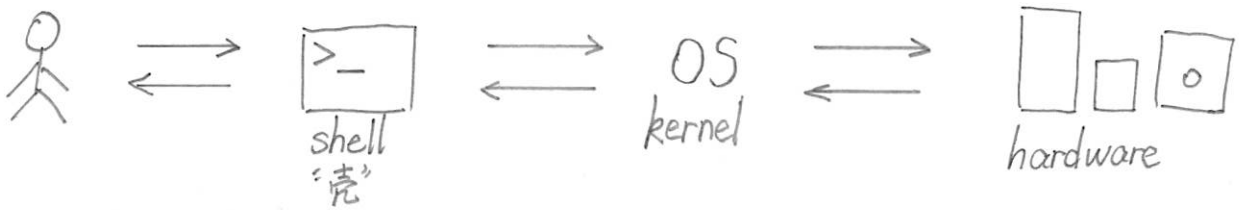
Lec10 认识与学习 BASH

认识 BASH 这个 shell

1. 使用终端下达指令，就是通过 bash 的环境来处理的。
2. 使用者通过 shell 与 kernel 沟通。

Q: 系统中有多少 shell？
为什么选择 bash？

硬件、核心与 shell



接口

为何需要文字接口的 shell？

1. 各家 distributions 使用的 bash 都是一样的
2. 远程管理：文字接口比较快
3. 方便进行大规模地管理

系统的合法 shell 与 /etc/shells 功能

- Bourne SHell (sh)
- C SHell (∵ Linux 为 C 语言撰写)
- K SHell
- TCSH (整合了 C SHell，并提供了更多功能)
- ...

Linux 使用的是 Bourne Again SHell (bash)，其是 Bourne SHell 的增强版
在 /etc/shells 这个文件下可以看到有哪些可用的 shell。
各家的 shell 的功能都差不多，但在某些语法下达方面却有所不同。Linux 预设的是 bash
可通过 `/etc/passwd` 查看每个用户的预设 shell。

bash shell 的功能

/bin/bash 是 Linux 预设的 shell，优点有：

1. history ← the `history` command isn't an executable file on your filesystem, like most commands, but a built-in function of bash

过去使用过的指令记录在家目录的 .bash_history 里。(~/.bash_history)
*有的操作系统会暂时把这一次登录时使用过的指令放在内存中，当用户 logout 时再写入 ~/.bash_history，可通过 `history -w` 进行写入。

2. Tab 按键：命令与文件的补全功能
   这个功能是bash里头才会的。
3. 命令别名的设定功能 alias
   `alias` ：显示目前的命令别名
   `alias fragrans="ls -la"`
4. 工作控制、前景控制、背景控制
   job control ： ctrl+c 来停掉程序
   foreground control & background control ：详见 Lec16
5. 程序化脚本 shell scripts
6. 通配符 wildcard
   如`ls -l /usr/bin/x*`

注：
use `help` to check all built-in

查询指令是否为 bash shell 的内建命令：type

如何知道某个指令是来自于外部（非bash提供的指令），还是内建在bash当中？

↑ 此时type可类似which　　↑ 例如 cd、history

指令的下达与快速编辑按钮

Enter ：换行

Enter ：不是换行

ctrl + a ：让光标移到最前面
ctrl + e ：让光标移到最后面

shell 的变量功能

什么是变量

略

变量的取用与设定：echo与 =

例如：

```
echo $PATH
echo ${PATH}
```

```
echo ${jasmine}
    ← 返回空的
jasmine=anhao
echo ${jasmine} ← 返回 anhao
```

在bash当中，当一个变量名称尚未被设定时，预设的内容是空的，同时也有一些规则。

每一种 shell 的语法都不相同

变量设定规则

① myname=jasmine

② myname = jasmine ✗

myname=jasmine mipha ✗

等号两边不能直接接空格

③ 变量名必须以字母或下划线开头，后跟字母、下划线或数字。

④ 变量内容中若含有空格，则可使用双引号或单引号括起来，但

双引号内的特殊字符，如$等，可以保有原本的特性；

单引号内的特殊字符则仅为一般字符(纯文本)

```
lover="her name is $myname"
echo ${lover} ← "her name is jasmine"
lover='her name is $myname'
echo ${lover} ← "her name is $myname"
```

⑤ 转义字符 \

myname=AC\ Milan

⑥ `<command>` 或 $(<command>)

```
temp=`ls`
temp=$(ls)
```

⑦ 若该变量内容可扩增时，则可以通过 "$<variable>" 或 ${<variable>} 来扩增

```
PATH="$PATH":/home/bin
PATH=${PATH}:/home/bin
name=${name}chen
```

变量 → 环境变量

export <variable>

取消变量：unset

unset <variable>

环境变量的功能

用`env`观察环境变量与常见环境变量说明

PATH SHELL HOME LANG RANDOM

用`set`观察所有变量（环境变量+自定义变量）

PS1：命令提示字符，

每次按 Enter 去执行某个命令时，最后要再次出现提示字符时，就会去主动读取这个变量。

如"i542016@VSa1179357:~>"

PS2：使用转义符\后下一行显示的内容

$：关于本 shell 的 PID

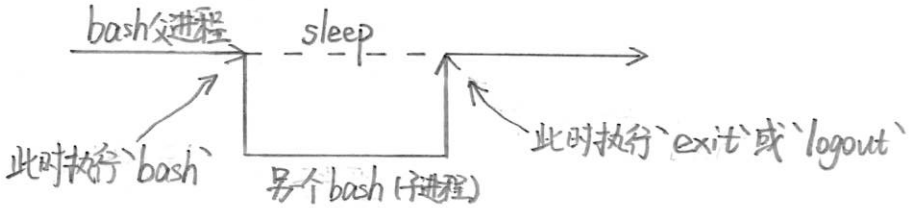
?：关于上个执行命令的回传值

用`export`将自定义变量转为环境变量

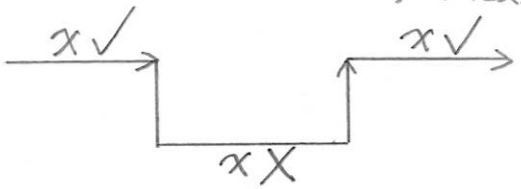
关键：该变量是否会被子程序所继续引用

当登入Linux并取得一个bash后，这个bash就是一个独立的进程（有PID）。

接下来在这个bash下所下达的任何命令都是由这个bash所衍生出来的，即子进程。



子进程仅会继承父进程的环境变量，而不会继承父进程的自定义变量。



影响显示结果的语言编码的变量 locale

`locale`

`cat /etc/locale.conf`

变量的有效范围

类比：

环境变量 = 全局变量 global variable

自定义变量 = 局部变量 local variable

原因：

内存配置

注：

环境变量 ≠ bash 的操作环境

变量键盘读取、数组与宣告：read、array、declare

```
read [-pt] <variable>      #读取来自键盘输入的变量
```

[-p] 可以接提示字符

[-t] 用于设定秒数（不会一直等待）

例：`read -p "what's your name?" -t 30 name`

```
declare [-aixr] <variable>  # 宣告变量的类型
```

[-a] 将 <variable> 定义为数组（array）类型

[-i] 将 <variable> 定义为整数数字（integer）类型

[-x] 将 <variable> 设为环境变量

[-r] 将 <variable> 设为 readonly，即变量不可改，也不能 unset，除非注销再登入

例：
```
declare -i sum=1+4+9
declare +x sum    #取消 sum 的环境变量的"地位"
```

注：

1. 变量类型默认为字符串。
2. bash 环境中的数值运算，预设最多仅能到整数。

例：
```
flowers[1]="jasmine"
flowers[2]="fragrans"
flowers[3]="plum"
echo "${flowers[1]}, ${flowers[3]}"
```

与文件系统及程序的限制关系：ulimit

bash是可以限制用户的某些系统资源的，

包括 { 可以开启的文件数量
可以使用的CPU时间
可以使用的内存总量
...

```
ulimit -a  # 列出当前用户的所有限制
ulimit [-SHacdfltu] <配额>
```

[-H] hard limit

[-S] soft limit

[-c] 当某些程序发生错误时，系统可能会将该程序在内存中的信息写成文件（排错用），这种文件就叫核心文件（core file）。此为限制每个核心文件的最大容量。

[-f] 此shell可以建立的最大文件容量，单位为blocks（SUSE12）。

[-d] data segment size

[-l] max locked memory

[-t] 可使用的最大CPU时间，单位为秒。

[-u] max user processes number

变量内容的删除、取代与替换

“将变量的内容进行微调”

略，需要时再查。

命令别名与历史命令

命令别名设定：alias、unalias

```
`alias anhao="ls -l"`
`unalias anhao`
`alias cls='clear'`
`alias dir='ls -l'`
```

历史命令：history

```
history <n>
history -c  # 将目前shell中的所有history内容全部清除
history [-raw]  # 对~/.bash_history操作
```

read append write

bash shell 的操作环境

指令运作的顺序

1. 以相对/绝对路径执行指令，例如 `/bin/ls` 或 `./ls`
2. 由 alias 找到该指令来执行
3. 由 bash 内建的 (builtin) 指令来执行
4. 通过 $PATH 这个变量的顺序搜寻到的第一个指令来执行

case. 假如 `ls` 命令已经有其它别名、且有多个名为 ls 的可执行的二进制文件，那么究竟哪个会被执行呢？

例：

```
i542016@vsa11779357: ~ > alias echo='echo -n'
i542016@vsa11779357: ~ > type -a echo
        echo is aliased to 'echo -n'
        echo is a shell builtin
        echo is /usr/bin/echo
```

bash 的进站与欢迎讯息：/etc/issue 、/etc/motd

略

bash 的环境配置文件

"为什么没有进行任何操作，但一进入 bash 就已经取得了一堆有用的变量了？"

系统存在一些环境配置文件，bash 在启动时会直接读取这些配置文件，以规划好 bash 的操作环境。

- 全体系统的配置文件
- 用户个人偏好配置文件

注：命令别名、自定义的变量等，会在注销 bash 后失效。

△ login 与 non-login shell （是否需要完整的登入流程）

这两个取得 bash 的情况中，读取的配置文件并不一样。

login shell 会读取：
- /etc/profile 系统整体的设定
- ~/.bash_profile 或 ~/.bash_login 或 ~/.profile 使用者个人的设定

non-login shell 仅会读取 ~/.bashrc

△ /etc/profile

（每个使用者登入、取得 bash 时一定会读取的配置文件）

此文件设定的变量有 PATH、MAIL、USER、HOSTNAME、HISTSIZE、umask 等

/etc/profile 不仅仅会设定这些变量（整体环境），还会去呼叫外部的设定数据。（呼叫出其他的配置文件）

在CentOS 7.x默认的情况下，会依序呼叫：

① /etc/profile.d/*.sh

需要且仅需使用者拥有 r 的权限，该文件就会被 /etc/profile 呼叫进来。

这些文件规范了：
- bash 操作接口的颜色
- 语系
- ll与ls 指令的命令别名
- vi的命令别名
- which 的命令别名
- ...

② /etc/locale.conf

此文件由 /etc/profile.d/lang.sh 呼叫进来。

决定 bash 预设使用何种语系、LANG/LC_ALL。

③ /usr/share/bash-completion/completions/*

tab 的妙用：命令补齐、文件名补齐、指令的选项/参数补齐，

就是从这个目录里面找到对应的指令来处理的，

这个目录底下的内容是由 /etc/profile.d/bash_completion.sh 载入。

△ ~/.bash_profile 或 ~/.bash_login 或 ~/.profile

bash 在读完了整体环境设定的 /etc/profile 并借此呼叫其他配置文件后，

接下来则会读取使用者的个人配置文件。

```
for f in ["~/.bash_profile", "~/.bash_login", "~/.profile"]:
----if os.exists(f):
--------read_profile(f)
--------break
```

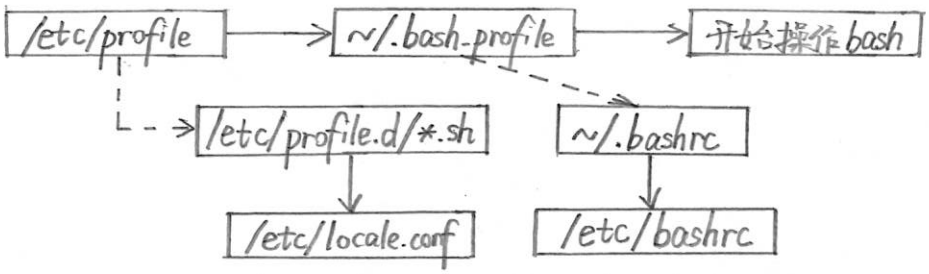


图. login shell 的配置文件读取流程

△ source：读入环境配置文件的指令

source <配置文件文件名> 或 . <配置文件文件名>

△ ~/.bashrc

non-login shell 会读

△其他相关配置文件

/etc/man_db.conf ：man page 的路径到哪去找

~/.bash_history ：历史命令

~/.bash_logout ："当我注销 bash 后，系统再帮我做完什么动作后离开"

终端机的环境设定：stty, set

在文字接口的终端（terminal）环境登入时，登入的时候我们可以取得一些字符设定的功能。

more generally，终端的输入环境的设定

例如：

backspace 可以删除命令行上的字符；

ctrl + C 强制终止一个指令的运行；

当输入错误时，会有声音跑出来警告；

stty（setting tty）：

```
stty -a     # 将目前所有的 stty 参数列出来
stty intr ^h  # 设定 interrupt（中断，原本为 ^c）为 ^h
```

除了 stty 之外，bash 还有自己的一些终端设定（利用 set 来设定）。

部分按键设定功能位于 /etc/inputrc，

与终端有关的环境配置文件位于 /etc/DIR_COLORS* 与 /usr/share/terminfo/* 等。

不建议修改 tty 的环境！

通配符与特殊符号

wildcard

表：一些常用的通配符

符号	意义	实例
*	代表 0 个到无穷多个任意字符	
?	代表一定有一个任意字符	
[ ]	代表一定有一个在括号内的字符（非任意）	[abcd]
[-]	代表在编码顺序内的所有字符	[0-9]
[^]	"反向选择"	[^ab]：不能是 a 或 b

表：部分特殊符号

符号	意义
#	注释
\	反义
|	管线（pipe）
;	下达连续指令的分隔符
~	用户的家目录
$	取变量前的前导符
&	工作控制（job control） ← 将指令变成背景下工作
!	逻辑运算的"非"
` `	先执行的指令，亦可用 $()

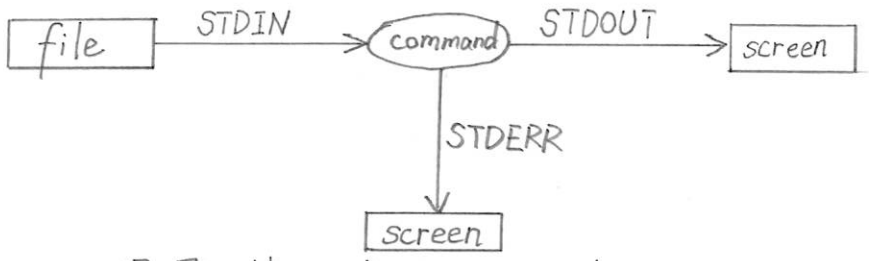
符号	意义
' '	不具有变量置换的功能（单纯文字）
" "	具有变量置换的功能（$有用）
( )	子 shell 的起始与结束
{ }	命令区块的组合

数据流重导向（redirect）

"将数据传导到其他地方"

将某个指令执行后，本该出现在屏幕上的数据，传输到其他地方，例如文件或装置。

什么是数据流重导向



图：默认情况下，指令执行过程中的数据传输情况

standard output：指令执行所回传的正确的讯息

standard error output：指令执行失败后，所回传的错误讯息

分别传送所用的特殊字符如下：

① 标准输入（stdin）：代码为 0，使用 < 或 <<

② 标准输出（stdout）：代码为 1，使用 > 或 >>

③ 标准错误输出（stderr）：代码为 2，使用 2> 或 2>>

} 将 stdout 与 stderr 分别重定向

2> 覆盖　2>> 追加

/dev/null：“垃圾桶”、“黑洞装置”

特殊写法：将指令的 stdout 与 stderr 写入同一个文件：`ls > tmp 2>&1` 或 `ls &> tmp`

standard input：将原本需要由键盘输入的数据，改由文件内容来取代。

例：

`cat > catfile` ← 按 ctrl + d 离开

`cat > catfile < ~/.bashrc`

`cat > catfile << "eof"`

结束的输入字符

注：

“为何我的 root 会收到系统 crontab 寄来的错误讯息呢？”

改进方法：2> errorfile

命令执行的判断依据：; && ||

△ <cmd>; <cmd>

△ $?（指令回传值）与 && 或 ||

“两个指令之间有相依性，而这个相依性在于前一个指令执行的结果是否正确”

<cmd1> && <cmd2>：若 <cmd1> 执行完毕且正确执行（$?=0），则开始执行 <cmd2>；
若 <cmd1> 执行完毕且为错误（$?≠0），则 <cmd2> 不执行。

<cmd1> || <cmd2>：若 <cmd1> 执行完毕且正确执行（$?=0），则 <cmd2> 不执行；
若 <cmd1> 执行完毕且为错误（$?≠0），则开始执行 <cmd2>。

例：不清楚 /tmp/flowers 是否存在，但就是要建立 /tmp/flowers/jasmine 文件

```
ls /tmp/flowers || mkdir /tmp/flowers && touch /tmp/flowers/jasmin
```

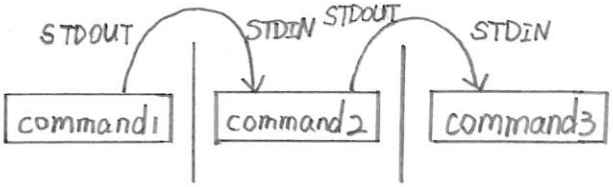
例：用 ls 命令判断 /tmp/zapdos 是否存在，并输出 "exist" 或 "not exist"

```
ls /tmp/zapdos && echo "exist" || echo "not exist"
```

这里的两个 echo 不能颠倒

管线命令（pipe）

管线命令能且仅能处理经由前面一个指令传来的正确信息，也就是 standard output 的信息，对于 standard error 并没有直接处理的能力。



注：

每个管线后面接的必定是指令，且这个指令必须能够接受 standard input 的数据！

可以接受 standard input 的命令：less more head tail 等

不可接受 standard input 的命令：ls cp mv 等

如果硬要让 standard error 可以被管线命令利用，则可通过数据流重导向实现。"2>&1"

撷取命令：cut，grep

```
cut -d '<分隔字符>' -f <fields>
```

找到第几个，例如 1 或 3,5

相当于 python 的 split('<分隔字符>')

```
cut -c <字符区间>
```

例如：

```
12-20  # 第12到第20个字符
12-    # [12:]
```

```
grep [-acinv] [--color=auto] '<搜寻字符串>' <file or stdin>
```

[-c] 计算找到 <搜寻字符串> 的次数

[-i] 忽略大小写

[-n] 输出行号

[-v] 反向选择，即显示出没有 <搜寻字符串> 的行

排序与统计：sort，wc，uniq

```
sort [-fnr] <file or stdin>
```

[-f] 忽略大小写

[-n] 使用纯数字进行排序（默认是以文字型态来排序）

[-r] 反向排序

uniq [-ic]　　# 排序完成后，将重复的资料仅列出一个显示

☆ 当重复的行并不相邻时，uniq命令是不起作用的

[-i] 忽略大小写

[-c] 进行计数

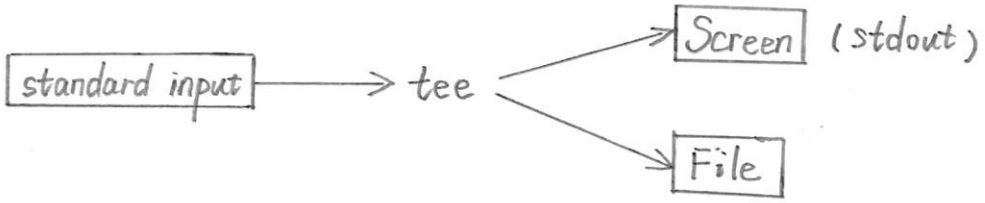
wc [-lwm] <file or stdin>

[-l] 统计出有多少行

[-w] 统计出有多少字

[-m] 统计出有多少字符

双向重导向：tee



tee [-a] <file>

[-a] 以累加的方式，将数据加入<file>当中

字符转换命令：tr, col, join, paste, expand

tr 可以用来删除文字(-d)，或是进行替换(-s)

col -x 可以将 tab 键转换成对等的空格键

以下略

分区命令：split

"将一个大文件，依据文件大小或行数，来分成若干小文件"

用法略

参数代换、xargs

产生某个指令的参数，

xargs 可以读入 stdin 的数据，并且以空格或者换行符将 stdin 分隔成 arguments.

xargs [-pn] <command>

[-p] 在执行每个指令的 argument 时，都会询问使用者

[-n] 后接一个数字，每次<command>指令执行时，会使用几个参数的意思

☆ 会使用 xargs 的原因：很多指令并不支持管线命令，因此需要通过 xargs 来使得指令可使用 stdin

减号 - 的用途

在管线命令当中，常常会使用前一个指令的 stdout 作为这次的 stdin，

某些指令要用到文件名(例如 tar)，该 stdin 与 stdout 可以利用减号"-"来代替。