第一部分

基本概念

在第 1 個部分裡，我們會將重點擺在 *make* 的功能上：它們能夠做什麼，以及如何正確地使用它們。我們首先會做個簡介，並且告訴你如何建立你的第一個 *makefile*。這個部分的內容涵蓋了 *make* 的規則、變數、函式以及命令稿。

看完第 1 個部分之後，你將會獲得相當完整的 GNU *make* 操作知識，以及掌握許多進階的用法。

第 1 章

如何撰寫一個簡單的 makefile

程式設計的技術通常遵循著一個極為簡單的慣例：編輯原始碼檔案、將原始碼編譯成可執行的形式，以及對成果除錯。儘管將原始碼轉換成執行檔被視為慣例，如果程式員的做法有誤也可能會浪費大量的時間在問題的除錯上。大多數開發者都經驗過這樣的挫敗：修改好一個函式之後，執行新的程式碼時卻發現瑕疵並未修正，接著發現再也無法執行這個經過修改的函式，因為其中某個程序有誤，像是無法重新編譯原碼、重新連結可執行檔或是重新建造 jar 檔。此外，當程式變的越來越複雜時，這些例行工作可能會因為需要（針對其他平台，或程式庫的其他版本，等等）為程式開發不同的版本，變得越來越容易發生錯誤。

make 程式可讓「將原始碼轉換成可執行檔」之類的例行工作自動化。相較於命令稿，make 的優點在於，你可以把程式中各元素之間的關係告訴 make，之後 make 會根據這些關係和時間戳記（timestamp）【譯註】，判斷應該重新進行哪些步驟，以產生你所需要的程式。有了這個資訊，make 還可以優化建造過程，跳過非必要的步驟。

GNU make（以及 make 的其他變體）可以準確完成此事。make 定義了一個語言，可用來描述原始檔、中間檔以及可執行檔的關係。它還提供了一些功能，可用來管理各種候選組態、實作可重用程式庫的細節，以及讓使用者以自訂巨集將過程參數化。簡言之，make 常被視為開發過程的核心，因為它為應用程式的元件以及這些元件的搭配方式提供了一個可依循的準則。

譯註　Unix 檔案具有三種時間屬性：atime（最近被讀取的時間）、ctime（模式被改變的時間）以及 mtime（最近被寫入的時間）。檔案的時間戳記（timestamp）就是指 mtime。

make 一般會將工作細節存放在一個名為 *makefile* 的檔案中。下面是一個可用來建造傳統 "Hello, World" 程式的 *makefile*：

```
hello: hello.c
        gcc hello.c -o hello
```

要建造此程式，你可以在命令列提示符號之後鍵入：

```
$ make
```

以便執行 make。這將會使得 make 程式讀入 *makefile* 檔案，並且建造它在該處所找到的第一個工作目標：

```
$ make
gcc hello.c -o hello
```

如果將某個工作目標（target）指定成命令列引數（command-line argument），make 就會特別針對該工作目標進行更新的動作。如果命令列上未指定任何工作目標，make 就會採用 *makefile* 檔案中第一個工作目標 — 稱為**預定目標**（default goal）。

在大多數 *makefile* 檔案中，預定的目標一般就是建造程式。這通常涉及許多步驟。程式的原始碼經常是不完整的，而且必須使用 *flex* 或 *bison* 之類的工具來產生原始碼。接著原始碼必須被編譯成二元目的檔（binary object file）—*.o* 檔用於 C/C++、*.class* 檔用於 Java，等等。然後，對 C/C++ 而言，連結器（通常調用自 *gcc* 編譯器）會將這些目的檔繫結在一起形成一個可執行檔。

修改原始檔中任何內容並重新調用 make，將會使得這些步驟中有某些（通常不是全部）被重複進行，因此原始碼中的變動會被適切地併入可執行檔。這個細節描述檔（specification file）或 *makefile* 檔中，描述了原碼檔、中間檔以及可執行檔之間的關係，使得 make 能夠以最少的工作量來完成更新可執行檔的工作。

所以 make 的主要價值在於，它有能力完成建造應用程式時所需要的一系列複雜步驟，以及當有可能縮短「編輯 - 編譯 - 除錯」（edit-compile-debug）週期時對這些步驟進行優化的動作。此外，make 極具彈性，你可以在任何具有檔案依存關係的地方使用它，範圍從 C/C++ 到 Java、$\TeX$、資料庫管理，等等。

1.1 工作目標與必要條件

基本上，*makefile* 檔案中包含了一組用來建造應用程式的規則。make 所看到的第一項規則，會被當成預定規則（default rule）使用。一項規則可分成三個部分：工作目標（target）、它的必要條件（prerequisites）以及所要執行的命令（commands）：

```
target: prereq1 prereq2
   commands
```

工作目標（*target*）是一個必須建造的檔案或東西。必要條件（*prerequisites*）或依存對象（*dependents*）是工作目標得以被成功建造之前，必須事先存在的那些檔案。而所要執行的命令（*commands*）則是必要條件成立時將會執行的那些 shell 命令。

下面這項規則係用來將一個 C 檔案 *foo.c* 編譯成一個目的檔 *foo.o*：

```
foo.o: foo.c foo.h
gcc -c foo.c
```

工作目標 *foo.o* 出現在冒號之前。必要條件 *foo.c* 和 *foo.h* 出現在冒號之後。命令稿（command script）通常出現在後續的文字列上，而且會擺在跳格字符（tab character）之後。

當 make 被要求處理某項規則時，它首先會找出必要條件和工作目標中所指定的檔案。如果必要條件中有任何檔案關連到其他規則，則 make 會先完成相應規則的更新動作，然後才會考慮到工作目標。如果必要條件中有任何檔案的時間戳記在工作目標的時間戳記之後【譯註】，則 make 會執行命令稿以便重新建造工作目標。命令稿會傳遞給 shell 並在其 subshell 中執行。如果其中有任何命令發生錯誤，則 make 會終止工作目標的建造動作並結束執行。

下面這個程式會在它的輸入中計算 fee、fie、foe 和 fum 等詞彙（單字）出現的次數。這個程式（檔名為 *count_words.c*）並不難，因為它使用了一個 *flex* 掃描器：

```
#include <stdio.h>

extern int fee_count, ◘e_count, foe_count, fum_count;
extern int yylex( void );

int main( int argc, char ** argv )
{
    yylex();
    printf( "%d %d %d %d\n", fee_count, ◘e_count, foe_count, fum_count );
    exit( 0 );
}
```

譯註　也就是說，必要條件中最近有檔案遭到修改，但是工作目標尚未進行更新的動作。

這個掃描器（檔名為 *lexer.l*）相當簡單：

```
          int fee_count = 0;
          int ␣e_count = 0;
          int foe_count = 0;
          int fum_count = 0;
%%
fee       fee_count++;
␣e        ␣e_count++;
foe       foe_count++;
fum       fum_count++;
```

用來建造這個程式的 *makefile* 也很簡單：

```
count_words: count_words.o lexer.o -l␣
        gcc count_words.o lexer.o -l␣ -o count_words

count_words.o: count_words.c
        gcc -c count_words.c

lexer.o: lexer.c
        gcc -c lexer.c

lexer.c: lexer.l
        ␣ex -t lexer.l > lexer.c
```

當這個 *makefile* 首次執行時，我們會看到：

```
$ make
gcc -c count_words.c
␣ex -t lexer.l > lexer.c
gcc -c lexer.c
gcc count_words.o lexer.o -l␣ -o count_words
```

現在我們已經建造好了一個可執行的程式。當然，此處所舉的例子有點簡化，因為實際的程式通常是由多個模組所構成。此外，看過後面的章節你就會知道，這個 *makefile* 並未用到 make 大部分的特性，所以顯得有點囉嗦。不過，它仍不失為一個實用的 *makefile*。舉例來說，在這個範例撰寫期間，為了試驗程式，我執行了這個 *makefile* 不下數十次。

當這個 *makefile* 範例執行時，你可能會發現 make 執行命令的順序幾乎和它們出現在 *makefile* 中的順序相反。這種「從上而下」（top-down）的風格是 *makefile* 檔案中常見的手法。一般來說，通則形式的工作目標會先描述在 *makefile* 檔案中，而細節則會擺在後面。make 程式對此風格的支援有許多方式。其中以 make 的兩階段執行模型（two-phase execution model）以及遞迴變數（recursive variable）最為重要。我們將會在稍後的章節深入探討相關細節。

1.2 檢查依存關係

make 如何知道自己該做什麼事呢？讓我們繼續探討前一個範例。

make 首先注意到命令列上並未指定任何工作目標，所以會想要建造預定目標 *count_words*。當 make 檢查其必要條件時看到了三個項目：*count_words.o*、*lexer.o* 以及 -l□。現在 make 會想要建造 *count_words.o* 並看到相應的規則。接著 make 會再次檢查必要條件並注意到 *count_words.c* 並未關連到任何規則，但存在 *count_words.c* 這個檔案，所以會執行相應的命令把 *count_words.c* 轉換成 *count_words.o*：

```
gcc -c count_words.c
```

這種「從工作目標到必要條件、從必要條件到工作目標、再從工作目標到必要條件」的鏈結（chaining）機制就是 make 分析 *makefile* 決定要執行哪些命令的典型做法。

必要條件的下一個項目會讓 make 相要建造 *lexer.o*。規則鏈會將 make 導向 *lexer.c*，但這次 *lexer.c* 並不存在。make 會從 *lexer.l* 找到產生 *lexer.c* 的規則，所以 make 會執行 *flex* 程式。現在 *lexer.c* 存在了，make 會接著執行 *gcc* 命令。

最後，make 看到 -l□。其中 -l 是個選項，用來要求 *gcc* 必須將其所指定的系統程式庫連結進應用程式。此處指定了 fl 這個參數，代表實際的程式庫名稱為 *libfl.a*。GNU make 對這個語法提供了特別的支援。當 -l<NAME> 形式的必要條件被發現時，make 會搜尋 lib*NAME*.so 形式的檔案；如果找不到相符的檔案，make 接著會搜尋 lib*NAME*.a 形式的檔案。在此例中，make 會找到 */usr/lib/libfl.a* 而且會進行最後的動作 — 連結。

1.3 儘量減少重新建造的工作量

執行這個程式時，我們發現它除了會印出 fee、fie、foe 和 fum 等單字出現的次數，還會印出來自輸入檔案的其他文字。這並非我們想要的結果。問題出在我們忽略了詞彙分析器（lexical analyzer）的一些規則，而且 *flex* 會將未被認出的文字送往輸出。我們只要加入一條「any character」（任何字符）規則以及一條 newline（換列字符）規則就可以解決這個問題：

```
          int fee_count = 0;
          int □e_count = 0;
          int foe_count = 0;
          int fum_count = 0;
%%
fee       fee_count++;
□e        □e_count++;
foe       foe_count++;
fum       fum_count++;
.
\n
```

編輯這個檔案之後，還需要重新建造應用程式以便測試我們所做的修正：

```
$ make
□ex -t lexer.l > lexer.c
gcc -c lexer.c
gcc count_words.o lexer.o -l□ -ocount_words
```

請注意，這次 *count_words.c* 檔案並未被重新編譯。分析規則的時候，make 發現 *count_words.o* 已存在，而且該檔案的時間戳記在其必要條件 *count_words.c* 之後，所以不需要採取任何更新的動作。不過，分析 *lexer.c* 的時候，make 發現必要條件 *lexer.l* 的時間戳記在其工作目標 *lexer.c* 之後，所以 make 必須更新 *lexer.c*。這會依次造成 *lexer.o*、*count_words* 的更新。執行這個重新建造的程式，你會看到如下的結果：

```
$ count_words < lexer.l
3 3 3 3
```

1.4 調用 make

前面的範例做了以下假設：

- 專案的所有程式碼以及 make 描述檔全都放在單一目錄中。
- make 描述檔的檔名為 *makefile*、*Makefile* 或 *GNUMakefile*。
- 執行 make 命令時，*makefile* 就放在使用者的當前目錄（current directory）中。

當 make 在上述情況下被調用時，make 會自動建造其所找到的第一個工作目標。要更新另一個不同的工作目標（或多個工作目標）請在命令列上指定工作目標的名稱：

```
$ make lexer.c
```

當 make 執行時，它將會讀取描述檔以及找到所要更新的工作目標。如果工作目標或其必要條件中有任一檔案尚未更新（或不存在），則會（以一次一個命令的方式）執行相應規則之命令稿中的 shell 命令。這些命令執行之後，make 會假定工作目標已完成更新的動作，於是移往下一個工作目標或是結束執行。

如果你所指定的工作目標已經更新（up to date），make 除了告訴你此狀況並立即結束執行，其他什麼事也不做：

```
$ make lexer.c
make: `lexer.c' is up to date.
```

如果你所指定的工作目標並未出現在 *makefile* 檔案中，也不存在與之相應的內定規則（implicit rule），make 將會做如下的回應：

```
$ make non-existent-target
make: *** No rule to make target `non-existent-target'.  Stop.
```

make 提供了許多命令列選項。其中最有用的選項之一是 --just-print（或 -n），用來要求 make 顯示它將為特定工作目標執行的命令，但不要實際執行它們。當你在撰寫 *makefile* 時，這個功能特別有用。你甚至還可以在命令列上設定幾乎任何的 *makefile* 變數，來改寫預設值或 *makefile* 檔案中所設定的值。

1.5 Makefile 的基本語法

對 make 有了基本的認識之後，現在你差不多可以撰寫自己的 *makefile* 了。這一節我們將會介紹 *makefile* 的基本語法和結構，讓你得以開始使用 make。

makefile 檔案中一般採用「從上而下」（top-down）的結構，所以預定會更新最上層的工作目標（通常叫做 all）。下層工作目標用來讓上層工作目標保持在最新的狀態，比如用來刪除無用之臨時檔案的 clean 工作目標，應該放在最下層。正如你的預期，工作目標並不是非得檔案名稱不可，你可以使用任何名稱。

在前面的範例裡我們所看到的是經過簡化的規則。下面是較完整（但可能仍然不夠完整）的規則：

```
target1 target2 target3 : prerequisite1 prerequisite2
        command1
        command2
        command3
```

冒號的左邊可以出現一或多個工作目標，而冒號的右邊可以出現零或多個必要條件。如果冒號的右邊沒有指定必要條件，那麼只有在工作目標所代表的檔案不存在時才會進行更新的動作。更新工作目標所要執行的那組命令有時會被稱為命令稿（command scripts），不過通常只會被稱為命令（commands）。

每道命令**必須**以跳格字符（tab character）開頭。這個（隱含的）語法用來要求 make 將跟在跳格之後內容傳給 subshell 執行。如果你不經意地在非命令列（noncommand line）的第一個字符上插入了一個跳格，在大多數情況下，make 將會把其後的文字當成命令來解譯。如果你很幸運，這個誤入歧途的跳格將被視為語法錯誤，你會因而收到如下的訊息：

```
$ make
Make□le:6: *** commands commence before □rst target.  Stop.
```

我們將會在第 2 章《規則》討論錯綜複雜的跳格字符。

make 會將井字號（#）視為註解字符（comment character）。從井字號開始到該列結束之間的所有文字都會被 make 所忽略。你可以對做為註解的文字列進行縮排或前置空白符號。註解字符 # 並不會在代表命令的文字列中導入 make 的註解功能，這一整列（包括 # 及其後的字符）會傳給 shell 執行。這列文字的處理方式取決於你所使用的 shell。

你可以使用標準的 Unix 規避字符（escape character）— 倒斜線（\）— 接續太長的文字列。倒斜線一般用來接續太長的命令，也可用來接續必要條件。稍後我們將會探討處理過長必要條件的其他方法。

你現在已經有能力撰寫簡單的 *makefile*。接下來我們將會在第 2 章探討規則的細節，在第 3 章探討 make 變數，以及在第 5 章探討命令。所以現在你應該避免使用變數、巨集以及多列命令。