序

這本書的英文版近一年前在美國 Amazon 公司出版後，很快在美國，加拿大，德國，和日本都有售出。最近，又獲得美國最權威的書評機構 Bookauthority 的推薦，並評為最佳網路程式設計書籍。今將之譯成中文，以享國內讀者。

這本書含有我在美國電腦軟體工業界，橫跨好幾個領域，逾三十幾年的實際寶貴經驗的精華，是任何想成為世界頂尖軟體工程師或總工程師者，所必知的知識與必備的技能。深信讀者會終身受益。許多的留美電腦博士（大多在美國已工作很久） 買了與讀了之後都說，這本書的內容既廣泛又深入，他們從中學到了很多從其它書學不到的寶貴實際經驗，知識，與技能。

在計算機系統及網路程式設計上，看這本書就對了！但願這是一本您一輩子都想帶在身邊的書！

陳金追 謹上 2022 年 2 月

軟體開發與軟體工程過程

2

CHAPTER

這一章介紹所有軟體工程師每一天都在使用與經歷的軟體開發與軟體工程（software engineering）過程。這些過程存在的主要目的，是在於提升生產力與軟體產品的品質。知道與熟悉這些過程，對於提高每位軟體工程師的生產力與品質，助益良多。

2-1 軟體開發過程

這個主題很重要。因為，從我在美國電腦軟體工業界工作了三十幾年的經驗，幾乎每年，公司都會聘請一些剛從學校畢業的電腦碩士與博士，而這就是他們需要掙扎一陣子的領域。

軟體工程師們每一天的工作，就是開發軟體產品或組件。這一章介紹他們每天經歷的軟體開發與軟體工程過程。

一般而言，軟體開發有下列的步驟：

1. 已知一個問題，需要用軟體解決這個問題。或是在現有產品上增加一個新的特色。
2. 做可行性的探討，與預估大約需要多少時間。
3. 訂出一個高層次的**功能規格**（functional specification），讓大家評論（review）。

4. 訂出一個更詳細的**設計規格**（design specification），讓大家評論。可能做更準確的工作量預估。
5. 開始寫程式（coding）。
6. 從事單元測試。
7. 程式碼評論。
8. 進行是否有退化（regression）的測試。
9. 正式提出並併入你的程式碼（submit or merge code changes）。
10. 整合性測試。
11. 性能測試
12. 壓力測試

所有軟體開發計劃都是從有一個問題待解決或現有產品必須增加一個新特色開始的。這個主意可能很大，是開發一個全新的產品，也可能小小的，就在現有產品上增加一個新的特色。觀念可能來自某位大企業家的創業理想和美夢，也可能是和客戶會談時客戶所提出的要求，也有可能是某位經理或工程師對於改進產品功能的一個意見。

一旦有個主意在手，最重要的是要先有明確的問題論述（problem statement），清楚地說明有什麼問題要解決，以及需要做些什麼。這第一步最重要。關鍵就在於能真正了解要解決的問題，且能把問題說清楚。

緊接第二步就是可行性的探討。這有時需要做點研究才能得知，有時也可能會花很多時間。一般而言，幾乎所有軟體問題，都是可以解決的，只是時間、人力與經驗的問題。有時也必須先試著做出產品簡單的**雛形**（prototype），才能真正知道題解是否可行。在製作雛形的過程中，你會進一步了解，有那些問題必須解決，以及實際會碰上什麼困難等等。根據問題的大小與複雜度的不同，這一步驟可能必須花上幾小時、幾天、幾週、幾個月，或甚至幾年不等的時間。

在知道一切可行之後，下一步就是寫出一個高層次的功能規格，闡述待解的問題是什麼，以及擬定的題解是什麼，有包括那些功能單元。功能規格必須維持在高層次，對問題與題解做個很宏觀的概要描述，主要讓公司的高層主管

能看懂與了解，進而支持與批准。功能規格必須說明什麼功能包括在內，以及什麼功能不包括在內。功能規格寫好以後，必須經過各方評論（review）與取得批准。

功能規格核准了之後，下一步就是撰寫更詳細的設計規格。設計規格目的是給像總設計師（architect）與其他技術工程師們看的。它應該含有更多且更詳細的技術細節在內。解釋打算如何解決各式各樣的技術問題，畫出題解的主要功能方塊和單元，主要由那些函數達成，包括有那些介面、資料結構、那些階層（layers），以及各階層或單元之間如何溝通等等。說明主要問題在技術上如何解決是功能規格的重點。

功能規格可列出解決問題所必須設計和撰寫的有那些函數, 其名稱，每一函數所必須用到的參數（parameters），所需的資料輸入，以及產生什麼樣的輸出結果或錯誤碼等等。其甚至可列出一些概略的程式碼（pseudo code）。

除了敘述問題在技術上要如何解決之外，設計規格典型還會包括諸如性能、安全、測試、文書（documentation）等方面的計劃，以及甚至是還有那些問題待解。其甚至可以包括必須用到那些資源（包含人力）等。

設計規格經核准後，下一步就是實際撰寫程式碼（coding）了。

程式碼撰寫完成後，就是測試了。作為一個軟體開發的工程師，通常必須做單元測試（unit test）。我一定都有做。每一件軟體產品都需要做很多測試。在設計規格撰寫的同時或之後，一個詳細說明這產品要如何測試的測試計畫書（test plan）也必須出爐。測試計畫書載明產品測試的計畫與執行，是開發自動的測試，或是人工的（manual）測試，抑是兩者兼具。需要多少資源做測試，以便能真正測出產品確實達成了其設計目標，而且沒有錯蟲（bugs）。測試計畫書可以是單獨的文件，也可以併在設計規格內。

一個測試，可以是自動的，也可以是人工的。最好是自動的。自動測試程式寫好後，每次只要需要，就可以執行一次，省時省力多了。很多產品都是每天晚上都自動將所有的自動測試程式跑一遍的。在有些公司，單元測試是要開發軟體的工程師自己寫的。但其他公司則由專門負責測試的部門來寫，兩者都有。

單元測試之後，通常就是對手或同事的**程式碼評論**（code review）了。這個步驟非常重要。其所根據的道理就是四隻或八隻眼睛勝於兩隻。多找一些人過目一下，總會比較放心。免得開發者一時有了疏忽。有一點很重要的是，在你正式提交併入你的程式碼與改變之前，最好自己先把退化測試全部跑一遍，確定你所增加或更改的，沒有造成任何退步或退化。

對手或同事評論之後，通常就是開發計畫領導者的批准，然後，這些新的程式碼就提交併入**原始程式控制系統**（source control system）內，讓大家都看到見，也變成是產品的一部分。在這時候，若在併入這些新的程式碼之後，產品無法建立（build），那就要解決,看不能建立的問題是出在那兒。

在新的程式碼併入之後，除了有時產品建立會有問題，必須解決外，跑退化測試（regression tests）時，有時也會出現問題。這些也都必須要一一去了解和解決。

一個產品，幾乎每天都有人在改或增加，為了提早發現問題，確保產品品質的穩定，通常每個產品都會每天晚上重新建立一次，並且執行所有或絕大部分的退化測試一次。這樣，萬一有新的改變造成任何退化，可以及早發現與修復。若退化很嚴重，有時還得必須將造成退化的改變暫時先拿掉。

記得，避免造成任何退化至為重要。所以，每一位軟體開發者，都要養成，在正式提交併入你的程式碼或改變之前，一定要將有關的退化測試都先跑一遍，確定你所改或加的東西，沒有造成任何的退化後，才能正式提交併入你所改的。

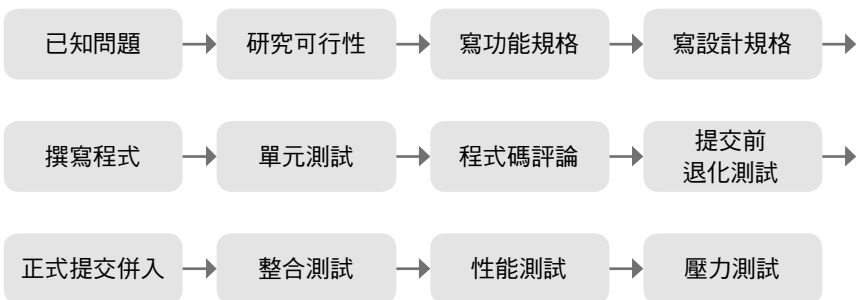

圖 2-1 軟體開發過程

在正式提交併入你的異動後，若沒問題，那最後一步就是整合測試了。除了功能上的整合測試外，很多產品經常還會做性能測試（performance test）與壓力測試（stress test）。

圖 2-1 所示即為一般軟體開發過程所包含的步驟。在現實上，不見得整個開發過程，從頭到尾都會很順利，有時會有進一步退兩步的情形。所以，其中的某些步驟，可能必須來回重複幾次，這並不奇怪。舉例而言，有時在你開始撰寫程式之後，你會發現，原來的設計，某些地方可能出了錯或必須稍作調整。因此，你得再回去更改你的設計規格，有時甚至也必須更改功能規格的也有。在測試時發現問題，必須回頭更改程式的也不是少見。總之一句話，真正做了以後，才會發現有一些從來沒想到的問題。這就是有些步驟必須來回反覆一、兩次的主要緣故。當然，這也和經驗有關，你的經驗愈豐富，出現這種情況的可能性就愈低。

下一節，我們討論一個經常會發生的實務問題。

2-1-1 程式碼評論的忠告

在軟體開發上，同儕間的程式評論（code review）是很稀鬆平常，對軟體品質的提升，也是很有貢獻的。其理論就是兩個或多個腦袋會勝過一個。事實的確是這樣，尤其是當有新手加入時。新手在做東西或更改產品時，若能有舊的，對產品比較熟悉的人稍微看一下，總是會比較放心，至少不會出現太離譜的差錯。此外，有時即使老手上路，但觸及或更改的是其它他比較不熟的領域，也會經常出錯。這時，若有對那一部分比較熟悉的人過目一下，多少總有助益。

所以，整體而言，程式評論絕對是正面的。但是，在工業界這麼多年，我也見識過很多負面的。總是，人有百樣。有些人會抓不到重點，有些人則會想藉機想整人，這些都是不對，也是不好的。

程式評論，最重要的是要能抓住重點，要把目標與重點放在功能上、正確性、性能、安全，與有無錯誤或造成退化上。但我發現有許多工程師，每次評論的都是有關個人的偏好，表面的細節問題，或風格與格調。這很不應該，是走偏了。記得，這些方面及項目對實際產品的品質根本沒有真正的影響。把時間和精力花在這些方面，不僅實際對產品品質的提升沒有幫助，反

而經常會造成彼此的爭論，傷害同事彼此之間的感情，很不值得！每個人有自己不同的品味，不可能每人相同。因此，評論必須避開。評論的主要目的，是要藉由更多人的過目，找到錯誤，加以更正。而不是在某些細節或格調上，爭論或爭執。評論者若要做任何評論，千萬要記得是針對功能的改善、正確性、性能與安全的提升、使用的簡易性、可靠度的提升等建言，才有意義。若針對的是小細節、美觀、個人喜好，或格調，就不值得，是在浪費時間。

有些公司總會訂有**寫碼規範**（coding standard），尤其是大公司。我發現這些大公司的寫碼規範，經常有很多過時與不合理的地方，也有很多是上面所提必須加以避免的。我看到了很多值得改進的地方。

2-2 原始碼控制系統

幾乎在任何公司都一樣，電腦軟體產品的開發，幾乎都是很多工程師同時一起努力的。換言之，許多工程師一起同時更改整個產品的程式。因此，兩個或兩個以上的工程師，欲同時更改同一個原始程式模組（source code module）或檔案的情形，經常出現。為了協調這麼多工程師的共同開發，並確保整個產品之原始碼的正確性，幾乎所有的公司都會把產品的原始程式碼，放在某種**原始程式碼控制系統**（source control system）下來管理。

原始程式碼控制系統是一種軟體。它的主要功能是記住每一原始碼檔案每一次更改的內容，協調不同工程師更改同一原始碼檔案的活動，確保彼此不會弄掉對方的更改內容，以及在正式提交併入之前，其他工程師看不見你所更改的內容等等。總之，原始程式碼控制系統讓許多工程師能共用產品的所有原始程式，能共時更新而不致於相互干擾，確保原始程式的正確性。

計算機工業界有許多不同的原始程式控制系統。有些很貴，但有些是開放原始碼（open source），不必用錢買的。有許多公司是專門開發這種產品，出售賺錢的，像 IBM 的 ClearCase 與微軟的 Source Safe 等。不用錢的產品包括 CVS、Perforce、Subversion，還有其他。有些大公司因自己內部所需，自己開發原始碼控制系統，自己用，不賣的也有很多。此外，有些作業系統軟體，買來時就附有原始程式碼控制系統的也有。譬如，AT&T UNIX 上有 SCCS，BSD UNIX 上也有 RCS（Revision Control System）。

不論你所用的是那一原始程式碼控制系統，其基本的功能大致雷同。只是指令的名稱可能都不一樣。因為這沒有工業界標準，因此，彼此之間的功能可能都有些許的差別。這一節，我們會將這些不同系統的共同核心功能以及執行各種常用作業的指令，作一簡單介紹，讓讀者有些許的概念。

原始程式控制系統的使用，通常有幾個步驟，以下我們就介紹這些基本共通的步驟。

1. 建立一個能看到所有原始程式檔案的**視野**（view）或**玩沙盒**（sandbox）。這個步驟建立一個能讓你看到你所開發之產品的所有原始程式碼檔案，能自己編譯所有程式碼，以及能建立一份只有你自己看得到的產品的一個環境。它讓你能擁有一個完全屬於自己的空間，可以更改原始程式檔案的內含、建立產品、執行測試等，而完全不會影響到其他人。做這個步驟的命令，有的叫 mkview、createview 或 mksandbox。在 IBM 的 ClearCase 則叫“ct mkview”。

2. 在原始程式碼控制系統內，建立一個新的原始程式檔案。

   這個命令一般叫 mkelem、create、ct mkelem（在 ClearCase 上）、cvs add（在 CVS 上），或 admin（在 SCCS 上）。

3. **借出**（check out）一個原始程式檔案，以便更改。

   在所有原始程式控制系統上都一樣，你必須先借出一個原始程式檔案，然後才能更改。

   借出檔案有兩個模式，你可將這個檔案加鎖（locked），讓其他人都不能與你同時借出，只有你可以。你也可以借出不加鎖，讓其他人也可以跟你一樣，同時借出。

   借出不加鎖的模式一般人最常用。但這有一點必須注意的是，在多人同時借出同一個檔案時，最先**還入**（check in）的人，比較稍佔便宜，因為，他的還入不會看到其他人更改的內容，不會有更改衝突必須排解。從第二個以後還入的人，就會看到先前還入這同一檔案的人的更改內容，因此，必須將兩人的更改內容，正確地合併（merge）在一起。視兩人更改的內容重疊多少而定，這合併的工作可很簡單，但有時也可能稍微麻煩。

借出原始程式檔案的命令，一般稱之為 checkout、co、ct co（ClearCase）、cvs checkout（CVS），或 get（SCCS）。

4. 使用文書編輯程式（editor），更改你所借出的檔案。你可以用任何你喜歡的文書編輯程式，如 vi 或 emacs 等。

5. 編譯與測試你所改的東西。

6. **還入**你所更改過的檔案。

   這個步驟將你所更改的內容，實際紀錄在原始程式控制系統內，也讓其他人可以看得見你所更改的內容。在你正式還入之前，你所改的，別人是完全都看不見的。此外，記得，在你還入之前，你的更改內容，是沒有記載在原始程式控制系統內的。是以，倘若你不慎丟掉了你所改過的原始檔案，那就無法找回的。因此，你最好習慣上，隨時都將你所改過、尚未完成的檔案，自己備存一份，以防萬一弄錯，搞丟了你的更改內容。

   還入更改過原始檔案的命令，通常叫 checkin、ci、ct ci（ClearCase）、cvs commit（CVS），或 delta（SCCS）。

7. 事先合併其他工程師在同一檔案上所做的更改。

   有時，你會同時借出一個檔案比較長的時間，而那之間其他工程師也更改了同一個檔案，且已還入。這時，你有兩個選擇，一是等到你真正還入時，再做合併。另一則是在正式還入前事先合併。事先合併其他工程師的更改內容，在 CVS 上，這個命令叫 cvs update。

8. 正式的**提交**（submit or merge）你所更改的內容。

   這一步驟因控制系統而異，並非所有控制系統都有此一步驟。

   在很多控制系統上，有了還入步驟就夠了。但其他的控制系統有加了這一步，這個步驟稱之 submit 或 merge。

9. 取消你的借出。

   有時，在你借出一個檔案，想更改之後，因故你又改變了主意。這時，你必須取消你的借出。這個命令叫 uncheckout、unco、ct unco（ClearCase），或 unget（SCCS）。

10. 經常，你會想知道，一個檔案，在某兩個不同版本之間，究竟改了什麼，以查出某人在某時所更改的內容。譬如，找出檔案版本 1.3 與 1.4 之間的變動。

    這個命令的名稱，一般叫 diff、ct diff（ClearCase）、cvs diff（CVS），sccsdiff 或 delta（SCCS 上）。

11. 除了以上所介紹的這些最主要且最常用的步驟與命令外，每一原始程式控制系統，一般都還會有其他的作業命令。這些就讓讀者在真正需要用到時，再去學了。舉例而言，有一可以列出一個檔案之全部更改版本與歷史的命令，一般叫做“ct lshistory”或“cvs history”。也有一個顯示你目前視野或玩沙盒之配置（configuration）的命令，叫做“ct catcs”等等。

總之，一般原始程式控制系統的使用，包括了以下這些步驟：

1. 建立一個能看到所有原始程式檔案的視野或玩沙盒。
2. 在原始程式控制系統內，建立一個新的原始程式檔案。此一步驟包含借出在內。
3. 借出一個現有的檔案，以便開始更改。
4. 事先併入他人的更改內容。
5. 還入一個你更改過的檔案，讓其他所有人看得到你所更改的內容。
6. 正式提交你改過的所有檔案。有些控制系統並無此一步驟。

圖 2-2 所示，即這些步驟與命令，在 ClearCase、SCCS 與 CVS 上的名稱。至於其他命令的名稱，請直接參考你所使用之控制系統的文書。

圖 2-2 原始程式控制系統的基本命令

原始程式控制軟體			
作業	SCCS	ClearCase	CVS
開創一個視野或玩沙盒		ct mkview	
進入一個視野或玩沙盒		ct setview	
列出所有的視野或玩沙盒		ct lsview	
退出視野或玩沙盒		ct endview	
建立一個新檔案	admin	ct mkelem	cvs add
剔除一個新檔案		ct rmelem	
借出一個檔案	get	ct co	cvs checkout
取消檔案的借出	unget	ct unco	
列出我所借出之檔案		ct lsco ct lscheckout	
還入借出檔案與改變	delta	ct ci	cvs commit
列出不同版本之差異	sccsdiff, delta	ct diff	cvs diff
列出檔案之更改歷史		ct lshistory	cvs history
顯示出視野的規格		ct catcs	

2-3 軟體釋出過程

一個電腦軟體，一旦問世之後，一般都會一直不斷地更新，過一陣子就會有更新的版本**釋出**（released）。如第一版、第二版、第三版等等。

對軟體開發者而言，每一軟體釋出（或版本），都會有一預計的出廠日期，以及該有的功能與特色。當所有該有的功能都齊全時，整個產品的原始碼樹林（source tree），就會從原始程式控制系統中，拷貝一份**分出**（branch out）作為新版本釋出的候選。這時，程式原始碼系列或樹林，就會一分為二。原來控制系統下的，稱為**主分支**（main branch），會繼續往前行，作為下下個更新版本開發的主幹。主分支上所有的永遠都是最新的版本。另外，在剛分叉出來，作為下一版本候選的，也變成獨立，自己存在，以便能進一步測試、抓錯蟲、除錯蟲，把它穩定下來，直到可以釋出成新的版本為止。在此時，至少就有二個原始碼樹林存在。一個是主分支，另一個是下一新版本的分支。

這時，在新版本分支測試上所發現的問題，修復時，工程師除了必須將其修復更改的內容，提交併入新版本分支上之外，也必須同時將之提交併入主分支。如此，這個問題才不至於又出現在下下個版本上。這是品質控制上很重要的一個步驟，確保沒有退步或退化。

圖 2-3 所示，即一個軟體產品的釋出過程與分支。你可看出，時間久了，版本多了，原始程式的分支也就不斷地增加。該維護與支援的版本，也愈來愈多。

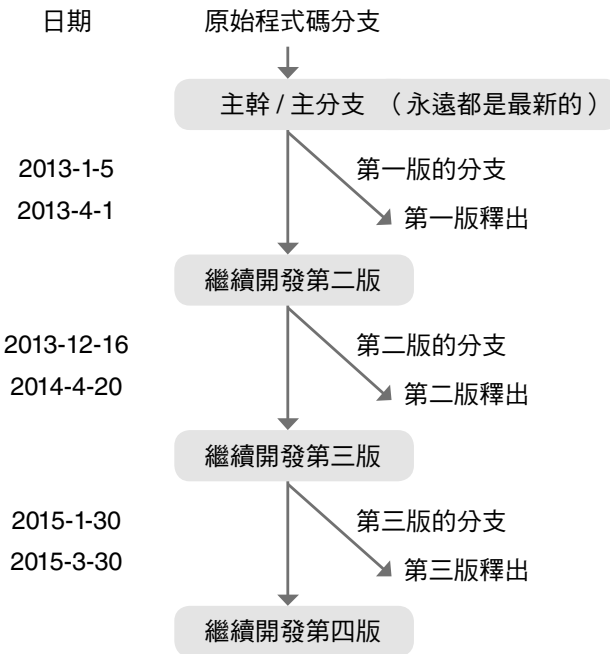

圖 2-3　軟體釋出版本與分支

2-4　產品建立的不同模式

一個軟體產品在出版送到客戶手上之前，都必須先建立與測試過。所以，在軟體開發的過程，必須經常建立產品無數次。建立產品就是將所有原始程式模組（source code modules）編譯，並連結成可執行的程式和可叫用的庫存函數。每一軟體產品的組成和建立程序或許會有些許的差異。在小的產品，

總計的原始程式檔案並不多，可能只分佈在少數幾個檔案夾。建立整個產品可能只需要幾個指令，幾分鐘即可完成。但在一個大產品上，通常檔案數都有上萬個，分佈在幾十或幾百個不同檔案夾內。不同組成單元之間互相依靠，建立程序有個一定的順序。建立過程也都自動化，經常需要幾個小時才能完成。諸如作業系統、資料庫管理系統、網路系統、群集系統等軟體，幾乎都這樣。

如果你是釋出工程師（Release Engineer），那負責每天建立整個產品，以及檢修各式各樣的建立上的問題，就是你每天的工作。假如你是軟體開發工程師，那你可能平常就只改產品的某一部分。通常你只須重建你所更改的那部分，不需全部重建。但在少數情況下，有時你還是得整個產品全部重建一次。譬如，你改到了某些很多單元都用到的庫存函數或前頭檔案（header files）之類的。每次改了某些東西時，有多少東西必須重建，那就是你的職責要去了解的。最保險的做法是整個產品全部重建，但那通常太費時了。日子久了，你會知道只要重建那一部分就可以的。

為釋出或除錯而重建

記得，視你重建整個產品之目的的不同而定，重建的方式略有差異。一般公司的做法是，在平常，產品還是處於正在開發與除錯階段時，產品都會建立成**除錯模式**（debugging mode）。這意謂，叫用編譯程式時，人們都會加上 -g **選項**（option），將程式編譯成含有符號名稱表格（symbol tables），可以隨時立即用除錯程式（debugger）加以除錯的形式。

例如，下面就是將一個 C 語言程式編譯成除錯模式的命令：

```
cc -g myprop.c
```

可是，當接近開發完成，或產品要出廠給客戶時，這除錯模式通常就會關掉。取而代之的是改成編譯成**最佳化模式**（optimized mode），以使產品在客戶真正使用時，取得最佳性能。譬如，上述的編譯命令，就會改成：

```
cc -o2 myprop.c
```

這 -o 選項就是將程式的性能最佳化。它通常有不同的層次。在最佳化模式所編譯出來的程式碼，佔用空間會比較小，執行效率會比較高。

2-5 產品建立的工具

不同作業系統上所提供的程式建立工具，會有不同。許多公司也都提供專門用以建立軟體產品的**整合開發環境**（Integrated Development Environment, IDE）。這些通常是一圖型介面（GUI-based）的開發環境與工具，讓軟體開發的工程師，能更簡易的建立整個產品與除錯用的。IDE 通常因不同程式語言而異。實際的產品例子，在 C 語言上，有 IBM AIX 上的 VisualAge、Oracle/Sun Solaris 上的 Sun Workbench、微軟之 Windows 系統上的 Visual Studio 等。在 Java 語言上，有 Eclipse、NetBeans、IntelliJ IDEA 等。當然，還有其他許多產品了。C 語言的整合開發環境，通常是要用錢買的。Java 的 IDEs 有一些是免費的。

這一節，我們要介紹一個在 UNIX 和 LINUX 作業系統上自動附來的，非常流行的簡單軟體建立工具，那就是 make。跟 IDEs 比較，它不用錢，很簡單，但功能也很強大。

Make 這個工具，在 1970 年代在 UNIX 上就有了。現在，所有的 UNIX 與 LINUX 作業系統上，也都有，是附在一起的。另外，GNU 也有自己版本的 make。值得一提的是，make 是包括在 POSIX 標準內。但現在的 make，有許多不同的版本。這一章，我們要介紹的是，make 共通的基本功能。但請你記得，不同的 make 有不同的特色，它們並非是完全一模一樣的。

2-5-1 make

在 UNIX 與 LINUX 作業系統上，最常見的軟體建立自動化工具，就是 make 公共常用命令或程式了。Make 可以自動走下你的原始程式樹林，建立你產品的所有庫存檔案與程式。Make 會讀取你在每一檔案夾上所置放的一個叫做 makefile 的規格檔案，從中得知它在目前這個檔案夾必須做些什麼。若你叫它編譯一些檔案，它會照做。若你叫它走到底下的幾個檔案夾，它也會照做。記得，你在 makefile 內規定依照什麼順序建立整個產品，make 就會照著辦。

欲用 make 建立你的整個軟體產品，首先你必須將你產品的所有原始程式組織成一個樹的結構，並在每一檔案夾建立一個 makefile。每一個檔案夾

內的 makefile，必須清楚地告訴 make，它必須做些什麼，包括在目前這檔案夾內必須做些什麼，以及在底下下一層有幾個檔案夾，也必須處理。

倘若你在 makefile 內規定的都正確無誤，指明建立每一目標（target）的規則，以及各目標間互相依賴的關係與順序，那建立整個產品就很簡單，你只要換到你希望建立之產品或組件所對應的最高檔案夾，然後打入 make 命令，make 就會全自動，把一切建立得好好的。就這麼簡單！

根據你在每一 makefile 中所訂的規則，make 會知道它必須走到再下一階層的那些副檔案夾（subdirectories），建立什麼組件或單元。且遵照你所規定的先後次序進行。建立整個產品可能需要花上幾個小時，但你只須做一件事，那就是跑到產品或組件的最高階檔案夾，下一道 make 命令。一切就會全部自動完成。

以 make 工具將建立軟體產品全部自動化有下列的優點：

1. 很簡單

   就如我們以上所說的，只要你把 makefile 都建立好了，那只要下一道命令，整個產品或組件，就會全部自動建立完成。

2. 全自動

   一切魔術神奇，都在 makefile 內。只要 makefile 建對了，整個產品建立過程，都是自動的。

3. make 是聰明的

   make 程式是有智慧的。它一個很重要的特色，就是能自動比對檔案的時間（timestamp）。找出有那些檔案已經過時，必須重建。然後只重建那些過時的檔案。倘若某些目標檔案沒有過時，不需要更新，那 make 就不會將之重建。譬如，在某一檔案夾內，目標檔案是 myprog.o。myprog.o 是 myprog.c 編譯而成的。在建立的過程中，只要 make 發現 myprog.o 已經存在，而且檔案的時間是在原始程式檔案 myprog.c 之後，那 make 就會自動跳過編譯這個檔案。因為，myprog.o 已經在上一次編譯過了，原始檔案在那之後，沒再更新。因此，不必再重建。

換言之，make 很聰明，它會知道那些原始檔案在上一次建立之後，有更新，並只重建那些檔案。當然，這假設你 makefile 建立得正確無誤。

4. 很容易

在讀完以下幾個章節之後，你會發現，其實學會怎麼寫 makefile，很容易，一點不難。在下面的幾個章節裡，我們會舉例，教你如何撰寫 makefile，讓你能將建立你的產品，全自動化。

至此，你已明白，一切魔術就在 makefile 裡。只要把 makefile 弄對了，make 就會自動走下整個原始程式的樹林，找到那些必須建立或重建的檔案，將之建立。重點是，若目標 A 必須依賴目標 B，那目標 B 就必須在目標 A 之前，先建立。

Makefile 的名稱

在每一檔案夾內的 makefile，可命名為 makefile 或 Makefile。從這兩個名字之中選一個，但不要兩個都有。若你兩個都有，絕大多數的 make 都會優先採用小寫的 makefile。

事實上，makefile 你也是可以使用其他的檔案名稱的。譬如，你的產品可能支援了多種不同的作業系統，而所需的 makefile 在這些作業系統上都有點差異。這時，在同一檔案夾內，你就可以有多個 makefile，並分別採用不同的名稱。譬如，如以下所示的，分別叫做：

Makefile.linux

Makefile.mac

Makefile.aix

⋮

當你的 makefile 不叫 makefile 或 Makefile 時，在你下 make 指令時，你就必須在你的 make 指令上，告知 make，你的 makefile 的名稱。舉例而言，你的 make 指令可能就必須改成像下面的樣子：

```
make -f Makefile.linux
```

而不再是只有 make 了。這個命令與純粹只有 make 一樣，只是它以-f 選項告訴了 make，要從檔案 Makefile.linux 中去讀取建立規格。

當然，你也可選擇寫一個適用多種不同作業系統的單一 makefile。

2-5-2 Makefile

一個 makefile，主要就是敘述一些產品建立的規則。每一規則的格式如下：

目標 [目標 2　目標 3 …]: 依賴 1　依賴 2 …

[TAB 鍵]命令 1

[TAB 鍵]命令 2

⋮

平常，每一個規則就說明如何建立一個目標，因此，它只有一個目標。目標就是你所要建立之檔案的名稱。但有時，同一個規則可以有多個目標。換言之，多個建立目標共用同一個規則。在有多個目標時，彼此之間以空格分開。

目標之後必須有個冒號（:）。然後，若有的話，緊接就是目標所依賴的其他目標，簡稱做**依賴**（dependent）。一個目標可能沒有其他的依賴，也可能有多個依賴。依賴就是目標的**先決必要條件**（pre-requisite），是必須先有不可的。當你在一個規則上，列出目標有依賴時，make 就會自動先去建立依賴。然後，再回來建立目標本身。倘若同時有多個依賴，那 make 會根據所有依賴先後出現的順序，依序建立完所有依賴之後，再回來建立目標。這就是你告訴 make，如何依序一一建立各項目標的方法。以目標與依賴的關係，以及你所列的依賴的順序，井然有序地訂出所有目標的建立順序。

在目標和依賴一行下面，你必須闡明如何去建立一個目標，需要執行那一或那些命令。請記得，如何建立目標的每一個命令之前，必須有一 TAB 鍵。除非每一命令之前都加有一 TAB 鍵，亦即整行以 TAB 鍵開始，否則，make 會變成看不懂你的命令。這時，make 會產生一個類似下面的錯誤訊息：

```
makefile: 10 : *** missing separator. stop.
```

就上面我們所列的規則而言，目前這個目標有二個依賴。建立這個目標，必須執行兩個命令。所以，在執行這個規則時，make 會先去建立第一個依賴，完了之後再去建立第二個依賴。在完成建立所有的依賴以後，make 會回來執行命令 1 與命令 2，完成此一目標的建立。在建立每一個依賴時，make 會去找到說明如何建立這個依賴的規則，亦即，找到那個目標就是這個依賴的規則。

每個目標的名稱是你所選定的。記得，它最好是一見分明，一看就知道它是什麼。一般的原則，它就是你所須建立之檔案、庫存或程式的名稱。

一個 makefile，可以有多個不同的規則在內，每一不同規則建立一個不同的目標。

在你引用 make 命令時，在命令上，你可指明你只要建立某一或某幾個目標。例如，

```
make clean prog1 prog2
```

這個命令等於教 make 建立 clean，prog1，與 prog2 三個目標。倘若你只打 make，命令上沒指明任何建立目標，那 make 就會去建立在 makefile 上所碰到的第一個目標。一般的習慣是把 makefile 內的第一個目標稱作 all，代表建立全部的意思。然後，這個建立 all 的規則，再告訴 make 如何去建立組成產品全部的每一單元或組件。

makefile 中可以加註解（comments）。每一註解的最前面第一個非空白文字，必須是 #。# 之前可以有空白。

以下是幾個 makefile 內之規則的例子。

1. 清除規則

```
clean:
  rm -fr *.o
```

每一個 makefile，幾乎都包括一個清除（clean）的規則。這個規則就是剔除所有建立的檔案，以便能從頭重新建立一切。這個目標通常去剔除目前已存在，建立過的目的檔案（object files）（亦即 *.o 或 *.obj 檔案），庫存檔案和程式檔案等。

在你每次想重新建立一切時，你就可先下一道“make clean”的命令，先清除一切，從零開始。有時，為了騰出所需的磁碟空間，你也會做這樣

的清除。平常，為了省時，你就不下這道命令了。因為，這樣可以把上一次所建立的留下，不必要重建的就不需再浪費時間去重建了。

所以，要清除一切時，你就執行以下的命令：

```
make clean
```

在執行時，make 就會去執行

```
rm -fr *.o
```

的命令。這命令指明那些檔案必須清除。

2. 從一個 *.c 檔案建立 *.o 檔案

```
prog1.o : prog1.c
  cc -c prog1.c -o prog1.o
```

這個規則說，目標 prog1.o 依賴著 prog1.c。建立 prog1.o 必須執行以下的命令：

```
cc -c prog1.c -o prog1.o
```

make 程式經由讀取 makefile 中的規則，與執行這些規則，達成產品的建立。

它會先看叫做 makefile 的檔案是否存在目前的檔案夾內，若有，則採用它。若沒有，它會再看叫做 Makefile 的檔案是否存在，若有，則用之。若沒有，那 make 就會出現“no makefile found”（找不到 makefile）的錯誤。前面說過了，假設你把你的 makefile 稱作 buildlib.mk，那你的命令就應改成

```
make -f buildlib.mk
```

2-5-2-1 典型的 makefile 目標

有幾個目標與建立步驟，是一般 makefile 經常用到的。下面三個就是例子。

make clean

make all

make install

以上所列的三個建立步驟，通常是照著所列順序執行的。

首先，“make clean”先清除所有舊有建立的檔案。緊接，“make all”就重新建立一切。最後，“make install”就是把所建立好的產品，安裝在目前這系統的某個檔案夾上。這個步驟通常將產品必須給客戶的所有庫存、程式和其他相關檔案，全部拷貝一份，存在某一個檔案夾上，或在產品真正安裝（install）時，存在安裝的檔案夾內。所有這些檔案的結構，就像正式安裝以後，產品所有檔案的結構。

2-5-3 Makefile 中的代號

makefile 的**代號**或**巨集**（macros）就像 C 語言程式中的代號或變數一樣。就像你可以在 C 語言程式中，定義你自己想用的代號一樣，你同樣可在 makefile 內，定義並使用你自己的代號。它的功能就像一個變數名稱一樣，以一個字眼或符號，代表某樣東西。使用代號的好處是，萬一你想或必須改它，那就只須改一次（就是定義那裡）就得了。代號的名稱，由你自己選定。習慣上，一般都用大寫字母做代號。

最常見的 makefile 代號，就是用以代表 C 語言編譯程式，連結程式（linker），編譯程式之選項（compiler options）與旗號，以及庫存等等。譬如，下面就是一些例子。

```
CC = cc
LD = ld
CFLAGS = -c -g
LDFLAGS =
LIBS = -lnsl -lpthread
```

這些定義意謂著，在這些定義之後，在 makefile 內，每一個 CC 出現的地方，就等於是 cc，每一個 LIBS 出現的地方，就是代表“-lnsl -lpthread”。代號在經過像上面這樣定義過後，真正要用它時，必須將之放在 $（ ）之內。譬如，在 makefile 內使用代號 CC 時，您就必須寫 $（CC）。請注意到，每一代號所代表的東西，不必用單括弧或雙括弧括起。

特殊代號

以上所提的是，你可以自己定義和使用的 makefile 代號。除了這個以外，有一些代號是 make 自己事先已定義好，你可以使用的。這些 makefile 特殊代號包括下列幾項：

1. $@ 這個代號代表目前正要建立的目標。
2. $< 這個代號代表引起目前之建立作業的檔案。例如，目前必須建立的目的檔案，如 prog1.o。
3. $? 這代表自上次建立以後，有更新過，而這次必須重建的依賴。

   這就是時間比目前欲建立之目標還新，且是目前目標所依賴，這次必須重建的東西。譬如說，是目標 prog1 程式所依賴的 prog1.o 與 mylib.o。
4. $* 這就是目標和依賴所共用的檔案名的字首（prefix）。換言之，$* 是目前目標的名字，截去或拿掉尾部（suffix）。例如，在一 .c .a 的建立規則上，$*.o 所代表的，即是相對應於必要條件之 .c 檔案中，所有必須重建的 .o 檔案。

舉例而言，有了這些特殊代號之後，在 makefile 內，你就可以有以下這些非常通用的規則，不必提及任何特定的檔案名，就能告訴 make，如何將一個 *.c 的檔案，編譯成一個 *.o 的檔案。這些規則，適用於任何檔案名的 *.c 檔案：

```
.c.o:
$(CC) -c $(CFLAGS) $<
```

或是

```
.c.o:
$(CC) -c $(CFLAGS) $< -o $@
```

就如我們所說的，這些是通用規則，適用將任何名稱的原始程式檔案翻譯成目的檔案（*.o）。注意，這兩個例子假設 CFLAGS 代號中不含 -c。若有，那命令中的 -c 就不需要了，要剔除。上述兩個命令唯一的不同，就是第二個命令指出了輸出檔案 “-o $@” 。由於這是既定的（default），因此，有或沒有寫，結果都一樣。

注意到，這規則上的目標寫的是 .c.o，它的意思是 “從 *.c 檔案建立 *.o 檔案”。假若需要編譯的原始檔案叫做 file1.c，那這時 $< 代表的就是 file1.c，而 $@ 所代表的就是 file1.o。假若我們把 $* 所代表的也用進來，那上述的規則也可寫成

```
.c.o:
$(CC) -c $(CFLAGS) $*.c
```

由此你可看出，三個不同的規則，其功能事實上都是一樣的。

假若你使用的是 c++ 語言，那規則中的目標就必須改成 .cpp.o，而不再是 .c.o 了。

值得一提的是，以上所提的這個建立規則，是 make 的隱含規則。因此，即使你不明確寫出，make 也是知道的。

傳統代號

事實上，makefile 所事先定義好的代號，還有其他。你若執行“make -p”命令，就可以看到全部事先定義好的代號。這包括下面幾個代號：

CC：C 語言編譯程式。既有值為 cc。

AS：組合語言編譯程式。既有值為 as。

AR：備存（archive）命令。既定值為 ar。

命令中的代號

記得，代號也是可以不在 makefile 內定義，而直接定義在 make 命令上的，像

```
$make CC=/usr/bin/cc
```

就是一個例子。

2-5-4 命令之前可加的特殊字號

有幾個特殊的字號（special characters），如果你在 makefile 內的命令之前加上這些特殊字號或文字，就可以稍稍改變它們既定或既有的行為。

首先，平常 make 程式在執行一個命令之前，都會將這個命令顯示出在螢幕上，讓你知道它正在執行那一個命令。倘若你在這個命令之前加上@文字，那 make 就會默默地執行命令，但不會將這個命令印出在螢幕上。這可用來減少 make 命令在螢幕上所顯示出的資料量。

make 命令在執行產品建立的作業時，若有碰到錯誤，它通常會立即停止。有時，你會希望 make 不要一碰到錯誤就馬上停止。假若你希望 make 在執行一個命令時，若碰到錯誤不要停止，繼續往下執行，那你就可以在這個命令之前加上一個減號（-）。

平常，若你執行“make -n”命令，make 就會印出所有它應執行的命令，但不會真正去執行它們。不過，若你在某一個命令之前加上了一個加號（+），那“make -n”就會真正去執行這個命令。

圖 2-4 示範特殊字號功能的 makefile

```
all:
    @echo "  A command starts with @ sign"
    +echo "  A command starts with + sign"
    -echo "  A command starts with - sign
    echo  "  A command after an error"
```

圖 2-4 所示，就是一個很簡單的 makefile，用來示範這些特殊字符的功能。以下是這個 makefile 執行的結果。

```
$ cat makefile
  special:
      @echo "  A command starts with @ sign"
      +echo "  A command starts with + sign"
      -echo "  A command starts with - sign
      echo  "  A command after an error"

$ make -n special
echo "  A command starts with @ sign"
echo "  A command starts with + sign"
  A command starts with + sign
echo "  A command starts with - sign
echo  "  A command after an error"

$ make special
  A command starts with @ sign
echo "  A command starts with + sign"
  A command starts with + sign
echo "  A command starts with - sign
/bin/sh: -c: line 0: unexpected EOF while looking for matching `"'
/bin/sh: -c: line 1: syntax error: unexpected end of file
make: [special] Error 2(ignored)
echo  "  A command after an error"
  A command after an error
```

```
$ make clean
(cd lib; make clean)
rm -fr file1.o file2.o mylib.a
(cd prog1; make clean)
rm -fr prog1 prog1.o

$ make
(cd lib; make)
cc -c file1.c -o file1.o
cc -c file2.c -o file2.o
ar rv mylib.a file1.o file2.o
  ar: Creating an archive file mylib.a.
  a - file1.o
  a - file2.o
(cd prog1; make)
cc -c prog1.c -o prog1.o
cc  -o prog1  prog1.o ../lib/mylib.a

$ ./prog1/prog1
Function func1 was called.
Function func2 was called.
```

再次提醒一下讀者，make 的基本功能在 POSIX 標準有規範，所以，這部分在所有作業系統都一樣。但是，每一作業系統下的 make，通常也都會有些微的差異。詳細情形，請讀者查看每一作業系統的文書。

這本書幾乎每一章節都有一些程式範例。屆時我們也會提供一個簡單的 makefile，一個是 Linux 作業系統上用的，另一個是 Apple Darwin 上用的。分別叫做 makefile.lin 與 makefile.mac。至於其他的平台，再請讀者拷貝並修改一下。

誠如我們說過的，makefile 有許多不同的建立方式。每一個人所寫的 makefile 可能都稍有不同。這本書裡的 makefiles，旨在簡單、迅速。因此，它們可能並非最佳狀態。但卻簡單又達目的。我們所給的，讀者只要改一下定義 OBJS 和 PROGS 代號兩行，將你所欲建立之目的檔案與可執行程式的名稱換上，就可以了，省時省事。

2-6 退化測試組套

絕大多數軟體產品都是一邊開發，一邊測試的。因此，在最後，都有了一套或一組測試在。在第一版本問世了之後，這套測試就變成了**退化測試組套**（regression suites）。只要產品有進一步的翻修或更改，這一套退化測試組套就可以用來確保新的改變沒有讓整個產品退化。所謂產生退化，就是原

來動作（works）的功能，現在反而不動作了。這是很丟臉，很不應該，也絕對不容許發生的。原來動作的，好的，就應該一直好下去。假若因為修了某個錯蟲，或新增了一些新的功能，就造成有退化的現象，那就是不對的。

增加新的功能，或為了修錯蟲而改變時，應該也要增加新的測試。沒有寫測試的缺點是，萬一有人不小心破壞了那一部分的功能，就沒辦法及早發現了。

及早發現產品有錯蟲或退化，至關緊要。若等到客戶用了以後，再發現那就不是很好，會影響產品的品質及聲譽。

測試必須愈完整愈好。所有的每一項重要功能及特色，都要有相關的測試守護著。所有的正常狀況、邊際狀況、非正常狀況，都應該測試到。正面測試絕對必要，負面測試也要盡量齊全。事實上，嚴格說來，該有的測試是幾乎無限量的，寫不完的。重點是，測試愈齊全與完整，測試涵蓋的範圍愈寬廣，你對產品的信心就愈高，產品的品質也相對提高。

總之，一套完整的退化測試組套，是產品必備的組件之一，絕對少不得。

2-7 編譯式與解譯式程式語言

2-7-1 程式語言與執行的型態

電腦程式可以寫成各種不同的語言。這些程式語言包括 C、C++、Java、C#、PASCAL、BASIC、Perl、Python、FORTRAN、COBOL、ADA、組合語言、劇本語言等等，還有其他許多語言。

並不是所有程式語言都一樣或相近的。有些程式語言，如 C 與 C++，就比其他程式語言，如 Java 與 Perl，來的精簡、嚴謹，與高效率。有些程式語言，如 C、C++、Java，就比其他語言，如 COBOL 與 FORTRAN 來的通用。

另外，有些程式語言是編譯式（compiled）的語言，而有些則是解譯式或即譯式的（interpretative）。譬如，C、C++、FORTRAN、COBOL 與 ADA 等，就是編譯式的語言，而 Perl、BASIC 與劇本（script）語言等，即是解譯式的語言。有些語言（如 Java、C#、Python）等則兼具編譯與解譯。其都先編譯成中間碼，再解譯執行。

寫成編譯式語言的程式，必須先經編譯成最終機器語言或中間式語言之後，才能正式執行。但寫成解譯式語言的程式，不須事先編譯，就能立即執行，邊解譯，邊執行。執行寫成解譯式語言的程式時，你通常要再叫用另一解譯程式來解釋並執行你的程式，而不是像編譯式程式一樣，直接叫用執行你的程式。

解譯式程式語言的優點是，易於除錯，除錯時間通常比較短，程式也比較少行。但它的缺點是，通常比較高階、鬆散、效率比較低。有些甚至非常臃腫累贅，且不夠嚴謹。

相對的，編譯式程式語言需要比較多步驟才能抵達執行的階段，而且需要除錯程式（debugger）幫忙除錯。但它們通常低階一些，一般直接叫用作業系統核心的系統叫用（system calls），因此，效率很高，且控制較緊。編譯程式也可以將之性能最佳化。若將 C 語言與 JAVA，Perl 或 Python 相比，C 相對就簡單、精簡、快速、嚴謹與精準多了，功能也強多了。效率與嚴謹度的差別，是相當大的。這就是為何幾乎所有的系統軟體，都是以 C 語言寫成的. 這是未來幾百年不會改變的。

上面提到 Java 是一既編譯且解譯的程式語言。所有 Java 語言的程式都必須先經 Java 編譯程式（一稱為 javac 的命令或程式）編譯成一種叫做**位元組碼**（byte code）的中間語言，然後，才能再以另一稱為 java 的解譯程式，加以解釋與執行。Java 語言這樣做，主要目的是希望能將 Java 程式，只要編譯一次，就能在任何不同硬體機器上執行。這是因為，位元組碼是硬體獨立的，它與計算機之硬體（及中央處理器）無關。而實際最後解釋你的 Java 語言程式的 java 命令，才和中央處理器硬體有關。java 解譯程式在每一種不同中央處理器上都不同，它才能將與硬體完全獨立無關的中間語言位元組碼，解釋成每一不同中央處理器的機器語言，執行你的 Java 程式。

所以，用編譯程式編譯

- C 語言程式（將之譯成機器語言）
- Java 語言程式（將之譯成中間語言，稱為位元組碼）

用解譯程式解譯且執行

- 劇本（script）語言程式
- Perl 程式
- Java 程式的位元組碼

總之，一般而言，編譯式程式語言，稍微低階一些，但較嚴謹與高效率。解譯式程式語言稍微高階一些，但通常較鬆散、累贅，且沒效率。有些解譯式語言，如 Perl，使用了許多庫存函數進行各種不同的作業，不僅效率很差，還經常在不同作業系統、功能就有所差異。理論上，Perl 語言程式應該是完全可移植的（portable），但其實實情並不是這樣。

2-7-2 程式的編譯過程

大部分軟體工程師都使用編譯式程式語言在做軟體開發。圖 2-11 所示，即這種開發方式下，軟體編譯、建立，與執行的步驟及過程。

由圖 2-11 可看出，建立與執行一個程式，通常會用到其他的四個程式：編譯程式（compiler）、組合語言編譯程式（assembler）、連結程式（linker），與載入程式（loader）。這個例子牽涉兩個原始程式，myprog.c 與 utils.s。C 編譯程式將 myprog.c 編譯成機器語言，結果存在硬碟上。這個目的檔案在 Unix 上就叫 myprog.o，而在 Windows 上就叫 myprog.obj。組合語言編譯程式將組合語言原始檔案 utils.s 翻譯成機器語言，並將結果存在硬碟上。這目的檔案就要叫 utils.o 或 utils.obj。

絕大多數程式都由多個原始程式檔案建立而成。因此，欲形成一個可執行的程式，下一個建立步驟就是要叫用連結程式，將所有這些有關的所有目的檔案，

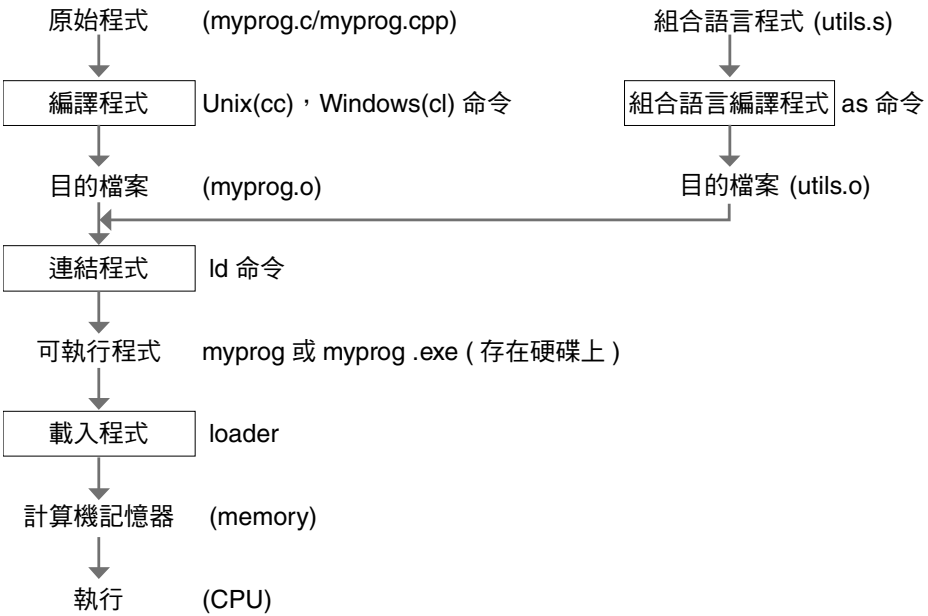

圖 2-11　編譯式程式的建立與執行過程

以及它們所叫用到的 C 庫存函數，全部連結在一起，以形成一個最終的可執行的程式。這個**連結程式**（linker）在大部分的作業系統，就是 ld 程式或命令。連結程式必須將你的程式所直接或間接叫用到的所有函數，包括庫存函數與你自己所寫的，全部找出並連結在一起，填入每一函數的位址，使程式變成一可以執行的狀態。這一個步驟最常見的錯誤就是有些函數或變數找不到“unresolved symbols”。遇到這個錯誤時，你就必須自己去找出那些連結程式找不到的函數或變數是定義在那個檔案或庫存內，並將它的名字加在現有的 ld 命令上。連結程式若大功告成，那它就會將可執行程式，存成一個可執行檔案，存至硬碟上。

你知道，每一個要讓中央處理器執行的程式，都必須存在記憶器內。因此，當你打入你所欲執行的程式的名字，企圖執行某個程式時，作業系統的母殼或命令殼（shell），就會叫用**載入程式**（loader），將這個程式，由硬碟拷貝一份，存入記憶器內。記得，這個載入程式通常存在作業系統的/sbin 檔案夾內。它與連結程式是兩個完全不同的程式。許多人都將它們搞混了，以為是同一個程式，那是錯的，兩個程式的名稱與功能完全不同。連結程式將一個應用程式有關的所有目的檔案和庫存函數，連結在一起，形成一個可執

行檔案，存在硬碟上。而載入程式則將一個可執行檔案，由硬碟上拷貝一份取出，存入記憶器內，以便其能為中央處理器所執行。

2-8 程式語言的選擇

幾十年來，電腦科學家所發明與設計的電腦程式語言，可能超過一百種。就在我的學習與工作過程，我就學過與用過將近十二種不同處理器的組合語言，以及 BASIC、PASCAL、FORTRAN、COBOL、C、C++、JAVA、C#、Perl，與腳本（shell scripts）等多種高階語言。隨著時間的過去，其中有些已逐漸褪色。

每一程式語言都有其優點與缺點，亦即，長處與短處。就目前而言，C 語言算是所有程式語言中，最精簡、高效率與高性能、最精準、最嚴謹，與功能最強的程式語言。幾乎所有當今最常用與最流行的作業系統（如 Unix 與 Linux），資料庫管理系統（如 Oracle）、網路系統（如 Cisco 的路遊系統程式）、群集（clustered）系統，以及許多其他的伺服系統與應用程式等，都是用 C 語言寫成的。這些軟體，在未來的幾拾年，甚至幾百年，都會一直繼續位處所有軟體的核心，繼續在網際網路的背後，履行幕後核心的處理作業。就功能與重要性而言，目前還未見任何可以取代 C 語言的其他語言出現，特別是在非常重要的系統軟體層次。

在應用程式的層次，有些應用軟體還是有用 C 寫的。但很多公司都有傾向採用 Java 的趨勢，追趕物件導向（object-oriented）的流行。在微軟的視窗系統上，C# 就相當於 Java，兩者有很多地方都有一對一的類似語言結構存在。採用 Java 的好處是，它是跨平台的，只要編譯一次，到處都可執行。對支援多平台，會省事許多。但 Java 有兩大弱點。第一是，太累贅了（cumbersome）。很多事情在 C 語言可以簡短幾行很精簡迅速做到的，在 Java 則是老太婆的裹腳布，又臭又長。很不精準又很沒效率。第二是，物件導向有個天生的缺點，那就是它是一切都是階級性的（hierarchical）。在現實的世界裡，並非所有事情都是階級性的。相反地，絕大多數都不是。因此，物件導向只適合於有效地解決某一些問題。對於解決其他問題，物件導向的階級結構反而是一種限制，綁手綁腳。至今無人以 Java 或 C# 開發前述的系統軟體。

最近幾年，Perl 與 Python 亦有逐漸流行的趨勢。這些語言相形之下，比較鬆散，不是很嚴謹，效率也較低，也較累贅。作者曾目睹一些本來應該選擇用 C 語言，但卻使用 Perl 與 Python 寫的開發計畫，最後都變得問題一大堆，產品品質挺不如人意。因此，記得任何軟體，想要品質可靠、高效率、高性能、精準，與可值性高，最好還是避開這些流行但鬆散、不夠嚴謹的語言。

這些語言通常都必須仰賴很多必須重新寫起的庫存函數，這些庫存函數經常是又臭又長，很沒效率，而且不是百分之百完全可移植的。譬如，Perl 照理說應該是百分之百可移植的，但事實並不然。問題出在它所用的許多庫存函數，在不同作業系統上，其功能就不盡相同。

繼續再往更上層的網際網路應用程式，同樣有多種不同的架構與語言可以選擇。但你會發現，這些不同的語言與環境架構，通常壽命都不是很長。過一陣子，你就會發現，不時會有新的語言與架構出來，出來時都擺明一副要取代過去舊有的一切的架勢。但最終你會發現，它們可能是有解決一些問題，但同時卻也帶來了新的問題。

總之一句話，選擇使用那一種程式語言來開發你的產品，關係至為重要。它事關你整個產品的品質、開發計畫的成敗，以及全部的成本。作者親自看過很多產品選擇了 Perl 與 Python，最終品質都不如理想。原因是，這些語言都太高階、太鬆散了。嚴謹度不及、效率太差，無法勝任真正嚴肅、嚴格的開發計畫。若你期望的是一個高品質、高性能、效率高、精準度高、可靠性高的軟體產品，記得選用一個嚴謹度與效率都高且不要太高階的語言。一般而言，C 程式語言至今為止還是無接近的對手！