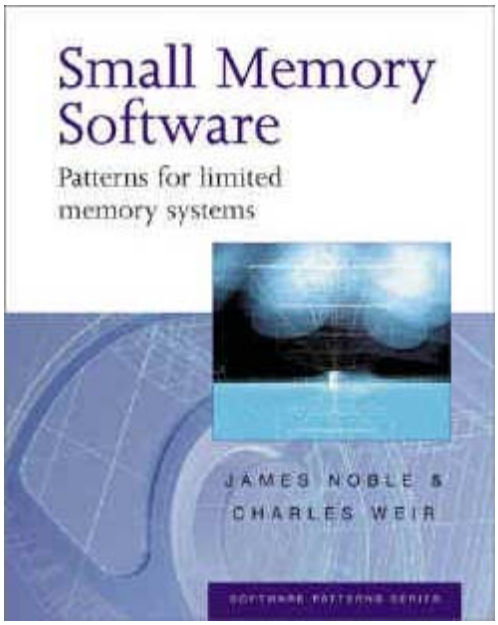

Small Memory Software
Patterns for limited memory systems
JAMES NOBLE &
CHARLES WEIR

Small Architecture（小容量架構，25）

Q：如何在整個系統中管理記憶體？A：讓每個組件（components）負責自己的記憶體運用狀態。

Memory Limit（記憶體限額,32）Q：如何在多個相互競爭的組件之間分配記憶體？A：爲每個組件設置限額，並拒絕超額配置。

Small Interface（小型介面,38）Q：如何減少組件介面帶來的記憶體額外負擔？A：設計出讓客戶得以控制資料傳輸的介面。

Partial Failure（局部毀棄，降格求全，48）Q：如何處理不可預見的記憶體需求？A：保證即使記憶體消耗殆盡，系統也能處於安全狀態。

Captain Oates(犧牲小我,57)　Q：如何滿足對記憶體的重點需求？A：犧牲非絕對必要之組件所使用的記憶體，避免拒絕更重要的任務。

Read-Only Memory（唯讀記憶體,65）Q：如何處置唯讀的程式碼和資料？A：把它們儲存於唯讀記憶體內。

Hooks（掛鉤,72）Q：如何更改唯讀儲存器內的資訊？A：在可寫（writable）儲存器內透過 hooks 存取唯讀資訊，然後更改 hooks 以便造成資訊更改的假象。

Secondary Storage（次儲存裝置，79）

Q：主儲存裝置耗盡了，該怎麼辦？A：把次儲存裝置當作執行期間的額外記憶體。

Application Switch（任務切換,84）Q：系統提供多種功能，如何降低其記憶體需求？A：把你的系統分解成獨立的執行個體，每次只運行一個。

Data Files（純資料檔，92）Q：如果資料太大，主記憶體容納不下怎麼辦？A：一次處理一點資料，把其餘部分放在次儲存裝置內。

Resource Files（純資源檔, 101）Q：如何管理爲數眾多的組態資料（configuration data）？A：在次儲存裝置內保存組態資料，必要時才載入或拋棄其中任何一項。

Packages（封包，108）Q：如何管理有許多可選（optional）成分的大型程式？A：把程式分解成多個封包，只在需要相關封包時才載入它們。

Paging（分頁，119）Q：如何製造出記憶體容量無限的幻象？A：將系統程式碼和資料存放在次儲存裝置內，必要時機載入（load）主記憶體或加以卸載（unload）。

Compression（壓縮, 135）

Q：如何才能把十公升資料放進一公升的記憶體盒罐內？A：以壓縮技術降低記憶體需求。

Table Compression（表格壓縮, 143）Q：如何壓縮大量短字串？A：以不同的位元數（bits）爲每個成員編碼（encode），令較常被使用的成員使用較少量的位元（bits）。

Difference Coding（差分編碼, 153）Q：如何減少資料序列（sequences of data）所佔用的記憶體？A：根據相鄰各項資料之間的差異重新塑造序列（sequences）。

Adaptive Compression（自省式壓縮, 160）Q：如何降低大量大型資料的記憶體需求？A：運用自省式壓縮演算法。

Small Data Structure（小資料結構, 169）

Q：如何減少資料佔用的記憶體？A：慎選「得以支援你所需要的操作」的最小結構。

Packed Data（資料包綑,174）Q：如何縮減儲存資料結構所需的記憶體？A：在結構內部包綑（pack）資料，令它佔用最少空間。

Sharing（共享，182）Q：用什麼方法消除同一資訊的多份副本？A：只儲存一份資訊，並在需要它的任何地方共享之。

Copy-on-Write（臨寫複製，191）Q：如何改變共享物件而不影響其他客戶？A：共享此一物件，並於你必須改變它的前一刻複製它，然後使用該份拷貝。

Embedded Pointers（內嵌式指標，198）Q：如何縮小一整群（collection）物件所佔用的空間？A：將維護群集（collection）所需的指標嵌入每個物件內。

Multiple Representations（多重表述，209）Q：如何支援同一物件的多種不同實作（implemenation）？A：讓所有實作都滿足共同介面。

Memory Allocation（記憶體配置, 219）

Q：如何配置記憶體以儲存你的資料結構？A：從滿足你需求的配置技術中遴選最簡單者。

Fixed Allocation（固定式配置，226）Q：如何確保永遠不會沒有記憶體可用？A：初始化階段預先配置物件（Pre-allocate objects during initialization）。

Variable Allocation（可變式配置，236）Q：如何避免出現未被運用的閒置空間？A：必要時機配置或釋放大小不一的物件（variable-sized objects）。

Memory Discard（記憶體拋棄, 244）Q：如何配置暫時物件？A：在暫態工作區（temporary workspace）中配置物件，工作完成後將它丟棄。

Pooled Allocation（池化配置，251）Q：如何配置大量相似物件？A：預先配置一個物件池（a pool of objects），循環使用其中未被使用的物件。

Compaction（夯實，密合，259）Q：如何回收因碎裂（fragmentation）而丟失的記憶體？A：移動物件，藉以移除物件之間的未用空間。

Reference Counting（參用計數, 268）Q：如何知道什麼時候才能刪除一個共享物件（shared object）？A：爲每個共享物件保持一個參用計數，計數爲 0 時便可刪除之。

Garbage Collection（垃圾回收，278）Q：如何知道什麼時候才能刪除共享物件（shared objects）？A：識別出那些不再被指涉（refer）的物件，然後便可釋放它們。

記憶體受限系統之程式開發

— 針對記憶體受限系統而整理的範式 —

Small Memory Software
Patterns for system with limited memory

James Noble and Charles Weir 著

Duane Bibby 插畫

侯捷 / 王飛 / 羅偉 合譯

To Julia and Katherine.

Who have suffered long and are kind.

商標說明

8086, Intel 432 是 Intel Corporation 的商標；ActiveX™, Internet Explorer, Microsoft C++, Microsoft Studio, MS PowerPoint™, MS Windows™, MS Word™, MS-DOS®; Windows 95®, Windows CE®, Windows NT® 是 Microsoft Corporation 的商標；Adobe PDF®, PostScript™, Photoshop™ 是 Adobe System Inc. 的商標；Apple Macintosh®, Apple-II®, Hypercard®, MacOS, NetwonScript™ 是 Apple Computer Inc. 的商標；CORBA™ 是 Object Management Group 的商標；Emacs 是 Sphinx Ltd 的商標；EPOC16, EPOC, EPOC32 是 Symbian Ltd 的商標；Inferno, Limbo© 是 Lucent Technologies© Inc.的商標；Java, Solaris 是 Sun Microsystems 的商標；Lotus Notes® 是 Lotus Development Corporation 的商標；Modula-3 是 Pine Creek Software 的商標；Netscape 是 Netscape Communications Corporation 的商標；ObjectPlus 是 Computer Associates International, Inc. 的商標；ObjectStore 是 Object Design Inc. 的商標；Palm Pilot™ , Palm Spotless JVM™, PalmOS™ 是 3Com Corporation 或 Palm Computing 的商標；Plan/9® 是 AT&T 和 Bell Laboraties 的商標；Psion 5© 是 Psion Computers 的商標；RISC OS 是 International Business Systems Corporation 的商標；Smalltalk, Smalltalk-80 是 Xerox Corporation 的商標；SparcWorks 是 Sparc International 的商標；UNIX® 是由 X/Open Company Ltd 授權；VisualWorks\Smalltalk 是 ParcPlace Inc. 的商標；VMS 是 Digital Equipment Corporation 的商標；X Window System 是 Massachusetts Institute of Technology(MIT™)的商標。

侯捷譯序

記憶體曾經是兵家必爭之地，曾經被喻爲 CPU 之外最寶貴的電腦硬體資源。在那「640K 天塹」的遠古年代裡，程式員對記憶體緇銖必較的程度，必然令生活於「虛擬記憶體」環境下的當今世代瞠目結舌。當時的人們（我也屬其中之一）即便在 config.sys 中揮汗調校節省區區數十個 bytes，都覺得歡欣鼓舞。

而後，PC 作業系統的技術有了突破性的發展，逐漸地運用 int67h 進入 EMS 記憶體，運用 int21h 進入 XMS 記憶體，最後是全面地、隱藏式地、通透性地提供了虛擬記憶體(virtual memory）。當虛擬記憶體作業系統（如 Windows、OS/2、Linux）走進群眾，苦樂俱往矣。非人道的痛苦折磨被迅速遺忘，緇銖必較的軼趣成了白頭宮女話天寶當年的回憶。我們不再被程式碼大小所限，也不再被資料量所限。所有記憶體不足的問題只要「加一條 256M 記憶體」就獲得解決。程式員「從此過著幸福美滿的日子」。

從極度嚴苛到極度自由，PC 程式員在記憶體用量上得到了完全的解放，開始胡天胡地了起來，不再把記憶體用量看在眼裡，想在心裡。當時的我也總這麼說：『能以小錢解決的問題，都不是問題』。這原本是好的發展，快樂的走向，但是當嵌入式系統逐漸大行其道，記憶體問題再度浮上檯面，而且較之 DOS 的「640K 天塹」遠遠更爲嚴苛。

如此嚴峻的環境下，還是有許多成功的嵌入式系統問世。它們是怎麼辦到的呢？本書收納多個嵌入式系統成功案例，從中萃取整理出記憶體受限環境下的軟體生存之道。本書以 patterns 的嚴謹描述方式，給予每一個方案以標準名稱、別名、問題出處、解決辦法、結果與影響、實作摘要、示例、旁徵參考。

這是一份良好的整理與總結，提供了一個完整涵括，適合給預備或即將在嵌入式系統上衝鋒陷陣的程式員閱讀。讀者必須具備相當的技術基礎，因爲本書告訴你的是「如果你要解決某某問題，請使用某某技術，它會帶來某某成果，並造成某某（正面或負面的）影響」。雖然本書也帶有數量頗豐的示例程式，但它們都是具體而微 — 能夠從這些小程式的蘊涵看出大世界的應用，你得具備我所謂的「相當程度的技術基礎和實務經驗」。

Small Memory Software

本書極端側重 **patterns**。這個詞在臺灣、大陸兩地的譯法，我見過三種：範式、樣式、模式。我個人最喜歡「範式」，足以說明 patterns 的「典範」意味。因此，本書以「範式」稱 patterns。本書所有 patterns 都保留英文名稱，並以特殊字型標示，例如 *Small Architecture, Memory Limit, Small Interfaces...*。本書也提到所謂 **forces**（作用力，意義詳見附錄），起始數章以中英並陳方式顯現，例如 *Memory requirements*（記憶體需求）, *Real-time response*（即時回應）, *Start-up time*（啓動時間）,*Power consumption*（電力消耗）, *Programmer effort*（編程心力）, *Programmer discipline*（編程素養）…，後繼篇幅才以中文獨現。所有 patterns 和 forces，在封面內頁和封底內頁，有完整的列表和簡要說明。

本書具有較爲濃厚的學術味道，讀者定位也比較高階。考量讀者的層次和需求，我時而將一些可能引起疑慮的術語以中英並陳的方式呈現。這樣或可彌補中文術語不統一造成的缺憾，以及做爲中文難以表達英文原意時的小小補償。書中有少部分技術超越我的領略能力，對此我的處理方式是：保留原文（1~2 個小句子）由讀者自行品味。

此外，一般談及記憶體「釋放」（'free' 或 'deallocate' 或 'release'），有時是指客端將記憶體還給 pool，有時是指 pool 管理器將記憶體還給系統（system heap），需要讀者根據上下文體會。一般而言我把 'free' 譯爲「釋放」，'deallocate' 譯爲「歸還」，'release' 譯爲「釋出、釋還、釋回」。但這只是通則，實際譯法視上下文調整。

本書由我和王飛、羅偉兩位先生共同完成。沒有他們的協助，這本書不可能在這個時間點上和各位見面。我要在此表達我的謝意。在合譯態度上，我個人向不喜歡一刀切的二分法或三分法，本書由王飛負責第三章（含）前初稿，羅偉負責第四章（含）後初稿，初稿以外的一切由我總攬。本書有繁簡兩個版本，繁體版採用臺灣術語，簡體版採用大陸術語。兩版勘誤皆可於以下網站獲得。

侯捷 2002.11.30 臺灣新竹
jjhou@jjhou.com
http://www.jjhou.com（繁體） http://jjhou.csdn.net（簡體）

Small Memory Software

王飛譯序

卻納須彌於芥子

計算機技術發展一日千里，如今的家用電腦堪與從前的商用小型機媲美。記憶體容量更是呈跳躍式擴大，於是今日編寫程式似乎再也不必考慮記憶體了，當年先輩爲了節約記憶體而發展出的種種巧思、妙技，似乎已成明日黃花。

但是，嵌入式系統的蓬勃發展卻使人驚嘆「小容量記憶體技術」復活了!

嵌入式設備受體積和電源供電能力等因素影響，記憶體容量很小。如果沿用 PC 上「揮霍無度」的編程方式，嵌入式系統一定會因爲記憶體耗盡而壽終正寢。科技的發展似乎總難以滿足人類的需求和慾望。爲了將須彌山之大的各種功能納入芥子般纖小的記憶體中，塵封已久的小容量記憶體技術重獲人們的青睞。我想，這正是本書的緣起。

本書兩位作者向大家展示的是節約記憶體的 patterns（範式），是從「道」的高度來分析與解決問題。我們不應指望從書中拷貝幾段程式碼，就能解決現實生活中的問題。誠如「獨孤九劍」講的是劍法而非劍招，需要大家不斷地在實踐中思考與揣摩，應勢而發，見招破招。

本書的 patterns（範式）提煉自許多成功系統，兩位作者對這些紛繁複雜的 patterns 進行了梳理和分類，輔以圖解和示例程式，文筆輕鬆詼諧，因此本書既實用又具親和力。對於從事嵌入式系統的朋友以及對小容量記憶體技術感興趣的朋友來說，本書精彩不容錯過。欲納須彌於芥子，請讀這本書！

Small Memory Software

這是我參與翻譯的第一本書，份量很重。我的工作範圍是第三章之前的所有內容。我要對侯捷老師和周筠老師給予我的寬容和關愛表示感謝，謝謝您們物質和精神上的幫助，請原諒我的無知和孟浪。我要對協譯者羅偉先生表示誠摯的謝意，很高興認識您這位遠方的朋友。我同樣要感謝本書的作者 James Noble，感謝您對我所提的問題所做的詳盡解答。我還要感謝我所在部門的趙治本、張志東、姜巍、劉波、張國華等各位領導和同事，感謝大家在生活和工作中給予我的關懷和幫助。我更要感謝我的母親和哥哥，感謝你們給予我的始終不渝的愛。

王飛 2002.12.01 於遼寧瀋陽

羅偉譯序

作爲建築學上第一個具體而直接的體系，古希臘時代產生的「柱式」，歷經兩千多年延伸至今。以致很長一段時期裡，歐洲的建築藝術教育就以研究「柱式」爲主要內容。它之所以能有如此強大的生命力，首先因爲它在實用基礎上已經上昇爲一套非常嚴謹、十分成熟的規範。在大工業化時代中，你也許無法得到像米開朗基羅式的教皇級待遇，身披創造者的榮耀，但與你共事的同僚，或你自己，也（因爲用了標準柱式而）不至於嘗試不入流的技巧，導致業主的圍攻。這倒也算得上另一種幸福。

計算機界常提到的 patterns（範式），與上面所說建築學上的柱式有異曲同工的意味。但只要稍加了解實際使用的 patterns，你或許會發現，這些 patterns 之於編程，與柱式之於建築，層面上著實有好大不同。初學 patterns 的人，也許在 *Abstract Factory* pattern 的行進過程中，就四肢無力搖旗投降了，這大抵是因爲你還缺少實際程式的歷練。畢竟 patterns 不是你拿著 Gang of Four（GOF）那本聖經（《*Design Patterns*》）就可以橫掃江湖的。GOF 只是一本指南！唯有實務鍛煉和經驗累積，才能使你對一個現實系統作出明智的解析。相比之下，柱式的各種比例測算、花式運用，散見於各類建築設計手冊，依葫蘆畫瓢對受過常年艱苦訓練的你來說，並不算一件天大的難事。

最早在網絡上看到這本《*Small Memory Software*》，對它有很高的期待，不僅因爲這本書是與嵌入式系統相關的第一本 patterns 總結資料，更因爲當時我正在思考物件導向思想於嵌入式器件裡的使用價值，很希望得到一些實務啓發。整本書研讀下來的結果並沒有讓我失望（當然，也沒有解決我的所有問題），它給我的思考指出了一些道路：不僅某種具體技術有其使用侷限，物件導向思想也有其實施上的侷限性。這對正處於上昇階段之嵌入式系統與傳統輝煌之物件導向思想的結合，有一定指導意義。畢竟，在總體設計階段把握住整個專案，決定最適合當前要求的解決方案，是絕對優先於 classes 設計與編碼的。

對嵌入式系統來說，系統分析人員有一定特殊性，因爲需求設計基本完成後，要進行軟體與硬體的劃分，而不是逕自奔向軟體總體分析。這是一個需要很高經驗和技巧的步驟。常用的硬體處理器不下 200 種，軟體系統平台也有 Vxwork, PSOS, VRTX, Nucleus, WinCE, QNX, CMX 等各種選擇。長期於 PC 和 workstation 上修煉的程式員一旦接觸嵌入式專案，一定會意識到，原本的世界真是個 big big world ☺。

本書有三大亮點：

一是對各種 patterns 的論述非常清晰，尤其在「環境討論」和「結果」這兩項，讓程式員得以高屋建瓴地審視應用環境；

其次，本書實作和範例使用了多種語言，避免過份依賴語言特性。此外，作者網頁上的 C++ 和 Java 範例程式碼，對理解書中討論內容有莫大作用，強烈建議你親自操練一遍；

最後，每一種 pattern 論述完畢後，通常會提供數種參考資料。本書參考資料達 210 種之多。這對計算機科班畢業或常年從事 PC/workstation 編程的程式員有特別意義，因爲這些資料充份顯示了作者的專業成長歷程。當然，對電子背景的工程師而言，軟硬體結合部份的參考資料稍嫌不足，不過要找到這種類型的書籍應當不難。

得到這本書的翻譯機會純屬偶然。也正是這次偶然之後的互動過程，讓我對侯先生更爲尊敬。感謝華中科技大學出版社的周筠女士一力促成我和侯先生的合作，祝你健康。我還要表達對本書的另一位譯者王飛的敬意，很高興能和千里之外的你成爲封面上的鄰居。特別地，我的朋友孟岩在我的 OO 之路起到重要作用，謝謝你十多年來的友誼。以我父母爲首的家人是我永遠的力量來源，你們的支持使我能振作起因牙痛而昏沉的頭腦，按時完成了譯稿。

恭喜你的書架上多了一本出色的技術書籍，願它帶給你快樂...Mmm...與財富！

羅偉 2002.11.30 於湖北武漢

序言

by John Vlissides

宣稱「個人電腦（Personal Computers, PCs）是唯一擁有龐大市場的電算平台」顯然很愚蠢。Forrester 調查顯示，1999 年 PCs 的銷售量已達巔峰，而 non-PC 電算設備的消費量卻如煙火般地直向上竄，而且成長趨勢看來還會持續。宣稱「記憶體不再是編寫程式時的主要限制」同樣愚蠢。目前流行的觀點是：如果記憶體是問題，多買一點不就得了！但是你要知道，所有桌面應用程式的背後，甚至是最節省記憶體的應用程式背後，隱藏著一頭「軟體基礎設施」（software infrastructure）巨獸，它對記憶體簡直貪得無厭。數百個 megabytes — 不久前用來描述硬碟和磁帶容量的數字 — 現在不過是價格上的入門選擇。稱得上大的記憶體一律按 gigabytes 計算。天哪，這些 RAM 裡面究竟裝了些什麼？

某些浪費有好的理由。透過程式碼重複運用，雖然浪費一點空間，卻可能縮短程式員的開發週期；編譯器實作技術上的折衷妥協，有可能造成二進制碼（binary code）的累贅；爲了讓程式反應速度更快，不得不以空間換取時間。記憶體茲意運用也可能反映出真正的記憶體密集型（memory-intensive）問題。也許，呵，也許以上兼而有之。

然而更常見的是，大量記憶體需求肇因於馬虎的編程、馬虎的設計、馬虎的開發方法和馬虎的組合。這類因素完全可以避免，悲劇在於它們往往不被認爲值得避免，至少短期內如此。市場的現實和「較差即較好」症候群鼓吹大家先把產品推出門，「日後」再修改。讓人沮喪的是，記憶體問題都太容易解決了（多買點不就好了），「日後」遂永不來臨。記憶體消費是典型的「最後處理」問題 — 如果它確實被處理了的話。

但令人驚奇的事情近來不斷發生：嵌入式系統和手持設備蓬勃發展，在 IBM 我們稱其爲「無處不在的電算」（pervasive computing）。受到當前粗糙的電池技術影響，這些系統的計算能力有限；受到尺寸或成本的影響，這些系統的 RAM 容量有限，迫使程式員將記憶體消耗量當作發展條件重新檢視。終於，人們發現，記憶體效能絕不是系統蹣跚之後才需著手解決的問題。它必須一開始就加以設計。最困難之處在於指出怎麼幹。

經過這麼久之後，程式員們不得不重拾記憶體節約技巧，儘管這些技巧自從 von Neumann（馮紐曼）架構就有了。確實，「節約記憶體」是電算界最初 25 年玩的遊戲，直到虛擬記憶體普遍被運用，這一局面才發生改變。儘管如此，唯有配備特別少量記憶體的電算設備，才能適應殘酷的小型化運動（miniaturization）。今昔差別在於，如今這類設備不論在數量或多樣性方面，都迅速超越了其他種類的計算機。

這正是本書耀眼之處。James 和 Charles 針對在這類設備上討生活的程式員所撰寫的這本書，如果算不上救生圈（lifesaver），起碼也是節約時間的法寶（timesaver）。作者以簡潔有力的方式把握了計算機領域中大多數古老而不可或缺，但非顯而易見的記憶體節約技術。這些技術從前也有人描述過，但從未像本書這樣將一系列範式（patterns）梳理成有機體，完全實用，技術新穎，而且立竿見影。不論我如何稱讚本書對軟體發展產生多麼巨大的潛在影響，都不爲過 — 不論你的軟體運行於 PDA, cell phone, pager, 或聞所未聞的嵌入式系統。每一位小容量記憶體程式撰寫者都可以從本書得到收穫。即使你認爲你有足夠的記憶體可供揮霍，這些範式（patterns）也會幫你樹立起一個健康的記憶體運用觀念，讓你的程式不那麼饕餮。

你可以逐頁遍讀本書，但這樣終究只是「讀書」。真正的利益在於遭遇實際問題時，找出相關範式（patterns）並用它來解決問題。因此我建議你在設計和實作程式時研究並實際運用書中的範式。一旦你這麼做，你會發現你必須修改範式使之適應具體問題。優秀的範式將使一切成爲可能，但並不是直接告訴你答案 — 永遠不可能那麼做，因爲沒有兩個問題完全相同 — 而是授你以知識。這就是書中 "forces" 段落彌足珍貴的原因：它們揭示問題解決過程中可能遇到的限制條件，把解域空間切割爲小得多的可能性集合（但這集合可能還是很大）。其中究竟何者最適合你，部分依賴於你願意並能夠做出哪些必要取捨，以及你無意中可能遭遇的限制，其餘就有賴你的創造力了。當然啦，這樣一本書能夠幫助你更具創造力，但創造力無法來自書中；創造力取決於你。

可以這麼說，如果你從事記憶體受限系統上的編程工作，而且你開始運用本書，很快你就會發現它實在是一份天賜的禮物。

John Vlissides
IBM, *T.J. Watson Research*

前言

by James & Charles

從前，計算機記憶體是地球上最昂貴的物品之一。當時大量的人類智慧耗費在超新星爆炸模擬上，全部的投入不過換得一個後來的諾貝爾獎得主和一大堆真空管。可現在許多人的手持電話、數位記事簿或微波爐裡的記憶體，就足以模擬多數星系的毀滅。

但至少兩件事在計算機歷史中恆長不變。軟體設計依舊艱難（Gamma et al. 1995），軟體功能不斷拓展以至填滿所有可用記憶體（Potter 1948）。本書同時涉及這兩個問題：範式（patterns）已被證實是記錄軟體設計知識的有效形式，而本書所論的範式專門用來解決記憶體需求方面的問題。

身爲本書作者，我們的寫作還有其他幾個目的。以範式研究人員和作家的身份，我們希望多學一些範式和範式編寫技巧；以軟體設計者（designers）和架構師（architects）的立場，我們希望研究目前存在的系統並向它們學習。更明確地說：

- 我們希望獲得並分享關於「可移植之小容量記憶體技術」的全面而深入的知識；這些技術在許多不同環境中都能發揮作用。
- 我們希望寫出一組完整的範式集，目標單一而集中，就是解決記憶體需求問題。
- 我們希望研究這些範式之間的關係，並根據這些關係將它們群組化、次序化。
- 我們想要一本親和力高的書，它讓你感覺輕鬆有趣，而不是讓你活受罪。

本書就是我們的成果。無論你在你的工作中是否遭遇記憶體的束縛，它都是像我們這樣的軟體開發者和架構師的好幫手。

爲使本書親和力更強（也爲寫作帶來樂趣），我們以輕鬆的態度完成大部份範式示例，並搭配 Duane Bibby 的漫畫。如果你對這些「輕浮的」☺東西不感興趣，請跳過漫畫和示例描述 — 是的，其他內容我們儘可能保持嚴肅。

本書仍在發展之中。我們已經把許多人的建議和評論納入，歡迎更多指教。您可以在我們的網站上與我們聯繫，網址是 http://www.smallmemory.com/

Small Memory Software

致謝

沒有任何作者可以獨享一本書的功勞。首先要感謝 John Vlissides，永不倦怠的編輯：我們手頭上還保留他對這份瘋狂作品的許多提議。從最初的 EuroPLoP 論文到最終的草稿，他對我們所整理的範式（patterns）作了大量評論，我們深爲感激。其次，如果沒有 Duane Bibby 的插畫，本書風貌將大不相同，希望大家和我們一樣喜愛這些插畫。

我們要爲本書的缺點負責，榮耀則應歸功於花時間閱讀和評論範式草案及本書草稿的各位 Memory Preservation Society 成員和特約研究員。這些人包括（以下可能有所疏漏）：John Vlissides（再次感謝）, Paul Dyson, Linda Rising, Klaus Marquardt, Liping Zhao （他們是 EuroPLoP 和 KoalaPLoP 研討會上的指導員）, Tim Bell, Jim Coplien, Frank Buschmann, Alan Dearle, Martine Devos, Martin Fowler, Nick Grattan, Neil Harrison, Benedict Heal, David Holmes, Ian Horrocks, Nick Healy, Dave Mery, Matt Millar, Alistair Moffat, Eliot Moss, Alan O'allaghan, Will Ramsey, Michael Richmond, Hans Rohnert, Andreas Rüping, Peter Sommerlad, Laurence Vanhelsuwe, Malcolm Weir, 以及設於 University of Illinois 和 Urbana-Champaign 的 Software Architecture Group，成員包括：Federico Balaguer, John Brant, Alan Carrol, Ian Chai, Diego Fernandez, Brian Foote, Alejandra Garrido, John Han, Peter Hatch, Ralph Johnson, Apu Kapadia, Aaron Klish, An Le, Dragos-Anton Manolescu, Brian Marick, Reza Razavi, Don Roberts, Paul Rubel, Les Tyrrell, Roger Whitney, Weerasak Witthawaskul, Joseph W. Yoder, Bosko Zivaljevic。

英國的 Addison-Wesley（或 Pearson Education，我們忘了究竟是哪個）工作團隊在地球的另一端幫了兩位作者大忙。我們感謝 Sally Mortimore （他啓動本案）, Allison Birtwell（他結束本案）, Katherin Ekstrom（他打破距離）。本書榮耀也要歸功於兩位藝術家：George Platts 構思書中插畫，Trevor Coard 親手繪製了多幅插畫。

最後，我們一定要感謝範式社群（patterns community）的全體成員，尤其要感謝曾經參加 Kloster Irsee 召開的 EuroPLoP 會議的諸位成員。我們兩人都是範式「文學」的仰慕者（就像人們可能是科幻小說迷一樣），我們希望這部範式選粹能爲範式文學增輝。

Small Memory Software

目錄

Small Memory Software

0

導讀

Introduction

『小巧即是美』 E. F. SCHUMACHER
『你永遠不可能既富有又苗條』 BARBARA HUTTON

設計「在有限記憶體空間內高效執行」的小型軟體，是一門行將就木的藝術，直到最近這種情況才有所改變。過去由於個人電腦、工作站、大型主機的記憶體容量和處理器速度呈指數規模遞增，程式員幾乎可以完全不必顧慮記憶體限制了。

身處新世紀之交，我們發現了一個伸手可及的龐大市場，數以億計的移動設備需要大量高規格軟體；但實際尺寸和能量（電力）供應限制，使它們的記憶體容量相對有限。與此同時，web 伺服器和資料庫伺服器的開發人員發現，他們的應用程式必須高效運用記憶體，同時支援數十萬個他們賴以生財的客戶。甚至 PC 和工作站上的程式員也發現，視訊需求和多媒體需求向系統記憶體容量提出了超乎合理的挑戰。小容量記憶體軟體（small memory software）復活了！

何謂「小容量記憶體軟體」？記憶體的大小就像財富或美貌一樣，並不是絕對的。某個數量的記憶體到底是大是小，取決於軟體被滿足的程度，也取決於底層軟硬體架構和其他許多東西。大型計算機上的氣象軟體也許和行動電話中的文字處理器或智慧卡（smart card）裡頭的嵌入式應用軟體一樣受到記憶體限制的困擾。所以：

「小容量記憶體軟體」就是「記憶體欲求不滿」的軟體

本書專爲小容量記憶體軟體的程式員、設計者和系統架構者而寫。也許你正著手設計並實作新系統，或維護現行系統，或只是爲了拓展軟體設計知識，都可以成爲本書讀者。

本書描述了我們在成功的小容量記憶體系統中見到的最重要編程技術。我們分析這些技術，把它們整理爲範式（patterns）— 一種「可有效運作之事物」的特別描述型式（Alexander

et al. 1977; 1979）。範式不是發明出來的，而是從現有系統和實踐中辨識或挖掘出來的。爲了提煉本書所談的範式，我們調查了許多在小型機器上成功運作的系統設計。本書萃取出最有可能促成系統成功的一些技術精髓。

本書涵蓋的範式只關注記憶體限制問題：以隨機存取記憶體（RAM）爲主，唯讀記憶體（ROM）和次儲存裝置（例如硬碟或依靠電池供電的 RAM）爲輔。現實系統受到多方限制；圖像、輸出資源、網路頻寬、處理功率、即時回應都可能存在諸多限制，本書只能談及其中一些。儘管我們關注的焦點是記憶體需求，但本書某些範式對於紓解前述限制可能也有助益；另一些則不一定適用，例如處理器功率極差時就不適合運用壓縮技術，分頁（paging）對於即時系統的性能提昇亦毫無幫助。我們已經在每個範式中指出它們如何幫助（或妨礙）各種限制的支援能力。

本章剩餘部分比較詳細地介紹各個範式，分爲以下主題：

- 如何使用本書　　如果你不想逐頁閱讀本書，這裡有一些建議。
- 小容量記憶體簡介　　詳細描述問題，對數個典型的「記憶體受限軟體類型」提出比較。
- 範式簡介　　介紹範式並解釋本書所採用的範式格式（pattern format）。
- 本書涵蓋的範式　　提出一些方法，告訴大家如何尋找（locating）、關聯（relating）、對照（contrasting）本書所有範式。

如何使用本書

毫無疑問，你可以從頭到尾遍讀本書。但更多人喜歡把它當作身兼數職的混合體：

- 爲程式員編寫的「小容量記憶體軟體」入門書籍；
- 你想運用的全部技術的一份快速概覽；
- 遇到難題時的一份查閱參考；
- 一份實作指南，展示常用（或不太常用）的某些範式的使用技巧和陷阱。

後繼內容說明，爲實現以上目的和其他特殊用途，應該如何運用本書。這些特殊用途包括：

- 解決特定的策略問題
- 學術研究
- 讓老闆開心

為程式員編寫的「小容量記憶體軟體」入門書籍

也許你是「記憶體受限」專案中的一名新手，以往從來沒幹過這類專案。

果真如此，你一定希望學到那些會影響你日常編程和設計層面的範式。通常一開始你最主要的設計關注點是 class（類別）設計，那就從直截了當的 *Packed data*（174）上路吧。接下來你可以鑽研記憶體受限系統中被大量運用的 *Small Data Structure*（169），以及 *Sharing*（182）、*Copy-on-Write*（191）和 *Embedded Pointers*（198）。

撰碼時立即面臨的選擇往往是如何配置資料結構，所以接下來你需要比較三種常用的記憶體配置方式：*Fixed Allocation*（226）、*Variable Allocation*（236）和 *Memory Discard*（244）。如何處理「記憶體耗盡」狀態也同樣重要，因此你有必要看看重要的 *Partial Failure*（48），或許也得瞧瞧比較簡單的 *Memory Limit*（32）。

最後，由於多數實際小容量記憶體系統都會借助硬體節省記憶體，因此你需要仔細研究研究 *Read-Only Memory*（65）、*Application Switch*（84）和 *Data Files*（92）。

以上各範式都討論了其他互補範式或替代品，所以閱讀這些範式你可以掌握最重要的技術，瞭解本書其餘部分的概貌。

本書所有技術的快速概覽

「彙集成群」的範式帶給我們的最大好處在於：它創造出「可供討論該主題」的一種共通語言（Gamma et al. 1995; Coplien 1996）。爲了掌握這門語言，你可以快速瀏覽所有範式，閱讀主幹內容，忽略細枝末節、程式碼、實作註解。我們仔細規劃了本書結構，所以上述閱讀方式對你應該輕而易舉。請從 *Small Architecture*（25）這一組開始，逐一讀下去，直到第一個中斷點如下：

❖ ❖ ❖

每個範式的第一部分提供給你你想了解的全部內容：問題、背景脈絡（context）、一個小例子、以及解決方案。沿著這條路線瀏覽一遍所有範式，認真的讀者得花上數個小時，他可以獲得全部範式的概貌和「藉以回想起範式名稱和基本想法」的足夠細節。

解決問題的參考

也許你正十萬火急地開發某個專案，卻遭遇了棘手難題，而你缺少簡單的答案。當此情況，不妨先參考一下封面內頁列舉的範式摘要。如果一或數個範式看起來派得上用場，就翻到相應地點，讀一讀小圓點標示出來的要點，以及「**因此**（*therefore*）」開頭的段落，看看它是否便是你的搜尋目標。

如果上述方法找不到理想的匹配，接下來請善用書後索引，搜尋與你的問題相關的關鍵字，然後再次瀏覽所有範式，看看哪些範式能派上用場。

如果這麼做還是無法讓你找到你的夢中情人，那就瞧瞧封底內頁的範式概要圖 — 其間或許可找出一些有用範式，它們與你已檢視過的範式有某種關連。請檢視那些看起來最有用的範式的「**參見**（*See Also*）」段落；很可能某個相關範式可以解決你的問題。

實作指南

也許你已經確知某個或某些個範式恰好可以爲你所用。你可能早就知道了該項技術，只是從未把它們當作範式來看待，你決定（或者被要求）用它來實作部分系統。

當此情況，你可以查閱每個選定範式的全文。通過封面內頁的摘要，找到該範式，跳至**實作**（*Implementation*）部分，那裡討論了運用該範式時會遭遇到的諸般問題，以及其他實作者成功解決問題時所採用的某些技術。如果你更愛審視原始碼之類細節，不妨先翻到**示例**（*Example*）章節，然後才回到**實作**章節。

你也可以查看**已知應用**（*Known Use*），說不定其中列舉的系統你很熟悉，因而得以輕鬆發現它們實作該範式的方法；**參見**（*See Also*）或許可以指引你找到可能有益的其他範式。

協助制定專案策略（project strategy）

如果你正在制定軟體專案的全盤戰略（Goldberg and Rubin 1995），你對記憶體限制之外的許多問題八成也很關心。你大概還得操心時間效率（time performance）、即時約束（real-time constraints）、急需提交的進度報告或「系統得持續使用數十年」的客戶需求。

遇到這種情況，請翻到附錄內的討論。這些被關心的題目稱作 ***force***s（Alexander 1979）。瀏覽附錄，嘗試選出你感興趣的 *forces*；*force* 的各部分講解會逐一告訴你哪些範式最能滿足你的需要。

學院派學習

無庸置疑，許多人看書還是喜歡從頭讀到尾。我們精心編寫此書，使之仍然能夠適用於這種傳統的、第二個千禧年風格的閱讀方式。

每一章都以比較簡單、易於理解的範式（patterns）起頭，演進至較爲複雜的範式；每一章先出現的範式會導出隨後出現的範式。各章之間採用近似的步進方式，從專案初期最需要的大規格範式（例如 *Small Architecture* (25)）開始，直到與實作息息相關的範式（例如 *Memory Allocation* (219)）。

讓老闆開心

也許你對這本破玩意兒根本不感興趣，但你的上司給你買了這本書，而你想保住你的美好聲譽。假如真是這樣，請打開封面，朝下擱在散熱器上三天。這樣看起來你好像已經讀過它了，唔，不費吹灰之力（Covey 1990）。

小容量記憶體（small memory）簡介

什麼樣的系統算「小」？我們假設本書範式最適用於「記憶體容量介於 50K~10M bytes 之間」的系統，但許多範式也適用於記憶體容量大得多的系統。

以下列舉四種不同的「記憶體欲求不滿」專案類型，它們可以從本書範式中獲益：

1. 移動式電算環境（Mobile Computing）

掌上電腦、傳呼器、行動電話等等移動設備日益重要。移動設備的用戶需要更多複雜而多樣的軟體，這些軟體最終應當堪與桌上電腦軟體媲美。然而與桌上系統相比，可攜式設備的硬體資源（尤其是記憶體）極爲有限。由於它們「到處存在」的天性（Norman 1998），這類設備應該比桌上電腦更強固（robust） — 一個不帶硬碟的電子記事簿不可能在重新啓動時不丟失資料。

開發這類設備，在記憶體限制方面，我們必須比面對 PCs 和工作站上的類似軟體傾注更大心血。事實上本書所有範式都可能與任何已知專案有關。

2. 嵌入式系統（Embedded System）

另一類尺寸纖細的設備是嵌入式系統，例如工序控制系統、醫療系統和智慧卡（smart cards）。嶄新時髦的汽車裡早已安裝了數百個微處理器，有朝一日我們的體內也會來上幾個微處理器，這麼一想，嵌入式系統可就是個舉足輕重的領域了。

嵌入式設備的記憶體有限，強固耐用的要求高，還常常必須滿足苛刻的即時處理底限。如果它們與生命交關或與任務成敗交關，它們還得滿足嚴格的品質控制和審核。

如果系統的記憶體容量遠低於 50K bytes，這樣的系統必須進行嚴格的記憶體最佳化措施。通常必須大刀闊斧地權衡取捨各項功能，使之與可用記憶體相配。特別需要指出的是，物件導向編程思維雖然仍可使用，但由於「heap 記憶體配置」不再可行，導致其價值大大降低。實作「記憶體容量小於 50K bytes」的系統時，*Memory Allocation*（219）和 *Data Structure*（169）大概是最重要的兩個範式。

3. 大餡餅中的小薄片

許多龐然大物，包括大型主機、迷你級電腦或 PC 伺服器，也可能面臨記憶體容量問題。當這些大塊頭支援數百、數千甚至數十萬個平行運算的 sessions 時，其成本效益最高。儘管它們塊頭巨大，擁有龐大的記憶體容量和通訊頻寬，然而它們承擔的巨大工作量使得每一個 session 只能分得數量相對有限的記憶體。

舉個例子，公元 2000 年時多數 Java 虛擬機器至少需要 10M bytes 記憶體方可運行。然而以 Java 爲建制基礎的 web 伺服器，或許得支援一萬個平行 sessions。如果輕率地爲每個 session 複製一個單用戶虛擬機器（single user virtual machine），那麼伺服器必須配備 100Giga bytes 記憶體，這還沒包括運行於各虛擬機器上的應用程式所需要的記憶體。本書所列範式，特別是 *Data Structure*，能夠令這類伺服器「支援巨量用戶」的能力大幅提高。

4. 大機器上的大問題

對於擁有 10M bytes 記憶體以上的單用戶系統而言，記憶體很少是應用軟體的主要關注點，但一般用途的計算機也會因記憶體容量限制而備受煎熬。

舉個例子，許多機構可能已經斥資購買了某些「記憶體容量固定且無法擴充」的硬體。如果你是一家銀行，有兩萬台用了三年的 PCs 擺在出納櫃台，你想軟體升級時會發生什麼事？如果你去年新買了一台 1Giga byte 伺服器，今年得處理 2Giga bytes 資料，卻又買不起今年新機種，會怎麼樣？即使你支付得起計算機升級費用，其他需求（比如員工獎金）或許需要優先考量。

換一種情形，你可能正致力於開發一個應用軟體，它必須處理數量驚人的資料，例如多媒體編輯、視訊處理、範式鑑別、天氣預報、或維護一組展現大千世界纖毫畢現的 bitmap 圖像。即使在大型系統，上述任何應用軟體都能輕而易舉地耗盡 RAM，所以你應當小心翼翼地設計，限制記憶體用量。對這些應用軟體而言，*Second Storage*（79）和 *Compression*（135）尤其重要。

追根究底，計算機記憶體總是不敷使用。用戶總會同時執行更多工作，處理更大量資料，或者爲追求相對低廉的價格而選擇記憶體容量較小的機器。從「小」的角度看去，唔，所有計算機的記憶體容量都很小。

記憶體侷限類型（constraint type）

想像你正在新環境中開始一個新專案。你如何確定哪些記憶體侷限有可能成爲攔路虎，哪些類型的侷限會引發最大麻煩？

硬體侷限（hardware constraints）

視系統的不同，你可能會遭遇以下一種或多種記憶體限制類型：

- RAM 記憶體　供執行中的程式碼、執行期 stack、短暫性資料和永續性資料使用。
- ROM 記憶體　供執行中的程式碼和唯讀資料使用。
- 次儲存裝置　用來保管程式碼本身、唯讀資料和永續性資料。

你還可能遇到更多特殊限制：例如 stack 的大小可能受限，或者你同時擁有 dynamic RAM（速度快但需借助電力來保持資料）和 static RAM（速度慢但只需要非常低的電力）。

通常 RAM 是最昂貴的一種記憶體，所以很多系統將程式碼存放於次儲存裝置，必要時才載至 RAM；系統也很可能在不同的用戶或不同的應用軟體之間共享已被載入的程式碼。

軟體侷限（software constraints）

大多數軟體開發環境並不以主記憶體、ROM、次儲存裝置來表現它們對記憶體的運用。作爲一個設計者，你經常會發現必須親自處理 heap、stack 和 file 大小。表 I.1 顯示軟體的典型屬性，以及每一種屬性如何映射至上述討論的各類實際記憶體。

表 I.1　軟體屬性與實際記憶體的映射關係

屬性	棲身處所
永續性資料（persistent data）	次儲存裝置或 RAM
heap 和 static data	RAM
程式碼本身（Code storage）	次儲存裝置或 ROM
執行中的程式碼（Executing code）	RAM 或 ROM
stack	RAM

不同類型的記憶體受限（memory constrained）系統

不同類型的系統擁有不同的資源和不同的限制。表 I.2 描述四種典型系統：嵌入式系統、行動電話或個人數位助理（PDA）、PCs 或工作站、大型主機伺服器。這張表只是一般指南，幾乎沒有實際的系統與之精準吻合。例如供網路卡使用的嵌入式系統理所當然有

網路支援；許多大型主機可能提供 GUI 終端畫面；遊戲控制台也許介於嵌入式系統和 PDA 之間。

表 I.2　不同類型系統之比較

	嵌入式系統	行動電話 PDA	PC 工作站	大型主機或 伺服器組
典型應用	設備控制、 協定轉換…等等	日記、電話地址簿、e-mail	文書處理、試算表、小型資料庫、會計	電子商務、大型資料庫應用、會計、庫存控制
UI	無	GUI，由 ROM 內的程式庫實現	GUI，由硬碟上的 DLLs 實現	由客戶、瀏覽器或終端機上的程式庫實現
網路服務	無、序列（serial）連接埠、工業用的 LAN	TCP/IP，構築於無線連接之上。	10 MBps LAN	100 MBps LAN
其他 IO	依需求而定 — 通常根據設備的主要用途	序列（serial）埠	序列（serial）與並行（parallel）埠、數據機（modem）	任何型式，透過 LAN 存取。

一般來說，這些環境都將短暫性程式和堆疊資料存放於 RAM，其他運用方式則大相逕庭。表 I.3 展示這些類型的系統對於每一種軟體記憶體的典型實作方式。

表 I.3　各類系統的記憶體使用

	嵌入式系統	行動電話 PDA	PC 工作站	大型主機或 伺服器組
廠商供應的程式碼	ROM	ROM	Disk, 載入到 RAM	Disk，載入到 RAM
第三方程式碼	無	從快閃（flash）記憶體載入 RAM	由廠商供應之程式碼決定	由廠商供應之程式碼決定
共享程式碼 Shared code	無	在多個應用程式之間共享 DLLs	在多個應用程式之間共享DLLs	在多個使用者之間共享 DLLs 和應用程式碼
永續性資料 Persistent data	無， 或 RAM	RAM 或 快閃（flash）記憶體	本機硬碟或 網路伺服器	次儲存裝置 （如磁碟）

請注意，行動電話和 PDAs 對待第三方程式碼的方式與對待廠商供應之程式碼的方式不同，因爲第三方程式碼不能內置於 ROM。

記憶體限制（memory constraints）的相對重要性

針對上述各類系統的典型用途，表 I.4 說明了各類記憶體限制所代表的不同重要性。三顆鑽石表示該限制通常是該類典型專案架構的主要推動因素；兩顆鑽石表示該限制屬於重點設計考慮對象。一顆鑽石表示該限制可能需要程式員付出一些努力，但或許不會嚴重影響架構；空欄則表示該限制與發展毫無干係。

表 I.4　記憶體限制（memory constraints）的重要性

	嵌入式系統	無線 PDA	PC 或工作站	大型主機或伺服器組
Code storage 程式碼本身	❖❖	❖❖		
Code working set 執行中的程式碼		❖❖	❖	
Heap 和 stack	❖❖❖	❖❖	❖	❖
Persistent data 永續性資料	❖❖❖	❖		

真實世界中的每個開發專案皆不相同；所以也會有一些完全沒有「heap 和 stack 記憶體限制」的智慧卡（smart card），以及一些「主要限制落於永續性資料儲存裝置」的大型主機應用。

範式（patterns）簡介

範式（patterns）究竟是什麼？簡明扼要地說，範式是「特定環境中的問題解答」（Alexander et al. 1977; Coplien 1996）。「環境、上下脈絡」（context）的強調極爲重要，因爲面對一大堆範式，人們很難甄選出最佳者。例如本書所有範式都用來解決同一個問題：記憶體匱乏問題，但解決之道各有不同。

範式並非特定問題的特定答案。「沒有兩個編程問題完全相像」正是「編程維艱」的原因之一，所以「解決某個特定問題」的技術，總體而言用途不大（Jackson 1995）。範式是一大類（a general class）問題的答案的泛化描述 — 就像演算法是計算方法的泛化描述而程式是演算法的特定實作品一樣。由於範式是通用性的描述，比演算法所處的層面更高，因此它們的實作並非一成不變。你不能指望從本書裡頭剪裁一些用以描述範式的示例程式，粘貼到你的程式裡，就妄想萬事大吉。相反地，你應當嘗試領略範式的總體理念，並在你所面對的特定問題的環境中應用它。

怎樣才能信賴某個範式？有用的範式一定曾經在已知系統中有效運作。爲了厲行這一點，我們保證所有範式必須遵循所謂「事必過三」原則：它必須在至少三個實用系統中有所表現。範式被使用的次數多多益善，這意味它是個已被驗證過的優秀解決方案。優秀的範式不是發明出來的，而是從既有系統和實踐中辨識或挖掘出來的。爲了歸納本書範式，我們調查了小型設備上的眾多成功案例。本書提煉出「最有可能促成系統成功」的技術精髓。

Forces（作用力）

優秀的範式應當有嚴密的邏輯。它應當舉出有力的論證，借由「從問題到解答的邏輯關係闡述」，證明該方案確實解決了它要解決的問題。爲了做到這一點，優秀的範式必須逐一列舉環境中重要的 ***forces***，並列舉該解決方案的正面效果和負面後果。

> 譯註：軟體開發就像下棋，在發展過程中要化解多方牽制力。因此我把 force 譯爲「作用力」。感謝 kylin 提供的建議。

所謂 ***force*** 就是「解決問題時應當詳加考量的任何方面」（*any aspect of the problem that should be considered when solving it*, Buschmann et al. 1996），例如必須滿足的要求、必須克服的限制、或應當具備的令人期待的性質。

本書範式最重要的 *force*s 便是：記憶體需求（*memory requirements*，亦即系統佔用的記憶體總量）、記憶體可預測性（*memory predictability*，亦即記憶體需求量能否事先決定）。這些範式也涉及即時效率（real time performance）和可用性（usability）等其他類型的 *force*s。優秀的範式既講優點（它解決了什麼 forces），也談缺點（它曝露出哪些 forces）。如果你使用某個範式，也許不得不借助另一個範式來彌補缺欠（Meszaros and Doble 1998）。

本書附錄討論了本書範式所解決的主要 forces，描述了解決或曝露每一種 force 的主要範式。封底內頁另有一份總結表。

範式群（Collections of patterns）

某些範式可以獨立存在，描述你所要做的全部工作。但也有許多範式是所謂的複合範式（Vlissides 1998; Riehle 1997）：以其他範式表達部分解決方案。這種情況下一個範式只能徹底解決部分問題，更多範式則能解決更多問題，並暴露出原本未曾被考慮的一些 forces。另一方面，後繼範式（往往規模較小）能夠處理先前範式遺留的問題和暴露出來的 forces。

Alexander 將他的範式組織成一門範式語言（*pattern language*），一個從最高階到最低階的範式序列（*a sequence of patterns*），每個範式都將讀者明白引向後繼範式。循著語言所呈現的範式路線，建築師便能夠完成一間房屋、一棟大廈或一座城市的完整設計（Alexander 1977）。

本書範式並非該種意義上的範式語言，我們根本沒打算描述或建立完整系統所需的全部編程技術！但是，只要可能，我們就會說明範式如何相互聯繫，某個範式的使用將如何引導你考慮其他範式的使用。最重要的聯繫詳述於封底內頁圖表。

範式（patterns）簡史

範式並非濫觴自程式設計。建築師 Christopher Alexander 發展了範式（patterns）形式，將其作爲成功建築實踐知識的記錄工具（這是建築學領域的傳說）（Alexander et al. 1977; Alexander 1979）。Kent Beck 和 Ward Cunningham 將範式加以改造，使之適用於軟體設計，並針對 Textronix 用戶介面的設計撰寫出數個範式。Hillside Group 的軟體工程師發展了許多提高範式撰寫水平的技術，開創了 PLoP 系列會議的先河。

Gamma 等人於 1995 年出版《*Design Patterns*》一書，發展出物件導向框架（OO frameworks）方面的範式。隨後湧現出一大批高水準範式書籍，尤其值得一提的是 *Patterns of Software Architecture*（Buschmann et al. 1996）, *Analysis Patterns*（Fowler 1997）, Addison-Wesley 的 *Pattern Languages of Program Design* 系列叢書，以及更多專家級圖書，像 *Smalltalk Companion to Design Patterns*（Alpert et al. 1998）等等。現在你可以從 *Patterns Almanac* （Rising 2000）書中找到軟體開發所有方面的範式。

我們如何撰寫本書

爲了構築出特定的範式群，我們收集了爲數眾多的記憶體節約技術，有軟體中見到的、別人告訴我們的、網路上看到的、聽說的。然後我們剔除那些貌似相關實則無用的技術（例如 bank switching 或電源管理），並將超出本書範圍的技術拒於門外（例如 UI 設計、專案管理），然後我們獲得了數百個不同的觀點、應用和示例。

我們把上述成果集結起來，大量搜尋提供底層（underlying）技術的主題或知識。每項技術構成一個範式的基礎。我們詳細記錄這些技術，形成草案，在 1998 年的一次範式編寫研討會上提交了這份草案（Noble and Weir 1998, 2000）。我們不斷充實草案，收到不少評論、批評和建議，爲了提供更完備的描述，我們「重構」（refactored, Fowler 1999）所有範式，擴充其範圍，減少或改變來自其他範式的干涉。我們剖析了每一個範式所處理的主要 forces 以及範式間的關聯，以便找到把範式綴成連貫群組的好辦法。整個成果就是你此刻正捧著的這本書。

和所有現實背後的故事一樣，實際上所發生的事情並非一步接著一步地完全循序漸進。真實情況更爲機動，更富創造性。整個工作橫跨數年，期間許多動作平行發生。但總體而言「循序漸進」還算是準確稱職的說法。

我們的範式格式（pattern format）

本書所有範式皆使用統一格式。請看圖 I.1，那是 *data structure* 範式的簡略版本。完整版本描述於 p.174。

Packed Data

另名：Bit Packing

如何縮減儲存資料結構所需的記憶體？

- 你有一個資料結構（一大群物件），它們需要大量記憶體。
- 你需要快速地隨機存取結構中任一物件的任一部分。

無論你的系統幹些什麼，遲早你得設計低層（low-level）資料結構，用以保存程式所需的資訊...

例如 Strap-It-On's Insanity-Phone 這個應用軟體，它可以儲存當地電話簿（200,000 個用戶）內的全部姓名和號碼。

由於這些物件（或資料結構）是你的程式的核心，所以它們必須在程式執行時唾手可得...

因此：包綑（Pack）結構內的資料項，使它們佔用最小空間。

有兩種方法可以減少物件佔用的記憶體數量...

一一考量各個欄位，考慮其中有多少資訊是真正必須儲存的，然後選擇語言層次中「得以包容該資訊」的最小資料型別，俾使編譯器或組譯器（assembler）能以最少量的記憶體空間來對它進行編碼...

再次考慮前述的 Insanity-Phone 例子，設計者意識到當地電話簿無須涵蓋「數值大於 32」的區碼，所以每一筆資料只需 5 bits 即可存放區碼...

結果

每個實體（instance）佔用更少記憶體，系統總體需求量因而降低。即便如此，同樣數量的資料還是可以隨機儲存、更新、存取...

然而：系統的時間性能打了折扣，因爲 CPUs 處理「未對齊邊界」（unaligned）資料時，速度比較緩慢...

❖ ❖ ❖

圖 I.1 範式摘錄

這個例子顯示出範式描述形式的各個區段，其中包括：

- 範式名稱（Pattern name）：每個範式都有一個獨一無二、容易記憶的名稱。
- 卡通漫畫（Cartoon）：輕鬆的卡通漫畫提供此一解決方案的一個視覺表達。
- 另名（Also known as）：範式或範式變體的其他常見名稱。

- 問題敘述（Problem statement）：一句話概括範式所要解決的主要問題。
- 環境概述（Context summary）：小圓點標示出問題所涉的主要 forces，對於「此範式是否適合我的特定問題」給個驚鴻一瞥的答案。
- 環境討論（Context discussion）：針對小圓點標記中的描述，展開較長的討論：這個範式何時可派上用場，什麼問題使它特別有趣或特別困難？這一部分還引入一個簡單的例題 — 往往天馬行空甚至荒誕不經 — 作爲解說。
- 解決方案（Solution）：以一句話概述答案。
- 方案描述（Solution description）：後續段落詳細說明解決方案，偶爾附有圖解或圖表。這部分也展示出此一解法如何解決前述例題。
- 結果（Consequences）：這部分指出範式的典型後果，不諱美，不掩惡。爲使優缺點涇渭分明，先講正面優點，再談負面缺欠，二者以「**然而：**」分界。其中**最重要的 forces 將以斜體字表現**，它們在附錄有一份交叉參考（cross-reference）。
- 分隔符號（Separator）：三個 ❖ 符號表示範式描述到此爲止。

本書從頭至尾以這樣的字體表示範式：*Packet Data*。

對於每一種範式的基礎知識，以上就是你所需要的全部閱讀內容。爲了幫助你更透徹理解範式 — 例如實際應用它 — 每種範式還提供更詳盡的資訊，分爲以下區段：

- 實作（Implementation）：以一組「實作注解」討論實際系統中該範式的實作細節。一般來說這部分最長，往往占用數頁篇幅。

  實作注解中的指令並非金科玉律；你或許可以在特定環境中找到更技高一籌的替代方法。這份注解記錄了寶貴經驗，動手實作之前你應該仔細閱讀它。

- 示例（Example）：這部分從特定的、通常相當簡單的實作品中抽取出程式實例，詳細討論其中做了什麼以及爲什麼那麼做。建議將這部分和實作部分結合起來閱讀。

  我們的網站（www.smallmemory.com）提供了大部份示例的完整原始碼。

- 已知應用（Known uses）：這部分描述運用此範式的現行成功系統，證實我們的範式「既有用又高效」。如果你希望尋找進一步資訊，它也給了一些建議。
- 參見（See also）：這部分將讀者引導至本書或其他地方的相關範式，也可能引導你查閱其他書籍和網頁 — 其中擁有更多相關資訊，或對你的實作有所幫助。

「主技術」和「範式」之間的關係

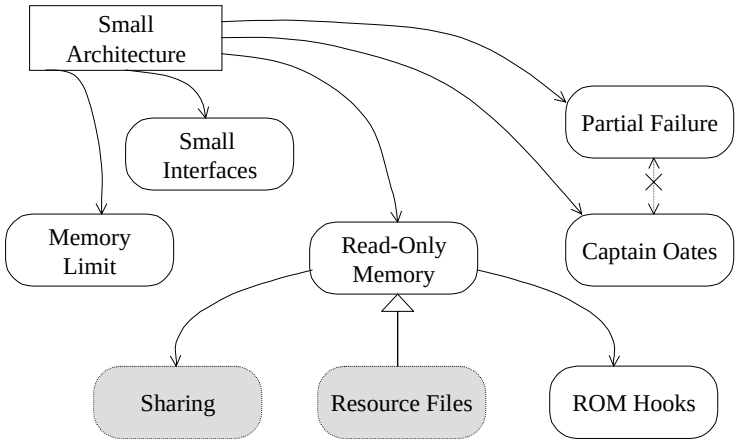
本書範式分成五章，每章展示一項小容量記憶體軟體的主要設計技術：*Small Architecture*（小容量架構），*Secondary Storage*（次儲存裝置），*Compression*（壓縮），*Small Data Structures*（小型資料結構），*Memory Allocation*（記憶體配置）。

儘管每一項主技術比其涵蓋的範式更抽象，但它們本身也是一個範式。是的，我們可以將主技術轉化爲一般範式的描述型式。

每一項主技術都有一個「特化範式（Specialized patterns）」描述區段，概述該章所涵蓋的所有範式，並以此取代原本在範式描述時應有的「示例」區段（如圖 I.2）。

特化範式（Specialized patterns）

以下章節描述六個特化範式，它們描述了「可減少 RAM 記憶體用量」的架構決策（architectural decisions）。下圖顯示它們之間的相互關聯...



本章範式包括：

Small Interfaces（38） 設計「組件之間用以明確管理記憶體」的介面（interface），降低其實作品（implementation）的記憶體需求量...

圖 I.2 「主技術」的描述摘錄

這張圖表展現當時那一章的各個範式之間的關係。圖中矩形代表主技術；白色圓框矩形代表本章範式；灰色圓框矩形代表他章範式。範式間的關係如下所示（Noble 1998）：

	使用（*Uses*）	如果你使用左側範式，你也應當考慮使用右側範式。右側規模較小（smaller-scale）範式解決了左側規模較大範式所暴露的 forces。例如，如果你使用 *Read-Only Memory* 範式，你就應該考慮也使用 *Hooks* 範式。
	特化（*Specializes*）	如果你使用左側範式，特定情勢下你可能會想使用右側範式。右側範式是左側範式的特化版本（a more specialized version），以更特殊的方法解決差不多相同的 forces。例如 *Resource Files* 是一種特殊的 *Read-Only Memory*。
	衝突（*Conflicts*）	兩個範式提供了同一問題的零和解決方案。它們以互不相容的方法解決相同（或類似）的問題。例如，*Partial Failure* 和 *Captain Oates* 都描述了記憶體耗盡時的應對方法，前者降低服務質量，後者終止其他組件。

圖 I.3　範式間的關係

現役實例（running example）

傲視同儕的 Strap-It-On 腕型 PC（圖 I.4）由大名鼎鼎的 StrapItOn 公司出品，是個面臨嚴峻記憶體挑戰的系統。藉由提取該系統內的實例，我們闡釋了許多範式。這個產品內含著名的 Word-O-Matic 文字處理器，配備革新的摩斯電報電碼（Morse Code）鍵盤和語音用戶介面（Voice User Interface，VUI）。

圖 I.4　Strap-It-On 腕型 PC

如果你一派天真地想要實作出任何一個我們提及的應用並因而發財，唔，祝你好運！

本書涵蓋的範式（patterns）

我們爲本書範式選擇了一個特定次序（請見封面內頁表格）。這個次序使這些範式比較容易學習，以由上而下（top-down）的方式運作，使第一個範式能夠爲後繼各章的範式鋪路。

還有許多有效的方法可以安排或討論這些範式的排列方式。本節從數個不同的角度來檢視這個題目：以重要技術來排列，以技術所處理的 forces 來排列、以不同的記憶體節約方法來排列，或是以個案研究來排列。可能你也會想參考附錄中的討論，它根據小容量記憶體軟體所涉及的類型，推薦適用的範式。

主技術

本書將範式組織爲五種主技術（成爲本書五章），概述於表 I.5 中。

表 I.5 五種主技術

Small Architecture	需系統內多個組件(s) 協同運作（co-operation）才能完成的一種記憶體節約技術。
Secondary Storage	以硬碟或等價物作爲 RAM 的輔弼。
Compression	這是以處理器運算爲基礎（Processing-based）的一種技術，藉由「自動壓縮」來減少資料量。
Small Data Structure	定義一些資料結構和演算法，以達到儘可能減少記憶體用量的目的。
Memory Allocation	從混沌無序的可用記憶體中產生某種資料結構，並在不再需要它時歸還之。

範式所處理的 forces

另一種審視範式的方式是比較各個範式所處理的 forces。

本書附錄詳細討論了每個 forces，以及用來處理它們的範式。下面兩張表格是該份附錄的非全面性摘要。表 I.6 總結出我們認爲最重要的十種 forces，表 I.7 展示各個範式如何處理表 I.6 的每一種 force。

表 I.6　10 個重要的 forces

memory requirements 記憶體需求	範式確實減少了系統執行時所需記憶體的絕對數量嗎？
memory predictability 記憶體可預測性	範式確實使系統記憶體需求量更容易被事先預測嗎？
time performance 時間效率	範式有助於提升系統的執行速度嗎？
real-time response 即時反應	範式是否能夠藉由程式執行效率的可預測性，降低程式對事件（events）反應的潛在因素？
start-up time 啓動時間	範式是否減少了「系統收到程式啓動請求後至程式開始執行」的時間？
local vs. global 區域 vs. 整體	範式有助於封裝應用程式的不同部分，使其獨立性更強嗎？
secondary storage 次儲存裝置	範式有助於將記憶體的使用從昂貴的 RAM 轉移至廉價的次儲存裝置嗎？
maintainability 易維護性	範式鼓勵更高設計品質嗎？此後系統的修改會變得更容易嗎？
programmer effort 程式員心力	範式有助於降低已知系統的創建過程中程式員所付出的總體努力嗎？
testing cost 測試成本	範式有助於降低應用程式發展過程中的總體測試負擔嗎？

表 I.7 展示每一種範式如何處理這些 forces。針對某個 force，如果某個範式大體而言有利於你解決該 force，我們就以 ☺ 表示；如果某個範式大體而言是個缺點（遺棄了該 force），我們就以 ☹ 表示。本書附錄更詳盡地探討了這些 forces。

減量（Reduce），復用（Reuse），循環（Recycle）

環保人士定出以下三個策略，可以降低人類文明對自然環境的衝擊：

- 減少（*reduce*）人工製品的消耗量和廢品數量。
- 將產品重新使用（*reuse*）於其他用途。
- 循環使用（*recycle*）產品中的原始材料，用以製造其他產品。

三者之中，減量（*reduction*）最具效率；如果你降低了廢品數量，也就不必再操心如何處理它。循環使用（*recycling*）的效率最低；需付出大量的能源和努力才能夠從廢品中生產新產品。本書描述的範式也可以以類似方法分組如下：

Table I.7　不同的範式對各種 forces 的處理

範式（patterns）	Time performance	Real-time	Start-up time	Local vs. global	Predictability	Quality & maintainability	Programmer effort	Testing cost	Usability
Architecture				😐	☺	☺	😐		☺
Memory Limit				☺	☺			😐	
Small Interfaces	☹	☺		☺	☺	☺	😐	😐	
Partial Failure				☹	☺	☺	☹	☹	☺
Captain Oates	☹			☺	😐		☹	☹	☺
Real-only Memory		☺	☺	😐		☹	☹	☺	
Hooks	☹					☺	☺	☹	
Secondary Storage		☹		☹			☹		☹
Application Switching	☹	☺	☺	😐	☺	☺	😐	☺	☹
Data Files	☹		☹	😐	☺		☹	☺	☹
Resource Files	☹		☹			☺	😐		
Packages		☹	☺			☺	☹	☹	😐
Paging	☹	☹		☺		☺	☺	☺	☺
Compression	☹	😐			☹	☹	☹	☹	
Table Compression	😐							☹	
Difference Coding	😐	☺			☺			☹	
Adaptive Compression	☹	☹					☹	☹	
Data Structures		😐		☺	😐	☺	☹	😐	☺
Packed Data	☹			☺		☹	☹		
Sharing	☺		☺	☹		😐	☹	☹	
Copy-on-Write	😐	☹	☺		☹		☹	☹	
Embedded Pointers	☺	☺		☹	☺	☹	☹		
Multiple Representations	☺			☺	☹	☺	😐	☹	
Allocation	😐	😐	😐	😐	☺				
Fixed Allocation	☺	☺	☹		☺		😐	☺	☹
Variable Allocation	☹	☹	☺	☹	☹	☺	☺	☹	
Memory Discard	☺	☺	☺	☺	☺		☺	☹	
Pooled Allocation	☺	☺	☹		☺			☹	
Compaction	☹	☹					☹	☹	
Reference Counting	☹	☺		☺		☺			
Garbage Collection	☺	☹		☺	☹	☺	☺		

- *Reduce*：降低程式的記憶體需求。像 *Packed Data*（174）、*Sharing*（182）和 *Compression*（135）之類的範式都是透過資料減量、去除冗餘的手法來降低記憶體絕對需求量。*Second Storages*（79）和 *Read-Only Memory*（65）則是借助其他儲存設備來降低 RAM 的需求。
- *Reuse*：面對不同的用途，重複運用記憶體。通常 *Fixed Allocation*（226）或 *Pooled Allocation*（251）所使用的記憶體可能會被（重新）用來儲存「型別大致相同」的物件，一個接一個。*Hooks*（72）允許軟體重複使用既有的唯讀碼而不必替換之。
- *Recycle*：在不同的時間，針對不同的用途，循環使用記憶體。*Variable Allocation*（236）,*Reference Counting*（268）, *Compaction*（259）, *Garbage Collection*（278）和 *Captain Oates*（57）都有助於程式在不同時間以截然不同的方式使用同一塊記憶體。

個案研究（Case studies）

這一節帶你看看三個簡單的個案研究，每個個案描述你可能用來成功完成實作品的一些範式。

1. 手持應用（Hand-held application）

想像你正在爲 Windows CE, PalmOs 或 EPOC smart-phone 之類的手持設備開發應用程式。

這個應用程式得有 GUI，以及一般 PC 應用程式具備的大部分基本功能。然而其記憶體配備遠遜 PC，可承受之最大容量可能只是 2Mb RAM，700 Kb 程式碼（非程式庫），以及應客戶需求而準備的數個 Mb 左右的永續性資料。

設備裡頭每一個應用程式的可用記憶體都很有限，你的程式當然也不例外。*Small Interfaces*（38）無處不在，至關重要。應用程式依據作業系統式樣指南（the operating system style guide）使用 *Resource Files*（101）。爲了將程式碼的體積和測試降至最低，你會希望運用廠商應許之程式庫所提供的 *Hooks*（72），將它置於 *Read-Only Memory*（65）內。由於作業系統支援多行程（processes），而你的行程不可能時刻處於運行狀態，因此你可以將永續性資料儲存於 *Data Files*（92）內。

具體操作環境可能需要其他類型的架構範式（architecture patters）：PalmOs 要求 *Application Switch*（84）；CE 要求 *Captain Oates*（57）；EPOC 想要 *Partial Failure*（48）。如果你不是爲系統廠商工作，你的應用程式就會是個「從次儲存裝置載入」的「協力廠商（第三方）應用程式」，於是你可能使用 *Packages*（108）來降低記憶體需求量。

你的應用程式裡的多數物件可以使用 *Variable Allocation*（236）或 *Memory Discard*（244）。某些「即時性極端重要」的組件 — 例如通訊驅動程式 — 可以使用 *Fixed Allocation*（226）和 *Embedded Pointers*（198）。

如果某個 class 將被生成許多實體（instances），可考慮使用 *Packed Data*（174），*Multiple Representations*（209），*Sharing*（182）或 *Copy-On-Write*（191）以減少其記憶體總量。如果你的物件被多個組件共享，你可以考慮使用 *Reference Counting*（268）。

2. 智慧卡（Smart card）

換個環境，設想你正致力開發供智慧卡（smart card）應用軟體 — 例如能夠置於 PC 裡頭的 PCMCIA 數據機。程式碼存放在 ROM 裡頭（實際上是把 flash RAM 當作 ROM 來用），一共大約 2Mb。RAM 不過區區 500K。唯一的用戶介面是 Hayes "AT" 指令集，透過序列連結（serial link）和 PC 相連；數據機也和電話線相連。

系統程式碼連同數據機協定所需的靜態表格都儲存於 *Read-Only Memory*（65）內。你只需要一個控制緒（control thread）和一個大約 50K 的 stack。

數據機的即時性能至關重要，所以大部份「長時間存在」的物件應該使用 *Fixed Allocation*（226）。短暫性資料可使用 *Memory Discard*（244）儲存於 stack 內。這個系統需要許多 I/O 緩衝區，可採用 *Pooled Allocation*（251）來完成；如果眾多組件共享一個緩衝區，你也可以採用 *Reference Counting*（268）。

數據機的主要工作大多是對電話線上的傳輸資料進行 *Compression*（135）。簡單的數據機通訊協定可能採用 *Difference Coding*（153）；複雜的協定則採用 *Table Compression*（143）和 *Adaptive Compression*（160）。爲了實作出這些複雜協定，你需要將大型、複雜的資料結構內建於 RAM，而爲了將記憶體用量降至最低並提高效率，可採用 *Packed Data* 和 *Embedded Pointers*（198）實作上述資料結構。

3. 大型 web 伺服器

你也可能正埋首開發一個 Java web 伺服器，它提供一個電子商務 web 和 WAP 介面，允許用戶購買產品或線上服務。此伺服器通過區域網路（LAN）和公司內部用來管理定價及庫存的計算機相連，也和儲存全體用戶詳細資料的資料庫相連。

RAM 記憶體和系統開發成本比較之下相對便宜，但伺服器能夠支援的實際記憶體數量畢竟有限。伺服器作業系統通常提供 *Paging*（119）以提高（外觀上）應用程式可用的記憶體數量，但由於大多數交易只佔相對短暫的時間，所以你不希望任何時刻總有一大堆記憶體置換動作（paged out）。伺服器同時支援數千個並行用戶，所

以你根本承擔不起「爲每個用戶配置數十個 megabytes」的沉重負荷。

你可以採用 *Sharing*（182）俾使所有用戶共享一個或數個 Java 虛擬機器實體（virtual machine instances）。只要資料是 *Read-Only*（65），就可以被共享。如果用戶進行的交易包含極複雜的資料結構，你可以對每一個用戶施行 *Memory Limit*（32）。編程的可維護性和簡易度很重要，所以幾乎所有物件都應該採用 *Variable Allocation*（236）和 *Garbage Collection*（278）。

網際連接（internet connections）滲透至每個終端用戶，形成巨大的效率瓶頸，所以只要 web 協定或 WAP 協定支援 *Adaptive Compression*（160），你就應該利用它來傳送資料。最後一點，你可能得透過 *Resource Files*（101）爲不同的用戶支援不同的語言和頁面佈局。

Small Memory Software

1

Small Architecture

小容量架構

你如何管理整個系統所用的記憶體？

- 記憶體數量如果受到限制，會束縛整個系統
- 系統由多個組件（components）構成
- 各個組件可由不同開發團隊完成
- 組件（components）的記憶體需求量會動態改變

系統記憶體消耗量是一個總體性關注焦點。「在受限的記憶體中大展身手」絕不是一項可以單獨切入的性質：你不能指望一組程式員在你的系統裡加進一些程式碼，系統的記憶體需求量就能降低。記憶體束縛（memory constraints）貫穿著系統設計的始終，影響著系統的每一部分。這正是記憶體受限系統的設計難點（Buschmann et al. 1996; Shaw and Garlan 1996; Bass et al. 1998; Bosch 2000）。

舉個例子，Strap-It-On 腕型 PC 有一個支援多種字型的電子郵件應用程式。爲了提高性能，該程式早期版本將所有載入字型以快取方式儲存起來（cached），並壓縮所有電子郵件，丟棄所有附件；如果載入某字型時記憶體耗盡，系統會崩潰，因此性能低劣，可用性極差。要知道，限制某處只能運用數百個 bytes，卻在他處浪費數百萬個 bytes，是沒有意義的。一旦程式崩潰，你的任何節約都將失去意義。

你可以將系統設計成一個「龐大獨立具整體性的單一組件」（a monolithic single component），或稱「大泥球」（Foote and Yoder 2000）。這種方式雖然看起來誘人，但只適用於最簡單系統，原因是這種設計很難將開發任務分割給不同的程式員或編程團隊；由於系統的各部分互相影響，發展出來的系統既難理解又難維護；而且你無法購入已有的可復用組件。

爲了牢牢掌握系統，你可以從你有能力獨立設計、建造、測試的組件開始一步一步構築系統。你的組件可以復用先前已開發完成者，或從外部購買，或從頭打造；某些組件的開發可能需要專門技能，某些組件囿於自身特性只能透過商業途徑購買。爲了避免重複投資，不同的組件可以分配給不同的發展團隊（Szyperski 1999）。

這些組件的類型可能各不相同，透過多種途徑交互作用：以 (1) 源碼程式庫（source libraries）或 (2) 目的碼程式庫（object libraries）的型式編入系統，或以 (3) 動態連結程式庫（dynamic-linked library）的型式於執行期間載入，或成爲「運用 CORBA、Java Beans、ActiveX 等技術之獨立位址空間」中的 (4) 執行期物件（runtime objects），或成爲獨立行程中的 (5) 可執行程式（separate executables）。所有組件邏輯上都是獨立的，如果它們互相通訊，必須透過介面。

不幸的是，把程式切割爲組件，並不能降低記憶體用量。整個系統的記憶體需求等於所有組件的記憶體需求總和。而且每個組件的記憶體需求量（乃至於整個系統的記憶體需求量）隨著系統的執行而動態變化。不過儘管記憶體消耗量整體性地影響架構，你還是應當儘可能將組件分開處理。現在，如果系統的確被劃分爲多個組件，你如何高效運用記憶體，爲客戶提供最佳服務？

因此：讓每個組件管理自己的記憶體運用狀態。

所謂系統架構，並不僅限於高階（high-level）組件的設計，以及組件間的相互連接，還必須定義出系統的架構策略（architecture strategy），包括組件共同政策、標準和假設（Bass et al. 1998; Brooks 1982）。記憶體受限系統的架構必須闡述記憶體管理策略，保證當系統作爲一個整體型式出現時，每個組件的記憶體配置均可行。

這意味在記憶體受限系統中，每個組件必須明白承擔「管理自身記憶體用量」策略的實作職責。你尤其應當注意 *Small Data Structures*（169），它可以用最少的記憶體儲存系統所需資訊。

當組件以靜態方式配置記憶體時（*Fixed Allocation* (226)），記憶體管理相當容易：組件很單純地擁有固定於其內部的記憶體。但 heap 屬於全域資源，當組件動態地從 heap 配置記憶體時（*Variable Allocation,* 236），管理的難度就提高了。「讓組件始終擁有所有動態配置物」是一個良好的開端（Cargill 1996）。爲了讓組件得以操控其動態配置行爲，你可能需要實作 *Memory Limit*（32）或是以 *Pooled Allocation*（251）方式來配置物件。當組件之間交換物件時，你可以運用 *Small Interfaces*（38）保證某個組件永遠負責管理交換過程中的記憶體需求。

所謂系統架構，還需設置仲裁策略，用以紓解「組件間相互競爭的記憶體索求」。當未經配置的記憶體所剩無多或甚至告罄時，系統對這種仲裁策略的需求尤爲迫切。如果系統無法滿足組件的記憶體需求，你應當保證這些組件只遭受 *Partial Failure*（48）的洗禮。或許，犧牲低優先權組件的記憶體之後（*Captain Oates,* 57），系統能夠繼續運作，勉力支撐到有更多記憶體可用爲止。

例如 Strap-It-On PC 的架構，將字型管理器與 E-mail 顯示器定義爲獨立組件，並爲每個組件設定了合理的記憶體耗用量。字型管理器的設計者實作出 *Memory Limit*，將字型快取裝置（cache）縮減到合理大小；E-mail 顯示組件的設計者發現他們可以因而獲得比預期好得多的效率。當 E-mail 組件顯示大型電子郵件時，便利用字型管理器的 *Small Interface* 縮減字型快取裝置的大小。同樣道理，一旦系統記憶體短缺，字型快取裝置便丟棄所有非使用中的字型。

結果（Consequences）（譯註：本段保留之 *Italic Times New Roman* 字型均爲 forces 名稱）

在程式架構中明白處理記憶體問題，可降低程式的記憶體需求（*memory requirement*），提高程式的記憶體用量可測性（*memory predictability*），也可能使程式伸縮性更好（*scalable*）、可用性更強（*usable*）。

以一致的方式處理記憶體，可以降低編程心力（*programmer effort*），因爲程式員無須重新決定每個組件的記憶體策略。各自獨立的模組和團隊能夠順利合作提供一致性的總體效果，客戶因而可以在記憶體數量很低時預測系統的最終行爲，也就提高了可用性（*usability*）。

總地來說，明確描述系統架構，可以提高系統的設計品質（*design quality*）和易維護性（*maintainability*）。

然而：設計小容量架構需要程式員付出更多努力。「保證組件根據架構規則來設計」亦要求較高的程式員素養（*programmer discipline*）。「將記憶體當作架構關注焦點」使得記憶體由獨立組件和團隊內的局部問題轉化爲總體關心焦點，範圍涵蓋整個專案。例如開發人員可能會以犧牲其他團隊開發的組件爲代價，將自己組件的記憶體需求降至最低。

如果外部組件不符合系統架構標準，將它們強行納入系統就得付出大量的編程心力（*programmer effort*）— 你可能不得不重新實作那些不適用的組件。

設計一個適合記憶體受限環境下的架構，會侷限程式的可伸縮性（*scalability*），因爲它把不必要的限制強加於程式之上。如果有更多記憶體可用，根本不必增加這些限制。

❖ ❖ ❖

實作（Implementation）

本範式背後的主要理念爲「一致性」與「責任」。將系統分解爲獨立組件，你便可以一塊一塊地設計、建造系統；借助統一的記憶體策略，你可以確保發展出來的各部分能夠高效率地協同工作。

用以表現組件的實際編程機制並不特別重要。組件可以是個 class，package 或 namespace，也可以是獨立的可執行程式（separate executable）或行程（process），可以是 COM 或 CORBA 等中介層（middleware）提供的組件，或是一組特別物件（objects）、資料結構、函式和程序（procedures）。在物件導向系統中，組件通常包括許多不同的物件，經常是不同類別的實體，並搭配一或多個物件充當 *Facades*（Gamma et al. 1995）以提供「訪問整個組件」的介面。

設計小容量系統的組件介面時，有兩個需要深入考慮的問題。

1. 可剪裁性（Tailorability）

不同的客戶所使用的相同組件，其記憶體需求亦不相同。設計可復用組件時尤其如此；這類組件將用於許多不同情境（contexts），各種情境下的記憶體需求可能迥然不同。

組件可以透過「在介面中納入裁剪記憶體的參數」來解決此一問題。客戶可以調整這些參數，使組件適用於其所處情境。例如任何運用 *Fixed Allocation*（226）的組件，必須提供生成期（creation time）參數，以便調整它們能夠儲存的元素數量。同樣道理，運用 *Variable Allocation*（236）的組件可以開放一些參數，讓使用者調整記憶體的運用狀態，例如最大容量、初始配置、自由空間（在 hash table 中，保留自由空間可提高查詢效率）。組件也可以支援直接操控其行爲的某些操作，例如要求資料庫自我壓縮，或要求快取裝置（cache）將自己淨空。

舉個例子，Java 的 vector class 有數個操控記憶體的 methods。生成 Vectors 時，我們可以爲它分配足夠記憶體以保存特定數量的元素（例如 10）：

```
Vector v = new Vector(10);
```

容量可以動態增加（例如增加爲 20）：

```
v.ensureCapacity(20);
```

我們也可以減少其容量，使其所供應的記憶體數量剛好足夠保存容器內的元素。例如：

```
v.addElement( new Object() );
v.trimToSize();
```

爲結構配置精確的大小，可節約爲數驚人的記憶體，並減少底層配置器或垃圾回收器的負荷。想像有個 vector 用來儲存 520 個元素，每次安插一個元素。這個 vector 初始配置空間可儲存 8 個元素，空間用光後就配置二倍空間，並將當前元素複製到新空間中，同時歸還原空間。爲了儲存 520 個元素，這個 vector 得改變大小七次，最終的記憶體用量幾乎是必要量的兩倍；整個過程中配置的記憶體數量大約是實際需要量的四倍。與之相比，令此 vector 一開始（初始化時）即爲 520 個元素大小，那麼就只需呼叫記憶體配置系統一次，而配得的記憶體數量又剛好與所需相同（Soukup 1994）。

2. 讓客戶負責管理組件的記憶體配置

有時候，組件需要支援迥然不同的記憶體配置策略 — 某些客戶也許希望使用 *Pooled Allocation*（251）動態配置 package 內的所有物件，其他客戶可能偏愛 *Memory Limit*（32）或 *Memory Discard*（244），另一些客戶可能更欣賞由 system heap 內直接配置物件（非常簡潔）。你如何在實作一個組件時迎合所有需求？

2.1 **運用 Callbacks 管理記憶體**。一個最簡單的方法就是讓組件呼叫「客戶提供的記憶體管理函式」。在 non-OO 環境中，你可以讓組件呼叫客戶提供的函式，並將客戶和組件連結（link）起來。在 C 語言裡頭，你也許可以使用函式指標，或是讓組件宣告一個「將由程式庫提供實體」的函式原型。例如 X Window System 的 Xt Window System Toolkit 就支援一個 callback 函式：`XAlloc` function hook；客戶可提供自己的一個記憶體配置函式，再提供一個自己的記憶體釋放函式（Gilly and O'Reilly 1990）。

2.2 **記憶體策略**（Memory Strategy）。在 OO 環境中，你可以運用 *Strategy* 範式（Gamma et al. 1995），先定義一個針對「一整群配置演算法」的介面，然後再爲個別組件寫出「與使用環境相配合」的演算法。例如在 C++ 中一個所謂的 strategy class 可能僅僅提供配置記憶體和釋放記憶體的操作函式：

```
class MemoryStrategy {
   virtual char* Alloc(size_t nBytes) = 0; // returns null when exhausted.
   virtual void Free(char* anItem;) = 0;
}
```

然後 `MemoryStrategy` class 的某個具體實作版本再實作出特定策略：`PooledStrategy` 實作出 *Pooled Allocation*；`LimitStrategy` 運用 *Memory Limit*；`TemporaryHeapStrategy` 實作出 *Memory Discard*；`HeapStrategy` 則簡單地以系統提供的 `malloc()`和 `free()`完成上述 `Alloc` 和 `Free` 兩項動作。

另一種 C++ 設計採用「編譯期的 template 參數」而非「執行期的 object」。 C++ STL 的 collection 和 string templates 接受一個 class 參數（稱爲配置器 `Allocator`），提供配置函

式和釋放函式 （Stroustrup 1997）。STL 並提供一個配置器實作版本，使用一般的 heap 操作。所以 STL 的 `set` template class 定義如下：

```
template <class Key, class Compare = less<Key>,
          class Allocator = allocator> class set;
```

請注意其中 `Allocator` 參數預設爲 `allocator`，後者是個 strategy class（策略類別），採用一般的 heap 配置手法。

❖ ❖ ❖

特化範式（Specialzation Patterns）

以下各節描述本章的六個特化範式，它們描述了可減少 RAM 記憶體用量的架構性決策。圖 1.1 顯示它們之間的相互關聯。本書另有數個範式與它們息息相關，圖中以灰色表示。

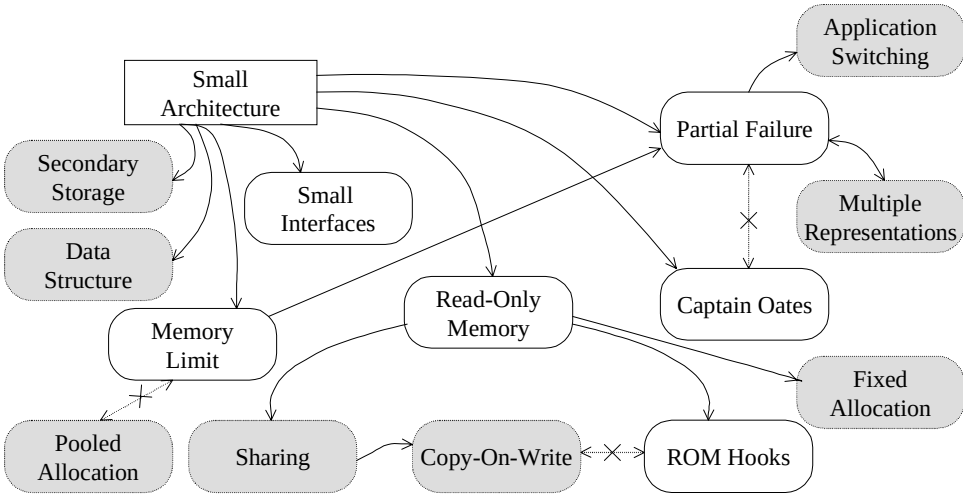


圖 1.1 範式間的關係

本章範式包括：

Memory Limit（32）— 爲組件可配置的記憶體數量設一個上限。

Small Interfaces（38）— 介於組件之間，用來明確管理記憶體，將組件實作品的記憶體需求量降至最低。

Partial Failure（48）— 當組件無法再配置任何記憶體時，保證這個組件能夠在「降級模式」下繼續執行，不至於終止生命或遺失現有資料。

Captain Oates（57）— 系統記憶體匱乏時，放棄重要性較低的組件所使用的記憶體，用以提升系統總體性能。

Read-only Memory（65）— 用來儲存不會被修改的組件。比起受限更多、價格更高的主記憶體，這種唯讀記憶體應該被優先使用。

Hooks（72）— 使儲存於 *Read-Only Memory*（或組件間共享記憶體）內的資訊好像能夠被改變。

❖ ❖ ❖

已知應用（Known uses）

C++ standard template library（Stroustrup 1997）和 Booch components（Booch 1987）都是爲支援不同記憶體策略而設計的物件導向 APIs。這些程式庫都提供有調整組件策略所需的參數（standard Java 和 Smalltalk collection classes 也在相對較低的程度上做到了這一點），它們預先配置數量精確的記憶體，藉以維護整個結構。

參見（See Also）

許多小容量架構利用 *Secondary Storage*（79）降低主記憶體需求量，或是設計 *Small Data Structures*（169）將記憶體用量降至最低，並鼓勵組件間的程式碼和資料共享（*Sharing*，182）。

Tom Cargill 的 *Localized Ownership* 範式（Cargill 1996）闡釋了如何確保每個組件始終精準地僅僅負責一個物件。*Hypoth-a-Sized Collection* 範式（Auer and Beck 1996）講解如何在生成期就爲 collections 配置足夠的記憶體以滿足客戶需求，不必再追加配置。

Software Requirements & Specifications（Jackson 1995）和 *Software Architecture*（Shaw and Garlan 1996）講述將系統劃分爲組件時保證架構連貫的方法。*Software Architecture in Practice*（Bass et al. 1998）大量描述軟體架構；*Design and Use of Software Architectures*（Bosch 2000）是一本新書，專論創建相似軟體系統的生產線。*Patterns in Software Architecture*（Buschmann et al. 1996）包括許多架構級（architecture-level）範式，有助於設計整體系統，值得一讀。

Practice of Programming（Kernighan and Pike 1999），*Pragmatic Programmer*（Hunt and Thomas 2000）和 *High-Level and Process Patterns from the Memory Preservation Society*（Noble and Weir 2000）講解系統組件記憶體消耗量估測，以及整個開發專案中的管理估測技術。

Memory Limit（記憶體限額）

另名：Fixed-sized Heap（定量 Heap），Memory Partitions（記憶體分割）

如何在多個相互競爭的組件之間分配記憶體？

- 你的系統包含許多組件，每個組件有各自的記憶體需求。
- 組件的記憶體需求隨著程式的執行而動態變化。
- 如果一個組件霸佔太多記憶體，會妨礙其他組件工作。
- 你可以爲每個任務（task）設定一個合理的記憶體上限（upper bounds）。

作爲 *Small Architecture*（25）設計哲學的一部分，你可以將系統分割爲多個組件，並讓各個組件負責管理自己的記憶體運用情況。每個組件的記憶體需求會隨著程式的執行而發生變化，視系統的負荷類型而定。如果不加限制，每個組件都會儘可能配置自己所需的記憶體，對其他組件的需求視若無物。由於其他組件也都需要記憶體，系統最終可能因爲記憶體耗盡而徹底完蛋。

例如，Strap-It-On 的虛擬實境遊戲 Stare War 內含數個組件：虛擬實境顯示，語音輸出，音樂概覽，語音辨識，以及協調整個遊戲的人工智慧腦。每個任務對記憶體的需求愈多愈好，但如果所有組件都試圖多配置一些記憶體，系統記憶體就會左支右絀不敷運用了。所以你必須審慎地在組件之間分配可用記憶體。

實作時不妨考慮 *Captain Oates*（57）範式，允許記憶體犯窮的組件從記憶體富裕的組件那裡偷些來用。然而 *Captain Oates* 仰賴組件設計者的菩薩心腸（願意釋出記憶體），實作時可能既困難又複雜。

你也可以考慮事先制定組件的記憶體用量。然而，除非有什麼方法可以保證組件一定遵守預算，否則無濟於事。「保證遵守預算」對於只使用 *Fixed Allocation*（226）的組件而言平淡無奇，但對其他組件而言，「模塑（*modeling*）其動態行爲並確保它們不會破壞你的計劃」殊非易事。

因此：爲每個組件設置限額。對於超出限額的配置請求，予以拒絕。

這個範式的應用分三步驟：

1. 記錄當前每個組件配置的記憶體數量。例如，你可以修改組件的記憶體配置常式（memory allocation routine），讓它在配置記憶體時增加記憶體計數，歸還記憶體時減少記憶體計數。
2. 確保組件所配置的記憶體數量不超過分配限額。組件在「超額配置」情況下所遭遇的失敗模式與組件在「系統缺乏可用記憶體」時所遭遇的失敗模式完全相同。爲保證即便觸及限額仍可繼續執行，組件應當支援 *Partial Failure*（48）。
3. 理想狀態下應當透過「試驗、檢視記憶體用量」的手段來爲每個組件設定限額。最後才設定限額似乎是在走回頭路，事實上你必須在發展過程中修改限額，或允許用戶爲適應其工作而調整限額。所以，請先建立流水帳機制（accounting mechanism），然後實驗、收集資訊、最後再設置你想推行的策略（policies）。

「所有組件的限額總和」應當等於或大於「可用記憶體總量」？答案取決於所有任務是否可能同時使用各自最大限額記憶體。現實中這種情況不太可能出現，而且 *Memory Limit*（32）的主要用途是防止某個組件獨霸全部記憶體。因此，只要保證每個任務得到的限額在全部記憶體中佔據合理份額就夠了。

請注意，只有爲「記憶體需求有所變化」的組件設置限額才有意義。記憶體限額對於那些「大多數結構使用 *Fixed Allocation*（226）」和「記憶體用量隨著時間並無顯著變化」的組件提供不了什麼好處。

例如，在 Stair Wars 遊戲中，人工智慧腦使用的記憶體數量大約和其支援的敵我雙方物質（entities）數成正比。經由試驗，開發人員確定這些物質的最大數量，然後調整智慧腦組件的記憶體限額，使之可以提供最大量物質同時存在情況下所需的記憶體。另一方面，螢幕顯示組件配置的是一塊大小固定的記憶體，因此 Stair Wars 不必爲此組件實施記憶體限額。

結果（Consequences）（譯註：本段保留之 *Italic Times New Roman* 字型均爲 forces 名稱）

由於每個組件的記憶體用量都有確定限額，所以你可以獨立地測試每一個組件，保證在最終系統中它還將以相同方式繼續工作。這就提高了系統的可預測性（*predictability*）。

通過檢視記憶體計數值，甄別出問題區域，並發現執行期間哪個組件因爲記憶體不足而失敗就輕而易舉了，這就提高了系統的區域化（*localization*）。

實作一個簡單的記憶體計數器只需付出一點點編程心力（*programmer effort*）。

然而：某些任務（tasks）正常工作時，另一些任務卻可能因爲缺乏記憶體而失敗；如果任務之間有大量互動，可能會引發一些不常見的錯誤狀態，難以重現和測試。即使系統中還有大量記憶體，組件也可能因爲觸及自身記憶體限額而失敗；所以本範式可能會浪費記憶體。大多數簡單的記憶體計數機制無法解決破碎（*fragmentation*，見 *Memory Allocation*, 219）引起的額外浪費。另一方面，更複雜的作業系統機制，例如爲所有組件分配獨立的 heaps，又會增加同樣的破碎（*fragmentation*）浪費。

❖ ❖ ❖

實作（Implementation）

有數種記憶體限額實作方法可供選擇：

1. 攔截記憶體管理操作函式

許多編程語言都允許你輕易攔截記憶體配置函式和歸還函式，並修改這些函式，以追蹤已配置的記憶體數量。計數值達到限額時，除非歸還函式傳回的計數值低於限額，否則此後的配置動作將會失敗。在 C++ 中，你可以改寫四個全域的 `new` 和 `delete` 運算子（Stroustrup 1995）。

本範式並不要求計數器有多高的精確度。只爲「主要的記憶體配置行爲」（例如大型緩衝區的配置）實作一個計數器亦足矣。如果被配置的記憶體中，較小項目與較大項目成比例，那麼這一限量便是間接統轄了整個任務（task）所用的記憶體總量。例如 Stair Wars 遊戲程式中不同物質（entities）使用的記憶體數量並不相同，但記憶體總用量大致和物質總數成正比，所以限制物質數量也就相當於實作出一個有效的記憶體限額。

在 C++ 裡頭，你可以爲某個 class（亦即相當於爲其所有 derived classes）改寫 `new` 和 `delete` 運算子，藉以實作出一份區域性更強的記憶體限額。這種作法還具有以下優點：即使是從同一個 heap 中配置記憶體，同一個程式的不同部分卻可以因此擁有不同的記憶體限額（Stroupstrup 1997）。

2. 分離的（separate）heaps

你可以讓每個組件各自使用分離的 heap，並各自管理自己的 heap，限制其最大容量。許多作業系統支援這樣的分離（separate）heaps，尤其是 Windows 和 Windows CE（Microsoft 1997a, Boling 1998）。

3. 分離的行程（separate processes）

你可以使每個組件成爲一個獨立行程，藉作業系統或虛擬機器之助，限制每個組件的記憶體用量。EPOC 以及大部份 UNIX 版本都允許你爲每個行程指定一個記憶體限額，系統禁止其內運行的行程超越此限額。使用這類限額幾乎不需耗費任何編程心力（*programmer effort*），而且作業系統還提供記憶體用量監視工具，便於你爲每個行程確定適當的記憶體限額。當然啦，你必須設計你的整個系統，使獨立組件成爲一個獨立行程 — 視系統之不同，這種做法或許十分繁複，或許只是小菜一碟。

許多作業系統以虛擬記憶體實作出 heap 限額。它們在虛擬記憶體位址空間中配置出全額大小的（full size）heap（見 *Paging*, 119）；當行程選擇存取記憶體區塊時，記憶體管理程式才將 heap 映射至實際記憶體。因此，heap 的大小在虛擬記憶體中是固定的，如果不使用 heap 就根本沒有實際的記憶體成本。此方法的缺點是：很少有虛擬記憶體系統能夠偵測出 heap 中的可用記憶體，並將其中的未用區塊還給系統。所以在大部份 VM 系統中，一個行程如果使用 full heap，會造成整個 heap 從那時起處於完全被配置狀態。

示例（Examples）

下面的 C++ 程式碼限制 `MemoryResrictedClass` 及其 subclasses 所使用的記憶體總量。如果超額配置，會觸發標準的「記憶體不足」異常：`bad_alloc`。這裡的總限額以編譯期的 `LIMIT_IN_BYTES` 指定：

```
class MemoryRestrictedClass {
public:
   enum { LIMIT_IN_BYTES = 10000 };
   static size_t totalMemoryCount;
   void* operator new(size_t aSize);
   void operator delete(void* anItem, size_t aSize);
};
size_t MemoryRestrictedClass::totalMemoryCount = 0;
```

這個 class 必須實作出一個 `operator new`，能夠檢查限額並於必要時擲出異常：

```
void* MemoryRestrictedClass::operator new(size_t aSize) {
   if (totalMemoryCount + aSize > LIMIT_IN_BYTES)
        throw (bad_alloc());

   totalMemoryCount += aSize;
   return malloc(aSize);
}
```

當然啦，對應的 delete 運算子必須減少記憶體計數值：

```
void MemoryRestrictedClass::operator delete(void* anItem, size_t aSize)
{
   totalMemoryCount -= aSize;
   free((char*)anItem);
}
```

如果要求完整，整個實作碼還得包括類似的 array 版本（Stroustrup 1995）。

Java 並未提供配置函式和刪除函式。然而，限制某個 class 的實體（instances）數量是完全可能的 — 只要保持一個 static 計數器，記錄已生成之實體個數即可。Java 並沒有提供簡單的歸還函式，但我們可以利用 finalization 攔截歸還動作。注意，許多 Java 虛擬機器實作 finalization 的效率不彰（如果真有實作的話），因此你應當把以下程式碼視爲一個可行方法的示範，而不是獲得推薦的實用手法（Gosling et al. 1996）。

以下 class 只允許產生一定數量的實體（instances）。其中記錄著實體個數，新物件被建構時會增加計數值，垃圾回收器（garbage collector）將它 "finalize" 時會減少計數值。由於物件只能在垃圾回收器執行起來時方可被 finalized，因此在任意時刻都可能存在一些尚未被 finalized 的垃圾物件。如果已經接近限額，爲保證我們不會非必要地拒絕配置請求，建構式（constructor）應在丟擲異常之前先做一次明顯的垃圾回收動作：

```
class RestrictedClass
{
   static final int maxNumberOfInstances = 5;
   static int numberOfInstances = 0;
   public RestrictedClass() {
        numberOfInstances++;
        if (numberOfInstances > maxNumberOfInstances) {
             System.gc();
        }
        if (numberOfInstances > maxNumberOfInstances) {
            throw new OutOfMemoryError("RestrictedClass can only have" +
                                       maxNumberOfInstances +
                                       "instances");
   }
}
```

在建構式中檢查記憶體，會帶來一點小爭議：即使我們擲出異常，物件還是被產生出來了。總的來說這不算什麼問題，因爲除非該物件的 superclass 建構式儲存著該物件的 reference，否則最終它肯定會被 finalized。

實際上的 finalization 碼平淡無奇：

```
public void finalize() {
    --numberOfInstances;
}
```

❖ ❖ ❖

已知應用（Known uses）

預設狀態下，UNIX 作業系統爲每一個用戶行程（user process）設定一個記憶體限額（Card et al. 1998）。此限額阻止任何一個行程獨霸系統的全部記憶體。只有擁有系統權限的行程才能改寫此限額。持續不斷的記憶體洩漏是導致「行程觸及限額」的最常見原因：行程執行了很長一段時間之後，記憶體請求將被拒絕，行程將終止執行而後被重新啓動（譯註：我不理解爲什麼會「重新啓動」）。

EPOC 將每個執行緒關聯一個 heap，並爲每個 heap 定義一個最大尺寸。應用程式的預設限額非常大，server threads（daemons）的限額通常就小得多，因爲它是被執行緒生成函式 `Rthread::Create()`的一個重載版本所產生。EPOC 的文化非常重視避免記憶體洩漏，所以限額只是用來限制系統的一個特殊成份所使用的資源。對系統用戶來說，EPOC 的伺服器常常不可見（invisible），因此防止它們過分膨脹就顯得非常重要。如果一個伺服器確實觸及記憶體限額，它應該採取 *Partial Failure*（48），壯士斷腕地放棄發生問題之特殊請求或 client session，而不是玉石俱焚地摧毀整個伺服器（Tasker et al. 2000）。

Microsoft Windows CE 和 Acorn Archimedes RISC OS 都允許用戶在執行期間調整系統組件的記憶體限額。Windows CE 限定程式可用的記憶體上限，以及資料可用的記憶體上限，RISC OS 則爲作業系統中的每個組件各自設定了限額（Boling 1998; RISC OS 2000）。

Java 虛擬機器通常提供執行期旗標，用以限制 heap 的總體大小，所以你能夠限定一個 Java 行程的大小（Lindholm and Yellin 1999）。Java「即時規格」並爲「在 heap 中進行的記憶體配置動作」設定限額。

參見（See Also）

典型的行程完全可能觸及限額，此時「遭受一次 *Partial Failure*，挽救整個行程」乃爲上策。在相互競爭的組件之間分配記憶體時，如果只用 *Fixed Allocation*（或 *Pooled Allocation*），技術上雖然比較簡單，但彈性卻相形遜色。

Small Interfaces（小型介面）

如何減少組件介面帶來的記憶體額外負擔？

- 你正在設計 *Small Architecture*，其宗旨是「各組件負責管理自身記憶體之運用」。
- 你的系統中有數個組件透過「顯式介面」（explicit interfaces）彼此進行通訊。
- 介面設計可能迫使組件或其客戶配置額外記憶體，這些記憶體僅供組件通訊使用。
- 可復用組件需要泛化介面（generic interfaces），這使我們需要的記憶體「比特定案例所必要之數量」更多。

你正在設計一個 *Small Architecture*，已經將系統分成若干組件，每個組件負責管理自身的記憶體運用。各組件通過其介面互相協作。很不幸，介面本身也需要臨時記憶體儲存引數和結果。在組件間傳送大量資訊可能相應地需要大量記憶體。

例如，Strap-It-On 'Spookivity' ghost hunter（鬼怪獵人）的支援應用程式使用 ROM 中的一個壓縮資料庫，其中儲存所有與給定規格相匹配的鬼怪詳細資料。資料庫組件的早期版本係爲小得多的 RAM 資料庫而設計，所以它們執行「搜尋」操作時僅僅傳回一個可變長度的 array，其中內含符合搜尋條件之所有鬼怪的詳細資料的副本。儘管功能正常，但這種介面設計意味即使回答一個常見查詢（像是找出「透明、帶白色、漂浮且已死亡的鬼怪」）也得配置數 Mbytes 暫用記憶體，啊，Strap-It-On 根本提供不了這麼多記憶體。

如果介面能夠撤消每個組件對記憶體配置的控制權，也會帶給 *Small Architecture* 麻煩。如果某物件由 A 組件配置，由 B 組件使用，最後由 C 組件刪除，那麼 A,B,C 任何一個組件都不能對此物件之記憶體負責。在 Spookivity 應用程式中，雖然鬼怪詳細資料所組成的 array 是由資料庫組件配置出來，但它卻經由某種方式由客戶負責管理。

可復用組件（reusable components）使得「控制記憶體的運用」愈發艱難。可復用組件的設計者經常要面對記憶體使用和其他因素（例如執行速度或失敗模式）的取捨。例如，

在正常處理過程中，爲支援快速反應，組件可能預先配置一些記憶體緩衝區。那麼，究竟應當配置多少記憶體呢？這類問題的答案嚴格依賴系統環境，還可能依賴客戶使用何種組件，甚至依賴當時客戶正在做些什麼。普通而常見的作法（也就是設計者根據一般應用程式的思路，對此問題所做的回答）不大可能在記憶體受限系統中產生令人滿意的結果。

因此：設計出讓客戶得以控制資料傳輸的介面。

設計組件介面有兩個主要步驟：

1. **將介面之間的資料傳輸量最小化**。small interfaces（Meyer 1997）和 strong design（Coplien 1994）兩原則指出，介面只應向其客戶提交最少量的資料和行爲。Small interfaces（小型介面）不應傳輸大部份組件或客戶並不需要的所謂「寄生」資訊。你可以降低介面間傳送的資料量，藉此降低組件帶來的記憶體額外負擔。
2. **決定資料傳輸的品質程度**。一旦你鑑別出需要在組件間傳遞的資料，你就可以決定以多好的品質來傳輸它。有許多不同的機制可以在組件之間傳遞資料，我們將在稍後的實作段落中討論最重要的數個。

舉個例子，新版本之 Spookivity 資料庫搜尋法傳回的是一個 *Iterator*（Gamma et al. 1995）。這個 iterator 的 `getNext()`函式傳回一個 reference to `GhostDetails` 物件，該物件提供「依次傳回每個鬼怪資料」的操作函式。這使得資料庫組件實作者能夠每次復用相同的 `GhostDetails`物件。其實作碼只含一個資料庫 ID，每次被呼叫便改變其值。`GhostDetails` 提供的所有操作都直接從高速資料庫存取資料。修訂後的這種介面只需數個 RAM bytes，而且由於資料庫本身就是按使用 iterators 的方式設計，所以效率不受影響。

結果（Consequences）（譯註：本段保留之 *Italic Times New Roman* 字型均爲 forces 名稱）

明確考慮每個組件介面的記憶體需求，你就可以降低介面間交換資訊的記憶體需求（*memory requirements*），進而降低整個系統的記憶體需求。由於介面間傳送資訊所用的記憶體大多數生命週期都很短暫，因此「消除（或減少）介面所需記憶體」可以讓你的程式的記憶體用量的可預測性（*predicatable*）更強，並支援更高階的即時（*real-time*）行爲。降低組件間的介面記憶體需求，可以降低「在一份設計中使用多個組件」所造成的額外開銷，提高區域性（*locality*）、設計品質（*design quality*）和可維護性（*maintainability*）。

然而：設計小型介面需要良好的編程素養（*programmer discipline*），增加團隊協作的額外負擔。「高效運用記憶體」使介面複雜度（*complex*）更高，因而需要更多程式碼和編程心力（*programmer effort*），也提高了測試成本（*test cost*）。和所有記憶體節約設計一樣，小型介面可以提高時間效率（*time performance*）。

❖ ❖ ❖

實作（Implementation）

在小容量系統中設計組件介面時，有許多需要考慮的問題和可選方案。不論資訊是「正方向」傳遞（也就是由客戶端至伺服器，與控制流方向一致），或是反方向傳遞（也就是由組件至客戶端），相同的技術都可以派上用場。

1. 傳值 vs. 傳址（pass by value vs. pass by reference）

資料可以採用 by value（傳值；拷貝資料）或 by reference（傳址；傳遞指標）兩種方式傳入和傳回。通常 by reference 比 by value 需要的記憶體少，可節約拷貝時間。Java 和 Smalltalk 程式通常使用 by reference 傳遞方式，它意味兩個組件此刻正 *Sharing*（182）資料，所以二者需透過某種方式合作管理記憶體。至於 by value 傳遞方式的接收端（組件）必須負責管理一塊暫時記憶體用以存放接收值。by value 方式在 C++ 中十分常見，可用來 *Discard*（244）stack 的記憶體。

2. 組件間交換記憶體（Exchanging memory across interfaces）

有三種常用策略，可供客戶通過組件介面傳送記憶體：

- 出借（Lending）— 客戶呼叫「提供服務之某組件」期間（或更長期間），由客戶出借一些記憶體給服務供應者。
- 借入（Borrowing）— 客戶獲得服務供應者（某組件）所擁有之物件的存取權。
- 竊取（Stealing）— 客戶接收服務供應者配置的物件，並負責歸還該物件。

呼叫期間，當資訊正向傳入，客戶經常可以出借記憶體給組件。由組件手上逆向傳回資訊給客戶比較困難。儘管客戶可以出借一些記憶體給服務供應者，但相比之下，客戶從伺服器那裡借入結果物件還是比較容易，竊取結果物件並無拘無束地盡情使用也是頗爲容易的。

下面各小節講解並對比上述三種方法。爲求簡便，我們描述「只傳回單一結果物件」的組件；不過如果組件傳回多個物件，相同的子範式（sub-patterns）也適用。

2.1 **出借**（Lending）。客戶將一個物件傳入組件的操作函式內；組件將該操作函式施行於該物件身上，存取其資料（圖 1.2）。如果客戶持有一個 reference 指向結果物件，他就

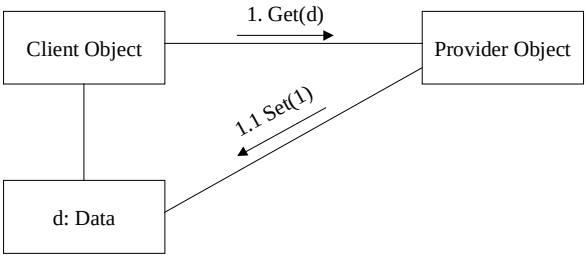


圖 1.2　出借（Lending）

可以直接存取該物件；抑或者組件可將結果物件傳回給客戶。例如以下 Java 程式碼概要展示一個使用「文書處理器（word processor）組件」的物件如何生成一份嶄新的「文件屬性」（document properties）物件，並將該物件傳給「文書處理器」— 後者會初始化該文件屬性物件，用以描述當前文件的屬性（properties）。

```
DocumentProperties d = new DocumentProperties();
wordProcessor.getCurrentDocumentProperties(d);
```

然後客戶可以操縱這個「文件屬性」（document properties）物件：

```
long docsize = d.getSize();
long doctime = d.getEditTime();
```

當這個文件屬性物件不再有用時，客戶必須釋出它：

```
d = null;
```

這是因爲客戶一直負責掌管該物件的記憶體。出借記憶體給組件時，客戶端管理物件（本例之「文件屬性」物件）的配置和壽命：可能採取靜態配置，也可能在 heap 或 stack 中配置。

當你期望客戶配置所有引數物件，而且你確定他們需要全部結果時，請考慮使用這種「出借」方式在介面之間傳遞引數。顯而易見，讓客戶擁有結果物件，便是賦予客戶相當的權力和靈活性。這種作法要求客戶配置一個新物件，用來接收結果，並在不再需要該物件時將它刪除，這需要相當的程式員素養（*programmer discipline*）。組件必須計算所有結果屬性，無論客戶是否需要。

在 C++ 程式庫中，這種技術的一般形式是以 by value（傳值）方式傳回結果，將結果從組件的臨時堆疊（stack）記憶體中拷貝至客戶出借的記憶體中。

另一個出借實例出現於「客戶將物件傳入供組件使用之緩衝區」場合。例如在 *Buffer Swap* 範式（Sane and Campbell 1996）中，組件需要即時記錄一個物件集合（例如聲音樣本），並將它們傳回給客戶。客戶首先給主要組件（main component）提供一個緩衝區，之後，每當緩衝區被填滿，便再提供一個新的。

2.2 **借入**（Borrowing）。組件擁有一個簡單物件或複合（composite）物件，並將其 reference 傳回給客戶。客戶使用「施行於物件身上」的操作函式來存取物件資料，並在客戶不再需要該物件時通知（signal）組件（圖 1.3）。

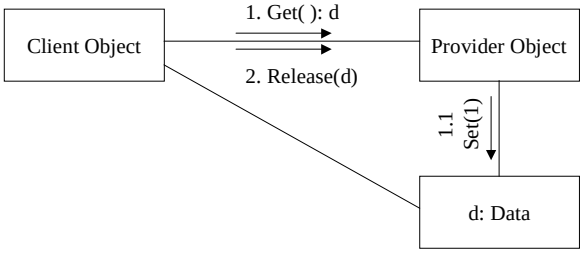


圖 1.3 借入（Borrowing）

例如，文書處理器組件允許客戶借用一個「表示當前文件屬性」的物件：

```
DocumentProperties d = wordProcessor.getDocumentProperties();
```

然後客戶可以操縱這個物件：

```
long docsize = d.getSize();
long doctime = d.getEditTime();
```

當客戶不再需要此一屬性物件時，必須告知文書處理器：

```
wordProcessor.releaseDocumentProperties(d);
```

與「出借」情況相仿，「借入」既可用於「向組件傳入資料」，也可用於「由組件傳出資料」。組件擁有結果物件，便是獲得了最大靈活性。組件可以每次配置新物件（*Variable allocation*, 236），也可以永久持有一個或多個實體（*Fixed allocation*, 226），甚至可以兩種方式並用。

另一方面，組件如今不得不管理結果物件的壽命 — 如果某一時刻同時有數個客戶或同時需要數個資料物件，這麼做就很困難。另一種作法是，組件可以只是靜態配置一個結果物件，並在每次調用（invocation）時循環使用它。其條件是組件一傳回資訊，客戶必須立刻加以拷貝。靜態結果物件（static result object）無法應付並行（concurrent）存取，但只要你保證任意時刻只有一個客戶，此法就暢行無礙。

還有一種作法是由組件介面提供一個明顯的 "release" 函式來刪除結果物件。這種作法在 Java 和 Smalltalk 中極罕見，因爲這些語言已經確保「當異常被擲出時，"release" 函式一定會被喚起」。這種作法在 C++ 介面中倒是屢見不鮮，因爲 C++ 允許組件針對物件實作出 *Reference Counting*（268），或乾脆在 `Release()`函式實作碼中執行 `delete this`。舉個例子，EPOC 撰碼風格（coding style, Tasker et al. 2000）就要求所有介面（'R classes'）必須提供一個 `Release()`函式（而不是一個解構式）。

當組件需要產生大型物件或是爲其客戶提供大型物件，而客戶不太可能長期保留這些物件時，請考慮使用以上所討論的「借入」手法。

2.3 **竊取**（Stealing）。組件配置一個簡單物件或複合物件，並將其管理責任移轉給客戶。客戶使用「實施於物件身上之操作函式」取得資料，然後釋放之（例如 C++）或依靠垃圾回收機制（例如 Java）來釋還記憶體（圖 1.4）。

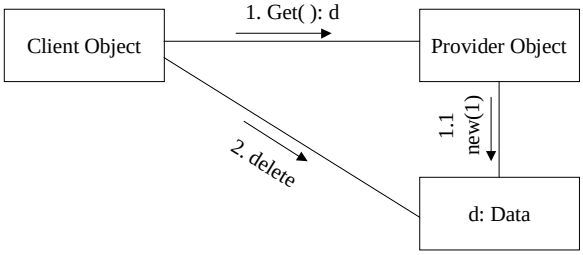


圖 1.4　竊取（Stealing）

例如文書處理器（word processor）可讓客戶竊取一個「文件屬性」物件：

```
DocumentProperties d = wordProcessor.getDocumentProperties();
```

允許客戶在需要時使用此物件：

```
long docsize = d.getSize();
long doctime = d.getEditTime();
```

不過如今客戶必須擔負管理或刪除物件之職責：

```
d = null;
```

這個例子展示客戶如何竊取最初隸屬於組件之物。當資料從客戶流往組件時，組件亦可竊取原屬於客戶的物件。物件管理責任（或物件所有權）的移轉很容易在程式中實現，這在 Java 和 Smalltalk 等「支援垃圾回收，無須明白執行 delete 操作」的語言中司空見慣。在 C++ 中，這種技術最適合應用於「未受限制的字串」（unbounded strings）這一類變長結構（variable size structures）。然而系統如果沒有垃圾回收機制，那麼除非傾注大量 *programmer discipline* 來刪除每一個傳回物件，否則這種技術可能引發記憶體洩漏。擁有

權的移轉迫使伺服器爲回返結果配置一個新物件，而此物件需要記憶體。無論客戶是否需要「被傳回物件」的全部屬性，伺服器都必須一一計算，無形中浪費了 *processing time*（處理時間）和記憶體。

當組件需要提供「客戶接收後將保留一段時間」的大型物件時，請考慮使用竊取手法。

3. 漸進式介面（Incremental interfaces）

經由介面，傳遞資料序列或資料群集（a sequence or collection of data items），是一件十分困難的事。在記憶體受限系統中，或記憶體經常破碎（fragmented）的地方，可能沒有足夠記憶體可用以儲存整個群集。這種情勢下介面應當被設計成漸進式（incremental），亦即執行一次以上的呼叫，將資訊從客戶端傳至組件，每次呼叫僅傳送一小部分資訊。漸進式介面可用於「進、出」雙向傳輸。客戶可以對著組件直接呼叫多次，也可以使用 *Iterator*（Gamma et al. 1995）作爲媒介物。

3.1 **客戶執行多次呼叫**。客戶對組件進行多次呼叫，每次呼叫都在呼叫期間出借（*loaning*）單一物件給組件。全部物件傳送完畢後，客戶再執行一次呼叫，通知組件說已送出完整群集，然後組件接下去處理（圖 1.5）。

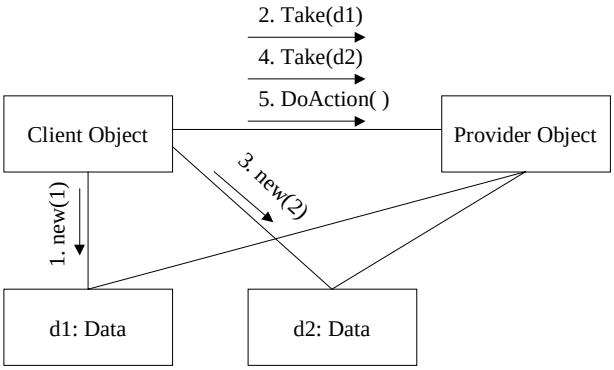


圖 1.5 客戶執行多次呼叫

例如，客戶可以將大量圖片插入文書處理器中 — 先呼叫 `addParagraph()`讓文書處理器接收每張圖片，再呼叫 `processAddedParagraphs()`處理並編排所有新段落：

```
for (int i = 0; i < num_new_paras; i++) {
        wordProcessor.addParagraph(paras[i]);
};
wordProcessor.processAddedParagraphs();
```

「客戶執行多次呼叫」是比較容易被理解的，所以編程新手常選擇這種方法，non-OO 語言中也常常用到它。但它迫使組件要嘛找到「以漸進方式處理資料」的作法（見 *Data files*, 92），要嘛則是進一步配置記憶體，產生屬於自己的「傳入物件所形成的群集」。另一種作法是，客戶在整個處理期間（而非每次呼叫期間）出借記憶體，但這麼做會強迫客戶保管所有配置完畢的資料，直至 `DoAction()`操作函式完成爲止。

爲了讓組件以漸進方式傳回資訊，客戶再次執行多次呼叫，但組件會以回返值或類似物標誌資料流的結束。

```
spookivity.findGhosts("transparent|dead");
while (spookivity.moreGhostsToProcess()) {
   ghostScreen.addDisplay(spookivity.getNextGhost());
};
```

3.2 **透過** `iterator` **傳送資料**。客戶可以不必執行多次呼叫，而是出借一個 iterator 給組件。然後組件藉由此一 iterator 進一步存取出借物。例如客戶可以傳遞一個「指向內部群集之一」的 iterator：

```
ghostScreen.displayAllGhosts(Spookivity.ghostIterator());
```

組件可以憑此 iterator 存取客戶資訊：

```
void displayAllGhosts(Iterator it) {
   while (it.hasNext()) {
        displayGhost((Ghost) it.next());
   }
}
```

傳入 iterator，便是逆轉了控制流，所以現在由組件調用（invoking）客戶。

概括而言，使用 iterator 比多次呼叫某個特殊介面更具彈性。由於組件可以藉由 iterator 存取物件群集，因此組件不必再儲存自己的物件群集。介面「使用抽象的 iterator 或抽象的 collection class（群集類別）」很重要；常見的介面錯誤設計是：以某個特定群集類別替代前二者，這會使客戶的實作受到限制。

3.3 **透過** `writable iterator` **傳回資料**。所謂 writable iterator 係指「非單純巡訪群集，而是能夠向群集插入元素」的 iterator。客戶產生的 writable iterator 用來實作「從組件到客戶」的外出流（outward flows），就像一般 iterator 用來實作入內流（inward flows）一樣。

```
Vector retrievedGhosts = new Vector();
spookivity.findGhosts("transparent|dead");
spookivity.returnAllGhosts(retrievedGhosts.writeableIterator());
```

請注意，本書寫作之時，Java 程式庫尚未包含 writable iterators。

3.4 **藉由傳回 iterator 傳回資料**。客戶從組件那裡或借入或竊取一個 iterator 物件，並透過它存取傳回值：

```
Iterator it = spookivity.findGhostsIterator("transparent|dead");
while (it.hasNext()) {
   ghostScreen.displayGhost((Ghost) it.next());
}
```

傳回 iterator，使得控制流由客戶指向組件，允許這個 iterator 被客戶碼操縱或傳遞給其他客戶組件。

❖ ❖ ❖

已知應用（Known uses）

介面俯拾可得。想找一個「適用於記憶體受限系統」的介面經典範例，請看 EPOC 或 PalmOs 作業系統所提供的 API 文件（Symbian 1999b, Palm 2000）。

作業系統的檔案 IO 呼叫必須在系統和應用程式之間傳送大量資訊。通常它們需要客戶出借緩衝區記憶體，然後直接讀或寫客戶端緩衝區。例如古典 UNIX（Ritchie and Thompson 1978）的檔案系統呼叫：

```
read(int fid, char *buf, int nchars);
```

便是從檔案 `fid` 中讀取最多 `nchars` 個字元，放入從 `buf` 開始的緩衝區。緩衝區不過是一塊生鮮而未經任何處理的記憶體（所謂 raw memory）。

EPOC 的主從介面（client-server interface）總是採用出借（lending）手法，因爲伺服器和其客戶分處於不同的記憶體空間中，因而只能將輸出拷貝至「客戶內部專爲輸出而保留」的記憶體，如此才能傳回輸出結果。這就保證了記憶體索求通常很小，客戶的記憶體需求量也可以固定下來。

許多標準介面都使用 iterators。例如 C++ iostreams 程式庫幾乎專門運用它們來存取容器類別（container classes, Stroustrup 1997）。Java 的 Zlib 壓縮程式庫將 iterators（streams）既用於輸入又用於輸出。

參見（See Also）

介面必須支援整個系統的總體記憶體策略，因此許多記憶體範式可以藉由組件間的介面反映出來。

介面可以提供一些函式，用來模擬組件中的記憶體失敗，進而對客戶和組件進行

Exhaustion Testing（Noble and Weir 2000）。如果介面傳回的 references 指向組件所擁有的物件，那麼介面就可以 *Share*（182）這些物件，還可以使用 *Reference Counting*（268）或 *Copy-On-Write*（191）。

介面，尤其是 C++ 介面，可以運用常數參數代表 *Read-Only Memory*（65），因而讓使用者無法改變其值。在其他語言中，這種手法是介面文件的一部分。凡組件使用 *Resource Files*（101）之處，介面通常使用字串或資源（而不使用結構）來表示資源 IDs。這麼做不但可以減少介面間的資訊傳送量，而且資源的記憶體成本將只由「實際具現之並使用之」的組件承擔。

如果組件（或編程環境）以 handles 支援 *Compaction*（259），那麼介面可以使用 handles（而不使用物件的 references）來表示組件內的物件。

Arguments and Results（Noble 2000）和 *Pattern Languages of Program Design 4*（Pryce 2000）所描述的範式闡釋如何引入物件，用以協助組件間的介面設計。Meyers 的 *Effective C++*（1998）和 Sutter 的 *Exceptional C++*（2000）描述優秀的 C++ 介面設計。Tony Simons 描述如何使用借入（borrowing）、拷貝（copying）和竊取（stealing）手法來設計 C++ classes（Simons 1998）。

Partial Failure（局部毀棄，降格求全）

另名：Graceful Degradation（優雅的降級）；Feast and Famine（盛宴與饑餓）

如何處理不可預見的記憶體需求？

- 無論怎樣降低程式的記憶體需求，你還是耗盡了記憶體。
- 寧可在非關鍵任務（trivial task）中失敗，也不要輕率放棄關鍵任務（critical task）。
- 持續不斷地執行，比始終完美地執行，更重要。
- 持續不斷地執行，比系統崩潰，更是重要得多。
- 系統中的可用記憶體的數量，隨著時間變化極大。

無論你做了多少工作降低程式的 *memory requirements*，它還是會耗盡記憶體。你可以悄無聲息地丟棄一段找不到儲存空間的資料，或提示一個粗暴的錯誤訊息而後終止處理，或繼續執行就好像你已經得到了你所需要的記憶體致使程式以不可預期的方式崩潰。無論如何你迴避不了這個問題。或明或暗，你都必須和「記憶體耗盡」打交道。在「傳統」系統中，記憶體短缺（low-memory）狀態極爲罕見，所以花費過多程式員心力來處理這個問題並不值得。預設的解決之道是「聽任程式崩潰」，這通常可以接受。別忘了，導致程式崩潰的原因還有很多，多那麼一兩個，用戶不會發現！但是在記憶體受限系統中，記憶體短缺狀態經常會出現，因此如果你繼續沿用老辦法，會嚴重影響系統的可用性，甚至使系統報廢。

例如，Word-O-Matic 文書處理器（word processor）爲每張圖片提供了語音輸出，並爲突顯拼寫、語法及政治正確性上的錯誤，添加了閃爍顏色；還提供一個浮動視窗持續不斷地提示句子的結尾和可能的改寫方式。這些功能都需要佔據大量記憶體，經常會耗光系統所有可用的 RAM 記憶體。

然而也有一些好消息要告訴你。首先，某些系統需求比其他需求更重要 — 因此如果你必須失去一些東西，那麼失去某些東西會比失去另一些東西好。其次，如果系統持續不斷地執行，「未能滿足一項需求」並不必然意味後續所有請求都將失敗。最後一點，你不太可能無限期缺乏記憶體。當系統空閒下來，其記憶體需求肯定比負載沉重時小得多。

例如在 Strap-It-On 腕型 PC 上，「讓系統持續執行並保證計時和鬧鈴功能始終能夠反映最新變化」，要比「讓任何華麗的編輯功能得以運作」更重要。在 Word-O-Matic 系統中，「把用戶辛苦透過二指鍵盤（two-finger keypad）輸入的文字保留下來」甚至比「顯示該段文字」更重要，比起「拚字檢查和語法檢查」及「改寫提示」更是重要得多。

因此：請確保即使記憶體消耗殆盡也要讓系統處於安全狀態。

保證每次記憶體配置失敗時，在失敗傳播開來之前，總能夠有應對策略。

一旦程式偵測到記憶體配置失敗，首先必須回復到一個安全穩定狀態，清除配置失敗所導致的任何矛盾。此一過程應當儘可能保證不丟失任何資料。視記憶體耗盡時所配置的物件的不同，有可能從「申請額外記憶體」的動作中撤回就足夠了（圖 1.6）。另一種情況是，你可以減少一個或多個組件所提供的功能，或甚至在錯誤發生時關閉該組件。

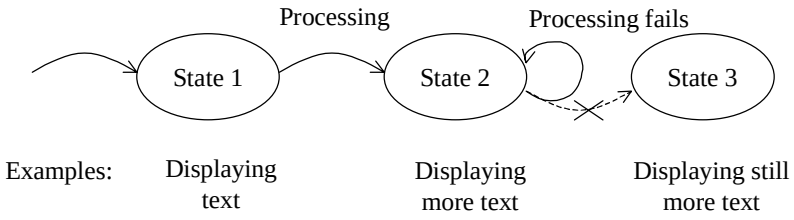


圖 1.6　拒絕「申請額外記憶體」動作

至關重要的一點是，從用戶的角度來看，必須保證一個動作要嘛完全成功，要嘛徹底失敗，兩種情況下系統都應處於穩定狀態。當一個足以影響用戶的重要行爲失敗時，用戶與組件介面應當有能力弄清楚：某些資料是否已被刪除？某項計算是否未被執行？

一旦程式到達穩定狀態，它應當盡最大努力繼續執行。理想情況下它應當在「降級模式」下繼續執行，提供儘可能多的功能，忽略較不重要而又嚴重消耗記憶體的功能（圖 1.7）。你可以提供一系列「逐漸降級模式」，應付記憶體逐漸減少的窘境。組件可以許多方式實作降級模式，例如對客戶隱藏記憶體耗盡情況；抑或先接受申請嗣後再處理；抑或提供一個品質較低的服務。舉個實例：處於「記憶體耗盡」（out of memory）狀態下的 Word-O-Matic 語音輸出模組，能夠接受客戶命令，但忽略不做任何回應。這種作法使編寫客戶程式得以簡單許多。

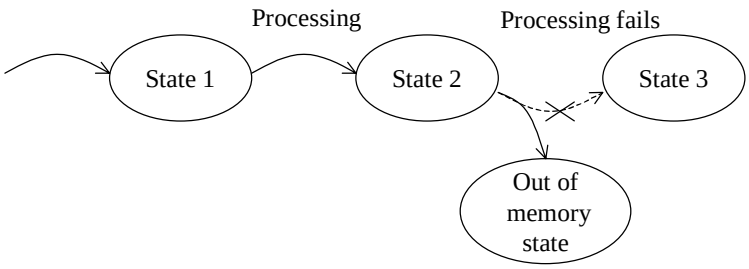


圖 1.7 失敗後進入記憶體耗盡狀態

一旦有更多可用記憶體時，系統應當恢復所有功能。系統與外部沉重負載苦苦纏鬥時，常會出現記憶體短缺；一旦負載消失，記憶體需求就會下降。用戶直接決定了多工環境如 MS Windows 和 EPOC 下的工作負載，所以他們可以關閉某些系統應用程式以釋放記憶體。執行於「降級模式」下的組件應當定期嘗試恢復完全功能，充分利用任何新增的可用記憶體。

舉個例子，當 Word-O-Matic 未能成功配置某一文件語音輸出所需的記憶體時，其顯示幕仍可正常運作。如果文字檢查器失敗，Word-O-Matic 就不再以高亮度突顯拼字錯誤；如果浮動視窗失敗，它就不再出現，但程式的其餘部分繼續執行。是的，錦上添花的特性並非不可或缺，多數用戶對於「只有文字顯示並能夠輸入更多文字」還是相當滿意的。

結果（Consequences）（譯註：本段保留之 *Italic Times New Roman* 字型均爲 forces 名稱）

Partial Failure 可以顯著提高程式的可用性（*usability*）。透過謹慎的設計，即使降級模式也能提供用戶完成工作所需的足夠基本功能。藉由「保證程式在一定數量的記憶體中可以繼續執行」，*Partial Failure* 可減少程式的最小記憶體需求（*memory requirements*），提高記憶體需求的可預測性（*predictability*）。

支援 *Partial Failure* 可以提高程式的設計品質（*design quality*）— 如果您能夠針對記憶體耗盡支援 *Partial Failure*，那麼針對網路錯誤或其他資源耗盡等問題而支援 *Partial Failure*（以及其他形式的失敗處理）也很容易。對於 *Partial Failure* 有著適度支援的系統，幾乎都是可信賴的（*reliable*）系統。

然而：*Partial Failure* 是件編程苦差事，需要良好的編程素養（*programmer discipline*），還需耗費大量編程心力（*programmer effort*）方可實現。

由於程式員必須提供「替代控制路徑」，並在失敗時釋出資源，所以「支援 *Partial Failure*」的語言機制 — 異常（exceptions）及類似手段 — 大大提高了系統的實作複雜度。

由於區域（*local*）事件（亦即「記憶體耗盡」）有可能經由影響系統中的其他模組而產生全域（*global*）後果，所以 *Partial Failure* 有可能增加系統的全域（*global*）複雜度。

支援 *Partial Failure*，會顯著增加每個模組的複雜度（因爲必須測試所有失敗模式），進而提高測試成本（*testing cost*）。

❖ ❖ ❖

實作（Implementation）

實作 *Partial Failure* 時請考慮以下問題：

1. 偵測記憶體耗盡

偵測記憶體耗盡的方法視你所使用的 *Memory Allocation*（219）類型而定。如果你從 heap 配置記憶體，物件生成函式會有某種偵測「配置失敗」的機制。如果你親自管理記憶體配置，例如使用 *Fixed Allocation*（226）或從 *Pool*（251）動態配置物件，那麼你就得保證程式得以檢查並確定你所使用的固定結構或 memory pool 何時已滿。*Memory Allocation* 一章將詳細討論這個問題。

至於程式內如何交流「記憶體耗盡」訊息，則仰賴編程語言提供的設施（facilities）。許多語言，包括 C 和 C++ 早期編譯器，這種錯誤的唯一通告方法是傳回一個錯誤碼（而非配製而得的記憶體）。不幸的是，每次都能細心檢查配置動作的傳回值，需要非常高水準的編程素養。如今有更多語言支援「異常」的變體，明確允許函式非正常地回返。大部份編程環境對於「記憶體不足」異常（out-of-memory exception）的預設處理辦法是將程式終止，所以實作 *Partial Failure* 的組件必須處理這些異常。

2. 到達安全狀態

一旦發現記憶體耗盡，你就必須確定如何到達安全狀態，也就是說你必須確定究竟系統中的多大部分將因爲得不到所需記憶體而無法繼續執行。典型情況下你只需拒絕提出申請的函式，其他情況下組件可能需要一個降級模式，抑或者對一個獨立程式來說，可能得徹底結束。

爲確定系統的多大部分無法恢復安全狀態，你得依次檢查每個組件，考慮其不變量（invariants），也就是考慮「爲了使能成功執行，哪些狀態必須保留（Hoare 1981; Meyer 1997）」。如果組件的不變量不受記憶體耗盡的影響，那麼組件就能夠繼續執行。如果不變量受記憶體配置失敗的影響，「刪除或改變組件內的其他資訊」也許能使組件回復協調狀態。

如果你無法將組件恢復至某個安全狀態，你就不得不關閉它。

如果你不得不捨棄整個應用程式，也許你還可以運用 *Application Switching*（84）獲得一個安全狀態。

3. 釋出資源（Releasing resources）

爲保證不留下任何副作用，凡未能配得記憶體的組件，稍後都必須完成清理工作：必須釋出所有配得但無法再使用的資源（尤其是記憶體），必須將其狀態（以及其他所有受影響的組件的狀態）恢復至其不變量原值。

C++ 的異常在 `throw` 述句和 `catch` 述句之間進行 stack unwind（堆疊輾轉開解, Stroustrup 1997）。預設情況下二者之間的所有 stack-based pointers 都會丟失，它們擁有的全部資源都將無人照管。C++ 異常保證會喚起 stack-based object 的解構式，於是 stack 上頭的所有物件都可以在解構式中完成清理工作。因此異常發生期間，各物件都能夠被清除乾淨。標準的 template class `auto_ptr` 內部包裝了一個指標，並在 stack unwound（堆疊輾轉開解）時刪除該指標：

```
auto_ptr<NetworkInterfaceClass> p(new NetworkInterfaceClass);
p->doSomethingWhichCallsAnException(); // the instance is deleted
```

儘管 Java 擁有垃圾回收機制，stack unwind 時你還是得釋放物件（實作手法是：移除物件的 references），並釋出外部資源。Java 沒有解構式，無論 `try` 區塊退出是否正常，其 `try.finally` 模型都會執行 `finally` 區塊。下例在集合中註冊一個 *Command*（Gamma et al. 1995）子類別實體，當擲出異常或 *Command* 的 `execute` 函式正常回返時再將它移除：

```
Command cmd = new LongWindedCommand();
setOfActiveCommands.add(cmd);

try {
   cmd.execute();
}
finally {
   setOfActiveCommands.remove(cmd);
}
```

身爲一個記憶體受限的作業系統，EPOC 將 *Partial Failure* 作爲其最基本架構原則之一（Symbian 1999a）。實際上任何操作都可能因記憶體耗盡而失敗，但這類失敗被儘可能限制起來，永遠不允許引發記憶體洩漏。EPOC 的 C++ 環境不使用 C++ 異常，改用作業系統提供的 TRAP 模型。基本上呼叫 `leave()`便會進行 stack unwinds（堆疊輾轉開解）（利用 C `longjmp()`函式），直到遇上一個 "TRAP harness call"。客戶碼明確地由 cleanup stack 加入或移除資料項，然後 `leave()`自動爲每一個儲存於 cleanup stack 上的物件呼叫

`cleanup()`。最上層的 EPOC 系統排程器爲所有正常用戶碼提供了一個 TRAP harness。預設情況下會彈出一個對話框，警告用戶操作失敗，然後繼續執行。

這裡有一個 EPOC 安全物件構建實例（Tasker et al. 2000）。*Factory Method*（Gamma et al. 1995）`NewL()`運用 `new(Eleave)`配置一個以 0 填充的（安全的）物件，然後呼叫下一個函式 `ConstructL()`執行一切有可能失敗的操作。藉由「將未初始化的物件壓入 cleanup stack」手法，如果 `ConstructL()`失敗，物件便會被自動刪除。一旦新物件完全建立起來，就可以把它從 cleanup stack 中移走了：

```
SafeObject* SafeObject::NewL(CEikonEnv* aEnv) {
   SafeObject* obj = new (ELeave) SafeObject(aEnv);
   CleanupStack::PushL(obj);
   obj->ConstructL();
   CleanupStack::Pop();    // obj is now OK, so remove it
   return obj;
}
```

Captain Oates（57）範式內含另一個 EPOC cleanup stack 實例。

4. 降級模式（Degraded modes）

記憶體配置失敗後，只要清理工作一完成，你的程式就應當在穩定狀態下繼續執行，儘管其效率將降低。例如：

- 載入某種字型時可能失敗；這種情況下你可以選用標準系統字型。
- 顯示某張圖片時可能失敗；你可以留下空白或顯示一條訊息。
- 快取裝置（cache）的內容也許不可用；你可以花一些時間取得原值。
- 詳細的計算（detailed calculation）可能失敗；你可以使用近似值。
- Undo 資訊可能無法保存（通常在發出警告之後）。

無論何時何處，只要可能，組件就應當對客戶隱匿 *partial failure*。這樣的封裝使組件容易設計，並且將失敗後果限制於失敗組件內。儘管組件介面可以爲感興趣的客戶提供附加函式用以「認識這些失敗」，但它不應強迫客戶知悉這些失敗。

你可以經常使用 *Multiple Representations*（209）來協助實現 *partial failure*。

5. 未雨綢繆

將欲取之，必先與之。賺錢之前你得先花點錢。處理記憶體耗盡，本身也需要記憶體。C++ 和 Java 透過異常的丟擲來通知記憶體耗盡，這需要額外記憶體來儲存異常物件；「顯示對話框以警告用戶」也需要額外記憶體儲存對話框物件。因此你應當預留一些記憶體以備不測；C++ 執行期系統也得預先配置足夠的記憶體用來儲存「記憶體耗盡」狀態時需

要用到的 bad_alloc 異常（Stroustrup 1997）。同樣道理，Windows CE 也得爲「記憶體不足」對話框預留足夠記憶體（Boling 1998）。Prograph 視覺編程語言採用一種更複雜的方法：提供一個 "rainy day fund" class（未雨綢繆類別）用來管理記憶體儲備，一旦主記憶體耗盡，這些儲備記憶體將立即被釋出（MacNeil and Proudfoot 1985）。

示例（Examples）

以下 Java 程式碼演示了一項運用 *Partial Failure* 處理錯誤的簡單技術。StrapFont.font() 嘗試找出一種字型並保證將它載入主記憶體。從客戶的角度看，它必須每次都成功，彈無虛發。

我們藉由「保證每次總有一種字型傳回」來實現某種安全狀態。在這裡，class 初始化時首先產生一種預設字型。如果操作失敗，會造成行程（process）初始化失敗，new 會丟擲一個此處並未捕捉的 OutOfMemoryError 異常，阻止用戶在這種情況下輸入資料：

```
class StrapFont {
    static Font myDefaultFont = new Font("Dialog", Font.PLAIN, 12);

    public static Font defaultFont() {
         return myDefaultFont;
    }
```

以下的 StrapFont.font()試圖根據 privateGetFont()產生新字型物件，而後者可能耗盡記憶體，擲出 OutOfMemoryError 異常。如果新字型物件沒能被成功產生出來，我們就傳回預設字型。這套機制也可以用來對不同的問題（例如字型不存在）進行安全處理：

```
    public static Font font(String name, int style, int size) {
         Font f;
         try {
              f = privateGetFont(name, style, size);
         }
         catch (BadFontException e) {
              return defaultFont();
         }
         catch (OutOfMemoryError e) {
              return defaultFont();
         }
         return f;
    }
}
```

每次客戶重繪螢幕時，都必須使用 `StrapFont.font()`重新載入字型，而不是將先前的傳回值以快取方式儲存起來；這就保證一旦記憶體充足必能載入正確字型。

❖ ❖ ❖

已知應用（Known uses）

Partial Failure 是一項重要的架構原則（architectural principle）。如果系統打算支援 *Partial Failure*，就必須一以貫之、鍥而不捨。最近有個專案，評估將一個第三方（third-party）資料庫程式庫移植到 EPOC 上成爲作業系統服務之一。看起來似乎萬事如意，因爲程式碼十分優雅，移植只是舉手之勞。不幸的是該程式庫當初是爲擁有大量記憶體的系統而設計，不支援 *Partial Failure*；所有記憶體配置動作要嘛成功，要嘛終止行程。然而在「供應許多 EPOC 應用程式同時使用」的服務中這種策略是絕對不可行的；EPOC 耗盡記憶體是很平常的事，系統設計者絕不容許出現「導致許多應用程式同時失敗」的情況發生。由於這個程式庫不支援 *Partial Failure*，所以它不適合 EPOC。

降級模式（Degraded modes）在 GUI 應用程式中很常見。如果 Netscape 因記憶體不足而載入字型失敗，它就使用標準字型繼續執行。Microsoft PowerPoint 則會使用標準字型並忽略圖像。PhotoShop 會先警告用戶，然後停止保存 undo 資訊。

讓我們到更低層面看看。如果 Microsoft Foundation Class（MFC）framework 在繪製視窗時偵測到異常，預設行爲是將視窗標記爲「已全部畫完」。雖然視窗的顯示可能因而不正常，但應用程式可以繼續執行；將來應用程式改變視窗內容時，視窗會被重新繪製。

EPOC 的文書處理器（word processor）在編排頁面時使用的記憶體最多。如果失敗，它就進入「記憶體不足」模式，將編排完畢之文字內容儘可能顯示出來。只要一發生用戶事件（例如捲動或重新顯示），文書處理器就會重新編排頁面，如果成功則脫離降級模式。EPOC 的架構還有一項很有趣的安全狀態策略：EPOC application framework 是事件驅動（event-driven）式的，任何應用程式都仰賴中央排程器（central scheduler）反覆呼叫自己而獲得執行。如果應用程式當下未被排程器執行，就是處於安全狀態，所以任何 EPOC 應用程式的失敗絲毫不影響其他程式（Tasker et al. 2000）。

參見（See Also）

Application Switching（84）可以捨棄某應用程式並執行起另一個應用程式，不必犧牲由多個應用程式組成的完整系統。*Multiple Representations*（209）也支援 partial failure，作法是以更高效更經濟的記憶體運用設計來替代標準表述方式。

另一種令記憶體索求者（某個組件）失敗的方法是 *Captain Oates*（57），捨棄比較不重要的組件。*Memory Allocation*（219）一章講解配置失敗的許多處理策略，例如推延請求、丟棄資訊、通知錯誤等等。

Ward Cunningham 的 *Checks* 範式語言討論了數種向用戶通告 *Partial Failure* 的方法（Cunningham 1995）。*Professional Symbian Programming*（Tasker et al. 2000）、*More Effective C++*（Meyers 1996）和 *Exceptional C++*（Sutter 2000）詳細講述運用「C++ 異常」實作出 *Partial Failure*（48）的編程技術和慣用手法。

Captain Oates（犧牲小我）

另名：Cache Release

如何滿足對記憶體的最重要需求？

- 許多系統都有執行於背景（後台, background）的組件。
- 許多應用程式爲了提高性能，會以快取方式（cache）存放資料。
- 與「系統爲了自身而運轉」的背景活動相較，用戶更關心自己的工作。

對作業系統而言，所有記憶體需求都是平等的。但是對用戶來說，某些需求比其他需求更重要（Orwell 1945）。

例如，當人們使用 Strap-It-On PC 文書處理器來編輯文件時，他們並不關心背景的碎形（fractal）畫面。你可以將稀罕資源投資於用戶真正需要的目標，用以提升系統的可用性（*usability*）。

許多系統包含背景組件，例如螢幕保護程式、聊天程式、密碼分析引擎（Hayes 1998），或是搜尋地球外智慧生命的傅立葉分析程式（Sullivan et al. 1997）。系統也會運用記憶體來讓用戶的操作更迅捷、更有趣味，例如下載音樂、快取 web 頁面、爲檔案系統建立索引等等。儘管從長遠觀點來看這些行爲都很重要，但當它們出現並從用戶端十萬火急、攸關生死的記憶體申請處取走稀有資源時，它們確實沒什麼幫助。

因此：你應該犧牲非絕對必要之組件所使用的記憶體，以免抗拒更重要的任務。

當記憶體消耗殆盡，尚存一些剩餘空間時，請你警告系統內的所有組件。組件收到警告後，應當釋放它所使用的「非至關重要」的記憶體，或是在更極端的情況下終止操作。

如果系統不支援「記憶體狀況通告」，行程（process）可透過規律性輪詢（regular polling）和空閒記憶體追蹤，於記憶體短缺時釋放（或關閉）「非至關重要」的資源。

例如當 Word-O-Matic 的記憶體即將耗盡時，IP networking stack 會清空 IP address map 的快取裝置，web 瀏覽器也會清空其頁面快取裝置。"Fizzy" 碎形生成器之類的背景服務行程會自動關閉。於是文書處理器的記憶體需求就能夠獲得滿足。圖 1.8 和圖 1.9 演示了 *Captain Oates* 系統層面上的一份實作。

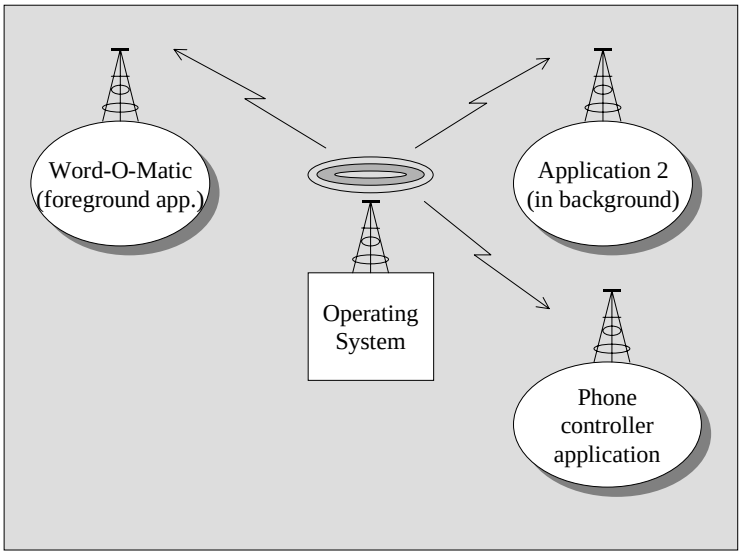


圖 1.8　momery-low（記憶體短缺）事件

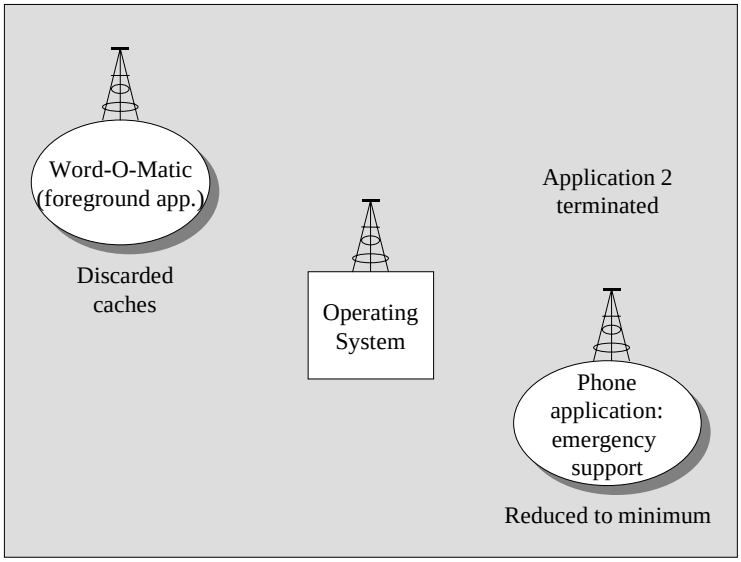


圖 1.9　momery-low（記憶體短缺）事件之結果

本範式的名稱用來頌揚維多利亞女王時代的一位知名探險家：Captain Lawrence 'Titus' Oates。Oates 是 Robert Falcon Scott 領導的不列顛探險隊一員，他到達南極後發現 Roald Amundsen 率領的挪威探險隊已經最先抵達。Scott 的探險隊返程途中出現補給短缺狀況，心情沮喪又患了凍瘡的 Oates 爲了給其他隊員創造生存機會，犧牲了自己，他走進暴風雪之中，留下了知名的一頁日記：I just going outside and may be some time（我出去轉轉，可能得一會兒）。Oates 的犧牲未能拯救探險隊其他成員，次年他們的遺骸在冰封的宿營地中被發現，連同 35 公斤的岩石標本，艱難地從極地返國（Limb and Cordingley 1982; Scott 1913）。

結果（Consequences）（譯註：本段保留之 *Italic Times New Roman* 字型均爲 forces 名稱）

本範式爲記憶體需求最殷切的地方配置記憶體，因而提高系統的可用性（*usability*），降低記憶體需求量（*memory requirements*）。這個範式會釋出臨時記憶體，因而得以提高系統記憶體用量的可預測性（*predictability*）。

然而：爲了自願釋出資源，*Captain Oates* 需要良好的編程素養（*programmer discipline*）。*Captain Oates* 對於其直接實作者通常沒什麼好處，所以開發團隊比較少有實作意願 — 得有強大的文化或架構推動力才能促使開發團隊願意實作這個範式。爲了實作和測試，程式員需要付出很大心力（*programmer effort*）。

Captain Oates 會在原本並無關連的組件之間引入耦合（coupling）關係，降低系統的可預測性（*predictability*）。釋出資源並且會降低程式的時間效率（*time performance*）。程式應當通過檢測來證明它們確實釋出了資源而後還能繼續執行。由於許多程式都必須處理記憶體不足訊號，所以 *Captain Oates* 在作業系統的支援下比較容易實作。這是又一個「導入 *local complexity*（區域複雜度）來處理訊號」的全域機制（*global mechanism*）。

❖ ❖ ❖

實作（Implementation）

Captain Oates 的主旨是：爲了讓高優先權活動可以繼續，釋出低優先權活動所用的記憶體。如果某個組件正在爲高優先權活動服務，就不該讓它釋出記憶體。但「記憶體不足」偵測機制往往不分青紅皂白地一律通知所有組件，我們該如何找出應犧牲的組件？

通常，UI 應用程式都能夠確定自己是否爲現行（current）應用程式 — 也就是說它是否擁有輸入（鍵盤）焦點因而可與用戶交談（圖 1.10）。果真如此，任何程式如果收到「記憶體不足」警訊，就不應該犧牲自己。

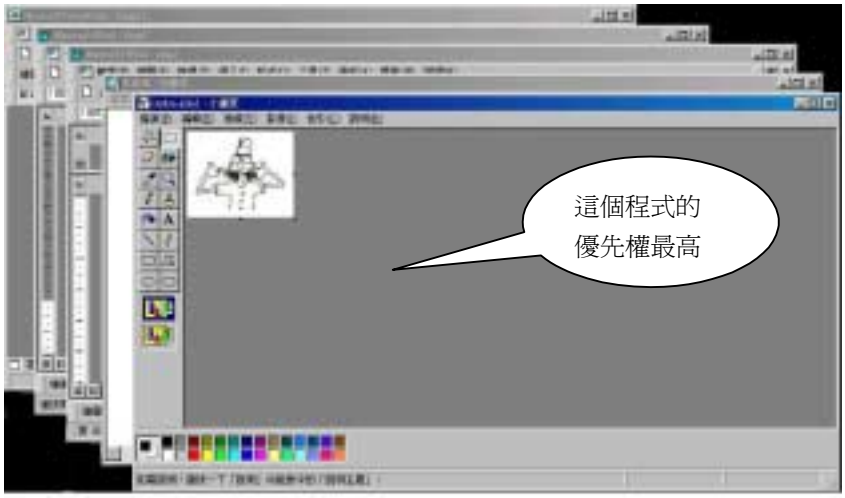


圖 1.10 現行（current）應用程式

背景行程（background process）往往無法向系統詢問自己的重要性。在 MS Windows 中，高優先權執行緒在等待某些事件時會形成阻塞（block）— 例如任務管理器（Task Manager）等待 Ctrl+Alt+Del 按鍵時擁有高優先權。然而當任務管理器偵測到某個事件發生後，它會把自身優先權降至普通級。所以，呼叫 `GetThreadPriority()`並不能準確顯示任務的重要性以及它是否正被使用。

然而大部份行程都可以透過其他資訊確定自身的重要性。例如，網絡連接管理組件能夠檢查自己是否擁有主動連接（active connections）。其他背景行程可能根本無從獲得那些資訊；又例如 web 頁面快取裝置可能對它所支援的應用程式一無所悉。這些行程絕不能直接與用戶交談（否則它們將可以瞭解更多的用戶行爲資訊），因而可以於必要時頗爲安全地釋出無關緊要的資源。

1. 偵測記憶體不足（low-memory）

記憶體可用量很低時，許多作業系統都會發佈事件（event）用以警告應用程式。MS Windows 和 MS Windows CE 會向所有視窗（不含背景行程）送出（send）`WM_COMPACTING` 和 `WM_HIBERNATE` 訊息，警告它們可用的系統記憶體數量正在降低（Boling 1998; Microsoft 1997）。記憶體數量很低時，某些作業系統或語言執行期程式庫（language runtimes）並不發送事件，而是回呼（call back）系統組件 — 請參見 *Partial Failure*（48）中的 C++ `new_handler` 的例子。

另一種方式是，如果系統提供一個函式可以顯示當前記憶體用量，組件就可以輪詢（poll）查看記憶體數量是否很低。如果確實如此，就釋出記憶體。由於處理器的活動會消耗電能，所以在電池供電機器中，輪詢（poll）的表現恐怕不能儘如人意。

2. 處理記憶體不足（low-memory）

記憶體不足事件發生時，如果每個組件都能夠確定系統記憶體的短缺程度，將會很有幫助。在 Java JDK 1.2 環境中，執行期物件（runtim object）的 `getMemoryAdvice()`會傳回以下四種模式之一：「綠」代表不缺乏，「黃」到「橙」代表記憶體數量正在降低，「紅」代表記憶體嚴重匱乏。MS Windows 的 `WM_COMPACTING` 事件會送出一個數值，表示「花費在記憶體分頁（paging）動作上」的時間比例：1/8 相當於「黃」，此時首度發出訊息；任何超過 1/4 的值都代表嚴重匱乏（Microsoft 1997）。

3. 良民（Good citizenship）

最簡單而且通常最容易的辦法就是：每個行程（process）自願放棄未真正用上、無關緊要的資源。只要監視一個單純的計時器，無論系統其餘部分狀態如何，你都可以在一段特定時間後釋出休眠的資源。例如 EPOC web 瀏覽器會動態載入一些 DLLs 用以處理特定類型之 web 資料，如果某個特殊類型的資料出現過一次，它很可能會立即再次出現，因此 DLL 載入器將上述每一個 DLL 以快取（cache）方式儲存起來。如果數秒鐘內 DLL 未被再次使用，載入器便將它釋出。

示例（Examples）

下面這個 C++ 示例實作出作業系統基礎建設的一小片段，支援 EPOC 作業系統所具備的 *Captain Oates* 簡單機制。*Captain Oates* 管控程式執行於背景，當記憶體數量太低，就關閉非現行的應用程式。由於關閉 EPOC 應用程式會自動保存該程式的狀態（這是 PC 同步機制的一項要求），所以這麼做不會丟失任何資料。但瞬間編輯狀態（Transient editing states，例如文件中光標所在位置或顯示於檔案瀏覽器上的當前項目）並未保留下來。

全部功能包含於 `COatesTerminator` class 中，如下所示（其中省略函式宣告）：

```
class COatesTerminator : public CBase {
private:
   RNotifier iPopupDialogNotifier;       // Provides user screen output
   CPeriodic* iTimer;                    // Timer mechanism
   CEikonEnv* iApplicationEnvironment;   // User I/O Handler for this app.

   enum {
        EPollPeriodInSeconds = 10,       // How often to check memory
        EDangerPercentage = 5 };         // Close applications when less free
   };                                    // memory than this.
```

class 內有很多設置「定時器」和「對話框通知器」的建構函式和初始化函式（此處未含）。

應用程式的核心 `TimerTickL()`函式會輪詢（polls）當前記憶體狀態，如果數量很低就關閉程式。如果其他應用程式所配置的記憶體數量比當前正在使用的多，則「空閒記憶體檢查器」所顯示的「記憶體數量稀少」可能並不真確。如果第一次檢查顯示記憶體很少，我們就壓縮所有 heaps；這可收回位於每個 heap 末端的空閒記憶體。第二次檢查可以精準估量所有空閒記憶體。如果第二次檢查結果仍然很低，我們就呼叫 `CloseAnApplicationL()`，將應用程式關閉：

```
void COatesTerminator::TimerTickL() {
   if ( GetMemoryPercentFree() <= EDangerPercentage ||
            (User::CompressAllHeaps(),
            GetMemoryPercentFree() <= EDangerPercentage )) {
        CloseAnApplicationL();
   }
}
```

`CloseAnApplicationL()`必須首先選擇一個適當的（可關閉的）應用程式 — 我們不希望關閉現行（前景）應用程式、系統外殼程式（shell）、以及本行程。我們關閉的是候選應用程式中 Z-order 最低者。是的，應用程式被系統劃分爲不同的「視窗群組」（Window groups, WG），爲找到合適的視窗，我們首先取得不想被關閉之視窗群組的辨識符號（`focusWg,defaultWg,thisWg`），然後取得 `WindowGroupList`，再沿此一 list 回溯，關閉我們找到的第一個合適的應用程式。

請注意 *Partial Failure*（48）中講述的 `CleanupStack()`的用法。一旦將持有 `WindowGroupList` 的 array 配置完畢後，我們立即將它推入 stack，並在函式結束時將它移除並摧毀。如果「取得視窗群組」的呼叫出現問題，我們立刻離開 `CloseAnApplicationL` 函式，位於 cleanup stack 中的 array 會被自動摧毀。

```
void COatesTerminator::CloseAnApplicationL() {
   RWsSession& windowServerSession =
        iApplicationEnvironment->WsSession();
TInt foregroundApplicationWG =
        windowServerSession.GetFocusWindowGroup();
   TInt systemShellApplicationWG =
        windowServerSession.GetDefaultOwningWindow();
   TInt thisApplicationWG =
        iApplicationEnvironment->RootWin().Identifier();
   TInt nApplications=windowServerSession.NumWindowGroups(0);
   CArrayFixFlat<TInt>* applicationList =
        new (ELeave) CArrayFixFlat<TInt>(nApplications);
   CleanupStack::PushL(applicationList);
   User::LeaveIfError(
        windowServerSession.WindowGroupList(0,applicationList));
   TInt applicationWG = 0;
```

```
    TInt i= applicationList->Count();
    for (i --; i>=0; i --) {
         applicationWG = applicationList->At( i );
         if (applicationWG  != thisApplicationWG &&
                   applicationWG != systemShellApplicationWG &&
                   applicationWG != foregroundApplicationWG)
              break;
         }
    }
```

如果找到了合適候選者，我們就以標準機制乾淨利落地結束它。請注意 `_LIT` 定義了一個儲存於 ROM 的字串字面常數 — 請見 *Read-Only Memory*（65）範式。

```
    if (i >= 0) {
         TApaTask task(windowServerSession);
         task.SetWgId(applicationWG);
         task.EndTask();
         _LIT(KMessage, "Application terminated");
         iPopupDialogNotifier.InfoPrint( KMessage );
    }
    CleanupStack::PopAndDestroy(); // applicationList
}
```

這種實作法的缺點在於需要輪詢（pooling），因而非必要地消耗了 CPU 時間和電池能源。更好的實作法是僅在寫入 RAM-based 檔案系統之後（這很直接明瞭）或在用戶輸入之後（這比較艱難）才進行輪詢（poll）；抑或根據可用記憶體的數量改變輪詢頻率。

❖ ❖ ❖

已知應用（Known uses）

MS Windows 附帶一套由波士頓 ObjectPLUS 公司出品的超媒體（hypercard）應用軟體，名爲 ObjectPLUS，它會對 `WM_COMPACTING` 訊息做出回應。當記憶體短缺越來越嚴重時，它會：

- 停止播放聲音；
- 壓縮圖像；
- 將由資料庫取得的 cached bitmaps 刪除。

這種做法對系統中的其他應用程式有好處。「因爲其他更重要的活動而釋出記憶體」也使 HyperCard 應用程式本身獲益良多。透過「在 Windows 事件處理常式（event handler）中實現這種行爲」，設計者得以令該行爲與應用程式中的其他處理有著架構層次（architecturally）上的分離。

Apple Macintosh 的記憶體管理器（*Compaction* 中另有討論）支援「可清理的記憶體區塊」，一旦記憶體匱乏，記憶體管理器可以回收這一類區塊（Apple 1985）。它們用於資源檔（Resource Files）、檔案系統快取裝置（file system cache），以及動態配置的程式記憶體。

MS Windows CE Shell 採用兩段法（two-phase approach）管理記憶體（Microsoft 1998; Boling 1998）。當記憶體數量很低時，它向所有應用程式發送（sends）`WM_HIBERNATE` 訊息；CE 應用程式會以「儘可能釋出系統資源」的行動回應該訊息。如果記憶體進一步減少，Shell 便向優先權最低的應用程式發送（sends）`WM_CLOSE` 訊息，要求它們關閉（和 EPOC 一樣，Windows CE 要求應用程式收到 `WM_CLOSE` 後不必提示用戶就直接保存狀態）。一旦系統有了更多可用資源，應用程式可以收到 `WM_ACTIVATE` 訊息，讓它們重建「接收 `WM_HIBERNATE` 時丟棄的」內部狀態。

大量分散式網際專案將 *Captain Oates* 用於螢幕保護程式（screen saver）。計算機被使用時，螢幕保護程式並不執行；計算機閒置數分鐘後，螢幕保護程式開始利用空閒的處理器功率搜尋外部訊息（Hayes 1998），或破解加密訊息（Sullivan et al. 1997）。

參見（See Also）

Captain Oates（57）描述系統中其他行程耗盡了記憶體時，程式該怎麼做。*Partial Failure*（48）描述的則是行程本身耗盡了記憶體時該怎麼做。許多用在 *Partial Failure* 上的技術（例如 *Multiple Representations,* 209 和 *Application Switching,* 84）也適用於 *Captain Oates*。

Fixed Allocation（226）描述 *Captain Oates* 範式的一種簡單實作法，其中每一個「活動」（activity）都僅僅只是個資料結構，僅僅只是「以新易舊」— 將新活動覆蓋於舊活動之上。

Scott 和其探險隊是大不列顛和紐西蘭文化中家喻戶曉的英雄人物。請參看 *Captain Oates : Soldier and Explorer*（Limb and Cordingley 1982）以及 *Scott's Last Expedition : The Personal Journals of Captain R. F. Scott, R.N., C.V.O., on his Journey to the South Pole*（Scott 1913）。

Read-Only Memory（唯讀記憶體）

另名：Use the ROM（充份運用 ROM）

如何處置唯讀（read only）的程式碼和資料？

- 許多系統既提供唯讀（read-only）記憶體也提供可寫（writable）記憶體。
- 唯讀記憶體比可寫記憶體便宜。
- 程式通常並不修改可執行碼（executable code）。
- 程式不修改資源檔（resource files），檢索表（lookup tables）以及其他預先初始化過的資料（pre-initialized data）。

程式通常包含許多唯讀程式碼和資料。例如 Word-O-Matic 文書處理器（word processor）有大量可執行碼和永不改變的大型主字典檔。如果把這些靜態資訊都儲存於主記憶體內，將會從「確確實實需要改變」的資料手中奪走可用的記憶體，因而總體上提高了程式的記憶體需求（*memory requirements*）。

許多硬體設備 — 尤其是小型設備 — 既支援唯讀記憶體也支援可寫記憶體。前者有可能是可從中央處理器（processor）直接存取的主儲存體，也可能是需間接存取的次儲存體。許多硬體技術都能夠提供唯讀記憶體，包括各種 ROMs、PROMS、flash ROMs、read-only compact discs，甚至包括 paper tape。其中多數形式在許多方面都優於相應的可寫記憶體 — 製造更簡單、價格更低廉、更可靠、更節約能源、散發的熱量更少、抗宇宙游離輻射的能力更強。

因此：最好將唯讀的程式碼和資料儲存於唯讀記憶體內。

把系統程式碼和資料分成「可改變」和「恆久不變」兩部分。把不變的部分儲存於唯讀記憶體內，執行期間將它們與可變部分再關聯（re-associate）起來。

例如，Word-O-Matic 程式碼內含於 Strap-It-On 腕型 PC 的 ROM 記憶體中。Word-O-Matic 的主字典（master dictionary）和其他資源檔儲存於唯讀的次儲存體（採用 flash ROM）；只有用戶文件和組態檔（configuration files）儲存於可寫記憶體內。

結果（Consequences）（譯註：本段保留之 *Italic Times New Roman* 字型均爲 forces 名稱）

本範式將可寫的主儲存體（writable main storage）替換爲唯讀儲存器（read-only storage），因而降低主儲存體的記憶體需求（*memory requirements*），並使測試更容易。從金融成本、電力消耗、可靠性三方面來看，唯讀儲存器比可寫儲存器更便宜。如果系統能夠從唯讀記憶體中直接執行程式，那麼使用唯讀記憶體還可以降低系統的啓動時間（*start-up time*）。

雖然你可能必須從唯讀次儲存器（*read-only secondary storage*）身上複製程式碼和資料到主記憶體，但你從主記憶體中刪除唯讀資訊時卻不必把它存回次儲存體。由於不能修改唯讀程式碼和資料，所以它們可以輕鬆地在程式或組件之間實現共享，進一步從總體上降低系統的記憶體需求（*memory requirements*）。

然而：將程式分爲唯讀和可寫兩部分，需要付出編程心力（*programmer effort*），而爲了維繫這種劃分，還需要良好的編程素養（*programmer discipline*）。儘管唯讀資訊和可寫資訊得由程式中每個組件區域性地（*locally*）實現，但從根本上來說，二者的區分卻是個全域性的（*global*）關注點。

唯讀記憶體中的程式碼或資料，比可寫的次儲存器（writable secondary storage）內的資訊更難以維護（*maintain*）。通常，將儲存於唯讀記憶體內的程式碼或資料替換出來的唯一辦法是實際更換儲存硬體。一般來說，可擦除、可覆寫的 flash memory（快閃記憶體）的更新程序很複雜，如果作業系統儲存於正被更新的記憶體中，那就更複雜了。

❖ ❖ ❖

實作（Implementation）

生成 ROM image（用以儲存「唯讀記憶體內之最終程式碼和資料」的拷貝）是個不折不扣的神奇過程，需要一流的咒語和詭秘的軟體因素 — 後者隨著你的環境而異。然而，使用唯讀記憶體時，大部份環境還是有許多共同之處。

1. 儲存可執行碼

如果你直接從唯讀記憶體執行程式，你就可以用它來儲存可執行碼。這通常引發兩個問題：如何在唯讀記憶體中表示程式碼？程式碼如何存取所需資料？

多數環境將程式儲存爲目的檔（object files），例如可執行檔和動態連結程式庫。它們並不指涉（refer to）記憶體中的絕對位址，而是在內部包含指向其他檔案或程式庫的符號式參考（symbolic references）。在目的檔（object files）可被執行之前，作業系統的執行期載入器（runtime loader）必須先繫結這些符號式參考，生成完全可執行的機器碼。

爲了將可執行碼儲存於唯讀記憶體內，你需要一個輔助工具：ROM 生成器（ROM builder），它將完成執行期載入器的工作，讀入目的檔，產生 ROM image。ROM 生成器賦予每個目的檔一個記憶體基礎位址（base address），然後將目的檔內容拷貝至 ROM image 的相應位置上，繫結（綁定）符號式參考（symbolic references），爲 heap memory、static memory 和 static data 指派可寫記憶體。EPOC 系統就包含了一個 Java ROM 生成器，它模仿 Java 執行期類別載入器的行爲，接受 .jar 和 .class 檔，並將這些檔案載入 ROM image 內。

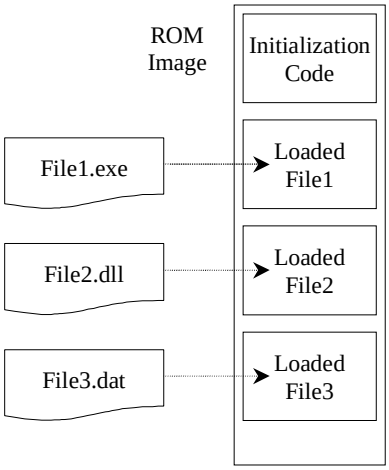


圖 1.11　ROM 映像

如果系統經由「執行唯讀記憶體內的程式碼」而啓動，那麼 ROM image 中還應包含「配置主記憶體資料結構」和「引導（bootstrap）整個系統」的初始化程式碼（initialization code，圖 1.11）。ROM 生成器知道有這麼一份引導碼，會將它安裝於正確位置上。

2. 在程式碼中包含資料

多數程式和編程語言既包含常量資料（constant data）又包含可執行碼。如果程式碼儲存於唯讀記憶體內，那麼常量資料應當相伴左右。爲實現此一目標，你得讓編譯器或組譯器（assembler）相信，那些資料確實不會改變。

例如，依照 C++ 標準規格（Ellis and Stroustrup 1990）定義，物件實體可以位於程式碼節區（code segment）內（於是乎也就位於唯讀記憶體內），前提是：

- 該實體被定義爲 `const`，而且
- 它沒有建構式（constructor）和解構式（destructor）。

於是：

```
const char myString[] = "Hello";    // In ROM
char* myString = "Hello";      // Not in ROM according to the Standard.
const String myString("Hello"); // Not in ROM, since it has a constructor
```

更明確地說，你可以生成能夠被編入 ROM 的 C++ 資料表（data tables）：

```
const int myTable[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }; // In ROM
```

請注意由於 non-const C++ 字串可以修改，因此一般不把它們置於程式碼節區中。不過某些編譯器支援「用以改變此一行爲」的某些旗標或 #pragma 宣告。

EPOC 系統使用 C++ macros 和 template classes 的混合體，在唯讀記憶體中生成字串實體，內含文本長度和文本（text），如下所示：

```
template <TInt S> class TLitC {
   // Various operators...
public:
   int iTypeLength; // This is the structure of a standard EPOC string
   char iBuf[S];
}

#define _LIT(name,s) const static TLitC<sizeof(s)> name={sizeof(s)-1,s}
```

這允許 EPOC 碼在 ROM 中使用 _LIT 巨集來定義字串：

```
_LIT(MyString, "Hello World");
User::InfoPrint(MyString); // Displays a message on the screen.
```

連結器（linker）能夠濾掉重複的常量定義，所以你甚至可以把 _LIT 定義式植入表頭檔。

2.1 **唯讀物件**。如果考慮物件的 bitwise state，C++ 編譯器堅持主張 const：const member functions 不能更改任何 data member，客戶也不能透過 const 指標刪除其所指的物件（Stroupstrup 1997）。如果一個精心設計的 class 絕對不會改變物件外部可見狀態，那麼令其所有 public function 都是 const，便能爲它提供邏輯常量性（logical const-ness）。下面這個簡單的 String class 同時提供了一個 const 和一個 non-const 存取運算子。客戶如果使用 const String&，將只能夠使用前者。

```
class String {
public:
   // Constructors etc. not shown...
   char operator[](int i) const { return rep[i]; }
   char& operator[](int i) { return rep[i]; }
private:
   char* rep;
}
```

C++ 支援以傳值（by value）方式傳遞參數，這種方式將在 stack 之中產生一個共享物的拷貝。如果物件規模大而且函式並不修改它，C++ 的常見風格是以 `const` reference 傳遞方式來 *Share* 某物。因此：

```
void function(const String& p);
```

通常優於

```
void function(String p);
```

因爲前者使用的 stack 空間較少。

2.2 Java **的唯讀物件**。Java 沒有 `const`，所以它對「什麼資料可以和程式碼一起儲存於唯讀記憶體中」的限制更加嚴格 — 惟有字串和單一基本型數值（single primitive values）才可以儲存於 Java 常數表中。例如下面程式碼：

```
final int myTable[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }; // Don't do this!
```

被編譯到一個建構 `myTable` 的特大函式中；元素一個個被指派給主記憶體內的 array。所以，「在唯讀記憶體中爲 Java 程式儲存資料」相當複雜。你可以將資料編碼爲 two-byte 整數，儲存於字串內；也可以透過 Java Native Interface（JNI）由 C++ 來管理和存取資料；或是將資料保存於資源檔內，並使用檔案存取函式（Lindholm and Yellin 1999）。

3. 靜態資料結構

某些程式需要相對龐大而內容不變的資料結構，例如：

- US Data Encryption Standard（DES）之類的加密演算法；
- log、sine、cosine 函數之類的數學演算法；
- 狀態躍遷表（state transition table），例如由支援 Shlaer-Mellor object-oriented methodology（1991）的工具所產生的表格。

這些表格可能相當龐大，所以將它們儲存於主記憶體中並非上策。它們恆定不變，所以可以將它們移至唯讀記憶體中。然而管理這些資料結構的發展是件十分艱巨的任務。

如果發展過程中表格資料經常變化，上上之策就是借助工具將表格生成爲獨立檔案，再由 ROM image 生成器將該獨立檔案併入。如果資料基本不變化，那麼比較容易的作法是將表格手工複製到程式碼中，再修改表格或其周圍程式碼，使編譯器能夠將其置於唯讀記憶體內。

4. 唯讀檔案系統

某些環境將唯讀記憶體當作檔案系統。這麼做有以下優點：檔案系統的結構可以將這些唯讀資料組織化；儘管應用程式無法修改唯讀資料，但它們可以透過一般的檔案操作函

式讀取資料。例如 EPOC 就支援一個邏輯檔案系統（Z:），該系統儲存於唯讀記憶體內，由 EPOC ROM 生成器構建完成，通常用戶不可見。所有 ROM-based 應用所需的資源檔都儲存於此檔案系統中。

通常，存取檔案系統會比直接存取記憶體慢。如果能夠將唯讀記憶體映射至應用程式的位址空間中，ROM 檔案系統內的資料就可以直接訪問了。例如 EPOC 的「字型和位元圖伺服器」就使用函式 `User::IsFileInROM()`直接存取 ROM 內的 bitmap 圖。

5. 版本控制

不同版本的 ROM image 會把同一份程式碼映射至不同位址。你得提供某種索引來幫助系統中的其他軟體和不同的 ROM 版本協同工作。例如 ROM image 常常以指標表格開頭，每個指標指向常式（routine）和資料結構的起始位址：外部軟體可以透過這個表格間接找到正確的呼叫位址（Smith 1985）。

Hooks 範式會談到「如何將表格儲存於可寫記憶體中」，這麼一來我們便得以擴展或替換許多常式（routines）。

示例（Examples）

下面例子利用唯讀的查詢表（lookup table）計算以徑度（radians）表示的 sine 函數。由於本例以 Java 寫成，而數字 arrays 不能儲存於 Java 常數表格中，所以我們必須將表格編碼爲字串（使用十六進制）。以下程式碼在我們的開發機器上執行，計算 256 個 sine 函數值，傳回 16-bit 整數。

```
final int nPoints = 256;
for (int i = 0; i<nPoints; i++) {
   double radians = i * Math.PI / nPoints;
   int tableValue = (int)(Math.sin(radians) * 65535);
   System.out.print("\\u"+Integer.toHexString(tableValue));
}
```

以上並非是完全正確的 Java 碼，因爲表格首尾的數個轉義碼（escape codes）缺少 "leading zeros"，但手工糾正此一錯誤比起在只執行一次的這種程式上花費時間有價值得多。

`sin` 函數在「表格提供的兩點」之間進行線性插入（linear interpolation）。爲求簡潔，我沒有列出整個表格：

```
static final String sinValues = "\u0000\u0324\u0648. . .\u0000";

public static float sin(float radians) {
   float point = (radians / (float)Math.PI) * sinValues.length();
   int lowVal = (int)point;
   int hiVal = lowVal + 1;
```

```
    float lowValSin = (float)sinValues.charAt(lowVal) / 65535;
    float hiValSin = (float)sinValues.charAt(hiVal) / 65535;
    float result = ((float)hiVal - point) * lowValSin
         + (point - (float)lowVal) * hiValSin;
    return result;
}
```

在配備算術協同處理器的快速機器上，此一 `sin` 函數比原生的（native）`Math.sin()`慢上許多個數量級！不過此程式之精確度超過二萬分之一，並且演示了查詢表（lookup table）技術，這種技術廣泛應用於不支援數學程式庫的環境中。如果你不想使用浮點數運算，而更青睞於整數運算（例如在低功率處理器上進行圖形壓縮與解壓縮），查詢表（lookup table）也應用得非常廣泛。

❖ ❖ ❖

已知應用（Known uses）

大多數嵌入式系統 — 從數字手錶和洗衣機遙控器，到移動電話和武器系統 — 都將程式碼和某些資料存放於唯讀記憶體（例如 PROMs 或 EPROMs）中。只有執行期資料才儲存於可塗寫的主記憶體內。Palmtops 和 Smartphones 通常將作業系統程式碼連同隨電話供應的應用軟體儲存於 ROM 中。第三方應用軟體則存放在次儲存體（由電池供電的 RAM）裡頭，必須載入主記憶體才能執行。類似情況，許多如 BBC Micro 之類的 1980 年代家用計算機都有複雜的 ROM 架構（Smith 1985）。

即便是「從次儲存裝置載入幾乎所有程式碼」的系統，啓動時仍然需要一些存放於 ROM 中的「引導用」初始化程式碼（'bootstrap' initialization code），才能從硬碟載入第一組指令。PCs 將這種引導碼（bootstrap）拓展爲 ROM-based Basic Input Output System（BIOS），提供對硬體的通用存取（generic access），使得「以一種作業系統（例如 DOS 或 MS Windows）支援多種類型的硬體」易如反掌（Chappel 1994）。

參見（See Also）

我們可以運用 *Copy-on-write*（91）和 *Hooks*（72）更改唯讀儲存器中的資料。*Copy-on-write* 和 *Hooks* 也允許將某些不經常變化（但並非恆定不變）的資料移至唯讀儲存器。

唯讀儲存器內所有的東西都適合在多個程式以及不同的組件之間 *Sharing*（182），也適合移至 *Secondary Storage*（79）中。

Paging（119）系統經常區分唯讀頁面（read-only pages）和可寫頁面（writable pages），忽略或阻止「對著唯讀頁面進行寫入動作」的企圖。多個行程（processes）可以安全共享同一個唯讀頁面 — 分頁（paging）系統無須「將其寫回硬碟」就可以丟棄之。

Hooks（掛鉤）

另名：Vector table, Jump table, Patch table, Interrupt table.

如何更改唯讀儲存器（read-only storage）內的資訊？

- 你正在使用唯讀記憶體。
- 一旦生成完畢，要修改唯讀記憶體的內容，不是很困難就是根本不可能。
- 唯讀記憶體中的程式碼或資料需要維護或升級。
- 你需要向儲存於唯讀記憶體中的資訊添加內容，並進行規模相對較小的修改。

唯讀儲存器的主要缺點在於它是唯讀的（read-only）。唯讀記憶體的內容可能設定於製造之時，也可能設定於升級之時；隨後它們將永遠固定不變。不幸的是它們總有臭蟲需要修正，或需要升級功能。例如 Strap-It-On 的 ROM 內的 Word-O-Matic 程式碼發佈版本可謂「蟲」害氾濫，對這些臭蟲所做的修改需要加入既有系統去。此外 Strap-It-On 市場部宣佈，他們將額外增加一項預測輸入功能，自動補足用戶的輸入，減少敲擊鍵盤的次數（Darragh et al. 1990）。

如果資訊儲存於 EPROMs 之類的「不完全可寫儲存器」（partly writable storage）中，你可以發佈一個全新的 ROM image，再想點什麼辦法說服所有客戶投入一些時間，冒點風險將 ROM image 升級。然而對顧客而言，ROMs 的升級實在苦不堪言，如果你不能總控整個系統，從商業角度談論 ROMs 的升級常常不切實際。進一步說，一個已發佈的 ROM 不太可能毛病多到必須徹底重新發佈一個。通常，需要修改的資訊量都很小，即使是大量修改，從整體角度來看仍然微不足道。

你可以忽略既有的唯讀記憶體，在可寫主記憶體中儲存一份資訊新拷貝。即使系統有充份的可寫記憶體，足夠容納唯讀記憶體內容的一份完整拷貝，通常你也沒辦法提供大量主記憶體用以儲存 ROM 的多份拷貝。

因此：在可寫儲存器（writable storage）內，透過 hooks 存取唯讀資訊，然後更改 hooks 以便造成資訊更改的假象。

不用唯讀記憶體而用可寫記憶體，將系統連接起來，是擴展唯讀儲存器的關鍵。設計系統使用唯讀儲存器時，千萬不要直接存取該儲存器。先在可寫記憶體中爲每個儲存於唯讀記憶體的進入點（entry point，指向函式、組件、資料結構或資源）配置一個 hook，然後初始化每個 hook，使其指向相應進入點。保證「每次存取進入點」都必須經過 writable hook — 所有存取動作，無論來自唯讀記憶體或可寫記憶體，都應該使用 hook。

如果打算升級唯讀記憶體，你只需複製需要修改的部分，並對著副本進行修改。然後你可以將修改後的副本儲存於可寫記憶體中，設置 hooks 指向修改後的碼。如有必要，修改後的碼也可以透過 hooks 間接呼叫程式其他部分（圖 1.12）。

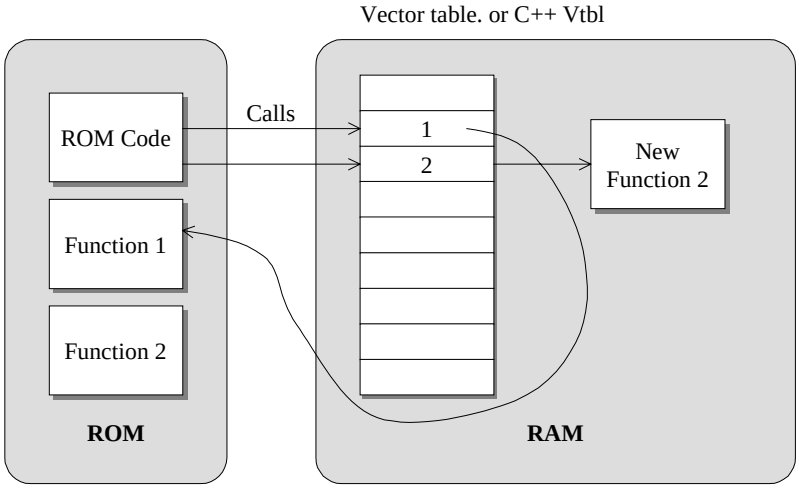


圖 1.12 Code hooks

例如，精心設計後的 Strap-It-On，每個主要組件都經由儲存於 RAM 之中的 hooks table 被間接呼叫，該 hooks table 是在系統被導入（booted）時初始化的。臭蟲的修正和擴展，以及第三方程式碼，都可以被載入系統主記憶體內，而後更改 hooks 指向它們。應用程式使用系統函式時，hooks 擔保一定能找到正確的程式節區 — 如果不是 ROM 內的原始程式碼，就是 RAM 內的新程式碼。

結果（Consequences）（譯註：本段保留之 *Italic Times New Roman* 字型均爲 forces 名稱）

Hooks 讓你得以擴展唯讀儲存器，不但降低唯讀儲存器的使用難度，也降低了程式對於可寫記憶體的需求（writable *memory requirements*）。

好的 hooks 可以提高程式設計質量，使未來的維護（*maintain*）和擴展更簡單。提供優秀

hooks 的 ROM-based 作業系統能夠大量減輕實作任何一項特定功能所需耗費的編程心力（*programmer effort*）。

然而：「將 hooks 設計到程式裡頭，並保證它們一定被使用」，需要良好的編程素養（*programmer discipline*）。由於必須測試 hooks 是否在恰當（正確）的時間被呼叫，因此會提高程式的測試成本。

透過 hooks 所進行的間接存取，比直接存取慢，降低了時間效率（*time performance*）；hook vectors 也需要佔用寶貴的可寫儲存器，因而稍稍增加了記憶體需求（*memory requirements*）。 正如許多病毒撰寫者所言，hook vectors 是攻擊系統完整性的理想點，所以使用 hooks 可能會降低系統的可靠性（*reliable*）。

❖ ❖ ❖

實作（Implementation）

實作 *Hooks* 範式時請考慮以下問題。

1. 從唯讀記憶體（read only memory）呼叫可寫記憶體（writable memory）

你無法預測儲存於主記憶體中的程式碼和資料的位址或進入點 — 真的，因爲記憶體可寫，所以記憶體位址會因程式版本的不同（甚至可能因爲程式正在執行）而變化。這使得 ROM 中的程式碼很難呼叫儲存於可寫記憶體中的碼，或依賴可寫記憶體中的資料。

解決之道是，你可以使用儲存於主記憶體已知位址上的附加 hooks，這些 hooks 指向主記憶體（而非唯讀記憶體）內的程式碼和資料。ROM 內的程式碼可以循著這些 hooks 找到它想使用的主記憶體組件的位址。

2. 擴展唯讀記憶體內的物件

物件導向環境將「操作行爲」和「被操作物件」聯繫起來的動作，稱作「動態派送」或「訊息發送」或「特殊多型」（dynamic dispatch，message sending or ad-hoc polymorphism）。你可以用它來實現相當靈活的 hooks。例如 EPOC 和 Windows CE 都支援位於 RAM 中的 C++ 衍生類別，它們繼承自儲存於 ROM 中的基礎類別。系統呼叫 C++ 虛擬函式時，執行起來的程式碼既可能在 ROM 中，也可能在 RAM 中，視函式所屬物件的類別而定。編譯器與執行期系統保證 C++ 虛擬函式表（vtbls）擁有每個函式的正確入口（entry），因此 vtbls 的行爲類似 hooks table（Ellis and Stroustrup 1990）。由於繼承機制並不真正區分 ROM classes 和 RAM classes，所以 ROM 程式員可以用許多物件導向設計範式（例如 *Factory Method* 和 *Abstract Factory*）實作出可擴展的程式碼（Gamma et al. 1995）。

這種作法在 Java 編譯環境中同樣運行良好。根據標的物的類別（target object's class），Java 的動態繫結允許 ROM-based 程式碼呼叫 ROM 或 RAM 之中的函式（methods）。

3. 擴展唯讀記憶體內的資料

當資料以 ROM 檔案系統的檔案形式存在時，更換 ROM-based 資料最是簡單。這種情況下只要保證「搜尋 ROM 檔案系統之前先搜尋其他檔案系統」就足夠了。例如 EPOC 會按驅動器字母順序，逐一搜尋每個驅動器上相同目錄內的資源檔。驅動器 Z，也就是 ROM 驅動器，理所當然最後一個被搜尋。

你也可以藉存取函式（accessor functions）之助，使用儲存於唯讀記憶體內的資料結構。如果透過 hooks 呼叫存取函式，你就可以藉由修改這些存取函式來修改「系統其餘部分從唯讀記憶體中取得的資料」。

如果你直接存取唯讀記憶體，那麼你需要很好的編程素養（*programmer discipline*）才有辦法撰寫可同時用於 ROM 和 RAM 的程式碼。讀取資料時，你應當先全面搜尋 RAM，然後才搜尋 ROM；寫入資料時，你只能對著 RAM 寫入。此種作法保證，如果你藉由「向 RAM 寫入資料」而替換了 ROM 內的資料，那麼 RAM 內修改過的版本將比 ROM 內的原始版本更早被找到。

示例（Examples）

Strap-It-On 的作業系統主要儲存於 ROM，需透過 hooks table 才能取用。我們可以透過修改 hooks 來更新作業系統。下列 C 程式碼實作出 hook table 的生成，並攔截作業系統提供的函式 `memalloc()` — 它用來配置記憶體。

Strap-It-On 作業系統的基本資料型別（basic data type）稱爲 `sysobj` — 它可能是個指向某塊記憶體的指標、或是一個 4-byte 整數，或兩個 2-byte 短整數…等等。所有系統呼叫都接收並傳回單個 `sysobj`，所以 hook table 實質上是個函式指標表，其內所有元素都是函式指標，指向「接收並傳回 `sysobjs`」的函式。

```
typedef void* sysobj;
const int SIO_HOOK_TABLE_SIZE = 100;
typedef sysobj (*sio_hook_function) (sysobj) ;

sio_hook_function sio_hook_table[SIO_HOOK_TABLE_SIZE];
```

系統開始運行時，會將一個函式指標（指向 `memalloc` 函式）儲存於 hook table 的適當位置上：

```
extern sysobj sio_memalloc(sysobj);
const int SIO_MEMALLOC = 0;
sio_hook_table[SIO_MEMALLOC] = sio_memalloc;
```

Strap-It-On 應用程式藉由「間接呼叫 hook table 內相應條目所指的函式」，完成 `memalloc()` 之類的系統呼叫：

```
void *memalloc(size_t bytesToAllocate) {
   return (void*)sio_hook_table[SIO_MEMALLOC]((sysobj)bytesToAllocate);
}
```

1. 利用 hook 改變函式

爲改變系統行爲，例如實作一個記憶體計數器，我們首先配置一個變數，保存 `memalloc()` 初始版本（在唯讀記憶體內）的位址。之所以要保存原始版本，是因爲我們的記憶體計數器只計算被索求的 bytes 數，之後還得呼叫原始版本才能實際配得記憶體：

```
static sio_hook_function original_memalloc = 0;
static size_t mem_counter = 0;
```

接下來我們可以撰寫一個替代函式，在其中計算記憶體需求量，並呼叫原始版本：

```
sysobj mem_counter_memalloc(sysobj size) {
   mem_counter += (size_t)size;
   return original_memalloc(size);
}
```

最後，我們將現行系統的 `memalloc()`位址從 hook table 拷貝至我們的變數中，安裝記憶體計數器，並將我們的新函式安裝於 hook table 內：

```
original_memalloc = sio_hook_table(SIO_MEMALLOC);
sio_hook_table(SIO_MEMALLOC) = mem_counter_memalloc;
```

現在，對 `memalloc()`的任何呼叫（在客戶端程式碼和作業系統中進行，因爲 ROM 也使用 hook table）都首先由記憶體計數器處理。

❖ ❖ ❖

已知應用（Known uses）

我們可以藉由 RAM 的 hook vectors 接觸到 Mac、BBC Micro 和 IBM PC 的 ROMs，也可以藉由改變 hooks 而更新它們。Emacs 大量使用 hooks 來擴展其執行碼（executable-only code）— 利用這種方式，多個用戶可以共享同一份 Emacs 二進制碼，每個用戶並且擁有專屬的訂製環境（Stallman 1984）。NewtonScript 允許物件繼承自唯讀物件，兼用 *Hooks*（72）和 *Copy-On-Write*（191），以便將來可被修改（Smith 1999）。

EPOC 的 "Time World" 應用程式擁有一個 ROM-based 大型資料庫，內含世界各大城市及其相關時區、電話號碼、地理位置等資訊。它允許用戶添加條目，並將新條目儲存於「與預先定義之 ROM 資料庫相似」的 RAM 資料庫中，用戶尋找城市時兩者皆會被搜尋。

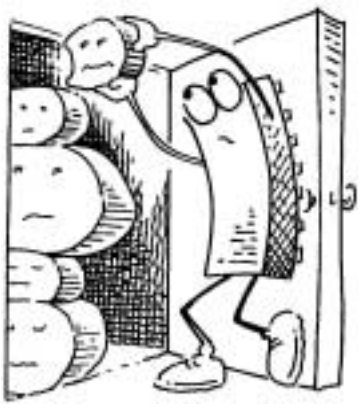
EPOC 採用另一種方法來更新 ROM。它提供一個設備驅動程式，用以「更改系統的虛擬記憶體映射表（virtual memory map）」，以求映射一或多個新的程式碼頁面（new pages of code），取代原有的 ROM 記憶體內容。由於新程式碼必須精準佔據原程式碼的相同空間，在完全相同的記憶體位址上提供完全相同的進入點，因此這種方法不太容易操控。

參見（See Also）

Copy-On-Write（191）是更改 *Read-Only Memory*（65）內容的一種互補技術，*Copy-On-Write* 和 *Hooks*（72）常常可以聯合使用。

配套使用 *Hooks* 和 *Read-Only Memory*，是 hooks 一般用途下的一個特殊例子，用來擴張人們無法直接修改的系統。*Object-Oriented Design Patterns*（Gamma et al. 1995）中的許多範式也和「不直接改變而能完成系統擴張」有關。

Hooks 是系統設計熱點方法（hot-spot approach）中的一個重要成份（Pree 1995）。



Small Memory Software

2

Second Sotrage

次儲存裝置

一旦你用光主記憶體，接下來該怎麼辦？

- 可用的主儲存裝置（primary storage）無法滿足你的記憶體需求。
- 你無法充分降低系統的記憶體需求。
- 你可以將次儲存裝置添加到「執行系統」的設備上。

有時，系統的主記憶體（primary memory）不足以滿足程式的記憶體需求。

舉個例子，Strap-It-On 的 Word-O-Matic 文書處理器（word processor）應當能夠允許用戶編輯大量文本，並支援格式化文本的供顯示或列印，以及拼寫檢查、語法檢查、語音輸出、郵件合併；還有特殊的 StoryDone 功能，用來爲短篇故事撰寫結局。不幸的是 Strap-It-On 只有 2 Mb RAM。在 Word-O-Matic 程式碼佔據大部份記憶體的情況下，程式員究竟如何實作上述那麼多功能？

本書裡頭有其他許多可以降低記憶體需求的技術。*Compression*（135）可以將資訊儲存在較小記憶體內。利用 *Memory Limit*（32）測試應用程式可保證程式被完美裝入狹小記憶體空間。你可以刪除某些特性或降低服務品質，藉以降低系統機能。然而許多時候這些技術無法充分降低程式的記憶體需求，例如「需被隨機存取的資料」難以壓縮，又例如程式必須提供市場期望的特性和品質。

但對大部份應用程式而言，希望長存人間。即使在小型系統中，任意時刻上，程式執行所需的記憶體數量往往只是全部記憶體的一小部分。因此，任意時刻程式所需的程式碼和資料應該儲存在哪裡，並非問題所在；相反地，把未來可能需要（或不需要）的其餘程式碼和資料儲存在何處，才是問題之所在。

因此：把次儲存裝置（second storage）當作執行期間的額外記憶體。

多數系統都有某種形式的快速次儲存裝置（secondary storage）。次儲存裝置與 RAM 截然不同，因爲處理器無法對前者的每一個位址直接寫入。但即使用戶不介入，應用程式存取次儲存裝置依然輕而易舉。大部份次儲存裝置都支援檔案系統（file systems ），俾使資料棲身於擁有文字名稱和目錄結構的檔案之中。通常每個檔案還支援對其資料的隨機存取（random access；例如「請將檔案起始處以下偏移位置 301 的 byte 給我」）。

如果你能把程式和資料劃分爲合適片段，你就可以只將任意時刻當時需要的程式碼和資料段載入主記憶體，其他部分繼續保存於次儲存裝置。當你不再需要目前主記憶體內的程式片段時，可以從次儲存裝置中選取更恰當的片段加以替換。

有許多種「可被修改且可隨機存取」的次儲存裝置：軟碟、硬碟、flash filing systems、 bubble memory cards、CD-ROM drives、writable CD-ROM file systems，以及經由網路存取的 gargantuan file servers。Palm Pilot 系統使用儲存於 secondary RAM 中的永續性「記憶體錄像」（memory record）。另一些次儲存裝置只提供序列（sequential）或唯讀（read only）存取：磁帶、CD-ROM、經由網際網絡（internet）存取的 web 頁面。

舉個例子，Strap-It-On（腕型 PC）附帶一套 CyberStrap，把 32 Mb bubble memory 儲存器連同 wrist-mounted 硬碟驅動器介面內置於錶帶中，於是 Word-O-Matic 開發者就擁有了可儲存大量資料的「硬碟」。Word-O-Matic 由數個具備 *Application Switching*（84）能力的獨立執行檔組成，它將所有未使用文件儲存於 *Data File*（92）之中；字典、語法規則及故事結尾骨幹都以 *Resource Files*（101）呈現；可選特性通常以 *Packages*（108）傳遞；最複雜的操作採用 object *Paging*（119）俾使可用的 RAM 看起來比實際多得多。

結果（Consequences）（譯註：本段保留之 *Italic Times New Roman* 字型均爲 forces 名稱）

使用 *Secondary Storage*（79）好似免費得到了大量額外記憶體，因而可以極大程度降低程式的主記憶體需求（*primary memory requirements*）。

然而：次儲存裝置需要管理，使資訊在主、次儲存裝置間傳送。管理需要付出時間效率（*time performance*）上的代價，還可能需要更多編程心力（*programmer effort*）和編程素養（*programmer discipline*），並爲了全域（*global*）機制而提高區域限制（*local restrictions*），降低程式的可用性（*usability*）。大部份次儲存裝置需要附加設備，也因此提高了系統的電力消耗（*power consumption*）。

❖ ❖ ❖

實作（Implementation）

想要高效使用次儲存裝置，有數個問題必須解決。

- 你要分割什麼？程式碼？資料？組態（configuration）資訊？抑或它們的某種組合？
- 誰執行分割：程式員？系統？抑或用戶？
- 誰發起載入（loading）和卸載（unloading）：程式員？系統？抑或用戶？
- 何時進行載入或卸載？

通常，程式愈細分，而且細分愈精細，程式對主記憶體的依賴就愈小，程式對次儲存裝置的運用也就愈多。粗糙的分割也許只是處理程式碼或只處理資料，這時候可能需要更多主記憶體，而加諸次儲存裝置的壓力也就輕得多。

讓程式員細分程式，比出於某種原因讓系統或用戶細分程式，需要付出更多心力；精細的分割也比粗糙的分割需要更多心力。因此通常唯有系統自動提供分割功能才可能獲得高精細度的分割；但建立自動系統需得付出艱辛的努力。如果讓用戶分割程式或資料，程式員幾乎無需付出任何代價，但卻會降低系統的可用性（usability）。

決定由誰控制各段分割的載入和卸載時，我們面臨到類似的取捨。如果由系統自動完成，可以減輕系統建立者之外的所有人的工作；否則重任就落在用戶和程式員肩上。循序載入和卸載是最容易實作的（對用戶而言效率往往是最差的）。更複雜的方案對用戶而言才是天衣無縫，例如視需要而載入或卸載（load and unload on demand）程式碼和資料，或甚至造成「次儲存裝置之依賴性」對用戶和程式員完全透明。

❖ ❖ ❖

特化範式（Specialzation Patterns）

本章後續包括五個特化範式，描述 *Secondary Storage* 的不同使用方法。圖 2.1 顯示這些範式及它們之間的相互聯繫：箭頭表示關係緊密；打叉的線條表示範式之間針鋒相對。

這些範式也構成了一個序列：由簡單範式開始，由複雜範式收束全局。前者只需在小區域內實作，並依賴編程素養（programmer discipline）糾正實作；後者需要硬體或作業系統支援，只需極少（或甚至完全不需要）編程素養。每個範式根據上述問題，在其所定義的設計空間中佔據著不同的位置，如下所示：

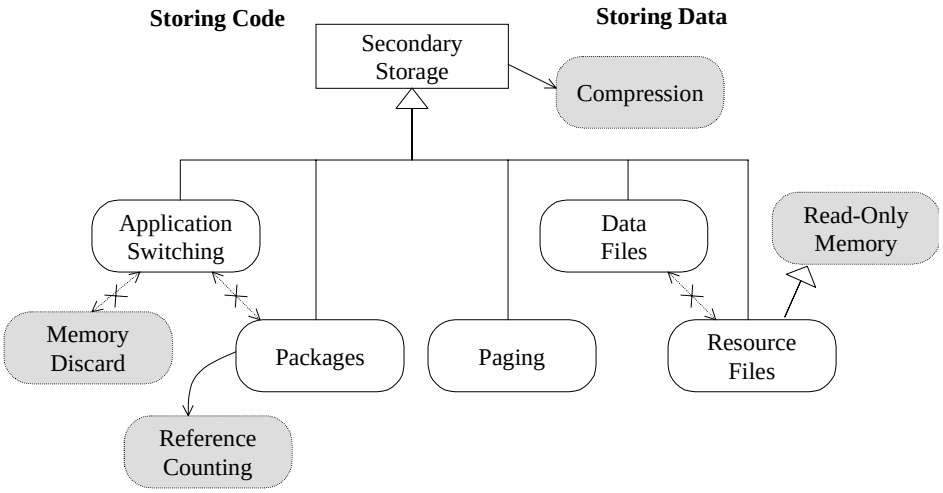


圖 2.1 *Secondary storage*（次儲存裝置）範式

Application switching（84）要求程式員把系統分成獨立的執行個體（executables），每次只可執行一個。各執行個體的執行次序可由它們自行決定，也可由外部腳本（external script）程式或用戶決定。

Data Files（92）將次儲存裝置作爲非活躍的程式資料的駐錫地。這些檔案可能（也可能不）爲用戶所見。

Resource Files（101）用來儲存靜態唯讀資料。當程式需要資源如字體、錯誤訊息、視窗描述（window description）時，就把資源從檔案載入臨時記憶體內；事後再釋出之。

Packages（108）用來儲存程式碼區塊（chunks of the program code）。程式員把程式碼分爲多個 packages（packages），執行期間按需要載入或卸載之。

Paging（119）將程式任意分解爲非常細小的單元（所謂的頁面，page），各單元自動在主、次儲存裝置間移動。*Paging* 可以處理程式碼和資料，支援唯讀資訊以及不同程式之間共享的資訊，並對大部份程式員和用戶透明。

以上所有範式，在某種意義上，都是和程式員所在之工作環境所提供的工具做買賣。環境（編譯工具和執行期系統）愈複雜，程式員所需負擔的記憶體管理工作愈少。然而發展出複雜的執行期環境需要付出很大心力，而且複雜的執行期環境在執行期間也有自己的記憶體需求。圖 2.2 顯示此一方案的每一個範式的位置。

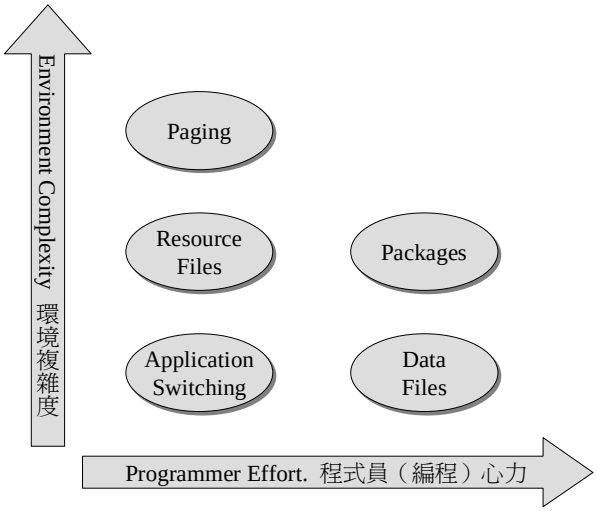


圖 2.2 實作心力 vs. 環境複雜度

參見（See Also）

無需將程式或資料的 *Read-only*（65）片段存回次儲存裝置，就可以直接刪除它們。

你可以運用 *Compression*（135）減少次儲存裝置的空間佔用量。

次儲存裝置管理是現代作業系統的主要功能之一。使用次儲存裝置技術的更多背景資訊和細節，可以在許多現代作業系統教材中找到（Tannenbaum 1992; Leffler et al. 1989; Goodheart and Cox 1994）。

Application Switching（任務切換）

另名: Phases, Program Chaining, Command Scripts

面對提供許多不同函式的系統，你如何降低其記憶體需求？

- 系統太龐大，主記憶體無法容納全部程式碼和資料。
- 用戶經常一次只需執行一項工作（task）。
- 單一工作（single task）只需自己的程式碼和資料就能執行；其他程式碼和資料可以存放在次儲存裝置內。
- 一次只編寫一組相關工作（或一個應用程式）更容易些。

某些系統是個龐然大物，大到同一時刻主記憶體無法容納全部可執行碼和資料。

例如 Strap-It-On 用戶可能會進行文書處理、運行試算表，閱讀 web 網頁、計賬、管理資料庫、打電動遊戲，或者利用遙控裝置 "StrapMan" 管理大型電信網絡的日常策略。程式員如何在 2 Mb RAM 裡頭讓上述所有功能都大顯身手呢？別忘了 StrapMan 的五項功能每一項都需要 1 Mb 程式碼和 0.5 Mb 臨時資料（需置放於 RAM）。

任一特定時刻，多數系統只需整體功能的一個小子集，能夠支援某項工作足矣。多數系統的大部分程式碼和資料在多數時間內並沒有被用到，但它們卻始終佔據寶貴的主記憶體空間。

系統愈複雜，發展團隊愈龐大，發展就愈困難。軟體開發人員一直對於「將系統架構分解爲獨立組件，減少組件間的依存性」青睞有加。無疑地，組件的確可以使系統發展更易於管理，但不能降低主記憶體需求。

因此：把你的系統分解成獨立執行個體（independent executables），每次只運行一個。

大部份作業系統都支援在次儲存裝置上以「可執行檔形式」呈現的獨立程式組件。運行中的執行個體我們稱爲行程（process），其程式碼和資料佔用著主記憶體。行程終止時，它使用的所有主記憶體都將完璧歸趙，還給系統。

你應該設計你的系統，使得「用戶即將一併使用或旋即就會使用」的行爲，能夠位於同一執行個體內。你應該提供某些設施，使得必要時得以啓動另一個執行個體並終止當前的執行個體。新行程（process）得以復用舊（已死）行程所釋出的所有記憶體。

許多作業系統只支援「一次執行一個行程」。在 MS-DOS 中一個執行個體必須提供「自殺」功能，用以在另一個執行個體運行前結束自我生命；在 MacOS 和 PalmOs 中，全體應用程式共享一項控制功能，使得以選擇另一個應用程式並切換過去（Chappell 1994; Apple 1985; Palm 2000）。多工系統經常使用 *Application Switching* 來降低主記憶體需求。

舉個例子，沒有任何 Strap-It-On 用戶會想在任何時刻執行一個以上的工作（tasks），因爲螢幕尺寸如此之小，那麼做根本就不現實（也不可能）。所以每項工作都以獨立執行個體（文書處理器、試算表、web 瀏覽器、會計、資料庫、遊戲）的形式出現。Strap-It-On 提供一個控制對話框，允許用戶終止當前工作（應用程式），啓動另一項工作。每個應用程式退出時都會保存狀態，重新啓動時恢復原狀態，因此用戶無法知道哪一個應用程式曾經終止而後被重新啓動（除了載入所帶來的等待之外）。但即使作爲單一執行個體，RAM 依然容納不下 StrapMan 程式。所以 StrapMan 的作者把它分成六個不同的執行個體（一個是主程式，其他五個各對應一項功能），必要時由主執行個體切換至其他各個執行個體。

結果（Consequences）（譯註：本段保留之 *Italic Times New Roman* 字型均爲 forces 名稱）

每個行程（process）的記憶體需求（*memory requirements*）都小於整個系統的記憶體需求。行程終止時由作業系統回收記憶體。因此這種方法可以減少管理記憶體所花費的編程心力（*programmer effort*），也減輕「記憶體洩漏」（memory leak）的可能性。

不同的執行個體可能由不同的語言實現 — 視需求而被解釋或編譯。某些執行個體可能是既存的老舊程式碼，也可能是作業系統提供的工具程式（utility）。因此 *Application Switching* 可以顯著降低研製系統所花費的編程心力（*programmer effort*），鼓勵復用（*reuse*），讓維護（*maintenance*）更輕鬆愜意。由於 script（腳本）極易修改，所以 Script-based 方法非常靈活。此外，由於應用程式傾向於經常性的停止、重啓，所以「造成行程結束」的那種嚴重錯誤並不會給用戶帶來太大麻煩，相對提高了系統的強固性（robustness）。

在諸如 PalmOs 之類的單行程環境中，所有行程佔據同一份記憶體空間，因而所需的記憶體數量容易預期（*predict*），提升了可靠性（*reliability*），使測試更容易（*testing easier*），移除了全域（*global*）記憶體的使用對每一個區域（*local*）應用程式的影響。你只需啓動第一個行程就可以讓系統運行起來，減少啓動時間（*start-up times*）。很容易就可以理解單行程環境發生了什麼事，因而簡化了即時（*real-time*）編程。

然而：將大型程式劃分爲漂亮的行程集（processes set）可能頗爲不易，所以設計多行程應用程式需要大量的編程心力（*programmer effort*），其實作品中會有很高的區域複雜度（*local complexity*）。

如果你有許多執行個體，「啓動每個個體並在其間傳輸資料」的代價將主宰系統的執行期效率（runtime performance）；如果不同行程間的控制流（control flow）頗爲複雜，而且行程頻繁地終止、重啓，那麼 *Application Switch* 也會形成問題。

單行程（single process）環境裡頭，用戶只能使用當前執行個體所提供的功能，因此切換動作會降低系統可用性（*usability*）。如果用戶不得不明確管理行程，也會降低可用性。

本範式不支援背景活動，例如 TCP/IP 協定、連繫移動電話、或背景下載電子郵件。即使用戶切換工作，這些行爲也必須繼續執行。背景工作的程式碼或是被忽略（這會降低可用性），或是棲身於某獨立行程中（這會增加編程心力，*programmer effort*），或是以中斷常式（interrupt routine）完成（這需要大量而特殊的編程心力，*programmer effort*）。

❖ ❖ ❖

實作（Implementation）

爲實作出 *Application Switching*，你必須將系統劃分爲獨立組件（請參見 *Small Architecture*, 25）。行程間的通訊頗爲棘手，所以總體而言，分割必須滿足以下規則：

- 行程間的控制流（control flow）必須簡單。
- 行程間幾乎不傳遞短暫性資料（transient data）。
- 分割必須對用戶有某種意義。

圖 2.3 顯示 *Application Switching* 的兩種主要實作法。

1. Program chaining（程式鏈）

一個行程可以明白地將控制權傳遞給後繼行程，此即所謂 "program chaining" — 因 BASIC 語言（某些版本）中的 `CHAIN` 命令而得名（Digital 1975; Steiner 1984）。Program chaining 要求每個執行個體知道下一個執行個體是誰。這可由每個應用程式明白編寫出來，或是成爲應用程式框架程式庫（application framework library）的一部分。如果有這樣一個框架，每個執行個體便能夠以它決定接下來該切換至哪個應用程式，並實際進行切換，而無需付出太多編程心力。MacOs（工作切換器，task switcher）和 PalmOs 應用程式框架就是這麼做的（Apple 1985; Palm 2000）。

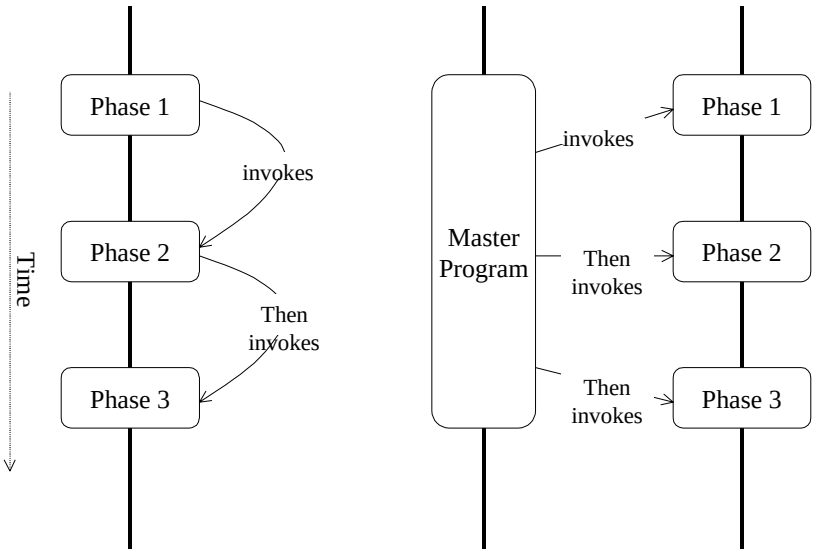


圖 2.3　兩組不同的實作階段（implementation phases）

2. Master program（主控程式）

另一種方法是令控制腳本（master script）或最上層命令程式（top level command program）依次喚起每個應用程式。由於每個執行個體不需對週遭環境（context）有太多了解，並且能夠獨立運作，所以主控程式鼓勵復用（reuse）。UNIX 環境首倡「能夠以此方式協同運作」的小型互操作工具（small interoperable tools）」（Kernighan and Pike 1984）。即使用了主控程式，終止程式（terminating program）仍然可以經由兩種方式協助我們決定接下來該執行哪個應用：使用結束碼（exit codes）將資訊傳回給主控程式，或產生輸出檔案臨時檔供主控程式讀取。

3. 行程間通訊（InterProcess Communicating，IPC）

當一次僅執行一個行程時，各個獨立組件如何通訊？你不可能運用主記憶體，因爲行程結束時主記憶體內容將一筆勾銷。你需要使用以下一種或多種機制：

- 將命令列參數（command line parameters）或環境變數傳入新行程；
- 次儲存裝置（檔案、記錄或資料庫），一個行程負責寫，另一個行程負責讀；
- 因環境而異的機制。例如當一個新行程覆寫（overwrites）當前行程時，許多 Basic 變體會以 `COMMON` 關鍵字補充 `CHAIN` 命令，以之指定「待保留的資料」（Steiner 1984）。

4. 管理資料

如何讓用戶感覺應用程式永不終止 — 即使它已被分割爲多個獨立行程？許多環境只支援數量很少的行程，甚至可能僅支援一個，但用戶可不希望每次啓動一個新行程時都必須重新生成全部狀態。他們需要「程式一直在背景中執行」的幻象。

解決之道是令應用程式退出時將狀態保存於 *Secondary Storage*（79），並於重新啓動時恢復原狀。許多 OO 程式庫和 OO 環境都支援「將所有重要物件（包括資料和狀態）的串流化（streaming）動作」變成單一操作，這需要一個二進制的 file stream，其中定義了用以讀寫原始型別（如 `int,char,float,string`）的 stream 函式，而每個「用以表示應用程式狀態」的 class 也都定義有自己的 stream 函式。

當你 'streaming out' 某個物件導向應用程式時，無論每個物件有多少個 references ，你應當保證該物件只「流」出一次。解決良策是讓 file stream 維護一張物件標誌符號表（object identifier table）— 每當 stream 收到「'stream out' 某個物件」的請求，首先搜尋該格，如果找到物件確在表格裡頭，就只保存一個 reference 指向該物件檔案位置，而不再次保存該物件（譯註：MFC 的 serialization 機制就有如此設計，其細節相當複雜，非三言兩語可盡）。

Java 程式庫透過 Serialization framework 支援物件永續（persistence；見 Chan et al. 1998）。所有具永續性的 classes 都必須實作 `Serializable` interface，此外不需任何其他程式碼，執行期環境便會次第寫入（並於未來次第讀出）物件的所有資料成員，並像上一段所說，每個物件只寫一次。Java 程式庫提供的 `ObjectOutputStream` 和 `ObjectInputStream` 提供了讀寫一群物件的函式：`writeObject()`和 `readObject()`。習慣上這樣產生出來的檔案通常有著 `.ser` 副檔名；某些應用程式會將原始 `.ser` 檔連同 JAR 檔案中的 Java 程式碼一起出貨。

示例（Examples）

這裡有個取自 MS Windows 3.1 的極簡單實例。我們無法在 MS Windows 執行期間使用 scandisk（一個硬碟檢查程式），所以我們先執行 scandisk 再執行 Windows：

```
@REM AUTOEXEC.BAT Command file to start MS Windows 3.1 from DOS
@REM [Commands to set paths and load device drivers omitted]
```

```
C:\WINDOWS\COMMAND\scandisk /autofix /nosummary
win
```

下面的 Java 函式會終止當前行程，並鏈接至（chains to，見 p86）另一個行程：

```
void ChainToCommand(String theCommand) throws IOException {
   Runtime.getRuntime().exec(theCommand);
   Runtime.getRuntime().exit( 0 );
}
```

請注意，如果這個函式被用來執行另一個 Java 程式，它會在終止現行 Java 虛擬機器前先創建另一個 Java 虛擬機器，兩個虛擬機器（VMs）會暫時共存，這需要大量記憶體。

UNIX `exec` 函式族系更適用於記憶體匱乏時的單行程鏈（single process chaining）：它們都在目前行程的空間內啓動一個新行程（Kernighan and Pike 1984）。下面的 C++ 函式使用 Microsoft C++ 提供的 `_execl` 變體（Microsoft 1997），並藉助 Windows 環境變數 `'COMSPEC'` 找到命令直釋器，因爲只有命令直釋器才清楚到哪裡找出各個可執行檔並正確解析命令：

```
void ChainToCommand( string command ) {
   const char *args[4];
   args[0] = getenv( "comspec" );
   args[1] = "/c";
   args[2] = command.c_str();
   args[3] = 0;
   _execv( args[0], args );
}
```

此函式永不回返。請注意，雖然所有的 RAM 記憶體都被丟棄，但 `execl` 並不關閉 file handles — 它們在「被鏈接行程」（chained process）中仍保持開啓狀態。如果你想進一步瞭解 execl 及其相關函式，請閱讀 C++ 或其程式庫說明文件。

下面的 EPOC C++ 程式碼爲一個簡單類別實作出 streaming 功能，以便在應用程式被切換出去時將資料保存於檔案。`TSerialPortConfiguration` class 代表一個印表機埠（printer port）的組態設置，其大部份資料成員如果不是值域範圍不大的 C++ `enums`，就是 1-byte 整數（C++ 採用 `char`，EPOC C++ 採用 `TInt8`）；`TOutputHandshake` 是另一個 class：

```
class TSerialPortConfiguration {
   // Various function declarations omitted . . .
   TBps iDataRate;
   TDataBits iDataBits;
   TStopBits iStopBits;
   TParity iParity;
   TOutputHandshake iHandshake;
   };
```

`InternalizeL()`和 `ExternalizeL()`分別從 stream 中讀寫物件。儘管 `iDataRate` 內部以 4-byte integers 和 enums 表示，但我們知道以 single byte 形式將它儲存爲 *Packed Data* 並不會丟失任何資訊。至於 `TOutputHandshake`，有自己的 streaming 函式，所以我們直接使用：

```
EXPORT_C void TSerialPortConfiguration::InternalizeL(
                          RReadStream& aStream) {
   iDataRate = (TBps) aStream.ReadInt8L();
   iDataBits = (TDataBits) aStream.ReadInt8L();
   iStopBits = (TStopBits) aStream.ReadInt8L();
   iParity = (TParity) aStream.ReadInt8L();
   iHandshake.InternalizeL(aStream);
   }

EXPORT_C void TSerialPortConfiguration::ExternalizeL(
                       RWriteStream& aStream) const {
   aStream.WriteInt8L(iDataRate);
   aStream.WriteInt8L(iDataBits);
   aStream.WriteInt8L(iStopBits);
   aStream.WriteInt8L(iParity);
   iHandshake.ExternalizeL(aStream);
   }
```

❖ ❖ ❖

已知應用（Known uses）

PalmOs 和早期的 MacOS 環境在任一時刻均只支援單一用戶行程；二者都提供 framework functions 爲用戶模擬多工（multi-tasking）。MacOS 採用 persistence（永續機制）保存應用資料，PalmOs 則採用次儲存裝置的「記憶體記錄」（Apple 1985; Palm 2000）。

在 UNIX 環境中，藉助適當的、被支援的腳本引擎（script engine），任何腳本可以和二進制可執行檔（binary executables）有著完全相同的執行方式。UNIX 環境鼓勵程式員經由腳本控制行程（Kernighan and Pike 1984）。在 MS Windows 和 DOS 環境中，儘管「產生一個簡單的 Windows BAT 檔，喚起諸如 Tcl（Ousterhout 1994）和 Perl（Wall et al. 1996）之類強大的腳本語言」易如反掌，但完全獲得支援的腳本格式只有十分簡單的 BAT 和 CMD 格式。

UNIX Make utility 使用 *Application Switching*（以及其中所需的 *Data Files,* 92）技術來編譯程式。它通常先逐一爲每個輸入檔執行起預處理器（pre-processors）和適當的編譯器行程（compiler process），然後再執行一個或多個連結器行程（linker process），產出完整的可執行檔（Kernighan and Pike 1984）。

參見（See Also）

Packages（108）在單一行程內提供了類似功能，其作法是將程式碼載入時機推遲至必要爲止。不過兩者仍有不同：如果採用 *Application Switching*，task 結束時由作業系統負責丟棄 task 持有之記憶體、程式碼和其他資源；如果採用 *Package*，則必須由 task 自行釋出資源。

Paging（119）提供更多的靈活性，它能夠兼而處理程式碼和資料。

我們也可以使用 *Compression*（135）把可執行檔儲存於 *Secondary Storage*（79）。

Memory Discard（244）有一股與 *Application Switching* 類似的原動力，但規模小得多。*Application Switching* 只在一個行程終止時方可重新獲得其所佔用的全部記憶體，*Memory Discard* 則允許應用程式在其執行過程中重新使用被一群物件佔用的記憶體。

Data Files（純資料檔）

另名: Batch Processing, Filter, Temporary File

如果資料量太大，主記憶體容納不下怎麼辦？

- 系統過分龐大，RAM 無法將全部的程式碼和資料一起容納進來。
- 程式碼本身可以裝入 RAM — 也許藉其他範式之助。
- RAM 容納不下資料。
- 一般而言，資料被循序（sequentially）寫入。

有時候，程式本身相當小巧，卻得處理大量資料。這種情況下的記憶體需求（*memory requirements*）意味程式本身可以進入主記憶體，但資料卻耗用太多記憶體。

例如，爲大部頭書籍編排格式時，Word-O-Matic 的 Text Formatter 的輸入和輸出資料量會超出 Strap-It-On 的主記憶體容量。一旦「將輸出資料、索引檔、交叉參考…等等全部植入 RAM」根本不可行時，假設先不考慮如何生成所有索引檔、如何更新全部交叉參考，試問 Word-O-Matic 設計者該如何實作程式，才能生成輸出用的 PostScript 資料？

將程式劃分爲多個更小區段（如 *Application Switching* 所示）能夠減少程式自身所需的記憶體，但這對於減少 I/O 所需的記憶體並無幫助。同樣道理，*Compression*（135）或許能減少儲存資料時所需的次儲存裝置量，卻無法減少處理資料時所需的主記憶體量。

其實大部份系統並不強求你把全部資料置於 RAM 中。現代作業系統使得「讀寫次儲存裝置上的檔案」毫不費力。而且大部份 tasks 並不會同時存取全部資料。

因此：一次只處理一部分資料；其餘放在次儲存裝置內。

利用循序存取（sequential access）或隨機存取（random access）讀取檔案中的每一筆資料並進行處理，再把處理完畢的資料循序寫回一個或多個檔案。你也可以把臨時資料寫至次儲存裝置，直到你打算使用它們爲止。只要謹慎行事，「處理一小部分資料」所需的主記憶體量將與「把所有資料統統置於主記憶體內」所需的總記憶體量，有著天壤之別。你必須能夠將輸入資料和輸出資料儲存爲 *Secondary Storage* 上的檔案，因此輸入和輸出資料必須被分割得很乾淨。

舉個例子，Word-O-Matic 把書上各章儲存爲各自獨立的文本檔案（text file）。Word-O-Matic Text Formatter（暱名 "Wombat", 澳洲特產「袋熊」）對這些檔案進行多回合處理，見圖 2.4。第一回合先逐一掃瞄各章，尋找「交叉參考和索引項」在檔案中的位置和標的，並把產生每一交叉參考所需的全部資訊寫入一個臨時的「交叉參考」檔內。然後，第二回合，掃瞄交叉參考檔，在記憶體中爲該檔產生索引，而後讀取各章檔案，爲每一章產生一個包含交叉參考和索引的短暫版本。Wombat 藉記憶體內索引之助，隨機存取索引檔，因而讀得交叉參考資料。由於章節內容的變化會造成頁碼變動，所以 Wombat 還在記憶體內維護一張表格，顯示變化過程中每一個參考標的物的變動情況。最後，第三回合，Wombat 分次讀取上述「暫時章節檔」內的一小部分，循序寫出 PostScript 資料，同時根據記憶體內的表格，修正索引和參考頁碼。利用這些技術，Wombat 可以在區區 50 Kb RAM 記憶體中編排一整本書。

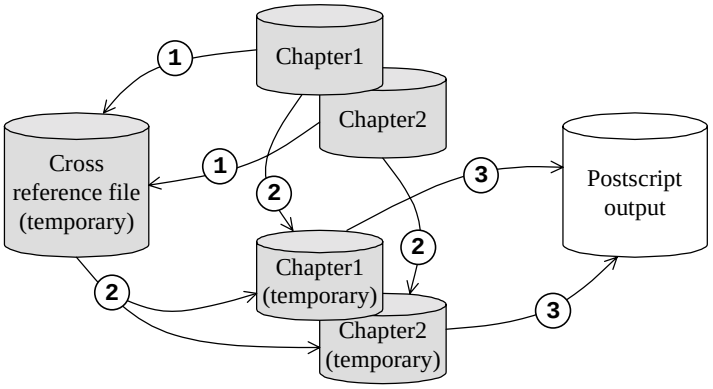


圖 2.4　Wombat 資料檔，及其各個處理階段（phases）

結果（Consequences）（譯註：本段保留之 *Italic Times New Roman* 字型均爲 forces 名稱）

記憶體需求（*memory requirements*）會下降，因爲大部份資料都位於次儲存裝置中。系統對於主記憶體的需求量更加可預期（*predictable*），因爲你可以配置固定量的記憶體來支援資料的處理，不必爲了大小可能改變的資料而配置大小可能改變的記憶體。

你可以運用「查看次儲存裝置檔」的工具程式（utility）檢視應用程式的輸入和輸出函式，這使得測試（*testing*）更加容易。*Data Files* 也使得「把應用程式分解爲不同的獨立組件，只透過純資料檔連接」變得更輕鬆，降低了區域性（*local*）改變所造成的全域性（*global*）影響，使維護（*maintenance*）更加容易。*Data Files* 還使得實作 phases（階段）的難度大減，並針對各個 phases 支援 *Application switching*（84），例如 Wombat 第一階段（phase 1）和第二、第三階段（phase2, phase3）就處於不同的執行個體內。

然而：爲求資料可被獨立處理，需要付出額外的編程心力（*Programmer effort*）。增量式（incrementally）處理資料會增加實作的區域複雜度（*local complexity*），原本這可藉由集中而整體地（*globally*）處理資料加以避免。如果你需要額外的環境資訊（context info.）以便處理資料，那麼管理這些資訊還會增加程式的整體複雜度（*global complexity*）。

「讀寫多份小型資料項」通常比「讀寫一份大型資料項」效率低，所以 *Data Files* 的執行效率（*runtime performance*）與「集中處理全部輸入」的作法比較之下相形見絀。重複存取次儲存裝置，會增加系統的電力耗損（*power consumption*），甚至可能縮短某些次儲存裝置（如 flash RAM 和軟碟）的壽命。資料檔本身的局限性（例如強制要求所有輸入必須先行排序，要求用戶或客端軟體必須管理檔案）會降低系統的可用性（*usability*）。

❖ ❖ ❖

實作（Implementation）

前面的 Wombat 例子闡述了作用於純資料檔的四種主要操作：

1. 簡單循序輸入（依次讀取各章）；
2. 簡單循序輸出（撰寫最終輸出檔案）；
3. 隨機存取（讀取交叉參考檔案）；
4. 循序輸出至數個檔案（撰寫臨時章節檔案）；

下面是使用資料檔案時需要注意的一些問題。

1. 增量式處理（Incremental processing）

操控純資料檔的一個簡單、常用的方法就是：從輸入設備循序讀取整個檔案，並（或）將另一個檔案循序寫至輸出裝置（見圖 2.5）。程式必須經過特殊裁剪才能以增量方式處理其輸入，因此需要付出額外的編程心力。由於程式以化整爲零的方式將大型檔案分成多份細小增量，因此程式員往往得負責遴選待處理之增量（此一工作也可以交給用戶，由用戶指定資料檔的增量邊界，或提供一些更小的資料檔集合）。

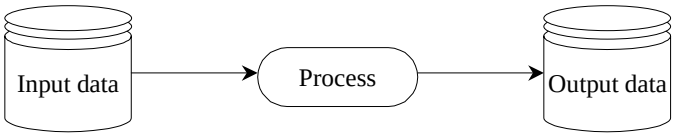


圖 2.5　增量處理（Incremental processing）

由於整個輸入檔是在單一行程中處理，因此增量處理會使「在各個處理階段（processing phases）之間維護全域環境資訊」更容易些，並且更容易產生最終輸出結果 — 最終結果正是由程式增量寫出而成。不幸的是，正因爲這是在一個長期運行的單一行程中進行，因此「將記憶體需求降至最低」也就比較困難。

2. 子檔處理（Subfile processing）

如果不以循序方式處理檔案，你也可以把資料劃分爲多個小型子檔（subfiles）。這時候你應該撰寫一個程式用以「處理單一子檔」並產出一個個獨立輸出檔。執行此程式數次（通常採用循序方式）以便處理每個子檔，然後合併所有子檔，產生所需的輸出（圖 2.6）

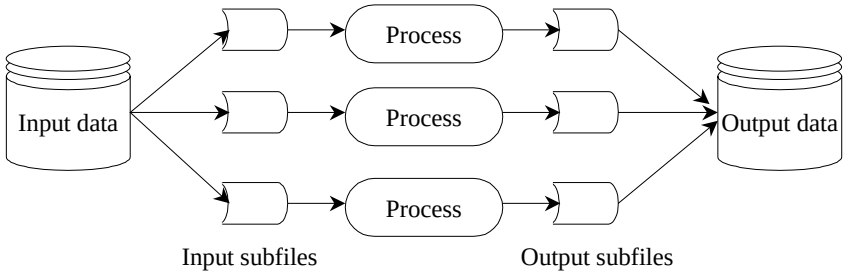


圖 2.6　子檔處理（Subfile processing）

如果資料的劃分很容易，那麼「子檔處理」擁有數個優點。由於每次只處理資料的一份子集，所以「子檔處理」需要的記憶體一般就少得多；由於資料檔的毀損和錯誤只會影響一個檔案，所以「子檔處理」對這些問題的抵禦能力更強。不幸的是，分解檔案有待程式員或用戶付出心力：「協調處理」和「合併子檔」都需要更多的編程心力。請參考 *Application switching*（84）對這類「行程間通訊」技術的討論。

許多編譯器都運用「子檔處理」技術：它們分開編譯各個程式檔，只在分離的連結階段（link phase）才把獲得的所有暫時目的檔（object files）合併起來。由於這項技術在「降低記憶體用量」上蘊涵巨大潛力，所以「子檔處理」在過去的批次磁帶處理（batch tape processing）中隨處可見（Knuth 1998）。

3. 隨機存取（Random access）

迥異於循序讀寫（無論是增量技術或子檔技術），你也可以隨機存取某一檔案，以任意次序甄選資訊、讀寫檔案（圖 2.7）。隨機存取通常比增量處理或子檔處理需要更多的編程心力，但它比後兩者遠爲靈活：你不需要預先決定資料項 — 它們甚至可能需要被分解爲子檔 — 的處理次序。

爲了運用隨機存取，每個行程必須能夠在次儲存裝置的檔案裡頭找出各筆資料項。一般來說你需要一份索引，列出每一筆資料距離檔案起始點的偏移位置。由於大部份檔案存取動作都需要用到此一索引，所以索引應該儲存在主記憶體內，或從主記憶體中可以便捷地取用。檔案的高效檢索本身就是一門大學問，但對於一般應用，有兩種直截了當的選擇：

- 檔案可以包含自身索引（或許位於檔案起始處），由行程讀入 RAM。*Resource files*（101）常常使用這種作法。
- 應用程式可以在啓動（start-up）時掃瞄檔案，生成自己的索引。Wombat 文字處理器藉助其交叉參考檔（cross-reference file）完成這項工作。

更複雜的系統可能會使用得以定位多個檔案的索引，甚至爲索引檔再建立一份索引。Folk et al.（1988）, Date（1999）和 Elmasri and Navathe（2000）等文獻都涵蓋有「檔案處理」這一主題。

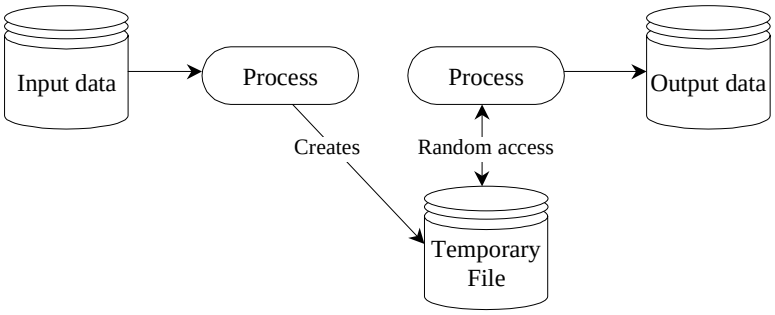


圖 2.7 隨機存取（Random access）

示例（Examples）

1. 簡單的子檔處理（subfile processing）

「檔案編譯」是子檔處理的一個典型示例。用戶把大型程式分解爲多個檔案，編譯器逐一處理每個檔案，然後連接器 `ld` 將所有 `.o` 輸出檔合併爲一個可執行程式 `testprog`：

```
cc main.c
cc datalib.c
cc transput.c
ld -o testprog main.o datalib.o transput.o
```

2. 增量處理（Incremental processing）

以下 Java 程式將輸入檔內的各行字元反轉。它把每一行讀入緩衝區，逆轉緩衝區內的字元，然後把緩衝區內容寫入另一個檔案。我們呼叫 `reverse()` 並傳入兩個參數 `BufferedReader` 和 `BufferedWriter`，用以取得標準 I/O 裝置 — 其效率高於直接使用硬碟讀/寫函式，但得付出一些記憶體成本：

```
reverse(new BufferedReader(new InputStreamReader(System.in)),
     new BufferedWriter(new OutputStreamWriter(System.out)));
```

由於 Java 的 `Strings` 不可修改，所以 `reverse()`使用兩個緩衝區（一個針對 `String` 另一個針對 `StringBuffer`）完成這項工作：

```
public void reverse(BufferedReader reader, BufferedWriter writer)
  throws IOException {
  String line;
  StringBuffer lineBuffer;

  while ((line = reader.readLine()) != null) {
       lineBuffer = new StringBuffer(line);
       lineBuffer.reverse();
       writer.write(lineBuffer.toString());
       writer.newLine();
  }
  writer.close();
}
```

這個例子的重點在於，它所需要的記憶體只要「足以容納 I/O 緩衝區以及一行待逆轉文字」即可，不需要「容納整個檔案」，所以它可以處理任意長度的檔案，絕無記憶體耗盡之虞。

3. 使用多個子檔進行處理（Processing with multiple subfiles）

現在讓我們考慮逆轉整個檔案的所有 bytes（而不僅限於每一行的 bytes）。由於前述的簡單增量技術仰賴「處理某行，不會影響其他任何一行」的事實，所以此處無法奏效。「逆轉檔案中的所有字元」會牽扯整個檔案，而不僅於單獨每行。

我們可以使用次儲存裝置上的子檔，藉此逆轉檔案，且不必將檔案全部放進記憶體。我們首先把大型檔案分解（`scatter()`）爲許多小型子檔，每個子檔都可以載入記憶體，然後逐一逆轉每個子檔。最後再以相反次序讀取（`gather()`）所有子檔，裝配成一個嶄新的、完全逆轉的檔案。

```
public void run() throws IOException {
   scatter(new BufferedReader(new InputStreamReader(System.in)));
   gather(new BufferedWriter(new OutputStreamWriter(System.out)));
}
```

爲了將檔案分解爲子檔，我們從輸入裝置 `reader` 讀取 `SubfileSize` 個 bytes 到緩衝區，而後逆轉緩衝區，而後再把緩衝區內容寫入一個新子檔：

```
protected void scatter(BufferedReader reader) throws IOException {
   int bytesRead;
   while ((bytesRead = reader.read(buffer, 0, SubfileSize)) > 0) {
        StringBuffer stringBuffer = new StringBuffer(bytesRead);
        stringBuffer.append(buffer, 0, bytesRead);
        stringBuffer.reverse();
        BufferedWriter writer =
             new BufferedWriter(new FileWriter(subfileName(nSubfiles)));
        writer.write(stringBuffer.toString());
        writer.close();
        nSubfiles++;
   }
}
```

每次逆轉檔案時都可以重複使用緩衝區（這是 *Fixed Allocation*, 226 的一個範例），但必須爲每個子檔產生一個新名稱。此外還應該計算所寫出的子檔數量，以便稍後可以將它們重新匯聚起來：

```
protected char buffer[] = new char[SubfileSize];

protected String subfileName(int n) {
   return "subxx" + n;
}
protected int nSubfiles = 0;
```

最後，我們把所有子檔匯聚爲一體。由於子檔都已經是逆序的，所以只需從最後一個子檔開始，打開每個子檔並讀取其內容，寫至一個輸出檔即可：

```
protected void gather(BufferedWriter writer) throws IOException {
   for (nSubfiles --; nSubfiles >= 0; nSubfiles --) {
        File subFile = new File(subfileName(nSubfiles));
        BufferedReader reader =
             new BufferedReader(new FileReader(subFile));
        int bytesRead = reader.read(buffer, 0, SubfileSize);
        writer.write(buffer, 0, bytesRead);
        reader.close();
        subFile.delete();
        }
   writer.close();
}
```

❖ ❖ ❖

已知應用（Known uses）

大部份編程語言都運用子檔處理技術來進行編譯。典型如 C, C++, FORTRAN, COBOL，一次編譯一個檔案，所有編譯階段（compilation phases）結束後，再以一個連結階段（link phase）將編譯所得的目的檔（object files）合併起來。C 和 C++ 還強迫程式員以表頭檔（header files）形式管理編譯過程所需的共享資料（Kernighan and Ritchie 1988）。Java 以相同手法編譯每個獨立的 .java 檔，不再有所謂的連結階段，而是（通常）將 .class 檔以 *Compression* 手法合併爲 JAR 檔（Chan et al. 1998）。

UNIX 環境鼓勵程式員使用純資料檔。它提供許多簡單的過濾器程式如 `wc`, `tee`, `grep`, `sed`, `awk`, `troff`（Kernighan and Pike 1984）。程式員可以利用 pipes（管線）將它們合併起來使用；作業系統安排每個過濾器每次只需處理很少量的資料。

許多膾炙人口的應用程式都使用純資料檔，並讓用戶明白看見當前檔案的名稱。Microsoft 的 MFC framework 在其 Document-View 架構中將這種設計奉若神明（Prosise 1999），支援多重文件（multiple documents），並令每一份文件對應一個資料檔。EPOC 的 AppArc 架構（Symbian 1999）一次只支援一份文件，其檔名對用戶不一定可見 — 視特定環境之視覺設計而定。

某些文書處理器和格式編排工具支援子檔案（subfiles），例如 Microsoft Word、TeX 和 FrameMaker（Microsoft Word 1997; Lamport 1986, Adobe 1997）。用戶可以產生一個主控文件（master document），以之參考一系列子文件；子文件被獨立編輯，列印時各子文件被逐一載入記憶體並列印。應用程式只需在記憶體中保持少量「跨子文件」之全域狀態即可。

EPOC 在每個檔案內支援所謂的 stream 子檔案；不同的 stream 以不同的整數 ID 標識之，

並以簡單的永續機制存取之。這使得「產生多個 output streams 並個別存取之」變得簡單易行。許多 EPOC 應用程式都使用這項特性。「在大型物件上運用永續性（persistence）」的組件，通常把每個物件保存在獨立的 stream 裡頭，因而得以推延物件的載入時間，直到實際需要該物件爲止。template class `TSwizzle` 提供了一個 *Multiple Representation*（209），使客戶端程式碼無法看見其中細節（Symbian 1999a）。EPOC 上的關聯式資料庫爲大約每 12 筆 "database rows" 產生一個新 stream，也爲它所儲存的每一個二進制物件（binary object）產生一個新的 stream。這麼一來 DBMS 伺服器要修改資料庫內的資料就易如反掌了：只要撰寫一個新的 stream 取代既有的 stream，並更新資料庫內「面向所有 streams」的索引（Thoelke 1999）。

印表機驅動程式（特別是內嵌於 bitmap-based 印表機內的驅動程式）經常使用 'banding' 技術：每次僅著色並列印一部分頁面。這種技術可以減少必須儲存的輸出圖像大小，但也降低了列印速度，因爲每一頁必須著色數次，每次一個 band（一個帶狀區域）。

參見（See Also）

Resource Files（101）可以替代唯讀資料。*Paging*（119）的實作雖然十分複雜，對程式員而言它卻十分簡明。*Data Files* 可以令 *Application switching*（84）的實作更加容易，你可以運用 *Compression*（135）把各個子檔儲存在 *Secondary Storage* 裡頭。

你可以單獨運用 *Fixed Allocation*（226）或 *Memory Discard*（244），抑或聯結使用它們，用以處理從 *Data File* 讀出來的各筆資料。

Pipes and Filters（Shaw and Garlan 1996）描述了一個基於各種過濾器（filters）的軟體架構風格。

也許簡單的 *Data File* 不能滿足你，也許你需要一個全功能資料庫（Connolly and Begg 1999; Date 1999; Elmasri and Navathe 2000）。在這個主題上，Wolfgang Keller 和 Jens Coldewey（1998）奉獻了一組「將物件由 OO 程式儲存至關聯式資料庫」的範式。

Resource Files（純資源檔）

如何管理爲數眾多的組態資料（configuration data）？

- 許多程式資料是一些唯讀組態資訊，程式不可更改之。
- 組態資料通常比程式碼變化更頻繁。
- 資料可能被程式的不同階段（phases）取用。
- 任何時候你都僅僅需要少數幾筆資料。
- 檔案系統支援隨機存取，使「單獨載入檔案某一部分」輕而易舉。

有時候程式的記憶體需求（*memory requirements*）包括爲「大量唯讀靜態資料」配置的空間；通常程式每次只使用其中少量部分。例如 Word-O-Matic 需要許多靜態資料，包括視窗佈局、圖標設計、字體尺度、拼字檢查字典。大部份這類資訊有可能在任意時刻被程式索求，但往往只被使用很短一段時間。如果這些資訊被儲存在主記憶體內 — 也許你把它編死到程式內部 — 它就會增加程式的記憶體總體需求。

猶有進者，你很可能需要脫離程式自行修改組態資訊。程式的不同變體（不同的語言，不同的用戶介面主題思想, user interface themes）使用同一份程式碼，但需要不同的組態資料，這時局面會怎樣？在某特定組態下，特殊資料項也許須臾不可或缺，例如視窗格局或字型，所以你不能採用 *Application Switching*（84）來帶入這些資料。然而一般言之，大部份資料在任何特定時刻下並未被使用。

因此：在次儲存裝置（secondary storage）內保存組態資料，必要時才載入或拋棄其中某一項。

作業系統提供了一種儲存唯讀靜態資料的簡單方法：將它保存在次儲存裝置檔案裡頭。檔案系統提供隨機存取，因此「只讀取檔案某一部分，忽略其他部分」不過是舉手之勞。你可以把檔案的一部分載入臨時記憶體，使用一會兒，然後丟棄之；如果你需要它，總可以再次取回。事實上只要稍稍增加一些複雜度，你就能夠把一個檔案轉化爲一個唯讀資料庫，其中每筆資料各與一個獨一無二的標識符號關聯。

所以，你不必刻意將各筆資料編死到程式碼裡頭，你只需賦予每筆資料一個獨一無二的標識符號。一旦程式需要該筆資料，就喚起一個特殊函式並傳給它標識符號；這個函式會從「資源檔」內讀取資料，並將資料傳回給程式。一旦程式不再需要這筆資料，就丟棄它。典型的「資源」（resources）有：

- 字串（strings）
- 螢幕佈局（screen layout）
- 字型（fonts）
- 位元圖（bitmap）、圖標（icon）和滑鼠（mouse）

舉個例子， Word-O-Matic 的所有視窗佈局、圖標設計和字串都儲存在資源檔（resource file）裡頭。需要它們時 Word-O-Matic 便從資源檔取出資料，儲存在一個臨時記憶體緩衝區內。一旦不再使用該資料，緩衝區便被收回復用。

結果（Consequences）（譯註：本段保留之 *Italic Times New Roman* 字型均爲 forces 名稱）

唯讀靜態資料並不塞滿於主記憶體內，因此可以降低程式的記憶體需求（*memory requirements*）。多個程式可以共享同一個資源檔，減少相關的編程心力（*programmer effort*）。某些作業系統會在同一個程式或程式庫的多個實體之間共享載入後的資源，進一步降低記憶體需求。這也使「不改動程式，只改變資料」（例如支援多語文字串），變得更容易些，提高程式的設計品質（*design quality*）。

然而：爲了能夠將資源安插至資源檔，並正確載入和釋出資源，本方法需要良好的編程素養（*programmer discipline*）。載入和卸載資源檔會略微降低程式的時間效率（*time performance*），特別是會對啓動時間（*start-up time*）造成衝擊。由於你需要某種機制來卸載（以及重新載入）資源，所以資源檔的實作也需要付出額外的編程心力（*programmer effort*）。如果作業系統提供這種支援，那是再妙不過了。

❖ ❖ ❖

實作（Implementation）

由於資源檔必須被隨機存取，所以應用程式需要一份索引，用以決定資料座落位置（參見 *Data Files*, 92）。大部份資源檔實作時將此索引置於資源檔自身之內，通常位於檔案起始處。這也就意味資源檔不再是人類可讀的文字格式，使用前必須經過編譯。資源編譯器通常也把資源轉化爲「應用程式碼得以輕鬆管理」的二進制格式，減少記憶體佔用量，提升應用程式的效率。

實際過程中你常常需要系統內不同資源之間的邏輯分隔：組件 A 所用的資源和組件 B 所用的資源可能截然不同，不同的團隊，其責任亦不相同。因此大部份資源檔框架（resource file frameworks）一次支援一個以上的資源檔。

實作資源檔案時，需要考慮以下問題。

1. 必須讓程式員得以輕鬆運用

載入資源並解釋其格式，這樣的任務絕非泛泛，所以大部份系統都提供了相關程式庫。你需要一些基本函式用來載入和釋出資源；通常也許還有更複雜的函式可以：

- 管理來自多個檔案的資源的載入和釋出；
- 根據載入的資源建立圖形對話框和構件；
- 直接從檔案向螢幕傳送 bitmap 和繪畫資源（字型、圖標、滑鼠、繪圖基本元素），不必向程式暴露其結構；
- 將參數插入資源中的字串。

只是在執行期間將資源載入和卸載，這是不夠的；你必須首先生成資源。編程環境也會提供工具，幫助你建立資源：

- 資源編譯器 — 它可以從一個文本檔案表述內容中創建出資源檔資料庫；
- 對話框編輯器。這些編輯器允許程式員使用「用戶控件」（user controls）來進行螢幕繪製；然後程式員便能夠從其結果產生資源檔文字描述（resource-file descriptions）。Microsoft Developer Studio 就是一例（見圖 2.8），此外還有許多資源編輯器。

2. 運用資源檔，節約記憶體

某些資源檔系統支援 *Compression*（135）。這得爲每一個被載入的資源付出一點時間成本，但可以減少檔案佔據空間。不過 *Adaptive Compression*（160）演算法並不適合壓縮整個檔案，因爲其實施前提是「不依賴檔案的其餘部分而能對任何資料獨立解碼」。如果單一資源夠大（例如圖像或聲音），你可以個別壓縮之。

付出一些努力，領會你的系統中的「資源載入」究竟如何進行，絕對物有所值，因爲這對記憶體的節約常常大有裨益。舉個例子，MS Windows 支援兩類資源：`PRELOAD`（預先載入）和 `LOADONCALL`（呼叫時才載入）。前者在程式首次執行時被載入，後者在客端

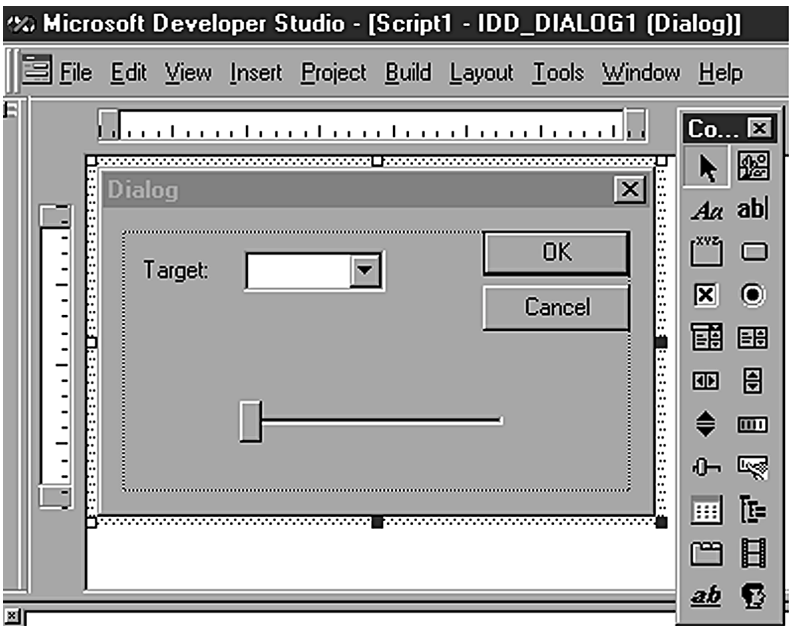


圖 2.8 對話框編輯器

程式碼明確請求後才載入。顯而易見，如果想節省記憶體，你應當更青睞 LOADONCALL。類似道理，Windows 3.1 並不載入單一字串，而是以「16 個 ID 號碼相連的字串所組成的區塊」爲載入單元。因此你可以精心安排字串的分塊，使同一區塊內的字串一起被使用，進而把記憶體用量降至最低。與之形成鮮明對比的是，Windows LoadIcon()函式本身並不存取資源檔，而是在螢幕驅動程式需要圖標（icons）時才會執行存取動作。所以呼叫 LoadIcon()並未使用多少記憶體。Petzold（1998）詳細討論了 Windows 資源檔對記憶體的運用情況。

3. 字型檔（Font files）

和其他類型的資源相較之下，你經常會希望以截然不同的方式來對待字型。首先一點，所有應用程式共享同一份字型集，而且字型描述往往比其他資源龐大得多。一個精巧的字型處理系統只會載入每種字型之中特定應用程式所需的部分：也許只是特定大小的字型實體，也許只是應用程式爲了某個特定字串而申請的字元。後一種方法尤其適合 Unicode 字元所使用的字型，因爲其中可能包括數千個圖像（Pike and Thompson 1993）。

4. 實作一個資源檔案系統（resource file system）

有時你需要實作自己的資源檔案系統。這時候請考慮以下問題。

4.1 **遴選變數**。系統如何決定應該載入哪一版本的資源檔？有各種選擇。有些系統每次發表總是只含一份資源檔，另一些系統（例如大部份 MS Windows 應用程式）支援多種語言，但只安裝一種語言版本；對於這種系統，更改語言意味覆蓋（overwriting）原有檔案。還有一些系統在程式初始化階段選擇適當版本，例如 EPOC 透過副檔名選擇版本（如果當前語言是 01 號，則應用程式 Word 便載入資源檔 `WORD.R01`）。其他系統甚至允許系統動態改變語言，雖然這毫無疑問會帶來應用程式之間複雜的交互影響。

4.2 **將參數插入字串**。資源最常用於字串。如今，顯示出來的字串常常包含可變參數：「你有 NN 件事情要做，CC」，其中數字 NN 和名稱 CC 依程式需求而變化。該如何插入這些參數呢？

常見手法是採用 C 語言的 `printf` 格式：「你有`%d` 件事情要做，`%s`」。這種方式合情合理，但有兩個嚴重缺點。第一，如果傳給資源字串的參數非其所需，那麼 `printf` 及其變體的一般實作品很容易就造成程式崩潰。因此一個有著訛誤或馬馬虎虎構建起來的資源檔會造成無法預期的程式錯誤。第二，支援不同語言之系統構建時，`printf` 格式並不十分靈活，例如德國人可能希望看到兩個參數的次序相反：「`%s`：你有`%d` 件事情要做」。

更具彈性的作法是在資源字串裡頭使用編號字串：「你有`%1` 件事情要做，`%2`」。程式碼負責把所有參數轉換爲字串（這很簡單，能夠依地域或國家而有相應的考量），再由一個標準函式把字串插入資源字串內。如果被傳遞字串的號碼與資源字串不相吻合，實作出一個「提供預設行爲」或「顯示錯誤訊息」的函式，是件很簡單的事。

示例（Examples）

下面是一個擁有 "about" 對話框的 MS Windows 資源檔：

```
// About Box Dialog
//

IDD_ABOUTBOX DIALOG DISCARDABLE 34, 22, 217, 55
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "About DEMOAPP"
FONT 8, "MS Sans Serif"
BEGIN
```

```
    ICON                2,IDC_STATIC,11,17,18,20
    LTEXT               "Demonstration Application by Charles Weir",
                                IDC_STATIC,40,10,79,8
    LTEXT               "Copyright \251 1999",IDC_STATIC,40,25,119,8
    DEFPUSHBUTTON "OK",IDOK,176,6,32,14,WS_GROUP
END
The C++ code to use this using the Microsoft Foundation Classes is remarkably
trivial:
////////////////////////////////////////////////////////////////////////
/
// CAboutDlg dialog

CAboutDlg::CAboutDlg(CWnd* pParent /*=NULL*/)
    : CDialog(CAboutDlg::IDD, pParent) {
    //{{AFX_DATA_INIT(CAboutDlg)
            // NOTE: the ClassWizard will add member initialization here
    //}}AFX_DATA_INIT
}
```

請注意註釋中的寫法：`{{AFX_DATA_INIT(CAboutDlg)`，這種寫法使得其他 Microsoft 工具和 'wizards' 得以辨識位置；Wizard 可以確定對話框中的任何變數欄位，於是插入程式碼將這些欄位初始化，俟對話框完成後再取回欄位值。本例沒有這一類變數，故而沒有程式碼。

❖ ❖ ❖

已知應用（Known uses）

所有 Apple Macintosh 程式和 MS Windows GUI 程式都以資源檔儲存 GUI 資源，尤以字型爲多（Apple 1985; Petzold 1998）。EPOC 把所有和語言相依的資訊（包括經過壓縮的 help 文字）都儲存在資源檔內，容許系統在執行期間選擇適當語言。EPOC 的「Unicode 字型處理」使用一個 *Fixed Allocation*（226）（Edwards 1997）緩衝區，儲存字型圖像的快取集合（cached set），將字型常式的記憶體使用量降至最低。EPOC16 利用 *Compression* 縮小資源檔。

許多計算機遊戲使用 *Resource Files* — 從附帶 static ROM 的手持設備、配備磁帶和軟碟的早期微型計算機，到最先進的 CD-ROMs 遊戲控制台，無一例外。本範式使它們可以提供更多的螢幕、關卡（遊戲中的 levels）、或地圖，那是主記憶體根本容納不下的東西。每一「關」都是次儲存裝置上的一個獨立資源檔，玩家過關斬將殺至某一關時，相應的檔案就被載入。任何時刻玩家都只可能在某一關中鏖戰，所以記憶體需求就減至一關所需要的儲存量。由於各關之間的順序始終可以預測，所以這種方式在 arcade-style 遊戲中運作良好 — 在這種遊戲中，玩家先攻一關，如果獲勝就進入下一關，如果失敗則一命

嗚呼。同樣道理，許多多用戶（multi-user）探險遊戲都保存著特定遊戲的細節：資源檔位置、區域性規則、妖魔鬼怪及武器裝備等等；這些資訊往往很龐大，所以常常以 *Compressed*（135）儲存。

參見（See Also）

Data Files（92）提供可寫資料儲存區（writable data storage）；*Resource Files* 對不可修改之資料施行什麼操作，*Application switching*（84）和 *Packages*（108）就對程式碼如法炮製。

所有資源檔都可以儲存在 *Secondary Storage*（79）或 *Read-only Memory*（65）裡頭，同時還可能使用 *Compression*。

Petzold（1998）和 Microsoft（1997）講述了 Microsoft Windows 資源檔。Tasker et al.（2000）講解了 EPOC 資源檔的使用。

Packages（封包）

另名: Components, Lazy Loading, Dynamic Loading, Code Segmentation

如何管理「有許多可選成份（optional pieces）」的大型程式？

- 你沒有足夠的記憶體空間容納全部程式碼和靜態資料。
- 系統有許多功能，但不會被同時使用。
- 你可能需要不同功能的任意組合。
- 如果已發展的組件之間有著明顯的分隔，發展將會最順利。

一些大型程式多數時間內其實只是小型程式 — 全部程式碼的記憶體需求遠遠大於任何一次特定運行中程式碼對記憶體的實際用量。例如 Strap-It-On 的 Spin-the-Web 網頁瀏覽器可以同時觀看多種不同類型的檔案，但通常它只閱讀供其 help 系統使用的 StrapTML 本地網頁。「支援其他檔案類型」會提高程式碼的記憶體需求量 — 即使不需要這些程式碼時也如此。

這種類型的應用程式無法事先預測你會需要哪些特性，也無法對這些特性進行排序以便每次可以只使用其中一項。因此 *Application Switching*（84）範式在此無能爲力，但你仍然希望享受該範式帶來的福利 — 藉由「在同一時間不將全部程式載入主記憶體」而降低系統的記憶體需求。

因此：把程式分解爲多個 packages（封包），只在系統需要某個 package 時才載入它。

所有「將程式碼儲存於硬碟檔案中」的執行期環境（runtime environment）都必須擁有一套機制，將「從硬碟載入的可執行檔」激活（activate）起來。只需付出相對較小的努力，你就可以擴展這套機制，使它載入額外的可執行碼到正在運行的程式中。當然啦，當「大部份程式運行過程中無需載入上述額外程式碼」時，此一作法才有價值。

你需要把程式分解成一個主程式和一個由「可被獨立載入之 packages」組成的群集。主程式被載入而後並開始運行；當它需要某個 package 時，系統某處的一段程式碼就載入適當的 package，並直接呼叫該 package。

例如，Spin-the-Web 的核心就是個主程式，其中分析每個 Web 頁面，並把適當的閱讀器（viewer）當作 package 載入進來。

結果（Consequences）（譯註：本段保留之 *Italic Times New Roman* 字型均爲 forces 名稱）

某些未被使用的程式碼始終存放於 *Secondary Storage*（79），因此記憶體需求減少了。

由於初始化時只需載入小小的主程式，因此啓動更快，而且能夠以更少的記憶體就開始執行。由於每個 package 都相當小巧，所以後續的 packages 可以被迅速載入，無須爲了更換程式階段（phases）而暫停 — 這種事情在 *Application Switching*（84）中會發生。

由於 packages 並非靜態連結至應用程式碼，因此動態載入機制允許第三方或後續開發人員不必更改（或甚至停止主程式）就可以增加功能，明顯提高了系統的可用性（*usability*）和維護性（*maintainability*）。

然而：把程式分解爲 packages 需要付出相當的編程心力（*Programmer effort*）。

許多執行環境從不卸載 packages，造成程式的記憶體需求（*memory requirements*）持續增加。除非任何一次特定運行都僅使用一小部分功能，否則程式仍然可能耗盡記憶體。以下行爲都需付出相當的編程心力（*programmer effort*）：實作動態載入機制、令 packages 遵循該機制、定義「何時載入及卸載」策略、將 packages 的分解辦法最佳化、將載入所需的額外開銷最小化。這個機制常常可以跨程式而被復用，或由作業系統提供；但是另一方面，許多執行環境卻完全不支援動態載入。

因爲 packages 不會不請自來，所以「動態載入」意味「直到程式載入完畢，系統才可能偵測到 packages 丟失」；這稍稍降低了程式的可用性（*usability*）。而且，如果 packages 的存取速度比較慢（例如透過 web），載入 packages 所花費的時間會降低程式的回應速度，也就降低了程式的可用性。這種反覆無常的延遲也使 *Packages* 不適用於即時（*real-time*）操作。

你可以遠程定位 packages，並在主程式之外修改它們。這會產生安全方面的隱憂，是的，一個充滿敵意的代理商可能會藉由修改 packages 而將病毒或安全漏洞引入系統。

❖ ❖ ❖

實作（Implementation）

欲支援 packages，需三寶俱足：

1. 你需要一個系統，能夠將程式碼載入 RAM 內並執行之；

2. 你必須能夠將軟體分割爲 packages，使正常情況下只有其中一個子集處於活躍狀態；
3. 你必須支援「可被動態載入」的 packages — 通常是一些「與位置無關」或「可重定位（relocatable）」的目的碼（object code）。

當你將 packages 視爲系統的一部分而使用它或實作它時，要考慮一些問題。

1. 把行程（process）視爲 packages

也許 package 的最單純形式就是一個獨立行程。儘管我們可能得付出效率和程式複雜度兩方面的龐大代價，但經過審慎的編程，我們可以讓用戶感覺「同時運行的兩個行程（兩個 packages）」就和一個行程沒有兩樣。將 package 實作爲獨立行程，有數個關鍵優點：

- package 和主程式各自運行於獨立位址空間中，因此 package 的致命錯誤不一定會終止主程式。
- 不再使用 package 時，只需終止其對應行程，就可輕鬆丟棄配給 package 的記憶體。
- 某些情況下，用戶亟需之 package 可能已經以應用程式的形式獨立存在了。例如我們可能希望獲得某些 packages 進行文書處理、繪圖或試算表管理。這樣的應用程式早已存在，通常被實作爲獨立行程。

將行程當作 package 來使用，有兩種常見作法。

1. 客戶可以採用「與作業系統 shell 相同的方式」來執行行程。行程會被執行直至結束，也許會讀取其標準輸出（見 *Data Files*，92）。
2. 客戶可利用作業系統的「行程間通訊」（Inter-Process Communication, IPC）機制，和行程進行溝通。

第二種作法已經被某些 Microsoft ActiveX（'COM'）框架、IBM's System Object Model（SOM）框架以及以 CORBA 爲基礎的各個框架採用了（Box 1998; Szyperski 1999; Henning and Vinoski 1999; Egremont 1998）。它們都運用某種形式的 *Proxy*（Gamma et al. 1995; Buschmann et al. 1996），賦予客戶能力使能存取 package 物件內的物件。環境的討論和比較請見 *Essential Distributed Objects Survival Guide*（Orfali et al. 1996）。

2. 把動態連結程式庫（DLL）當作 C++ packages

你也可以考慮把共享的或動態連結的程式庫（DLLs）當作 packages 使用。通常一個執行個體在初始化階段會載入其所有 DLLs，因此預設情況下 DLLs 並不像 packages。然而，大部份環境都提供了附加機制，可以在執行期間動態載入和卸載 DLLs。

某些框架（frameworks）使用「DLLs 延遲載入機制」。例如你可以把 Microsoft COM 物件實作成「當物件首次被取用時才自動載入」的 DLLs。雖然 COM 的設計使用了 C++ virtual function tables（虛擬函式表），但其他許多語言也提供了存取 COM 物件所需的黏合劑（Box 1998）。

另有一些環境只是簡單提供「把 DLL 檔案載入 RAM，並喚起 DLL 內的函式」的機制。如何以這種機制實作 packages 呢？

通常你可以透過函式名稱或序號（第一個被匯出函式的序號爲 0，而後是 1，而後是 2…，依此類推），指明 DLLs 中的「外部可呼叫匯出函式」。無論採用哪種方法，「爲所有供客戶使用的函式提供印記（stubs）並將之修補爲 DLL 上的正確位置」都可謂任務艱巨。

你可以改用物件導向的動態繫結（dynamic binding）來減輕工作。這種方法只需對某個 DLL 進入點（entry point，通常在索引 0 或 1 處）進行一次呼叫；該函式應該返回一個指標，指向某個 class 實體，該 class 支援一個爲客戶所知的介面。此後客戶便可以呼叫該實體所具備的函式；OO 語言對動態繫結的支援是：唯有正確的程式碼才會執行。一般而言上述所說的 class 是個 *Abstract Factory*，或是提供了 *Factory Methods*（Gamma et al. 1995）。圖 2.9 展示一個這樣的程式庫以及它所支援的 classes。

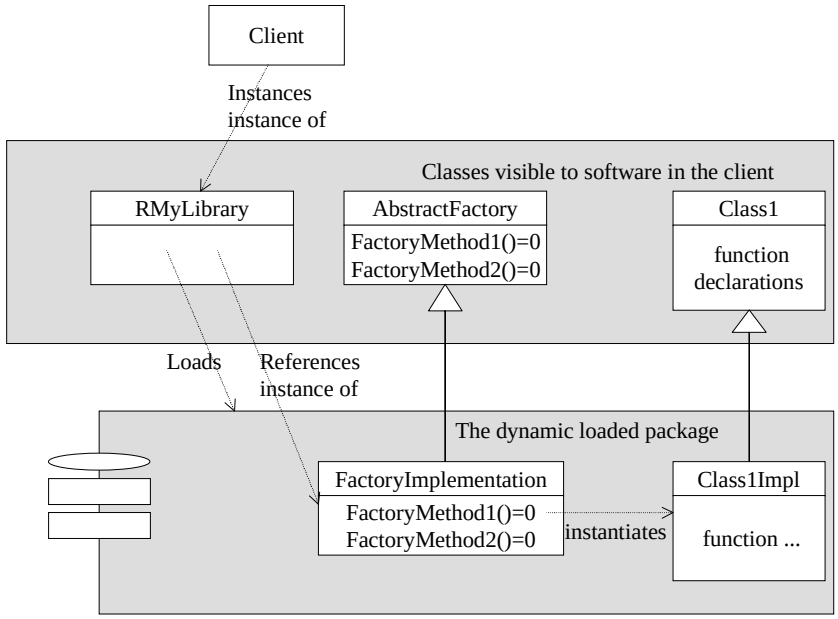


圖 2.9　Abstract Factory 和「被動態載入的 packages」

3. 以 code segmentation（程式碼分段）實作 packages

許多處理器及作業系統都提供 *code segmentation*。這使我們得以在機器碼或目的碼（object code）層面支援 packages。所謂分段式架構（segmented architecture）係把程式和其所取用的資料視爲一些相互獨立的 segments，而非渾然一體的大塊記憶體空間（Tannenbaum 1992）。

通常每個 segmetns 都有自己的記憶體保護屬性 — data segments 可能被單一行程讀寫，來自共享程式庫的 code segments 可被所有行程讀取。有了 packages，各個 segments 就可以被作業系統置入（或取出於）次儲存裝置 — 也許是自動進行，也許在程式員操控之下。分段式（segmented）系統的連結器所產生的程式被劃分爲多個 segments，同樣可能自動進行，或是在程式碼內的指令（directives）的控制下。

許多比較古老的 CPUs 都明白支援 segmentation，運用數個 segment registers（分段暫存器）來加速 segments 的存取，並確保 segments 中的程式碼或資料被存取時與 segments 的實際記憶體無關。處理器的局限往往限制了 segments 的最大尺寸（8086 架構中最大尺寸爲 64K）。現代化處理器傾向於把 *Packages* 和 *Paging*（119）結合起來。

4. 載入 packages

如果此刻你並未使用 code segmentation 或 Java packages，沒法子，你必須在每個應用程式的某些地方寫點程式碼來載入 packages。有兩種標準載入法。

4.1. **手工載入**。客戶明白載入 packages。以下場合使用此法最適宜：

1. 客戶必須從數個帶有相同介面的 packages 中指出需要哪一個（例如載入一個印表機驅動程式）。或者
2. 程式庫提供相對簡單的功能，而且卸載時機十分明確。

4.2. **自動載入**。客戶呼叫程式庫支援的任何函式。這些函式實際上是客戶提供的 stub（戳記）；當它被呼叫時，它會載入程式庫並喚起適當進入點。以下場合使用此法比較好：

1. 你要爲客戶提供一個簡單介面，或者
2. 多個 packages 之間存在錯綜複雜的相互依賴關係，所以不存在簡單的演算法可以決定何時載入 packages。

以上兩種方法都很常見。Microsoft's COM 框架和大部份 EPOC 應用程式都執行顯式載入；Emacs 文字編輯器則進行自動載入（Box 1998; Tasker et al. 2000; Stallman 1984）。

5. 卸載（Unloading）packages

如果有某種機制能夠卸載不再被使用的 packages，那麼許多記憶體便都得以節省下來。爲實現這個目標，你需要一種方法得以偵測已被載入之程式碼何時不再被需要。OO 環境中這很容易決定：一旦不再有 packages 支援的任何物件實體，已載入的程式碼就不再被需要了。因此你可以運用 *Reference Counting*（268）或 *Garbage Collection*（278）來決定何時卸載程式碼。

載入程式碼頗耗費時間，因此對某些產品來說，即使客戶通知它們可以卸載 packages，它們仍然選擇推遲。理想情況下，當系統記憶體漸顯匱乏時，它們就必須卸載這些先前被推遲了的 packages — 請參考 *Captain Oates*（57）範式。

6. 版本控制（version control）及二進位相容（binary compatibility）

你必須確保每個被載入的 packages 都能和其載入者（某組件）正確無誤地運作 — 即使這兩塊程式碼發展、發貨於不同時間。這種需求常常被稱作「二進位相容」，與「原始碼相容」截然不同。「二進位相容性」的需求既依賴語言又依賴編譯系統，典型包括：

- 客戶端新版本期望看到相同的外部進入點、相同的參數和返回值；新的服務（services）必須能夠提供它們。
- 新客戶不增加額外參數；新服務不增加額外回返值。這與 subtyping 規則有關 — 參見 Meyer（1997）。
- 新服務支援一如以往的外部可見狀態。
- 除非現行客戶有辦法處理，否則新服務不增加新的異常或錯誤狀況。

發展專案的過程中，當多個團隊平行發展數個 packages 時，版本控制可能成爲一個主要的頭痛問題。例如 Java 並未提供內建機制保證兩個 packages 二進位相容；不相容的版本往往可以被載入，但會產生微妙的程式缺陷。爲解決這個問題，某些環境在程式庫中提供了版本控制。例如 Solaris 支援其 DLLs 的版本主號與次號；次號變化並不影響二進位相容性，主號變化則不然。

Drossopoulou et al.（1998）詳盡討論了 Java 中的規則。Symbian Knowledgebase（2000）則討論了 C++ 二進位相容性的規則。

7. 最佳化（Optimizing）packages

如果你使用 *Packages*，那麼任意特定時刻就只會有一小部分程式碼和資料位於記憶體內，也就是所謂的 working set（工作集）。使用什麼技術可以使 working set 保持最小？你應當確保「一併被使用的程式碼」儲存在相同的 packages 裡頭。令人遺憾的是，雖然

「以 classes 和 modules 爲依據，對編譯後的程式碼進行最佳化」可謂善始，但並非上上之策。例如 Strap-it-On 的 Mind-Mapping 應用程式裡頭有許多可視物件，每一個都擁有諸多功能：(1) 可根據含混的文字描述生成自我、(2) 可爲螢幕上的動畫著色、(3) 可以詭秘刺激的方式交談、(4) 可將自身保存至儲存器並再次回復自己。然而施行於 Mind-Mapping 上的一個典型操作僅僅使用這些功能中的一種，而非使用所有 classes（圖 2.10）。

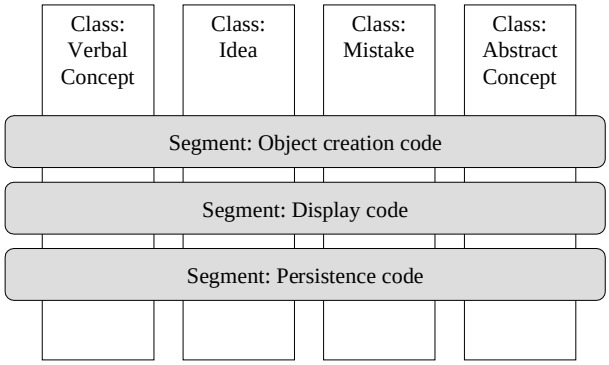


圖 2.10 實例 — class 的劃分並未能夠產生適當的 segments

你可以重新組織程式碼，使編譯單元對應於你所希望的 segment — 不過結果很難讓程式員加以管理和維護。借用 Soni et al.（1995）的說法，問題出在我們必須根據系統的執行架構（*execution architecture*）來組織「編譯後的程式碼」，而原始碼卻是根據概念架構（*conceptual architecture*）組織起來的。多數開發環境都提供「展示執行架構」的分析器，因此由程式員決定 segmentation structure（分段結構）是可行的 — 只是需要付出一些編程心力（*programmer effort*）— 然而它們究竟如何實作呢？

某些編譯環境提供了解決方案。例如 Microsoft 的 C++ 編譯器和 DEC 的 FORTRAN 編譯器，都允許用戶把每個編譯單元劃分爲多個由單一函式構成的編譯單元，稱爲 `COMDAT`s。然後程式員就可以運用連結選項 `/ORDER:@filename` 將這些單元排列成適當的 segments（Microsoft 1997）。Sun 公司的 SparcWorks' `analyzer` 工具令此程序的自動化程度更上層樓，允許以不同的 segmentation 選項加上分析所得的數據進行實驗，並提供一個工具程式（`er_mapgen`）直接根據這些試驗產生連結器的 'map' 檔。

對於那些缺乏上述選項的連結器，另一個替代方法是：預先處理源碼，爲每個函式產生一個檔案，然後在連結器命令列中明白地對結果檔案進行排列。此法需要額外的編程素養（*programmer discipline*），因爲它不讓程式碼和資料對每個源碼檔而言形成區域化（譯註：技術上無體會；原文如下：it prevents you making code and data local to each source file）。

示例（Examples）

這裡的 EPOC C++ 實例做出了圖 2.11 所示「使用 *Abstract Factory*，動態載入 packages」的方法。此組件在一個虛擬實境（virtual reality）應用程式中使用動畫物件；動畫物件種類繁多（人物、機械、動物、河流等等），每次只有少數幾種需要登台亮相，新實作品將在以後增加。於是動畫物件的實作品棲身於 packages 中，一有需求就被載入。

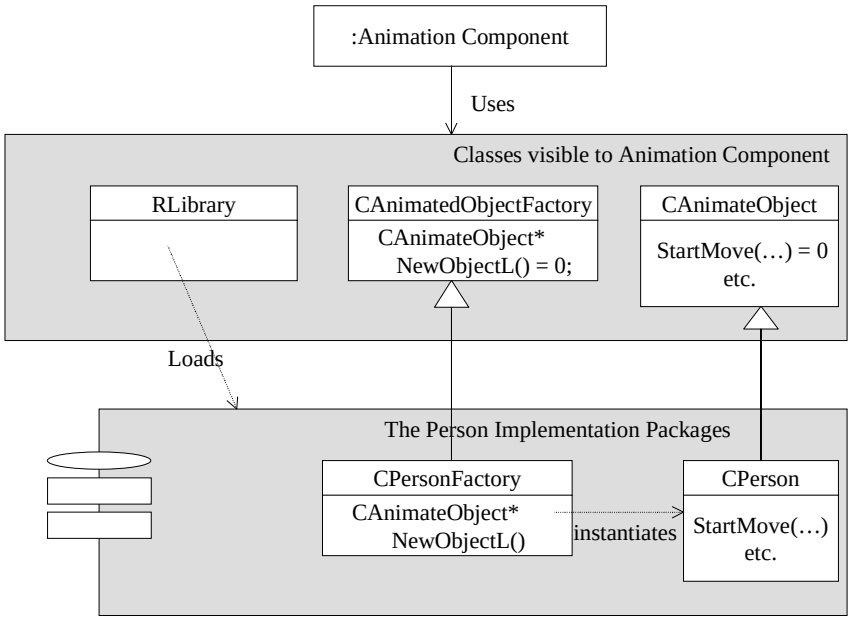


圖 2.11 示例用的 classes

1. 動畫組件的實作

EPOC 以 DLLs 實作 packages。動畫組件必須載入 DLL；任何 DLL-based 物件只要仍使用 DLL 程式碼，就得一直保留一個「指向 DLL」的 handle。它可能會以 C++ 產生一個新的、和下面差不多的 `CAnimatedObject`（在這裡，當前物件的生存期比 packages 裡任何物件的生存期都要長，`iAnimatedObjectFactory` 是型別 `CAnimatedObjectFactory*` 的一個變數）

```
iAnimatedObjectFactory = CreateAnimatedObjectFactoryFromDLL(fileName);
CAnimatedObject* newAnimatedObject =
   iAnimatedObjectFactory->NewObjectL();
```

`CreateAnimatedObjectFactoryFromDLL()` 實作如下。它把 EPOC 的 class `RLibrary` 做爲程式庫的 handle；函式 `RLibrary::Load()`用來載入程式庫；`RLibrary::Close()`

卸載之。就像所有的 EPOC 程式碼一樣，如果載入失敗就必須實作 *Partial Failure*（48）。`libraryHandle` 是個堆疊變數，所以我們必須確保它是 `Closed` — 如果稍後的操作做出 *Partial Failure* 的話。我們將使用善後用的堆疊函式 `CleanupClosePushL()`。

```
CAnimatedObjectFactory* CreateAnimatedObjectFactoryFromDLL(
                          const TDesC& aFileName) {
   RLibrary libraryHandle;
   TInt r = libraryHandle.Load(aFileName);
   if (r != KErrNone)
        User::Leave(r);
   CleanupClosePushL(libraryHandle);
```

我們必須保證程式庫已被驗明正身，核對無誤。EPOC 的每個程式庫（以及資料檔）以檔案最前端的三個 Unique Identifier（UID）互相區別。其中第二個 UID（索引爲 1）用來標識檔案類型：

```
if(libraryHandle.Type()[1] != TUid::Uid(KUidAnimationLibraryModuleV01))
        User::Leave(KErrBadLibraryEntryPoint);
```

EPOC DLLs 並不根據名稱，而是根據序號匯出函式（Tasker 1999b）。按慣例，呼叫序號爲 1 的程式庫進入點，會傳回 *Factory Object*（本例爲 `CAnimatedObjectFactory`）的一份實體。

```
typedef CAnimatedObjectFactory* (*TAnimatedObjectFactoryNewL)();
TAnimatedObjectFactoryNewL libEntryL =
   reinterpret_cast<TAnimatedObjectFactoryNewL>(libraryHandle.Lookup(1));
if (libEntryL == NULL)
   User::Leave(KErrBadLibraryEntryPoint);
CAnimatedObjectFactory* factoryObject = (*libEntryL)();
CleanupStack::PushL(factoryObject);
```

我們將在 packages 生存期間始終保持這個 *factory object*，因此我們把 `RLibrary` handle 傳給其建構函式（construction function）：

```
   factoryObject->ConstructL(libraryHandle);
   CleanupStack::Pop(2); // libraryHandle, factoryObject
   return factoryObject;
}
```

`CAnimatedObjectFactory` *factory object* 簡潔明瞭。它只儲存程式庫的 handle。就像幾乎每一個擁有資源的 EPOC 物件一樣，它衍生自 `CBase`，並提供 `ConstructL` 函式（Tasker et al. 2000）。它的一些函式會被「跨越 DLL 邊界」而呼叫；我們使用 EPOC 的 `IMPORT_C` 和 `EXPORT_C` 巨集，告知編譯器產生額外的連結碼（linkup code）。

```
class CAnimatedObjectFactory : public CBase {
public:
   IMPORT_C ~CAnimatedObjectFactory();
   IMPORT_C void ConstructL(RLibrary& aLib);
```

```
    IMPORT_C virtual CAnimatedObject* NewAnimatedObjectL() = 0;
private:
    RLibrary iLibraryHandle;
};
```

建構函式（construction function）與解構式的實作非常簡單：

```
EXPORT_C void CAnimatedObjectFactory::ConstructL(RLibrary& aLib) {
    iLibraryHandle = aLib;
}
EXPORT_C CAnimatedObjectFactory::~CAnimatedObjectFactory() {
    iLibraryHandle.Close();
}
```

2. packages 的實作

packages 本身必須實作進入點，用以傳回一個新的 *factory object*。因此它需要一個衍生自 `CAnimatedObjectFactory` 的 class：

```
class CPersonFactory : public CAnimatedObjectFactory {
public:
    virtual CAnimatedObject* NewAnimatedObjectL();
};
CAnimatedObject* CPersonFactory::NewAnimatedObjectL() {
    return new(ELeave) CPerson;
}
```

這個 packages 還要求 class 必須實作 `CAnimatedObject` 物件本身：

```
class CPerson : public CAnimatedObject {
public:
    CPerson();
    // etc.
};
```

最後，程式庫進入點（entry point）簡單傳回 concrete *factory object* 的一個新實體（如果記憶體配置失敗，則傳回 null）。`EXPORT_C` 可以確保此函式是個程式庫進入點。在 MS C++ 中，藉由編輯 'DEF' 檔，我們可以確保此函式對應於程式庫中的序號 1（Microsoft 1997）。

```
EXPORT_C CAnimatedObjectFactory* LibEntry() {
    return new CPersonFactory;
}
```

❖ ❖ ❖

已知應用（Known uses）

大多數現代作業系統如 UNIX, MS Windows, WinCE, EPOC 等，都支援動態連結程式庫（Goodheart and Cox 1994; Petzold 1998, Symbian 1999b）。許多應用程式將特定 DLLs 的載入時機延遲 — 特別是第三方提供的額外功能。Lotus Notes 按需要載入各類 viewer DLLs；Netscape 和 Internet Explorer 動態載入閱讀器（諸如 Adobe PDF 閱讀器）；MS Word 載入文件轉換器，並利用 DLLs 支援網頁編輯之類的擴充。某些 EPOC 應用程式會明白載入 packages：Web 應用程式爲每一種傳輸機制（HTTP, FTP…等等等）載入驅動程式，並爲每一種資料類型載入對應的閱讀器。

人們常將印表機驅動程式實作成 *Package*s。這可使你不必重新啓動任何應用程式，即可增添新的印表機驅動程式。只要有必要，所有 EPOC 應用程式都會動態載入印表機驅動程式。MS Windows 95 和 NT 亦同。

許多 Lisp 系統採用動態載入。例如 GNU Emacs 由一個「核心文字編輯器 packages」加上一些「自動載入的設施」組成。GNU Emacs 引人入勝的特性大多以 packages 形式出現：智能語言、拼寫檢查、電子郵件、web 瀏覽器、終端模擬…等等，不一而足（Stallman 1984）。

Java 廣泛使用動態載入。Java 只在需要某個 class 時才載入之，所以每個 class 實際上都是 *Package*s。一旦 Java 程式不再需要某些 classes，使用垃圾回收器吧，不過目前還有許多環境並不這麼做。Java applets 也被視爲由 web 瀏覽器動態載入的 packages。瀏覽器會載入目前顯示的頁面中的 applets，並執行之。一旦該頁面不再顯示，瀏覽器便停止執行 applets 並卸載之（Lindholm and Yellin 1999）。Palm Spotless JVM classes 幾乎都是動態載入，甚至像 `String` 那樣實實在在的 Java class 也未能例外（Taivalsaari et al. 1999）。

很多早期處理器在其架構中明白支援 segmentation。8086 和 PDP-11 處理器都實作出 segment 暫存器。這些環境裡頭的程式員常常必須敏銳地知曉由於固定的 segment 大小而帶來的限制；MS Windows 1,2,3 無一不在編程介面中明白反映出這種 segmented（分段）架構（Hamacher et al. 1984; Chappell 1994）。

參見（See Also）

Application switching（84）是本範式較爲簡單的替代品，當 task 被分爲獨立 phases 時適用。*Paging*（119）是比較複雜的替代品。卸載下來的 packages 可以存放於 *Secondary Storage*（79）裡頭，或許還可以使用 *Compression*（135）。

Abstract Factory 提供了「將客戶介面與 packages 實作分開」的優秀實作機制。*Virtual Proxies* 可用來自動載入各個獨立的 packages（Gamma et al. 1995）。你或許需要 *Reference Counting*（268）或 *Garbage Collection*（278）來決定何時可以卸載某個 packages。

Coplien 所著的 *Advanced C++ Programming Styles and Idioms*（1994）詳細講述如何將 C++ 函式動態載入一個正執行中的程式。

Paging（分頁）

另名 Virtual Memory, Paging OO DBMS

如何製造出記憶體數量無限的幻象[1]？

- 程式碼和資料的記憶體需求過於龐大，RAM 無法負荷。
- 程式需要對其程式碼和資料進行隨機存取。
- 你有一個快速的次儲存裝置，可存放目前不用的程式碼和資料。
- 爲減輕編程心力、提高可用性，程式員和用戶不應當知曉程式正在使用次儲存裝置。

有些系統的記憶體需求實在太大，可用記憶體根本容納不下。或許因爲程式的資料結構大於系統的 RAM 記憶體，或許因爲主記憶體無法容納整個系統 — 即使每個獨立組件本身都很小。

例如 Strap-It-On 的天氣預報系統 Rain-Sight，由其無線電網絡連結載入相對少量的天氣資訊，企圖計算出用戶是否會遭遇風雨。爲了做到這一點，它得處理一些超大規模矩陣 — 規模之大即使無其他任何應用程式，記憶體也容納不下。因此 Rain-Sight 市場團隊已經同意該程式的每一份出貨隨附一個 5 Gb 'coin-disk' pack（譯註：尺寸如同美金一分錢硬幣的一種高容量硬碟），這用來存放 Rain-Sight 的資料就綽綽有餘了。Rain-Sight 開發者面臨的問題只是如何操控這新穎的設備。

你可以明白運用 *Data File*（92）管理次儲存裝置上的資料，但有兩個缺點：

- 你的程式碼必須將「處理資料」和「在主、次儲存裝置之間施展資料乾坤大挪移」兩者結合起來。其結果既繁雜又難以維護，需花費大型編程心力才能實作出來，亦需良好的編程素養才能正確使用，因爲程式員不但需要理解特定領域的需求，還必須清楚資料存取的細微處。

[1] 有時候因爲程式太大、太複雜，或是你太懶，以至於難以對程式碼加工（segment, subdivide, chain, phase, slice, dice, vitamize…等等等）。爲什麼程式員就活該擔憂記憶體呢！「無限記憶體」對所有人都應該是天賜人權，不該是恩典！那些在小記憶體系統中汲汲營營的傢伙們其實是命運不濟的失敗者！

- 此外，這種方法對資料的隨機存取往往效率不彰。如果每次存取一筆資料你都得讀取它，操作完畢後還得寫回，那麼就得執行多次緩慢的次儲存裝置存取動作。

其他諸如 *Compression*（135）和 *Packed Data*（174）之類的技術，當然可以降低 RAM 記憶體需求，但降低的幅度有限；說句極端的話，任何系統都可以擁有比 RAM 所能容納的更多資料。

因此：將系統程式碼和資料存放在次儲存裝置內，必要時才載入主記憶體，或加以卸載。

沒有任何軟體能夠完完全全地隨機存取記憶體；任一特定時刻，一個典型的系統只能使用程式碼和資料的一個小型子集。所以你只需在記憶體中保持一個相對小的工作集（working set）即可；系統的其餘部分可以存放於次儲存裝置中。由於工作集（working set）位於主記憶體內，所以存取速度非常快。如果要存取次儲存裝置上的資訊，你就必須變換工作集，讀取所需的新資料，並將原本佔據空間的資料寫出或丟棄。

你必須確保工作集（working set）的更換很慢 — 也就是軟體必須展現所謂「參考的區域性」（*locality of reference*）：存取同一個物件或記憶區，遠多於完全隨機存取。這有助於記憶體配置器把配置所得的資料項彙集在同一個記憶區內。尤其當你使用 *Fixed Allocation*（226）時，配置所得的物件通常會在記憶體中實質比鄰。

本範式目前有三種使用形式（Tannenbaum 1992; Goodheart and Cox 1994）：

Demand paging 是最被熟悉的形式。「記憶體管理相關硬體」或「直譯器環境」負責實作出虛擬記憶體，以一個額外的 page table（分頁表）將軟體所使用的位址映射至實際記憶體的 pages 上。如果軟體試圖存取 page table 內無相應條目的記憶體位址，這個環境便釋放一個 page（首先得將其內容保存於次儲存裝置），然後將新資料從次儲存裝置載入實際記憶體中，並傳回實際記憶體的位址。

Swapping 是 paging 的一個簡單替代形式，此法令操作環境停止某個行程，並將其全部資料寫入次儲存裝置。一旦該行程需要處理某個事件（event），操作環境再從次儲存裝置中重新載入全部資料，恢復執行。此法在可攜式 PCs 中極爲常見，整個環境都被保存至硬碟 — 儘管其目的是爲了省電而非爲了節約記憶體。

Object-oriented databases 和 demand paging 很類似，但 paged memory 的單位是「某個物件，及其相關並擁有的物件（或者，爲了高效起見，可能是一簇此類物件）。本法所需的編程心力比 demand paging 多，但可使資料具備永續性（persistence），也允許多個行程共享同一物件。

舉個例子，Rain-Sight 團隊決定運用 Paging，讓他們的硬碟大展神威。Strap-OS 作業系統不支援以硬體爲基礎的 paging，所以其團隊修改了一個 Java 直譯器，爲每個 Java 物件實作 paging。然後，該團隊定義了一些物件，用以實作出 Rain-Sight 矩陣的每個相關部分（它們總是被一起存取），賦予它們可被接受的效率，和一塊「看起來只受 'coin disk' 大小限制」的記憶體空間。（譯註：'coin disk'，尺寸如同美金一分錢硬幣的高容量硬碟）

結果（Consequences）（譯註：本段保留之 *Italic Times New Roman* 字型均爲 forces 名稱）

Paging 是面臨記憶體挑戰的程式員的最終解脫。由於 paging 造成了「記憶體無限」幻象 — 程式的記憶體需求（*memory requirements*）不再成爲問題，所以和其他技術相比，程式員對於 paging 最是可以不聞不問。由於記憶體需求不再是壓倒一切的重要議題，所以 paging 可以提高系統的設計品質（*design quality*）和維護性（*maintainability*）。

使用 paging 幾乎不需要特別的編程心力（*programmer effort*）和編程素養（*programmer discipline*），因爲我們無須對程式進行邏輯分解。由於 paging 不要求任何人工分割 — 將程式分割爲多個階段（phases）或將資料分割至檔案，所以系統更具可用性（*usable*）。使用 paging 機制的程式可以僅僅藉由少量 pages 就輕而易舉地提供更多記憶體，因此 paging 也提高了擴展性（*scalability*）。

在「記憶體的運用乃全域分佈於許多不同組件」的情況下，paging 對於記憶體有很好的區域性運用。這是因爲不同的組件通常在不同的時間使用資料。

然而：由於某些記憶體的存取需要讀寫次儲存裝置，因此 paging 降低了程式的時間效率（*time performance*）。它同時也降低了回應時間的可預測性，因而不適用於即時系統（*real-time systems*）。如果記憶體存取未能呈現「參考的地域性」（*locality of reference*），那麼 paging 的表現將會很差，這可能需要付出編程心力（*programmer effort*）進行修正。

爲了表現出高效率，paging 需要快速的次儲存裝置。當然啦，快是個相對概念；許多系統以磁碟進行 paging。由於 paging 傾向於執行多次小資料傳送，而非寥寥數次的大型資料傳送，所以次儲存裝置的延遲時間（latency）往往比其總處理能力更加重要。此外，paging 連續存取次儲存裝置，會增加系統的電能消耗（*power consumption*），磨損 flash RAM 和磁碟等儲存介質的壽命。

由於 paging 並不要求編程素養（*programmer discipline*），因此 paged 系統中的應用程式的記憶體需求（*memory requirements*）會趨於增加，需要更多次儲存裝置的幫忙，並會影響程式的時間效率（*time performance*）。Paging 並不要求程式內部的支援，但需要低層面的總體支援，後者常常由硬體和作業系統、直譯器或資料管理器提供。由於中間資訊可以被 'paged out' 至次儲存裝置，所以除非次儲存裝置也像主儲存裝置一樣被保護起來，否則 paging 會影響系統的安全性（*security*）。

❖ ❖ ❖

實作（Implementation）

通常我們以兩種主要的資料結構來支援 paging；見圖 2.12。

page frames 位於主記憶體內，包含程式的 "paged in" RAM 資料。每個 page frame 也擁有控制資訊：(1) 當前資料所對應的「次儲存裝置位置」，(2) 一個 *dirty bit*，一旦由次儲存裝置載入的 page memory 被改變，這個 bit 便被設立起來。

page table 也位於主記憶體內；次儲存裝置上的每一個 page 在其中都有一個對應條目。它儲存著「page 是否駐留於記憶體內」的訊息；如果答案肯定，那麼就還儲存著「page 存在於哪一個 page frame 內」的訊息。圖 2.12 展示 page table 和 page frames。

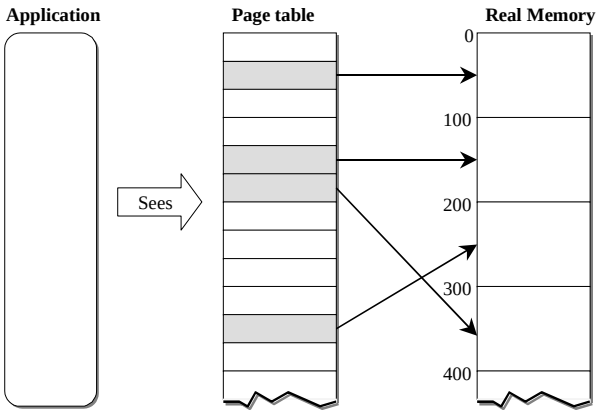


圖 2.12 Page table 和 page frames

當你執行 paging 應用程式，它透過 page table 存取記憶體。'paged in' 的記憶體可被直接讀寫；如果對 page 執行塗寫動作，就應當設立該 page 的 *dirty bit*。當你試圖存取 'paged out'（不在主記憶體中）的 page 時，系統必須從次儲存裝置載入該 page。爲了給新 page 騰出空間，可能得將記憶體內現有的 page 存回次儲存裝置。試圖存取次儲存裝置內的 page，會造成 page fault。爲了處理 page fault，或是爲了配置一個新的 page，系統必須找到一個未使用的 page frame。一般而言，雀屏中選的 frame 往往內含作用中的資料（active data），這些資料必須被丟棄；如果其 *dirty bit* 被設立，那麼系統一定得將內容全部寫回次儲存裝置中。一旦新的 frame 配置完畢，或者已從次儲存裝置載入內容，page table 會被刷新，程式繼續執行。

paging 的實作需考慮以下一些問題。

1. 攔截記憶體存取動作（intercepting memory access）

實作 paging 時難上加難的部分大概是攔截記憶體存取動作了。這種攔截一定要把塗寫動作和讀取動作嚴格區分開來，前者必須設立 *dirty bit*，後者不必。

下面是數種可行的機制：

1. *MMU*。許多現代系統除了 CPU（中央處理單元）之外尚有 MMU（記憶體管理單元）。MMU 提供一個虛擬記憶體映射（*virtual memory maps*）集（通常一個行程一個 map），可將程式碼所請求的記憶體位置，映射至不同的實際記憶體位址。如果程式存取了一個未曾被載入的位址，將會產生一次 page fault 中斷，於是中斷驅動程式會從次儲存裝置載入 page。MMU 也把 pages 分爲「唯讀」和「可讀可寫」兩種。企圖對唯讀的 page 執行塗寫動作，也會導致 page fault 中斷，這使得該 page 對應的 dirty bit 的設立不費吹灰之力。
2. *Interpreter*（直譯器）。爲直譯環境（interpreted environments）實作 paging，是件十分直截了當的事。執行期直譯器必須實作出「對程式及其資料」進行的所有存取，因此，攔截存取並區分讀寫，就相對容易多了。
3. *Process Swap*。當你置換（swap）整個行程，如果行程於換出（swapped out）時並未運行，你就無須偵測記憶體存取。
4. *Data manager*。如果操作環境中並無內建的 paging 機制，我們可以採用 'smart pointers' to classes 來攔截對物件的所有存取動作。因此一個 data manager 可以保證物件位於儲存器內並管理物件的載入、快取和置換動作。

   這種情況下 'page in' 或 'page out' 整個物件，而非任意大小的 pages，可謂舉止得宜。

2. 頁面替換（Page replacement）

如何選擇 page frame、釋放空間，以便接納新的或被載入的 page？最佳演算法是「移除未來將最後被使用」的 page — 也就是「就系統的立即需求而言」最不重要的 page（Tannenbaum 1992）。不幸的是這種演算法通常無法具體實現，因此你不得不改變手法，根據最新情況推演未來。移除最不常被使用的（least frequently used, LFU）pages 可以產生最精確的估測結果，但實作難度相當大。效率不相上下但更容易實作的是 least recently used（LRU）演算法 — 只需保持所有 page frames 構成的一份 list，一旦某個 page 被使用就把它移至 list 頂端。「隨機挑選一個可替換的 page」頗爲容易，許多情況下性能也頗令人滿意。

大部份 MMU paging 實作手法也包含了分段（segmentation）技術（參見 *Packages*, 108）。由於你已經令行程的虛擬「**資料**」記憶體分割爲 pages，對著「**程式碼**」如法炮製一番，自然是一種顯而易見的擴充。程式碼是「唯讀」性質（65），把它從次儲存裝置載入記憶體時，改動通常微乎其微。因此在 swap file 中爲程式碼浪費空間根本不值得；需要 code pages 時可以直接從 code file 取得；不需要它時逕自丟棄它們就是了。

3. 工作集的大小（Working set size）

程式的 *working set* 大小是指在「不產生過多 page faults」的前提下運行所需的最小記憶體量。籠統地說，page size 愈大，working set size 就愈大。程式的 working set 大小決定程式能否在任何已知的 paging 系統中運行良好。如果該大小比配置給 page frames 的實際記憶體大，那麼 page faults 的數量將非常驚人。系統將開始 *thrashing*（遭受痛擊，作業系統術語），花費許多時間在主記憶體和次儲存裝置之間往復置換 pages，對應用程式的執行毫無幫助。

爲避免造成 thrashing，請讓你的程式少做點事，並爲系統增加多一些記憶體。或者利用 *Packages*（108）所討論的技術，將程式的記憶體佈局最佳化。

4. 分頁控制（Program control of paging）

有些程式並未實現所謂「指涉地域性」（*locality of reference*）。例如某個程式可能巡訪全部資料一遍，因此按序讀取每個 page 一次（就這一次）。這種情況下每個 page 只被置入（paged in）一次，而最適合被替換的 page 便是最常或最近被使用的那個。爲解決此類問題，某些系統提供介面讓程式員能夠控制其 paging 機制。該介面或許會支援「把特定 page 置換出去」的請求，或者可能接受「把特定 page 置換進來」的請求 — 例如上述程式在處理當前 page 之前便知道下次需要哪個 page。

其他程式碼可能有些即時性的約束（real-time constraints）。例如設備驅動程式通常必須在數個微秒之內回應中斷事件，因此設備驅動程式的資料不能被 'paged out'。大部份系統以不同屬性標識程式碼的特定區域，藉以支援上述想法。例如 Microsoft 的 PE（Portable Executable）可執行檔格式便對其虛擬裝置驅動程式（VxD）的可執行檔支援了兩個選項：Pre-load（預先載入）和 Memory-resident（記憶體長駐）（Microsoft 1997）。

示例（Examples）

以下程式碼實作出一個簡單框架，用以對單個 C++ 物件進行分頁（paging）。

在缺乏作業系統的支援下，要攔截「指向某個 C++ 物件」的全部 references，確實異常艱難，尤其是我們無法攔截成員函式裡頭的 `this` 指標。所以將分頁目標放在任何一個「擁有成員函式之 C++ 物件」身上，可謂打錯了主意。讓我們換一種手法，令各個物件把自己的資料儲存在一個獨立的資料結構內，並透過「得以控制 paging」的特殊成員函式來存取該結構。物件本身充當資料的替身（*Proxy*，Gamma et al. 1995），我們將不儲存資料的指標，改而儲存 page 號碼。圖 2.13 展現一種典型的情況。

page table 實現了「對 RAM 內資料存取」的最佳化：如果某特定 page 的對應條目是 non-null，該 page 便是已被載入 RAM，程式物件可以直接存取其資料。如果程式物件試圖存取一個 page，而 page table 上的對應條目爲 null，那就意味該物件的資料並未被載入。這種情況下 paging 機制會回存或丟棄一個現有的 page frame，並從硬碟載入物件資料。

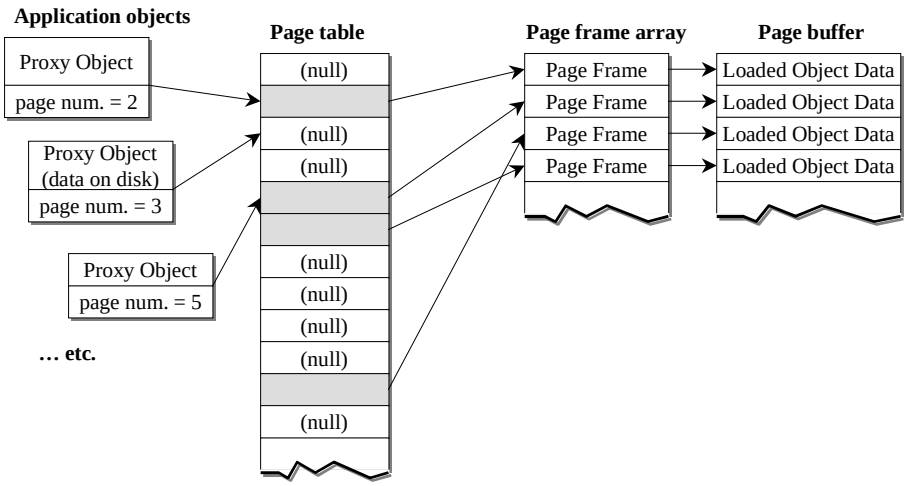


圖 2.13　paging 實例中，記憶體內的物件

1. 客戶端的實作

這裡是一份使用 paging 機制的客戶端實例。它是一個僅僅包含像素（pixels）的簡單 bitmap 圖。其他 paged 資料結構可以包含任何 C++ 基本型別或 `structs`，包括 pointers to object（當然，不能夠是 pointers to paged data structure instances，因爲它們即將被 'paged out'）。

```
typedef char Pixel;
class BitmapImageData {
   friend class BitmapImage;
   Pixel pixels[SCREEN_HEIGHT * SCREEN_WIDTH];
};
```

Proxy class `BitmapImage` 的 paging 功能衍生自更一般化的 `ObjectWithPagedData`。其實作主要限制在於「對資料物件的所有存取動作」都必須透過 base class 的 `GetPagedData()`，後者保證資料被 'paged into' RAM。它把資料強制轉型爲正確型別，而後得以存取這些資料：

```
class BitmapImage : public ObjectWithPagedData {
private:
   BitmapImageData* GetData()
        { return static_cast<BitmapImageData*>(GetPagedData()); }
   const BitmapImageData* GetData() const
        { return static_cast<const BitmapImageData*>(GetPagedData()); }
```

建構式必須指定 `PageFile` 物件，並將資料結構初始化。請注意這些函式都可以是 inline：

```
public:
   BitmapImage(PageFile& thePageFile)
        : ObjectWithPagedData(thePageFile) {
        memset(GetData(), 0, sizeof(BitmapImageData));
   }
```

其他所有函式都透過 `GetData()`存取資料。請注意 C++ 的常數性（constness）如何保證我們取得正確版本的函式；`GetData()`的 non-const 版本會設立 page 的 'dirty bit'，於是一旦該 page 被 'paged out' 時，便會被寫回檔案（譯註：而非逕自丟棄）。

```
   Pixel GetPixel(int pixelNumber) const {
        return GetData()->pixels[pixelNumber];
   }
   void SetPixel(int pixelNumber, Pixel newValue) {
        GetData()->pixels[pixelNumber] = newValue;
   }
};
```

這就是完整的客戶端實作。簡單至極，不是嗎？

爲了使用它，我們得建立一個 page file。這裡有個例子，只有四個 page buffers：

```
PageFile pageFile("testfile.dat", sizeof(BitmapImageData), 4);
```

於是我們便可以像使用其他 C++ 物件一樣地使用 `BitmapImage`：

```
BitmapImage* newImage = new BitmapImage(pageFile);
newImage->SetPixel(0, 0);
delete newImage;
```

2. paging framework 概觀

圖 2.14 以 UML 表示法展示 paging framework 的邏輯結構（Fowler and Scott 1999）。一般字型所表現的是框架中的 classes 名稱；其他東西實作如下：

- *Page Table Entry* 是 pageTable 指標陣列裡的一個條目（entry）。
- *Page in RAM* 是一個簡單的 `(void*)` 緩衝區。
- *Page on Disk* 是硬碟檔案裡頭的一個固定 page。
- *Client Implementation* 是類似前述 `BitmapImage` class 的任何一個客戶類別。

page frames 和 page table 是個 *Fixed Allocation*，總是佔用同一塊 RAM 記憶體。

3. 實作 ObjectWithPagedData

`ObjectWithPagedData` 是客戶實作類別的基礎類別。它只包含資料的 page 號碼，和一個 reference to `PageFile` object。這使我們得以擁有數種不同型別且各自分頁（paged）的客戶物件。

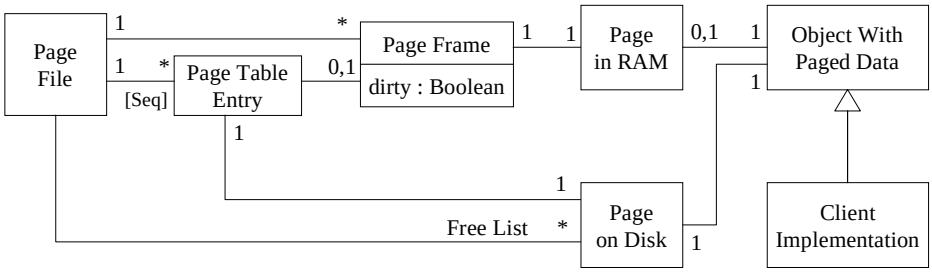


圖 2.14　UML 圖：物件分頁系統（object paging system）的邏輯結構

```
class ObjectWithPagedData {
private:
   PageFile& pageFile;
   const int pageNumber;
```

所有成員函式都是 protected，因爲只有客戶實作碼使用它們。建構式和解構式以 `PageFile` 提供的函式來配置和釋放 data page：

```
ObjectWithPagedData::ObjectWithPagedData(PageFile& thePageFile)
   :    pageFile(thePageFile),
        pageNumber(thePageFile.NewPage())
{}

ObjectWithPagedData::~ObjectWithPagedData() {
   pageFile.DeletePage(pageNumber);
}
```

我們同時需要 const 和 non-const 兩個版本的函式來存取 paged data。二者都保證會存在一個 page frame，然後存取緩衝區；non-const 版本所用的函式會設立 page frame 的 dirty-bit：

```
const void* ObjectWithPagedData::GetPagedData() const {
   PageFrame* frame = pageFile.FindPageFrameForPage( pageNumber );
   return frame->GetConstPage();
}
void* ObjectWithPagedData::GetPagedData() {
   PageFrame* frame = pageFile.FindPageFrameForPage( pageNumber );
   return frame->GetWritablePage();
}
```

4. 實作 PageFrame

一個 `PageFrame` 物件代表「真實記憶體」的唯一一塊緩衝區。如果 page frame 處於「使用」（In Use）狀態，表示客戶物件已經存取過該緩衝區，而緩衝區內容尚未被保存到硬碟或被丟棄。如果 frame 處於「使用」狀態，就設定成員 `currentPageNumber` 代表適當

的 page，否則設爲 INVALID_PAGE_NUMBER。PageFrame 會儲存緩衝區的 dirty-bit，並在任何一個客戶呼叫 GetWritablePage()時設立之。

```
class PageFrame {
   friend class PageFile;

private:
   enum {    INVALID_PAGE_NUMBER = -1 };
   bool      dirtyFlag;
   int  currentPageNumber;
   void*     bufferContainingCurrentPage;
```

建構式和解構式只是很簡單地將各成員初始化：

```
PageFrame::PageFrame(int pageSize)
   :    bufferContainingCurrentPage(new char[pageSize]),
        dirtyFlag(false),
        currentPageNumber(PageFrame::INVALID_PAGE_NUMBER)
{}

PageFrame::~PageFrame() {
   delete [] (char*)(bufferContainingCurrentPage);
}
```

GetConstPage 和 GetWritablePage 提供緩衝區存取能力：

```
const void* PageFrame::GetConstPage() {
   return bufferContainingCurrentPage;
}

void* PageFrame::GetWritablePage() {
   dirtyFlag = true;
   return bufferContainingCurrentPage;
}
```

另兩個成員函式也很平常：

```
int PageFrame::PageNumber() {
   return currentPageNumber;
}
bool PageFrame::InUse() {
   return currentPageNumber != INVALID_PAGE_NUMBER;
}
```

5.實作 PageFile

PageFile 物件用來管理 paging 系統的全部重要行爲。它擁有暫時檔，並實作出用來「交換資料緩衝區」的函式。

PageFile 主體結構如下：

pageTable 是個 vector，每個元素對應 page file 裡頭的一個 page。如果 page 被置換至次儲存裝置，則元素爲 null；如果 page 在 RAM 裡頭，則元素指向一個 PageFrame 物件。

pageFrameArray 內含所有 PageFrame 物件。這是個陣列，對於「隨機選擇欲丟棄之物件」過程實現了簡化。

listOfFreePageNumbers 內含「已被刪除之 pages」所構成的一個 queue。我們無法把 pages 從 page file 中移除，因此另闢蹊徑，錄下 page 號碼，必要時重新指定。

綜合以上所說，得到 private data 如下：

```
class PageFile {
   friend class ObjectWithPagedData;
private:
   vector<PageFrame*> pageTable;
   vector<PageFrame*> pageFrameArray;
   list<int> listOfFreePageNumbers;
   const int pageSize;
   fstream fileStream;
```

PageFile 的建構式必須將檔案初始化，並配置所有的 *Fixed Allocation*（226）。如果開檔失敗，要有放棄的辦法；本例只是簡單運用 ASSERT 巨集的一個變體來進行檢查：

```
PageFile::PageFile(char* fileName, int pageSizeInBytes,
                   int nPagesInCache)
   :    fileStream(fileName,
                   ios::in|ios::out|ios::binary|ios::trunc),
        pageSize(pageSizeInBytes) {
   ASSERT_ALWAYS( fileStream.good() );
   for (int i = 0; i<nPagesInCache; i++) {
         pageFrameArray.push_back(new PageFrame(pageSize));
   }
}
```

解構式必須收拾記憶體並關閉檔案。如果考慮完整，還應刪除檔案：

```
PageFile::~PageFile() {
   for (vector<PageFrame*>::iterator i = pageFrameArray.begin();
        i != pageFrameArray.end(); i++)
        delete *i;
   fileStream.close();
}
```

函式 NewPage()可爲新的客戶物件在硬碟上配置 page。如果硬碟上有空閒的 page，它就使用之，否則就配置一個新 pageTable 條目，並向 page file 末尾寫入一個 page 的隨機資料（因而擴展 page file）：

```
int PageFile::NewPage() {
   int pageNumber;
   if (!listOfFreePageNumbers.empty()) {
        pageNumber = listOfFreePageNumbers.front();
        listOfFreePageNumbers.pop_front();
   } else {
        pageNumber = pageTable.size();
        pageTable.push_back(0);
        int newPos = fileStream.rdbuf()->pubseekoff(0, ios::end);
        fileStream.write(
             (char*)pageFrameArray[0]->bufferContainingCurrentPage,
             PageSize());
   }
   return pageNumber;
}
```

對應的 `DeletePage()`函式很簡單：

```
void PageFile::DeletePage(int pageNumber) {
   listOfFreePageNumbers.push_front[pageNumber];
}
```

函式 `FindPageFrameForPage()`爲給定的 page 號碼指派一個 `PageFrame`，並保證該 page 一定在 RAM 內。如果該 page 已經有了個 `PageFrame`，此函式就傳回指標；否則就找一個 `PageFrame` 並以硬碟獲得之 page 內容填充之。

```
PageFrame* PageFile::FindPageFrameForPage(int pageNumber) {
   PageFrame* frame = pageTable[pageNumber];
   if (frame == 0) {
        frame = MakeFrameAvailable();
        LoadFrame(frame, pageNumber);
        pageTable[pageNumber] = frame;
   }
   return frame;
}
```

函式 `MakeFrameAvailable()`藉由 'paging out' 或「丟棄」一個隨機選擇的現有 page，來指派 frame：

```
PageFrame* PageFile::MakeFrameAvailable() {
   PageFrame* frame =
             pageFrameArray[(rand() * pageFrameArray.size()) / RAND_MAX];
   if (frame->InUse()) {
        SaveOrDiscardFrame( frame );
   }
   return frame;
}
```

函式 SaveOrDiscardFrame()提供 paging 演算法的實質內容。它把 page 寫到檔案中的相應位置（若有必要），並重新指定 page table 條目：

```
void PageFile::SaveOrDiscardFrame(PageFrame* frame) {
   if (frame->dirtyFlag) {
        int newPos = fileStream.rdbuf()-
                          frame->PageNumber()*PageSize(), ios::beg);
        fileStream.write((char*)frame->bufferContainingCurrentPage,
                          PageSize());
        frame->dirtyFlag = false;
   }
   pageTable[frame->PageNumber()] = 0;
   frame->currentPageNumber = PageFrame::INVALID_PAGE_NUMBER;
}
```

最後，對應的 frame 載入函式實作如下：

```
void PageFile::LoadFrame(PageFrame* frame, int pageNumber) {
   int newPos = fileStream.rdbuf()->pubseekoff(pageNumber * PageSize(),
                          ios::beg);
   fileStream.read((char*)frame->bufferContainingCurrentPage,
PageSize());
   frame->currentPageNumber = pageNumber;
}
```

❖ ❖ ❖

已知應用（Known uses）

幾乎任何一個現代磁碟作業系統(modern disk O.S.)都提供分頁式虛擬記憶體(paged virtual memory），包括大部份版本的 UNIX、Linux、MacOS、MS Windows（Goodheart and Cox 1994; Card et al. 1998; Microsoft 1997a）。

OO 資料庫幾乎都採用某種形式的 object paging（物件分頁）。ObjectStore 直接使用 UNIX（或 NT）的 paging 支援，但以「使用永續性資料以滿足 OO 程式需求」的驅動程式取代作業系統提供的 paging 驅動程式（Chaudhri and Loomis 1997）。

Infocom 遊戲在 Apple-IIs 及早期 PCs 機器上實作了 paged 直譯器，把主記憶體 'paging' 至磁碟（Blank and Galley 1980）。於是遊戲可以運行在記憶體大小不等的機器上。當然啦，可用的主記憶體愈少，遊戲就運行得愈慢。LOOM 系統在 Smalltalk 上實作了 paging 機制（Kaehler and Krasner 1983）。

參見（See Also）

本章其他範式 — *Application Switching*（226）、*Data Files*（92）、*Packages*（108）及 *Resource Files*（101），提供了本範式之外的其他選擇。*Paging* 也可採用 *Copy-on-Write*（191）對唯讀儲存器的存取進行最佳化，亦可被擴展以支援 *Sharing*（182）。系統記憶體是全域資源，因而某些作業系統會實作 *Captain Oates*（57），在不同的行程（而非來自「要求新 page」的行程）中丟棄節區（segments）。

Interpreter（Gamma et al. 1995）可使 *Paging* 對用戶程式透明。*Virtual Proxies* 和 *Bridges*（Gamma et al. 1995）以及 *Envelope/Letter* 或 *Handle/Body*（Coplien 1994）可以不影響物件的客戶介面而爲物件提供 *Paging*。

Small Memory Software