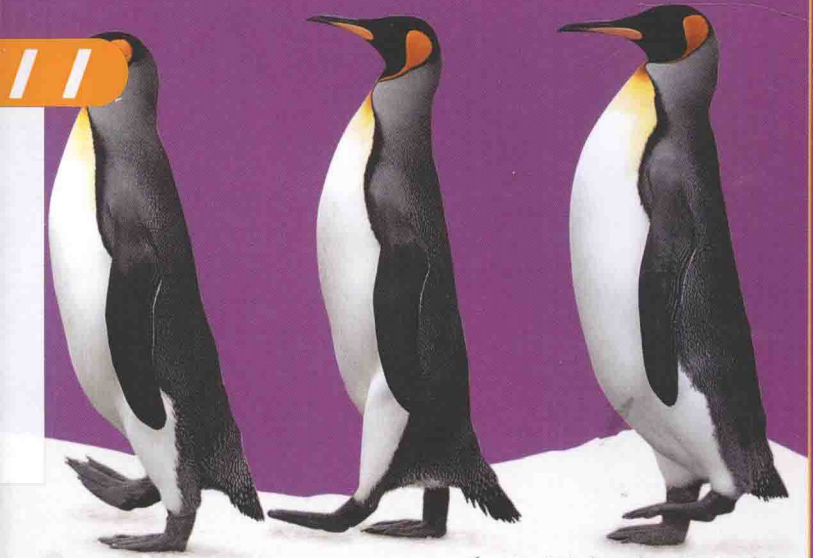
LINUX

- 从基本概念着手，通过丰富、实用的范例，深入浅出地讲解 Shell 编程语言
- 配有全程视频教学光盘，方便读者学习

Shell 编程从入门到精通

（第 2 版）

张昊 程国钢 编著



中国工信出版集团
人民邮电出版社
POSTS & TELECOM PRESS

Linux

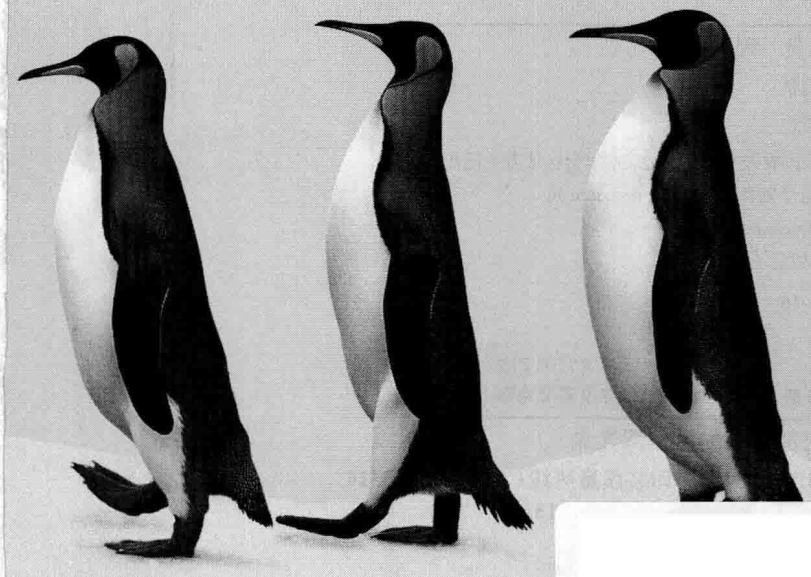
附赠全程视频教学光盘

LINUX

Shell 编程从入门到精通

（第2版）

张昊 程国钢 编著



Linux

人民邮电出版社
北京

图书在版编目（CIP）数据

Linux Shell编程从入门到精通 / 张昊，程国钢编著
. -- 2版. -- 北京 : 人民邮电出版社，2015.9
ISBN 978-7-115-40004-8

Ⅰ. ①L… Ⅱ. ①张… ②程… Ⅲ. ①Linux操作系统
—程序设计 Ⅳ. ①TP316.89

中国版本图书馆CIP数据核字(2015)第191897号

内 容 提 要

本书由浅入深、循序渐进地详细讲解了 Linux Shell 编程的基本知识，主要包括 Shell 编程的基本知识、文本处理的工具和方法、正则表达式、Linux 系统知识等。

本书旨在通过理清 Linux Shell 编程的脉络，从基本概念着手，以丰富、实用的实例作为辅助，使读者能够深入浅出地学习 Linux Shell 编程。本书以丰富的范例、详细的源代码讲解、独到的作者心得、实用的综合案例等为读者全面讲解 Linux Shell 编程。

本书的每章都配有综合案例，这些综合案例不仅可以使读者复习前面所学知识，还可以增强开发项目的经验。这些案例实用性很强，许多代码可以直接应用到 Linux 系统管理实践中。

本书附赠超大容量的 DVD 光盘，包含书中源代码、全程录像的视频讲解光盘，读者可以将视频与书配合使用，可以更快、更好地掌握 Linux Shell 编程技巧。

本书适合于 Linux Shell 编程的初学者和有一定 Linux Shell 编程基础但还需要进一步提高技能的人员。另外，本书对于有一定编程经验的程序员也有很好的参考价值。

◆ 编　　著　张　昊　程国钢
责任编辑　李永涛
责任印制　杨林杰

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路 11 号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京昌平百善印刷厂印刷

◆ 开本：787×1092　1/16
印张：21.25
字数：530 千字　　2015 年 9 月第 2 版
印数：5 001 – 7 500 册　　2015 年 9 月北京第 1 次印刷

定价：59.00 元（附光盘）

读者服务热线：(010)81055410　印装质量热线：(010)81055316
反盗版热线：(010)81055315

前　言

大一时，我刚刚开始接触 Linux。那时候的我，沉醉于 Linux 华丽的用户界面，沉醉于使用 Beryl 软件（现在叫做 Compiz Fusion）带来的图形效果，并且自我感觉良好：能够在姑娘们面前炫耀她们从没有见过的操作系统，应该算是一个计算机高手了。

直到某一天，参加一个学长（此人现在为南京大学高性能计算机研究所老师）的一个 Linux 讲座。他使用的是最简陋的图形界面（gnome 默认配置），用两台运行着 Ubuntu Linux 系统的机器和一个摄像头，各自打开一个命令行进行演示，让我目瞪口呆。首先，他将两台机器配置成联网状态，然后将一台机器（A 机器）连接摄像头，对着我们，另一台机器（B 机器） 连接投影仪，打到大屏幕上。然后，他在 A 机器的命令行中输入了一串长长的命令，又在 B 机器的命令行中输入另一串命令，按回车键。最后，我们发现现场观众的实时动画被投影到大屏幕上！现场一片哗然！他解释道，这是应用管道实现的效果。在 A 机器上用读取命令将图像从摄像头中读取出来，通过管道连接压缩程序，压缩程序将一帧一帧的图像压缩，再传输到管道中；此时管道就通过无线局域网连接到 B 机器上。B 机器上的解压程序从管道出口将压缩帧解压，通过流媒体播放器播放出来，再投放到大屏幕上！

管道！从此我爱上了黑乎乎的命令行，沉醉于它更强大的功能并且更有利于程序间的交互。这让我有种去除表象抓住实质的感觉。后来再见到 Linux 用户炫耀他们华丽的图形界面时，我的脑海中总会蹦出一个单词：Fish（菜鸟）。

的确，Linux 命令行就是 Linux 的灵魂。而用户界面只是运行在灵魂上的皮囊而已。和 Windows 的命令行不同，Linux 命令行的确是一个强大的操纵系统的工具。你可以在命令行里完成几乎一切日常操作，并且比图形界面高效和强大得多。

有许多人用 Linux 当 Windows 用，这样的人大约是得了命令行恐惧症，认为那个黑乎乎的交互界面似乎应该是一些计算机 Geek（极客）才用的。另外，有的人用了多年的 Linux 命令行还仅仅只会 ls、cp、mv 等几个简单命令，如果他的老板让他写一个 Linux Shell 脚本来完成某批处理任务，就一筹莫展了。而真正的 Linux 高手应是能够驾驭复杂的命令行和 Shell 语言的 Linux Shell 编程强人。

让我们一起走进 Linux Shell 编程的世界吧！

本书讲的是什么？

本书是 Linux Shell 编程的入门书籍。与市场上许多介绍 Linux 的书籍不同的是，这本书偏重于 Linux Shell 编程，将 Shell 当作一门语言来讲，而不是只有一两章提到 Shell。实际上，一两章是绝对不够介绍 Shell 编程的，只能算蜻蜓点水而已。

本书内容讲解全面，涵盖了 Linux Shell 编程的方方面面。

第 1 章介绍了 Shell 的一些背景知识。我们从如何运行一个 Shell 程序开始讲起，循序渐进地介绍 Shell 的一些背景知识，如 Shell 运行的环境变量、Shell 的本质等。最后，对 Shell 语言的优势进行探讨。

第 2 章是一个类似于总括的章节，主要讲解 Shell 编程的基础。包括 Shell 脚本参数的传递方式，Shell 中命令的重形象与管道，基本文本检索的方法，UNIX/Linux 系统的设计思想以及 UNIX 编程的基本原则。

第 3 章主要讲编程的基本元素。Linux Shell 编程的基本元素包括变量、函数、条件控制和流程控制，以及非常重要的循环。学习本章将会对这些元素的使用有初步的认识。

第 4 章跳出 Shell 本身的范畴，介绍了正则表达式。Shell 的强大之处在于文本处理，而正则表达式又是文本匹配的利器。关于正则表达式，除了介绍其基本知识外，还以两个案例给出了具体的应用场景，当然是在 Linux Shell 中完成。

第 5 章主要讲基本文本处理。大部分 Linux Shell 脚本都与文本处理相关，因此本章需要读者重点学习掌握。本章主要介绍一些文本处理的功能，如排序、去重、统计、打印、字段处理和文本替换。

第 6 章讲解文件和文件系统。主要介绍文件的查看、寻找与比较，还介绍了文件系统的定义与选择。

第 7 章介绍 sed。sed 也称为流编辑器，它可以对整行文本流进行处理。本章和第 8 章关系紧密，sed 和 awk 常常被一起使用。

第 8 章介绍 awk。与 sed 不同，awk 往往更善于对字段进行处理。awk 也是一门紧凑的语言，包括几乎所有语言的常见属性。

第 9 章主要介绍关于进程一些相关知识。Linux 中的进程很多，本章介绍了进程的查看与管理，进程间通信。此处举了两个例子，一个是 Linux 中的第一个进程 init，另一个是 Linux 系统中进程间管道的实现。然后介绍了 Linux 任务管理工具，最后，将 Linux 中的进程和线程做了一个比较，分析不同的应用场景。

第 10 章主要介绍 Linux 中的工具。包括不同的 Shell，远程登录的工具 SSH，管理多个终端的工具 screen，以及文本编辑工具 VIM。

第 11 章主要讲解了几个 Linux Shell 编程的实例。通过这些实例，巩固前面所学知识，并加深对 Linux Shell 编程的理解。

谁适合读这本书？

本书适合 Linux Shell 编程的初学者和有一定 Linux Shell 编程基础知识，但还希望在此领域进一步学习的人。

另外，本书还适合在 C、C++、JAVA 或 VB 等领域对其中任何一门计算机语言有所了解的专业人员、初学者或爱好者使用。

这本书能帮助你什么？

本书的目标在于，帮助一个 Linux Shell 新手掌握 Linux Shell 脚本编程，从而能更深刻地理解与应用 Linux 系统的交互方式。

当然，仅仅靠本书还是不够的，还需要读者勤加练习。

如何联系作者？

如果您有任何意见或建议，可以通过邮箱联系我们。我们的邮箱是 ollir@live.com。我们将会在第一时间给您回复。

感谢

感谢我曾经的导师和学校（南京大学），他们系统地教会我使用 Shell 编程与实用技巧。

感谢在大学阶段参与创建的一个 Linux 社团 Open Association（http://njuopen.com），是社团促进了我的成长，并带领我走进 Linux 的广袤世界。

感谢我的女朋友，她做出了一定牺牲，让我周末有时间写稿，而不是陪她逛街。

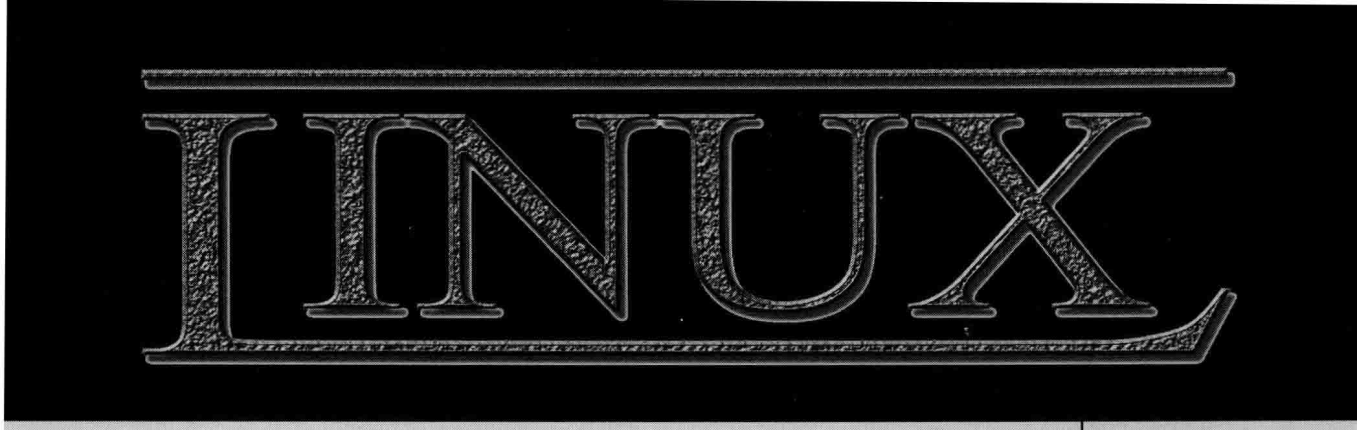
感谢马泽民、逯永广、吕平、高克臻、张云霞、张璐、许小荣、王冬、王龙、张银芳、周新国、陈可汤、陈作聪、苏静、周艳丽、祁招娣、张秀梅、张玉兰、李爽、卿前华、王文婷、肖岳平、肖斌、蔡娜等同志，他们参与了本书的编写和最终的整理。

感谢出版社对稿件的校对和发行做出了极大努力。没有他们，我不可能完成这本书。

编者

2015 年 5 月

目　录



第1章 初识Shell程序

欢迎来到 Linux Shell 编程世界。让我们开始吧。

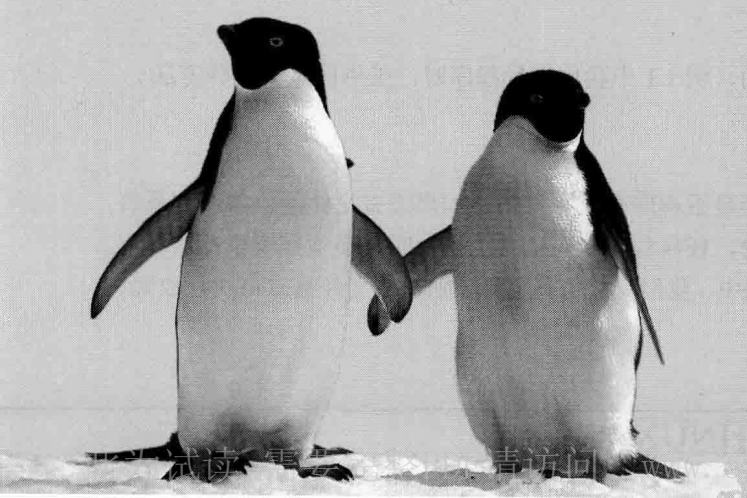
在本章中，你将会学习如下知识。

（1）编译型语言与解释型语言的差异，Linux Shell 编程的优势。

（2）如何编写和运行 Linux Shell 程序。

（3）Linux Shell 运行在环境变量中，环境变量的设置。

本章涉及的 Linux 命令有：sh，bash，echo，pwd，chmod，source，rm，more，set，unset，export 和 env。



1.1 第一道菜

许多的 UNIX 书籍的开篇都会从各种 UNIX 版本和分支讲起，内容冗长，缺少实用性，还是让我们跳过这部分吧。

我们先来看一个实例，echo.sh。

实例：echo.sh

```
1    #! /bin/sh
2    cd /tmp
3    echo"hello world!"
```

这是一个完整的，可执行的 Linux Shell 程序。

它是一个相对简单的程序，以至于你一眼就能看出它“葫芦里卖的是什么药”（如果你知道 echo 命令的话）。别急着往后跳，因为程序并不是本章的重点。

现在运行一下这个程序，看看运行结果。

例 1.1　运行实例 echo.sh

```
alloy@ubuntu:~/LinuxShell/ch1$ pwd                    #查看当前工作目录
/home/alloy/LinuxShell/ch1                            #当前工作目录
alloy@ubuntu:~/LinuxShell/ch1$ chmod +x echo.sh       #修改文件权限为可执行
alloy@ubuntu:~/LinuxShell/ch1$ ./echo.sh              #运行可执行文件
“hello world!”                                        #Shell 程序的执行结果
alloy@ubuntu:~/LinuxShell/ch1$ pwd                    #再次查看当前的工作目录
/home/alloy/LinuxShell/ch1                            #当前工作目录未发生改变
```

好，程序已经发挥作用了。很简单，不是吗？别高兴得太早，现在我要出 2 个问题，接招吧。

（1）程序第一行“#! /bin/sh”是什么意思？

（2）如何运行程序？

1.2 如何运行程序

运行 Linux 程序有 3 种方法。

（1）使文件具有可执行权限，直接运行文件。

（2）直接调用命令解释器[①]执行程序。

（3）使用 source 执行文件。

第三种方法运行结果和前两种是不同的。例 1.1 中我们运行程序时，采用的是第一种方法。

1.2.1 选婿：位于第一行的#!

当命令行 Shell 执行程序时，首先判断是否程序有执行权限。如果没有足够的权限，则系统会提示用户：“权限不够”。从安全角度考虑，任何程序要在机器上执行时，必须判断执行这个程序的用户是否具有相应权限。在第一种方法中，我们直接执行文件，则需要文件具有可执行权限。

① 参见 1.4 节“Linux Shell 是解释型语言”。

chmod 命令可以修改文件的权限。+x 参数使程序文件具有可执行权限。

命令行 Shell 接收到我们的执行命令，并且判定我们有执行权限后，则调用 Linux 内核命令新建（fork）一个进程，在新建的进程中调用我们指定的命令。如果这个命令文件是编译型的（二进制文件），则 Linux 内核知道如何执行文件。不幸的是，我们的 echo.sh 程序文件并不是编译型的文件，而是文本文件，内核并不知道如何执行，于是，内核返回“not executable format file”（不是可执行的文件类型）出错信息。Shell 收到这个信息时说：“内核不知道怎么运行，我知道，这一定是个脚本！”

Shell 知道这是个脚本后，启动了一个新的 Shell 进程来执行这个程序。但是现在的 Linux 系统往往拥有好几个 Shell，到底挑选哪个夫婿呢？这就要看脚本中意哪个了。在第一行中，脚本通过“#! /bin/sh”告诉命令行：“我只和他好，让他来执行吧！”

这种选婿方法有助于执行方式的通用化。用户在编写脚本时，在程序的第一行通过#!来设置运行 Shell 创建一个什么样的进程来执行此脚本。在我们的 echo.sh 中，Shell 创建了一个/bin/sh（标准 Shell）进程来执行脚本。

命令行在扫过第一行，发现#!时，开始试图读取#!之后的字符，搜寻解释器的完整路径。如果在第一行中的解释器也有参数，则一并读取。例如，我们可以这样来引用我们的解释器：

```
#! /bin/bash-l
```

这样，命令行 Shell 会启用一个新的 bash 进程来执行程序的每一行。并且，-l 参数使得这个 bash 进程的反应与登录 Shell 相似。

这种选婿方法，使得我们可以调用任何的解释器，并不局限于 Linux Shell。例如，我们可以创建这样一个 python[①]程序：

```
1 #! /usr/bin/python
2 print"hello world!"
```

当这个文件被赋予可执行权限，并且用第一种方式运行时，就像调用了 python 解释器来执行一样。

NOTE:

填写完整的解释器路径。如果不知道某解释器的完整路径，可使用 whereis 命令查询。

```
alloy@ubuntu:~/LinuxShell/ch1$ whereis bash
bash: /bin/bash /etc/bash.bashrc /usr/share/man/man1/bash.1.gz
```

每个脚本的头都指定了一个不同的命令解释器，为了帮助你打破#!的神秘性，我们可以这样来写一个脚本，如例 1.2 所示。

例 1.2　自删除脚本

```
1 #!/bin/rm
2 # 自删除脚本
3 # 当你运行这个脚本时，基本上什么都不会发生……当然这个文件消失不见了
4 WHATEVER=65
5 echo "This line will never print!"
6 exit $WHATEVER  # 不要紧，脚本是不会在这退出的
```

当然，你还可以试试在一个 README 文件的开头加上一个#!/bin/more，并让它具有执行权限。结果将是文档自动列出自己的内容。

① 参见 http://www.python.org。

1.2.2 找碴：程序执行的差异

3 种程序运行方法中，如果#!中指定的 Shell 解释器和第二种指定的 Shell 解释器相同的话，这两种的执行结果是相同的。我们来看看第三种方法的执行过程。

例 1.3

```
alloy@ubuntu:~/LinuxShell/ch1$ pwd                    #查看当前工作目录
/home/alloy/LinuxShell/ch1                            #当前工作目录
alloy@ubuntu:~/LinuxShell/ch1$ source echo.sh         #执行 echo.sh 文件
“hello world!”                                        #输出运行结果
alloy@ubuntu:/tmp$ pwd
/tmp                                                  #工作目录改变
```

细心的你，一定发现了不同！是的，当前目录发生了改变！

我们再来看例 1.4。

例 1.4

```
alloy@ubuntu:~/LinuxShell/ch1$ pwd                    #查看当前工作目录
/home/alloy/LinuxShell/ch1
alloy@ubuntu:~/LinuxShell/ch1$cd /tmp                 #改变当前工作目录
alloy@ubuntu:/tmp$ pwd
/tmp                                                  #工作目录改变
```

为什么例 1.3 和例 1.4 的 cd 命令可以改变工作目录，而例 1.1 中的工作目录并没有改变呢？这个问题的答案，我们将在 1.2.3 小节揭晓。

1.2.3 Shell 的命令种类

Linux Shell 可执行的命令有 3 种：内建命令、Shell 函数和外部命令。

（1）内建命令就是 Shell 程序本身包含的命令。这些命令集成在 Shell 解释器中，例如，几乎所有的 Shell 解释器中都包含 cd 内建命令来改变工作目录。部分内建命令的存在是为了改变 Shell 本身的属性设置，在执行内建命令时，没有进程的创建和消亡；另一部分内建命令则是 I/O 命令，例如 echo 命令。

（2）Shell 函数是一系列程序代码，以 Shell 语言写成，它可以像其他命令一样被引用。我们在后面将详细介绍 Shell 函数。

（3）外部命令是独立于 Shell 的可执行程序。例如 find、grep、echo.sh。命令行 Shell 在执行外部命令时，会创建一个当前 Shell 的复制进程来执行。在执行过程中，存在进程的创建和消亡。外部命令的执行过程如下：

① 调用 POSIX 系统 fork 函数接口，创建一个命令行 Shell 进程的复制（子进程）；

② 在子进程的运行环境中，查找外部命令在 Linux 文件系统中的位置。如果外部命令给出了完全路径，则跳过查找这一步；

③ 在子进程里，以新程序取代 Shell 复制并执行（exec），此时父进程进入休眠，等待子进程执行完毕；

④ 子进程执行完毕后，父进程接着从终端读取下一条命令。过程如图 1-1 所示。

NOTE:

（1）子进程在创建初期和父进程一模一样，但是子进程不能改变父进程的参数变量。

（2）只有内建命令才能改变命令行 Shell 的属性设置（环境变量）。

我们回到例 1.1。在这个例子中，我们使用 cd（内建命令）试图改变工作目录。但是未获成功。为了理解失败的原因，图 1-2 说明了执行的过程。

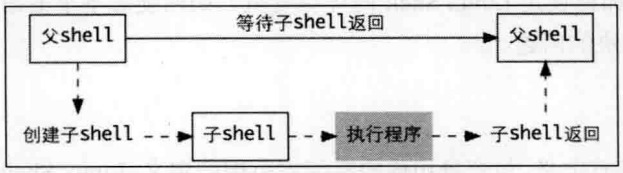


图 1-1　创建进程

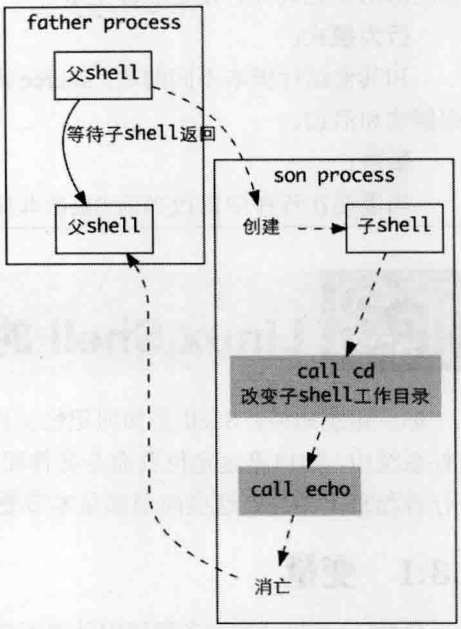


图 1-2　echo.sh 的执行过程

在我们运行 Shell 程序的 3 种方法中，前两种方法的执行过程都可以用图 1-2 解释。

（1）父进程接收到命令“./echo.sh”或“/bin/sh echo.sh”，发现不是内建命令，于是创建了一个和自己一模一样的 Shell 进程来执行这个外部命令。

（2）这个 Shell 子进程用/bin/sh 取代自己，sh 进程设置自己运行环境变量，其中包括$PWD 变量（标识当前工作目录）。

（3）sh 进程依次执行内建命令 cd 和 echo，在此过程中，sh 进程（子进程）的环境变量$PWD 被 cd 命令改变，注意：父进程的环境变量并没有改变。

（4）sh 子进程执行完毕，消亡。一直在等待的父进程醒来继续接收命令。

这样，例 1.1 中 cd 命令失效的原因就可以理解了！聪明的你，一定猜到了例 1.3 中使用 source 命令为什么可以改变命令行 Shell 的环境变量了吧！

这也是在例 1.1 中目录没有改变的原因：父进程的当前目录（环境变量）无法被子进程改变！

NOTE:

使用 source 执行 Shell 脚本时，不会创建子进程，而是在父进程中直接执行！

---

source

语法：

```
source file
. file
```

描述：

使用 Shell 进程本身执行脚本文件。souce 命令也被称为“点命令”，通常用于重新执行刚修改的初始化文件。使之立即生效。

行为模式：

和其他运行脚本不同的是，source 命令影响 Shell 进程本身。在脚本执行过程中，并没有进程创建和消亡。

警告：

当需要在程序中修改当前 Shell 本身环境变量时，使用 source 命令。

1.3 Linux Shell 的变量

你一定想知道，Shell 是如何记忆工作目录的改变的？当 Shell 接收到外部命令时，在庞大的文件系统中，如何迅速定位到命令文件呢？如何设定 Linux Shell 的环境变量？如何使黑乎乎的命令行看起来更漂亮？这些问题都是本节要解决的问题。

1.3.1 变量

变量(variable)在许多程序设计语言中都有定义，与变量相伴的有使用范围的定义。Linux Shell 也不例外。变量，本质上就是一个键值对。例如，str = “hello”，就是将字符串值（value）“hello”赋予键（key）str。在 str 的使用范围内，我们都可以用 str 来引用“hello”值，这个操作叫做变量替换。

Shell 变量的名称以一个字母或下划线符号开始，后面可以接任意长度的字母、数字或下划线。和许多其他程序设计语言不同的是，Shell 变量名称字符并没有长度限制。Linux Shell 并不对变量区分类型。一切值都是字符串，并且和变量名一样，值并没有字符长度限制。神奇的是，bash 也允许比较操作和整数操作。其中关键因素是：变量中的字符串值是否为数字。例如 1.5 所示。

例 1.5　Linux Shell 中的变量

```
alloy@ubuntu:~/LinuxShell/ch1$ long_str="Linux_Shell_programming"
alloy@ubuntu:~/LinuxShell/ch1$ echo $long_str
Linux_Shell_programming
alloy@ubuntu:~/LinuxShell/ch1$ add_1=100
alloy@ubuntu:~/LinuxShell/ch1$ add_2=200
alloy@ubuntu:~/LinuxShell/ch1$ echo $(($add_1+$add_2))
300
```

由例 1.5 可见，虽然 Linux Shell 中的变量都是字符串类型的，但是同样可以执行比较操作和整数操作，只要变量字符串值是数字。

变量赋值的方式为：变量名称=值，其中“=”两边不要有任何空格。当你想使用变量名称来获得值时，在名称前加上“$”。例如，$long_str。当赋值的内容包含空格时，请加引号，例如：

```
alloy@ubuntu:~/LinuxShell/ch1$ with_space="this contains spaces."
alloy@ubuntu:~/LinuxShell/ch1$ echo $with_space
This contains spaces
```

注意：$with_space 事实上只是${with_space}的简写形式，在某些上下文中$with_space 可能会引起错误，这时候你就需要用${with_space}了。

当变量“裸体”出现的时候（没有$前缀的时候），变量可能存在如下几种情况：变量被声明或被赋值；变量被 unset；或者变量被 export。

变量赋值可以使用“=”（比如 var=27），也可以在 read 命令中或者循环头进行赋值，例如，for var2 in 1 2 3。

被一对双引号（" "）括起来的变量替换是不会被阻止的。所以双引号被称为部分引用，有时候又被称为“弱引用”。但是如果使用单引号的话（' '），那么变量替换就会被禁止了，变量名只会被解释成字面的意思，不会发生变量替换。所以单引号被称为“全引用”，有时候也被称为“强引用”。例如：

```
alloy@ubuntu:~/LinuxShell/ch1$var=123
alloy@ubuntu:~/LinuxShell/ch1$ echo '$var'      #此处是单引号
$var
alloy@ubuntu:~/LinuxShell/ch1$ echo "$var"      #此处是双引号
123
```

在这个例子中，单引号中的$var 没有替换成变量值 123，也就是说，变量替换被禁止了；而双引号中的$var 发生了变量替换。即：单引号为全引用（强应用），双引号为弱引用。

在 Shell 的世界里，变量值可以是空值（“NULL”值），就是不包含任何字符。这种情况很常见，并且也是合理的。但是在算术操作中，这个未初始化的变量常常看起来是 0。但是这是一个未文档化（并且可能是不可移植）的行为。例如：

```
alloy@ubuntu:~/LinuxShell/ch1$ echo "$uninit"              #未初始化变量
                                                           #此行为空，没有输出
alloy@ubuntu:~/LinuxShell/ch1$ let "uninit+= 5"            #未初始化变量加 5
alloy@ubuntu:~/LinuxShell/ch1$ echo "$uninit"
5                                                  #此行输出结果为 5
alloy@ubuntu:~/LinuxShell/ch1$
```

Linux Shell 中的变量类型有两种：局部变量和全局变量。

- 顾名思义，局部变量的可见范围是代码块或函数中。这一点与大部分编程语言是相同的。但是，局部变量必须明确以 local 声明，否则即使在代码块中，它也是全局可见的。
- 环境变量是全局变量的一种。全局变量在全局范围内可见，在声明全局变量时，不需要加任何修饰词。

例 1.6　测试全局变量和局部变量的适用范围

```
1  #! /bin/sh
2  # 测试全局变量和局部变量的适用范围
3  num=123
4  func1 ()
5  {
6  num=321                          #在代码块中声明的变量
7  echo $num
8  }
9  func2()
10 {
11  local num=456                   #声明为局部变量
12 echo $num
13 }
14 echo $num                        #显示初始时的 num 变量
15 func1                           #调用 func1，在函数体中赋值（声明？）变量
16 echo $num                       #测试 num 变量是否被改变
```