

第 1 章：Shell 基础（开胃菜）

1. Shell 是什么？1 分钟理解 Shell 的概念！

现在我们使用的操作系统（Windows、Mac OS、[Android](#)、iOS 等）都是带图形界面的，简单直观，容易上手，对专业用户（程序员、网管等）和普通用户（家庭主妇、老年人等）都非常适用；计算机的普及离不开图形界面。

然而在计算机的早期并没有图形界面，我们只能通过一个一个地命令来控制计算机，这些命令有成百上千之多，且不说记住这些命令非常困难，每天面对没有任何色彩的“黑屏”本身就是一件枯燥的事情；这个时候的计算机还远远谈不上炫酷和普及，只有专业人员才能使用。



图：早期的电脑，都是“黑纸白字”

猛击 [《带你逛西雅图活电脑博物馆》](#) 可以欣赏更多早期的计算机。

对于图形界面，用户点击某个图标就能启动某个程序；对于命令行，用户输入某个程序的名字（可以看做一个命令）就能启动某个程序。这两者的基本过程都是类似的，都需要查找程序在硬盘上的安装位置，然后将它们加载到内存运行。

关于程序的运行原理，请猛击 [《载入内存，让程序运行起来》](#)。

换句话说，图形界面和命令行要达到的目的是一样的，都是让用户控制计算机。

然而，真正能够控制计算机硬件（CPU、内存、显示器等）的只有操作系统内核（Kernel），图形界面和命令行只是架设在用户和内核之间的一座桥梁。

如果你不了解操作系统的作用，请转到 [《操作系统是什么》](#)。

由于安全、复杂、繁琐等原因，用户不能直接接触内核（也没有必要），需要另外再开发一个程序，让用户直接使用这个程序；该程序的作用就是接收用户的操作（点击图标、输入命令），并进行简单的处理，然后再传递给内核，这样用户就能间接地使用操作系统内核了。你看，在用户和内核之间增加一层“代理”，既能简化用户的操作，又能保障内核的安全，何乐不为呢？

用户界面和命令行就是这个另外开发的程序，就是这层“代理”。在 Linux 下，这个命令行程序叫做 **Shell**。

Shell 是一个应用程序，它连接了用户和 Linux 内核，让用户能够更加高效、安全、低成本地使用 Linux 内核，这就是 Shell 的本质。

Shell 本身并不是内核的一部分，它只是站在内核的基础上编写的一个应用程序，它和 QQ、迅雷、Firefox 等其它软件没有什么区别。然而 Shell 也有着它的特殊性，就是开机立马启动，并呈现在用户面前；用户通过 Shell 来使用 Linux，不启动 Shell 的话，用户就没办法使用 Linux。

Shell 是如何连接用户和内核的？

Shell 能够接收用户输入的命令，并对命令进行处理，处理完毕后再将结果反馈给用户，比如输出到显示器、写入到文件等，这就是大部分读者对 Shell 的认知。你看，我一直都在使用 Shell，哪有使用内核哦？我也没有看到 Shell 将我和内核连接起来呀？！

其实，Shell 程序本身的功能是很弱的，比如文件操作、输入输出、进程管理等都得依赖内核。我们运行一个命令，大部分情况下 Shell

都会去调用内核暴露出来的接口，这就是在使用内核，只是这个过程被 Shell 隐藏了起来，它自己在背后默默进行，我们看不到而已。

接口其实就是一个一个的函数，使用内核就是调用这些函数。这就是使用内核的全部内容了吗？嗯，是的！除了函数，你没有别的途径使用内核。

比如，我们都知道在 Shell 中输入 `cat log.txt` 命令就可以查看 log.txt 文件中的内容，然而，log.txt 放在磁盘的哪个位置？分成了几个数据块？在哪里开始？在哪里终止？如何操作探头读取它？这些底层细节 Shell 统统不知道的，它只能去调用内核提供的 `open()` 和 `read()` 函数，告诉内核我要读取 log.txt 文件，请帮助我，然后内核就乖乖地按照 Shell 的吩咐去读取文件了，并将读取到的文件内容交给 Shell，最后再由 Shell 呈现给用户（其实呈现到显示器上还得依赖内核）。整个过程中 Shell 就是一个“中间商”，它在用户和内核之间“倒卖”数据，只是用户不知道罢了。

Shell 还能连接其它程序

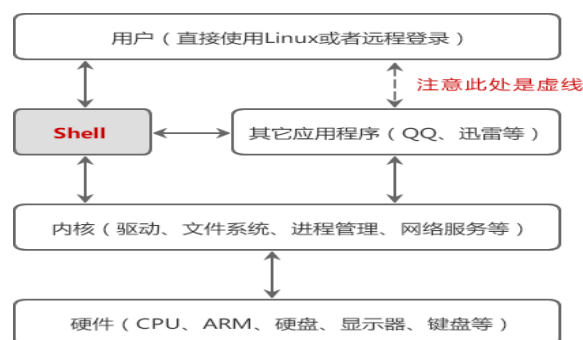
在 Shell 中输入的命令，有一部分是 Shell 本身自带的，这叫做[内置命令](#)；有一部分是其它的应用程序（一个程序就是一个命令），这叫做外部命令。

Shell 本身支持的命令并不多，功能也有限，但是 Shell 可以调用其他的程序，每个程序就是一个命令，这使得 Shell 命令的数量可以无限扩展，其结果就是 Shell 的功能非常强大，完全能够胜任 Linux 的日常管理工作，如文本或字符串检索、文件的查找或创建、大规模软件的自动部署、更改系统设置、监控服务器性能、发送报警邮件、抓取网页内容、压缩文件等。

更加惊讶的是，Shell 还可以让多个外部程序发生连接，在它们之间很方便地传递数据，也就是把一个程序的输出结果传递给另一个程序作为输入。

大家所说的 Shell 强大，并不是 Shell 本身功能丰富，而是它擅长使用和组织其他的程序。Shell 就是一个领导者，这正是 Shell 的魅力所在。

可以将 Shell 在整个 Linux 系统中的地位描述成下图所示的样子。注意“用户”和“其它应用程序”是通过虚线连接的，因为用户启动 Linux 后直接面对的是 Shell，通过 Shell 才能运行其它的应用程序。



Shell 也支持编程

Shell 并不是简单的堆砌命令，我们还可以在 Shell 中编程，这和使用 [C++](#)、[C#](#)、[Java](#)、[Python](#) 等常见的编程语言并没有什么两样。

Shell 虽然没有 C++、Java、Python 等强大，但也支持了基本的编程元素，例如：

- if...else 选择结构，case...in 开关语句，for、while、until 循环；
- 变量、数组、字符串、注释、加减乘除、逻辑运算等概念；
- 函数，包括用户自定义的函数和内置函数（例如 `printf`、`export`、`eval` 等）。

站在这个角度讲，Shell 也是一种编程语言，它的编译器（解释器）是 Shell 这个程序。我们平时所说的 Shell，有时候是指连接用户和内核的这个程序，有时候又是指 Shell 编程。

Shell 主要用来开发一些实用的、自动化的小工具，而不是用来开发具有复杂业务逻辑的中大型软件，例如检测计算机的硬件参数、搭建 Web 运行环境、日志分析等，Shell 都非常合适。

使用 Shell 的熟练程度反映了用户对 Linux 的掌握程度，运维工程师、网络管理员、程序员都应该学习 Shell。

尤其是 Linux 运维工程师，Shell 更是必不可少的，是必须掌握的技能，它使得我们能够自动化地管理服务器集群，否则你就得一个一个地登录所有的服务器，对每一台服务器都进行相同的设置，而这些服务器可能有成百上千之多，会浪费大量的时间在重复性的工作上。

Shell 是一种脚本语言

任何代码最终都要被“翻译”成二进制的形式才能在计算机中执行。

有的编程语言，如 C/C++、Pascal、Go 语言、汇编等，必须在程序运行之前将所有代码都翻译成二进制形式，也就是生成可执行文件，用户拿到的是最终生成的可执行文件，看不到源码。

这个过程叫做**编译（Compile）**，这样的编程语言叫做**编译型语言**，完成编译过程的软件叫做**编译器（Compiler）**。

而有的编程语言，如 Shell、[JavaScript](#)、Python、[PHP](#) 等，需要一边执行一边翻译，不会生成任何可执行文件，用户必须拿到源码才能运行程序。程序运行后会即时翻译，翻译完一部分执行一部分，不用等到所有代码都翻译完。

这个过程叫做**解释**，这样的编程语言叫做**解释型语言**或者**脚本语言（Script）**，完成解释过程的软件叫做**解释器**。

编译型语言的优点是执行速度快、对硬件要求低、保密性好，适合开发操作系统、大型应用程序、数据库等。

脚本语言的优点是使用灵活、部署容易、跨平台性好，非常适合 Web 开发以及小工具的制作。

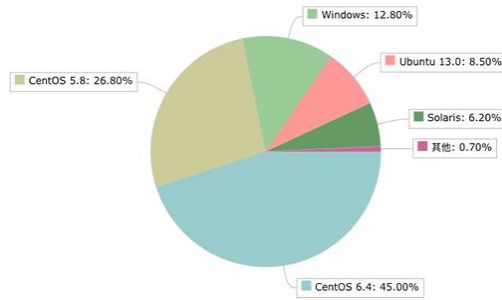
Shell 就是一种脚本语言，我们编写完源码后不用编译，直接运行源码即可。

2. Shell 是运维人员必须掌握的技能

Linux 运维人员就是负责 Linux 服务器的运行和维护。随着互联网的爆发，Linux 运维在最近几年也迎来了春天，出现了大量的职位需求，催生了一批 Linux 运维培训班。

如今的 IT 服务器领域是 Linux、UNIX、Windows 三分天下，Linux 系统可谓后起之秀，特别是“互联网热”以来，Linux 在服务器端的市场份额不断扩大，每年增长势头迅猛，开始对 Windows 和 UNIX 的地位构成严重威胁。

下图是 2016 年初国内服务器端各个操作系统的市场份额：



可以看出来，Linux 占 80% 左右（包括 CentOS、Ubuntu 等），Windows 占 12.8%，Solaris 占 6.2%。在未来的服务器领域，Linux 是大势所趋。

Linux 在服务器上的应用非常广泛，可以用来搭建 Web 服务器、数据库服务器、负载均衡服务器（CDN）、邮件服务器、DNS 服务器、反向代理服务器、VPN 服务器、路由器等。用 Linux 作为服务器系统不但非常高效和稳定，还不用担心版权问题，不用付费。

正是由于 Linux 服务器的大规模应用，才需要一批专业的人才去管理，这群人就是 **Linux 运维工程师（OPS）**。

OPS 的主要工作就是搭建起运行环境，让程序员写的代码能够高效、稳定、安全地在服务器上运行，他们属于后勤部门。OPS 的要求并不比程序员低，优秀的 OPS 拥有架设服务器集群的能力，还会编程开发常用的工具。

OPS 这项工作的细节内容包括：

- 安装操作系统，例如 CentOS、Ubuntu 等。
- 部署代码运行环境，例如网站后台语言采用 [PHP](#)，就需要安装 Nginx、Apache、[MySQL](#)、PHP 运行时等。
- 及时修复漏洞，防止服务器被攻击，这包括 Linux 本身漏洞以及各个软件的漏洞。
- 根据项目需求升级软件，例如 PHP 7.0 在性能方面获得了重大突破，如果现在服务器压力比较大，就可以考虑将旧版的 PHP 5.x 升级到 PHP 7.0。
- 监控服务器压力，别让服务器宕机。例如淘宝双十一的时候就会瞬间涌入大量用户，导致部分服务器宕机，网页没法访问，甚至连支付宝都不能使用。
- 分析日志，及时发现代码或者环境的问题，通知相关人员修复。

这些任务只要登录远程服务器，或者去机房连接服务器（下图所示）就能够完成，为什么要用 Shell 编程呢？



图：OPS 在机房中用笔记本连接服务器

因为 OPS 面对的是成千上万台的服务器，不是十台八台，你总不能把同样的工作重复成千上万遍吧，那时估计黄花菜都凉了，市场也成一片红海了。

服务器一旦多了，这些人力工作都需要自动化起来，跑一段代码就能在成千上万台服务器上完成相同的工作，例如服务的监控、代码快速部署、服务启动停止、数据备份、日志分析等。

Shell 脚本很适合处理纯文本类型的数据，而 Linux 中几乎所有的配置文件、日志文件（如 NFS、Rsync、Httpd、Nginx、MySQL 等），以及绝大多数的启动文件都是纯文本类型的文件。

下面的手链形象地展示了 Shell 在运维工作中的地位：



运维“手链”的组成：每颗“珍珠”都是一项服务，将珍珠穿起来的“线”就是 Shell。

Shell 脚本是实现 Linux 系统自动管理以及自动化运维所必备的工具，Linux 的底层以及基础应用软件的核心大都涉及 Shell 脚本的内容。每一个合格的 Linux 系统管理员或运维工程师，都应该能够熟练的编写 Shell 脚本，只要这样才能提升运维人员的工作效率，减少不必要的重复劳动，为个人的职场发展奠定较好的基础。

Shell、Python 和 Perl

除了 Shell，能够用于 Linux 运维的脚本语言还有 Python 和 Perl。

1) Perl 语言

Perl 比 Shell 强大很多，在 2010 年以前很流行，它的语法灵活、复杂，在实现不同的功能时可以用多种不同的方式，缺点是不易读，团队协作困难。

Perl 脚本已经成为历史了，现在的 Linux 运维人员几乎不需要了解 Perl 了，最多可以了解一下 Perl 的安装环境。

2) Python 语言

Python 是近几年非常流行的语言，它不但可以用于脚本程序开发，也可以实现 Web 程序开发（知乎、豆瓣、YouTube、Instagram 都是用 Python 开发），甚至还可以实现软件的开发（大名鼎鼎的 OpenStack、SaltStack 都是 Python 语言开发）、游戏开发、[大数据](#)开发、移动端开发。

现在越来越多的公司要求运维人员会 Python 自动化开发，Python 也成了运维人员必备的技能，每一个运维人员在熟悉了 Shell 之后，都应该再学习 Python 语言。

3) Shell

Shell 脚本的优势在于处理偏操作系统底层的业务，例如，Linux 内部的很多应用（有的是应用的一部分）都是使用 Shell 脚本开发的，因为有 1000 多个 Linux 系统命令为它作支撑，特别是 Linux 正则表达式以及三剑客 grep、awk、sed 等命令。

对于一些常见的系统脚本，使用 Shell 开发会更简单、更快速，例如，让软件一键自动化安装、优化，监控报警脚本，软件启动脚本，日志分析脚本等，虽然 Python 也能做到这些，但是考虑到掌握难度、开发效率、开发习惯等因素，它们可能就不如 Shell 脚本流行以及有优势了。对于一些常见的业务应用，使用 Shell 更符合 Linux 运维**简单、易用、高效**的三大原则。

Python 语言的优势在于开发**复杂的运维软件**、Web 页面的管理工具和 Web 业务的开发（例如 CMDB 自动化运维平台、跳板机、批量管理软件 SaltStack、[云计算](#) OpenStack 软件）等。

我们在开发一个应用时，应该根据业务需求，结合不同语言的优势以及自己擅长的语言来选择，扬长避短，从而达到高效开发、易于自己维护的目的。

3. 常用的 Shell 有哪些？

Linux 是一个开源的操作系统，由分布在世界各地的多个组织机构或个人共同开发完成，每个组织结构或个人负责一部分功能，最后组合在一起，就构成了今天的 Linux。例如：

- Linux 内核最初由芬兰黑客 Linus Torvalds 开发，后来他组建了团队，Linux 内核由这个团队维护。
- GNU 组织开发了很多核心软件和基础库，例如 GCC 编译器、C 语言标准库、文本编辑器 Emacs、进程管理软件、Shell 以及 GNOME 桌面环境等。
- VIM 编辑器由荷兰人 Bram Moolenaar 开发。

Windows、Mac OS、Android 等操作系统不一样，它们都由一家公司开发，所有的核心软件和基础库都由一家公司做决定，容易形成统一的标准，一般不会开发多款功能类似的软件。

而 Linux 不一样，它是“万国牌”，由多个组织机构开发，不同的组织机构为了发展自己的 Linux 分支可能会开发出功能类似的软件，它们各有优缺点，用户可以自由选择。Shell 就是这样的一款软件，不同的组织机构开发了不同的 Shell，它们各有所长，有的占用资源少，有的支持高级编程功能，有的兼容性好，有的重视用户体验。

Shell 既是一种脚本编程语言，也是一个连接内核和用户的软件。

常见的 Shell 有 sh、bash、csh、tcsh、ash 等。

sh

sh 的全称是 Bourne shell，由 AT&T 公司的 Steve Bourne 开发，为了纪念他，就用他的名字命名了。

sh 是 UNIX 上的标准 shell，很多 UNIX 版本都配有 sh。sh 是第一个流行的 Shell。

csh

sh 之后另一个广为流传的 shell 是由柏克莱大学的 Bill Joy 设计的，这个 shell 的语法有点类似 C 语言，所以才得名为 C shell，简称为 csh。

Bill Joy 是一个风云人物，他创立了 BSD 操作系统，开发了 vi 编辑器，还是 Sun 公司的创始人之一。

BSD 是 UNIX 的一个重要分支，后人在此基础上发展出了很多现代的操作系统，最著名的有 FreeBSD、OpenBSD 和 NetBSD，就连 Mac OS X 在很大程度上也基于 BSD。

tcsh

tcsh 是 csh 的增强版，加入了命令补全功能，提供了更加强大的语法支持。

ash

一个简单的轻量级的 Shell，占用资源少，适合运行于低内存环境，但是与下面讲到的 bash shell 完全兼容。

bash

bash shell 是 Linux 的默认 shell，本教程也基于 bash 编写。

bash 由 GNU 组织开发，保持了对 sh shell 的兼容性，是各种 Linux 发行版默认配置的 shell。

bash 兼容 sh 意味着，针对 sh 编写的 Shell 代码可以不加修改地在 bash 中运行。

尽管如此，bash 和 sh 还是有一些不同之处：

- 一方面，bash 扩展了一些命令和参数；
- 另一方面，bash 并不完全和 sh 兼容，它们有些行为并不一致，但在大多数企业运维的情况下区别不大，特殊场景可以使用 bash 代替 sh。

查看 Shell

Shell 是一个程序，一般都是放在 `/bin` 或者 `/usr/bin` 目录下，当前 Linux 系统可用的 Shell 都记录在 `/etc/shells` 文件中。`/etc/shells` 是一个纯文本文件，你可以在图形界面下打开它，也可以使用 `cat` 命令查看它。

通过 `cat` 命令来查看当前 Linux 系统的可用 Shell：

```
$ cat /etc/shells
/bin/sh
/bin/bash
/sbin/nologin
/usr/bin/sh
/usr/bin/bash
/usr/sbin/nologin
/bin/tcsh
/bin/csh
```

在现代的 Linux 上，sh 已经被 bash 代替，`/bin/sh` 往往是指向 `/bin/bash` 的符号链接。

如果你希望查看当前 Linux 的默认 Shell，那么可以输出 SHELL 环境变量：

```
$ echo $SHELL
/bin/bash
```

输出结果表明默认的 Shell 是 bash。

`echo` 是一个 Shell 命令，用来输出变量的值，我们将在《[Shell echo](#)》一节中详细介绍它的用法。`SHELL` 是 Linux 系统中的环境变量，它指明了当前使用的 Shell 程序的位置，也就是使用的哪个 Shell。

4. 进入 Shell 的两种方式

在 Linux 发展的早期，唯一能用的工具就是 Shell，Linux 用户都是在 Shell 中输入文本命令，并查看文本输出；如果有必要的话，Shell 也能显示一些基本的图形。

而如今 Linux 的环境已经完全不同，几乎所有的 Linux 发行版都使用某种图形桌面环境（例如 GNOME、KDE、Unity 等），这使得原生的 Shell 入口被隐藏了，进入 Shell 仿佛变得困难起来。

进入 Linux 控制台

一种进入 Shell 的方法是让 Linux 系统退出图形界面模式，进入控制台模式，这样一来，显示器上只有一个简单的带着白色文字的“黑屏”，就像图形界面出现之前的样子。这种模式称为 **Linux 控制台 (Console)**。

现代 Linux 系统在启动时会自动创建几个虚拟控制台（Virtual Console），其中一个供图形桌面程序使用，其他的保留原生控制台的样子。虚拟控制台其实就是 Linux 系统内存中运行的虚拟终端（Virtual Terminal）。

从图形界面模式进入控制台模式也很简单，往往按下 `Ctrl + Alt + Fn(n=1,2,3,4,5...)` 快捷键就能够来回切换。

例如，CentOS 在启动时会创建 6 个虚拟控制台，按下快捷键 `Ctrl + Alt + Fn(n=2,3,4,5,6)` 可以从图形界面模式切换到控制台模式，按下 `Ctrl + Alt + F1` 可以从控制台模式再切换回图形界面模式。也就是说，1 号控制台被图形桌面程序占用了。

下图就是进入了控制台模式：



```
CentOS Linux 7 (Core)
Kernel 3.10.0-123.el7.x86_64 on an x86_64

localhost login: mozhayan → 输入用户名
Password: → 输入密码
Last login: Mon Jul 3 17:42:08 on tty5
mozhayan@localhost ~]$ → Shell命令提示符
```

输入用户名和密码，登录成功后就可以进入 Shell 了。`$` 是命令提示符，我们可以在它后面输入 Shell 命令。

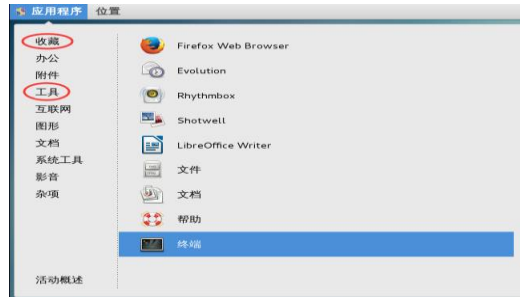
在图形界面模式下，输入密码时往往会显示为 *，密码有几个字符就显示几个 *；而在控制台模式下，输入密码什么都不会显示，好像按键无效一样，这一点请大家不要惊慌，只要输入的密码正确就能够登录。

图形界面也是一个程序，会占用 CPU 时间和内存空间，当 Linux 作为服务器系统时，安装调试完毕后，应该让 Linux 运行在控制台模式下，以节省服务器资源。正是由于这个原因，很多服务器甚至不安装图形界面程序，管理员只能使用命令来完成各项操作。

使用终端

进入 Shell 的另外一种方法是使用 **Linux 桌面环境** 中的终端模拟包（Terminal emulation package），也就是我们常说的 **终端 (Terminal)**，这样在图形桌面中就可以使用 Shell。

以 CentOS 为例，可以在“应用程序”菜单中找到终端，如下图所示：



图：在“收藏”和“工具”分类中都可以找到终端

打开终端后，就可以输入 Shell 命令了：



CentOS 默认的图形界面程序是 GNOME，该终端模拟包也是 GNOME 自带的。

除了 GNOME 终端，Linux 还有其他的终端模拟包，例如：

- **xterm 终端**

最古老最基础的 X Windows 桌面程序自带的终端模拟包就是 xterm。xterm 在 X Windows 出现之前便已经存在了，默认包含在大多数 X Windows 中。xterm 虽然没有太多炫目的特性，但是运行它不需要太多的资源，所以 xterm 在针对老硬件设计的 Linux 发行版中仍然很常见，比如 fluxbox 图形桌面环境就用它作为默认的终端模拟包。

- **Konsole 终端**

KDE 桌面项目也开发了自己的终端模拟包，名为 Konsole。Konsole 整合了基本的 xterm 特性以及一些更高级的类似 Windows 应用程序的特性。

5. Linux Shell 命令的基本格式

进入 Shell 以后，我们就可以输入命令来使用 Linux 的各种功能了，但是在真正使用 Shell 命令之前，我们有必要先学习一下 Shell 命令的基本格式。

[进入 Shell](#) 之后第一眼看到的内容类似下面这种形式：

```
[mozhiyan@localhost ~]$
```

这叫做命令提示符，看见它就意味着可以输入命令了。命令提示符不是命令的一部分，它只是起到一个提示作用，我们将在《[Shell 命令提示符](#)》一节中详细分析，本节只分析 Shell 命令的基本格式。

Shell 命令的基本格式如下：

```
command [选项] [参数]
```

`[]` 表示可选的，也就是可有可无。有些命令不写选项和参数也能执行，有些命令在必要的时候可以附带选项和参数。

`ls` 是常用的一个命令，它属于目录操作命令，用来列出当前目录下的文件和文件夹。`ls` 可以附带选项，也可以不带，不带选项的写法为：

```
[mozhiyan@localhost ~]$ cd demo
[mozhiyan@localhost demo]$ ls
abc          demo.sh    a.out      demo.txt
getsum       main.sh    readme.txt a.sh
module.sh    log.txt    test.sh    main.c
```

先执行 `cd demo` 命令进入 `demo` 目录，这是我在自己的主目录下创建的文件夹，用来保存教学使用的各种代码和数据。

接着执行 `ls` 命令，它列出了 `demo` 目录下的所有文件，并且进行了格式对齐。

使用选项

`ls` 命令之后不加选项和参数也能执行，不过只能执行最基本的功能，即显示当前目录下的文件名。那么加入一个选项，会出现什么结果？

```
[mozhiyan@localhost demo]$ ls -l
总用量 140
-rwxrwxr-x. 1 mozhiyan mozhiyan 8675 4月  2 15:01 a.out
-rwxr-xr-x. 1 mozhiyan mozhiyan 116 4月  3 09:24 a.sh
-rw-rw-r--. 1 mozhiyan mozhiyan  44 4月  2 16:41 check.sh
-rw-r--r--. 1 mozhiyan mozhiyan 399 3月 11 17:12 demo.sh
-rw-rw-r--. 1 mozhiyan mozhiyan   4 4月  8 17:56 demo.txt
-rw-rw-r--. 1 mozhiyan mozhiyan   0 4月 15 17:26 log.txt
-rw-rw-r--. 1 mozhiyan mozhiyan 650 4月 10 11:06 main.c
```

```
-rwxrwxr-x. 1 mozhiyan mozhiyan 69 3月 26 10:13 main.sh

-rw-rw-r--. 1 mozhiyan mozhiyan 111 3月 26 09:56 module.sh

-rw-rw-r--. 1 mozhiyan mozhiyan 352 3月 22 17:40 out.log

-rw-rw-r--. 1 mozhiyan mozhiyan 61 4月 16 11:19 output.txt

-rw-r--r--. 1 mozhiyan mozhiyan 5 4月 11 15:16 readme.txt

-rwxr-xr-x. 1 mozhiyan mozhiyan 88 4月 15 17:23 test.sh
```

如果加一个 `-l` 选项，则可以看到显示的内容明显增多了。`-l` 是长格式 (long list) 的意思，也就是显示文件的详细信息。

可以看到，选项的作用是调整命令功能。如果没有选项，那么命令只能执行最基本的功能；而一旦有选项，则能执行更多功能，或者显示更加丰富的数据。

短格式选项和长格式选项

Linux 的选项又分为短格式选项和长格式选项。

- 短格式选项是长格式选项的简写，用一个减号 `-` 和一个字母表示，例如 `ls -l`。
- 长格式选项是完整的英文单词，用两个减号 `--` 和一个单词表示，例如 `ls --all`。

一般情况下，短格式选项是长格式选项的缩写，也就是一个短格式选项会有对应的长格式选项。当然也有例外，比如 `ls` 命令的短格式选项 `-l` 就没有对应的长格式选项，所以具体的命令选项还需要通过帮助手册来查询。

使用参数

参数是命令的操作对象，一般情况下，文件、目录、用户和进程等都可以作为参数被命令操作。例如：

```
[mozhiyan@localhost demo]$ ls -l main.c

-rw-rw-r--. 1 mozhiyan mozhiyan 650 4月 10 11:06 main.c
```

但是为什么一开始 `ls` 命令可以省略参数？那是因为默认参数。命令一般都需要加入参数，用于指定命令操作的对象是谁。如果可以省略参数，则一般都有默认参数。例如 `ls`：

```
[mozhiyan@localhost ~]$ cd demo

[mozhiyan@localhost demo]$ ls

abc          demo.sh    a.out      demo.txt

getsum       main.sh    readme.txt  a.sh

module.sh    log.txt    test.sh     main.c
```

这个 `ls` 命令后面如果没有指定参数的话，默认参数是当前所在位置，所以会显示当前目录下的文件名。

选项和参数一起使用

Shell 命令可以同时附带选项和参数，例如：

```
[mozhiyan@localhost ~]$ echo "http://c.biancheng.net/shell/"
http://c.biancheng.net/shell/

[mozhiyan@localhost ~]$ echo -n "http://c.biancheng.net/shell/"
http://c.biancheng.net/shell/[mozhiyan@localhost ~]$
```

`-n` 是 `echo` 命令的选项，`"http://c.biancheng.net/shell/"` 是 `echo` 命令的参数，它们被同时用于 `echo` 命令。

`echo` 命令用来输出一个字符串，默认输出完成后会换行；给它增加 `-n` 选项，就不会换行了。

选项附带的参数

有些命令的选项后面也可以附带参数，这些参数用来补全选项，或者调整选项的功能细节。

例如，`read` 命令用来读取用户输入的数据，并把读取到的数据赋值给一个变量，它通常的用法为：

```
read str
```

`str` 为变量名。

如果我们只是想读取固定长度的字符串，那么可以给 `read` 命令增加 `-n` 选项。比如读取一个字符作为性别的标志，那么可以这样写：

```
read -n 1 sex
```

`1` 是 `-n` 选项的参数，`sex` 是 `read` 命令的参数。

`-n` 选项表示读取固定长度的字符串，那么它后面必然要跟一个数字用来指明长度，否则选项是不完整的。

总结

Shell 命令的选项用于调整命令功能，而命令的参数是这个命令的操作对象。有些选项后面也需要附带参数，以补全命令的功能。

6. Shell 命令的本质到底是什么？如何自己实现一个命令？

《[Shell 是什么](#)》一节中讲到，用户通过在 Shell 中输入一些命令来使用 Linux。给命令附带不同的选项后，同一个命令的功能也会有所差异。

Shell 命令分为两种：

- Shell 自带的命令称为内置命令，它在 Shell 内部可以通过函数来实现，当 Shell 启动后，这些命令所对应的代码（函数体代码）也被加载到内存中，所以使用内置命令是非常快速的。
- 更多的命令是外部的应用程序，一个命令就对应一个应用程序。运行外部命令要开启一个新的进程，所以效率上比内置命令差很多。

用户输入一个命令后，Shell 先检测该命令是不是内置命令，如果是就执行，如果不是就检测有没有对应的外部程序：有的话就转而执行外部程序，执行结束后再回到 Shell；没有的话就报错，告诉用户该命令不存在。

内置命令

内置命令不宜过多，过多的内置命令会导致 Shell 程序本身体积膨胀，运行 Shell 程序后就会占用更多的内存。Shell 是一个常驻内存的程序，占用过多内存会影响其它的程序。

只有那些最常用的命令才有理由成为内置命令，比如 cd、kill、echo 等；你可以转到《[Shell 内置命令](#)》来了解所有的内置命令，以及如何判断一个命令是否是内置命令。

外部命令

外部命令可能是读者比较疑惑的，一个外部的应用程序究竟是如何变成一个 Shell 命令的呢？

应用程序就是一个文件，只不过这个文件是可以执行的。既然是文件，那么它就有名字，并且存放在文件系统中。用户在 Shell 中输入一个外部命令后，只是将可执行文件的名字告诉了 Shell，但是并没有告诉 Shell 去哪里寻找这个文件。

难道 Shell 要遍历整个文件系统，查看每个目录吗？这显然是不能实现的。

为了解决这个问题，Shell 在启动文件中增加了一个叫做 PATH 的环境变量，该变量就保存了 Shell 对外部命令的查找路径，如果在这些路径下找不到同名的文件，Shell 也不会再去其它路径下查找了，它就直接报错。

你不用关心启动文件（我们将在《[Shell 启动文件](#)》中详解），只需要知道 PATH 变量保存了检索路径即可。

我们使用 echo 命令输出 PATH 变量的值，看看它保存了哪些检索路径：

```
[mozhiyan@localhost ~]$ echo $PATH
/usr/local/bin:/usr/local/sbin:/usr/bin:/usr/sbin:/bin:/sbin:/home/mozhiyan/.local/bin:/home/mozhiyan/bin
```

不同的路径之间以 `:` 分隔。你看，Shell 只会几个固定的路径中查找外部命令。

如果我们自己用 C 语言或者 C++ 编写一个应用程序，并将它放到这几个目录下面，那么我们的程序也会成为 Shell 命令。当然，你也可以修改 PATH 变量给它增加另外的路径，不过这并不是本文的重点，有兴趣的读者请转到《[编写自己的 Shell 配置文件](#)》。

我自己使用 C 语言编写了一个叫做 getsum 的程序，它用来计算从 m 累加到 n 的和，代码如下：

```

#include <stdio.h>

#include <unistd.h>

#include <getopt.h>

#include <stdlib.h>

int main(int argc, char *argv[]) {

    int start = 0;

    int end = 0;

    int sum = 0;

    int opt;

    char *optstring = ":s:e:";

    while((opt = getopt(argc, argv, optstring)) != -1) {

        switch(opt) {

            case 's': start = atoi(optarg); break;

            case 'e': end = atoi(optarg); break;

            case ':': puts("Missing parameter"); exit(1);

        }

    }

    if(start < 0 || end <= start) {

        puts("Parameter error"); exit(2);

    }

    for(int i=start; i<=end; i++) {

        sum+=i;

    }

    printf("%d\n", sum);

    return 0;

}

```

将这段代码编译成名为 getsum 的应用程序，并放在 `~/bin` 目录（`~` 表示用户主目录）下，然后在 Shell 中输入下面的命令，就可以计算 `1+2+3 +99+100` 的值。

```
[mozhiyan@localhost ~]$ getsum -s 1 -e 100
```

```
5050
```

`-s` 选项表示起始数字，`-e` 选项表示终止数字。

对于不了解 C 语言的读者，我也提供了编译好的 getsum 程序，请[猛击这里](#)下载。

总结

Shell 内置命令的本质是一个自带的函数，执行内置命令就是调用这个自带的函数。因为函数代码在 Shell 启动时已经被加载到内存了，所以内置命令的执行速度很快。

Shell 外部命令的本质是一个应用程序，执行外部命令就是启动一个新的应用程序。因为要创建新的进程并加载应用程序的代码，所以外部命令的执行速度很慢。

7. Shell 命令的选项和参数在本质上到底是什么？

很多 Shell 命令都是可以附带选项和参数的，不同的选项和参数也使得命令的功能细节有所差异。

Shell 命令附带参数的例子：

- `cd demo` 命令表示进入当前目录下的 `demo` 目录，其中 `demo` 就是 `cd` 命令的参数。
- `echo "123xyz"` 命令表示输出字符串并换行，其中 `"123xyz"` 就是 `echo` 命令的参数。

Shell 命令附带选项的例子：

`ls -l` 命令用来显示当前目录下的所有文件以及它们的详细信息，其中 `-l` 就是 `ls` 命令的选项。

`echo -n "http://c.biancheng.net/shell/"` 表示在输出字符串后不换行，其中 `-n` 是 `echo` 命令的选项，`"http://c.biancheng.net/shell/"` 是 `echo` 命令的参数。

有些命令的选项后面也可以附带参数：

- `getsum -s 1 -e 100` 命令用来计算从 1 累加到 100 的和，其中 `-s` 和 `-e` 是 `getsum` 命令的选项，1 和 100 分别是 `-s` 和 `-e` 选项的参数。
- `read -n 1 sex` 命令用来读取一个字符并赋值给 `sex` 变量，其中 `-n` 是 `read` 命令的选项，1 是 `-n` 选项的参数，`sex` 是 `read` 命令的参数。

你是否对这些形形色色的选项和参数感到好奇？你是否想知道它们在底层是如何实现的？你是否也想自己动手对它们进行解析？本节就来给你揭晓答案！

死磕这个细节并不是闲得无聊，它能帮助我们理解命令的真正含义。好了，废话不多说，让我们赶紧转入正题吧。

上节我们讲到，一个 Shell 内置命令就是一个内部的函数，一个外部命令就是一个应用程序。内置命令后面附带的所有数据（所有选项和参数）最终都以参数的形式传递给了函数，外部命令后面附带的所有数据（所有选项和参数）最终都以参数的形式传递给了应用程序。

也就是说，不管是内置命令还是外部命令，它后面附带的所有数据都会被“打包”成参数，这些参数有的传递给了函数，有的传递给了应用程序。

有编程经验的读者应该知道，C 语言或者 C++ 程序的入口函数是 `int main(int argc, char *argv[])`，传递给应用程序的参数最终都被 `main` 函数接收了。从这个角度看，传递给应用程序的参数其实也是传递给了函数。

有了以上认知，我们就不用再区分函数和应用程序了，我们就认为：**不管是内置命令还是外部命令，它后面附带的数据最终都以参数的形式传递给了函数。**实现一个命令的一项重要工作就是解析传递给函数的参数。

注意，命令后面附带的数据并不是被合并在一起，作为一个参数传递给函数的；这些数据是由空格分隔的，它们被分隔成了几份，就会转换成几个参数。例如 `getsum -s 1 -e 100` 要向函数传递四个参数，`read -n 1 sex` 要向函数中传递三个参数。

并且，命令后面附带的数据都是“原汁原味”地传递给了函数，比如 `getsum -s 1 -e 100` 要传递的四个参数分别是 `-s`、`1`、`-e`、`100`，减号-也会一起传递过去，在函数内部，减号-可以用来区分该参数是否是命令的选项。

至于在函数内部如何解析这些参数，对于外部命令来说那就是 C/C++ 程序员的工作了，这里不再过多赘述，只给出演示代码。

上节我给大家演示了一个 `getsum` 程序，本节依然使用该程序演示参数的解析，只是对代码进行了微调。

```
#include <stdio.h>

#include <unistd.h>

#include <getopt.h>

#include <stdlib.h>
```

```

int main(int argc, char *argv[]) {

    int start = 0;

    int end = 0;

    int sum = 0;

    int opt;

    char *optstring = ":s:e:";

    //分析接收到的参数

    while((opt = getopt(argc, argv, optstring)) != -1) {

        switch(opt) {

            case 's': start = atoi(optarg); break;

            case 'e': end = atoi(optarg); break;

            case ':': puts("Missing parameter"); exit(1);

        }

    }

    //检测参数是否有效

    if(start < 0 || end <= start) {

        puts("Parameter error"); exit(2);

    }

    //打印接收到的参数

    printf("Received parameters: ");

    for(int i=0; i<argc; i++) {

        printf("%s ", argv[i]);

    }

    printf("\n");

    //计算累加的和

    for(int i=start; i<=end; i++) {

        sum+=i;

    }

    printf("sum=%d\n", sum);

    return 0;

}

```

第 11~20 行是解析参数的关键代码，getopt.h 头文件中的 getopt() 函数是值得重点研究的，有了该函数我们就不用自己去解析参数了，省了很大的力气。

第 27~32 行将接收到的参数打印出来，以便读者更好地观察。

根据上节给出的办法就可以运行 getsum 命令：

```
[mozhiyan@localhost ~]$ getsum -s 1 -e 100  
  
Received parameters: getsum  -s 1  -e 100  
  
sum=5050
```

8. Linux Shell 命令提示符

启动 [Linux 桌面环境](#) 自带的终端模拟包，或者从 Linux 控制台登录后，便可以看到 Shell 命令提示符。看见命令提示符就意味着可以输入命令了。命令提示符不是命令的一部分，它只是起到一个提示作用。

不同的 Linux 发行版使用的提示符格式大同小异，例如在 CentOS 中，默认的提示符类似下面这样：

```
[mozhiyan@localhost ~]$
```

各个部分的含义如下：

- `[]` 是提示符的分隔符号，没有特殊含义。
- `mozhiyan` 表示当前登录的用户，我现在使用的是 mozhiyan 用户登录。
- `@` 是分隔符号，没有特殊含义。
- `localhost` 表示当前系统的简写主机名（完整主机名是 localhost.localdomain）。
- `~` 代表用户当前所在的目录为主目录（home 目录）。如果用户当前位于主目录下的 bin 目录中，那么这里显示的就是 bin。
- `$` 是命令提示符。Linux 用这个符号标识登录的用户权限等级：如果是超级用户（root 用户），提示符就是 `#`；如果是普通用户，提示符就是 `$`。

总结起来，Linux Shell 默认的命令提示符的格式为：

```
[username@host directory]$
```

或者

```
[username@host directory]#
```

什么是主目录？

Linux 系统是纯字符界面，用户登录后，要有一个初始登录的位置，这个初始登录位置就称为用户的主目录（home 目录）。超级用户的主目录为 `/root/`，普通用户的主目录为 `/home/用户名/`。

有的资料也称为“家目录”，“家”是 home 的直译，它们都是一个意思。

用户在自己的主目录中拥有完整权限，所以我们也建议操作实验可以放在主目录中进行。

我们使用 `cd` 命令切换一下用户所在目录，看看有什么效果。

```
[mozhiyan@localhost ~]$ cd demo
```

```
[mozhiyan@localhost demo]$ cd /usr/local
```

```
[mozhiyan@localhost local]$
```

仔细看，如果切换用户所在目录，那么命令提示符中会变成用户当前所在目录的最后一个目录（不显示完整的所在目录 `/usr/local/`，只显示最后一个目录 `local`）。

第二层命令提示符

有些命令不能在一行内输入完成，需要换行，这个时候就会看到第二层命令提示符。第二层命令提示符默认为 `>`，请看下面的例子：

```
[mozhiyan@localhost ~]$ echo "Shell 教程"
```

Shell 教程

```
[mozhiyan@localhost ~]$ echo "
```

```
> http://
```

```
> c.biancheng.net
```

```
> "
```

```
http://
```

```
c.biancheng.net
```

第一个 echo 命令在一行内输入完成，不会出现第二层提示符。第二个 echo 命令需要多行才能输入完成，提示符 `>` 用来告诉用户命令还没输入完成，请继续输入。

echo 命令用来输出一个字符串。字符串是一组由 `"` 包围起来的字符序列，echo 将第一个 `"` 作为字符串的开端，将第二个 `"` 作为字符串的结尾。对于第二个 echo 命令，我们将字符串分成多行，echo 遇到第一个 `"` 认为是不完整的字符串，所以会继续等待用户输入，直到遇见第二个 `"`。

命令提示符的格式不是固定的，用户可以根据自己的喜好来修改，下节《[修改 Linux 命令提示符](#)》将会展开讲解。

9. Shell 修改命令提示符

Shell 通过 PS1 和 PS2 这两个环境变量来控制提示符的格式，修改 PS1 和 PS2 的值就能修改命令提示符的格式。

- PS1 控制最外层的命令提示符格式。
- PS2 控制第二层的命令提示符格式。

在修改 PS1 和 PS2 之前，我们先用 echo 命令输出它们的值，看看默认情况下是什么样子的：

```
[mozhiyan@localhost ~]$ echo $PS1

[\u@\h \W]\$

[mozhiyan@localhost ~]$ echo $PS2

>
```

Linux 使用以\为前导的特殊字符来表示命令提示符中包含的要素，这使得 PS1 和 PS2 的格式看起来可能有点奇怪。下表展示了可以在 PS1 和 PS2 中使用的特殊字符。

Bash shell 命令提示符可以包含的要素	
字符	描述
\a	铃声字符
\d	格式为“日 月 年”的日期
\e	ASCII 转义字符
\h	本地主机名
\H	完全合格的限定域主机名
\j	shell 当前管理的作业数
\l	shell 终端设备名的基本名称
\n	ASCII 换行字符
\r	ASCII 回车
\s	shell 的名称
\t	格式为“小时:分钟:秒”的 24 小时制的当前时间
\T	格式为“小时:分钟:秒”的 12 小时制的当前时间
\@	格式为 am/pm 的 12 小时制的当前时间
\u	当前用户的用户名
\v	bash shell 的版本
\V	bash shell 的发布级别
\w	当前工作目录
\W	当前工作目录的基本名称
\!	该命令的 bash shell 历史数
\#	该命令的命令数量
\\$	如果是普通用户，则为美元符号\$；如果超级用户（root 用户），则为井号#。
\nnn	对应于八进制值 nnn 的字符

\\	斜杠
\l	控制码序列的开头
\j	控制码序列的结尾

注意，所有的特殊字符均以反斜杠\开头，目的是与普通字符区分开来。您可以在命令提示符中使用以上任何特殊字符的组合。

【实例】通过修改 PS1 变量的值来修改命令提示符的格式：

```
[mozhiyan@localhost ~]$ PS1="\t[\u]\$ "
```

```
[12:51:43][mozhiyan]$ PS1="[c.biancheng.net]\$ "
```

```
[c.biancheng.net]$
```

第一次修改后可以显示当前的时间和用户名，第二次修改后显示 C 语言中文网的域名。为了保留版权，证明该教程出自 C 语言中文网，后续文章中我经常会使用[c.biancheng.net]\$这种命令提示符，大家不要觉得奇怪。

遗憾的是，通过这种方式修改的命令提示符只在当前的 Shell 会话期间有效，再次启动 Shell 后将重新使用默认的命令提示符。

如果希望持久性地修改 PS1，让它对任何 Shell 会话都有效，那么就得把 PS1 变量的修改写入到 Shell 启动文件中，我们将在《[编写自己的 Shell 配置文件](#)》一节中展开讨论。

10. 第一个 Shell 脚本

几乎所有编程语言的教程都是从使用著名的“Hello World”开始的，出于对这种传统的尊重（或者说落入俗套），我们的第一个 Shell 脚本也输出“Hello World”。

打开文本编辑器，新建一个文本文件，并命名为 test.sh。

扩展名 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。

在 test.sh 中输入代码：

```
#!/bin/bash
```

```
echo "Hello World !" #这是一条语句
```

第 1 行的 #! 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；后面的 /bin/bash 就是指明了解释器的具体位置。

第 2 行的 echo 命令用于向标准输出文件（Standard Output，stdout，一般就是指显示器）输出文本。在 .sh 文件中使用命令与在终端直接输入命令的效果是一样的。

第 2 行的 # 及其后面的内容是注释。Shell 脚本中所有以 # 开头的都是注释（当然以 #! 开头的除外）。写脚本的时候，多写注释是非常有必要的，以方便其他人能看懂你的脚本，也方便后期自己维护时看懂自己的脚本——实际上，即便是自己写的脚本，在经过一段时间后也很容易忘记。

下面给出了一段稍微复杂的 Shell 脚本：

```
#!/bin/bash
```

```
# Copyright (c) http://c.biancheng.net/shell/
```

```
echo "What is your name?"
```

```
read PERSON
```

```
echo "Hello, $PERSON"
```

第 5 行中表示从终端读取用户输入的数据，并赋值给 PERSON 变量。read 命令用来从标准输入文件（Standard Input，stdin，一般就是指键盘）读取用户输入的数据。

第 6 行表示输出变量 PERSON 的内容。注意在变量名前边要加上 \$，否则变量名会作为字符串的一部分处理。

11. 执行 Shell 脚本（多种方法）

上节我们编写了一个简单的 Shell 脚本，这节课我们就让它运行起来。运行 Shell 脚本有两种方法，一种在新进程中运行，一种是在当前 Shell 进程中运行。

在新进程中运行 Shell 脚本

在新进程中运行 Shell 脚本有多种方法。

1) 将 Shell 脚本作为程序运行

Shell 脚本也是一种解释执行的程序，可以在终端直接调用（需要使用 `chmod` 命令给 Shell 脚本加上执行权限），如下所示：

```
[mozhiyan@localhost ~]$ cd demo           #切换到 test.sh 所在的目录

[mozhiyan@localhost demo]$ chmod +x ./test.sh #给脚本添加执行权限

[mozhiyan@localhost demo]$ ./test.sh        #执行脚本文件

Hello World !                               #运行结果
```

第 2 行中，`chmod +x` 表示给 `test.sh` 增加执行权限。

第 3 行中，`./` 表示当前目录，整条命令的意思是执行当前目录下的 `test.sh` 脚本。如果不写 `./`，Linux 会到系统路径（由 `PATH` 环境变量指定）下查找 `test.sh`，而系统路径下显然不存在这个脚本，所以会执行失败。

通过这种方式运行脚本，脚本文件第一行的 `#!/bin/bash` 一定要写对，好让系统查找到正确的解释器。

2) 将 Shell 脚本作为参数传递给 Bash 解释器

你也可以直接运行 Bash 解释器，将脚本文件的名字作为参数传递给 Bash，如下所示：

```
[mozhiyan@localhost ~]$ cd demo           #切换到 test.sh 所在的目录

[mozhiyan@localhost demo]$ /bin/bash test.sh #使用 Bash 的绝对路径

Hello World !                               #运行结果
```

通过这种方式运行脚本，不需要在脚本文件的第一行指定解释器信息，写了也没用。

更加简洁的写法是运行 `bash` 命令。`bash` 是一个外部命令，Shell 会在 `/bin` 目录中找到对应的应用程序，也即 `/bin/bash`，这点我们已在《[Shell 命令的本质到底是什么](#)》一节中提到。

```
[mozhiyan@localhost ~]$ cd demo

[mozhiyan@localhost demo]$ bash test.sh

Hello World !
```

这两种写法在本质上是一样的：第一种写法给出了绝对路径，会直接运行 Bash 解释器；第二种写法通过 `bash` 命令找到 Bash 解释器所在的目录，然后再运行，只不过多了一个查找的过程而已。

检测是否开启了新进程

有些读者可能会疑问，你怎么知道开启了新进程？你有什么证据吗？既然如此，那我就来给大家验证一下吧。

Linux 中的每一个进程都有一个唯一的 ID，称为 PID，使用 `$$` 变量就可以获取当前进程的 PID。`$$` 是 Shell 中的特殊变量，稍后我会在《[Shell 特殊变量](#)》一节中展开讲解，读者在此不必深究。

首先编写如下的脚本文件，并命名为 `check.sh`：

1. `#!/bin/bash`
2. `echo $$ #输出当前进程 PID`

然后使用以上两种方式来运行 `check.sh`：

```
[mozhiyan@localhost demo]$ echo $$

2861  #当前进程的 PID

[mozhiyan@localhost demo]$ chmod +x ./check.sh

[mozhiyan@localhost demo]$ ./check.sh

4597  #新进程的 PID

[mozhiyan@localhost demo]$ echo $$

2861  #当前进程的 PID

[mozhiyan@localhost demo]$ /bin/bash check.sh

4584  #新进程的 PID
```

你看，进程的 PID 都不一样，当然就是两个进程了。

在当前进程中运行 Shell 脚本

这里需要引入一个新的命令——`source` 命令。`source` 是 [Shell 内置命令](#)的一种，它会读取脚本文件中的代码，并依次执行所有语句。你也可以理解为，`source` 命令会强制执行脚本文件中的全部命令，而忽略脚本文件的权限。

`source` 命令的用法为：

```
source filename
```

也可以简写为：

```
. filename
```

两种写法的效果相同。对于第二种写法，注意点号 `.` 和文件名中间有一个空格。

例如，使用 `source` 运行上节的 `test.sh`：

```
[mozhiyan@localhost ~]$ cd demo          #切换到 test.sh 所在的目录

[mozhiyan@localhost demo]$ source ./test.sh #使用 source
```

```
Hello World !

[mozhiyan@localhost demo]$ source test.sh    #使用 source

Hello World !

[mozhiyan@localhost demo]$ ./test.sh        #使用点号

Hello World !

[mozhiyan@localhost demo]$ . test.sh        #使用点号

Hello World !
```

你看，使用 `source` 命令不用给脚本增加执行权限，并且写不写 `/` 都行，是不是很方便呢？

检测是否在当前 Shell 进程中

我们仍然借助 `$$` 变量来输出进程的 PID，如下所示：

```
[mozhiyan@localhost ~]$ cd demo

[mozhiyan@localhost demo]$ echo $$

5169    #当前进程 PID

[mozhiyan@localhost demo]$ source ./check.sh

5169    #Shell 脚本所在进程 PID

[mozhiyan@localhost demo]$ echo $$

5169    #当前进程 PID

[mozhiyan@localhost demo]$ ./check.sh

5169    #Shell 脚本所在进程 PID
```

你看，进程的 PID 都是一样的，当然是同一个进程了。

总结

作为初学者，你可能看不懂这些运行方式有什么区别，没关系，暂时先留个疑问吧，后续教程中我们会逐一讲解。

如果需要在子进程中运行 Shell 脚本，我一般使用 `bash test.sh` 这种写法；如果在当前进程中运行 Shell 脚本，我一般使用 `./test.sh` 这种写法。这是我个人的风格。

最后再给大家演示一个稍微复杂的例子。本例中使用 `read` 命令从键盘读取用户输入的内容并赋值给 `URL` 变量，最后在显示器上输出。

```
1.  #!/bin/bash
2.  # Copyright (c) http://c.biancheng.net/shell/
3.
4.  echo "What is the url of the shell tutorial?"
5.  read URL
```

```
6. echo "$URL is very fast!"
```

运行脚本：

```
[mozhiyan@localhost demo]$ ./test.sh
```

```
What is the url of the shell tutorial?
```

```
http://c.biancheng.net/shell/↵
```

```
http://c.biancheng.net/shell/ is very fast!
```

↵ 表示按下回车键。

12. Shell 四种运行方式（启动方式）精讲

Shell 是一个应用程序，它的一端连接着 Linux 内核，另一端连接着用户。Shell 是用户和 Linux 系统沟通的桥梁，我们都是通过 Shell 来管理 Linux 系统。

我们可以直接使用 Shell，也可以输入用户名和密码后再使用 Shell；第一种叫做非登录式，第二种叫做登录式。

我们可以在 Shell 中一个个地输入命令并及时查看它们的输出结果，整个过程都在跟 Shell 不停地互动，这叫做交互式。我们也可以运行一个 Shell 脚本文件，让所有命令批量化、一次性地执行，这叫做非交互式。

总起来说，Shell 一共有四种运行方式：

- 交互式的登录 Shell；
- 交互式的非登录 Shell；
- 非交互式的登录 Shell；
- 非交互式的非登录 Shell。

判断 Shell 是否是交互式

判断是否为交互式 Shell 有两种简单的方法。

1) 查看变量 `_` 的值，如果值中包含了字母 `i`，则表示交互式（interactive）。

【实例 1】在 CentOS GNOME 桌面环境自带的终端下输出 `_` 的值：

```
[c.biancheng.net]$ echo $_  
himBH
```

包含了 `i`，为交互式。

【实例 2】在 Shell 脚本文件中输出 `_` 的值：

```
[c.biancheng.net]$ cat test.sh  
  
#!/bin/bash  
  
echo $_  
  
[c.biancheng.net]$ bash ./test.sh  
  
hB
```

不包含 `i`，为非交互式。注意，必须在新进程中[运行 Shell 脚本](#)。

2) 查看变量 `PS1` 的值，如果非空，则为交互式，否则为非交互式，因为非交互式会清空该变量。

【实例 1】在 CentOS GNOME 桌面环境自带的终端下输出 `PS1` 的值：

```
[mozhiyan@localhost]$ echo $PS1
```

```
[\\u@\\h \\W]\\$
```

非空，为交互式。

【实例 2】在 Shell 脚本文件中输出 PS1 的值：

```
[c.biancheng.net]$ cat test.sh

#!/bin/bash

echo $PS1

[c.biancheng.net]$ bash ./test.sh
```

空值，为非交互式。注意，必须在新进程中运行 Shell 脚本。

判断 Shell 是否为登录式

判断 Shell 是否为登录式也非常简单，只需执行 `shopt login_shell` 即可，值为 `on` 表示为登录式，`off` 为非登录式。

`shopt` 命令用来查看或设置 Shell 中的行为选项，这些选项可以增强 Shell 的易用性。

【实例 1】在 CentOS GNOME 桌面环境自带的终端下查看 `login_shell` 选项：

```
[c.biancheng.net]$ shopt login_shell

login_shell    off
```

【实例 2】按下 `Ctrl+Alt+Fn` 组合键切换到虚拟终端，输入用户名和密码登录后，再查看 `login_shell` 选项：

```
[c.biancheng.net]$ shopt login_shell

login_shell    on
```

【实例 3】在 Shell 脚本文件中查看 `login_shel` 选项：

```
[c.biancheng.net]$ cat test.sh

#!/bin/bash

shopt login_shell

[c.biancheng.net]$ bash ./test.sh

login_shell    off
```

同时判断交互式、登录式

要同时判断是否为交互式和登录式，可以简单使用如下的命令：

```
echo $PS1; shopt login_shell
```

或者

```
echo $-; shopt login_shell
```

常见的 Shell 启动方式

1) 通过 Linux 控制台（不是桌面环境自带的终端）或者 ssh 登录 Shell 时（这才是正常登录方式），为交互式的登录 Shell。

```
[c.biancheng.net]$ echo $PS1;shopt login_shell

[\u@\h \W]\$

login_shell      on
```

2) 执行 bash 命令时默认是非登录的，增加 `--login` 选项（简写为 `-l`）后变成登录式。

```
[c.biancheng.net]$ cat test.sh

#!/bin/bash

echo $-; shopt login_shell

[c.biancheng.net]$ bash -l ./test.sh

hB

login_shell      on
```

3) 使用由 `()` 包围的[组命令](#)或者[命令替换](#)进入子 Shell 时，子 Shell 会继承父 Shell 的交互和登录属性。

```
[c.biancheng.net]$ bash

[c.biancheng.net]$ (echo $PS1;shopt login_shell)

[\u@\h \W]\$

login_shell      off

[c.biancheng.net]$ bash -l

[c.biancheng.net]$ (echo $PS1;shopt login_shell)

[\u@\h \W]\$
```

```
login_shell    on
```

4) ssh 执行远程命令，但不登录时，为非交互非登录式。

```
[c.biancheng.net]$ ssh localhost 'echo $PS1;shopt login_shell'
```

```
login_shell    off
```

5) 在 [Linux 桌面环境](#)下打开终端时，为交互式的非登录 Shell。



13. Shell 配置文件（配置脚本）的加载

无论是交互式，还是登录式，Bash Shell 在启动时总要配置其运行环境，例如初始化环境变量、设置命令提示符、指定系统命令路径等。这个过程是通过加载一系列配置文件完成的，这些配置文件其实就是 Shell 脚本文件。

与 Bash Shell 有关的配置文件主要有 /etc/profile、~/bash_profile、~/bash_login、~/.profile、~/bashrc、/etc/bashrc、/etc/profile.d/*.sh，不同的启动方式会加载不同的配置文件。

~表示用户主目录。*是通配符，/etc/profile.d/*.sh 表示 /etc/profile.d/ 目录下所有的脚本文件（以.sh 结尾的文件）。

登录式的 Shell

Bash 官方文档说：如果是登录式的 Shell，首先会读取和执行 /etc/profiles，这是所有用户的全局配置文件，接着会到用户主目录中寻找 ~/.bash_profile、~/bash_login 或者 ~/.profile，它们都是用户个人的配置文件。

不同的 Linux 发行版附带的个人配置文件也不同，有的可能只有其中一个，有的可能三者都有，笔者使用的是 CentOS 7，该发行版只有 ~/.bash_profile，其它两个都没有。

如果三个文件同时存在的话，到底应该加载哪一个呢？它们的优先级顺序是 ~/.bash_profile > ~/.bash_login > ~/.profile。

如果 ~/.bash_profile 存在，那么一切以该文件为准，并且到此结束，不再加载其它的配置文件。

如果 ~/.bash_profile 不存在，那么尝试加载 ~/.bash_login。~/bash_login 存在的话就到此结束，不存在的话就加载 ~/.profile。

注意，/etc/profiles 文件还会嵌套加载 /etc/profile.d/*.sh，请看下面的代码：

```
for i in /etc/profile.d/*.sh ; do

    if [ -r "$i" ]; then

        if [ "${-#*i}" != "$-" ]; then

            . "$i"

        else

            . "$i" >/dev/null

        fi

    fi

done
```

同样，~/bash_profile 也使用类似的方式加载 ~/.bashrc：

```
if [ -f ~/.bashrc ]; then

    . ~/.bashrc

fi
```

非登录的 Shell

如果以非登录的方式启动 Shell，那么就不会读取以上所说的配置文件，而是直接读取 `~/bashrc`。

`~/bashrc` 文件还会嵌套加载 `/etc/bashrc`，请看下面的代码：

```
if [ -f /etc/bashrc ]; then
. /etc/bashrc
fi
```

14. 如何编写自己的 Shell 配置文件（配置脚本）？

学习了《[Shell 配置文件的加载](#)》一节，读者应该知道 Shell 在登录和非登录时都会加载哪些配置文件了。对于普通用户来说，也许 `~/.bashrc` 才是最重要的文件，因为不管是否登录都会加载该文件。

我们可以将自己的一些代码添加到 `~/.bashrc`，这样每次启动 Shell 都可以个性化地配置。如果你有代码洁癖，也可以将自己编写的代码放到一个新文件中（假设叫 `myconf.sh`），只要在 `~/.bashrc` 中使用类似 `./myconf.sh` 的形式将新文件引入进来就行了

使用 `source` 命令引入其它代码文件时有一些细节需要注意，我们将在《[Shell 模块化](#)》一节中展开讨论。

实例 1：给 PATH 变量增加新的路径

你曾经是否感到迷惑，Shell 是怎样知道去哪里找到我们输入的命令的？例如，当我们输入 `ls` 后，Shell 不会查找整个计算机系统，而是在指定的几个目录中检索（最终在 `/bin/` 目录中找到了 `ls` 程序），这些目录就包含在 `PATH` 变量中。

当用户登录 Shell 时，`PATH` 变量会在 `/etc/profile` 文件中设置，然后在 `~/.bash_profile` 也会增加几个目录。如果没有登录 Shell，`PATH` 变量会在 `/etc/bashrc` 文件中设置。

如果我们想增加自己的路径，可以将该路径放在 `~/.bashrc` 文件中，例如：

```
PATH=$PATH:$HOME/addon
```

将主目录下的 `addon` 目录也设置为系统路径。假如此时在 `addon` 目录下有一个 `getsum` 程序，它的作用是计算从 `m` 累加到 `n` 的和，那么我们不用 `cd` 到 `addon` 目录，直接输入 `getsum` 命令就能得到结果。

在《[Shell 命令的本质到底是什么](#)》一节中我已经给出了 `getsum` 程序及其源代码，有兴趣的读者可以[猛击这里下载](#)。下载完成后请配置环境变量，然后输入如下的命令就可以得到结果：

```
[c.biancheng.net]$ getsum -s 1 -e 100
```

```
5050
```

`-s` 选项表示起始（start）数字，`-e` 选项表示终止（end）数字，以上命令用来计算从 1 累加到 100 的和。

实例 2：修改命令提示符的格式

在《[修改 Linux 命令提示符](#)》一节中我曾提到，修改 `PS1` 变量的值就可以修改命令提示符的格式，但是那个时候大家还不了解 Shell 启动文件，所以只能临时性地修改，并不能持久。

现在我们已经知道，在 `~/.bashrc` 文件中修改 `PS1` 变量的值就可以持久化，每个使用 Shell 的用户都会看见新的命令提示符。

将下面的代码添加到 `~/.bashrc` 文件中，然后重新启动 Shell，命令提示符就变成了 `[c.biancheng.net]$`。

```
PS1="[c.biancheng.net]\$ "
```

第 2 章：Shell 编程

1. Shell 变量：Shell 变量的定义、赋值和删除

变量是任何一种编程语言都必不可少的组成部分，变量用来存放各种数据。脚本语言在定义变量时通常不需要指明类型，直接赋值就可以，Shell 变量也遵循这个规则。

在 Bash shell 中，每一个变量的值都是字符串，无论你给变量赋值时有没有使用引号，值都会以字符串的形式存储。

这意味着，Bash shell 在默认情况下不会区分变量类型，即使你将整数和小数赋值给变量，它们也会被视为字符串，这一点和大部分的编程语言不同。例如在 C 语言或者 [C++](#) 中，变量分为整数、小数、字符串、布尔等多种类型。

当然，如果有必要，你也可以使用 [Shell declare](#) 关键字显式定义变量的类型，但在一般情况下没有这个需求，Shell 开发者在编写代码时自行注意值的类型即可。

定义变量

Shell 支持以下三种定义变量的方式：

```
variable=value
variable='value'
variable="value"
```

variable 是变量名，value 是赋给变量的值。如果 value 不包含任何空白符（例如空格、Tab 缩进等），那么可以不使用引号；如果 value 包含了空白符，那么就必须使用引号包围起来。使用单引号和使用双引号也是有区别的，稍后我们会详细说明。

注意，赋值号 = 的周围不能有空格，这和你熟悉的大部分编程语言都不一样。

Shell 变量的命名规范和大部分编程语言都一样：

- 变量名由数字、字母、下划线组成；
- 必须以字母或者下划线开头；
- 不能使用 Shell 里的关键字（通过 help 命令可以查看保留关键字）。

变量定义举例：

```
1. url=http://c.biancheng.net/shell/
2. echo $url
3. name='C 语言中文网'
4. echo $name
5. author="严长生"
6. echo $author
```

使用变量

使用一个定义过的变量，只要在变量名前面加美元符号 \$ 即可，如：

```
1. author="严长生"
```

```
2. echo $author
3. echo ${author}
```

变量名外面的花括号{}是可选的，加不加都行，加花括号是为了帮助解释器识别变量的边界，比如下面这种情况：

```
1. skill="Java"
2. echo "I am good at ${skill}Script"
```

如果不给 skill 变量加花括号，写成 `echo "I am good at $skillScript"`，解释器就会把 \$skillScript 当成一个变量（其值为空），代码执行结果就不是我们期望的样子了。

推荐给所有变量加上花括号{}，这是个良好的编程习惯。

修改变量的值

已定义的变量，可以被重新赋值，如：

```
1. url="http://c.biancheng.net"
2. echo ${url}
3. url="http://c.biancheng.net/shell/"
4. echo ${url}
```

第二次对变量赋值时不能在变量名前加\$，只有在使用变量时才能加\$。

单引号和双引号的区别

前面我们还留下一个疑问，定义变量时，变量的值可以由单引号'包围，也可以由双引号"包围，它们到底有什么区别呢？不妨以下面的代码为例来说明：

```
1. #!/bin/bash
2.
3. url="http://c.biancheng.net"
4. website1='C 语言中文网: ${url}'
5. website2="C 语言中文网: ${url}"
6. echo $website1
7. echo $website2
```

运行结果：

C 语言中文网: \${url}

C 语言中文网: http://c.biancheng.net

以单引号'包围变量的值时，单引号里面是什么就输出什么，即使内容中有变量和命令（命令需要反引起来）也会把它们原样输出。这种方式比较适合定义显示纯字符串的情况，即不希望解析变量、命令等的场景。

以双引号" "包围变量的值时，输出时会先解析里面的变量和命令，而不是把双引号中的变量名和命令原样输出。这种方式比较适合字符串中附带有变量和命令并且想将其解析后再输出的变量定义。

我的建议：如果变量的内容是数字，那么可以不加引号；如果真的需要原样输出就加单引号；其他没有特别要求的字符串等最好都加上双引号，定义变量时加双引号是最常见的使用场景。

将命令的结果赋值给变量

Shell 也支持将命令的执行结果赋值给变量，常见的有以下两种方式：

```
variable=`command`  
variable=$(command)
```

第一种方式把命令用反引号`（位于 Esc 键的下方）包围起来，反引号和单引号非常相似，容易产生混淆，所以不推荐使用这种方式；第二种方式把命令用\$()包围起来，区分更加明显，所以推荐使用这种方式。

例如，我在 demo 目录中创建了一个名为 log.txt 的文本文件，用来记录我的日常工作。下面的代码中，使用 cat 命令将 log.txt 的内容读取出来，并赋值给一个变量，然后使用 echo 命令输出。

```
[mozhiyan@localhost ~]$ cd demo  
  
[mozhiyan@localhost demo]$ log=$(cat log.txt)  
  
[mozhiyan@localhost demo]$ echo $log  
  
严长生正在编写 Shell 教程，教程地址：http://c.biancheng.net/shell/  
  
[mozhiyan@localhost demo]$ log=`cat log.txt`  
  
[mozhiyan@localhost demo]$ echo $log  
  
严长生正在编写 Shell 教程，教程地址：http://c.biancheng.net/shell/
```

只读变量

使用 **readonly** 命令可以将变量定义为只读变量，只读变量的值不能被改变。

下面的例子尝试更改只读变量，结果报错：

```
1.  #!/bin/bash  
2.  
3.  myUrl="http://c.biancheng.net/shell/"  
4.  readonly myUrl  
5.  myUrl="http://c.biancheng.net/shell/"
```

运行脚本，结果如下：

```
bash: myUrl: This variable is read only.
```

删除变量

使用 **unset** 命令可以删除变量。语法：

```
1. unset variable_name
```

变量被删除后不能再次使用；unset 命令不能删除只读变量。

举个例子：

[纯文本复制](#)

```
1. #!/bin/sh
2.
3. myUrl="http://c.biancheng.net/shell/"
4. unset myUrl
5. echo $myUrl
```

上面的脚本没有任何输出。

2. Shell 变量的作用域：全局变量、环境变量和局部变量

Shell 变量的**作用域 (Scope)**，就是 Shell 变量的有效范围（可以使用的范围）。

在不同的作用域中，同名的变量不会相互干涉，就好像 A 班有个叫小明的同学，B 班也有个叫小明的同学，虽然他们都叫小明（对应于变量名），但是由于所在的班级（对应于作用域）不同，所以不会造成混乱。但是如果同一个班级中有两个叫小明的同学，就必须用类似于“大小明”、“小小明”这样的命名来区分他们。

Shell 变量的作用域可以分为三种：

- 有的变量只能在函数内部使用，这叫做**局部变量 (local variable)**；
- 有的变量可以在当前 Shell 进程中使用，这叫做**全局变量 (global variable)**；
- 而有的变量还可以在子进程中使用，这叫做**环境变量 (environment variable)**。

Shell 局部变量

Shell 也支持自定义函数，但是 Shell 函数和 C++、Java、C# 等其他编程语言函数的一个不同点就是：在 Shell 函数中定义的变量默认也是全局变量，它和在函数外部定义变量拥有一样的效果。请看下面的代码：

```
1.  #!/bin/bash
2.
3.  #定义函数
4.  function func(){
5.      a=99
6.  }
7.
8.  #调用函数
9.  func
10.
11. #输出函数内部的变量
12. echo $a
```

输出结果：
99

a 是在函数内部定义的，但是在函数外部也可以得到它的值，证明它的作用域是全局的，而不是仅限于函数内部。

要想变量的作用域仅限于函数内部，可以在定义时加上 `local` 命令，此时该变量就成了局部变量。请看下面的代码：

```
1.  #!/bin/bash
2.
3.  #定义函数
```



```
4.  function func() {
5.      local a=99
6.  }
7.
8.  #调用函数
9.  func
10.
11. #输出函数内部的变量
12. echo $a
```

输出结果为空，表明变量 `a` 在函数外部无效，是一个局部变量。

Shell 变量的这个特性和 JavaScript 中的变量是类似的。在 JavaScript 函数内部定义的变量，默认也是全局变量，只有加上 `var` 关键字，它才会变成局部变量。

本节只是演示了函数的定义和调用，并没有对语法细节作过多说明，后续我们将在《[Shell 函数](#)》一节中进行深入讲解。

Shell 全局变量

所谓全局变量，就是指变量在当前的整个 Shell 进程中都有效。每个 Shell 进程都有自己的作用域，彼此之间互不影响。在 Shell 中定义的变量，默认就是全局变量。

想要实际演示全局变量在不同 Shell 进程中的互不相关性，可在图形界面下同时打开两个 Shell，或使用两个终端远程连接到服务器（SSH）。

首先打开一个 Shell 窗口，定义一个变量 `a` 并赋值为 99，然后打印，这时在同一个 Shell 窗口中是可正确打印变量 `a` 的值的。然后再打开一个新的 Shell 窗口，同样打印变量 `a` 的值，但结果却为空，如图 1 所示。

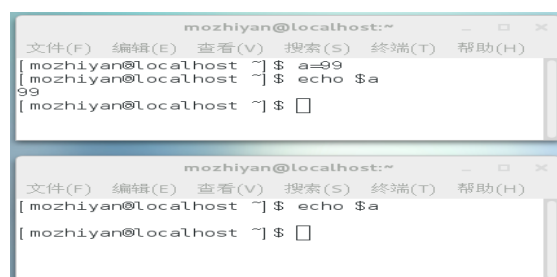


图 1：打开两个 Shell 窗口

这说明全局变量 `a` 仅仅在定义它的第一个 Shell 进程中有效，对新的 Shell 进程没有影响。这很好理解，就像小王家和小徐家都有一部电视机（变量名相同），但是同一时刻小王家和小徐家的电视中播放的节目可以是不同的（变量值不同）。

需要强调的是，全局变量的作用范围是当前的 Shell 进程，而不是当前的 Shell 脚本文件，它们是不同的概念。打开一个 Shell 窗口就创建了一个 Shell 进程，打开多个 Shell 窗口就创建了多个 Shell 进程，每个 Shell 进程都是独立的，拥有不同的进程 ID。在一个 Shell 进程中可以使用 `source` 命令执行多个 Shell 脚本文件，此时全局变量在这些脚本文件中都有效。

例如，现在有两个 Shell 脚本文件，分别是 `a.sh` 和 `b.sh`。`a.sh` 的代码如下：

1. `#!/bin/bash`
2. `echo $a`
3. `b=200`

b.sh 的代码如下：

1. `#!/bin/bash`
2. `echo $b`

打开一个 Shell 窗口，输入以下命令：

```
[c.biancheng.net]$ a=99

[c.biancheng.net]$ ./a.sh

99

[c.biancheng.net]$ ./b.sh

200
```

这三条命令都是在一个进程中执行的，从输出结果可以发现，在 Shell 窗口中以命令行的形式定义的变量 a，在 a.sh 中有效；在 a.sh 中定义的变量 b，在 b.sh 中也有效，变量 b 的作用范围已经超越了 a.sh。

注意，必须在当前进程中运行 Shell 脚本，不能在新进程中运行 Shell 脚本，不了解的读者请转到《[执行 Shell 脚本](#)》。

Shell 环境变量

全局变量只在当前 Shell 进程中有效，对其它 Shell 进程和子进程都无效。如果使用 `export` 命令将全局变量导出，那么它就在所有的子进程中也有效了，这称为“环境变量”。

环境变量被创建时所处的 Shell 进程称为父进程，如果在父进程中再创建一个新的进程来执行 Shell 命令，那么这个新的进程被称作 Shell 子进程。当 Shell 子进程产生时，它会继承父进程的环境变量为自己所用，所以说环境变量可从父进程传给子进程。不难理解，环境变量还可以传递给孙进程。

注意，两个没有父子关系的 Shell 进程是不能传递环境变量的，并且环境变量只能向下传递而不能向上传递，即“传子不传父”。

创建 Shell 子进程最简单的方式是运行 `bash` 命令，如图 2 所示。



图 2：进入 Shell 子进程

通过 `exit` 命令可以一层一层地退出 Shell。

下面演示一下环境变量的使用：

```
[c.biancheng.net]$ a=22      #定义一个全局变量

[c.biancheng.net]$ echo $a    #在当前 Shell 中输出 a，成功

22

[c.biancheng.net]$ bash      #进入 Shell 子进程

[c.biancheng.net]$ echo $a    #在子进程中输出 a，失败

exit

[c.biancheng.net]$ exit      #退出 Shell 子进程，返回上一级 Shell

exit

[c.biancheng.net]$ export a   #将 a 导出为环境变量

[c.biancheng.net]$ bash      #重新进入 Shell 子进程

[c.biancheng.net]$ echo $a    #在子进程中再次输出 a，成功

22

[c.biancheng.net]$ exit      #退出 Shell 子进程

exit

[c.biancheng.net]$ exit      #退出父进程，结束整个 Shell 会话
```

可以发现，默认情况下，a 在 Shell 子进程中是无效的；使用 export 将 a 导出为环境变量后，在子进程中就可以使用了。

`export a` 这种形式是在定义变量 a 以后再将它导出为环境变量，如果想在定义的同时导出为环境变量，可以写作 `export a=22`。

我们一直强调的是环境变量在 Shell 子进程中有效，并没有说它在所有的 Shell 进程中都有效；如果你通过终端创建了一个新的 Shell 窗口，那它就不是当前 Shell 的子进程，环境变量对这个新的 Shell 进程仍然是无效的。请看下图：



第一个窗口中的环境变量 a 在第二个窗口中就无效。

环境变量也是临时的

通过 export 导出的环境变量只对当前 Shell 进程以及所有的子进程有效，如果最顶层的父进程被关闭了，那么环境变量也就随之消失了，其它的进程也就无法使用了，所以说环境变量也是临时的。

有读者可能会问，如果我想让一个变量在所有 Shell 进程中都有效，不管它们之间是否存在父子关系，该怎么办呢？

只有将变量写入 Shell 配置文件中才能达到这个目的！Shell 进程每次启动时都会执行配置文件中的代码做一些初始化工作，如果将变

量放在配置文件中，那么每次启动进程都会定义这个变量。不知道如何修改配置文件的读者请猛击 [《Shell 配置文件的加载》](#) [《编写自己的 Shell 配置文件》](#)。

3. Shell 命令替换：将命令的输出结果赋值给变量

Shell 命令替换是指将命令的输出结果赋值给某个变量。比如，在某个目录中输入 ls 命令可查看当前目录中所有的文件，但如何将输出内容存入某个变量中呢？这就需要使用命令替换了，这也是 Shell 编程中使用非常频繁的功能。

Shell 中有两种方式可以完成命令替换，一种是反引号```，一种是`$()`，使用方法如下：

```
variable=`commands`  
variable=$(commands)
```

其中，variable 是变量名，commands 是要执行的命令。commands 可以只有一个命令，也可以有多个命令，多个命令之间以分号`;`分隔。

例如，date 命令用来获得当前的系统时间，使用命令替换可以将它的结果赋值给一个变量。

```
1.  #!/bin/bash  
2.  
3.  begin_time=`date`      #开始时间，使用``替换  
4.  sleep 20s              #休眠 20 秒  
5.  finish_time=$(date)    #结束时间，使用$()替换  
6.  
7.  echo "Begin time: $begin_time"  
8.  echo "Finish time: $finish_time"
```

运行脚本，20 秒后可以看到输出结果：

```
Begin time: 2019 年 04 月 19 日 星期五 09:59:58 CST  
Finish time: 2019 年 04 月 19 日 星期五 10:00:18 CST
```

使用 date 命令的`%s` 格式控制符可以得到当前的 UNIX 时间戳，这样就可以直接计算脚本的运行时间了。UNIX 时间戳是指从 1970 年 1 月 1 日 00:00:00 到目前为止的秒数，不了解的读者请[猛击这里](#)。

```
1.  #!/bin/bash  
2.  
3.  begin_time=`date +%s`  #开始时间，使用``替换  
4.  sleep 20s              #休眠 20 秒  
5.  finish_time=$(date +%s) #结束时间，使用$()替换  
6.  run_time=$((finish_time - begin_time)) #时间差  
7.  
8.  echo "begin time: $begin_time"  
9.  echo "finish time: $finish_time"  
10. echo "run time: ${run_time}s"
```

运行脚本，20 秒后可以看到输出结果：

```
begin time: 1555639864  
finish time: 1555639884
```

run time: 20s

第 6 行代码中的`(())`是 Shell 数学计算命令。和 [C++](#)、[C#](#)、[Java](#) 等编程语言不同，在 Shell 中进行数据计算不那么方便，必须使用专门的数学计算命令，`(())`就是其中之一。更多细节我们将会在《[Shell 数学计算](#)》一节中详细讲解。

注意，如果被替换的命令的输出内容包括多行（也即有换行符），或者含有多个连续的空白符，那么在输出变量时应该将变量用双引号包围，否则系统会使用默认的空白符来填充，这会导致换行无效，以及连续的空白符被压缩成一个。请看下面的代码：

```
1.  #!/bin/bash
2.
3.  LSL=`ls -l`
4.  echo $LSL  #不使用双引号包围
5.  echo "-----"  #输出分隔符
6.  echo "$LSL"  #使用引号包围
```

运行结果：

```
total 8 drwxr-xr-x. 2 root root 21 7月 1 2016 abc -rw-rw-r--. 1 mozhiyan mozhiyan 147 10月 31 10:29 demo.sh -rw-rw-r--. 1 mozhiyan mozhiyan 35 10月 31 10:20 demo.sh~
-----

total 8

drwxr-xr-x. 2 root      root      21 7月   1 2016 abc
-rw-rw-r--. 1 mozhiyan mozhiyan 147 10月 31 10:29 demo.sh
-rw-rw-r--. 1 mozhiyan mozhiyan  35 10月 31 10:20 demo.sh~
```

所以，为了防止出现格式混乱的情况，我建议在输出变量时加上双引号。

再谈反引号和 `$()`

原则上讲，上面提到的两种变量替换的形式是等价的，可以随意使用；但是，反引号毕竟看起来像单引号，有时候会对查看代码造成困扰，而使用 `$()` 就相对清晰，能有效避免这种混乱。而且有些情况必须使用 `$()`：`$()` 支持嵌套，反引号不行。

下面的例子演示了使用计算 `ls` 命令列出的第一个文件的行数，这里使用了两层嵌套。

```
[c.biancheng.net]$ Fir_File_Lines=$(wc -l $(ls | sed -n '1p'))
[c.biancheng.net]$ echo "$Fir_File_Lines"
36 anaconda-ks.cfg
```

要注意的是，`$()` 仅在 Bash Shell 中有效，而反引号可在多种 Shell 中使用。所以这两种命令替换的方式各有特点，究竟选用哪种方式全看个人需求。

4. Shell 位置参数（命令行参数）

我们先来说一下 Shell 位置参数是怎么回事。

运行 Shell 脚本文件时我们可以给它传递一些参数，这些参数在脚本文件内部可以使用 `$n` 的形式来接收，例如，`$1` 表示第一个参数，`$2` 表示第二个参数，依次类推。

同样，在调用函数时也可以传递参数。Shell 函数参数的传递和其它编程语言不同，没有所谓的形参和实参，在定义函数时也不用指明参数的名字和数目。换句话说，定义 Shell 函数时不能带参数，但是在调用函数时却可以传递参数，这些传递进来的参数，在函数内部就使用 `$n` 的形式接收，例如，`$1` 表示第一个参数，`$2` 表示第二个参数，依次类推。

这种通过 `$n` 的形式来接收的参数，在 Shell 中称为**位置参数**。

在讲解**变量的命名**时，我们提到：变量的名字必须以字母或者下划线开头，不能以数字开头；但是位置参数却偏偏是数字，这和变量的命名规则是相悖的，所以我们将它们视为“特殊变量”。

除了 `$n`，Shell 中还有 `$#`、 `$*`、 `$@`、 `$?`、 `$$` 几个特殊参数，我们将在下节讲解。

1) 给脚本文件传递位置参数

请编写下面的代码，并命名为 `test.sh`：

```
1.  #!/bin/bash
2.
3.  echo "Language: $1"
4.  echo "URL: $2"
```

运行 `test.sh`，并附带参数：

```
[mozhiyan@localhost ~]$ cd demo

[mozhiyan@localhost demo]$ ./test.sh Shell http://c.biancheng.net/shell/

Language: Shell

URL: http://c.biancheng.net/shell/
```

其中 `Shell` 是第一个位置参数，`http://c.biancheng.net/shell/` 是第二个位置参数，两者之间以空格分隔。

2) 给函数传递位置参数

请编写下面的代码，并命名为 `test.sh`：

```
1.  #!/bin/bash
2.
3.  #定义函数
4.  function func() {
```

```
5.     echo "Language: $1"
6.     echo "URL: $2"
7. }
8.
9. #调用函数
10. func C++ http://c.biancheng.net/cplusplus/
```

运行 test.sh :

```
[mozhiyan@localhost ~]$ cd demo

[mozhiyan@localhost demo]$ ./test.sh

Language: C++

URL: http://c.biancheng.net/cplusplus/
```

关于函数定义和调用的具体语法请访问：[Shell 函数定义和调用](#)、[Shell 函数参数](#)

注意事项

如果参数个数太多，达到或者超过了 10 个，那么就得用 `${n}` 的形式来接收了，例如 `${10}`、`${23}`。`{ }` 的作用是为了帮助解释器识别参数的边界，这跟使用变量时加 `{ }` 是一样的效果。

下节展望

在 Shell 中，传递位置参数时除了能单独取得某个具体的参数，还能取得所有参数的列表，以及参数的个数等信息，下节我们将会详细讲解。

5. Shell 特殊变量：Shell \$#、\$*、\$@、\$?、\$\$

上节我们讲到了 \$n，它是特殊变量的一种，用来接收位置参数。本节我们继续讲解剩下的几个特殊变量，它们分别是：\$#、\$*、\$@、\$?、\$。

Shell 特殊变量及其含义	
变量	含义
\$0	当前脚本的文件名。
\$n (n≥1)	传递给脚本或函数的参数。n 是一个数字，表示第几个参数。例如，第一个参数是 \$1，第二个参数是 \$2。
\$#	传递给脚本或函数的参数个数。
\$*	传递给脚本或函数的所有参数。
\$@	传递给脚本或函数的所有参数。当被双引号" "包含时，\$@ 与 \$* 稍有不同，我们将在《Shell \$*和\$@的区别》一节中详细讲解。
\$?	上个命令的退出状态，或函数的返回值，我们将在《Shell \$?》一节中详细讲解。
\$\$	当前 Shell 进程 ID。对于 Shell 脚本，就是这些脚本所在的进程 ID。

下面我们通过两个例子来演示。

1) 给脚本文件传递参数

编写下面的代码，并保存为 test.sh：

```
1.  #!/bin/bash
2.
3.  echo "Process ID: $$"
4.  echo "File Name: $0"
5.  echo "First Parameter : $1"
6.  echo "Second Parameter : $2"
7.  echo "All parameters 1: $@"
8.  echo "All parameters 2: $*"
9.  echo "Total: $#"
```

运行 test.sh，并附带参数：

```
[mozhiyan@localhost demo]$ ./test.sh Shell Linux

Process ID: 5943

File Name: bash

First Parameter : Shell

Second Parameter : Linux
```

```
All parameters 1: Shell Linux
```

```
All parameters 2: Shell Linux
```

```
Total: 2
```

2) 给函数传递参数

编写下面的代码，并保存为 test.sh：

```
1.  #!/bin/bash
2.
3.  #定义函数
4.  function func() {
5.      echo "Language: $1"
6.      echo "URL: $2"
7.      echo "First Parameter : $1"
8.      echo "Second Parameter : $2"
9.      echo "All parameters 1: $@"
10.     echo "All parameters 2: $*"
11.     echo "Total: $#"
```

```
12. }
13.
14. #调用函数
15. func Java http://c.biancheng.net/java/
```

运行结果为：

Language: Java

URL: http://c.biancheng.net/java/

First Parameter : Java

Second Parameter : http://c.biancheng.net/java/

All parameters 1: Java http://c.biancheng.net/java/

All parameters 2: Java http://c.biancheng.net/java/

Total: 2

6. Shell \$*和\$@之间的区别

\$* 和 \$@ 都表示传递给函数或脚本的所有参数，我们已在《[Shell 特殊变量](#)》一节中进行了演示，本节重点说一下它们之间的区别。

当 \$* 和 \$@ 不被双引号 " 包围时，它们之间没有任何区别，都是将接收到的每个参数看做一份数据，彼此之间以空格来分隔。

但是当它们被双引号 " 包含时，就会有区别了：

- "\$*"会将所有的参数从整体上看做一份数据，而不是把每个参数都看做一份数据。
- "\$@"仍然将每个参数都看作一份数据，彼此之间是独立的。

比如传递了 5 个参数，那么对于"\$*"来说，这 5 个参数会合并到一起形成一份数据，它们之间是无法分割的；而对于"\$@"来说，这 5 个参数是相互独立的，它们是 5 份数据。

如果使用 echo 直接输出"\$*"和"\$@"做对比，是看不出区别的；但如果使用 for 循环来逐个输出数据，立即就能看出区别来。

关于 for 循环的用法请猛击：[Shell for 循环和 for int 循环详解](#)

编写下面的代码，并保存为 test.sh：

```
1.  #!/bin/bash
2.
3.  echo "print each param from \"$*"\"
4.  for var in "$*"
5.  do
6.      echo "$var"
7.  done
8.
9.  echo "print each param from \"$@"\"
10. for var in "$@"
11. do
12.     echo "$var"
13. done
```

运行 test.sh，并附带参数：

```
[mozhiyan@localhost demo]$ ./test.sh a b c d
print each param from "$*"
a b c d
print each param from "$@"
a
b
c
d
```

从运行结果可以发现，对于"\$*"，只循环了 1 次，因为它只有 1 份数据；对于"\$@"，循环了 5 次，因为它有 5 份数据。

7. Shell \$? : 获取函数返回值或者上一个命令的退出状态

\$? 是一个特殊变量，用来获取上一个命令的退出状态，或者上一个函数的返回值。

所谓退出状态，就是上一个命令执行后的返回结果。退出状态是一个数字，一般情况下，大部分命令执行成功会返回 0，失败返回 1，这和 C 语言的 main() 函数是类似的。

不过，也有一些命令返回其他值，表示不同类型的错误。

1) \$? 获取上一个命令的退出状态

编写下面的代码，并保存为 test.sh：

```
1.  #!/bin/bash
2.
3.  if [ "$1" == 100 ]
4.  then
5.      exit 0 #参数正确，退出状态为 0
6.  else
7.      exit 1 #参数错误，退出状态 1
8.  fi
```

`exit` 表示退出当前 Shell 进程，我们必须在新进程中运行 test.sh，否则当前 Shell 会话（终端窗口）会被关闭，我们就无法取得它的退出状态了。

例如，运行 test.sh 时传递参数 100：

```
[mozhiyan@localhost ~]$ cd demo

[mozhiyan@localhost demo]$ bash ./test.sh 100 #作为一个新进程运行

[mozhiyan@localhost demo]$ echo $?

0
```

再如，运行 test.sh 时传递参数 89：

```
[mozhiyan@localhost demo]$ bash ./test.sh 89 #作为一个新进程运行

[mozhiyan@localhost demo]$ echo $?

1
```

2) \$? 获取函数的返回值

编写下面的代码，并保存为 test.sh：

```
1.  #!/bin/bash
2.
3.  #得到两个数相加的和
4.  function add() {
5.      return `expr $1 + $2`
6.  }
7.
8.  add 23 50 #调用函数
9.  echo $? #获取函数返回值
```

运行结果：

73

有 [C++](#)、[C#](#)、[Java](#) 等编程经验的读者请注意：严格来说，Shell 函数中的 return 关键字用来表示函数的退出状态，而不是函数的返回值；Shell 不像其它编程语言，没有专门处理返回值的关键字。

以上处理方案在其它编程语言中没有任何问题，但是在 Shell 中是非常错误的，Shell 函数的返回值和其它编程语言大有不同，我们将在《[Shell 函数返回值](#)》中展开讨论。

8. Shell 字符串详解

字符串（String）就是一系列字符的组合。字符串是 Shell 编程中最常用的数据类型之一（除了数字和字符串，也没有其他类型了）。

字符串可以由单引号`'`包围，也可以由双引号`"`包围，也可以不用引号。它们之间是有区别的，稍后我们会详解。

字符串举例：

```
1.  str1=c.biancheng.net
2.  str2="shell script"
3.  str3='C 语言中文网'
```

下面我们说一下三种形式的区别：

1) 由单引号`'`包围的字符串：

- 任何字符都会原样输出，在其中使用变量是无效的。
- 字符串中不能出现单引号，即使对单引号进行转义也不行。

2) 由双引号`"`包围的字符串：

- 如果其中包含了某个变量，那么该变量会被解析（得到该变量的值），而不是原样输出。
- 字符串中可以出现双引号，只要它被转义了就行。

3) 不被引号包围的字符串

- 不被引号包围的字符串中出现变量时也会被解析，这一点和双引号`"`包围的字符串一样。
- 字符串中不能出现空格，否则空格后边的字符串会作为其他变量或者命令解析。

我们通过代码来演示一下三种形式的区别：

```
1.  #!/bin/bash
2.
3.  n=74
4.  str1=c.biancheng.net$n str2="shell \"script\" $n"
5.  str3='C 语言中文网 $n'
6.
7.  echo $str1
8.  echo $str2
9.  echo $str3
```

运行结果：

```
c.biancheng.net74
shell "script" 74
C 语言中文网 $n
```

str1 中包含了 `$n`，它被解析为变量 `n` 的引用。`$n` 后边有空格，紧随空格的是 `str2`；Shell 将 `str2` 解释为一个新的变量名，而不是作为字符串 `str1` 的一部分。

str2 中包含了引号，但是被转义了（由反斜杠 `\` 开头的表示转义字符）。str2 中也包含了 `$n`，它也被解析为变量 `n` 的引用。

str3 中也包含了 `$n`，但是仅仅是作为普通字符，并没有解析为变量 `n` 的引用。

获取字符串长度

在 Shell 中获取字符串长度很简单，具体方法如下：

```
${#string_name}
```

`string_name` 表示字符串名字。

下面是具体的演示：

[纯文本复制](#)

```
1.  #!/bin/bash
2.
3.  str="http://c.biancheng.net/shell/"
4.  echo ${#str}
```

运行结果：

29

9. Shell 字符串拼接（连接、合并）

在脚本语言中，字符串的拼接（也称字符串连接或者字符串合并）往往都非常简单，例如：

- 在 [PHP](#) 中，使用 `.` 即可连接两个字符串；
- 在 [JavaScript](#) 中，使用 `+` 即可将两个字符串合并为一个。

然而，在 Shell 中你不需要使用任何运算符，将两个字符串并排放在一起就能实现拼接，非常简单粗暴。请看下面的例子：

```
1.  #!/bin/bash
2.
3.  name="Shell"
4.  url="http://c.biancheng.net/shell/"
5.
6.  str1=$name$url  #中间不能有空格
7.  str2="$name $url"  #如果被双引号包围，那么中间可以有空格
8.  str3=$name": "$url  #中间可以出现别的字符串
9.  str4="$name: $url"  #这样写也可以
10. str5="${name}Script: ${url}index.html"  #这个时候需要给变量名加上大括号
11.
12. echo $str1
13. echo $str2
14. echo $str3
15. echo $str4
16. echo $str5
```

运行结果：

```
Shellhttp://c.biancheng.net/shell/
Shell http://c.biancheng.net/shell/
Shell: http://c.biancheng.net/shell/
Shell: http://c.biancheng.net/shell/
ShellScript: http://c.biancheng.net/shell/index.html
```

对于第 7 行代码，`$name` 和 `$url` 之间之所以不能出现空格，是因为当字符串不被任何一种引号包围时，遇到空格就认为字符串结束了，空格后边的内容会作为其他变量或者命令解析，这一点在《[Shell 字符串](#)》中已经提到。

对于第 10 行代码，加 `{ }` 是为了帮助解释器识别变量的边界，这一点在《[Shell 变量](#)》中已经提到。

Shell 这种拼接字符串的方式和 [Python](#) 非常类似，Python 既支持用 `+` 拼接字符串，也支持将两个字符串放在一起，读者可以猛击《[Python 字符串](#)》了解详情。

10.Shell 字符串截取（非常详细）

Shell 截取字符串通常有两种方式：从指定位置开始截取和从指定字符（子字符串）开始截取。

从指定位置开始截取

这种方式需要两个参数：除了指定起始位置，还需要截取长度，才能最终确定要截取的字符串。

既然需要指定起始位置，那么就涉及到计数方向的问题，到底是字符串左边开始计数，还是从字符串右边开始计数。答案是 Shell 同时支持两种计数方式。

1) 从字符串左边开始计数

如果想从字符串的左边开始计数，那么截取字符串的具体格式如下：

```
${string: start :length}
```

其中，string 是要截取的字符串，start 是起始位置（从左边开始，从 0 开始计数），length 是要截取的长度（省略的话表示直到字符串的末尾）。

例如：

```
1. url="c.biancheng.net"
2. echo ${url: 2: 9}
```

结果为 biancheng。

再如：

```
1. url="c.biancheng.net"
2. echo ${url: 2} #省略 length，截取到字符串末尾
```

结果为 biancheng.net。

2) 从右边开始计数

如果想从字符串的右边开始计数，那么截取字符串的具体格式如下：

```
${string: 0-start :length}
```

同第 1) 种格式相比，第 2) 种格式仅仅多了 0-，这是固定的写法，专门用来表示从字符串右边开始计数。

这里需要强调两点：

- 从左边开始计数时，起始数字是 0（这符合程序员思维）；从右边开始计数时，起始数字是 1（这符合常人思维）。计数方向不同，起始数字也不同。
- 不管从哪边开始计数，截取方向都是从左到右。

例如：

```
1. url="c.biancheng.net"
```

```
2. echo ${url: 0-13: 9}
```

结果为 biancheng。从右边数，b 是第 13 个字符。

再如：

```
1. url="c.biancheng.net"
```

```
2. echo ${url: 0-13} #省略 length，直接截取到字符串末尾
```

结果为 biancheng.net。

从指定字符（子字符串）开始截取

这种截取方式无法指定字符串长度，只能从指定字符（子字符串）截取到字符串末尾。Shell 可以截取指定字符（子字符串）右边的所有字符，也可以截取左边的所有字符。

1) 使用 # 号截取右边字符

使用 # 号可以截取指定字符（或者子字符串）右边的所有字符，具体格式如下：

```
${string#*chars}
```

其中，string 表示要截取的字符，chars 是指定的字符（或者子字符串），* 是通配符的一种，表示任意长度的字符串。*chars 连起来使用的意思是：忽略左边的所有字符，直到遇见 chars（chars 不会被截取）。

请看下面的例子：

```
1. url="http://c.biancheng.net/index.html"
```

```
2. echo ${url#*:}
```

结果为 //c.biancheng.net/index.html。

以下写法也可以得到同样的结果：

```
1. echo ${url#*p:}
```

```
2. echo ${url#*ttp:}
```

如果不需要忽略 chars 左边的字符，那么也可以不写 *，例如：

```
1. url="http://c.biancheng.net/index.html"
```

```
2. echo ${url#http://}
```

结果为 c.biancheng.net/index.html。

注意，以上写法遇到第一个匹配的字符（子字符串）就结束了。例如：

```
1. url="http://c.biancheng.net/index.html"
```

```
2. echo ${url#*/}
```

结果为 `/c.biancheng.net/index.html`。url 字符串中有三个 `/`，输出结果表明，Shell 遇到第一个 `/` 就匹配结束了。

如果希望直到最后一个指定字符（子字符串）再匹配结束，那么可以使用 `##`，具体格式为：

```
${string##*chars}
```

请看下面的例子：

```
1. #!/bin/bash
2.
3. url="http://c.biancheng.net/index.html"
4. echo ${url#*/}    #结果为 /c.biancheng.net/index.html
5. echo ${url##*/}   #结果为 index.html
6.
7. str="---aa+++aa@@"
8. echo ${str#aa}    #结果为 +++aa@@
9. echo ${str##aa}   #结果为 @@@
```

2) 使用 % 截取左边字符

使用 `%` 号可以截取指定字符（或者子字符串）左边的所有字符，具体格式如下：

```
${string%chars*}
```

请注意 `*` 的位置，因为要截取 chars 左边的字符，而忽略 chars 右边的字符，所以 `*` 应该位于 chars 的右侧。其他方面 `%` 和 `#` 的用法相同，这里不再赘述，仅举例说明：

```
1. #!/bin/bash
2.
3. url="http://c.biancheng.net/index.html"
4. echo ${url%/*}    #结果为 http://c.biancheng.net
5. echo ${url%%/*}   #结果为 http:
6.
7. str="---aa+++aa@@"
8. echo ${str%aa*}   #结果为 ---aa+++
9. echo ${str%%aa*}  #结果为 ---
```

汇总

最后，我们对以上 8 种格式做一个汇总，请看下表：

格式	说明
<code>\${string: start :length}</code>	从 string 字符串的左边第 start 个字符开始，向右截取 length 个字符。
<code>\${string: start}</code>	从 string 字符串的左边第 start 个字符开始截取，直到最后。
<code>\${string: 0-start :length}</code>	从 string 字符串的右边第 start 个字符开始，向右截取 length 个字符。
<code>\${string: 0-start}</code>	从 string 字符串的右边第 start 个字符开始截取，直到最后。
<code>\${string#*chars}</code>	从 string 字符串第一次出现 *chars 的位置开始，截取 *chars 右边的所有字符。
<code>\${string##*chars}</code>	从 string 字符串最后一次出现 *chars 的位置开始，截取 *chars 右边的所有字符。
<code>\${string%*chars}</code>	从 string 字符串第一次出现 *chars 的位置开始，截取 *chars 左边的所有字符。
<code>\${string%%*chars}</code>	从 string 字符串最后一次出现 *chars 的位置开始，截取 *chars 左边的所有字符。

11.Shell 数组：Shell 数组定义以及获取数组元素

和其他编程语言一样，Shell 也支持数组。数组（Array）是若干数据的集合，其中的每一份数据都称为元素（Element）。

Shell 并且没有限制数组的大小，理论上可以存放无限量的数据。和 [C++](#)、[Java](#)、[C#](#) 等类似，Shell 数组元素的下标也是从 0 开始计数。

获取数组中的元素要使用下标[]，下标可以是一个整数，也可以是一个结果为整数的表达式；当然，下标必须大于等于 0。

遗憾的是，常用的 Bash Shell 只支持一维数组，不支持多维数组。

Shell 数组的定义

在 Shell 中，用括号()来表示数组，数组元素之间用空格来分隔。由此，定义数组的一般形式为：

```
array_name=(ele1 ele2 ele3 ... elen)
```

注意，赋值号=两边不能有空格，必须紧挨着数组名和数组元素。

下面是一个定义数组的实例：

```
1. nums=(29 100 13 8 91 44)
```

Shell 是弱类型的，它并不要求所有数组元素的类型必须相同，例如：

```
1. arr=(20 56 "http://c.biancheng.net/shell/")
```

第三个元素就是一个“异类”，前面两个元素都是整数，而第三个元素是字符串。

Shell 数组的长度不是固定的，定义之后还可以增加元素。例如，对于上面的 nums 数组，它的长度是 6，使用下面的代码会在最后增加一个元素，使其长度扩展到 7：

```
1. nums[6]=88
```

此外，你也无需逐个元素地给数组赋值，下面的代码就是只给特定元素赋值：

```
1. ages=([3]=24 [5]=19 [10]=12)
```

以上代码就只给第 3、5、10 个元素赋值，所以数组长度是 3。

获取数组元素

获取数组元素的值，一般使用下面的格式：

```
${array_name[index]}
```

其中，array_name 是数组名，index 是下标。例如：

```
1. n=${nums[2]}
```

表示获取 nums 数组的第二个元素，然后赋值给变量 n。再如：

```
1. echo ${nums[3]}
```

表示输出 nums 数组的第 3 个元素。

使用@或*可以获取数组中的所有元素，例如：

```
1. ${nums[*]}
```

```
2. ${nums[@]}
```

两者都可以得到 nums 数组的所有元素。

完整的演示：

```
1. #!/bin/bash
2.
3. nums=(29 100 13 8 91 44)
4. echo ${nums[@]} #输出所有数组元素
5. nums[10]=66 #给第 10 个元素赋值（此时会增加数组长度）
6. echo ${nums[*]} #输出所有数组元素
7. echo ${nums[4]} #输出第 4 个元素
```

运行结果：

29 100 13 8 91 44

29 100 13 8 91 44 66

91

12. Shell 获取数组长度

所谓数组长度，就是数组元素的个数。

利用 `@` 或 `*`，可以将数组扩展成列表，然后使用 `#` 来获取数组元素的个数，格式如下：

```
#{array_name[@]}
#{array_name[*]}
```

其中 `array_name` 表示数组名。两种形式是等价的，选择其一即可。

如果某个元素是字符串，还可以通过指定下标的方式获得该元素的长度，如下所示：

```
#{arr[2]}
```

获取 `arr` 数组的第 2 个元素（假设它是字符串）的长度。

回忆字符串长度的获取

回想一下 Shell 是如何获取字符串长度的呢？其实和获取数组长度如出一辙，它的格式如下：

```
#{string_name}
```

`string_name` 是字符串名。

实例演示

下面我们通过实际代码来演示一下如何获取数组长度。

[纯文本复制](#)

```
1.  #!/bin/bash
2.
3.  nums=(29 100 13)
4.  echo ${#nums[*]}
5.
6.  #向数组中添加元素
7.  nums[10]="http://c.biancheng.net/shell/"
8.  echo ${#nums[@]}
9.  echo ${#nums[10]}
10.
11. #删除数组元素
12. unset nums[1]
13. echo ${#nums[*]}
```

运行结果：

3

4

29

3

13. Shell 数组拼接，Shell 数组合并

所谓 Shell 数组拼接（数组合并），就是将两个数组连接成一个数组。

拼接数组的思路是：先利用 `@` 或 `*`，将数组扩展成列表，然后再合并到一起。具体格式如下：

```
array_new=(${array1[@]} ${array2[@]})  
array_new=(${array1[*]} ${array2[*]})
```

两种方式是等价的，选择其一即可。其中，`array1` 和 `array2` 是需要拼接的数组，`array_new` 是拼接后形成的新数组。

下面是完整的演示代码：

```
1.  #!/bin/bash  
2.  
3.  array1=(23 56)  
4.  array2=(99 "http://c.biancheng.net/shell/")  
5.  array_new=(${array1[@]} ${array2[*]})  
6.  
7.  echo ${array_new[@]}  #也可以写作 ${array_new[*]}
```

运行结果：

23 56 99 <http://c.biancheng.net/shell/>

14. Shell 删除数组元素（也可以删除整个数组）

在 Shell 中，使用 `unset` 关键字来删除数组元素，具体格式如下：

```
unset array_name[index]
```

其中，`array_name` 表示数组名，`index` 表示数组下标。

如果不写下标，而是写成下面的形式：

```
unset array_name
```

那么就是删除整个数组，所有元素都会消失。

下面我们通过具体的代码来演示：

```
1.  #!/bin/bash
2.
3.  arr=(23 56 99 "http://c.biancheng.net/shell/")
4.  unset arr[1]
5.  echo ${arr[@]}
6.
7.  unset arr
8.  echo ${arr[*]}
```

运行结果：

```
23 99 http://c.biancheng.net/shell/
```

注意最后的空行，它表示什么也没输出，因为数组被删除了，所以输出为空

15. Shell 关联数组（下标是字符串的数组）

现在最新的 Bash Shell 已经支持关联数组了。关联数组使用字符串作为下标，而不是整数，这样可以做到见名知意。

关联数组也称为“键值对（key-value）”数组，键（key）也即字符串形式的数组下标，值（value）也即元素值。

例如，我们可以创建一个叫做 color 的关联数组，并用颜色名字作为下标。

```
1. declare -A color
2. color["red"]="ff0000"
3. color["green"]="00ff00"
4. color["blue"]="0000ff"
```

也可以在定义的同时赋值：

```
1. declare -A color=(["red"]="ff0000", ["green"]="00ff00", ["blue"]="0000ff")
```

不同于普通数组，关联数组必须使用带有 `-A` 选项的 `declare` 命令创建。关于 `declare` 命令的详细用法请访问：[Shell declare 和 typeset 命令：设置变量属性](#)

访问关联数组元素

访问关联数组元素的方式几乎与普通数组相同，具体形式为：

```
array_name["index"]
```

例如：

```
1. color["white"]="ffffff"
2. color["black"]="000000"
```

加上 `$()` 即可获取数组元素的值：

```
${array_name["index"]}
```

例如：

```
1. echo ${color["white"]}
2. white=${color["black"]}
```

获取所有元素的下标和值

使用下面的形式可以获得关联数组的所有元素值：

```
 ${array_name[@]}  
 ${array_name[*]}
```

使用下面的形式可以获取关联数组的所有下标值：

```
 ${!array_name[@]}  
 ${!array_name[*]}
```

获取关联数组长度

使用下面的形式可以获得关联数组的长度：

```
 ${#array_name[*]}  
 ${#array_name[@]}
```

关联数组实例演示：

```
1.  #!/bin/bash  
2.  
3.  declare -A color  
4.  color["red"]="ff0000"  
5.  color["green"]="00ff00"  
6.  color["blue"]="0000ff"  
7.  color["white"]="ffffff"  
8.  color["black"]="000000"  
9.  
10. #获取所有元素值  
11. for value in ${color[*]}  
12. do  
13.     echo $value  
14. done  
15. echo "*****"  
16.  
17. #获取所有元素下标（键）  
18. for key in ${!color[*]}  
19. do  
20.     echo $key  
21. done  
22. echo "*****"
```

```
23.  
24. #列出所有键值对  
25. for key in ${!color[@]}  
26. do  
27.     echo "${key} -> ${color[$key]}"  
28. done
```

运行结果：

#ff0000

#0000ff

#ffffff

#000000

#00ff00

red

blue

white

black

green

red -> #ff0000

blue -> #0000ff

white -> #ffffff

black -> #000000

green -> #00ff00

16.Shell 内建命令（内置命令）

所谓 Shell 内建命令，就是由 Bash 自身提供的命令，而不是文件系统中的某个可执行文件。

例如，用于进入或者切换目录的 cd 命令，虽然我们一直在使用它，但如果不加以注意很难意识到它与普通命令的性质是不一样的：该命令并不是某个外部文件，只要在 Shell 中你就一定可以运行这个命令。

可以使用 type 来确定一个命令是否是内建命令：

```
[root@localhost ~]# type cd
cd is a Shell builtin
[root@localhost ~]# type ifconfig
ifconfig is /sbin/ifconfig
```

由此可见，cd 是一个 Shell 内建命令，而 ifconfig 是一个外部文件，它的位置是 /sbin/ifconfig。

还记得系统变量 \$PATH 吗？\$PATH 变量包含的目录中几乎聚集了系统中绝大多数的可执行命令，它们都是外部命令。

通常来说，内建命令会比外部命令执行得更快，执行外部命令时不但会触发磁盘 I/O，还需要 fork 出一个单独的进程来执行，执行完成后再退出。而执行内建命令相当于调用当前 Shell 进程的一个函数。

下表列出了 Bash Shell 中直接可用的内建命令。

Bash Shell 内建命令	
命令	说明
:	扩展参数列表，执行重定向操作
.	读取并执行指定文件中的命令（在当前 shell 环境中）
alias	为指定命令定义一个别名
bg	将作业以后台模式运行
bind	将键盘序列绑定到一个 readline 函数或宏
break	退出 for、while、select 或 until 循环
builtin	执行指定的 shell 内建命令
caller	返回活动子函数调用的上下文
cd	将当前目录切换为指定的目录
command	执行指定的命令，无需进行通常的 shell 查找
compgen	为指定单词生成可能的补全匹配
complete	显示指定的单词是如何补全的
compgopt	修改指定单词的补全选项
continue	继续执行 for、while、select 或 until 循环的下一迭代
declare	声明一个变量或变量类型。
dirs	显示当前存储目录的列表
disown	从进程作业表中删除指定的作业
echo	将指定字符串输出到 STDOUT
enable	启用或禁用指定的内建 shell 命令
eval	将指定的参数拼接成一个命令，然后执行该命令
exec	用指定命令替换 shell 进程

exit	强制 shell 以指定的退出状态码退出
export	设置子 shell 进程可用的变量
fc	从历史记录中选择命令列表
fg	将作业以前台模式运行
getopts	分析指定的位置参数
hash	查找并记住指定命令的全路径名
help	显示帮助文件
history	显示命令历史记录
jobs	列出活动作业
kill	向指定的进程 ID(PID) 发送一个系统信号
let	计算一个数学表达式中的每个参数
local	在函数中创建一个作用域受限的变量
logout	退出登录 shell
mapfile	从 STDIN 读取数据行，并将其加入索引数组
popd	从目录栈中删除记录
printf	使用格式化字符串显示文本
pushd	向目录栈添加一个目录
pwd	显示当前工作目录的路径名
read	从 STDIN 读取一行数据并将其赋给一个变量
readarray	从 STDIN 读取数据行并将其放入索引数组
readonly	从 STDIN 读取一行数据并将其赋给一个不可修改的变量
return	强制函数以某个值退出，这个值可以被调用脚本提取
set	设置并显示环境变量的值和 shell 属性
shift	将位置参数依次向下降一个位置
shopt	打开/关闭控制 shell 可选行为的变量值
source	读取并执行指定文件中的命令（在当前 shell 环境中）
suspend	暂停 Shell 的执行，直到收到一个 SIGCONT 信号
test	基于指定条件返回退出状态码 0 或 1
times	显示累计的用户和系统时间
trap	如果收到了指定的系统信号，执行指定的命令
type	显示指定的单词如果作为命令将会如何被解释
typeset	声明一个变量或变量类型。
ulimit	为系统用户设置指定的资源的上限
umask	为新建的文件和目录设置默认权限
unalias	删除指定的别名
unset	删除指定的环境变量或 shell 属性
wait	等待指定的进程完成，并返回退出状态码

接下来的几节我们将重点讲解几个常用的 Shell 内置命令。

17. Shell alias：给命令创建别名

alias 用来给命令创建一个别名。若直接输入该命令且不带任何参数，则列出当前 Shell 进程中使用了哪些别名。现在你应该能理解类似 ll 这样的命令为什么与 ls -l 的效果是一样的吧。

下面让我们来看一下有哪些命令被默认创建了别名：

```
[mozhiyan@localhost ~]$ alias
alias cp='cp -i'
alias l.='ls -d .* --color=tty'
alias ll='ls -l --color=tty'
alias ls='ls --color=tty'
alias mv='mv -i'
alias rm='rm -i'
alias which='alias | /usr/bin/which --tty-only --read-alias --show-dot --show-tilde'
```

你看，为了让我们使用方便，Shell 会给某些命令默认创建别名。

使用 alias 命令自定义别名

使用 alias 命令自定义别名的语法格式为：

```
alias new_name='command'
```

比如，一般的关机命令是 shutdown -h now，写起来比较长，这时可以重新定义一个关机命令，以后就方便多了。

```
alias myShutdown='shutdown -h now'
```

再如，通过 date 命令可以获得当前的 UNIX 时间戳，具体写法为 date +%s，如果你嫌弃它太长或者不容易记住，那可以给它定义一个别名。

```
alias timestamp='date +%s'
```

在《[Shell 命令替换](#)》一节中，我们使用 date +%s 计算脚本的运行时间，现在学了 alias，就可以简化代码了。

```
1.  #!/bin/bash
2.
3.  alias timestamp='date +%s'
4.
5.  begin=`timestamp`
6.  sleep 20s
7.  finish=$(timestamp)
8.  difference=$((finish - begin))
9.
10. echo "run time: ${difference}s"
```


运行脚本，20 秒后看到输出结果：
run time: 20s

别名只是临时的

在代码中使用 alias 命令定义的别名只能在当前 Shell 进程中使用，在子进程和其它进程中都不能使用。当前 Shell 进程结束后，别名也随之消失。

要想让别名对所有的 Shell 进程都有效，就得把别名写入 Shell 配置文件。Shell 进程每次启动时都会执行配置文件中的代码做一些初始化工作，将别名放在配置文件中，那么每次启动进程都会定义这个别名。不知道如何修改配置文件的读者请猛击 [《Shell 配置文件的加载》](#) [《编写自己的 Shell 配置文件》](#)。

使用 unalias 命令删除别名

使用 unalias 内建命令可以删除当前 Shell 进程中的别名。unalias 有两种使用方法：

- 第一种用法是在命令后跟上某个命令的别名，用于删除指定的别名。
- 第二种用法是在命令后接 -a 参数，删除当前 Shell 进程中所有的别名。

同样，这两种方法都是在当前 Shell 进程中生效的。要想永久删除配置文件中定义的别名，只能进入该文件手动删除。

```
# 删除 ll 别名
[mozhiyan@localhost ~]$ unalias ll
# 再次运行该命令时，报“找不到该命令”的错误，说明该别名被删除了
[mozhiyan@localhost ~]$ ll
-bash: ll: command not found
```

18.Shell echo 命令：输出字符串

echo 是一个 [Shell 内建命令](#)，用来在终端输出字符串，并在最后默认加上换行符。请看下面的例子：

```
1.  #!/bin/bash
2.
3.  name="Shell 教程"
4.  url="http://c.biancheng.net/shell/"
5.
6.  echo "读者，你好！" #直接输出字符串
7.  echo $url #输出变量
8.  echo "${name} 的网址是：${url}" #双引号包围的字符串中可以解析变量
9.  echo '${name} 的网址是：${url}' #单引号包围的字符串中不能解析变量
```

运行结果：

```
读者，你好！
http://c.biancheng.net/shell/
Shell 教程的网址是：http://c.biancheng.net/shell/
${name}的网址是：${url}
```

不换行

echo 命令输出结束后默认会换行，如果不希望换行，可以加上 `-n` 参数，如下所示：

```
1.  #!/bin/bash
2.
3.  name="Tom"
4.  age=20
5.  height=175
6.  weight=62
7.
8.  echo -n "${name} is ${age} years old, "
9.  echo -n "${height}cm in height "
10. echo "and ${weight}kg in weight."
11. echo "Thank you!"
```

运行结果：

```
Tom is 20 years old, 175cm in height and 62kg in weight.
Thank you!
```

输出转义字符

默认情况下，echo 不会解析以反斜杠开头的转义字符。比如，`\n` 表示换行，echo 默认会将它作为普通字符对待。请看下面的例子：

```
[root@localhost ~]# echo "hello \nworld"
hello \nworld
```

我们可以添加 `-e` 参数来让 echo 命令解析转义字符。例如：

```
[root@localhost ~]# echo -e "hello \nworld"
hello
world
```

`\c` 转义字符

有了 `-e` 参数，我们也可以使用转义字符 `\c` 来强制 echo 命令不换行了。请看下面的例子：

```
1.  #!/bin/bash
2.
3.  name="Tom"
4.  age=20
5.  height=175
6.  weight=62
7.
8.  echo -e "${name} is ${age} years old, \c"
9.  echo -e "${height}cm in height \c"
10. echo "and ${weight}kg in weight."
11. echo "Thank you!"
```

运行结果：

```
Tom is 20 years old, 175cm in height and 62kg in weight.
Thank you!
```

19. Shell read 命令：读取从键盘输入的数据

read 是 [Shell 内置命令](#)，用来从标准输入中读取数据并赋值给变量。如果没有进行重定向，默认就是从键盘读取用户输入的数据；如果进行了重定向，那么可以从文件中读取数据。

后续我们会在《[Linux Shell 重定向](#)》一节中深入讲解重定向的概念，不了解的读者可以不用理会，暂时就认为：read 命令就是从键盘读取数据。

read 命令的用法为：

```
read [-options] [variables]
```

`options` 表示选项，如下表所示；`variables` 表示用来存储数据的变量，可以有一个，也可以有多个。

`options` 和 `variables` 都是可选的，如果没有提供变量名，那么读取的数据将存放到环境变量 `REPLY` 中。

Shell read 命令支持的选项	
选项	说明
-a array	把读取的数据赋值给数组 array，从下标 0 开始。
-d delimiter	用字符串 delimiter 指定读取结束的位置，而不是一个换行符（读取到的数据不包括 delimiter）。
-e	在获取用户输入的时候，对功能键进行编码转换，不会直接显示功能键对应的字符。
-n num	读取 num 个字符，而不是整行字符。
-p prompt	显示提示信息，提示内容为 prompt。
-r	原样读取（Raw mode），不把反斜杠字符解释为转义字符。
-s	静默模式（Silent mode），不会在屏幕上显示输入的字符。当输入密码和其它确认信息的时候，这是很有必要的。
-t seconds	设置超时时间，单位为秒。如果用户没有在指定时间内输入完成，那么 read 将会返回一个非 0 的退出状态，表示读取失败。
-u fd	使用文件描述符 fd 作为输入源，而不是标准输入，类似于重定向。

【实例 1】使用 read 命令给多个变量赋值。

```
1. #!/bin/bash
2.
3. read -p "Enter some information > " name url age
4. echo "网站名字: $name"
5. echo "网址: $url"
6. echo "年龄: $age"
```

运行结果：

```
Enter some information > C 语言中文网 http://c.biancheng.net 7
网站名字: C 语言中文网
网址: http://c.biancheng.net
年龄: 7
```

注意，必须在一行内输入所有的值，不能换行，否则只能给第一个变量赋值，后续变量都会赋值失败。

本例还使用了 `-p` 选项，该选项会用一段文本来提示用户输入。

【示例 2】只读取一个字符。

```
1.  #!/bin/bash
2.
3.  read -n 1 -p "Enter a char > " char
4.  printf "\n" #换行
5.  echo $char
```

运行结果：

```
Enter a char > 1
1
```

`-n 1` 表示只读取一个字符。运行脚本后，只要用户输入一个字符，立即读取结束，不用等待用户按下回车键。

`printf "\n"` 语句用来达到换行的效果，否则 `echo` 的输出结果会和用户输入的内容位于同一行，不容易区分。

【实例 3】在指定时间内输入密码。

```
1.  #!/bin/bash
2.
3.  if
4.      read -t 20 -sp "Enter password in 20 seconds(once) > " pass1 && printf "\n" && #第一次输入密码
5.      read -t 20 -sp "Enter password in 20 seconds(again)> " pass2 && printf "\n" && #第二次输入密码
6.      [ $pass1 == $pass2 ] #判断两次输入的密码是否相等
7.  then
8.      echo "Valid password"
9.  else
10.     echo "Invalid password"
11.  fi
```

这段代码中，我们使用 `&&` 组合了多个命令，这些命令会依次执行，并且从整体上作为 `if` 语句的判断条件，只要其中一个命令执行失败（退出状态为非 0 值），整个判断条件就失败了，后续的命令也就没有必要执行了。

如果两次输入密码相同，运行结果为：

```
Enter password in 20 seconds(once) >
```

```
Enter password in 20 seconds(again)>
```

```
Valid password
```

如果两次输入密码不同，运行结果为：

Enter password in 20 seconds(once) >

Enter password in 20 seconds(again)>

Invalid password

如果第一次输入超时，运行结果为：

Enter password in 20 seconds(once) > Invalid password

如果第二次输入超时，运行结果为：

Enter password in 20 seconds(once) >

Enter password in 20 seconds(again)> Invalid password

20. Shell exit 命令：退出当前进程

exit 是一个 [Shell 内置命令](#)，用来退出当前 Shell 进程，并返回一个退出状态；使用 \$? 可以接收这个退出状态，这一点已在《[Shell \\$?](#)》中进行了讲解。

exit 命令可以接受一个整数值作为参数，代表退出状态。如果不指定，默认状态值是 0。

一般情况下，退出状态为 0 表示成功，退出状态为非 0 表示执行失败（出错）了。

exit 退出状态只能是一个介于 0~255 之间的整数，其中只有 0 表示成功，其它值都表示失败。

Shell 进程执行出错时，可以根据退出状态来判断具体出现了什么错误，比如打开一个文件时，我们可以指定 1 表示文件不存在，2 表示文件没有读取权限，3 表示文件类型不对。

编写下面的脚本，并命名为 test.sh：

```
1.  #!/bin/bash
2.
3.  echo "befor exit"
4.  exit 8
5.  echo "after exit"
```

运行该脚本：

```
[mozhiyan@localhost ~]$ bash ./test.sh
befor exit
```

可以看到，"after exit"并没有输出，这说明遇到 exit 命令后，test.sh 执行就结束了。

注意，exit 表示退出当前 Shell 进程，我们必须在新进程中运行 test.sh，否则当前 Shell 会话（终端窗口）会被关闭，我们就无法看到输出结果了。

我们可以紧接着使用 \$? 来获取 test.sh 的退出状态：

```
[mozhiyan@localhost ~]$ echo $?
8
```

< [Shell read 命令](#) [Shell declare 和 typeset 命令](#) >

21. Shell declare 和 typeset 命令：设置变量属性

declare 和 typeset 都是 Shell 内建命令，它们的用法相同，都用来设置变量的属性。不过 typeset 已经被弃用了，建议使用 declare 代替。

declare 命令的用法如下所示：

```
declare [+/-] [aAfFgiLprtux] [变量名=变量值]
```

其中，-表示设置属性，+表示取消属性，aAfFgiLprtux 都是具体的选项，它们的含义如下表所示：

选项	含义
-f [name]	列出之前由用户在脚本中定义的函数名称和函数体。
-F [name]	仅列出自定义函数名称。
-g name	在 Shell 函数内部创建全局变量。
-p [name]	显示指定变量的属性和值。
-a name	声明变量为普通数组。
-A name	声明变量为关联数组（支持索引下标为字符串）。
-i name	将变量定义为整数型。
-r name[=value]	将变量定义为只读（不可修改和删除），等价于 readonly name。
-x name[=value]	将变量设置为环境变量，等价于 export name[=value]。

【实例 1】将变量声明为整数并进行计算。

```
1.  #!/bin/bash
2.
3.  declare -i m n ret #将多个变量声明为整数
4.  m=10
5.  n=30
6.  ret=$m+$n
7.  echo $ret
```

运行结果：

40

【实例 2】将变量定义为只读变量。

```
[c.biancheng.net]$ declare -r n=10

[c.biancheng.net]$ n=20

bash: n: 只读变量

[c.biancheng.net]$ echo $n

10
```

【实例 3】显示变量的属性和值。


```
[c.biancheng.net]$ declare -r n=10
```

```
[c.biancheng.net]$ declare -p n
```

```
declare -r n="10"
```

22. Shell 数学计算（算术运算，加减乘除运算）

如果要执行算术运算（数学计算），就离不开各种运算符号，和其他编程语言类似，Shell 也有很多算术运算符，下面就给大家介绍一下常见的 Shell 算术运算符，如下表所示。

Shell 算术运算符一览表	
算术运算符	说明/含义
+, -	加法（或正号）、减法（或负号）
*, /, %	乘法、除法、取余（取模）
**	幂运算
++, --	自增和自减，可以放在变量的前面也可以放在变量的后面
!, &&,	逻辑非（取反）、逻辑与（and）、逻辑或（or）
<, <=, >, >=	比较符号（小于、小于等于、大于、大于等于）
==, !=, =	比较符号（相等、不相等；对于字符串，= 也可以表示相当于）
<<, >>	向左移位、向右移位
~, , &, ^	按位取反、按位或、按位与、按位异或
=, +=, -=, *=, /=, %=	赋值运算符，例如 a+=1 相当于 a=a+1，a-=1 相当于 a=a-1

但是，Shell 和其它编程语言不同，Shell 不能直接进行算数运算，必须使用数学计算命令，这让初学者感觉很困惑，也让有经验的程序员感觉很奇葩。

下面我们先来看一个反面的例子：

```
[c.biancheng.net]$ echo 2+8
2+8
[c.biancheng.net]$ a=23
[c.biancheng.net]$ b=$a+55
[c.biancheng.net]$ echo $b
23+55
[c.biancheng.net]$ b=90
[c.biancheng.net]$ c=$a+$b
[c.biancheng.net]$ echo $c
23+90
```

从上面的运算结果可以看出，默认情况下，Shell 不会直接进行算术运算，而是把 + 两边的数据（数值或者变量）当做字符串，把 + 当做字符串连接符，最终的结果是把两个字符串拼接在一起形成一个新的字符串。

这是因为，在 Bash Shell 中，如果不特别指明，每一个变量的值都是字符串，无论你给变量赋值时有没有使用引号，值都会以字符串的形式存储。

换句话说，Bash shell 在默认情况下不会区分变量类型，即使你将整数和小数赋值给变量，它们也会被视为字符串，这一点和大部分的编程语言不同。

这一点我们已在《[Shell 变量](#)》中提到，读者可以猛击链接回忆。

数学计算命令

要想让数学计算发挥作用，必须使用数学计算命令，Shell 中常用的数学计算命令如下表所示。

Shell 中常用的六种数学计算方式	
运算操作符/运算命令	说明
(())	用于整数运算，效率很高， 推荐使用 。
let	用于整数运算，和 <code>(())</code> 类似。
\$[]	用于整数运算，不如 <code>(())</code> 灵活。
expr	可用于整数运算，也可以处理字符串。比较麻烦，需要注意各种细节，不推荐使用。
bc	Linux 下的一个计算器程序，可以处理整数和小数。Shell 本身只支持整数运算，想计算小数就得使用 <code>bc</code> 这个外部的计算器。
declare -i	将变量定义为整数，然后再进行数学运算时就不会被当做字符串了。功能有限，仅支持最基本的数学运算（加减乘除和取余），不支持逻辑运算、自增自减等，所以在实际开发中很少使用。

如果大家时间有限，只学习 `(())` 和 `bc` 即可，不用学习其它的了：`(())` 可以用于整数计算，`bc` 可以小数计算。

在接下来的章节中，我们将逐一为大家讲解 Shell 中的各种运算符号及运算命令。

23. Shell (())：对整数进行数学运算

双小括号 (()) 是 Bash Shell 中专门用来进行整数运算的命令，它的效率很高，写法灵活，是企业运维中常用的运算命令。

注意：(()) 只能进行整数运算，不能对小数（浮点数）或者字符串进行运算。后续讲到的 bc 命令可以用于小数运算。

Shell (()) 的用法

双小括号 (()) 的语法格式为：

((表达式))

通俗地讲，就是将数学运算表达式放在((和))之间。

表达式可以只有一个，也可以有多个，多个表达式之间以逗号分隔。对于多个表达式的情况，以最后一个表达式的值作为整个 (()) 命令的执行结果。

可以使用\$获取 (()) 命令的结果，这和使用\$获得变量值是类似的。

表 1：(()) 的用法	
运算操作符/运算命令	说明
((a=10+66)) ((b=a-15)) ((c=a+b))	这种写法可以在计算完成后给变量赋值。以 ((b=a-15)) 为例，即将 a-15 的运算结果赋值给变量 c。 注意，使用变量时不用加\$前缀，(()) 会自动解析变量名。
a=\$((10+66)) b=\$((a-15)) c=\$((a+b))	可以在 (()) 前面加上\$符号获取 (()) 命令的执行结果，也即获取整个表达式的值。以 c=\$((a+b)) 为例，即将 a+b 这个表达式的运算结果赋值给变量 c。 注意，类似 c=((a+b)) 这样的写法是错误的，不加\$就不能取得表达式的结果。
((a>7 && b==c))	(()) 也可以进行逻辑运算，在 if 语句中常会使用逻辑运算。
echo \$((a+10))	需要立即输出表达式的运算结果时，可以在 (()) 前面加\$符号。
((a=3+5, b=a+10))	对多个表达式同时进行计算。

在 (()) 中使用变量无需加上\$前缀，(()) 会自动解析变量名，这使得代码更加简洁，也符合程序员的书写习惯。

Shell (()) 实例演示

【实例 1】利用 (()) 进行简单的数值计算。

```
[c.biancheng.net]$ echo $((1+1))
```

2

```
[c.biancheng.net]$ echo $((6-3))
```

3

```
[c.biancheng.net]$ i=5

[c.biancheng.net]$ ((i=i*2)) #可以简写为 ((i*=2))。

[c.biancheng.net]$ echo $i #使用 echo 输出变量结果时要加 $。

10
```

【实例 2】用 (()) 进行稍微复杂一些的综合算术运算。

```
[c.biancheng.net]$ ((a=1+2**3-4%3))

[c.biancheng.net]$ echo $a

8

[c.biancheng.net]$ b=$((1+2**3-4%3)) #运算后将结果赋值给变量，变量放在了括号的外面。

[c.biancheng.net]$ echo $b

8

[c.biancheng.net]$ echo $((1+2**3-4%3)) #也可以直接将表达式的结果输出，注意不要丢掉 $ 符号。

8

[c.biancheng.net]$ a=$((100*(100+1)/2)) #利用公式计算 1+2+3+...+100 的和。

[c.biancheng.net]$ echo $a

5050

[c.biancheng.net]$ echo $((100*(100+1)/2)) #也可以直接输出表达式的结果。

5050
```

【实例 3】利用 (()) 进行逻辑运算。

```
[c.biancheng.net]$ echo $((3<8)) #3<8 的结果是成立的，因此，输出了 1，1 表示真

1

[c.biancheng.net]$ echo $((8<3)) #8<3 的结果是不成立的，因此，输出了 0，0 表示假。

0

[c.biancheng.net]$ echo $((8==8)) #判断是否相等。

1

[c.biancheng.net]$ if ((8>7&&5==5))

> then

> echo yes

> fi
```

```
yes
```

最后是一个简单的 if 语句的格式，它的意思是，如果 $8 > 7$ 成立，并且 $5 = 5$ 成立，那么输出 yes。显然，这两个条件都是成立的，所以输出了 yes。

【实例 4】利用 $(())$ 进行自增 (++) 和自减 (--) 运算。

```
[c.biancheng.net]$ a=10

[c.biancheng.net]$ echo $((a++)) #如果++在 a 的后面，那么在输出整个表达式时，会输出 a 的值, 因为 a 为 10，所以表达式的值为
10。

10

[c.biancheng.net]$ echo $a #执行上面的表达式后，因为有 a++，因此 a 会自增 1，因此输出 a 的值为 11。

11

[c.biancheng.net]$ a=11

[c.biancheng.net]$ echo $((a--)) #如果--在 a 的后面，那么在输出整个表达式时，会输出 a 的值，因为 a 为 11，所以表达式的值的
为 11。

11

[c.biancheng.net]$ echo $a #执行上面的表达式后，因为有 a--，因此 a 会自动减 1，因此 a 为 10。

10

[c.biancheng.net]$ a=10

[c.biancheng.net]$ echo $((--a)) #如果--在 a 的前面，那么在输出整个表达式时，先进行自增或自减计算，因为 a 为 10，且要自
减，所以表达式的值为 9。

9

[c.biancheng.net]$ echo $a #执行上面的表达式后，a 自减 1, 因此 a 为 9。

9

[c.biancheng.net]$ echo $((++a)) #如果++在 a 的前面，输出整个表达式时，先进行自增或自减计算，因为 a 为 9，且要自增 1，所
以输出 10。

10

[c.biancheng.net]$ echo $a #执行上面的表达式后，a 自增 1, 因此 a 为 10。

10
```

本教程假设读者具备基本的编程能力，相信读者对于前自增（前自减）和后自增（后自减）的区别也非常清楚，这里就不再赘述，只进行简单的说明：

- 执行 `echo $((a++))` 和 `echo $((a--))` 命令输出整个表达式时，输出的值即为 a 的值，表达式执行完毕后，会再对 a 进行 ++、-- 的运算；
- 而执行 `echo $((++a))` 和 `echo $((--a))` 命令输出整个表达式时，会先对 a 进行 ++、-- 的运算，然后再输出表达式的值，即为 a 运算后的值。

【实例 5】利用 `(())` 同时对多个表达式进行计算。

```
[c.biancheng.net]$ ((a=3+5, b=a+10)) #先计算第一个表达式，再计算第二个表达式
```

```
[c.biancheng.net]$ echo $a $b
```

```
8 18
```

```
[c.biancheng.net]$ c=$((4+8, a+b)) #以最后一个表达式的结果作为整个 (( )) 命令的执行结果
```

```
[c.biancheng.net]$ echo $c
```

```
26
```

24. Shell let 命令：对整数进行数学运算

let 命令和双小括号 (()) 的用法是类似的，它们都是用来对整数进行运算，读者已经学习了《[Shell \(\(\)\)](#)》，再学习 let 命令就相当简单了。

注意：和双小括号 (()) 一样，let 命令也只能进行整数运算，不能对小数（浮点数）或者字符串进行运算。

Shell let 命令的语法格式为：

```
let 表达式
```

或者

```
let "表达式"
```

或者

```
let '表达式'
```

它们都等价于 ((表达式))。

当表达式中含有 Shell 特殊字符（例如 | ）时，需要用双引号 " " 或者单引号 ' ' 将表达式包围起来。

和 (()) 类似，let 命令也支持一次性计算多个表达式，并且以最后一个表达式的值作为整个 let 命令的执行结果。但是，对于多个表达式之间的分隔符，let 和 (()) 是有区别的：

- let 命令以空格来分隔多个表达式；
- (()) 以逗号 , 来分隔多个表达式。

另外还要注意，对于类似 let x+y 这样的写法，Shell 虽然计算了 x+y 的值，但却将结果丢弃；若不想这样，可以使用 let sum=x+y 将 x+y 的结果保存在变量 sum 中。

这种情况下 (()) 显然更加灵活，可以使用 \${x+y} 来获取 x+y 的结果。请看下面的例子：

```
[c.biancheng.net]$ a=10 b=20

[c.biancheng.net]$ echo ${a+b}

30

[c.biancheng.net]$ echo let a+b #错误，echo 会把 let a+b 作为一个字符串输出

let a+b
```

Shell let 命令实例演示

【实例 1】给变量 i 加 8：

```
[c.biancheng.net]$ i=2

[c.biancheng.net]$ let i+=8

[c.biancheng.net]$ echo $i
```


10

`let i+=8` 等同于 `((i+=8))` , 但后者效率更高。

【实例 2】`let` 后面可以跟多个表达式。

```
[c.biancheng.net]$ a=10 b=35
```

```
[c.biancheng.net]$ let a+=6 c=a+b #多个表达式以空格为分隔
```

```
[c.biancheng.net]$ echo $a $c
```

```
16 51
```

25. Shell `$[]`：对整数进行数学运算

和 `()`、`let` 命令类似，`$[]` 也只能进行整数运算。

Shell `$[]` 的用法如下：

```
$[表达式]
```

`$[]` 会对表达式进行计算，并取得计算结果。如果表达式中包含了变量，那么你可以加`$`，也可以不加。

Shell `$[]` 举例：

```
[c.biancheng.net]$ echo ${3*5}    #直接输出结算结果

15

[c.biancheng.net]$ echo ${ (3+4)*5}  #使用()

35

[c.biancheng.net]$ n=6

[c.biancheng.net]$ m=${n*2}    #将计算结果赋值给变量

[c.biancheng.net]$ echo ${m+n}

18

[c.biancheng.net]$ echo ${m*$n}  #在变量前边加$也是可以的

72

[c.biancheng.net]$ echo ${4*(m+n)}

72
```

需要注意的是，不能单独使用 `$[]`，必须能够接收 `$[]` 的计算结果。例如，下面的用法是错误的：

```
[c.biancheng.net]$ ${3+4}

bash: 7: 未找到命令...

[c.biancheng.net]$ ${m+3}

bash: 15: 未找到命令...
```

26. Shell expr 命令：对整数进行运算

expr 是 evaluate expressions 的缩写，译为“表达式求值”。Shell expr 是一个功能强大，并且比较复杂的命令，它除了可以实现整数计算，还可以结合一些选项对字符串进行处理，例如计算字符串长度、字符串比较、字符串匹配、字符串提取等。

本节只讲解 expr 在整数计算方面的应用，并不涉及字符串处理，有兴趣的读者请自行研究。

Shell expr 对于整数计算的用法为：

expr 表达式

expr 对表达式的格式有几点特殊的要求：

- 出现在表达式中的运算符、数字、变量和小括号的左右两边至少要有一个空格，否则会报错。
- 有些特殊符号必须用反斜杠\进行转义（屏蔽其特殊含义），比如乘号*和小括号()，如果不用\转义，那么 Shell 会把它们误解为正则表达式中的符号（*对应通配符，()对应分组）。
- 使用变量时要加\$前缀。

【实例 1】expr 整数计算简单举例：

```
[c.biancheng.net]$ expr 2 +3 #错误：加号和 3 之前没有空格

expr: 语法错误

[c.biancheng.net]$ expr 2 + 3 #这样才是正确的

5

[c.biancheng.net]$ expr 4 * 5 #错误：乘号没有转义

expr: 语法错误

[c.biancheng.net]$ expr 4 \* 5 #使用 \ 转义后才是正确的

20

[c.biancheng.net]$ expr ( 2 + 3 ) \* 4 #小括号也需要转义

bash: 未预期的符号 `2' 附近有语法错误

[c.biancheng.net]$ expr \( 2 + 3 \) \* 4 #使用 \ 转义后才是正确的

20

[c.biancheng.net]$ n=3

[c.biancheng.net]$ expr n + 2 #使用变量时要加 $

expr: 非整数参数

[c.biancheng.net]$ expr $n + 2 #加上 $ 才是正确的

5

[c.biancheng.net]$ m=7

[c.biancheng.net]$ expr $m \* \( $n + 5 \)
```

以上是直接使用 `expr` 命令，计算结果会直接输出，如果你希望将计算结果赋值给变量，那么需要将整个表达式用反引号```（位于 Tab 键的上方）包围起来，请看下面的例子。

【实例 2】将 `expr` 的计算结果赋值给变量：

```
[c.biancheng.net]$ m=5  
  
[c.biancheng.net]$ n=`expr $m + 10`  
  
[c.biancheng.net]$ echo $n  
  
15
```

你看，使用 `expr` 进行数学计算是多么的麻烦呀，需要注意各种细节，我奉劝大家还是省省心，老老实实用 `()`、`let` 或者 `[]` 吧。

27. Linux bc 命令：一款数学计算器

Bash Shell 内置了对整数运算的支持，但是并不支持浮点运算，而 Linux bc 命令可以很方便的进行浮点运算，当然整数运算也不再话下。

bc 甚至可以称得上是一种编程语言了，它支持变量、数组、输入输出、分支结构、循环结构、函数等基本的编程元素，所以 Linux 手册中是这样来描述 bc 的：

An arbitrary precision calculator language

翻译过来就是“一个任意精度的计算器语言”。

在终端输入 `bc` 命令，然后回车即可进入 bc 进行交互式的数学计算。在 Shell 编程中，我们也可以通过管道和输入重定向来使用 bc。

本节我们先学习如何在交互式环境下使用 bc，然后再学习如何在 Shell 编程中使用 bc，这样就易如反掌了。

从终端进入 bc

在终端输入 bc 命令，然后回车，就可以进入 bc，请看下图：



bc 命令还有一些选项，可能会用到，请看下表。

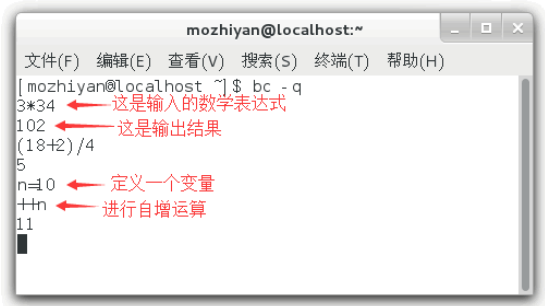
选项	说明
-h --help	帮助信息
-v --version	显示命令版本信息
-l --mathlib	使用标准数学库
-i --interactive	强制交互
-w --warn	显示 POSIX 的警告信息
-s --standard	使用 POSIX 标准来处理
-q --quiet	不显示欢迎信息

例如你不想输入 bc 命令后显示一堆没用的信息，那么可以输入 `bc -q`：



在交互式环境下使用 bc

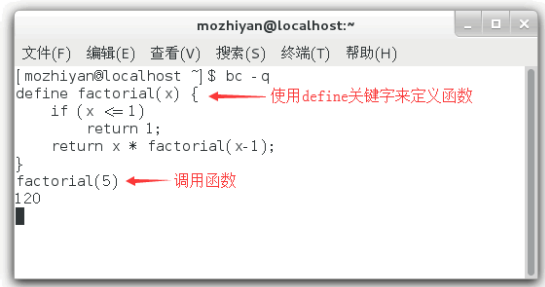
使用 bc 进行数学计算是非常容易的，像平常一样输入数学表达式，然后按下回车键就可以看到结果，请看下图。



```
moshiyan@localhost:~  
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)  
[moshiyan@localhost ~]$ bc -q  
3*34 ← 这是输入的数学表达式  
102 ← 这是输出结果  
(18+2)/4  
5  
n=10 ← 定义一个变量  
++n ← 进行自增运算  
11
```

值得一提的是，我们定义了一个变量 n，然后在计算中也使用了 n，可见 bc 是支持变量的。

除了变量，bc 还支持函数、循环结构、分支结构等常见的编程元素，它们和其它编程语言的语法类似。下面我们定义一个求阶乘的函数：



```
moshiyan@localhost:~  
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)  
[moshiyan@localhost ~]$ bc -q  
define factorial(x) { ← 使用define关键字来定义函数  
    if (x <= 1)  
        return 1;  
    return x * factorial(x-1);  
}  
factorial(5) ← 调用函数  
120
```

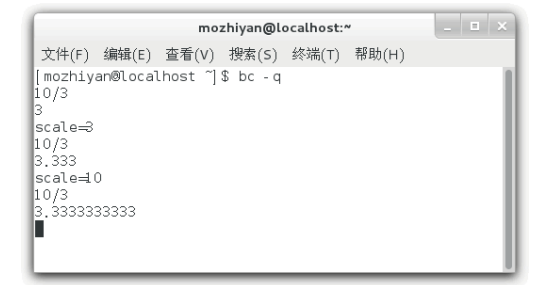
其实我们很少使用这么复杂的功能，大部分情况下还是把 bc 作为普通的数学计算器，求一下表达式的值而已，所以大家不必深究，了解一下即可。

内置变量

bc 有四个内置变量，我们在计算时会经常用到，如下表所示：

变量名	作用
scale	指定精度，也即小数点后的位数；默认为 0，也即不使用小数部分。
ibase	指定输入的数字的进制，默认为十进制。
obase	指定输出的数字的进制，默认为十进制。
last 或者 .	表示最近打印的数字

【实例 1】scale 变量用法举例：



```
moshiyan@localhost:~  
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)  
[moshiyan@localhost ~]$ bc -q  
10/3  
3  
scale=3  
10/3  
3.333  
scale=10  
10/3  
3.3333333333
```

刚开始的时候，10/3 的值为 3，不带小数部分，就是因为 scale 变量的默认值为 0；后边给 scale 指定了一个大于 0 的值，就能看到小数部分了。

【实例 2】ibase 和 obase 变量用法举例：



```
mozhiyan@localhost:~  
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)  
[mozhiyan@localhost ~]$ bc -q  
23*2  
46  
obase=16  
23*2  
2E  
obase=10  
ibase=16  
10*10  
256
```

注意：obase 要尽量放在 ibase 前面，因为 ibase 设置后，后面的数字都是以 ibase 的进制来换算的。

内置函数

除了内置变量，bc 还有一些内置函数，如下表所示：

函数名	作用
s(x)	计算 x 的正弦值，x 是弧度值。
c(x)	计算 x 的余弦值，x 是弧度值。
a(x)	计算 x 的反正切值，返回弧度值。
l(x)	计算 x 的自然对数。
e(x)	求 e 的 x 次方。
j(n, x)	贝塞尔函数，计算从 n 到 x 的阶数。

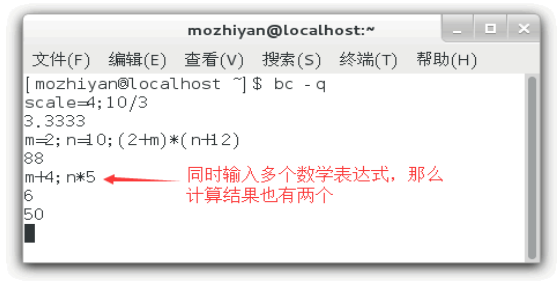
要想使用这些数学函数，在输入 bc 命令时需要使用 -l 选项，表示启用数学库。请看下面的例子：



```
mozhiyan@localhost:~  
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)  
[mozhiyan@localhost ~]$ bc -q -l  
x=5  
s(x)  
-.95892427466313846889  
e(x)  
148.41315910257660342111
```

在一行中使用多个表达式

在前边的例子中，我们基本上是一行一个表达式，这样看起来更加舒服；如果你愿意，也可以将多个表达式放在一行，只要用分号隔开就行。请看下面的例子：



```
mozhiyan@localhost:~  
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)  
[mozhiyan@localhost ~]$ bc -q  
scale=4; 10/3  
3.3333  
m=2; n=10; (2+m)*(n+2)  
88  
m+4; n*5  
6  
50
```

同时输入多个数学表达式，那么计算结果也有两个

在 Shell 中使用 bc 计算器

在 Shell 脚本中，我们可以借助管道或者输入重定向来使用 bc 计算器。

- 管道是 Linux 进程间的一种通信机制，它可以将前一个命令（进程）的输出作为下一个命令（进程）的输入，两个命令之间使用竖线 | 分隔。
- 通常情况下，一个命令从终端获得用户输入的内容，如果让它从其他地方（比如文件）获得输入，那么就需要重定向。

此处我们并不打算展开讲解管道和重定向，不了解的小伙伴请自行百度。

借助管道使用 bc 计算器

如果读者希望直接输出 bc 的计算结果，那么可以使用下面的形式：

```
echo "expression" | bc
```

expression 就是希望计算的数学表达式，它必须符合 bc 的语法，上面我们已经进行了介绍。在 expression 中，还可以使用 Shell 脚本中的变量。

使用下面的形式可以将 bc 的计算结果赋值给 Shell 变量：

```
variable=$(echo "expression" | bc)
```

variable 就是变量名。

【实例 1】最简单的形式：

```
[c.biancheng.net]$ echo "3*8"|bc
24

[c.biancheng.net]$ ret=$(echo "4+9"|bc)

[c.biancheng.net]$ echo $ret
13
```

【实例 2】使用 bc 中的变量：

```
[c.biancheng.net]$ echo "scale=4;3*8/7"|bc
3.4285

[c.biancheng.net]$ echo "scale=4;3*8/7;last*5"|bc
3.4285

17.1425
```

【实例 3】使用 Shell 脚本中的变量：

```
[c.biancheng.net]$ x=4

[c.biancheng.net]$ echo "scale=5;n=$x+2;e(n)"|bc -l
```



```
403.42879
```

在第二条命令中，`$x` 表示使用第一条 Shell 命令中定义的变量，`n` 是在 `bc` 中定义的新变量，它和 Shell 脚本是没关系的。

【实例 4】进制转换：

#十进制转十六进制

```
[mozhayan@localhost ~]$ m=31
```

```
[mozhayan@localhost ~]$ n=$(echo "obase=16;$m"|bc)
```

```
[mozhayan@localhost ~]$ echo $n
```

```
1F
```

#十六进制转十进制

```
[mozhayan@localhost ~]$ m=1E
```

```
[mozhayan@localhost ~]$ n=$(echo "obase=10;ibase=16;$m"|bc)
```

```
[mozhayan@localhost ~]$ echo $n
```

```
30
```

借助输入重定向使用 `bc` 计算器

可以使用下面的形式将 `bc` 的计算结果赋值给 Shell 变量：

```
variable=$(bc << EOF
expressions
EOF
)
```

其中，`variable` 是 Shell 变量名，`express` 是要计算的数学表达式（可以换行，和进入 `bc` 以后的书写形式一样），`EOF` 是数学表达式的开始和结束标识（你也可以换成其它的名字，比如 `aaa`、`bbb` 等）。

请看下面的例子：

```
[c.biancheng.net]$ m=1E
```

```
[c.biancheng.net]$ n=$(bc << EOF
```

```
> obase=10;
```

```
> ibase=16;
```

```
> print $m
```

```
> EOF
```

```
> )
```

```
[c.biancheng.net]$ echo $n
```

```
30
```

如果你有大量的数学计算，那么使用输入重定向就比较方便，因为数学表达式可以换行，写起来更加清晰明了。

28. Shell declare -i : 将变量声明为整数类型

在《[Shell declare 命令](#)》一节中，我们已经讲解了 declare 命令的各种选项，为了让 Shell 进行整数运算，本节我们重点讲解-i 选项。

默认情况下，Shell 中每一个变量的值都是字符串（不了解的读者请猛击《[Shell 变量](#)》），即使你给变量赋值一个数字，它其实也是字符串，所以在进行数学计算时会出错。

使用 declare 命令的 `-i` 选项可以将一个变量声明为整数类型，这样在进行数学计算时就不会作为字符串处理了，请看下面的例子：

```
1.  #!/bin/bash
2.
3.  declare -i m n ret
4.  m=10
5.  n=30
6.
7.  ret=$m+$n
8.  echo $ret
9.
10. ret=$n/$m
11. echo $ret
```

运行结果：

40

3

除了将 m、n 定义为整数，还必须将 ret 定义为整数，如果不这样做，在执行 `ret=$m+$n` 和 `ret=$n/$m` 时，Shell 依然会将 m、n 视为字符串。

此外，你也不能写类似 `echo $m+$n` 这样的语句，这种情况下 m、n 也会被视为字符串。

总之，除了将参与运算的变量定义为整数，还得将承载结果的变量定义为整数，而且只能用整数类型的变量来承载运算结果，不能直接使用 echo 输出。

和 `()`、`let`、`$[]` 不同，`declare -i` 的功能非常有限，仅支持最基本的数学运算（加减乘除和取余），不支持逻辑运算（比较运算、与运算、或运算、非运算），所以在实际开发中很少使用。

29. Shell if else 语句（详解版）

和其它编程语言类似，Shell 也支持选择结构，并且有两种形式，分别是 if else 语句和 case in 语句。本节我们先介绍 if else 语句，case in 语句将会在《[Shell case in](#)》中介绍。

如果你已经熟悉了 C 语言、[Java](#)、[JavaScript](#) 等其它编程语言，那么你可能会觉得 Shell 中的 if else 语句有点奇怪。

if 语句

最简单的用法就是只使用 if 语句，它的语法格式为：

```
if condition
then
    statement(s)
fi
```

`condition` 是判断条件，如果 condition 成立（返回“真”），那么 then 后边的语句将会被执行；如果 condition 不成立（返回“假”），那么不会执行任何语句。

从本质上讲，if 检测的是命令的退出状态，我们将在下节《[Shell 退出状态](#)》中深入讲解。

注意，最后必须以 `fi` 来闭合，fi 就是 if 倒过来拼写。也正是有了 fi 来结尾，所以即使有多条语句也不需要 `{ }` 包围起来。

如果你喜欢，也可以将 then 和 if 写在一行：

```
if condition; then
    statement(s)
fi
```

请注意 condition 后边的分号，当 if 和 then 位于同一行的时候，这个分号是必须的，否则会有语法错误。

实例 1

下面的例子使用 if 语句来比较两个数字的大小：

```
1.  #!/bin/bash
2.
3.  read a
4.  read b
5.
6.  if (( $a == $b ))
7.  then
8.      echo "a 和 b 相等"
9.  fi
```

运行结果：

84✓

84✓

a 和 b 相等

在《[Shell \(0\)](#)》一节中我们讲到，`(())` 是一种数学计算命令，它除了可以进行最基本的加减乘除运算，还可以进行大于、小于、等于等关系运算，以及与、或、非逻辑运算。当 a 和 b 相等时，`(($a == $b))` 判断条件成立，进入 if，执行 then 后边的 echo 语句。

实例 2

在判断条件中也可以使用逻辑运算符，例如：

```
1.  #!/bin/bash
2.
3.  read age
4.  read iq
5.
6.  if (( $age > 18 && $iq < 60 ))
7.  then
8.      echo "你都成年了，智商怎么还不及格！"
9.      echo "来 C 语言中文网（http://c.biancheng.net/）学习编程吧，能迅速提高你的智商。"
10. fi
```

运行结果：

20✓

56✓

你都成年了，智商怎么还不及格！

来 C 语言中文网（<http://c.biancheng.net/>）学习编程吧，能迅速提高你的智商。

`&&`就是逻辑“与”运算符，只有当`&&`两侧的判断条件都为“真”时，整个判断条件才为“真”。

熟悉其他编程语言的读者请注意，即使 then 后边有多条语句，也不需要`{ }`包围起来，因为有 fi 收尾呢。

if else 语句

如果有两个分支，就可以使用 if else 语句，它的格式为：

```
if condition
then
    statement1
else
    statement2
fi
```

如果 condition 成立，那么 then 后边的 statement1 语句将会被执行；否则，执行 else 后边的 statement2 语句。

举个例子：

```
1.  #!/bin/bash
2.
3.  read a
4.  read b
5.
6.  if (( $a == $b ))
```

```
7.  then
8.      echo "a 和 b 相等"
9.  else
10.     echo "a 和 b 不相等，输入错误"
11. fi
```

运行结果：

10✓

20✓

a 和 b 不相等，输入错误

从运行结果可以看出，a 和 b 不相等，判断条件不成立，所以执行了 else 后边的语句。

if elif else 语句

Shell 支持任意数目的分支，当分支比较多时，可以使用 if elif else 结构，它的格式为：

```
if condition1
then
    statement1
elif condition2
then
    statement2
elif condition3
then
    statement3
.....
else
    statementn
fi
```

注意，if 和 elif 后边都得跟着 then。

整条语句的执行逻辑为：

- 如果 condition1 成立，那么就执行 if 后边的 statement1；如果 condition1 不成立，那么继续执行 elif，判断 condition2。
- 如果 condition2 成立，那么就执行 statement2；如果 condition2 不成立，那么继续执行后边的 elif，判断 condition3。
- 如果 condition3 成立，那么就执行 statement3；如果 condition3 不成立，那么继续执行后边的 elif。
- 如果所有的 if 和 elif 判断都不成立，就进入最后的 else，执行 statementn。

举个例子，输入年龄，输出对应的人生阶段：

```
1.  #!/bin/bash
2.
3.  read age
4.
5.  if (( $age <= 2 )); then
```

```
6.     echo "婴儿"
7.     elif (( $age >= 3 && $age <= 8 )); then
8.         echo "幼儿"
9.     elif (( $age >= 9 && $age <= 17 )); then
10.        echo "少年"
11.    elif (( $age >= 18 && $age <= 25 )); then
12.        echo "成年"
13.    elif (( $age >= 26 && $age <= 40 )); then
14.        echo "青年"
15.    elif (( $age >= 41 && $age <= 60 )); then
16.        echo "中年"
17.    else
18.        echo "老年"
19.    fi
```

运行结果 1 :

19
成年

运行结果 2 :

100
老年

再举一个例子，输入一个整数，输出该整数对应的星期几的英文表示：

```
1.  #!/bin/bash
2.
3.  printf "Input integer number: "
4.  read num
5.
6.  if ((num==1)); then
7.      echo "Monday"
8.  elif ((num==2)); then
9.      echo "Tuesday"
10. elif ((num==3)); then
11.     echo "Wednesday"
12. elif ((num==4)); then
13.     echo "Thursday"
14. elif ((num==5)); then
15.     echo "Friday"
```

```
16.  elif ((num==6)); then
17.      echo "Saturday"
18.  elif ((num==7)); then
19.      echo "Sunday"
20.  else
21.      echo "error"
22.  fi
```

运行结果 1：

Input integer number: 4

Thursday

运行结果 2：

Input integer number: 9

error

30. Shell 退出状态

每一条 Shell 命令，不管是 Bash 内置命令（例如 cd、echo），还是外部的 Linux 命令（例如 ls、awk），还是自定义的 Shell 函数，当它退出（运行结束）时，都会返回一个比较小的整数值给调用（使用）它的程序，这就是命令的**退出状态 (exit statu)**。

很多 Linux 命令其实就是一个 C 语言程序，熟悉 C 语言的读者都知道，main() 函数的最后都有一个 `return 0`，如果程序想在中间退出，还可以使用 `exit 0`，这其实就是 C 语言程序的退出状态。当有其它程序调用这个程序时，就可以捕获这个退出状态。

if 语句的判断条件，从本质上讲，判断的就是命令的退出状态。

按照惯例来说，退出状态为 0 表示“成功”；也就是说，程序执行完成并且没有遇到任何问题。除 0 以外的其它任何退出状态都为“失败”。

之所以说这是“惯例”而非“规定”，是因为也会有例外，比如 diff 命令用来比较两个文件的不同，对于“没有差别”的文件返回 0，对于“找到差别”的文件返回 1，对无效文件名返回 2。

有编程经验的读者请注意，Shell 的这个部分与你所熟悉的其它编程语言正好相反：在 C 语言、[C++](#)、[Java](#)、[Python](#) 中，0 表示“假”，其它值表示“真”。

在 Shell 中，有多种方式取得命令的退出状态，其中 `$?` 是最常见的一种。上节《[Shell if else](#)》中使用了 `(0)` 进行数学计算，我们不妨来看一下它的退出状态。请看下面的代码：

```
1.  #!/bin/bash
2.
3.  read a
4.  read b
5.
6.  (( $a == $b ));
7.
8.  echo "退出状态: "$?
```

运行结果 1：
26
26
退出状态：0

运行结果 2：
17
39
退出状态：1

退出状态和逻辑运算符的组合

Shell if 语句的一个神奇之处是允许我们使用逻辑运算符将多个退出状态组合起来，这样就可以一次判断多个条件了。

Shell 逻辑运算符		
运算符	使用格式	说明
&&	expression1 && expression2	逻辑与运算符，当 expression1 和 expression2 同时成立时，整个表达式才成立。 如果检测到 expression1 的退出状态为 0，就不会再检测 expression2 了，因为不管 expression2 的退出状态是什么，整个表达式必然都是不成立的，检测了也是多此一举。
	expression1 expression2	逻辑或运算符，expression1 和 expression2 两个表达式中只要有一个成立，整个表达式就成立。

		如果检测到 expression1 的退出状态为 1，就不会再检测 expression2 了，因为不管 expression2 的退出状态是什么，整个表达式必然都是成立的，检测了也是多此一举。
!	!expression	逻辑非运算符，相当于“取反”的效果。如果 expression 成立，那么整个表达式就不成立；如果 expression 不成立，那么整个表达式就成立。

【实例】将用户输入的 URL 写入到文件中。

```
1.  #!/bin/bash
2.
3.  read filename
4.  read url
5.
6.  if test -w $filename && test -n $url
7.  then
8.      echo $url > $filename
9.      echo "写入成功"
10. else
11.     echo "写入失败"
12. fi
```

在 Shell 脚本文件所在的目录新建一个文本文件并命名为 urls.txt，然后运行 Shell 脚本，运行结果为：

urls.txt ✓

http://c.biancheng.net/shell/ ✓

写入成功

test 是 Shell 内置命令，可以对文件或者字符串进行检测，其中，`-w` 选项用来检测文件是否存在并且可写，`-n` 选项用来检测字符串是否非空。下节《[Shell test](#)》中将会详细讲解。

`>` 表示重定向，默认情况下，echo 向控制台输出，这里我们将输出结果重定向到文件。

31. Shell test 命令 (Shell []) 详解，附带所有选项及说明

test 是 Shell 内置命令，用来检测某个条件是否成立。test 通常和 if 语句一起使用，并且大部分 if 语句都依赖 test。

test 命令有很多选项，可以进行数值、字符串和文件三个方面的检测。

Shell test 命令的用法为：

```
test expression
```

当 test 判断 expression 成立时，退出状态为 0，否则为非 0 值。

test 命令也可以简写为 `[]`，它的用法为：

```
[ expression ]
```

注意 `[]` 和 `expression` 之间的空格，这两个空格是必须的，否则会导致语法错误。`[]` 的写法更加简洁，比 test 使用频率高。

test 和 `[]` 是等价的，后续我们会交替使用 test 和 `[]`，以让读者尽快熟悉。

在《[Shell if else](#)》中，我们使用 `(())` 进行数值比较，这节我们就来看一下如何使用 test 命令进行数值比较。

```
1.  #!/bin/bash
2.
3.  read age
4.
5.  if test $age -le 2; then
6.      echo "婴儿"
7.  elif test $age -ge 3 && test $age -le 8; then
8.      echo "幼儿"
9.  elif [ $age -ge 9 ] && [ $age -le 17 ]; then
10.     echo "少年"
11. elif [ $age -ge 18 ] && [ $age -le 25 ]; then
12.     echo "成年"
13. elif test $age -ge 26 && test $age -le 40; then
14.     echo "青年"
15. elif test $age -ge 41 && [ $age -le 60 ]; then
16.     echo "中年"
17. else
18.     echo "老年"
19. fi
```

其中，`-le` 选项表示小于等于，`-ge` 选项表示大于等于，`&&` 是逻辑与运算符。

学习 test 命令，重点是学习它的各种选项，下面我们就逐一讲解。

1) 与文件检测相关的 test 选项

表 1：test 文件检测相关选项列表	
文件类型判断	
选 项	作 用
-b filename	判断文件是否存在，并且是否为块设备文件。
-c filename	判断文件是否存在，并且是否为字符设备文件。
-d filename	判断文件是否存在，并且是否为目录文件。
-e filename	判断文件是否存在。
-f filename	判断文件是否存在，并且是否为普通文件。
-L filename	判断文件是否存在，并且是否为符号链接文件。
-p filename	判断文件是否存在，并且是否为管道文件。
-s filename	判断文件是否存在，并且是否为非空。
-S filename	判断该文件是否存在，并且是否为套接字文件。
文件权限判断	
选 项	作 用
-r filename	判断文件是否存在，并且是否拥有读权限。
-w filename	判断文件是否存在，并且是否拥有写权限。
-x filename	判断文件是否存在，并且是否拥有执行权限。
-u filename	判断文件是否存在，并且是否拥有 SUID 权限。
-g filename	判断文件是否存在，并且是否拥有 SGID 权限。
-k filename	判断该文件是否存在，并且是否拥有 SBIT 权限。
文件比较	
选 项	作 用
filename1 -nt filename2	判断 filename1 的修改时间是否比 filename2 的新。
filename1 -ot filename2	判断 filename1 的修改时间是否比 filename2 的旧。
filename1 -ef filename2	判断 filename1 是否和 filename2 的 inode 号一致，可以理解为两个文件是否为同一个文件。这个判断用于判断硬链接是很好的方法

Shell test 文件检测举例：

```
1.  #!/bin/bash
2.
3.  read filename
4.  read url
5.
6.  if test -w $filename && test -n $url
```

```
7.  then
8.      echo $url > $filename
9.      echo "写入成功"
10. else
11.      echo "写入失败"
12. fi
```

在 Shell 脚本文件所在的目录新建一个文本文件并命名为 urls.txt，然后运行 Shell 脚本，运行结果为：

```
urls.txt ✓
http://c.biancheng.net/shell/ ✓
写入成功
```

2) 与数值比较相关的 test 选项

表 2：test 数值比较相关选项列表	
选 项	作 用
num1 -eq num2	判断 num1 是否和 num2 相等。
num1 -ne num2	判断 num1 是否和 num2 不相等。
num1 -gt num2	判断 num1 是否大于 num2。
num1 -lt num2	判断 num1 是否小于 num2。
num1 -ge num2	判断 num1 是否大于等于 num2。
num1 -le num2	判断 num1 是否小于等于 num2。

注意，test 只能用来比较整数，小数相关的比较还得依赖 [bc 命令](#)。

Shell test 数值比较举例：

```
1.  #!/bin/bash
2.
3.  read a b
4.
5.  if test $a -eq $b
6.  then
7.      echo "两个数相等"
8.  else
9.      echo "两个数不相等"
10. fi
```

运行结果 1：

```
10 10
两个数相等
```

运行结果 2：
10 20
两个数不相等

3) 与字符串判断相关的 test 选项

表 3：test 字符串判断相关选项列表	
选项	作用
-z str	判断字符串 str 是否为空。
-n str	判断字符串 str 是否为非空。
str1 = str2 str1 == str2	<code>=</code> 和 <code>==</code> 是等价的，都用来判断 str1 是否和 str2 相等。
str1 != str2	判断 str1 是否和 str2 不相等。
str1 \> str2	判断 str1 是否大于 str2。 <code>\></code> 是 <code>></code> 的转义字符，这样写是为了防止 <code>></code> 被误认为成重定向运算符。
str1 \< str2	判断 str1 是否小于 str2。同样， <code>\<</code> 也是转义字符。

有 C 语言、[C++](#)、[Python](#)、[Java](#) 等编程经验的读者请注意，`==`、`>`、`<` 在大部分编程语言中都用来比较数字，而在 Shell 中，它们只能用来比较字符串，不能比较数字，这是非常奇葩的，大家要习惯。

其次，不管是比较数字还是字符串，Shell 都不支持 `>=` 和 `<=` 运算符，切记。

Shell test 字符串比较举例：

```
1.  #!/bin/bash
2.
3.  read str1
4.  read str2
5.
6.  #检测字符串是否为空
7.  if [ -z "$str1" ] || [ -z "$str2" ]
8.  then
9.      echo "字符串不能为空"
10.     exit 0
11. fi
12.
13. #比较字符串
14. if [ $str1 = $str2 ]
15. then
16.     echo "两个字符串相等"
```

```
17.  else
18.      echo "两个字符串不相等"
19.  fi
```

运行结果：

```
http://c.biancheng.net/
http://c.biancheng.net/shell/
两个字符串不相等
```

细心的读者可能已经注意到，变量 \$str1 和 \$str2 都被双引号包围起来，这样做是为了防止 \$str1 或者 \$str2 是空字符串时出现错误，本文的后续部分将为你分析具体原因。

4) 与逻辑运算相关的 test 选项

表 4：test 逻辑运算相关选项列表

选项	作用
expression1 -a expression2	逻辑与，表达式 expression1 和 expression2 都成立，最终的结果才是成立的。
expression1 -o expression2	逻辑或，表达式 expression1 和 expression2 有一个成立，最终的结果就成立。
!expression	逻辑非，对 expression 进行取反。

改写上面的代码，使用逻辑运算选项：

```
1.  #!/bin/bash
2.
3.  read str1
4.  read str2
5.
6.  #检测字符串是否为空
7.  if [ -z "$str1" -o -z "$str2" ] #使用 -o 选项取代之前的 ||
8.  then
9.      echo "字符串不能为空"
10.     exit 0
11. fi
12.
13. #比较字符串
14. if [ $str1 = $str2 ]
15. then
16.     echo "两个字符串相等"
17. else
```

```
18.      echo "两个字符串不相等"
19.  fi
```

前面的代码我们使用两个 `if` 命令，并使用 `||` 运算符将它们连接起来，这里我们改成 `-o` 选项，只使用一个 `if` 命令就可以了。

在 test 中使用变量建议用双引号包围起来

`test` 和 `[]` 都是命令，一个命令本质上对应一个程序或者一个函数。即使是一个程序，它也有入口函数，例如 C 语言程序的入口函数是 `main()`，运行 C 语言程序就从 `main()` 函数开始，所以也可以将一个程序等效为一个函数，这样我们就不用再区分函数和程序了，直接将一个命令和一个函数对应起来即可。

有了以上认知，就很容易看透命令的本质了：使用一个命令其实就是调用一个函数，命令后面附带的选项和参数最终都会作为实参传递给函数。

假设 `test` 命令对应的函数是 `func()`，使用 `test -z $str1` 命令时，会先将变量 `$str1` 替换成字符串：

- 如果 `$str1` 是一个正常的字符串，比如 `abc123`，那么替换后的效果就是 `test -z abc123`，调用 `func()` 函数的形式就是 `func("-z abc123")`。`test` 命令后面附带的所有选项和参数会被看成一个整体，并作为实参传递进函数。
- 如果 `$str1` 是一个空字符串，那么替换后的效果就是 `test -z`，调用 `func()` 函数的形式就是 `func("-z ")`，这就比较奇怪了，因为 `-z` 选项没有和参数成对出现，`func()` 在分析时就会出错。

如果我们给 `$str1` 变量加上双引号，当 `$str1` 是空字符串时，`test -z "$str1"` 就会被替换为 `test -z ""`，调用 `func()` 函数的形式就是 `func("-z \"")`，很显然，`-z` 选项后面跟的是一个空字符串（`\` 表示转义字符），这样 `func()` 在分析时就不会出错了。

所以，当你在 `test` 命令中使用变量时，我强烈建议将变量用双引号 `"` 包围起来，这样能避免变量为空值时导致的很多奇葩问题。

总结

`test` 命令比较奇葩，`>`、`<`、`==` 只能用来比较字符串，不能用来比较数字，比较数字需要使用 `-eq`、`-gt` 等选项；不管是比较字符串还是数字，`test` 都不支持 `>=` 和 `<=`。有经验的程序员需要慢慢习惯 `test` 命令的这些奇葩用法。

对于整型数字的比较，我建议大家使用 `()`，这在《[Shell if else](#)》中已经进行了演示。`()` 支持各种运算符，写法也符合数学规则，用起来更加方便，何乐而不为呢？

几乎完全兼容 `test`，并且比 `test` 更加强大，比 `test` 更加灵活的是 `[]`；`[]` 不是命令，而是 Shell 关键字，下节《[Shell \[\]](#)》我们将会讲解。

32. Shell [] 详解：检测某个条件是否成立

`[[]]` 是 Shell 内置关键字，它和 [test 命令](#) 类似，也用来检测某个条件是否成立。

`test` 能做到的，`[[]]` 也能做到，而且 `[[]]` 做的更好；`test` 做不到的，`[[]]` 还能做到。可以认为 `[[]]` 是 `test` 的升级版，对细节进行了优化，并且扩展了一些功能。

`[[]]` 的用法为：

```
[[ expression ]]
```

当 `[[]]` 判断 `expression` 成立时，退出状态为 0，否则为非 0 值。注意 `[[]]` 和 `expression` 之间的空格，这两个空格是必须的，否则会导致语法错误。

[[]] 不需要注意某些细枝末节

`[[]]` 是 Shell 内置关键字，不是命令，在使用时没有给函数传递参数的过程，所以 `test` 命令的某些注意事项在 `[[]]` 中就不存在了，具体包括：

- 不需要把变量名用双引号 `"` 包围起来，即使变量是空值，也不会出错。
- 不需要、也不能对 `>`、`<` 进行转义，转义后会出错。

请看下面的演示代码：

```
1.  #!/bin/bash
2.
3.  read str1
4.  read str2
5.
6.  if [[ -z $str1 ]] || [[ -z $str2 ]] #不需要对变量名加双引号
7.  then
8.      echo "字符串不能为空"
9.  elif [[ $str1 < $str2 ]] #不需要也不能对 < 进行转义
10. then
11.     echo "str1 < str2"
12. else
13.     echo "str1 >= str2"
14. fi
```

运行结果：

```
http://c.biancheng.net/shell/
http://data.biancheng.net/
str1 < str2
```

[[]] 支持逻辑运算符

对多个表达式进行逻辑运算时，可以使用逻辑运算符将多个 `test` 命令连接起来，例如：

```
[ -z "$str1" ] || [ -z "$str2" ]
```

你也可以借助选项把多个表达式写在一个 `test` 命令中，例如：

```
[ -z "$str1" -o -z "$str2" ]
```

但是，这两种写法都有点“别扭”，完美的写法是在一个命令中使用逻辑运算符将多个表达式连接起来。我们的这个愿望在 `[[]]` 中实现了，`[[]]` 支持 `&&`、`||` 和 `!` 三种逻辑运算符。

使用 `[[]]` 对上面的语句进行改进：

```
[[ -z $str1 || -z $str2 ]]
```

这种写法就比较简洁漂亮了。

注意，`[[]]` 剔除了 `test` 命令的 `-o` 和 `-a` 选项，你只能使用 `||` 和 `&&`。这意味着，你不能写成下面的形式：

```
[[ -z $str1 -o -z $str2 ]]
```

当然，使用逻辑运算符将多个 `[[]]` 连接起来依然是可以的，因为这是 Shell 本身提供的功能，跟 `[[]]` 或者 `test` 没有关系，如下所示：

```
[[ -z $str1 ]] || [[ -z $str2 ]]
```

该表总结了各种写法的对错

test 或 []		[[]]	
[-z "\$str1"] [-z "\$str2"]	✓	[[-z \$str1]] [[-z \$str2]]	✓
[-z "\$str1" -o -z "\$str2"]	✓	[[-z \$str1 -o -z \$str2]]	✗
[-z \$str1 -z \$str2]	✗	[[-z \$str1 -z \$str2]]	✓

[[]] 支持正则表达式

在 Shell `[[]]` 中，可以使用 `=~` 来检测字符串是否符合某个正则表达式，它的用法为：

```
[[ str =~ regex ]]
```

`str` 表示字符串，`regex` 表示正则表达式。

下面的代码检测一个字符串是否是手机号：

```
1. #!/bin/bash
2.
3. read tel
4.
```

```
5.  if [[ $tel =~ ^1[0-9]{10}$ ]]
6.  then
7.      echo "你输入的是手机号码"
8.  else
9.      echo "你输入的不是手机号码"
10. fi
```

运行结果 1：
13203451100
你输入的是手机号码

运行结果 2：
132034511009
你输入的不是手机号码

对 `^1[0-9]{10}$` 的说明：

- `^` 匹配字符串的开头（一个位置）；
- `[0-9]{10}` 匹配连续的十个数字；
- `$` 匹配字符串的末尾（一个位置）。

本文并不打算讲解正则表达式的语法，不了解的读者请猛击《[正则表达式 30 分钟入门教程](#)》。

总结

有了 `[[]]`，你还有什么理由使用 `test` 或者 `[]`，`[[]]` 完全可以替代之，而且更加方便，更加强大。

但是 `[[]]` 对数字的比较仍然不友好，所以我建议，以后大家使用 `if` 判断条件时，用 `(())` 来处理整型数字，用 `[[]]` 来处理字符串或者文件。

33. Shell case in 语句详解

和其它编程语言类似，Shell 也支持两种分支结构（选择结构），分别是 if else 语句和 case in 语句。在《[Shell if else](#)》一节中我们讲解了 if else 语句的用法，这节我们就来讲解 case in 语句。

当分支较多，并且判断条件比较简单时，使用 case in 语句就比较方便了。

《[Shell if else](#)》一节的最后给出了一个例子，就是输入一个整数，输出该整数对应的星期几的英文表示，这节我们就用 case in 语句来重写代码，如下所示。

```
1.  #!/bin/bash
2.
3.  printf "Input integer number: "
4.  read num
5.
6.  case $num in
7.      1)
8.          echo "Monday"
9.          ;;
10.     2)
11.         echo "Tuesday"
12.         ;;
13.     3)
14.         echo "Wednesday"
15.         ;;
16.     4)
17.         echo "Thursday"
18.         ;;
19.     5)
20.         echo "Friday"
21.         ;;
22.     6)
23.         echo "Saturday"
24.         ;;
25.     7)
26.         echo "Sunday"
27.         ;;
28.     *)
29.         echo "error"
```

30. **esac**

运行结果：

Input integer number:3 ✓

Wednesday

看了这个例子，相信大家对 case in 语句有了一个大体上的认识，那么，接下来我们就正式开始讲解 case in 的用法，它的基本格式如下：

```
case expression in
  pattern1)
    statement1
    ;;
  pattern2)
    statement2
    ;;
  pattern3)
    statement3
    ;;
  .....
  *)
    statementn
esac
```

case、in 和 esac 都是 Shell 关键字，expression 表示表达式，pattern 表示匹配模式。

- expression 既可以是一个变量、一个数字、一个字符串，还可以是一个数学计算表达式，或者是命令的执行结果，只要能够得到 expression 的值就可以。
- pattern 可以是一个数字、一个字符串，甚至是一个简单的正则表达式。

case 会将 expression 的值与 pattern1、pattern2、pattern3 逐个进行匹配：

- 如果 expression 和某个模式（比如 pattern2）匹配成功，就会执行这模式（比如 pattern2）后面对应的所有语句（该语句可以有一条，也可以有多条），直到遇见双分号`;;`才停止；然后整个 case 语句就执行完了，程序会跳出整个 case 语句，执行 esac 后面的其它语句。
- 如果 expression 没有匹配到任何一个模式，那么就执行`*)`后面的语句（`*)`表示其它所有值），直到遇见双分号`;;`或者 `esac` 才结束。`*)`相当于多个 if 分支语句中最后的 else 部分。

如果你有 C 语言、[C++](#)、[Java](#) 等编程经验，这里的`;;`和`*)`就相当于其它编程语言中的 break 和 default。

对`*)`的几点说明：

- Shell case in 语句中的`*)`用来“托底”，万一 expression 没有匹配到任何一个模式，`*)`部分可以做一些“善后”工作，或者给用户一些提示。
- 可以没有`*)`部分。如果 expression 没有匹配到任何一个模式，那么就不执行任何操作。

除最后一个分支外（这个分支可以是普通分支，也可以是`*)`分支），其它的每个分支都必须以`;;`结尾，`;;`代表一个分支的结束，不写的话会有语法错误。最后一个分支可以写`;;`，也可以不写，因为无论如何，执行到 esac 都会结束整个 case in 语句。

上面的代码是 case in 最常见的用法，即 expression 部分是一个变量，pattern 部分是一个数字或者表达式。

case in 和正则表达式

case in 的 pattern 部分支持简单的正则表达式，具体来说，可以使用以下几种格式：

格式	说明
*	表示任意字符串。
[abc]	表示 a、b、c 三个字符中的任意一个。比如，[15ZH] 表示 1、5、Z、H 四个字符中的任意一个。
[m-n]	表示从 m 到 n 的任意一个字符。比如，[0-9] 表示任意一个数字，[0-9a-zA-Z] 表示字母或数字。
	表示多重选择，类似逻辑运算中的或运算。比如，abc xyz 表示匹配字符串 "abc" 或者 "xyz"。

如果不加以说明，Shell 的值都是字符串，expression 和 pattern 也是按照字符串的方式来匹配的；本节第一段代码看起来是判断数字是否相等，其实是判断字符串是否相等。

最后一个分支*)并不是什么语法规定，它只是一个正则表达式，*表示任意字符串，所以不管 expression 的值是什么，*)总能匹配成功。

下面的例子演示了如何在 case in 中使用正则表达式：

```
1.  #!/bin/bash
2.
3.  printf "Input a character: "
4.  read -n 1 char
5.
6.  case $char in
7.      [a-zA-Z])
8.          printf "\nletter\n"
9.          ;;
10.     [0-9])
11.         printf "\nDigit\n"
12.         ;;
13.     [0-9])
14.         printf "\nDigit\n"
15.         ;;
16.     [.,?!])
17.         printf "\nPunctuation\n"
18.         ;;
19.     *)
20.         printf "\nerror\n"
21.  esac
```

运行结果 1 :

Input integer number: S

letter

运行结果 2 :

Input integer number: ,

Punctuation

34. Shell while 循环详解

while 循环是 Shell 脚本中最简单的一种循环，当条件满足时，while 重复地执行一组语句，当条件不满足时，就退出 while 循环。

Shell while 循环的用法如下：

```
while condition
do
    statements
done
```

condition 表示判断条件，statements 表示要执行的语句（可以只有一条，也可以有多条），do 和 done 都是 Shell 中的关键字。

while 循环的执行流程为：

- 先对 condition 进行判断，如果该条件成立，就进入循环，执行 while 循环体中的语句，也就是 do 和 done 之间的语句。这样就完成了一次循环。
- 每一次执行到 done 的时候都会重新判断 condition 是否成立，如果成立，就进入下一次循环，继续执行 do 和 done 之间的语句，如果不成立，就结束整个 while 循环，执行 done 后面的其它 Shell 代码。
- 如果一开始 condition 就不成立，那么程序就不会进入循环体，do 和 done 之间的语句就没有执行的机会。

注意，在 while 循环体中必须有相应的语句使得 condition 越来越趋近于“不成立”，只有这样才能最终退出循环，否则 while 就成了死循环，会一直执行下去，永无休止。

while 语句和 if else 语句中的 condition 用法都是一样的，你可以使用 test 或 [] 命令，也可以使用 (()) 或 [[]]，遗忘的读者请猛击下面的链接回顾：

- [Shell if else](#)
- [Shell 退出状态](#)
- [Shell test 命令](#)
- [Shell \[\[\]\]](#)

while 循环举例

【实例 1】计算从 1 加到 100 的和。

```
1.  #!/bin/bash
2.
3.  i=1
4.  sum=0
5.
6.  while ((i <= 100))
7.  do
8.      ((sum += i))
9.      ((i++))
10. done
```



```
11. echo "The sum is: $sum"
```

运行结果：

The sum is: 5050

在 while 循环中，只要判断条件成立，循环就会执行。对于这段代码，只要变量 i 的值小于等于 100，循环就会继续。每次循环给变量 sum 加上变量 i 的值，然后再给变量 i 加 1，直到变量 i 的值大于 100，循环才会停止。

`i++` 语句使得 i 的值逐步增大，让判断条件越来越趋近于“不成立”，最终退出循环。

对上面的例子进行改进，计算从 m 加到 n 的值。

```
1.  #!/bin/bash
2.
3.  read m
4.  read n
5.  sum=0
6.
7.  while ((m <= n))
8.  do
9.      ((sum += m))
10.     ((m++))
11. done
12. echo "The sum is: $sum"
```

运行结果：

1✓

100✓

The sum is: 5050

【实例 2】实现一个简单的加法计算器，用户每行输入一个数字，计算所有数字的和。

```
1.  #!/bin/bash
2.
3.  sum=0
4.
5.  echo "请输入您要计算的数字，按 Ctrl+D 组合键结束读取"
6.  while read n
7.  do
8.      ((sum += n))
9.  done
10.
```

```
11. echo "The sum is: $sum"
```

运行结果：

12 ✓

33 ✓

454 ✓

6767 ✓

1 ✓

2 ✓

The sum is: 7269

在终端中读取数据，可以等价于在文件中读取数据，按下 Ctrl+D 组合键表示读取到文件流的末尾，此时 read 就会读取失败，得到一个非 0 值的退出状态，从而导致判断条件不成立，结束循环。

35. Shell until 循环用法详解

until 循环和 while 循环恰好相反，当判断条件不成立时才进行循环，一旦判断条件成立，就终止循环。

until 的使用场景很少，一般使用 while 即可。

Shell until 循环的用法如下：

```
until condition
do
    statements
done
```

condition 表示判断条件，statements 表示要执行的语句（可以只有一条，也可以有多条），do 和 done 都是 Shell 中的关键字。

until 循环的执行流程为：

- 先对 condition 进行判断，如果该条件不成立，就进入循环，执行 until 循环体中的语句（do 和 done 之间的语句），这样就完成了一次循环。
- 每一次执行到 done 的时候都会重新判断 condition 是否成立，如果不成立，就进入下一次循环，继续执行循环体中的语句，如果成立，就结束整个 until 循环，执行 done 后面的其它 Shell 代码。
- 如果一开始 condition 就成立，那么程序就不会进入循环体，do 和 done 之间的语句就没有执行的机会。

注意，在 until 循环体中必须有相应的语句使得 condition 越来越趋近于“成立”，只有这样才能最终退出循环，否则 until 就成了死循环，会一直执行下去，永无休止。

上节《[Shell while 循环](#)》演示了如何求从 1 加到 100 的值，这节我们改用 until 循环，请看下面的代码：

```
1.  #!/bin/bash
2.
3.  i=1
4.  sum=0
5.
6.  until ((i > 100))
7.  do
8.      ((sum += i))
9.      ((i++))
10. done
11. echo "The sum is: $sum"
```

运行结果：

The sum is: 5050

在 while 循环中，判断条件为 ((i<=100))，这里将判断条件改为 ((i>100))，两者恰好相反，请读者注意区分。

36. Shell for 循环和 for int 循环详解

除了 while 循环和 until 循环，Shell 脚本还提供了 for 循环，它更加灵活易用，更加简洁明了。Shell for 循环有两种使用形式，下面我们逐一讲解。

C 语言风格的 for 循环

C 语言风格的 for 循环的用法如下：

```
for((exp1; exp2; exp3))
do
    statements
done
```

几点说明：

- exp1、exp2、exp3 是三个表达式，其中 exp2 是判断条件，for 循环根据 exp2 的结果来决定是否继续下一次循环；
- statements 是循环体语句，可以有一条，也可以有多条；
- do 和 done 是 Shell 中的关键字。

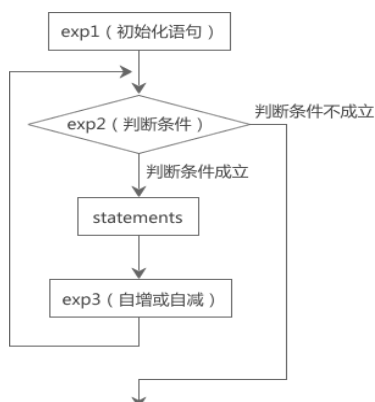
它的运行过程为：

- 1) 先执行 exp1。
- 2) 再执行 exp2，如果它的判断结果是成立的，则执行循环体中的语句，否则结束整个 for 循环。
- 3) 执行完循环体后再执行 exp3。
- 4) 重复执行步骤 2) 和 3)，直到 exp2 的判断结果不成立，就结束循环。

上面的步骤中，2) 和 3) 合并在一起算作一次循环，会重复执行，for 语句的主要作用就是不断执行步骤 2) 和 3)。

exp1 仅在第一次循环时执行，以后都不会再执行，可以认为这是一个初始化语句。exp2 一般是一个关系表达式，决定了是否还要继续下次循环，称为“循环条件”。exp3 很多情况下是一个带有自增或自减运算的表达式，以使循环条件逐渐变得“不成立”。

for 循环的执行过程可用下图表示：



下面我们给出一个实际的例子，计算从 1 加到 100 的和。

```
1. #!/bin/bash
```

```
2.
3.  sum=0
4.
5.  for ((i=1; i<=100; i++))
6.  do
7.      ((sum += i))
8.  done
9.
10. echo "The sum is: $sum"
```

运行结果：

The sum is: 5050

代码分析：

1) 执行到 for 语句时，先给变量 i 赋值为 1，然后判断 i<=100 是否成立；因为此时 i=1，所以 i<=100 成立。接下来会执行循环体中的语句，等循环体执行结束后（sum 的值为 1），再计算 i++。

2) 第二次循环时，i 的值为 2，i<=100 成立，继续执行循环体。循环体执行结束后（sum 的值为 3），再计算 i++。

3) 重复执行步骤 2)，直到第 101 次循环，此时 i 的值为 101，i<=100 不再成立，所以结束循环。

由此我们可以总结出 for 循环的一般形式为：

```
for(( 初始化语句; 判断条件; 自增或自减 ))
do
    statements
done
```

for 循环中的三个表达式

for 循环中的 exp1（初始化语句）、exp2（判断条件）和 exp3（自增或自减）都是可选项，都可以省略（但分号必须保留）。

1) 修改“从 1 加到 100 的和”的代码，省略 exp1：

```
1.  #!/bin/bash
2.
3.  sum=0
4.  i=1
5.
6.  for ((; i<=100; i++))
7.  do
8.      ((sum += i))
9.  done
10.
11. echo "The sum is: $sum"
```

可以看到，将 `i=1` 移到了 `for` 循环的外面。

2) 省略 `exp2`，就没有了判断条件，如果不作其他处理就会成为死循环，我们可以在循环体内部使用 `break` 关键字强制结束循环：

```
1.  #!/bin/bash
2.
3.  sum=0
4.
5.  for ((i=1; ; i++))
6.  do
7.      if(( i>100 )); then
8.          break
9.      fi
10.     ((sum += i))
11. done
12.
13. echo "The sum is: $sum"
```

`break` 是 Shell 中的关键字，专门用来结束循环，后续章节还会深入讲解。

3) 省略了 `exp3`，就不会修改 `exp2` 中的变量，这时可在循环体中加入修改变量的语句。例如：

```
1.  #!/bin/bash
2.
3.  sum=0
4.
5.  for ((i=1; i<=100; ))
6.  do
7.      ((sum += i))
8.      ((i++))
9.  done
10.
11. echo "The sum is: $sum"
```

4) 最后给大家看一个更加极端的例子，同时省略三个表达式：

```
1.  #!/bin/bash
2.
3.  sum=0
```

```
4.  i=0
5.
6.  for (( ; ; ))
7.  do
8.      if(( i>100 )); then
9.          break
10.     fi
11.     ((sum += i))
12.     ((i++))
13. done
14.
15. echo "The sum is: $sum"
```

这种写法并没有什么实际意义，仅仅是为了给大家做演示。

Python 风格的 for in 循环

Python 风格的 for in 循环的用法如下：

```
for variable in value_list
do
    statements
done
```

variable 表示变量，value_list 表示取值列表，in 是 Shell 中的关键字。

in value_list 部分可以省略，省略后的效果相当于 in \$@，本文末尾的「[value_list 使用特殊变量](#)」将会详细讲解。

每次循环都会从 value_list 中取出一个值赋给变量 variable，然后进入循环体（do 和 done 之间的部分），执行循环体中的 statements。直到取完 value_list 中的所有值，循环就结束了。

Shell for in 循环举例：

```
1.  #!/bin/bash
2.
3.  sum=0
4.
5.  for n in 1 2 3 4 5 6
6.  do
7.      echo $n
8.      ((sum+=n))
9.  done
10.
```

```
11. echo "The sum is "$sum
```

运行结果：

```
1
2
3
4
5
6
The sum is 21
```

对 value_list 的说明

取值列表 value_list 的形式有多种，你可以直接给出具体的值，也可以给出一个范围，还可以使用命令产生的结果，甚至使用通配符，下面我们——讲解。

1) 直接给出具体的值

可以在 in 关键字后面直接给出具体的值，多个值之间以空格分隔，比如 `1 2 3 4 5`、`"abc" "390" "tom"`等。

上面的代码中用一组数字作为取值列表，下面我们再演示一下用一组字符串作为取值列表：

```
1.  #!/bin/bash
2.
3.  for str in "C 语言中文网" "http://c.biancheng.net/" "成立 7 年了" "日 IP 数万"
4.  do
5.      echo $str
6.  done
```

运行结果：

```
C 语言中文网
http://c.biancheng.net/
成立 7 年了
日 IP 数万
```

2) 给出一个取值范围

给出一个取值范围的具体格式为：

```
{start..end}
```

start 表示起始值，end 表示终止值；注意中间用两个点号相连，而不是三个点号。根据笔者的实测，这种形式只支持数字和字母。

例如，计算从 1 加到 100 的和：

```
1.  #!/bin/bash
2.
3.  sum=0
4.
```



```
5.  for n in {1..100}
6.  do
7.      ((sum+=n))
8.  done
9.
10. echo $sum
```

运行结果：
5050

再如，输出从 A 到 z 之间的所有字符：

```
1.  #!/bin/bash
2.
3.  for c in {A..z}
4.  do
5.      printf "%c" $c
6.  done
```

输出结果：
ABCDEFGHIJKLMNOPQRSTUVWXYZ^_`abcdefghijklmnopqrstuvwxyz

可以发现，Shell 是根据 ASCII 码表来输出的。

3) 使用命令的执行结果

使用反引号```或者`$()`都可以取得命令的执行结果，我们在《[Shell 变量](#)》一节中已经进行了详细讲解，并对比了两者的优缺点。本节我们使用`$()`这种形式，因为它不容易产生混淆。

例如，计算从 1 到 100 之间所有偶数的和：

```
1.  #!/bin/bash
2.
3.  sum=0
4.
5.  for n in $(seq 2 2 100)
6.  do
7.      ((sum+=n))
8.  done
9.
10. echo $sum
```

运行结果：
2550

seq 是一个 Linux 命令，用来产生某个范围内的整数，并且可以设置步长，不了解的读者请自行百度。seq 2 2 100 表示从 2 开始，每次增加 2，到 100 结束。

再如，列出当前目录下的所有 Shell 脚本文件：

```
1. #!/bin/bash
2.
3. for filename in $(ls *.sh)
4. do
5.     echo $filename
6. done
```

运行结果：

```
demo.sh
test.sh
abc.sh
```

ls 是一个 Linux 命令，用来列出当前目录下的所有文件，*.sh 表示匹配后缀为.sh 的文件，也就是 Shell 脚本文件。

4) 使用 Shell 通配符

Shell 通配符可以认为是一种精简化的正则表达式，通常用来匹配目录或者文件，而不是文本，不了解的读者请猛击《[Linux Shell 通配符 \(glob 模式\)](#)》。

有了 Shell 通配符，不使用 ls 命令也能显示当前目录下的所有脚本文件，请看下面的代码：

```
1. #!/bin/bash
2.
3. for filename in *.sh
4. do
5.     echo $filename
6. done
```

运行结果：

```
demo.sh
test.sh
abc.sh
```

5) 使用特殊变量

Shell 中有多个特殊的变量，例如 \$#、\$*、\$@、\$?、\$\$ 等（不了解的读者请猛击《[Shell 特殊变量](#)》），在 value_list 中就可以使用它们。

```
1. #!/bin/bash
2.
3. function func() {
4.     for str in $@
```

```
5.     do
6.         echo $str
7.     done
8. }
9.
10. func C++ Java Python C#
```

运行结果：

```
C++
Java
Python
C#
```

其实，我们也可以省略 value_list，省略后的效果和使用\$@一样。请看下面的演示：

```
1.  #!/bin/bash
2.
3.  function func() {
4.      for str
5.      do
6.          echo $str
7.      done
8.  }
9.
10. func C++ Java Python C#
```

运行结果：

```
C++
Java
Python
C#
```

37. Shell select in 循环详解

select in 循环用来增强交互性，它可以显示出带编号的菜单，用户输入不同的编号就可以选择不同的菜单，并执行不同的功能。

select in 是 Shell 独有的一种循环，非常适合终端（Terminal）这样的交互场景，C 语言、[C++](#)、[Java](#)、[Python](#)、[C#](#) 等其它编程语言中是没有的。

Shell select in 循环的用法如下：

```
select variable in value_list
do
    statements
done
```

variable 表示变量，value_list 表示取值列表，in 是 Shell 中的关键字。你看，select in 和 [for in](#) 的语法是多么地相似。

我们先来看一个 select in 循环的例子：

```
1.  #!/bin/bash
2.
3.  echo "What is your favourite OS?"
4.  select name in "Linux" "Windows" "Mac OS" "UNIX" "Android"
5.  do
6.      echo $name
7.  done
8.  echo "You have selected $name"
```

运行结果：

```
What is your favourite OS?
1) Linux
2) Windows
3) Mac OS
4) UNIX
5) Android
#? 4↵
You have selected UNIX
#? 1↵
You have selected Linux
#? 9↵
You have selected
#? 2↵
You have selected Windows
#? ^D
```

#? 用来提示用户输入菜单编号；^D 表示按下 Ctrl+D 组合键，它的作用是结束 select in 循环。

运行到 select 语句后，取值列表 value_list 中的内容会以菜单的形式显示出来，用户输入菜单编号，就表示选中了某个值，这个值就会赋给变量 variable，然后再执行循环体中的 statements（do 和 done 之间的部分）。

每次循环时 select 都会要求用户输入菜单编号，并使用环境变量 PS3 的值作为提示符，PS3 的默认值为 #?，修改 PS3 的值就可以修

改提示符。

如果用户输入的菜单编号不在范围之内，例如上面我们输入的 9，那么就会给 variable 赋一个空值；如果用户输入一个空值（什么也不输入，直接回车），会重新显示一遍菜单。

注意，select 是无限循环（死循环），输入空值，或者输入的值无效，都不会结束循环，只有遇到 break 语句，或者按下 Ctrl+D 组合键才能结束循环。

完整实例

select in 通常和 [case in](#) 一起使用，在用户输入不同的编号时可以做出不同的反应。

修改上面的代码，加入 case in 语句：

```
1.  #!/bin/bash
2.
3.  echo "What is your favourite OS?"
4.  select name in "Linux" "Windows" "Mac OS" "UNIX" "Android"
5.  do
6.      case $name in
7.          "Linux")
8.              echo "Linux 是一个类 UNIX 操作系统，它开源免费，运行在各种服务器设备和嵌入式设备。"
9.              break
10.             ;;
11.         "Windows")
12.             echo "Windows 是微软开发的个人电脑操作系统，它是闭源收费的。"
13.             break
14.             ;;
15.         "Mac OS")
16.             echo "Mac OS 是苹果公司基于 UNIX 开发的一款图形界面操作系统，只能运行与苹果提供的硬件之上。"
17.             break
18.             ;;
19.         "UNIX")
20.             echo "UNIX 是操作系统的开山鼻祖，现在已经逐渐退出历史舞台，只应用在特殊场合。"
21.             break
22.             ;;
23.         "Android")
24.             echo "Android 是由 Google 开发的手机操作系统，目前已经占据了 70% 的市场份额。"
25.             break
26.             ;;
27.         *)
```

```
28.             echo "输入错误，请重新输入"
29.         esac
30.     done
```

用户只有输入正确的编号才会结束循环，如果输入错误，会要求重新输入。

运行结果 1，输入正确选项：

```
What is your favourite OS?
1) Linux
2) Windows
3) Mac OS
4) UNIX
5) Android
#? 2
Windows 是微软开发的个人电脑操作系统，它是闭源收费的。
```

运行结果 2，输入错误选项：

```
What is your favourite OS?
1) Linux
2) Windows
3) Mac OS
4) UNIX
5) Android
#? 7
输入错误，请重新输入
#? 4
UNIX 是操作系统的开山鼻祖，现在已经逐渐退出历史舞台，只应用在特殊场合。
```

运行结果 3，输入空值：

```
What is your favourite OS?
1) Linux
2) Windows
3) Mac OS
4) UNIX
5) Android
#?
1) Linux
2) Windows
3) Mac OS
4) UNIX
5) Android
#? 3
Mac OS 是苹果公司基于 UNIX 开发的一款图形界面操作系统，只能运行与苹果提供的硬件之上。
```

38. Shell break 和 continue 跳出循环详解

使用 while、until、for、select 循环时，如果想提前结束循环（在不满足结束条件的情况下结束循环），可以使用 break 或者 continue 关键字。

在 C 语言、C++、C#、Python、Java 等大部分编程语言中，break 和 continue 只能跳出当前层次的循环，内层循环中的 break 和 continue 对外层循环不起作用；但是 Shell 中的 break 和 continue 却能够跳出多层循环，也就是说，内层循环中的 break 和 continue 能够跳出外层循环。

在实际开发中，break 和 continue 一般只用来跳出当前层次的循环，很少有需要跳出多层循环的情况。

break 关键字

Shell break 关键字的用法为：

```
break n
```

n 表示跳出循环的层数，如果省略 n，则表示跳出当前的整个循环。break 关键字通常和 if 语句一起使用，即满足条件时便跳出循环。

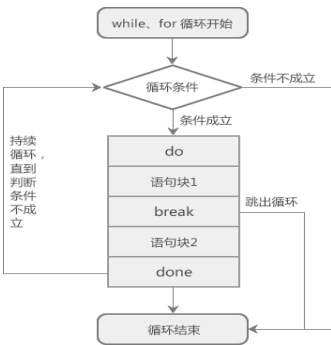


图 1：Shell break 关键字原理示意图

【实例 1】不断从终端读取用户输入的正数，求它们相加的和：

```
1.  #!/bin/bash
2.
3.  sum=0
4.
5.  while read n; do
6.      if ((n>0)); then
7.          ((sum+=n))
8.      else
9.          break
10.     fi
11. done
12.
```

```
13. echo "sum=$sum"
```

运行结果：

10✓

20✓

30✓

0✓

sum=60

while 循环通过 read 命令的退出状态来判断循环条件是否成立，只有当按下 Ctrl+D 组合键（表示输入结束）时，`read n` 才会判断失败，此时 while 循环终止。

除了按下 Ctrl+D 组合键，你还可以输入一个小于等于零的整数，这样会执行 break 语句来终止循环（跳出循环）。

【实例 2】使用 break 跳出双层循环。

如果 break 后面不跟数字的话，表示跳出当前循环，对于有两层嵌套的循环，就得使用两个 break 关键字。例如，输出一个 4*4 的矩阵：

```
1.  #!/bin/bash
2.
3.  i=0
4.  while ((++i)); do #外层循环
5.      if((i>4)); then
6.          break #跳出外层循环
7.      fi
8.
9.      j=0;
10.     while ((++j)); do #内层循环
11.         if((j>4)); then
12.             break #跳出内层循环
13.         fi
14.         printf "%-4d" $((i*j))
15.     done
16.
17.     printf "\n"
18. done
```

运行结果：

```
1  2  3  4
2  4  6  8
3  6  9 12
```


当 $j > 4$ 成立时，执行第二个 `break`，跳出内层循环；外层循环依然执行，直到 $i > 4$ 成立，跳出外层循环。内层循环共执行了 4 次，外层循环共执行了 1 次。

我们也可以在 `break` 后面跟一个数字，让它一次性地跳出两层循环，请看下面的代码：

```
1.  #!/bin/bash
2.
3.  i=0
4.  while ((++i)); do #外层循环
5.      j=0;
6.      while ((++j)); do #内层循环
7.          if ((i>4)); then
8.              break 2 #跳出内外两层循环
9.          fi
10.         if ((j>4)); then
11.             break #跳出内层循环
12.         fi
13.         printf "%-4d" $((i*j))
14.     done
15.
16.     printf "\n"
17. done
```

修改后的代码将所有 `break` 都移到了内层循环里面。读者需要重点关注 `break 2` 这条语句，它使得程序可以一次性跳出两层循环，也就是先跳出内层循环，再跳出外层循环。

continue 关键字

Shell `continue` 关键字的用法为：

```
continue n
```

`n` 表示循环的层数：

- 如果省略 `n`，则表示 `continue` 只对当前层次的循环语句有效，遇到 `continue` 会跳过本次循环，忽略本次循环的剩余代码，直接进入下一次循环。
- 如果带上 `n`，比如 `n` 的值为 2，那么 `continue` 对内层和外层循环语句都有效，不但内层会跳过本次循环，外层也会跳过本次循环，其效果相当于内层循环和外层循环同时执行了不带 `n` 的 `continue`。这么说可能有点难以理解，稍后我们通过代码来演示。

continue 关键字也通常和 if 语句一起使用，即满足条件时便跳出循环。

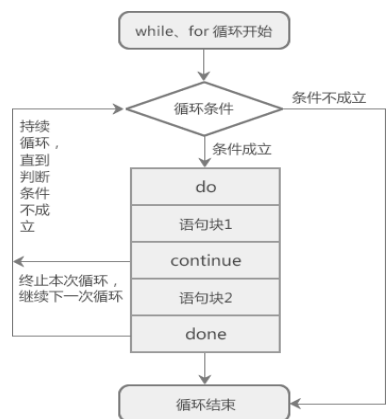


图 2：Shell continue 关键字原理示意图

【实例 1】不断从终端读取用户输入的 100 以内的正数，求它们的和：

```
1.  #!/bin/bash
2.
3.  sum=0
4.
5.  while read n; do
6.      if((n<1 || n>100)); then
7.          continue
8.      fi
9.      ((sum+=n))
10. done
11.
12. echo "sum=$sum"
```

运行结果：

```
10✓
20✓
-1000✓
5✓
9999✓
25✓
sum=60
```

变量 sum 最终的值为 60，-1000 和 9999 并没有计算在内，这是因为 -1000 和 9999 不在 1~100 的范围内，if 判断条件成立，所以执行了 continue 语句，跳过了当次循环，也就是跳过了 `((sum+=n))` 这条语句。

注意，只有按下 Ctrl+D 组合键输入才会结束，read n 才会判断失败，while 循环才会终止。

【实例 2】使用 continue 跳出多层循环，请看下面的代码：

```
1.  #!/bin/bash
2.
3.  for((i=1; i<=5; i++)); do
4.      for((j=1; j<=5; j++)); do
5.          if((i*j==12)); then
6.              continue 2
7.          fi
8.          printf "%d*%d=%-4d" $i $j $((i*j))
9.      done
10.  printf "\n"
11. done
```

运行结果：

```
1*1=1   1*2=2   1*3=3   1*4=4   1*5=5
2*1=2   2*2=4   2*3=6   2*4=8   2*5=10
3*1=3   3*2=6   3*3=9   4*1=4   4*2=8   5*1=5   5*2=10   5*3=15   5*4=20   5*5=25
```

从运行结果可以看出，遇到 `continue 2` 时，不但跳过了内层 `for` 循环，也跳过了外层 `for` 循环。

break 和 continue 的区别

break 用来结束所有循环，循环语句不再有执行的机会；**continue** 用来结束本次循环，直接跳到下一次循环，如果循环条件成立，还会继续循环。

39.Shell 函数详解（函数定义、函数调用）

Shell 函数的本质是一段可以重复使用的脚本代码，这段代码被提前编写好了，放在了指定的位置，使用时直接调取即可。

Shell 中的函数和 [C++](#)、[Java](#)、[Python](#)、[C#](#) 等其它编程语言中的函数类似，只是在语法细节有所差别。

Shell 函数定义的语法格式如下：

```
function name() {  
    statements  
    [return value]  
}
```

对各个部分的说明：

- `function` 是 Shell 中的关键字，专门用来定义函数；
- `name` 是函数名；
- `statements` 是函数要执行的代码，也就是一组语句；
- `return value` 表示函数的返回值，其中 `return` 是 Shell 关键字，专门用在函数中返回一个值；这一部分可以写也可以不写。

由 `{ }` 包围的部分称为函数体，调用一个函数，实际上就是执行函数体中的代码。

函数定义的简化写法

如果你嫌麻烦，函数定义时也可以不写 `function` 关键字：

```
name() {  
    statements  
    [return value]  
}
```

如果写了 `function` 关键字，也可以省略函数名后面的小括号：

```
function name {  
    statements  
    [return value]  
}
```

我建议使用标准的写法，这样能够做到“见名知意”，一看就懂。

函数调用

调用 Shell 函数时可以给它传递参数，也可以不传递。如果不传递参数，直接给出函数名字即可：

```
name
```

如果传递参数，那么多个参数之间以空格分隔：

```
name param1 param2 param3
```

不管是哪种形式，函数名字后面都不需要带括号。

和其它编程语言不同的是，Shell 函数在定义时不能指明参数，但是在调用时却可以传递参数，并且给它传递什么参数它就接收什么参数。

Shell 也不限制定义和调用的顺序，你可以将定义放在调用的前面，也可以反过来，将定义放在调用的后面。

实例演示

1) 定义一个函数，输出 Shell 教程的地址：

```
1.  #!/bin/bash
2.
3.  #函数定义
4.  function url() {
5.      echo "http://c.biancheng.net/shell/"
6.  }
7.
8.  #函数调用
9.  url
```

运行结果：

http://c.biancheng.net/shell/

你可以将调用放在定义的前面，也就是写成下面的形式：

```
1.  #!/bin/bash
2.
3.  #函数调用
4.  url
5.
6.  #函数定义
7.  function url() {
8.      echo "http://c.biancheng.net/shell/"
9.  }
```

2) 定义一个函数，计算所有参数的和：

```
1.  #!/bin/bash
2.
```

```
3.  function getsum() {
4.      local sum=0
5.
6.      for n in $@
7.      do
8.          ((sum+=n))
9.      done
10.
11.     return $sum
12. }
13.
14. getsum 10 20 55 15 #调用函数并传递参数
15. echo $?
```

运行结果：

100

`$@`表示函数的所有参数，`$?`表示函数的退出状态（返回值）。关于如何获取函数的参数，我们将在《[Shell 函数参数](#)》一节中详细讲解。

此处我们借助 `return` 关键字将所有数字的和返回，并使用`$?`得到这个值，这种处理方案在其它编程语言中没有任何问题，但是在 `Shell` 中是非常错误的，`Shell` 函数的返回值和其它编程语言大有不同，我们将在《[Shell 函数返回值](#)》中展开讨论。

40. Shell 函数参数

和 [C++](#)、[C#](#)、[Python](#) 等大部分编程语言不同，Shell 中的函数在定义时不能指明参数，但是在调用时却可以传递参数。

函数参数是 [Shell 位置参数](#) 的一种，在函数内部可以使用 `$n` 来接收，例如，`$1` 表示第一个参数，`$2` 表示第二个参数，依次类推。

除了 `$n`，还有另外三个比较重要的变量：

- `$#` 可以获取传递的参数的个数；
- `$@` 或者 `$*` 可以一次性获取所有的参数（猛击《[Shell \\$*和\\$@的区别](#)》可以了解更多内容）。

`$n`、 `$#`、 `$@`、 `$*` 都属于特殊变量，不了解的读者请转到《[Shell 特殊变量](#)》。

【实例 1】使用 `$n` 来接收函数参数。

```
1.  #!/bin/bash
2.
3.  #定义函数
4.  function show() {
5.      echo "Tutorial: $1"
6.      echo "URL: $2"
7.      echo "Author: ~$3"
8.      echo "Total $# parameters"
9.  }
10.
11. #调用函数
12. show C# http://c.biancheng.net/csharp/ Tom
```

运行结果：

Tutorial: C#

URL: http://c.biancheng.net/csharp/

Author: Tom

Total 3 parameters

注意，第 7 行代码的写法有点不同，这里使用了 [Shell 字符串拼接](#) 技巧。

【实例 2】使用 `$@` 来遍历函数参数。

定义一个函数，计算所有参数的和：

```
1.  #!/bin/bash
2.
3.  function getsum() {
4.      local sum=0
5.
```

```
6.     for n in $@
7.     do
8.         ((sum+=n))
9.     done
10.
11.     echo $sum
12.     return 0
13. }
14.
15. #调用函数并传递参数，最后将结果赋值给一个变量
16. total=$(getsum 10 20 55 15)
17. echo $total
18.
19. #也可以将变量省略
20. echo $(getsum 10 20 55 15)
```

运行结果：

100

100

41. Shell 函数返回值精讲

在 C++、Java、C#、Python 等大部分编程语言中，返回值是指函数被调用之后，执行函数体中的代码所得到的结果，这个结果就通过 return 语句返回。

但是 Shell 中的返回值表示的是函数的退出状态：返回值为 0 表示函数执行成功了，返回值为非 0 表示函数执行失败（出错）了。if、while、for 等语句都是根据函数的退出状态来判断条件是否成立。

Shell 函数的返回值只能是一个介于 0~255 之间的整数，其中只有 0 表示成功，其它值都表示失败。

函数执行失败时，可以根据返回值（退出状态）来判断具体出现了什么错误，比如一个打开文件的函数，我们可以指定 1 表示文件不存在，2 表示文件没有读取权限，3 表示文件类型不对。

如果函数体中没有 return 语句，那么使用默认的退出状态，也就是最后一条命令的退出状态。如果这就是你想要的，那么更加严谨的写法为：

```
return $?
```

\$? 是一个特殊变量，用来获取上一个命令的退出状态，或者上一个函数的返回值，请猛击《[Shell \\$?](#)》了解更多。

如何得到函数的处理结果？

有人可能会疑惑，既然 return 表示退出状态，那么该如何得到函数的处理结果呢？比如，我定义了一个函数，计算从 m 加到 n 的和，最终得到的结果该如何返回呢？

这个问题有两种解决方案：

- 一种是借助全局变量，将得到的结果赋值给全局变量；
- 一种是在函数内部使用 echo、printf 命令将结果输出，在函数外部使用 \${} 或者 `` 捕获结果。

下面我们具体来定义一个函数 getsum，计算从 m 加到 n 的和，并使用以上两种解决方案。

【实例 1】将函数处理结果赋值给一个全局变量。

```
1.  #!/bin/bash
2.
3.  sum=0  #全局变量
4.
5.  function getsum() {
6.      for((i=$1; i<=$2; i++)); do
7.          ((sum+=i))  #改变全局变量
8.      done
9.
10.     return $?  #返回上一条命令的退出状态
11. }
12.
```

```

13. read m
14. read n
15.
16. if getsum $m $n; then
17.     echo "The sum is $sum" #输出全局变量
18. else
19.     echo "Error!"
20. fi

```

运行结果：

```

1
100
The sum is 5050

```

这种方案的弊端是：定义函数的同时还得额外定义一个全局变量，如果我们仅仅知道函数的名字，但是不知道全局变量的名字，那么也是无法获取结果的。

【实例 2】在函数内部使用 echo 输出结果。

```

1.  #!/bin/bash
2.
3.  function getsum() {
4.      local sum=0 #局部变量
5.      for((i=$1; i<=$2; i++)); do
6.          ((sum+=i))
7.      done
8.
9.      echo $sum
10.     return $?
11. }
12.
13. read m
14. read n
15.
16. total=$(getsum $m $n)
17. echo "The sum is $total"
18.
19. #也可以省略 total 变量，直接写成下面的形式
20. #echo "The sum is "$(getsum $m $n)

```

运行结果：

1 ✓

100 ✓

The sum is 5050

代码中总共执行了两次 echo 命令，但是却只输出一次，这是因为 `$()` 捕获了第一个 echo 的输出结果，它并没有真正输出到终端上。除了 `$()`，你也可以使用 ``` 来捕获 echo 的输出结果，请猛击《[Shell 变量](#)》了解两者的区别。

这种方案的弊端是：如果不使用 `$()`，而是直接调用函数，那么就会将结果直接输出到终端上，不过这貌似也无所谓，所以我推荐这种方案。

总起来说，虽然 C 语言、C++、Java 等其它编程语言中的返回值用起来更加方便，但是 Shell 中的返回值有它独特的用途，所以不要带着传统的编程思维来看待 Shell 函数的返回值。

第 3 章：Shell 高级教程

1. Linux Shell 重定向（输入输出重定向）精讲

Linux Shell 重定向分为两种，一种输入重定向，一种是输出重定向；从字面上理解，输入输出重定向就是「改变输入与输出的方向」的意思。

那么，什么是输入输出方向呢？标准的输入输出方向又是什么呢？

一般情况下，我们都是从键盘读取用户输入的数据，然后再把数据拿到程序（C 语言程序、Shell 脚本程序等）中使用；这就是标准的输入方向，也就是从键盘到程序。

反过来说，程序中也会产生数据，这些数据一般都是直接呈现到显示器上，这就是标准的输出方向，也就是从程序到显示器。

我们可以把观点提炼一下，其实输入输出方向就是数据的流动方向：

- 输入方向就是数据从哪里流向程序。数据默认从键盘流向程序，如果改变了它的方向，数据就从其它地方流入，这就是输入重定向。
- 输出方向就是数据从程序流向哪里。数据默认从程序流向显示器，如果改变了它的方向，数据就流向其它地方，这就是输出重定向。

硬件设备和文件描述符

计算机的硬件设备有很多，常见的输入设备有键盘、鼠标、麦克风、手写板等，输出设备有显示器、投影仪、打印机等。不过，在 Linux 中，标准输入设备指的是键盘，标准输出设备指的是显示器。

Linux 中一切皆文件，包括标准输入设备（键盘）和标准输出设备（显示器）在内的所有计算机硬件都是文件。

为了表示和区分已经打开的文件，Linux 会给每个文件分配一个 ID，这个 ID 就是一个整数，被称为文件描述符（File Descriptor）。

表 1：与输入输出有关的文件描述符			
文件描述符	文件名	类型	硬件
0	stdin	标准输入文件	键盘
1	stdout	标准输出文件	显示器
2	stderr	标准错误输出文件	显示器

Linux 程序在执行任何形式的 I/O 操作时，都是在读取或者写入一个文件描述符。一个文件描述符只是一个和打开的文件相关联的整数，它的背后可能是一个硬盘上的普通文件、FIFO、管道、终端、键盘、显示器，甚至是一个网络连接。

stdin、stdout、stderr 默认都是打开的，在重定向的过程中，0、1、2 这三个文件描述符可以直接使用。

Linux Shell 输出重定向

输出重定向是指命令的结果不再输出到显示器上，而是输出到其它地方，一般是文件中。这样做的最大好处就是把命令的结果保存起来，当我们需要的时候可以随时查询。Bash 支持的输出重定向符号如下表所示。

表 2：Bash 支持的输出重定向符号		
类 型	符 号	作 用

标准输出重定向	command >file	以覆盖的方式，把 command 的正确输出结果输出到 file 文件中。
	command >>file	以追加的方式，把 command 的正确输出结果输出到 file 文件中。
标准错误输出重定向	command 2>file	以覆盖的方式，把 command 的错误信息输出到 file 文件中。
	command 2>>file	以追加的方式，把 command 的错误信息输出到 file 文件中。
正确输出和错误信息同时保存	command >file 2>&1	以覆盖的方式，把正确输出和错误信息同时保存到同一个文件（file）中。
	command >>file 2>&1	以追加的方式，把正确输出和错误信息同时保存到同一个文件（file）中。
	command >file1 2>file2	以覆盖的方式，把正确的输出结果输出到 file1 文件中，把错误信息输出到 file2 文件中。
	command >>file1 2>>file2	以追加的方式，把正确的输出结果输出到 file1 文件中，把错误信息输出到 file2 文件中。
	command >file 2>file	【 不推荐 】这两种写法会导致 file 被打开两次，引起资源竞争，所以 stdout 和 stderr 会互相覆盖，我们将在《 结合 Linux 文件描述符谈重定向，彻底理解重定向的本质 》一节中深入剖析。
	command >>file 2>>file	

在输出重定向中，>代表的是覆盖，>>代表的是追加。

注意

输出重定向的完整写法其实是 fd>file 或者 fd>>file，其中 fd 表示文件描述符，如果不写，默认为 1，也就是标准输出文件。

当文件描述符为 1 时，一般都省略不写，如上表所示；当然，如果你愿意，也可以将 command >file 写作 command 1>file，但这样做是多此一举。

当文件描述符为大于 1 的值时，比如 2，就必须写上。

需要重点说明的是，fd 和>之间不能有空格，否则 Shell 会解析失败；>和 file 之间的空格可有可无。为了保持一致，我习惯在>两边都不加空格。

下面的语句是一个反面教材：

```
echo "c.biancheng.net" 1 >log.txt
```

注意 1 和>之间的空格。echo 命令的输出结果是 c.biancheng.net，我们的初衷是将输出结果重定向到 log.txt，但是当你打开 log.txt 文件后，发现文件的内容为 c.biancheng.net 1，这就是多余的空格导致的解析错误。也就是说，Shell 将该条语句理解成了下面的形式：

```
echo "c.biancheng.net" 1 1>log.txt
```

输出重定向举例

【实例 1】将 echo 命令的输出结果以追加的方式写入到 demo.txt 文件中。

```
1. #!/bin/bash
2.
3. for str in "C 语言中文网" "http://c.biancheng.net/" "成立 7 年了" "日 IP 数万"
4. do
5.     echo $str >>demo.txt #将输入结果以追加的方式重定向到文件
```

6. **done**

运行以上脚本，使用 `cat demo.txt` 查看文件内容，显示如下：

C 语言中文网

<http://c.biancheng.net/>

成立 7 年了

日 IP 数万

【实例 2】将 `ls -l` 命令的输出结果重定向到文件中。

```
[c.biancheng.net]$ ls -l #先预览一下输出结果

总用量 16

drwxr-xr-x. 2 root    root      21 7 月   1 2016 abc

-rw-r--r--. 1 mozhiyan mozhiyan 399 3 月   11 17:12 demo.sh

-rw-rw-r--. 1 mozhiyan mozhiyan  67 3 月   22 17:16 demo.txt

-rw-rw-r--. 1 mozhiyan mozhiyan 278 3 月   16 17:17 main.c

-rwxr-xr-x. 1 mozhiyan mozhiyan 187 3 月   22 17:16 test.sh

[c.biancheng.net]$ ls -l >demo.txt #重定向

[c.biancheng.net]$ cat demo.txt #查看文件内容

总用量 12

drwxr-xr-x. 2 root    root      21 7 月   1 2016 abc

-rw-r--r--. 1 mozhiyan mozhiyan 399 3 月   11 17:12 demo.sh

-rw-rw-r--. 1 mozhiyan mozhiyan   0 3 月   22 17:21 demo.txt

-rw-rw-r--. 1 mozhiyan mozhiyan 278 3 月   16 17:17 main.c

-rwxr-xr-x. 1 mozhiyan mozhiyan 187 3 月   22 17:16 test.sh
```

错误输出重定向举例

命令正确执行是没有错误信息的，我们必须刻意地让命令执行出错，如下所示：

```
[c.biancheng.net]$ ls java #先预览一下错误信息

ls: 无法访问 java: 没有那个文件或目录

[c.biancheng.net]$ ls java 2>err.log #重定向

[c.biancheng.net]$ cat err.log #查看文件

ls: 无法访问 java: 没有那个文件或目录
```

正确输出和错误信息同时保存

【实例 1】把正确结果和错误信息都保存到一个文件中，例如：

```
[c.biancheng.net]$ ls -l >out.log 2>&1

[c.biancheng.net]$ ls java >>out.log 2>&1

[c.biancheng.net]$ cat out.log

总用量 12

drwxr-xr-x. 2 root    root      21 7月   1 2016 abc

-rw-r--r--. 1 mozhiyan mozhiyan 399 3月   11 17:12 demo.sh

-rw-rw-r--. 1 mozhiyan mozhiyan 278 3月   16 17:17 main.c

-rw-rw-r--. 1 mozhiyan mozhiyan   0 3月   22 17:39 out.log

-rwxr-xr-x. 1 mozhiyan mozhiyan 187 3月   22 17:16 test.sh

ls: 无法访问 java: 没有那个文件或目录
```

out.log 的最后一行是错误信息，其它行都是正确的输出结果。

【实例 2】上面的实例将正确结果和错误信息都写入同一个文件中，这样会导致视觉上的混乱，不利于以后的检索，所以我建议把正确结果和错误信息分开保存到不同的文件中，也即写成下面的形式：

```
ls -l >>out.log 2>>err.log
```

这样一来，正确的输出结果会写入到 out.log，而错误的信息则会写入到 err.log。

/dev/null 文件

如果你既不想把命令的输出结果保存到文件，也不想把命令的输出结果显示到屏幕上，干扰命令的执行，那么可以把命令的所有结果重定向到 /dev/null 文件中。如下所示：

```
ls -l &&>/dev/null
```

大家可以把 /dev/null 当成 Linux 系统的垃圾箱，任何放入垃圾箱的数据都会被丢弃，不能恢复。

Linux Shell 输入重定向

输入重定向就是改变输入的方向，不再使用键盘作为命令输入的来源，而是使用文件作为命令的输入。

表 3：Bash 支持的输出重定向符号	
符号	说明
command <file	将 file 文件中的内容作为 command 的输入。
command <<END	从标准输入（键盘）中读取数据，直到遇见分界符 END 才停止（分界符可以是任意的字符串，用户自己定义）。
command <file1 >file2	将 file1 作为 command 的输入，并将 command 的处理结果输出到 file2。

和输出重定向类似，输入重定向的完整写法是 `fd<file`，其中 fd 表示文件描述符，如果不写，默认为 0，也就是标准输入文件。

输入重定向举例

【示例 1】统计文档中有多少行文字。

Linux `wc` 命令可以用来对文本进行统计，包括单词个数、行数、字节数，它的用法如下：

```
wc [选项] [文件名]
```

其中，`-c` 选项统计字节数，`-w` 选项统计单词数，`-l` 选项统计行数。

统计 `readme.txt` 文件中有多少行文本：

```
[c.biancheng.net]$ cat readme.txt #预览一下文件内容

C 语言中文网

http://c.biancheng.net/

成立 7 年了

日 IP 数万

[c.biancheng.net]$ wc -l <readme.txt #输入重定向

4
```

【实例 2】逐行读取文件内容。

```
#!/bin/bash

while read str; do

    echo $str

done <readme.txt
```

运行结果：

```
C 语言中文网
http://c.biancheng.net/
成立 7 年了
日 IP 数万
```

这种写法叫做代码块重定向，也就是把一组命令同时重定向到一个文件，我们将在《[Shell 代码块重定向](#)》一节中详细讲解。

【实例 3】统计用户在终端输入的文本的行数。

此处我们使用输入重定向符号 `<<`，这个符号的作用是使用特定的分界符作为命令输入的结束标志，而不使用 `Ctrl+D` 键。

```
[c.biancheng.net]$ wc -l <<END

> 123

> 789
```



```
> abc
```

```
> xyz
```

```
> END
```

```
4
```

wc 命令会一直等待用输入，直到遇见分界符 **END** 才结束读取。

<< 之后的分界符可以自定义，只要再碰到相同的分界符，两个分界符之间的内容将作为命令的输入（不包括分界符本身）。

2. Linux 中的文件描述符到底是什么？

[Linux 中一切皆文件](#)，比如 C++ 源文件、视频文件、Shell 脚本、可执行文件等，就连键盘、显示器、鼠标等硬件设备也都是文件。

一个 Linux 进程可以打开成百上千个文件，为了表示和区分已经打开的文件，Linux 会给每个文件分配一个编号（一个 ID），这个编号就是一个整数，被称为文件描述符（File Descriptor）。

这只是一个形象的比喻，为了让读者容易理解我才这么说。如果你也仅仅理解到这个层面，那不过是浅尝辄止而已，并没有看到文件描述符的本质。

本篇文章的目的就是拨云见雾，从底层实现的角度来给大家剖析一下文件描述符，看看文件描述符到底是如何表示一个文件的。

不过，阅读本篇文章需要你有一定的 C 语言编程基础，至少要理解数组、指针和结构体；如果理解内存，那就更好了，看了这篇文章你会醍醐灌顶。

好了，废话不多说，让我们马上进入正题吧。

Linux 文件描述符到底是什么？

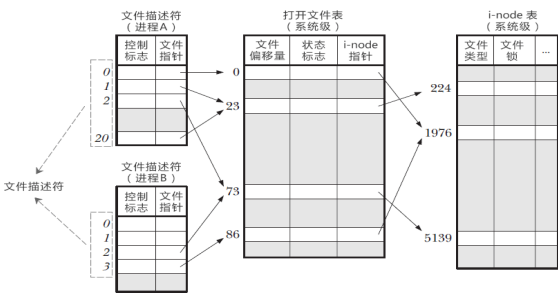
一个 Linux 进程启动后，会在内核空间中创建一个 PCB 控制块，PCB 内部有一个文件描述符表（File descriptor table），记录着当前进程所有可用的文件描述符，也即当前进程所有打开的文件。

内核空间是虚拟地址空间的一部分，想死磕的读者请猛击[《C 语言内存精讲》](#)，不想纠缠细节的读者可以这样理解：进程启动后要占用内存，其中一部分内存分配给了文件描述符表。

除了文件描述符表，系统还需要维护另外两张表：

- 打开文件表（Open file table）
- i-node 表（i-node table）

文件描述符表每个进程都有一个，打开文件表和 i-node 表整个系统只有一个，它们三者之间的关系如下图所示。



从本质上讲，这三种表都是结构体数组，0、1、2、73、1976 等都是数组下标。表头只是我自己添加的注释，数组本身是没有的。实线箭头表示指针的指向，虚线箭头是我自己添加的注释。

你看，文件描述符只不过是一个数组下标吗！

通过文件描述符，可以找到文件指针，从而进入打开文件表。该表存储了以下信息：

- 文件偏移量，也就是文件内部指针偏移量。调用 `read()` 或者 `write()` 函数时，文件偏移量会自动更新，当然也可以使用 `lseek()` 直接修改。
- 状态标志，比如只读模式、读写模式、追加模式、覆盖模式等。

- i-node 表指针。

然而，要想真正读写文件，还得通过打开文件表的 i-node 指针进入 i-node 表，该表包含了诸如以下的信息：

- 文件类型，例如常规文件、套接字或 FIFO。
- 文件大小。
- 时间戳，比如创建时间、更新时间。
- 文件锁。

对上图的进一步说明：

- 在进程 A 中，文件描述符 1 和 20 都指向了同一个打开文件表项，标号为 23（指向了打开文件表中下标为 23 的数组元素），这可能是通过调用 `dup()`、`dup2()`、`fcntl()` 或者对同一个文件多次调用了 `open()` 函数形成的。
- 进程 A 的文件描述符 2 和进程 B 的文件描述符 2 都指向了同一个文件，这可能是在调用 `fork()` 后出现的（即进程 A、B 是父子进程关系），或者是不同的进程独自去调用 `open()` 函数打开了同一个文件，此时进程内部的描述符正好分配到与其他进程打开该文件的描述符一样。
- 进程 A 的描述符 0 和进程 B 的描述符 3 分别指向不同的打开文件表项，但这些表项均指向 i-node 表的同一个条目（标号为 1976）；换言之，它们指向了同一个文件。发生这种情况是因为每个进程各自对同一个文件发起了 `open()` 调用。同一个进程两次打开同一个文件，也会发生类似情况。

有了以上对文件描述符的认知，我们很容易理解以下情形：

- 同一个进程的不同文件描述符可以指向同一个文件；
- 不同进程可以拥有相同的文件描述符；
- 不同进程的同一个文件描述符可以指向不同的文件（一般也是这样，除了 0、1、2 这三个特殊的文件）；
- 不同进程的不同文件描述符也可以指向同一个文件。

3. 结合文件描述符谈重定向，彻底理解重定向的本质！

《Linux 重定向》一节讲解了输入输出重定向的各种写法，并提到了文件描述符的概念；《Linux 文件描述符》一节从底层剖析了文件描述符的本质，它只不过是一个数组下标。本节我们就将两者结合起来，看看 Shell 是如何借助文件描述符实现重定向的。

Linux 系统这个“傻帽”只有一根筋，每次读写文件的时候，都从文件描述符下手，通过文件描述符找到文件指针，然后进入打开文件表和 i-node 表，这两个表里面才真正保存了与打开文件相关的各种信息。

试想一下，如果我们改变了文件指针的指向，不就改变了文件描述符对应的真实文件吗？比如文件描述符 1 本来对应显示器，但是我们偷偷将文件指针指向了 log.txt 文件，那么文件描述符 1 也就和 log.txt 对应起来了。

文件指针只不过是一个内存地址，修改它是轻而易举的事情。文件指针是文件描述符和真实文件之间最关键的“纽带”，然而这条纽带却非常脆弱，很容易被修改。

Linux 系统提供的函数可以修改文件指针，比如 dup()、dup2()；Shell 也能修改文件指针，输入输出重定向就是这么干的。

对，没错，输入输出重定向就是通过修改文件指针实现的！更准确地说，发生重定向时，Linux 会用文件描述符表（一个结构体数组）中的一个元素给另一个元素赋值，或者用一个结构体变量给数组元素赋值，整体上的资源开销相当低。

你看，发生重定向的时候，文件描述符并没有改变，改变的是文件描述符对应的文件指针。对于标准输出，Linux 系统始终向文件描述符 1 中输出内容，而不管它的文件指针指向哪里；只要我们修改了文件指针，就能向任意文件中输出内容。

以下面的语句为例来说明：

```
echo "c.biancheng.net" 1>log.txt
```

文件描述符表本质上是一个结构体数组，假设这个结构体的名字叫做 FD。发生重定向时，Linux 系统首先会打开 log.txt 文件，并把各种信息添加到 i-node 表和文件打开表，然后再创建一个 FD 变量（通过这个变量其实就能读写文件了），并用这个变量给下标为 1 的数组元素赋值，覆盖原来的内容，这样就改变了文件指针的指向，完成了重定向。

Shell 对文件描述符的操作

前面提到，`>`是输出重定向符号，`<`是输入重定向符号；更准确地说，它们应该叫做文件描述符操作符。`>`和`<`通过修改文件描述符改变了文件指针的指向，所以能够实现重定向的功能。

除了`>`和`<`，Shell 还支持`<>`，它的效果是前面两者的总和。

Shell 文件描述符操作方法一览表		
分类	用法	说明
输出	n>filename	以输出的方式打开文件 filename，并绑定到文件描述符 n。n 可以不写，默认为 1，也即标准输出文件。
	n>&m	用文件描述符 m 修改文件描述符 n，或者说用文件描述符 m 的内容覆盖文件描述符 n，结果就是 n 和 m 都代表了同一个文件，因为 n 和 m 的文件指针都指向了同一个文件。 因为使用的是 <code>></code> ，所以 n 和 m 只能用作命令的输出文件。n 可以不写，默认为 1。
	n>&-	关闭文件描述符 n 及其代表的文件。n 可以不写，默认为 1。
	&>filename	将正确输出结果和错误信息全部重定向到 filename。
输入	n<filename	以输入的方式打开文件 filename，并绑定到文件描述符 n。n 可以不写，默认为 0，也即标准输入文件。
	n<&m	类似于 n>&m，但是因为使用的是 <code><</code> ，所以 n 和 m 只能用作命令的输入文件。n 可以不写，默认为 0。

	n<&-	关闭文件描述符 n 及其代表的文件。n 可以不写，默认为 0。
输入和输出	n<> filename	同时以输入和输出的方式打开文件 filename，并绑定到文件描述符 n，相当于 n>filename 和 n<filename 的总和。。n 可以不写，默认为 0。

【实例 1】前面的文章中提到了下面这种用法：

```
command >file 2>&1
```

它省略了文件描述符 1，所以等价于：

```
command 1>file 2>&1
```

这个语句可以分成两步：先执行 1>file，让文件描述符 1 指向 file；再执行 2>&1，用文件描述符 1 修改文件描述符 2，让 2 和 1 的内容一样。最终 1 和 2 都指向了同一个文件，也就是 file。所以不管是向 1 还是向 2 中输出内容，最终都输出到 file 文件中。

这里需要注意执行顺序，多个操作符在一起会从左往右依次执行。对于上面的语句，就是先执行 1>file，再执行 2>&1；如果写作下面的形式，那就南辕北辙了：

```
command 2>&1 1>file
```

Shell 会先执行 2>&1，这样 1 和 2 都指向了标准错误输出文件，也即显示器；接着执行 1>file，这样 1 就指向了 file 文件，但是 2 依然指向显示器。最终的结果是，正确的输出结果输出到了 file 文件，错误信息却还是输出到显示器。

【实例 2】一个比较奇葩的重定向写法。

```
echo "C 语言中文网" 10>log.txt >&10
```

先执行 10>log.txt，打开 log.txt，并给它分配文件描述符 10；接着执行>&10，用文件描述符 10 来修改文件描述符 1（对于>，省略不写的话默认为 1），让 1 和 10 都指向 log.txt 文件，最终的结果是向 log.txt 文件中输出内容。

这条语句其实等价于 echo "C 语言中文网" >log.txt，我之所以写得这么绕，是为了让大家理解各种操作符的用法。

文件描述符 10 只用了一次，我们在末尾最好将它关闭，这是一个好习惯。

```
echo "C 语言中文网" 10>log.txt >&10 10>&-
```

4. 使用 exec 命令操作文件描述符

exec 是 [Shell 内置命令](#)，它有两种用法，一种是执行 Shell 命令，一种是操作文件描述符。本节只讲解后面一种，前面一种请大家自行学习。

使用 exec 命令可以永久性地重定向，后续命令的输入输出方向也被确定了，直到再次遇到 exec 命令才会改变重定向的方向；换句话说，一次重定向，永久有效。

嗯？什么意思？难道说我们以前使用的重定向都是临时的吗？是的！前面使用的重定向都是临时的，它们只对当前的命令有效，对后面的命令无效。

请看下面的例子：

```
[mozhiyan@localhost ~]$ echo "c.biancheng.net" > log.txt

[mozhiyan@localhost ~]$ echo "C 语言中文网"

C 语言中文网

[mozhiyan@localhost ~]$ cat log.txt

c.biancheng.net
```

第一个 echo 命令使用了重定向，将内容输出到 log.txt 文件；第二个 echo 命令没有再次使用重定向，内容就直接输出到显示器上了。很明显，重定向只对第一个 echo 有效，对第二个 echo 无效。

有些脚本文件的输出内容很多，我们不希望直接输出到显示器上，或者我们需要把输出内容备份到文件中，方便以后检索，按照以前的思路，必须在每个命令后面都使用一次重定向，写起来非常麻烦。如果以后想修改重定向的方向，那工作量也是不小的。

exec 命令就是为解决这种困境而生的，它可以让重定向对当前 Shell 进程中的所有命令有效，它的用法为：

exec 文件描述符操作

在《[结合 Linux 文件描述符谈重定向，彻底理解重定向的本质](#)》一节讲到的所有对文件描述符的操作方式 exec 都支持，请看下面的例子：

```
[mozhiyan@localhost ~]$ echo "重定向未发生"

重定向未发生

[mozhiyan@localhost ~]$ exec >log.txt

[mozhiyan@localhost ~]$ echo "c.biancheng.net"

[mozhiyan@localhost ~]$ echo "C 语言中文网"

[mozhiyan@localhost ~]$ exec >&2

[mozhiyan@localhost ~]$ echo "重定向已恢复"

重定向已恢复

[mozhiyan@localhost ~]$ cat log.txt

c.biancheng.net

C 语言中文网
```

对代码的说明：

- `exec >log.txt` 将当前 Shell 进程的所有标准输出重定向到 `log.txt` 文件，它等价于 `exec 1>log.txt`。
- 后面的两个 `echo` 命令都没有在显示器上输出，而是输出到了 `log.txt` 文件。
- `exec >&2` 用来恢复重定向，让标准输出重新回到显示器，它等价于 `exec 1>&2`。2 是标准错误输出的文件描述符，它也是输出到显示器，并且没有遭到破坏，我们用 2 来覆盖 1，就能修复 1，让 1 重新指向显示器。
- 接下来的 `echo` 命令将结果输出到显示器上，证明 `exec >&2` 奏效了。
- 最后我们用 `cat` 命令来查看 `log.txt` 文件的内容，发现就是中间两个 `echo` 命令的输出。

重定向的恢复

类似 `echo "1234" >log.txt` 这样的重定向只是临时的，当前命名执行完毕后会自动恢复到显示器，我们不用担心。但是诸如 `exec >log.txt` 这种使用 `exec` 命令的重定向都是持久的，如果我们想再次回到显示器，就必须手动恢复。

以输出重定向为例，手动恢复的方法有两种：

- `/dev/tty` 文件代表的就是显示器，将标准输出重定向到 `/dev/tty` 即可，也就是 `exec >/dev/tty`。
- 如果还有别的文件描述符指向了显示器，那么也可以别的文件描述符来恢复标号为 1 的文件描述符，例如 `exec >&2`。注意，如果文件描述符 2 也被重定向了，那么这种方式就无效了。

下面的例子演示了输入重定向的恢复：

```
1.  #!/bin/bash
2.
3.  exec 6<&0  #先将 0 号文件描述符保存
4.  exec <nums.txt  #输入重定向
5.
6.  sum=0
7.  while read n; do
8.      ((sum += n))
9.  done
10. echo "sum=$sum"
11.
12. exec 0<&6 6<&-  #恢复输入重定向，并关闭文件描述符 6
13.
14. read -p "请输入名字、网址和年龄：" name url age
15. echo "$name 已经$age 岁了，它的网址是 $url"
```

将代码保存到 `test.txt`，并执行下面的命令：

```
[mozhiyan@localhost ~]$ cat nums.txt
```

80

```
33
129
71
100
222
8

[mozhiyan@localhost ~]$ bash ./test.sh

sum=643

请输入名字、网址和年龄：C 语言中文网 http://c.biancheng.net 7

C 语言中文网已经 7 岁了，它的网址是 http://c.biancheng.net
```

5. Shell 代码块重定向（对一组命令进行重定向）

所谓代码块，就是由多条语句组成的一个整体；for、while、until 循环，或者 if...else、case...in 选择结构，或者由{}包围的命令都可以称为代码块。

请转到《[Shell 组命令](#)》了解更多关于{}的细节。

将重定向命令放在代码块的结尾处，就可以对代码块中的所有命令实施重定向。

【实例 1】使用 while 循环不断读取 nums.txt 中的数字，计算它们的总和。

```
1.  #!/bin/bash
2.
3.  sum=0
4.  while read n; do
5.      ((sum += n))
6.  done <nums.txt #输入重定向
7.  echo "sum=$sum"
```

将代码保存到 test.sh 并运行：

```
[c.biancheng.net]$ cat nums.txt

80

33

129

71

100

222

8

[c.biancheng.net]$ ./test.sh

sum=643
```

对上面的代码进行改进，记录 while 的读取过程，并将输出结果重定向到 log.txt 文件：

```
1.  #!/bin/bash
2.
3.  sum=0
4.  while read n; do
5.      ((sum += n))
6.      echo "this number: $n"
7.  done <nums.txt >log.txt #同时使用输入输出重定向
8.  echo "sum=$sum"
```

将代码保存到 test.sh 并运行：

```
[c.biancheng.net]$ . ./test.sh

sum=643

[c.biancheng.net]$ cat log.txt

this number: 80

this number: 33

this number: 129

this number: 71

this number: 100

this number: 222

this number: 8
```

【实例 2】对{}包围的代码使用重定向。

```
1.  #!/bin/bash
2.
3.  {
4.      echo "C 语言中文网";
5.      echo "http://c.biancheng.net";
6.      echo "7"
7.  } >log.txt #输出重定向
8.
9.  {
10.     read name;
11.     read url;
12.     read age
13.  } <log.txt #输入重定向
14.
15. echo "$name 已经$age 岁了，它的网址是 $url"
```

将代码保存到 test.sh 并运行：

```
[c.biancheng.net]$ . ./test.sh

C 语言中文网已经 7 岁了，它的网址是 http://c.biancheng.net

[c.biancheng.net]$ cat log.txt

C 语言中文网
```

<http://c.biancheng.net>

7

6. Shell Here Document (内嵌文档/立即文档)

Shell 还有一种特殊形式的重定向叫做 “Here Document” ，目前没有统一的翻译，你可以将它理解为 “嵌入文档” “内嵌文档” “立即文档” 。

所谓文档，就是命令需要处理的数据或者字符串；所谓嵌入，就是把数据和代码放在一起，而不是分开存放（比如将数据放在一个单独的文件中）。有时候命令需要处理的数据量很小，将它放在一个单独的文件中有点“大动干戈”，不如直接放在代码中来得方便。

Here Document 的基本用法为：

```
command <<END
  document
END
```

`command` 是 Shell 命令，`<<END` 是开始标志，`END` 是结束标志，`document` 是输入的文档（也就是一行一行的字符串）。

这种写法告诉 Shell 把 `document` 部分作为命令需要处理的数据，直到遇见终止符 `END` 为止（终止符 `END` 不会被读取）。

注意，终止符 `END` 必须独占一行，并且要定顶格写。

分界符（终止符）可以是任意的字符串，由用户自己定义，比如 `END`、`MARKER` 等。分界符可以出现在正常的数据流中，只要它不是顶格写的独立的一行，就不会被作为结束标志。

【实例 1】 `cat` 命令一般是从文件中读取内容，并将内容输出到显示器上，借助 Here Document，`cat` 命令可以从键盘上读取内容。

```
[mozhiyan@localhost ~]$ cat <<END
```

```
> Shell 教程
```

```
> http://c.biancheng.net/shell/
```

```
> 已经进行了三次改版
```

```
> END
```

```
Shell 教程
```

```
http://c.biancheng.net/shell/
```

```
已经进行了三次改版
```

`<` 是第二层命令提示符。

正文中也可以出现结束标志 `END`，只要它不是独立的一行，并且不顶格写，就没问题。

```
[mozhiyan@localhost ~]$ cat <<END
```

```
> END 可以出现在行首
```

```
> 出现在行尾的 END
```

```
> 出现在中间的 END 也是允许的
```

```
> END
```

```
END 可以出现在行首
```

```
出现在行尾的 END
```

```
出现在中间的 END 也是允许的
```

【实例 2】在脚本文件中使用 Here Document，并将 document 中的内容转换为大写。

```
1.  #!/bin/bash
2.  #在脚本文件中使用立即文档
3.
4.  tr a-z A-Z <<END
5.  one two three
6.  Here Document
7.  END
```

将代码保存到 test.sh 并运行，结果为：

```
ONE TWO THREE
HERE DOCUMENT
```

忽略命令替换

默认情况下，正文中出现的变量和命令也会被求值或运行，Shell 会先将它们替换以后再交给 command，请看下面的例子：

```
[mozhiyan@localhost ~]$ name=C 语言中文网

[mozhiyan@localhost ~]$ url=http://c.biancheng.net

[mozhiyan@localhost ~]$ age=7

[mozhiyan@localhost ~]$ cat <<END

> ${name} 已经${age} 岁了， 它的网址是 ${url}

> END

C 语言中文网已经 7 岁了，它的网址是 http://c.biancheng.net
```

你可以将分界符用单引号或者双引号包围起来使 Shell 替换失效：

```
[mozhiyan@localhost ~]$ name=C 语言中文网

[mozhiyan@localhost ~]$ url=http://c.biancheng.net

[mozhiyan@localhost ~]$ age=7

[mozhiyan@localhost ~]$ cat <<'END'  #使用单引号包围

> ${name} 已经${age} 岁了， 它的网址是 ${url}

> END

${name} 已经${age} 岁了，它的网址是 ${url}
```

忽略制表符

默认情况下，行首的制表符也被当做正文的一部分。

```
1.  #!/bin/bash
2.
3.  cat <<END
4.      Shell 教程
5.      http://c.biancheng.net/shell/
6.      已经进行了三次改版
7.  END
```

将代码保存到 test.sh 并运行，结果如下：

```
Shell 教程
http://c.biancheng.net/shell/
已经进行了三次改版
```

这里的制表符仅仅是为了格式对齐，我们并不希望它作为正文的一部分，为了达到这个目的，你可以在 `<<` 和 `END` 之间增加 `-`，请看下面的代码：

```
1.  #!/bin/bash
2.
3.  #增加了减号-
4.  cat <<-END
5.      Shell 教程
6.      http://c.biancheng.net/shell/
7.      已经进行了三次改版
8.  END
```

这次的运行结果为：

```
Shell 教程
http://c.biancheng.net/shell/
已经进行了三次改版
```

总结

如果你尝试在脚本嵌入一小块多行数据，使用 Here Document 是很有用的，而嵌入很大的数据块是一个不好的习惯。你应该保持你的逻辑（你的代码）和你的输入（你的数据）分离，最好是在不同的文件中，除非是输入一个很小的数据集。

Here Document 最常用的功能还是向用户显示命令或者脚本的用法信息，例如类似下面的函数：

```
1.  usage() {
2.      cat <<-END
```

```
3.      usage: command [-x] [-v] [-z] [file ...]
4.      A short explanation of the operation goes here.
5.      It might be a few lines long, but shouldn't be excessive.
6.  END
7.  }
```

7.Shell Here String (内嵌字符串 , 嵌入式字符串)

Here String 是 [Here Document](#) 的一个变种 , 它的用法如下 :

```
command <<< string
```

command 是 Shell 命令，string 是字符串（它只是一个普通的字符串，并没有什么特别之处）。

这种写法告诉 Shell 把 string 部分作为命令需要处理的数据。例如，将小写字符串转换为大写：

```
[mozhiyan@localhost ~]$ tr a-z A-Z <<< one
```

```
ONE
```

Here String 对于这种发送较短的数据到进程是非常方便的，它比 Here Document 更加简洁。

双引号和单引号

一个单词不需要使用引号包围，但如果 string 中带有空格，则必须使用双引号或者单引号包围，如下所示：

```
[mozhiyan@localhost ~]$ tr a-z A-Z <<< "one two three"
```

```
ONE TWO THREE
```

双引号和单引号是有区别的，双引号会解析其中的变量（当然不写引号也会解析），单引号不会，请看下面的代码：

```
[mozhiyan@localhost ~]$ var=two
```

```
[mozhiyan@localhost ~]$ tr a-z A-Z <<<"one $var there"
```

```
ONE TWO THERE
```

```
[mozhiyan@localhost ~]$ tr a-z A-Z <<<'one $var there'
```

```
ONE $VAR THERE
```

```
[mozhiyan@localhost ~]$ tr a-z A-Z <<<one${var}there
```

```
ONETWOTHERE
```

有了引号的包围，Here String 还可以接收多行字符串作为命令的输入，如下所示：

```
[mozhiyan@localhost ~]$ tr a-z A-Z <<<"one two there
```

```
> four five six
```

```
> seven eight"
```

```
ONE TWO THERE
```

```
FOUR FIVE SIX
```

```
SEVEN EIGHT
```


总结

与 Here Document 相比，Here String 通常是相当方便的，特别是发送变量内容（而不是文件）到像 grep 或者 sed 这样的过滤程序时。

8. Shell 组命令（把多条命令看做一个整体）

所谓组命令，就是将多个命令划分为一组，或者看成一个整体。

Shell 组命令的写法有两种：

```
{ command1; command2; command3; ... }  
(command1; command2; command3; ... )
```

两种写法的区别在于：由花括号`{}`包围起来的组命名在当前 Shell 进程中执行，而由小括号`()`包围起来的组命令会创建一个子 Shell，所有命令都在子 Shell 中执行。

对于第一种写法，花括号和命令之间必须有一个空格，并且最后一个命令必须用一个分号或者一个换行符结束。

子 Shell 就是一个子进程，是通过当前 Shell 进程创建的一个新进程。但是子 Shell 和一般的子进程（比如 `bash ./test.sh` 创建的子进程）还是有差别的，我们将在《[子 Shell 和子进程](#)》一节中深入讲解，读者暂时把子 Shell 和子进程等价起来就行。

组命令可以将多条命令的输出结果合并在一起，在使用重定向和管道时会特别方便。

例如，下面的代码将多个命令的输出重定向到 `out.txt`：

```
1. ls -l > out.txt #>表示覆盖  
2. echo "http://c.biancheng.net/shell/" >> out.txt #>>表示追加  
3. cat readme.txt >> out.txt
```

本段代码共使用了三次重定向。

借助组命令，我们可以将以上三条命令合并在一起，简化成一次重定向：

```
{ ls -l; echo "http://c.biancheng.net/shell/"; cat readme.txt; } > out.txt
```

或者写作：

```
(ls -l; echo "http://c.biancheng.net/shell/"; cat readme.txt) > out.txt
```

使用组命令技术，我们节省了一些打字时间。

类似的道理，我们也可以将组命令和管道结合起来：

```
{ ls -l; echo "http://c.biancheng.net/shell/"; cat readme.txt; } | lpr
```

这里我们把三个命令的输出结果合并在一起，并把它们用管道输送给命令 `lpr` 的输入，以便产生一个打印报告。

两种组命令形式的对比

虽然两种 Shell 组命令形式看起来相似，它们都能用在重定向中合并输出结果，但两者之间有一个很重要的不同：由`{}`包围的组命令在当前 Shell 进程中执行，由`()`包围的组命令会创建一个子 Shell，所有命令都会在这个子 Shell 中执行。

在子 Shell 中执行意味着，运行环境被复制给了一个新的 shell 进程，当这个子 Shell 退出时，新的进程也会被销毁，环境副本也会消失，所以在子 Shell 环境中的任何更改都会消失（包括给变量赋值）。因此，在大多数情况下，除非脚本要求一个子 Shell，否则使用`{}`比使用`()`更受欢迎，并且`{}`的进行速度更快，占用的内存更少。

9.Shell 进程替换（把一个命令的输出传递给另一个命令）

进程替换和命令替换非常相似。**命令替换**是把一个命令的输出结果赋值给另一个变量，例如 `dir_files=`ls -l`` 或 `date_time=$(date)`；而进程替换则是把一个命令的输出结果传递给另一个（组）命令。

为了说明进程替换的必要性，我们先来看一个使用管道的例子：

```
1. echo "http://c.biancheng.net/shell/" | read
2. echo $REPLY
```

以上代码输出结果总是为空，因为 `echo` 命令在父 Shell 中执行，而 `read` 命令在子 Shell 中执行，当 `read` 执行结束时，子 Shell 被销毁，`REPLY` 变量也就消失了。管道中的命令总是在子 Shell 中执行的，任何给变量赋值的命令都会遭遇到这个问题。

使用 `read` 读取数据时，如果没有提供变量名，那么读取到的数据将存放到环境变量 `REPLY` 中，这一点已在《[Shell read](#)》中讲到。幸运的是，Shell 提供了一种“特异功能”，叫做进程替换，它可以用来解决这种麻烦。

Shell 进程替换有两种写法，一种用来产生标准输出，借助输入重定向，它的输出结果可以作为另一个命令的输入：

```
<(commands)
```

另一种用来接受标准输入，借助输出重定向，它可以接收另一个命令的输出结果：

```
>(commands)
```

`commands` 是一组命令列表，多个命令之间以分号分隔。注意，`<`或`>`与圆括号之间是没有空格的。

例如，为了解决上面遇到的问题，我们可以像下面这样使用进程替换：

```
1. read <<(echo "http://c.biancheng.net/shell/")
2. echo $REPLY
```

输出结果：

```
http://c.biancheng.net/shell/
```

整体上来看，Shell 把 `echo "http://c.biancheng.net/shell/"` 的输出结果作为 `read` 的输入。`<()` 用来捕获 `echo` 命令的输出结果，`<` 用来将该结果重定向到 `read`。

注意，两个`<`之间是有空格的，第一个`<`表示输入重定向，第二个`<`和`()`连在一起表示进程替换。

本例中的 `read` 命令和第二个 `echo` 命令都在当前 Shell 进程中运行，读取的数据也会保存到当前进程的 `REPLY` 变量，大家都在一个进程中，所以使用 `echo` 能够成功输出。

而在前面的例子中我们使用了管道，`echo` 命令在父进程中运行，`read` 命令在子进程中运行，读取的数据也保存在子进程的 `REPLY` 变量中，`echo` 命令和 `REPLY` 变量不在一个进程中，而子进程的环境变量对父进程是不可见的，所以读取失败。

再来看一个进程替换用作「接受标准输入」的例子：

```
echo "C 语言中文网" >>(read; echo "你好，$REPLY")
```

运行结果：

```
你好，C 语言中文网
```

因为使用了重定向，read 命令从 `echo "C 语言中文网"` 的输出结果中读取数据。

Shell 进程替换的本质

为了能够在不同进程之间传递数据，实际上进程替换会跟系统中的文件关联起来，这个文件的名字为 `/dev/fd/n`（`n` 是一个整数）。该文件会作为参数传递给 `()` 中的命令，`()` 中的命令对该文件是读取还是写入取决于进程替换格式是 `<` 还是 `>`：

- 如果是 `>()`，那么该文件会给 `()` 中的命令提供输入；借助输出重定向，要输入的内容可以从其它命令而来。
- 如果是 `<()`，那么该文件会接收 `()` 中命令的输出结果；借助输入重定向，可以将该文件的内容作为其它命令的输入。

使用 `echo` 命令可以查看进程替换对应的文件名：

```
[c.biancheng.net]$ echo >(true)

/dev/fd/63

[c.biancheng.net]$ echo <(true)

/dev/fd/63

[c.biancheng.net]$ echo >(true) <(true)

/dev/fd/63 /dev/fd/62
```

`/dev/fd/` 目录下有很多序号文件，进程替换一般用的是 63 号文件，该文件是系统内部文件，我们一般查看不到。

我们通过下面的语句进行实例分析：

```
echo "shellsript" > >(read; echo "hello, $REPLY")
```

第一个 `>` 表示输出重定向，它把第一个 `echo` 命令的输出结果重定向到 `/dev/fd/63` 文件中。

`>()` 中的第一个命令是 `read`，它需要从标准输入中读取数据，此时就用 `/dev/fd/63` 作为输入文件，把该文件的内容交给 `read` 命令，接着使用 `echo` 命令输出 `read` 读取到的内容。

可以看到，`/dev/fd/63` 文件起到了数据中转或者数据桥梁的作用，借助重定向，它将 `>()` 内部的命令和外部的命令联系起来，使得数据能够在这些命令之间流通。

10. Linux Shell 管道详解

通过前面的学习，我们已经知道了怎样从文件重定向输入，以及重定向输出到文件。Shell 还有一种功能，就是可以将两个或者多个命令（程序或者进程）连接到一起，把一个命令的输出作为下一个命令的输入，以这种方式连接的两个或者多个命令就形成了**管道（pipe）**。

Linux 管道使用竖线`|`连接多个命令，这被称为**管道符**。Linux 管道的具体语法格式如下：

```
command1 | command2
command1 | command2 [ | commandN... ]
```

当在两个命令之间设置管道时，管道符`|`左边命令的输出就变成了右边命令的输入。只要第一个命令向标准输出写入，而第二个命令是从标准输入读取，那么这两个命令就可以形成一个管道。大部分的 Linux 命令都可以用来形成管道。

这里需要注意，command1 必须有正确输出，而 command2 必须可以处理 command2 的输出结果；而且 command2 只能处理 command1 的正确输出结果，不能处理 command1 的错误信息。

为什么使用管道？

我们先看下面一组命令，使用 `mysqldump`（一个数据库备份程序）来备份一个叫做 `wiki` 的数据库：

```
mysqldump -u root -p '123456' wiki > /tmp/wikidb.backup
gzip -9 /tmp/wikidb.backup
scp /tmp/wikidb.backup username@remote_ip:/backup/mysql/
```

上述这组命令主要做了如下任务：

- `mysqldump` 命令用于将名为 `wike` 的数据库备份到文件 `/tmp/wikidb.backup`；其中 `-u` 和 `-p` 选项分别指出数据库的用户名和密码。
- `gzip` 命令用于压缩较大的数据库文件以节省磁盘空间；其中 `-9` 表示最慢的压缩速度最好的压缩效果。
- `scp` 命令（secure copy，安全拷贝）用于将数据库备份文件复制到 IP 地址为 `remote_ip` 的备份服务器的 `/backup/mysql/` 目录下。其中 `username` 是登录远程服务器的用户名，命令执行后需要输入密码。

上述三个命令依次执行。然而，如果使用管道的话，你就可以将 `mysqldump`、`gzip`、`ssh` 命令相连接，这样就避免了创建临时文件 `/tmp/wikidb.backup`，而且可以同时执行这些命令并达到相同的效果。

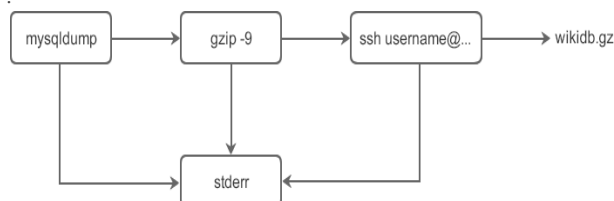
使用管道后的命令如下所示：

```
mysqldump -u root -p '123456' wiki | gzip -9 | ssh username@remote_ip "cat > /backup/wikidb.gz"
```

这些使用了管道的命令有如下特点：

- 命令的语法紧凑并且使用简单。
- 通过使用管道，将三个命令串联到一起就完成了远程 `mysql` 备份的复杂任务。
- 从管道输出的标准错误会混合到一起。

上述命令的数据流如下图所示：



重定向和管道的区别

乍看起来，管道也有重定向的作用，它也改变了数据输入输出的方向，那么，管道和重定向之间到底有什么不同呢？

简单地说，重定向操作符>将命令与文件连接起来，用文件来接收命令的输出；而管道符|将命令与命令连接起来，用第二个命令来接收第一个命令的输出。如下所示：

```
command > file
command1 | command1
```

有些读者在学习管道时会尝试如下的命令，我们来看一下会发生什么：

```
command1 > command2
```

答案是，有时尝试的结果将会很糟糕。这是一个实际的例子，一个 Linux 系统管理员以超级用户（root 用户）的身份执行了如下命令：

```
cd /usr/bin
ls > less
```

第一条命令将当前目录切换到了大多数程序所存放的目录，第二条命令是告诉 Shell 用 ls 命令的输出重写文件 less。因为 /usr/bin 目录已经包含了名称为 less（less 程序）的文件，第二条命令用 ls 输出的文本重写了 less 程序，因此破坏了文件系统上的 less 程序。

这是使用重定向操作符错误重写文件的一个教训，所以在使用它时要谨慎。

Linux 管道实例

【实例 1】将 ls 命令的输出发送到 grep 命令：

```
[c.biancheng.net]$ ls | grep log.txt

log.txt
```

上述命令是查看文件 log.txt 是否存在于当前目录下。

我们可以在命令的后面使用选项，例如使用 -al 选项：

```
[c.biancheng.net]$ ls -al | grep log.txt

-rw-rw-r--.  1 mozhiyan mozhiyan    0 4 月  15 17:26 log.txt
```

管道符|与两侧的命令之间也可以不存在空格，例如将上述命令写作 ls -al|grep log.txt；然而我还是推荐在管道符|和两侧的命令之间使用空格，以增加代码的可读性。

我们也可以重定向管道的输出到一个文件，比如将上述管道命令的输出结果发送到文件 output.txt 中：

```
[c.biancheng.net]$ ls -al | grep log.txt >output.txt

[c.biancheng.net]$ cat output.txt

-rw-rw-r--.  1 mozhiyan mozhiyan    0 4 月  15 17:26 log.txt
```

【实例 2】使用管道将 cat 命令的输出作为 less 命令的输入，这样就可以将 cat 命令的输出每次按照一个屏幕的长度显示，这对于查看长度大于一个屏幕的文件内容很有帮助。

```
cat /var/log/message | less
```

【实例 3】查看指定程序的进程运行状态，并将输出重定向到文件中。

```
[c.biancheng.net]$ ps aux | grep httpd > /tmp/ps.output

[c.biancheng.net]$ cat /tem/ps.output

mozhiyan  4101      13776  0   10:11 pts/3   00:00:00 grep httpd

root      4578      1        0   Dec09  ?        00:00:00 /usr/sbin/httpd

apache    19984     4578    0   Dec29  ?        00:00:00 /usr/sbin/httpd

apache    19985     4578    0   Dec29  ?        00:00:00 /usr/sbin/httpd

apache    19986     4578    0   Dec29  ?        00:00:00 /usr/sbin/httpd

apache    19987     4578    0   Dec29  ?        00:00:00 /usr/sbin/httpd

apache    19988     4578    0   Dec29  ?        00:00:00 /usr/sbin/httpd

apache    19989     4578    0   Dec29  ?        00:00:00 /usr/sbin/httpd

apache    19990     4578    0   Dec29  ?        00:00:00 /usr/sbin/httpd

apache    19991     4578    0   Dec29  ?        00:00:00 /usr/sbin/httpd
```

【实例 4】显示按用户名排序后的当前登录系统的用户的信息。

```
[c.biancheng.net]$ who | sort

mozhiyan :0                2019-04-16 12:55 (:0)

mozhiyan pts/0            2019-04-16 13:16 (:0)
```

who 命令的输出将作为 sort 命令的输入，所以这两个命令通过管道连接后会显示按照用户名排序的已登录用户的信息。

【实例 5】统计系统中当前登录的用户数。

```
[c.biancheng.net]$ who | wc -l

5
```

管道与输入重定向

输入重定向操作符<可以在管道中使用，以用来从文件中获取输入，其语法类似下面这样：

```
command1 < input.txt | command2
command1 < input.txt | command2 -option | command3
```

例如，使用 `tr` 命令从 `os.txt` 文件中获取输入，然后通过管道将输出发送给 `sort` 或 `uniq` 等命令：

```
[c.biancheng.net]$ cat os.txt

redhat

suse

centos

ubuntu

solaris

hp-ux

fedora

centos

redhat

hp-ux

[c.biancheng.net]$ tr a-z A-Z <os.txt | sort

CENTOS

CENTOS

FEDORA

HP-UX

HP-UX

REDHAT

REDHAT

SOLARIS

SUSE

UBUNTU

[c.biancheng.net]$ tr a-z A-Z <os.txt | sort | uniq

CENTOS

FEDORA

HP-UX

REDHAT

SOLARIS
```


SUSE

UBUNTU

管道与输出重定向

你也可以使用重定向操作符>或>>将管道中的最后一个命令的标准输出进行重定向，其语法如下所示：

```
command1 | command2 | ... | commandN > output.txt
command1 < input.txt | command2 | ... | commandN > output.txt
```

【实例 1】使用 mount 命令显示当前挂载的文件系统的信息，并使用 column 命令格式化列的输出，最后将输出结果保存到一个文件中。

```
[c.biancheng.net]$ mount | column -t >mounted.txt

[c.biancheng.net]$ cat mounted.txt

proc          on  /proc          type  proc          (rw,nosuid,nodev,noexec,relatime)

sysfs         on  /sys           type  sysfs         (rw,nosuid,nodev,noexec,relatime,seclabel)

devtmpfs      on  /dev           type  devtmpfs      (rw,nosuid,seclabel,size=496136k,nr_inodes=124034,mode=755)

securityfs    on  /sys/kernel/security type securityfs (rw,nosuid,nodev,noexec,relatime)

tmpfs         on  /dev/shm       type  tmpfs         (rw,nosuid,nodev,seclabel)

devpts        on  /dev/pts       type  devpts        (rw,nosuid,noexec,relatime,seclabel,gid=5,mode=620,ptmxmode=000)

tmpfs         on  /run           type  tmpfs         (rw,nosuid,nodev,seclabel,mode=755)

tmpfs         on  /sys/fs/cgroup type  tmpfs         (rw,nosuid,nodev,noexec,seclabel,mode=755)

#####此处省略部分内容#####
```

【实例 2】使用 tr 命令将 os.txt 文件中的内容转化为大写，并使用 sort 命令将内容排序，使用 uniq 命令去除重复的行，最后将输出重定向到文件 ox.txt.new。

```
[c.biancheng.net]$ cat os.txt

redhat

suse

centos

ubuntu

solaris

hp-ux
```

```
fedora

centos

redhat

hp-ux

[c.biancheng.net]$ tr a-z A-Z <os.txt | sort | uniq >os.txt.new

[c.biancheng.net]$ cat os.txt.new

CENTOS

FEDORA

HP-UX

REDHAT

SOLARIS

SUSE

UBUNTU
```

11. Shell 过滤器

我们已经知道，将几个命令通过管道符组合在一起就形成一个管道。通常，通过这种方式使用的命令就被称为过滤器。过滤器会获取输入，通过某种方式修改其内容，然后将其输出。

简单地讲，过滤器可以概括为以下两点：

- 如果一个 Linux 命令是从标准输入接收它的输入数据，并在标准输出上产生它的输出数据（结果），那么这个命令就被称为过滤器。
- 过滤器通常与 Linux 管道一起使用。

常用的被作为过滤器使用的命令如下所示：

命令	说明
awk	用于文本处理的解释性程序设计语言，通常被作为数据提取和报告的工具。
cut	用于将每个输入文件（如果没有指定文件则为标准输入）的每行的指定部分输出到标准输出。
grep	用于搜索一个或多个文件中匹配指定模式的行。
tar	用于归档文件的应用程序。
head	用于读取文件的开头部分（默认是 10 行）。如果没有指定文件，则从标准输入读取。
paste	用于合并文件的行。
sed	用于过滤和转换文本的流编辑器。
sort	用于对文本文件的行进行排序。
split	用于将文件分割成块。
strings	用于打印文件中可打印的字符串。
tac	与 cat 命令的功能相反，用于倒序地显示文件或连接文件。
tail	用于显示文件的结尾部分。
tee	用于从标准输入读取内容并写入到标准输出和文件。
tr	用于转换或删除字符。
uniq	用于报告或忽略重复的行。
wc	用于打印文件中的总行数、单词数或字节数。

接下来，我们通过几个实例来演示一下过滤器的使用。

在管道中使用 awk 命令

关于 awk 命令的具体用法，请大家自行学习，本节我们仅通过几个简单的实例来了解一下 awk 命令在管道中的使用。

实例 1

查看系统中的所有的账号名称，并按名称的字母顺序排序。

```
[c.biancheng.net]$ awk -F: '{print $1}' /etc/passwd | sort

adm

apache

avahi
```

```
avahi-autoipd

bin

daemon

dbus

ftp

games

...
```

在上例中，使用冒号`:`作为列分隔符，将文件 `/etc/passwd` 的内容分为了多列，并打印了第一列的信息（即用户名），然后将输出通过管道发送到了 `sort` 命令。

实例 2

列出当前账号最常使用的 10 个命令。

```
[c.biancheng.net]$ history | awk '{print $2}' | sort | uniq -c | sort -rn | head

140 echo

 75 man

 71 cat

 63 su

 53 ls

 50 vi

 47 cd

 40 date

 26 let

 25 paste
```

在上例中，`history` 命令将输出通过管道发送到 `awk` 命令，`awk` 命令默认使用空格作为列分隔符，将 `history` 的输出分为了两列，并把第二列内容作为输出通过管道发送到了 `sort` 命令，使用 `sort` 命令进行排序后，再将输出通过管道发送到了 `uniq` 命令，使用 `uniq` 命令统计了历史命令重复出现的次数，再用 `sort` 命令将 `uniq` 命令的输出按照重复次数从高到低排序，最后使用 `head` 命令默认列出前 10 个的信息。

实例 3

显示当前系统的总内存大小，单位为 KB。

```
[c.biancheng.net]$ free | grep Mem | awk '{print $2}'

2029860
```

在管道中使用 cut 命令

cut 命令被用于文本处理。你可以使用这个命令来提取文件中指定列的内容。

实例 1

查看系统中登录 Shell 是 “/bin/bash” 的用户名和对应的用户主目录的信息：

```
[c.biancheng.net]$ grep "/bin/bash" /etc/passwd | cut -d: -f1,6  
  
root:/root  
  
mozhiyan:/home/mozhiyan
```

如果你对 Linux 系统有所了解，你会知道，/etc/passwd 文件被用来存放用户账号的信息，此文件中的每一行会记录一个账号的信息，每个字段之间用冒号分隔，第一个字段即是账号的账户名，而第六个字段就是账号的主目录的路径。

实例 2

查看当前机器的 CPU 类型。

```
[c.biancheng.net]$ cat /proc/cpuinfo | grep name | cut -d: -f2 | uniq  
  
Intel(R) Core(TM) i5-2520M CPU @ 2.50GHz
```

上例中，执行命令 `cat /proc/cpuinfo | grep name` 得到的内容如下所示：

```
[c.biancheng.net]$ cat /proc/cpuinfo | grep name  
  
model name      : Intel(R) Core(TM) i5-2520M CPU @ 2.50GHz  
  
model name      : Intel(R) Core(TM) i5-2520M CPU @ 2.50GHz  
  
model name      : Intel(R) Core(TM) i5-2520M CPU @ 2.50GHz  
  
model name      : Intel(R) Core(TM) i5-2520M CPU @ 2.50GHz
```

然后，我们使用 cut 命令将上述输出内容以冒号作为分隔符，将内容分为了两列，并显示第二列的内容，最后使用 uniq 命令去掉了重复的行。

实例 3

查看当前目录下的子目录数。

```
[c.biancheng.net]$ ls -l | cut -c 1 | grep d | wc -l  
  
5
```

上述管道命令主要做了如下操作：

- 命令 `ls -l` 输出的内容中，每行的第一个字符表示文件的类型，如果第一个字符是 `d`，就表示文件的类型是目录。
- 命令 `cut -c 1` 是截取每行的第一个字符。
- 命令 `grep d` 来获取文件类型是目录的行。
- 命令 `wc -l` 用来获得 grep 命令输出结果的行数，即目录个数。

在管道中使用 grep 命令

grep 命令是在管道中比较常用的一个命令。

实例 1

查看系统日志文件中的错误信息。

```
[c.biancheng.net]$ grep -i "error:" /var/log/messages | less
```

实例 2

查看系统中 HTTP 服务的进程信息。

```
[c.biancheng.net]$ ps auxwww | grep httpd

apache 18968 0.0 0.0 26472 10404 ?    S    Dec15   0:01 /usr/sbin/httpd
apache 18969 0.0 0.0 25528  8308 ?    S    Dec15   0:01 /usr/sbin/httpd
apache 18970 0.0 0.0 26596 10524 ?    S    Dec15   0:01 /usr/sbin/httpd
```

实例 3

查找我们的程序列表中所有命令名中包含关键字 zip 的命令。

```
[c.biancheng.net]$ ls /bin /usr/bin | sort | uniq | grep zip

bunzip2

bzip2

bzip2recover

gunzip

gzip
```

实例 4

查看系统安装的 kernel 版本及相关的 kernel 软件包。

```
[c.biancheng.net]$ rpm -qa | grep kernel

kernel-2.6.18-92.el5

kernel-debuginfo-2.6.18-92.el5

kernel-debuginfo-common-2.6.18-92.el5

kernel-devel-2.6.18-92.el5
```

实例 5

查找 /etc 目录下所有包含 IP 地址的文件。

```
[c.biancheng.net]$ find /etc -type f -exec grep '[0-9][0-9]*[.][0-9][0-9]*[.][0-9][0-9]*[.][0-9][0-9]*' {} \;
```

在管道中使用 tar 命令

tar 命令是 Linux 系统中最常用的打包文件的程序。

实例 1

你可以使用 tar 命令复制一个目录的整体结构。

```
[c.biancheng.net]$ tar cf - /home/mozhiyan | ( cd /backup/; tar xf - )
```

实例 2

跨网络地复制一个目录的整体结构。

```
[c.biancheng.net]$ tar cf - /home/mozhiyan | ssh remote_host "( cd /backup/; tar xf - )"
```

实例 3

跨网络地压缩复制一个目录的整体结构。

```
[c.biancheng.net]$ tar czf - /home/mozhiyan | ssh remote_host "( cd /backup/; tar xzf - )"
```

实例 4

检查 tar 归档文件的大小，单位为字节。

```
[c.biancheng.net]$ cd /; tar cf - etc | wc -c

215040
```

实例 5

检查 tar 归档文件压缩为 tar.gz 归档文件后所占的大小。

```
[c.biancheng.net]$ tar czf - etc.tar | wc -c

58006
```

实例 6

检查 tar 归档文件压缩为 tar.bz2 归档文件后所占的大小。

```
[c.biancheng.net]$ tar cjf - etc.tar | wc -c

50708
```

在管道中使用 head 命令

有时，你不需要一个命令的全部输出，可能只需要命令的前几行输出。这时，就可以使用 head 命令，它只打印命令的前几行输出。默认的输出行数为 10 行。

实例 1

显示 ls 命令的前 10 行输出。

```
[c.biancheng.net]$ ls /usr/bin | head

addftinfo

afmtodit

apropos

arch

ash

awk

base64

basename

bash

bashbug
```

实例 2

显示 ls 命令的前 5 行内容。

```
[c.biancheng.net]$ ls / | head -n 5

bin

cygdrive

Cygwin.bat

Cygwin.ico

Cygwin-Terminal.ico
```

在管道中使用 uniq 命令

uniq 命令用于报告或删除重复的行。我们将使用一个测试文件进行管道中使用 uniq 命令的实例讲解，其内容如下所示：

```
[c.biancheng.net]$ cat testfile

This line occurs only once.
```



```
This line occurs twice.  
  
This line occurs twice.  
  
This line occurs three times.  
  
This line occurs three times.  
  
This line occurs three times.
```

实例 1

去掉输出中重复的行。

```
[c.biancheng.net]$ sort testfile | uniq  
  
This line occurs only once.  
  
This line occurs three times.  
  
This line occurs twice.
```

实例 2

显示输出中各重复的行出现的次数，并按次数多少倒序显示。

```
[c.biancheng.net]$ sort testfile | uniq -c | sort -nr  
  
3 This line occurs three times.  
  
2 This line occurs twice.  
  
1 This line occurs only once.
```

在管道中使用 wc 命令

wc 命令用于统计包含在文本流中的字符数、单词数和行数。

实例 1

统计当前登录到系统的用户数。

```
[c.biancheng.net]$ who | wc -l
```

实例 2

统计当前的 Linux 系统中的进程数。

```
[c.biancheng.net]$ ps -ef | wc -l
```

12.子 Shell 和子进程到底有什么区别？

Shell 中有很多方法产生子进程，比如以新进程的方式运行 Shell 脚本，使用组命令、管道、命令替换等，但是这些子进程是有区别的。

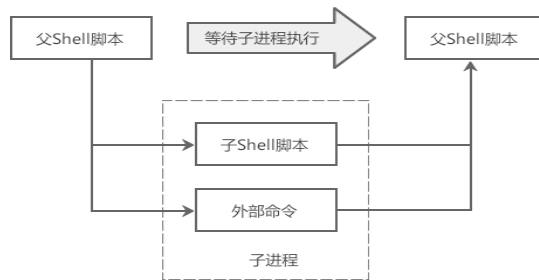
子进程的概念是由父进程的概念引申而来的。在 Linux 系统中，系统运行的应用程序几乎都是从 init (pid 为 1 的进程) 进程派生而来的，所有这些应用程序都可以视为 init 进程的子进程，而 init 则为它们的父进程。

使用 `ps tree -p` 命令就可以看到 init 及系统中其他进程的进程树信息（包括 pid）：

```
systemd(1) ┌─── ModemManager(796) ┌─── {ModemManager}(821)
            │                    └── {ModemManager}(882)
            └─── NetworkManager(975) ┌─── {NetworkManager}(1061)
                                      └── {NetworkManager}(1077)
            └─── abrt-watch-log(774)
            └─── abrt-watch-log(776)
            └─── abrt-d(773)
            └─── accounts-daemon(806) ┌─── {accounts-daemon}(839)
                                     └── {accounts-daemon}(883)
            └─── alsactl(768)
            └─── at-spi-bus-laun(1954) ┌─── dbus-daemon(1958) ┌─── {dbus-daemon}(1960)
                                     │                    │
                                     │                    └── {at-spi-bus-laun}(1955)
                                     │
                                     └── {at-spi-bus-laun}(1957)
                                     └── {at-spi-bus-laun}(1959)
            └─── at-spi2-registr(1962) ┌─── {at-spi2-registr}(1965)
                                     │
                                     └── atd(842)
            └─── auditd(739) ┌─── audispd(753) ┌─── sedispatch(757)
                           │                │
                           │                └── {audispd}(759)
                           └── {auditd}(752)
```

本教程基于 CentOS 7 编写，CentOS 7 为了提高启动速度使用 systemd 替代了 init。CentOS 7 之前的版本依然使用 init。

Shell 脚本是从上至下、从左至右依次执行的，即执行完一个命令之后再执行下一个。如果在 Shell 脚本中遇到子脚本（即脚本嵌套，但是必须以新进程的方式运行）或者外部命令，就会向系统内核申请创建一个新的进程，以便在该进程中执行子脚本或者外部命令，这个新的进程就是子进程。子进程执行完毕后才能回到父进程，才能继续执行父脚本中后续的命令及语句。



子进程的创建

了解 Linux 编程的读者应该知道，使用 `fork()` 函数可以创建一个子进程；除了 PID（进程 ID）等极少的参数不同外，子进程的一切都来自父进程，包括代码、数据、堆栈、打开的文件等，就连代码的执行位置（状态）都是一样的。

也就是说，`fork()` 克隆了一个一模一样的自己，身高、体重、颜值、噪音、年龄等各种属性都相同。当然，后期随着各自的发展轨迹不同，两者会变得不一样，比如 A 好吃懒做越来越肥，B 经常健身成了一个肌肉男；但是在 `fork()` 出来的那一刻，两者都是一样的。

Linux 还有一种创建子进程的方式，就是子进程被 `fork()` 出来以后立即调用 `exec()` 函数加载新的可执行文件，而不使用从父进程继承来的一切。什么意思呢？

比如在 `~/bin` 目录下有两个可执行文件分别叫 `a.out` 和 `b.out`。现在我运行 `a.out`，就会产生一个进程，比如叫做 A。在进程 A 中我又调用 `fork()` 函数创建了一个进程 B，那么 B 就是 A 的子进程，此时它们是一模一样的。但是，我调用 `fork()` 后立即又调用 `exec()` 去加载 `b.out`，这可就坏了，B 进程中的一切（包括代码、数据、堆栈等）都会被销毁，然后再根据 `b.out` 重建建立一切。这样一折腾，B 进程除了 ID 没有变，其它的都变了，再也没有属于 A 的东西了。

你看，同样是创建子进程，但是结果却大相径庭：

- 第一种只使用 `fork()` 函数，子进程和父进程几乎是一模一样的，父进程中的函数、变量、别名等在子进程中仍然有效。
- 第二种使用 `fork()` 和 `exec()` 函数，子进程和父进程之间除了硬生生地维持一种“父子关系”外，再也没有任何联系了，它们就是两个完全不同的程序。

对于 Shell 来说，以新进程的方式运行脚本文件，比如 `bash ./test.sh`、`chmod +x ./test.sh; ./test.sh`，或者在当前 Shell 中使用 `bash` 命令启动新的 Shell，它们都属于第二种创建子进程的方式，所以子进程除了能继承父进程的环境变量外，基本上也不能使用父进程的什么东西了，比如，父进程的全局变量、局部变量、文件描述符、别名等在子进程中都无效。

但是，组命令、命令替换、管道这几种语法都使用第一种方式创建进程，所以子进程可以使用父进程的一切，包括全局变量、局部变量、别名等。我们将这种子进程称为 **子 Shell (sub shell)**。

子 Shell 虽然能使用父 Shell 的一切，但是如果子 Shell 对数据做了修改，比如修改了全局变量，那么这种修改只能停留在子 Shell，无法传递给父 Shell。不管是子进程还是子 Shell，都是“传子不传父”。

总结

子 Shell 才是真正继承了父进程的一切，这才像“一个模子刻出来的”；普通子进程和父进程是完全不同的两个程序，只是维持着父子关系而已。

13.如何检测子 Shell 和子进程？

上节我们说了子 Shell 和子进程的区别，这节就来看一下如何检测它们。

我们都知道使用 \$ 变量可以获取当前进程的 ID，我在父 Shell 和子 Shell 中都输出 \$ 的值，只要它们不一样，不就是创建了一个新的进程吗？那我们就来试一下吧。

```
[mozhiyan@localhost ~]$ echo $$ #父 Shell PID

3299

[mozhiyan@localhost ~]$ (echo $$) #组命令形式的子 Shell PID

3299

[mozhiyan@localhost ~]$ echo "http://c.biancheng.net" | { echo $$; } #管道形式的子 Shell PID

3299

[mozhiyan@localhost ~]$ read <<(echo $$) #进程替换形式的子 Shell PID

[mozhiyan@localhost ~]$ echo $REPLY

3299
```

你看，子 Shell 和父 Shell 的 ID 都是一样的，哪有产生新进程了？作者你是不是骗人呢？

其实不是我骗人，而是你掉坑里了，因为 \$ 变量在子 Shell 中无效！Base 官方文档说，在普通的子进程中，\$ 确实被展开为子进程的 ID；但是在子 Shell 中，\$ 却被展开成父进程的 ID。

除了 \$，Bash 还提供了另外两个环境变量——SHLVL 和 BASH_SUBSHELL，用它们来检测子 Shell 非常方便。

SHLVL 是记录多个 Bash 进程实例嵌套深度的累加器，每次进入一层普通的子进程，SHLVL 的值就加 1。而 BASH_SUBSHELL 是记录一个 Bash 进程实例中多个子 Shell (sub shell) 嵌套深度的累加器，每次进入一层子 Shell，BASH_SUBSHELL 的值就加 1。

1) 我们还是用实例来说话吧，先说 SHLVL。创建一个脚本文件，命名为 test.sh，内容如下：

```
#!/bin/bash

echo "$SHLVL $BASH_SUBSHELL"
```

然后打开 Shell 窗口，依次执行下面的命令：

```
[mozhiyan@localhost ~]$ echo "$SHLVL $BASH_SUBSHELL"

2 0

[mozhiyan@localhost ~]$ bash #执行 bash 命令开启一个新的 Shell 会话

[mozhiyan@localhost ~]$ echo "$SHLVL $BASH_SUBSHELL"

3 0

[mozhiyan@localhost ~]$ bash ./test.sh #通过 bash 命令运行脚本

4 0
```

```

[mozhiyan@localhost ~]$ echo "$SHLVL  $BASH_SUBSHELL"

3  0

[mozhiyan@localhost ~]$ chmod +x ./test.sh  #给脚本增加执行权限

[mozhiyan@localhost ~]$ ./test.sh

4  0

[mozhiyan@localhost ~]$ echo "$SHLVL  $BASH_SUBSHELL"

3  0

[mozhiyan@localhost ~]$ exit  #退出内层 Shell

exit

[mozhiyan@localhost ~]$ echo "$SHLVL  $BASH_SUBSHELL"

2  0

```

SHLVL 和 BASH_SUBSHELL 的初始值都是 0，但是输出结果中 SHLVL 的值从 2 开始，我猜测 Bash 在初始化阶段可能创建了子进程，我们暂时不用理会它，将关注点放在值的变化上。

仔细观察的读者应该会发现，使用 bash 命令开启新的会话后，需要使用 exit 命令退出才能回到上一级 Shell 会话。

`bash ./test.sh` 和 `chmod +x ./test.sh; ./test.sh` 这两种运行脚本的方式，在脚本运行期间会开启一个子进程，运行结束后立即退出子进程。

2) 再说一下 BASH_SUBSHELL，请看下面的命令：

```

[mozhiyan@localhost ~]$ echo "$SHLVL  $BASH_SUBSHELL"

2  0

[mozhiyan@localhost ~]$ (echo "$SHLVL  $BASH_SUBSHELL")  #组命令

2  1

[mozhiyan@localhost ~]$ echo "hello" | { echo "$SHLVL  $BASH_SUBSHELL"; }  #管道

2  1

[mozhiyan@localhost ~]$ var=$(echo "$SHLVL  $BASH_SUBSHELL")  #命令替换

[mozhiyan@localhost ~]$ echo $var

2  1

[mozhiyan@localhost ~]$ ( ( ( (echo "$SHLVL  $BASH_SUBSHELL") ) ) ) )  #四层组命令

2  4

```

你看，组命令、管道、命令替换这几种写法都会进入子 Shell。

注意，“进程替换”看起来好像产生了一个子 Shell，其实只是玩了一个障眼法而已。进程替换只是借助文件在 `()` 内部和外部的命令之间传递数据，但是它并没有创建子 Shell；换句话说，`()` 内部和外部的命令是在一个进程（也就是当前进程）中执行的。

我们不妨来实际检测一下：

```
[mozhiyan@localhost ~]$ echo "$SHLVL  $BASH_SHELL"

2  0

[mozhiyan@localhost ~]$ echo "hello" > >(echo "$SHLVL  $BASH_SHELL")

2  0
```

SHLVL 和 BASH_SHELL 变量的值都没有发生改变，说明进程替换既没有进入子进程，也没有进入子 Shell。

14.Linux 中的信号是什么？

在 Linux 中，理解信号的概念是非常重要的。这是因为，信号被用于通过 Linux 命令行所做的一些常见活动中。例如，每当你按 Ctrl+C 组合键来从命令行终结一个命令的执行，你就使用了信号。每当你使用如下命令来结束一个进程时，你就使用了信号：

```
kill -9 [PID]
```

所以，至少知道信号的基本原理是非常有用的。

Linux 中的信号

在 Linux 系统（以及其他类 Unix 操作系统）中，信号被用于进程间的通信。信号是一个发送到某个进程或同一进程中的特定线程的异步通知，用于通知发生的一个事件。从 1970 年贝尔实验室的 Unix 面世便有了信号的概念，而现在它已经被定义在了 POSIX 标准中。

对于在 Linux 环境进行编程的用户或系统管理员来说，较好地理解信号的概念和机制是很重要的，在某些情况下可以帮助我们更高效地编写程序。对于一个程序来说，如果每条指令都运行正常的话，它会连续地执行。但如果在程序执行时，出现了一个错误或任何异常，内核就可以使用信号来通知相应的进程。

信号同样被用于通信、同步进程和简化进程间通信，在 Linux 中，信号在处理异常和中断方面，扮演了极其重要的角色。信号已经在没有任何较大修改的情况下被使用了将近 30 年。

当一个事件发生时，会产生一个信号，然后内核会将事件传递到接收的进程。有时，进程可以发送一个信号到其他进程。除了进程到进程的信号外，还有很多种情况，内核会产生一个信号，比如文件大小达到限额、一个 I/O 设备就绪或用户发送了一个类似于 Ctrl+C 或 Ctrl+Z 的终端中断等。

运行在用户模式下的进程会接收信号。如果接收的进程正运行在内核模式，那么信号的执行只有在该进程返回到用户模式时才会开始。

发送到非运行进程的信号一定是由内核保存，直到进程重新执行为止。休眠的进程可以是可中断的，也可以是不可中断的。如果一个在可中断休眠状态的进程（例如，等待终端输入的进程）收到了一个信号，那么内核会唤醒这个进程来处理信号。如果一个在不可中断休眠状态的进程收到了一个信号，那么内核会拖延此信号，直到该事件完成为止。

当进程收到一个信号时，可能会发生以下 3 种情况：

- 进程可能会忽略此信号。有些信号不能被忽略，而有些没有默认行为的信号，默认会被忽略。
- 进程可能会捕获此信号，并执行一个被称为信号处理器的特殊函数。
- 进程可能会执行信号的默认行为。例如，信号 15(SIGTERM) 的默认行为是结束进程。

当一个进程执行信号处理时，如果还有其他信号到达，那么新的信号会被阻断直到处理器返回为止。

信号的名称和值

每个信号都有以 SIG 开头的名称，并定义为唯一的正整数。在 Shell 命令行提示符下，输入 kill -l 命令，将显示所有信号的信号值和相应的信号名，类似如下所示：

```
[c.biancheng.net]$ kill -l

1) SIGHUP      2) SIGINT      3) SIGQUIT     4) SIGILL      5) SIGTRAP
6) SIGABRT     7) SIGBUS      8) SIGFPE      9) SIGKILL     10) SIGUSR1
11) SIGSEGV    12) SIGUSR2    13) SIGPIPE     14) SIGALRM    15) SIGTERM
```

16) SIGSTKFLT	17) SIGCHLD	18) SIGCONT	19) SIGSTOP	20) SIGTSTP
21) SIGTTIN	22) SIGTTOU	23) SIGURG	24) SIGXCPU	25) SIGXFSZ
26) SIGVTALRM	27) SIGPROF	28) SIGWINCH	29) SIGIO	30) SIGPWR
31) SIGSYS	34) SIGRTMIN	35) SIGRTMIN+1	36) SIGRTMIN+2	37) SIGRTMIN+3
38) SIGRTMIN+4	39) SIGRTMIN+5	40) SIGRTMIN+6	41) SIGRTMIN+7	42) SIGRTMIN+8
43) SIGRTMIN+9	44) SIGRTMIN+10	45) SIGRTMIN+11	46) SIGRTMIN+12	47) SIGRTMIN+13
48) SIGRTMIN+14	49) SIGRTMIN+15	50) SIGRTMAX-14	51) SIGRTMAX-13	52) SIGRTMAX-12
53) SIGRTMAX-11	54) SIGRTMAX-10	55) SIGRTMAX-9	56) SIGRTMAX-8	57) SIGRTMAX-7
58) SIGRTMAX-6	59) SIGRTMAX-5	60) SIGRTMAX-4	61) SIGRTMAX-3	62) SIGRTMAX-2
63) SIGRTMAX-1	64) SIGRTMAX			

信号值被定义在文件 `/usr/include/bits/signum.h` 中，其源文件是 `/usr/src/linux/kernel/signal.c`。

在 Linux 下，可以查看 `signal(7)` 手册页来查阅信号名列表、信号值、默认的行为和它们是否可以被捕获。其命令如下所示：

man 7 signal

下标所列出的信号是 POSIX 标准的一部分，它们通常被缩写成不带 `SIG` 前缀，例如，`SIGHUP` 通常被简单地称为 `HUP`。

信 号	默认行为	描 述	信号值
SIGABRT	生成 core 文件然后终止进程	这个信号告诉进程终止操作。ABRT 通常由进程本身发送，即当进程调用 <code>abort()</code> 函数发出一个非正常终止信号时	6
SIGALRM	终止	警告时钟	14
SIGBUS	生成 core 文件然后终止进程	当进程引起一个总线错误时，BUS 信号将被发送到进程。例如，访问了一部分未定义的内存对象	10
SIGCHLD	忽略	当了进程结束、被中断或是在被中断之后重新恢复时，CHLD 信号会被发送到进程	20
SIGCONT	继续进程	CONT 信号指不操作系统重新开始先前被 STOP 或 TSTP 暂停的进程	19
SIGFPE	生成 core 文件然后终止进程	当一个进程执行一个错误的算术运算时，FPE 信号会被发送到进程	8
SIGHUP	终止	当进程的控制终端关闭时，HUP 信号会被发送到进程	1
SIGILL	生成 core 文件然后终止进程	当一个进程尝试执行一个非法指令时，ILL 信号会被发送到进程	4
SIGINT	终止	当用户想要中断进程时，INT 信号被进程的控制终端发送到进程	2
SIGKILL	终止	发送到进程的 KILL 信号会使进程立即终止。KILL 信号不能被捕获或忽略	9
SIGPIPE	终止	当一个进程尝试向一个没有连接到其他目标的管道写入时，PIPE 信号会被发送到进程	13
SIGQUIT	终止	当用户要求进程执行 core dump 时，QUIT 信号由进程的控制终端发送到进程	3
SIGSEGV	生成 core 文件然后终止进程	当进程生成了一个无效的内存引用时，SEGV 信号会被发送到进程	11
SIGSTOP	停止进程	STOP 信号指示操作系统停止进程的执行	17

SIGTERM	终止	发送到进程的 TERM 信号用于要求进程终止	15
SIGTSTP	停止进程	TSTP 信号由进程的控制终端发送到进程来要求它立即终止	18
SIGTTIN	停止进程	后台进程尝试读取时，TTIN 信号会被发送到进程	21
SIGTTOU	停止进程	后台进程尝试输出时，TTOU 信号会被发送到进程	22
SIGUSR1	终止	发送到进程的 USR1 信号用于指示用户定义的条件	30
SIGUSR2	终止	同上	31
SIGPOLL	终止	当一个异步输入/输出时间事件发生时，POLL 信号会被发送到进程	23
SIGPROF	终止	当仿形计时器过期时，PROF 信号会被发送到进程	27
SIGSYS	生成 core 文件然后终止进程	发生有错的系统调用时，SYS 信号会被发送到进程	12
SIGTRAP	生成 core 文件然后终止进程	追踪捕获/断点捕获时，会产生 TRAP 信号。	5
SIGURG	忽略	当命一个 socket 有紧急的或是带外数据可被读取时，URG 信号会被发送到进程	16
SIGVTALRM	终止	当进程使用的虚拟计时器过期时，VTALRM 信号会被发送到进程	26
SIGXCPU	终止	当进程使用的 CPU 时间超出限制时，XCPU 信号会被发送到进程	24
SIGXFSZ	生成 core 文件然后终止进程	当文件大小超过限制时，会产生 XFSZ 信号	25

15. Bash Shell 中的信号简述

当没有任何捕获时，一个交互式 Bash Shell 会忽略 SIGTERM 和 SIGQUIT 信号。由 Bash 运行的非内部命令会使用 Shell 从其父进程继承的信号处理程序。如果没有启用作业控制，异步执行的命令会忽略除了有这些信号处理程序之外的 SIGINT 和 SIGQUIT 信号。由于命令替换而运行的命令会忽略键盘产生的作业控制信号 SIGTTIN、SIGTTOU 和 SIGTSTP。

默认情况下，Shell 接收到 SIGHUP 信号后会退出。在退出之前，一个交互式的 Shell 会向所有的作业，不管是正在运行的还是已停止的，重新发送 SIGHUP 信号。对已停止的作业，Shell 还会发送 SIGCONT 信号以确保它能够接收到 SIGHUP 信号。

若要阻止 Shell 向某个特定的作业发送 SIGHUP 信号，可以使用内部命令 disown 将它从作业表中移除，或是用 “disown -h” 命令阻止 Shell 向特定的作业发送 SIGHUP 信号，但并不会将特定的作业从作业表中移除。

我们通过如下实例，来了解一下 disown 命令的作用：

```
#将 sleep 命令放在后台执行，休眠 30 秒

[c.biancheng.net]$ sleep 30 &

[1] 8052


#列出当前 Shell 下所有作业的信息

[c.biancheng.net]$ jobs -l

[1]+ 8052 Running      sleep 30 &


#将作业 1 从作业表中移除

[c.biancheng.net]$ disown %1


#再次列出当前 Shell 下所有作业的信息

[c.biancheng.net]$ jobs -l


#查找 sleep 进程

[c.biancheng.net]$ ps -ef | grep sleep

mozhayan 8052 8092 consl 11:28:21 /usr/bin/sleep


#打印当前 Shell 的进程号

[c.biancheng.net]$ echo $$

8092
```

在上述实例中，我们首先将命令 “sleep 30” 放在后台运行，此时，我们使用命令 “jobs -l” 可以看到作业表中有一个正在运行的作业，然后，我们使用命令 “disown %1” 将作业 1 从作业表中移除，再使用命令 “jobs -l” 会看到作业表中已经没有了作业，但是我们发现其实 “sleep 30” 这个命令的进程仍然存在。此时，Shell 若接收到 SIGHUP 信号，它就不会向作业 1 重新发送 SIGHUP 信号，此时如果我们退出 Shell，这个作业仍将继续运行，而不会被终止。

我们再来看一下命令 “disown -h” 的用途：

#将 sleep 命令放在后台执行，休眠 30 秒

```
[c.biancheng.net]$ sleep 30 &
```

```
[1] 3184
```

#列出当前 Shell 下所有作业的信息

```
[c.biancheng.net]$ jobs -l
```

```
[1]+ 3184 Running    sleep 30 &
```

#阻止 Shell 向作业 1 发送 SIGHUP 信号

```
[c.biancheng.net]$ disown -h %1
```

```
[c.biancheng.net]$ jobs -l
```

```
[1]+ 3184 Running    sleep 30 &
```

我们看到，在执行了命令“disown -h %1”后，作业 1 并没有从作业表中移除，但它已经被标记，所以即使 Shell 收到 SIGHUP 信号也不会向此作业发送 SIGHUP 信号。因此，如果此时我们退出 Shell，这个作业也仍将继续运行，而不会被终止。

注意：如果使用内部命令 shopt 打开了 Shell 的 huponexit 选项，当一个交互式的登录 Shell 退出时，会向所有的作业发送 SIGHUP 信号。

16. Linux 进程简明教程

进程是 Linux 操作系统中最重要的基本概念之一，这一节我们将了解学习 Linux 进程的一些基础知识。

进程是运行在 Linux 中的程序的一个实例。这是一个你之前就可能已经听说过的基本定义。

当你在 Linux 系统中执行一个程序时，系统会为这个程序创建特定的环境。这个环境包含系统运行这个程序所需的任何东西。

每当你在 Linux 中执行一个命令，它都会创建，或启动一个新的进程。比如，当你尝试运行命令“ls -l”来列出目录的内容时，你就启动了一个进程。如果有两个终端窗口显示在屏幕上，那么你可能运行了两次同样的终端程序，这时会有两个终端进程。

每个终端窗口可能都运行了一个 Shell，每个运行的 Shell 都分别是一个进程。当你从 Shell 调用一个命令时，对应的程序就会在一个新进程中执行，当这个程序的进程执行完成后，Shell 的进程将恢复运行。

操作系统通过被称为 PID 或进程 ID 的数字编码来追踪进程。系统中的每一个进程都有一个唯一的 PID。

现在我们通过一个实例来了解 Linux 中的进程。我们在 Shell 命令行下执行如下命令：

```
[c.biancheng.net]$ sleep 10 &

[1] 3324
```

因为程序会等待 10 秒，所以我们快速地在当前 Shell 上查找任何进程名为 sleep 的进程：

```
[c.biancheng.net]$ ps -ef | grep sleep

mozhiyan 3324 5712 cons1 17:11:46 /usr/bin/sleep
```

我们看到进程名为 /usr/bin/sleep 的进程正运行在系统中（其 PID 与我们在上一命令中得到的 PID 相同）。

现在，我们尝试并行地从 3 个不同的终端窗口运行上述的 sleep 命令，上述命令的输出将类似如下所示：

```
[c.biancheng.net]$ ps -ef | grep sleep

mozhiyan 896 5712 cons1 17:16:51 /usr/bin/sleep

mozhiyan 5924 5712 cons1 17:16:52 /usr/bin/sleep

mozhiyan 2424 5712 cons1 17:16:50 /usr/bin/sleep
```

我们看到 sleep 程序的每一个实例都创建了一个单独的进程。

每个 Linux 进程还有另一个 ID 号码，即父进程的 ID(ppid)。系统中的每一个用户进程都有一个父进程。

命令“ps -f”就会列出进程的 PID 和 PPID。此命令的输出类似如下所示：

```
[c.biancheng.net]$ ps -f

  UID      PID  PPID   TTY      STIME   COMMAND
mozhiyan  4124    228  cons0   21:37:09  /usr/bin/ps
mozhiyan   228     1   cons0   21:32:23  /usr/bin/bash
```

你在 Shell 命令行提示符下运行的命令都把当前 Shell 的进程作为父进程。例如，你在 Shell 命令行提示符下输入 ls 命令，Shell 将执行 ls 命令，此时 Linux 内核会复制 Shell 的内存页，然后执行 ls 命令。

在 Unix 中，每一个进程是使用 fork 和 exec 方法创建的。然而，这种方法会导致系统资源的损耗。

在 Linux 中，fork 方法是使用写时拷贝内存页实现的，所以它导致的仅是时间和复制父进程的内存页表所需的内存的损失，并且会为子

进程创建一个唯一的任务结构。

写时拷贝模式在创建新进程时避免了创建不必要的结构拷贝。例如，用户在 Shell 命令行提示符下输出 ls 命令，Linux 内核将会创建一个 Shell 的子进程，即 Shell 的进程是父进程，而 ls 命令的进程是子进程，ls 命令的进程会指向与此 Shell 相同的内存页，然后子进程使用写时拷贝技术执行 ls 命令。

前台进程和后台进程

当你启动一个进程时（运行一个命令），可以如下两种方式运行该进程：

- 前台进程
- 后台进程

默认情况下，你启动的每一个进程都是运行在前台的。它从键盘获取输入并发送它的输出到屏幕。

当一个进程运行在前台时，我们不能在同一命令行提示符下运行任何其他命令（启动任何其他进程），因为在程序结束它的进程之前命令行提示符不可用。

启动一个后台进程最简羊的方法是添加一个控制操作符 “&” 到命令的结尾。例如，如下命令将启动一个后台进程：

```
[c.biancheng.net]$ sleep 10 &

[1] 5720
```

现在 sleep 命令被放在后台运行。当 Bash 在后台启动一个作业时，它会打印一行内容显示作业编号（[1]）和进程号（PID-5720）。当作业完成时，作业会发送类似如下的信息到终端程序，来显示此作业已完成，其内容类似如下所示：

```
[1]+ Done  sleep 10
```

将进程放在后台运行的好处是：你可以继续运行其他命令，而不需要等待此进程运行完成再运行其他命令。

进程的状态

每个 Linux 进程都有它自己的生命周期，比如，创建、执行、结束和清除。每个进程也都有各自的状态，显示进程中当前正发生什么。

进程可以有如下几种状态：

- D（不可中断休眠状态）——进程正在休眠并且不能恢复，直到一个事件发生为止。
- R（运行状态）——进程正在运行。
- S（休眠状态）——进程没有在运行，而在等待一个事件或是信号。
- T（停止状态）——进程被信号停止，比如，信号 SIGINT 或 SIGSTOP。
- Z（僵死状态）——标记为 <defunct> 的进程是僵死的进程，它们之所以残留是因为它们的父进程适当地销毁它们。如果父进程退出，这些进程将被 init 进程销毁。

若要查看指定进程的状态，可以使用如下命令：

```
ps -C processName -o pid=,cmd,stat
```

例如：

```
[c.biancheng.net]$ ps -C sleep -o pid=,cmd,stat
```

	CMD	STAT
9434	sleep 20	S

17. Linux 使用什么命令查看进程

通过前面章节的一些实例的学习，想必你已经知道了使用 `ps` 命令可以查看进程的信息，但除了 `ps` 命令，我们还可以使用 `pstree` 命令和 `pgrep` 命令查看当前进程的信息。

使用 `ps` 命令，可以查看当前的进程。默认情况下，`ps` 命令只会输出当前用户并且是当前终端（比如，当前 Shell）下调用的进程的信息。其输出将类似如下所示：

```
[c.biancheng.net]$ ps

PID  TTY      TIME    CMD
4380 pts/0    00:00:00  bash
4414 pts/0    00:00:00  ps
```

我们从上面的输出中可以看到，默认情况下，`ps` 命令会显示进程 ID(PID)、与进程关联的终端（TTY）、格式为“[dd-]hh:mm:ss”的进程累积 CPU 时间（TIME），以及可执行文件的名称（CMD）。并且，输出内容默认是不排序的。

使用标准语法显示系统中的每个进程：

```
[c.biancheng.net]$ ps -ef | head -2

UID    PID  PPID  C  STIME  TTY  TIME    CMD
root    1    0    0  Jan14   ?   00:00:02  init [5]
```

使用 BSD 语法显示系统中的每个进程：

```
[c.biancheng.net]$ ps aux | head -2

USER    PID  %CPU  %MEM  VSZ   RSS TTY  STAT  START   TIME    COMMAND
root     1    0.0   0.0   2160  648  ?    Ss    Jan14   0:02    init [5]
```

使用 BSD 样式选项会增加进程状态（STAT）等信息作为默认显示，你也可以使用 `PS_FORMAT` 环境变量重写默认的输出格式。

查看系统中 `httpd` 进程的信息：

```
ps aux | grep httpd
```

使用 `pstree` 命令，可以显示进程树的信息：

```
[c.biancheng.net]$ pstree

init--acpid
    |--atd
    |--auditd--audispd---{audispd}
    |   |--{auditd}
    |   |--automount---4*[{automount}]
    |   |--avahi-daemon---avahi-daemon
    |   |--crond---5*[crond--m.j.sh]
    |   |--sendmail]
    |--cupsd
```

```
|dbus-daemon---{dbus-daemon}

|-events/0

|-events/1

|-gam_server

|-gpm

|-hald---hald-runner-+-hald-addon-acpi

|                               |-hald-addon-keyb

|                               `--hald-addon-stor

|-hcid

|-hidd

|-hpiod

|-java-+-java---17*[{java}]

|      `--14*[{java}]

|-java-+-java---29*[{java}]

|      `--14*[{java}]

|-java-+-java---34*[{java}]

|      `--14*[{java}]

|-java---20*[{java}]

|-java---292*[{java}]

|-khelper

|-klogd

|-krfcommd

|-ksoftirqd/0

|-ksoftirqd/1

|-kthread-+-aio/0

|          |-aio/1

|          |-ata/0

|          |-ata/1

|          |-ata_aux

|          |-cqueue/0

|          |-cqueue/1
```



```
|      |-hd-audio0

|      |-kacpid

|      |-kauditd

|      |-kblockd/0

|      |-kblockd/1

|      |-khubd

|      |-khungtaskd

|      |-2*[kjournald]

|      |-kmpath_handlerd

|      |-kmpathd/0

|      |-kmpathd/1

|      |-kondemand/0

|      |-kondemand/1

|      |-kpsmoused

|      |-kseriod

|      |-ksnapd

|      |-kstriped

|      |-kswapd0

|      |-2*[pdflush]

|      |-rpciod/0

|      |-rpciod/1

|      |-scsi_eh_0

|      |-scsi_eh_1

|      |-scsi_eh_2

|      |-scsi_eh_3

|      |-scsi_eh_4

|      |-scsi_eh_5

|-loop0

|-mcstransd

|-migration/0

|-migration/1
```

```

|-6*[mingetty]

|-mj.sh---make---java---11*[{java}]

|-ntpd

|-pcscd---{pcscd}

|-portmap

|-python

|-restorecond

|-rpc.idmapd

|-rpc.statd

|-screen---bash---update.sh---cvs

|-sendmail---2*[sendmail]

|-sendmail

|-setroubleshootd---2*[{setroubleshootd}]

|-smartd

|-sshd+-sshd---bash---update_and_rest---cvs

|   |-sshd---bash---pstree

|   `--sshd---bash

|-start_derby.sh---java---45*[{java}]

|-surf---8*[{surf}]

|-syslogd

|-tomcat---sleep

|-udev

|-watchdog/0

|-watchdog/1

|-xfs

|-xinetd

`--yum-updatesd

```

`ps`命令以树形结构的形式显示系统中所有当前运行的进程的信息。此树形结构以指定的 PID 为根，若没有指定 PID，则以 init 进程为根。下面，我们看一个显示指定 PID 的进程树的例子：

```

[c.biancheng.net]$ ps -ef | grep httpd
httpd-11*[httpd]

```

上述输出内容的含义是，PID 是 4578 的 httpd 进程下有 11 个 httpd 子进程。在显示时，pstree 命令会将一样的分支合并到一个方括号中，并在方括号前显示重复的次数。

如果 pstree 命令指定的参数是用户名，那么就会显示以此用户的进程为根的所有进程树的信息。其显示内容将类似如下所示：

```
[c.biancheng.net]$ pstree mozhiyan

Xvnc

dbus-daemon

dbus-launch

dcopserver

gconfd-2

kded

kdeinit+-bt-applet

    |-esc+-esc---9*[{esc}]

    | `--esc---6*[{esc}]

    |-2*[kio_file]

    |-kio-media

    |-klauncher

    `--kwin

kdesktop

kicker

klipper

ksmserver

bash---pstree

start_kdeinit

xstartup---startkde---kwrapper
```

使用 pgrep 命令，可以基于名称或其他属性查找进程。

pgrep 命令会检查当前运行的进程，并列出与选择标准相匹配的进程的 ID。例如，查看 root 用户的 sshd 进程的 PID：

```
[c.biancheng.net]$ pgrep -u root sshd

2877

6572

18563
```

列出所有者是 root 和 daemon 的进程的 PID :

```
pgrep -u root,daemon
```

18.Shell 向进程发送信号 (kill、pkill 和 killall 命令)

我们可以使用键盘或 pkill 命令、kill 命令和 killall 命令向进程发送各种信号。

使用键盘发送信号

在 Bash Shell 下，我们可以使用键盘发送信号，如下表所示。

可以发送信号的组合键	
组合键	含 义
Ctrl+C	中断信号，发送 SIGINT 信号到运行在前台的进程。
Ctrl+Y	延时挂起信号，使运行的进程在尝试从终端读取输入时停止。控制权返回给 Shell，使用户可以将进程放在前台或后台，或杀掉该进程。
Ctrl+Z	挂起信号，发送 SIGTSTP 信号到运行的进程，由此将其停止，并将控制权返回给 Shell。

kill 命令发送信号

大多数主流的 Shell，包括 Bash，都有内置的 kill 命令。Linux 系统中，也有 kill 命令，即 /bin/kill。如果使用 /bin/kill，则系统可能会激活一些额外的选项，比如，杀掉不是你自己的进程，或指定进程名作为参数，类似于 pgrep 和 pkill 命令。不过两种 kill 命令默认都是发送 SIGTERM 信号。

当准备杀掉一个进程或一连串的进程时，我们的常识是从尝试发送最安全的信号开始，即 SIGTERM 信号。以这种方式，关心正常停止运行的程序，当它收到 SIGTERM 信号时，有机会按照已经设计好的流程执行，比如，清理和关闭打开的文件。

如果你发送一个 SIGKILL 信号到进程，你将消除进程先清理而后关闭的机会，而这可能会导致不幸的结果。但如果一个有序地终结不管用，那么发送 SIGINT 或 SIGKILL 信号就可能是唯一的方法了。例如，当一个前台进程使用 Ctrl+C 组合键杀不掉时，那最好就使用命令 “kill -9 PID” 了。

在前面的学习中我们已经了解，kill 命令可以发送多种信号到进程。特别有用的信号包括：

- SIGHUP (1)
- SIGINT (2)
- SIGKILL (9)
- SIGCONT (18)
- SIGSTOP (19)

在 Bash Shell 中，信号名或信号值都可作为 kill 命令的选项，而作业号或进程号则作为 kill 命令的参数。

实例 1

发送 SIGKILL 信号到 PID 是 123 的进程。

```
kill -9 123
```

或是：

```
kill -KILL 123
```

也可以是：

```
kill -SIGKILL 123
```

实例 2

使用 kill 命令终结一个作业。

```
#将 sleep 命令放在后台执行，休眠 30 秒
```

```
[c.biancheng.net]$ sleep 30 &
```

```
[1] 20551
```

```
#列出当前 Shell 下所有作业的信息
```

```
[c.biancheng.net]$ jobs -l
```

```
[1]+ 20551 Running      sleep 30 &
```

```
#终结作业 1
```

```
[c.biancheng.net]$ kill %1
```

```
[1]+ 20551 Terminated  sleep 30
```

```
#查看当前 Shell 下的作业的信息
```

```
[c.biancheng.net]$ jobs -l
```

killall 命令发送信号

killall 命令会发送信号到运行任何指定命令的所有进程。所以，当一个进程启动了多个实例时，使用 killall 命令来杀掉这些进程会更方便些。

注意：在生产环境中，若没有经验，使用 killall 命令之前请先测试该命令，因为在一些商业 Unix 系统中，它可能不像所期望的那样工作。

如果没有指定信号名，killall 命令会默认发送 SIGTERM 信号。例如，使用 killall 命令杀掉所有 firefox 进程：

```
killall firefox
```

发送 KILL 信号到 firefox 的进程：

```
killall -s SIGKILL firefox
```

pkill 命令发送信号

使用 pkill 命令，可以通过指定进程名、用户名、组名、终端、UID、EUID 和 GID 等属性来杀掉相应的进程。pkill 命令默认也是发送 SIGTERM 信号到进程。

实例 1

使用 pkill 命令杀掉所有用户的 firefox 进程。

```
pkill firefox
```

实例 2

强制杀掉用户 mozhiyan 的 firefox 进程。

```
kill -KILL -u mozhiyan firefox
```

实例 3

让 sshd 守护进程重新加载它的配置文件。

```
kill -HUP sshd
```

19. Linux Shell trap 命令：捕获信号

到目前为止，我们在本教程所见的脚本中还没有需要信号处理功能的，因为它们的内容相对比较简单，执行时间很短，而且不会创建临时文件。而对于较大的或者更复杂的脚本来说，如果脚本具有信号处理机制可能就比较有用了。

当我们设计一个大型复杂的脚本时，考虑到当脚本运行时出现用户退出或系统关机会发生什么是很重要的。当这样的事件发生时，一个

信号将会发送到所有受影响的进程。相应地，这些进程的程序可以采取一些措施以确保程序正常有序地终结。比如说，我们编写了一个会在执行时生成临时文件的脚本。在好的设计过程中，我们会让脚本在执行完成时删除这些临时文件。同样聪明的做法是，如果脚本接收到了指示程序将提前结束的信号，也应删除这些临时文件。

接下来，就让我们开始学习，如何在脚本中进行这些处理。

trap 命令

Bash Shell 的内部命令 `trap` 让我们可以在 Shell 脚本内捕获特定的信号并对它们进行处理。`trap` 命令的语法如下所示：

```
trap command signal [ signal ... ]
```

上述语法中，`command` 可以是一个脚本或是一个函数。`signal` 既可以用信号名，也可以用信号值指定。

你可以不指定任何参数，而直接使用 `trap` 命令，它将会打印与每个要捕获的信号相关联的命令的列表。

当 Shell 收到信号 `signal(s)` 时，`command` 将被读取和执行。比如，如果 `signal` 是 0 或 EXIT 时，`command` 会在 Shell 退出时被执行。如果 `signal` 是 DEBUG 时，`command` 会在每个命令后被执行。

`signal` 也可以被指定为 ERR，那么每当一个命令以非 0 状态退出时，`command` 就会被执行（注意，当非 0 退出状态来自一个 if 语句部分，或来自 while、until 循环时，`command` 不会被执行）。

下面我们通过几个简单的实例来学习 `trap` 命令的用法。

首先，我们定义一个变量 `FILE`：

```
[c.biancheng.net]$ FILE=`mktemp -u /tmp/testtrap.$$ XXXXXX`
```

这里使用 `mktemp` 命令创建一个临时文件；使用 `-u` 选项，表示并不真正创建文件，只是打印生成的文件名；“XXXXXX”表示生成 6 位随机字符。

然后，我们定义捕获错误信号：

```
[c.biancheng.net]$ trap "echo There exist some error!" ERR
```

查看已经定义的捕获：

```
[c.biancheng.net]$ trap
```

```
trap -- 'echo There exist some error!' ERR
```

此时，当我们尝试使用 `rm` 命令删除变量 `$FILE` 代表的并不存在的文件时，就会显示类似如下的错误信息：

```
[c.biancheng.net]$ rm $FILE
```

```
rm: cannot remove '/tmp/testtrap.8020.zafuo4': No such file or directory
```

```
There exist some error!
```

从上面的输出中我们看到，Shell 捕获到了文件 `/tmp/testtrap.8020.zafuo4` 不存在的这个错误信号，并执行了 `echo` 命令，显示了我们指定的错误信息。

当调试较大的脚本时，你可能想要赋予某个变量一个踪迹属性，并捕获变量的调试信息。通常，你可能只使用一个简单的赋值语句，比如，`VARIABLE=value`，来定义一个变量。若使用类似如下的语句替换上述的变量定义，可能会为你提供更有用的调试信息：

```
1.  #声明变量 VARIABLE，并赋予其踪迹属性
2.  declare -t VARIABLE=value
3.
4.  #捕获 DEBUG
5.  trap "echo VARIABLE is being used here." DEBUG
6.
7.  #脚本的余下部分
```

现在，我们创建一个名称为 `testtrap1.sh` 的脚本，其内容如下所示：

```
1.  #!/bin/bash
2.
3.  #捕获退出状态 0
4.  trap 'echo "Exit 0 signal detected..."' 0
5.
6.  #打印信息
7.  echo "This script is used for testing trap command."
8.
9.  #以状态（信号）0 退出此 Shell 脚本
10. exit 0
```

此脚本运行结果将类似如 f 所示：

```
[c.biancheng.net]$ bash ./testtrap1.sh
This script is used for testing trap command.
Exit 0 signal detected...
```

在上述的脚本中，`trap` 命令语句设置了一个当脚本以 0 状态退出时的捕获，所以当脚本以 0 状态退出时，会打印一条信息 “Exit 0 signal detected...”。

我们再创建一个名称为 `testtrap2.sh` 的脚本，其内容类似如下所示：

```
1.  #!/bin/bash
2.
3.  #捕获信号 SIGINT，然后打印相应信息
4.  trap "echo 'You hit control+C! I am ignoring you.'" SIGINT
5.
6.  #捕获信号 SIGTERM，然后打印相应信息
7.  trap "echo 'You tried to kill me! I am ignoring you.'" SIGTERM
```

```
8.  
9.  #循环 5 次  
10.  
11. for i in {1..5}; do  
12.     echo "Iteration $i of 5"  
13.     #暂停 5 秒  
14.     sleep 5  
15. done
```

当你运行上述脚本时，如果敲击 CTRL+C 组合键，将会中断 sleep 命令，进入下一次循环，并看到输出信息 “You hit control+C! I am ignoring you.”，但脚本 testtrap2.sh 并不会停止运行。此脚本的运行结果将类似如下所示：

```
[c.biancheng.net]$ bash ./testtrap2.sh  
Iteration 1 of 5  
You hit control+C! I am ignoring you.  
Iteration 2 of 5  
Iteration 3 of 5  
Iteration 4 of 5  
You hit control+C! I am ignoring you.  
Iteration 5 of 5
```

当将上述脚本放在后台运行时，如果我们同时在另一个终端窗口尝试使用 kill 命令终结此脚本，此脚本并不会被终结，而是会显示信息 “You tried to kill me! I am ignoring you.”，此脚本的运行结果将会类似如下所示：

```
[c.biancheng.net]$ sh ./testtrap2.sh &  
[1] 2320  
[c.biancheng.net]$ Iteration 1 of 5  
You tried to kill me! I am ignoring you.  
Iteration 2 of 5  
Iteration 3 of 5  
Iteration 4 of 5  
You tried to kill me! I am ignoring you.  
Iteration 5 of 5  
You tried to kill me! I am ignoring you.  
[1]+ Done  sh ./testtrap2.sh
```

有时，接收到一个信号后你可能不想对其做任何处理。比如，当你的脚本处理较大的文件时，你可能希望阻止一些错误地输入 Ctrl+C 或 Ctrl+\ 组合键的做法，并且希望它能执行完成而不被用户中断。这时就可以使用空字符串 "" 或 ' ' 作为 trap 的命令参数，那么 Shell 将忽略这些信号。其用法类似如下所示：

```
$ trap ' ' SIGHUP SIGINT [ signal... ]
```

20. Linux Shell trap 命令捕获信号实例演示

通过前面内容的学习，我们已经知道，信号多用于以友好的方式结束一个进程的执行，即允许进程在退出之前有机会做一些清理工作。然而，信号同样还可用于其他用途。例如，当终端窗口的大小改变时，在此窗口中运行的 Shell 都会接收到信号 SIGWINCH。通常，这个信号是被忽略的，但是，如果一个程序关心窗口大小的变化，它就可以捕获这个信号，并用特定的方式处理它。

注意：除 SIGKILL 信号以外，其他任何信号都可以被捕获并通过调用 C 语言函数 signal 处理。

接下来，我就以一个脚本为实例演示捕获并处理 SIGWINCH 信号。我们创建名为 sigwinch_handler.sh 的脚本，其内容如下所示：

```
1.  #!/bin/bash
2.
3.  #打印信息
4.  echo "Adjust the size of your window now."
5.
6.  #捕获 SIGWINCH 信号
7.  trap "echo Window size changed." SIGWINCH
8.
9.  #定义变量
10. COUNT COUNT=0
11.
12. #while 循环 30 次
13. while [ $COUNT -lt 30 ]; do
14.     #将 COUNT 变量的值加 1
15.     COUNT=$((COUNT + 1))
16.     #休眠 1 秒
17.     sleep 1
18. done
```

当上述的 Shell 脚本运行时，若改变了此脚本运行所在终端窗口的大小，脚本的进程就会收到 SIGWINCH 信号，从而调用 chwinsize 函数，以作出相应的处理。此脚本的运行结果将类似如下所示：

```
[c.biancheng.net]$ chmod +x sigwinch_handler.sh

[c.biancheng.net]$ ./sigwinch_handler.sh

Adjust the size of your window now.

Window size changed.

Window size changed.
```

我们通过上一节《[trap 命令](#)》的学习已经知道，在 trap 命令中可以调用函数来处理相应的信号。下面我们就以脚本 trapbg_clearup.sh 为例，来进一步学习如何使用 trap 语句调用函数来处理信号，其脚本内容如下所示：

```
1.  #!/bin/bash
2.
3.  #捕获 INT 和 QUIT 信号，如果收到这两个信号，则执行函数 my_exit 后退出
4.  trap 'my_exit; exit' SIGINT SIGQUIT
5.
6.  #捕获 HUP 信号
```

```
7.  trap 'echo Going down on a SIGHUP - signal 1, no exiting...; exit' SIGHUP
8.
9.  #定义 count 变量
10. count=0
11.
12. #创建临时文件
13. tmp_file=`mktemp /tmp/file.$$XXXXXX`
14.
15. #定义函数 my_exit
16. my_exit()
17. {
18.     echo "You hit Ctrl-C/Ctrl-\, now exiting..."
19.     #清除临时文件
20.     rm -f $tmp_file >& /dev/null
21. }
22.
23. #向临时文件写入信息
24. echo "Do someting..." > $tmp_file
25.
26. #执行无限 while 循环
27. while :
28. do
29.     #休眠 1 秒
30.     sleep 1
31.     #将 count 变量的值加 1
32.     count=$((expr $count + 1))
33.     #打印 count 变量的值
34.     echo $count
35. done
```

当上述脚本运行时，接收到 SIGINT 或 SIGQUIT 信号后会调用 my_exit 函数后退出（trap 命令列表中的 exit 命令），my_exit 函数会做一些清理临时文件的操作。我们运行此脚本，然后在另一个终端窗口中查看此脚本创建的临时文件：

```
[c.biancheng.net]$ ls -trl /tmp/ | tail -l
```

将会看到类似如下的文件信息：

```
-rw----- 1 mozhiyan mozhiyan 15 Feb 6 22:09 file.6668.RI6669
```

现在，在脚本运行的终端窗口，我们输入 Ctrl+C 或 Ctrl+\ 组合键来终结或退出此脚本，将会看到类似如下的信息：

```
[c.biancheng.net]$ ./trapbg_cleanup.sh

1

2

3

4

5

6

7

8

9

You hit Ctrl+C/Ctrl+, now exiting...
```

然后我们再查看一下脚本创建的临时文件是否已被清理：

```
[c.biancheng.net]$ ls -l /tmp/file.6668.RI6669

ls: /tmp/file.6668.RI6669: No such file or directory
```

当脚本运行在后台时，同样可以捕获信号。我们将上例中的脚本 trapbg_cleanup.sh 放在后台运行：

```
[c.biancheng.net]$ ./trapbg_cleanup.sh &

[1] 16957

[c.biancheng.net]$ 1

2

3
```

现在从另一个终端窗口，发送 HUP 信号来杀掉这个运行脚本的进程：

```
[c.biancheng.net]$ kill -1 16957
```

现在，在脚本运行的终端窗口，将看到类似如下的信息：

```
[c.biancheng.net]$ ./trapbg_cleanup.sh &

[1] 16957
```

```
[c.biancheng.net]$ 1

2

3

4

5

6

7

8

9

10

Going down on a SIGHUP - signal 1, now exiting...

[1]+  Done      ./trapbg_clearup.sh
```

LINENO 和 BASH_COMMAND 变量

Bash Shell 中有两个内部变量可以方便地在处理信号时，为我们提供更多的与脚本终结相关的信息。这两个变量分别是 LINENO 和 BASH_COMMAND。BASH_COMMAND 是 Bash 中特有的。这两个变量分别用于报告脚本当前执行的行号和脚本当前运行的命令。

下面，我们以脚本 trap_report.sh 为实例，学习如何在脚本中使用变量 LINENO 和 BASH_COMMAND 在脚本终结时为我们提供更多的错误信息，其脚本内容类似如下所示：

```
1.  #!/bin/bash
2.  #捕获 SIGHUP、SIGINT 和 SIGQUIT 信号。如果收到这些信号，将执行函数 my_exit 后退出
3.  trap 'my_exit $LINENO $BASH_COMMAND; exit' SIGHUP SIGINT SIGQUIT
4.
5.  #函数 my_exit
6.  my_exit()
7.  {
8.      #打印脚本名称，及信号被捕获时所运行的命令和行号
9.      echo "$(basename $0) caught error on line : $1 command was: $2"
10.
11.     #将信息记录到系统日志中
12.     logger -p notice "script: $(basename $0) was terminated: line: $1, command was $2"
13.
14.     #其他一些清理命令
15. }
```

```
16.
17. #执行无限 while 循环
18. while :
19. do
20.     #休眠 1 秒
21.     sleep 1
22.     #将变量 count 的值加 1
23.     count=$(expr $count + 1)
24.     #打印 count 变量的值
25.     echo $count
26. done
```

当上述脚本运行时，向脚本发送 SIGHUP、SIGINT 和 SIGQUIT 信号后，脚本将会调用 my_exit 函数，此函数将解析参数 \$(LINENO) 和 \$2(BASH_COMMAND)，显示信号被捕获时脚本所运行的命令及其行号，同样 logger 语句会记录信息到日志文件 /var/log/messages 中。如果需要，还可以在此函数中执行一些清理命令，然后脚本将会退出（trap 命令列表中的 exit 命令）。

此脚本的运行结果将会类似如下所示：

```
[c.biancheng.net]$ ./trap_report.sh

1
2
3
4
5

trap_report.sh caught error on line : 34 command was: sleep
```

在 /var/log/messages 文件中，将会看到一条类似如下的记录：

```
Feb 7 16:48:13 localhost mozhiyan: script: trap_report.sh was terminated: line: 34, command was sleep
```

我们在上一节《[trap 命令](#)》中已经学习了，使用 trap 语句可以忽略信号。你也同样可以在脚本的一部分中忽略某些信号，然后，当你希望捕获这些信号时，可以重新定义它们来采取一些行动。我们以脚本 trapoff_on.sh 为例，在此脚本中我们将忽略信号 SIGINT 和 SIGQUIT，直到 sleep 命令结束运行后为止。然后当下一个 sleep 命令开始时，如果接收到终结信号，trap 语句将采取相应的行动。

其脚本的内容如下所示：

```
1. #!/bin/bash
2.
3. #忽略 SIGINT 和 SIGQUIT 信号
4. trap ' ' SIGINT SIGQUIT
```

```
5.
6.  #打印提示信息
7.  echo "You cannot terminate using ctrl+c or ctrl+\!"
8.
9.  #休眠 10 秒
10. sleep 10
11.
12. #重新捕获 SIGINT 和 SIGQUIT 信号。如果捕获到这两个信号，则打印信息后退出
13. #现在可以中断脚本了
14. trap 'echo Terminated!; exit' SIGINT SIGQUIT
15.
16. #打印提示信息
17. echo "OK! You can now terminate me using those keystrokes"
18.
19. #休眠 10 秒
20. sleep 10
```

此脚本的运行结果将类似如下所示：

```
[c.biancheng.net]$ chmod +x trapoff_on.sh

[c.biancheng.net]$ ./trapoff_on.sh

You cannot terminate using ctrl+c or ctrl+\!

OK! You can now terminate me using those keystrokes.

Terminated!
```

21. Linux Shell 移除（重置）信号捕获

如果我们在脚本中应用了捕获，我们通常会在脚本的结尾处，将接收到信号时的行为处理重置为默认模式。重置（移除）捕获的语法如下所示：

```
$ trap - signal [ signal ... ]
```


从上述语法中可以看出，使用破折号作为 trap 语句的命令参数，就可以移除信号的捕获。

下面，我们以脚本 trap_reset.sh 为例，来学习如何在脚本中移除先前定义的捕获。其脚本的内容类似如下所示：

```
1.  #!/bin/bash
2.
3.  #定义函数 cleanup
4.  function cleanup {
5.      #如果变量 msgfile 所指定的文件存在
6.      if [[ -e $msgfile ]]; then
7.          #将文件重命名（或移除）
8.          mv $msgfile $msgfile.dead
9.      fi
10.
11.      exit
12.  }
13.
14.  #捕获 INT 和 TERM 信号
15.  trap cleanup INT TERM
16.
17.  #创建一个临时文件
18.  msgfile=`mktemp /tmp/testtrap.$$ XXXXXX`
19.
20.  #通过命令行向此临时文件中写入内容
21.  cat > $msgfile
22.
23.  #接下来，发送临时文件的内容到指定的邮件地址，你自己完善此部分代码
24.  #send the contents of $msgfile to the specified mail address...
25.
26.  #删除临时文件
27.  rm $msgfile
28.
29.  #移除信号 INT 和 TERM 的捕获
30.  trap - INT TERM
```

上述脚本中，在用户已经完成了发送邮件的操作之后，临时文件会被删除。这时，因为已经不再需要清理操作，我们可以重置信号的捕获到默认状态，所以我们在脚本的最后一行重置了 INT 和 TERM 信号的捕获。

22. 关于 Linux Shell 中进程、信号和捕获的总结

下面我们总结一下前面几节学到的关于进程、信号和捕获的主要知识。

在 Linux 系统和其他类 Unix 或 Unix 操作系统中，信号被用于进程间的通信。

信号是一个发送到某个进程或同一进程中的特定线程的异步通知，用于通知发生的一个事件。

在 Linux 中，信号在处理异常和中断方面，扮演了极其重要的角色。

当一个事件发生时，会产生一个信号，然后内核会将事件传递到接收的进程。

运行在用户模式下的进程会接收信号。如果接收的进程正运行在内核模式，那么信号的执行只有在该进程返回到用户模式时才会开始。

当进程收到一个信号时，可能会发生以下 3 种情况：

- 进程可能会忽略此信号。有些信号不能被忽略，而有些没有默认行为的信号，默认会被忽略。
- 进程可能会捕获此信号，并执行一个被称为信号处理器的特殊函数。
- 进程可能会执行信号的默认行为。例如，信号 15(SIGTERM) 的默认行为是结束进程。

在 Shell 命令行提示符下，输入 “kill -l” 命令，可以显示所有信号的信号值和相应的信号名。

由 Bash Shell 运行的非内部命令会使用 Shell 从其父进程继承的信号处理程序。

默认情况下，Shell 接收到 SIGHUP 信号后会退出。在退出之前，一个交互式的 Shell 会向所有的作业，不管是正在运行的还是已停止的，重新发送 SIGHUP 信号。

若要阻止 Shell 向某个特定的作业发送 SIGHUP 信号，可以使用内部命令 disown 将它从作业表中移除，或是用 “disown -h” 命令阻止 Shell 向特定的作业发送 SIGHUP 信号，但并不会将特定的作业从作业表中移除。

进程是运行在 Linux 中的程序的一个实例。

每当你在 Linux 中执行一个命令，它都会创建或启动一个新的进程。

有两种运行方式的进程：前台进程和后台进程。

进程可以有 5 种状态：不可中断休眠状态 (D)、运行状态 (R)、休眠状态 (S)、停止状态 (T) 和僵死状态 (Z)。

使用 ps 命令，可以查看当前的进程；使用 pstree 命令，可以显示进程树的信息；使用 pgrep 命令，可以基于名称或其他属性查找进程。

当准备杀掉一个进程或一连串的进程时，我们的常识是从尝试发送最安全的信号开始，即 SIGTERM 信号。

如果发送一个 SIGKILL 信号到进程，将消除进程先清理而后关闭的机会，这可能导致不幸的结果。但如果一个有序地终结不管用，那么发送 SIGINT 或 SIGKILL 信号就可能是唯一的方法了。

killall 命令会发送信号到运行任何指定命令的所有进程。

使用 pkill 命令，可以通过指定进程名、用户名、组名、终端、UID、EUID 和 GID 等属性来杀掉相应的进程。pkill 命令默认也是发送 SIGTERM 信号到进程。

Bash 的内部命令 trap，让我们可以在 Shell 脚本内捕获特定的信号并对它们进行处理。

使用空字符串 "" 或 "作为 trap 的命令参数，可以让 Shell 忽略指定的信号。

除 SIGKILL 信号以外，其他任何信号都可以被捕获并通过调用 C 语言函数 signal 处理。

Bash 中有两个内部变量 LINENO 和 BASH_COMMAND 可以方便地在处理信号时，分别用于报告脚本当前执行的行号和脚本当前运行的命令。

使用破折号作为 trap 语句的命令参数，就可以移除指定信号的捕获。

23. Shell 模块化（把代码分散到多个脚本文件中）

所谓模块化，就是把代码分散到多个文件或者文件夹。对于大中型项目，模块化是必须的，否则会在一个文件中堆积成千上万行代码，这简直是一种灾难。

基本上所有的编程语言都支持模块化，以达到代码复用的效果，比如，Java 和 Python 中有 `import`，C/C++ 中有 `#include`。在 Shell 中，我们可以使用 `source` 命令来实现类似的效果。

在《[执行 Shell 脚本](#)》一节中我们已经提到了 source 命令，这里我们再来讲解一下。

source 命令的用法为：

```
source filename
```

也可以简写为：

```
. filename
```

两种写法的效果相同。对于第二种写法，注意点号`.`和文件名中间有一个空格。

source 是 [Shell 内置命令](#)的一种，它会读取 filename 文件中的代码，并依次执行所有语句。你也可以理解为，source 命令会强制执行脚本文件中的全部命令，而忽略脚本文件的权限。

实例

创建两个脚本文件 func.sh 和 main.sh：func.sh 中包含了若干函数，main.sh 是主文件，main.sh 中会包含 func.sh。

func.sh 文件内容：

```
1.  #计算所有参数的和
2.  function sum() {
3.      local total=0
4.
5.      for n in $@
6.      do
7.          ((total+=n))
8.      done
9.
10.     echo $total
11.     return 0
12. }
```

main.sh 文件内容：

```
1.  #!/bin/bash
2.
3.  source func.sh
4.
5.  echo $(sum 10 20 55 15)
```

运行 main.sh，输出结果为：

100

source 后边可以使用相对路径，也可以使用绝对路径，这里我们使用的是相对路径。

避免重复引入

熟悉 C/C++ 的读者都知道，C/C++ 中的头文件可以避免被重复引入；换句话说，即使被多次引入，效果也相当于一次引入。这并不是 #include 的功劳，而是我们在头文件中进行了特殊处理。

Shell source 命令和 C/C++ 中的 #include 类似，都没有避免重复引入的功能，只要你使用一次 source，它就引入一次脚本文件中的代码。

那么，在 Shell 中究竟该如何避免重复引入呢？

我们可以在模块中额外设置一个变量，使用 if 语句来检测这个变量是否存在，如果发现这个变量存在，就 return 出去。

这里需要强调一下 return 关键字。return 在 C++、C#、Java 等大部分编程语言中只能退出函数，除此以外再无他用；但是在 Shell 中，return 除了可以退出函数，还能退出由 source 命令引入的脚本文件。

所谓退出脚本文件，就是在被 source 引入的脚本文件（子文件）中，一旦遇到 return 关键字，后面的代码都不会再执行了，而是回到父脚本文件中继续执行 source 命令后面的代码。

return 只能退出由 source 命令引入的脚本文件，对其它引入脚本的方式无效。

下面我们通过一个实例来演示如何避免脚本文件被重复引入。本例会涉及到两个脚本文件，分别是主文件 main.sh 和 模块文件 module.sh。

模块文件 module.sh：

```
1.  if [ -n "$__MODULE_SH__" ]; then
2.      return
3.  fi
4.  __MODULE_SH__='module.sh'
5.
6.  echo "http://c.biancheng.net/shell/"
```

注意第一行代码，一定要是使用双引号把\$__MODULE_SH__包围起来，具体原因已经在《[Shell test](#)》一节中讲到。

主文件 main.sh：

```
1.  #!/bin/bash
2.
3.  source module.sh
4.  source module.sh
5.
6.      echo "here executed"
```

./表示当前文件，你也可以直接写作 source module.sh。

运行 main.sh，输出结果为：
http://c.biancheng.net/shell/
here executed

我们在 main.sh 中两次引入 module.sh，但是只执行了一次，说明第二次引入是无效的。

main.sh 中的最后一条 echo 语句产生了输出结果，说明 return 只是退出了子文件，对父文件没有影响。

第 4 章 Bash Shell 快捷键

1. Bash Shell 快捷键大全

开玩笑地说，我经常把 Unix 描述为“这个操作系统是为喜欢敲键盘的人们服务的。”当然，Unix 甚至还有一个命令行，这个事实是个确凿的证据，证明了我所说的话。但是命令行用户不喜欢敲入那么多字。那又为什么如此多的命令会有这样简短的命令名，像 cp，ls，mv，和 rm？

事实上，命令行最为珍视的目标之一就是懒惰；用最键次数少的击来完成最多的工作。另一个目标是你的手指永远不必离开键盘，永不触摸鼠标。

在这一节我们将看一下 Bash Shell 的特性，这些特性使键盘使用起来更加迅速，更加高效。

命令行编辑

Bash Shell 使用了一个名为 Readline 的库（共享的线程集合，可以被不同的程序使用），来实现命令行编辑。我们已经看到一些例子。我们知道，例如，箭头按键可以移动鼠标，此外还有许多特性。想想这些额外的工具，我们可以在工作中使用。学会所有的特性并不重要，但许多特性非常有帮助。选择自己需要的特性。

注意：下面一些按键组合（尤其使用 Alt 键的组合），可能会被 GUI 拦截来触发其它的功能。当使用虚拟控制台时，所有的按键组合都应该正确地工作。

移动光标

下表列出了移动光标所使用的按键。

光标移动快捷键	
按键	作用
Ctrl+a	移动光标到行首。
Ctrl+e	移动光标到行尾。
Ctrl+f	光标前移一个字符；和右箭头作用一样。
Ctrl+b	光标后移一个字符；和左箭头作用一样。
Alt+f	光标前移一个字。
Alt+b	光标后移一个字。
Ctrl+l	清空屏幕，移动光标到左上角。clear 命令完成同样的工作。

修改文本

下面这些快捷键用来在命令行中编辑字符。

文本编辑快捷键	
按键	作用
Ctrl+d	删除光标位置的字符。
Ctrl+t	光标位置的字符和光标前面的字符互换位置。
Alt+t	光标位置的字和其前面的字互换位置。
Alt+l	把从光标位置到字尾的字符转换成小写字母。
Alt+u	把从光标位置到字尾的字符转换成大写字母。

剪切和粘贴文本

Readline 的文档使用术语 `killring` 和 `yanking` 来指我们平常所说的剪切和粘贴。剪切下来的文本被存储在一个叫做剪切环 (`kill-ring`) 的缓冲区中。

剪切和粘贴快捷键	
按键	作用
Ctrl+k	剪切从光标位置到行尾的文本。
Ctrl+u	剪切从光标位置到行首的文本。
Alt+d	剪切从光标位置到词尾的文本。
Alt+Backspace	剪切从光标位置到词头的文本。如果光标在一个单词的开头，剪切前一个单词。
Ctrl+y	把剪切环中的文本粘贴到光标位置。

【冷知识】元键

如果你冒险进入到 Readline 的文档中，你会在 `bash` 手册页的 `READLINE` 段落，遇到一个术语“元键 (`meta key`)”。在当今的键盘上，这个元键是指 `Alt` 键，但并不总是这样。

回到昏暗的年代 (在 `PC` 之前 `Unix` 之后)，并不是每个人都有他们自己的计算机。他们可能有一个叫做终端的设备。一个终端是一种通信设备，它以一个文本显示屏幕和一个键盘作为其特色，它里面有足够的电子器件来显示文本字符和移动光标。它连接到 (通常通过串行电缆) 一个更大的计算机或者是一个大型计算机的通信网络。

有许多不同的终端产品商标，它们有着不同的键盘和特征显示集。因为它们都倾向于至少能理解 `ASCII`，所以软件开发者想要符合最低标准的可移植的应用程序。`Unix` 系统有一个非常精巧的方法来处理各种终端产品和它们不同的显示特征。

因为 `Readline` 程序的开发者们，不能确定一个专用多余的控制键的存在，他们发明了一个控制键，并把它做“元 (`meta`)”。然而在现代的键盘上，`Alt` 键作为元键来服务。如果你仍然在使用终端 (在 `Linux` 中，你仍然可以得到一个终端)，你也可以按下和释放 `Esc` 键来得到如控制 `Alt` 键一样的效果。

2. Bash Shell 命令自动补全功能

`shell` 能帮助你的另一种方式是通过一种叫做自动补全的机制。当你敲入一个命令时，按下 `tab` 键，自动补全就会发生。

让我们看一下这是怎样工作的。给出一个看起来像这样的主目录：

```
[c.biancheng.net]$ ls

Desktop ls-output.txt Pictures Templates Videos

....
```

试着输入下面的命令，但不要按下 Enter 键：

```
[c.biancheng.net]$ ls l
```

现在按下 tab 键：

```
[c.biancheng.net]$ ls ls-output.txt
```

看一下 shell 是怎样补全这一行的？让我们再试试另一个例子。这回，也不要按下 Enter：

```
[c.biancheng.net]$ ls D
```

按下 tab：

```
[c.biancheng.net]$ ls D
```

没有补全，只是嘟嘟响。因为“D”不止匹配目录中的一个条目。为了自动补全执行成功，你给它的“线索”必须不模棱两可。如果我们继续输入：

```
[c.biancheng.net]$ ls Do
```

然后按下 tab：

```
[c.biancheng.net]$ ls Documents
```

自动补全成功了。

这个实例展示了路径名自动补全，这是最常用的形式。自动补全也能对变量起作用（如果字的开头是一个“\$”），用户名字（单词以“ ”开始），命令（如果单词是一行的第一个单词），和主机名（如果单词的开头是“@”）。主机名自动补全只对包含在文件 /etc/hosts 中的主机名有效。

有一系列的快捷键与自动补全相关联：

命令自动补全快捷键	
按键	功能
Alt+?	显示可能的自动补全列表。在大多数系统中，你也可以完成这个通过按两次 tab 键，这会更容易些。
Alt+*	插入所有可能的自动补全。当你想要使用多个可能的匹配项时，这个很有帮助。

可编程自动补全

目前的 bash 版本有一个叫做可编程自动补全工具。可编程自动补全允许你（更可能是，你的发行版提供商）来加入额外的自动补全规则。通常需要加入对特定应用程序的支持，来完成这个任务。例如，有可能为一个命令的选项列表，或者一个应用程序支持的特殊文件类型加入自动补全。

默认情况下，Ubuntu 已经定义了一个相当大的规则集合。可编程自动补全是由 shell 函数实现的。

如果你感到好奇，试一下：

```
set | less
```

查看一下如果你能找到它们的话。默认情况下，并不是所有的发行版都包括它们。

3. Shell history : 历史命令

bash shell 维护着一个已经执行过的命令的历史列表。这个命令列表被保存在你主目录下，一个叫做 `.bash_history` 的文件里。这个 history 工具是个有用资源，因为它可以减少你敲键盘的次数，尤其当和命令行编辑联系起来时。

搜索历史命令

在任何时候，我们都可以浏览历史列表的内容，通过：

```
[c.biancheng.net]$ history | less
```

在默认情况下，bash 会存储你所输入的最后 500 个命令。在随后的章节里，我们会知道怎样调整这个数值。

比方说我们想要找到列出目录 /usr/bin 内容的命令。一种方法，我们可以这样做：

```
[c.biancheng.net]$ history | grep /usr/bin
```

比方说在我们的搜索结果之中，我们得到一行，包含了有趣的命令，像这样：

```
88 ls -l /usr/bin > ls-output.txt
```

数字“88”是这个命令在历史列表中的行号。随后在使用另一种展开类型时，叫做历史命令展开，我们会用到这个数字。我们可以这样做，来使用我们所发现的行：

```
[c.biancheng.net]$ !88
```

bash 会把“!88”展开成为历史列表中 88 行的内容。还有其它的历史命令展开形式，我们一会儿讨论它们。

bash 也具有按递增顺序来搜索历史列表的能力。这意味着随着字符的输入，我们可以告诉 bash 去搜索历史列表，每一个附加字符都进一步提炼我们的搜索。

启动递增搜索，输入 Ctrl+r，其后输入你要寻找的文本。当你找到它以后，你可以敲入 Enter 来执行命令，或者输入 Ctrl+j，从历史列表中复制这一行到当前命令行。再次输入 Ctrl+r，来找到下一个匹配项（向上移动历史列表）。

输入 Ctrl+g 或者 Ctrl+c，退出搜索，实际来体验一下：

```
[c.biancheng.net]$
```

首先输入 Ctrl+r：

```
(reverse-i-search)`':
```

提示符改变，显示我们正在执行反向递增搜索。搜索过程是“反向的”，因为我们按照从“现在”到过去某个时间段的顺序来搜寻。下一步，我们开始输入要查找的文本。在这个例子里是“/usr/bin”：

```
(reverse-i-search)`/usr/bin': ls -l /usr/bin > ls-output.txt
```

即刻，搜索返回我们需要的结果。我们可以执行这个命令，按下 Enter 键，或者我们可以复制这个命令到我们当前的命令行，来进一步编辑它，输入 Ctrl+j。

复制它，输入 Ctrl+j：

```
[c.biancheng.net]$ ls -l /usr/bin > ls-output.txt
```

我们的 shell 提示符重新出现，命令行加载完毕，正准备行动！下表列出了一些按键组合，这些按键用来操作历史列表。

历史命令快捷键	
按键	作用

Ctrl+p	移动到上一个历史条目。类似于上箭头按键。
Ctrl+n	移动到下一个历史条目。类似于下箭头按键。
Alt+<	移动到历史列表开头。
Alt+>	移动到历史列表结尾，即当前命令行。
Ctrl+r	反向递增搜索。从当前命令行开始，向上递增搜索。
Alt+p	反向搜索，不是递增顺序。输入要查找的字符串，然后按下 Enter，执行搜索。
Alt+n	向前搜索，非递增顺序。
Ctrl+o	执行历史列表中的当前项，并移到下一个。如果你想要执行历史列表中一系列的命令，这很方便。

历史命令展开

通过使用 `!` 字符，shell 为历史列表中的命令，提供了一个特殊的展开类型。我们已经知道一个感叹号，其后再加上一个数字，可以把来自历史列表中的命令插入到命令行中。还有一些其它的展开特性：

历史展开命令	
命令	作用
!!	重复最后一次执行的命令。可能按下上箭头按键和 enter 键更容易些。
! <i>number</i>	重复历史列表中第 <i>number</i> 行的命令。
! <i>string</i>	重复最近历史列表中，以这个字符串开头的命令。
! <i>?string</i>	重复最近历史列表中，包含这个字符串的命令。

应该小心谨慎地使用 “!*string*” 和 “!*?string*” 格式，除非你完全确信历史列表条目的内容。

在历史展开机制中，还有许多可利用的特点，但是这个题目已经太晦涩难懂了，如果我们再继续讨论的话，我们的头可能要爆炸了。bash 手册页的 HISTORY EXPANSION 部分详尽地讲述了所有要素。

扩展阅读

除了 bash 中的命令历史特性，许多 Linux 发行版包括一个叫做 script 的程序，这个程序可以记录整个 shell 会话，并把 shell 会话存在一个文件里面。这个命令的基本语法是：

```
script [file]
```

命令中的 file 是指用来存储 shell 会话记录的文件名。如果没有指定文件名，则使用文件 typescript。查看脚本的手册页，可以得到一个关于 script 程序选项和特点的完整列表。