第一章 操作系统及UNIX Shell

- 1.5 Unix/Linux Shell概述
  - 1.5.1 Shell的种类
  - 1.5.2 Shell程序的识别
  - 1.5.3 Shell 环境
  - 1.5.4 Unix命令和工具表达形式的约定
- 1.6 Unix/Linux Shell 命令
- 1.7 Unix/Linux Shell 命令进阶
- 1.8 Unix/Linux Shell 编程

➢1.5.1 shell的种类

➢B shell

B shell(Bourne shell)的作者是Steven Bourne，简称shell，是UNIX最初使用的shell，并且在所有的UNIX上都可以使用。

缺点在于它处理用户的输入方面，键入命令会很麻烦，尤其当你键入很多相似的命令时。

➢C shell

C shell由Bill Joy所写，是一种与C语言很相似的shell，比B shell更适于编程。它的用户界面友好，支持象命令补齐(command-line completion)等一些Bourne shell所不支持的特性。

虽然普遍认为C shell的编程接口做的不如B shell，但C shell被很多C程序员使用，这也是C shell名称的由来。

➢1.5.1 shell的种类

➢Bash shell

Bash shell(Bourne Again Shell)是Linux操作系统基本的shell，简称Bash(GNU)。Bash与B shell完全向后兼容，即所有B shell的规范和程序都可以在Bash中使用。

Bash有许多特色，它包含了很多C shell和K shell中的优点，可以提供如命令补全、命令编辑和命令历史表(记忆使用过的命令、可用箭头键、提供history命令和命令行自动完成)等功能，有灵活和强大的编程接口，同时又有很友好的用户界面。

➢1.5.1 shell的种类

➢更多的shell脚本语言

还有许多语言虽然不是UNIX的用户界面，但都能够作为shell编程的脚本语言，例如perl，tcl等语言。

➢如何找到shell语言/脚本语言

所有的shell语言/脚本语言的可执行文件，通常存放在UNIX的标准目录/bin或者/usr/bin下。在使用一种UNIX操作系统时，可以用ls命令或者find命令查找这些文件是否存在：

/bin/ksh，/usr/bin/perl，/usr/bin/tcl，/usr/bin/tclsh。

而/bin/sh，/bin/csh一定存在，/bin/bash在Linux中一定存在。

find命令的使用示例：

$ find / -name “ksh” -print

➢1.5.2 Shell程序的识别

UNIX/Linux为每个用户都设定了某种shell环境：

- **B shell　　UNIX的基本shell操作语言**
- **Bash shell　　Linux的基本shell操作语言**
- **C shell　　类C语言的shell操作语言**

➢1.5.2 Shell程序的识别

UNIX/Linux为每个用户都设定了某种shell环境：

➢不同shell的区别

shell	B shell	Bash shell	C shell
启动shell的命令	sh	bash	csh
shell的待命符	$	$	%
配置文件	.profile	.bashrc	.cshrc
		.bash_profile	.login

配置文件的作用是在用户登录UNIX系统时设定用户的环境。由于它们都是以“.”为文件的首字符，所以都是隐含文件。

用户登录时，将处在所设定的shell环境中。但是可以使用启动shell的命令来转到另一种shell环境下，例如从B shell转到C shell，或者从Bash shell转到C shell等。

➢1.5.3 Shell环境

系统管理员建立用户帐户时需要设置某一种shell环境，通常是B shell(UNIX)或者Bash shell(Linux)。SHELL是一个环境变量，可以用echo命令显示。例如：

$ echo $SHELL	可以获知目前启动的是哪一种shell
/bin/sh	说明用户将使用B shell(UNIX)

如果希望转到另一种shell(例如C shell)，需要执行命令：

$ csh	启动C shell
%	%为C shell的缺省待命符
% ...	在C shell环境下工作
% exit 或者 logout	结束csh
$	返回原先的B shell或者Bash shell

如果希望在某种shell环境下转入另一种shell，应该使用正确的启动shell的命令。例如，从B shell转入C shell的命令为csh，从C shell转入的B shell的命令为sh。但是，不要连续启动不同的shell环境，因为这样将执行过多的进程，影响运行速度。而应该先退出一种shell环境，再启动另一种shell。例如，

以下做法是不恰当的，因为执行了过多的进程： $ csh % sh $ csh	以下是正确的做法： $ csh % exit $ 退出C shell，因为原来就是在B shell中

➢1.5.4 UNIX命令和工具表达形式的约定

在介绍shell和一些UNIX的工具时，需要对命令格式和表达方式事先给予约定，否则容易令人产生误解。本课程采用以下约定：

- 回车符

作为一行文本或者一行命令的结束，总是需要键入回车符<CR>的。在显示一行文字时，行尾也必定存在换行符<CR>。在大多数情况下，虽然<CR>不会显示出来，但是不会产生误解。因此我们只有在必要的情况下会特地标明<CR>以表示强调。

例如：

$ echo “Good morning!”<CR>仅在需要强调时才标明<CR>

$ echo “Good morning!”　　通常可以省略<CR>

➢1.5.4 UNIX命令和工具表达形式的约定

➢空格的表示

必要的空格，用□表示。不能有空格的地方，也将专门说明。

例如：

$ echo□$a□$b	**表示不能省略空格**
$ echo $a $b	**明显需要空格的地方，省略空格□**
$ echo "$a□□$b"	**表示必须使用两个空格**
xy=1_3	**加以说明：等号两侧不能有空格**

➢控制符

控制符由CTRL键和字母键*X* 组成，可以采用CONTROL+*X* ，^*X* ，<CTRL/*X* >或者<CTRL/*X* > 等方式表示。例如：

CONTROL+C，^C ，<CTRL/C>或者<CTRL-C>的意义相同。

➢1.5.4 UNIX命令和工具表达形式的约定

➢斜体字

命令中的变量或者参数，需要由用户决定其输入内容的，用*斜体字*表示。

例如变量赋值语句的格式为：

变量=值　　*变量*和*值*的实际值需要由用户决定

➢下划线

执行命令时用户键入的数据或者字符，用下划线表示。但是，由于在待命符后面的命令必定是用户键入的，所以在绝大多数情况可以省略而不需要用下划线表示。

例如，执行命令：

$ my_cat　　my_cat必定是键入的命令，可省略下划线标记

Hello!　　用户从键盘输入的字符串，需要下划线标记

Hello!　　屏幕显示的字符串，不应该用下划线标记

^D　　用户从键盘键入控制符^D，需要下划线标记

在比较熟悉的情况下，可以省略所有的下划线标记。

第一章 操作系统及UNIX Shell

- 1.5 Unix/Linux Shell概述
- 1.6 Unix/Linux Shell 命令
  - 1.6.1 单行和多行命令
  - 1.6.2 输入输出定向
  - 1.6.3 文件名的通配符
  - 1.6.4 Shell变量
  - 1.6.5 特殊字符
- 1.7 Unix/Linux Shell 命令进阶
- 1.8 Unix/Linux Shell编程

1.6.1 单行和多行命令

- 单行表达多个命令

  可以在一行中键入多个命令，在两个命令之间用分号“;”作为分隔。例如：

  $ cd ; pwd　　等价为以下两行命令

  $ cd

  $ pwd

- 多行表达命令

  如果某个命令的字符数较多，可以用几行完成键入，则需要在命令没有完成的一行行尾使用反斜杠“\”作为续行符。例如：

  **$ tar cvf /dev/rst0 /users/usr1 **

  > /users/eejm/eejma01/shell

  其中，当产生继续行时，shell将显示二级待命符>，而$称为一级待命符。

➢1.6.1 单行和多行命令

- 执行多个后台命令

可以用一对圆括号“()”将一组命令括起，一起作为后台命令执行。例如：

$ (date ; ps -ef) &　　使date和ps均为后台命令

$ date ; ps -ef &　　仅使ps为后台命令, 而date为前台命令

➢1.6.2 输入输出重定向

- UNIX指定了三个标准的输入输出定向

(1) 标准输入(stdin)，缺省值为键盘输入。
stdin的来源：shell的read语句，C语言的scanf()等。
(2) 标准输出(stdout)，缺省值为屏幕显示。
stdout的来源：shell的echo语句，C语言的printf()等。
(3)标准出错输出(stderr)，缺省值为屏幕显示。
stderr的来源：C语言的fprintf(stderr,...)等

➢1.6.2 输入输出重定向

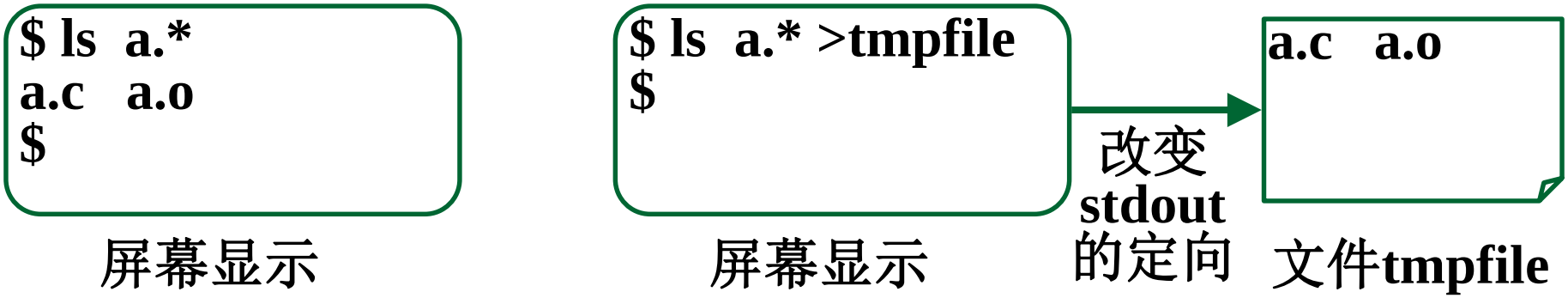
- 改变stdout的定向

格式为：> *file*，表示将stdout的输出改为存入文件*file*。例如:

$ ls a.* ls的输出为stdout

a.c a.o 缺省输出在屏幕显示

$ ls a.* >tmpfile 将ls的输出从stdout改为存入文件tmpfile



【例1-1】改变stdout的定向示例。

C程序文件my_cat.c为：

```
#include <stdio.h>
main()                          /* 从stdin读入字符，输出到stdout    */
{
    int c;
    while( ( c = getchar() ) != EOF)
        putchar(c);
}
```

通过编译，生成可执行文件my_cat。

如果执行命令：

```
$ my_cat                    从stdin读入字符，输出到stdout
Hello!                      从键盘输入字符串“Hello!”
Hello!                      在屏幕上显示字符串“Hello!”
^D                          键入^D等于文件结束符EOF
```

则屏幕上显示的两行字符串，第一行是键入的，第二行是my_cat的输出。

【例1-1】改变stdout的定向示例。

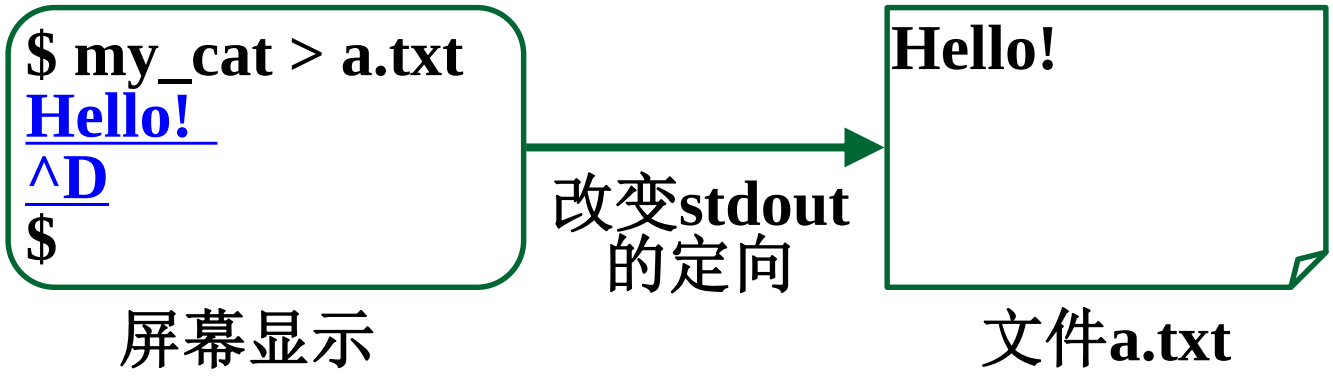
如果执行命令:

```
$ my_cat > a.txt
Hello!
^D
$
```

从stdin读入字符，存入文件a.txt
从键盘输入字符串“Hello!”
键入^D等于文件结束符EOF

则字符串“Hello!”将存入文件a.txt。

以上操作可以产生一些简单的文件，起到相当于使用文本编辑程序产生文件的功能。



➢1.6.2 输入输出重定向

- 改变stdin的定向

格式为：< *file*，表示将输入stdin改为取自文件*file*。

如果执行命令:

$ my_cat < a.txt　　从文件a.txt读入字符

Hello!　　在屏幕上显示字符串“Hello!”

$

以上操作相当于shell命令cat的作用，即：

$ cat a.txt　　在屏幕上显示文件a.txt

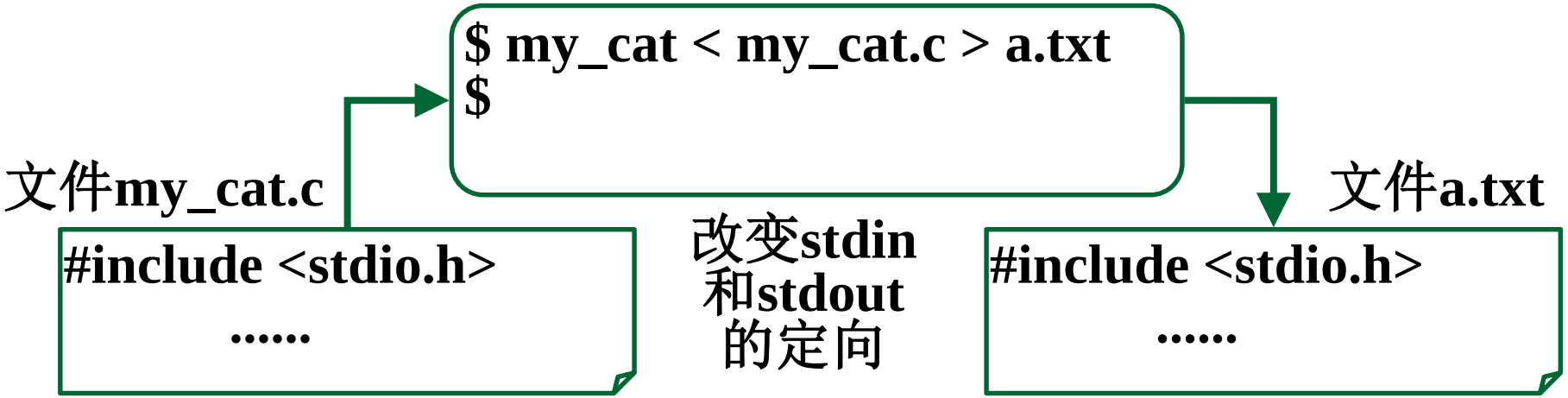
➢1.6.2 输入输出重定向

- 改变stdin和stdout的定向

  如果执行命令：

$ my_cat < my_cat.c > a.txt

表示将my_cat从stdin获得输入改为从文件my_cat.c获得输入，将输出到stdout改为存入文件a.txt，实现了相当于将文件my_cat.c复制为a.txt的功能。



➢1.6.2 输入输出重定向

- 附加标准输出

格式为：>> ***file***，表示将**stdout**的输出附加在文件***file***后面。例如，将以下**ls**命令的结果加到文件**tmpfile**的最后：

$ ls b.* >> tmpfile

- 改变**stderr**的定向（**B shell/Bash**）

格式为：**2>** ***file***，表示将**stderr**的输出改为存入文件***file***。例如，将**cc(C**编译程序**)**的出错信息存入文件**errfile**，操作为：

$ cc b.c 2> errfile

➢1.6.2 输入输出重定向

➢合并stdout和stderr

Bash增加了合并标准输出stdout和标准出错输出stderr的功能。在B shell中，已知改变stdout和stderr的方式分别为：

> *file*　　表示将stdout的输出改为存入文件*file*，以及

2> *file*　　表示将stderr的输出改为存入文件*file*。

如果某个程序或者命令既产生标准输出，比如C程序的编译命令cc或者gcc，又产生标准出错输出。则在B shell中，命令为：

cc *C-file* > *outfile* 2> *errfile*

如果希望将stdout和stderr指向同一个文件，命令为：

cc *C-file* > *file* 2> *file*

在Bash中，可以合并stdout和stderr的指向，命令为：

cc *C-file* > *file* 2>&1

➢1.6.2 输入输出重定向

➢ C shell中不能单独改变stderr的定向

同时改变标准输出和标准出错输出：
只需要用>&即可，不能用2>&1和2>

>& *file*

表示将stderr的输出改为存入文件*file*。
例如，将cc(C编译程序)的出错信息存入文件errfile。
在B shell和Bash中，操作为：

```
$ cc  b.c  2> errfile
```

在C shell中，操作为：

```
% cc  b.c  >& errfile
```

➢1.6.2 输入输出重定向

管道(流水线，pipeline)

如果有两个命令或者程序依次执行，前者的输出是指向stdout，后者的输入是来自stdin，这时可用管道(流水线，pipe line)将两个命令串接起来，使得可以从前者的输出获得后者的输入数据。

例如，以下命令实现了将my_cat的输出存入文件a.tmp，命令sort再从a.tmp获得输入。此时将产生一个临时文件a.tmp，此后需要将其删除。

```
$ my_cat  < /etc/passwd   >a.tmp
$ sort <a.tmp
```

如果使用管道，就可以将my_cat的输出直接送给sort的输入，从而不再产生临时文件。例如：

```
$ my_cat  < /etc/passwd | sort
```

再例如，以下是一系列命令的管道操作：

```
$ get_it | check_it | process_it | format_it > store_file
```

➢课本例题[1-2]

text:

0.□stdin:□Standard□input
1.□stdout:□Standard□output
2.□stderr:□Standard□error□output

$ tr□-cs□"[A-Z][a-z][0-9]\."□"[\012*]"<text | sort | uniq -c

tr命令

0.
stdin
Standard
input
1.
stdout
Standard
output
2.
stderr
Standard
error
output

sort命令

0.
1.
2.
Standard
Standard
Standard
error
input
output
output
stderr
stdin
stdout

uniq命令

1 0.
1 1.
1 2.
3 Standard
1 error
1 input
2 output
1 stderr
1 stdin
1 stdout

➢1.6.2 输入输出重定向

➢Here Document

在许多需要标准输入的命令中，可以使用“<<”改变标准输入定向(Here Document)。格式为：

$ command [*option ...*] [*variable ...*] << *InputFromHere*

input

InputsFromHere

其中，***input***是用户输入的若干行文字，***InputFromHere***称为分界符，是用户定义的任何一个字符串，位于“<<”之后，并且必须在***input***之后再出现。其功能是该命令将把分界符包含的***input***视为标准输入。

【注】在B shell、C shell和Bash中都可以使用“<<”的这个功能。

常用命令cat和ftp等常常会使用<<的功能。

<<还经常用来快速创建文本文件或者在文本文件中追加文字内容。

【例1-23】Here Document的应用示例

$ cat > a.txt	从stdin读入字符，存入文件a.txt
Hello!	从键盘输入字符串“Hello!”
Great!	从键盘输入字符串“Great!”
^D	键入^D等于文件结束符EOF

如果使用Here Document，将改为：

$ cat > a.txt <<END	用END作为分界符
Hello!	从键盘输入字符串“Hello!”
Great!	从键盘输入字符串“Great!”
END	键入分界符END作为标准输入结束

例如，用户jack需要在隐含配置文件.profile或者.bashrc中增加一个环境变量PS1的定义，命令操作如下：

$ cat >> .bashrc <<END	在.bashrc中追加文字
PS1=“[$LOGNAME@RedHat]□” ; export PS1	
END	
$ . .bashrc	运行配置文件
[jack@RedHat]□	用户jack的待命符PS1被改动

➢1.6.3 文件名的通配符

- 什么是通配符

星号“*”，问号“?”，一对方括号“[...]”是shell的三种对文件名的替代符或者通配符(File name generation)，它们的作用如下：

通配符	匹配内容
*	与任意多个字符串(包括 0 个字符)相匹配
?	与任意一个字符相匹配
[…]	以下两种情况的组合： [*xxx*]：与 *xxx* 枚举的一个字符相匹配 [*x*-*y*]：与 *x* 到 *y* 之间(按 ASCII 顺序)的一个字符相匹配

➢1.6.3 文件名的通配符

- 什么是通配符

在shell命令的参数中如果出现通配符，它们将对相应目录下存在的文件进行匹配。例如，在以下命令中出现通配符，将寻找匹配的文件名：

$ ls *.c 列出所有后缀为“.c”的文件

$ rm ../abc.?

删除父目录下所有前缀为abc后缀为一个字符的文件

$ more [ACD_a-e].a 查看前缀是匹配[ACD_a-e]的“.a”文件

【例1-2】通配符示例。若当前目录下有以下文件：

.at.out　　.o　　a.out　　c.c
.ctc　　a.a.o　　at.　　pc.o
.kc　　a.o　　at.out　　tc.out

*	匹配任意多个字符
?	匹配一个字符
[...]	匹配一个字符范围

以ls命令为例，以下为通配符及执行文件列表ls的效果：

(1) 用*匹配任意多个字符。列出以“a”为首，后有任意个字符的文件：

```
$ ls  a*
a.a.o      a.o      a.out      at.      at.out
```

(2) 用*匹配任意多个字符。列出在“.o”前有任意个字符的文件：

```
$ ls  *.o
a.a.o      a.o      pc.o
```

看起来，“.o”也是应该列出的文件，但是首字符为“.”的文件称为隐含文件，不能简单地用*匹配。

【例1-2】通配符示例。若当前目录下有以下文件：

.at.out　　.o　　a.out　　c.c
.ctc　　a.a.o　　at.　　pc.o
.kc　　a.o　　at.out　　tc.out

*	匹配任意多个字符
?	匹配一个字符
[...]	匹配一个字符范围

以ls命令为例，以下为通配符及执行文件列表ls的效果：

(3) 使用可选项“-a”，列出隐含文件：

```
$ ls –a
.at.out  .ctc  .kc  .o  a.a.o  a.o  at.  a.out  at.out  c.c
pc.o  tc.out
```

(4) 用?匹配一个字符。列出在“.out”前有两个字符的文件：

```
$ ls  ??.out
at.out  tc.out
```

(5) 用?匹配一个字符。列出在“.”的前后各有一个字符的文件：

```
$ ls  ?.?
a.o  c.c
```

【例1-2】通配符示例。若当前目录下有以下文件:

.at.out .o a.out c.c
.ctc a.a.o at. pc.o
.kc a.o at.out tc.out

*	匹配任意多个字符
?	匹配一个字符
[...]	匹配一个字符范围

以ls命令为例，以下为通配符及执行文件列表ls的效果：

(6) 同时使用*和?。列出在“.”前有两个字符，后有任意个字符的文件：

```
$ ls  ??.*
at.     at.out    pc.o      tc.out
```

(7) 用[...]匹配一个字符。列出以“a”或“t”为首字符引导的“*.out”文件：

```
$ ls  [at]*.out
a.out     at.out     tc.out
```

【例1-2】通配符示例。若当前目录下有以下文件:

.at.out	.o	a.out	c.c
.ctc	a.a.o	at.	pc.o
.kc	a.o	at.out	tc.out

*	匹配任意多个字符
?	匹配一个字符
[...]	匹配一个字符范围

以ls命令为例，以下为通配符及执行文件列表ls的效果:

(8) 用[...]匹配某个范围的一个字符。列出以“.”或者小写字母引导的“.out”文件:

```
$ ls  [.a-z]*.out
.at.out   a.out  at.out    tc.out
$ ls .*.out
.at.out
```

➢1.6.4 Shell变量

- 变量定义

不需要事先定义，可随时用字符串(标识符)定义一个变量。

- 变量赋值
  - B shell和Bash的变量定义语句

    变量=*值*　　注意：=的左右不能存在空字符(包括空格)
  - C shell的变量定义语句

    set *变量*=*值*

- 变量引用

$*变量*　　或者　　${*变量*}

➢1.6.4 Shell变量

➢ 变量的撤消

1、B Shell和Bash的变量撤销语句：

变量=

2、B shell，Bash和C shell的变量删除语句：

unset *变量*

例如（**Bash或B shell**）：

```
$ x=finish                定义x的值为finish
$ echo $x
finish                    运行结果：显示x的值为finish
$ set | grep x            set查看已定义变量
x=finish
$ unset x                 删除变量
$ set | grep x
$                         运行结果：变量列表中找不到x，表示x被删除
```

【注】用set命令可查已定义变量

➢1.6.4 Shell变量

⊙C shell增加的测试变量定义的特殊变量

无论是普通变量还是环境变量，C shell提供测试某个变量是否已定义的特殊变量：

$?*变量*

则将获得一个数值。如果为0，表示该变量尚未定义。如果为1，表示该变量已经定义。

【例1-34】C shell变量定义示例

```
% set x=finish                    定义x的值为finish
% echo $?x
   1                              变量已定义
% unset x                        删除变量
% echo $?x
   0                              表示x未定义
```

➢1.6.4 Shell变量

Shell类型	B	Bash	C
$ a=5	√	√	
$ set a=5			√
$ a=	√	√	
$ unset a	√	√	√
$?a			√

【例1-3】变量引用示例—变量串接（Bash/B shell）

$ a=xy	**变量a赋值**
$ b=1_3	**变量b赋值**
$ echo $a□$b	**用□分隔变量a和变量b的引用**
xy□1_3	**显示变量a和b的值**
$ echo $a□□□$b	**在$a和$b之间键入三个空格**
xy□1_3	**在变量a和b的值之间只输出一个空格**
$ echo “$a□□□$b”	**用双引号括起三个空格**
xy□□□1_3	**在变量a和b的值之间输出了三个空格**

【例1-3】变量引用示例一变量串接（Bash/B shell）

如果引用若干个变量时，相互之间不加空格或者制表符，而是紧连在一起，表示将它们的值相串接。

```
$ c=$a$b            变量a和b串接后对c赋值
$ echo $c
xy1_3               显示变量a和b串接后的值
$ c=$a/$b           变量a、/和变量b串接后对c赋值
$ echo $c
xy/1_3              显示变量a、/和变量b串接后的值
$ echo $a1/$b       变量a1、/和变量b串接后对c赋值
/1_3                变量a1没有定义，
                    只显示/和变量b串接后的值
$ echo ${a}1/$b     采用花括号引用变量，
                    ${a}1分隔了变量a和1
xy1/1_3             显示变量a、1/和变量b串接后的值
```

➢1.6.4 Shell变量

shell变量分为普通变量和环境变量。

➢普通变量

普通变量是指由用户定义的变量，通常用小写字母表示。

例如（**Bash/B shell**）：

```
$ root=my_proj
$ dir=src
$ my_dir=$root/$dir
```

➢1.6.4 Shell变量

➢环境变量

- 环境变量不仅仅可在shell命令中引用，还可以在shell程序以及用户程序(例如C程序)中引用。
- **B shell / Bash中**

  环境变量通常全部用大写字母表示。

用户也可添加和修改环境变量，并且用export命令确认，否则将视为普通变量。例如：

```
$ PATH=.:$HOME/bin:$PATH
$ export PATH
```

表示在原来设置的环境变量PATH中，增加两个可执行路径：

“.”(当前目录)

$HOME/bin(注册目录下的bin目录)

➢1.6.4 Shell变量

➢常用环境变量（B shell/Bash）

- **PATH**　　可执行路径

每执行一个命令(UNIX内部命令除外)，如果命令表示为绝对路径或者相对路径，例如

```
$ /bin/sh          绝对路径
$ ../bin/sh1       相对路径
$ ./sh2            相对路径
```

shell将执行该路径下的命令。如果没有说明路径，例如

```
$ sh1
```

shell将在PATH所定义的目录中依次寻找该命令，找到则执行，找不到将显示出错信息：“command not found”。

➢1.6.4 Shell变量

➢常用环境变量（B shell/Bash）

➢ PATH　　可执行路径

通常需要修改PATH的定义，例如当前PATH为“.:/bin:/usr/bin”，表示可执行路径为“.”(当前目录)，/bin，/usr/bin三个目录。

如果需要增加一个“$HOME/shell”到PATH中，则命令为：

$ PATH=$HOME/shell:$PATH ； export PATH

另外假定在/bin和/usr/bin下都存在一个同名的命令xyz，如果命令操作为：

$ xyz

将执行/bin/xyz。

如果需要执行/usr/bin/xyz，可有两个办法，或者说明xyz的路径，即：

$ /usr/bin/xyz

或者修改PATH为“ . :/usr/bin:/bin”，使得/usr/bin比/bin优先，然后执行：

$ xyz

1.6.4 Shell变量

常用环境变量（B shell/Bash）

- **LOGNAME** 用户名(user name)，即注册时用的账号只能引用，不能修改。
- **HOME** 用户的注册目录

例如：

```
$ cd  $HOME/bin
```

等价为：

```
$ cd
$ cd  bin
```

➢1.6.4 Shell变量

➢常用环境变量（B shell/Bash）

➢ SHELL

由于本课程要求学会使用UNIX中的B shell和C shell，以及Linux中的Bash，因此每当登录一个操作系统时应该明白自己当前使用的是哪种shell环境。否则，由于shell种类的不同，将使得执行的命令发生错误。了解当前shell环境的命令为：

```
$ echo $SHELL
```

➢1.6.4 Shell变量

- 常用环境变量（B shell/Bash）
  - PS1和PS2　　一级待命符和二级待命符

系统将PS1和PS2分别设为$和>，用户可以修改。例如：

$ PS1="[$LOGNAME]□" ; export PS1

[wenqing]□　　一级待命符被改为用户名(wenqing)

- 查看环境变量的方法（B shell/Bash）

$ echo $*环境变量*

$ env 或者 set

区别：

- set:显示(设置)shell变量，包括的普通变量以及环境变量
- env:显示(设置)环境变量

➢普通变量与环境变量示例（B shell/Bash）

```
$ aaa=bbb          --shell变量设定
$ echo $aaa
bbb
$ env | grep aaa   --env查看环境变量列表，并没有aaa
$ set | grep aaa   --set查看已定义变量列表，有aaa
aaa=bbb
$ export aaa       --那么用export 导出一下
$ env | grep aaa   --发现aaa存在于环境变量列表中了
aaa=bbb
```

➢1.6.4 Shell变量

➢环境变量

➢ C shell中

C shell的环境变量通常全部用小写字母表示。同时，在C shell环境下，也能引用B shell(用大写字母表示)的环境变量。

定义C shell环境变量的语句格式为：

setenv□*环境变量*□*值*

使用setenv定义环境变量，可以直接生效，而不需要使用export命令（实际上cshell不认识export命令）。例如：

% setenv□PATH□“.:$PATH”

➢1.6.4 Shell变量

➢ 常用环境变量（C shell）

⊙USER　　用户名，即注册时用的账号

⊙~　　用户的注册目录

在Bash和C Shell中：

“cd ~”等价为“cd”或者“cd $HOME”，

“cd ~/bin”等价为“cd $HOME/bin”或者“cd ; cd bin”。

在C Shell中，~还可以用来匹配其他用户的注册目录。例如，用户me01想查看另外一个用户me00注册目录下的文件，可以执行以下命令：

% ls□~me00

⊙PATH　　可执行路径

例如：已设置的可执行路径为“/bin:/usr/bin”。如果需要增加两个可执行路径“.”(当前目录)和“~/shell”，则命令为：

% setenv PATH□.:~/shell:$PATH

➢1.6.4 Shell变量

➢ 常用环境变量（C shell）

⊙**prompt** 缺省的一级待命符

通常系统管理员将**prompt**的初值设为%，用户可以修改。例如：

% setenv prompt “[$username@`host`]\!>□”

[wenqing@fdme00]5>□

其中，**`host`**表示将执行**host**命令，以获得当前使用的宿主机名，**\!**表示当前的命令序号(命令序号将在命令史中介绍)。

※查看环境变量的方法

% echo $*环境变量*

% printenv *环境变量*

% env

% set

区别：

- **set:**显示(设置)**shell**变量，包括的普通变量以及环境变量
- **env:**显示(设置)环境变量

➢1.6.4 Shell变量

用户定义的环境变量在撤消登录后将失效。下次登录需要重新定义。如果希望每次登录时都能够自动生效，必须将环境变量的设置写入配置文件。例如把设定PATH的两行命令写入配置文件，每次登录环境变量PATH都将自动生效。

shell	B shell	Bash shell	C shell
配置文件	.profile	.bashrc	.cshrc
		.bash_profile	.login

如果将环境变量的设置在写入配置文件后希望立即生效，必须运行配置文件。配置文件的运行方式为：

B shell和Bash： **.□*配置文件***

C shell： **source□*配置文件***

例如：

$ . .profile　　或　　% source□~/.cshrc

➢1.6.4 Shell变量

➢特殊变量

【注】 $$和$?是在B shell、C shell和Bash中都可以使用的特殊变量。

- 进程号

$$表示当前执行的进程号。由于进程号是唯一的，通常用来在shell程序中命名文件，以免文件重名。

例如：

```
$ ls
a.out      test.txt
$ ls > $$.$$          改变stdout的定向，产生由进程号命名的文件
$ echo $$  $$.$$                  显示进程号
280   280.280
$ ls
280.280    a.out     test.txt     显示增加了由进程号命名的文件
```

➢1.6.4 Shell变量

➢特殊变量

➢ 进程状态值

$?为进程状态值，表示前一进程执行结果的状态值，或者称为命令返回值，表示上个命令的返回值。$?的值为0表示进程执行成功，为非0值表示进程失败。

例如：

```
$ ls
a.out     test.txt
$ echo $?                    上一个命令执行成功，状态值为0
0
$ ls  tst.txt                文件tst.txt不存在，出错
ls: tst.txt: No such file or directory
$ echo $?                    上一个命令执行失败，状态值为2
2
$
```

➢1.6.5 特殊字符

shell中以下字符称为特殊字符，它们通常表达的不是字符本身，而是作为特殊的用途：

;	命令分隔符(单行多命令)	*	字符串通配符
( )	命令组或运算括号	?	单字符匹配符
&	后台进程	[ ]	枚举字符匹配符
< >	输入输出定向	#	注释行引导符
|	管道符	$	shell变量的取值符

➢1.6.5 特殊字符

~ C shell中的home，类似于B shell的$HOME

\ 有三种作用：

1) 续行符(命令的多行表达)

2) 和个别字符一起构成特殊含义

例如，在shell程序中使用的\n(换行)和\c(不换行)

3) 将一个特殊字符转义为正常字符(escape)

例如：\#表示字符#，\\表示字符\

- `` 命令结果替换符(重音符，抑音符，反引号)

```
$ p=`pwd`        （立即替换）
$ echo p=$p
p=/users/usr2
```

- ”” 弱转义(弱引用)，将双引号内的特殊字符转义为正常字符(消除特殊含义)，但是$、`和\除外
- ’’ 强转义(强引用)，将单引号内的特殊字符转义为正常字符

```
$ x=*
$ echo $x
hello.sh menus.sh misc.sh phonebook tshift.sh
$ echo "$x"
*
$ echo '$x'
$x
```

➢作　　业

- 上机操作、复习课堂讲过的命令
- shell命令上机及习题，将运行结果写在作业本上

  E-2 (shell变量)1-19.1

  E-4 (shell变量)1-21.1

  E-8 (shell变量及特殊字符)1-31.1, 1-31.2, (通配符)1-31.4
- C语言编程练习 E-18：3-4
- 3月30日及4月5日上机安排说明：
  - 4月5日（周四）清明节放假
  - 原周四下午上机的同学本周安排在3月30日下午4:20-5:05上机
  - 原周五下午上机的同学本周安排在3月30日下午3:25-4:10上机
  - 上周已完成相关上机内容的同学可不必参加本周上机