
在最简单的情况下，脚本只不过是存储在文件中的系统命令列表。 至少，这节省了每次调用特定命令序列时重新键入该命令的工作量。

Example 2-1. cleanup: A script to clean up log files in /var/log

```
# Cleanup
# Run as root, of course.

cd /var/log
cat /dev/null > messages
cat /dev/null > wtmp
echo "Log files cleaned up."
```

这里没有什么不寻常的，只有一组命令可以很容易地从控制台上的命令行或终端窗口中一个一个地调用。 将命令放在脚本中的好处远不止不必一次又一次地重新键入它们。 脚本变成了一个程序 —— 一个工具 —— 并且可以很容易地为特定的应用程序修改或定制。

Example 2-2. cleanup: An improved clean-up script

```
#!/bin/bash
# Proper header for a Bash script.

# Cleanup, version 2

# Run as root, of course.
# Insert code here to print error message and exit if not root.

LOG_DIR=/var/log
# Variables are better than hard-coded values.
cd $LOG_DIR

cat /dev/null > messages
cat /dev/null > wtmp


echo "Logs cleaned up."

exit #  The right and proper method of "exiting" from a script.
     #  A bare "exit" (no parameter) returns the exit status
     #+ of the preceding command.
```

现在这开始看起来像一个真正的脚本。 但我们可以走得更远。 . .

Example 2-3. cleanup: An enhanced and generalized version of above scripts.

```
#!/bin/bash
# Cleanup, version 3

#  Warning:
#  -------
#  This script uses quite a number of features that will be explained
#+ later on.
#  By the time you've finished the first half of the book,
#+ there should be nothing mysterious about it.



LOG_DIR=/var/log
ROOT_UID=0     # Only users with $UID 0 have root privileges.
LINES=50       # Default number of lines saved.
E_XCD=86       # Can't change directory?
E_NOTROOT=87   # Non-root exit error.


# Run as root, of course.
if [ "$UID" -ne "$ROOT_UID" ]
then
  echo "Must be root to run this script."
  exit $E_NOTROOT
fi

if [ -n "$1" ]
# Test whether command-line argument is present (non-empty).
then
  lines=$1
else
  lines=$LINES # Default, if not specified on command-line.
fi


#  Stephane Chazelas suggests the following,
#+ as a better way of checking command-line arguments,
#+ but this is still a bit advanced for this stage of the tutorial.
#
#    E_WRONGARGS=85  # Non-numerical argument (bad argument format).
#
#    case "$1" in
#    ""      ) lines=50;;
#    *[!0-9]*) echo "Usage: `basename $0` lines-to-cleanup";
#     exit $E_WRONGARGS;;
#    *       ) lines=$1;;
#    esac
```

```
#
#* Skip ahead to "Loops" chapter to decipher all this.


cd $LOG_DIR

if [ `pwd` != "$LOG_DIR" ]  # or   if [ "$PWD" != "$LOG_DIR" ]
                            # Not in /var/log?
then
  echo "Can't change to $LOG_DIR."
  exit $E_XCD
fi  # Doublecheck if in right directory before messing with log file.

# Far more efficient is:
#
# cd /var/log || {
#   echo "Cannot change to necessary directory." >&2
#   exit $E_XCD;
# }




tail -n $lines messages > mesg.temp # Save last section of message log file.
mv mesg.temp messages               # Rename it as system log file.


#  cat /dev/null > messages
#* No longer needed, as the above method is safer.

cat /dev/null > wtmp  #  ': > wtmp' and '> wtmp'  have the same effect.
echo "Log files cleaned up."
#  Note that there are other log files in /var/log not affected
#+ by this script.

exit 0
#  A zero return value from the script upon exit indicates success
#+ to the shell.
```

由于您可能不希望清除整个系统日志，因此此版本的脚本将消息日志的最后一部分保持不变。 您将不断发现微调以前编写的脚本以提高效率的方法。

* * *

脚本开头的 sha-bang (#!) [1] 告诉您的系统该文件是一组命令，将提供给指定的命令解释器。 这 #! 实际上是一个两字节的 [2] 幻数，一个指定文件类型的特殊标记，或者在这种情况下是一个可执行的 shell 脚本（键入 man magic 以获取有关这个迷人主题的更多详细信息）。 紧跟在 sha-bang 后面的是路径名。 这是解释脚本中命令的程序的路径，无论它是 shell、编程语言还是实用程序。 然后，此命令解释器执行脚本中的命令，从顶部（sha-bang 行之后的行）开始，并忽略注释。 [3]

```
#!/bin/sh
#!/bin/bash
#!/usr/bin/perl
#!/usr/bin/tcl
#!/bin/sed -f
#!/bin/awk -f
```

上述每个脚本标题行都调用不同的命令解释器，无论是 /bin/sh、默认 shell（Linux 系统中的 bash）还是其他。 [4] 使用 #!/bin/sh（大多数商业 UNIX 变体中的默认 Bourne shell）使脚本可移植到非 Linux 机器，尽管您牺牲了 Bash 特定的功能。 但是，该脚本将符合 POSIX [5] sh 标准。

请注意，“sha-bang”处给出的路径必须正确，否则会出现错误消息——通常是 “找不到命令” 。 -- 将是运行脚本的唯一结果。 [6]

#! 如果脚本仅由一组通用系统命令组成，不使用内部 shell 指令，则可以省略。 上面的第二个示例需要初始 #!，因为变量赋值行 lines=50 使用特定于 shell 的构造。 [7] 再次注意 #!/bin/sh 调用默认的 shell 解释器，在 Linux 机器上默认为 /bin/bash。

- 本教程鼓励使用模块化方法来构建脚本。 记下并收集可能在未来脚本中有用的“样板”代码片段。 最终，您将构建一个相当广泛的漂亮例程库。 例如，以下脚本 prolog 测试是否已使用正确数量的参数调用脚本。

```
E_WRONG_ARGS=85
script_parameters="-a -h -m -z"
#                   -a = all, -h = help, etc.

if [ $# -ne $Number_of_expected_args ]
then
echo "Usage: `basename $0` $script_parameters"
# `basename $0` is the script's filename.
exit $E_WRONG_ARGS
fi
```

- 很多时候，您将编写一个执行特定任务的脚本。 本章的第一个脚本就是一个示例。 稍后，您可能会想到将脚本概括为执行其他类似的任务。 用变量替换文字（“硬连线”）常量是朝这个方向迈出的一步，用函数替换重复的代码块也是如此。

Notes

[1] 在文献中更常见的是 she-bang 或 sh-bang。 这源自于标记Sharp (#) 和bang (!) 的串联。

---

[2] 一些 UNIX 风格（基于 4.2 BSD 的）据称采用四字节幻数，在 ! 之后需要一个空格。 —— #! /bin/sh。 根据 Sven Mascheck 的说法，这可能是一个神话。

---

[3] 这 #! shell 脚本中的行将是命令解释器（sh 或 bash）首先看到的内容。 由于该行以 # 开头，因此当命令解释器最终执行脚本时，它将被正确解释为注释。 该行已经达到了它的目的——调用命令解释器。
事实上，如果脚本包含一个额外的 #! 行，然后 bash 会将其解释为注释。

```
#!/bin/bash

echo "Part 1 of script."
a=1

#!/bin/bash
# This does *not* launch a new script.

echo "Part 2 of script."
echo $a  # Value of $a stays at 1.
```

---

[4] 这允许一些可爱的技巧。

```
#!/bin/rm
# Self-deleting script.

# Nothing much seems to happen when you run this... except that the file
disappears.

WHATEVER=85

echo "This line will never print (betcha!)."

exit $WHATEVER  # Doesn't matter. The script will not exit here.
                # Try an echo $? after script termination.
                # You'll get a 0, not a 85.
```

另外，尝试使用 #!/bin/more 开始一个 README 文件，并使其可执行。 结果是一个自列出的文档文件。 （使用 cat 的 here 文档可能是更好的选择——参见示例 19-3）。

---

[5] 便携式操作系统接口，尝试标准化类 UNIX 操作系统。 POSIX 规范列在 Open Group 网站上。

---

[6] 为了避免这种可能性，脚本可以以 #!/bin/env bash sha-bang 行开头。 这在 bash 不在 /bin 中的 UNIX 机器上可能很有用

[7] 如果 Bash 是您的默认 shell，那么 #! 在脚本的开头不是必需的。 但是，如果从不同的 shell（例如 tcsh）启动脚本，则需要 #!。