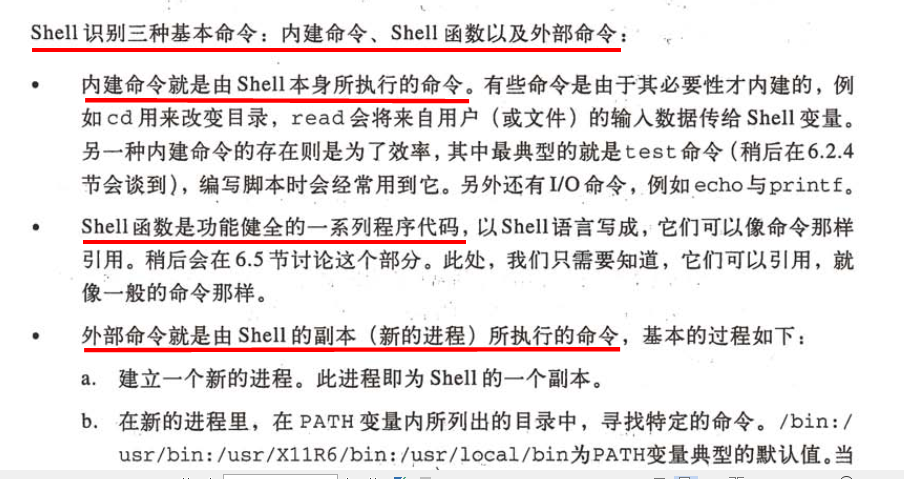
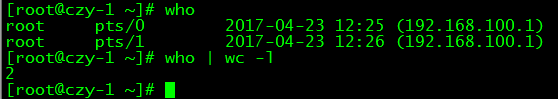
* http://c.biancheng.net/cpp/view/6996.html

****

**计算机器登陆的个数**

[root@czy-1 ~]# who | wc –l



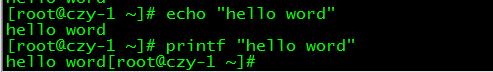
**输出**

[root@czy-1 ~]# echo "hello word"

hello word

[root@czy-1 ~]# printf "hello word" （自动换行）

hello word[root@czy-1 ~]# （不自动换行）



[root@czy-1 ~]# printf "hello word \n"

hello word



# Shell有两种执行命令的方式

|  |
| --- |
| * 交互式（Interactive）：解释执行用户的命令，用户输入一条命令，Shell就解释执行一条。 * 批处理（Batch）：用户事先写一个Shell脚本(Script)，其中有很多条命令，让Shell一次把这些命令执行完，而不必一条一条地敲命令。 * Shell是一种脚本语言，那么，就必须有解释器来执行这些脚本。  Unix/Linux上常见的Shell脚本解释器有bash、sh、csh、ksh等，习惯上把它们称作一种Shell。我们常说有多少种Shell，其实说的是Shell脚本解释器。 |

# 常用脚本解释器

|  |
| --- |
| **bash**  bash是Linux标准默认的shell，本教程也基于bash讲解。bash由Brian Fox和Chet Ramey共同完成，是BourneAgain Shell的缩写，内部命令一共有40个。  Linux使用它作为默认的shell是因为它有诸如以下的特色：   * 可以使用类似DOS下面的doskey的功能，用方向键查阅和快速输入并修改命令。 * 自动通过查找匹配的方式给出以某字符串开头的命令。 * 包含了自身的帮助功能，你只要在提示符下面键入help就可以得到相关的帮助。 |
| **sh**  sh 由Steve Bourne开发，是Bourne Shell的缩写，sh 是Unix 标准默认的shell。 |

# 编程语言

|  |
| --- |
| 很多传统的程序设计语言，例如Fortran、Ada、Pascal、C、C++和Java，都是编译型语言。这类语言需要预先将我们写好的源代码(source code)转换成目标代码(object code)，这个过程被称作“编译”。  运行程序时，直接读取目标代码(object code)。由于编译后的目标代码(object code)非常接近计算机底层，因此执行效率很高，这是编译型语言的优点。  但是，由于编译型语言多半运作于底层，所处理的是字节、整数、浮点数或是其他机器层级的对象，往往实现一个简单的功能需要大量复杂的代码。例如，在C++里，就很难进行“将一个目录里所有的文件复制到另一个目录中”之类的简单操作。 |

# 解释型语言

|  |
| --- |
| 解释型语言也被称作“脚本语言”。执行这类程序时，解释器(interpreter)需要读取我们编写的源代码(source code)，并将其转换成目标代码(object code)，再由计算机运行。因为每次执行程序都多了编译的过程，因此效率有所下降。  使用脚本编程语言的好处是，它们多半运行在比编译型语言还高的层级，能够轻易处理文件与目录之类的对象；缺点是它们的效率通常不如编译型语言。不过权衡之下，通常使用脚本编程还是值得的：花一个小时写成的简单脚本，同样的功能用C或C++来编写实现，可能需要两天，而且一般来说，脚本执行的速度已经够快了，快到足以让人忽略它性能上的问题。脚本编程语言的例子有awk、Perl、Python、Ruby与Shell。  因为Shell似乎是各UNIX系统之间通用的功能，并且经过了POSIX的标准化。因此，Shell脚本只要“用心写”一次，即可应用到很多系统上。因此，之所以要使用Shell脚本是基于：   * 简单性：Shell是一个高级语言；通过它，你可以简洁地表达复杂的操作。 * 可移植性：使用POSIX所定义的功能，可以做到脚本无须修改就可在不同的系统上执行。 * 开发容易：可以在短时间内完成一个功能强大又妤用的脚本。   但是，考虑到Shell脚本的命令限制和效率问题，下列情况一般不使用Shell：   1. 资源密集型的任务，尤其在需要考虑效率时（比如，排序，hash等等）。 2. 需要处理大任务的数学操作，尤其是浮点运算，精确运算，或者复杂的算术运算（这种情况一般使用C++或FORTRAN 来处理）。 3. 有跨平台（操作系统）移植需求（一般使用C 或Java）。 4. 复杂的应用，在必须使用结构化编程的时候（需要变量的类型检查，函数原型，等等）。 5. 对于影响系统全局性的关键任务应用。 6. 对于安全有很高要求的任务，比如你需要一个健壮的系统来防止入侵、破解、恶意破坏等等。 7. 项目由连串的依赖的各个部分组成。 8. 需要大规模的文件操作。 9. 需要多维数组的支持。 10. 需要数据结构的支持，比如链表或数等数据结构。 11. 需要产生或操作图形化界面 GUI。 12. 需要直接操作系统硬件。 13. 需要 I/O 或socket 接口。 14. 需要使用库或者遗留下来的老代码的接口。 15. 私人的、闭源的应用（shell 脚本把代码就放在文本文件中，全世界都能看到）。   如果你的应用符合上边的任意一条，那么就考虑一下更强大的语言吧——或许是Perl、Tcl、Python、Ruby——或者是更高层次的编译语言比如C/C++，或者是Java。即使如此，你会发现，使用shell来原型开发你的应用，在开发步骤中也是非常有用的。 |

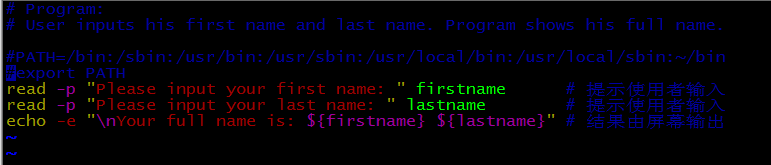
# 编写shell

|  |
| --- |
| 打开文本编辑器，新建一个文件，扩展名为sh（sh代表shell），扩展名并不影响脚本执行，见名知意就好，如果你用php写shell 脚本，扩展名就用php好了。  输入一些代码：   1. #!/bin/bash 2. echo "Hello World !"   “#!” 是一个约定的标记，它告诉系统这个脚本需要什么解释器来执行，即使用哪一种Shell。echo命令用于向窗口输出文本。 |
| **运行Shell脚本有两种方法**  **一、作为可执行程序**  将上面的代码保存为test.sh，并 cd 到相应目录：  chmod +x ./test.sh #使脚本具有执行权限  ./test.sh #执行脚本  注意，一定要写成./test.sh，而不是test.sh。运行其它二进制的程序也一样，直接写test.sh，linux系统会去PATH里寻找有没有叫test.sh的，而只有/bin, /sbin, /usr/bin，/usr/sbin等在PATH里，你的当前目录通常不在PATH里，所以写成test.sh是会找不到命令的，要用./test.sh告诉系统说，就在当前目录找。  通过这种方式运行bash脚本，第一行一定要写对，好让系统查找到正确的解释器。  这里的"系统"，其实就是shell这个应用程序（想象一下Windows Explorer），但我故意写成系统，是方便理解，既然这个系统就是指shell，那么一个使用/bin/sh作为解释器的脚本是不是可以省去第一行呢？是的。  **二、作为解释器参数**  这种运行方式是，直接运行解释器，其参数就是shell脚本的文件名，如：  /bin/sh test.sh  /bin/php test.php  这种方式运行的脚本，不需要在第一行指定解释器信息，写了也没用。 |

# my

下面的脚本使用 read 命令从 stdin 获取输入并赋值给 ${\*} 变量，最后在输出：

|  |
| --- |
| read -p "Please input your first name: " firstname # 提示使用者输入  read -p "Please input your last name: " lastname # 提示使用者输入  echo -e "\nYour full name is: ${firstname} ${lastname}" # 结果由屏幕输出 |



运行脚本：chmod +x showname.sh

./ showname.sh

