第 3 章 Shell

学习目标

- 了解 Shell 的概念、分类与使用技巧
- 熟悉 Shell 中的变量
- 掌握 Shell 中的符号
- 掌握正则表达式
- 掌握 grep 命令的使用
- 熟悉 sed 命令的使用
- 熟悉 awk 的使用
- 掌握 Shell 脚本的语法格式

在 IT 环境维护中，为了提高工作效率，减少因手工操作出现的错误，人们常选择使用脚本处理大量重复性工作。Shell 是 Linux 系统中最常使用的脚本语言，使用 Shell 脚本可实现有针对性的自动化运维。本章将针对与编写 Shell 脚本相关的知识，包括 Shell 的语法、正则表达式、文本处理工具等进行讲解。

3.1 Shell 概述

在计算机中，用户是无法直接与硬件或内核交互的。用户一般通过应用程序发送指令给内核，内核在收到指令后分析用户需求，调度硬件资源来完成操作。在 Linux 系统中，这个应用程序就是 Shell，本节将针对 Shell 进行详细讲解。

3.1.1 Shell 的概念

Shell 是一种具备特殊功能的程序，处于用户与内核之间，提供用户与内核进行交互的接口。换言之，Shell 可接收用户输入的命令，将命令送入内核中执行。内核接收到用户的命令后调度硬件资源完成操作，再将结果返回给用户。Shell 与内核及用户间的关系如图 3-1 所示。

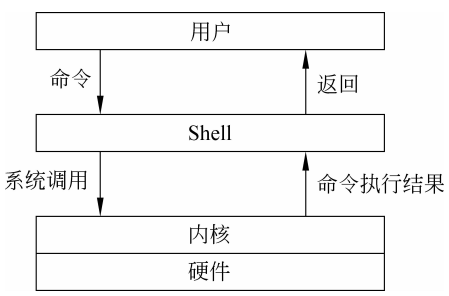

图 3-1 Shell 与内核及用户的关系

Shell 在帮助用户与内核完成交互的过程中还提供了解释功能：传递命令时，Shell 将命令解释为二进制形式；返回结果时，Shell 将结果解释为字符形式，因此 Shell 又被称为命令解释器。Shell 拥有内建的命令集，第 2 章中介绍的多种命

令，实际上都是 Shell 命令集中的命令。

Shell 也是一个解释型的程序设计语言，使用 Shell 语言编写的程序称为 Shell 脚本。Shell 脚本支持变量、数组和控制结构(如选择结构、循环结构等)，也支持 Shell 命令。Shell 编程语言简单易学，一旦掌握后它将是最得力的工具。

Shell 提供了两种方式以实现用户与内核的通信：交互式通信(Interactive)和非交互式通信(Shell Script)。交互式通信指用户输入一条命令，Shell 就解释执行一条命令，此种方式下用户输入的命令可以立即得到响应；非交互式通信指按照 Shell 语言规范编写程序并保存为文件，在需要时执行 Shell 文件，一次性执行文件中的所有命令。

3.1.2　Shell 的分类

Linux 中 Shell 种类很多，常见的 Shell 有 Bourne Shell(sh)、Bourne-Again Shell(bash)、C Shell(csh/tcsh)、Korn Shell(ksh)、Z Shell(zsh)几种。

1. Bourne Shell(sh)

sh 是 Linux/UNIX 操作系统最初使用的 Shell，在任一个操作系统上都可以使用。sh 在 Shell 编程方面非常优秀，但是在用户与内核的交互性方面不如其他几种 Shell。

2. C Shell(csh/tcsh)

csh 由以 William Joy 为代表的共计 47 位作者编写而成(tcsh 是 csh 的扩展)。因为 csh 的语法和 C 语言类似，所以 csh 被很多 C 程序员使用，这也是 csh 名称的由来。

csh 提供了友好的用户界面，并且增强了与用户的交互功能，如作业控制、命令行历史和别名等。虽然 csh 的功能很强大，但它的运行速度非常慢。

3. Korn Shell(ksh)

ksh 结合了 csh 的交互式特性，并融入了 sh 的语法。除此之外，ksh 还新增了一些功能，例如，数学计算、进程协作、行内编辑等。在 ksh 的基础上，又扩展出了 pdksh，它支持任务控制，可以在命令行上挂起、后台执行、唤醒或终止程序。

4. Bourne-Again Shell(bash)

bash 是 sh 的扩展，在 sh 的基础上增加了许多特性，而且它融合了 C Shell 和 ksh 的功能优势，如作业控制、命令行历史、支持任务控制等。相比 sh，bash 具有以下特色：

- 可以使用方向键查阅、输入、修改命令。
- 支持命令补齐功能。
- 自身包含帮助功能，用户只要在提示符下面键入 help 就可以得到相关的帮助信息。

bash 有很灵活和强大的编程接口，同时又有很友好的用户界面，是 Linux 操作系统默认的 Shell。

5. Z Shell(zsh)

zsh 是 Linux 操作系统中最大的 Shell，它包含 84 个内置命令。在完全兼容 bash 的同

时，zsh 还有很多性能方面的提升，例如，更高效、更好的自动补全功能等。但是 zsh 需要使用者手动安装，比较麻烦。一般的 Linux 操作系统都不会使用 zsh。

Linux 系统中的每种 Shell 都有各自的优势和不足，用户在使用时可根据具体情况酌情选择。

使用 cat 命令查看/etc/shells 文件中的内容可确定主机支持的 Shell 类型，具体操作与命令的输出结果如下：

```
[itheima@localhost ~]$ cat /etc/shells
/bin/sh
/bin/bash
/sbin/nologin
/usr/bin/sh
/usr/bin/bash
/usr/sbin/nologin
/bin/tcsh
/bin/csh
```

由输出结果可知，本机中有 sh、bash、tcsh 和 csh 4 种。用户还可以使用 echo 命令，通过打印环境变量 $SHELL（$SHELL 是一个环境变量，它记录了 Linux 当前用户所使用的 Shell 类型，关于环境变量将在 4.2.2 节进行讲解）的值来查看本机当前正在使用的 Shell，具体操作与命令的输出结果如下：

```
[itheima@localhost ~]$ echo $SHELL
/bin/bash
```

由输出结果可知，本机当前使用的是 bash，即系统默认的 Shell。接下来的章节中提到的 Shell，没有特殊说明，一般都指 bash。

用户使用的 Shell 是可以更改的，直接输入各类 Shell 的文件名就可以开启一个新的 Shell，以开启 csh 为例，具体操作如下：

```
[itheima@localhost ~]$/bin/csh
```

此命令启动了一个新的 Shell（csh），这个 Shell 在 bash 之后登录，称为 bash 的下级 Shell 或子 Shell。使用 echo $SHELL 只能显示用户登录的 Shell，无法显示子 Shell，用户可使用如下命令来显示系统中运行的所有 Shell，包括所有的子 Shell。

```
[itheima@localhost ~]$ ps
  PID TTY      TIME         CMD
 3943 pts/0    00:00:00     bash
 4201 pts/0    00:00:00     csh
 4217 pts/0    00:00:00     ps
```

在上述命令中，ps 用于显示系统中正在运行的进程，“|”是管道符，grep 是一个搜索命令工具，该命令的含义是搜索系统中运行的 sh 进程。管道符将在 3.1.3 节讲解，grep 命令将在 3.5 节讲解。由输出结果可知，子 Shell（csh）正在运行，使用 exit 命令可以退出这个子 Shell。

3.1.3　Shell 的使用技巧

Shell 有很多常用的使用技巧，对于初学者来说，掌握这些使用技巧对提升使用 Linux 的效率有很大帮助。接下来介绍几种常用的 Shell 使用技巧。

1. 自动补齐

Shell 具有命令行自动补齐的功能。利用 Tab 键可以根据输入的字符串自动查找匹配的命令、文件、目录等，如果匹配结果唯一，Shell 会自动补齐；如果有多个可以匹配的名称，按两次 Tab 键后 Shell 会列出所有匹配项。

例如，用户要从当前目录进入到/usr/src/kernels/3.10.0-514.el7.x86_64/目录，在进入目录时就可以利用自动补齐的功能快速切换这个目录。

```
[itheima@localhost ~]$cd /u<Tab>/sr<Tab>/ke<Tab>/3.10<Tab>
```

对以上命令进行分析，具体如下：

(1) /u＜Tab＞会自动补齐为/usr/。

(2) /usr/sr＜Tab＞会自动补齐为/usr/src/。在这一步，如果只写出一个 s 字符，则会出现 sbin/、share/、src/ 3 个目录供选择，因此示例中匹配以 sr 开头的字符串。

(3) /usr/src/ke＜Tab＞会自动补齐为/usr/src/kernels/。

(4) usr/src/kernels/3.10＜Tab＞会自动补齐为/usr/src/kernels/3.10.0-514.el7.x86_64/。

有了自动补齐功能，在碰到名字特别长的文件名时使用自动补齐功能非常方便，例如，要安装某一个软件，安装包的名字为 vsfrpd-2.0.2-10.el5-linux-x64.rpm，这时只要在命令行输入如下命令就可以自动识别安装包：

```
[root@localhost itheima]# rpm -ivh vsfrpd<Tab>
```

2. 查找命令行历史记录

Shell 提供了 4 种方法查找命令行历史记录，具体如下所示。

(1) 查找家目录下的.history 文件，Shell 在执行命令时将命令的操作记录保存在用户家目录下的.bash_history 文件中，通过这个文件可以查询 Shell 命令的执行历史。

(2) 使用 history 命令，Shell 提供了 history 命令用于查询命令行历史记录。

(3) 使用键盘上下方向键，通过键盘上的上下方向键可以逐条查询命令行历史记录。如果要重新执行不久前执行的命令，使用上下方向键查找更方便。

(4) 使用快捷键 Ctrl＋R，按下此快捷键，会出现 reverse-i-search 提示，输入之前执行过的命令，每当输入一个字符，终端都会滚动显示历史命令，当显示到想要查找的命令时直接按回车键就执行了该命令。不想查找时，按 Esc 键或方向键退出查找。

3. 命令替换

在进行 Shell 编程时经常会用到命令替换，命令替换有两种方式：反引号(`)和 $()符号。

反引号在 Esc 键下方，它的作用是将命令字符替换为命令的执行结果。例如，替换 ls 命令，输出当前目录下的文件和目录。

```
[itheima@localhost ~]$ echo `ls`
公共 模板 视频 图片 文档 下载 音乐 桌面
```

以上命令的作用和直接执行 ls 命令相同，同样的功能也可以使用 $()符号来实现。

```
[itheima@localhost ~]$ echo $(ls)
公共 模板 视频 图片 文档 下载 音乐 桌面
```

这两种方式在替换时实现机制略有不同，但最终结果都是将命令替换成命令的执行结果。

4. I/O 重定向

Shell 默认可接收用户输入到终端的命令，并在执行后将错误信息和输出结果打印到终端，但在实际应用中，并非任何情境下我们都希望 Shell 执行这项默认操作，此时，可通过 Linux 系统提供的一些功能，改变 Shell 获取信息和输出信息的方向。

Linux 系统中的输入输出分为以下 3 类：

(1) 标准输入(STDIN)，标准输入文件的描述符是 0，默认的设备是键盘，命令在执行时从标准输入文件中读取需要的数据。

(2) 标准输出(STDOUT)，标准输出文件的描述符是 1，默认的设备是显示器，命令执行后其输出结果会被发送到标准输出文件。

(3) 标准错误(STDERR)，标准错误文件的描述符是 2，默认的设备是显示器，命令执行时产生的错误信息会被发送到标准错误文件。

Linux 允许对以上 3 种资源重定向。所谓重定向，即使用用户指定的文件，而非默认资源(键盘、显示器)来获取或接收信息。

下面分别针对上述 3 种文件，讲解其重定向的方法。

1) 输入重定向

输入重定向运算符“<”可以指定其右值为左值的输入，具体格式如下：

```
命令 0<文件名
```

其中 0 是标准输入文件标识符，可以被省略。以 wc 命令为例，将 file 中的内容作为命令 wc 的输入，统计文件中的行数。

```
[itheima@localhost ~]$wc -l <file
```

2）输出重定向

输出重定向运算符“>”可以将其右值作为左值的输出端，其格式如下：

```
命令 1>文件名
```

其中 1 是标准输出文件标识符，也可以被省略。以 cat 命令为例，将/etc/passwd 文件中的内容重定向到 file 文件的具体操作如下：

```
[itheima@localhost ~]$cat /etc/passwd >file
```

以上命令会将“cat /etc/passwd”的执行结果以覆盖的形式输出到 file 文件中。若想保留 file 文件的内容，可以使用“>>”运算符，以追加的形式将结果输出到 file 文件。

```
[itheima@localhost ~]$cat /etc/passwd >>file
```

3）错误重定向

错误重定向也使用“>”符号，其格式如下：

```
命令 2>文件名
```

其中 2 是标准错误文件描述符，不可以被省略。例如，打开一个不存在的文件会报出错误信息，如果使用错误重定向可以将错误信息输出到文件中。下面打开不存在的文件 cfile，将其错误信息重定向到文件 newfile 中，命令如下所示：

```
[itheima@localhost ~]$cat c 2>newfile
```

执行该命令，屏幕上不会输出错误信息，错误信息被重定向到 newfile 文件中了。查看 newfile 文件，结果如下所示：

```
[itheima@localhost ~]$ cat newfile
cat: c: 没有那个文件或目录
```

同样，错误重定向也可以使用运算符“>>”，以追加的方式将错误输出到指定的文件。

```
[itheima@localhost ~]$cat cfile 2>>file
```

5. 管道

管道符号为“|”，它可以将多个简单的命令连接起来，使一个命令的输出，作为另外一个命令的输入，由此来完成更加复杂的功能。其格式如下：

```
命令 1 | 命令 2 |…| 命令 n
```

以 ls 命令和 grep 命令的组合为例来演示管道符的使用，具体如下所示：

```
[itheima@localhost ~]$ls -l /etc | grep init
```

在以上示例中，管道符“|”连接了 ls 命令和 grep 命令，其作用为：输出 etc 目录下文件

信息包含 init 关键字的行。若不使用管道，则必须使用两步来完成这个任务，具体步骤如下：

```
[itheima@localhost ~]$ls -l /etc >tmp.txt
[itheima@localhost ~]$grep init <tmp.txt
```

3.2 Shell 中的变量

Shell 提供了一些变量，这些变量可以保存路径名、文件名或者数值等。Shell 中常用的变量有 4 种：本地变量、环境变量、位置变量和特殊变量，本节将针对这 4 种变量进行详细讲解。

3.2.1 本地变量

本地变量相当于 C 语言中的局部变量，它只在本 Shell 中有效，如果 Shell 退出，本地变量将被销毁。本地变量的定义格式如下所示：

```
NAME=value
```

NAME 是变量名，value 是赋给变量的值。如果 value 没有指定，变量将被赋值为空字符串。在使用变量时，要在变量前面加“$”符号。例如，定义一个变量 NAME，其值为 Tom，在输出时，要以 $NAME 的形式输出。

```
[itheima@localhost ~]$ NAME=Tom
[itheima@localhost ~]$ echo $NAME
Tom
[itheima@localhost ~]$ echo My name is $NAME
My name is Tom
```

Shell 支持连续输出多个变量的值，例如，再定义一个变量 AGE，其值为 23，然后同时输出变量 NAME 与 AGE。

```
[itheima@localhost ~]$ AGE=23
[itheima@localhost ~]$ echo $NAME $AGE
Tom 23
```

在定义本地变量时，还可以使用 read 命令从标准输入中读取变量值，其中 read 的-p 选项可以设置输入提示信息。

```
[itheima@localhost ~]$ read -p "please input an int number: " INTNUM
please input an int number: 100
[itheima@localhost ~]$ echo $INTNUM
100
```

删除所定义的变量，可以使用 unset 命令。

```
[itheima@localhost ~]$ unset AGE
[itheima@localhost ~]$ echo $AGE
```

调用 unset 命令删除 AGE 变量后，再输出该变量时值不再显示，仅输出一个空白行，表明这个变量已被删除。

3.2.2　环境变量

环境变量是 Shell 中非常重要的一个变量，用于初始化 Shell 的启动环境。下面将针对 Shell 中的环境变量进行详细的讲解。

1. 环境变量的定义与清除

环境变量在 Shell 编程和 Linux 系统管理方面都起着非常重要的作用，它一般用来存储路径列表，这些路径可用于搜索可执行文件、库文件等。环境变量定义格式如下所示：

```
export ENVIRON-VARIABLE=value
```

环境变量必须要使用 export 关键字导出，export 关键字的作用是声明此变量为环境变量。例如，定义 APPSPATH 变量并赋值为/usr/local，然后利用 export 将 APPSPATH 声明为环境变量。

```
[itheima@localhost ~]$ export APPSPATH=/usr/local
[itheima@localhost ~]$ echo $APPSPATH
/usr/local
```

在命令行中使用 export 定义的环境变量只在当前 Shell 与子 Shell 中有效，Shell 重启后这些环境变量将丢失。

使用 env 命令可以查看所有的环境变量，包括用户自定义的环境变量。

```
[itheima@localhost ~]$env
```

删除环境变量和删除本地变量的方式相同，也是调用 unset 命令。

```
[itheima@localhost ~]$ unset APPSPATH
[itheima@localhost ~]$ echo $APPSPATH
```

2. 几个重要的环境变量

bash 中预设了很多环境变量，其中有几个比较重要的环境变量，Linux 系统及诸多应用程序的正常运行都依赖它们。

1) PATH

PATH 是 Linux 中一个极为重要的环境变量，它用于帮助 Shell 找到用户所输入的命令。用户输入的每个命令都是一个可执行程序，计算机执行这个程序以实现这个命令的功能。可执行程序存在于不同目录下，PATH 变量就记录了这一系列的目录列表。

输出 PATH 变量的值，结果如下所示：

```
[itheima@localhost ~]$ echo $PATH
/usr/local/bin: /usr/local/sbin: /usr/bin: /usr/sbin: /bin: /sbin: /root/bin
```

由输出结果可知，PATH 中包含了多个目录，它们之间用冒号分隔，这些目录中保存着命令的可执行程序，例如，输入 ls 命令，PATH 就会去这些目录中查找 ls 命令的可执行程序，首先在/usr/local/bin 目录查找，找到就执行该命令；没找到就继续查找下一个目录，直到找到为止。如果 PATH 值存储的目录列表中的所有目录都不包含相应文件，则 Shell 会提示“未找到命令……”。

PATH 变量的值可以被修改，但在修改时要注意不可以直接赋新值，否则 PATH 现有的值将会被覆盖。如果要在 PATH 中添加新目录，可以使用下面的命令格式：

```
PATH=$PATH: /new directory
```

以上格式中 $PATH 表示原来的 PATH 变量，new directory 表示要添加的新路径，中间用冒号隔开，旧的 PATH 变量加上新增路径之后再赋值给 PATH 变量。

2）PWD 和 OLDPWD

PWD 记录当前的目录路径，当利用 cd 命令切换到其他目录时，系统自动更新 PWD 的值，OLDPWD 保存旧的工作目录。输出这两个变量的值，结果如下所示：

```
[itheima@localhost ~]$ echo $PWD
/root
[itheima@localhost ~]$ echo $OLDPWD
/usr/local
```

这表示当前所在目录是/root，之前所在目录是/usr/local。

3）HOME

HOME 记录当前用户的家目录，例如，在本机中有两个用户 root、itheima，分别用这两个用户输出 $HOME 变量的值，具体如下所示：

```
[itheima@localhost]$ echo $HOME
/home/itheima
[itheima@localhost root]$ echo $HOME
/root
```

4）SHELL

SHELL 变量的值是/bin/bash，表示当前的 Shell 是 bash。如果有必要使用其他 Shell，则需要重置 SHELL 变量的值。

5）USER 和 UID

USER 和 UID 是用于保存用户信息的环境变量，USER 保存已登录用户的名字，UID 则保存已登录用户的 ID。使用 echo 命令打印这两个环境变量，具体如下所示：

```
[itheima@localhost root]$ echo $USER $UID
itheima 1000
```

由以上打印结果可知，当前登录用户为 itheima，用户 ID 为 1000。

6）PS1 和 PS2

PS1 和 PS2 称为提示符变量，用于设置提示符格式。例如，“[itheima@localhost ～]$”就是 Shell 提示符，[]里包含了当前用户名、主机名和当前目录等信息，这些信息并不是固定不变的，可以通过 PS1 和 PS2 的设置而改变。

PS1 用于设置一级 Shell 提示符，也称为主提示符。使用 echo 命令查看 PS1 的值。

```
[root@localhost ~]# echo $PS1
[\u@\h \W]\$
```

由以上输出结果可知，变量 PS1 包含 4 项内容，这 4 项内容的含义分别如下：

- \u 表示即当前用户名；
- \h 表示主机名；
- \w 表示当前目录名；
- \$ 是命令提示符，普通用户是 $ 符号，如果是 root 用户，命令提示符是 # 符号。

PS2 用于设置二级 Shell 提示符，使用 echo 命令查看 PS2 的值，其结果如下所示：

```
[itheima@localhost ~]$ echo $PS2
>
```

PS2 的值为＞符号，当输入命令不完整时，将出现二级提示符。

```
[root@localhost ~]# echo "hello
>"                                  #二级提示符
hello
```

3. 环境变量的配置文件

Linux 中环境变量包括系统级和用户级，系统级的环境变量对每个用户都有效，而用户级的环境变量只对当前用户有效。环境变量的配置文件也分为系统级和用户级，系统级的文件有很多，例如/etc/profile、/etc/profile. d、/etc/bashrc、/etc/environment 等，在这些文件中定义的环境变量对所有用户都是永久有效的。用户级的环境变量配置文件主要是. bash_profile 和. bashrc 两个文件，它们位于用户的家目录下。例如，以 itheima 用户登录，它们位于/home/itheima/目录下，使用 cat 命令查看两个文件中的内容，具体如下所示：

```
[itheima@localhost ~]$ cat .bash_profile
# .bash_profile

# Get the aliases and functions
if [ -f ~/.bashrc ]; then
    . ~/.bashrc
fi

# User specific environment and startup programs

PATH=$PATH: $HOME/.local/bin: $HOME/bin

export PATH0
```

```
[itheima@localhost ~]$ cat .bashrc
# .bashrc

# Source global definitions
if [ -f /etc/bashrc ]; then
    . /etc/bashrc
fi

# Uncomment the following line if you don't like systemctl's auto-paging feature:
# export SYSTEMD_PAGER=

# User specific aliases and functions
```

.bash_profile 文件主要定义当前 Shell 环境变量，.bashrc 文件主要用于定义子 Shell 环境变量。如果当前 Shell 创建了一个子 Shell，则.bashrc 文件使得子 Shell 的环境变量与当前 Shell 的环境变相分离。

用户在上述文件中均可以定义永久有效的环境变量，但要区分开环境变量是对所有用户有效还是对当前用户有效。

3.2.3 位置变量

位置变量主要用于接收传入 Shell 脚本的参数，因此位置变量也被称为位置参数。位置变量的名称由“$”与整数组成，命名规则如下所示：

```
$n
```

$n 用于接收传递给 Shell 脚本的第 n 个参数，如变量 $1 接收传入脚本的第一个参数。当位置变量名中的整数大于 9 时，需使用{}将其括起来，如脚本中的第 11 个位置参数应表示为 ${11}。位置变量是 Shell 中唯一全部使用数字命名的变量。需要注意的是，n 是从 1 开始的，$0 表示脚本自身的名称。

接下来通过一个 Shell 脚本来演示位置变量的用法，如例 3-1 所示。

例 3-1 编写 test.sh 脚本，其内容如下所示：

```
#!/bin/bash

echo "The script's name is : $0"
echo "Parameter #1: $1"
echo "Parameter #2: $2"
echo "Parameter #3: $3"
echo "Parameter #4: $4"
echo "Parameter #5: $5"
echo "Parameter #6: $6"
echo "Parameter #7: $7"
echo "Parameter #8: $8"
echo "Parameter #9: $9"
echo "Parameter#10: ${10}"
```

执行这个脚本，并传入相应的参数，结果如下所示：

```
[itheima@localhost ~]$ bash test.sh a b c d e f g h i j
The script's name is : test.sh
Parameter #1: a
Parameter #2: b
Parameter #3: c
Parameter #4: d
Parameter #5: e
Parameter #6: f
Parameter #7: g
Parameter #8: h
Parameter #9: i
Parameter #10: j
```

在接受参数时，位置变量只根据位置来接受相应参数，比如修改 test.sh 脚本。

```
#!/bin/bash

echo "The script's name is : $0"
echo "Parameter #8: $8"
```

修改之后的脚本只保留一个第 8 个位置变量，再次执行这个脚本，还是传递 10 个参数。结果如下所示：

```
[itheima@localhost ~]$ bash test.sh a b c d e f g h i j
The script's name is : test.sh
Parameter #8: h
```

在传入的参数中，第 8 个位置是 h，$8 读取到了相应位置的参数。如果传入的参数不足 8 个，那么 $8 值为空。

3.2.4　特殊变量

除了上述几个变量之外，Shell 还定义了一些特殊变量，主要用来查看脚本的运行信息。Shell 中的常用的特殊变量如下所示。

- $#：传递到脚本的参数数量。
- $* 和 $@：传递到脚本的所有参数。
- $?：命令退出状态，0 表示正常退出，非 0 表示异常退出。
- $$：表示进程的 PID。

接下来修改例 3-1 中的 test.sh 脚本来演示特殊变量的用法，如例 3-2 所示。

例 3-2　修改 test.sh，增加几行代码，如下所示：

```
#!/bin/bash

echo "The script's name is : $0"
echo "Parameter #1: $1"
```

```
echo "Parameter #2: $2"
echo "Parameter #3: $3"
echo "Parameter #4: $4"
echo "Parameter #5: $5"
echo "Parameter #6: $6"
echo "Parameter #7: $7"
echo "Parameter #8: $8"
echo "Parameter #9: $9"
echo "Parameter #10: ${10}"
#新增代码
echo "Parameter count: $#"          #传递给脚本参数数量
echo "All parameters: $* "          #传递给脚本的所有参数
echo "All parameters: $@"           #传递给脚本的所有参数
echo "PID: $$"                      #本程序的进程 ID
```

在 test.sh 脚本中使用特殊变量显示该脚本运行的信息。执行 test.sh 脚本，传入相应参数，结果如下所示：

```
[itheima@localhost ~]$ bash test.sh a b c d e f g h i j
The script's name is : test.sh
Parameter #1: a
Parameter #2: b
Parameter #3: c
Parameter #4: d
Parameter #5: e
Parameter #6: f
Parameter #7: g
Parameter #8: h
Parameter #9: i
Parameter #10: j
Parameter count: 10
All parameters: a b c d e f g h i j
All parameters: a b c d e f g h i j
PID: 7997
```

由后 4 行输出结果可知：使用变量 $ # 可获取脚本执行时接收的参数总量，使用变量 $ * 与 $ @ 可获取传入脚本的全部参数，使用变量 $ $ 可获取当前脚本的进程 ID。

3.3 Shell 中的符号

Shell 除了命令，还有一些作用很强大的符号，如引号、通配符、连接符等。这些符号在 Shell 命令中有着各种各样的作用，借助这些符号，用户可以用命令完成更复杂的功能。本节将对 Shell 中常用的符号进行讲解。

3.3.1 引号

在 Shell 中，引号主要用来转换元字符的含义。所谓元字符，是指那些在正则表达式

(正则表达式将在 3.4 节学习)中具有特殊处理能力的字符,如 $ 、\、>等字符。

Shell 中的引号有 3 种:单引号('')、双引号("")与反引号(``)。接下来分别介绍这几种引号。

1. 单引号

单引号可以将它中间的字符还原为字面意义,实现屏蔽 Shell 元字符的功能。引号里的字符串就是一个单纯的字符串,没有任何含义。例如,定义变量 NUM=100,在输出变量时需要添加 $ 符号,如果这个变量加上单引号输出,则直接将 $ 符号与变量整体作为一个字符串输出,命令如下所示:

```
[itheima@localhost ~]$ NUM=100
[itheima@localhost ~]$ echo $NUM
100
[itheima@localhost ~]$ echo '$NUM'
$NUM
```

在第二次加单引号输出 $NUM 时,直接输出了一个字符串而不是值 100,单引号将 $ 符号的功能屏蔽了。

注意:不可以在两个单引号中间单独插入一个单引号,单引号必须要成对出现。

2. 双引号

双引号也具有屏蔽作用,但它不会屏蔽 $ 符号、\符号和`符号。将刚才定义的变量 NUM 加双引号输出,具体如下所示:

```
[itheima@localhost ~]$ echo "$NUM"
100
```

由以上输出结果可知,使用双引号输出变量 NUM 时,$ 符号的功能不会被屏蔽。

注意:双引号也可以屏蔽单引号的作用,在一对双引号中,单引号不必成对出现。

3. 反引号

在 3.1.3 节讲解过反引号,它可以进行命令替换。反引号与双引号可以结合使用。例如,输出系统的时间,具体操作如下:

```
[itheima@localhost ~]$ echo "Today is `date`"
Today is 2017年 09月 07日 星期四 10:50:26 CST
```

以上所示的命令中用到了命令 date,该命令的功能是打印系统当前的时间。

可以把反引号嵌入到双引号中,但是当把反引号嵌入到单引号中时,单引号会屏蔽掉反引号的功能。例如,把`date`嵌入单引号中,将不会打印出当前的时间。

```
[itheima@localhost ~]$ echo 'Today is `date`'
Today is `date`
```

3.3.2 通配符

Shell 的通配符一般用于数据处理或文件名匹配，常用的通配符如表 3-1 所示。

表 3-1 Shell 中的通配符

符号	说 明	符号	说 明
*	与零个或多个字符匹配	[]	与[]中的任一字符匹配
?	与任何单个字符匹配	[!]	与[]之外的任一字符匹配

下面将对表 3-1 中的通配符逐一讲解。

1. 通配符“*”

如果用户想要列出/etc 目录下以 sys 开头的所有文件，可以使用如下命令：

```
[itheima@localhost~]$ ls -d /etc/sys*
sysconfig  sysctl.conf  sysctl.d  systemd  system-release  system-release-cpe
```

在以上命令中，sys* 表示匹配以字符串 sys 开头的所有文件。-d 选项表示仅对目标目录本身进行处理，不递归处理目录中的文件。

如果想输出以.conf 结尾的所有文件，则可以使用如下命令：

```
[itheima@localhost~]$ ls /etc/*.conf
asound.conf               fuse.conf      man_db.conf      rsyncd.conf
autofs.conf               GeoIP.conf     mke2fs.conf      rsyslog.conf
autofs_ldap_auth.conf     host.conf      mtools.conf      sestatus.conf
brltty.conf               idmapd.conf    nfsmount.conf    sos.conf
…
```

在这个命令中，*.conf 表示匹配所有以.conf 字符串结尾的文件，此命令会输出所有以.conf 结尾的文件。因为文件太多，在这里只截取一部分。

2. 通配符“?”

通配符“?”每次只能匹配一个字符，通常与其他通配符结合使用。如果想查找/etc 目录下文件名是由两个字符组成的文件，可以使用如下命令：

```
[itheima@localhost~]$ ls -d /etc/??
hp  pm
```

3. 通配符“[]”

通配符“[]”表示与[]中的任一字符匹配，它通常是一个范围。例如，在/etc 目录，列出以 f～h 范围的字母开头，并以.conf 结尾的文件，可以使用如下命令：

```
[itheima@localhost~]$ls /etc/[f-h]*.conf
fprintd.conf  fuse.conf  GeoIP.conf  host.conf
```

由输出结果可知，/etc 目录下以 f～h 范围内的字母开头，并以.conf 结尾的文件有 4 个。

4. 通配符“[!]”

通配符“[!]”表示除了[]里的字符，与其他任一字符匹配。例如，如果查找以 y 开头且不以.conf 结尾的文件，可以使用如下命令：

```
[itheima@localhost~]$ ls -d /etc/y* [!.conf]
yum  yum.repos.d
```

由输出结果可知，/etc 目录下符合条件的匹配项有两个。

3.3.3　连接符

Shell 中提供了一组用于连接命令的符号，包括“;”“&&”以及“||”，使用这些符号，可以对多条 Shell 指令进行连接，使这些指令顺序或根据命令执行结果有选择地执行。下面将对这些符号的功能分别进行介绍。

1. “;”连接符

使用“;”连接符间隔的命令，会按照先后次序依次执行。假如现在有一系列确定的操作需要执行，且这一系列操作的执行需要耗费一定时间，如安装 gdb 包时，在下载好安装包后，还需要逐个执行以下命令：

```
[root@localhost ~]#tar -xzvf gdb-7.11.1.tar.gz
[root@localhost ~]#cd gdb-7.11.1
[root@localhost ~]#./configure
[root@localhost ~]#make
[root@localhost ~]#make install
[root@localhost ~]#gdb -v
```

且在大多数命令开始执行后，都需要一定的时间，等待命令执行完毕。若此时使用“;”连接符连接这些命令，具体如下所示：

```
[root@localhost ~]#tar -xzvf gdb-7.11.1.tar.gz ;cd gdb-7.11.1;./configure;
make;make install;gdb -v
```

系统会自动执行这一系列命令。

2. “&&”连接符

使用“&&”连接符连接的命令，其前后命令的执行遵循逻辑与关系，只有该连接符之前的命令执行成功后，它后面的命令才被执行。

3. “||”连接符

使用“||”连接符连接的命令，其前后命令的执行遵循逻辑或关系，只有该连接符之前的

命令执行失败时，它后面的命令才会执行。

3.4 正则表达式

正则表达式是预先定义好的一组规则(也称为模式)，这组规则通常应用于文本搜索与替换中。由于其语法简练、功能强大，许多编程语言，如 C++ 、Java、Perl、Shell 等都提供了对正则表达式的支持。

3.4.1 正则表达式的概念

正则表达式可以对文本进行过滤，而它之所以拥有过滤文本的功能，是因为它定义了一系列的元字符。元字符与其他字符组合起来形成一定的规则，只有符合规则的文本才能保留下来，而不符合规则的文本会被过滤，如检测文本中是否包含字符串"ab"，其原理如图 3-2 所示。

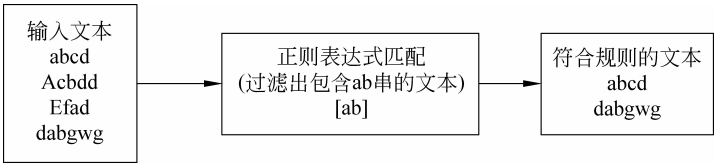

图 3-2 正则表达式对文本的处理

正则表达式也可理解为描述字符串结构模式的形式化表达“方法”或“思想”，它与一些特定工具搭配使用，可实现包括添加、删除、分离、插入、选择等各种文本分析功能。

注意：正则表达式的元字符与 Shell 中的通配符有一些重复，但它们的意义却不完全相同。

3.4.2 元字符

POSIX 规范规定了两种标准的正则表达式语法：一种是基本正则表达式，另一种是扩展正则表达式。这两种正则表达式的元字符组成略有不同。下面分别来学习这两种正则表达式的元字符。

1. 基础正则表达式元字符

1) 限定符“ * ”

符号“ * ”用于匹配前导字符 0 次或多次，具体示例如下：

```
hel*o
```

以上示例中“ * ”符号之前是普通字符 l，“ * ”符号就表示匹配 l 字符 0 次或多次，字符串 helo、hello、hellllllo 都可以与 hel * o 匹配。

2) 点字符“.”

符号“.”用来匹配除换行符“\n”外任意的单个字符。当正则表达式中出现“.”符号时，意味着该位置应有一个字符，具体示例如下：

```
..U73.
```

“.”只能匹配一个字符，因此上述字符串表示前两个字符是任意字符，第 3～5 个字符是 U73，最后一个字符也是任意字符。字符串 MHU73、4JU73H 等都可与..U73.匹配。

3）行首定位符“^”

符号“^”用来匹配行首字符，表示行首字符是“^”后面那个字符。例如，列举出/etc 目录下以字符串"sys"开头的文件，可以使用如下命令：

```
[itheima@localhost ~]$ ls /etc | grep "^sys"
sysconfig
sysctl.conf
sysctl.d
systemd
system-release
system-release-cpe
```

注意：尽管以上示例匹配的是以字符串"sys"开头的内容，但读者应理解为匹配以字符"s"开头，第二、第三个字符依次为 y 和 s 的行，这种理解方式更符合正则表达式的思维。这种思维在学习正则表达式时非常重要，读者应熟练掌握。

4）行尾定位符“$”

行尾定位符“$”用来匹配文本行末尾的字符，与“^”符号的作用正相反。例如，查找/etc 目录下以 conf 结尾的文件，可以使用如下命令：

```
[itheima@localhost ~]$ ls /etc | grep conf$
asound.conf
autofs.conf
autofs_ldap_auth.conf
brltty.conf
cgconfig.conf
cgrules.conf
cgsnapshot_blacklist.conf
…
```

同样，读者在理解行尾定位符的时候也应该从字符的角度去理解，即“conf$”匹配的是以字符 f 结尾，同时倒数第 2～4 个字符依次为 n、o、c 的文本行。

5）字符组“[]”

“[]”符号的功能比较特殊，它是用来指定一个字符集合的，其基本语法如下：

```
[abc]
```

其中 a、b 和 c 表示任意单个字符，只要某个字符串在方括号所在的位置出现了方括号中的任意一个字符，就能满足匹配规则。

另外，对于连续的数字或字母可以使用符号“-”来表示一个范围，例如，[a-z]，表示匹配到 a～z 中的任意一个字母。下面通过查看/etc 目录下以 rc 开头，并且 rc 后面紧跟一个数字的文件有哪些来演示“[]”符号的使用，查看的命令具体如下所示：

```
[itheima@localhost ~]$ ls /etc | grep "^rc[0-9]"
rc0.d
rc1.d
rc2.d
rc3.d
rc4.d
rc5.d
rc6.d
```

由输出结果可知，符合规则的文件一共有 7 个。

注意：元字符“*”或“.”位于“[]”符号之中，便仅表示一个普通的字符，不再具有特殊意义。

6）排除型字符组“[^]”

“[^]”表示不匹配其中列出的任意字符，其语法格式如下所示：

```
[^abc]
```

它的用法与“[]”符号相反，此处不再赘述。

2. 扩展正则表达式元字符

与基本正则表达式相比，扩展正则表达式新增加了一些元字符，它支持比基本正则表达式更多的元字符。下面介绍这些新增加的元字符。

1）限定符“+”

“+”符号与“*”符号类似，都可匹配其前导字符多次，但“*”符号支持匹配 0 次，而“+”符号要求至少匹配 1 次。例如，在/etc 目录下查找以 ss 开头的文件，分别使用“+”符号与“*”符号进行匹配，具体命令如下所示：

```
[itheima@localhost ~]$ ls /etc | grep "^sss*"
ssh
ssl
sssd
[itheima@localhost ~]$ ls /etc | egrep "^sss+"
sssd
```

由输出结果可知，第一个命令使用“*”符号进行匹配输出了 3 个文件，因为“*”符号可以匹配 0 次，所以 ssh 与 ssl 符合匹配规则；第二个命令使用“+”符号进行匹配，输出一个文件，因为“+”符号至少匹配一次，因此只有 sssd 符合匹配规则。

注意：在使用“+”符号时，使用了 egrep 命令，这是因为 grep 命令使用的是基本正则表达式，而 egrep 命令默认使用扩展正则表达式。如果使用 grep 命令要加上-E 选项。

2）限定符“?”

“?”也是一个限定符，它限定前导字符最多出现 1 次，即前导字符只能出现 0 次或 1 次。例如，查询/etc 目录下以"sss"或"ss"字符串开头的文件，使用“?”符号进行匹配，具体命令

如下所示：

```
[root@localhost ~]# ls /etc | egrep "^sss?"
ssh
ssl
sssd
```

在字符串"ss"之后可以出现 0 个或 1 个字符 s，因此这 3 个文件全部被输出。

3）“|”符号和“()”符号

“|”符号实现正则表达式之间的“或”运算，其格式如下所示：

```
表达式 1 | 表达式 2 |…|表达式 n
```

“()”表示一组可选值的集合。“()”经常结合“|”一起使用，表示一组可选的值。例如，筛选/etc 目录下包含字符串"ssh"或"ssl"或以"yum"开头的文本行，可以使用如下命令：

```
[itheima@localhost ~]$ ls /etc | egrep "(ssh|ssl|^yum)"
ssh
ssl
yum
yum.conf
yum.repos.d
```

“()”符号与“|”符号结合使用也可以实现“[]”符号的功能，如(a|b|c)。但“()”与“|”组合后的功能比“[]”更强大，“[]”只能匹配单个字符，而“()”与“|”结合使用可以匹配多个字符。

3.5　文本处理工具

3.4 节介绍了正则表达式，正则表达式主要用于过滤文本，它常与文本处理工具结合使用，Shell 提供了 3 个强大的文本处理工具：grep、sed、awk。本节将针对这 3 个文本处理工具进行详细的讲解。

3.5.1　grep

grep 是 Global Search Regular Expression and Print out the line(全局搜索正则表达式并打印文本行)的缩写，它是一个强大的文本搜索命令。它会从一个或多个文件中搜索与指定模式匹配的文本行，并打印匹配结果。grep 命令的基本格式如下：

```
grep [选项] [模式] [文件名]
```

以上格式中的模式是匹配规则，模式后的文件名用于指定搜索目标，文件名可以有多个，其间以空格分隔，模式前的选项用于对模式进行补充说明，grep 命令的常用选项如表 3-2 所示。