = $salary\n";
```

以上程序执行输出结果为：

```
Age = 25

Name = runoob

Salary = 1445.5
```

## 数组变量

数组是用于存储一个有序的标量值的变量。

数组 @ 开始。

要访问数组的变量，可以使用美元符号(\$) + 变量名，并指定下标来访问，实例如下所示：

### 实例

```
#!/usr/bin/perl
@ages = (25, 30, 40);
@names = ("google", "runoob", "taobao");
print "\$ages[0] = $ages[0]\n";
print "\$ages[1] = $ages[1]\n";
print "\$ages[2] = $ages[2]\n";
print "\$names[0] = $names[0]\n";
print "\$names[1] = $names[1]\n";
print "\$names[2] = $names[2]\n";
```

以上程序执行输出结果为：

```
$ages[0] = 25

$ages[1] = 30

$ages[2] = 40

$names[0] = google

$names[1] = runoob

$names[2] = taobao
```

程序中我们在 \$ 标记前使用了转义字符 (\)，这样才能输出字符 \$。

## 哈希变量



哈希是一个 **key/value** 对的集合。

哈希 % 开始。

如果要访问哈希值，可以使用 **\$ + {key}** 格式来访问：

实例

```
#!/usr/bin/perl
%data = ('google', 45, 'runoob', 30, 'taobao', 40);
print "\$data{'google'} = $data{'google'}\n";
print "\$data{'runoob'} = $data{'runoob'}\n";
print "\$data{'taobao'} = $data{'taobao'}\n";
```

以上程序执行输出结果为：

```
$data{'google'} = 45

$data{'runoob'} = 30

$data{'taobao'} = 40
```

## 变量上下文

所谓上下文：指的是表达式所在的位置。

上下文是由等号左边的变量类型决定的，等号左边是标量，则是标量上下文，等号左边是列表，则是列表上下文。

Perl 解释器会根据上下文来决定变量的类型。实例如下：

实例

```
#!/usr/bin/perl
@names = ('google', 'runoob', 'taobao');
@copy = @names; # 复制数组
$size = @names; # 数组赋值给标量，返回数组元素个数
print "名字为 : @copy\n";
print "名字数为 : $size\n";
```

以上程序执行输出结果为：

```
名字为 : google runoob taobao

名字数为 : 3
```

代码中 **@names** 是一个数组，它应用在了两个不同的上下文中。第一个将其复制给另外一个数组，所以它输出了数组的所有元素。第二个我们将数组赋值给一个标量，它返回了数组的元素个数。

以下列出了多种不同的上下文：

序号	上下文及描述
1	<b>标量 -</b> 赋值给一个标量变量，在标量上下文的右侧计算
2	<b>列表 -</b> 赋值给一个数组或哈希，在列表上下文的右侧计算。
3	<b>布尔 -</b> 布尔上下文是一个简单的表达式计算，查看是否为 <b>true</b> 或 <b>false</b> 。
4	<b>Void -</b> 这种上下文不需要关系返回什么值，一般不需要返回值。

5	插值 - 这种上下文只发生在引号内。
---	-----------------------

[Perl 数据类型](#)

Perl 标量 [□](#)

[点我分享笔记](#)

[反馈/建议](#)

Copyright © 2013-2018 菜鸟教程 [runoob.com](#) All Rights Reserved. 备案号：闽ICP备15012807号-1



[首页](#) [HTML](#) [CSS](#) [JS](#) [本地书签](#)

[Perl 变量](#)

Perl 数组 [□](#)

## Perl 标量

标量是一个简单的数据单元。

标量可以是一个整数，浮点数，字符，字符串，段落或者一个完整的网页。

以下实例演示了标量的简单应用：

### 实例

```
#!/usr/bin/perl
$age = 20; # 整数赋值
$name = "Runoob"; # 字符串
$salary = 130.50; # 浮点数
print "Age = $age\n";
print "Name = $name\n";
print "Salary = $salary\n";
```

执行以上程序，输出结果为：

```
Age = 20

Name = Runoob

Salary = 130.5
```

## 数字标量

标量通常是一个数字或字符串，以下实例演示了不同类型的数字标量的使用：

### 实例

```
#!/usr/bin/perl
$integer = 200;
$negative = -300;
$floating = 200.340;
$bigfloat = -1.2E-23;
# 八进制 377 ，十进制为 255
```

```
$octal = 0377;
# 十六进制 FF, 十进制为 255
$hexa = 0xff;
print "integer = $integer\n";
print "negative = $negative\n";
print "floating = $floating\n";
print "bigfloat = $bigfloat\n";
print "octal = $octal\n";
print "hexa = $hexa\n";
```

执行以上程序，输出结果为：

```
integer = 200

negative = -300

floating = 200.34

bigfloat = -1.2e-23

octal = 255

hexa = 255
```

## 字符串标量

以下实例演示了不同类型的字符串标量的使用，注意单引号和双引号的使用区别：

### 实例

```
#!/usr/bin/perl
$var = "字符串标量 - 菜鸟教程!";
$quote = '我在单引号内 - $var';
$double = "我在双引号内 - $var";
$escape = "转义字符使用 -\tHello, World!";
print "var = $var\n";
print "quote = $quote\n";
print "double = $double\n";
print "escape = $escape\n";
```

执行以上程序，输出结果为：

```
var = 字符串标量 - 菜鸟教程!

quote = 我在单引号内 - $var

double = 我在双引号内 - 字符串标量 - 菜鸟教程!

escape = 转义字符使用 -      Hello, World!
```

## 标量运算

以下实例演示了标量的简单运算：

### 实例

```
#!/usr/bin/perl
$str = "hello" . "world"; # 字符串连接
$num = 5 + 10; # 两数相加
$mul = 4 * 5; # 两数相乘
$mix = $str . $num; # 连接字符串和数字
print "str = $str\n";
```

```
print "num = $num\n";
print "mix = $mix\n";
```

执行以上程序，输出结果为：

```
str = helloworld

num = 15

mix = helloworld15
```

## 多行字符串

我们可以使用单引号来输出多行字符串，如下所示：

### 实例

```
#!/usr/bin/perl
$string = '
菜鸟教程
— 学的不仅是技术，更是梦想！
';
print "$string\n";
```

执行以上程序，输出结果为：

```
菜鸟教程

— 学的不仅是技术，更是梦想！
```

你也可以使用 "here" document 的语法格式来输出多行：

### 实例

```
#!/usr/bin/perl
print <<EOF;
菜鸟教程
— 学的不仅是技术，更是梦想！
EOF
```

执行以上程序，输出结果为：

```
菜鸟教程

— 学的不仅是技术，更是梦想！
```

## 特殊字符

以下我们将演示 Perl 中特殊字符的应用，如 `__FILE__`、`__LINE__` 和 `__PACKAGE__` 分别表示当前执行脚本的文件名，行号，包名。

注意：`__` 是两条下划线，`__FILE__` 前后各两条下划线。

这些特殊字符是单独的标记，不能写在字符串中，例如：

### 实例

```
#!/usr/bin/perl
print "文件名 " . __FILE__ . "\n";
print "行号 " . __LINE__ . "\n";
print "包名 " . __PACKAGE__ . "\n";
```

```
# 无法解析
print "__FILE__ __LINE__ __PACKAGE__\n";
```

执行以上程序，输出结果为：

文件名 test.pl

行号 4

包名 main

\_\_FILE\_\_ \_\_LINE\_\_ \_\_PACKAGE\_\_

## v 字符串

一个以 v 开头,后面跟着一个或多个用句点分隔的整数,会被当作一个字符串文本。

当你想为每个字符 直接声明其数值值时,v-字符串提供了一种更清晰的构造这类字符串的方法，而不像 "\x{1}\x{14}\x{12c}\x{fa0}" 这种不易于理解，我们可以看下面的实例：

### 实例

```
#!/usr/bin/perl
$smile = v9786;
$foo = v102.111.111;
$martin = v77.97.114.116.105.110;
print "smile = $smile\n";
print "foo = $foo\n";
print "martin = $martin\n";
```

执行以上程序，输出结果为：

Wide character in print at test.pl line 7.

smile = &#x263a;

foo = foo

martin = Martin

☐ Perl 变量

Perl 数组 ☐

☐ 点我分享笔记

反馈/建议

## Perl 数组

Perl 数组一个是存储标量值的列表变量，变量可以是不同类型。

数组变量以 **@** 开头。访问数组元素使用 **\$ + 变量名称 + [索引值]** 格式来读取，实例如下：

### 实例

```
#!/usr/bin/perl
@hits = (25, 30, 40);
@names = ("google", "runoob", "taobao");
print "\$hits[0] = $hits[0]\n";
print "\$hits[1] = $hits[1]\n";
print "\$hits[2] = $hits[2]\n";
print "\$names[0] = $names[0]\n";
print "\$names[1] = $names[1]\n";
print "\$names[2] = $names[2]\n";
```

程序中 **\$** 符号使用了 **\** 来转义，让他原样输出。

执行以上程序，输出结果为：

```
$hits[0] = 25

$hits[1] = 30

$hits[2] = 40

$names[0] = google

$names[1] = runoob

$names[2] = taobao
```

## 创建数组

数组变量以 **@** 符号开始，元素放在括号内，也可以以 **qw** 开始定义数组。

```
@array = (1, 2, 'Hello');

@array = qw/这是一个数组/;
```

第二个数组使用 **qw//** 运算符，它返回字符串列表，数组元素以空格分隔。当然也可以使用多行来定义数组：

```
@days = qw/google

taobao

...

runoob/;
```

你也可以按索引来给数组赋值，如下所示：

```
$array[0] = 'Monday';
```

```
...
```

```
$array[6] = 'Sunday';
```

## 访问数组元素

访问数组元素使用 **\$+ 变量名称 + [索引值]** 格式来读取，实例如下：

### 实例

```
@sites = qw/google taobao runoob/;
print "$sites[0]\n";
print "$sites[1]\n";
print "$sites[2]\n";
print "$sites[-1]\n"; # 负数，反向读取
```

执行以上程序，输出结果为：

```
google
```

```
taobao
```

```
runoob
```

```
runoob
```

数组索引值从 **0** 开始，即 **0** 为第一个元素，**1** 为第二个元素，以此类推。

负数从反向开始读取，**-1** 为第一个元素，**-2** 为第二个元素

## 数组序列号

Perl 提供了可以按序列输出的数组形式，格式为 **起始值 + .. + 结束值**，实例如下：

### 实例

```
#!/usr/bin/perl
@var_10 = (1..10);
@var_20 = (10..20);
@var_abc = (a..z);
print "@var_10\n"; # 输出 1 到 10
print "@var_20\n"; # 输出 10 到 20
print "@var_abc\n"; # 输出 a 到 z
```

执行以上程序，输出结果为：

```
1 2 3 4 5 6 7 8 9 10
```

```
10 11 12 13 14 15 16 17 18 19 20
```

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
```

## 数组大小

数组大小由数组中的标量上下文决定。：

```
@array = (1,2,3);
```

```
print "数组大小: ",标量 @array,"\n";
```

数组长度返回的是数组物理大小，而不是元素的个数，我们可以看以下实例：

### 实例

```
#!/uer/bin/perl
@array = (1,2,3);
$array[50] = 4;
$size = @array;
$max_index = $#array;
print "数组大小: $size\n";
print "最大索引: $max_index\n";
```

执行以上程序，输出结果为：

```
数组大小: 51

最大索引: 50
```

从输出的结果可以看出，数组元素只有四个，但是数组大小为 51。

### 添加和删除数组元素

Perl 提供了一些有用的函数来添加和删除数组元素。

如果你之前没有编程经验，可能会问什么是函数，其实我们之前使用的 **print** 即是一个输出函数。

下表列出了数组中常用的操作函数：

序号	类型和描述
1	<b>push @ARRAY, LIST</b> 将列表的值放到数组的末尾
2	<b>pop @ARRAY</b> 弹出数组最后一个值，并返回它
3	<b>shift @ARRAY</b> 弹出数组第一个值，并返回它。数组的索引值也依次减一。
4	<b>unshift @ARRAY, LIST</b> 将列表放在数组前面，并返回新数组的元素个数。

### 实例

```
#!/usr/bin/perl
# 创建一个简单数组
@sites = ("google","runoob","taobao");
print "1. \@sites = @sites\n";
# 在数组结尾添加一个元素
push(@sites, "baidu");
print "2. \@sites = @sites\n";
# 在数组开头添加一个元素
unshift(@sites, "weibo");
print "3. \@sites = @sites\n";
# 删除数组末尾的元素
pop(@sites);
print "4. \@sites = @sites\n";
# 移除数组开头的元素
shift(@sites);
print "5. \@sites = @sites\n";
```

执行以上程序，输出结果为：



1. @sites = google runoob taobao
2. @sites = google runoob taobao baidu
3. @sites = weibo google runoob taobao baidu
4. @sites = weibo google runoob taobao
5. @sites = google runoob taobao

## 切割数组

我们可以切割一个数组，并返回切割后的新数组：

### 实例

```
#!/usr/bin/perl
@sites = qw/google taobao runoob weibo qq facebook 网易/;
@sites2 = @sites[3,4,5];
print "@sites2\n";
```

执行以上程序，输出结果为：

```
weibo qq facebook
```

数组索引需要指定有效的索引值，可以是正数后负数，每个索引值使用逗号隔开。

如果是连续的索引，可以使用 .. 来表示指定范围：

### 实例

```
#!/usr/bin/perl
@sites = qw/google taobao runoob weibo qq facebook 网易/;
@sites2 = @sites[3..5];
print "@sites2\n";
```

执行以上程序，输出结果为：

```
weibo qq facebook
```

## 替换数组元素

Perl 中数组元素替换使用 splice() 函数，语法格式如下：

```
splice @ARRAY, OFFSET [ , LENGTH [ , LIST ] ]
```

参数说明：

@ARRAY: 要替换的数组。

OFFSET: 起始位置。

LENGTH: 替换的元素个数。

LIST: 替换元素列表。

以下实例从第6个元素开始替换数组中的5个元素：

### 实例

```
#!/usr/bin/perl
@nums = (1..20);
print "替换前 - @nums\n";
splice(@nums, 5, 5, 21..25);
print "替换后 - @nums\n";
```

执行以上程序，输出结果为：

```
替换前 - 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
```

```
替换后 - 1 2 3 4 5 21 22 23 24 25 11 12 13 14 15 16 17 18 19 20
```

## 将字符串转换为数组

Perl 中将字符串转换为数组使用 `split()` 函数，语法格式如下：

```
split [ PATTERN [ , EXPR [ , LIMIT ] ] ]
```

参数说明：

**PATTERN:** 分隔符，默认为空格。

**EXPR:** 指定字符串数。

**LIMIT:** 如果指定该参数，则返回该数组的元素个数。

## 实例

```
#!/usr/bin/perl
# 定义字符串
$var_test = "runoob";
$var_string = "www-runoob-com";
$var_names = "google,taobao,runoob,weibo";
# 字符串转为数组
@test = split('', $var_test);
@string = split('-', $var_string);
@names = split(',', $var_names);
print "$test[3]\n"; # 输出 o
print "$string[2]\n"; # 输出 com
print "$names[3]\n"; # 输出 weibo
```

执行以上程序，输出结果为：

```
o

com

weibo
```

## 将数组转换为字符串

Perl 中将数组转换为字符串使用 `join()` 函数，语法格式如下：

```
join EXPR, LIST
```

参数说明：

**EXPR:** 连接符。

**LIST:** 列表或数组。

## 实例

```
#!/usr/bin/perl
# 定义字符串
$var_string = "www-runooob-com";
$var_names = "google,taobao,runoob,weibo";
# 字符串转为数组
@string = split('-', $var_string);
@names = split(',', $var_names);
# 数组转为字符串
$string1 = join( '-', @string );
$string2 = join( ',', @names );
print "$string1\n";
print "$string2\n";
```

执行以上程序，输出结果为：

```
www-runooob-com
```

```
google,taobao,runoob,weibo
```

## 数组排序

Perl 中数组排序使用 `sort()` 函数，语法格式如下：

```
sort [ SUBROUTINE ] LIST
```

参数说明：

SUBROUTINE：指定规则。

LIST：列表或数组。

## 实例

```
#!/usr/bin/perl
# 定义数组
@sites = qw(google taobao runoob facebook);
print "排序前: @sites\n";
# 对数组进行排序
@sites = sort(@sites);
print "排序后: @sites\n";
```

执行以上程序，输出结果为：

```
排序前: google taobao runoob facebook
```

```
排序后: facebook google runoob taobao
```

注意：数组排序是根据 ASCII 数字值来排序。所以我们在对数组进行排序时最好先将每个元素转换为小写后再排序。

## 特殊变量：\$[

特殊变量 `$[` 表示数组的第一索引值，一般都为 0，如果我们将 `$[` 设置为 1，则数组的第一个索引值即为 1，第二个为 2，以此类推。实例如下：

## 实例

```
#!/usr/bin/perl
```

```
# 定义数组
@sites = qw(google taobao runoob facebook);
print "网站: @sites\n";
# 设置数组的第一个索引为 1
$[ = 1;
print "\@sites[1]: $sites[1]\n";
print "\@sites[2]: $sites[2]\n";
```

执行以上程序，输出结果为：

```
网站: google taobao runoob facebook

@sites[1]: google

@sites[2]: taobao
```

一般情况我们不建议使用特殊变量 `$[`，在新版 *Perl* 中，该变量已废弃。

## 合并数组

数组的元素是以逗号来分割，我们也可以使用逗号来合并数组，如下所示：

### 实例

```
#!/usr/bin/perl
@numbers = (1,3,(4,5,6));
print "numbers = @numbers\n";
```

执行以上程序，输出结果为：

```
numbers = 1 3 4 5 6
```

也可以在数组中嵌入多个数组，并合并到主数组中：

### 实例

```
#!/usr/bin/perl
@odd = (1,3,5);
@even = (2, 4, 6);
@numbers = (@odd, @even);
print "numbers = @numbers\n";
```

执行以上程序，输出结果为：

```
numbers = 1 3 5 2 4 6
```

## 从列表中选择元素

一个列表可以当作一个数组使用，在列表后指定索引值可以读取指定的元素，如下所示：

### 实例

```
#!/usr/bin/perl
$var = (5,4,3,2,1)[4];
print "var 的值为 = $var\n"
```

执行以上程序，输出结果为：

```
var  的值为 = 1
```

同样我们可以在数组中使用 .. 来读取指定范围的元素：

### 实例

```
#!/usr/bin/perl
@list = (5,4,3,2,1)[1..3];
print "list  的值 = @list\n";
```

执行以上程序，输出结果为：

```
list  的值 = 4 3 2
```

[Perl 标量](#)

[Perl 哈希](#)



2 篇笔记  
#2

[写笔记](#)



#### splice()函数小结

1. 替换数组元素。

```
splice(@ARRAY,OFFSET,LENGTH,LIST)
```

@ARRAY: 要替换的数组。

OFFSET: 起始位置。

LENGTH: 替换的元素个数。

LIST: 替换元素列表。

示例：

```
@a = (1..5);

@b = (a..h);

print "原始  @a\n";

splice(@a , 2 , 2 , @b);

print "插入  @a\n";
```

结果：

```
原始  1 2 3 4 5
```

```
插入  1 2 a b c d e f g h 5
```

2. 删除。

```
splice(@ARRAY,OFFSET,LENGTH)
```

@ARRAY: 要删除的数组。

OFFSET: 起始位置。

LENGTH: 删除的元素个数。

示例：

```
@a = (1..20);

print "原始  @a\n";

splice(@a , 2 , 6);

print "删除  @a\n";
```

结果:

```
原始  1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

删除  1 2 9 10 11 12 13 14 15 16 17 18 19 20
```

3.删除到末尾。

```
splice(@ARRAY,OFFSET)
```

@ARRAY: 要删除到结尾的数组。  
OFFSET: 起始位置。  
示例:

```
@a = (1..20);

print "原始  @a\n";

splice(@a , 2);

print "删除  @a\n";
```

结果:

```
原始  1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

删除  1 2
```

zhi\_00712个月前 (10-15)

#1



关于数组排序的总结  
sort 函数遍历原始数组的每两个元素，把左边的值放入变量 \$a，右边的值放入变量 \$b。然后调用比较函数。如果 \$a 的内容应该在左边的话， 比较函数会返回 1；如果 \$b 应该在左边的话，返回 -1，两者一样的话，返回 0。  
依据数值大小排序:

```
my @line=qw /1 2 34 9 12 4 3 8 67 53 1 42 1 17 3 0/;

print "sorted numbers: ";

foreach my $item(sort {$a <=> $b} @line){

    print "$item ";

}
```

代码中使用 <=> 比较函数，表示依据数值字面大小排序，此为升序，若要降序排序，仅需调换标量 \$a 和 \$b 位置即可。  
依据第一个字符 ASCII 值排序:将上述代码中的比较函数改为 cmp 函数即可，这也是 perl 默认的比较函数。

tangxuan7个月前 (03-16)

[首页](#) [HTML](#) [CSS](#) [JS](#) [本地书签](#)[Perl 数组](#)[Perl 条件语句](#)

## Perl 哈希

哈希是 **key/value** 对的集合。

Perl中哈希变量以百分号 (%) 标记开始。

访问哈希元素格式: **\${key}**。

以下是一个简单的哈希实例:

### 实例

```
#!/usr/bin/perl
%data = ('google', 'google.com', 'runoob', 'runoob.com', 'taobao', 'taobao.com');
print "\$data{'google'} = $data{'google'}\n";
print "\$data{'runoob'} = $data{'runoob'}\n";
print "\$data{'taobao'} = $data{'taobao'}\n";
```

执行以上程序, 输出结果为:

```
$data{'google'} = google.com
$data{'runoob'} = runoob.com
$data{'taobao'} = taobao.com
```

## 创建哈希

创建哈希可以通过以下两种方式:

### 一、为每个 **key** 设置 **value**

```
$data{'google'} = 'google.com';

$data{'runoob'} = 'runoob.com';

$data{'taobao'} = 'taobao.com';
```

### 二、通过列表设置

列表中第一个元素为 **key**, 第二个为 **value**。

```
%data = ('google', 'google.com', 'runoob', 'runoob.com', 'taobao', 'taobao.com');
```

也可以使用 => 符号来设置 **key/value**:

```
%data = ('google'=>'google.com', 'runoob'=>'runoob.com', 'taobao'=>'taobao.com');
```

以下实例是上面实例的变种，使用 - 来代替引号：

```
%data = (-google=>'google.com', -runoob=>'runoob.com', -taobao=>'taobao.com');
```

使用这种方式 **key** 不能出现空格，读取元素方式为：

```
$val = $data{-google}
```

```
$val = $data{-runoob}
```

## 访问哈希元素

访问哈希元素格式：**\${key}**，实例如下：

### 实例

```
#!/usr/bin/perl
%data = ('google'=>'google.com', 'runoob'=>'runoob.com', 'taobao'=>'taobao.com');
print "\$data{'google'} = $data{'google'}\n";
print "\$data{'runoob'} = $data{'runoob'}\n";
print "\$data{'taobao'} = $data{'taobao'}\n";
```

执行以上程序，输出结果为：

```
$data{'google'} = google.com
$data{'runoob'} = runoob.com
$data{'taobao'} = taobao.com
```

## 读取哈希值

你可以像数组一样从哈希中提取值。

哈希值提取到数组语法格式：**@{key1,key2}**。

### 实例

```
#!/usr/bin/perl
%data = (-taobao => 45, -google => 30, -runoob => 40);
@array = @data{-taobao, -runoob};
print "Array : @array\n";
```

执行以上程序，输出结果为：

```
Array : 45 40
```

## 读取哈希的 **key** 和 **value**

### 读取所有**key**

我们可以使用 **keys** 函数读取哈希所有的键，语法格式如下：

```
keys %HASH
```

该函数返回所有哈希的所有 **key** 的数组。

### 实例

```
#!/usr/bin/perl
```



```
%data = ('google'=>'google.com', 'runoob'=>'runoob.com', 'taobao'=>'taobao.com');
@names = keys %data;
print "$names[0]\n";
print "$names[1]\n";
print "$names[2]\n";
```

执行以上程序，输出结果为：

```
taobao

google

runoob
```

类似的我们可以使用 **values** 函数来读取哈希所有的值,语法格式如下：

```
values %HASH
```

该函数返回所有哈希的所有 **value** 的数组。

## 实例

```
#!/usr/bin/perl
%data = ('google'=>'google.com', 'runoob'=>'runoob.com', 'taobao'=>'taobao.com');
@urls = values %data;
print "$urls[0]\n";
print "$urls[1]\n";
print "$urls[2]\n";
```

执行以上程序，输出结果为：

```
taobao.com

runoob.com

google.com
```

## 检测元素是否存在

如果你在哈希中读取不存在的 **key/value** 对，会返回 **undefined** 值，且在执行时会有警告提醒。

为了避免这种情况，我们可以使用 **exists** 函数来判断**key**是否存在，存在的时候读取：

## 实例

```
#!/usr/bin/perl
%data = ('google'=>'google.com', 'runoob'=>'runoob.com', 'taobao'=>'taobao.com');
if( exists($data{'facebook'}) ){
print "facebook 的网址为 $data{'facebook'} \n";
}
else
{
print "facebook 键不存在\n";
}
```

执行以上程序，输出结果为：

```
facebook 键不存在
```

以上代码中我们使用了 **IF...ELSE** 语句，在后面的章节我们会具体介绍。

## 获取哈希大小

哈希大小为元素的个数，我们可以通过先获取 **key** 或 **value** 的所有元素数组，再计算数组元素多少来获取哈希的大小，实例如下：

### 实例

```
#!/usr/bin/perl
%data = ('google'=>'google.com', 'runoob'=>'runoob.com', 'taobao'=>'taobao.com');
@keys = keys %data;
$size = @keys;
print "1 - 哈希大小: $size\n";
@values = values %data;
$size = @values;
print "2 - 哈希大小: $size\n";
```

执行以上程序，输出结果为：

```
1 - 哈希大小: 3
```

```
2 - 哈希大小: 3
```

## 哈希中添加或删除元素

添加 **key/value** 对可以通过简单的赋值来完成。但是删除哈希元素你需要使用 **delete** 函数：

### 实例

```
#!/usr/bin/perl
%data = ('google'=>'google.com', 'runoob'=>'runoob.com', 'taobao'=>'taobao.com');
@keys = keys %data;
$size = @keys;
print "1 - 哈希大小: $size\n";
# 添加元素
$data{'facebook'} = 'facebook.com';
@keys = keys %data;
$size = @keys;
print "2 - 哈希大小: $size\n";
# 删除哈希中的元素
delete $data{'taobao'};
@keys = keys %data;
$size = @keys;
print "3 - 哈希大小: $size\n";
```

执行以上程序，输出结果为：

```
1 - 哈希大小: 3
```

```
2 - 哈希大小: 4
```

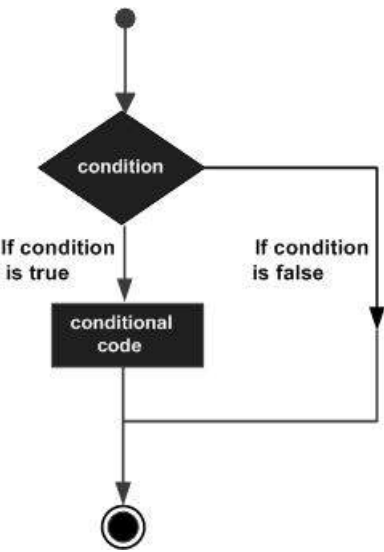
```
3 - 哈希大小: 3
```



# Perl 条件语句

Perl 条件语句是通过一条或多条语句的执行结果（**True**或者**False**）来决定执行的代码块。

可以通过下图来简单了解条件语句的执行过程：



注意，数字 0，字符串 '0'、""，空 list ()，和 undef 为 **false**，其他值均为 **true**。true 前面使用 **!** 或 **not** 则返回 **false**。

Perl 提供了下拉的条件语句：

语句	描述
<a href="#">if 语句</a>	一个 <b>if</b> 语句 由一个布尔表达式后跟一个或多个语句组成。
<a href="#">if...else 语句</a>	一个 <b>if</b> 语句 后可跟一个可选的 <b>else</b> 语句， <b>else</b> 语句在布尔表达式为假时执行。
<a href="#">if...elsif...else 语句</a>	您可以在一个 <b>if</b> 语句后可跟一个可选的 <b>elsif</b> 语句，然后再跟另一个 <b>else</b> 语句。
<a href="#">unless 语句</a>	一个 <b>unless</b> 语句 由一个布尔表达式后跟一个或多个语句组成。
<a href="#">unless...else 语句。</a>	一个 <b>unless</b> 语句 后可跟一个可选的 <b>else</b> 语句。
<a href="#">unless...elsif..else statement</a>	一个 <b>unless</b> 语句 后可跟一个可选的 <b>elsif</b> 语句，然后再跟另一个 <b>else</b> 语句。
<a href="#">switch 语句</a>	在最新版本的 Perl 中，我们可以使用 <b>switch</b> 语句。它根据不同的值执行对应的代码块。

## 三元运算符？：

我们可以使用 **条件运算 ？：**来简化 **if...else** 语句的操作。通常格式为：

```
Exp1 ? Exp2 : Exp3;
```

如果 **Exp1** 表达式为 **true**，则返回 **Exp2** 表达式计算结果，否则返回 **Exp3**。

实例如下所示：

## 实例

```
#!/usr/local/bin/perl
$name = "菜鸟教程";
$favorite = 10; # 喜欢数
$status = ($favorite > 60)? "热门网站" : "不是热门网站";
print "$name - $status\n";
```

执行以上程序，输出结果为：

```
菜鸟教程 - 不是热门网站
```

[Perl 哈希](#)

[Perl IF 语句](#)

[点我分享笔记](#)

[反馈/建议](#)

Copyright © 2013-2018 菜鸟教程 [runoob.com](#) All Rights Reserved. 备案号：闽ICP备15012807号-1



[首页](#) [HTML](#) [CSS](#) [JS](#) [本地书签](#)

[Perl switch 语句](#)

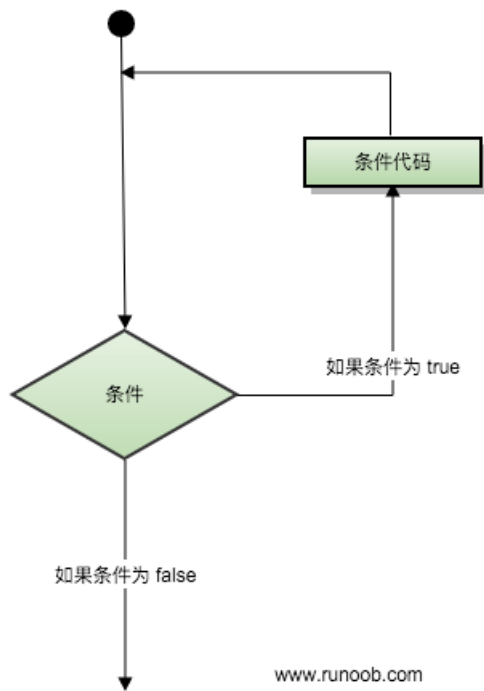
[Perl while 循环](#)

## Perl 循环

有的时候，我们可能需要多次执行同一块代码。一般情况下，语句是按顺序执行的：函数中的第一个语句先执行，接着是第二个语句，依此类推。

编程语言提供了更为复杂执行路径的多种控制结构。

循环语句允许我们多次执行一个语句或语句组，下面是大多数编程语言中循环语句的流程图：



注意，数字 0, 字符串 '0'、'', 空 list (), 和 undef 为 **false**，其他值均为 **true**。true 前面使用 **!** 或 **not** 则返回 false。

Perl 语言提供了以下几种循环类型：

循环类型	描述
<a href="#">while 循环</a>	当给定条件为 <b>true</b> 时，重复执行语句或语句组。循环主体执行之前会先测试条件。
<a href="#">until 循环</a>	重复执行语句或语句组，直到给定的条件为 <b>true</b> 。循环主体执行之前会先测试条件。
<a href="#">for 循环</a>	多次执行一个语句序列，简化管理循环变量的代码。
<a href="#">foreach 循环</a>	foreach 循环用于迭代一个列表或集合变量的值。
<a href="#">do..while 循环</a>	除了它是在循环主体结尾测试条件外，其他与 <b>while</b> 语句类似。
<a href="#">嵌套循环</a>	您可以在 <b>while</b> 、 <b>for</b> 或 <b>do..while</b> 循环内使用一个或多个循环。

## 循环控制语句

循环控制语句改变了代码的执行顺序，通过它你可以实现代码的跳转。

Perl 提供了下列的循环控制语句：

控制语句	描述
<a href="#">next 语句</a>	停止执行从next语句的下一语句开始到循环体结束标识符之间的语句，转去执行continue语句块，然后再返回到循环体的起始处开始执行下一次循环。
<a href="#">last 语句</a>	退出循环语句块，从而结束循环
<a href="#">continue 语句</a>	continue 语句块通常在条件语句再次判断前执行。
<a href="#">redo 语句</a>	redo 语句直接转到循环体的第一行开始重复执行本次循环，redo语句之后的语句不再执行，continue语句块也不再执行；
<a href="#">goto 语句</a>	Perl 有三种 goto 形式：got LABEL，goto EXPR，和 goto &NAME。

## 无限循环

如果条件永远不为 **false**，则循环将变成无限循环。

**for** 循环在传统意义上可用于实现无限循环。

由于构成循环的三个表达式中任何一个都不是必需的，您可以将某些条件表达式留空来构成一个无限循环。

## 实例

```
#!/usr/bin/perl
for( ; ; )
{
    printf "循环会无限执行。\\n";
}
```

你可以按下 **Ctrl + C** 键来终止循环。

当条件表达式不存在时，它被假设为 **true**。您也可以设置一个初始值和增量表达式，但是一般情况下，Perl 程序员偏向于使用 **for(;;)** 结构来表示一个无限循环。

❏ 点我分享笔记

反馈/建议



# Perl 运算符

运算符是一种告诉编译器执行特定的数学或逻辑操作的符号，如: **3+2=5**。

Perl 语言内置了丰富的运算符，我们来看下常用的几种：

- 算术运算符
- 比较运算符
- 逻辑运算符
- 赋值运算符
- 位运算符
- 引号运算符
- 其他运算符
- 运算符优先级

## 算术运算符

表格实例中我们设置变量 **\$a** 为 10，**\$b** 为 20。

运算符	描述	实例
-----	----	----

+	加法运算	<code>\$a + \$b</code> 结果为 30
-	减法运算	<code>\$a - \$b</code> 结果为 -10
*	乘法运算	<code>\$a * \$b</code> 结果为 200
/	除法运算	<code>\$b / \$a</code> 结果为 2
%	求余运算，整除后的余数	<code>\$b % \$a</code> 结果为 0
**	乘幂	<code>\$a**\$b</code> 结果为 10 的 20 次方

## 实例

```
#!/usr/bin/perl
$a = 10;
$b = 20;
print "\$a = $a , \$b = $b\n";
$c = $a + $b;
print '$a + $b = ' . $c . "\n";
$c = $a - $b;
print '$a - $b = ' . $c . "\n";
$c = $a * $b;
print '$a * $b = ' . $c . "\n";
$c = $a / $b;
print '$a / $b = ' . $c . "\n";
$c = $a % $b;
print '$a % $b = ' . $c . "\n";
$a = 2;
$b = 4;
$c = $a ** $b;
print '$a ** $b = ' . $c . "\n";
```

以上程序执行输出结果为：

```
$a = 10 , $b = 20

$a + $b = 30

$a - $b = -10

$a * $b = 200

$a / $b = 0.5

$a % $b = 10

$a ** $b = 16
```

## 比较运算符

表格实例中我们设置变量 `$a` 为 10，`$b` 为 20。

运算符	描述	实例
<code>==</code>	检查两个操作数的值是否相等，如果相等则条件为 <code>true</code> ，否则为 <code>false</code> 。	<code>(\$a == \$b)</code> 为 <code>false</code>
<code>!=</code>	检查两个操作数的值是否相等，如果不相等则条件为 <code>true</code> ，否则为 <code>false</code> 。	<code>(\$a != \$b)</code> 为 <code>true</code> 。
<code>&lt;=&gt;</code>	检查两个操作数的值是否相等，如果左边的数小于右边的数返回 -1，如果相等返回 0，如果左边的数大于右边的数返回 1。	<code>(\$a &lt;=&gt; \$b)</code> 返回 -1。

>	检查左操作数的值是否大于右操作数的值，如果是则条件为 <b>true</b> ，否则为 <b>false</b> 。	(\$a > \$b) 返回 <b>false</b> 。
<	检查左操作数的值是否小于右操作数的值，如果是则条件为 <b>true</b> ，否则返回 <b>false</b> 。	(\$a < \$b) 返回 <b>true</b> 。
>=	检查左操作数的值是否大于或等于右操作数的值，如果是则条件为 <b>true</b> ，否则返回 <b>false</b> 。	(\$a >= \$b) 返回 <b>false</b> 。
<=	检查左操作数的值是否小于或等于右操作数的值，如果是则条件为 <b>true</b> ，否则返回 <b>false</b> 。。	(\$a <= \$b) 返回 <b>true</b> 。

## 实例

```
#!/usr/bin/perl
$a = 10;
$b = 20;
print "\$a = $a , \$b = $b\n";
if( $a == $b ){
    print "$a == $b 结果 true\n";
}else{
    print "\$a == $b 结果 false\n";
}
if( $a != $b ){
    print "\$a != $b 结果 true\n";
}else{
    print "\$a != $b 结果 false\n";
}
$c = $a <=> $b;
print "\$a <=> $b 返回 $c\n";
if( $a > $b ){
    print "\$a > $b 结果 true\n";
}else{
    print "\$a > $b 结果 false\n";
}
if( $a >= $b ){
    print "\$a >= $b 结果 true\n";
}else{
    print "\$a >= $b 结果 false\n";
}
if( $a < $b ){
    print "\$a < $b 结果 true\n";
}else{
    print "\$a < $b 结果 false\n";
}
if( $a <= $b ){
    print "\$a <= $b 结果 true\n";
}else{
    print "\$a <= $b 结果 false\n";
}
}
```

以上程序执行输出结果为：

```
$a = 10 , $b = 20

$a == $b 结果 false

$a != $b 结果 true

$a <=> $b 返回 -1

$a > $b 结果 false

$a >= $b 结果 false
```



\$a < \$b 结果 true

\$a <= \$b 结果 true

以下表格实例中设置变量 \$a 为 "abc"，\$b 为 "xyz"，然后使用比较运算符来计算结果。

运算符	描述	实例
lt	检查左边的字符串是否小于右边的字符串，如果是返回 true，否则返回 false。	(\$a lt \$b) 返回 true。
gt	检查左边的字符串是否大于右边的字符串，如果是返回 true，否则返回 false。	(\$a gt \$b) 返回 false。
le	检查左边的字符串是否小于或等于右边的字符串，如果是返回 true，否则返回 false。	(\$a le \$b) 返回 true
ge	检查左边的字符串是否大于或等于右边的字符串，如果是返回 true，否则返回 false。	(\$a ge \$b) 返回 false。
eq	检查左边的字符串是否等于右边的字符串，如果是返回 true，否则返回 false。	(\$a eq \$b) 返回 false。
ne	检查左边的字符串是否不等于右边的字符串，如果是返回 true，否则返回 false。	(\$a ne \$b) 返回 true
cmp	如果左边的字符串大于右边的字符串返回 1，如果相等返回 0，如果左边的字符串小于右边的字符串返回 -1。	(\$a cmp \$b) 返回 -1。

### 实例

```
#!/usr/bin/perl
$a = "abc";
$b = "xyz";
print "\$a = $a , \$b = $b\n";
if( $a lt $b ){
print "$a lt \$b 返回 true\n";
}else{
print "\$a lt \$b 返回 false\n";
}
if( $a gt $b ){
print "\$a gt \$b 返回 true\n";
}else{
print "\$a gt \$b 返回 false\n";
}
if( $a le $b ){
print "\$a le \$b 返回 true\n";
}else{
print "\$a le \$b 返回 false\n";
}
if( $a ge $b ){
print "\$a ge \$b 返回 true\n";
}else{
print "\$a ge \$b 返回 false\n";
}
if( $a ne $b ){
print "\$a ne \$b 返回 true\n";
}else{
print "\$a ne \$b 返回 false\n";
}
$c = $a cmp $b;
print "\$a cmp \$b 返回 $c\n";
```

以上程序执行输出结果为：

\$a = abc , \$b = xyz

```
abc lt $b 返回 true

$a gt $b 返回 false

$a le $b 返回 true

$a ge $b 返回 false

$a ne $b 返回 true

$a cmp $b 返回 -1
```

赋值运算符

表格实例中我们设置变量 \$a 为 10， \$b 为 20。

运算符	描述	实例
=	简单的赋值运算符，把右边操作数的值赋给左边操作数	\$c = \$a + \$b 将把 \$a + \$b 的值赋给 \$c
+=	加且赋值运算符，把右边操作数加上左边操作数的结果赋值给左边操作数	\$c += \$a 等于 \$c = \$c + \$a
-=	减且赋值运算符，把左边操作数减去右边操作数的结果赋值给左边操作数	\$c -= \$a 等于 \$c = \$c - \$a
*=	乘且赋值运算符，把右边操作数乘以左边操作数的结果赋值给左边操作数	\$c *= \$a 等于 \$c = \$c * \$a
/=	除且赋值运算符，把左边操作数除以右边操作数的结果赋值给左边操作数	\$c /= \$a 等于 \$c = \$c / \$a
%=	求模且赋值运算符，求两个操作数的模赋值给左边操作数	\$c %= \$a 等于 \$c = \$c % a
**=	乘幂且赋值运算符，求两个操作数的乘幂赋值给左边操作数	\$c **= \$a 等于 \$c = \$c ** \$a

实例

```
#!/usr/bin/perl
$a = 10;
$b = 20;
print "\$a = $a , \$b = $b\n";
$c = $a + $b;
print "赋值后 \$c = $c\n";
$c += $a;
print "\$c = $c , 运算语句 \$c += \$a\n";
$c -= $a;
print "\$c = $c , 运算语句 \$c -= \$a\n";
$c *= $a;
print "\$c = $c , 运算语句 \$c *= \$a\n";
$c /= $a;
print "\$c = $c , 运算语句 \$c /= \$a\n";
$c %= $a;
print "\$c = $c , 运算语句 \$c %= \$a\n";
$c = 2;
$a = 4;
print "\$a = $a , \$c = $c\n";
$c **= $a;
print "\$c = $c , 运算语句 \$c **= \$a\n";
```

以上程序执行输出结果为：

```
$a = 10 , $b = 20

赋值后 $c = 30

$c = 40 , 运算语句 $c += $a

$c = 30 , 运算语句 $c -= $a
```

```
$c = 300 , 运算语句 $c *= $a

$c = 30 , 运算语句 $c /= $a

$c = 0 , 运算语句 $c %= $a

$a = 4 ,   $c = 2

$c = 16 , 运算语句 $c **= $a
```

## 位运算

位运算符作用于位，并逐位执行操作。

设置 **\$a = 60**, **\$b = 13**，现在以二进制格式表示，它们如下所示：

```
$a = 0011 1100

$b = 0000 1101

-----

$a&$b = 0000 1100

$a|$b = 0011 1101

$a^$b = 0011 0001

~$a   = 1100 0011
```

Perl 支持的位运算符如下表所示：

运算符	描述	实例
&	如果同时存在于两个操作数中，二进制 <b>AND</b> 运算符复制一位到结果中。	( <b>\$a &amp; \$b</b> ) 将得到 <b>12</b> ，二进制为 <b>0000 1100</b>
	如果存在于任一操作数中，二进制 <b>OR</b> 运算符复制一位到结果中。	( <b>\$a   \$b</b> ) 将得到 <b>61</b> ，二进制为 <b>0011 1101</b>
^	如果存在于其中一个操作数中但不同时存在于两个操作数中，二进制异或运算符复制一位到结果中。	( <b>\$a ^ \$b</b> ) 将得到 <b>49</b> ，二进制为 <b>0011 0001</b>
~	二进制反码运算符是一元运算符，具有"翻转"位效果，即 <b>0</b> 变成 <b>1</b> ， <b>1</b> 变成 <b>0</b> 。	( <b>~\$a</b> ) 将得到 <b>-61</b> ，二进制为 <b>1100 0011</b> ，一个有符号二进制数的反码形式。
<<	二进制左移运算符。左操作数的值向左移动右操作数指定的位数。	<b>\$a &lt;&lt; 2</b> 将得到 <b>240</b> ，二进制为 <b>1111 0000</b>
>>	二进制右移运算符。左操作数的值向右移动右操作数指定的位数。	<b>\$a &gt;&gt; 2</b> 将得到 <b>15</b> ，二进制为 <b>0000 1111</b>

实例

```
#!/usr/bin/perl
use integer;
$a = 60;
$b = 13;
print "\$a = $a , \$b = $b\n";
$c = $a & $b;
print "\$a & \$b = $c\n";
$c = $a | $b;
print "\$a | \$b = $c\n";
$c = $a ^ $b;
print "\$a ^ \$b = $c\n";
$c = ~$a;
print "~\$a = $c\n";
$c = $a << 2;
print "\$a << 2 = $c\n";
$c = $a >> 2;
print "\$a >> 2 = $c\n";
```

以上程序执行输出结果为：

```
$a = 60 , $b = 13

$a & $b = 12

$a | $b = 61

$a ^ $b = 49

~$a = -61

$a << 2 = 240

$a >> 2 = 15
```

逻辑运算符

Perl 逻辑运算符如下表所示。  
表格实例中我们设置变量 \$a 为 true, \$b 为 false。

运算符	描述	实例
and	逻辑与运算符。如果两个操作数都为 true，则条件为 true。	(\$a and \$b) 为 false。
&&	C 风格的逻辑与运算符。如果两个操作数都为 true，则条件为 true	(\$a && \$b) 为 false。
or	逻辑或运算符。如果两个操作数中有任意一个非零，则条件为 true。	(\$a or \$b) 为 true。
	C 风格逻辑或运算符。如果两个操作数中有任意一个非零，则条件为 true。	(\$a    \$b) 为 true。
not	逻辑非运算符。用来反转操作数的逻辑状态。如果条件为 true，则逻辑非运算符将使其为 false。	not(\$a and \$b) 为 true。

实例

```
#!/usr/bin/perl
$a = true;
$b = false;
print "\$a = $a , \$b = $b\n";
$c = ($a and $b);
print "\$a and \$b = $c\n";
$c = ($a && $b);
print "\$a && \$b = $c\n";
```

```
$c = ($a or $b);
print "\$a or \$b = $c\n";
$c = ($a || $b);
print "\$a || \$b = $c\n";
$a = 0;
$c = not($a);
print "not(\$a)= $c\n";
```

以上程序执行输出结果为：

```
$a = true , $b = false

$a and $b = false

$a && $b = false

$a or $b = true

$a || $b = true

not($a)= 1
```

## 引号运算

Perl 引号运算符如下表所示。

运算符	描述	实例
q{ }	为字符串添加单引号	q{abcd} 结果为 'abcd'
qq{ }	为字符串添加双引号	qq{abcd} 结果为 "abcd"
qx{ }	为字符串添加反引号	qx{abcd} 结果为 `abcd`

## 实例

```
#!/usr/bin/perl
$a = 10;
$b = q{a = $a};
print "q{a = \"$a\"} = $b\n";
$b = qq{a = $a};
print "qq{a = \"$a\"} = $b\n";
# 使用 unix 的 date 命令执行
$t = qx{date};
print "qx{date} = $t\n";
```

以上程序执行输出结果为：

```
q{a = $a} = a = $a

qq{a = $a} = a = 10

qx{date} = 2016年 6月10日 星期五 16时22分33秒 CST
```

## 其他运算符

除了以上我们提到的运算符外，Perl 还支持以下运算符：

运算符	描述	实例
-----	----	----

.	点号 (.) 用于连接两个字符串。	如果 \$a="run", \$b="oob" , \$a.\$b 结果为 "runoob"
x	x 运算符返回字符串重复的次数。	(' ' x 3) 输出为 一。
..	.. 为范围运算符。	(2..5) 输出结果为 (2, 3, 4, 5)
++	自增运算符, 整数值增加 1	\$a =10, \$a++ will 输出为 11
—	自减运算符, 整数值减少 1	\$a =10, \$a— 输出为 9
->	箭号用于指定一个类的方法	\$obj->\$a 表示对象 \$obj 的 \$a 方法。

实例

```
#!/usr/bin/perl
$a = "run";
$b = "oob";
print "\$a = $a ,  \$b = $b\n";
$c = $a . $b;
print "\$a . \$b = $c\n";
$c = "-" x 3;
print "\"-\" x 3 = $c\n";
@c = (2..5);
print "(2..5) = @c\n";
$a = 10;
$b = 15;
print "\$a = $a ,  \$b = $b\n";
$a++;
$c = $a ;
print "\$a 执行 \$a++ = $c\n";
$b--;
$c = $b ;
print "\$b 执行 \$b-- = $c\n";
```

以上程序执行输出结果为:

```
$a  = run ,  $b = oob

$a . $b = runoob

"-" x 3 = ---

(2..5) = 2 3 4 5

$a  = 10 ,  $b = 15

$a  执行 $a++ = 11

$b  执行 $b-- = 14
```

运算符优先级

下表列出了 Perl 语言的运算符优先级:

运算符	结合性
++, --	无
~, ~!, !	从右到左
**	从右到左

=~, !~	从左到右
*, /, %, x	从左到右
+, -, .	从左到右
<<, >>	从左到右
-e, -r,	无
<, <=, >, >=, lt, le, gt, ge	从左到右
==, !=, <=>, eq, ne, cmp	从左到右
&	从左到右
, ^	从左到右
&&	从左到右
	从左到右
..	从左到右
? and :	从右到左
=, +=, -=, *=,	从右到左
其他	
,	从左到右
not	从左到右
and	从左到右
or, xor	从左到右

## 实例

```
#!/usr/bin/perl
$a = 20;
$b = 10;
$c = 15;
$d = 5;
$e;
print "\$a = $a, \$b = $b, \$c = $c , \$d = $d\n";
$e = ($a + $b) * $c / $d;
print "(\$a + \$b) * \$c / \$d = $e\n";
$e = (($a + $b) * $c) / $d;
print "((\$a + \$b) * \$c) / \$d = $e\n";
$e = ($a + $b) * ($c / $d);
print "(\$a + \$b) * (\$c / \$d) = $e\n";
$e = $a + ($b * $c) / $d;
print "\$a + (\$b * \$c) / \$d = $e\n";
```

以上程序执行输出结果为:

```
$a  = 20, $b = 10, $c = 15 , $d = 5

($a + $b) * $c / $d  = 90
```

```
(( $a + $b ) * $c ) / $d  = 90

( $a + $b ) * ( $c / $d )  = 90

$a + ( $b * $c ) / $d  = 50
```



# Perl 时间日期

本章节为大家介绍 Perl 语言对时间日期的处理。

Perl中处理时间的函数有如下几种：

- 1、**time()** 函数： 返回从1970年1月1日起累计的秒数
- 2、**localtime()** 函数： 获取本地时区时间
- 3、**gmtime()** 函数： 获取格林威治时间

## 当前时间和日期

接下来让我们看下 **localtime()** 函数，该函数在没有参数的情况下返回当前的时间和日期。

以下 9 个符号代表不同的时间日期参数：

```
sec,      # 秒, 0 到 61

min,      # 分钟, 0 到 59

hour,     # 小时, 0 到 24

mday,     # 天, 1 到 31

mon,      # 月, 0 到 11

year,     # 年, 从 1900 开始

yday,     # 星期几, 0-6,0表示周日

yday,     # 一年中的第几天,0-364,365

isdst     # 如果夏令时有效,则为真
```



实例演示如下：

## 实例

```
#!/usr/bin/perl
@months = qw( 一月 二月 三月 四月 五月 六月 七月 八月 九月 十月 十一月 十二月 );
@days = qw(星期天 星期一 星期二 星期三 星期四 星期五 星期六);
($sec,$min,$hour,$mday,$mon,$year,$wday,$yday,$isdst) = localtime();
print "$mday $months[$mon] $days[$wday]\n";
```

以上实例执行输出结果为：

```
12 六月 星期天
```

如果直接调用 `localtime()`，它返回系统当期设置时区的时间，实例如下：

## 实例

```
#!/usr/bin/perl
$datestring = localtime();
print "时间日期为: $datestring\n";
```

以上实例执行输出结果为：

```
时间日期为: Sun Jun 12 11:27:31 2016
```

## 格林威治时间 (GMT)

函数 `gmtime()` 与 `localtime()` 类似，但它返回标准格林威治时间。

## 实例

```
#!/usr/bin/perl
$local_datestring = localtime();
print "本地时间日期为: $local_datestring\n";
$gmt_datestring = gmtime();
print "GMT 时间日期为: $gmt_datestring\n";
```

以上实例执行输出结果为：

```
本地时间日期为: Sun Jun 12 11:32:14 2016
```

```
GMT 时间日期为: Sun Jun 12 03:32:14 2016
```

从实例中我们可以看出，中国的时间和格林威治时间相差了8小时。

## 格式化日期和时间

我们可以使用 `localtime()` 函数的 9 个时间元素来输出需要制定的格式时间。格式化输出使用 `printf()` 函数：

## 实例

```
#!/usr/bin/perl
($sec,$min,$hour,$mday,$mon,$year,$wday,$yday,$isdst) = localtime();
printf("格式化时间: HH:MM:SS\n");
printf("%02d:%02d:%02d", $hour, $min, $sec);
```

以上实例执行输出结果为：

```
格式化时间：HH:MM:SS

11:35:23
```

## 新纪元时间(Epoch Time)

我们可以使用 `time()` 函数来获取新纪元时间，该函数返回从1970年1月1日起累计的秒数。实例如下：

```
实例

#!/usr/bin/perl
$epoc = time();
print "从1970年1月1日起累计的秒数为: $epoc\n";
```

以上实例执行输出结果为：

```
从1970年1月1日起累计的秒数为: 1465702883
```

我们可以输出一个我们想要的时间格式：

```
实例

#!/usr/bin/perl
($sec,$min,$hour,$mday,$mon,$year,$wday,$yday,$isdst) = localtime();
print "当前时间和日期: ";
printf("%d-%d-%d %d:%d:%d", $year+1900,$mon+1,$mday,$hour,$min,$sec);
print "\n";
$epoc = time();
$epoc = $epoc - 24 * 60 * 60; # 一天前的时间秒数
($sec,$min,$hour,$mday,$mon,$year,$wday,$yday,$isdst) = localtime($epoc);
print "昨天时间和日期: ";
printf("%d-%d-%d %d:%d:%d", $year+1900,$mon+1,$mday,$hour,$min,$sec);
print "\n";
```

以上实例执行输出结果为：

```
当前时间和日期: 2017-3-15 12:47:54

昨天时间和日期: 2017-3-14 12:47:54
```

## POSIX 函数 strftime()

函数 `strftime()` 可以将时间格式化为我们想要的格式。

下表列出了一些格式化的符号，\* 号表示想要依赖本地时间：

符号	描述	实例
%a	星期几的简称（ Sun..Sat） *	Thu
%A	星期几的全称（ Sunday..Saturday） *	Thursday
%b	月的简称（ Jan..Dec） *	Aug
%B	月的全称（ January..December） *	August

%c	日期和时间 *	Thu Aug 23 14:55:02 2001
%C	年份除于 <b>100</b> ，并取整 (00-99)	20
%d	一个月的第几天 (01-31)	23
%D	日期, MM/DD/YY 相等于 %m/%d/%y	08/23/01
%e	一个月的第几天，使用空格填充个位数 ( 1-31)	23
%F	YYYY-MM-DD 的简写类似于 %Y-%m-%d	2001-08-23
%g	年份的最后两位数 (00-99)	01
%g	年	2001
%h	月的简称 * (和%b选项相同)	Aug
%H	<b>24</b> 小时制 (00-23)	14
%I	<b>12</b> 小时制 (01-12)	02
%j	一年的第几天 (001-366)	235
%m	月 (01-12)	08
%M	分钟 (00-59)	55
%n	新行 ('\\n')	
%p	显示出 <b>AM</b> 或 <b>PM</b>	PM
%r	时间 (hh: mm: ss <b>AM</b> 或 <b>PM</b> )， 12小时 *	02:55:02 pm
%R	<b>24</b> 小时 HH:MM 时间格式,相等于 %H:%M	14:55
%S	秒数 (00-61)	02
%t	水平制表符 ('\\t')	
%T	时间 (24小时制) (hh:mm:ss)， 相等于%H:%M:%S	14:55
%u	<b>ISO 8601</b> 的星期几格式，星期一为1 (1-7)	4
%U	一年中的第几周，星期天为第一天(00-53)	33
%V	<b>ISO 8601</b> 第几周 (00-53)	34
%w	一个星期的第几天 (0代表星期天) (0-6)	4
%W	一年的第几个星期，星期一为第一天 (00-53)	34
%x	显示日期的格式 (mm/dd/yy) *	08/23/01
%X	显示时间格式 *	14:55:02
%y	年，两位数 (00-99)	01
%Y	年	2001
%z	<b>ISO 8601</b> 与 <b>UTC</b> 的时区偏移(1 minute=1, 1 hour=100)	+100

%Z	当前时区的名称,如"中国标准时间" *	CDT
%%	% 符号	%

实例

```
#!/usr/bin/perl
use POSIX qw(strftime);
$datestring = strftime "%Y-%m-%d %H:%M:%S", localtime;
printf("时间日期 - $datestring\n");
# GMT 格式化时间日期
$datestring = strftime "%Y-%m-%d %H:%M:%S", gmtime;
printf("时间日期 - $datestring\n");
```

以上实例执行输出结果为:

```
时间日期 - 2016-06-12 12:15:13

时间日期 - 2016-06-12 04:15:13
```

[❏ 点我分享笔记](#)



# Perl 子程序(函数)

Perl 子程序也就是用户定义的函数。

Perl 子程序即执行一个特殊任务的一段分离的代码，它可以使减少重复代码且使程序易读。

Perl 子程序可以出现在程序的任何地方，语法格式如下：

```
sub subroutine{

    statements;

}
```

调用子程序语法格式：

```
subroutine( 参数列表 );
```

在 Perl 5.0 以下版本调用子程序方法如下：

```
&subroutine( 参数列表 );
```

在新版本上，虽然也支持该调用方法，但不推荐使用。

接下来我们来看一个简单实例：

## 实例

```
#!/usr/bin/perl
# 函数定义
sub Hello{
    print "Hello, World!\n";
}
# 函数调用
Hello();
```

执行以上程序，输出结果为：

```
Hello, World!
```

## 向子程序传递参数

Perl 子程序可以和其他编程一样接受多个参数，子程序参数使用特殊数组 `@_` 标明。

因此子程序第一个参数为 `$_[0]`，第二个参数为 `$_[1]`，以此类推。

不论参数是标量型还是数组型的，用户把参数传给子程序时，perl默认按引用的方式调用它们。

## 实例

```
#!/usr/bin/perl
# 定义求平均值函数
sub Average{
    # 获取所有传入的参数
    $n = scalar(@_);
    $sum = 0;
    foreach $item (@_){
        $sum += $item;
    }
    $average = $sum / $n;
    print "传入的参数为 : ", "@_\n"; # 打印整个数组
    print "第一个参数值为 : $_[0]\n"; # 打印第一个参数
    print "传入参数的平均值为 : $average\n"; # 打印平均值
}
# 调用函数
Average(10, 20, 30);
```

执行以上程序，输出结果为：

```
传入的参数为 : 10 20 30
```

```
第一个参数值为 : 10
```

```
传入参数的平均值为 : 20
```

用户可以通过改变 `@_` 数组中的值来改变相应实际参数的值。

## 向子程序传递列表

由于 `@_` 变量是一个数组，所以它可以向子程序中传递列表。

但如果我们需要传入标量和数组参数时，需要把列表放在最后一个参数上，如下所示：

### 实例

```
#!/usr/bin/perl
# 定义函数
sub PrintList{
my @_list = @_;
print "列表为 : @list\n";
}
$a = 10;
@b = (1, 2, 3, 4);
# 列表参数
PrintList($a, @b);
```

以上程序将标量和数组合并了，输出结果为：

```
列表为 : 10 1 2 3 4
```

我们可以向子程序传入多个数组和哈希，但是在传入多个数组和哈希时，会导致丢失独立的标识。所以我们需要使用引用（下一章节会介绍）来传递。

## 向子程序传递哈希

当向子程序传递哈希表时，它将复制到 `@_` 中，哈希表将被展开为键/值组合的列表。

### 实例

```
#!/usr/bin/perl
# 方法定义
sub PrintHash{
my (%hash) = @_;
foreach my $key ( keys %hash ){
my $value = $hash{$key};
print "$key : $value\n";
}
}
%hash = ('name' => 'runoob', 'age' => 3);
# 传递哈希
PrintHash(%hash);
```

以上程序执行输出结果为：

```
age : 3

name : runoob
```

## 子程序返回值

子程序可以向其他编程语言一样使用 `return` 语句来返回函数值。

如果没有使用 `return` 语句，则子程序的最后一行语句将作为返回值。

### 实例

```
#!/usr/bin/perl
# 方法定义
sub add_a_b{
# 不使用 return
```

```
$_[0]+$_[1];  
# 使用 return  
# return $_[0]+$_[1];  
}  
print add_a_b(1, 2)
```

以上程序执行输出结果为：

3

子程序中我们可以返回标量，数组和哈希，但是在返回多个数组和哈希时，会导致丢失独立的标识。所以我们需要使用引用（下一章节会介绍）来返回多个数组和函数。

## 子程序的私有变量

默认情况下，**Perl** 中所有的变量都是全局变量，这就是说变量在程序的任何地方都可以调用。

如果我们需要设置私有变量，可以使用 **my** 操作符来设置。

**my** 操作符用于创建词法作用域变量，通过 **my** 创建的变量，存活于声明开始的地方，直到闭合作用域的结尾。

闭合作用域指的可以是一对花括号中的区域，可以是一个文件，也可以是一个 **if**, **while**, **for**, **foreach**, **eval** 字符串。

以下实例演示了如何声明一个或多个私有变量：

```
sub somefunc {  
  
    my $variable; # $variable 在方法 somefunc() 外不可见  
  
    my ($another, @an_array, %a_hash); # 同时声明多个变量  
  
}
```

## 实例

```
#!/usr/bin/perl  
# 全局变量  
$string = "Hello, World!";  
# 函数定义  
sub PrintHello{  
# PrintHello 函数的私有变量  
my $string;  
$string = "Hello, Runoob!";  
print "函数内字符串: $string\n";  
}  
# 调用函数  
PrintHello();  
print "函数外字符串: $string\n";
```

以上程序执行输出结果为：

函数内字符串: Hello, Runoob!

函数外字符串: Hello, World!

## 变量的临时赋值

我们可以使用 **local** 为全局变量提供临时的值，在退出作用域后将原来的值还回去。

**local** 定义的变量不存在于主程序中，但存在于该子程序和该子程序调用的子程序中。定义时可以给其赋值，如：

## 实例

```
#!/usr/bin/perl
# 全局变量
$string = "Hello, World!";
sub PrintRunoob{
# PrintHello 函数私有变量
local $string;
$string = "Hello, Runoob!";
# 子程序调用的子程序
PrintMe();
print "PrintRunoob 函数内字符串值: $string\n";
}
sub PrintMe{
print "PrintMe 函数内字符串值: $string\n";
}
sub PrintHello{
print "PrintHello 函数内字符串值: $string\n";
}
# 函数调用
PrintRunoob();
PrintHello();
print "函数外部字符串值: $string\n";
```

以上程序执行输出结果为:

```
PrintMe 函数内字符串值: Hello, Runoob!

PrintRunoob 函数内字符串值: Hello, Runoob!

PrintHello 函数内字符串值: Hello, World!

函数外部字符串值: Hello, World!
```

## 静态变量

**state**操作符功能类似于C里面的**static**修饰符，**state**关键字将局部变量变得持久。

**state**也是词法变量，所以只在定义该变量的词法作用域中有效，举个例子：

## 实例

```
#!/usr/bin/perl
use feature 'state';
sub PrintCount{
state $count = 0; # 初始化变量
print "counter 值为: $count\n";
$count++;
}
for (1..5){
PrintCount();
}
```

以上程序执行输出结果为:

```
counter 值为: 0

counter 值为: 1

counter 值为: 2

counter 值为: 3

counter 值为: 4
```



注1: `state` 仅能创建闭合作用域为子程序内部的变量。

注2: `state` 是从Perl 5.9.4开始引入的，所以使用前必须加上 `use`。

注3: `state` 可以声明标量、数组、哈希。但在声明数组和哈希时，不能对其初始化（至少Perl 5.14不支持）。

## 子程序调用上下文

子程序调用过程中，会根据上下文来返回不同类型的值，比如以下 `localtime()` 子程序，在标量上下文返回字符串，在列表上下文返回列表：

### 实例

```
#!/usr/bin/perl
# 标量上下文
my $datestring = localtime( time );
print $datestring;
print "\n";
# 列表上下文
($sec,$min,$hour,$mday,$mon, $year,$wday,$yday,$isdst) = localtime(time);
printf("%d-%d-%d %d:%d:%d", $year+1990,$mon+1,$mday,$hour,$min,$sec);
print "\n";
```

以上程序执行输出结果为：

```
Sun Jun 12 15:58:09 2016

2106-6-12 15:58:9
```



## Perl 引用

引用就是指针，Perl 引用是一个标量类型可以指向变量、数组、哈希表（也叫关联数组）甚至子程序，可以应用在程序的任何地方。

### 创建引用

定义变量的时候，在变量名前面加个`\`，就得到了这个变量的一个引用，比如：

```
$scalarref = \ $foo;      # 标量变量引用

$arrayref  = \@ARGV;      # 列表的引用

$hashref   = \%ENV;       # 哈希的引用

$coderef   = \&handler;   # 子过程引用

$globref   = \*foo;        # GLOB句柄引用
```

在数组中我们可以用匿名数组引用，使用 `[]` 定义：

```
$aref= [ 1,"foo",undef,13 ];
```

匿名数组的元素仍然可以是匿名数组，所以我们可以用这种方法构造数组的数组，可以构造任意维度的数组。

```
my $aref = [

    [1, 2, 3],

    [4, 5, 6],

    [7, 8, 9],

]
```

在哈希中我们可以用匿名哈希引用，使用 `{}` 定义：

```
$href= { APR =>4, AUG =>8 };
```

我们也可以创建一个没有子程序名的匿名子程序引用：

```
$coderef = sub { print "Runoob!\n" };
```

## 取消引用

取消引用可以根据不同的类型使用 `$`, `@` 或 `%` 来取消，实例如下：

### 实例

```
#!/usr/bin/perl
$var = 10;
# $r 引用 $var 标量
$r = \ $var;
# 输出本地存储的 $r 的变量值
print "$var 为 : ", $r, "\n";
@var = (1, 2, 3);
# $r 引用 @var 数组
$r = \@var;
# 输出本地存储的 $r 的变量值
print "@var 为: ", @$r, "\n";
%var = ( 'key1' => 10, 'key2' => 20);
# $r 引用 %var 数组
$r = \%var;
```

```
# 输出本地存储的 $r 的变量值
print "%var 为 : ", %$r, "\n";
```

执行以上实例执行结果为:

```
10 为 : 10

1 2 3 为: 123

%var 为 : key110key220
```

如果你不能确定变量类型, 你可以使用 **ref** 来判断, 返回值列表如下, 如果没有以下的值返回 **false**:

```
SCALAR

ARRAY

HASH

CODE

GLOB

REF
```

实例如下:

## 实例

```
#!/usr/bin/perl
$var = 10;
$r = \ $var;
print "r 的引用类型 : ", ref($r), "\n";
@var = (1, 2, 3);
$r = \@var;
print "r 的引用类型 : ", ref($r), "\n";
%var = ('key1' => 10, 'key2' => 20);
$r = \%var;
print "r 的引用类型 : ", ref($r), "\n";
```

执行以上实例执行结果为:

```
r 的引用类型 : SCALAR

r 的引用类型 : ARRAY

r 的引用类型 : HASH
```

## 循环引用

循环引用在两个引用相互包含时出现。你需要小心使用, 不然会导致内存泄露, 如下实例:

## 实例

```
#!/usr/bin/perl
my $foo = 100;
$foo = \ $foo;
print "Value of foo is : ", $$foo, "\n";
```

执行以上实例执行结果为:

Value of foo is : REF(0x9aae38)

## 引用函数

函数引用格式: \&

调用引用函数格式: & + 创建的引用名。

实例如下:

### 实例

```
#!/usr/bin/perl
# 函数定义
sub PrintHash{
my (%hash) = @_ ;
foreach $item (%hash){
print "元素 : $item\n";
}
}
%hash = ( 'name' => 'runoob', 'age' => 3 );
# 创建函数的引用
$cref = \&PrintHash;
# 使用引用调用函数
&$cref(%hash);
```

执行以上实例执行结果为:

元素 : age

元素 : 3

元素 : name

元素 : runoob

[Perl 子程序\(函数\)](#)

[Perl 格式化输出](#)

[点我分享笔记](#)

[反馈/建议](#)

Copyright © 2013-2018 菜鸟教程 [runoob.com](#) All Rights Reserved. 备案号: 闽ICP备15012807号-1



[首页](#) [HTML](#) [CSS](#) [JS](#) [本地书签](#)

[Perl 引用](#)

[Perl 文件操作](#)

# Perl 格式化输出

Perl 是一个非常强大的文本数据处理语言。

Perl 中可以使用 **format** 来定义一个模板，然后使用 **write** 按指定模板输出数据。

Perl 格式化定义语法格式如下：

```
format FormatName =  
  
fieldline  
  
value_one, value_two, value_three  
  
fieldline  
  
value_one, value_two  
  
.
```

参数解析：

**FormatName**：格式化名称。

**fieldline**：一个格式行，用来定义一个输出行的格式,类似 @,<,>,>| 这样的字符。

**value\_one,value\_two.....**：数据行，用来向前面的格式行中插入值,都是perl的变量。

**.**：结束符号。

以下是一个简单是格式化实例：

## 实例

```
#!/usr/bin/perl  
$text = "google runoob taobao";  
format STDOUT =  
first: ^<<<<< # 左边对齐，字符长度为6  
$text  
second: ^<<<<< # 左边对齐，字符长度为6  
$text  
third: ^<<<< # 左边对齐，字符长度为5，taobao 最后一个 o 被截断  
$text  
.  
write
```

执行以上实例输出结果为：

```
first: google  
  
second: runoob  
  
third: taoba
```

## 格式行(图形行)语法

格式行以 @ 或者 ^ 开头，这些行不作任何形式的变量代换。

@ 字段(不要同数组符号 @ 相混淆)是普通的字段。

@,< 后的 <,>,>| 长度决定了字段的长度，如果变量超出定义的长度,那么它将被截断。

<,>,>| 还分别表示,左对齐,右对齐,居中对齐。

^ 字段用于多行文本块填充。

## 值域格式

值域的格式，如下表所示：

格 式	值域含义
@<<<	左对齐输出
@>>>	右对齐输出
@	中对齐输出
@##.##	固定精度数字
@*	多行文本

每个值域的第一个字符是行填充符，当使用@字符时，不做文本格式化。

在上表中，除了多行值域@\*，域宽都等于其指定的包含字符@在内的字符个数，例如：

@###.##

表示七个字符宽，小数点前四个，小数点后两个。

实例如下：

## 实例

```
#!/usr/bin/perl  
format EMPLOYEE =  
  
=====
```

@<<<<<<<<<<<<<<<<<< @<<

```
$name, $age  
@#####.##  
$salary  
  
=====
```

.

```
select(STDOUT);  
$~ = EMPLOYEE;  
  
@n = ("Ali", "Runoob", "Jaffer");  
@a = (20, 30, 40);  
@s = (2000.00, 2500.00, 4000.00);  
$i = 0;  
  
foreach (@n){  
    $name = $_;  
    $age = $a[$i];  
    $salary = $s[$i++];  
    write;  
}
```

以上实例输出结果为:

```
=====
Ali                                     20
2000.00
=====
=====
=====
Runoob                                  30
2500.00
=====
```

```
=====

Jaffer                40

4000.00

=====
```

## 格式变量

`$~ ($FORMAT_NAME)`：格式名字 `$^ ($FORMAT_TOP_NAME)`：当前的表头格式名字存储在

`% ($FORMAT_PAGE_NUMBER)`：当前输出的页号

`$= ($FORMAT_LINES_PER_PAGE)`：每页中的行数

`$| ($FORMAT_AUTOFLUSH)`：是否自动刷新输出缓冲区存储

`$\L ($FORMAT_FORMFEED)`：在每一页(除了第一页)表头之前需要输出的字符串存储在

以下是一个简单是使用 `$~` 格式化的实例：

### 实例

```
#!/usr/bin/perl
$~ = "MYFORMAT"; # 指定缺省文件变量下所使用的格式
write; # 输出 $~ 所指定的格式
format MYFORMAT = # 定义格式 MYFORMAT
=====
Text # 菜鸟教程
=====
.
write;
```

执行以上实例输出结果为：

```
=====

Text # 菜鸟教程

=====

=====

Text # 菜鸟教程

=====
```

如果不指定`$~`的情况下，会输出名为`STDOUT`的格式：

### 实例

```
#!/usr/bin/perl
write; # 不指定$~的情况下会寻找名为STDOUT的格式
format STDOUT =
~用~号指定的文字不会被输出
-----
STDOUT格式
-----
.
```

执行以上实例输出结果为：

-----

STDOUT格式

-----

以下实例我们通过添加报表头部信息来演示 `$^` 或 `$FORMAT_TOP_NAME` 变量的使用：

## 实例

```
#!/usr/bin/perl
format EMPLOYEE =
=====
@<<<<<<<<<<<<<<<<<<< @<<
$name, $age
@#####.##
$salary
=====
.
format EMPLOYEE_TOP =
=====
Name Age
=====
.
select(STDOUT);
$_ = EMPLOYEE;
^ = EMPLOYEE_TOP;
@n = ("Ali", "Runoob", "Jaffer");
@a = (20, 30, 40);
@s = (2000.00, 2500.00, 4000.00);
$i = 0;
foreach (@n){
    $name = $_;
    $age = $a[$i];
    $salary = $s[$i++];
    write;
}
```

以上实例输出结果为:

Name	Age
Ali	20
2000.00	
Runoob	30
2500.00	
Jaffer	40



=====

## 实例

}

=====

=====

=====

2000.00

=====

=====

2500.00

=====

---

Jaffer 40

```
4000.00
```

```
=====
```

## 输出到其它文件

默认情况下函数**write**将结果输出到标准输出文件**STDOUT**，我们也可以使它将结果输出到任意其它的文件中。最简单的方法就是把文件变量作为参数传递给**write**，如：

```
write(MYFILE);
```

以上代码**write**就用缺省的名为**MYFILE**的打印格式输出到文件**MYFILE**中。

但是这样就不能用**\$~**变量来改变所使用的打印格式。系统变量**\$~**只对默认文件变量起作用，我们可以改变默认文件变量，改变**\$~**，再调用**write**。

### 实例

```
#!/usr/bin/perl
if (open(MYFILE, ">tmp")) {
    $~ = "MYFORMAT";
    write MYFILE; # 含文件变量的输出，此时会打印与变量同名的格式，即MYFILE。$~里指定的值被忽略。
    format MYFILE = # 与文件变量同名
    =====
    输入到文件中
    =====
    .
    close MYFILE;
}
```

执行成功后，我们可以查看 **tmp** 文件的内容，如下所示：

```
$ cat tmp
```

```
=====
```

```
    输入到文件中
```

```
=====
```

我们可以使用**select**改变默认文件变量时，它返回当前默认文件变量的内部表示，这样我们就可以创建子程序，按自己的想法输出，又不影响程序的其它部分。

### 实例

```
#!/usr/bin/perl
if (open(MYFILE, ">>tmp")) {
    select (MYFILE); # 使得默认文件变量的打印输出到MYFILE中
    $~ = "OTHER";
    write; # 默认文件变量，打印到select指定的文件中，必使用$~指定的格式 OTHER
    format OTHER =
    =====
    使用定义的格式输入到文件中
    =====
    .
    close MYFILE;
}
```

执行成功后，我们可以查看 **tmp** 文件的内容，如下所示：

```
$ cat tmp
```

```
=====
```

输入到文件中

```
=====
```

```
=====
```

使用定义的格式输入到文件中

```
=====
```

[Perl 引用](#)

[Perl 文件操作](#)

[点我分享笔记](#)

[反馈/建议](#)

Copyright © 2013-2018 菜鸟教程 [runoob.com](#) All Rights Reserved. 备案号：闽ICP备15012807号-1



[首页](#) [HTML](#) [CSS](#) [JS](#) [本地书签](#)

[Perl 格式化输出](#)

[Perl 目录操作](#)

## Perl 文件操作

Perl 使用一种叫做文件句柄类型的变量来操作文件。

从文件读取或者写入数据需要使用文件句柄。

文件句柄(file handle)是一个I/O连接的名称。

Perl提供了三种文件句柄:STDIN,STDOUT,STDERR, 分别代表标准输入、标准输出和标准出错输出。

Perl 中打开文件可以使用以下方式：

```
open FILEHANDLE, EXPR
```

```
open FILEHANDLE
```

```
sysopen FILEHANDLE, FILENAME, MODE, PERMS
```

```
sysopen FILEHANDLE, FILENAME, MODE
```

参数说明：

**FILEHANDLE:** 文件句柄，用于存放一个文件唯一标识符。

EXPR: 文件名及文件访问类型组成的表达式。

MODE: 文件访问类型。

PERMS: 访问权限位(permission bits)。

## Open 函数

以下代码我们使用 `open` 函数以只读的方式(<)打开文件 `file.txt`:

```
open(DATA, "<file.txt");
```

<表示只读方式。

代码中的 `DATA` 为文件句柄用于读取文件，以下实例将打开文件并将文件内容输出：

### 实例

```
#!/usr/bin/perl
open(DATA, "<file.txt") or die "file.txt 文件无法打开, $!";
while(<DATA>){
    print "$_";
}
```

以下代码以写入(>)的方式打开文件 `file.txt`:

```
open(DATA, ">file.txt") or die "file.txt 文件无法打开, $!";
```

>表示写入方式。

如果你需要以读写方式打开文件，可以在 > 或 < 字符前添加 + 号：

```
open(DATA, "+<file.txt"); or die "file.txt 文件无法打开, $!";
```

这种方式不会删除文件原来的内容，如果要删除，格式如下所示：

```
open DATA, "+>file.txt" or die "file.txt 文件无法打开, $!";
```

如果要向文件中追加数据，则在追加数据之前，只需要以追加方式打开文件即可：

```
open(DATA, ">>file.txt") || die "file.txt 文件无法打开, $!";
```

>> 表示向现有文件的尾部追加数据，如果需要读取要追加的文件内容可以添加 + 号：

```
open(DATA, "+>>file.txt") || die "file.txt 文件无法打开, $!";
```

下表列出了不同的访问模式：

模式	描述
< 或 r	只读方式打开，将文件指针指向文件头。
> 或 w	写入方式打开，将文件指针指向文件头并将文件大小截为零。如果文件不存在则尝试创建之。

>> 或 a	写入方式打开，将文件指针指向文件末尾。如果文件不存在则尝试创建之。
+< 或 r+	读写方式打开，将文件指针指向文件头。
+> 或 w+	读写方式打开，将文件指针指向文件头并将文件大小截为零。如果文件不存在则尝试创建之。
+>> 或 a+	读写方式打开，将文件指针指向文件末尾。如果文件不存在则尝试创建之。

## Sysopen函数

**sysopen** 函数类似于 **open** 函数，只是它们的参数形式不一样。

以下实例是以读写(+<filename)的方式打开文件：

```
sysopen(DATA, "file.txt", O_RDWR);
```

如果需要在更新文件前清空文件，则写法如下：

```
sysopen(DATA, "file.txt", O_RDWR|O_TRUNC );
```

你可以使用 **O\_CREAT** 来创建一个新的文件，**O\_WRONLY** 为只写模式，**O\_RDONLY** 为只读模式。

The **PERMS** 参数为八进制属性值，表示文件创建后的权限，默认为 **0x666**。

下表列出了可能的模式值：

模式	描述
O_RDWR	读写方式打开，将文件指针指向文件头。
O_RDONLY	只读方式打开，将文件指针指向文件头。
O_WRONLY	写入方式打开，将文件指针指向文件头并将文件大小截为零。如果文件不存在则尝试创建之。
O_CREAT	创建文件
O_APPEND	追加文件
O_TRUNC	将文件大小截为零
O_EXCL	如果使用O_CREAT时文件存在,就返回错误信息,它可以测试文件是否存在
O_NONBLOCK	非阻塞I/O使我们的操作要么成功，要么立即返回错误，不被阻塞。

## Close 函数

在文件使用完后，要关闭文件，以刷新与文件句柄相关联的输入输出缓冲区，关闭文件的语法如下：

```
close FILEHANDLE

close
```

**FILEHANDLE** 为指定的文件句柄，如果成功关闭则返回 **true**。

```
close(DATA) || die "无法关闭文件";
```

## 读写文件

向文件读写信息有以下几种不同的方式：

### <FILEHANDLE> 操作符

从打开的文件句柄读取信息的主要方法是 <FILEHANDLE> 操作符。在标量上下文中，它从文件句柄返回单一行。例如：

#### 实例

```
#!/usr/bin/perl
print "菜鸟教程网址?\n";
$name = <STDIN>;
print "网址: $name\n";
```

以上程序执行后，会显示以下信息，我们输入网址后 `print` 语句就会输出：

```
菜鸟教程网址？
http://www.runoob.com
网址： http://www.runoob.com
```

当我们使用 <FILEHANDLE> 操作符时，它会返回文件句柄中每一行的列表，例如我们可以导入所有的行到数组中。

实现创建 `import.txt` 文件，内容如下：

```
$ cat import.txt
```

```
1
```

```
2
```

```
3
```

读取 `import.txt` 并将每一行放到 `@lines` 数组中：

#### 实例

```
#!/usr/bin/perl
open(DATA,"<import.txt") or die "无法打开数据";
@lines = <DATA>;
print @lines; # 输出数组内容
close(DATA);
```

执行以上程序，输出结果为：

```
1
```

```
2
```

```
3
```

## getc 函数

`xgetc` 函数从指定的 `FILEHANDLE` 返回单一的字符，如果没指定返回 `STDIN`：

```
getc FILEHANDLE
```

```
getc
```

如果发生错误，或在文件句柄在文件末尾，则返回 `undef`。

## read 函数

`read` 函数用于从缓冲区的文件句柄读取信息。

这个函数用于从文件读取二进制数据。

```
read FILEHANDLE, SCALAR, LENGTH, OFFSET
```

```
read FILEHANDLE, SCALAR, LENGTH
```

参数说明：

**FILEHANDLE:** 文件句柄，用于存放一个文件唯一标识符。

**SCALAR:** 存贮结果，如果没有指定**OFFSET**，数据将放在**SCALAR**的开头。否则数据放在**SCALAR**中的**OFFSET**字节之后。

**LENGTH:** 读取的内容长度。

**OFFSET:** 偏移量。

如果读取成功返回读取的字节数，如果在文件结尾返回 `0`，如果发生错误返回 `undef`。

## print 函数

对于所有从文件句柄中读取信息的函数，在后端主要的写入函数为 `print`：

```
print FILEHANDLE LIST
```

```
print LIST
```

```
print
```

利用文件句柄和 `print` 函数可以把程序运行的结果发给输出设备(**STDOUT**：标准输出)，例如：

```
print "Hello World!\n";
```

## 文件拷贝

以下实例我们将打开一个已存在的文件 `file1.txt`，并读取它的每一行写入到文件 `file2.txt` 中：

### 实例

```
#!/usr/bin/perl
# 只读方式打开文件
open(DATA1, "<file1.txt");
# 打开新文件并写入
open(DATA2, ">file2.txt");
# 拷贝数据
while(<DATA1>)
{
    print DATA2 $_;
}
close( DATA1 );
close( DATA2 );
```

## 文件重命名

以下实例，我们将已存在的文件 `file1.txt` 重命名为 `file2.txt`，指定的目录是在 `/usr/runoob/test/` 下：

```
#!/usr/bin/perl

rename ("/usr/runoob/test/file1.txt", "/usr/runoob/test/file2.txt" );
```

函数 **renames** 只接受两个参数，只对已存在的文件进行重命名。

## 删除文件

以下实例我们演示了如何使用 **unlink** 函数来删除文件：

### 实例

```
#!/usr/bin/perl
unlink ("/usr/runoob/test/file1.txt");
```

## 指定文件位置

你可以使用 **tell** 函数来获取文件的位置，并通过使用 **seek** 函数来指定文件内的的位置：

### tell 函数

**tell** 函数用于获取文件位置：

```
tell FILEHANDLE

tell
```

如果指定 **FILEHANDLE** 该函数返回文件指针的位置，以字节计。如果没有指定则返回默认选取的文件句柄。

### seek 函数

**seek()**函数是通过文件句柄来移动文件读写指针的方式来读取或写入文件的，以字节为单位进行读取和写入：

```
seek FILEHANDLE, POSITION, WHENCE
```

参数说明：

**FILEHANDLE**：文件句柄，用于存放一个文件唯一标识符。

**POSITION**：表示文件句柄(读写位置指针)要移动的字节数。

**WHENCE**：表示文件句柄(读写位置指针)开始移动时的起始位置，可以取的值为0、1、2；分别表示文件开头、当前位置和文件尾。

以下实例为从文件开头读取 256 个字节：

```
seek DATA, 256, 0;
```

## 文件信息

Perl 的文件操作也可以先测试文件是否存在，是否可读写等。

我们可以先创建 **file1.txt** 文件，内如如下：

```
$ cat file1.txt

www.runoob.com
```



## 实例

```
#!/usr/bin/perl
my $file = "/usr/test/runoob/file1.txt";
my (@description, $size);
if (-e $file)
{
    push @description, '是一个二进制文件' if (-B _);
    push @description, '是一个socket(套接字)' if (-S _);
    push @description, '是一个文本文件' if (-T _);
    push @description, '是一个特殊块文件' if (-b _);
    push @description, '是一个特殊字符文件' if (-c _);
    push @description, '是一个目录' if (-d _);
    push @description, '文件存在' if (-x _);
    push @description, (($size = -s _) ? "$size 字节" : '空');
    print "$file 信息: ", join(' ', @description), "\n";
}
```

执行以上程序，输出结果为：

file1.txt 信息: 是一个文本文件, 15 字节

文件测试操作符如下表所示：

操作符	描述
-A	文件上一次被访问的时间(单位：天)
-B	是否为二进制文件
-C	文件的(inode)索引节点修改时间(单位：天)
-M	文件上一次被修改的时间(单位：天)
-O	文件被真实的UID所有
-R	文件或目录可以被真实的UID/GID读取
-S	为socket(套接字)
-T	是否为文本文件
-W	文件或目录可以被真实的UID/GID写入
-X	文件或目录可以被真实的UID/GID执行
-b	为block-special (特殊块)文件(如挂载磁盘)
-c	为character-special (特殊字符)文件(如I/O 设备)
-d	为目录
-e	文件或目录名存在
-f	为普通文件
-g	文件或目录具有setgid属性
-k	文件或目录设置了sticky位
-l	为符号链接
-o	文件被有效UID所有

-p	文件是命名管道(FIFO)
-r	文件可以被有效的UID/GID读取
-s	文件或目录存在且不为0(返回字节数)
-t	文件句柄为TTY(系统函数isatty())的返回结果；不能对文件名使用这个测试)
-u	文件或目录具有setuid属性
-w	文件可以被有效的UID/GID写入
-x	文件可以被有效的UID/GID执行
-z	文件存在，大小为0(目录恒为false)，即是否为空文件，

[Perl 格式化输出](#)

[Perl 目录操作](#)

[点我分享笔记](#)

反馈/建议

Copyright © 2013-2018 菜鸟教程 [runoob.com](#) All Rights Reserved. 备案号：闽ICP备15012807号-1



[首页](#) [HTML](#) [CSS](#) [JS](#) [本地书签](#)

[Perl 文件操作](#)

[Perl 错误处理](#)

## Perl 目录操作

以下列出了一些操作目录的标准函数：

```
opendir DIRHANDLE, EXPR # 打开目录

readdir DIRHANDLE        # 读取目录

rewinddir DIRHANDLE      # 定位指针到开头

telldir DIRHANDLE        # 返回目录的当前位置

seekdir DIRHANDLE, POS   # 定位指定到目录的 POS 位置

closedir DIRHANDLE       # 关闭目录
```

### 显示所有的文件

显示目录下的所有文件，以下实例使用了 glob 操作符，演示如下：

#### 实例

```
#!/usr/bin/perl
# 显示 /tmp 目录下的所有文件
$dir = "/tmp/*";
my @files = glob( $dir );
foreach (@files ){
print $_ . "\n";
}
# 显示 /tmp 目录下所有以 .c 结尾的文件
$dir = "/tmp/*.c";
@files = glob( $dir );
foreach (@files ){
print $_ . "\n";
}
# 显示所有隐藏文件
$dir = "/tmp/.*";
@files = glob( $dir );
foreach (@files ){
print $_ . "\n";
}
# 显示 /tmp 和 /home 目录下的所有文件
$dir = "/tmp/* /home/*";
@files = glob( $dir );
foreach (@files ){
print $_ . "\n";
}
```

以下实例可以列出当前目录下的所有文件：

## 实例

```
#!/usr/bin/perl
opendir (DIR, '.') or die "无法打开目录, $!";
while ($file = readdir DIR) {
print "$file\n";
}
closedir DIR;
```

如果你要显示 /tmp 目录下所有以 .c 结尾的文件，可以使用以下代码：

## 实例

```
#!/usr/bin/perl
opendir(DIR, '.') or die "无法打开目录, $!";
foreach (sort grep(/^.*\.c$/,readdir(DIR))){
print "$_\n";
}
closedir DIR;
```

## 创建一个新目录

我们可以使用 **mkdir** 函数来创建一个新目录，执行前你需要有足够的权限来创建目录：

## 实例

```
#!/usr/bin/perl
$dir = "/tmp/perl";
# 在 /tmp 目录下创建 perl 目录
mkdir( $dir ) or die "无法创建 $dir 目录, $!";
print "目录创建成功\n";
```

## 删除目录

我们可以使用 **rmdir** 函数来删除目录，执行该操作需要有足够权限。另外要删除的目录必须是空目录：

## 实例

```
#!/usr/bin/perl
$dir = "/tmp/perl";
# 删除 /tmp 目录下的 perl 目录
rmdir( $dir ) or die "无法删除 $dir 目录, $!";
```

```
print "目录删除成功\n";
```

## 切换目录

我们可以使用**chdir** 函数来切换当期目录，执行该操作需要有足够权限。实例如下：

### 实例

```
#!/usr/bin/perl
$dir = "/home";
# 将当期目录移动到 /home 目录下
chdir( $dir ) or die "无法切换目录到 $dir , $!";
print "你现在所在的目录为 $dir\n";
```

执行以上程序，输出结果为：

```
你现在所在的目录为 /home
```

[Perl 文件操作](#)

[Perl 错误处理](#)

[点我分享笔记](#)

[反馈/建议](#)

Copyright © 2013-2018 菜鸟教程 [runoob.com](#) All Rights Reserved. 备案号：闽ICP备15012807号-1



[首页](#) [HTML](#) [CSS](#) [JS](#) [本地书签](#)

[Perl 目录操作](#)

[Perl 特殊变量](#)

## Perl 错误处理

程序运行过程中，总会碰到各式各样的错误，比如打开一个不存在的文件。

程序运行过程中如果出现错误就会停止，我们就需要使用一些检测方法来避免错误，从而防止程序退出。

Perl 提供了多中处理错误发方法，接下来我们一一介绍。

### if 语句

**if 语句** 可以判断语句的返回值，实例如下：

```
if(open(DATA, $file)){

    ...

}else{

    die "Error: 无法打开文件 - $!";

}
```

程序中变量 `$!` 返回了错误信息。 我们也可以将以上代码简化为如下代码：

```
open(DATA, $file) || die "Error: 无法打开文件 - $!";
```

## unless 函数

**unless** 函数与 **if** 相反，只有在表达式返回 **false** 时才会执行，如下所示：

```
unless(chdir("/etc")){  
  
    die "Error: 无法打开目录 - $!";  
  
}
```

**unless** 语句在你设置错误提醒时是非常有用的。我么也可以将以上代码简写为：

```
die "Error: 无法打开目录!: $!" unless(chdir("/etc"));
```

以上错误信息只有在目录切换错误的情况下才会输出。

## 三元运算符

以下是一个三元运算符的简单实例：

```
print(exists($hash{value}) ? '存在' : '不存在',"\\n");
```

以上实例我们使用了三元运算符来判断哈希的值是否存在。

实例中包含了一个表达式两个值，格式为：**表达式 ? 值一 : 值二**。

## warn 函数

**warn** 函数用于触发一个警告信息，不会有其他操作，输出到 **STDERR**(标准输出文件)，通常用于给用户提示：

```
chdir('/etc') or warn "无法切换目录";
```

## die 函数

**die** 函数类似于 **warn**，但它会执行退出。一般用作错误信息的输出：

```
chdir('/etc') or die "无法切换目录";
```

## Carp 模块

在 **Perl** 脚本中，报告错误的常用方法是使用 **warn()** 或 **die()** 函数来报告或产生错误。而对于 **Carp** 模块，它可以对产生的消息提供额外级别的控制，

尤其是在模块内部。

标准 **Carp** 模块提供了 **warn()** 和 **die()** 函数的替代方法，它们在提供错误定位方面提供更多信息，而且更加友好。当在模块中使用时，错误消息中包含模块名称和行号。

## carp 函数

**carp**函数可以输出程序的跟踪信息，类似于 **warn** 函数，通常会将该信息发送到 **STDERR**:

```
package T;

require Exporter;

@ISA = qw/Exporter/;

@EXPORT = qw/function/;

use Carp;

sub function {

    carp "Error in module!";

}

1;
```

在脚本调用以下程序:

```
use T;

function();
```

执行以上程序，输出结果为:

```
Error in module! at test.pl line 4
```

## cluck 函数

**cluck()** 与 **warn()** 类似，提供了从产生错误处的栈回溯追踪。

```
package T;

require Exporter;

@ISA = qw/Exporter/;

@EXPORT = qw/function/;

use Carp qw(cluck);
```

```
sub function {

    cluck "Error in module!";

}

1;
```

在脚本调用以下程序:

```
use T;

function();
```

执行以上程序, 输出结果为:

```
Error in module! at T.pm line 9

    T::function() called at test.pl line 4
```

## croak 函数

croak() 与 die() 一样, 可以结束脚本。

```
package T;

require Exporter;

@ISA = qw/Exporter/;

@EXPORT = qw/function/;

use Carp;

sub function {

    croak "Error in module!";

}

1;
```

在脚本调用以下程序:

```
use T;

function();
```

执行以上程序，输出结果为：

```
Error in module! at test.pl line 4
```

## confess 函数

`confess()` 与 `die()` 类似，但提供了从产生错误处的栈回溯追踪。

```
package T;

require Exporter;

@ISA = qw/Exporter/;

@EXPORT = qw/function/;

use Carp;

sub function {

    confess "Error in module!";

}

1;
```

在脚本调用以下程序：

```
use T;

function();
```

执行以上程序，输出结果为：

```
Error in module! at T.pm line 9

    T::function() called at test.pl line 4
```

[☐ Perl 目录操作](#)

[Perl 特殊变量](#) ☐

[☐ 点我分享笔记](#)

[反馈/建议](#)



[首页](#) [HTML](#) [CSS](#) [JS](#) [本地书签](#)[Perl 错误处理](#)[Perl 正则表达式](#)

## Perl 特殊变量

Perl 语言中定义了一些特殊的变量, 通常以 `$`, `@`, 或 `%` 作为前缀, 例如: `$_`。

很多特殊的变量有一个很长的英文名, 操作系统变量 `$!` 可以写为 `$OS_ERROR`。

如果你想使用英文名的特殊变量需要在程序头部添加 **use English;**。这样就可以使用具有描述性的英文特殊变量。

最常用的特殊变量为 `$_`, 该变量包含了默认输入和模式匹配内容。实例如下:

### 实例

```
#!/usr/bin/perl
foreach ('Google', 'Runoob', 'Taobao') {
    print $_;
    print "\n";
}
```

执行以上程序, 输出结果为:

Google

Runoob

Taobao

以下实例我们不使用 `$_` 来输出内容:

### 实例

```
#!/usr/bin/perl
foreach ('Google', 'Runoob', 'Taobao') {
    print;
    print "\n";
}
```

执行以上程序, 输出结果为:

Google

Runoob

Taobao

实例中, 首先输出 "Google", 接着输出 "Runoob", 最后输出 "Taobao"。

在迭代循环中, 当前循环的字符串会放在 `$_` 中, 然后 通过 `print` 输出。另外 `print` 在不指定输出变量, 默认情况下使用的也是 `$_`。

以下是几处即使没有写明 `Perl` 也会假定使用 `$_` 的地方:

各种单目函数, 包括像 `ord()` 和 `int()` 这样的函数以及除 `"-t"` 以外所有的文件 测试操作 (`"-f"`, `"-d"`), `"-t"` 默认操作 `STDIN`。

各种列表函数, 例如 `print()` 和 `unlink()`。

没有使用 `"=~"` 运算符时的模式匹配操作 `"m/"`、`"s/"` 和`"tr/"`。

在没有给出其他变量时是 `"foreach"` 循环的默认迭代变量。

`grep()` 和 `map()` 函数的隐含迭代变量。

当 `"while"` 仅有唯一条件，且该条件是对 `""`操作的结果进行测试时，`$_` 就是存放输入记录的默认位置。除了`"while"` 测试条件之外不会发生这种情况。(助记：下划线在特定操作中是可以省略的。)

## 特殊变量类型

根据特殊的变量的使用性质，可以分为以下几类：

全局标量特殊变量。

全局数组特殊变量。

全局哈希特殊变量。

全局特殊文件句柄。

全局特殊常量。

正则表达式特殊变量。

文件句柄特殊变量。

## 全局标量特殊变量

以下列出了所有的标量特殊变量，包含了特殊字符与英文形式的变量：

<code>\$_</code>	默认输入和模式匹配内容。
<code>\$ARG</code>	
<code>\$.</code>	前一次读的文件句柄的当前行号
<code>\$NR</code>	
<code>\$/</code>	输入记录分隔符,默认是新行字符。如用 <code>undef</code> 这个变量,将读到文件结尾。
<code>\$RS</code>	
<code>\$,</code>	输出域分隔符
<code>\$OFS</code>	
<code>\$\</code>	输出记录分隔符
<code>\$ORS</code>	
<code>\$"</code>	该变量同 <code>\$,</code> 类似，但应用于向双引号引起的字符串(或类似的内插字符串)中内插数组和切片值的场合。默认为一个空格。
<code>\$LIST_SEPARATOR</code>	
<code>\$;</code>	在仿真多维数组时使用的分隔符。默认为 <code>"\034"</code> 。
<code>\$\$SUBSCRIPT_SEPARATOR</code>	
<code>\$\L</code>	发送到输出通道的走纸换页符。默认为 <code>"\f"</code> 。
<code>\$\$FORMAT_FORMFEED</code>	
<code>\$:</code>	The current set of characters after which a string may be broken to fill continuation fields (starting with <code>^</code> ) in a format. Default is <code>"\n"</code> .
<code>\$\$FORMAT_LINE_BREAK_CHARACTERS</code>	

<code>^A</code>	打印前用于保存格式化数据的变量
<code>\$ACCUMULATOR</code>	
<code>\$#</code>	打印数字时默认的数字输出格式（已废弃）。
<code>\$OFMT</code>	
<code>\$?</code>	返回上一个外部命令的状态
<code>\$CHILD_ERROR</code>	
<code>\$!</code>	这个变量的数字值是 <code>errno</code> 的值,字符串值是对应的系统错误字符串
<code>\$OS_ERROR</code> or <code>\$ERRNO</code>	
<code>\$@</code>	命令 <code>eval</code> 的错误消息.如果为空,则表示上一次 <code>eval</code> 命令执行成功
<code>\$EVAL_ERROR</code>	
<code>\$\$</code>	运行当前Perl脚本程序的进程号
<code>\$PROCESS_ID</code> or <code>\$PID</code>	
<code>\$&lt;</code>	当前进程的实际用户号
<code>\$REAL_USER_ID</code> or <code>\$UID</code>	
<code>\$&gt;</code>	当前进程的有效用户号
<code>\$EFFECTIVE_USER_ID</code> or <code>\$EUID</code>	
<code>\$(</code>	当前进程的实际组用户号
<code>\$REAL_GROUP_ID</code> or <code>\$GID</code>	
<code>\$)</code>	当前进程的有效组用户号
<code>\$EFFECTIVE_GROUP_ID</code> or <code>\$EGID</code>	
<code>\$0</code>	包含正在执行的脚本的文件名
<code>\$PROGRAM_NAME</code>	
<code>\$[</code>	数组的数组第一个元素的下标,默认是 0。
<code>\$]</code>	Perl的版本号
<code>\$PERL_VERSION</code>	
<code>^D</code>	调试标志的值
<code>\$DEBUGGING</code>	
<code>^E</code>	在非UNIX环境中的操作系统扩展错误信息
<code>\$EXTENDED_OS_ERROR</code>	
<code>^F</code>	最大的文件描述符数值
<code>\$SYSTEM_FD_MAX</code>	

\$^H	由编译器激活的语法检查状态
\$^I	内置控制编辑器的值
\$INPLACE_EDIT	
\$^M	备用内存池的大小
\$^O	操作系统名
\$OSNAME	
\$^P	指定当前调试值的内部变量
\$PERLDB	
\$^T	从新世纪开始算起,脚本本以秒计算的开始运行的时间
\$BASETIME	
\$^W	警告开关的当前值
\$WARNING	
\$^X	Perl二进制可执行代码的名字
\$EXECUTABLE_NAME	
\$ARGV	从默认的文件句柄中读取时的当前文件名

全局数组特殊变量

@ARGV	传给脚本的命令行参数列表
@INC	在导入模块时需要搜索的目录列表
@F	命令行的数组输入

全局哈希特殊变量

%INC	散列表%INC包含所有用do或require语句包含的文件.关键字是文件名,值是这个文件的路径
%ENV	包含当前环境变量
%SIG	信号列表及其处理方式

全局特殊文件句柄

ARGV	遍历数组变量@ARGV中的所有文件名的特殊文件句柄
STDERR	标准错误输出句柄
STDIN	标准输入句柄
STDOUT	标准输出句柄
DATA	特殊文件句柄引用了在文件中 __END__ 标志后的任何内容包含脚本内容。或者引用一个包含文件中 __DATA__ 标志后的所有内容，只要你在同一个包有读取数据，__DATA__ 就存在。
_ (下划线)	特殊的文件句柄用于缓存文件信息(fstat、stat和lstat)。

全局特殊常量

__END__	脚本的逻辑结束，忽略后面的文本。
__FILE__	当前文件名
__LINE__	当前行号
__PACKAGE__	当前包名，默认的包名是main。

正则表达式特殊变量

\$n	包含上次模式匹配的第n个子串
\$&	前一次成功模式匹配的字符串
\$MATCH	
\$`	前次匹配成功的子串之前的内容
\$PREMATCH	
\$'	前次匹配成功的子串之后的内容
\$POSTMATCH	
\$+	与上个正则表达式搜索格式匹配的最后一个括号。例如：
\$LAST_PAREN_MATCH	<div>/Version: (.*) Revision: (.*)/ &amp;&amp; (\$rev = \$+);</div>

文件句柄特殊变量

\$	如果设置为零,在每次调用函数write或print后，自动调用函数flush,将所写内容写回文件
\$OUTPUT_AUTOFLUSH	
\$%	当前输出页号
\$FORMAT_PAGE_NUMBER	
\$=	当前每页长度。默认为 60。
\$FORMAT_LINES_PER_PAGE	
\$-	当前页剩余的行数
\$FORMAT_LINES_LEFT	
\$~	当前报表输出格式的名称。默认值是文件句柄名。
\$FORMAT_NAME	
\$^	当前报表输出表头格式的名称。默认值是带后缀"_TOP"的文件句柄名。
\$FORMAT_TOP_NAME	

[首页](#) [HTML](#) [CSS](#) [JS](#) [本地书签](#)[Perl 特殊变量](#)[Perl 发送邮件](#)

## Perl 正则表达式

正则表达式(regular expression)描述了一种字符串匹配的模式，可以用来检查一个串是否含有某种子串、将匹配的子串做替换或者从某个串中取出符合某个条件的子串等。

Perl语言的正则表达式功能非常强大，基本上是常用语言中最强大的，很多语言设计正则式支持的时候都参考Perl的正则表达式。

Perl的正则表达式的三种形式，分别是匹配，替换和转化：

匹配：**m//**（还可以简写为**//**，略去**m**）

替换：**s//**

转化：**tr//**

这三种形式一般都和 **=~** 或 **!~** 搭配使用，**=~** 表示相匹配，**!~** 表示不匹配。

## 匹配操作符

匹配操作符 **m//** 用于匹配一个字符串语句或者一个正则表达式，例如，要匹配 标量 **\$bar** 中的 "run",代码如下所示：

### 实例

```
#!/usr/bin/perl
$bar = "I am runoob site. welcome to runoob site.";
if ($bar =~ /run/){
print "第一次匹配\n";
}else{
print "第一次不匹配\n";
}
$bar = "run";
if ($bar =~ /run/){
print "第二次匹配\n";
}else{
print "第二次不匹配\n";
}
```

执行以上程序，输出结果为：

第一次匹配

第二次匹配

## 模式匹配修饰符

模式匹配有一些常用的修饰符，如下表所示：

修饰符	描述
-----	----

i	忽略模式中的大小写
m	多行模式
o	仅赋值一次
s	单行模式，"."匹配"\n"（默认不匹配）
x	忽略模式中的空白
g	全局匹配
cg	全局匹配失败后，允许再次查找匹配串

## 正则表达式变量

perl处理完后会给匹配到的值存在三个特殊变量名：

**\$:** 匹配部分的前一部分字符串

**\$&:** 匹配的字符串

**\$':** 还没有匹配的剩余字符串

如果将这三个变量放在一起,你将得到原始字符串。

实例如下：

### 实例

```
#!/usr/bin/perl
$string = "welcome to runoob site.";
$string =~ m/run/;
print "匹配前的字符串: $\n";
print "匹配的字符串: $&\n";
print "匹配后的字符串: $('n";
```

执行以上程序输出结果为：

匹配前的字符串: welcome to

匹配的字符串: run

匹配后的字符串: oob site.

## 替换操作符

替换操作符 **s///** 是匹配操作符的扩展，使用新的字符串替换指定的字符串。基本格式如下：

```
s/PATTERN/REPLACEMENT/;
```

PATTERN 为匹配模式，REPLACEMENT 为替换的字符串。

例如我们将以下字符串的 "google" 替换为 "runoob"：

### 实例

```
#!/usr/bin/perl
$string = "welcome to google site.";
$string =~ s/google/runoob/;
print "$string\n";
```

执行以上程序输出结果为：

```
welcome to runoob site.
```

### 替换操作修饰符

替换操作修饰符如下表所示：

修饰符	描述
i	如果在修饰符中加上"i", 则正则将会取消大小写敏感性, 即"a"和"A" 是一样的。
m	默认的正则开始"^"和结束"\$"只是对于正则字符串如果在修饰符中加上"m", 那么开始和结束将会指字符串的每一行: 每一行的开头就是"^", 结尾就是"\$"。
o	表达式只执行一次。
s	如果在修饰符中加入"s", 那么默认"."代表除了换行符以外的任何字符将会变成任意字符, 也就是包括换行符!
x	如果加上该修饰符, 表达式中的空白字符将会被忽略, 除非它已经被转义。
g	替换所有匹配的字符串。
e	替换字符串作为表达式

### 转化操作符

以下是转化操作符相关的修饰符：

修饰符	描述
c	转化所有未指定字符
d	删除所有指定字符
s	把多个相同的输出字符缩成一个

以下实例将变量 `$string` 中的所有小写字母转化为大写字母：

```
#!/usr/bin/perl

$string = 'welcome to runoob site.';

$string =~ tr/a-z/A-z/;

print "$string\n";
```

执行以上程序输出结果为：

```
WELCOME TO RUNOOB SITE.
```

以下实例使用 `/s` 将变量 `$string` 重复的字符删除：



## 实例

```
#!/usr/bin/perl
$string = 'runoob';
$string =~ tr/a-z/a-z/s;
print "$string\n";
```

执行以上程序输出结果为：

```
runob
```

更多实例：

```
$string =~ tr/\d/ /c;      # 把所有非数字字符替换为空格

$string =~ tr/\t //d;      # 删除tab和空格

$string =~ tr/0-9/ /cs     # 把数字间的其它字符替换为一个空格。
```

## 更多正则表达式规则

表达式	描述
.	匹配除换行符以外的所有字符
x?	匹配 0 次或一次 x 字符串
x*	匹配 0 次或多次 x 字符串,但匹配可能的最少次数
x+	匹配 1 次或多次 x 字符串,但匹配可能的最少次数
.*	匹配 0 次或多次的任何字符
.+	匹配 1 次或多次的任何字符
{m}	匹配刚好是 m 个 的指定字符串
{m,n}	匹配在 m个 以上 n个 以下 的指定字符串
{m,}	匹配 m个 以上 的指定字符串
[]	匹配符合 [] 内的字符
[^]	匹配不符合 [] 内的字符
[0-9]	匹配所有数字字符
[a-z]	匹配所有小写字母字符
[^0-9]	匹配所有非数字字符
[^a-z]	匹配所有非小写字母字符
^	匹配字符开头的字符
\$	匹配字符结尾的字符
\d	匹配一个数字的字符,和 [0-9] 语法一样

\d+	匹配多个数字字符串,和 [0-9]+ 语法一样
\D	非数字,其他同 \d
\D+	非数字,其他同 \d+
\w	英文字母或数字的字符串,和 [a-zA-Z0-9_] 语法一样
\w+	和 [a-zA-Z0-9_]+ 语法一样
\W	非英文字母或数字的字符串,和 [^a-zA-Z0-9_] 语法一样
\W+	和 [^a-zA-Z0-9_]+ 语法一样
\s	空格,和 [\n\t\r\f] 语法一样
\s+	和 [\n\t\r\f]+ 一样
\S	非空格,和 [^\n\t\r\f] 语法一样
\S+	和 [^\n\t\r\f]+ 语法一样
\b	匹配以英文字母,数字为边界的字符串
\B	匹配不以英文字母,数值为边界的字符串
a b c	匹配符合a字符 或是b字符 或是c字符 的字符串
abc	匹配含有 abc 的字符串 (pattern) () 这个符号会记住所找寻到的字符串,是一个很实用的语法.第一个 () 内所找到的字符串变成 \$1 这个变量或是 \1 变量,第二个 () 内所找到的字符串变成 \$2 这个变量或是 \2 变量,以此类推下去.
/pattern/i	i 这个参数表示忽略英文大小写,也就是在匹配字符串的时候,不考虑英文的大小写问题. \ 如果要在 pattern 模式中找寻一个特殊字符,如 "*",则要在该字符前加上 \ 符号,这样才会让特殊字符失效

[☐ Perl 特殊变量](#)

[Perl 发送邮件](#) ☐

[☐ 点我分享笔记](#)

[反馈/建议](#)

Copyright © 2013-2018 菜鸟教程 [runoob.com](#) All Rights Reserved. 备案号：闽ICP备15012807号-1



[首页](#) [HTML](#) [CSS](#) [JS](#) [本地书签](#)

[☐ Perl 正则表达式](#)

[Perl Socket 编程](#) ☐

## Perl 发送邮件

如果你的程序在 **Linux/Unix** 系统上运行，你就可以在 **Perl** 中使用 **sendmail** 工具来发送邮件。

以下是一个简单的脚本实例用于发送邮件：

## 实例

```
#!/usr/bin/perl
# 接收邮箱, 这里我设置为我的 QQ 邮箱, 你需要修改它为你自己的邮箱
$to = '429240967@qq.com';
#发送者邮箱
$from = 'test@runoob.com';
#标题
$subject = '菜鸟教程 Perl 发送邮件测试';
$message = '这是一封使用 Perl 发送的邮件。';
open(MAIL, "|/usr/sbin/sendmail -t");
# 邮件头部
print MAIL "To: $to\n";
print MAIL "From: $from\n";
print MAIL "Subject: $subject\n\n";
# 邮箱信息
print MAIL $message;
close(MAIL);
print "邮件发送成功\n";
```

执行以上程序, 输出结果为:

邮件发送成功

正常情况下, 以上邮件会被 QQ 邮箱拦截, 我么可以把它加入白名单, 操作方式可以点击: <https://kf.qq.com/faq/120322fu63YV130805rYRFzu.html>

加入白名单后就可以正常接收邮件了。

### 菜鸟教程 Perl 发送邮件测试 ☆

发件人: **test** <test@runoob.com>   
(由 tianqixin@tianqixindeMacBook-Pro.local 代发)   
时 间: 2016年6月16日(星期四) 下午2:03  
收件人:  <429240967@qq.com>

这是一封使用 Perl 发送的邮件。

## 发送 HTML 格式邮件

我们可以在邮件头部添加 **Content-type: text/html** 来发送 HTML 格式的邮件, 实例如下:


## 实例

```
#!/usr/bin/perl
# 接收邮箱, 这里我设置为我的 QQ 邮箱, 你需要修改它为你自己的邮箱
$to = '429240967@qq.com';
#发送者邮箱
$from = 'test@runoob.com';
#标题
$subject = '菜鸟教程 Perl 发送邮件测试';
$message = '<h1>这是一封使用 Perl 发送的邮件<h1><p>你好, 我来自菜鸟教程, 地址是:http://www.runoob.com。</p>';
open(MAIL, "|/usr/sbin/sendmail -t");
# 邮件头部
print MAIL "To: $to\n";
print MAIL "From: $from\n";
print MAIL "Subject: $subject\n";
print MAIL "Content-type: text/html\n";
# 邮箱信息
print MAIL $message;
close(MAIL);
```

```
print "邮件发送成功\n";
```

执行成功后，查看邮件内容，如下所示：

## 菜鸟教程 Perl 发送邮件测试 ☆

发件人: **test** <test@runoob.com>   
(由 tianqixin@tianqixindeMacBook-Pro.local 代发) ?

时 间: 2016年6月16日(星期四) 下午2:03

收件人:  <429240967@qq.com>

这是一封使用 Perl 发送的邮件。

## 使用 MIME::Lite 模块

如果你使用的是 window 系统，没有 sendmail 工具。这时你就可以使用 perl 的 MIME::Lite 模块作为邮件客户端来发送邮件。

MIME::Lite 模块 下载地址为: [MIME-Lite-3.030.tar.gz](http://www.cpan.org/modules/by-module/MIME/MIME-Lite-3.030.tar.gz)。

这里我们直接用 cpan 来安装(需要 root 权限)，不用下载：

```
$ cpan -i MIME::Lite

.....

/usr/bin/make install -- OK
```

安装成功后，我们来演示一个实例：

## 实例

```
#!/usr/bin/perl
use MIME::Lite;
# 接收邮箱，这里我设置为我的 QQ 邮箱，你需要修改它为你自己的邮箱
$to = '429240967@qq.com';
# 抄送者，多个使用逗号隔开
# $cc = 'test1@runoob.com, test2@runoob.com';
# 发送者邮箱
$from = 'test@runoob.com';
# 标题
$subject = '菜鸟教程 Perl 发送邮件测试';
$message = '这是一封使用 Perl 发送的邮件，使用了 MIME::Lite 模块。';
$msg = MIME::Lite->new(
    From => $from,
    To => $to,
    Cc => $cc,
    Subject => $subject,
    Data => $message
);
$msg->send;
print "邮件发送成功\n";
```

执行成功后，查看邮件内容，如下所示：

## 菜鸟教程 Perl 发送邮件测试 ☆

发件人: **test** <test@runoob.com> 

时 间: 2016年6月16日(星期四) 下午2:46

收件人:  <429240967@qq.com>

这是一封使用 Perl 发送的邮件，使用了 MIME::Lite 模块。

### 发送 HTML 格式邮件

我们可以在邮件头部添加 **Content-type: text/html** 来发送 HTML 格式的邮件，实例如下：

#### 实例

```
#!/usr/bin/perl
use MIME::Lite;
# 接收邮箱，这里我设置为我的 QQ 邮箱，你需要修改它为你自己的邮箱
$to = '429240967@qq.com';
# 抄送者，多个使用逗号隔开
# $cc = 'test1@runoob.com, test2@runoob.com';
# 发送者邮箱
$from = 'test@runoob.com';
# 标题
$subject = '菜鸟教程 Perl 发送邮件测试';
$message = '<h1>这是一封使用 Perl 发送的邮件</h1><p>使用了 MIME::Lite 模块。</p><p>来自菜鸟教程，地址是: http://www.runoob.com.</p>';
$msg = MIME::Lite->new(
    From => $from,
    To => $to,
    Cc => $cc,
    Subject => $subject,
    Data => $message
);
# 添加头部信息
$msg->attr("content-type" => "text/html");
$msg->send;
print "邮件发送成功\n";
```

执行成功后，查看邮件内容，如下所示：

#### 菜鸟教程 Perl 发送邮件测试 ☆

发件人: **test** <test@runoob.com> 

时 间: 2016年6月16日(星期四) 下午2:49

收件人:  <429240967@qq.com>

这是一封使用 Perl 发送的邮件

使用了 MIME::Lite 模块。

来自菜鸟教程，地址是: <http://www.runoob.com>。

### 发送带有附件的邮件

发送带有附件的邮件实例如下：

#### 实例

```
#!/usr/bin/perl
use MIME::Lite;
# 接收邮箱，这里我设置为我的 QQ 邮箱，你需要修改它为你自己的邮箱
$to = '429240967@qq.com';
# 抄送者，多个使用逗号隔开
# $cc = 'test1@runoob.com, test2@runoob.com';
# 发送者邮箱
$from = 'test@runoob.com';
```

```
#标题
$subject = '菜鸟教程 Perl 发送邮件测试';
$message = '这是一封使用 Perl 发送的邮件，使用了 MIME::Lite 模块，包含了附件。';
$msg = MIME::Lite->new(
    From => $from,
    To => $to,
    Cc => $cc,
    Subject => $subject,
    Type => 'multipart/mixed' # 附件标记
);
$msg->attach (
    Type => 'TEXT',
    Data => $message
);# 指定附件信息
$msg->attach(Type => 'TEXT',
    Path => './runoob.txt', # 当前目录下
    Filename => 'runoob.txt',
    Disposition => 'attachment'
);
$msg->send;
print "邮件发送成功\n";
```

执行成功后，查看邮件内容，如下所示：

菜鸟教程 Perl 发送邮件测试 ☆

发件人：test <test@runoob.com>

时 间：2016年6月16日(星期四) 下午3:11

收件人：<429240967@qq.com>

附 件：1 个 (runoob.txt)

这是一封使用 Perl 发送的邮件，使用了 MIME::Lite 模块，包含了附件。

附件(1 个)

普通附件



runoob.txt (14字节)

下载 预览 收藏 转存

附件

你可以通过使用多个 \$msg->attach 来添加多个附件。

□ Perl 正则表达式

Perl Socket 编程 □

□ 点我分享笔记

反馈/建议

Copyright © 2013-2018 菜鸟教程 runoob.com All Rights Reserved. 备案号：闽ICP备15012807号-1

首页 HTML CSS JS 本地书签

□ Perl 发送邮件

Perl 面向对象 □

## Perl Socket 编程

Socket又称"套接字"，应用程序通常通过"套接字"向网络发出请求或者应答网络请求，使主机间或者一台计算机上的进程间可以通讯。

本章节我们为大家接收 Perl 语言中如何使用 Socket 服务。

### 创建服务端

使用 **socket** 函数来创建 **socket**服务。

使用 **bind** 函数绑定端口。

使用 **listen** 函数监听端口。

使用 **accept** 函数接收客户端请求。

### 创建客户端

使用 **socket** 函数来创建 **socket** 服务。

使用 **connect** 函数连接到 **socket** 服务端。

以下图表演示了客户端与服务端之间的通信流程：

□

## 服务端 socket 函数

### socket 函数

Perl 中，我们用 **socket**（）函数来创建套接字，语法格式如下：

```
socket( SOCKET, DOMAIN, TYPE, PROTOCOL );
```

参数解析：

**DOMAIN** 创建的套接字指定协议集。 例如：

**AF\_INET** 表示IPv4网络协议

**AF\_INET6** 表示IPv6

**AF\_UNIX** 表示本地套接字（使用一个文件）

**TYPE** 套接字类型可以根据是面向连接的还是非连接分为**SOCK\_STREAM**或**SOCK\_DGRAM**

**PROTOCOL** 应该是 **(getprotobyname("tcp"))[2]**。指定实际使用的传输协议。

所以 **socket** 函数调用方式如下：

```
use Socket      # 定义了 PF_INET 和 SOCK_STREAM

socket(SOCKET,PF_INET,SOCK_STREAM,(getprotobyname('tcp'))[2]);
```

### bind() 函数

使用 **bind()** 为套接字分配一个地址：

```
bind( SOCKET, ADDRESS );
```

**SOCKET** 一个socket的描述符。 **ADDRESS** 是 **socket** 地址 ( TCP/IP ) 包含了三个元素：

地址簇 (TCP/IP, 是 AF\_INET, 在你系统上可能是 2)

端口号 (例如 21)

网络地址 (例如 10.12.12.168)

使用`socket()`创建套接字后，只赋予其所使用的协议，并未分配地址。在接受其它主机的连接前，必须先调用`bind()`为套接字分配一个地址。  
简单实例如下：

```
use Socket          # 定义了 PF_INET 和 SOCK_STREAM

$port = 12345;      # 监听的端口

$server_ip_address = "10.12.12.168";

bind( SOCKET, pack_sockaddr_in($port, inet_aton($server_ip_address)))

or die "无法绑定端口! \n";
```

**or die** 在绑定地址失败后执行。

通过设置 `setsockopt()` 可选项 `SO_REUSEADDR` 设置端口可立即重复使用。

**pack\_sockaddr\_in()** 函数将地址转换为二进制格式。

## listen() 函数

当`socket`和一个地址绑定之后，`listen()`函数会开始监听可能的连接请求。然而，这只能在有可靠数据流保证的时候使用：

```
listen( SOCKET, QUEUESIZE );
```

**SOCKET**：一个`socket`的描述符。

**QUEUESIZE**：是一个决定监听队列大小的整数，当有一个连接请求到来，就会进入此监听队列；当一个连接请求被`accept()`接受，则从监听队列中移出；当队列满后，新的连接请求会返回错误。

一旦连接被接受，返回0表示成功，错误返回-1。

## accept() 函数

`accept()` 函数接受请求的`socket`连接。如果成功则返回压缩形式的网络地址，否则返回**FALSE**：

```
accept( NEW_SOCKET, SOCKET );
```

**NEW\_SOCKET**：一个`socket`的描述符。

**SOCKET**：一个`socket`的描述符。

`accept()` 通常应用在无限循环当中：

```
while(1) {

    accept( NEW_SOCKET, SOCKT );

    .....

}
```



以上实例可以实时监听客户端的请求。

## 客户端函数

### connect() 函数

connect()系统调用为一个套接字设置连接，参数有文件描述符和主机地址。

```
connect( SOCKET, ADDRESS );
```

以下创建一个连接到服务端 **socket** 的实例：

```
$port = 21;      # ftp 端口

$server_ip_address = "10.12.12.168";

connect( SOCKET, pack_sockaddr_in($port, inet_aton($server_ip_address)))

    or die "无法绑定端口! \n";
```

## 完整实例

接下来我们通过一个完整实例来了解下所有 **socket** 函数的应用：

服务端 **server.pl** 代码：

### 实例

```
#!/usr/bin/perl -w
# Filename : server.pl
use strict;
use Socket;

# 使用端口 7890 作为默认值
my $port = shift || 7890;
my $proto = getprotobyname('tcp');
my $server = "localhost"; # 设置本地地址
# 创建 socket, 端口可重复使用, 创建多个连接
socket(SOCKET, PF_INET, SOCK_STREAM, $proto)
or die "无法打开 socket $!\n";
setsockopt(SOCKET, SOL_SOCKET, SO_REUSEADDR, 1)
or die "无法设置 SO_REUSEADDR $!\n";
# 绑定端口并监听
bind( SOCKET, pack_sockaddr_in($port, inet_aton($server)))
or die "无法绑定端口 $port! \n";
listen(SOCKET, 5) or die "listen: $!";
print "访问启动: $port\n";
# 接收请求
my $client_addr;
while ($client_addr = accept(NEW_SOCKET, SOCKET)) {
    # send them a message, close connection
    my $name = gethostbyaddr($client_addr, AF_INET );
    print NEW_SOCKET "我是来自服务端的信息";
    print "Connection recieved from $name\n";
    close NEW_SOCKET;
}
```

打开一个终端，执行以下代码：

```
$ perl sever.pl
```

访问启动：7890

客户端 client.pl 代码:

## 实例

```
#!/usr/bin/perl -w
# Filename : client.pl
use strict;
use Socket;
# 初始化地址与端口
my $host = shift || 'localhost';
my $port = shift || 7890;
my $server = "localhost"; # 主机地址
# 创建 socket 并连接
socket(SOCKET,PF_INET,SOCK_STREAM,(getprotobyname('tcp'))[2])
or die "无法创建 socket $!\n";
connect( SOCKET, pack_sockaddr_in($port, inet_aton($server)))
or die "无法连接: port $port! \n";
my $line;
while ($line = <SOCKET>) {
    print "$line\n";
}
close SOCKET or die "close: $!";
```

打开另外一个终端，执行以下代码:

```
$ perl client.pl
```

我是来自服务端的信息

[Perl 发送邮件](#)

[Perl 面向对象](#)

[点我分享笔记](#)

[反馈/建议](#)



[Perl Socket 编程](#)

[Perl 数据库连接](#)

# Perl 面向对象

Perl 中有两种不同地面向对象编程的实现:

一是基于匿名哈希表的方式，每个对象实例的实质就是一个指向匿名哈希表的引用。在这个匿名哈希表中，存储来所有的实例属性。

二是基于数组的方式，在定义一个类的时候，我们将为每一个实例属性创建一个数组，而每一个对象实例的实质就是一个指向这些数组中某一行索引的引用。在这些数组中，存储着所有的实例属性。

## 面向对象基础概念

面向对象有很多基础概念，这里我们接收三个：对象、类和方法。

**对象**：对象是对类中数据项的引用。。

**类**：类是个Perl包，其中含提供对象方法的类。

**方法**：方法是个Perl子程序，类名是其第一个参数。

Perl 提供了 `bless()` 函数，`bless` 是用来构造对象的，通过 `bless` 把一个引用和这个类名相关联，返回这个引用就构造出一个对象。

## 类的定义

一个类只是一个简单的包。

可以把一个包当作一个类用，并且把包里的函数当作类的方法来用。

Perl 的包提供了独立的命名空间，所以不同包的方法与变量名不会冲突。

Perl 类的文件后缀为 `.pm`。

接下来我们创建一个 `Person` 类：

```
package Person;
```

类的代码范围到脚本文件的最后一行，或者到下一个 `package` 关键字前。

## 创建和使用对象

创建一个类的实例 (对象) 我们需要定义一个构造函数，大多数程序使用类名作为构造函数，Perl 中可以使用任何名字。

你可以使用多种 Perl 的变量作为 Perl 的对象。大多数情况下我们会使用引用数组或哈希。

接下来我们为 `Person` 类创建一个构造函数，使用了 Perl 的哈希引用。

在创建对象时，你需要提供一个构造函数，它是一个子程序，返回对象的引用。

实例如下：

### 实例

```
package Person;
sub new
{
    my $class = shift;
    my $self = {
        _firstName => shift,
        _lastName => shift,
        _ssn => shift,
    };
    # 输出用户信息
    print "名字: $self->{_firstName}\n";
    print "姓氏: $self->{_lastName}\n";
    print "编号: $self->{_ssn}\n";
    bless $self, $class;
    return $self;
}
```

接下来我们创建一个对象：

```
$object = new Person( "小明", "王", 23234345);
```

## 定义方法

Perl类的方法只不过是Perl子程序而已，也即通常所说的成员函数。

Perl面向对象中Perl的方法定义不提供任何特别语法，但规定方法的第一个参数为对象或其被引用的包。

Perl 没有提供私有变量，但我们可以通过辅助的方式来管理对象数据。

接下来我们定义一个获取名字的方法：

```
sub getFirstName {  
  
    return $self->{_firstName};  
  
}
```

同样也可以这么写：

```
sub setFirstName {  
  
    my ( $self, $firstName ) = @_;  
  
    $self->{_firstName} = $firstName if defined($firstName);  
  
    return $self->{_firstName};  
  
}
```

接下来我们修改 `Person.pm` 文件的代码，如下所示：

## 实例

```
#!/usr/bin/perl  
package Person;  
sub new  
{  
    my $class = shift;  
    my $self = {  
        _firstName => shift,  
        _lastName => shift,  
        _ssn => shift,  
    };  
    # 输出用户信息  
    print "名字: $self->{_firstName}\n";  
    print "姓氏: $self->{_lastName}\n";  
    print "编号: $self->{_ssn}\n";  
    bless $self, $class;  
    return $self;  
}  
sub setFirstName {  
    my ( $self, $firstName ) = @_;  
    $self->{_firstName} = $firstName if defined($firstName);  
    return $self->{_firstName};  
}  
sub getFirstName {  
    my( $self ) = @_;  
    return $self->{_firstName};  
}  
1;
```

`employee.pl` 脚本代码如下：

## 实例

```
#!/usr/bin/perl  
use Person;  
$object = new Person( "小明", "王", 23234345);  
# 获取姓名  
$firstName = $object->getFirstName();  
print "设置前姓名为 : $firstName\n";
```

```
# 使用辅助函数设置姓名
$object->setFirstName( "小强" );
# 通过辅助函数获取姓名
$firstName = $object->getFirstName();
print "设置后姓名为 : $firstName\n";
```

执行以上程序后，输出结果为：

```
$ perl employee.pl

名字: 小明

姓氏: 王

编号: 23234345

设置前姓名为 : 小明

设置后姓名为 : 小强
```

## 继承

Perl 里 类方法通过@ISA数组继承，这个数组里面包含其他包（类）的名字，变量的继承必须明确设定。

多继承就是这个@ISA数组包含多个类（包）名字。

通过@ISA只能继承方法，不能继承数据。

接下来我们创建一个 Employee 类继承 Person 类。

Employee.pm 文件代码如下所示：

### 实例

```
#!/usr/bin/perl
package Employee;
use Person;
use strict;
our @ISA = qw(Person); # 从 Person 继承
```

现在 Employee 类包含了 Person 类的所有方法和属性，我们在 main.pl 文件中输入以下代码，并执行：

### 实例

```
#!/usr/bin/perl
use Employee;
$object = new Employee( "小明", "王", 23234345);
# 获取姓名
$firstName = $object->getFirstName();
print "设置前姓名为 : $firstName\n";
# 使用辅助函数设置姓名
$object->setFirstName( "小强" );
# 通过辅助函数获取姓名
$firstName = $object->getFirstName();
print "设置后姓名为 : $firstName\n";
```

执行以上程序后，输出结果为：

```
$ perl main.pl

名字: 小明

姓氏: 王

编号: 23234345
```

设置前姓名为 ： 小明

设置后姓名为 ： 小强

## 方法重写

上面实例中，**Employee** 类继承了 **Person** 类，但如果 **Person** 类的方法无法满足需求，就需要对其方法进行重写。

接下来我们在 **Employee** 类中添加一些新方法，并重写了 **Person** 类的方法：

### 实例

```
#!/usr/bin/perl
package Employee;
use Person;
use strict;
our @ISA = qw(Person); # 从 Person 继承
# 重写构造函数
sub new {
    my ($class) = @_ ;
    # 调用父类的构造函数
    my $self = $class->SUPER::new( $_[1], $_[2], $_[3] );
    # 添加更多属性
    $self->{_id} = undef;
    $self->{_title} = undef;
    bless $self, $class;
    return $self;
}
# 重写方法
sub getFirstName {
    my( $self ) = @_ ;
    # 这是子类函数
    print "这是子类函数\n";
    return $self->{_firstName};
}
# 添加方法
sub setLastName{
    my ( $self, $lastName ) = @_ ;
    $self->{_lastName} = $lastName if defined($lastName);
    return $self->{_lastName};
}
sub getLastName {
    my( $self ) = @_ ;
    return $self->{_lastName};
}
1;
```

我们在 **main.pl** 文件中输入以下代码，并执行：

### 实例

```
#!/usr/bin/perl
use Employee;
$object = new Employee( "小明", "王", 23234345);
# 获取姓名，使用修改后的构造函数
$firstName = $object->getFirstName();
print "设置前姓名为 ： $firstName\n";
# 使用辅助函数设置姓名
$object->setFirstName( "小强" );
# 通过辅助函数获取姓名
$firstName = $object->getFirstName();
print "设置后姓名为 ： $firstName\n";
```

执行以上程序后，输出结果为：

```
$ perl main.pl
```

名字: 小明

姓氏：王

编号：23234345

这是子类函数

设置前姓名为 ： 小明

这是子类函数

设置后姓名为 ： 小强

## 默认载入

如果在当前类、当前类所有的基类、还有 **UNIVERSAL** 类中都找不到请求的方法，这时会再次查找名为 **AUTOLOAD()** 的一个方法。如果找到了 **AUTOLOAD**，那么就会 调用，同时设定全局变量 **\$AUTOLOAD** 的值为缺失的方法的全限定名称。

如果还不行，那么 **Perl** 就宣告失败并出错。

如果你不想继承基类的 **AUTOLOAD**，很简单，只需要一句：

```
sub AUTOLOAD;
```

## 析构函数及垃圾回收

当对象的最后一个引用释放时，对象会自动析构。

如果你想在析构的时候做些什么，那么你可以在类中定义一个名为"**DESTROY**"的方法。它将在适合的时机自动调用，并且按照你的意思执行额外的清理动作。

```
package MyClass;

...

sub DESTROY

{

    print "MyClass::DESTROY called\n";

}
```

**Perl** 会把对象的引用作为 唯一的参数传递给 **DESTROY**。注意这个引用是只读的，也就是说你 cannot 通过访问 **\$\_[0]** 来修改它。（译者注：参见 **perlsub**）但是对象自身（比如 **"\${\$\_[0]}"** 或者 **"@{\$\_[0]}"** 还有 **"%{\$\_[0]}"** 等等）还是可写的。

如果你在析构器返回之前重新 **bless** 了对象引用，那么 **Perl** 会在析构器返回之后接着调用你重新 **bless** 的那个对象的 **DESTROY** 方法。这可以让你有机会调用基类或者你指定的其它类的析构器。需要说明的是，**DESTROY** 也可以手工调用，但是通常没有必要这么做。

在当前对象释放后，包含在当前对象中的其它对象会自动释放。

## Perl 面向对象实例

我们可以通过以下实例来进一步理解**Perl**面向对象的应用：

### 实例

```
#!/usr/bin/perl
# 下面是简单的类实现
package MyClass;
sub new
```

```

{
print "MyClass::new called\n";
my $type = shift; # 包名
my $self = {}; # 引用空哈希
return bless $self, $type;
}
sub DESTROY
{
print "MyClass::DESTROY called\n";
}
sub MyMethod
{
print "MyClass::MyMethod called!\n";
}
# 继承实现
package MySubClass;
@ISA = qw( MyClass );
sub new
{
print "MySubClass::new called\n";
my $type = shift; # 包名
my $self = MyClass->new; # 引用空哈希
return bless $self, $type;
}
sub DESTROY
{
print "MySubClass::DESTROY called\n";
}
sub MyMethod
{
my $self = shift;
$self->SUPER::MyMethod();
print " MySubClass::MyMethod called!\n";
}
# 调用以上类的主程序
package main;
print "调用 MyClass 方法\n";
$myObject = MyClass->new();
$myObject->MyMethod();
print "调用 MySubClass 方法\n";
$myObject2 = MySubClass->new();
$myObject2->MyMethod();
print "创建一个作用域对象\n";
{
my $myObject2 = MyClass->new();
}
# 自动调用析构函数
print "创建对象\n";
$myObject3 = MyClass->new();
undef $myObject3;
print "脚本执行结束...\n";
# 自动执行析构函数

```

执行以上程序，输出结果为：

```

调用 MyClass 方法

MyClass::new called

MyClass::MyMethod called!

调用 MySubClass 方法

MySubClass::new called

MyClass::new called

MyClass::MyMethod called!

```



```
MySubClass::MyMethod called!
```

创建一个作用域对象

```
MyClass::new called
```

```
MyClass::DESTROY called
```

创建对象

```
MyClass::new called
```

```
MyClass::DESTROY called
```

脚本执行结束...

```
MyClass::DESTROY called
```

```
MySubClass::DESTROY called
```

[Perl Socket 编程](#)

[Perl 数据库连接](#)

[点我分享笔记](#)

[反馈/建议](#)

Copyright © 2013-2018 菜鸟教程 [runoob.com](#) All Rights Reserved. 备案号: 闽ICP备15012807号-1



[首页](#) [HTML](#) [CSS](#) [JS](#) [本地书签](#)

[Perl 面向对象](#)

[Perl CGI编程](#)

## Perl 数据库连接

本章节我们将为大家介绍 **Perl** 数据库的连接。

**Perl 5** 中我们可以使用 **DBI** 模块来连接数据库。

**DBI** 英文全称: **Database Independent Interface**, 中文称为数据库独立接口。

**DBI** 作为 **Perl** 语言中和数据库进行通讯的标准接口, 它定义了一系列的方法, 变量和常量, 提供一个和具体数据库平台无关的数据库持久层。

### DBI 结构

**DBI** 和具体数据库平台无关, 我们可以将其应用在 **Oracle**, **MySQL** 或 **Informix**, 等数据库中。

□

图表中 **DBI** 获取所有 **API** (**Application Programming Interface**: 应用程序接口) 发送过来的 **SQL** 数据, 然后分发到对应的驱动上执行, 最后再获取数据返回。

### 变量名约定

以下设置了比较常用的变量命名方法:

<code>\$dsn</code>	驱动程序对象的句柄
<code>\$dbh</code>	一个数据库对象的句柄
<code>\$sth</code>	一个语句或者一个查询对象的句柄
<code>\$h</code>	通用的句柄 ( <code>\$dbh</code> , <code>\$sth</code> , 或 <code>\$drh</code> ), 依赖于上下文
<code>\$rc</code>	操作代码返回的布什值 ( <code>true</code> 或 <code>false</code> )
<code>\$rv</code>	操作代码返回的整数
<code>@ary</code>	查询返回的一行值的数组 (列表)
<code>\$rows</code>	操作代码返回的行数值
<code>\$fh</code>	文件句柄
<code>undef</code>	NULL 值表示未定义
<code>\%attr</code>	引用属性的哈希值并传到方法上

## 数据库连接

接下来我们以 MySQL 数据库为例演示 Perl 是如何对数据库进行操作的。

这里我们在 MySQL 数据库创建 RUNOOB 数据库, 数据表为 Websites, 表结构及数据如下图所示:

id	name	url	alexa	country
1	Google	<a href="https://www.google.cm/">https://www.google.cm/</a>	1	USA
2	淘宝	<a href="https://www.taobao.com/">https://www.taobao.com/</a>	13	CN
3	菜鸟教程	<a href="http://www.runoob.com/">http://www.runoob.com/</a>	5000	USA
4	微博	<a href="http://weibo.com/">http://weibo.com/</a>	20	CN
5	Facebook	<a href="https://www.facebook.com/">https://www.facebook.com/</a>	3	USA
7	stackoverflow	<a href="http://stackoverflow.com/">http://stackoverflow.com/</a>	0	IND

下载该数据表: [http://static.runoob.com/download/websites\\_perl.sql](http://static.runoob.com/download/websites_perl.sql)

接下来我们使用以下代码来连接数据库:

## 实例

```
#!/usr/bin/perl -w
use strict;
use DBI;
my $host = "localhost"; # 主机地址
my $driver = "mysql"; # 接口类型 默认为 localhost
my $database = "RUNOOB"; # 数据库
# 驱动程序对象的句柄
my $dsn = "DBI:$driver:database=$database:$host";
my $userid = "root"; # 数据库用户名
my $password = "123456"; # 数据库密码
# 连接数据库
my $dbh = DBI->connect($dsn, $userid, $password) or die $DBI::errstr;
my $sth = $dbh->prepare("SELECT * FROM Websites"); # 预处理 SQL 语句
$sth->execute(); # 执行 SQL 操作
# 注释这部分使用的是绑定值操作
# $alexa = 20;
# my $sth = $dbh->prepare("SELECT name, url
# FROM Websites
# WHERE alexa > ?");
# $sth->execute( $alexa ) or die $DBI::errstr;
# 循环输出所有数据
while ( my @row = $sth->fetchrow_array() )
{
    print join('\t', @row)."\n";
}
```

```
}  
$sth->finish();  
$dbh->disconnect();
```

## 插入操作

执行步骤:

使用 `prepare()` API 预处理 SQL 语句。

使用 `execute()` API 执行 SQL 语句。

使用 `finish()` API 释放语句句柄。

最后如果一切顺利就会提交以上执行操作。

```
my $sth = $dbh->prepare("INSERT INTO Websites  
  
                        (name, url, alexa, conutry )  
  
                        values  
  
                        ('Twitter', 'https://twitter.com/', 10, 'USA')");  
  
$sth->execute() or die $DBI::errstr;  
  
$sth->finish();  
  
$dbh->commit or die $DBI::errstr;
```

应用程序还可以绑定输出和输入参数,下面例子通过用变量 取代 ? 占位符的位置来执行一条插入查询:

```
my $name = "Twitter";  
  
my $url = "https://twitter.com/";  
  
my $alexa = 10;  
  
my $conutry = "USA";  
  
my $sth = $dbh->prepare("INSERT INTO Websites  
  
                        (name, url, alexa, conutry )  
  
                        values  
  
                        (?, ?, ?, ?)");  
  
$sth->execute($name,$url,$alexa, $conutry)  
  
                        or die $DBI::errstr;  
  
$sth->finish();  
  
$dbh->commit or die $DBI::errstr;
```

## 更新操作

执行步骤:

使用 `prepare()` API 预处理 SQL 语句。

使用 `execute()` API 执行 SQL 语句。

使用 `finish()` API 释放语句句柄。

最后如果一切顺利就会提交以上执行操作。

```
my $sth = $dbh->prepare("UPDATE Websites  
  
    SET    alexa = alexa + 1  
  
    WHERE country = 'CN'");  
  
$sth->execute() or die $DBI::errstr;  
  
print "更新的记录数 :" + $sth->rows;  
  
$sth->finish();  
  
$dbh->commit or die $DBI::errstr;
```

应用程序还可以绑定输出和输入参数,下面例子通过用变量取代 ? 占位符的位置来执行一条更新查询:

```
$name = '菜鸟教程';  
  
my $sth = $dbh->prepare("UPDATE Websites  
  
    SET    alexa = alexa + 1  
  
    WHERE name = ?");  
  
$sth->execute($name) or die $DBI::errstr;  
  
print "更新的记录数 :" + $sth->rows;  
  
$sth->finish();
```

当然我们也可以绑定要设置的值,如下所示将 `country` 为 `CN` 的 `alexa` 都修改为 1000:

```
$country = 'CN';  
  
$alexa = 1000; ;  
  
my $sth = $dbh->prepare("UPDATE Websites  
  
    SET    alexa = ?  
  
    WHERE country = ?");  
  
$sth->execute($alexa, $country) or die $DBI::errstr;  
  
print "更新的记录数 :" + $sth->rows;  
  
$sth->finish();
```

## 删除数据

执行步骤:

使用 `prepare()` API 预处理 SQL 语句。

使用 `execute()` API 执行 SQL 语句。

使用 `finish()` API 释放语句句柄。

最后如果一切顺利就会提交以上执行操作。

以下数据将 `Websites` 中 `alexa` 大于 1000 的数据都删除:

```
$alexa = 1000;

my $sth = $dbh->prepare("DELETE FROM Websites

                        WHERE alexa = ?");

$sth->execute( $alexa ) or die $DBI::errstr;

print "删除的记录数 :" + $sth->rows;

$sth->finish();

$dbh->commit or die $DBI::errstr;
```

## 使用 `do` 语句

`do` 语句可以执行 `UPDATE`, `INSERT`, 或 `DELETE` 操作, 使用他比较简短, 执行成功返回`true`, 执行失败返回 `false`, 实例如下:

```
$dbh->do('DELETE FROM Websites WHERE alexa>1000');
```

## COMMIT 操作

`commit` 为提交事务, 完成数据库的操作:

```
$dbh->commit or die $dbh->errstr;
```

## ROLLBACK 操作

如果在 `SQL` 执行过程中发生错误, 可以回滚数据, 不做任何改变:

```
$dbh->rollback or die $dbh->errstr;
```

事务

和其它的语言一样, `perl DBI`对数据库的操作也支持事务处理, 它的实现方式有两个:

### 1、在连接数据库的时候就开始一个事务

```
$dbh = DBI->connect($dsn, $userid, $password, {AutoCommit => 0}) or die $DBI::errstr;
```

以上代码在连接的时候设置了`AutoCommit`为`false`, 也就是说当你数据库进行更新操作的时候, 它不会自动地把那些更新直接写到数据库里, 而是要程序通过 `$dbh->commit` 来使数据真正地写到数据库里, 或 `$dbh->rollback` 来回滚刚才的操作。

## 2、通过\$dbh->begin\_work()语句来开始一个事务

这种方式就不需要在连接数据库的时候设置 `AutoCommit = 0`。

可以一次数据库连接进行多次事务操作，不用每一次事务的开始都去连接一次数据库。

```
$rc = $dbh->begin_work or die $dbh->errstr;
```

```
#####
```

```
##这里执行一些 SQL 操作
```

```
#####
```

```
$dbh->commit;    # 成功后操作
```

```
-----
```

```
$dbh->rollback;  # 失败后回滚
```

## 断开数据库连接

如果我们需要断开数据库连接，可以使用 `disconnect` API:

```
$rc = $dbh->disconnect or warn $dbh->errstr;
```

[☐ Perl 面向对象](#)

[Perl CGI编程](#) ☐

[☐ 点我分享笔记](#)

反馈/建议

Copyright © 2013-2018 菜鸟教程 [runoob.com](#) All Rights Reserved. 备案号: 闽ICP备15012807号-1



[首页](#) [HTML](#) [CSS](#) [JS](#) [本地书签](#)

[☐ Perl 数据库连接](#)

[Perl 包和模块](#) ☐

## Perl CGI编程

### 什么是CGI

CGI 目前由NCSA维护，NCSA定义CGI如下：

CGI(Common Gateway Interface),通用网关接口,它是一段程序,运行在服务器上如：[HTTP服务器](#)，提供同客户端HTML页面的接口。

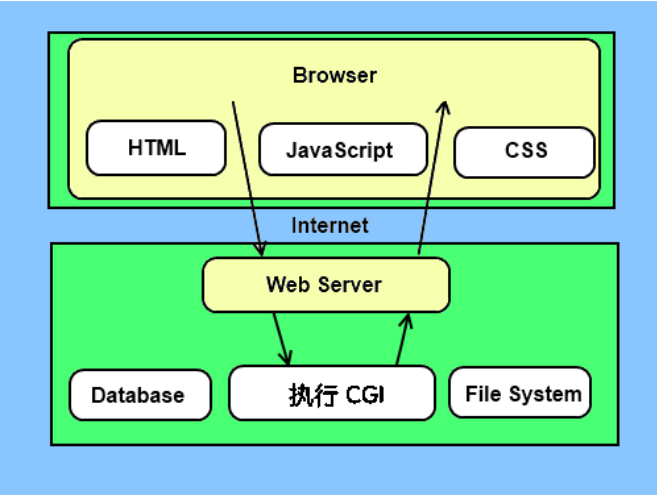
# 网页浏览

为了更好的了解CGI是如何工作的，我们可以从在网页上点击一个链接或URL的流程：

- 1、使用你的浏览器访问URL并连接到HTTP web 服务器。
- 2、Web服务器接收到请求信息后会解析URL，并查找访问的文件在服务器上是否存在，如果存在返回文件的内容，否则返回错误信息。
- 3、浏览器从服务器上接收信息，并显示接收的文件或者错误信息。

CGI程序可以是Python脚本，PERL脚本，SHELL脚本，C或者C++程序等。

## CGI架构图



## Web服务器支持及配置

在你进行CGI编程前，确保您的Web服务器支持CGI及已经配置了CGI的处理程序。

Apache 支持CGI 配置：

设置好CGI目录：

```
ScriptAlias /cgi-bin/ /var/www/cgi-bin/
```

所有的HTTP服务器执行CGI程序都保存在一个预先配置的目录。这个目录被称为CGI目录，并按照惯例，它被命名为/var/www/cgi-bin目录。

CGI文件的扩展名为.cgi，Perl 也可以使用.pl扩展名。

默认情况下，Linux服务器配置运行的cgi-bin目录中为/var/www。

如果你想指定其他运行CGI脚本的目录，可以修改httpd.conf配置文件，如下所示：

```
<Directory "/var/www/cgi-bin">

    AllowOverride None

    Options +ExecCGI

    Order allow,deny

    Allow from all

</Directory>
```

在 AddHandler 中添加 .pl 后缀，这样我们就可以访问 .pl 结尾的 Perl 脚本文件：

```
AddHandler .pl
```

```
AddHandler cgi-script .cgi .pl .py
```

## 第一个 CGI 程序

下面我们创建一个 `test.cgi` 文件，代码如下所示：

test.cgi 代码

```
#!/usr/bin/perl
print "Content-type:text/html\r\n\r\n";
print '<html>';
print '<head>';
print '<meta charset="utf-8">';
print '<title>菜鸟教程(runoob.com)</title>';
print '</head>';
print '<body>';
print '<h2>Hello Word! </h2>';
print '<p>来自菜鸟教程第一个 CGI 程序。</p>';
print '</body>';
print '</html>';
1;
```

然后通过浏览器打开 `http://localhost/cgi-bin/test.cgi`，输出结果如下：

脚本第一行的输出内容"Content-type:text/html\r\n\r\n"发送到浏览器并告知浏览器显示的内容类型为"text/html"。

## HTTP头部

test.cgi文件内容中的" Content-type:text/html"即为HTTP头部的一部分，它会发送给浏览器告诉浏览器文件的内容类型。

HTTP头部的格式如下：

HTTP 字段名： 字段内容

例如：

Content-type:text/html\r\n\r\n

以下表格介绍了CGI程序中HTTP头部经常使用的信息：

头	描述
Content-type:	请求的与实体对应的MIME信息。例如: Content-type:text/html
Expires: Date	响应过期的日期和时间
Location: URL	用来重定向接收方到非请求URL的位置来完成请求或标识新的资源
Last-modified: Date	请求资源的最后修改时间
Content-length: N	请求的内容长度
Set-Cookie: String	设置Http Cookie

## CGI环境变量

所有的CGI程序都接收以下的环境变量，这些变量在CGI程序中发挥了重要的作用：

变量名	描述
CONTENT_TYPE	这个环境变量的值指示所传递来的信息的MIME类型。目前，环境变量CONTENT_TYPE一般都是：application/x-www-form-urlencoded,他表示数据来自于HTML表单。



CONTENT_LENGTH	如果服务器与CGI程序信息的传递方式是 <b>POST</b> ，这个环境变量即使从标准输入 <b>STDIN</b> 中可以读到的有效数据的字节数。这个环境变量在读取所输入的数据时必须使用。
HTTP_COOKIE	客户机内的 <b>COOKIE</b> 内容。
HTTP_USER_AGENT	提供包含了版本数或其他专有数据的客户浏览器信息。
PATH_INFO	这个环境变量的值表示紧接在 <b>CGI</b> 程序名之后的其他路径信息。它常常作为 <b>CGI</b> 程序的参数出现。
QUERY_STRING	如果服务器与 <b>CGI</b> 程序信息的传递方式是 <b>GET</b> ，这个环境变量的值即使所传递的信息。这个信息经跟在 <b>CGI</b> 程序名的后面，两者中间用一个问号'?'分隔。
REMOTE_ADDR	这个环境变量的值是发送请求的客户机的 <b>IP</b> 地址，例如上面的 <b>192.168.1.67</b> 。这个值总是存在的。而且它是 <b>Web</b> 客户机需要提供给 <b>Web</b> 服务器的唯一标识，可以在 <b>CGI</b> 程序中用它来区分不同的 <b>Web</b> 客户机。
REMOTE_HOST	这个环境变量的值包含发送 <b>CGI</b> 请求的客户机的主机名。如果不支持你想查询，则无需定义此环境变量。
REQUEST_METHOD	提供脚本被调用的方法。对于使用 <b>HTTP/1.0</b> 协议的脚本，仅 <b>GET</b> 和 <b>POST</b> 有意义。
SCRIPT_FILENAME	<b>CGI</b> 脚本的完整路径
SCRIPT_NAME	<b>CGI</b> 脚本的的名称
SERVER_NAME	这是你的 <b>WEB</b> 服务器的主机名、别名或 <b>IP</b> 地址。
SERVER_SOFTWARE	这个环境变量的值包含了调用 <b>CGI</b> 程序的 <b>HTTP</b> 服务器的名称和版本号。例如，上面的值为 <b>Apache/2.2.14(Unix)</b>

以下是一个简单的**CGI**脚本输出**CGI**的环境变量：

### 实例

```
#!/usr/bin/perl
print "Content-type: text/html\n\n";
print '<meta charset="utf-8">';
print "<font size=+1>环境变量: </font>\n";
foreach (sort keys %ENV)
{
    print "<b>$_</b>: $ENV{$_}<br>\n";
}
1;
```

### 文件下载

如果我们想通过 **Perl CGI** 实现文件下载，需要设置不同的头部信息，如下所示：

### 实例

```
#!/usr/bin/perl
# HTTP Header
print "Content-Type:application/octet-stream; name=\"FileName\"\\r\\n";
print "Content-Disposition: attachment; filename=\"FileName\"\\r\\n\\n";
# Actual File Content will go hear.
open( FILE, "<FileName" );
while(read(FILE, $buffer, 100) )
{
    print("$buffer");
}
```

### 使用GET方法传输数据

**GET**方法发送编码后的用户信息到服务端，数据信息包含在请求页面的**URL**上，以"?"号分割, 如下所示：

```
http://www.test.com/cgi-bin/test.cgi?key1=value1&key2=value2
```

有关 GET 请求的其他一些注释:

GET 请求可被缓存

GET 请求保留在浏览器历史记录中

GET 请求可被收藏为书签

GET 请求不应在处理敏感数据时使用

GET 请求有长度限制

GET 请求只应当用于取回数据

## 简单的url实例：GET方法

以下是一个简单的URL，使用GET方法向test.cgi程序发送两个参数：

```
/cgi-bin/test.cgi?name=菜鸟教程&url=http://www.runoob.com
```

以下为test.cgi文件的代码：

### 实例

```
#!/usr/bin/perl
local ($buffer, @pairs, $pair, $name, $value, %FORM);
# 读取文本信息
$ENV{'REQUEST_METHOD'} =~ tr/a-z/A-Z/;
if ($ENV{'REQUEST_METHOD'} eq "GET")
{
    $buffer = $ENV{'QUERY_STRING'};
}
# 读取 name/value 对信息
@pairs = split(/&/, $buffer);
foreach $pair (@pairs)
{
    ($name, $value) = split(/=/, $pair);
    $value =~ tr/+// ;
    $value =~ s/%(..)/pack("C", hex($1))/eg;
    $FORM{$name} = $value;
}
$name = $FORM{name};
$url = $FORM{url};
print "Content-type:text/html\r\n\r\n";
print "<html>";
print "<head>";
print '<meta charset="utf-8">';
print '<title>菜鸟教程(runoob.com)</title>';
print "</head>";
print "<body>";
print "<h2>$name网址: $url</h2>";
print "</body>";
print "</html>";
1;
```

查看浏览器，输出结果如下：

## 简单的表单实例：GET方法

以下是一个通过HTML的表单使用GET方法向服务器发送两个数据，提交的服务器脚本同样是test.cgi文件，test.html 代码如下：

### test.html 文件代码

```
<!DOCTYPE html>
<html>
```

```
<head>
<meta charset="utf-8">
<title>菜鸟教程(runoob.com)</title>
</head>
<body>
<form action="/cgi-bin/test.cgi" method="get">
站点名称: <input type="text" name="name"> <br />
站点 URL: <input type="text" name="url" />
<input type="submit" value="提交" />
</form>
</body>
</html>
```

浏览器中，执行效果如下所示：

□

## 使用POST方法传递数据

使用POST方法向服务器传递数据是更安全可靠，像一些敏感信息如用户密码等需要使用POST传输数据。

以下同样是 `test.cgi`，它也可以处理浏览器提交的POST表单数据：

### test.cgi 代码

```
#!/usr/bin/perl
local ($buffer, @pairs, $pair, $name, $value, %FORM);
# 读取文本信息
$ENV{'REQUEST_METHOD'} =~ tr/a-z/A-Z/;
if ($ENV{'REQUEST_METHOD'} eq "POST")
{
    read(STDIN, $buffer, $ENV{'CONTENT_LENGTH'});
}else {
    $buffer = $ENV{'QUERY_STRING'};
}
# 读取 name/value 对信息
@pairs = split(/&/, $buffer);
foreach $pair (@pairs)
{
    ($name, $value) = split(/=/, $pair);
    $value =~ tr/+// ;
    $value =~ s/%(..)/pack("C", hex($1))/eg;
    $FORM{$name} = $value;
}
$name = $FORM{name};
$url = $FORM{url};
print "Content-type:text/html\r\n\r\n";
print "<html>";
print "<head>";
print '<meta charset="utf-8">';
print '<title>菜鸟教程(runoob.com)</title>';
print "</head>";
print "<body>";
print "<h2>$name网址: $url</h2>";
print "</body>";
print "</html>";
1;
```

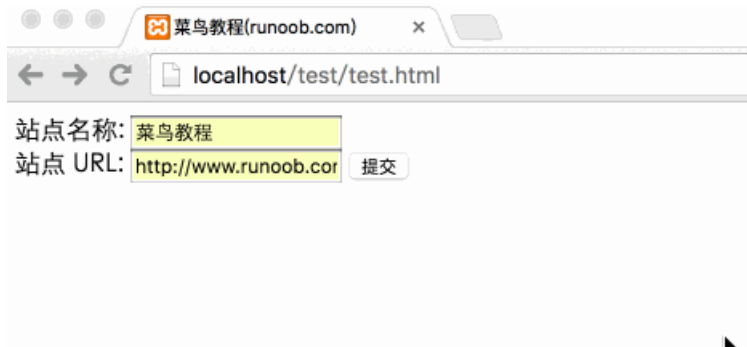
以下是一个通过HTML的表单使用GET方法向服务器发送两个数据，提交的服务器脚本同样是test.cgi文件，test.html 代码如下：

### test.html 代码

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>菜鸟教程(runoob.com)</title>
</head>
<body>
<form action="/cgi-bin/test.cgi" method="post">
站点名称: <input type="text" name="name"> <br />
站点 URL: <input type="text" name="url" />
<input type="submit" value="提交" />
</form>
```

```
</form>
</body>
</html>
```

浏览器中，执行效果如下所示：



## 通过CGI程序传递checkboxbox数据

checkboxbox用于提交一个或者多个选项数据，test.html 代码如下：

### test.html 代码

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>菜鸟教程(runoob.com)</title>
</head>
<body>
<form action="/cgi-bin/test.cgi" method="POST" target="_blank">
<input type="checkbox" name="runoob" value="on" /> 菜鸟教程
<input type="checkbox" name="google" value="on" /> Google
<input type="submit" value="选择站点" />
</form>
</body>
</html>
```

以下为 test.cgi 文件的代码：

### test.cgi 代码

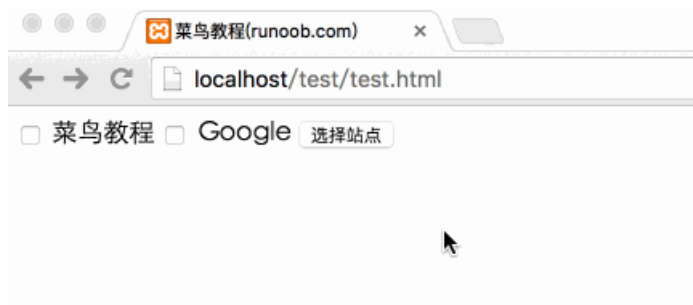
```
#!/usr/bin/perl
local ($buffer, @pairs, $pair, $name, $value, %FORM);
# 读取信息
$ENV{'REQUEST_METHOD'} =~ tr/a-z/A-Z/;
if ($ENV{'REQUEST_METHOD'} eq "POST")
{
read(STDIN, $buffer, $ENV{'CONTENT_LENGTH'});
}else {
$buffer = $ENV{'QUERY_STRING'};
}
# 读取 name/value 对信息
@pairs = split(/&/, $buffer);
foreach $pair (@pairs)
{
($name, $value) = split(/=/, $pair);
$value =~ tr/+//;
$value =~ s/%(..)/pack("C", hex($1))/eg;
$FORM{$name} = $value;
}
if( $FORM{runoob} ){
$runoob_flag = "ON";
}else{
$runoob_flag = "OFF";
}
if( $FORM{google} ){
$google_flag = "ON";
}else{
$google_flag = "OFF";
}
}
```

```

print "Content-type:text/html\r\n\r\n";
print "<html>";
print "<head>";
print '<meta charset="utf-8">';
print '<title>菜鸟教程(runoob.com)</title>';
print "</head>";
print "<body>";
print "<h2> 菜鸟教程选中状态 : $runoob_flag</h2>";
print "<h2> Google 选择状态 : $google_flag</h2>";
print "</body>";
print "</html>";
1;

```

浏览器中，执行效果如下所示：



## 通过CGI程序传递Radio数据

Radio 只向服务器传递一个数据，test.html 代码如下：

### test.html 代码

```

<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>菜鸟教程(runoob.com)</title>
</head>
<body>
<form action="/cgi-bin/test.cgi" method="post" target="_blank">
<input type="radio" name="site" value="runoob" /> 菜鸟教程
<input type="radio" name="site" value="google" /> Google
<input type="submit" value="提交" />
</form>
</body>
</html>

```

test.cgi 脚本代码如下：

### test.cgi 代码

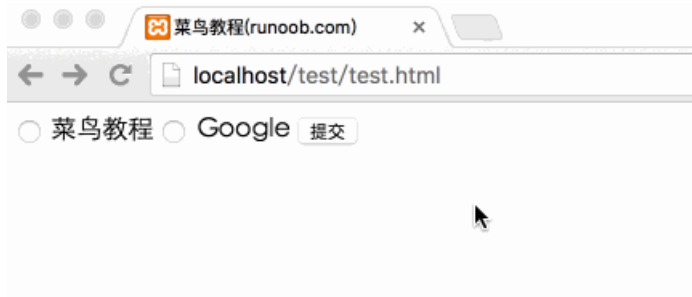
```

#!/usr/bin/perl
local ($buffer, @pairs, $pair, $name, $value, %FORM);
# 读取信息
$ENV{'REQUEST_METHOD'} =~ tr/a-z/A-Z/;
if ($ENV{'REQUEST_METHOD'} eq "POST")
{
read(STDIN, $buffer, $ENV{'CONTENT_LENGTH'});
}else {
$buffer = $ENV{'QUERY_STRING'};
}
# 读取 name/value 对信息
@pairs = split(/&/, $buffer);
foreach $pair (@pairs)
{
($name, $value) = split(/=/, $pair);
$value =~ tr/+//;
$value =~ s/%(..)/pack("C", hex($1))/eg;
$FORM{$name} = $value;
}
$site = $FORM{site};
print "Content-type:text/html\r\n\r\n";

```

```
print "<html>";
print "<head>";
print '<meta charset="utf-8">';
print '<title>菜鸟教程(runoob.com)</title>';
print "</head>";
print "<body>";
print "<h2> 选择的网站 $site</h2>";
print "</body>";
print "</html>";
1;
```

浏览器中，执行效果如下所示：



## 通过CGI程序传递 Textarea 数据

Textarea 向服务器传递多行数据，test.html 代码如下：

### test.html 代码

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>菜鸟教程(runoob.com)</title>
</head>
<body>
<form action="/cgi-bin/test.cgi" method="post" target="_blank">
<textarea name="textcontent" cols="40" rows="4">
在这里输入内容...
</textarea>
<input type="submit" value="提交" />
</form>
</body>
</html>
```

test.cgi 脚本代码如下：

### test.cgi 代码

```
#!/usr/bin/perl
local ($buffer, @pairs, $pair, $name, $value, %FORM);
# 读取信息
$ENV{'REQUEST_METHOD'} =~ tr/a-z/A-Z/;
if ($ENV{'REQUEST_METHOD'} eq "POST")
{
read(STDIN, $buffer, $ENV{'CONTENT_LENGTH'});
}else {
$buffer = $ENV{'QUERY_STRING'};
}
# 读取 name/value 对信息
@pairs = split(/&/, $buffer);
foreach $pair (@pairs)
{
($name, $value) = split(/=/, $pair);
$value =~ tr/+//;
$value =~ s/%(..)/pack("C", hex($1))/eg;
$FORM{$name} = $value;
}
$text_content = $FORM{textcontent};
print "Content-type:text/html\r\n\r\n";
print "<html>";
```

```
print "<head>";
print '<meta charset="utf-8">';
print '<title>菜鸟教程(runoob.com)</title>';
print "</head>";
print "<body>";
print "<h2>输入的文本内容为: $text_content</h2>";
print "</body>";
print "</html>";
1;
```

浏览器中，执行效果如下所示：

□

## 通过 CGI 程序传递下拉数据

HTML 下拉框代码如下：

### test.html 代码

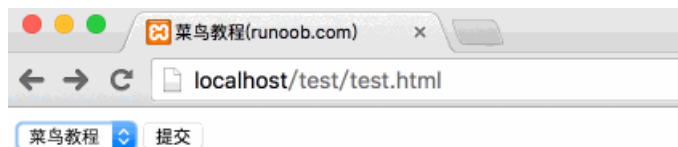
```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>菜鸟教程(runoob.com)</title>
</head>
<body>
<form action="/cgi-bin/test.cgi" method="post" target="_blank">
<select name="dropdown">
<option value="runoob" selected>菜鸟教程</option>
<option value="google">Google</option>
</select>
<input type="submit" value="提交"/>
</form>
</body>
</html>
```

test.cgi 脚本代码如下所示：

### test.cgi 代码

```
#!/usr/bin/perl
local ($buffer, @pairs, $pair, $name, $value, %FORM);
# 读取信息
$ENV{'REQUEST_METHOD'} =~ tr/a-z/A-Z/;
if ($ENV{'REQUEST_METHOD'} eq "POST")
{
read(STDIN, $buffer, $ENV{'CONTENT_LENGTH'});
}else {
$buffer = $ENV{'QUERY_STRING'};
}
# 读取 name/value 对信息
@pairs = split(/&/, $buffer);
foreach $pair (@pairs)
{
($name, $value) = split(/=/, $pair);
$value =~ tr/+ / /;
$value =~ s/%(..)/pack("C", hex($1))/eg;
$FORM{$name} = $value;
}
$site = $FORM{dropdown};
print "Content-type:text/html\r\n\r\n";
print "<html>";
print "<head>";
print '<meta charset="utf-8">';
print '<title>菜鸟教程(runoob.com)</title>';
print "</head>";
print "<body>";
print "<h2>选择的网站是: $site</h2>";
print "</body>";
print "</html>";
1;
```

浏览器中，执行效果如下所示：



## CGI中使用Cookie

在 http 协议一个很大的缺点就是不对用户身份的的判断，这样给编程人员带来很大的不便，而 cookie 功能的出现弥补了这个不足。

cookie 就是在客户访问脚本的同时，通过客户的浏览器，在客户硬盘上写入纪录数据，当下次客户访问脚本时取回数据信息，从而达到身份判别的功能，cookie 常用在身份校验中。

### cookie的语法

http cookie的发送是通过http头部来实现的，他早于文件的传递，头部set-cookie的语法如下：

```
Set-cookie:name=name;expires=date;path=path;domain=domain;secure
```

**name=name:** 需要设置cookie的值(name不能使用";"和",")，有多个name值时用 ";" 分隔，例如：**name1=name1;name2=name2;name3=name3**。

**expires=date:** cookie的有效期限,格式：**expires="Wdy,DD-Mon-YYYY HH:MM:SS"**

**path=path:** 设置cookie支持的路径,如果path是一个路径，则cookie对这个目录下的所有文件及子目录生效，例如：**path="/cgi-bin/"**，如果path是一个文件，则cookie指对这个文件生效，例如：**path="/cgi-bin/cookie.cgi"**。

**domain=domain:** 对cookie生效的域名，例如：**domain="www.runoob.com"**

**secure:** 如果给出此标志，表示cookie只能通过SSL协议的https服务器来传递。

cookie的接收是通过设置环境变量HTTP\_COOKIE来实现的，CGI程序可以通过检索该变量获取cookie信息。

## Cookie设置

Cookie的设置非常简单，cookie会在http头部单独发送。以下实例在cookie中设置了UserID、Password 和 expires：

### 实例

```
#!/usr/bin/perl
print "Set-Cookie:UserID=XYZ;\n";
print "Set-Cookie:Password=XYZ123;\n";
print "Set-Cookie:Expires=Tuesday, 31-Dec-2017 23:12:40 GMT;\n";
print "Set-Cookie:Domain=www.runoob.com;\n";
print "Set-Cookie:Path=/perl;\n";
print "Content-type:text/html\r\n\r\n";
.....其他 HTML 内容
```

### 查找 Cookie

Cookie信息检索页非常简单，Cookie信息存储在CGI的环境变量HTTP\_COOKIE中，存储格式如下：

### 实例

```
#!/usr/bin/perl
$rcvd_cookies = $ENV{'HTTP_COOKIE'};
@cookies = split /;/, $rcvd_cookies;
foreach $cookie ( @cookies ){
    ($key, $val) = split(/=/, $cookie); # splits on the first =.
    $key =~ s/^\s+//;
    $val =~ s/^\s+//;
    $key =~ s/\s+$//;
```



```
$val =~ s/\s+$//;
if( $key eq "UserID" ){
$user_id = $val;
}elseif($key eq "Password"){
$password = $val;
}
}
print "User ID = $user_id\n";
print "Password = $password\n";
```

以上实例输出结果为：

```
User ID = XYZ

Password = XYZ123
```

## CGI 模块

Perl 提供了很多内置的 CGI 模块，常用以下两个：

[CGI 模块](#)

[Berkeley cgi-lib.pl](#)

[Perl 数据库连接](#)

Perl 包和模块 [Perl](#)

[点我分享笔记](#)

反馈/建议



[Perl CGI编程](#)

Perl 进程管理 [Perl](#)

## Perl 包和模块

Perl 中每个包有一个单独的符号表，定义语法为：

```
package mypack;
```

此语句定义一个名为 **mypack** 的包，在此后定义的所有变量和子程序的名字都存贮在该包关联的符号表中，直到遇到另一个 **package** 语句为止。

每个符号表有其自己的一组变量、子程序名，各组名字是不相关的，因此可以在不同的包中使用相同的变量名，而代表的是不同的变量。

从一个包中访问另外一个包的变量，可通过"包名 + 双冒号(::) + 变量名"的方式指定。

存贮变量和子程序的名字的默认符号表是与名为 **main** 的包相关联的。如果在程序里定义了其它的包，当你想切换回去使用默认的符号表，可以重新指定 **main** 包：

```
package main;
```

这样，接下来的程序就好象从没定义过包一样，变量和子程序的名字象通常那样存贮。

以下实例中文件有 `main` 和 `Foo` 包。特殊变量 `__PACKAGE__` 用于输出包名：

## 实例

```
#!/usr/bin/perl
# main 包
$i = 1;
print "包名 : " , __PACKAGE__ , " $i\n";
package Foo;
# Foo 包
$i = 10;
print "包名 : " , __PACKAGE__ , " $i\n";
package main;
# 重新指定 main 包
$i = 100;
print "包名 : " , __PACKAGE__ , " $i\n";
print "包名: " , __PACKAGE__ , " $Foo::i\n";
1;
```

执行以上程序，输出结果为：

```
包名 : main 1

包名 : Foo 10

包名 : main 100

包名: main 10
```

## BEGIN 和 END 模块

Perl语言提供了两个关键字：`BEGIN`，`END`。它们可以分别包含一组脚本，用于程序体运行前或者运行后的执行。

语法格式如下：

```
BEGIN { ... }

END { ... }

BEGIN { ... }

END { ... }
```

每个 **BEGIN** 模块在 Perl 脚本载入和编译后但在其他语句执行前执行。

每个 **END** 语句块在解释器退出前执行。

**BEGIN** 和 **END** 语句块在创建 Perl 模块时特别有用。

如果你还不大理解，我们可以看个实例：

## 实例

```
#!/usr/bin/perl
package Foo;
print "Begin 和 Block 实例\n";
BEGIN {
print "这是 BEGIN 语句块\n"
```

```
}  
END {  
    print "这是 END 语句块\n"  
}  
1;
```

执行以上程序，输出结果为：

```
这是 BEGIN 语句块  
  
Begin 和 Block 实例  
  
这是 END 语句块
```

## 什么是 Perl 模块？

Perl5 中用 Perl 包来创建模块。

Perl 模块是一个可重复使用的包，模块的名字与包名相同，定义的文件后缀为 **.pm**。

以下我们定义了一个模块 **Foo.pm**，代码如下所示：

### 实例

```
#!/usr/bin/perl  
package Foo;  
sub bar {  
    print "Hello $_[0]\n"  
}  
sub blat {  
    print "World $_[0]\n"  
}  
}  
1;
```

Perl 中关于模块需要注意以下几点：

函数 **require** 和 **use** 将载入一个模块。

**@INC** 是 Perl 内置的一个特殊数组，它包含指向库例程所在位置的目录路径。

**require** 和 **use** 函数调用 **eval** 函数来执行代码。

末尾 **1**；执行返回 **TRUE**，这是必须的，否则返回错误。

## Require 和 Use 函数

模块可以通过 **require** 函数来调用，如下所示：

### 实例

```
#!/usr/bin/perl  
require Foo;  
Foo::bar( "a" );  
Foo::blat( "b" );
```

也可以通过 **use** 函数来引用：

### 实例

```
#!/usr/bin/perl  
use Foo;  
bar( "a" );  
blat( "b" );
```

我们注意到 **require** 引用需要使用包名指定函数，而 **use** 不需要，二者的主要区别在于：

- 1、**require**用于载入module或perl程序(.pm后缀可以省略，但.pl必须有)

- 2、Perl `use`语句是编译时引入的，`require`是运行时引入的
- 3、Perl `use`引入模块的同时，也引入了模块的子模块。而`require`则不能引入，要在重新声明
- 4、`USE`是在当前默认的`@INC`里面去寻找，一旦模块不在`@INC`中的话，用`USE`是不可以引入的，但是`require`可以指定路径
- 5、`USE`引用模块时，如果模块名称中包含`::`双冒号，该双冒号将作为路径分隔符，相当于Unix下的/或者Windows下的\。如：

```
use MyDirectory::MyModule
```

通过添加以下语句 `use` 模块就可以从模块中导出列表符号：

```
require Exporter;

@ISA = qw(Exporter);
```

`@EXPORT`数组包含默认导出的变量和函数的名字：

```
package Module;

require Exporter;

@ISA = qw(Exporter);

@EXPORT = qw(bar blat); # 默认导出的符号

sub bar { print "Hello $_[0]\n" }

sub blat { print "World $_[0]\n" }

sub splat { print "Not $_[0]\n" } # Not exported!

1;
```

## 创建 Perl 模块

通过 Perl 分发自带的工具 `h2xs` 可以很简单的创建一个 Perl 模块。

你可以在命令行模式键入 `h2xs` 来看看它的参数列表。

`h2xs` 语法格式：

```
$ h2xs -AX -n ModuleName
```

参数说明：

- A 忽略 `autoload` 机制
- X 忽略 `XS` 元素
- n 指定扩展模块的名字

例如，如果你的模块在 **Person.pm** 文件中，使用以下命令：

```
$ h2xs -AX -n Person
```

执行以上程序将输出：

```
Writing Person/lib/Person.pm

Writing Person/Makefile.PL

Writing Person/README

Writing Person/t/Person.t

Writing Person/Changes

Writing Person/MANIFEST
```

**Person** 目录下你可以看到新增加的目录及文件说明：

**README**：这个文件包含一些安装信息，模块依赖性，版权信息等。

**Changes**：这个文件作为你的项目的修改日志（**changelog**）文件。

**Makefile.PL**：这是标准的 **Perl Makefile** 构造器。用于创建 **Makefile.PL** 文件来编译该模块。

**MANIFEST**：本文件用于自动构建 **tar.gz** 类型的模块版本分发。这样你就可以把你的模块拿到 **CPAN** 发布或者分发给其他人。它包含了你在这个项目中所有文件的列表。

**Person.pm**：这是主模块文件，包含你的 **mod\_perl** 句柄代码（**handler code**）。

**Person.t**：针对该模块的一些测试脚本。默认情况下它只是检查模块的载入，你可以添加一些新的测试单元。

**t/**：测试文件

**lib/**：实际源码存放的目录

你可以使用 **tar (Linux 上)** 命令来将以上目录打包为 **Person.tar.gz**。

## 安装 Perl 模块

我们可以对刚才压缩的 **Person.tar.gz** 文件进行解压安装，执行步骤如下：

```
tar xvfz Person.tar.gz

cd Person

perl Makefile.PL

make

make install
```

首先运行 "**perl Makefile.PL**" 在当前目录生成 **Makefile**；

然后运行 "**make**" 编译并创建所需的库文件；

之后用 "**make test**" 测试编译结果是否正确；最后运行 "**make install**" 将库文件安装到系统目录，至此整个编译过程结束。



## Perl 进程管理

Perl 中你可以以不同的方法来创建进程。

本教程将讨论一些进程的管理方法。

你可以使用特殊变量 `$$` 或 `$PROCESS_ID` 来获取进程 ID。

`%ENV` 哈希存放了父进程，也就是shell中的环境变量，在Perl中可以修改这些变量。

`exit()` 通常用于退出子进程，主进程在子进程全部退出后再退出。

所有打开的句柄会在子程序中被 `dup()` 函数复制，所有关闭进程所有句柄不会影响其他进程。

## 反引号运算符

使用反引号运算符可以很容易的执行 **Unix** 命令。你可以在反引号中插入一些简单的命令。命令执行后将返回结果：

```
#!/usr/bin/perl

@files = `ls -l`;

foreach $file (@files){

    print $file;

}

1;
```

执行以上程序，输出结果如下：

```
drwxr-xr-x 3 root root 4096 Sep 14 06:46 9-14

drwxr-xr-x 4 root root 4096 Sep 13 07:54 android

-rw-r--r-- 1 root root 574 Sep 17 15:16 index.htm

drwxr-xr-x 3 544 401 4096 Jul 6 16:49 MIME-Lite-3.01

-rw-r--r-- 1 root root 71 Sep 17 15:16 test.pl

.....
```

## system() 函数

你也可以使用 **system()** 函数执行 **Unix** 命令, 执行该命令将直接输出结果。默认情况下会送到目前**Perl**的**STDOUT**指向的地方, 一般是屏幕。你也可以使用重定向运算符 **>** 输出到指定文件:

执行以上程序, 输出结果如下:

```
drwxr-xr-x 3 root root 4096 Sep 14 06:46 9-14

drwxr-xr-x 4 root root 4096 Sep 13 07:54 android

-rw-r--r-- 1 root root 574 Sep 17 15:16 index.htm

drwxr-xr-x 3 544 401 4096 Jul 6 16:49 MIME-Lite-3.01

-rw-r--r-- 1 root root 71 Sep 17 15:16 test.pl

.....
```

你需要注意命令包含环境变量如 **\$PATH** 或 **\$HOME** 的输出结果, 如下所示:

### 实例

```
#!/usr/bin/perl
$PATH = "我是 Perl 的变量";
system('echo $PATH'); # $PATH 作为 shell 环境变量
system("echo $PATH"); # $PATH 作为 Perl 的变量
system("echo \ $PATH"); # 转义 $
1;
```

执行以上程序, 输出结果如下:

```
/usr/local/bin:/bin:/usr/bin:/usr/local/sbin:/usr/sbin:/sbin

我是 Perl 的变量

/usr/local/bin:/bin:/usr/bin:/usr/local/sbin:/usr/sbin:/sbin
```

## fork() 函数

**Perl fork()** 函数用于创建一个新进程。

在父进程中返回子进程的**PID**, 在子进程中返回**0**。如果发生错误(比如, 内存不足)返回**undef**, 并将**\$\_**设为对应的错误信息。

**fork** 可以和 **exec** 配合使用。**exec** 函数执行完引号中的命令后进程即结束。

### 实例

```
#!/usr/bin/perl
```

```
if(!defined($pid = fork())) {
# fork 发生错误返回 undef
die "无法创建子进程: $!";
}elseif ($pid == 0) {
print "通过子进程输出\n";
exec("date") || die "无法输出日期: $!";
} else {
# 在父进程中
print "通过父进程输出\n";
$ret = waitpid($pid, 0);
print "完成的进程ID: $ret\n";
}
1;
```

执行以上程序，输出结果如下：

通过父进程输出

通过子进程输出

2016年 6月19日 星期日 22时21分14秒 CST

完成的进程ID: 47117

如果进程退出时,会向父进程发送一个CHLD的信号后,就会变成僵死的进程,需要父进程使用wait和waitpid来终止。当然,也可以设置SIG{CHLD}为IGNORE:

## 实例

```
#!/usr/bin/perl
local $SIG{CHLD} = "IGNORE";
if(!defined($pid = fork())) {
# fork 发生错误返回 undef
die "无法创建子进程: $!";
}elseif ($pid == 0) {
print "通过子进程输出\n";
exec("date") || die "无法输出日期: $!";
} else {
# 在父进程中
print "通过父进程输出\n";
$ret = waitpid($pid, 0);
print "完成的进程ID: $ret\n";
}
1;
```

执行以上程序，输出结果如下：

通过父进程输出

通过子进程输出

2016年 6月19日 星期日 22时30分56秒 CST

完成的进程ID: -1

## Kill 函数

Perl kill('signal', (Process List))给一组进程发送信号。signal是发送的数字信号，9为杀掉进程。

首先看看linux中的常用信号,见如下列表：



信号名	值	标注	解释
<hr/>			
HUP	1	A	检测到挂起
INT	2	A	来自键盘的中断
QUIT	3	A	来自键盘的停止
ILL	4	A	非法指令
ABRT	6	C	失败
FPE	8	C	浮点异常
KILL	9	AF	终端信号
USR1	10	A	用户定义的信号1
SEGV	11	C	非法内存访问
USR2	12	A	用户定义的信号2
PIPE	13	A	写往没有读取者的管道
ALRM	14	A	来自闹钟的定时器信号
TERM	15	A	终端信号
CHLD	17	B	子进程终止
CONT	18	E	如果被停止则继续
STOP	19	DF	停止进程
TSTP	20	D	<b>tty</b> 键入的停止命令
TTIN	21	D	对后台进程的 <b>tty</b> 输入
TTOU	22	D	对后台进程的 <b>tty</b> 输出

以下实例向进程 104 和 102 发送 SIGINT 信号：

### 实例

```
#!/usr/bin/perl
kill('INT', 104, 102);
1;
```

反馈/建议



# Perl POD 文档

Perl 中可以在模块或脚本中嵌入 POD（Plain Old Documentation）文档。

POD 是一种简单而易用的标记型语言（置标语言）。

POD 文档使用规则：

POD 文档以 `=head1` 开始，`=cut` 结束，`=head1` 前与 `=cut` 后添加一空行。

Perl 会忽略 POD 中的文档。实例如下：

## 实例

```
#!/usr/bin/perl
print "Hello, World\n";
=head1 Hello, World 实例
这是一个 Perl 的简单实例。
=cut
print "Hello, Runoob\n";
```

执行以上程序，输出结果为：

```
Hello, World

Hello, Runoob
```

我们还可以使用 `"__END__"` 或 `"__DATA__"` 将所在行之后的内容全部"注释"掉：

## 实例

```
#!/usr/bin/perl
print "Hello, World\n";
while(<DATA>){
    print $_;
}
__END__
=head1 Hello, World 实例
这是一个 Perl 的简单实例。
print "Hello, Runoob\n";
```

执行以上程序，输出结果为：

```
Hello, World

=head1 Hello, World 实例

这是一个 Perl 的简单实例。

print "Hello, Runoob\n";
```

以下实例不读取 POD 文档：

## 实例

```
#!/usr/bin/perl
print "Hello, World\n";
__END__
=head1 Hello, World 实例
这是一个 Perl 的简单实例。
print "Hello, Runoob\n";
```

执行以上程序，输出结果为：

```
Hello, World
```

## 什么是 POD?

Pod(Plain Old Documentation), 是一种简单而易用的标记型语言（置标语言），它经常用于在perl程序和模块中的文档书写。

Pod 的 转化器可以将 Pod 转换成很多种格式，例如 text, html, man 等很多。

Pod 标记语言包含三种基本基本类型：普通, 原文, 和 命令。

**普通段落:** 你可以在普通段落中使用格式化代码，如黑体，斜体，或代码风格，下划线等。

**原文段落:** 原文段落，用于代码块或者其他不需要转换器处理的部分，而且不需要段落重排。

**命令段落:** 命令段落作用于整个的文档，通常用于标题设置或列表标记。

所有的命令段落（他只有一行的长度）使用 "=" 开始，然后是一个标识符。随后的文本将被这条命令所影响。现在被广泛使用的命令包括

```
=pod （开始文档）

=head1 标题文本

=head2 标题文本

=head3 标题文本

=head4 标题文本

=over 缩进空格数量

=item 前缀

=back （结束列表）

=begin 文档格式

=end 结束文档格式

=for 格式文本

=encoding 编码类型

=cut （文档结束）
```

在perl中，可以使用 pod2html \*.pod >\*.html 来生成html格式的pod文档。

考虑以下 POD 实例：

## 实例

```
=begin html
=encoding utf-8
```

```
=head1 菜鸟教程
=cut
```

pod2html时会原文拷贝此段代码。

使用 pod2html 命令执行，将其转换为 HTML 代码：

```
$ pod2html test.pod > test.html
```

在浏览器中打开 test.html，链接部分为索引，显示如下：

- [菜鸟教程](#)

## 菜鸟教程

以下实例在 POD 文档中直接写入 HTML：

```
=begin html

=encoding utf-8


<h1>菜鸟教程</h1>

<p> www.runoob.com </p>


=end html
```

pod2html时会原文拷贝此段代码。

使用 pod2html 命令执行，将其转换为 HTML 代码：

```
$ pod2html test.pod > test.html
```

在浏览器中打开 test.html，链接部分为索引，显示如下：

## 菜鸟教程

www.runoob.com

☐ Perl 进程管理

☐ 点我分享笔记

反馈/建议