第二章

Linux 核心

在前一章的「作業系統到底有何重要？」一節中，我們知道了作業系統主要的作用，在於將各種硬體抽象化，同時為我們提供 API。這些 API 的程式化功能，讓我們毋需操心程式執行場合及方式，就能寫出應用程式。說穿了，就是核心為程式提供了這樣的 API。

本章會探討何謂 Linux 核心，以及你應該如何以整體方式看待它與相關元件。讀者們會學到 Linux 的整體架構，以及 Linux 核心所扮演的基本角色。本章的主要目的之一就是讓大家理解，儘管核心提供所有的中心功能，但光憑核心自己是無法構成完整的作業系統，它只是構成作業系統最中心的部分。

首先，我們還是從鳥瞰的角度開始，觀察核心如何配合並與底層硬體互動。然後我們會檢視計算的中心概念，並探討各種 CPU 架構、以及它們與核心的關係。接著會近距觀察個別的核心元件，並探討核心為程式所提供、可讓你執行的 API。最後則會觀察如何自訂及擴充 Linux 核心。

本章的目的，在於讓你了解相關的術語，幫你熟悉程式與核心之間的界面，並協助你對核心的功能有基本的認識。本章不打算將你塑造成核心開發達人，或是懂得設定與編譯核心的系統管理員。如果你真有意於此，我會在本章結尾時為你指出幾條路徑。

現在，讓我們躍身進入深水區：Linux 架構和核心在其中扮演的中心角色。

Linux 架構

從高階角度來看，Linux 的架構就如同圖 2-1 所示。你可以將事物概括區分成三個壁壘分明的階層：

硬體

從 CPU、主記憶體到磁碟機、網路界面、以及鍵盤及螢幕監視器之類的周邊裝置等等。

核心

這是本章後續篇幅的重點所在。注意有些元件位於核心與使用者空間中間的灰色地帶，像是 init 系統及系統服務（例如網路）等等，但是嚴格說起來，其實不算是核心的一部分。

使用者空間

這是大部分應用程式運行所在之處，包括作業系統元件，如 shell（第 3 章介紹）、像是 `ps` 或是 `ssh` 之類的工具程式，還有以 X 視窗系統作為桌面的圖形使用者界面等等。

本書會著重在圖 2-1 的上兩層，亦即核心與使用者領域。我們只會在這一章及少數其他章節的相關內容中略為提及硬體層。

Linux 作業系統會替兩個不同層面間的界面做清楚的定義，並納入作為套件的一部分。位於核心和使用者空間中間的，是稱為系統呼叫（*system call*，簡稱 *syscall*）的界面。我們會在第 26 頁的「系統呼叫」小節詳細說明。

而硬體與核心之間的界面則與系統呼叫不同，並非單一的個體。它包含的是一系列的個別界面，通常會按照硬體種類區分：

1. CPU 界面（請參閱第 16 頁的「CPU 架構」）
2. 與主記憶體的界面，在第 21 頁的「記憶體管理」會介紹
3. 網路界面和驅動程式（包括有線和無線網路；請參閱第 23 頁的「網路」）
4. 檔案系統和區塊裝置的驅動程式界面（請參閱 24 頁的「檔案系統」）
5. 字元裝置、硬體中斷、以及裝置驅動程式，像是鍵盤之類的輸入裝置、終端機和其他 I/O 裝置等等（請參閱 24 頁的「裝置驅動程式」）

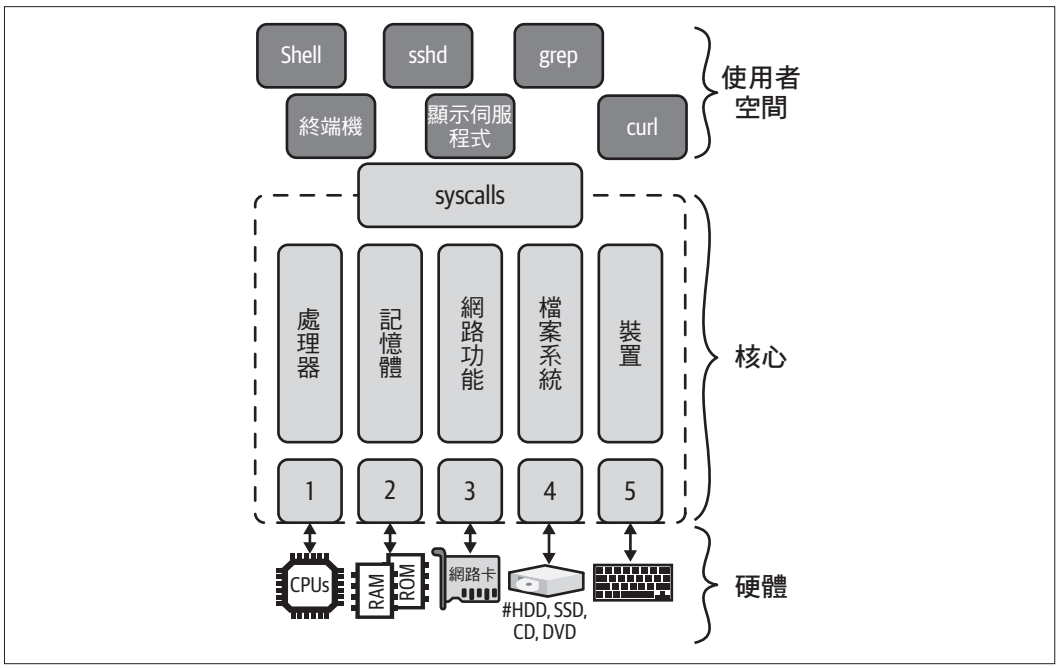

圖 2-1　Linux 架構的高階觀點

如各位所見，許多我們通常視為 Linux 作業系統一部分的事物，像是 shell 或 `grep`、`find` 和 `ping` 之類的工具程式，其實都並非核心的一部分，反而像是你可以下載的應用程式那樣，屬於使用者領域的一部分。

在使用者領域這個主題，你會經常耳聞使用者模式與核心模式這兩個名詞。其實它們代表的則是對於硬體存取的特權等級、以及既有抽象層受限的程度。

一般來說，**核心模式**意味著抽象程度低、執行也較快，而**使用者模式**則代表相對較慢、但也較為安全、而且更為便利的抽象層。除非你是核心開發者，不然幾乎可以完全無視核心模式，因為你所有的應用程式都會在使用者領域執行。另一方面，了解如何與核心互動（第 26 頁的「系統呼叫」）才是關鍵所在，也是我們要考量的部分。

有了這份對於 Linux 架構的概括理解，我們就可以從硬體開始說明了。

第三章

Shells 與 Scripting

在這一章裡，我們要專注在用終端機與 Linux 互動這碼事上，亦即透過 shell 呈現的命令列介面（command-line interface, CLI）。能夠有效地透過 shell 來完成日常作業，是一件至關緊要的事，因此我們要特別專注在其可用性上。

首先我們會複習一些術語，並簡單扼要地介紹 shell 的基礎須知。然後我們會檢視若干近年來更趨友善的 shell，像是 Fish shell 等等。我們也會看到 shell 的組態及其常見的任務。然後我們會繼續介紹，如何以終端機多工器（terminal multiplexer）有效地運用 CLI，讓你可以同時在本地端或遠端操作多個會談。而本章最後會談到如何在 shell 中以指令碼（scripts）將任務自動化，包括如何寫出安全、可攜的指令碼的最佳實務做法，以及如何潤飾（lint）與測試指令碼。

從 CLI 的角度來看，與 Linux 互動的方式主要有兩種。第一種是手動的，亦即人身使用者坐在終端機前，從鍵盤鍵入命令並取得輸出，以此進行互動。這種互動方式適於大部分要在 shell 中進行的日常事務，包括：

- 列舉目錄、尋找檔案、或是搜尋檔案中的內容
- 在目錄間複製檔案、甚至是複製到遠端機器上
- 閱讀電子郵件或新聞，或是從終端機發一篇推文

此外，我們還會學到如何方便有效地同時操作數個 shell 會談。

另外一種操作模式則是透過某種特殊檔案、自動地處理一系列的命令，shell 會為你解譯該檔案的內容、並自行加以執行。這種模式通常稱作 *shell scripting*、或簡稱 *scripting*。

通常你會想用指令碼來代勞特定的重複性任務，而非事必躬親。還有，指令碼是許多組態及安裝系統的基礎。它十分方便。然而如果使用不慎，指令碼也可能十分危險。因此當你考慮撰寫指令碼時，請把圖 3-1 所示的 XKCD 網路漫畫記在心裡。

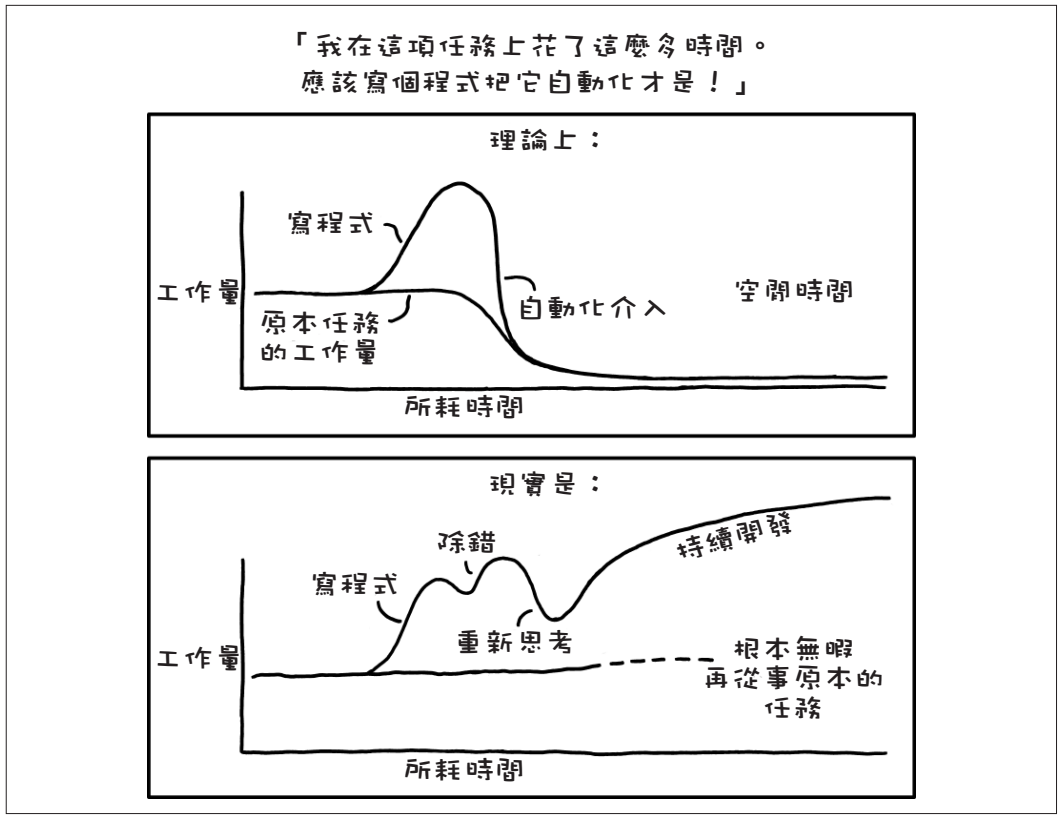

圖 3-1　XKCD 對於自動化的諷刺（*https://oreil.ly/GSKUb*）。原畫是 Randall Munroe（基於 CC BY-NC 2.5 授權引用）

筆者鄭重建議讀者們在手邊準備一套 Linux 的環境，以便隨時實驗本書中的範例。現在大家是否已經準備好動手了呢？如果一切都就緒了，我們就先從術語及基本的 shell 使用開始講起。

基礎

在我們一頭栽進各種選項和設定之前，先來熟悉一下若干基本的術語，例如**終端機**（*terminal*）和 *shell* 之類。在這個小節裡，筆者會定義各項術語、並向讀者們說明如何在 shell 中完成各種日常任務。我們同時也會瀏覽各種現代化的命令，以及它們的實際用途。

終端機

我們先從終端機（terminal）、或者說是終端機模擬器（terminal emulator）、軟體終端機（soft terminal）談起，它們代表的都是同樣的事物：現在提到**終端機**，都是指可以提供文字化使用者介面的程式。也就是說，終端機會從鍵盤讀取輸入，並同時將其顯示在螢幕上。多年以前，這些動作是由同一套整合裝置共同完成的（那時的鍵盤和螢幕還是完整的一組硬體裝置），但如今的終端機都只不過是軟體而已。

除了擔任基本的字元型態輸入與輸出以外，終端機還支援所謂的**跳脫序列**（*escape sequences*），又稱為**跳脫碼**（*escape codes*），便於處理游標及畫面之用、有時還可提供彩色畫面。舉例來說，按下 Ctrl+H 便有倒退鍵的作用，這會刪除游標左側的字元。

而環境變數 `TERM` 則含有正在使用何種終端機模擬器的資訊，其組態可透過以下的 `infocmp` 命令來觀察[1]（注意輸出文字已經過精簡）：

```
$ infocmp ❶
#       Reconstructed via infocmp from file: /lib/terminfo/s/screen-256color
screen-256color|GNU Screen with 256 colors,
        am, km, mir, msgr, xenl,
        colors#0x100, cols#80, it#8, lines#24, pairs#0x10000,
        acsc=++\,\,--..00``aaffgghhiijjkkllmmnnooppqqrrssttuuvvwwxxyyzz{{||}}~~,
        bel=^G, blink=\E[5m, bold=\E[1m, cbt=\E[Z, civis=\E[?25l,
        clear=\E[H\E[J, cnorm=\E[34h\E[?25h, cr=\r,
        ...
```

❶ `infocmp` 輸出的訊息可能一時難以理解。如果讀者們有意理解其功能，請參閱 terminfo[2] 資料庫。舉例來說，以上的輸出便代表支援單頁每行 80 個字（`cols#80`）及 24 行（`lines#24`）的輸出畫面，同時可以有 256 種顏色（寫成 `colors#0x100` 的十六進位註記法）。

1 譯註：若要觀察環境變數 `TERM` 內容，請使用 `echo $TERM`。

2 *https://oreil.ly/qjwiv*

終端機模擬器的例子，並不只限於 `xterm`、`rxvt` 及 Gnome 終端機這幾種，新一代的終端機模擬器甚至可以運用到繪圖處理器（GPU）的功能，像是 Alacritty[3]、kitty[4]、以及 warp[5] 等等。

等大家讀到 61 頁的「終端機多工器」時，我們會再來談談終端機。

Shells

接下來要談談 *shell*，這是一種運作在終端機裡的程式，其任務在於解譯命令。Shell 會以串流（streams）的方式處理輸入與輸出，同時也支援變數，自身還具備若干內建的命令可供操作，它負責處理命令的執行與狀態，通常還支援互動式操作（interactive usage）及指令碼式操作（scripted usage）（ 請參閱第 68 頁的「Scripting」）。

Shell 的正式定義為 `sh`[6]，此外我們也會經常看到 POSIX shell[7] 一詞，當我們談到指令碼及可攜性時，這個名詞會變得更加關鍵。

最早的 `sh` 是採用 Bourne shell，這是以開發者命名的，但如今的 shell 卻常為 bash shell 所取代（這名字是針對原始名稱玩出來的促狹文字，亦即「Bourne Again Shell」（又一種 Bourne shell 之意）），而 bash 已廣泛作為預設 shell 使用。

如果你好奇自己使用的究竟是何種 shell，只需輸入 `file -h /bin/sh` 命令就可得知，如果不行，試試 `echo $0` 或 `echo $SHELL` 也行。

在本小節裡，除非另行指明，我們指的一律都是 bash shell（`bash`）。

`sh` 還有許多實作及變種，像是 Korn shell（`ksh`）和 C shell（`csh`）等等，但如今它們並不常見。第 54 頁的「更為友善的 shell」一節中會再檢視一些更新穎的 `bash` 替代品。

現在我們要來看一下 shell 的兩大基本功能：串流與變數。

3 *https://oreil.ly/zm9M9*
4 *https://oreil.ly/oxyMn*
5 *https://oreil.ly/WBG9S*
6 *https://oreil.ly/ISxwU*
7 *https://oreil.ly/rkfqG*

串流

我們先從輸入（串流）和輸出（也是串流）這兩大主題談起，它們也合稱為 I/O。你要如何為程式提供輸入？又如何控制程式的輸出送往何處（例如顯示在終端機上、或寫入檔案）？

首先，shell 為每個程序（process）都配置了三個預設的檔案描述符（file descriptors，簡稱 FD），專供輸入與輸出使用：

- `stdin`（FD 0）
- `stdout`（FD 1）
- `stderr`（FD 2）

如圖 3-2 所示，這些檔案描述符預設都是分別連接到你的螢幕及鍵盤。換句話說，除非你另行指定，不然你輸入到 shell 的命令必定是從鍵盤取得輸入（`stdin`），而輸出（`stdout`）則是送往螢幕。

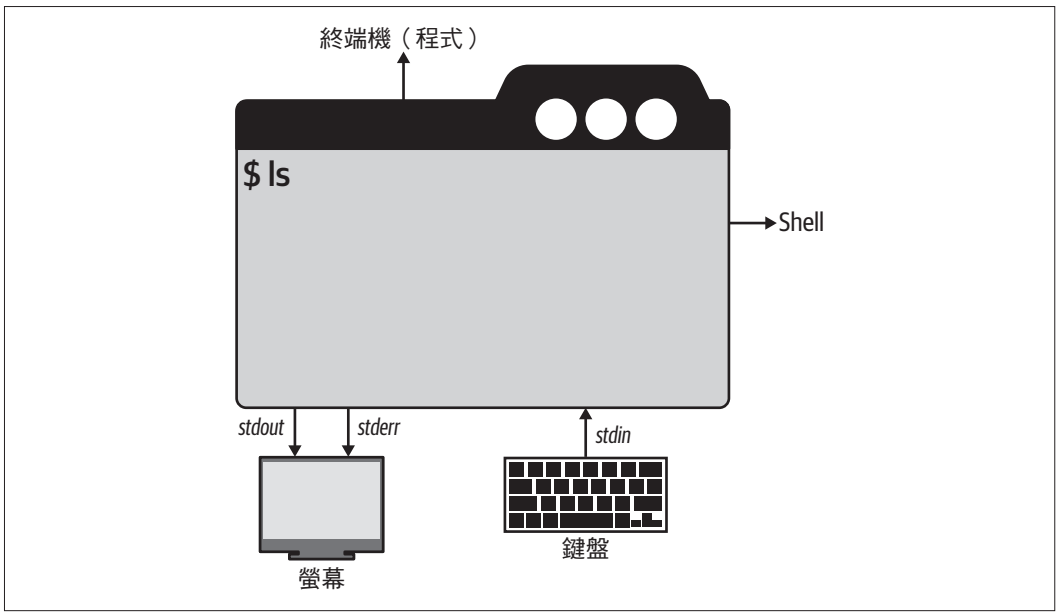

圖 3-2　Shell 的 I/O 預設串流

以下與 shell 的互動便展示了預設的行為模式：

```
$ cat
This is some input I type on the keyboard and read on the screen^C
```

我們在上列的範例中使用了 `cat`，以便觀察其預設行為模式。注意筆者在最後加上了 Ctrl+C（顯示為 `^C`），以便結束命令。

如果你不想按照 shell 提供的預設方式進行，例如說，不想把 `stderr` 的輸出送到畫面上，而是想把它寫到檔案裡，就可以將串流轉向（redirect）。

你可以利用檔案描述符 `$FD>` 和 `<$FD` 將程序的輸出串流轉向。舉例來說，`2>` 代表將 `stderr` 輸出轉向。注意 `1>` 和 `>` 的意思都是一樣的，因為後者是 `stdout` 的預設寫法。如果你想同時將 `stdout` 和 `stderr` 轉向，就寫成 `&>`，如果你想把串流中的訊息棄置不理，將其轉向至 `/dev/null` 即可。

我們來看看以上概念在實際範例是如何運作的，我們試著用 `curl` 下載一些 HTML 的檔案內容：

```
$ curl https://example.com &> /dev/null ❶

$ curl https://example.com > /tmp/content.txt 2> /tmp/curl-status ❷
$ head -3 /tmp/content.txt
<!doctype html>
<html>
<head>
$ cat /tmp/curl-status
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  1256  100  1256    0     0   3187      0 --:--:-- --:--:-- --:--:--  3195

$ cat > /tmp/interactive-input.txt ❸

$ tr < /tmp/curl-status [A-Z] [a-z] ❹
  % total    % received % xferd  average speed   time    time     time  current
                                 dload  upload   total   spent    left  speed
100  1256  100  1256    0     0   3187      0 --:--:-- --:--:-- --:--:--  3195
```

❶ 將 `stdout` 和 `stderr` 都轉向至 */dev/null*，藉此將所有輸出皆棄置不理。

❷ 將輸出和狀態分別轉向，儲存至不同的檔案。

❸ 以互動方式輸入、並將輸入內容存至檔案；此時需以 Ctrl+D 停止捕捉輸入的內容、並結束儲存。

❹ 將所有的字改成小寫，這是透過 `tr` 命令進行的，而命令的輸入是從 `stdin` 讀入的。

Shell 通常都可以理解各種特殊字元的用意，例如：

Ampersand（*&*）

位於命令的最末端，它會把命令放到背景端執行（請參閱第 45 頁的「工作控管」一節）

反斜線（\）

用來延續下一行的命令內容，這是為了方便閱讀冗長的命令設計的

管線（|）

把一個程序的 `stdout` 串接到下一個程序的 `stdin`，這樣一來就可以直接傳遞資料，無須先暫存在檔案中作為傳遞之用。

管線與 UNIX 哲學

雖說管線乍看之下沒什麼，但其實其中大有學問。筆者曾與當初發明管線處理概念的原作者 Doug McIlroy 有過一番饒富趣味的探討。當時我寫了「2018 年對於 Unix 哲學的重新省思」一文（Revisiting the Unix Philosophy in 2018[8]）一文，文中比較了 UNIX 與微服務（microservices）之間的關係。有人在文章下面提出了見解，而那篇文章引出了 Doug 寫給我的一封電子郵件（這令我十分意外，我甚至得求證信件是否真為 Doug 本人所發）以便澄清一些觀念。

同樣地，我們再來觀察一些理論內容在實際中的例子。我們要用 `curl` 下載一個 HTML 檔案，然後用管線將檔案內容轉送給 `wc` 這個工具，藉以計算檔案中總共有幾行文字：

```
$ curl https://example.com 2> /dev/null | \ ❶
  wc -l ❷
46
```

❶ 利用 `curl` 從 URL 下載檔案內容，同時將過程中輸出至 `stderr` 的狀態訊息棄之不理。（注意：其實你可以用 `curl` 的選項 `-s` 來抑制狀態訊息的輸出，但我們總還是要驗證一下新學到的知識嘛。）

❷ `curl` 的 `stdout` 被轉向給 `wc` 的 `stdin`，後者會藉由選項 `-l` 緊接著計算檔案內容的行數。

8 *https://oreil.ly/KTU4q*

現在你對於命令、串流和轉向都有基本的認識了，我們要繼續介紹 shell 的另一項核心功能，亦即對於變數的處理。

變數

在面對 shell 的相關題材時，你會經常遇到的名詞之一，就是變數（*variables*）。每當你不想要（或不能）把某項資料值寫死（hardcode），就可以利用變數來儲存它、並在必要時加以變更。運用的例子如下：

- 當你要處理 Linux 提供的設定項目時，例如 shell 要判斷何處尋找可執行檔時，便會參閱 `$PATH` 變數的內容。這便是一種可供讀寫變數的介面。
- 當你要以互動方式對使用者查詢資料值的時候，例如當指令碼執行到一半、但需要使用者提供某些內容的時候。
- 當你想透過定義一長串的值，例如某個 HTTP API 的 URL，來縮短輸入內容的時候。這個例子相當於程式語言中的常數值，是因為一旦宣告了這類變數，便不會再變更其中資料值的緣故。

我們會將變數分為兩類：

環境變數

: 這是在 shell 間通用的設定值；可以用 `env` 來列舉。

Shell 變數

: 只在當下執行的 shell 中有效；在 bash 中可以用 `set` 來列舉。Shell 變數並不會為子程序所繼承。

在 bash 中，你可以用 `export` 來建立環境變數[9]。當你要取用變數值時，必須在變數名稱前面加上 `$`，如果要解除變數，便以 `unset` 為之。

一下子講太多怕大家吸收不了。我們先來看一些實際的例子（以 bash 為例）：

```
$ set MY_VAR=42 ❶
$ set | grep MY_VAR ❷
_=MY_VAR=42

$ export MY_GLOBAL_VAR="fun with vars" ❸

$ set | grep 'MY_*' ❹
```

9 譯註：C shell 使用 setenv，不過如今的 csh 也少見了。

```
MY_GLOBAL_VAR='fun with vars'
_=MY_VAR=42

$ env | grep 'MY_*' ❺
MY_GLOBAL_VAR=fun with vars

$ bash ❻
$ echo $MY_GLOBAL_VAR ❼
fun with vars

$ set | grep 'MY_*' ❽
MY_GLOBAL_VAR='fun with vars'

$ exit ❾
$ unset $MY_VAR
$ set | grep 'MY_*'
MY_GLOBAL_VAR='fun with vars'
```

❶ 這會建立名為 MY_VAR 的 shell 變數，並賦值為 42[10]。

❷ 列出全部的 shell 變數，並特地挑出 MY_VAR 來顯示。注意 _= 這個字樣，它代表變數並未匯出變成環境變數。

❸ 建立新的環境變數，名為 MY_GLOBAL_VAR。

❹ 列出所有名稱開頭有 MY_ 字樣的 shell 變數。一如預期，我們會看到以上步驟建立的所有變數。

❺ 只列出環境變數。當然這時就只會出現 MY_GLOBAL_VAR。

❻ 建立新的 shell 會談（亦即一個屬於現行 shell 會談的子程序），因而其中不會繼承 MY_VAR 變數。

❼ 取出環境變數 MY_GLOBAL_VAR 來看看。

❽ 再度列出所有 shell 變數，這時便只有 MY_GLOBAL_VAR 會出現了，因為我們此時處於一個子程序當中。

❾ 退出子程序，同時將 shell 變數 MY_VAR 卸除，再度列出 shell 變數試試。這時 MY_VAR 會如同預期般消失無蹤。

10 譯註：用 set 為變數賦值原本是 csh 風格的作法，bash 應該是用 var=value 的方式就可以為變數 $var 賦值。

筆者在表 3-1 中列出了常見的 shell 變數及環境變數。你幾乎在四處都可看到這些變數的蹤影，而它們的用途也很要緊，值得了解一番。對於任何變數，你都可以用 `echo $XXX` 的方式來查閱其中的資料值，而 `XXX` 就是變數名稱。

表 3-1　常見的 shell 變數及環境變數

變數	類型	語境
`EDITOR`	環境	預設用來編輯檔案的程式路徑
`HOME`	POSIX	現行使用者的家目錄路徑
`HOSTNAME`	bash shell	當下所在主機的名稱
`IFS`	POSIX	列出分離欄位用的字元；shell 用來區隔展開的文字字樣
`PATH`	POSIX	其中含有 shell 會去尋找可執行檔（亦即二進位檔案或指令碼檔案）的目錄清單
`PS1`	環境	原始的 shell 命令輸入提示字串
`PWD`	環境	現行工作目錄的完整路徑
`OLDPWD`	bash shell	執行上一個 last cd 命令之前所在目錄的完整路徑
`RANDOM`	bash shell	一個介於 0 到 32767 之間的隨機值
`SHELL`	環境	含有目前使用的 shell 名稱
`TERM`	環境	正在使用的終端機模擬器
`UID`	環境	現行使用者的獨特識別碼（整數）
`USER`	環境	現行使用者的名稱
`_`	bash shell	前一個在前景執行命令的最後一個引數 [11]
`?`	bash shell	退出狀態；請參閱第 44 頁的「退出的狀態」一節
`$`	bash shell	現行程序的識別碼（整數）
`0`	bash shell	現行程序的名稱

此外，你可以參閱 bash 特有變數的完整清單 [12]，此外請注意，表 3-1 中的變數將會在第 68 頁「Scripting」一節中再度登場。

11 譯註：有趣的是，如果前一個命令正好是 echo 變數內容，$_ 取出的便不是作為引數的變數名稱、而是變數的內容。

12 *https://oreil.ly/EIgVc*

退出的狀態

Shell 會利用一種稱為**退出狀態**（*exit status*）的變數來通知執行程式的一方，其命令已經執行完畢。Linux 通常會在命令終止時傳回一個狀態。它可以是正常結束的終止（一切如意）、或是不正常的中止（中途發生問題了）。退出狀態若為 `0`，便代表命令執行成功，而且未曾發生錯誤，若狀態為介於 1 和 255 之間的非零值，便表示有故障發生。若要查詢退出狀態，請用 `echo $?` 來做 [13]。

此外請注意管道中的退出狀態處理，因為有些 shell 只會留存最後一個狀態碼。如果要因應這種限制，請改用 `$PIPESTATUS`[14]。

內建命令

Shell 附帶一組內建命令。其中相當有用的就包括 `yes`、`echo`、`cat` 或是 `read` 等等（看你的 Linux 發行版而定，有些命令可能不是以 shell 的內建形式存在，而是位於 */usr/bin* 之下）。你可以用 `help` 命令將內建命令列出來。但請記住，除此以外的命令都是 shell 的外部程式，通常位於 */usr/bin* 底下（亦即供使用者操作的命令）、或是位於 */usr/sbin* 底下（亦即供管理者操作的命令）。

那麼，你要如何得知在哪裡可以找得到某個執行檔呢？以下便是：

```
$ which ls
/usr/bin/ls

$ type ls
ls is aliased to `ls --color=auto'
```

本書的技術審閱者之一指出，`which` 並非 POSIX 標準，而是屬於外部程式，因此不見得隨處可用。此外，審稿人員也建議改用 *`command`* `-v` 的形式來取代 `which`[15]，以便判斷其背後是程式路徑或是 shell 的別名 / 函式。詳情請參閱 shellcheck 文件 [16]。

13 譯註：注意！若要查詢前一次執行的結果狀態，必須在執行目標完成後立即以 `echo $?` 查詢；若重複執行 `echo $?`，你查到的其實已經是 `echo $?` 本身的執行成敗紀錄。

14 譯註：例如執行 ls -al | more 以便觀察一頁以上的目錄內檔案清單，就算執行無誤，$? 呈現的也會是 more 執行成功的結果。讀者們不妨故意把管線前的命令打錯字，譬如 ld -al | more 好了，這時若直接檢視 $?，其結果仍為 0（因為 more 沒有發生錯誤，錯誤發生在我們故意把 ls 拼錯字的 ld）；若再度製造一次同樣錯誤（記住 $? 記憶的是前一次執行的退出狀態碼哦），這時改以 $PIPESTATUS 檢查，就會查出 1 的狀態碼了（表示管線中出現過錯誤）。

15 譯註：例如要查詢 ls 的背景，就不要用 which ls 來查，而是改用 command -v ls 來查。

16 *https://oreil.ly/5toUM*

工作控管

大部分 shell 還支援另一項功能，亦即**工作控管**（*job control*）。依照預設，當你鍵入一道命令，它便掌控了螢幕和鍵盤，這稱為**前景執行**（*running in the foreground*）。然而如果你不想以互動方式執行，或是像伺服器的情況那樣、執行時根本不會有任何輸入來自 `stdin` 時，該當如何？這時便是工作控管和背景工作登場的時候了：你可以在背景端發起程序，只需執行時在尾端加上 `&` 字元即可，或是用 Ctrl+Z 按鍵組合，將還在前景運作的程序送往背景端繼續執行。

以下範例就是實際的運作方式，讓大家體驗一下：

```
$ watch -n 5 "ls" & ❶

$ jobs ❷
Job     Group   CPU     State   Command
1       3021    0%      stopped watch -n 5 "ls" &

$ fg ❸
Every 5.0s: ls                                          Sat Aug 28 11:34:32 2021

Dockerfile
app.yaml
example.json
main.go
script.sh
test
```

❶ 在命令結尾放上一個 `&`，藉以將命令放到背景端執行。

❷ 列出所有的工作。

❸ 以 `fg` 命令將程序帶往前景。如果你想結束執行 `watch`，只需按下 Ctrl+C 就好。

如果你想在 shell 關閉後仍能讓背景的程序持續執行，可以在前面加上 `nohup` 命令。此外，對於已在執行當中、而且執行時尚未在前面加上 `nohup` 的程序，你可以事後執行 `disown` 來達到同樣的效果。最後一點是，如果你想清除一個執行中的程序，請用 `kill` 命令加上各種不等的強制等級（請參閱 230 頁的「訊號」一節）。

除卻工作控管以外，筆者建議大家多多利用終端機多工器（terminal multiplexer），等下在 61 頁的「終端機多工器」一節便會介紹。這類程式會處理最常見的運作案例（例如 shell 的關閉、多重程序的運行、以及它們之間的協作等等），同時也可以在遠端系統上使用。

我們繼續來介紹一些常用核心命令（幾乎是從 Unix 存在以來就有）的近代替代品。

現代化的命令

你會發現有很多命令是每天都要一再用到的。這些命令包括瀏覽目錄用的 `cd`、列舉目錄內容的 `ls`、尋找檔案用的 `find`、以及顯示檔案內容用的 `cat` 跟 `less` 等等。既然你這麼常用到它們，也許你會想在操作相關功能時能更有效率，因為每多打一個字都是浪費時間。

有些常用命令已有更現代化的變種存在。其中有的是平行取代、有些則是延伸了既有的功能。但它們全都為最通用的操作方式先提供了合理的預設值，此外也提供了更豐富的輸出，不但便於判讀、通常在輸入時也更精簡省力，輸入寥寥數個字便能完成相同的任務。這讓大家在操作 shell 時比較不吃力，不但工作起來更愉快、也可以讓流程更順暢。如果你想更進一步了解現代化工具，請參閱附錄 B。但請注意，如果你想在自己的企業用環境中運用這類知識：筆者個人與這些工具並無瓜葛，純粹只是我自己覺得它們好用、所以推薦給讀者而已。若要安裝與試用這類工具，最好的方式是選擇你使用的 Linux 發行版所驗證過的版本。

以 exa 列舉目錄內容

每當你想要知道目錄中的內容，通常都是用 `ls`、或是用 `ls` 的變種命令加上參數為之。以 bash 為例，筆者會改用以別名處理過的 `ls -GAhltr`。但其實有更好的方式：`exa`[17]，它是 `ls` 的現代化替代品，以 Rust 撰寫而成，內建支援 Git 和樹狀圖繪製。請問讀者諸君，一旦列出了目錄內容，你覺得接下來最常用的命令是什麼？依筆者自身經驗，是清空螢幕畫面，而人們通常都是以 `clear` 來完成任務的。但此舉需要鍵入五個字元、還沒算上按下 `ENTER` 一個動作。但有了 exa，這動作只需要一個按鍵組合（Ctrl+L）就能完成。

17 *https://oreil.ly/5lPAl*

用 bat 檢視檔案內容

假設你已列出了目錄內容，也找到了你要檢查的檔案。也許你會用 `cat` 來檢視，是嗎？其實有更好的工具，筆者建議你試試 `bat`[18]。正如圖 3-3 所示，`bat` 命令帶有語法凸顯功能，能顯示無法印出的字元，也支援 Git，同時內建換頁功能（亦即當檔案顯示內容超過一個螢幕大小的頁面時）[19]。

用 rg 尋找檔案內容

通常大家都用 `grep` 來尋找檔案內容。然而，這也有現代化替代品可用，就是 `rg`[20]，它既快速又強大。

下例中會比較 `rg` 和組合 `find` 與 `grep` 之間的效果，範例目標是找出含有字串「sample」字樣的 YAML 檔案：

```
$ find . -type f -name "*.yaml" -exec grep "sample" '{}' \; -print ❶
      app: sample
        app: sample
./app.yaml

$ rg -t "yaml" sample ❷
app.yaml
9:      app: sample
14:        app: sample
```

❶ 合併使用 `find` 和 `grep`，在 YAML 檔案中尋找字串。

❷ 以 `rg` 進行同樣的任務。

如果你比較上例中的命令和執行結果，就會發現 `rg` 不僅使用起來簡單，其結果資訊也比較豐富（包括更多背景資訊，以本例來說就是行號）。

18 *https://oreil.ly/w3K76*
19 譯註：譯者在 Ubuntu 22.04 上測試，裝好 bat 後，命令會變成 batcat。
20 *https://oreil.ly/u3Sfw*

File: main.go

```
1  package main
2
3  import (
4      "fmt"
5      "net/http"
6  )
7
8  func main() {
9      http.HandleFunc("/", HelloServer)
10     http.ListenAndServe(":8080", nil)
11 }
12
13 func HelloServer(w http.ResponseWriter, r *http.Request) {
14     fmt.Fprintf(w, "Hello, %s!", r.URL.Path[1:])
15 }
```

File: app.yaml

```
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: something
5 +  namespace: xample
6  spec:
7    selector:
8      matchLabels:
9        app: sample
10 ~  replicas: 2
11   template:
12     metadata:
13       labels:
14         app: sample
15     spec:
16       containers:
17         - name: example
18 _         image: public.ecr.aws/mhausenblas/example:stable
```

圖 3-3　用 bat 列出一個 Go 語言檔案（上方）及一個 YAML 檔案（下方）的內容

以 jq 處理 JSON 資料

現在到了加分時間了。我們要介紹 `jq` 這個命令，它其實並不算是替代品，而更像是 JSON 的專用工具，JSON 本身就是一種極受歡迎的文字資料格式。你可以在 HTTP API 及類似的設定檔中看到 JSON 的身影。

所以現在都用 `jq`[21] 取代 `awk` 或 `sed` 來擷取特定資料值。舉例來說，如果用 JSON generator[22] 來產生一些隨機資料，筆者手邊便有一個 2.4 KB 的 JSON 檔案 *example.json*，看起來就像下面這樣（以下只顯示其中第一筆紀錄）：

```
[
  {
    "_id": "612297a64a057a3fa3a56fcf",
    "latitude": -25.750679,
    "longitude": 130.044327,
    "friends": [
      {
        "id": 0,
        "name": "Tara Holland"
      },
      {
        "id": 1,
        "name": "Giles Glover"
      },
      {
        "id": 2,
        "name": "Pennington Shannon"
      }
    ],
    "favoriteFruit": "strawberry"
  },
...
```

假設我們只對那些最喜歡的水果是「草莓」的人的第一位朋友有興趣，亦即 `friends` 這個陣列中的第 0 個元素。如果用 `jq` 查詢，可以這樣寫：

```
$ jq 'select(.[].favoriteFruit=="strawberry") | .[].friends[0].name' example.json
"Tara Holland"
"Christy Mullins"
"Snider Thornton"
"Jana Clay"
"Wilma King"
```

21 *https://oreil.ly/9s7yh*
22 *https://oreil.ly/bcT9d*

這些 CLI 都很有趣是吧？如果讀者們有意閱讀更多關於現代化命令、以及其他可以用來替代的候補對象之類的題材，請參閱 modern-unix repo[23]，其中列舉了諸多建議。接下來，我們要將注意力轉移到一些常見的任務上，而不再僅限於瀏覽目錄及檢視檔案內容了。

常見的任務

你會發現有些事是自己經常要做的，而 shell 裡有很多辦法可以幫你加速處理這類事務。我們來看看有哪些常見任務可以做得更有效率。

簡化常用的命令

一旦深入探討介面時，其中一項基本議題便是：最常用的命令用起來應該最不費力，它們應該很快就能輸入完畢。現在請把這個概念用到 shell 上：你不該鍵入 `git diff --color-moved` 這一長串命令，而應該以極精簡的 `d`（單一字母）就能完成任務，因為筆者一天當中常常需要一再地檢視儲藏庫中的變動達數百次之多。根據不同的 shell，也有不同的方式可以簡化這一點：在 bash 裡通常是透過**別名**（*alias*[24]）的方式為之，而在 Fish 裡（請參閱第 54 頁的「Fish Shell」）則會利用縮寫功能（abbreviations[25]）做到同一點。

瀏覽

當你在 shell 的提示後鍵入命令時，有些動作是很常做的，例如在行中移動（像是將游標移至行首之類）、或是處理一行文字（像是刪除游標以左全部內容）等等。表 3-2 便列出常用的 shell 捷徑。

表 3-2　Shell 瀏覽及編輯用的捷徑

動作	命令	註解
將游標移至一行的開頭	Ctrl+a	-
將游標移至一行的結尾	Ctrl+e	-
將游標往前移一個字元	Ctrl+f	-
將游標往後移一個字元	Ctrl+b	-
將游標往前移一整個單字	Alt+f	僅限左邊的 Alt 鍵
將游標往後移一整個單字	Alt+b	-
刪除現在位置的字元	Ctrl+d	-

23 *https://oreil.ly/cBAXt*
24 *https://oreil.ly/fbBvm*
25 *https://oreil.ly/rrmNI*

動作	命令	註解
刪除游標左側的字元	Ctrl+h	-
刪除游標左側的單字	Ctrl+w	-
刪除游標右側所有內容	Ctrl+k	-
刪除游標左側所有內容	Ctrl+u	-
清空畫面	Ctrl+l	-
取消命令	Ctrl+c	-
還原	Ctrl+_	僅限 bash
搜尋歷程	Ctrl+r	只有部分 shell 可用
取消搜尋	Ctrl+g	只有部分 shell 可用

注意，不是所有的 shell 都支援以上全部的捷徑，而且有些動作，例如命令的歷程管理，在某些 shell 裡實作的方式便有所不同。此外你也許需要知道，這些捷徑的按鍵組合都是源於 Emacs 的編輯鍵。如果你偏好 `vi`，可以利用在 *.bashrc* 檔案裡加上 `set -o vi` 的字樣，藉以改用 `vi` 風格的按鍵組合來進行命令列編輯。最後，請以表 3-2 為出發點，試試看你的 shell 支援哪些功能，還有你能否將其設定為適合你的習慣用法。

檔案內容的管理

如果只是要加一行文字，不一定要大費周章地啟用 `vi` 這樣的編輯器才能進行。有時你甚至還沒有 `vi` 可以用，例如撰寫指令碼的時候（參閱第 68 頁的「Scripting」一節）。

那麼，你要如何操作文字的內容？來看幾個例子：

```
$ echo "First line" > /tmp/something ❶

$ cat /tmp/something ❷
First line

$ echo "Second line" >> /tmp/something && \ ❸
  cat /tmp/something
First line
Second line

$ sed 's/line/LINE/' /tmp/something ❹
First LINE
Second LINE

$ cat << 'EOF' > /tmp/another ❺
First line
Second line
Third line
```

```
EOF

$ diff -y /tmp/something /tmp/another ❻
First line                                                      First line
Second line                                                    Second line
                                                             > Third line
```

❶ 藉由將 echo 的輸出轉向，建立一個檔案。

❷ 檢視新建檔案的內容。

❸ 利用 >> 運算子，在檔案底部附加新的一行文字，並隨即檢視其內容。

❹ 利用 sed 替換檔案內的內容，並將輸出傳給 stdout[26]。

❺ 利用即席文件（here document[27]）新建另一個檔案。

❻ 比較兩個檔案的差異。

現在大家都知道基本的檔案內容操作方式了，我們要來看一些更進階的檔案內容檢視方式。

檢視冗長的檔案

對於冗長的檔案來說（亦即檔案行數超過 shell 可以在螢幕上以一頁顯示的篇幅時），就可以用 less 或 bat 之類的分頁顯示工具（bat 自己內建了分頁工具）。藉由分頁顯示，程式便能將輸出拆成剛好可以讓一個螢幕畫面顯示的分頁、再提供一些操作命令以便瀏覽（例如前後翻頁等等）。

另一種處理冗長檔案的方式，是只顯示檔案中選定的部分，例如只有開頭幾行。這樣方便的命令有兩種可以引用：head 和 tail。

舉例來說，若要顯示檔案開頭部分：

```
$ for i in {1..100} ; do echo $i >> /tmp/longfile ; done ❶

$ head -5 /tmp/longfile ❷
1
2
3
4
5
```

26 譯註：此處 sed 只會把完成替換後的內容顯示到 stdout，但是原始檔案 */tmp/something* 的內容其實隻字未動。如果你確實要把編輯過的內容寫回原檔案，請利用 sed -i 來做。

27 *https://oreil.ly/FPWqT*

❶ 刻意建立一個冗長的檔案（這裡是 100 行）。

❷ 只顯示該檔案的前五行。

或者你想追蹤一個持續增長的檔案的最近更新內容，就這樣做：

```
$ sudo tail -f /var/log/Xorg.0.log ❶
[ 36065.898] (II) event14 - ALPS01:00 0911:5288 Mouse: device is a pointer
[ 36065.900] (II) event15 - ALPS01:00 0911:5288 Touchpad: device is a touchpad
[ 36065.901] (II) event4  - Intel HID events: is tagged by udev as: Keyboard
[ 36065.901] (II) event4  - Intel HID events: device is a keyboard
...
```

❶ 用 `tail` 顯示一個日誌檔的結尾部分，並加上選項 `-f` 以便追蹤內容異動，亦即自動顯示新進的內容。

本小節最後要來談談對於日期與時間的處理。

日期與時間的處理

如果需要產生獨一無二的檔案名稱，`date` 命令十分好用。它允許你產生各種格式的日期，包括 Unix 時間戳記，以及在各種不同的日期與時間格式之間轉換。

```
$ date +%s ❶
1629582883

$ date -d @1629742883 '+%m/%d/%Y:%H:%M:%S' ❷
08/21/2021:21:54:43
```

❶ 建立一個 UNIX 時間戳記 [28]。

❷ 將一筆 UNIX 時間戳記轉換成可以供人判讀的日期。

關於 UNIX 紀元時間

UNIX 紀元時間（UNIX epoch time，有時也簡稱為 UNIX 時間）是一段以秒計算的時間長度，從 1970-01-01T00:00:00Z 開始累計。UNIX 時間將每一天都視為剛好 86,400 秒來計算。

28 譯註：`+%s` 的意思就是要產生一筆 Unix 紀元時間。

如果你處理的軟體會將 UNIX 時間儲存為有號的 32 位元整數，你可能需要特別留意，因為到了 2038-01-19 便會發生問題，即計時器會出現溢位，這又稱為「2038 年問題[29]」。

你也可以利用線上轉換工具[30]進行更多樣化的操作，甚至可以解析到微秒與毫秒的程度。

到此本小節已經講述過 shell 的基本知識了。現在讀者們應該已經對終端機與 shell 有了正確的認識，也知道如何運用它們來遂行日常任務，像是瀏覽檔案系統、尋找檔案等等。接下來要進行下一個主題：操作起來更為友善的 shell。

更為友善的 Shells

雖說 bash shell 可能依舊是最廣為使用的 shell，但它卻不見得是操作起來最友善的。Bash 問世於約莫 1980 年代晚期，已經頗有年歲了。而坊間已經出現了不少標榜著人性化的現代化 shell，筆者鄭重建議大家自行嘗試一下，不要只使用 bash。

我們會先以實際案例詳細介紹一種現代化且更友善的 shell，亦即 Fish shell，再簡要介紹其他的類似產品，讓各位知道有多少種選擇。本小節的結尾，會在第 61 頁「我該使用哪一種 Shell？」提出建議和結論，作為總結。

Fish Shell

Fish shell[31] 宣稱自己是一種聰明而且對使用者友善的命令列 shell。我們先來看一些相關的基本操作，然後再進展到與組態相關的題材。

基本操作

對於許多日常任務來說，大家都不會注意到 bash 裡的輸入方式有何特別之處；表 3-2 所列的大部分命令幾乎都可以使用。不過 `fish` 有兩處是跟 `bash` 有所不同、而且還更為方便的：

29 *https://oreil.ly/dKiWx*
30 *https://oreil.ly/Z1a4A*
31 *https://fishshell.com*

沒有明顯的歷程管理。

你只需鍵入寥寥數字，就能看到先前執行過的命令。然後就可以用上下方向鍵選擇要執行的內容（請參閱圖 3-4）。

```
└─> $ exa --long --all --git
app.yaml  Dockerfile  example.json  main.go  script.sh  test
```

圖 3-4　Fish 的命令歷程處理實例

許多命令都帶有自動補齊的建議功能。

如圖 3-5 所示。此外，當你按下 Tab 鍵時，Fish shell 就會嘗試自動補齊命令名稱、其引數、甚至是路徑，還會用有顏色的文字讓你可以一眼看出各種提示，連你打錯字時都會以紅色凸顯有錯誤發生。

```
└─> $ ls -@Ahltr
-1                                  (List one entry per line)  -l                          (Long listing format)
-@                          (for -l: Display extended attributes)  -m     (Comma-separated format, fills across screen)
-A                                (Show hidden except . and ..)  -n            (Long format, numerical UIDs and GIDs)
-a                                    (Show hidden entries)  -O                    (for -l: Show file flags)
-B                     (Octal escapes for non-graphic characters)  -o                 (Long format, omit group names)
-b                         (C escapes for non-graphic characters)  -P                       (Don't follow symlinks)
-C                                (Force multi-column output)  -p                   (Append directory indicators)
-c                 (Sort (-t) by modified time and show time (-l))  -q        (Replace non-graphic characters with '?')
-d                          (List directories, not their content)  -R                (Recursively list subdirectories)
-e           (for -l: Print ACL associated with file, if present)  -r                        (Reverse sort order)
-F  (Append indicators. dir/ exec* link@ socket= fifo| whiteout%)  -S                              (Sort by size)
-f                                (Unsorted output, enables -a)  -s                          (Show file sizes)
-G                                    (Enable colorized output)  -T             (for -l: Show complete date and time)
-g                    (Show group instead of owner in long format)  -t    (Sort by modification time, most recent first)
-H                           (Follow symlink given on commandline)  -U   (Sort (-t) by creation time and show time (-l))
-h                                   (Human-readable sizes)  -u     (Sort (-t) by access time and show time (-l))
-i                              (Show inode numbers for files)  -W     (Display whiteouts when scanning directories)
-k                         (for -s: Display sizes in kB, not blocks)  -w  (Force raw printing of non-printable characters)
-L                           (Follow all symlinks Cancels -P option)  -x        (Multi-column output, horizontally listed)
```

圖 3-5　Fish 的自動補齊建議實例

表 3-3 列出了若干常用的 `fish` 命令。請特別留意這時處理環境變數的方式。

表 3-3　Fish shell 的參考

任務	命令
匯出環境變數 KEY 並賦值為 VAL	`set -x KEY VAL`
刪除環境變數 KEY	`set -e KEY`
將環境變數 KEY 賦予命令 cmd	`env KEY=VAL cmd`
將顯示路徑長度精簡到 1 個字元	`set -g fish_prompt_pwd_dir_length 1`
管理縮寫	`abbr`
管理函式	`functions` 和 `funcd`

fish 與其他的 shell 不同之處，在於它將上一個命令的退出狀態儲存在變數 $status 中，而非 $?[32]。

如果你原本用慣了 bash，也許可以參考一下 Fish FAQ[33]，其中解釋了各種疑問。

組態

如要設定 Fish shell 的組態，只需鍵入 fish_config 命令即可（視發行版的不同，你可能需要加上子命令 browse[34]），然後 fish 會在 *http://localhost:8000* 啟動一個本機的網頁伺服程式，並自動以你的預設瀏覽器來開啟一個花俏的使用介面，如圖 3-6 所示，這樣就可以檢視和變更設定了。

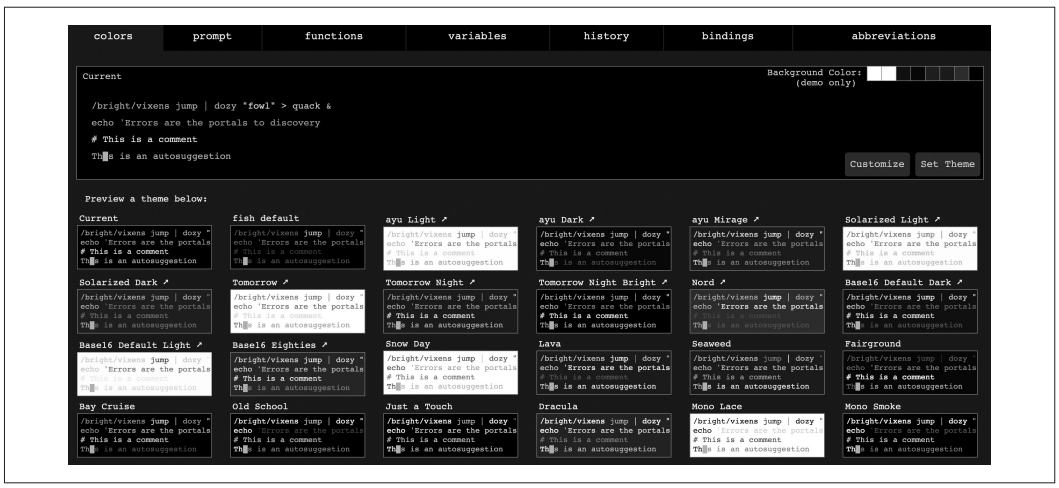

圖 3-6　以瀏覽器設定 Fish　shell

如欲在 vi 與 Emacs（預設）兩種命令列瀏覽按鍵組合風格間切換，只需鍵入 fish_vi_key_bindings 就能啟用 vi 模式，若再輸入 fish_default_key_bindings 便能復原到 Emacs 模式。注意一旦進行變更，就會立即套用到所有仍在使用中的 shell 會談。

32 譯註：csh 也是以 $status 代表前一指令的退出狀態。

33 *https://oreil.ly/Nk2S2*

34 譯註：如果你開啟瀏覽器後遇上「找不到檔案」的錯誤，是因為 Ubuntu 22.04 預設瀏覽器 Firebox 無權操作 /tmp 目錄之故；簡單的規避方式是改用其他預設瀏覽器例如 Chrome，就看得到 fish_config 的畫面了。

結論

本章著重在以終端機、也就是文字化使用者介面來操作 Linux。我們介紹了關於 shell 的術語，也提供了關於 shell 基本操作的上手說明，還檢視了各種常見的任務、以及如何透過一些特定命令的現代化版本，來提升你在 shell 上的生產力（例如以 `exa` 取代 `ls`）。

接著我們又看過了現代更人性化的 shell，特別是 `fish`，也解釋了如何設定和使用這些 shell。此外，我們也以 `tmux` 為上手的實例，探討了終端機多工器，讓讀者們能同時操作多個本地端或遠端的會談。運用現代化的 shell 和多工器，可以大幅地提升你在命令列的工作效率，筆者十分鄭重地建議大家考慮採用它們。

最後我們探討了如何撰寫安全及可攜的 shell 指令碼，藉以將任務自動化，也提到如何潤飾和測試指令碼。記住，shell 其實就只是命令的直譯工具，就像任一種語言一樣，你得經常練習才能流利地運用它。話說到此，讀者們已經具備了以命令列操作 Linux 的基礎知識，也就能應付坊間大多數採用 Linux 的系統了，不論是嵌入式系統還是雲端虛擬機都一樣。在任何情況下，你都可以找到取得終端機介面的方式，並以互動方式發號施令、或是執行指令碼。

倘若讀者們還想繼續深入本章探討的各項題材，以下是若干額外參考資源：

終端機

- 「Anatomy of a Terminal Emulator」（*https://oreil.ly/u2CFr*）
- 「The TTY Demystified」（*https://oreil.ly/8GT6s*）
- 「The Terminal, the Console and the Shell—What Are They?」（*https://oreil.ly/vyVAV*）
- 「What Is a TTY on Linux? (and How to Use the tty Command)」（*https://oreil.ly/E0EGG*）
- 「Your Terminal Is Not a Terminal: An Introduction to Streams」（*https://oreil.ly/xIEoZ*）

Shells

- 「Unix Shells: bash, Fish, ksh, tcsh, zsh」（*https://oreil.ly/4pepC*）
- 「Comparison of Command Shells」（*https://oreil.ly/RQfS6*）
- 「bash vs zsh」 thread on reddit（*https://oreil.ly/kseEe*）
- 「Ghost in the Shell—Part 7—ZSH Setup」（*https://oreil.ly/1KGz6*）

終端機多工器

- 「A tmux Crash Course」（*https://oreil.ly/soqPv*）
- 「A Quick and Easy Guide to tmux」（*https://oreil.ly/0hVCS*）
- 「How to Use tmux on Linux（and Why It's Better Than screen）」（*https://oreil.ly/Q75TR*）
- *The Tao of tmux*（*https://oreil.ly/QDsYI*）
- *tmux 2: Productive Mouse-Free Development*（*https://oreil.ly/eO9y2*）
- Tmux Cheat Sheet & Quick Reference website（*https://oreil.ly/SWCa5*）

Shell 指令碼

- 「Shell Style Guide」（*https://oreil.ly/3cxAw*）
- 「bash Style Guide」（*https://oreil.ly/zfy1v*）
- 「bash Best Practices」（*https://oreil.ly/eC1ol*）
- 「bash Scripting Cheatsheet」（*https://oreil.ly/nVroM*）
- 「Writing bash Scripts That Are Not Only bash: Checking for bashisms and Testing with Dash」（*https://oreil.ly/D0zwe*）

掌握了本章的 shell 基礎知識以後，我們要進行下一個主題：Linux 的存取控制與實施方式。