Power Shell：A

目錄

1. 認識 Shell 與 Shell Script

在 Linux 系統中可以分成三層：

- Kernel（核心）：直接與硬體溝通，負責系統資源管理。
- Shell（殼層）：接收使用者輸入的指令，把它轉交給 Kernel 執行，再把結果顯示給使用者。
- File System Structure（檔案系統結構）：管理檔案和資料夾。

簡單說：Shell 是使用者與核心之間的「翻譯官」。

1.1 Shell Script

Shell 也可以寫成腳本檔案（.sh），一次執行多行指令。

用途：自動化工作、系統管理、定時排程

Shell Script 用 Shell 語法編寫，執行時由 Shell 解譯。

1.2 Linux 中常見的 Shell：

- Bourne Shell (sh)：最早的 Unix Shell
- C Shell (csh)：語法類似 C 語言
- Korn Shell (ksh)：結合 Bourne 與 C Shell 優點
- Bourne Again Shell (bash)：Linux 預設 Shell，功能最完整
- T shell (tcsh)：加強版 C Shell

2. 認識 Bash

Bash 全名 Bourne-Again Shell，是 GNU 專案開發的 Unix Shell。

Bash 是最常見的 shell，也是許多 Linux 系統的預設 shell。

2.1 Bash 的環境變數

Bash 啟動時會載入一組環境變數，影響系統運作與使用者操作。

# 查看變數 	
echo $變數名
	# 設定變數（只對當前 session 有效）
	# 如果要永久生效：修改 ~/.bashrc 或 /etc/profile
	變數名=值
	export 變數名

2.2 常見環境變數

變數名	功能
PATH	命令搜尋路徑（用 : 分隔）
HOME	使用者家目錄
USER	使用者名稱
SHELL	當前使用的 Shell 路徑
PS1	主提示符號格式
HISTSIZE	歷史指令記錄數量
LANG	語系設定
PWD	當前工作目錄
RANDOM	產生隨機數
UID	使用者 ID

範例：查看查看 Bash 版本 echo $BASH_VERSION

3. 變數

要加上「$」符號，等號左右不可以有空格。

3.1 範例：初始化變數 hello

```
# !/bin/bash
hello="Hello World"
echo $hello
```

3.2 範例：變數沒有一定資料類型

```
# !/bin/bash
t=100              # t 是整數
let "t+=23"        # 整數運算，t = t + 23
```

```
echo "t = $t "     # 印出 t=123

s=AB3
echo "s = $s" # s = AB3
```

3.3 範例：空變數自動轉成整數

```
z=""             # 空字串
echo "z = $z"    # 會輸出 z =
let "z += 1"     # 嘗試用整數運算，空字串會當 0
echo "z = $z"    # 變成 z=1
```

3.4 區域變數

```
#!/bin/bash
# 區域變數

hello="hello world"

# 使用函數
function hello01(){
    local hello="這是 hello01"
    echo $hello
}

function hello02(){
    local hello="這是 hello01"
    echo $hello
}

echo $hello
hello01
hello02
```

3.5 參數變數

在 Shell Script 中的參數,會自動變成程式內的變數。

變數	意義
$0	腳本名稱
$1	第一個參數
$2	第二個參數
$3	第三個參數
$@	可以在迴圈中安全地取得每一個參數（遇到空白不會被切掉）。

Power Shell：A

$*	跟 $@ 很像，但會把全部參數當成一個字串。

```
#!/bin/bash
echo "\$0 = $0"
echo "\$1 = $1"
echo "\$2 = $2"
echo "\$3 = $3"
echo "\$@ = $@"

# 執行方式 bash test.sh arg1 arg2 arg3
# $0 = test.sh
# $1 = arg1
# $2 = arg2
# $3 = arg3
# $@ = arg1 arg2 arg
```

3.6 接收傳遞 script 或函式的參數 (Receiving Arguments)

執行腳本時，可以在檔名後面加上參數，腳本內部可以用特殊變數來接收。

- $#: 代表傳入參數的總個數。
- $1, $2, ...: 依序代表第一個、第二個...參數。
- shift: 這個指令可以讓參數列表往前移。例如執行 shift 後，原本的 $2 會變成 $1，原本的 $3 會變成 $2，以此類推。

```
#!/bin/bash

echo "Number of arguments: $#"

# 當還有參數存在時，就繼續迴圈
while [ "$#" -gt "0" ]; do
    echo "第一個參數是: $1"
    shift  # 將參數往前移
done

> bash test.sh a b c d
# Number of arguments: 4
# 第一個參數是: a
# 第一個參數是: b
# 第一個參數是: c
# 第一個參數是: d
```

3.7 取消定義變數 (Unsetting Variables)

當你不再需要一個變數時，可以使用 unset 指令將它從記憶體中完全移除。這比將它設為空字串 (var="") 更徹底，因為 unset 後，變數本身就不存在了。

```
#!/bin/bash
my_secret="TOP_SECRET_INFO"
echo "執行 unset 之前: $my_secret"
unset my_secret # 取消定義該變數
echo "執行 unset 之後: $my_secret" # 再次嘗試存取，會得到空值

if [ -z "$my_secret" ]; then
  echo "變數 'my_secret' 已經不存在了。"
fi
```

3.8 查詢變數是否被定義 (Checking if a Variable is Set)

使用 -n 和 -z 運算子，通常與 if 條件式搭配使用

```
if [ -n "$username" ]; # 檢查變數的字串長度是否 "不為零"
if [ -z "$user_email" ]; # 檢查變數的字串長度是否 "為零"
```

```
#!/bin/bash

# 情況 1: username 有被設定值
username="gordon"
if [ -n "$username" ]; then
  echo "變數 'username' 已定義，值為: $username"
fi

# 情況 2: user_email 未被定義
if [ -z "$user_email" ]; then
  echo "變數 'user_email' 是空的或未定義。"
fi

# 情況 3: user_id 被定義為空字串
user_id=""
if [ -z "$user_id" ]; then
  echo "變數 'user_id' 是空的。"
fi
```

3.9 在變數未被定義時，使用預設值或顯示錯誤

Shell 提供「參數擴展」(Parameter Expansion)，可以讓你用非常簡潔的語法處理變數未定義的情況。

```
${VARIABLE:-default_value}
```

Power Shell：A

```
#!/bin/bash

# 1. color 未定義，使用預設值 'blue'
echo "你最喜歡的顏色是 ${color:-blue}。"
echo "檢查變數本身: color 的值是 '$color'" # 變數 color 依然是空的

echo "---"

# 2. color 有被定義，使用它自己的值
color="red"
echo "你最喜歡的顏色是 ${color:-blue}。"
```

3.10 將檔案的內容寫入變數 (Assigning File Content to a Variable)

```
# 要先建立一個檔案 text.txt，裡面隨便寫些東西
#!/bin/bash

animal=`cat text.txt`
echo $animal
```

3.11 將變數的值設為唯讀 (Read-only Variables)

一旦變數被設為唯讀 (readonly)，它的值就不能再被修改或 unset。

```
#!/bin/bash

readonly myname="jack"
myname="sue" # 任何嘗試修改唯讀變數的行為都會導致錯誤
```

4. 運算子

$? 會儲存上一個指令的退出碼（Exit Status）。

0 代表指令執行成功，非 0 代表失敗或條件不成立。

範例：條件成立	範例：條件不成立

`#!/bin/bash`
`[ 3 -lt 2 ]  # 判斷 3 是否小於 2`
`echo $?  # 輸出 1（不成立）`
	`#!/bin/bash`
	`[ 3 -lt 2 ]  # 判斷 3 是否小於 2`
	`echo $?  # 輸出 1（不成立）`

Bash 的數字比較運算子（在 [ ] 或 test 裡用）

比較符號	意義
-eq	等於
-ne	不等於

Power Shell：A

-lt	小於
-le	小於等於
-gt	大於
-ge	大於等於

Bash 的字串比較運算子

比較符號	意義
=	相等
!=	不相等
-z s	長度為 0（空字串）
-n s	長度非 0

```
#!/bin/bash
x=10
y=20
if [ $x -lt $y ]; then
    echo "$x 小於 $y" # 10 小於 20
fi
```

```
#!/bin/bash
name="Amy"
if [ "$name" = "Amy" ]; then
    echo "名字一樣"
fi
```

檔案測試運算子

運算子	意義
-e file	存在
-f file	是一般檔
-d file	是目錄
-s file	檔案大小 > 0
-r file	可讀
-w file	可寫
-x file	可執行

檔案與目錄的比較

運算子	意義
file1 -nt file2	file1 比 file2 新

Power Shell：A

file1 -ot file2	file1 比 file2 舊
file1 -ef file2	兩者指向同一檔案（硬連結）

5. 判斷式

5.1 if 判斷語法

```
if [ 條件 1 ]; then
    動作 1
elif [ 條件 2 ]; then
    動作 2
elif [ 條件 3 ]; then
    動作 3
else
    動作 4
fi
```

5.2 範例：密碼比對

```
#!/bin/bash
I_PASSWORD="abc123"  # 預設密碼

echo "Please enter the password:"
read PASSWORD         # 讀取使用者輸入

if [ "$PASSWORD" == "$I_PASSWORD" ]; then
    echo "Welcome login!"
else
    echo "ACCESS DENIED!"
fi
```

5.3 範例：成績判斷 (if-elif-else)

```
#!/bin/bash
echo "Please enter your score (0-100):"
read grade

if [ "$grade" -lt 0 ] || [ "$grade" -gt 100 ]; then
    echo "Sorry, you are out of range."
elif [ "$grade" -lt 60 ]; then
    echo "Not pass."
elif [ "$grade" -lt 80 ]; then
    echo "Good!"
```

```
elif [ "$grade" -le 90 ]; then
    echo "Great!"
else
    echo "Excellent!"
fi
```

6. 基本的數值運算

6.1 算數擴展運用

這個技巧介紹 Bash 內建的算數擴展功能，使用 $(()) 語法來進行整數運算。

- $((...))：在括號內可以寫算數表達式，Shell 會自動計算結果並返回。支援的運算符號包括 +（加）、-（減）、*（乘）、/（除）、%（取餘數）等。

```
#!/bin/bash
a=$((3+2))
b=$((6-3))
c=$((4*5))
d=$((10/3))
e=$((10%3))
echo $a $b $c $d $e

a=$((a+b))
b=$((c-b))
echo $a $b
```

6.2 位元左移或右移

- $((...))：位元運算同樣在算數擴展的括號內執行。
- <<：左移，相當於乘以 2 的次方。例如 10<<1 是 10 * 2^1 = 20。
- >>：右移，相當於除以 2 的次方（取整數）。例如 10>>1 是 10 / 2^1 = 5。

```
#!/bin/bash
a=$((2<<3))
b=$((2>>1))
echo $a $b
```

6.3 and 與 or 的用法

- &：位元 AND，只有當兩個位元都是 1 時結果才為 1。
- |：位元 OR，只要其中一個位元為 1 時結果就為 1。

```
#!/bin/bash
a=$((8&12))
b=$((8|12))
echo $a $b
```

6.4 let 的用法

let 也是一個內建命令，用於進行算術運算。

let：後面直接跟算術表達式，不需要 $ 符號。

```
#!/bin/bash
let i=1+4
echo $i
```

6.5 產生亂數

這個技巧介紹如何產生一個範圍內的亂數。

- $RANDOM：一個特殊的 Shell 變數，每次存取時都會產生一個介於 0 到 32767 的亂數。
- $((RANDOM % N + 1))：產生介於 1 到 N 之間的亂數。

```
#!/bin/bash
for i in 1 2 3 4 5
do
    echo $((RANDOM % 10 + 1))
done
```

7. 控制流程

7.1 控制流程基礎

```
# tty -s 是 Linux 中的 tty 命令的一個選項，用來靜默地檢查標準輸入是否連接到終端設備。這個選項不會輸出任何文字，只會根據結果回傳不同的退出碼。
if tty -s; then
    echo "Running in a terminal"
else
    echo "Not running in a terminal"
fi
```

7.2 在 IF 中寫多條件

```
#!/bin/bash
hour=`date +%H`
if [ $hour -ge 5 -a $hour -lt 12 ]then;
    echo "good morning"
elif [ $hour -ge 12 -a $hour -lt 17 ]then;
    echo "good afternoon"
elif [ $hour -ge 17 -a $hour -lt 23 ]then;
    echo "good evening"
else
```

```
    echo "You have go to bed"
fi
```

7.3 在 IF 條件逆轉

```
#!/bin/bash
if ! true;then
    echo "true"
else
    echo "false"
fi
```

7.4 使用&&與||處理

&&	指令&&指令	左邊指令回傳 0 時，右邊自動執行
||	指令||指令	左邊指令回傳 1 時，右邊自動執行

```
#!/bin/bash
# -e 是 test 指令的一個選項，代表 "exist"（存在）
test -e test.sh && echo "Yes, it exist"
test -e no.file || echo "No, it does not exist"
```

7.5 case 判斷語法

```
case $變數 in
    模式 1) 動作 1 ;;
    模式 2) 動作 2 ;;
    模式 3) 動作 3 ;;
    *)      預設動作 ;;
esac
```

7.6 範例：輸入 1~5 對應文字

```
#!/bin/bash
echo "Enter a number between 1 to 5:"
read NUM

case $NUM in
    1) echo "one" ;;
    2) echo "two" ;;
    3) echo "three" ;;
    4) echo "four" ;;
    5) echo "five" ;;
```

```
    *) echo "INVALID NUMBER!" ;;
esac
```

7.7 使用 case 根據字串模式進行判斷

```
#!/bin/bash
read -p "enter your answer: " answer
case "$answer" in
    [Yy]*) # 如果 answer 的開頭是 Y 或 y（例如 yes, Yup, y），就輸出
        echo "your answer is yes";;
    [Nn]*) - 如果 answer 的開頭是 N 或 n（例如 no, Nah, n），就輸出
        echo "your answer is no";;
    *)
        echo "your answer is unknow";;
esac
```

8. 迴圈

8.1 for 迴圈

```
for 變數 in 值1 值2 值3 ...
do
    指令
done
```

```
for name in Alice Bob Carol
do
    echo "Hello, $name"
done
```

8.2 while 迴圈

```
while [ 條件判斷式 ]
do
    指令
done
```

```
count=1
while [ $count -le 5 ]
do
    echo "count = $count"
```

```
    let count++
done
```

8.3 在條件成立期間內，執行迴圈內容

```
#!/bin/bash
while [ -n "$1" ] # 判斷 $1 是否為非空。
do
    echo "$1" # 輸出
    shift # 把所有參數往左移一位，讓 $2 變成 $1，$3 變成 $2，依此類推。
done
# bash test.sh 7 8 9
```

8.4 無窮迴圈的設計

```
while :
do
    echo "running"
    sleep 2
done
```

8.5 使用 while 計算平方和：

輸入 3 時，迴圈會計算 9+4+1=14。

當輸入 5 時，迴圈會計算 25+16+9+4+1=55。

```
#!/bin/bash

input=$1
value=0
while [ $input -gt 0 ]
do
    value=$(($value+$input*$input))
    input=$(($input-1))
done
echo $value

# bash test.sh 3
# bash test.sh 5
```

8.6 跳到下一次的迴圈執行：continue

```
#!/bin/bash
for i in 1 2 3 4 5 6 7 8 9 10
do
    rem=$(($i % 2))
```

```
    if [ $rem -eq 0 ]
    then
        continue
    fi
    echo "$i is odd"
done
```

8.7 中斷迴圈進

8.8 行下一步處理：break

```
#!/bin/bash
for i in 1 2 3 4 5 stop 6 7 8 9
do
    if [ "$i" = "stop" ]
    then
        break
    fi
    echo "$i"
done
```

8.9 對所有的參數進行重複處理

```
#!/bin/bash
for i in "$@"
do
    echo "list"
    echo "$i"
done
# or i in "$@" 會遍歷所有命令行參數。每次迭代都會執行 echo "list" 和 echo "$i"。
```

8.10 對指令的輸出結果做重複處理

```
#!/bin/bash
for name in `ls`
do
    if [ -d "$name" ]
    then
        echo "目錄: $name"
    elif [ -f "$name" ]
    then
        echo "檔案: $name"
    else
        echo "其他: $name"
    fi
```

```
done
```

8.11 對檔名做重複處理

* 是萬用字元，代表所有檔案和目錄名稱。這是一種更常見的遍歷方法。

```
#!/bin/bash
for name in /etc/*
do
    if [ -f "$name" ]
    then
        echo "檔案: $name"
    elif [ -d "$name" ]
    then
        echo "目錄: $name"
    else
        echo "其他: $name"
    fi
done
```

8.12 在背景進行重複處理

這個技巧教您如何讓一個迴圈在背景執行，不阻塞當前的終端機。

```
#!/bin/bash
for i in 1 2 3 4
do
    echo "$i"
    sleep 1
done &
sleep 5
```

8.13 將重複處理的結果存為檔案

```
#!/bin/bash
for name in *
do
    echo "$name"
done > out.txt
```

8.14 使用 select 顯示選單並做迴圈處理

select 是一個非常方便的命令，用於在腳本中創建互動式選單。

- select item in list：select 會將 list 中的每個項目都顯示為一個帶編號的選單，並等待使用者輸入。
- $REPLY：使用者輸入的編號會被儲存在 $REPLY 變數中。

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

```
PS3="Please make a selection -> "
select color in red orange yellow green blue purple quit
do
    case $color in
        red|yellow|blue)
            echo "Pure Color"
            ;;
        orange|green|purple)
            echo "Mixed Color"
            ;;
        quit)
            break
            ;;
        *)
            echo "ERROR: Invalid selection"
            ;;
    esac
done
```

9. 函數

9.1 定義函式 (Defining a Function)

```
函式名稱() {
    # 要執行的指令...
    echo "You are a boy"
}
```

9.2 範例：使用函數

```
#!/bin/bash
myfunction() {
    echo "you are a pretty girl"
}

myfunction # you are a pretty girl
```

9.3 取消定義函式 (Unsetting a Function)

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

```

# 定義函式
funtion() {
    echo "you are a boy"
}

funtion # 第一次呼叫，正常執行
unset -f funtion # 取消函式定義

# 第二次呼叫，會發生錯誤
funtion # test.sh: line 15: funtion: command not found
```

9.4 在呼叫函式時傳遞參數 (Passing Arguments to a Function)

```
#!/bin/bash
echo() {
    echo "總共有 $# 個參數"
    echo "第一個是: $1"
    echo "第二個是: $2"
    echo "第三個是: $3"
}

# 呼叫函式並傳遞三個參數
echo YA NO YO
```

9.5 在函式內部取得結果 (Getting a Return Value from a Function)

```
#!/bin/bash

# 這個函式用來判斷今天是不是星期一
ismonday() {
    # `date +%A` 會輸出今天的星期幾 (例如: Monday)
    if [ "$(date +%A)" == "Monday" ]; then
        return 0  # 如果是星期一，回傳 0 (成功)
    else
        return 1  # 如果不是，回傳 1 (失敗)
    fi
}

# 呼叫 ismonday 函式
ismonday

```

```
# `$?` 變數會儲存上一個指令 (也就是 ismonday) 的回傳值
if [ $? -eq 0 ]; then
    echo "今天是星期一，工作順利"
else
    echo "今天不是星期一"
fi
```

9.6 遞迴：呼叫函式本身 (Recursive Function)

```
#!/bin/bash

# 計算階乘的函式
CallFunction() {
    if [ $1 -eq 1 ]; then
        Value=1
    else
        # 呼叫自己，但參數減 1
        CallFunction $(($1 - 1))
        Value=$(($1 * $Value))
    fi
}

# 計算 5!
CallFunction 5
echo $Value
```

9.7 呼叫其他檔案中定義的函式 (Sourcing Functions from Other Files)

你可以將常用的函式整理在一個獨立的檔案中，然後在需要使用的腳本裡，透過 source 指令或 . 指令將它載入進來。這樣可以提高程式碼的重複使用性。

```
├── a.sh
└── test.sh(主檔案)
```

```
# a.sh
#!/bin/bash

a123() {
    echo "pig"
}
```

```
# test.sh
#!/bin/bash
```

```
. ./a.sh # 要空格歐

a123
echo "dog"
```

10. 使用字串 (Using Strings)

10.1 寫一個橫跨多行的字串 (Creating a Multi-line String)

有時候你需要處理的字串會跨越多行。常見的方法是使用 cat 搭配 << (Here Document)。

cat <<'結束標記' 的語法可以讓你輸入多行文字，直到你再次輸入 結束標記 為止。所有在此之間的內容，都會被 cat 指令輸出，然後可以透過指令替換 (`...` 或 $(...)) 將這些多行內容存入一個變數中。

```
#!/bin/bash

# 使用 Here Document 將多行文字存入變數 str
# 'EOF' 是一個自訂的結束標記，可以是任何你喜歡的詞
# 結束標記前後都不能有任何空格
str=$(cat <<'EOF'
a/b/c/d/apple/e/tomato/g
a/brother/sister/g
apple
EOF
)

echo "$str" # 印出這個多行變數的內容

# test.sh: line 15: warning: here-document at line 11 delimited by end-of-file (wanted
`EOF')
# a/b/c/d/apple/e/tomato/g
# a/brother/sister/g
# apple
```

10.2 跳脫字元使用方法 (Using Escape Characters)

跳脫字元 (Escape Character)，通常是反斜線 \，它能讓緊跟在它後面的那個特殊字元失去其特殊意義，變回一個普通的字元。echo 指令搭配 -E (預設) 或 -e 選項時，對跳脫字元的處理會不同：

- echo (或 echo -E): 不會解釋跳脫序列，\ 就只是一個 \。
- echo -e: 會解釋跳脫序列，例如 \n 會變成換行，\t 會變成 Tab。

```
#!/bin/bash

# 使用 -e 來讓 \t 被解釋為 Tab 鍵
echo -e "Hello\tWorld" # Hello   World
```

```
# 不使用 -e，\t 就只是普通字元
echo "Hello\tWorld" # Hello\tWorld
```

10.3 字串切片

取出字串中的第 M 個字元到第 N 個字元的內容 (Extracting a Substring)

```
${變數:起始位置:長度}
```

```
#!/bin/bash

word=applebanana

# 從第 2 個位置 (第三個字元 'p') 開始，取 5 個字元
getout=${word:2:5}

echo $getout # pleba
```

11. 檔案內的字串處理

11.1 查詢字串的長度 (Shell 內建功能)

```
#!/bin/bash
word="rjiwerjqwerjij ijirjeriqwjei ije iji"
echo ${#word}
```

11.2 轉換字串的大小寫 (使用 tr)

```
word=wergfergtgierjirtekgweqj
# 將所有小寫 a-z 轉換成大寫 A-Z
echo "$word" | tr 'a-z' 'A-Z'
```

11.3 查詢檔案內的字串 (使用 grep)

從檔案或輸出中，找出包含指定關鍵字的那些行。

- -r (recursive): 遞迴搜尋，會連同指定目錄下的所有子目錄一起找。
- -i: 忽略大小寫。
- -v: 反向查找，只顯示**不包含**關鍵字的行。
- -n: 顯示行號。

```
#!/bin/bash
# 在 /etc/ 目錄下，遞迴地找出所有包含 "network" 字串的行
grep -r network /etc/*
```

11.4 替換字串 (使用 sed)

```
sed 's/要被取代的字串/新的字串/選項'
```

常用選項 (g)：

- 如果沒有 g (global)，sed 只會取代每一行中**第一個**匹配到的字串。
- 如果加上 g，則會取代每一行中**所有**匹配到的字串。

```
#!/bin/bash
paper="something"

# 只取代第一個 "some"
echo "$paper" | sed "s/some/nothing/"

# 取代所有 "some"
echo "$paper" | sed "s/some/nothing/g"
```

11.5 抽取出字串 (使用 awk)

```
# -F':' 指定分隔符為冒號
# '{print $1, $6}' 印出第一個欄位 (帳號) 和第六個欄位 (家目錄)
# 處理 /etc/passwd 檔案
awk -F':' '{print $1, $6}' /etc/passwd
```

11.6 抽取出字串 (使用 cut)

cut 是一個比 awk 更輕量級的工具，專門用來 "切" 出指定的欄位或字元。

- -c: 按字元 (character) 位置來切。
- -f: 按欄位 (field) 來切。
- -d: 搭配 -f 使用，指定欄位的分隔符號。

```
# -c 1-3: 切出每一行的第 1 到 3 個字元
cut -c 1-3 /etc/passwd

# -d':' -f 1,6: 以 : 為分隔符，切出第 1 和第 6 個欄位
cut -d':' -f 1,6 /etc/passwd
```

12. 檔案的欄位處理 (Field Processing in Files)

主要圍繞著 awk 這個強大的工具，教我們如何像處理 Excel 表格一樣，對以欄位分隔的文字檔進行各種操作。

12.1 更換逗點分隔的欄位順序

當你的資料是以逗號 (,) 分隔時，awk 預設就能正確處理。這個技巧展示如何調換欄位的輸出順序。

```
test="10240246,john,male,10"

# 使用 echo 傳資料給 awk
# -F',' 指定用逗號當分隔符號
# '{print $2, ",", $1, ",", $3}' 調換第 1 和第 2 欄位的順序，並用逗號隔開
```

```
echo "$test" | awk -F',' '{print $2 "," $1 "," $3}'
```

12.2 增加一個固定的欄位

在現有的欄位後面，加上一個新的、固定的或是動態產生的欄位。

```
test="10240246,john,male,10"

# 使用 echo 傳資料給 awk
# -F',' 指定用逗號當分隔符號
# '{print $2, ",", $1, ",", $3}' 調換第 1 和第 2 欄位的順序，並用逗號隔開
# 在原本的欄位後面，加上一個由 date 指令產生的日期欄位
# `date +'%Y%m%d'` 會先被 shell 執行，得到結果 (如 20250812)
# 然後 awk 才執行，將這個結果作為一個固定字串印在最後

echo "$test" | awk -F',' '{print $1, ",", $2, ",", $3, ",", "'$(date +'%Y%m%d')'"}'
```

12.3 變更欄位的分隔符號

改變輸出的欄位分隔符號，例如將原本的逗號分隔，改為用空格分隔。

```
test="10240246,john,male,10"

# -v OFS=' ' 設定輸出的欄位分隔符號 (Output Field Separator) 為一個空格
echo "$test" | awk -F',' -v OFS=' ' '{print $1, $2, $3, $4}'
```

12.4 取出符合條件的內容

awk 最強大的功能之一：根據特定欄位的內容來篩選要處理的行。

13. 檔案操作技巧

13.1 從檔名取得檔名或資料夾名稱

在 Shell Script 中，如果你從一個完整的路徑中取得檔名，可以使用 basename 和 dirname 這兩個指令。

- basename <路徑>：會取出路徑中的檔名部分。
- dirname <路徑>：會取出路徑中的資料夾部分。

```
#!/bin/bash
pathname="/etc/security/limits.conf"

filename=$(basename "$pathname")
dirname=$(dirname "$pathname")

echo "檔名: $filename"
echo "資料夾路徑: $dirname"
```

13.2 產生內容為空的檔案

```
# 產生一個名為 "-abc" 的檔案
touch -- -abc
```

13.3 變更檔案的更新日期

touch 指令最主要的功能是變更檔案的存取或修改時間。

- touch -a：只改變檔案的存取時間。
- touch -m：只改變檔案的修改時間。
- touch -t <時間>：將檔案的時間設定為特定值。時間格式為 [[CC]YY]MMDDhhmm[.ss]。

```
# 將 myfile.txt 的修改時間變更為 2016 年 2 月 10 日 22:00
touch -t 201602102200 myfile.txt
```

13.4 使用暫存檔

在 Shell Script 中，有時需要暫時儲存一些中間結果。mktemp 指令可以幫助你產生一個獨一無二的暫存檔，避免檔名衝突。

```
#!/bin/bash
tmpfile=$(mktemp)
echo "這個是暫存檔案: $tmpfile"

echo "這是暫存內容" > "$tmpfile"
cat "$tmpfile"

# 程式結束後，通常會將暫存檔刪除
rm "$tmpfile"
```

13.5 查詢指令所在的位置

- which <指令>：查詢指令的可執行檔案路徑。
- whereis <指令>：查詢指令的二進位檔案、原始碼和 man page 的位置。

```
which ls
# 執行結果：/usr/bin/ls

whereis ls
# 執行結果：ls: /usr/bin/ls /usr/share/man/man1/ls.1.gz
```

13.6 列出資料夾內的檔案

```
#!/bin/bash
dir="/etc/ssh"  # 定義要搜尋的資料夾
set -- $(ls "$dir") # 將 ls 的結果設定為腳本的參數
number_of_files=$#   # 取得檔案數量

```

```
echo "directory: $dir"
echo "number of files: $number_of_files"
echo "list of files:"

for file in "$@"; do
    echo "$dir/$file"
done
```

13.7 搜尋檔名與檔案內容

- grep：用於在檔案內容中搜尋指定的字串。
- find：用於在指定的資料夾中搜尋符合條件的檔案。

```
# 在目前資料夾及其子資料夾中，搜尋檔名中包含 ".sh" 的檔案
find . -name "*.sh"

# 在 /usr/bin 資料夾中，搜尋檔案內容中包含 "grep" 字串的檔案
grep -r "grep" /usr/bin/
```

13.8 找出檔名與樣式符合的檔案

find 指令支援萬用字元（如 *）來進行更彈性的檔名搜尋。

```
# 在 /usr/share/doc 資料夾中，搜尋所有以 "find" 或 "grep" 開頭的文字檔
find /usr/share/doc -name "find*" -o -name "grep*"
```

13.9 讀入文字資料

在 Shell Script 中，我們可以使用 while 迴圈搭配 read 指令來逐行讀取檔案內容。

```
#!/bin/bash
# 假設我們有一個檔案名為 data.txt，內容如下：
# A 1
# B 2
# C 3

while read letter number; do
    echo "字母是 $letter, 數字是 $number"
done < data.txt
```

13.10 處理 CSV 格式的檔案

CSV 是一種常用的檔案格式。由於其欄位之間是以逗號分隔，我們可以調整 IFS（Internal Field Separator）變數來處理它。

```
#!/bin/bash
# 假設有一個 csv 檔案 records.csv
```

```
# 內容為: name,id
# alice,1
# bob,2

# 將 IFS 設定為逗號
IFS=','

# 讀取 records.csv
while read name id; do
    echo "名字: $name, ID: $id"
done < records.csv
```

14. 日期、時間與時區值 (Date, Time, and Timezone Values)

14.1 日期、時間與時區值 (Date, Time, and Timezone Values)

```
# date: 直接執行，輸出預設格式的系統時間。

date +"%Y/%m/%d %H:%M:%S":

# + 後面接的是格式化字串。
# %Y: 四位數年份 (e.g., 2025)
# %m: 兩位數月份 (01-12)
# %d: 兩位數日期 (01-31)
# %H: 24 小時制的小時 (00-23)
# %M: 分鐘 (00-59)
# %S: 秒 (00-59)
```

14.2 計算指定的日期與時間

```
# 計算「2016/6/8」的「10 天後」是幾號
date -d "2016/6/8 +10 days"

# 計算 100 秒前，是什麼時間
date -d "100 seconds ago"
```