

目 錄

讀者來函

知 莫大乎棄疑（侯捷自序）

慮而後能得（第一版自序）

目錄

前言

本書定位

合適的讀者

最佳閱讀方式

各章主題

編譯工具

中英術語的運用風格

英文術語採用原則

版面風格

網上服務

推薦讀物

第 0 章 編譯平台、作業平台、閱讀基礎

0.1 簡介：語法、語意、物件模型、應用框架、設計樣式

0.2 百花齊放的 C++ 編譯器

0.3 編譯器命令行模式（command line mode）環境設定

0.4 make file 與 batch file

0.5 C++ 程式與作業系統的關係

0.5.1 address space (位址空間)

0.5.2 processes (行程)

0.5.3 threads (執行緒)

0.5.4 code segment (編碼節區)

0.5.5 data segment (資料節區)

0.5.6 heap (堆積)

0.5.7 free store (自由空間)

0.5.8 stack (堆疊)

0.5.9 Const Data (常數資料)

0.5.10 global/static objects (全域/靜態 物件)

0.6 C++ 編程基礎認識

0.6.1 程式進入點 (entry point)

0.6.2 宣告與定義 (declaration and definition)

0.6.3 標記式與原型 (signature and prototype)

0.6.4 參數與引數 (parameter and argument)

0.6.5 傳值與傳址 (by value and by reference)

0.6.6 左值與右值 (lvalue and rvalue)

0.6.7 指標與化名 (pointer and reference)

0.6.8 含入檔 (included file)、表頭檔 (header file)、
實作檔 (implementation file)

0.7 UML (Unified Modeling Language) 圖形表示法

0.8 範例與練習

第 1 章 Classes 基本語法和語意

第 2 章 Classes 階層體系

第 3 章 C++ 物件模型

第 4 章 型別轉換 (Type Casting) 與 #, ## 運算子

第 5 章 再談繼承與多型 (Inheritance & Polymorphism)

第 6 章 Application Framework 核心建設 – OO 技術極致展現

6.1 MFCLite 3

6.1.1 MFCLite 演進與 3.x 版簡介

6.1.2 模組切割與編譯設定

檔案組態

命令行 (Command Line) 編譯設定

整合開發環境 (Integrated Development Environment, IDE)

6.1.3 設計樣式 "Template Method" 之大量實現

6.1.4 MFCLite 的型別設定和常數定義

6.2 三大基礎建設之問題分析

6.2.1 類別型錄網 與 CRuntimeClass

6.2.2 巨集 v.s. 虛擬函式

6.2.3 根類別 CObject

6.2.4 程式產生器：巧妙的 # 運算子和 ## 運算子

6.2.5 容器相關類別：CDWordArray, CObList, CPtrList

6.2.6 形狀類別：CShape 階層體系

6.3 執行期型別辨識，第一層巨集 (x_DYNAMIC)

6.4 動態生成，第二層巨集 (x_DYNCREATE)

6.4.1 動態生成有多麼重要

6.5 次第檔案讀寫，第三層巨集 (x_SERIAL)

6.5.1 物件永續機制之輕量級實現 (a lightweight persistence)

6.5.2 將 operator>> 和 operator<< 多載化

6.5.3 檔案相關類別：CArchive, CFile

簡介 CFile

簡介 CArchive

將 object 寫入檔案

將 object 自檔案讀出

6.5.4 Serialization 檔案格式深入探討

6.6 應用框架（application framework）之浮現

6.6.1 框架模組與應用模組之乾淨切割：論資源檔

割斷耦合

資源檔（.res）

6.6.2 應用程式相關類別 CWinApp, CWinThread 與骨幹應用程式之執行流程

6.6.3 訊息相關機制之模擬

::SendMessage(), ::TranslateMessage(), ::DispatchMessage(), ::PostMessage()

6.6.4 視窗相關類別 CWnd, CFrameWnd

6.6.5 MDI 相關類別 CMDIFrameWnd, CMDIChildWnd

6.7 訊息映射（message mapping）

6.7.1 命令訊息與 CCmdTarget

6.7.2 建立一個訊息映射表（message map）

DECLARE-, BEGIN-, END- _MESSAGE_MAP

6.7.3 訊息映射表之填寫

ON_COMMAND, ON_WM_CREATE, ON_WM_PAINT...

pointer to member function 的型別轉換

Contravariance rule

6.7.4 羅塞達石碑：解開型別包裹與型別轉換之謎

6.8 訊息繞送（message routing）

6.8.1 單刀直入的視窗訊息遞送方式（直線上溯）

6.8.2 雨露均霑的命令訊息（WM_COMMAND）繞送方式

轉轍器：OnCmdMsg()

this 指標 觀念大回顧

6.8.3 適當的訊息攔截點

- 6.8.4 加入 MDI 之後
- 6.9 MVC 模型 (Model-View-Controller)
 - 6.9.1 MFC(Lite) 中的 MVC 模型總覽
 - 6.9.2 文件管理類別：CDocManager, CDocTemplate, CMultiDocTemplate
 - 6.9.3 文件，CDocument
 - 文件生成之一：開新檔案 [File/New]
 - 文件生成之二：開啓舊檔 [File/Open]
 - 文件儲存之一：儲存檔案 [File/Save]
 - 文件儲存之二：另存新檔 [File/SaveAs]
 - 6.9.4 視圖，CView
 - 6.9.5 文件外框視窗 (docFrame)，CChildFrame
 - 6.9.6 爲目前作用中的 document 增加一個 view
 - 6.9.7 文件與視圖之間的 subscribe-notify 協定
- 6.10 一個完整的測試程式
 - 6.10.1 新增一個顯示手法不同的 view
- 6.11 除錯機制
 - 6.11.1 追蹤訊息映射表與訊息繞行路線
 - 6.11.2 追蹤 CRuntimeClass 類別型錄網
 - 6.11.3 追蹤 document-view 現狀
 - 6.11.4 呼叫堆疊 (Call Stack)
 - 6.11.5 土法鍊鋼與現代化器械
- 6.12 強化 MFCLite
 - 6.12.1 CPtrList 的垃圾回收機制 (Pool 技巧實現)
 - 6.12.2 Object Persistence 再探討
 - 三種標籤 (tags) 的設計
 - CArchive 介面增修
 - 輸出至檔案

從檔案輸入

6.12.3 文件清理和視窗關閉

錯誤設計 — 挖東補西鋸箭療傷

MFC 的視窗關閉應對措施（CallStack 的觀察）

細說從頭 — 視窗關閉的 Win32 APIs 層面和訊息層面

MFCLite 的簡易視窗管理

擴充 MFCLite，增加文件關閉功能、視窗關閉功能

關閉文件：CDocument::OnFileClose()，熱鍵 'C'

關閉視窗：CFrameWnd::OnClose()，熱鍵 'c' 或 'x'

結束程式：CWinApp::OnAppExit()，熱鍵 'X'

第 7 章 設計樣式（Design Patterns）於 MFC(Lite) 之應用

7.1 設計樣式（Design Patterns）概述

7.2 OO 古典語錄

7.3 Adapter（Wrapper）

7.4 Bridge（Handle/Body）

7.5 Chain-of-Responsibility

7.6 Composite

7.7 Factory Method（Virtual Constructor）

7.8 Iterator（Cursor）

7.9 MVC（Model-View-Controller）

7.10 Observer（Dependents, Publish-Subscribe）

7.11 Singleton

7.12 Strategy

7.13 Template Method

附錄 A 推薦書目（Bibliography）

附錄 B 英中繁簡術語 對照

附錄 C 本書支援網站 簡介

附錄 D C++ 的愛戀與沉迷 / 侯捷

附錄 E 本書範例與練習 完整源碼

附錄 F MFCLite 3.0 源碼介紹與 UML 圖示

索引 (index)

6

Application Framework

核心、建設

OO 技術極致展現

設計大型軟體很不容易，設計可重複使用的大型軟體又更困難。從 OO 的角度來看，你必須找到適切的 objects，抽取它們的特性，使之成為 classes，定義其介面，厘清各個 classes 之間的繼承關係，組成一個合理體系。你的設計不僅要能夠解決特定問題，也必須有足夠的彈性以應付未來的需求。

聰明而不知達變的人，喜歡一切從輪子造起。聰明而知達變的人，很快就選好零組件，組裝出一輛汽車。軟體世界充滿了聰明達變的人，所以我們總是能夠在應用程式中發現一個或多個所謂的 toolkits（工具箱，此處指的是 libraries）。在 OO 世界裡，toolkits 是由預先定義好的 classes 所組成的類別庫（class libraries），其中的 classes 彼此獨立（或有輕微關連），各自具備良好的復用性與通用機能，例如做為容器的所謂 collection classes library，做為輸出入串流的所謂 I/O stream library。這些程式庫並不會影響你的程式的根本形貌與設計意識，它們只是提供機能，幫助你的程式完成工作；它們讓你不必重頭撰寫一般性的通用機能，它們強調的是程式碼本身的復用性（code reuse），而不是設計架構的復用性。

所謂 framework，所謂 application framework

所謂 framework（框架），不同於 toolkits，是一組緊密關連的 classes，強調彼此配合以遂行某種可重複運用的設計概念。例如 C++ 標準程式庫提供的 STL，便是一組應用於資料結構和演算法的 framework，其六大組件有著相當密切的關連，以特定方式合作，彼此不可或缺。它們相當程度影響了你的程式形貌。

在各類 framework 之中，有一種格局最宏大，影響最深遠的類型，稱為 **application framework**，用來規劃應用程式骨幹，讓程式遵循一定的流程和動線，展現一定的風貌和功能。這種產品之發展主要是基於現代化視窗作業平台（例如 Microsoft Windows, Unix Xwindow, IBM OS2 Presentation Manager...）日趨複雜，又有特殊的事件驅動（event-driven）編程模式，因而造成應用程式的撰寫難度大幅提高。為了讓程式員將精力放在專業領域，不必費力於通用性功能的繁文褥節，遂有 application framework 的誕生。

Application framework 規範你的程式架構。當你使用 toolkit，你負責撰寫程式主體，並呼叫（重複運用）別人寫好的碼；當你使用 application framework，你改而重複運用別人架構好的程式主體，並負責撰寫它（程式主體）所呼叫的程式碼，如圖 6-1。這些待呼叫的程式碼當然是以虛擬函式的形式出現，因此你必須以特定形式（具體而言就是特定的函式名稱、特定的參數列、特定的回返值型別）來實作這些虛擬函式。傳統編程方式像是寫申論題，application framework 輔助之下的編程形式卻像寫填充題。填充題可比申論題簡單多了。

使用 application framework，不僅應用程式的建立快得多，程式結構也比較類似。「程式結構類似」意味軟體界得以發展出程式碼產生器（例如 Visual C++ 之 AppWizard, Borland C++ 之 AppExpert），用以快速產生程式骨幹，也意味軟體的維護成本降低、終端用戶的學習成本降低。當然，隨之而來的就是程式員喪失了某些創新機會，因為程式骨幹（主架構）已經被決定了。這很公平：你要某種自由，你就喪失某種復用性（reusability），你要完全自由，你就完全喪失復用性。關鍵在於，上述自由度的喪失完全無損軟體的價值，亦無損程式員的尊嚴，反而讓程式員得以集中精力於專業領域的開發。

如果說撰寫 application 困難，那麼，撰寫 toolkits 更困難，撰寫 framework 又更困難，撰寫 application framework 則達到困難的頂端。如今，藉由一個 4000 行左右的小程式 **MFCLite**，我要帶領各位一窺 application framework 的神秘面紗，一探 application framework 的核心技術。實際追蹤、觸摸、剖析、實踐一個 OO 大架構，和純粹只是聽只是想只是講，層次完全不同，請不要因為懷疑而遲疑猶豫。

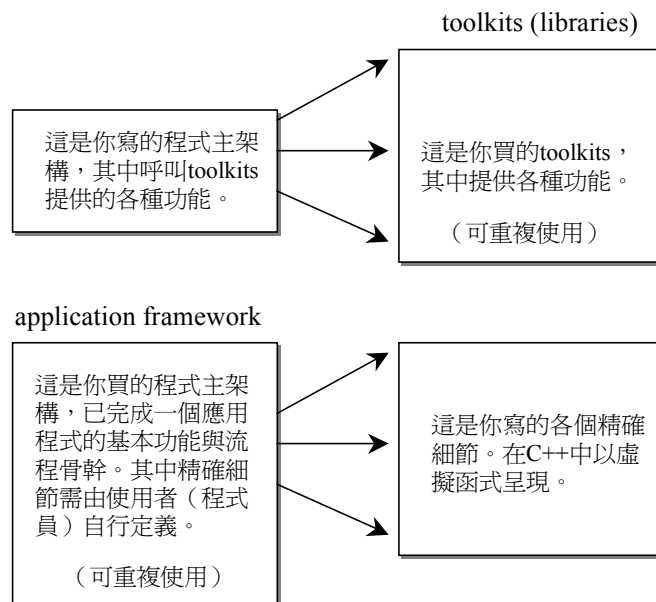


圖 6-1 toolkits 和 application framework 的差別



閱讀建議：application framework 是一種非常複雜的軟體。本章技術含量非常高濃，欲有體會，非三五遍閱讀之功。但也請不要一頭鑽進細節，見樹而不見林。最好多瀏覽幾遍，對 MFCLite 的設計有點體會了，對 MFCLite 的架構成竹在胸了，再開始仔細研究實作技術。

Application framework 的設計是來回返復不斷修正改善的。即使有了 MFC 源碼在手，MFCLite 的發展過程也是點滴擴充逐一強化的。下面是 MFCLite 的堆累次序：

- (1) RTTI/Dynamic-Creation/Serialization 三大基礎建設
- (2) application framework 與 application 之模組切割
- (3) 訊息映射表 Msg-Map
- (4) 訊息繞行 Msg-Routing
- (5) MVC model，涵蓋 Document Manager, Document Template, Document, View
- (6) msg 和 msg-queue 模擬
- (7) MDI 模擬
- (8) Window-Close system。

「翼德於百萬軍中，取上將之首如探囊取物」。我希望這一章扮演張翼德的角色，讓你在十數萬行計的 MFC 源碼中取其神髓，進而一窺所謂 application framework 的神妙。

6.1 MFCLite 3

眼下軟體界所採行的規則是，application framework 都以白盒子（white box，也就是開放源碼）的形式供貨。這有必要，一來使用者不見得完全接受 application framework 的技術封裝，或許某些時候需要自己動點手腳，二來 application framework 錯綜複雜，說明文件難免掛一漏萬，我們需要源碼在手以解萬一之惑。

源碼在手，這就帶給了我們大好的學習機會。但是一個 application framework 動輒十數萬行程式碼、數百個檔案，如果沒有好的導引，曠日費時免不了，更怕的是徒勞無功。而且，我們所渴望學習的 OO 分析與設計，及其實作技術，與視窗作業系統的特性夾雜糾葛，更需要一把明快的刀，將它們謹慎割離。

我曾經在《深入淺出 MFC》書中進行 MFC（Microsoft Foundation Classes）關鍵技術的六大模擬，也曾經在《多型與虛擬》第一版完成一個更加強化（加入了 Serialization 功能）的所謂 MFCLite 程式。在這裡，我將呈現給你 MFCLite3，大幅提昇模擬的廣度和深度，並將過去所有片片段段整合起來，真正成爲一個具體而微的小型 application framework。本書讀者很可能曾經閱讀過上述兩本書籍，稍後我會列出各版本的差異，方便你學習。

之所以一再挑選 MFC 爲觀察對象，原因是，做爲 Visual C++ 的內建產品，MFC 成爲市場佔有率最高、影響最深遠的一個 C++ application framework。也許你並不使用 MFC，也許你並不欣賞 MFC，也許你認爲 MFC 的設計還有許多值得議論之處，但「市場佔有率最高」的背後畢竟蘊含豐富的意義，再且 MFC 所採用的設計手法自有其道理。一旦我們有機會近距離觀察、親手觸摸、乃至於實際操刀解剖重建一個小型 MFC，過程之中可以落實許多寶貴的 OO 技術。我所看重的，不是 MFC 還有多大市場或多長壽命，而是演練 MFCLite 的設計與實作過程中獲得的寶貴思維。

MFCLite3 以標準 C++ 完成，適用於所有支援標準 C++ 的編譯平台和執行平台。

6.1.1 MFCLite 演進與 3.x 版簡介

我曾經在《深入淺出 MFC》第三章進行了 MFC 關鍵技術的六大模擬，可視為 **MFCLite 1.0**：

- frame1：模擬 MFC 關鍵類別的從屬（繼承）關係。
- frame2：模擬 MFC 應用程式的初始化過程。
- frame3：模擬 RTTI（RunTime Type Identification）。
- frame4：進一步模擬 RTTI。
- frame5：因書籍版次演化而保留號碼，實際為空。
- frame6：模擬動態生成（dynamic creation）。
- frame7：模擬訊息映射（message map）。
- frame8：模擬訊息繞行（message routing）。

這個版本的缺點是：(1) 應用模組與框架模組切割不乾淨，(2) 做為 application framework，規模還差一大截，(3) 檔案讀寫（serialization）機制由於牽連太廣，未加模擬，(4) Document-View-DocTemplate 架構，未加模擬，(5) 訊息映射與繞行，未以真正的訊息形式測試之，(6) 所有特性零散分佈，未加整合。

我也曾經在《多型與虛擬》第一版完成一個補強的 MFCLite，可視為 **2.0** 版：

- MFCLite：模擬 RTTI、動態生成（dynamic creation）、檔案讀寫（serialization）。最重要的是檔案讀寫的模擬，補 1.0 版不足。此版並未加入訊息相關機制，因為此版純粹只為示範 C++ 的多型、虛擬函式、巨集、多載化運算子…。

現在，你即將看到的 **MFCLite 3**，改善了先前版本的所有缺點，是一個五臟俱全具體而微的小型 application framework。除了圖像視窗難以模擬（亦非本書重點），基本上 MFCLite3 在 4000 行左右（註解不計）的源碼規模內展現了 MFC 的所有 application framework 相關核心技術。這個版本的特色是：

- 應用模組（application）與框架模組（application framework）兩端切割完全，無一絲一毫耦合。這其中涉及所謂的資源編譯器。我仿照 VC++ 模擬了一個簡單的資源編譯器（RC.EXE）。

- 模擬 application framework 三大核心建設（型別辨識、動態生成、檔案讀寫）。
- 模擬訊息佇列（message queue）、訊息攫取、訊息派送（dispatching）。
- 模擬訊息處理的相關機制（映射、繞送）。
- 模擬著名的 smalltalk-80 UI 模型：MVC（Model-View-Controller），此即 MFC 所謂的 Document-View 模型。
- 不足之處：文字模式，並未模擬 GUI 圖像視窗。



除了極少數情況，使我在簡化過程中必須因為技術牽連過廣而不得不變更 MFC 的原始作法（每個這樣的改變都有明確註解），MFCLite 基本上是原汁原味的 MFC 簡化版。MFCLite 不是創新，而是濃度極高的簡化。所有源碼的榮耀都屬於 Microsoft AFX 小組，侯捷唯一的貢獻就是把它濃縮為 4000 行的小東西，並且加上大量說明，使它適合教育目的。以下當我說 **MFC(Lite)** 如何如何，我說的是 MFC 和 MFCLite 的共同作法，如果我說 **MFC** 如何如何或 **MFCLite** 如何如何，我說的便分別是 MFC 或 MFCLite 各自的作法。



99.99999999999999% 的程式員綜其一生不需要寫個 application framework，但是 99.99999999999999% 的程式員綜其一生會用到一個 application framework。學習 application framework 核心技術（尤其在了一本好書的帶引下☺），可以將你的 OO 技術層次提昇到最高境界。請特別注意 application framework 如何架構骨幹、預留彈性空間，如何與 application 切割乾淨而又帶給 application 極大的好處。後續閱讀的過程中，你必須時時確認角色的變換：此刻所講的是 application framework 設計者的思維呢，還是 application 撰寫者的思維？

6.1.2 模組切割與編譯設定

做為一個本質極其複雜的 application framework，我們必須為它建構一個 classes 階層體系，在適當的 classes 內設計適當的介面，並設計適當的虛擬函式，給予客戶覆寫（override）的機會。設計過程中，整個 classes 階層體系是逐漸累進並循環改善的，沒有所謂線性發展、一次底定之事。雖然我嘗試以線性方式帶領你學習和觀摩，但請記住，實際的開發過程是來回不斷的反省和重構（refactoring）。我自己在一點一滴模擬、簡化、擴充的過程中，充分認清了每一個歷程和它們應有的

圖 6-2 列出 MFCLite3 的最終結果。灰色方塊是客戶（應用程式員）附加的 classes，白色方塊是 MFC 正規成份，其中分為程式主架構（*Application Architecture*）、視窗支援（*Window Support*）、群集（*Collections*）、檔案服務（*File Services*），以及 CObject 繼承體系外的一些 classes。基於本書前數章的討論，我們知道，這裡適用 single（單根）繼承和 public（公開）繼承。採用 single 繼承的好處是：堪用而且簡單，採用 public 繼承的原因則是：這裡的每一個上下類別都是 *is-a* 的關係。

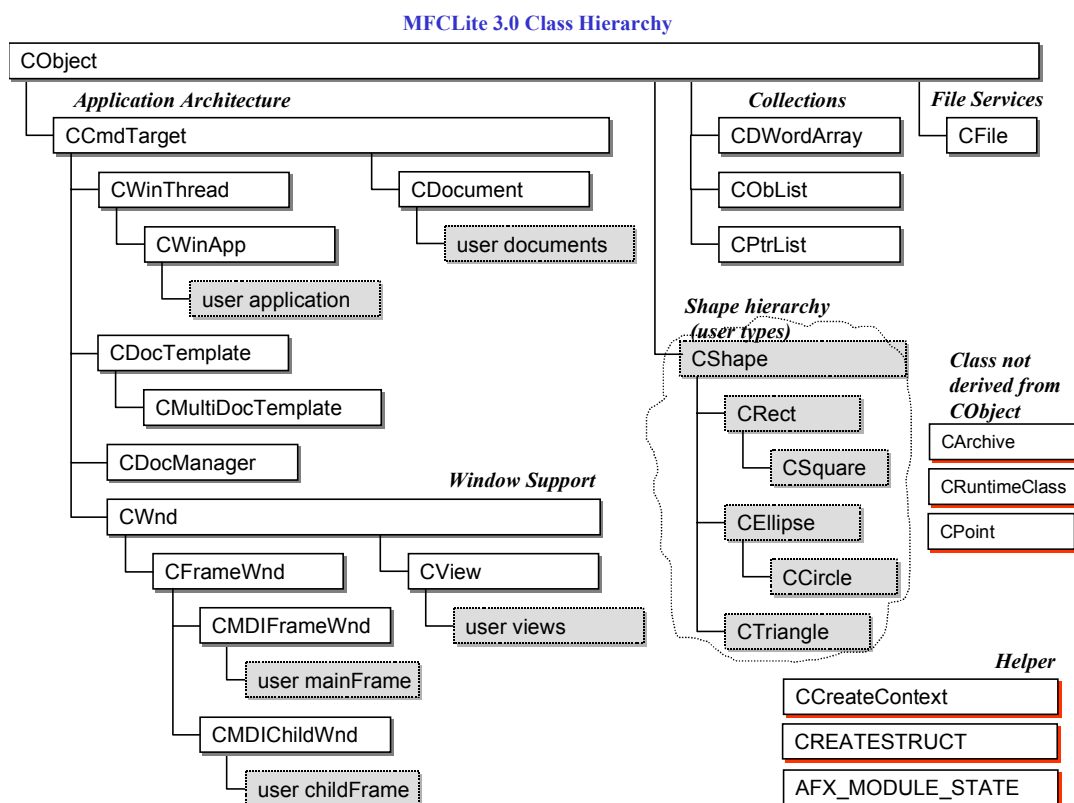


圖 6-2 MFC(Lite) 階層體系

Polymorphism in C++, 2e

要的前置宣告（forward declarations）：

```
// in mfclite.h
class CObject;
class CCmdTarget;
class CWinThread;
class CWinApp;
class CDocTemplate;
class CMultiDocTemplate;
class CDocument;
class CWnd;
class CFrameWnd;
class CMDIFrameWnd;
class CMDIChildWnd;
class CView;
class CFile;
class CDocManager;
class CDWordArray;
class CObList;
class CPtrList;
class CArchive;
class CRuntimeClass;
class CPoint;
```

檔案組態

Application framework 如此龐大複雜，動輒數百個檔案（MFC 4.2+ 有 279 個實作檔和 73 個表頭檔）。為了便於閱讀與學習，我將 MFCLite 濃縮在單一表頭檔（mfclite.h）和單一實作檔（mfclite.cpp）內。但是另有兩組獨特的定義，被我獨立出來放在 afxmsg_.h 和 afxres.h，這完全是為了模擬 MFC 的檔案組態。圖 6-3a 便是 MFC 的檔案組態，圖 6-3b 是 MFCLite 的檔案組態，其中綠底區域的 *shape* 類別體系，可視為與 MFCLite 相容的另一個模組（例如一個購入的繪圖類別庫）。灰色不帶陰影者，是應用程式員利用 MFC(Lite) 所開發的應用程式。

注意，圖 6-3a 的動態聯結在不同的作業平台上有不同的檔案格式與寫碼型式，而動態聯結並非本章關心的技術，因此我改用圖 6-3b 所示的靜態聯結——事實上 Visual C++ 整合開發環境中也有 MFC 靜態聯結選項。這麼一來本章的 MFCLite 便可跨任何編譯平台與任何作業平台（因為它使用標準 C++）。目前測試通過的編譯平台有 VC6, CB5, CYGWIN, GNU-GCC，測試通過的執行平台有 Win32, Solaris, FreeBSD, Linux。

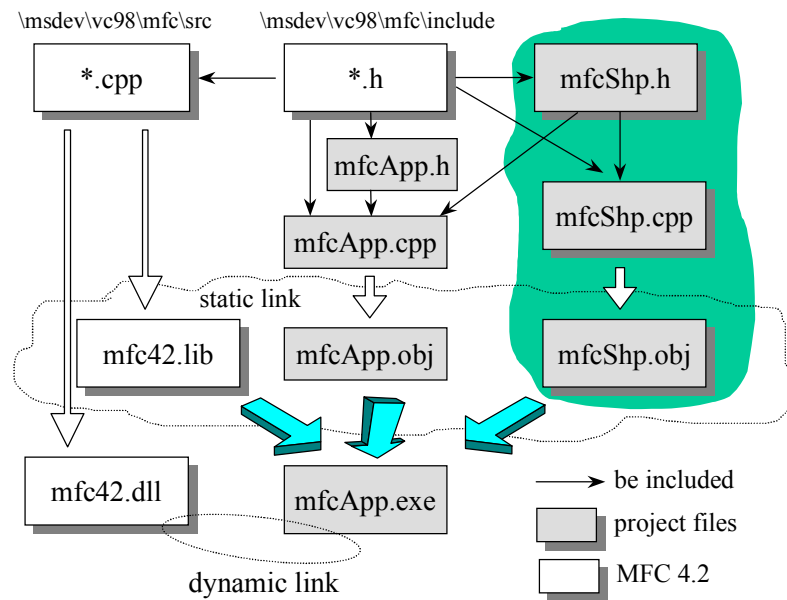


圖 6-3a MFC 和其應用程式。動態聯結。

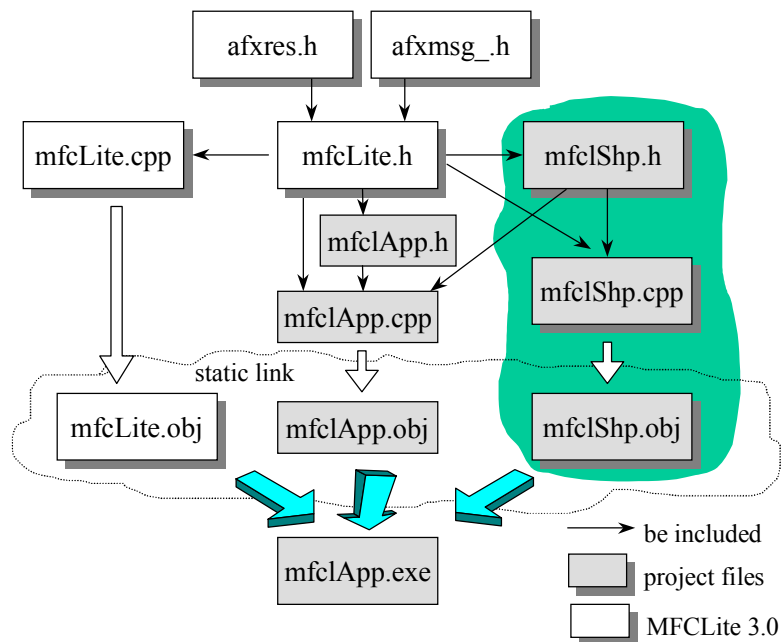


圖 6-3b MFCLite 和其應用程式。靜態聯結。

MFCLite 檔案路徑安排如圖 6-4，這也正是你從本書支援網站下載檔案並解壓縮後，獲得的路徑。

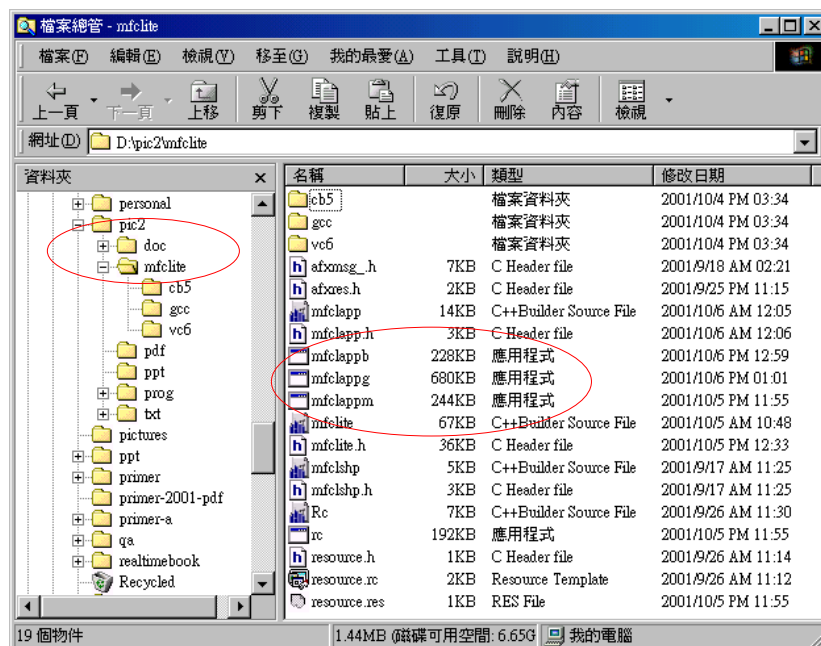


圖 6-4 MFCLite 和其應用程式 檔案路徑安排

命令行 (Command Line) 編譯設定

MFCLite 程式源碼無編譯版本之分。爲了以三種編譯器測試，我分別建立三個子目錄：VC6, CB5, GCC。三個編譯器的命令行模式 (command line mode) 環境設定請參考第 0 章。每個編譯環境製造出來的可執行檔，分別被拷貝到上層目錄，並加尾綴詞：'m' 代表 Microsoft Visual C++ 版本，'b' 代表 Borland C++Builder 版本，'g' 代表 GNU GCC 版本。每個子目錄內各有一個 makefile，只要在該目錄下執行工具程式 make (可能是 make.exe 或 nmake.exe)，便能夠將 MFCLite 連同其測試程式重新編譯連結。下面是適用於 VC6 的一個完整 makefile，請注意編譯選項 /GX 和 /GR。

mfclite.mak for Microsoft Visual C++ 6 :

```
# filename : makefile
```

```

# makefile for mfclite and App (plain C++ application)
# build : setting proper environment for VC 6.0+
#         then C:\pic2\mfclite\vc6> nmake <Enter>
all:mfclapp.exe \
    rc.exe

CC = cl
CFLAGS = -c -GR -GX

mfclapp.exe : mfclapp.obj mfclite.obj mfclshp.obj
    $(CC) mfclapp.obj mfclite.obj mfclshp.obj
    copy mfclapp.exe ..\mfclappm.exe

rc.exe : rc.obj
    $(CC) rc.obj
    copy rc.exe ..

..\resource.res : ..\resource.rc
    cd ..
    rc.exe
    cd vc6

mfclite.obj :    ..\mfclite.cpp ..\mfclite.h
    $(CC) $(CFLAGS) ..\mfclite.cpp

mfclshp.obj :    ..\mfclshp.cpp ..\mfclshp.h ..\mfclite.h
    $(CC) $(CFLAGS) ..\mfclshp.cpp

mfclapp.obj :    ..\mfclapp.cpp ..\mfclapp.h ..\mfclite.h ..\mfclshp.h
    $(CC) $(CFLAGS) ..\mfclapp.cpp

rc.obj :    ..\rc.cpp
    $(CC) $(CFLAGS) ..\rc.cpp

```

只要修改上述巨集 CC 和 CFLAGS，便可完成其他編譯環境下的 makefile，例如：

```

CC = bcc32                (CB5 的編譯器名稱)
CFLAGS = -c -w-aus -w-par (CB5 的編譯選項)

```

由於 CB5 編譯器不厭其煩地將所有「未被使用的參數」和所有「賦值後未被使用的變數」視為一種可能的錯誤並給予警訊，而我在發展 MFCLite3 的過程中為了讓介面相容於 MFC，保留了不少參數，實際未予運用，導致大量上述警訊。為避免看到那些令人煩心的東西，我把這兩個警訊以 `-w-aus -w-par` 抑制下來。

以下是 GCC 的編譯設定：

```

CC = g++
CFLAGS = -c

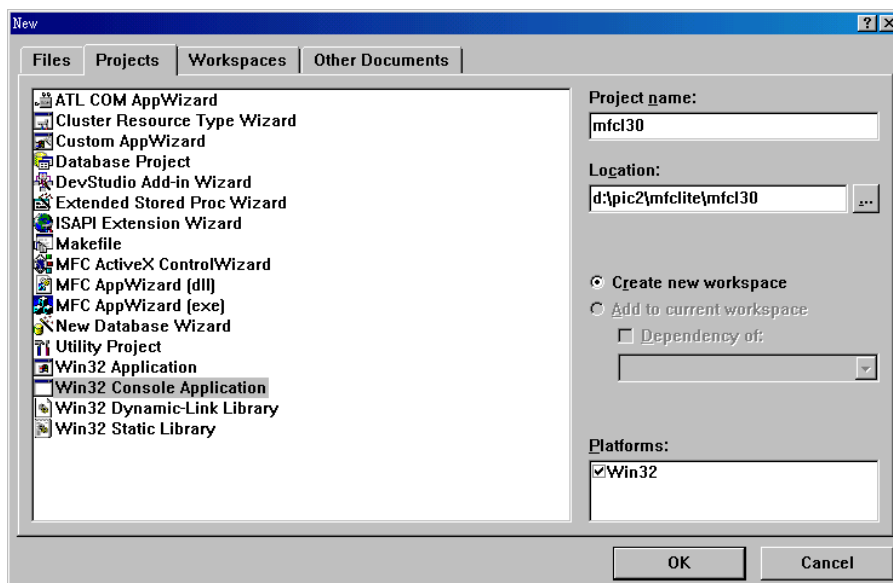
```

整合開發環境（Integrated Development Environment, IDE）

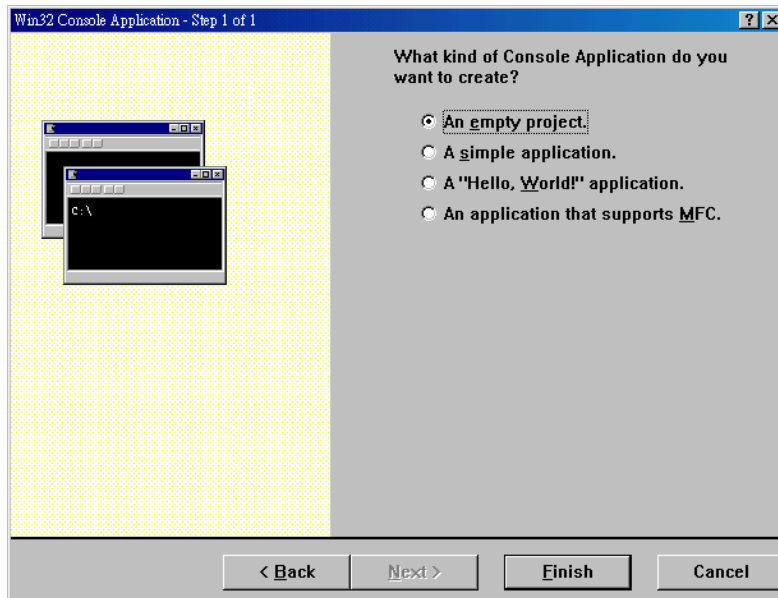
雖然 MFCLite 發展過程中主要以命令行模式來進行編譯（對於基礎教育，我的一貫理念是：儘可能選用最基本的工具），並因此預留了許多除錯機制（詳見 6.11 節），但最終爲了幾個難纏的臭蟲，我還是動用了整合開發環境（IDE）中的除錯器，才得以掃除障礙。這使我感覺，還是有必要告訴各位如何將 MFCLite 放進整合環境中 — 或許你會想要利用除錯器的步進功能（Step by Step），觀察本章未曾探討的某些細節。

棘手的是，各家整合開發環境有完全不同的使用方式。所幸我們需要的幾個設定動作，在各家整合環境中幾乎都類似或相通。以下我只以 VC++ 爲例。

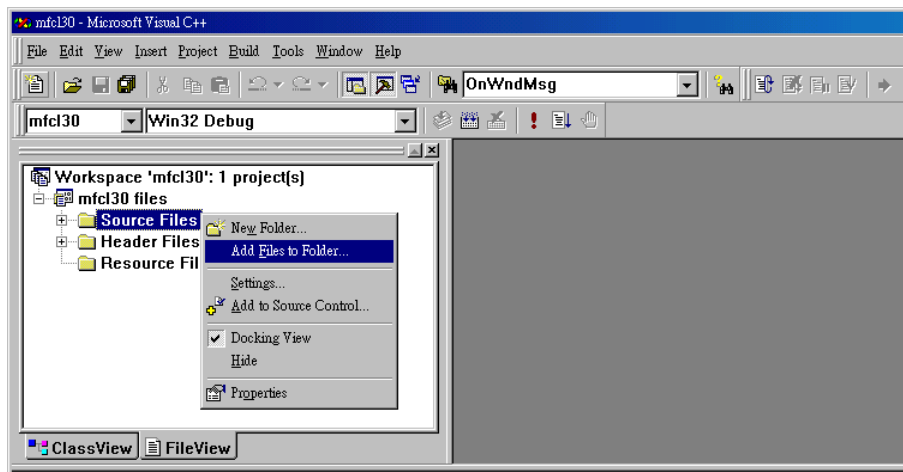
首先在 VC++ 整合環境中產生一個新專案（project）如下，右側的磁碟目錄請依個人需求自行設定：



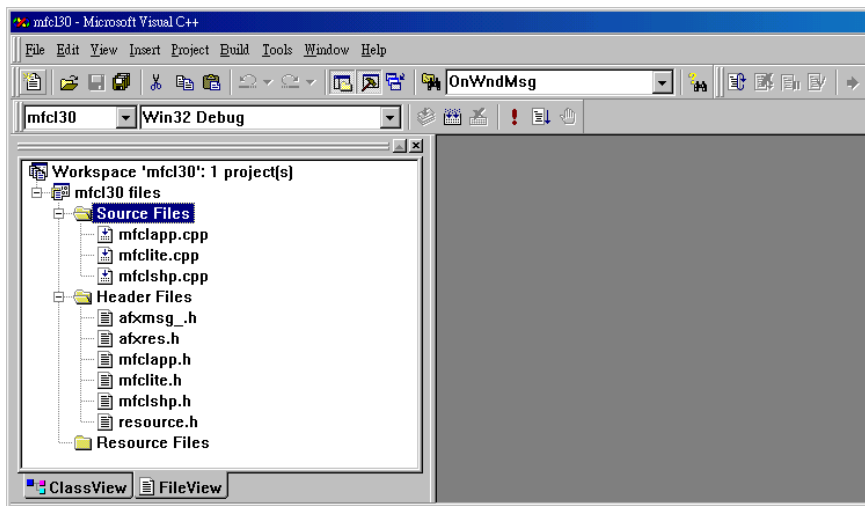
按下 [OK]，出現以下畫面：



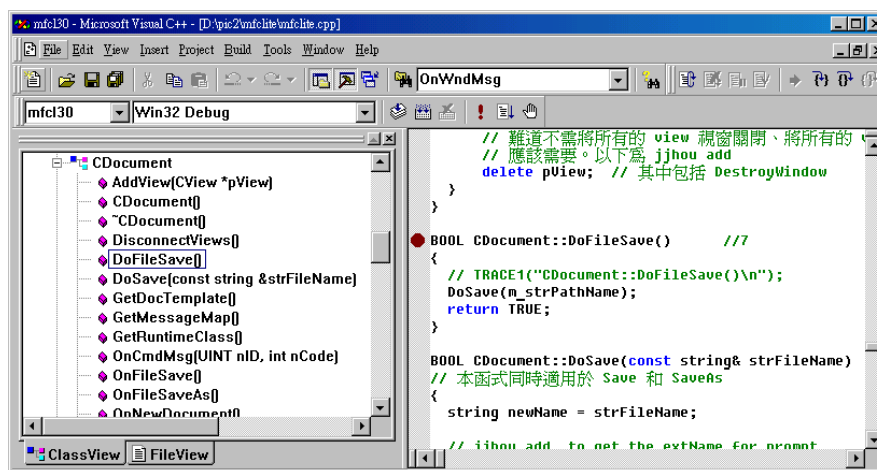
由於 MFCLite 本身是個完整程式，所以這裡應該選擇 [An empty project]。按下 [Finish] 即獲得一個除了專案維護檔 (*.DSP 和 *.DSW) 之外什麼也沒有的專案。接下來將所有實作檔 \pic2\mfclite*.cpp 一一加入 [Source Files] 資料夾內，並將所有表頭檔 \pic2\mfclite*.h 一一加入 [Header Files] 資料夾內，動作如下：



最後結果是：



此時打開上圖左下角的 [ClassView] 資料夾，你會發現整合環境已經將所有 classes 連同其內的 data members 和 member functions 都整理好了，方便我們瀏覽。只要以滑鼠按鍵雙擊其中任何一個欄位，右視窗立刻出現該欄位所在的檔案源碼並跳至該欄位所在位置。例如：

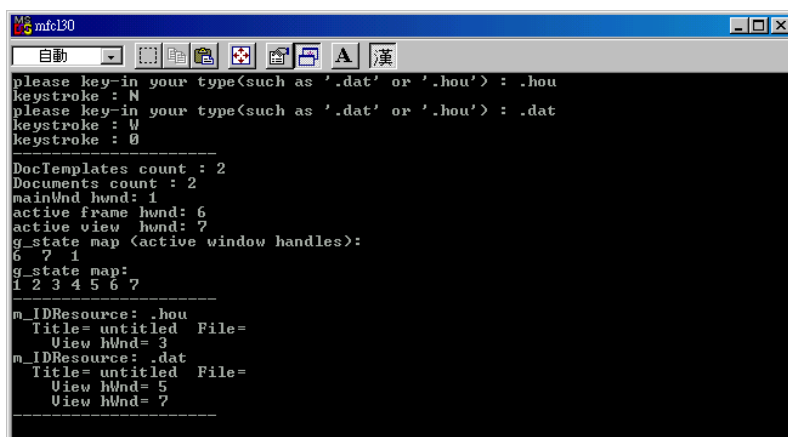


右視窗中的紅點，是我爲了除錯而設定的執行中斷點。關於中斷點的設定以及除錯器的各項功能，這裡就不介紹了，請閱讀 VC++ 使用手冊。

在 VC++ 整合環境中編譯 MFCLite，會產生數十個警告，這是命令行模式所不會

發生的。這些警告全都無關痛癢，可以不予理會。例如最大宗的警告說：identifier was truncated to '255' characters in the debug information，這是運用 templates 時幾乎無可避免會產生的警告，因為 template 往往被編譯器展開成為很長的句子。但是…唔…等等，為什麼 MFCLite 裡頭會有 templates 呢？我的模擬對象 — MFC — 本身並未使用任何 templates 技術（除了三個做為容器使用的所謂 *Typed Template Collections*：CTypedPtrArray, CTypedPtrList, CTypedPtrMap；但我並未模擬它們，因為它們和 application framework 核心技術無關）。答案是，MFCLite 為了模擬視窗系統的訊息機制和視窗管理機制，運用了 C++ 標準程式庫所提供的兩個容器：std::queue 和 std::map，它們都是 templates。稍後講述細節時再做說明。

整合環境中不論以發行（Release）或除錯（Win32 Debug）模式執行 mfclApp.exe，都會引發一個主控台（console，亦即 DOS 視窗），程式中所有輸往 cout（標準輸出設備）的文字都將被丟往該視窗：



```

please key-in your type(such as '.dat' or '.hou') : .hou
keystroke : N
please key-in your type(such as '.dat' or '.hou') : .dat
keystroke : U
keystroke : 
-----
DocTemplates count : 2
Documents count : 2
mainWnd hWnd: 1
active frame hWnd: 6
active view hWnd: 7
g_state map (active window handles):
6 7 1
g_state map:
1 2 3 4 5 6 7
-----
m_IDResource: .hou
Title= untitled File=
View hWnd= 3
m_IDResource: .dat
Title= untitled File=
View hWnd= 5
View hWnd= 7
-----

```

這和以命令行模式建造程式而後直接於 DOS 視窗下執行，形式和結果都完全相同。唯一不同就是，在整合環境中，我們可以令除錯器提供各種奧援。如此則不僅有助於找出臭蟲，也有助於我們徹底了解 MFCLite 的流程。舉個例子，如果你不是非常清楚 this 指標在虛擬函式的喚起過程中所扮演的角色，那麼 CMDIFrameWnd::OnCommand() 的行為絕對會讓你陰溝裡翻船（如果你對自己深具信心，不妨注意 6.6.5 節對此函式的說明是否帶給你衝擊），這時候除錯器的步進執行（Step by Step）就是一帖良藥。

6.1.3 設計樣式 "Template Method" 的大量實現

作為造橋鋪路的先鋒，application framework 必須為應用程式奠定基礎設施，它因而大量運用一種名為 **algorithm skeleton** 的設計手法。顧名思義，這種手法可以先將演算法的骨幹架設好，留下細節或無法代為處理的部分讓應用程式員填寫。這些留待填寫的部分，就是一個個等待被覆寫（overridden）的虛擬函式。

舉個例子，當應用程式偵測到使用者按下 [File/Open] 按鈕，準備開啓一份文件（對應於一個檔案），它大致應該這麼回應：

- 給出一個對話盒，詢問檔名。
- 檢查檔案狀態（存在否、是否已開啓...）。
- 以正確方式（「讀取模式」）開啓檔案。
- 讀取資料。
- 關閉檔案。
- 通知所有視圖區（Views），令它們將文件內容顯示出來。這是 MVC 模型下的工作方式，詳見 6.9 節。

這些動作中，只有 4 無法由 application framework 代勞，其他都可以先行完成（這也正是 application framework 的價值所在）。如果我們把整個 [File/Open] 動作視為一個演算法，application framework 便是先將這個演算法的骨幹架設好，留下最關鍵的、相依於應用程式的、無法代勞的部分，讓應用程式員補足。實際手法如下：

```
#include <iostream>
using namespace std;

// 這裡扮演 application framework 的角色
class CDocument
{
public:
    void OnFileOpen() // 此即所謂 Template Method
    {
        // 這是一個演算法，骨幹已完成。每個 cout 輸出權且代表一個實際應有動作
        cout << "dialog..." << endl;
        cout << "check file status..." << endl;
        cout << "open file..." << endl;
        Serialize(); // 它會喚起下面哪一個函式？
    }
};
```

```

        cout << "close file..." << endl;
        cout << "update all views..." << endl;
    }

    virtual void Serialize() { };
};

// 以下扮演 application 的角色
class CMyDocument : public CDocument
{
public:
    virtual void Serialize()
    {
        // 只有應用程式員自己才知道如何讀取自己的文件檔
        cout << "CMyDocument::Serialize()" << endl;
    }
};

int main()
{
    CMyDocument myDoc;

    // 假設主視窗表單中的 [File/Open] 被按下後，執行至此。
    myDoc.OnFileOpen();
}

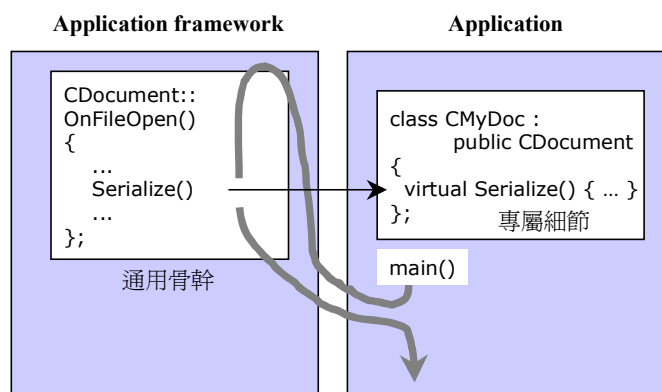
```

執行結果如下：

```

dialog...
check file status...
open file...
CMyDocument::Serialize()
close file...
update all views...

```



這種設計手法有一個響亮的名稱：**Template Method**，收錄於《Design Pattern》一書。這本著名的書籍收集了 23 個被廣泛運用的樣式 (patterns)，其中對於 Template Method 的描述如下：



Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

定義演算法骨幹，延緩其中某些步驟，使它們在 subclasses 中執行。Template Method 使 subclasses 得以重新定義演算法內的某些動作，而不需改變演算法的總體結構。

Template Method 散佈於任何一個 application framework 之中(當然包括微軟的 MFC 和侯捷的 MFCLite ☺)，處處可見，非常重要。這是因為，所謂 application framework 正是要預先設計好應用程式的骨幹，把細節的、無法代為處理的工作留給應用程式去完成。這不正是 Template Method 的定義和用途嗎！

6.1.4 MFCLite 的型別設定和常數定義

為標準型別再加一層定義，是許多 framework 的常用手法，目的是保留一層彈性，對於可攜性有點幫助。一些頗有用處但非語言標準的型別，例如 WORD，也可以通過這種手法定義出來。下面是 MFCLite 的型別定義，並標示出參考來源 (VC++ 檔案)：

```
#define TRACE0 printf          // 用以輸出系統錯誤訊息
#define TRACE1 printf          // 除錯用
#define TRACEmr printf         // 針對「訊息繞行」除錯用
typedef unsigned int  UINT;     // windef.h
typedef bool  BOOL;    // 若編譯器未支援 bool，可改用 int (windef.h)
typedef unsigned long DWORD;   // windef.h
typedef unsigned short WORD;   // windef.h
typedef unsigned char BYTE;    // windef.h
typedef long LONG;             // winnt.h
typedef void* LPVOID;          // windef.h (原為 void far*)
#define TRUE true              // 若編譯器未支援 true，可代之以 1 (windef.h)
#define FALSE false           // 若編譯器未支援 false，可代之以 0 (windef.h)
typedef FILE* HFILE;           // 以 ANCI-C files 模擬 Win32 files
typedef UINT HWND;             // window handle
typedef UINT WPARAM;           // windef.h
typedef LONG LPARAM;           // windef.h
```

```
typedef LONG          LRESULT;    // windef.h
#define ASSERT(x)    assert(x)
#define HIWORD(l) ((WORD)((DWORD)(l) >> 16) & 0xFFFF) // windef.h
#define LOWORD(l)     ((WORD)(l)) // windef.h
typedef char* LPTSTR;           // winnt.h
```

6.2 三大基礎建設之問題分析

就像所有 application framework（例如 MFC 或 OWL 或 VCL）一樣，也像許多大型 class libraries（如 Java 標準程式庫）一樣，MFC Lite 以一個根類別（root）開枝散葉。應用程式的任何 classes，如果希望享受 framework 帶來的好處，就必須繼承自這個根類別。此一類別在 MFC(Lite) 名為 CObject，在 Java 名為 Object，在 OWL 和 VCL 名為 TObject。

愛因斯坦說：讓事情簡單，但是不要過份簡單。這話很適用於 classes 介面設計。現在我們必須想想，CObject 提供怎樣的介面，才是最精簡又迫切必要的？以下所謂三大基礎建設，皆為期望獲得 MFC(Lite) 服務與好處的所有 classes 所祈求；唯有獲得這三項設施，才能融入 MFC(Lite) 所建構的體系，成為其服務對象：

1. 執行時期型別辨識（Runtime Type Identification）
2. 動態生成（Dynamic Creation）
3. 次第檔案讀寫（Serialization）

每一項設施都涵蓋前一設施。讓我們從下向上說起。

我們希望自己的程式能夠獲得物件永續（**persistence**）機能。所謂物件永續，就是將物件的狀態寫入永久儲存裝置內，以便下一次程式再執行時能夠恢復物件原狀。

所謂 **Serialization**（次第檔案讀寫），是指將資料依照程式設計者的規範，以一定次序寫入檔案，再以相同次序從檔案讀出。由於一連串資料的讀寫動作是：循序、不間斷、不跳躍，所以稱為「次第」檔案讀寫。這種物件永續手法並非唯一，也不一定最佳，卻最直觀。

的確，將資料依序寫入檔案再依序讀出，再直觀不過了。但是在直觀之中有一絲不易查覺的困難。將資料寫入檔案很簡單，但如果只是寫入未經加工的原始資料（raw data），讀出來就絕對無法還原為物件 — 我們連讀得的究竟是什麼樣的資料型態都不知道呢。因此資料的寫入勢必連帶附加「object 相關資訊」如 class 名稱、object 大小、容器實際大小等等。這好辦。然而當我們從檔案讀出一個 class 名稱，即便知道它繼承自 CObject，也很難藉此造出一個 object：

```
char className[50] = GetClassName(); // 假設 GetClassName() 可用。
CObject* pObj = new className;      // 失敗。
```

是的，任何 C++ 編譯器都要求你在 new 運算子之後放置真正的 class 名稱，不可以放置一個字串變數 — 即使其中內含 class 名稱。

解決之道是「全面比對法」：

```
string className = GetClassName(); // 假設 GetClassName() 可用。
if (className == "Class1"
    new Class1;
else if (className == "Class2"
    new Class2;
else if (className == "Class3"
    new Class3;
...
```

很笨，很暴力，但畢竟解決了問題。然而暴力只能治標，不能治本。

我們必須清楚，我們現在扮演著 application framework 的角色，期望幫助客戶（程式員）快速實現檔案讀寫功能。你當然不能假設你的客戶一定不會「自行設計一些 classes 並將其 objects 寫入檔案並於稍後讀出」。是的，你不能有這種天真的假設，因此你在 application framework 這一端採用上述的「暴力全面比對法」，就稱不上「全面」了 — 你不可能知道你的客戶會寫出什麼樣的 classes 名稱來。

如果不要求客戶端做一絲一毫的努力，上述這個問題在 C++ 中無解。唯一的辦法就是讓客戶端也做點動作，application framework 則設法將客戶端的勞動降至最低（那就比較不容易出錯）。

一個想法是：令每個 classes — 不論由 application framework 提供或由應用程式員自行設計 — 都供應一個函式，執行自己的「物件生成」動作。於是，名為 CFoo

的類別提供如下函式：

```
CObject* CFoo::CreateObject()    // 注意回返型別。多型的表現。  
{ return new CFoo; }
```

名為 CBar 的類別提供如下函式：

```
CObject* CBar::CreateObject()    // 注意回返型別。多型的表現。  
{ return new CBar; }
```

然後，如果我把 CreateObject() 設計為虛擬函式，問題是否就解決了呢？

不要手上拿了把榔頭，便把一切都看成釘子。虛擬函式很好用，但無法解決這個問題。首先，名稱比對問題還是沒有著落，其次，我們不能認為「如果比對成功，就呼叫該物件的虛擬函式 CreateObject()，把物件產生出來」。你不覺得其中有語病嗎？此刻的情況是，我們獲得了一個 class 名稱，希望藉此產生相應的 object，這和「手上有個型別未知的 object 而我們打算呼叫它的某個虛擬函式以實現因型別而異的動作」是不同的。後者是虛擬函式的典型用途。

為了進行比對，我們需要所有 classes 的名稱 — 包括來自 application framework 和來自應用程式的所有 classes。只要有這份資料在手得以進行比對，也就表示我們同時擁有了執行期型別辨識能力。如此，程式一旦動態獲得 class 名稱，便可採用比對法（也是唯一可行的辦法），再藉由比對結果，呼叫對應的 CreateObject() 函式。這些對應的 CreateObject() 函式，內容如同上述，必須設計為 static，因為我們很可能在尚未有任何 object 之前便呼叫它 — 它正是為了產生 object。

在這裡，如果你對設計樣式有點研究，你可能會以為 CreateObject() 是所謂的 **factory method**（或名 **virtual constructor**）。不，味道有點像，但不完全是。詳見第 7 章說明。

這樣的設計思維逐漸勾勒出一幅實作藍圖：每個 application 在執行初期，都（必須）擁有一整套「與其所有 classes 呼應」的結構，提供所有 classes 名稱和相應的 CreateObject() 函式。這個結構我名之為「類別型錄網」。稱其為型錄（category）是因它提供查找功能，稱其為「網」則是因為它和存在的所有 classes 一一對應，形成樹狀（但實際以 linked list 即可完成，稍後可見）。

爲方便客戶，MFC(Lite) 提供三組巨集（對應不同的三層基礎服務），讓客戶端輕鬆加入適當地點。短短兩行（每組巨集只有兩行）便能輕鬆建立三項基礎設施。如果客戶被限制必須運用 application framework 搭配的開發工具，不得自己動手寫碼，那麼程式碼自動產生器（例如 VC++ 的 AppWizard）負責產出標準的應用程式骨幹，就更萬無一失了：你的客戶不至於寫出錯誤的應用程式骨幹，你的 application framework 不至於因爲應用程式端頻出狀況而灰頭土臉，你也不至於聽到任何莫名其妙的錯誤或嫌麻煩的怨言。

第三層基礎建設「次第檔案讀寫」的完整流程相當複雜，然而其中最關鍵、首先需要解決的技術就是前述的「動態生成」：執行期動態獲得 class 名稱，並據此產生相應的 object。

動態生成不僅應用於物件的檔案讀寫，還發生在許多地方。這是因爲 application framework 內的許多設計都做了彈性考量，允許客戶在某些場合指定運用自己比較喜歡的 classes。爲了這樣的彈性，勢必得在該處實施動態生成。舉個例子，MFC(Lite) 以這種方式讓客戶指定 MVC 模型（6.9 節）所需的三個 classes：

```
CMultiDocTemplate* pDocTemplate;  
pDocTemplate = new CMultiDocTemplate(  
    IDR_MYDOCTYPE,  
    RUNTIME_CLASS(CMyDocument),  
    RUNTIME_CLASS(CChildFrame),  
    RUNTIME_CLASS(CMyView) // 你也可以搞個 CMyView2  
);  
AddDocTemplate(pDocTemplate);
```

而後 MVC 模型即能夠在開檔、讀檔之時，根據上述登錄的 classes，動態生成相應的 objects。諸如此類的動態生成行爲，在 application framework 中佔很重要的地位。6.9 節有更詳細的說明。

6.2.1 類別型錄網與 CRuntimeClass

要實現「類別型錄網」，我們需要思考兩件事：(1) 網中的每一個節點採用怎樣的格式？(2) 什麼時機以怎樣的型式將整個網串起？

下面的 CRuntimeClass 便是「類別型錄網」的節點結構：

```

struct CRuntimeClass
{
    char* m_lpszClassName;    // class 名稱
    int m_nObjectSize;        // object 大小
    UINT m_wSchema;           // 版本編號 (schema number)
    CObject* (*m_pfnCreateObject)(); // 函式指標
    CRuntimeClass* m_pBaseClass; // 指向 base，用以模擬 tree。

    // 以下爲了維護簡單的 linked list
    CRuntimeClass* m_pNextClass; // 下一節點
    static CRuntimeClass* pFirstClass;
    // 起始節點 (只需一份，所以爲 static)

    // for Dynamic Creation (動態生成)
    CObject* CreateObject();

    // for Persistence (物件永續)。以下兩函式定義於 6.5.3 節。
    void Store(CArchive& ar) const;
    static CRuntimeClass* Load(CArchive& ar, UINT* pwSchemaNum);
};
// 「版本號碼」和「物件永續」相關欄位將在稍後小節中詳述。

```

這樣，我們便畫出了它的物件模型如圖 6-5。當然你知道，第 3 章已經告訴你，member functions 不在物件記憶體區塊內。

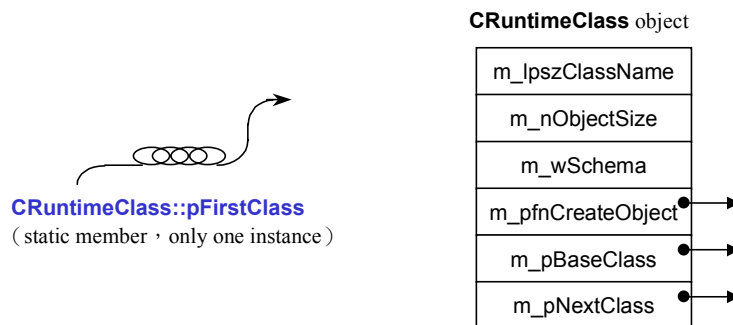


圖 6-5 CRuntimeClass 物件模型

再來的問題是，什麼時候，以怎樣的型式，將整個「類別型錄網」串起？如果我們能夠設計出良好的巨集，讓客戶端使用，客戶就不必直接面對上述這個不算不複雜的結構。巨集可用來填充資料結構的欄位，這是大家常有的經驗，但如果我們的巨集還能夠自動將「類別型錄網」串接起來，客戶端就更省事了。而且，這

個串接動作的發生時機必須非常非常早，因為也許程式一啟動就需要執行動態生成（這在 application framework 中是絕對可能的事）。

這樣的巨集可能嗎？可能！詳見 6.3 節。

我們希望，每個 MFC(Lite) 應用程式一執行起來，就有一大串「類別型錄網」在手，如圖 6-6。這樣一個類別型錄網，對應於圖 6-7 的樹狀想像。

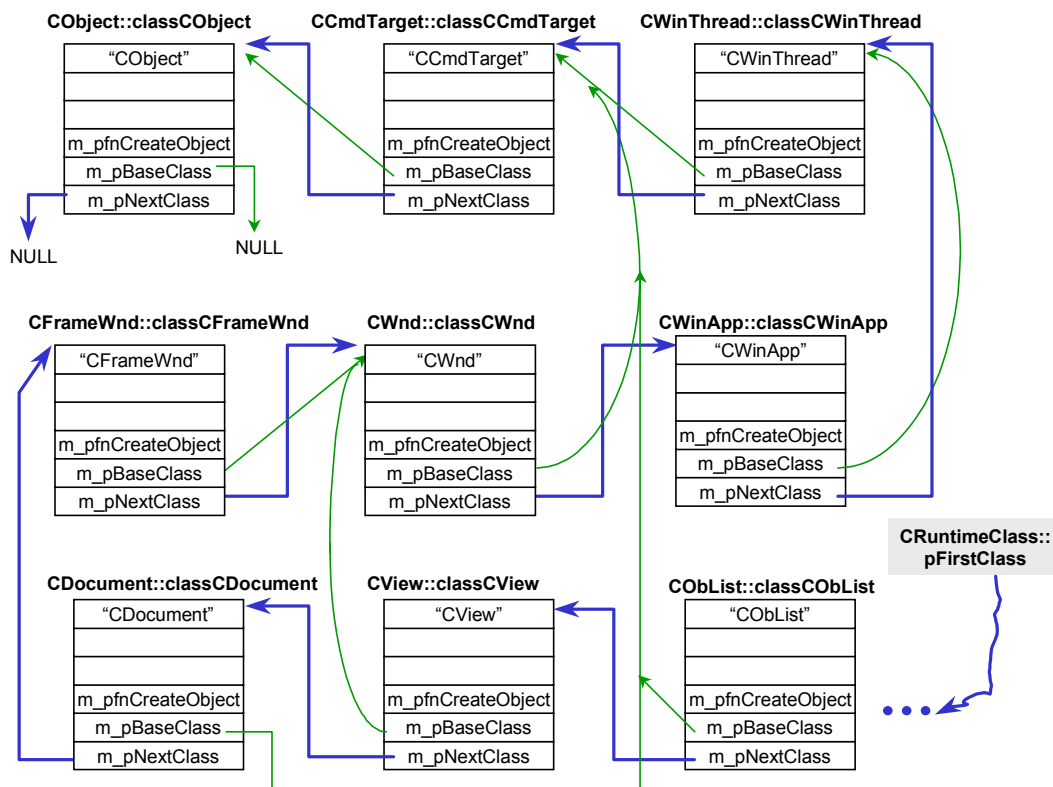


圖 6-6 MFC(Lite) 類別型錄網

請注意，在這裡，MFC(Lite) 採用了特殊的命名方式：每個 class 名稱加上前綴詞 "class"，即成為對應之 CRuntimeClass object 名稱。這些 objects 以複合型式（composition）成為一個個 class member（這是合理的設計），因此才會出現諸如 CObject::classCObject, CWinApp::classCWinApp, CObList::classCObList

等等名稱。此外，這些 `CRuntimeClass` object 應該只與 `class` 呼應；不論相應的 `class` 產出多少 `objects`，其 `CRuntimeClass` object 都應該只有一份。因此它應該是其對應的 `class` 內的一個 `static member`。

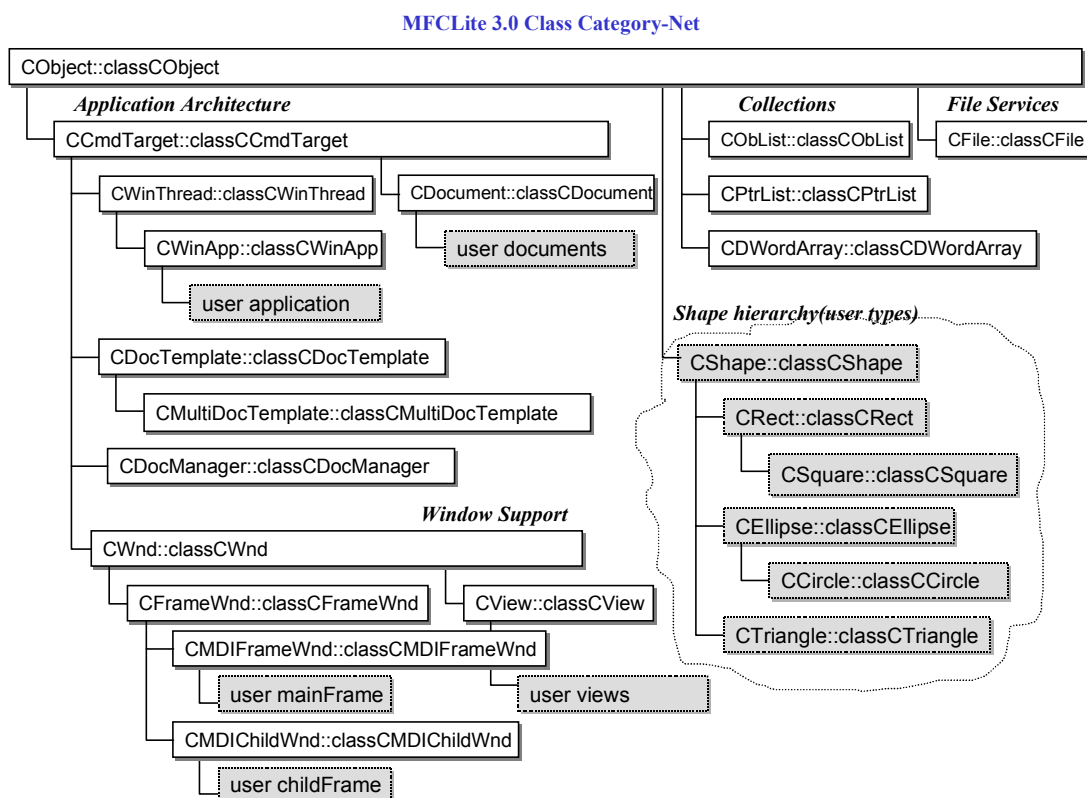


圖 6-7 類別型錄網的樹狀想像。請對應觀察圖 6-2 的 MFC(Lite) 類別階層體系

把上述的 `CRuntimeClass` 納入考量之後，一個 `CFoo` 應該有這樣的宣告和實作：

```
class CFoo : public CObject
{
public:
    static CRuntimeClass classCFoo; // 爲了支援三大基礎建設
    static CObject* CreateObject(); // 如果需要「動態生成」
    ...
};
... // CFoo::classCFoo 的相關設定 (相當繁瑣)
... // CFoo::CreateObject() 的定義 (return new CFoo;)。
```


但是這樣曝露了太多細節，造成應用程式員撰碼時的負擔。稍後我們要開發一些巨集，減輕這些瑣屑的勞役。

有了這樣的具體架構做為後援，現在我們再整理一次動態生成概念。任何時候想要執行動態生成，有三種方式：

1. 如果你已確知 class 名稱，就直接呼叫該 class 的 static `CreateObject()`。不過請注意，動態生成往往發生在我們不確切知道 class 名稱的情況下，因為作為 application framework 的設計者，我們正在撰寫「演算法骨幹」，無從得知執行期動態獲得的確切 class 名稱。
2. 根據執行期獲得的某個 class 名稱，找出其 `CRuntimeClass` object，再呼叫其中的 `(*m_pfnCreateObject)()`。這是個好方法。
3. 為了友善介面，再加入一個 `CRuntimeClass::CreateObject()`（如圖 6-8 最下），於是演變成：根據執行期獲得的某個 class 名稱，找出其 `CRuntimeClass` object，再呼叫後者的 `CreateObject()`。這是最好的方法。

圖 6-8 表現出上述三種作法。

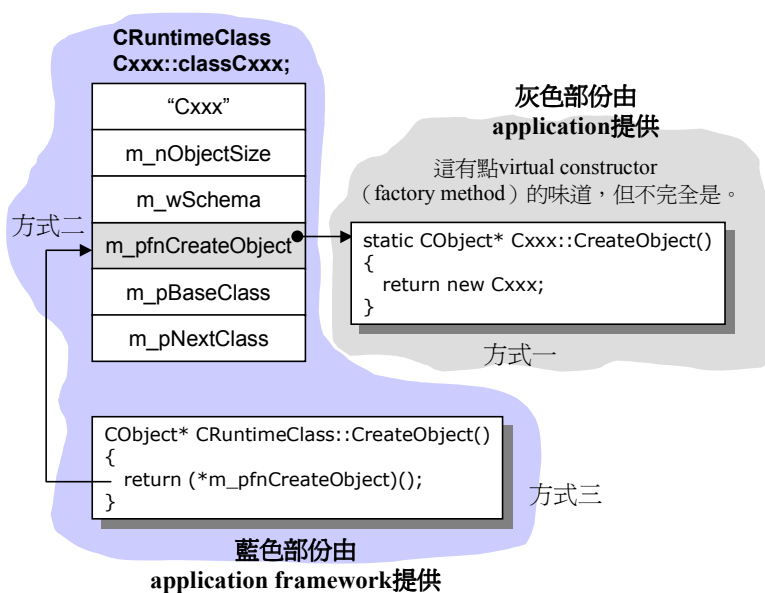


圖 6-8 動態生成與 `CRuntimeClass` object

6.2.2 巨集 v.s. 虛擬函式

OO（物件導向編程技術）並不強調巨集的價值，也不甚鼓勵大家使用巨集。但，能夠抓到老鼠的就是好貓，更何況先前我們在三大基礎建設的設計中已經討論過了，有時候虛擬函式派不上用場。

MFC(Lite) 動用兩大套巨集，一套負責三大基礎建設，一套負責稍後討論的訊息映射。這些巨集的名稱是：

- `DECLARE_DYNAMIC / IMPLEMENT_DYNAMIC` 基礎建設第一層
- `DECLARE_DYNCREATE / IMPLEMENT_DYNCREATE` 基礎建設第二層
- `DECLARE_SERIAL / IMPLEMENT_SERIAL` 基礎建設第三層
- `BEGIN_MESSAGE_MAP / END_MESSAGE_MAP` 訊息映射表的形成
- `ON_COMMAND / ON_WM_CREATE / ON_WM_MOUSEMOVE / ...` 設定訊息處理常式

動態生成無法靠虛擬函式達成，原因已於上節提過。至於訊息映射，主要目的是希望讓想要攔截訊息的 `classes` 能夠如願以償。好，如果我們把各種訊息處理常式（`message handler`）都設計為虛擬函式，讓所有 `CObject-derived classes` 繼承下去，讓有意覆寫的人覆寫，那麼也可以達到目的。這固然是不錯的想法，但只能說你 OO 考了一百分，卻落了個不食人間煙火。要知道，任何一個視窗系統的訊息都不下百千種，如果為它們一一設計各自的處理常式，並令它們統統成為虛擬函式，一層層繼承下去，額外負荷（`overhead`）太重了。現實程式中，各個 `classes` 所關心的訊息也許只有十數種（本章結束後你會發現，區區 6 種訊息就讓一個 `MFC Lite` 應用程式跑起來）。如果能夠不付出沉重的賦稅又達到既定的目標，無論 `application` 或 `application framework` 都會擊節讚賞 — 非純 OO 又何妨 ☺

使用巨集的另一個好處是，搭配兩個十分奇特（但符合 C++ 標準）的運算子，巨集竟然成了程式碼產生器，效果保證讓你目瞪口呆。稍後我們便來剖析這些曼妙的巨集。

6.2.3 根類別 CObject

根據截至目前的討論，我們將 CObject 的介面設計如下：

```
class CObject
{
public:
    virtual ~CObject() { }    // virtual dtor 絕對必要，見 2.x 節。
    BOOL IsKindOf(const CRuntimeClass* pClass) const;
    virtual CRuntimeClass* GetRuntimeClass() const;
    virtual void Serialize(CArchive&);
public:
    static CRuntimeClass classCObject; // 只要一份就好，所以是 static
};
```

其中 IsKindOf() 用來支援執行期型別辨識，GetRuntimeClass() 用來間接支援動態生成，Serialize() 用來支援次第檔案讀寫。後兩者的實際動作因型別而有變異，所以設計為 virtual。前兩者的動作不會改變任何 data member，所以設計為 const。CRuntimeClass object 只應有一份，所以必須為 static member。

下面便是三個 member functions 的實作細目：

```
BOOL CObject::IsKindOf(const CRuntimeClass* pClass) const
{
    // 首先定位自己
    CRuntimeClass* pClassThis = GetRuntimeClass();

    // 然後從自己位置開始，往上走遍整個族系，尋找與 pClass 吻合者。
    while (pClassThis != NULL)
    {
        if (pClassThis == pClass)
            return TRUE;
        pClassThis = pClassThis->m_pBaseClass; // 注意，找的是 Base
    }
    return FALSE;    // 走完了，沒找到。
}
```

```
CRuntimeClass* CObject::GetRuntimeClass() const
{
    return &CObject::classCObject;    // 取物件位址
}
```

```
void CObject::Serialize(CArchive&)
{
}
```

`GetRuntimeClass()` 用來取出自己所對應的 `CRunTimeClass` object 的位址。有了這個位址，我們便可以在 `IsKindOf()` 函式中實現「類別名稱比對法」。於是，當我們於程式執行期獲得一個型別未知的 object 或 pointer 或 reference，便可如此判定其實際型別是否為某種 class（或其後裔）：

```
// 假設以下 obj 和 pobj 和 robj 為執行期獲得的 object 和 pointer 和 reference.
// 假設它們的 class 都衍生自 CObject。
obj.IsKindOf(RUNTIME_CLASS(CDocument));
pobj->IsKindOf(RUNTIME_CLASS(CView));
robj.IsKindOf(RUNTIME_CLASS(CCmdTarget));
```

其中 `RUNTIME_CLASS` 是個巨集，用來取出某個 class 對應的 `CRunTimeClass` object，功能和 `GetRuntimeClass()` 相同，但某些應用場合（例如上述所列）需要這樣一個同等價值的巨集。實際定義詳見 6.3 節。

`Serialize()` 是個空函式，為什麼不設計為純虛擬函式呢？如果那麼做，許多 non-serializable `CObject`-derived class 原本不需覆寫 `Serialize()`，將因此變得必須覆寫，這並不好。此外，保留一個空函式，也就保留了一些彈性，將來也許有必要在這裡加上一些通用性動作。不過這麼一來每個 serializable `CObject`-derived class 都必須在自己的 `Serialize()` 函式起始便呼叫 `CObject::Serialize()`，才能保證與將來的 MFC(Lite) 版本相容¹。

至此，我們還需將 `CObject::classCObject` 的欄位填妥，才算成功德圓滿：

```
static char szCObject[] = "CObject";
struct CRunTimeClass CObject::classCObject =
{ szCObject, sizeof(CObject), 0xFFFF, NULL, NULL, NULL };

static AFX_CLASSINIT _init_CObject(&CObject::classCObject);
// AFX_CLASSINIT 用來串接「類別型錄網」，詳見 6.3 節。
```

稍後你即將看到的三大巨集，基本上做的也就是上述 `CObject` 所做的工作，那麼何不讓 `CObject` 直接使用三大巨集之一呢？這是因為 `CObject` 並無父類別，而三大巨集中的 `IMPLEMENT_X` 巨集卻要求使用者必須指定父類別。

¹ 是的，MFC 的確給了使用者這樣的規範，要求每一個 `Serialize()` 函式必須在第一時間呼叫 `CObject::Serialize()`。

6.2.4 程式產生器：巧妙的 # 運算子和 ## 運算子

本書第 4 章已經介紹過 # 運算子和 ## 運算子。它們都會改變程式源碼。如果把它們應用在巨集身上，就出現「程式碼產生器（code generator）」的效果。

為什麼需要「程式碼產生器」？先前的設計，包括 MFC(Lite) 類別型錄網所需的 CRuntimeClass，以及三大基礎建設所需的介面 IsKindOf()、GetRuntimeClass()、Serialize()，都是 application framework 提供的服務，都最好不要增加應用程式員的負擔。然而由於它們的複雜性，一旦所有細節呈現在應用程式員面前，無可避免會帶來很大的困擾（學習到這裡，你不也頭疼欲裂了嗎☹）。因此，配合先前規劃的命名方式，MFC(Lite) 提供了一些巨集，為應用程式員分憂解勞。

按說，每一個 CObject-derived class，如果希望享受第一層基礎建設，應該有以下介面：

```
class CFoo : public CObject
{
public:
    static CRuntimeClass classCFoo;
    // BOOL IsKindOf() 繼承自 CObject，不必再寫
    virtual CRuntimeClass* GetRuntimeClass() const;
    ...
};
... // CFoo::classCFoo 相關設定
... // CFoo::GetRuntimeClass() 定義式
```

如果希望享受第二層基礎建設，應該有以下介面：

```
class CFoo : public CObject
{
public:
    static CRuntimeClass classCFoo;
    virtual CRuntimeClass* GetRuntimeClass() const;
    static CObject* CreateObject();
    ...
};
... // CFoo::classCFoo 相關設定
... // CFoo::GetRuntimeClass() 定義式
... // CFoo::CreateObject() 定義式。
```

如果希望享受第三層基礎建設，應該有以下介面（詳見 6.5 節）：

```

class CFoo : public CObject
{
public:
    static CRuntimeClass classCFoo;
    virtual CRuntimeClass* GetRuntimeClass() const;
    static CObject* CreateObject();
    friend CArchive& operator>>(CArchive& ar, const class_name*& pObj);
    friend CArchive& operator>>(CArchive& ar, class_name*& pObj);
    ...
};
... // CFoo::classCFoo 相關設定
... // CFoo::GetRuntimeClass() 定義式
... // CFoo::CreateObject() 定義式。
... // 兩個 operator>> 定義式。

```

爲了減輕程式員的負擔，framework 設計者希望應用端只要這麼寫就好了：

```

class CFoo : public CObject
{
    DECLARE_DYNAMIC(CFoo);          // 或 DECLARE_DYNCREATE
                                     // 或 DECLARE_SERIAL
};

IMPLEMENT_DYNAMIC(CFoo, CObject);   // 或 IMPLEMENT_DYNCREATE
                                     // 或 IMPLEMENT_SERIAL

```

這便要求上述巨集扮演程式碼產生器的角色。關於巨集的展開，乃至於 # 運算子和 ## 運算子如何發揮作用，詳見 6.3~6.5 節。

6.2.5 容器相關類別：CDWordArray, CObList, CPtrList

現在，讓我們從極度複雜的 application framework 中抽離，先完成一些工具類別，以備稍後運用。MFC(Lite) 提供有三個容器類別，分別是：

1. CDWordArray：一個動態陣列，元素型別爲 DWORD (unsigned long)。
2. CObList：一個雙向串列，每個元素都是 CObject-derived object。
3. CPtrList：一個雙向串列，每個元素都是指標。

其中 (1) 和 (2) 都支援 Serialization，所以必須選用第三層巨集。(3) 的每一個元素用來存放一個 void* 指標；由於所指目標不明確，不確定爲 CObject-derived object，所以無法支援檔案讀寫，只能支援第一層 RTTI 基礎設施。以下是它們的介面，實作細節見書附源碼。

```

class CDWordArray : public CObject
{
    DECLARE_SERIAL(CDWordArray)
public:
    CDWordArray();
    ~CDWordArray();

    // 在尾端增加一個元素
    int CDWordArray::Add(DWORD newElement);
    // 取出 nIndex 位置上的元素。傳回一個右值 (rvalue)
    DWORD CDWordArray::GetAt(int nIndex) const;
    // 取出 nIndex 位置上的元素。傳回一個左值 (lvalue)。
    DWORD& CDWordArray::ElementAt(int nIndex);
    // 取出 nIndex 位置上的元素。傳回一個右值 (rvalue)。
    DWORD CDWordArray::operator[] (int nIndex) const;
    // 取出 nIndex 位置上的元素。傳回一個左值 (lvalue)。
    DWORD& CDWordArray::operator[] (int nIndex);
    // 詢問目前的 array 大小。
    int CDWordArray::GetSize() const;
    // 在 nIndex 位置上設定元素值。如果 array 大小不足，應該成長。
    void SetAtGrow(int nIndex, DWORD newElement);
    // 設定大小與單次成長量。
    void SetSize(int nNewSize, int nGrowBy = -1);
    // 資料循序讀寫 (至檔案)
    void Serialize(CArchive&);

protected:
    DWORD* m_pData;        // 元素儲存處
    int m_nSize;            // 目前大小 (元素個數)
    int m_nMaxSize;        // 目前最大空間
    int m_nGrowBy;         // 單次成長量
};

```

```

class CObList : public CObject
{
    DECLARE_SERIAL(CObList)
protected:
    struct CNode {          // 串列節點結構
        CNode* pNext;
        CNode* pPrev;
        CObject* data;      // 每個元素都是 CObject*
    };
public:
    CObList();
    ~CObList();

    // 目前的元素個數
    int CObList::GetCount() const;
    // 串列是空的嗎？

```

```

    BOOL CObList::IsEmpty() const;
    // 取得最前端（第一個）節點
    POSITION CObList::GetHeadPosition() const;
    // 取得 rPosition 位置的下一個節點元素
    CObject* CObList::GetNext(POSITION& rPosition) const;
    POSITION AddTail(const CObject* newElement); // 尾端加一個新元素
    void RemoveAll();                          // 移除所有元素
    void Serialize(CArchive&);                 // 資料循序讀寫（至檔案）

protected:
    CNode* NewNode(CNode*, CNode*);           // 增加一個新節點
    CNode* m_pNodeHead;                       // 指向頭元素
    CNode* m_pNodeTail;                       // 指向尾元素
    int m_nCount;                             // 目前元素個數
};

```

```

// CPtrList 的精巧實作 (memory pool & GC) 見 6.12.1 節
class CPtrList : public CObject
{
    DECLARE_DYNAMIC(CPtrList)
protected:
    struct CNode {
        CNode* pNext;
        CNode* pPrev;
        void* data;      // 每個元素都是 void
    };
public:
    CPtrList(int nBlockSize=10); // 預設每個 CPtrList object 內有 10 個 CNodes
    ~CPtrList();
    void RemoveAll();
    POSITION AddTail(void* newElement);
    void RemoveAt(POSITION position);
    void* RemoveHead();
    POSITION Find(void* searchValue, POSITION startAfter = NULL) const;
        // 預設從頭端 HEAD 開始找出（第二參數預設值為 NULL）。
        // 如果沒找到就傳回 NULL。
    BOOL CPtrList::IsEmpty() const;
    int CPtrList::GetCount() const;
    void*& CPtrList::GetHead();
    void* CPtrList::GetHead() const;
    POSITION CPtrList::GetHeadPosition() const;
    POSITION CPtrList::GetTailPosition() const;
    void*& CPtrList::GetNext(POSITION& rPosition);
    void* CPtrList::GetNext(POSITION& rPosition) const;
protected:
    CNode* m_pNodeFree; // jjhou: garbage list
    CNode* m_pNodeHead;
    CNode* m_pNodeTail;
    int m_nCount;

```



```

struct CPlex* m_pBlocks; // CPlex 見 6.12.1 節。
int m_nBlockSize;       // 此值表示每個 CPlex 內將有多少個 CNodes
CNode* NewNode(CNode*, CNode*);
void FreeNode(CNode*);
};

```

我得帶你徹底研究 CDWordArray 和 CObList 的 Serialize() 函式，這有助於我們徹底了解 object 被寫入檔案後的形式，從而了解其優缺點。

CDWordArray::Serialize()

```

void CDWordArray::Serialize(CArchive& ar)
{
    CObject::Serialize(ar); // 首先呼叫 base's serialize()

    if (ar.IsStoring())
    {
        ar.WriteCount(m_nSize);           // 寫入元素個數
        ar.Write(m_pData, m_nSize * sizeof(DWORD));
        // 一次寫入所有元素。唯有佔用連續空間的容器，才得如此。
    }
    else
    {
        DWORD nOldSize = ar.ReadCount(); // 讀出元素個數
        SetSize(nOldSize);               // 設定 array 大小
        ar.Read(m_pData, m_nSize * sizeof(DWORD));
        // 一次讀出所有元素。唯有佔用連續空間的容器，才得如此。
    }
}

```

上述函式的唯一參數型別 CArchive 將於 6.5.3 節介紹，請暫時想像它代表一個檔案。對它做 operator<< 便是將資料寫入檔案，對它做 operator>> 便是從檔案讀取資料。

由於 array 在記憶體中佔用連續空間，因此 array 的檔案讀寫非常簡單：先讀寫元素個數，然後一次讀寫所有元素。如果是讀取（而非寫入）動作，必須根據讀得的元素個數預先配置足夠空間。如果元素是個 class object（而非基本型別），我們就得處理其相關的「class 相關資訊」— 然而 CDWordArray 的每個元素都是基本型別 DWORD，因此可以省下這個動作。

下面是 DWordArray 的檔案讀寫測試，獲得圖 6-9 的輸出。

```

CDWordArray* pDArray; // 設計為 CMyDocument 的一個 data member。
...
CMyDocument* pDoc = GetDocument();
// 以上動作於 6.9 節介紹，用來取得一個指標，指向 CMyDocument object
pDoc->pDArray->Add(9);
pDoc->pDArray->Add(7);
pDoc->pDArray->Add(5);
pDoc->pDArray->Add(3);
pDoc->pDArray->Add(1);
ar << pDArray;          // ar 是一個 CArchive object，6.5.3 節介紹。
                        // 這樣便將整個 array 寫入檔案。

```

```

Turbo Dump  Version 3.1 Copyright (c) 1988, 1992 Borland International
              Display of File JJ.HOU

000000: 01 00 0B 00 43 44 57 6F  72 64 41 72 72 61 79 05  ....CDWordArray.
000010: 00 09 00 00 00 07 00 00  00 05 00 00 00 03 00 00  .....
000020: 00 01 00 00 00                .....

```

記憶體內容 (big-endian, 正放)	實值	意義
00 01	1	schema no (版本號碼)
00 0B	11	class 名稱長度
43 44 57 6F 72 64 41 72 72 61 79	CDWordArray	class 名稱
00 05	5	陣列長度 (元素個數)
00 00 00 09	9	第一元素值 9
00 00 00 07	7	第二元素值 7
00 00 00 05	5	第三元素值 5
00 00 00 03	3	第四元素值 3
00 00 00 01	1	第五元素值 1

圖 6-9 CDWordArray 檔案讀寫測試結果。上為檔案傾印結果，下為檔案格式剖析。
(請注意：6.12.2 節對以上格式有進一步優化措施，以滿足更複雜情況下的需求)

CObList::Serialize()

list 的檔案讀寫動作比較複雜：首先讀寫 list 的元素個數，然後一一讀寫每個元素：包括 object 內容及其相應的 class 資訊。讀取期間會發生 object 動態生成。CObList 內的每個元素可以是任意的 CObject-derived class object。

```

void CObList::Serialize(CArchive& ar)
{
    CObject::Serialize(ar);          // 首先呼叫 base's serialize()

    if (ar.IsStoring())

```

```

{
    ar.WriteCount(m_nCount);    // 寫入元素個數
    for (CNode* pNode = m_pNodeHead;
         pNode != NULL;
         pNode = pNode->pNext)
    {
        ar << pNode->data;      // 寫入每個元素內容
        // 這裡不能使用 Serialize(); 因為如果這麼做，讀取時也要配合，
        // 而彼時不知道節點型別為何，如何知道呼叫哪個 Serialize() 呢？
    }
}
else
{
    DWORD nNewCount = ar.ReadCount(); // 讀出元素個數
    CObject* newData; // 也可以是 const CObject* newData;
    while (nNewCount--)
    {
        ar >> newData;          // 讀出每個元素內容
        // 如果右端是 const CObject*,
        // 就喚起 operator>>(CArchive&, const CObject*&);
        // 如果右端是 CObject*,
        // 就喚起 operator>>(CArchive&, CObject*&);

        AddTail(newData);        // 加到串列尾端
        // AddTail 接受的是 const CObject*
    }
}
}

```

我要告訴你一個事實，並看著你的頭上升起一朵疑雲。這個事實是：CArchive 支援 DWORD 檔案讀寫（詳 6.5.3 節）。你頭上的疑雲則是：為什麼上述程式碼讀寫元素個數時，不直接這麼做：

```

ar << m_nCount; // 寫入
ar >> nNewCount; // 讀出

```

卻要寫成這樣呢：

```

ar.WriteCount(m_nCount); // 寫入元素個數
nNewCount = ar.ReadCount(); // 讀出元素個數

```

原因是 WriteCount() 和 ReadCount() 會判斷它所接受的參數（一個 DWORD）是否真有 DWORD 那麼大。如果小到只是一個 WORD，它們就只讀寫一個 WORD。真是無所不用其極地為客戶著想呀。

下面是 CObList 的檔案讀寫測試動作，其中所使用的 CShape 階層體系，詳見下

一小節，都是 CObject-derived classes，都支援 Serialization。這份測試獲得圖 6-10 的輸出。

```
COBList myList; // 設計為 CMyDocument 的一個 data member。
...
CMyDocument* pDoc = GetDocument(); // 此動作於 6.9 節介紹，用來
// 取得一個指標，指向 CMyDocument object
CShape* pShape[8]; // 關於 CShape 階層體系，見 6.2.6 節。
pShape[0] = new CEllipse(3.0, 3.0, 7.0, 21.0);
pShape[1] = new CCircle(5.0, 5.0, 7.0);
pShape[2] = new CTriangle(0.0, 0.0, 1.0, 0.0, 0.0, 1.0);
pShape[3] = new CRect(5.6, 6.8, 3.0, 9.0);
pShape[4] = new CSquare(3.2, 4.3, 6.0);
pShape[5] = new CStroke;
CStroke* pStroke = dynamic_cast<CStroke*>(pShape[5]);
assert(pStroke);
pStroke->m_DArray.Add(1);
pStroke->m_DArray.Add(2);
pStroke->m_DArray.Add(3);
pStroke->m_DArray.Add(4);
pStroke->m_DArray.Add(5);
pStroke->m_DArray.Add(6);
pStroke->m_DArray.Add(7);
pShape[6] = new CTriangle(0.0, 0.0, 3.5, 0.0, 0.0, 5.3);
pShape[7] = new CRect(9.9, 9.9, 4.8, 9.8);

for (int i=0; i<8; i++)
    pDoc->myList.AddTail(pShape[i]);

myList.Serialize(ar); // ar 是一個 CArchive object，6.5.3 節介紹。
// 這樣便將整個 list 寫入檔案。
```

```
Turbo Dump Version 3.1 Copyright (c) 1988, 1992 Borland International
Display of File JJ.DAT

000000: 08 00 00 00 08 00 43 45 6C 6C 69 70 73 65 00 00 .....CEllipse..
000010: 40 40 00 00 40 40 00 00 E0 40 00 00 A8 41 00 00 @...@...@...A..
000020: 07 00 43 43 69 72 63 6C 65 00 00 A0 40 00 00 A0 ..CCircle...@...
000030: 40 00 00 E0 40 00 00 09 00 43 54 72 69 61 6E 67 @...@....CTriang
000040: 6C 65 00 00 00 00 00 00 00 00 00 00 80 3F 00 00 le.....?..
000050: 00 00 00 00 00 00 00 00 80 3F 00 00 05 00 43 52 .....?....CR
000060: 65 63 74 33 33 B3 40 9A 99 D9 40 00 00 40 40 00 ect33...@...@..
000070: 00 10 41 00 00 07 00 43 53 71 75 61 72 65 CD CC ..A....CSquare..
000080: 4C 40 9A 99 89 40 00 00 C0 40 00 00 07 00 43 53 L@...@...@....CS
000090: 74 72 6F 6B 65 07 00 01 00 00 00 02 00 00 00 03 troke.....
0000A0: 00 00 00 04 00 00 00 05 00 00 00 06 00 00 00 07 .....
0000B0: 00 00 00 00 00 09 00 43 54 72 69 61 6E 67 6C 65 .....CTriangle
```

```

0000C0: 00 00 00 00 00 00 00 00 00 00 60 40 00 00 00 00 .....~@....
0000D0: 00 00 00 00 9A 99 A9 40 00 00 05 00 43 52 65 63 .....@....CRec
0000E0: 74 66 66 1E 41 66 66 1E 41 9A 99 99 40 CD CC 1C tff.Aff.A...@...
0000F0: 41 00 00 00 00 .....A.....

```

記憶體內容 (little endian, 逆置)	實值	意義
08 00	8	元素個數
00 00	0	版本號碼
08 00	8	class 名稱長度
43 45 6C 6C 69 70 73 65	CEllipse	class 名稱
00 00 40 40 00 00 40 40 00 00 E0 40		元素內容
00 00 A8 41		
07 00	7	class 名稱長度
43 43 69 72 63 6C 65	CCircle	class 名稱
00 00 A0 40 00 00 A0 40 00 00 E0 40		元素內容
00 00	0	版本號碼
09 00	9	class 名稱長度
43 54 72 69 61 6E 67 6C 65	CTriangle	class 名稱
00 00 00 00 00 00 00 00 00 00 80 3F		元素內容
00 00 00 00 00 00 00 00 00 00 80 3F		
00 00	0	版本號碼
05 00	5	class 名稱長度
43 52 65 63 74	CRect	class 名稱
33 33 B3 40 9A 99 D9 40 00 00 40 40		元素內容
00 00 10 41		
00 00	0	版本號碼
07 00	7	class 名稱長度
43 53 71 75 61 72 65	CSquare	class 名稱
CD CC 4C 40 9A 99 89 40 00 00 C0 40		元素內容
00 00	0	版本號碼
07 00	7	class 名稱長度
43 53 74 72 6F 6B 65	CStroke	class 名稱
07 00	7	陣列元素個數
01 00 00 00 02 00 00 00 03 00 00 00	1, 2, 3, 4, 5,	陣列所有元素
04 00 00 00 05 00 00 00 06 00 00 00	6, 7	內容
07 00 00 00		
00 00	0	版本號碼
09 00	9	class 名稱長度
43 54 72 69 61 6E 67 6C 65	CTriangle	class 名稱
00 00 00 00 00 00 00 00 00 00 60 40		元素內容

00 00 00 00 00 00 00 00 9A 99 A9 40		
00 00	0	版本號碼
05 00	5	class 名稱長度
43 52 65 63 74	CRect	class 名稱
66 66 1E 41 66 66 1E 41 9A 99 99 40		元素內容
CD CC 1C 41		

圖 6-10 CObList 檔案讀寫測試結果。上為檔案傾印結果，下為檔案格式剖析。
（請注意：6.12.2 節對以上格式有進一步優化措施，以滿足更複雜情況下的需求）

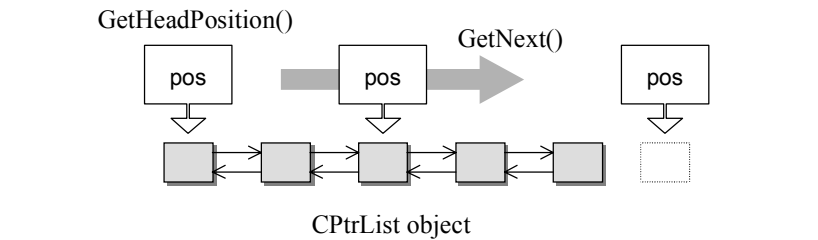
你也許會奇怪，圖 6-9 的傾印內容中有 CWordArray 名稱，圖 6-10 為什麼沒有了呢？這是因為前者為物件指標，採用 operator<< 進行寫入動作，後者卻是一個內嵌物件，採用 Serialize() 進行寫入動作。至於為什麼兩者導致不同的結果，什麼時候使用哪一種動作，詳見 6.5.2 節和 6.5.4 節。

MFC(Lite) 的第三個容器類別是 CPtrList，它並不支援檔案讀寫，主要用於 MFC(Lite) 內部維護。例如 CDocManager, CMultiDocument, CDocument（它們稍後將陸續出場）之中都有一個 CPtrList object，做為資料管理之用。

CPtrList 的運用方式頗有意思。下面是其標準使用形式：

```
CPtrList m_templateList;
...
int nCount = 0;
POSITION pos = m_templateList.GetHeadPosition();
while (pos != NULL)
{
    CDocTemplate* pTemplate = (CDocTemplate*)m_templateList.GetNext(pos);
    ...
}
```

在這裡，pos 代表所謂的「位置指示器」，又稱為 iterator（迭代器），意義如下：



其中 POSITION 定義如下：

```
struct __POSITION { };
typedef __POSITION* POSITION;
```

運用這種觀念與手法，任何一個複雜容器（資料結構）不必曝露太多細節，只需提供 `GetHeadPosition()` 和 `GetNext()` 介面，使用者就可以循序走訪容器中的每一個元素。這種手法在《Design Patterns》一書中被稱為 **Iterator**，定義如下：



Iterator : Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

提供一種依次存取「聚合物件內各元素」的方法，並且不曝露聚合物的內部細節。

`CPtrList` 的設計比 `CObList` 精巧複雜許多，主要是因為它扛起 `MFC(Lite)` 內部維護管理之責，任務繁重，對空間效率和時間效率都必須有高標準。`CPtrList` 提供垃圾回收機制（garbage collection），相當於一個小型 pool 機制。我將在 6.12.1 節詳細說明其實作手法。

6.2.6 形狀類別：CShape 及其子類別

上一節測試 `CObList` 的檔案讀寫功能時，我使用了一些形狀類別。這些類別全都繼承自 `CObject`。除了 `CShape`（一個抽象類別）之外，它們都選用第三層基礎建設 `Serialization`。下面只摘錄此處所需的部分宣告（其餘詳見書附源碼）：

```
class CShape : public CObject
{
    DECLARE_DYNCREATE(CShape)
public:
    virtual void display() { }
    virtual void Serialize(CArchive&) { };
};

class CTriangle : public CShape
{
    DECLARE_SERIAL(CTriangle)
public:
    virtual void display();
    virtual void Serialize(CArchive&);
    CTriangle(...);
protected:
    float x1, y1, x2, y2, x3, y3;
};
```

```

class CStroke : public CShape
{
    DECLARE_SERIAL(CStroke)
public:
    virtual void display();
    virtual void Serialize(CArchive&);

public:
    CDWordArray m_DArray;    // hold elements by CDWordArray
};

```

這些「形狀」類別都提供 `display()` 函式以表現自己（在文字模式下將列印出各自形狀的資料成員如圓心、半徑、邊長、角點等等），並提供 `Serialize()` 以供檔案讀寫之用。以下是某兩個 `Serialize()` 函式定義：

```

void CTriangle::Serialize(CArchive& ar)
{
    CObject::Serialize(ar);

    if (ar.IsStoring())
        ar << x1 << y1 << x2 << y2 << x3 << y3;
    else
        ar >> x1 >> y1 >> x2 >> y2 >> x3 >> y3;
}

void CStroke::Serialize(CArchive& ar)
{
    CObject::Serialize(ar);

    if (ar.IsStoring())
        m_DArray.Serialize(ar); // CDWordArray 相關資訊將因此不會被寫入
    else
        m_DArray.Serialize(ar); // 配合上面的動作。
}

```

請注意，這裡對於 `m_DArray` 的檔案讀寫動作並不採用 `operator<<` 或 `operator>>`，而是直接呼叫其 `Serialize()`，這是因為 `m_DArray` 是個內嵌物件，不是一個物件指標。詳見 6.5.2 節和 6.5.4 節。

我們可以把這個 `CShape` 階層體系視為一套第三方開發模組（程式庫），以此測試 MFCLite 和其應用程式之間的關係，這就更真實地模擬了現實世界的可能性。

6.3 執行期型別辨識，第一層巨集 (x_DYNAMIC)

第一層巨集支援執行期型別辨識 (RTTI)，源碼如下：

```
// 以下巨集用來宣告第一層基礎建設
//-----
#define DECLARE_DYNAMIC(class_name) \
public: \
    static CRuntimeClass class##class_name; \
    virtual CRuntimeClass* GetRuntimeClass() const;

// 以下巨集用來設定 CRuntimeClass 欄位，並串接「類別型錄網」
//-----
#define _IMPLEMENT_RUNTIMECLASS(class_name, base_class_name, \
                                wSchema, pfnNew) \
    static char _lpsz##class_name[] = #class_name; \
    CRuntimeClass class_name::class##class_name = { \
        _lpsz##class_name, \
        sizeof(class_name), \
        wSchema, \
        pfnNew, \
        RUNTIME_CLASS(base_class_name), \
        NULL }; \
    static AFX_CLASSINIT _init_##class_name( \
        &class_name::class##class_name); \
    CRuntimeClass* class_name::GetRuntimeClass() const \
    { return &class_name::class##class_name; }

// 以下巨集用來實作第一層基礎建設
//-----
#define IMPLEMENT_DYNAMIC(class_name, base_class_name) \
    _IMPLEMENT_RUNTIMECLASS(class_name, base_class_name, \
                            0xFFFF, NULL)
```

其中所使用的其他巨集或結構，定義如下：

```
#define RUNTIME_CLASS(class_name) \
    (&class_name::class##class_name)

struct AFX_CLASSINIT
{ AFX_CLASSINIT(CRuntimeClass* pNewClass); }; // ctor
// 稍後討論
```

於是，以下動作：

```
class CView : public CWnd
{
    DECLARE_DYNAMIC(CView);
};

IMPLEMENT_DYNAMIC(CView, CWnd);
```

就會被編譯器的前處理器 (pre-processor) 展開如下，正是我們所要的格式：

```
class CView : public CWnd
{
public:
    static CRuntimeClass classCView;
    virtual CRuntimeClass* GetRuntimeClass() const;
};

static char _lpszCView[] = "CView";
CRuntimeClass CView::classCView =
    _lpszCView,
    sizeof(CView),
    0xFFFF,
    0,
    (&CWnd::classCWnd),
    0 };

static AFX_CLASSINIT _init_CView(&CView::classCView);
CRuntimeClass* CView::GetRuntimeClass() const
{ return &CView::classCView; }
```

先前我曾說過，整個類別型錄網必須在及早建構完成，因為程式很可能一開始就需要三大基礎服務。及早？多早？比程式進入點 `main()` 更早，夠早了吧 ☺。爲了讓建構動作比 `main()` 更早，我們必須借助 `global object`：它的建構式是唯一早於 `main()` 的動作。爲此才有上述的 `AFX_CLASSINIT`。其建構式設計如下：

```
AFX_CLASSINIT::AFX_CLASSINIT(CRuntimeClass* pNewClass)
// 此建構式負責建立及維繫「類別型錄網」
{
    pNewClass->m_pNextClass = CRuntimeClass::pFirstClass;
    CRuntimeClass::pFirstClass = pNewClass;
}
```

當然，系統之中獨一無二的 `CRuntimeClass::pFirstClass` 必須有個初始設定。

此動作發生於全域範疇（global scope）內；由於是個 static object，所以可視為全域變數般地給予初值：

```
CRuntimeClass* CRuntimeClass::pFirstClass = NULL;
```

圖 6-11 說明「類別型錄網」的維護過程。

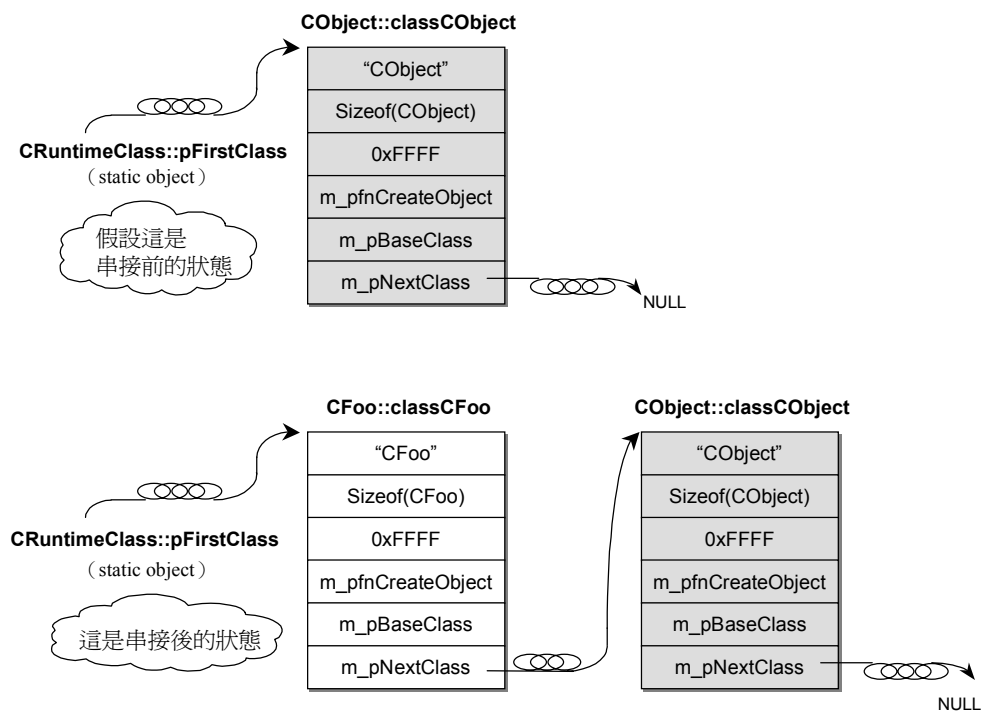


圖 6-11 「類別型錄網」的維護（串接）過程

6.4 動態生成，第二層巨集 (x_DYNCREATE)

第二層巨集支援動態生成 (Dynamic Creation)，源碼如下：

```
#define DECLARE_DYNCREATE(class_name) \
    DECLARE_DYNAMIC(class_name) \
    static CObject* CreateObject();

#define IMPLEMENT_DYNCREATE(class_name, base_class_name) \
    CObject* class_name::CreateObject() \
    { return new class_name; } \
    _IMPLEMENT_RUNTIMECLASS(class_name, base_class_name, \
        0xFFFF, class_name::CreateObject)
```

於是，以下動作：

```
class CMyView : public CView
{
    DECLARE_DYNCREATE(CMyView);
};

IMPLEMENT_DYNCREATE(CMyView, CView);
```

就會被編譯器的前處理器 (pre-processor) 展開如下，正是我們所要的格式：

```
class CMyView : public CView
{
public:
    static CRuntimeClass classCMyView;
    virtual CRuntimeClass* GetRuntimeClass() const;
    static CObject* CreateObject();
};

CObject* CMyView::CreateObject() { return new CMyView; }

static char _lpszCMyView[] = "CMyView";
CRuntimeClass CMyView::classCMyView =
    _lpszCMyView,
    sizeof(CMyView),
    0xFFFF,
    CMyView::CreateObject,
    (&CView::classCView),
    0 };

static AFX_CLASSINIT _init_CMyView(&CMyView::classCMyView);
CRuntimeClass* CMyView::GetRuntimeClass() const
{ return &CMyView::classCMyView; }
```

有趣的是，第二層巨集的實作（`IMPLEMENT_x`），為什麼不像其宣告（`DECLARE_x`）一樣地運用「層層涵蓋」的手法呢？其實意義上的確是層層涵蓋，只因各層給予 `_IMPLEMENT_RUNTIMECLASS` 巨集的最後兩個引數各不相同，不得不採用現在你所看到的變通手法。

現在請看巨集展開後的 `CRuntimeClass` 欄位設定動作，第 4 欄位的型別是：

```
CObject* (* m_pfnCreateObject)();          // (A)
```

並被設值為 `CMyView::CreateObject` — 函式名稱就是函式位址，這種寫法等同於 `&CMyView::CreateObject`。但是從語法上看起來 `m_pfnCreateObject` 明明是個 `pointer to function`，如何能強迫它接受一個 `member function` 的位址呢？

強扭的瓜不甜，這裡根本不存在強扭的事實。上述 (A) 式的確是個道地的 `pointer to non-member function`；它之所以能夠接受 `&CMyView::CreateObject`，關鍵在於後者其實地位相當於一個 `pointer to non-member function`（因為它是 `static` 函式，見 1.x 節），只不過貌似 `pointer to member function` 罷了。

6.4.1 動態生成有多重要

動態生成技術適用於檔案讀取時產生物件。讀取檔案內容時，我們可以這麼寫：

```
char className[50] = GetClassName(); // 假設 GetClassName() 可用。
...      // 將上述字串拿來和「類別型錄網」中的所有 class 名稱比對，
          // 獲得一個相應的 CRuntimeClass object。
          // 假設以 CRuntimeClass* pClassRef 指向它。
CObject* pObj = pClassRef->CreateObject();
```

當然啦，首先必須保證我們所讀取的 `class` 名稱，是一個 `CObject-derived class`，並且曾經採用第二層基礎建設。如果當初將 `object` 寫入檔案時是透過第三層設施 `Serialization` 完成，那就沒問題了 — 第三層設施涵蓋第二層設施，因此該 `class` 一定擁有動態生成能力。上述整個行為過程，在 6.9 節的 MVC 模型中有詳細說明。

MFC(Lite) 內部也大量運用動態生成技術，回應使用者撰寫程式時之彈性設定（設定使用某些 `classes`），進而動態調整相互合作的 `classes` 種類。是的，MFC(Lite) 只供應演算法骨幹，保留了細節部分讓使用者彈性指定，所以它必須根據使用者的指示，動態生成那些被指定的 `classes`。6.9 節的 MVC 模型對此有更多介紹。

6.5 次第檔案讀寫，第三層巨集 (x_SERIAL)

第三層巨集支援次第檔案讀寫 (Serialization)，源碼如下：

```
#define DECLARE_SERIAL(class_name) \
    DECLARE_DYNCREATE(class_name) \
    friend CArchive& operator>>(CArchive& ar, const class_name*& pObj); \
    friend CArchive& operator>>(CArchive& ar, class_name*& pObj);

#define IMPLEMENT_SERIAL(class_name, base_class_name, wSchema) \
    CObject* class_name::CreateObject() \
    { return new class_name; } \
    CArchive& operator>>(CArchive& ar, class_name*& pObj) \
    { pObj = (class_name*)ar.ReadObject( \
        RUNTIME_CLASS(class_name)); \
        return ar; } \
    CArchive& operator>>(CArchive& ar, const class_name*& pObj) \
    { pObj = (const class_name*)ar.ReadObject( \
        RUNTIME_CLASS(class_name)); \
        return ar; } \
    _IMPLEMENT_RUNTIMECLASS(class_name, base_class_name, \
        wSchema, class_name::CreateObject)

// 以上的 operator>> to "const" 是 MFCLite 新增，MFC 並沒有這麼做。
// 為的是將資料讀出放進 const container 中。測試碼見 mfclslt2.cpp
```

於是，以下動作：

```
class CObList : public CObject
{
    DECLARE_SERIAL(CObList);
};

IMPLEMENT_SERIAL(CObList, CObject, 1);
```

會被編譯器的前處理器 (pre-processor) 展開如下，正是我們所要的格式：

```
class CObList : public CObject
{
public:
    static CRuntimeClass classCObList;
    virtual CRuntimeClass* GetRuntimeClass() const;
    static CObject* CreateObject();
    friend CArchive& operator>>(CArchive& ar, const CObList*& pObj);
    friend CArchive& operator>>(CArchive& ar, CObList*& pObj);
};
```

```

CObject* CObList::CreateObject() { return new CObList; }

CArchive& operator>>(CArchive& ar, CObList*& pObj)
{ pObj = (CObList*)ar.ReadObject((&CObList::classCObList));
  return ar; }

CArchive& operator>>(CArchive& ar, const CObList*& pObj)
{ pObj = (const CObList*)ar.ReadObject((&CObList::classCObList));
  return ar; }

static char _lpzCObList[] = "CObList";
CRuntimeClass CObList::classCObList =
    _lpzCObList,
    sizeof(CObList),
    1,
    CObList::CreateObject,
    (&CObject::classCObject),
    0 };

static AFX_CLASSINIT _init_COBList(&CObList::classCObList);
CRuntimeClass* CObList::GetRuntimeClass() const
{ return &CObList::classCObList; }

```

第三層巨集的實作（`IMPLEMENT_x`），雖然意義上涵蓋第二層，但因各層給予 `_IMPLEMENT_RUNTIMECLASS` 巨集的最後兩個引數各不相同，所以不能直觀地像其宣告（`DECLARE_x`）那樣，在第三層巨集中直接涵蓋第二層巨集。

6.5.1 物件永續機制的輕量級實現（a lightweight persistence）

只要能將物件狀態儲存起來，不受程式生命的影響，就是所謂「物件永續」。次第讀寫不是唯一的物件永續之道，卻是最直觀的作法，也是 MFC(Lite) 所提供的作法。

為什麼說 **Serialization** 是最直觀的作法呢？這個字眼有「次第、逐一」完成的意思，其設計理念是，要求使用者（應用程式員）配合，將所有資料集中於 `CDocument-derived class` 內，然後由 **Serialization** 機制提供方便的操作（一個 `operator<<` 和一個 `operator>>`），便能夠讓使用者將整個 `CDocument-derived object` 寫入或讀出檔案。寫入或讀出時，循著使用者所覆寫的 `Serialize()` 虛擬函式中指定的次序（而非 `object` 內的成員宣告次序），次第寫入或讀出每個成員。如果成員本身又有成員，則自動引發次級成員的 `operator<<` 或 `operator>>`（當

然該次級成員必須也支援 `Serialization` 才行）。

因此，假設程式的所有資料都安排在 `CMyDocument` 內，如下：

```
class CMyDocument : public CDocument
{
...
private:
    CObList myList;      // 已於 6.2.5 節介紹過，支援 Serialization。
};
```

程式執行過程中產生了許多 `CObject-derived objects`，並被安置於 `myList` 內。現在，欲將所有資料寫入檔案，只要這麼做就行了：

```
void CMyDocument::Serialize(CArchive& ar)    // CArchive 稍後解釋
{
    CObject::Serialize(ar);  // 呼叫根類別的檔案讀寫動作，以備相容於未來

    if (ar.IsStoring()) {
        myList.Serialize(ar);  // persistence, write to file
        // ar << myList;      // 某種情況下甚至可採用這種型式。6.5.2 詳述。
    }
    else {
        myList.Serialize(ar);  // persistence, read from file
        // ar >> myList;      // 某種情況下甚至可採用這種型式。6.5.2 詳述。
    }
}
```

如果安置於 `myList` 內的元素 (`CObject-derived object`) 本身又是個複雜容器，當它被讀寫時，又會引發其 `Serialize()` 函式。舉個例子，假設 `myList` 的第三元素是個 `CDWordArray` (已於 6.2.5 節介紹過，支援 `Serialization`)，那麼讀寫至該元素，便會引發 `CDWordArray::Serialize()`。

這裡因此有了一個「分層負責」的觀念：每一種容器（大至複雜的 `linked-list`，小至簡單的基本型別），都只負責自己的讀寫動作就行了。有了這樣的想法，整個設計脈絡便豁然開朗。剩下的問題是，如何讓程式在終端使用者於圖像視窗介面中做了個按鈕動作，便引發一系列行為（詢問檔名、安全檢查、開啓檔案...），最終進入上述的 `CMyDocument::Serialize()` 呢？再一個問題是，漂亮俐索的 `operator>>` 和 `operator<<` 究竟怎麼做出來的？竟能夠將原始資料連同其類別資訊一併讀寫？

第一個問題將於 6.9 節的 MVC 模型中詳細解答。現在讓我回答第二個問題。

6.5.2 將 operator>> 和 operator<< 多載化

爲了讓應用程式者得以方便地這麼寫：

```
ar << pMyList;    // ar 是個 CArchive object，請暫時想像爲一個檔案。
ar >> pMyList;    // pMyList 是個 CObList*
```

身爲 application framework 設計者的我們必須針對 CArchive 提供兩個多載化運算子。正如前面談到物件永續（persistence）時所說，程式除了讀寫 object 的原始資料，還需讀寫其 class 資訊，因此我們將上述讀寫動作分別導至 ar.ReadObject() 和 ar.WriteObject()，稍後介紹 CArchive 時會有這兩個函式的特寫鏡頭。

舉個例子，面對 COblist，我們應該爲它準備以下兩個全域性的多載化運算子（請暫時不要在意爲何以下註解的編號是跳躍的）：

```
inline CArchive& operator<<(CArchive& ar, const CObList* pObj) // (1)
{ ar.WriteObject(pObj);    return ar; }
```

```
CArchive& operator>>(CArchive& ar, CObList*& pObj) // (4)
{ pObj = (CObList*)ar.ReadObject((&CObList::classCObList));
  return ar; }
```

由於 object 被寫入檔案時所用的 operator<< 動作不因 object 的型別而有異，所以我們可以把這個多載化運算子提昇爲更泛用、更與型別無關的型式如下（注意參數型別）：

```
inline CArchive& operator<<(CArchive& ar, const CObject* pObj) // (1)
{ ar.WriteObject(pObj);    return ar; }
```

這麼一來它就適用於所有的 CObject-derived serializable class。

以上 (4) 式適用於我們已經明確知道「待讀入的是個 CObList object」，正如本節一開始所示範。但如果程式讀取 object 時尚未明確知道它隸屬何種型別（只知道它必然屬於 CObject-derived），就必須使用更泛化的型式：

```
inline CArchive& operator>>(CArchive& ar, CObject*& pObj) // (2)
{ pObj = ar.ReadObject(NULL);
  return ar; }
```

將 `CArchive::ReadObject()` 的引數設為 `NULL`，會造成什麼影響？本小節最後提出了說明。

至此，我們應該為 `CObList` 準備以上三個多載化運算子。如果考慮更週詳一些，加上對應的 `const` 版本，總共是以下五個多載化運算子，其中 (2),(3) 和 (4),(5) 只是 `non-const` 版和 `const` 版之差 (MFCLite 正是如此設計。MFC 則缺乏 `const` 版本，可視為一個誤漏)：

```
inline CArchive& operator<<(CArchive& ar, const CObject* pObj) // (1)
{ ar.WriteObject(pObj); return ar; }
inline CArchive& operator>>(CArchive& ar, CObject*& pObj) // (2)
{ pObj = ar.ReadObject(NULL); return ar; }
inline CArchive& operator>>(CArchive& ar, const CObject*& pObj) // (3)
{ pObj = ar.ReadObject(NULL); return ar; }
```

```
CArchive& operator>>(CArchive& ar, CObList*& pObj) // (4)
{ pObj = (CObList*)ar.ReadObject((&CObList::classCObList));
  return ar; }
CArchive& operator>>(CArchive& ar, const CObList*& pObj) // (5)
{ pObj = (const CObList*)ar.ReadObject((&CObList::classCObList));
  return ar; }
```

前三個運算子通用於任何 `classes`，所以只要在全域範疇 (`global scope`) 內撰寫一份即可。後兩個運算子是 `class` (本例為 `CObList`) 專屬，所以應該設計由第三層巨集產生。請回頭看看 6.5 節的第三層巨集定義，及其展開結果。各運算子的正確運用時機，都將在 6.10 節的完整測試程式中顯現。

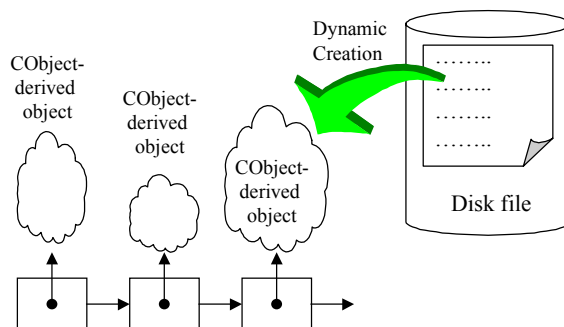
這些運算子中有一些看似詭譎的語法，有必要詳加說明。`operator>>` (2) 式的第二參數型別如下：

```
CObject*& pObj;
```

其意義是：

```
(CObject*)& pObj;
```

也就是說這個參數所接受的是個指標，但以 `by reference` 方式傳遞。為什麼要這樣？因為應用端往往希望在 `CObList` 內放置指標，如此才能達成多型效果。而我們必須以 `by reference` 方式將指標傳遞到這個函式內，才有可能在「從檔案讀出 `object` 資料並動態生成該 `object`」時，改變 `CObList` 內的指標值，使它指向新生 `object`：



至於 `operator>>` (3) 式的第二參數的型別：

```
const CObject*& pObj;
```

其意義是：

```
(const CObject*)& pObj;
```

也就是說這個參數所接受的是個「指向常數物件」的指標，但以 `by reference` 方式傳遞。指標本身可變（於是才能被設定指向新生成的 `object`），但指標所指的 `object` 本身是個常量。

`operator>>` (4),(5) 兩式的第二參數型別，雖然語法同上，運用場合卻有不同。在這裡，讀取對象已確定是個 `CObList` `object`，適用於以下這種情況：

```
class CMyDocument : public CDocument
{
...
private:
    CObList* pMyList;    // 注意，是個指標，不同於前一節（該處是個 object）
};

void CMyDocument::Serialize(CArchive& ar)    // CArchive 稍後解釋
{
    CObject::Serialize(ar);    // 呼叫根類別的檔案讀寫動作，以備相容於未來
    if (ar.IsStoring())
        ar << pMyList;        // 6.5.1 節是呼叫 Serialize()
    else
        ar >> pMyList;        // 6.5.1 節是呼叫 Serialize()
}
```

這肯定引起你的疑惑：究竟何時該使用 `operator>>` 和 `operator<<`，何時該使用 `Serialize()` 呢？請記住一個原則（絕對不能錯亂）：如果面對的是「物件實體」

如 6.5.1 節實例，應該使用 `Serialize()`，如果面對的是「物件指標」如本例，應該使用 `operator<<` 和 `operator>>`。這兩種作法所導致的檔案內容也不相同，稍後剖析檔案格式時我會有更深入的討論。

先前的 `operator>>()` (2)(3) 兩式為什麼要將 `ReadObject()` 的引數設為 `NULL` 呢？因為這麼做代表我們對讀入之 `class` 相關資訊無所預期，詳見 6.5.3 節的 `CArchive::ReadObject()` 和 `CArchive::ReadClass()` 源碼解說。



以下各節(至本章結尾)探討之 MFCLite classes，其 data members 和 member functions 的 UML 表示法，整理於附錄 F，對於全局掌握 MFCLite 的方方面面，乃至於源碼的追蹤，很有幫助。請務必時時參考，按圖索驥。

6.5.3 檔案相關類別：CArchive, CFile

由於 C++ 提供了運算子重載（operator overloading）能力，我們得以納須彌於芥子，將非常複雜的檔案讀寫動作收納於一個 operator<< 和一個 operator>> 運算子內。現在是跳出芥子體會須彌的時候了。

簡介 CFile

CFile 只是將檔案讀寫動作做一層薄薄包裝，其 member functions 一一對應於各個「檔案相關系統呼叫」。爲了實現 MFCLite 的可攜性，我把原本 MFC 所採用的系統呼叫（Win32 APIs）改爲 C 標準函式庫所提供的函式如 fopen(), fread, fwrite()。CFile 內部記錄有檔案的名稱和權柄（handle）。由於 MFC 以 HFILE（file handle）代表檔案的擁有權，爲相容於此，我在 MFCLite 中做了這樣的模擬：

```
typedef FILE* HFILE; // 以 ANCI-C FILE* 模擬 Win32 HFILE。
```

下面是 CFile 的介面，實作細節請見書附源碼。

```
class CFile : public CObject
{
    DECLARE_DYNAMIC(CFile)
public:
    // Flag values
    enum OpenFlags {
        modeRead = 0x0000, // 讀檔模式
        modeWrite = 0x0001, }; // 寫檔模式

    CFile(const char* lpszFileName, UINT nOpenFlags);
    virtual ~CFile();
    virtual BOOL Open(const char* lpszFileName,
                     UINT nOpenFlags); // 開檔
    virtual void Close(); // 關檔
    virtual UINT Read(void* lpBuf, UINT nCount); // 讀檔
    virtual void Write(const void* lpBuf, UINT nCount); // 寫檔

    HFILE m_hFile; // 對應的檔案（HFILE 就是 FILE*）

protected:
    char m_strFileName[40]; // 檔名
};
```

簡介 CArchive

CArchive 是 CFile 的更上層包裝，其價值在於：

- 可藉由它實現緩衝能力 (buffering)，增加檔案讀寫的效率。
- 可藉由這層間接性，將 Serialization 所需的許多煩瑣細節納入其中，免除應用的勞役。
- 將實際的檔案存取層 (CFile) 加以隔離，中介一層 CArchive。這是設計樣式 **Bridge** 的實現 (詳見第 7 章)。

讀寫緩衝 (buffering) 並非我所關注的技術，因此 MFCLite 並未加以模擬。CArchive 對 Serialization 的支援，主要在於 WriteClass(), WriteObject(), ReadClass(), ReadObject() 四個函式，以及眾多的 operator<<, operator>> 運算子。下面是 CArchive 介面，實作細節請見書附源碼。關鍵說明稍後即示。

```
// 以下為簡化版本。複雜 (與 MFC 完全相同) 之版本見 6.12.2 節。
class CArchive
{
public:
    // Flag values
    enum Mode { store = 0, load = 1 };          // 儲存 (寫檔) 或載出 (讀檔)

    CArchive(CFile* pFile, UINT nMode);
    ~CArchive();

    CFile* CArchive::GetFile() const
    { return m_pFile; }
    BOOL CArchive::IsStoring() const             // 儲存 (寫檔) 模式嗎?
    { return (m_nMode & CArchive::load) == 0; }
    BOOL CArchive::IsLoading() const             // 載出 (讀檔) 模式嗎?
    { return (m_nMode & CArchive::load) != 0; }

    UINT Read(void* lpBuf, UINT nMax);           // 讀取
    void Write(const void* lpBuf, UINT nMax);    // 寫入
    void Close();                                // 關閉

    // insertion operations (inlining)。寫出函式全名以利識別。
    CArchive& CArchive::operator<<(int i)
    { return CArchive::operator<<((LONG)i); }
    CArchive& CArchive::operator<<(unsigned u)
    { return CArchive::operator<<((LONG)u); }
    // ... 依此類推，針對各種基本型別而設計之 operator<< 運算子。
    // 詳見源碼檔案。
```

```

// extraction operations (inlining)。寫出函式全名以利識別。
CArchive& CArchive::operator>>(int& i)
{ return CArchive::operator>>((LONG&)i); }
CArchive& CArchive::operator>>(unsigned& u)
{ return CArchive::operator>>((LONG&)u); }
// ... 依此類推，針對各種基本型別而設計之 operator<< 運算子。
// 詳見源碼檔案。

// object read/write
CObject* ReadObject(const CRuntimeClass* pClass);
void WriteObject(const CObject* pObj);

// class read/write
void WriteClass(const CRuntimeClass* pClassRef);
CRuntimeClass* ReadClass(const CRuntimeClass*
                        pClassRefRequested = NULL);

DWORD ReadCount(); // 讀出一個 DWORD (也許是 class 名稱的長度)
void WriteCount(DWORD dwCount); // 寫入一個 DWORD

protected:
    BOOL m_nMode; // 開啓模式
    CFile* m_pFile; // 對應的 CFile object
};

```

除此之外就是各個 `Serializable classes` 針對 `CArchive` 而重載的 `operator>>`, `operator<<` 運算子。這些運算子由巨集自動產生，6.5.2 節才剛說過。

當終端用戶按下檔案相關命令如 [File/Open] 或 [File/Save]，程式該有的相應動作是詢問檔名、檢查檔案(這些將在 6.9 節的 MVC 模型中引入)，然後產生一個 `CFile` object，再據此產生一個 `CArchive` object。然後，程式就可以對這個 `CArchive` object 做檔案讀寫動作。

將 object 寫入檔案

將 object 原始內容 (raw data) 寫入檔案之前，必須先將其 class 相關資訊寫入 (以備讀出時動態生成之用)。這兩個寫入動作 (寫入 object 相關資訊和 object 本身內容) 必須在設計上區分開來。先前設計 `operator<<` 時我們讓它呼叫 `CArchive::WriteObject()`，後者又先以 `CArchive::WriteClass()` 將 class 相關資訊寫入檔案：

```
// 以下為簡化版本。複雜（與 MFC 完全相同）之版本見 6.12.2 節。
void CArchive::WriteObject(const CObject* pObj)
{
    // 此函式被 ar << xxx; 呼叫。不論傳入什麼型別，這裡都以 CObject* 接受之，
    // 只要再使用 GetRuntimeClass()，即可取出 object 對應的 CRuntimeClass，
    // 然後便可精確做事。

    // 此處應檢查是否 (pObj==null)，因為 null ptr 應該有特別的儲存方式。略。

    // 首先寫入 object 所屬的 class 的相關資訊（根據其 CRuntimeClass）
    CRuntimeClass* pClassRef = pObj->GetRuntimeClass();
    WriteClass(pClassRef);

    // 令 object 自行負責寫入檔案
    ((CObject*)pObj)->Serialize(*this);
}
```

所謂「class 相關資訊」不外乎 class 名稱（這裡採用 "LString" 概念，因此必須記錄名稱長度），其寫入動作不因任何 class 而異，因此「class 相關資訊」的寫入設計可提昇至 CRuntimeClass 中：

```
// 以下為簡化版本。複雜（與 MFC 完全相同）之版本見 6.12.2 節。
void CArchive::WriteClass(const CRuntimeClass* pClassRef)
{
    // 這裡應檢查 pClassRef 是否為 serializable (schema no. 不得為 0xFFFF)
    // 略。

    // 這裡應判斷是否為先前出現過的 class（詳見 6.5.4 節說明）。略。

    pClassRef->Store(*this); // 寫入 class 相關資訊
}
```

```
void CRuntimeClass::Store(CArchive& ar) const
{
    // 將 class 相關資訊寫入檔案
    WORD wLen = (WORD)::strlen(m_lpszClassName); // class 名稱長度
    ar << (WORD)m_wSchema << wLen; // 寫入 schema no. 及 class 名稱長度
    ar.Write(m_lpszClassName, wLen*sizeof(char)); // 寫入名稱
}
```

圖 6-12 表現出這些動作的概觀。

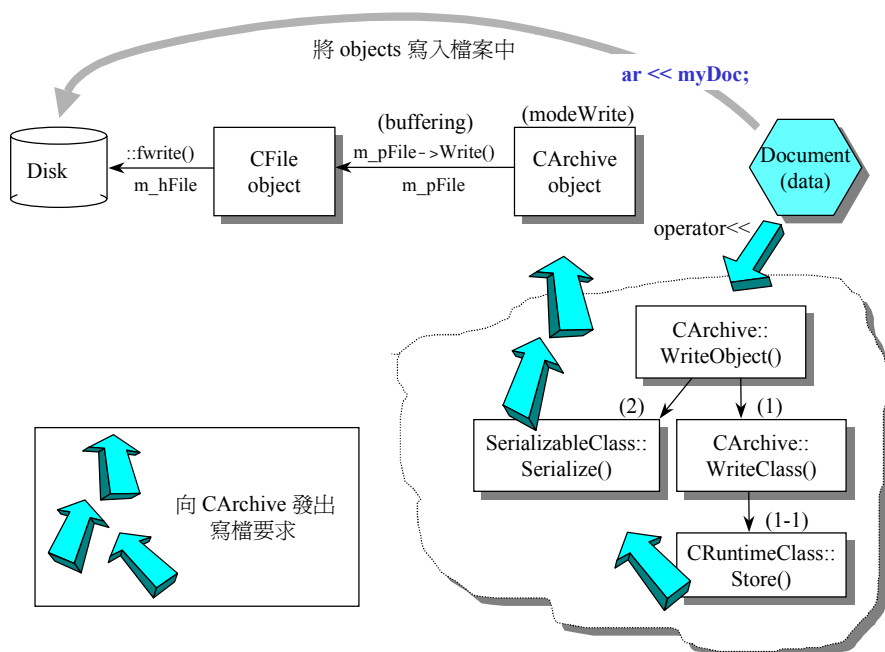


圖 6-12 將 object 寫入檔案，所引發的程式動作

將 object 自檔案讀出

相應於 object 的寫入動作，讀取時也必須先將 object 的「class 相關資訊」讀出，動態生成一個對應的 object，然後才令該 object 讀取實際內容。這兩個讀取動作必須在設計上區分開來才比較理想。先前設計 operator>> 時我們曾讓它呼叫 CArchive::ReadObject()，現在，後者必須先以 CArchive::ReadClass() 將 class 相關資訊從檔案中讀出：

```
// 以下為簡化版本。複雜（與 MFC 完全相同）之版本見 6.12.2 節。
CObject* CArchive::ReadObject(const CRuntimeClass*
                             pClassRefRequested)
{
    // 此函式被 "ar >> xxx;" 喚起。呼叫端先根據 xxx 取得其 CRuntimeClass，
    // 然後傳入這裡。函式內根據這個引數讀取相關資訊，看看是否能夠成功。
    // 讀取的作用並非為了獲得，而是為了檢驗。
```

```

// 首先讀出 object 所屬的 class 的相關資訊，並以 CRuntimeClass 呈現。
CRuntimeClass* pClassRef = ReadClass(pClassRefRequested);

// 驗證 class 相關資訊是否被正確讀出
CObject* pObj;
if (pClassRef == NULL) // 失敗
{
    TRACE0("bad class\n");
}
else /* gcc291 的大臭蟲：else 之後不能加單行註解 (//)。真要命 */
{
    // 根據 class 相關資訊 (亦即 class 名稱)，動態生成一個 object
    pObj = pClassRef->CreateObject();
    if (pObj == NULL)
        TRACE0("Dynamic Creation Fail\n");

    // 令 object 自行負責從檔案讀出其自身內容
    pObj->Serialize(*this);
}
return pObj;
}

```

所謂 class 相關資訊不外乎 class 名稱 (這裡採用 "LString" 概念，因此必須記錄名稱長度)，其讀取動作不因任何 class 而有異，因此「class 相關資訊」的讀取設計可提昇至 CRuntimeClass 中：

```

// 以下為簡化版本。複雜 (與 MFC 完全相同) 之版本見 6.12.2 節。
CRuntimeClass* CArchive::ReadClass(const CRuntimeClass*
                                   pClassRefRequested)
{
    assert(IsLoading()); // 確為讀出模式

    if (pClassRefRequested != NULL &&
        pClassRefRequested->m_wSchema == 0xFFFF) // non-serializable
        // 注意，如果寫入時採用 serialization 機制，版本號碼絕不會是 -1
    {
        TRACE0("Warning: Cannot call ReadClass/ReadObject for %hs.\n",
              pClassRefRequested->m_lpszClassName);
    }

    // 這裡可以加上 cache 設計，以提昇速度和空間上的效率。為求簡化，略。
    // 見 6.5.4 節說明。

    // 如果 pClassRefRequested == NULL，不就到這裡來了嗎？好像不妥...
    // 並無不妥，NULL 表示對讀入之 class 相關資訊無所預期。
}

```

```

CRuntimeClass* pClassRef;
UINT nSchema;

if ((pClassRef = CRuntimeClass::Load(*this, &nSchema)) == NULL)
    TRACE0("bad class\n");
if (pClassRef->m_wSchema != nSchema) { // 檢驗版本號碼
    TRACE0("bad schema: %s", pClassRef->m_lpszClassName);
    abort(); // 真實世界中不應這麼殘酷
}
return pClassRef;
}

```

```

CRuntimeClass* CRuntimeClass::Load(CArchive& ar, UINT* pwSchemaNum)
{
    // 讀出一份 class 相關資訊。注意，此為 static 函式，為什麼？
    // 為的是在尚未有 CRuntimeClass object 時，就能夠呼叫它，
    // 呼叫它正是為了獲得一個 CRuntimeClass*。

    WORD wLen;
    WORD wSchemaNum;
    char szClassName[64];
    CRuntimeClass* pClass;

    ar >> wSchemaNum; // 讀出版本號碼
    *pwSchemaNum = wSchemaNum; // 設定好，準備傳回
    ar >> wLen; // 讀出長度
    ar.Read(szClassName, wLen*sizeof(char)); // 讀出 class 名稱
    szClassName[wLen] = '\0'; // 設定正確的字串型式

    // 將上述名稱拿來和類別型錄網中的 class 名稱相比，如果吻合，
    // 就傳回該 CRuntimeClass。如無吻合者，表示這個 class 缺乏
    // 動態生成的能力。
    for (pClass = pFirstClass;
         pClass != NULL;
         pClass = pClass->m_pNextClass)
    {
        if (::strcmp(szClassName, pClass->m_lpszClassName) == 0)
            return pClass;
    }

    TRACE0("Error: Class not found: %s \n", szClassName);
    return NULL; // 沒找到就傳回 NULL
}

```

這些動作可以圖 6-13 表現出來。

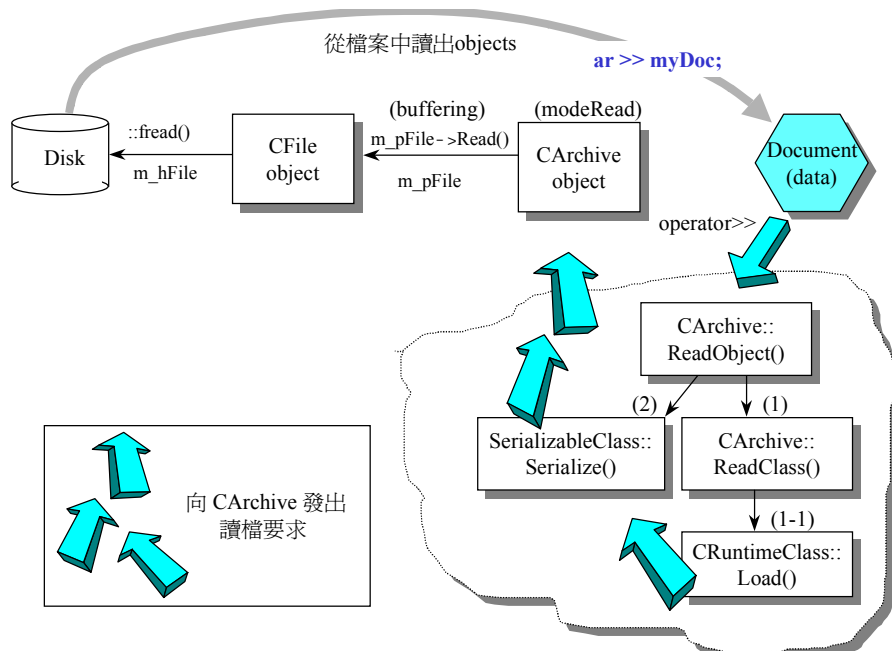


圖 6-13 自檔案讀出 object，所引發的程式動作

6.5.4 Serialization 檔案格式深入探討

綜合以上討論，我們獲得以下結論，形成 MFC(Lite) Serialization 的一般通則：

- 如果待寫之物是個 **pointer to object**，寫入時應該使用 `operator<<`，這會引發 `CArchive::WriteObject()`，進而先引發 `CArchive::WriteClass()` 將「class 相關資訊」寫入，然後才引發 `class::Serialize()` 將 **object** 原始內容 (raw data) 寫入。
- 上述動作之所以需要先寫入「class 相關資訊」，為的是讀出時可用於動態生成。畢竟，待寫之物如果是個 **pointer to object**，讀出時承接的（十之八九）也是個 **pointer to object**，那就必須先生成一個 **object**，才有記憶體空間可供置放資料。
- 如果待寫之物是個 **object**，寫入時應該直接呼叫其 `Serialize()` 函式，這會導致該 **object** 內容被寫入檔案，不含其 class 相關資料。
- 上述情況之所以不需要寫入其 class 相關資訊，是因為既然待寫之物為一個

object，讀出時承接的（十之八九）也會是一個 object，此時已有 object 空間（記憶體）可供置放資料，不必動態生成。

- 上述第一點提過，`operator<<` 引發 `CArchive::WriteObject()`，後者又分別引發 `CArchive::WriteClass()` 和 `class::Serialize()`（如圖 6-12）。同樣道理，`operator>>` 引發 `CArchive::ReadObject()`，後者又分別引發 `CArchive::ReadClass()` 和 `class::Serialize()`（如圖 6-13）。
- 所謂「class 相關資訊」，基本上只需涵蓋版本號碼（schema no.）、class 名稱長度、class 名稱，就足夠了。但面對複雜情況，就有許多優化設計，詳見 6.12.2 節。
- object 的實際內容以什麼形式寫入（或讀出）檔案，完全視 `Serialize()` 如何設計而定。如果 object 是個容器，那麼必須先寫入容器大小（元素個數），然後才是每個元素的內容。
- 綜合以上，任何 object 欲被寫入檔案，設計上的唯一思考就是，如何才能正確讀出資料並還原 object 的原本狀態 — 這才符合「物件永續」的意義。
- 為了盡量節省空間，`CArchive` 特別針對 `DWORD`（通常被用來做為計數器）的讀寫動作設計了 `WriteCount()` 和 `ReadCount()`，兩者都會判斷 `DWORD` 的實際大小，一旦發現數值小到以 `WORD` 表示即足夠時，就絕不會以一個 `DWORD` 來表示。
- 假設一份文件內含 1,000,000 個相同型別的 object，class 名稱長度為 7。如果執行「物件永續（persistence）」動作時，每一個 object 的 class 名稱都被寫入檔案，就會浪費近乎 7,000,000 bytes。雖然硬碟很便宜，但是網絡頻寬很寶貴，網絡傳輸速度將會因為這些贅餘資料而受到極大的衝擊。再者，資料量大，讀取時間也就比較長。為了節省這些不必要的空間和時間耗費，良好的設計應該加上 cache（快取裝置）。此外，如果兩個 pointers 相同（亦即指向同一個 object），當它們被寫入檔案再被讀出，應該還是相同（還是指向同一個 object）。本節的簡化版本無法解決上述這些問題。這些技術牽連廣泛而複雜，將在 6.12.2 節詳述並實現。



以下各節(至本章結尾)探討之 MFCLite classes，其 data members 和 member functions 的 UML 表示法，整理於附錄 F，對於全局掌握 MFCLite 的方方面面，乃至於源碼的追蹤，很有幫助。務請時時參考，按圖索驥。

6.6 應用框架（application framework）之浮現

好啦，終於完成三大基礎建設，讓我們換個角度看看 application framework。

正如我在本章起始時說過，使用 application framework 和使用 toolkits 時吾人所扮演的角色完全相反。當你使用 toolkit，你負責撰寫程式主體，並呼叫別人寫好的程式碼；當你使用 application framework，你改而使用別人架構好的程式主體，並負責撰寫它（程式主體）所呼叫的程式碼。

現在就讓我們來看看 application framework 如何架構我們的應用程式主體。

使用 application framework，我們沒有權力（也沒有必要）問為什麼程式架構長成這樣或那樣，但我們要問為什麼這樣或那樣配合就可以達到這樣或那樣的效果。面對一個全面決定我們的 application 行為模式的東西，我們必須清楚其肌理、了解其脈絡，絕不能老是照著葫蘆畫瓢——別人是巫師，你卻成了傀儡，自己都不知道自己在做什麼。

6.6.1 框架模組與應用模組之乾淨切割：謝資原檔

所謂「乾淨切割」就是，application framework 不能受 application 之任何發展的影響。假設 application 定義了一個表單命令（menu item），application framework 的命令服務機制雖然必須服務它，其源碼卻不該重新編譯，更不該有任何改變。別忘了，對應用程式員而言，application framework 是一個購入的產品，雖然源碼在手，卻只能拆解不該修改，否則你就得冒極大的風險。

圖 6-14 上方是 MFC 所切割的楚河漢界。其中為了 precompile-headers²的因素，出現 stdafx.h 和 stdafx.cpp 這樣的檔案。這不是我的模擬重點，因此 MFCLite 所切割的楚河漢界如圖 6-14 下方，就簡化了些。

² 所謂 precompile-headers 是指，任何表頭檔如果保持不變（例如 MFC 所提供者），就不該在程式每次重新建造（build）時重新編譯一遍，畢竟 C++ 表頭檔不像 C 那樣只有 #define 或 typedef 或 macros 而已，會耗用不少編譯時間。為此，絕大多數編譯器都有一套能力，將無變化的表頭檔編譯為 .obj 檔，於下次重新建造（build）時直接使用。

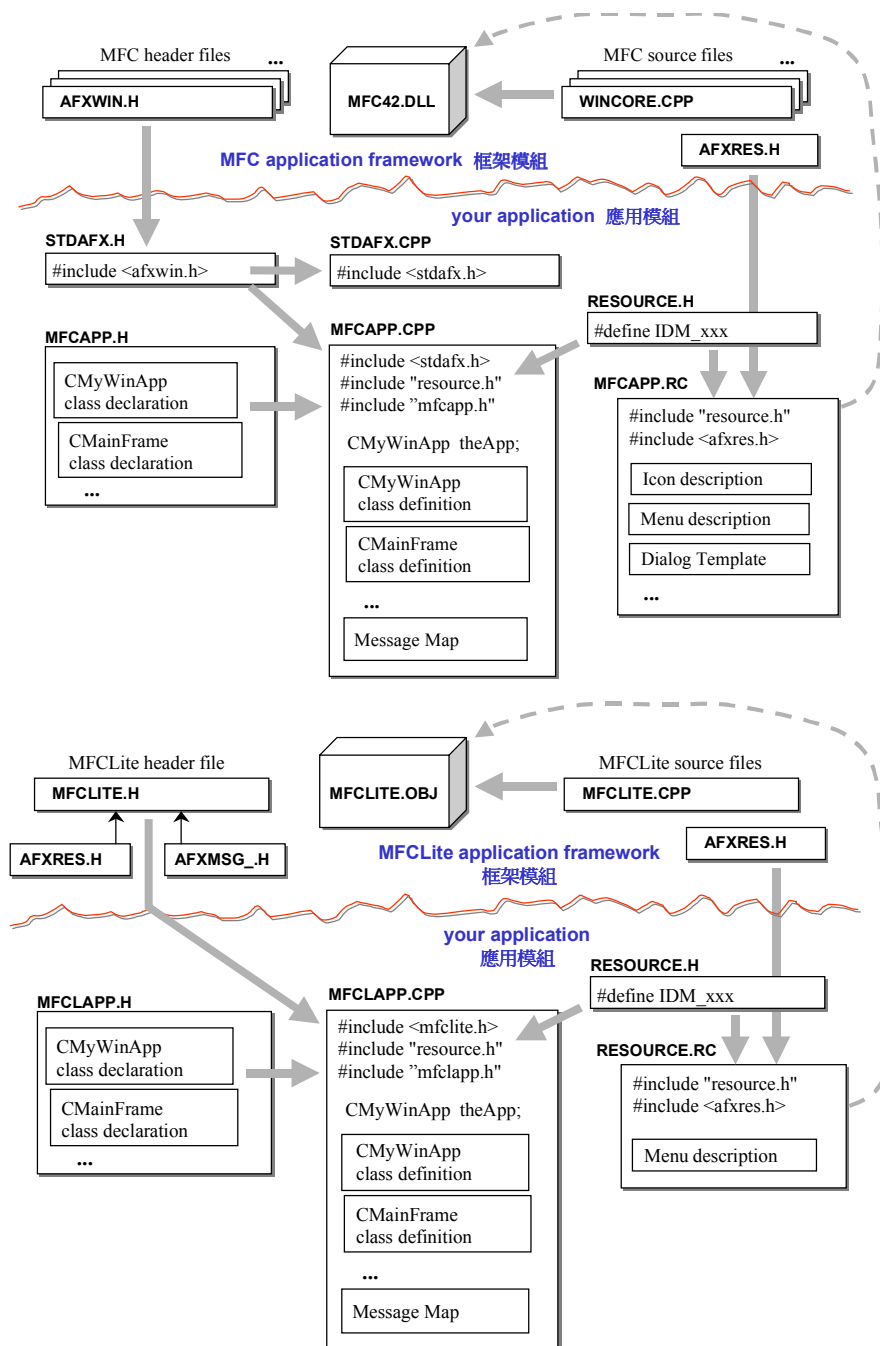


圖 6-14 MFC (上) 和 MFCLite (下) 所切割的楚河漢界。圖中虛線表示，
.rc 檔被資源編譯器解讀後轉譯為 .res 檔，為 MFC(Lite) 所用。

割斷耦合

應用 **Template Method** 設計手法（6.1.3 節），application framework 可輕易割斷與 application 之間的任何耦合（coupling）關係。但是，有一個極細微的地方卻帶來不小的困擾。

圖 6-14 右側出現 **resource** 這個字眼。在 MFC（乃至於一般 Win32 程式開發）中，所謂 resource 涵蓋了 menu, dialog, stringtable, icon, bitmap...等諸多項目，定義於 .rc 檔中（MFC Lite 只模擬其中的 menu 功能）。下面是應用模組可能定義的 resource.rc 的部分內容：

```
#include "resource.h"
#include "afxres.h"

IDR_GRAPHTYPE MENU PRELOAD DISCARDABLE
BEGIN
    POPUP "&File"
    BEGIN
        MENUITEM "&New\tCtrl+N",          ID_FILE_NEW
        MENUITEM "&Open...\tCtrl+O",      ID_FILE_OPEN
        MENUITEM "&Save\tCtrl+S",          ID_FILE_SAVE
        MENUITEM "Save &As...",           ID_FILE_SAVE_AS
        MENUITEM SEPARATOR
        MENUITEM "e&Xit",                  ID_APP_EXIT
    END

    POPUP "&Edit"
    BEGIN
        MENUITEM "clea&R All",             ID_EDIT_CLEAR_ALL
    END
    ...
END
```

簡單地說，一個 POPUP 代表一個下拉式表單，其內每個 MENUITEM 都代表一個表單命令，其後可指定文字描述、快捷鍵、命令識別碼（ID_x）。這些識別碼被集中定義於 resource.h。更詳細的說明請參考 Windows 程式設計相關書籍。

現在請思考一個問題（此題雖與 OO 技術無關，卻是模組切割乾淨的關鍵）：應用模組所定義的這些表單命令，其識別碼如何讓框架模組知道，好讓框架模組所架設的訊息服務系統（6.6.3, 6.7, 6.8 節）也能夠為這些表單命令服務？欲獲得這些識別碼，必須含入 resource.h，難題在於框架模組的設計者不可能含入 resource.h，

因為它是事後由應用模組所開發的檔案，框架模組設計之時並不存在；如果含入它，會引發編譯錯誤。

要解決這個問題，一定要把視野從編譯期（compile time）移到執行期（run time）。

資源檔（.res）

如果框架模組這一端提供某種工具（資源編譯器，RC.EXE），能夠吃進 resource.rc，將其中的命令識別碼代以真正的（定義於 resource.h 的）數值，並將它們儲存為另一種格式（resource.res），再將框架模組設計為「執行期間才從 resource.res 讀取命令識別碼」，那麼兩個模組之間的耦合問題就可以化解。圖 6-14 的虛線，正是代表這個意思。

如果你曾經開發過 Windows 應用程式，你應該知道，應用模組的 .rc 檔是可以自由命名的，上一段卻令 MFCLite 固定使用 resource.rc 名稱，是否因此而仍舊沒有解除某種耦合或束縛呢？不，不是這樣，真正的整合開發環境（IDE）總是在專案（project）開始時允許應用程式員指定專案名稱，進而指定各種檔案名稱，並記錄下來做為日後讀取之用。一個被命名為 abc.rc 的資源描述檔一定會被整合環境所提供的資源編譯器改編為 abc.res，不能錯亂。MFCLite 只不過是未曾提供這份靈活罷了。這種情況無獨有偶，VC++ 的 AppWizard 也將任何專案用以置放各種 ID 定義的檔案命名為 resource.h，不給任何彈性呢☺。

這是「乾淨切割」的一個關鍵點。基於上述理由，我們也因此知道，一定得有一個中介工具，將應用模組中的 RC 檔內的 ID 定義（文字敘述），轉換為真正的數值，才能為框架模組所用。這個中介工具當然可以設計於框架模組內，但如此這般的任務分組並不理想，所以我仿照 Visual C++ 整合環境，寫了一個所謂的資源編譯器（RC.EXE）做為中介工具。當然，由於 MFCLite 並不支援視窗、對話盒、滑鼠等圖形介面，只以特定按鍵來粗糙模擬表單命令，因此 resource.rc 中沒有能夠讓你定義太多圖形資源（事實上也就只支援表單了）。這對我們的核心探討已經完全足夠。

將問題從編譯期移到執行期來思考，是上述模組乾淨切割的唯一方法，否則無解。

由於 MFCLite 對於表單只支援熱鍵，不支援滑鼠，也不支援圖像，上述 .rc 轉換為 .res 的設計也就格外簡單：

- .rc 檔的 menu（表單）格式完全與正式的 Windows 程式開發相同。
- 資源編譯器（RC.EXE）讀入（而非含入）來自應用模組的 resource.h 和來自框架模組的 afxres.h，取得每一個 ID 的文字名稱和對應數值。注意，一定得是「讀入」而非「含入（#include）」，因為我們必須建立各個 ID 的「文字名稱 vs. 對應數值」檢索表，供下一步驟使用，像這樣：

文字名稱	實值	
ID_FILE_SAVE	57603	(註：0xE103)
ID_FILE_NEW	57600	(註：0xE100)
...		

如果採用含入方式，無論如何得不到 ID 的文字名稱。

- 資源編譯器（RC.EXE）讀入 .rc 內的每一行文字，並只對 MENUITEM 採取動作：取其 '&' 符號後的第一個字元做為熱鍵，再取其識別碼文字名稱，檢索上一步驟所得表格，獲得識別碼實值，然後將熱鍵連同其後的識別碼組成一個 pair，寫入 resource.res。格式如下：

熱鍵	實值	
'S'	57603	(註：0xE103)
'N'	57600	(註：0xE100)
...		

圖 6-15 展示本節的概念與作法。



以下開始將大量討論框架模組和應用模組中的各個 classes。本書採用一個命名習慣，凡應用模組中繼承自任何 MFC(Lite) classes 者，皆以前綴詞 **CMy** 開頭。因此 CMyDocument 繼承自 CDocument，CMyView 繼承自 CView，CMyWinApp 繼承自 CWinApp。唯一例外是 CMainFrame（代表主視窗）繼承自 CMDIFrameWnd，CChildFrame（代表文件外框視窗）繼承自 CMDIChildWnd。此外所有全域函式皆以前綴詞 Afx 開頭命名，例如 AfxGetApp()，AfxWndProc()。所有 Win32 API 函式出現在文字說明時皆以 :: 開頭（此一符號乃 C++ scope operator；前面如果不加任何 scope 名稱，即代表「全域」），例如 ::SendMessage()，::DispatchMessage()。

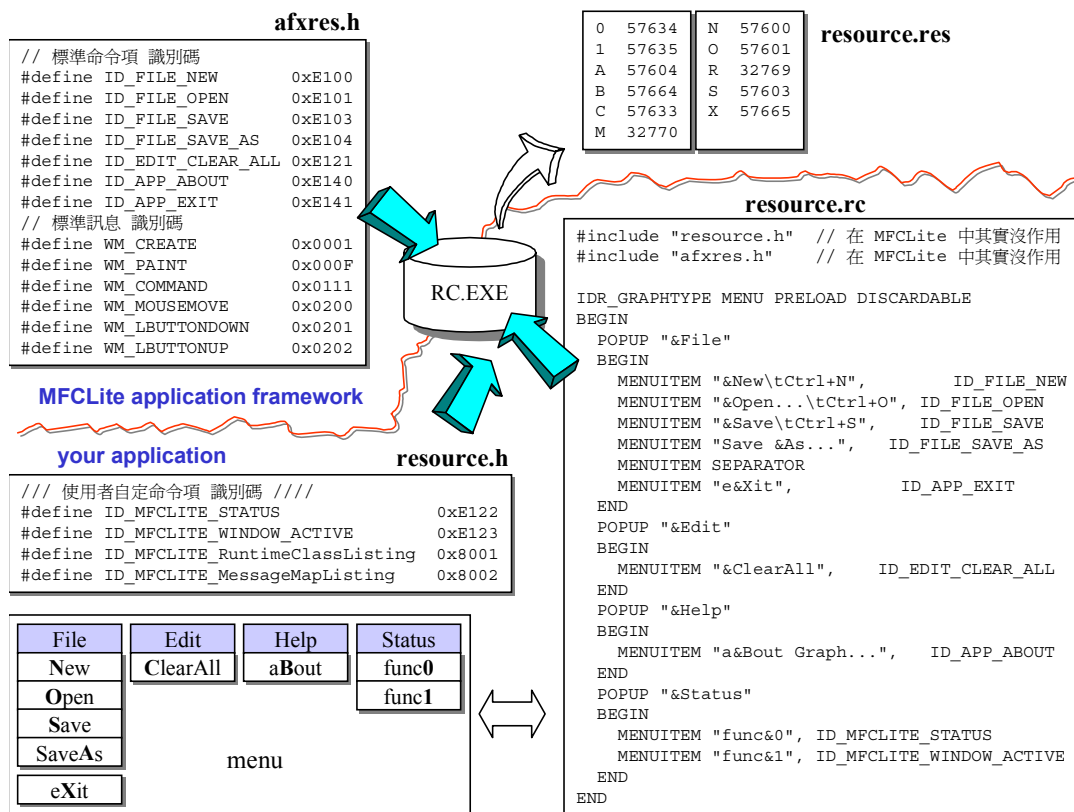


圖 6-15 「框架模組」與「應用模組」乾淨切割的關鍵點：.res 檔。圖中三個藍色箭頭表示 RC.EXE 的三份資料來源，產出右上角的 .res。 .res 不僅內含框架模組所定義的常數，也內含應用模組所定義的常數，如此一來框架模組讀取 .res 時就一併讀入了應用模組的定義，不必再含入（#include）應用模組端的表頭檔（那對乾淨切割是一種傷害）。注意切割線所區分出來的楚河漢界，一端為框架模組，一端為應用模組。圖左下角的意思是，視化工具可將 menu 直接轉換為文字描述，而 Windows GDI&USER 模組也可以將 .rc 文字描述轉化為圖像菜單。

請注意：Windows 應用程式開發工具中，資源編輯器（RC.EXE）除了解讀資源描述檔（.rc）並轉換為資源檔（.res），還有一個功能：將資源檔（.res）和可執行檔（.exe）結合起來成為一個獨立的可執行檔（.exe），執行時不再需要資源檔（.res）的存在。這項技術無關 OO，但相當實用。本書提供的 RC.EXE 並未模擬此一功能。參見 6.12 節「強化 MFCLite」。

6.6.2 應用程式相關類別 CWinApp, CWinThread 與骨幹應用程式之執行流程

應用程式的主架構被設計於框架模組中的 CWinApp 和 CWinThread，前者代表 application，後者代表 thread（執行緒）。由於 application 必是一個（*is a*）thread，它們的繼承關係也就確立了下來：CWinApp 繼承自 CWinThread。

MFCLite 只模擬 application framework 核心技術，並不打算模擬視窗圖像。然而與視窗之生命和活動有密切關聯的視窗訊息，卻又是視窗作業系統之核心與視窗程式生命之所繫。因此 MFCLite 不能或缺地必須模擬事件驅動特性和訊息派送機制。訊息的攫取和推送，代表程式的生命，而程式的生命在執行緒（threads），所以訊息的攫取和推送被設計於 CWinThread 內。CWinApp 負責的則是「不同的 applications 可能會有不同的作為」——例如可能需要不同的視窗風貌。當然，所有開放給應用模組覆寫（override）的功能，必然都是虛擬函式。

圖 6-16 是 MFC(Lite) 和其應用程式的互動關係，以源碼形式呈現。MFC(Lite) 規定，應用模組必須覆寫 CWinApp::InitInstance()，俾在其中產生主視窗。產生主視窗之前，應用模組還有許多重要動作要做（主要是設定 MVC 模型，詳見 6.9 節），但本節只討論「視窗誕生」這個部分，旁支暫略。圖左半是 MFC(Lite)，圖右部是 MFC(Lite) App。圖中標示的序號，清楚說明了程式的流程。

圖右側所示的應用程式骨幹，寫法標準而制式，因此以應用程式碼產生器（類似 VC++ AppWizard 之類的產品）來完成，絕無問題——只需幾個問答就好。

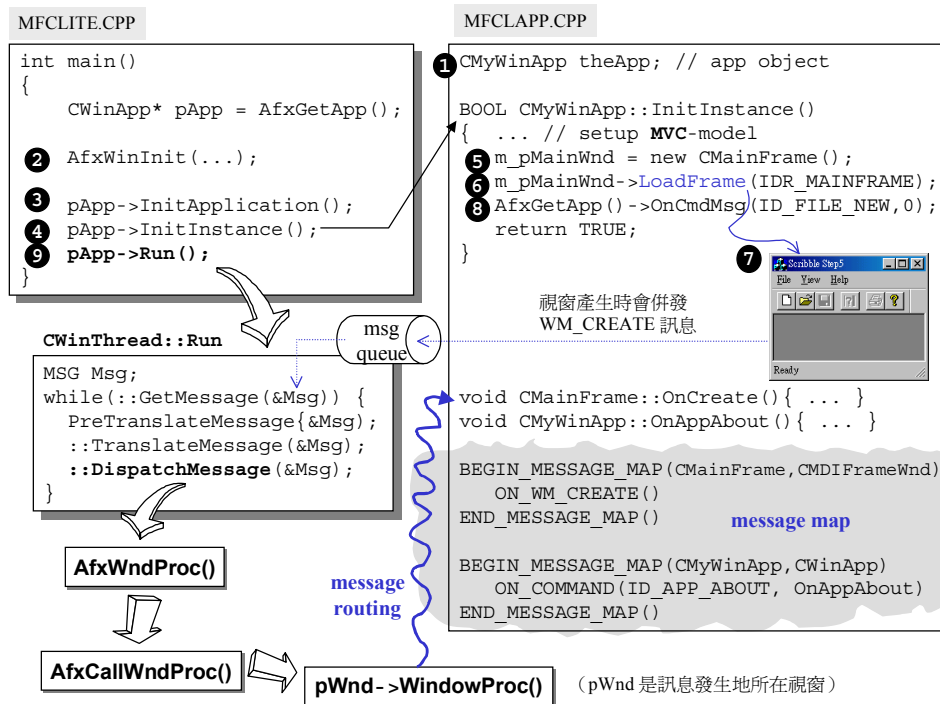


圖 6-16 以源碼方式呈現的 MFC(Lite) 和應用程式互動關係

下面便是圖 6-16 的運行過程。文字前如有號碼，便是對應圖 6-16 內的號碼。

- ❶ C++ 程式應該以 objects 為主角來運行。應用模組因此必須首先產生一個 CMyWinApp object，簡稱 **app object**，這個全域物件是整個應用程式的引爆點，其建構式更早於 main() 之前執行。MFC(Lite) 要求應用程式員必須在其 CMyWinApp 之中覆寫 CWinApp 的虛擬函式 InitInstance()，後述。
- 流程進入 main()。MFC 中的它會呼叫 AfxWinMain() 函式，後者執行圖 6-16 左上角所描述的動作。為求簡化，MFCLite 直接在 main() 中完成那些動作。
- **app object** 由應用模組產生，但是 MFC(Lite) 必須獲得這個 object 才能開始運作。由於程式員可對 app object 任意命名，MFC(Lite) 有什麼方法可以動態獲得這個 object？這再次考驗框架模組與應用模組之間的乾淨切割技巧。辦法之一是利用「app object 之生成一定得通過 CWinApp::CWinApp()」這個機會，彼時的 this object 便是 app object，因此框架模組便能正確取得 app object 並記錄下來，之後便能透過這個 app object，以應用模組的角色在

框架模組中發號施命。MFC(Lite) 並提供一個全域函式 `AfxGetApp()`，用以傳回被記錄下來的那個 `app` object。依此設計理念，只要為 `CWinApp` 設計一個 `CWinApp*` data member 用以儲存 `this` 指標即可，但由於 MFC 內有更多複雜事務需要處理，而 `MFCLite` 又模擬了這些複雜動作的介面（只不過為求簡化而遺留了許多空殼），因此實際源碼比上述稍稍複雜一些。圖 6-17 是以上描述的示意圖。

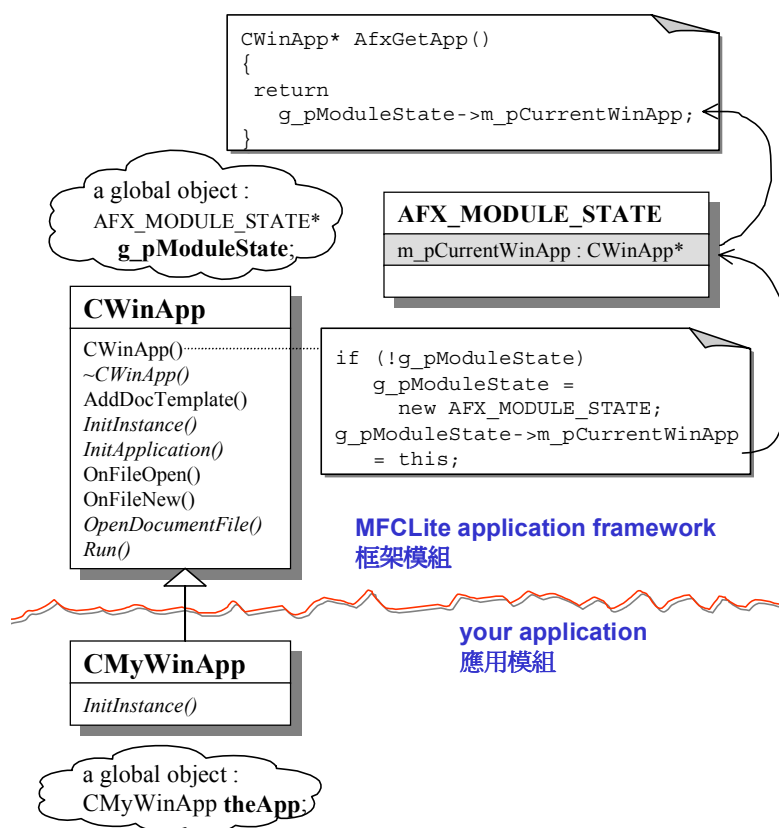


圖 6-17 應用模組端產生全域物件 `theApp` 時，一定會經過位於框架模組端的 base class ctor `CWinApp::CWinApp()`，只要該處將 `this` 記錄下來，那便是 `CMyWinApp` object。此後任何時候需要這個 object，呼叫 `AfxGetApp()` 便可得到。有了以上機制，MFC(Lite) 便能確保它在任何時候呼叫 `AfxGetApp()` 時，都能正確取得 **app object**（這是 MFC(Lite) 發展之時並不存在的一個東西），因而得以「正確的身份」來驅使所有動作。

- `main()` 之中利用 `AfxGetApp()` 取得 **app object**，這正是上述手法。
- ❷ `AfxWinInit()` 原本用以做視窗註冊動作。MFC Lite 並不支援圖形介面，所以略而不顧，但卻在這裡安排了表單命令識別碼的讀取，詳見 6.6.1 節。
- ❸ `pApp->InitApplication()` 用以呼叫為每個應用程式而設計的專屬行為。這些行為應該被寫在 `CMyWinApp::InitApplication()` 中。一般而言，視窗系統的每個應用程式的共通性質差異不大，因此應用模組通常不需覆寫此一虛擬函式。
- ❹ `pApp->InitInstance()` 用以呼叫為每個應用程式之執行實體 (instance) 而設計的專屬行為。這些行為應該被寫在 `CMyWinApp::InitInstance()` 中。一般而言視窗系統的應用程式的每一個執行實體的唯一差異是：它們必須為自己產生一個可能略帶些微差異（但也可能無任何差異）的主視窗。因此框架模組不提供任何預設行為，應用模組一定得覆寫此一虛擬函式。
- ❺ 由於 Template Method 的作用，上一個步驟之後，流程由應用模組接管。`CMyWinApp::InitInstance()` 的標準動作是，首先設定 MVC 模型（詳見 6-9 節），然後產生主視窗。任何視窗系統（當然包括 Microsoft Windows）都以 handles（號碼牌）代表一個實際的、作業系統層面的、以 system calls (APIs) 產生出來的視窗。MFC(Lite) 介入後，視窗的誕生分為兩組動作，首先以 `new` 方式產生出一個用於管理的、屬於 MFC(Lite) 層面的 `CWnd-derived class object`，然後才以 `system call` 產生一個屬於視窗作業系統層面的實際視窗，並將實際視窗的 window handle 記錄於 `CWnd-derived class object` 的某個 data member 內。請參考圖 6-18。

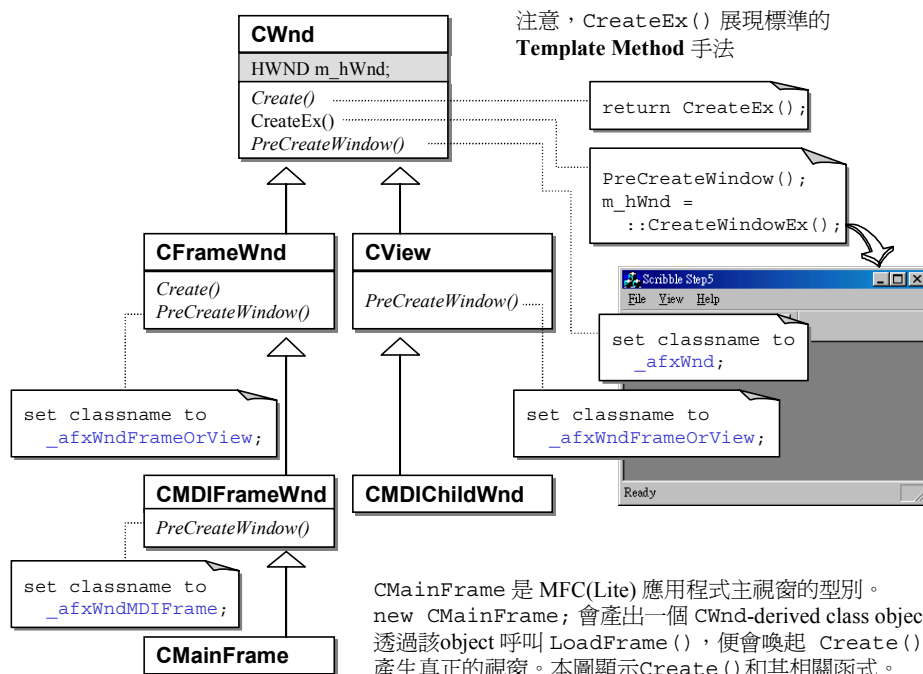


圖 6-18 ⑥ 在 MFC(Lite) 中，視窗的生成行動總是分為兩部分，一是 OO 層面用於管理與行動策劃的 CWnd-derived object，一是視窗系統層面的實際視窗。後續章節對於 CWnd-derived classes 另有詳細介紹。本圖顯示，在 `::CreateWindowEx()` 產出實際視窗之前，有一個前置動作 `PreCreateWindow()`，主要用來「登錄視窗類別」。此處所說的視窗類別（**window class**）並非一個 C++ class，而是某個 C struct 的泛稱，內含「某種」視窗該有的各種屬性，參見 6.6.3 節和 6.7 節，或其他 Windows 程式設計書籍。本章的 MFCLite 並不真正模擬視窗實相，所以不需要 window class（至於必要的訊息的繞送與處理，詳見 6.7 節），也就不需要為圖中各個 classes 設計 `PreCreateWindow()`，但仍保留其介面（詳見書附源碼）。圖中的藍色文字都是 MFC 內建的 window class；MFCLite 不打算產生真正的視窗，所以也就不需要它們；MFCLite 源碼已將它們全部標注起來。

❶ 圖 6-16 右側應用模組中的這兩行：

```
m_pMainWnd = new CMainFrame();
m_pMainWnd->LoadFrame(IDR_MAINFRAME); // 產生視窗，過程複雜（詳 6.9）
```

於是便依序先產生一個 Cwnd-derived class object 然後產生一個從屬的「實際視窗」。LoadFrame() 的參數與 MVC 模型有密切關連，6.9 節詳述。

- ❷ 流程進入 AfxGetApp() -> OnCmdMsg(ID_FILE_NEW, 0)。正規的（由 VC++ AppWizard 產生出來的）MFC 應用程式骨幹中，這裡原本動作比較複雜，詳見書附源碼中的註解，主要是分析命令行參數（command line parameters）是否帶有文件檔名，如果沒有（通常如此）就開啓一份未具名的空文件，這也就是你一啓動任何 Windows 程式時幾乎都會自動出現一份空白文件的原因。MFCLite 把上述動作簡化，不做命令行參數剖析，直接要求開空白文件。式中的 OnCmdMsg() 關係到訊息的傳遞，空白文件的開啓則關係到 MVC 模型，均於後續章節詳述。
- CMYWinApp::InitInstance() 結束之前，應該有些動作將主視窗顯示出來並爲它畫上初始畫面。MFC 應用程式採行的是 pMainFrame->ShowWindow() 和 pMainFrame->UpdateWindow()。由於 MFCLite 並不模擬實相視窗，所以省略這兩個動作。
- ❸ 流程再度回到框架模組。pApp->Run() 是程式脈動中樞，它所喚起的是 CWinApp::Run()，其內又喚起 CWinThread::Run()。後者以迴圈方式不斷呼叫各個函式，從 msg queue 抓取訊息（::GetMessage()）、轉換訊息（CWinThread::PreTranslateMessage(), ::TranslateMessage()）、派送訊息（::DispatchMessage）。於是，應用程式進入 message-based, event-driven 的生命模式。「訊息相關機制」之模擬，請見 6.6.3 節和 6.7, 6.8 節。

以上便是 MFC(Lite) 框架模組與 MFC(Lite) 應用模組之間的合作概要。應用模組端的動作幾乎完全制式化，因此我們可輕易發展出一個「應用程式骨幹產生器」，利用簡單幾個問答（例如希望取用什麼 class 名稱等等），讓程式員輕鬆獲得一個應用程式骨幹，從而獲得一個具備基本功能的視窗應用程式。這所謂「應用程式骨幹產生器」正是 Visual C++ AppWizard 所扮演的角色。

6.6.3 訊息相關機制之模擬

```
::SendMessage(), ::TranslateMessage(),  
::DispatchMessage(), ::PostMessage()
```

雖然「訊息相關機制」偏屬視窗作業系統的成份遠比偏屬 OO 的成份為多，但現今任何商用軟體幾乎都在各種視窗作業系統上開發，因此任何一個具實用價值的 application framework 不可能不規劃訊息處理機制³。

為了模擬 MFC 中的訊息映射和訊息繞送機制，MFCLite 必須逼真模擬實際訊息以供流動、以供處理，否則測試程式無法動起來。MFCLite 以儘可能簡化的方式來模擬訊息的發生和攫取，其介面完全相容於 MFC，只是實作上簡化許多。

談到訊息，我們必須先對 Windows 系統的訊息機制（圖 6-19）有一個基本認識。訊息起源於週邊設備驅動程式，彼等偵測到設備狀態有變化（稱為事件 **event**，例如滑鼠移動或滑鼠左鍵被按下，或某個鍵盤按鍵被按下），便將 event 相關資訊交給 Windows 作業系統中的 USER 模組，由該模組根據目前的視窗狀態，判斷這個 event 屬於哪個視窗，從而製作出一個對應的訊息（`struct MSG`），放入系統佇列（**system queue**）。另一個訊息來源是，其他視窗也可能以 `::PostMessage()` 將某個訊息傳給另一個視窗，藉此達到 IPC（InterProcess Communication）的效果；這類訊息被放在所謂的應用佇列（**app queue**）中。任何 Windows 程式都必須有一個所謂的訊息迴圈（**message loop**），不斷抓取訊息，並將訊息派送（**dispatching**）至視窗行為中樞——所謂的視窗函式（**window function**，或稱 **window procedure**）去。在那兒，訊息將被辨識、被分門別類地處理。視窗行為中樞必須由應用模組來設計——只有你才知道你想處理哪些訊息，也只有你才有權決定怎麼處理它們。**window function** 可被任意命名，但使用之前必須先向作業系統登錄（利用 `::RegisterClass()`）；這個函式由應用模組提供，卻只能被作業系統呼叫，是為所謂回呼（**call back**）函式。

³ 這裡所謂訊息，是視窗作業系統的訊息，是驅動程式偵測週邊設備如滑鼠、鍵盤的動態而包裝成的一份資料結構，不是傳統 OO 世界所謂「送訊息給某個 object」的訊息。在傳統 OO 世界中，「送訊息給某個 object」意味喚起該 object 的某個 function（或稱 method）。

訊息，大致可分為視窗訊息和命令訊息兩大類。視窗訊息可能用來表示滑鼠的移動（WM_MOUSEMOVE）或按鍵（WM_LBUTTONDOWN, WM_LBUTTONUP），或是視窗的誕生（WM_CREATE）或摧毀（WM_DESTROY），或是通知視窗重繪（WM_PAINT）。命令訊息（WM_COMMAND）則泛指所有因表單（menu）上的選按動作而引發的訊息，其中內含一個欄位，標示出被選按的命令項識別碼（ID）。

這樣的觀念在任何視窗作業系統上都有，只不過大同小異罷了。

在 MFC 中，訊息迴圈被隱藏於 CWinThread::Run()，先前你已經看過了；數種標準視窗的視窗函式（window function）亦已事先設計好。由於 MFCLite 並不打算實現真正的視窗，也就不需模擬「視窗函式的設計與登錄動作」。

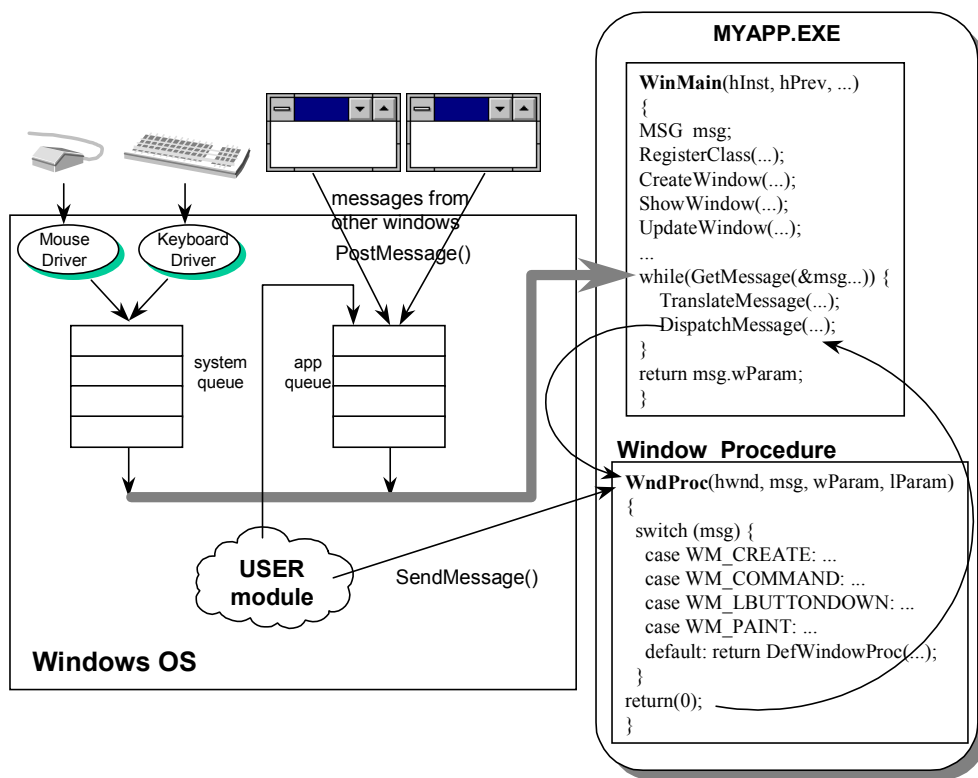


圖 6-19 Windows 訊息機制。圖左為作業系統，圖右為應用程式。圖右所呈現的便是所謂 Win32 API（或 SDK）編程模型（programming model）。

爲了簡化訊息機制，MFCLite 做了以下設計：

- 只接受鍵盤按鍵，不接受滑鼠動作。
- 目前版本只模擬極少量視窗訊息，這已足夠驗證 6.7 節之訊息映射和 6.8 節之訊息繞行的正確性，並足以驅動一個 MFCLite 應用程式之基本運行。這些訊息分別是：'c' 代表 WM_CLOSE，'p' 代表 WM_PAINT，'u' 代表 WM_LBUTTONDOWN，'d' 代表 WM_LBUTTONDOWN，'m' 代表 WM_MOUSEMOVE。你隨時可以修改 MFCLite 源碼，支援更多訊息。
- 可接受任意命令訊息 — 由應用模組自行設定（詳見 6.7 節）。
- 只模擬一個 message queue，無分 system queue 或 app queue。
- 訊息結構定義如下，比正規 Windows 訊息簡化兩個欄位：

```
typedef struct tagMSG {
    HWND      hWnd;
    UINT      nMsg;
    WPARAM    wParam;
    LPARAM    lParam;
    // DWORD   time;
    // POINT   pt;
} MSG, *PMSG;
```

- 採用 C++ 標準程式庫中的 Standard Template Library (STL)，以 `queue<MSG>` 做爲 message queue：

```
#include <queue>
...
queue<MSG> g_msgQueue;    // 模擬 message queue.
```
- 「訊息的落點視窗」原本需由滑鼠落點或視窗焦點判斷之，這是 Windows USER 模組的任務。MFCLite 既未模擬實相視窗，亦無必要模擬如此龐大複雜的 USER 模組。我讓 MFCLite 永遠追蹤記錄當前作用視窗（active window，只能有一個）；一旦訊息發生，便視之爲「發生於當前作用視窗中」。「取得當前作用視窗」的動作被我設計爲 `JJGetActiveFrame()`。至於作用視窗之切換，在真正的視窗系統中通常由終端使用者以滑鼠點選進行，如今在簡化的模擬世界中必須有新方法。我將採用「熱鍵切換」方式，後述。
- 真實視窗系統允許程式員自定視窗訊息。MFCLite 不允許如此，但它和 MFC 一樣，允許應用程式員定義自己的命令項。MFC(Lite) 內建了一些標準命令項如 `ID_FILE_OPEN`，`ID_FILE_SAVE`，`ID_APP_ABOUT`。
- 6.6.1 節已經說過，框架模組必須讀取應用模組所定義的命令項，才能針對這些命令項提供訊息繞行服務，但這些命令項在框架模組設計當時是未知的，

不可能在編譯期間被含入，必得在執行期間讀入。為此我也模擬 VC++ 整合環境發展了一個資源編譯器（亦名 RC.EXE），用來讀取應用模組的資源描述檔 .rc 檔並轉換為資源檔 .res，詳見圖 6-15。MFCLite 支援的 UI 資源種類只有表單（menu），其描述語法與正規 Windows 應用程式完全相同，每個 MENUITEM 描述句中的 '&' 後繼字元便是快捷鍵，例如 MENUITEM "e&Xit" ID_APP_EXIT 表示，一旦使用者按下大寫鍵 'X'，程式便送出命令訊息且其命令項為 ID_APP_EXIT(0xE141)。這些命令項由 MFCLite 的 AfxWinInit() 讀入，置於一個 STL map 中：

```
map<char, unsigned int> g_MenuTable;
```

其中第一參數表示快捷鍵，第二參數表示命令識別碼。稍後當程式開始接收訊息（見稍後的 ::GetMessage()），只要將按鍵拿來和 g_MenuTable 的所有元素的第一欄位比對，立時可知它是否代表「下了一道命令」。

- 攫取（GetMessage）、轉換（TranslateMessage）、派送（DispatchMessage）、傳送（PostMessage）等訊息處理動作模擬如下：

```

BOOL GetMessage(MSG* pMsg)
{
    if (!g_msgQueue.empty()) {           // 如果 msg queue 不為空。
        *pMsg = g_msgQueue.front();      // 注意，一定要拷貝出來。
        g_msgQueue.pop();                // 從 queue 中移除第一個元素。

        if (pMsg->nMsg == WM_QUIT)
            return FALSE; // 結束訊息迴圈

        return TRUE;
    }

    // msg queue 中沒有訊息，試圖從鍵盤取得 event，包裝成 MSG 訊息。
    cout << "->";           // MFCLite 提示號
#ifdef __GNUC__
    char c = getchar();      // 取得一個按鍵（需按 Enter）
#else
    char c = getche();        // 取得一個按鍵
    cout << endl;
#endif

    // MFCLite 希望最好以大寫鍵模擬命令訊息，小寫鍵模擬視窗訊息（但是你有自由）
    map<char, unsigned int>::iterator pos = g_MenuTable.find(c);
    if (pos != g_MenuTable.end()) {           // 確有吻合者
        pMsg->hWnd = JJGetMainFrame();        // 命令必定來自 MainFrame
        pMsg->nMsg = WM_COMMAND;              // 命令訊息
    }
}

```

```

    pMsg->wParam = pos->second;          // 命令項（識別碼）
    return TRUE;
}

// 非命令訊息。判斷是否為以下視窗訊息。每個非命令訊息都判給當前作用的
// 視圖視窗（active view）所有。這是合理的假設。
// 以下是 MFCLite 支援的數個視窗訊息。
switch (c) {
    case 'p' : // WM_PAINT
        pMsg->hWnd = JJGetActiveView();
        pMsg->nMsg = WM_PAINT;
        return TRUE;

    case 'd' : // WM_LBUTTONDOWN
        pMsg->hWnd = JJGetActiveView();
        pMsg->nMsg = WM_LBUTTONDOWN;
        return TRUE;

    case 'u' : // WM_LBUTTONUP
        pMsg->hWnd = JJGetActiveView();
        pMsg->nMsg = WM_LBUTTONUP;
        return TRUE;

    case 'm' : // WM_MOUSEMOVE
        pMsg->hWnd = JJGetActiveView();
        pMsg->nMsg = WM_MOUSEMOVE;
        return TRUE;

    case 'G' : // system hotkey for debu'G'
        ... // 除錯用。詳見 6.11 節。
        return TRUE;

    case 'c' : // WM_CLOSE，見 6.12.3 節。
    default :
        pMsg->hWnd = JJGetActiveView();
        pMsg->nMsg = WM_UNKNOWN;
        return TRUE;
}
}

```

```

void TranslateMessage(MSG* pMsg)
{
    // 此動作原本為轉換某些按鍵訊息。MFCLite 保留此一介面，忽略其中動作。
}

```

```

void DispatchMessage(MSG* pMsg)
{
    // 原本這裡應該將訊息派送到它所發生的地點（視窗）的對應的
    // 視窗函式（Wnd Proc）去。在 MFC2.5 中，四個標準視窗的視窗函式
    // 都是 AfxWinProc()，所以應流往 AfxWinProc()。MFC4.x 的作法更複雜些，
    // 但經由 hooking 和 subclassing 的手法，仍然流往 AfxWinProc()。
}

```

```
// 以下直接呼叫 AfxWinProc()，那是訊息繞行起點。
AfxWinProc(pMsg->hWnd, pMsg->nMsg, pMsg->wParam, pMsg->lParam);
}
```

```
BOOL PostMessage(HWND hWnd, UINT Msg, WPARAM wParam, LPARAM lParam)
{
    MSG msg;
    msg.hWnd = hWnd;
    msg.nMsg = Msg;
    msg.wParam = wParam;
    msg.lParam = lParam;
    g_msgQueue.push(msg);

    return TRUE;
}
```

以上實作技巧中，需要特別注意的是，從 `msg queue` 取出訊息時，不能採用 `reference` 語意，換言之不能這麼做：

```
pMsg = &(g_msgQueue.front());
```

那會造成 `alias`（別名）現象。果真如此，稍後執行以下動作時：

```
g_msgQueue.pop(); // 從 queue 之中移除第一個元素。
```

`pMsg` 所指的訊息將同時被摧毀，造成後續對該訊息的所有動作都發生錯誤。

我們必須採取 `value` 語意：

```
*pMsg = g_msgQueue.front(); // 拷貝出來。
```

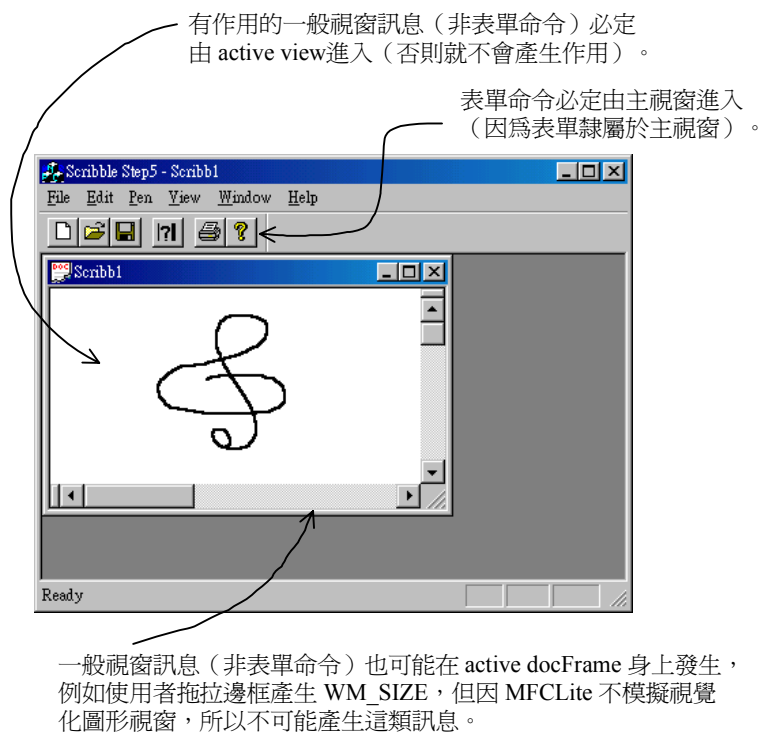
另外需要注意的一點是，`GetMessage()` 用到了我自行定義的兩個以 "JJ_" 起頭的全域函式。同類型的全域函式還有數個，但此處尚未用到。這三個函式在 `MFCLite` 的視窗模擬環境（無真正圖像，純為觀念）中有以下功能：

`JJGetMainFrame()`：取得唯一一個 `mainFrame`（主）視窗。

`JJGetActiveView()`：取得目前作用中的 `view`（視圖）視窗。

`JJGetActiveFrame()`：取得目前作用中的 `docFrame`（文件外框）視窗。

`GetMessage()` 之中令 `WM_COMMAND` 永遠來自 `mainFrame` 視窗，令其他 `WM_x` 訊息永遠來自 `active view` 視窗，這是合理的假設，原因如下圖所示：



現在，我們可以取得訊息了，也可以讓應用模組自由自在地設計其命令項而不至於影響框架模組。接下來當然應該探討訊息映射和訊息繞送。但是我們必須先對訊息誕生地：視窗，有更多的了解與實踐。

6.6.4 視窗相關類別 CWnd, CFrameWnd

本節只介紹視窗相關物件和實象視窗的「誕生與管理」。至於視窗訊息的處理，詳見 6.7, 6.8 節。

視窗，在任何視窗作業系統中都是當然的主角。但是如果要模擬出真正的視覺化圖像視窗，非等閒可為——那不只是圖像表達而已，還需包括視窗管理：上下層覆疊、大小改變、位置拖放…。整個工作幾乎相當於 Microsoft Windows 作業系統中的 USER 模組加上一小部分 GDI 模組。

視窗雖是 Windows 作業系統的核心，卻不是眼下我所關注的 application framework

核心，所以我不打算在 MFCLite 中模擬它。但是眼中可以沒有視窗，心中不能沒有視窗概念，否則訊息無安身立命之所，MFCLite 的教育價值將大打折扣。在這個議題上，我對 MFCLite 的規劃是（更多說明請見 6.12.3 節內的「MFCLite 簡易視窗管理」）：

- 完全模擬 MFC 對於視窗誕生與摧毀等各種動作所提供的介面，但是最後關頭並不真正產生視窗，只產生一個用以代表視窗的權柄（window handle）。
- 簡化視窗管理。既然沒有真正的圖像視窗，也就沒有視窗在螢幕上活動所衍生的各種複雜狀態。因此 MFCLite 只以一個 map 來管理所有視窗，並以兩個全域變數記錄 active view 視窗和 active docFrame 視窗。關於 View-docFrame，詳見 6.9 節。
- 上述之所以使用 map，是因為考量了一個事實：MFC(Lite) 中的每個實象視窗實際上和特定的某個 CWnd-derived class object 緊緊對應，成為生命共同體⁴。為此，我們時常需要在獲得一個 window handle 時反求其對應的 CWnd-derived class object。MFC 有實際的視窗系統（USER 模組）在背後支援，得以輕易利用各種 APIs 得到所要的東西；MFCLite 沒有這種奧援，所以乾脆在視窗誕生之際將 window handle 和 CWnd-derived class object 分別當作 map 的鍵值（key）和實值（value）記錄起來，以後就很容易反推了。

欲探討 MFCLite 中視窗的誕生，我們必須先探討實際視窗系統中的視窗的誕生。圖 6-19 已經告訴你，一個視窗在被 ::CreateWindow() 或 ::CreateWindowEx() 產生出來之前，必須先經過 RegisterClass()，後者設定視窗的恆長屬性，前者才能據以實現出一個個略帶差異風貌的視窗實體。所謂「恆長屬性」包括視窗的行為中樞（亦即所謂 window function）和視窗的定位（一般視窗、對話盒、或...），所謂「稍有差異的風貌」包括視窗誕生位置、視窗標題...⁵。

所有與視窗相關的 MFC(Lite) classes，全都以 CWnd 為根源。如果我們把產生視窗

⁴ 實象視窗負責實際的視窗行為（包括圖像的各種變化以及訊息的攫取），CWnd-derived class object 負責訊息在 MFC(Lite) classes 體系中的流動與比對，最終再將訊息交還給視窗作業系統（例如 Microsoft Windows 的 USER 模組）完成處理。詳見 6.7, 6.8 節。

⁵ 請參考視窗程式設計（API 層面）相關書籍。如果你面對的是 Microsoft Windows，我建議你閱讀《Programming Window》by Charles Petzold，或《Windows95: A Developer's Guide》by Jeffrey Richter & Jonathan Locke。

的動作設計在 `CWnd::Create()` 內，再在其中呼叫 `PreCreateWindow()`，並令後者為虛擬函式，便可達到 **Template Method** 的效果：由 base class 完成演算法骨幹，並將「因型別而有所變異」的部分交由 derived classes 完成。

這些 CWnd-derived 相關概念已在圖 6-18 中表示得非常清楚。關鍵源碼非常簡短，直接列於圖中。也許你會認為，圖中所示的 `Create()` 和 `CreateEx()` 其實可以合而為一，不必分割為兩個函式。我同意你的看法，不過為了儘量與 MFC 介面相容，我還是保留了它們。

圖 6-18 帶來的結論是：(1) 只要你想產生視窗，就想辦法喚起 `CWnd::Create()` —— 可能直接喚起，也可能透過層層包裝後喚起（例如透過 `LoadFrame()`，詳見 6.9 節）。當然也請注意，面對任何一個「不曾覆寫虛擬函式 `Create()`」的 CWnd derived classes（例如 `CView`, `CMDIFrameWnd`, `CMDIChildWnd`），呼叫其 `Create()` 函式其實也就是喚起 `CWnd::Create()`，因為它被繼承下去了。(2) 如果你需要為你的視窗登錄新的視窗類別（*register a new window class*；當然 MFCLite 應用程式是絕對不需要的），請在相應的 CWnd-derived class 中覆寫 `PreCreateWindow()`，並在其中完成登錄動作。

下面是 MFCLite 的視窗產生動作和視窗管理動作。有一些需要特別注意的實作手法，已在程式註解中說明：

```
// 全域物件 (global objects)
queue<MSG> g_msgQueue; // 模擬 message queue.
map<HWND, CWnd*> g_state; // 記錄所有的 window-handles/CWnd-objects
```

```
BOOL CWnd::CreateEx(/*...*/ LPVOID lpParam)
{
    CREATESTRUCT cs;
    PreCreateWindow(cs); // 被喚起的可能是任何 CWnd-driven classes
                        // 的 PreCreateWindow()。
    m_hWnd = ::CreateWindowEx(lpParam); // 真正產生視窗。
    g_state[m_hWnd] = this; // 視窗管理。

    return TRUE;
}
```

```
> HWND CreateWindowEx(/*...*/ LPVOID lpParam) // jjhou change
{
```

```

// MFCLite 並不真正產生視窗，只給個模擬的 window handle。
static HWND hWnd=0;
++hWnd;    // window handle 將以 1,2,3,4...的趨勢產生出來。

// 模擬 WM_CREATE 訊息的產生（並進入 msg queue 中）
// WM_CREATE 的 lParam 應置入 CREATESTRUCT（視窗生成時的所有資訊）位址
// 本例已簡化 CREATESTRUCT，使它只含一個成員：lpCreateParams，
// 其值按 MFC 介面規格，應設為本函式接獲的最後一個引數：lpParam。
CREATESTRUCT* pcs = new CREATESTRUCT;
if (lpParam == NULL)
    pcs->lpCreateParams = lpParam;    // 既然是 NULL，淺拷貝即可。
else
    pcs->lpCreateParams = new
        CCreateContext(*((CCreateContext*)lpParam));
// 如果不採用深拷貝（deep copy），會造成 alias。是的，當訊息包裝完成，
// 回返至呼叫端，由於該處的 CCreateClass object 會因為離開 scope
// 而被摧毀，此處準備好的 MSG 內的指標也就成了 dangling pointer。

// 注意，以上用了兩個 heap object，因此訊息接受者（例如 OnCreate()）
// 用畢訊息之後，應注意 clean-up。
MSG msg;
msg.hWnd = hWnd;
msg.nMsg = WM_CREATE;
msg.wParam = 0;
msg.lParam = (LPARAM)pcs;    // 訊息接受者應該負責 clean-up。
g_msgQueue.push(msg);    // 將訊息推入 msg queue。
return hWnd;
}

```

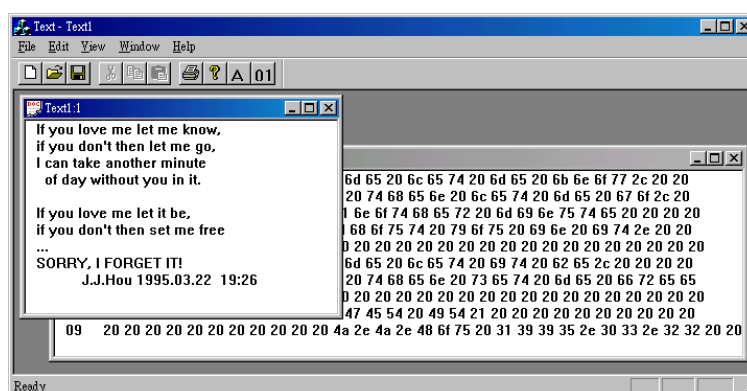
```

BOOL DestroyWindow(HWND hWnd)
{
    // 這裡應該真正摧毀視窗。
    // 文字模式中無法產生視窗（且非本例焦點），所以略之。
    return TRUE;
}

```

6.6.5 MDI 相關類別 CMainFrameWnd, CMDIChildWnd

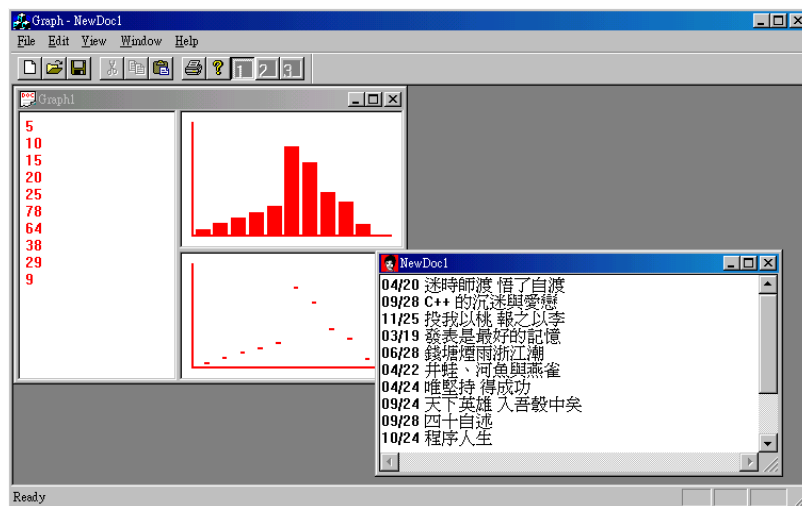
Windows 系統提供了一種所謂的 **MDI（Multiple Document Interface）** 視窗風貌，允許多個子視窗同時存在於一個父視窗內（並有良好的管理介面），像這樣：



和 MDI 相反的是所謂 **SDI：Single Document Interface**，表示父視窗內不能有多個子視窗存在（並有良好管理）。MFC 同時支援 MDI 和 SDI 兩種視窗管理風貌。

多視窗風格和所謂 "Document" 扯得上什麼關係呢？為什麼名稱之中有個 "Document" 呢？基本上 Windows 技術術語把視窗所顯示的資料視為一份 "document"（文件），上述多個視窗擁有多份資料，便稱為 multiple documents。然而，MFC(Lite) 允許同一份資料以不同形式呈現（此為 6.9 節 MVC 模型的主題），上圖兩個子視窗所映射的其實是同一份資料的兩種示相。這樣是不是也能稱為 multiple documents 呢？依我之見，將 Multiple Document Interface 斷句為 Multiple Document Interface 或許要比斷為 Multiple Document Interface 為佳。

無論如何，我們知道了什麼是 MDI。MFC 不僅允許主視窗有 MDI 的能力，子視窗也可以 MDI（如果循環下去，便可有無數層次的 MDI）。例如，以下主視窗內的左子視窗又有三個子視窗：



如果三個子視窗又都採用 MDI 風格，那就又可以…。一般而言我們不會把 UI 做得那麼複雜，通常我們會在三個子視窗上選用視圖視窗（View），詳見 6.9 節。

但是，即使僅僅實現如上圖「第一層子視窗（docFrame 視窗）內又有子視窗」的風貌，對於我所設定的教育目標而言也是太複雜了。事實上 MFCLite3 的設計過程中，原本並未模擬 MDI，一樣可以良好表現 application framework 的核心技術。只因 MDI 可以使 MFC(Lite) 呈現出設計樣式 **Observer** 的效果，本身又是設計樣式 **Composite** 的實踐（詳見第 7 章），我才決定納入它。至於放棄「子視窗中再度擁有 MDI」，並不影響 **Observer** 和 **Composite** 的展示，所以我做了以下決定：

MFCLite 允許主視窗有 MDI 功能，但不允許子視窗也有 MDI 功能
（亦即不允許類似上圖左側的三岔視窗）

換句話說：

- MFCLite 應用程式永遠只有一個主視窗，其中可有多個子視窗（6.9 節稱之為 docFrame 視窗，因為它像外框一樣地包覆了一份 document）。
- MFCLite 應用程式一旦產出 n 個 docFrame 視窗，也就同時擁有了 n 個 views 視窗（詳見 6.9 節）。

下面是 MFC(Lite) 及其應用程式中四個用途不同的 CFrameWnd-derived classes：

```
// in application framework
```

Polymorphism in C++, 2e

```
class CMDIFrameWnd : public CFrameWnd;
class CMDIChildWnd : public CFrameWnd;

// in application
class CMainFrame : public CMDIFrameWnd;      // 用於主視窗
class CChildFrame : public CMDIChildWnd;      // 用於 docFrame 視窗
```

MFCLite 要求，每個應用程式主視窗都必須繼承自 `CMDIFrameWnd`，每個文件視窗（`docFrame`）都必須繼承自 `CMDIChildWnd`。這樣便保持了與「一般」MFC 應用程式相同的編程風貌⁶。在此同時，爲了簡化，MFCLite 又令 `CMDIChildWnd` 暫略 MDI 能力，你可以從書附源碼看出，`CMDChildWnd` 幾乎等同於 `CFrameWnd`。

MDI 視窗的誕生機制，和一般視窗沒有兩樣，圖 6-18 已經做了說明。比較麻煩的是其訊息繞送機制，詳見 6.8 節。

⁶ 「一般」MFC 應用程式採用這種風格，但作爲一個商業用途的 application framework，MFC 還允許應用程式做出各種其他變化。

6.7 訊息映射 (message mapping)

原本是兩條平行線！

是的，視窗系統和物件導向，原本在兩條平行線上發展。因著技術的發展，視窗系統需要物件導向的協助，物件導向需要在視窗系統風潮中證明其技術的優越性。

於是我們有了各種 application frameworks，其中有許多 classes，一層又一層包裝了視窗系統中的程式、執行緒、視窗、對話盒、檔案、繪圖工具，乃至於文件（資料）、群集（資料結構）、同步工具（synchronization）…。這都沒有問題。最大的問題是，以 message-base, event-driven 為最大特徵之視窗程式，其生命所繫的「訊息（message）」該如何處理，才能配合 application framework 所衍生出來的 classes 階層體系？

問題的起源是這樣的。原本，以 Windows 系統服務（APIs）撰寫而成的程式（或謂 SDK 程式），所產生的每個視窗都必須由應用模組提供一個行為中樞，此即所謂視窗函式（**window function**），負責接收 `::DispatchMessage()` 派送過去的訊息，並攔下某些訊息做進一步處理，不感興趣的訊息則放手讓 `::DefWindowProc()` 處理，如圖 6-20。這些 window function 必須被登錄於對應的 **window class** 之中（亦即圖 6-19 右側 `WinMain()` 內的 `RegisterClass()` 動作，不過該處未列出應有的參數）。此處所謂 **window class** 並非是個 C++ class，而是 Windows 程式設計原本就有的觀念，是 Windows 程式設計領域的一個術語，和 C++ 無關⁷。

當 application framework 被引入，其主要目的是復用性，希望為所有應用程式服務。為此，設計者希望，由框架模組事先定義好數個 window classes，代表數個經常被運用而且行為已完全確立的視窗種類，這些視窗類別（window classes，例如 `_afxWnd`，`_afxWndFrameOrView`，`_afxWndMDIFrame`，`_afxWndControlBar`，`_afxWndOleControl`）中的視窗函式指標（pointer to **window function**）一律指向

⁷ 關於 window procedure, window class, `CreateWindow()`, `RegisterClass()`，請參考 Win32 API 程式設計相關書籍，例如《Programming Windows》by Charles Petzold 或《Windows95: A Developer's Guide》by Jeffrey Richter & Jonathan Locke。

同一個視窗函式：AfxWndProc()，如圖 6-21。

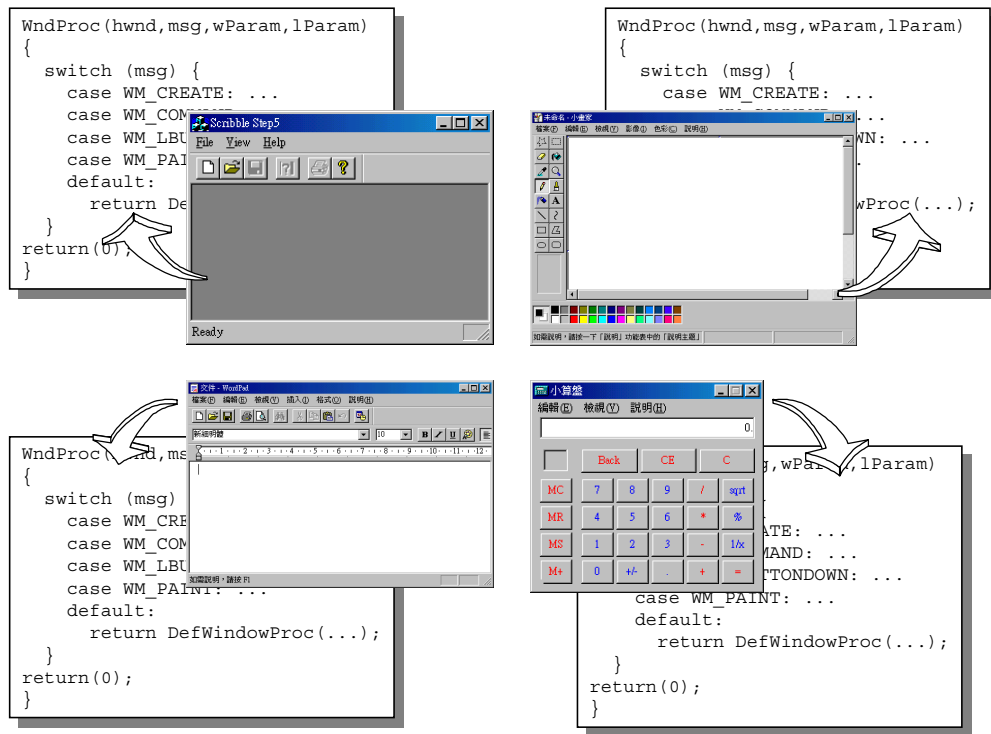


圖 6-20 每個視窗的背後，都有一個視窗函式 (window function) 作為行為中樞，決定哪些訊息該處理、該如何處理。

這麼一來，每一個 `CWnd-derived classes` 所產生的（對應的）視窗，其視窗函式就都是 `AfxWndProc()` 了，這就表示，只要 `MFC(Lite)` 在這個函式內設計好處理程序，所有 `CWnd-derived classes` 的對應視窗便有了預設的行為模式。但是，由於不同的應用程式需要不同的視窗行為，每個 `CWnd-derived objects` 必須有機會（但也可以放棄）自行決定處理哪些訊息。為了達到這一點，我們有兩種選擇。比較直觀（同時也比較 OO）的作法是，框架模組在 `CWnd` 中為每一個視窗訊息撰寫處理常式（message handler），並令它們皆為虛擬函式，於是所有 `CWnd-derived classes`（包括應用模組中的各個主視窗、子視窗…）便可依各自需求，覆寫各自感興趣的虛擬函式，達到訊息選擇和訊息處理型式的自主性。

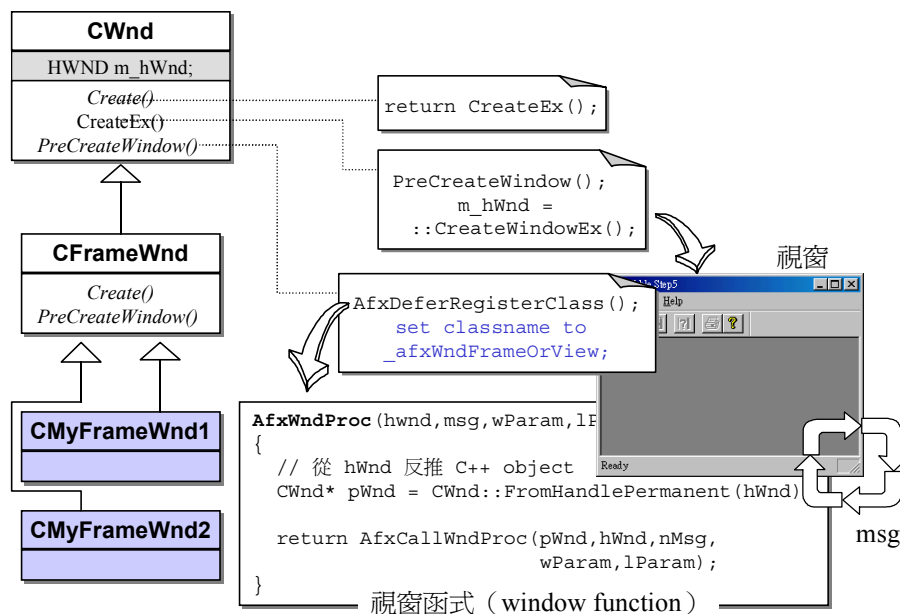


圖 6-21 爲了復用性，application framework 做出這樣的設計：每一個 CWnd-derived classes 所產生的視窗，其視窗程序都是同一個（AfxWndProc()）。不同的 CWnd-derived classes 如果想要有自己的訊息選擇權和處理權，必須使用訊息映射機制（詳下節）。圖中偽碼（虛擬碼）所示的 AfxDeferRegisterClass() 在 MFC 之中負責註冊視窗類別（Register WindowClass）。MFC Lite 主旨不在圖形介面，所以沒有模擬圖像視窗，也沒有模擬視窗類別（window class）及其註冊動作，當然也就沒有模擬 AfxDeferRegisterClass()。

這樣的設計，一如我在本章最初討論中所言，就 OO 思維而言是無可挑剔的，卻留有一絲遺憾和一道未解難題：

- 任何視窗系統都有上百個訊息，Microsoft Windows 就有三四百個訊息之多。一一爲它們設計虛擬函式，再被所有 CWnd-derived classes 一一繼承下去，會產生大量的負荷。但是視窗作業系統的訊息處理機制要求很高的效率。這是速度上的遺憾。
- 視窗訊息只能被 CWnd-derived classes 攫取，因爲只有它們才真正擁有視窗。然而在整個 application framework classes 階層體系中，並不是只讓 CWnd-derived classes 收到訊息就好。舉個例子，使用者按下 [Edit/Clear All] 表單

命令項，由於所有文件資料都被集中於 CDocument-derived object 內（這是 MVC 模型的要求，詳 6.9 節），因此如果訊息不能流至 CDocument-derived object，其他 objects 獲得這個命令訊息後將很不容易取得文件，即使能夠取得（那純粹是因為框架模組設計者發了慈悲心，開通了某種介面），也縛手縛腳不甚方便。噢，用來處理文件的表單命令，當然應該被 CDocument-derived object 攫取，才最自然，不是嗎？這是前述的虛擬函式解法所無法攻克的問題。

為此，MFC 決定採用看起來似乎不怎麼 OO 的技術，將上述的遺憾和難題畢其功於一役，那就是「訊息映射」和「訊息繞行」機制。

6.7.1 命令訊息與 CCmdTarget

爲了讓命令訊息 (WM_COMMAND) 有特立獨行的流動機會，MFC(Lite) 實現了一個所謂的 CCmdTarget，顧名思義就是「命令訊息的去處 (Command Target)」。這個 class 的原本設計與 OLE (ActiveX) 有大量關連，對於此刻我們所專注的 application framework 核心技術而言，並沒有 (也無必要) 提供太多介面，其價值純粹在於 classes 階層管理。凡能夠接收命令訊息的 classes，我們便讓它衍生自 CCmdTarget。回頭看看 MFC(Lite) classes 階層架構 (圖 6-2)，Application Architecture 和 Window Support 兩大類都是合理的命令訊息接收者，因此它們統統被置於 CCmdTarget 之下。其他如 File Services, Collections (以及未被 MFCLite 模擬的眾多 MFC classes) 都沒有收受命令訊息的必要。

下面是 CCmdTarget 的介面：

```
class CCmdTarget : public CObject
{
    DECLARE_DYNAMIC(CCmdTarget)    // 先前介紹過了
public:
    DECLARE_MESSAGE_MAP()           // 稍後介紹
    virtual ~CCmdTarget() { }
    virtual BOOL OnCmdMsg(UINT nID, int nCode); // 稍後介紹
};
```

其中 DECLARE_MESSAGE_MAP() 與訊息映射有關，OnCmdMsg() 與訊息繞送有關，都將於後續章節中介紹。

6.7.2 建立一個訊息映射表 (message map)

DECLARE-, BEGIN-, END- _MESSAGE_MAP

爲了解決虛擬函式帶來的遺憾與死角，MFC(Lite) 採用如下的訊息機制：

- 每個 CCmdTarget-derived classes 備妥專屬的表格，內含希望處理的訊息識別碼、處理常式的位址、命令項識別碼（唯命令訊息才需要）、其他相關資訊。這個表格稱爲訊息映射表（message map）。
- 所有 CCmdTarget-derived classes 的訊息映射表，都根據某種指示（詳後），串接成一個大型的、樹狀的訊息映射表（圖 6-22）。

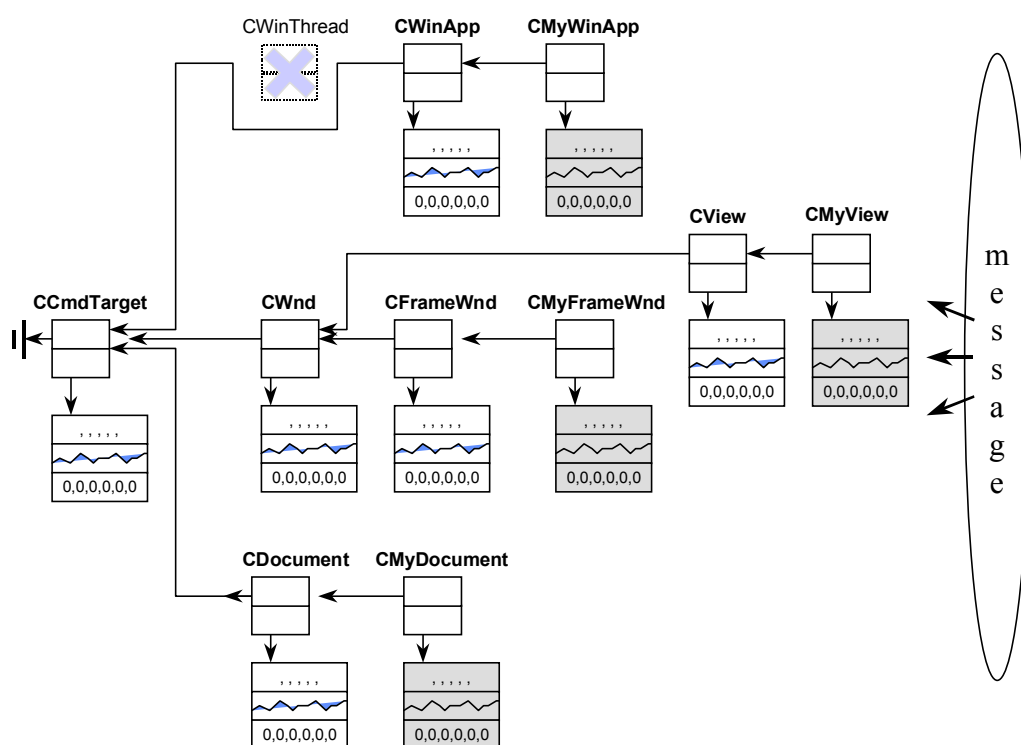
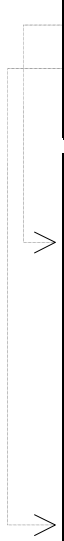


圖 6-22 大型的、全系統的、樹狀的訊息映射表。每個 class 有一個專屬的小型映射表，其中記錄訊息種類、處理常式位址、其他相關資訊；此一小型表格最後以「欄位全部爲 0」收尾。

- 框架模組提供一個訊息推動引擎（或稱為 **message pump**，訊息邦浦），根據特定的路線（詳見 6.8 節「訊息繞送」），令訊息「流」過整個訊息映射表的樹狀結構。流動過程中如果遇到吻合的表格條目（**table entry**），就表示該處打算處理此一訊息，這便呼叫其相應的訊息處理常式。

有了這樣的構想之後，我們首先要考量訊息映射表的結構設計，其次要考量如何能夠在不曝露過多繁瑣細節的情況下，讓應用模組以最方便的方式參與整個訊息映射機制 — 畢竟整個遊戲不能關起門來進行，必須考慮應用模組的參與。

我們希望，每一個希望加入訊息映射表的 **class**，都有這樣的宣告：



```

class CFoo : public CCmdTarget
{
private:
    static AFX_MSGMAP_ENTRY _messageEntries[];
    static AFX_MSGMAP messageMap;
    virtual AFX_MSGMAP* GetMessageMap() const;
};

typedef void (CCmdTarget::*AFX_PMSG)(void);

struct AFX_MSGMAP_ENTRY
{
    UINT nMessage; // windows message
    UINT nCode;    // control code or WM_NOTIFY code
    UINT nID;      // control ID (or 0 for windows messages)
    UINT nLastID;  // used for entries specifying a range of ctrl id's
    UINT nSig;     // signature type (action) or pointer to msg #
    AFX_PMSG pfn;  // routine to call (or special value)
};

struct AFX_MSGMAP
{
    AFX_MSGMAP* pBaseMessageMap;
    AFX_MSGMAP_ENTRY* lpEntries;
};
  
```

圖 6-23 表現出上述 **AFX_MSGMAP** 和 **AFX_MSGMAP_ENTRY** 的關係，至於後者六個欄位的詳細探討，請見 6.7.3 節。請注意，這些資料結構均為 **static members**，因此，不論產生了多少個 **CFoo** objects，這樣的訊息映射表永遠只需一份。嗯，非常合理。

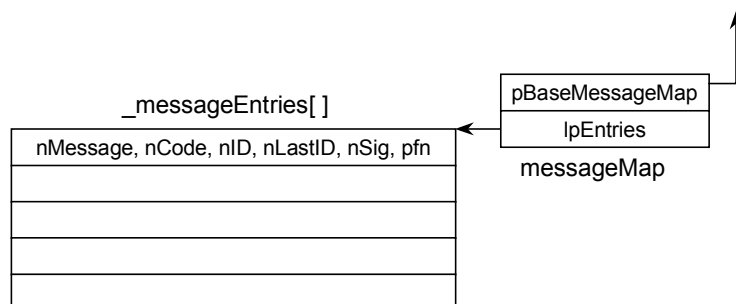


圖 6-23 訊息映射表的結構

上述作法曝露了過多細節，造成應用端的困擾。爲了避免這個缺點，MFC(Lite) 再次師法三大基礎建設，運用巨集來達到程式碼自動生成效果。首先設計一個巨集如下：

```

#define DECLARE_MESSAGE_MAP() \
    static AFX_MSGMAP_ENTRY _messageEntries[]; \
    static AFX_MSGMAP messageMap; \
    virtual AFX_MSGMAP* GetMessageMap() const;
  
```

然後令每一個需要訊息映射表的 classes，都在宣告式中如此這般地使用上述巨集：

```

class CFoo : public CCmdTarget
{
private:
    DECLARE_MESSAGE_MAP()
};
  
```

請注意，和三大基礎建設不同的是，這個巨集之內並未設立存取權級 (access level)，因此你把它放在 public: 之下，整個訊息映射表就是 public，你把它放在 private: 之下，整個訊息映射表就是 private。這不會帶來任何「正確性」問題，因為，稍後在訊息繞送機制中你將看到，取用此一訊息映射表的都是自家人（與映射表隸屬相同 class 的某個 member function），因此映射表若爲 private，沒有問題，若爲 public，更沒有問題。但 public 畢竟破壞了封裝性。或許 MFC 希望所有應用程式員都透過 VC++ 的 AppWizard 和 ClassWizard 來進行撰碼動作，在工具的協助之下也就不太容易做出越矩行爲。無論如何，就 OO 封裝觀點而言，還是選用 private 比較理想，所以 DECLARE_MESSAGE_MAP() 巨集最理想的形式應該是在最前面加一

行 `private:` 敘述句。爲了完全模擬 MFC，我並未讓 MFCLite 這麼做。

接下來是訊息映射表的填寫方式。兩個相關巨集定義如下：

```
#define BEGIN_MESSAGE_MAP(theClass, baseClass) \
    AFX_MSGMAP* theClass::GetMessageMap() const \
    { return &theClass::messageMap; } \
    AFX_MSGMAP theClass::messageMap = \
    { &(baseClass::messageMap), \
      (AFX_MSGMAP_ENTRY*) &(theClass::_messageEntries) }; \
    AFX_MSGMAP_ENTRY theClass::_messageEntries[] = \
    {

#define END_MESSAGE_MAP() \
    { 0, 0, 0, 0, AfxSig_end, (AFX_PMSG)0 } \
};
```

使用者應該在全域範疇 (global scope) 的任何位置以下列方式運用上述兩個巨集：

```
// 以下在框架模組中
BEGIN_MESSAGE_MAP(CWnd, CCmdTarget)
    ON_WM_CREATE() // 攔截並處理 WM_CREATE 訊息。後述。
END_MESSAGE_MAP()

BEGIN_MESSAGE_MAP(CFrameWnd, CWnd)
    ON_WM_CREATE()
END_MESSAGE_MAP()

BEGIN_MESSAGE_MAP(CDocument, CCmdTarget)
    ON_COMMAND(ID_FILE_SAVE_AS, OnFileSaveAs)
    ON_COMMAND(ID_FILE_SAVE, OnFileSave) // 攔截並處理 WM_COMMAND。後述
END_MESSAGE_MAP()

// 以下在應用模組中
BEGIN_MESSAGE_MAP(CMyDocument, CDocument)
    ON_COMMAND(ID_EDIT_CLEAR_ALL, OnEditClearAll)
END_MESSAGE_MAP()
```

這便形成類似圖 6-24 的訊息映射表。

請注意，做爲訊息流動的最終站，由於 `CCmdTarget` 的訊息映射表不復再有 `base class`，所以不能夠以前述巨集實現出來，必須手動完成，像這樣：

```
AFX_MSGMAP* CCmdTarget::GetMessageMap() const
{
    return &CCmdTarget::messageMap;
}
```

```

AFX_MSGMAP CCmdTarget::messageMap =
// static object initialization
{
    NULL,
    &CCmdTarget::_messageEntries[0]
};

AFX_MSGMAP_ENTRY CCmdTarget::_messageEntries[] =
// static object initialization
{
    { 0, 0, 0, 0, AfxSig_end, 0 }    // nothing here
};
// AfxSig_end 被定義為 0。稍後敘述。

```

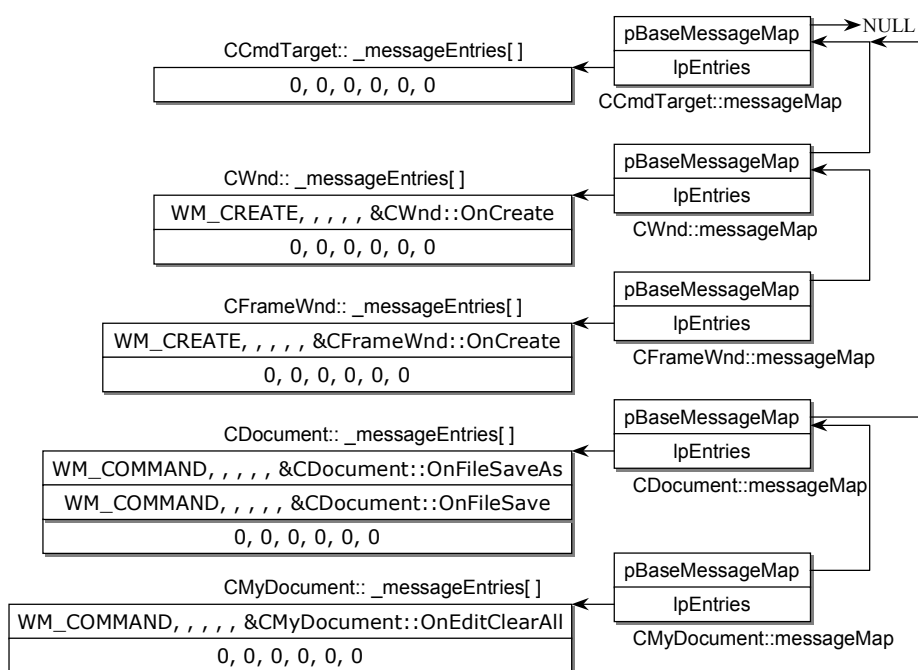


圖 6-24 訊息映射表的某種可能組成。這是 MFCLite 訊息映射表的一部分（雖然規模不大，對於模擬效果而言，已經足夠），MFC 訊息映射表則龐大得多。請注意映射表中所記錄的訊息處理常式，是個 class member function，而非一般 function，稍後另有討論。

6.7.3 訊息映射表之填寫

ON_COMMAND(), ON_WM_CREATE()...

訊息映射表中各欄位的填入，由以下巨集擔綱服務（**請注意**，以下各轉型動作採用 C-style 風格，是舊的作法；MFC7 已改採 `static_cast<>` 以尋求安全轉型。MFCLite3 源碼已改採 MFC7 新法，與此處稍有不同）：

```
// 以下定義三種 member function 型別：AFX_PMSG, AFX_PMSGW, AFX_PMSGT,
// 分別用於：CCmdTarget-, CWnd-, CWinTread-derived classes.
typedef void (CCmdTarget::*AFX_PMSG)(void);
typedef void (CWnd::*AFX_PMSGW)(void);
typedef void (CWinThread::*AFX_PMSGT)(void);

#define CN_COMMAND 0

// 定義於 afxmsg.h
#define ON_COMMAND(id, memberFxn) \
    { WM_COMMAND, CN_COMMAND, (WORD)id, (WORD)id, AfxSig_vv, \
      (AFX_PMSG)memberFxn },

#define ON_WM_CREATE() \
    { WM_CREATE, 0, 0, 0, AfxSig_is, \
      (AFX_PMSG)(AFX_PMSGW)(int (CWnd::*)(LPCREATESTRUCT))&OnCreate },

#define ON_WM_MOUSEMOVE() \
    { WM_MOUSEMOVE, 0, 0, 0, AfxSig_vwp, \
      (AFX_PMSG)(AFX_PMSGW)(void (CWnd::*)(UINT, CPoint))&OnMouseMove },

#define ON_WM_LBUTTONDOWN() \
    { WM_LBUTTONDOWN, 0, 0, 0, AfxSig_vwp, \
      (AFX_PMSG)(AFX_PMSGW)(void (CWnd::*)(UINT, CPoint))&OnButtonDown },

#define ON_WM_LBUTTONUP() \
    { WM_LBUTTONUP, 0, 0, 0, AfxSig_vwp, \
      (AFX_PMSG)(AFX_PMSGW)(void (CWnd::*)(UINT, CPoint))&OnButtonUp },

#define ON_WM_PAINT() \
    { WM_PAINT, 0, 0, 0, AfxSig_vv, \
      (AFX_PMSG)(AFX_PMSGW)(void (CWnd::*)(void))&OnPaint },

... more
```

每個巨集代表某種「訊息映射表條目 (message map entry)」的植入。面對真實世界的視窗系統，這類巨集有數百個之多。MFCLite 只模擬以上 6 個。除了 ON_COMMAND，其他巨集都不需要任何參數，其訊息處理常式的名稱是固定的（和巨集名稱相呼應，你很容易觀察出其間規律）。

pointer to member function 的型別轉換

在這裡，我們必須深入探討每一筆訊息條目的每一個欄位的目的和型別設計。尤其是第 6 欄位的型別，暗藏許多 pointer to member function 的轉型技巧。

做為一個比對標準與映射依據，訊息映射表中的條目（entries）必須提供以下欄位（請參考先前列出的 AFX_MSGMAP_ENTRY 源碼）：

- **nMessage**：訊息種類（識別碼），例如 WM_COMMAND, WM_CREATE...
- **nCode**：MFCLite 略而未用，但仍保留。
- **nID**：命令項識別碼（若為一般視窗訊息而非命令訊息，此欄為 0）
- **nLastID**：MFCLite 略而未用，但仍保留，並令它永遠等同上一欄位。
- **nSig**：一個代碼，用以表示下一欄位 pfn 的函式型態。
- **pfn**：一個 pointer to member function，型別為 AFX_PMSG，指向訊息處理常式。

訊息的流動有 4 條路線（詳見 6.8 節）。當訊息推動引擎決定了其中一條路線，就循著「由下而上，由 derived 而 base」的方向一一比對映射表中的條目。一旦訊息種類吻合（如為命令訊息，還需命令識別碼也吻合），推動引擎就直接喚起第 6 個欄位所登錄的訊息處理常式。

這樣的「比對而後呼叫」邏輯相當簡單而直觀，實作時卻頗為棘手，主要在於訊息處理常式的型別安排。第 6 欄位 pfn 的型別必須有海納百川的肚量，才能夠接納（接受指定）所有可能的訊息處理常式。這裡的困難不僅在於訊息處理常式可以是任何 CCmdTarget 衍生類別的 member functions（而你知道，CCmdTarget 可以擁有各式各樣的衍生類別），而且每一個訊息處理常式又可能擁有全然不同的參數列和回返回值。各種情況的總和，太多了，太多了。

當你面對這種困境，必須以一抵百的時候，唯一能做的就是找出最通用的型式。

Contravariance rule

由於訊息處理常式是個 member function，而且必定屬於某個 CCmdTarget-derived class，因此將第 6 個欄位設計為這樣是合理的（也算是多型的一種實現吧☺）：

```
AFX_PMSG pfn;
// pfn 實際必然是個 pointer to member function of CCmdTarget-derived class.
```

其中 `AFX_PMSG` 定義為：

```
typedef void (CCmdTarget::*AFX_PMSG)(void);
// 一個 pointer to member function 有三個構成要素：
// (1) 回返型別 (2) 所屬 class (3) 參數列
```

這麼一來，只要能夠將任何一個 member function of `CCmdTarget`-derived class 轉型為 `AFX_PMSG`，就可以被放進映射條目的第 6 欄位中。

但是一個 pointer to member function of `CCmdTarget`-derived class 不能被自動（隱式，implicitly）轉型為一個 pointer to member function of `CCmdTarget` class。這和我們過去以來獲得的多型（polymorphism）印象大異其趣：

```
CCmdTarget* pobj = new CWnd; // 這是多型的典型展現
// 多型：pointer to base class 總是能夠指向 derived class object
// 或說：pointer to derived class 總是能夠隱式轉型為 pointer to base class
```

如今面對 pointer to member function，自動（隱式）轉型的方向卻是：「a pointer to member function of base class 可被隱喻轉換為 a pointer to member function of derived class，而且兩造之參數列必須相同」。此稱為 **Contravariance rule**。

為此，我們有必要在「訊息映射條目」的植入巨集（各個 `ON_WM_x` 巨集）中，將視窗訊息的處理常式做三階段強制轉型：

- 首先將訊息處理常式轉換為 pointer to member function of `CWnd`，參數列維持不變。注意，特定的視窗訊息處理常式接受特定的引數，形式固定而無變化。`CWnd` 是所有視窗相關類別中位於階層架構最頂端者。這個轉換是 **Contravariance rule** 的逆向，因此需要顯式（explicitly）轉換。
- 將上述結果再轉換為同型函式，但令參數為 `void`。⁸
- 將上述結果再轉換為 pointer to member function of `CCmdTarget`，參數列維持不變（`void`）。

以 `ON_WM_LBUTTONDOWN()` 為例，假設 `CMyView` 使用了這個巨集，那麼上述三個轉換（轉換為一個 pointer to member function）將分別是：

⁸ 參數列（parameter list）竟然可以轉換！這在一般的 pointer to function 中是絕對不能夠的。換句話說，pointer to member function 的轉型，存在許多危險性（詳見 3.x 節）

- 將 `void(CMyView::*)(UINT, CPoint)` 轉換為 `void(CWnd::*)(UINT, CPoint)`
- 將上述結果轉換為 `void (CWnd::*)(void)`
- 將上述結果再轉換為 `void (CCmdTarget::*)(void)`

當然啦，被轉為最泛化型式 `void (CCmdTarget::*)(void)`（也就是 **AFX_PMSG**）之後，許多重要的資訊都流失了，因此將來比對成功之後，還必須將完整型別轉換回來，這就有賴「訊息映射條目」第 5 欄位 `n_Sig` 的協助（詳見 6.7.4 節）。

上述是一般視窗訊息。命令訊息又是另一種光景，比一般視窗訊息簡單一些。`ON_COMMAND` 巨集要求使用者必須明確指出命令識別碼和處理常式。命令訊息的處理常式永遠是「參數為 `void`，回返值為 `void`」，所以植入映射表時，只需一次轉換即可：

- 將 `void (CCmdTarget-derived::*)(void)` 轉換為 `void (CCmdTarget::*)(void)`。
轉換後的型別也就是 **AFX_PMSG**。

6.7.4 羅塞達石碑⁹：解開型別之謎與型別轉換之謎

讓我們暫且將訊息的推動、訊息條目的比對放在一旁（那是 6.8 節的事情）。假設某個訊息在映射表中被比對成功了，接下來應該喚起其訊息處理常式。但是該函式的完整型別已經因為上一小節所說的型別轉換，而被切割得七零八落，如何能夠正確呼叫呢？

為此我們需要一個用以記錄「訊息處理常式之原始型別」的東西以為輔助。MFC(Lite)設計了這樣一個 `enum`：

```
enum AfxSig
{
    AfxSig_end = 0,      // [marks end of message map]
    AfxSig_is,           // int (LPTSTR)
    AfxSig_vv,           // void (void)
    AfxSig_vvp,          // void (UINT, CPoint)
```

⁹ 羅塞達石碑（Rosetta Stone），1799 年拿破崙遠征埃及時，由一名官員在尼羅河口羅塞達發現。石碑是黑色玄武岩，高 114 公分，厚 28 公分，寬 72 公分，經法國學者 Jean-Francois Champollion 研究後，揭開了古埃及象形文字之謎，從此世人得以順利研讀古埃及文獻。在古埃及考古歷史上，佔有樞紐地位。

```
// ... more
};
```

每一個元素代表一種函式型別。簡單地說，每一個元素皆以 `AfxSig_` 為前綴詞，其後接續的第一個字元代表回返值型別，第二個字元起，每個字元代表一個參數型別，例如 `v` 代表 `void`，`w` 代表 `WORD (UINT)`，`i` 代表 `int`，`s` 代表 `LPTSTR`，`p` 代表 `CPoint`。這只是適用於此處目的的一種設計，不是什麼標準規範。

由於每一種訊息的處理常式都有其特定的參數個數、參數型別、回返值型別，因此 `ON_WM_` 巨集或 `ON_COMMAND` 巨集在展開程式碼的同時，就可以將特定的 `AfxSig_` 符號（編號）寫入映射條目的第 5 欄位。當訊息推動引擎成功比對了某個訊息，便可根據第 5 欄位，把原本已被極端泛化的成員函式型別 `void (CCmdTarget::*)(void)` 正確轉化為其原本型別。當然啦，這中間還需要一個 `union` 的協助，詳見圖 6-25 及書附源碼。

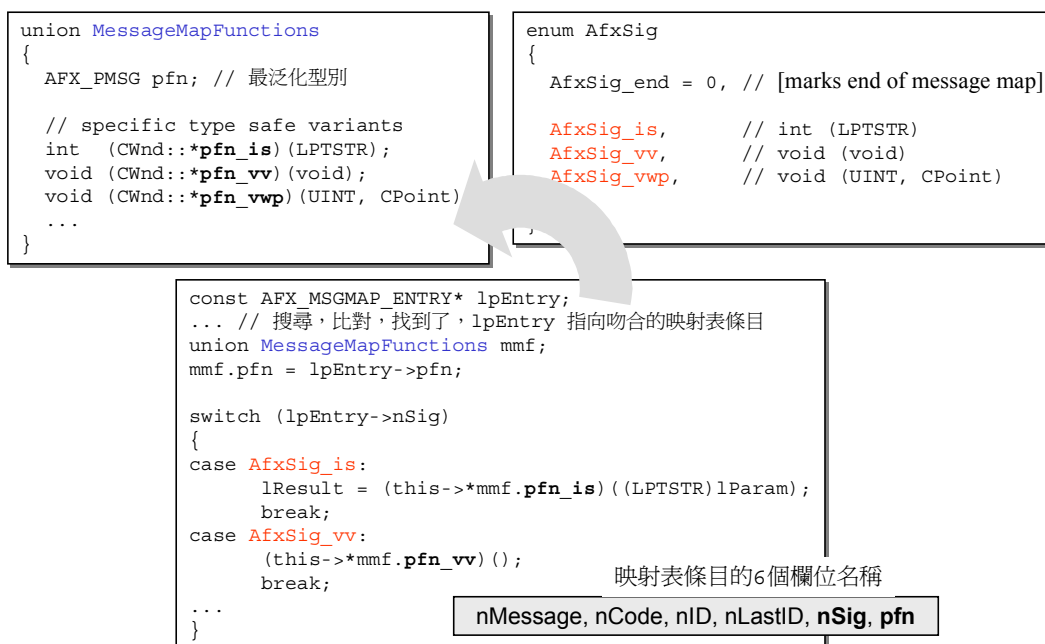


圖 6-25 透過本圖所示的 `enum` 和 `union`，我們得以將 `pointer to member function` 的「型別相關資訊」包裹起來。於是，即使經過層層轉型，仍可再轉回原型別。

6.8 訊息繞送 (message routing)

MFC(Lite) 訊息映射表並非單線串列，而是一個樹狀結構（圖 6-22, 6-24），訊息總是從葉節點流向（被推往）根節點。由於訊息總是發生於圖形介面（視窗）上，如果沒有做特別的處理，訊息將總是在 *Window Support* 系列那一條線上被推動，其他如 *Application Architecture* 系列根本沒有機會獲得訊息。然而我們早已討論過，某些命令訊息（`WM_COMMAND`）在視窗以外的 classes 中被處理才是比較自然、比較方便、比較有效率的。

我們必須讓訊息的前進路線轉彎 — 以某種特定規則轉彎！圖 6-26 便是這個特定規則。從 application framework 實作技術的角度來看，我們不必在乎為什麼定下這樣的規則（那屬於 Windows 視窗管理層次），只需知道如何實現這樣的規則。為了實現這個規則，又將大量動用虛擬函式和設計樣式 *Template Method*。

命令訊息接收物的型態	處理次序
Frame 視窗	1. View 2. Frame 視窗本身 3. app object
View	1. View 本身 2. Document
Document	1. Document 本身 2. Document Template

圖 6-26 MFC (MFCLite) 對於命令訊息 `WM_COMMAND` 的特殊處理方式

讓我對問題和解法再做一次分析。基於 6.7 節的討論，我們已經知道，由於不同機能的視窗（例如 `docFrame` 或 `View`，詳見 6.9 節）使用相同的視窗函式（`window function`），因此這些視窗所發生的訊息最終都會被送往 `AfxWndProc()`。在那裡，將會根據訊息結構（`struct MSG`）中的 `window handle` 找出對應的 `CWnd-derived object`，然後取其對應的訊息映射表，從彼處開始循著映射表的樹狀路線上溯，被誰攔住就被誰處理。

這樣的上溯路線終究只能到達 `CWnd-derived classes` 所屬的映射表，無法到達整個樹狀映射表的其他旁支。是的，視窗訊息一般用來反應視窗本身的狀態變化（移動、大小改變、重繪…等等），由 `CWnd-derived classes` 來處理，很好，但另有所謂的「命令訊息」（表單選項按下後觸發），並非用來反應視窗狀態，而是用來讓使用者對程式下達某種命令，因此它們必須由 `MFC(Lite)` 中的 *Application Architecture* 系列（如 `CWinApp`, `CDocument`, `CDocTemplate` 等）處理。然而，訊息映射表的各個樹狀旁支彼此並無通路，因此發生於視窗身上的命令訊息無法到達 `CWnd-derived` 以外的 `classes` 所設定的映射表。我們必須寫出適當的「轉轍器」，才能在適當時候讓訊息按圖 6-26 的既定規則流動。這就是所謂的訊息繞行 (message routing)。



訊息繞行機制的實作手法非常複雜，動用了多層虛擬函式，而這些虛擬函式的命名（例如 `OnWndMsg()`, `OnCommand()`, `OnCmdMsg()`）又極易給讀者帶來混亂。你在這裡停滯不前是很正常的情况。這裡的學習方式有兩種，一是大略吸收概念就好，一是徹底了解實作細節並確實追蹤其執行流程。後者雖然辛苦，一旦通過考驗，對實務的掌控當然從此脫胎換骨。如果你選擇後者，你需要超強的靜定功夫、許多個寧靜的夜晚、許多杯提神的咖啡（就別吸煙啦☺），並對虛擬函式和 `this` 指標有充分的理解。

當然，我也要提醒你，圖 6-26 的繞行路線是為總綱，實現手法不會只有一種。這裡所列是 `MFC` 的作法，如果你認為其中有值得大刀闊斧修改之處，並且你有具體的想法，動手吧，別在權威面前低頭☺。

由於訊息繞行太過繁複，6.8.1 節至 6.8.3 節皆作一個假設：假設系統並不支援 `MDI`（何謂 `MDI`？見 6.6.5 節），因此，所有視窗訊息均從 `CView` 進入，所有命令訊息均從 `CFrameWnd` 進入。直到 6.8.4 節才併入 `MDI`（也就是併入 `CMDIFrameWnd` 和 `CMDIChildWnd`）加以討論。

6.8.1 單工式的訊息傳送方式（直線上溯）

圖 6-27 顯示的便是命令訊息繞行的既定策略。此圖顯示，命令訊息（總是發生於 frame 視窗，亦即總是由 CFrameWnd 的訊息映射表出發）所流經的四條路線。每一條路線都走到底（CCmdTarget）而後轉至下一條路線，換言之所謂「訊息繞行」就是這四條路線的逐一上溯。

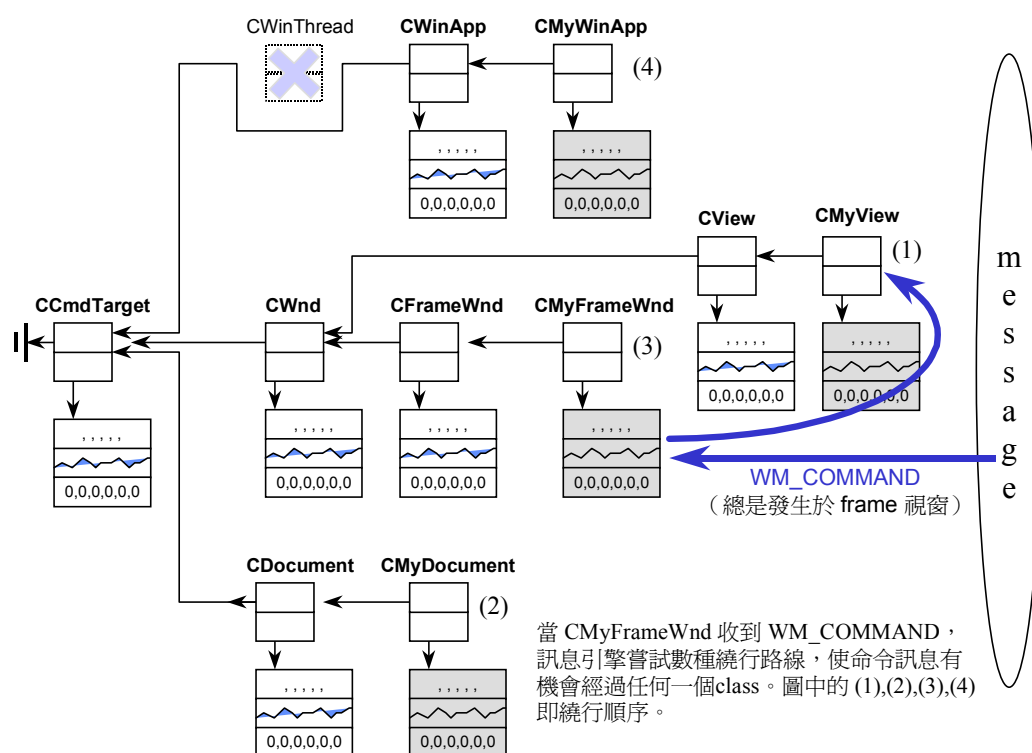


圖 6-27 命令訊息發生於 Frame 視窗後，有四條流動路線。每一條路線都走到底而後（如果未被攔截的話）轉至下一條路線。

現在，讓我們看看直線上溯的技法。

AfxWndProc() 是訊息繞行的起點。經過「一些動作」之後（MFCLite 不支援圖形介面，因此 MFC 裡頭的「一些動作」在此被我做了大量簡化），訊息流到轉轍站

的起點：`CWnd::WindowProc()`，開始流動繞行。如果最終沒有任何 classes 對此訊息感興趣，它就被交給 `::DefWindowProc()` 處理 — 這正是 Win32 API 撰碼風格（或稱 Windows SDK 風格）的重現¹⁰。MFCLite 並未模擬視窗實體，所以它的 `::DefWindowProc()` 裡頭什麼都不做。

下面便是訊息繞行起點 `AfxWndProc()` 和訊息轉轍站起點 `CWnd::WindowProc()` 的源碼：

```

LRESULT AfxWndProc(HWND hWnd, UINT nMsg,
                  WPARAM wParam, LPARAM lParam)
// 訊息繞行起點
{
    CWnd* pWnd = CWnd::FromHandlePermanent(hWnd); //hWnd 反推 C++物件
    return AfxCallWndProc(pWnd, hWnd, nMsg, wParam, lParam);
}

LRESULT AfxCallWndProc(CWnd* pWnd, HWND hWnd, UINT nMsg,
                      WPARAM wParam, LPARAM lParam)
// 輔助函式
{
    // 授權 (委派) 至 object 的 WindowProc()
    return pWnd->WindowProc(nMsg, wParam, lParam); // (A)
}

LRESULT CWnd::WindowProc(UINT nMsg, WPARAM wParam, LPARAM lParam)
{
    LRESULT lResult = 0;
    if (!OnWndMsg(nMsg, wParam, lParam, &lResult))
        lResult = DefWindowProc(nMsg, wParam, lParam);
    return lResult;
}

LRESULT DefWindowProc(HWND hWnd, UINT nMsg,
                      WPARAM wParam, LPARAM lParam)
{
    ... // 詳見 6.11.2 節
}

```

¹⁰ Windows 規定，任何訊息都必須被處理，如果你對它不感興趣，你得交給 Win32 API `::DefWindowProc()` 去處理，見圖 6-19 右下角視窗程序的最下一行。這是因為許多訊息的發生是用來表現視窗的狀態變化，如果你對它不感興趣，而又把它吃掉了，沒有交給 `::DefWindowProc()`，Windows 系統就無法對這些訊息做出正確回應。

這裡必須先說明我對視窗的模擬手法。MFCLite 無法讓你看到真正的視窗形貌，換言之有視窗之實而無視窗之形。每當 MFCLite 程式（不論是框架模組或應用模組）產生一個視窗，我就把它記錄在一個 map 內（詳見 6.6.4 節）¹¹，這個 map 的每個元素有兩個欄位，第一欄位是 window handle，第二欄位是該 handle 所對應的 CWnd-derived object。如此一來任何時候我獲得一個 window handle，很容易就可以從這個 map 中獲得其對應的 C++ object。只要你想將工作重心從視窗系統的基本要素「視窗(window handle)」轉移到物件導向中(視窗所對應)的 C++ object，上述的 window object → C++ object 就是必然要做的一步。

第二個需要說明的是上述源碼中的 (A) 式所喚起的函式。WindowProc() 是個虛擬函式，因此究竟 CWnd-derived classes 中哪一個 member function 被喚起，端視當時的 pWnd 而定。然而雖然 WindowProc() 被設計為虛擬函式，事實上幾乎沒有人會去覆寫它，框架模組不會，應用模組更不會（我們實在沒有必要改變既有的訊息繞行演算法 — 那是高難度高危險性的行為，行動之前你得三思，並評量自己的斤兩），因此被喚起的幾乎唯一可能就是 CWnd::WindowProc()。圖 6-28 展示 MFC(Lite) classes 之中與訊息繞行機制相關的所有 member functions。當你閱讀本章或書附源碼，此圖可帶來很大幫助。

回頭看看上一頁的源碼。CWnd::WindowProc() 所呼叫的 OnWndMsg() 也是個虛擬函式，然而基於上述相同的道理，幾乎沒有人會去覆寫它 — 框架模組不會，應用模組更不會。因此被喚起的是其唯一實作 CWnd::OnWndMsg()。這個函式首先判斷訊息是否為命令訊息，如果不是，就直線上溯。以下是 CWnd::OnWndMsg() 的源碼，其中的訊息比對和比對完成後的呼叫動作，以及呼叫前的型別轉換動作，你應該已經印象深刻了 — 6.7 節的訊息映射已經介紹過這些概念。

¹¹ 真實視窗系統中其實也存在有視窗管理所需的各種資料結構（通常是串列，list），例如用來記錄目前所有的視窗及其父視窗（parent）和擁有者（owner）。這些知識可以從 Windows 系統層面的書籍獲得。關於視窗管理，我看過寫得最好的一本書是《Windows Internals》，by Matt Pietrek, Addison Wesley, 1993.

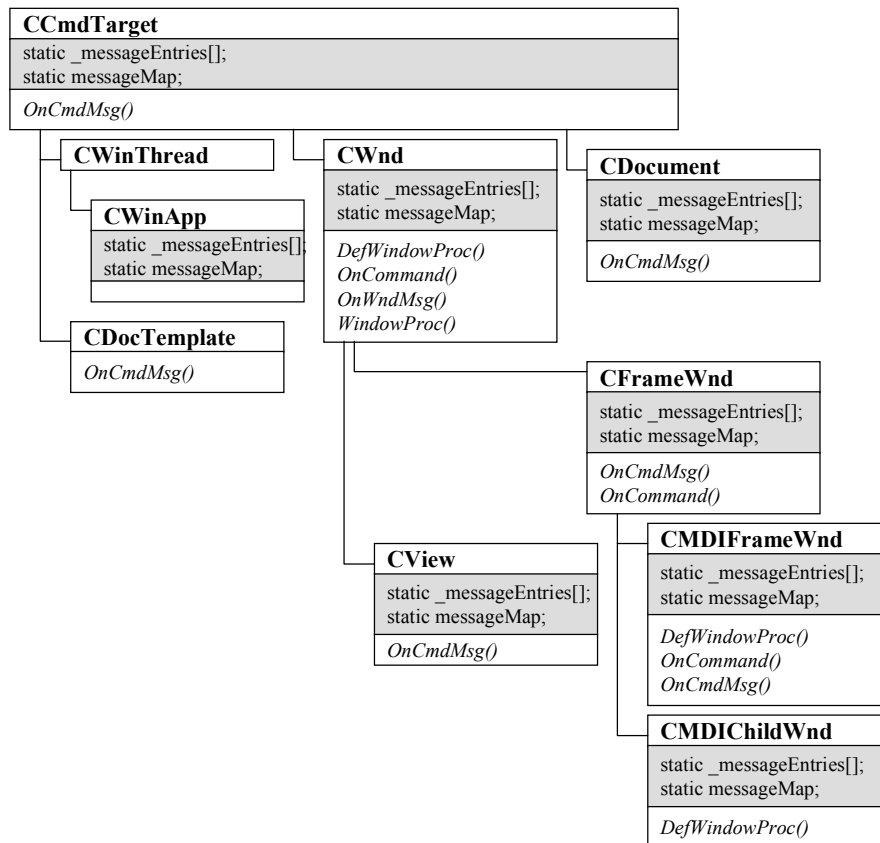


圖 6-28 MFC(Lite) classes 之中與訊息繞行機制相關的所有 members。本圖遵循 UML 表示法，以斜體表現虛擬函式。這些虛擬函式幾乎不可能在應用模組中被覆寫（除非你有特殊的需求），因為訊息繞行機制是 MFC(Lite) 的「根本大法」。

```

BOOL CWnd::OnWndMsg(UINT nMsg, WPARAM wParam,
                    LPARAM lParam, LRESULT* pResult)
{
    LRESULT lResult = 0;
    if (nMsg == WM_COMMAND) // 判知是命令訊息，需另做處理
        return OnCommand(wParam, lParam);

    // 這裡應該再檢查是否為 WM_NOTIFY，本處略。

    // 既非 WM_COMMAND 也非 WM_NOTIFY，那麼便是一般視窗訊息，採直線上溯法。
    AFX_MSGMAP* pMessageMap = GetMessageMap();
    const AFX_MSGMAP_ENTRY* lpEntry;
  
```

```

// MFC 維護了一個 cache，使速度加快。本例略。
for (; pMessageMap != NULL;
      pMessageMap = pMessageMap->pBaseMessageMap)
{
    if ((lpEntry = AfxFindMessageEntry(pMessageMap->lpEntries,
                                       nMsg, 0, 0))
        != NULL)
        goto LDispatch; // 找到了，跳開迴圈。
}
return FALSE; // 直線上溯，都沒找到。

LDispatch: // 找到了
union MessageMapFunctions mmf;
mmf.pfn = lpEntry->pfn;

switch (lpEntry->nSig) // 本例只處理以下數種 msg handler 型態
{
    case AfxSig_vv:
        (this->*mmf.pfn_vv)();
        break;
    case AfxSig_vwp:
        {
            CPoint point((DWORD)lParam);
            (this->*mmf.pfn_vwp)(wParam, point);
            break;
        }
    case AfxSig_is:
        lResult = (this->*mmf.pfn_is)((LPTSTR)lParam);
        break;
}

if (pResult != NULL)
    *pResult = lResult;
return TRUE;
}

```

以上，MFCLite 只模擬三種型式的訊息處理常式：AfxSig_vv, AfxSig_vwp, AfxSig_is，其意義已於 6.7.3 節介紹過。其中 AfxSig_vwp 主要用於滑鼠相關訊息的處理常式，所以需要準備一個 CPoint object 做為它的引數。

如果 CWnd::OnWndMsg() 判知流入的訊息是個命令訊息，它就不能貿貿然直線上溯；此時必須有流動方向上的決策，這個決策被設計於 OnCmdMsg()。但由於工作分層的需求，MFC(Lite) 並不在 OnWndMsg() 內直接呼叫 OnCmdMsg()，而是在這兩個執行區塊之間再安插一個 OnCommand() 函式。

6.8.2 揭露均等的命令訊息 (WM_COMMAND) 繞送方式

延續上一節討論，兩個 `OnCommand()` 被實作於 `CWnd` 和 `CFrameWnd` 內，下面是其源碼。`CFrameWnd::OnCommand()` 最終呼叫了 `CWnd::OnCommand()`，而後者又把轉轍決策權交給 `OnCmdMsg()` 統籌處理：

```

> BOOL CWnd::OnCommand(WPARAM wParam, LPARAM lParam)
{
    UINT nID = LOWORD(wParam);
    int nCode = HIWORD(wParam);

    // 這裡首先檢查 nID 所對應的 menu item 是否被 disable。會用到 lParam。
    // 如果未曾被 disable，才開始訊息的繞行。
    // 本例略。

    nCode = CN_COMMAND; // 定義於 afxmsg.h
    return OnCmdMsg(nID, nCode);
    // CWnd 並未覆寫 CCmdTarget::OnCmdMsg()，
    // 所以以上呼叫的是 CFrameWnd::OnCmdMsg() 或 CView::OnCmdMsg()
    // OnCommand() 是標準的 Template Method 手法。
}

```

```

    BOOL CFrameWnd::OnCommand(WPARAM wParam, LPARAM lParam)
    {
        // MFC 在這裡有一些關於 help mode 的處理。MFCLite 略。

        // 視同一般命令訊息來繞行
        return CWnd::OnCommand(wParam, lParam);
    }

```

> 轉轍器：OnCmdMsg()

`OnCmdMsg()` 是個十分標準的 **Template Method** 手法：「鋪設好演算法骨幹，並將未定細節留給 sub-type」。這裡的未定細節就是「轉轍決策」，因為圖 6-26 已經告訴我們，不同的 class 有不同的轉轍決策。頂層操控者 `CFrameWnd::OnCmdMsg()` 唯一確定的就是，它必須依序走過圖 6-26 所示的三條策略路線。

稍後我們可以從各個 `class::OnCmdMsg()` 清楚看出，它們正是實踐了圖 6-26 的訊息流動規則。注意，程式註解中所謂「唧送通過 Cxxx 自身」，意思是該訊息將循著 Cxxx 的映射表直線上溯。直線上溯的實作手法並不因 class 的不同而有不同，因此，從 OO 角度觀之，很適合把它規劃在 base class 的虛擬函式裡頭（本例為

CCmdTarget::OnCmdMsg())。這是因為，base class 的虛擬函式的實作內容，就物理意義而言相當於「derived classes 內的相應虛擬函式的預設行為」(2.x 節)。

圖 6-29 顯示參與「訊息繞行」的所有函式的喚起次序。

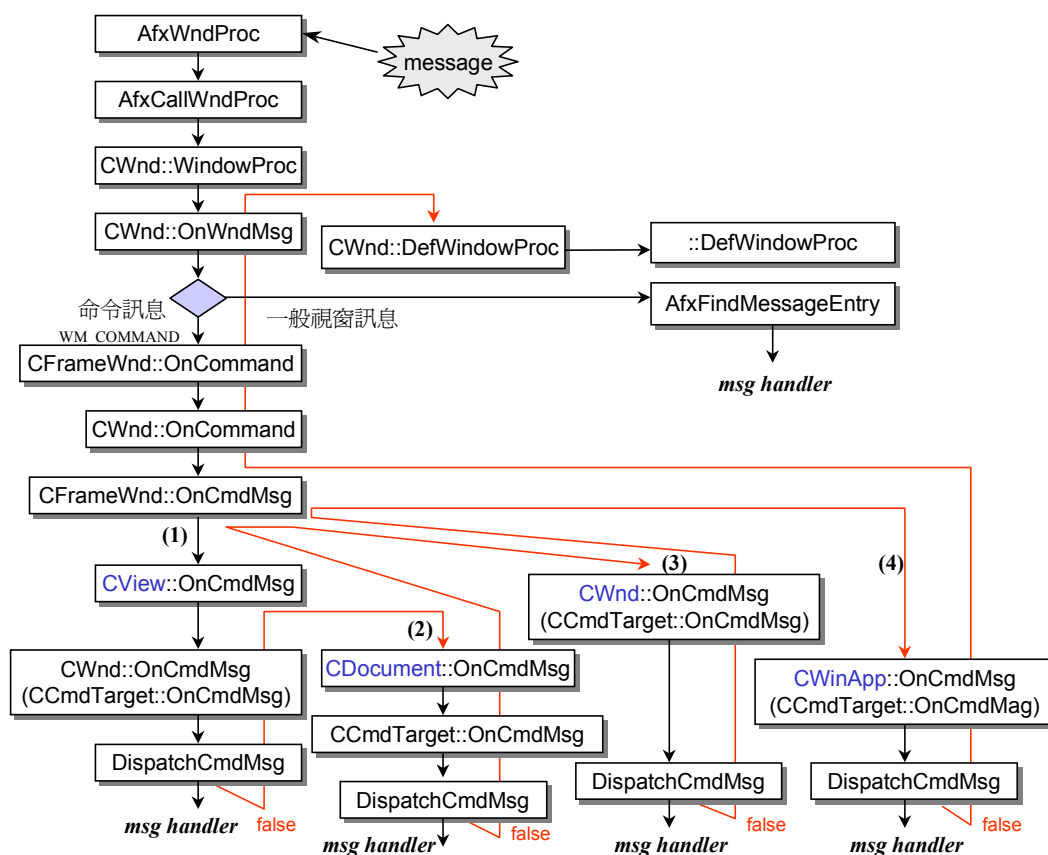


圖 6-29 參與「訊息繞行機制」的所有函式 (MDI 除外) 的喚起次序。最上方的 AfxWndProc() 可由圖 6-16 最下方接續過來。本圖標示的許多明確函式，其實是 this 指標在明確情境 (context) 下的代換結果。換句話說在不同的情境下，同一套程式碼可能引發不同的呼叫 (動態繫結之故)。如果考慮 MDI (6.8.4 節)，情況將遠比上圖複雜。本圖只是 MDI 情境下「訊息繞行」的一個部分子集。

下面就是 OnCmdMsg 虛擬函式在各個 CCmdTarget-derived class 中的實作內容。其中所呼叫的 AfxFindMessageEntry() 和 _AfxDispatchCmdMsg() 分別負責直線上溯行為中的「訊息比對」和「訊息常式呼叫」，實作手法與上一節的 CWnd::OnWndMsg() 大致相同，詳見書附源碼。_AfxDispatchCmdMsg() 在 MFC 中被設計為 static 全域函式，為的是將它侷限於實作檔範圍內 (cmdtarg.cpp)，也因此才給予名稱前置詞 '_'。MFCLite 已將所有實作檔集中於 mfclite.cpp，但仍設為 static，限制它不能被外界 (應用模組) 呼叫。



```

BOOL CFrameWnd::OnCmdMsg(UINT nID, int nCode)
{
    // 先從 active view 開始啣送 (pump) (A)
    CView* pView = GetActiveView();
    if (pView->OnCmdMsg(nID, nCode))
        return TRUE;

    // 其次，啣送通過 frame 自身 (B)
    if (CWnd::OnCmdMsg(nID, nCode)) // CWnd 並未覆寫 OnCmdMsg(), 所以
        return TRUE;                // 喚起 CCmdTarget::OnCmdMsg()

    // 最後啣送通過 app (C)
    CWinApp* pApp = AfxGetApp();
    if (pApp->OnCmdMsg(nID, nCode)) // CWinApp 並未覆寫 OnCmdMsg(), 所
        return TRUE;                // 以喚起 CCmdTarget::OnCmdMsg()

    return FALSE;
}

BOOL CView::OnCmdMsg(UINT nID, int nCode)
{
    // 首先，啣送通過 view 自身 (A-1)
    if (CWnd::OnCmdMsg(nID, nCode)) // CWnd 並未覆寫 OnCmdMsg(), 所以
        return TRUE;                // 喚起 CCmdTarget::OnCmdMsg()

    // 其次，啣送通過 document (A-2)
    BOOL bHandled = FALSE;
    bHandled = m_pDocument->OnCmdMsg(nID, nCode);
    return bHandled;
}

BOOL CCmdTarget::OnCmdMsg(UINT nID, int nCode)
{
    // 直線上溯，看看會不會被攔截。
    const AFX_MSGMAP* pMessageMap = GetMessageMap(); // 注意 this

```

```

const AFX_MSGMAP_ENTRY* lpEntry;
UINT nMsg = WM_COMMAND; // 有待考慮
// MFC 維護 cache，使速度加快。此處略
for (; pMessageMap != NULL;
      pMessageMap = pMessageMap->pBaseMessageMap)
{
    if ((lpEntry = AfxFindMessageEntry(pMessageMap->lpEntries,
                                       nMsg, nCode, nID))
        != NULL)
    {
        // 找到了，設法呼叫之。
        return ::_AfxDispatchCmdMsg(this, nID, nCode,
                                     lpEntry->pfn, lpEntry->nSig);
    }
    return FALSE; // 直線上溯，沒找到。
}

BOOL CDocument::OnCmdMsg(UINT nID, int nCode)
{
    // 首先，嚕送通過 document 自身 (A-2-1)
    if (CCmdTarget::OnCmdMsg(nID, nCode))
        return TRUE;

    // 其次，嚕送通過 doc-template (A-2-2)
    if (m_pDocTemplate != NULL &&
        m_pDocTemplate->OnCmdMsg(nID, nCode))
        return TRUE;

    return FALSE;
}

BOOL CDocTemplate::OnCmdMsg(UINT nID, int nCode)
{
    BOOL bReturn;

    // 嚕送通過 doc-template 自身 (A-2-2-1)
    bReturn = CCmdTarget::OnCmdMsg(nID, nCode);
    return bReturn;
}

```

CCmdTarget::OnCmdMsg() 內採用了標準的 **Template Method** 手法：「將撰寫當時無法掌握的細節留給 sub-types 執行」。此處「撰寫當時無法掌握的細節」就是：取得 xxx 自身映射表，做為直線上溯的起點。是的，「xxx 自身」有可能是應用模組的 class object 如 CMyFrameWnd, CMyView，也有可能是框架模組的 class object 如某些標準控件（它們不會被應用模組再衍生新的 sub-class，而是被應用模組直接拿來使用。MFCLite 並未模擬這類 classes）。CCmdTarget::OnCmdMsg() 一開

始所呼叫的 `GetMessageMap()` 是個虛擬函式，了解 **Template Method** 手法的人這便發出了會心的微笑(圖 6-30)。◦`GetMessageMap()` 以巨集 `DECLARE_MESSAGE_MAP()` 完成宣告，並以巨集 `BEGIN_MESSAGE_MAP()` 完成定義。

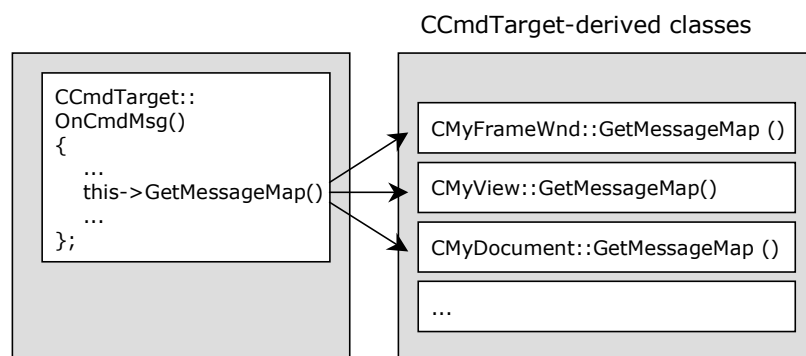


圖 6-30 Template Method 的再次實現。注意，源碼中呼叫 `GetMessageMap()` 也就是呼叫 `this->GetMessageMap()`，這在 2.x 節和 3.x 節已經說過。

追蹤訊息繞行機制的整個函式呼叫歷程，鮮有不頭昏腦脹者。最容易陷入的一個陷阱是，其間似乎有許多相同的呼叫動作，我們很多人既不清楚其語法差別，也不清楚其語意上的企圖。是的，回頭看看前數頁的 `class::OnCmdMsg()` 源碼，你會發現，其中的 (B)，(C)，(A-1)，(A-2-1)，(A-2-2-1) 五個式子統統喚起同一個函式 `CCmdTarget::OnCmdMsg()`，其間到底有什麼差別呢？

this 指標 觀念大回顧

首先我要告訴你，在這個設計中，喚起 `CCmdTarget::OnCmdMsg()`，目的就是要取「目前這個 object」所隸屬之 class 的訊息映射表起點，並從該點開始直線上溯。

「目前這個 object」在哪裡呢？就在 member function 的 `this` 指標所指處。`this` 指標是什麼意義？3.x 節說它是喚起某一 member function 的「始作俑者」。因此，以下 (B) (C) 兩式的意義並不相同：

```

BOOL CFrameWnd::OnCmdMsg(UINT nID, int nCode)
{
    ...
    // 其次，卿送通過 frame 自身 (B)
    // 以下將目前的 this 指標傳給 CCmdTarget::OnCmdMsg()，作為其 this 指標。
    if (CWnd::OnCmdMsg(nID, nCode))

```



```

        return TRUE;

// 最後啣送通過 app (C)
CWinApp* pApp = AfxGetApp();
// 以下將 pApp 傳給 CCmdTarget::OnCmdMsg() 作為其 this 指標。
if (pApp->OnCmdMsg(nID, nCode))
    return TRUE;
}

```

不同的 this 指標，完全影響了上述所喚起的 `CCmdTarget::OnCmdMsg()` 的行為結果，因為後者一開始就這麼做：

```

BOOL CCmdTarget::OnCmdMsg(UINT nID, int nCode)
{
    const AFX_MSGMAP* pMessageMap = GetMessageMap(); // 注意 this

```

上一行相當於：

```

    const AFX_MSGMAP* pMessageMap = this->GetMessageMap();

```

因此，不同的 this 指標，取得的訊息映射表並不相同，也就決定了不同的「訊息上溯路線」。上述 (B) (C) 兩個動作將分別取得圖 6-27 的 (3) (4) 兩個起點。

6.8.3 適當的訊息攔截點

有了上述的訊息映射機制和訊息繞行機制，任何 `CCmdTarget-derived classes` 只要這麼做，就可以將訊息攔截到自己手上：

```

// 以下發生於框架模組 (mfclite.cpp)
BEGIN_MESSAGE_MAP(CWnd, CCmdTarget)
    ON_WM_CREATE()
END_MESSAGE_MAP()

BEGIN_MESSAGE_MAP(CFrameWnd, CWnd)
    ON_WM_CREATE()
END_MESSAGE_MAP()

BEGIN_MESSAGE_MAP(CDocument, CCmdTarget)
    ON_COMMAND(ID_FILE_SAVE_AS, OnFileSaveAs)
    ON_COMMAND(ID_FILE_SAVE, OnFileSave)
END_MESSAGE_MAP()

BEGIN_MESSAGE_MAP(CView, CWnd)
    ON_WM_CREATE()
END_MESSAGE_MAP()

```

```

BEGIN_MESSAGE_MAP(CWinApp, CCmdTarget)
END_MESSAGE_MAP()

// 以下發生於應用模組 (mfclapp.cpp)
BEGIN_MESSAGE_MAP(CMyWinApp, CWinApp)
    ON_COMMAND(ID_FILE_NEW, CWinApp::OnFileNew)
    ON_COMMAND(ID_FILE_OPEN, CWinApp::OnFileOpen)
    ON_COMMAND(ID_APP_ABOUT, OnAppAbout)

    ON_COMMAND(ID_APP_HOTKEYHELP, OnAppHotKeyHelp) // 6.x 節
END_MESSAGE_MAP()

BEGIN_MESSAGE_MAP(CMyFrameWnd, CFrameWnd) // 稍後加入 MDI，這裡要改。
    ON_WM_CREATE() // 見書附源碼
END_MESSAGE_MAP()

BEGIN_MESSAGE_MAP(CMyDocument, CDocument)
    ON_COMMAND(ID_EDIT_CLEAR_ALL, OnEditClearAll)
END_MESSAGE_MAP()

BEGIN_MESSAGE_MAP(CMyView, CView)
    ON_WM_LBUTTONDOWN()
    ON_WM_LBUTTONUP()
    ON_WM_MOUSEMOVE()
END_MESSAGE_MAP()

```

注意，不可以因為某些 `BEGIN-`, `END-` `_MESSAGE_MAP` 巨集組中沒有任何 `ON_x` 巨集，就將它們拿掉。即使沒有任何 `ON_x` 巨集，`BEGIN-`, `END-` `_MESSAGE_MAP` 巨集組仍然是有作用的，它會架構起訊息映射表的連貫性（詳見 6.7.2 節）。

既然訊息可以流經所有 `classes` 的訊息映射表，那麼「哪個訊息該安排在哪裡處理」這一議題有沒有意義呢？唔，首先，上一句話並不精確，應該說「命令訊息可以流經所有 `classes` 的訊息映射表」，一般視窗訊息則只是直線上溯。回到「該在哪裡處理哪個訊息」這一題目來，請注意，應用程式所在意的終端用戶動作，不外乎是 `view` 視窗中的滑鼠操作（滑鼠移動啦、左右鍵按下放開啦），以及表單（`menu`）上的點選動作。因此，應用模組的一般設計通則是，在 `CMyView` 之內處理滑鼠訊息，在 `CMyDocument` 之內處理與文件相關的命令（例如 `Clear All` 或 `Select All`）。至於其他命令，你認為和 `view`, `document`, `docFrame`, `winApp` 四者之中哪一個比較有關係，你就在那個 `classes` 中設立訊息處理條目，把訊息攔到自己手上。

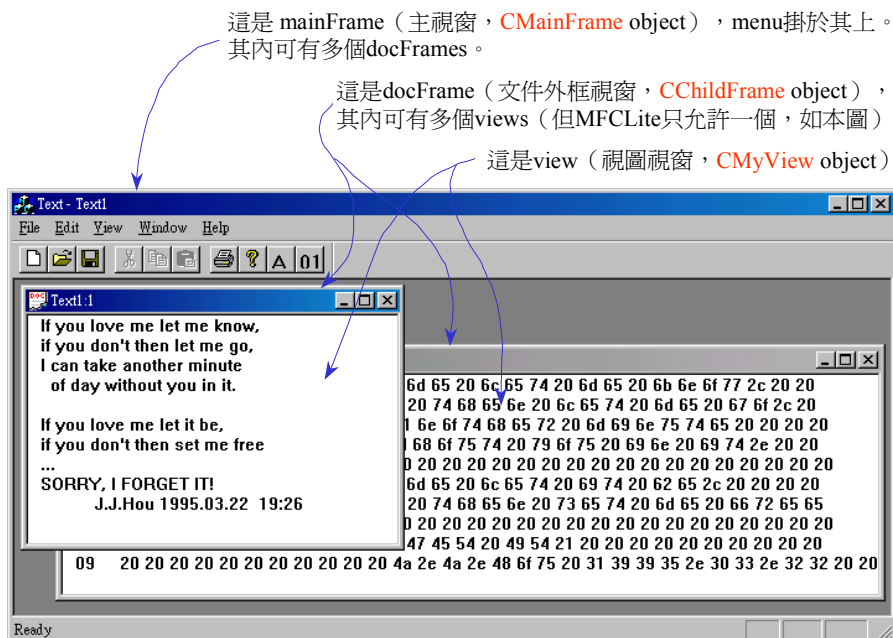
6.8.4 加個 MDI 之後

6.8.11 節和 6.8.2 節關於「訊息繞行」的討論中，所有函式動作都由 `AfxWndProc()` 出發，並假設命令訊息來自 `CFrameWnd`（精確地說應該是來自應用模組端的 `CMyFrameWnd`）。彼時曾經假設 `MFCLite` 系統尚未支援 MDI 功能，所以上述假設是合理的。

MDI 風貌是 `MFC(Lite)` 最終最完整的 UI 支援。下面是 `MFC(Lite)` 及其應用程式中四個與 MDI 相關的 classes，它們在程式中的用途如下圖所示：

```
// in application framework
class CMDIFrameWnd : public CFrameWnd;
class CMDIChildWnd : public CFrameWnd;

// in application (通常採用以下命名)
class CMainFrame : public CMDIFrameWnd;           // 用於主視窗
class CChildFrame : public CMDIChildWnd;           // 用於 docFrame 視窗
```



一旦加入這些 MDI classes，MFC 不能不讓它們有機會處理各種訊息，因為真實世界中的 MDI 視窗有自己的訊息預設處理函式 (::DefFrameProc()) 和特殊的 window class (_afxWndMDIFrame)；MFC 也不能不讓 MDI classes 有機會參與實際視窗的誕生，因為真實世界中的 MDI 視窗有自己獨特的三層視窗組合 (Frame-Client-Child)¹²。因此，CMDIFrameWnd 應該覆寫以下和訊息繞行及視窗生成有關的虛擬函式：

```
// 以下是 MFCLite 版本；其中少部分函式參數做了簡化（比之 MFC 而言）。
class CMDIFrameWnd : public CFrameWnd
{
    DECLARE_DYNCREATE(CMDIFrameWnd)
public:
    virtual BOOL LoadFrame(const string& nIDResource,
                           CCreateContext* pContext=NULL);
    virtual BOOL OnCmdMsg(UINT nID, int nCode);
    virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
protected:
    virtual LRESULT DefWindowProc(UINT nMsg, WPARAM wParam,
                                   LPARAM lParam);
    virtual BOOL OnCommand(WPARAM wParam, LPARAM lParam);
    void OnWindowNew();           // 稍後敘述
    DECLARE_MESSAGE_MAP()
};
```

在 OnCommand() 和 OnCmdMsg() 中，你會看到非常詭譎的行為：

```
BOOL CMDIFrameWnd::OnCommand(WPARAM wParam, LPARAM lParam)
{
    // 途徑 1
    // 首先傳送到 MDI child - 稍後會透過 OnCmdMsg 再傳送一次
    CMDIChildWnd* pActiveChild = MDIGetActive();

    if (pActiveChild != NULL &&
        AfxCallWndProc(pActiveChild, pActiveChild->m_hWnd,
                       WM_COMMAND, wParam, lParam) != 0)
        return TRUE; // 表示已由 child 處理完畢。

    // 途徑 2-1
    // 透過正常機制 (MDI child 或 frame) 來處理
```

¹² 關於 MDI 的種種底部構造，請參考 Windows API (SDK) 程式設計書籍，例如《Programming Window》by Charles Petzold，或《Windows95: A Developer's Guide》by Jeffrey Richter & Jonathan Locke。

```

    if (CFrameWnd::OnCommand(wParam, lParam))
        return TRUE;
    return FALSE; // 沒有獲得處理
}

BOOL CMDIFrameWnd::OnCmdMsg(UINT nID, int nCode)
{
    // 途徑 2-2
    // 首先嚮送通過 active child
    CMDIChildWnd* pActiveChild = MDIGetActive();
    if (pActiveChild != NULL)
    {
        if (pActiveChild->OnCmdMsg(nID, nCode))
            // this 將是一個 CMDIChildWnd-derived obj
            return TRUE;
        // 如果 active child 是個 CMDIChildWnd，由於它並未覆寫
        // OnCmdMsg()，所以上式喚起的是 CFrameWnd::OnCmdMsg()。
    }

    // 途徑 3
    // 通過 normal frame
    return CFrameWnd::OnCmdMsg(nID, nCode);
    // this 將是個 CMDIFrameWnd-derived obj
}

```

歸納可得其行為模式如下。從 `mainFrame` 進來的訊息（必為命令訊息），有三條繞送途徑：

- 途徑 1，透過 `AfxCallWndProc()` 繞送訊息，呼叫歷程一如圖 6-29；該圖所喚起之各個成員函式的 `this` 指標將是上述 `Afx_` 函式的第一引數 — 本例情境下將是一個 `CMDIChildWnd-derived` object。如果經此歷程，命令訊息未被處理，進入途徑 2。
- 途徑 2-1，透過 `CFrameWnd::OnCommand()` 傳送，夾帶而去的 `this` 指標正是目前的 `this` 指標 — 本例而言是個 `CMDIFrameWnd-derived` obj。由於 `this` 之故，圖 6-29 進入其路線(1) 之前喚起的函式是 `CMDIFrameWnd::OnCmdMsg()`（源碼如上）。
- 途徑 2-2，`CMDIFrameWnd::OnCmdMsg()` 取得 `active` MDI-child，並據以喚起 `OnCmdMsg()`。由 `MFC(Lite)` 繼承體系觀之，由於 `CMDIChildWnd` 並未覆寫 `OnCmdMsg()`，因此上述行為必然喚起 `CFrameWnd::OnCmdMsg()`，並夾帶一個指向 `CMDIChildWnd-derived` obj 的 `this` 指標。如果經此歷程，命令訊息仍然未被處理，進入途徑 3。
- 途徑 3，透過 `CFrameWnd::OnCmdMsg()` 傳送，夾帶的 `this` 指標指向一個

CMDIFrameWnd-derived obj。

這些路徑展開來雖有不少反覆走過，卻能夠最完整地考量每一個該走訪的角落。在 MFCLite 這麼「相對於 MFC 而言十分簡陋」的情況下，也許你可以將上述整個訊息繞行機制更形簡化，但是在真正 application framework 如 MFC 中，恐怕有許多細微之處是簡化不得的。

this 不但影響整個訊息繞行歷程之中喚起的函式，也影響它們（那些函式）所取得的訊息映射表（一如 6.8.2 節末尾所述）。試看這種情況：

```
// MFC(Lite)內建一個[Window/New]表單命令，並在主視窗攔截該命令訊息。
// 很合理的攔截地點，因為，欲針對 active document type 產生一個新視窗，
// 必須先知道 active MDI-child，而取 active MDI-child 的最方便地點
// 就是在 mainFrame 內。
BEGIN_MESSAGE_MAP(CMDIFrameWnd, CFrameWnd)
    ON_COMMAND(ID_WINDOW_NEW, OnWindowNew)
END_MESSAGE_MAP()
```

當命令 ID_WINDOW_NEW 發生，只有在前述第三條路徑上才能正確取得訊息映射條目而正確喚起 CMDIFrameWnd::OnWindowNew()。因為只有在第三條路徑上，喚起 CFrameWnd::OnCmdMsg() 時夾帶的 this 指標指向 CMDIFrameWnd-derived obj，進而才能夠使 ID_WINDOW_NEW 命令流經 CMDIFrameWnd 的訊息映射表。

追蹤 MFC(Lite) 的訊息繞行機制，實在是一個巨大的挑戰。如果你想實際驗證所有被喚起的函式，只需將書附源碼中的 TRACEmr() 的註解標記全部解除，就可以在執行時期看到每一個命令訊息所引發的函式。TRACEmr() 是我在所有與訊息繞行機制有關的函式的起始處放置的一個追蹤（除錯）裝置，形式如下：

```
#define TRACEmr printf // 定義於 mfclite.h

BOOL CMDIFrameWnd::OnCommand(WPARAM wParam, LPARAM lParam)
{
    //TRACEmr("CMDIFrameWnd::OnCommand() \n");
    ...
}
```

mr 正是代表 "message routing" 之意。

6.9 MVC 模型^U (Model-View-Controller)

所謂 **MVC** (**Model/View/Controller**) 是三個一組的 classes，用來在 Smalltalk-80 中建立圖形使用介面 (GUI)。這個模型非常著名，繼 Smalltalk-80 之後也被許多系統 (包括 MFC) 採用。MVC 已經成為圖形使用介面的經典手法，被視為一種設計樣式 (design pattern) — 此一領域的經典名著《Design Patterns》第一章便對 MVC 有一個概要介紹，以下是其重點描述。

MVC 包括三種 objects。Model 是 application object，View 是其螢幕呈現，Controller 定義出 UI 對使用者輸入的回應方式。MVC 出現之前，UI 設計者往往把這些 objects 總括在一起，MVC 則把它們分開來，增加其彈性和復用性。

MVC 解除了 views 和 models 之間的耦合 (coupling) 關係，作法是在它們之間建立起一個 **subscribe-notify** (訂閱-通知) 協定。在此協定中，view 必須保證其畫面時刻反映出 model 現狀；一旦 model 內容有變，就通知各個相依的 views，使每一個 views 有機會更新自己。這種作法使我們得以讓多個 views 附著於一個 model 身上並提供各不相同的表現形式。

MVC 的表面價值是，反映出一種將 views 和 models 解除耦合關係的設計。然而此種模型其實還可以應用到更一般化的問題上：解除 objects 間的耦合關係，使某個 object 的改變反映在其他 objects 身上，而彼此不需要知道對方太多實作細節。這個更一般化的設計即是所謂的 **Observer** 設計樣式。此外 MVC 還帶有其他諸如 **Composite**、**Strategy**、**Factory Method**、**Decorator**...等設計樣式。

圖 6-31 顯示三個 views 依附於一個 model 身上。為了簡化，圖中並未顯示 controller。

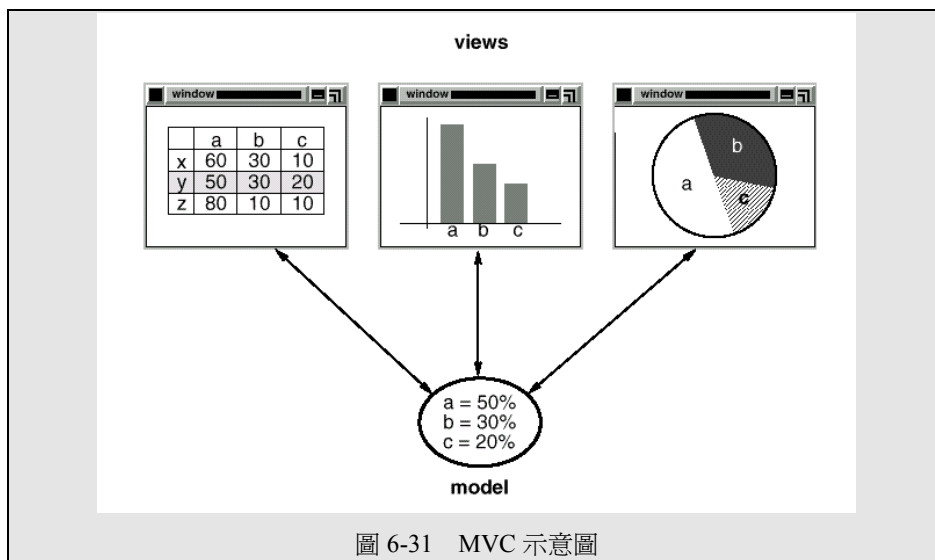


圖 6-31 MVC 示意圖

6.9.1 MFC(Lite) 中的 MVC 模型總覽

關於 MVC 模型的「資料與畫面分開處理」以及「subscribe-notify (訂閱-通知) 協定」, MFC(Lite) 有一個很好的實踐, 就是所謂的 Document-View (文件-視圖)。Document 負責實際資料的存放, 因此它應該具備容器功能、物件永續 (檔案讀寫) 功能; View 負責資料的顯現, 因此它應該具備繪圖功能以及接受使用者輸入的功能。MFC(Lite) 將 view 視為一種視窗, 這麼一來就自然而然能夠繪圖、自由縮放大小、接受輸入。為了更加區隔功能, MFC(Lite) 把 view 視窗的外框拿掉, 實際顯現時必須外置一個框架視窗, 或稱 docFrame 視窗。我們可以想像 view 充斥於 docFrame 視窗的內部, 隨著 docFrame 視窗的移動而移動、縮放而縮放, 如圖 6-32。

MFC(Lite) 支援多重文件, 也就是說同一個程式可以處理不同型態的文件資料 (就好像 Microsoft Word 可同時支援 doc、txt、rtf... 等格式的文件一樣)。每一種文件型態通常擁有自己專屬的 UI (其中最引人注目的是 menu), 這很合理, 因為不同的文件型態需要提供不同的命令給終端使用者操作。每一種文件型態稱為一個 document template (意思是可由這個「模板」生成一份一份的實際文件), 所有 document templates 由一個所謂的 document manager 管理之。整個管理架構如圖 6-33。圖 6-34 是 document-view 的所有相關類別。

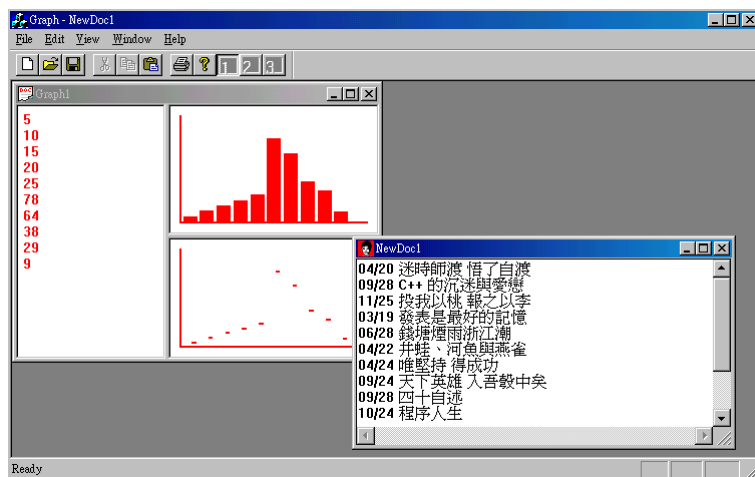


圖 6-32a 這個 MFC 應用程式支援兩個 DocTemplate（亦即兩種文件型態），目前開了兩份 documents，左側 document 同時開了三個 views（以 MDI 形式呈現），每個 view 以不同手法展現資料。右側 document 只開一個 view。不同的文件型態作用起來時，應該有相應的 UI（尤其引人注意的是 menu 或 toolbar）掛到 mainFrame 視窗上。至於 docFrame 視窗，並不允許掛上 menu。

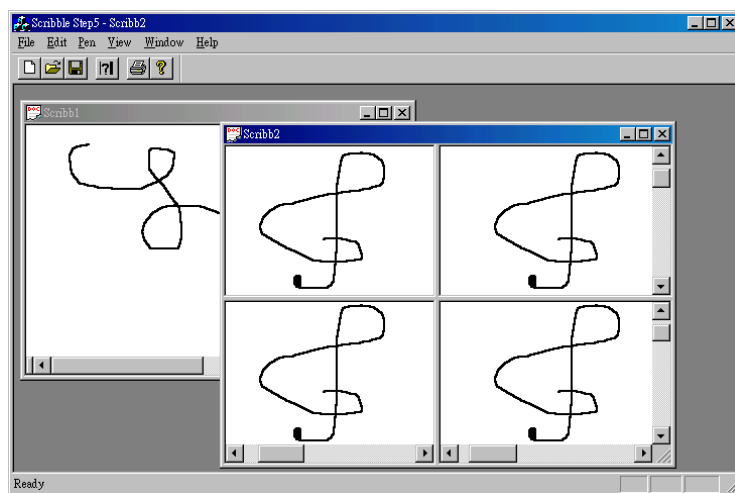


圖 6-32b 這個 MFC 應用程式支援一個 DocTemplate（亦即一種文件型態），目前開了兩份 documents。左側 document 開了一個 view。右側 document 開了四個 views（以 MDI 形式呈現），每個 view 都以相同手法展現同一份 document。

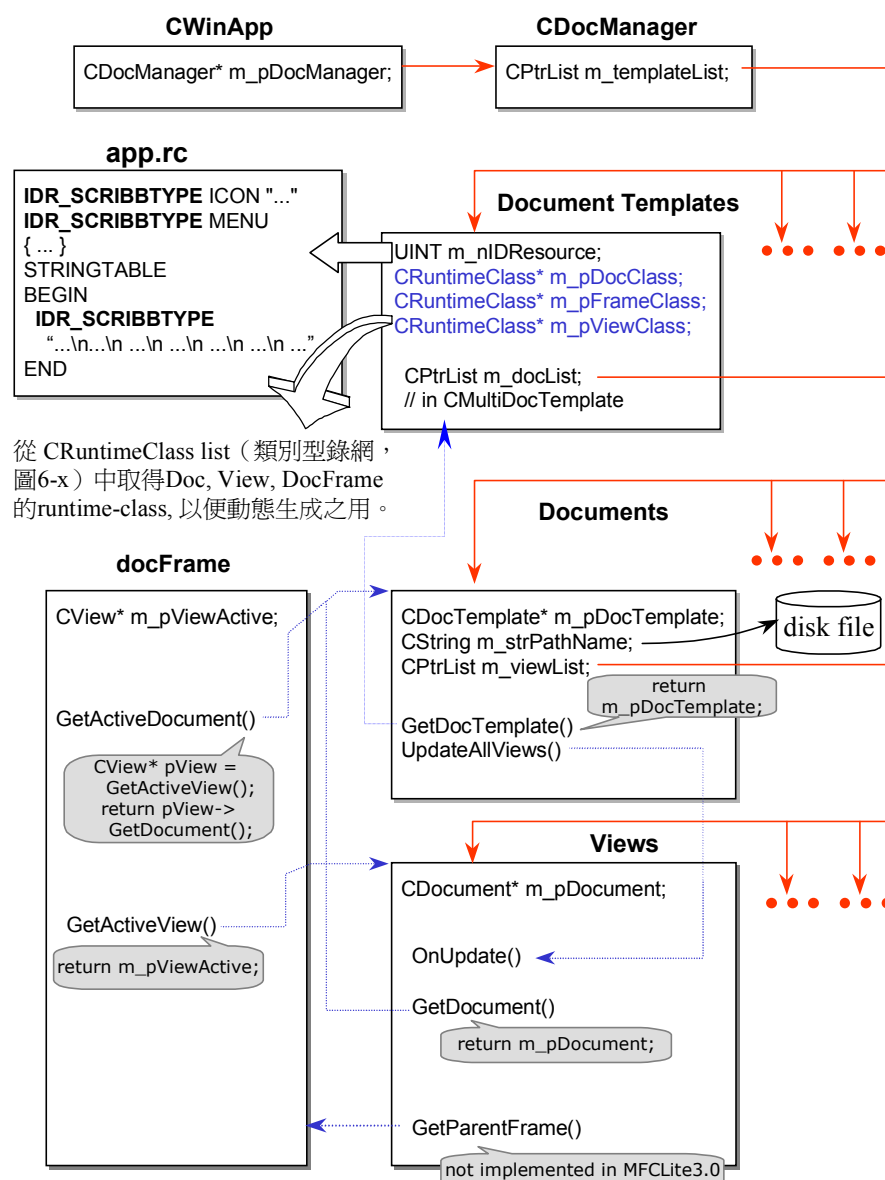


圖 6-33 MFC(Lite) 以本圖所示架構來管理 document-views。每個應用程式有一個 `CDocManager` object，負責維護一個 list，其內各元素都（可以）是 `CMultiDocTemplate` object；後者內部也有一個 list，其內各元素都是 `CDocument-derived` object；後者內部又有一個 list，其內各元素都是 `CView-derived` object；每個 view 對應一個 `docFrame` 視窗。MFC 允許多個 view 對應同一個 `docFrame`，但 **MFCLite 做⁷簡化，一個 view 必須對應一個 `docFrame`。**

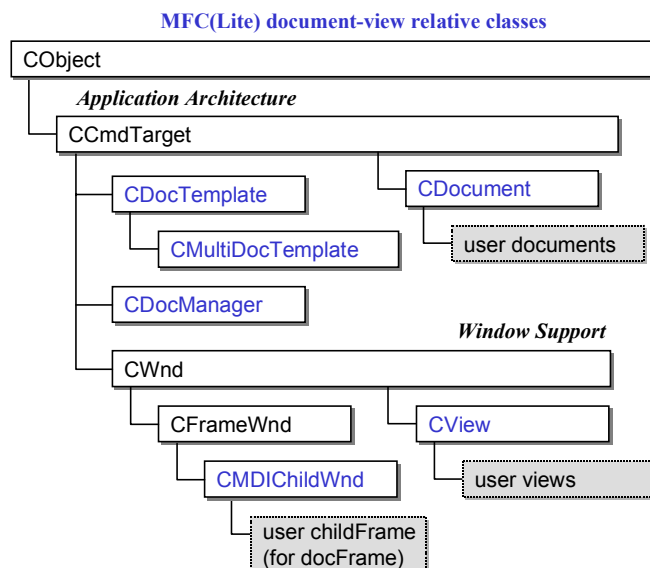


圖 6-34 MFC(Lite) 內的所有 document-view 相關類別

6.9.4 文件管理類別：

CDocManager, CDocTemplate, CMultiDocTemplate

正如圖 6-33 所示，每一個 MFC(Lite) 應用程式在其 CWinApp sub-object 內都有一個 CDocManager object，用來管理 document templates：

```

class CWinApp : public CWinThread
{
...
public:
    CDocManager* m_pDocManager;
};

class CDocManager : public CObject
{
public:
    virtual void AddDocTemplate(CDocTemplate* pTemplate);
public: // 在 MFC 之中原為 protected，為求方便 MFCLite 改為 public。
    CPtrList m_templateList; // CPtrList 詳見 6.2.5 節
    ...
};
  
```

不同型態的 document 需要不同的操作介面，因而需要不同的 menu。在 MFC 架構中所有 menu 都只能掛在 mainFrame 視窗上而不能掛在 docFrame 視窗上，因為後者只負責提供畫面搬移、縮放、極大極小等標準（基本）視窗功能。每當使用者在多重文件應用程式中切換至某一型態的文件，程式必須有能力在 mainFrame 視窗上切換一套對應的 menu。因此，MFC(Lite) 記錄 document 型態的同時，亦必須記錄其相應的 menu（更廣泛地說是一整套相應的 UI）。記錄方式如下。

欲加入新的文件型態，可透過 `CDocManager::AddDocTemplate()`：它接受一個 `CDocTemplate*` 參數。當我們決定使用多重文件時，我們會多次呼叫它並傳進一個 `CMultiDocTemplate*` 引數，例如：

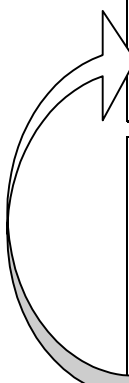
```
// 以下是 MFC(Lite) 應用模組
CMultiDocTemplate* pDocTemplate;
pDocTemplate = new CMultiDocTemplate(
    IDR_MYDOCTYPE,           // UI
    RUNTIME_CLASS(CMyDocument), // document
    RUNTIME_CLASS(CChildFrame), // docFrame
    RUNTIME_CLASS(CMyView)    // view
);
AddDocTemplate(pDocTemplate); // 登錄第一個文件型態

CMultiDocTemplate* pDocTemplate2;
pDocTemplate2 = new CMultiDocTemplate(
    IDR_MYDOCTYPE2,          // UI
    RUNTIME_CLASS(CMyDocument2), // document
    RUNTIME_CLASS(CChildFrame), // docFrame
    RUNTIME_CLASS(CMyView2)    // view
);
AddDocTemplate(pDocTemplate2); // 登錄第二個文件型態

// 以下是 MFC(Lite) 框架模組
void CWinApp::AddDocTemplate(CDocTemplate* pTemplate)
{
    if (m_pDocManager == NULL)
        m_pDocManager = new CDocManager; // 產生一個 DocManager
    m_pDocManager->AddDocTemplate(pTemplate); // 加入一個 DocTemplate
}

void CDocManager::AddDocTemplate(CDocTemplate* pTemplate)
{
    pTemplate->LoadTemplate(); // 載入 UI 相關物件 (如 menu, string)。
    // 對此，MFCLite 只維護一個空殼，實際什麼也沒做。
    m_templateList.AddTail(pTemplate); // 加入 list 之中。
}
```

圖 6-34 顯示，上列的 AddDocTemplate() 在多重文件時接受的 CMultiDocTemplate 係繼承自 CDocTemplate，後者內含一組資料，記錄以下四樣東西：



```

class CDocTemplate : public CCmdTarget
{
...
protected:
    string m_nIDResource;    // 原本是 IDR_ for frame/menu/accel
                           // 為簡化，改為只表現文件型態（副檔名）
    CRuntimeClass* m_pDocClass; // class for creating new documents
    CRuntimeClass* m_pFrameClass; // class for creating new docFrames
    CRuntimeClass* m_pViewClass; // class for creating new views
};

class CMultiDocTemplate : public CDocTemplate
{
...
public:
    CMultiDocTemplate(const string& nIDResource,
                      CRuntimeClass* pDocClass,
                      CRuntimeClass* pFrameClass,
                      CRuntimeClass* pViewClass)
        : CDocTemplate(nIDResource, pDocClass, pFrameClass, pViewClass) {}
protected:
    CPtrList m_docList; // 維護本型態（DocTemplate）之所有已開啓文件
};

```

四筆記錄之中，後三者分別是 document, view, docFrame 的 CRuntimeClass 指標，讓應用程式得以進行動態生成（6.4 節）。至於第一筆記錄，在 MFC 中原本是一個 RC 檔（資源描述檔）內以 UINT 代表的整套 UI 資源，包括表單、快速鍵、字串：

```

// 登錄一個文件型態
CMultiDocTemplate* pDocTemplate;
pDocTemplate = new CMultiDocTemplate(
    IDR_ScribTYPE,           // UI
    RUNTIME_CLASS(CMyDocument), // document
    RUNTIME_CLASS(CChildFrame), // docFrame
    RUNTIME_CLASS(CMyView)    // view
);
AddDocTemplate(pDocTemplate);

// 以下摘錄自 MFC 應用程式的 RC 檔
IDR_ScribTYPE ICON ...
IDR_ScribTYPE MENU ...
STRINGTABLE
BEGIN
    IDR_ScribTYPE "\nScrib\nScrib\nScribble Files(*.scb)\n.n.SCB\n
                  Scribble.Document\nScrib Document"

```

(注意，以上是連續字串。爲了版面閱讀方便，所以斷行)
(此一字串內含 7 個子字串，分別代表 document 的某些重要
資訊如型態名稱、副檔名、docFrame 視窗標題...)

END

但由於 MFCLite 不支援任何圖形介面，因此，就本章所關心的核心技術而言，唯一需要記錄的 document 相關資料只有「型態名稱」。Windows 作業系統往往以副檔名做爲文件型態的判斷依據(例如 .doc 爲 Microsoft Word 文件檔，.ppt 爲 Microsoft PowerPoint 文件檔、.txt 爲一般文字檔...)，沿續這個習慣，我將上述第一筆記錄的型別由 UINT 改爲 string，記錄副檔名，用以代表 document 型態。

以上描述的是多重文件型態的管理。剛才所展示的 CMultiDocTemplate 宣告式中出現的 m_docList 成員，將被用來置放所有已開啓的同型態文件。至於 document 和其所對應的 view 以及 docFrame 如何建立並維護關係，是接下來數個小節的主題。

6.9.3 文件，CDocument

圖 6-33 顯示，當應用程式登錄了一個文件型態，便在 CMultiDocTemplate object 內維護了一個 list：

```
CPtrList m_docList;    // 用來置放所有被開啓的同型態文件
```

文件生成 - 開新檔案 [File/New]

文件的產生，以 [File/New] (開新檔案) 或 [File/Open] (開啓舊檔) 兩種方式完成。無論如何，產生出來的文件都會被加入上述 list 之中。從這兩個 menu 命令項的按下 (MFCLite 係以熱鍵模擬之)，到啓動次第檔案讀寫 (Serialization) 機制，到動態生成 (Dynamic Creation) 機制，有一條相當長遠的路。所幸 6.5 節的「檔案讀寫」已經負擔了部分說明，此處只需再補足另一半，連接上去即可。

「開新檔案」或「開啓舊檔」兩個命令訊息被 MFC(Lite) 安排在這裡攔截：

```
BEGIN_MESSAGE_MAP(CMyWinApp, CWinApp)
    ON_COMMAND(ID_FILE_NEW, CWinApp::OnFileNew)
    ON_COMMAND(ID_FILE_OPEN, CWinApp::OnFileOpen)
END_MESSAGE_MAP()
```

後續呼叫動作分別描述於下。以下皆為 MFCLite 源碼和虛擬碼；MFC 必須考慮 GUI 及更多面向，所以某些地方更複雜些。

```
void CWinApp::OnFileNew()
{
    m_pDocManager->OnFileNew();
}
```

```
void CDocManager::OnFileNew()
{
    CDocTemplate* pBestTemplate;
    // 如果存在一個以上的 document templates，就詢問文件型態（MFCLite 中以
    // 副檔名代表），然後找出吻合的 document template，記錄於 pBestTemplate。
    // 如果只有一個 document templates，就直接用之。
    pBestTemplate->OpenDocumentFile(string()); // 引數是個空字串
    return;
}
```

```
// 由於 CDocTemplate::OpenDocumentFile() 是個純虛擬函式，所以上面的
// 呼叫動作喚起的是 CMultiDocTemplate::OpenDocumentFile()。
// 此處只列函式關鍵動作，細節請參考書附源碼。
CDocument* CMultiDocTemplate::OpenDocumentFile(
    const string& strFileName)
{
    // 以下開啓具名檔案。如果檔名長度為 0，產生一份新文件。
    // 產生一個 CDocument object（文件檔則於稍後開啓）
    CDocument* pDocument = CreateNewDocument(); // (A)
    // 以下產生一組 "CFrameWnd object + frame 視窗"，
    // 和一組 "CView object + view 視窗"
    CFrameWnd* pFrame = CreateNewFrame(pDocument, NULL); // (B)

    if (strFileName.length() == 0) // 表示來自 [File!New]
    {
        pDocument->SetTitle("untitled");
        // 開啓一個新的 document
        if (!pDocument->OnNewDocument()) { // (C)
            // 開檔失敗，警告使用者 ...
        }
    }
    else // 來自 [File!Open]
    {
        // 打開一個既有的 document（一個檔案）
        if (!pDocument->OnOpenDocument(strFileName)) { // (D)
            // 開檔失敗，警告使用者 ...
        }
    }
    return pDocument;
}
```

```

CDocument* CDocTemplate::CreateNewDocument()
{
    // 動態生成 document object
    CDocument* pDocument= (CDocument*)m_pDocClass->CreateObject();
    AddDocument(pDocument); // 加入管理行列。
    return pDocument;
}

```

```

// 本函式將產生一個 docFrame 視窗，對應於指定的 pDoc。
CFrameWnd* CDocTemplate::CreateNewFrame(CDocument* pDoc,
                                         CFrameWnd* pOther)
{
    CCreateContext context;
    context.m_pCurrentFrame = pOther;
    context.m_pCurrentDoc = pDoc;
    context.m_pNewViewClass = m_pViewClass;
    context.m_pNewDocTemplate = this;
    // 動態生成 CFrameWnd-derived object
    CFrameWnd* pFrame = (CFrameWnd*)m_pFrameClass->CreateObject();

    // 以下動作爲 jjhou 所加，爲的是能夠在 CView::OnCreate() 中
    // 很方便地設定 frame/view 的關係
    context.m_pCurrentFrame = pFrame;

    pFrame->LoadFrame(m_nIDResource, &context); // 產生 frame 視窗
    JJSetActiveFrame(pFrame->m_hWnd);
    // 上一動作爲 jjhou 補充，因爲 MFCLite 有自己的視窗管理方式。

    return pFrame;
}

```

以上兩段程式碼中所呼叫的 `CreateObject()` 分別造成 `CDocument-derived object` 和 `CFrameWnd-derived object` 的動態生成 (6.4 節)。至於 `LoadFrame()`，雖然在 `CFrameWnd` 和 `CMDIFrameWnd` 中都有實作，但此處的 `pFrame` 不可能是一個 pointer to `CMDIFrameWnd` (因爲 `pFrame` 賴以誕生的憑藉 `m_pFrameClass`，代表的是一個 `CMDIChildWnd-derived class`，見前述的 `CMultiDocTemplate` 初值設定)，所以喚起的必定是：

```

BOOL CFrameWnd::LoadFrame(const string& nIDResource,
                        CCreateContext* pContext /* =NULL */)
{
    // 企圖產生 frame 視窗。nIDResource 略而不用，因爲 MFCLite 並不支援 GUI。
    Create(pContext); // this->Create(). 詳見 6.6.4 節。
    return TRUE;
}

```


這裡，埋藏在 application framework 深處，由於缺少直接喚起動作，最隱晦最難被察覺的一段行為就是：docFrame 視窗的誕生觸發了 WM_CREATE 訊息（詳見 6.6.4 節），進而喚起 CMDIChildWnd::OnCreate()：

```
BEGIN_MESSAGE_MAP(CMDIChildWnd, CFrameWnd)
    ON_WM_CREATE()
END_MESSAGE_MAP()
```

```
int CMDIChildWnd::OnCreate(LPCREATESTRUCT lpcs)
{
    // MFC 原本在這裡處理 MDI 相關事務。由於 MFCLite 簡化了問題，不允許
    // docFrame (a CMDIChildWnd object) 再有子視窗（亦即否定其 MDI 功能），
    // 因此以下做了簡化，把 CMDIChildWnd 視同 CFrameWnd。
    if (CFrameWnd::OnCreate(lpcs) == -1)
        return -1;
    return 0; // success
}
```

```
int CFrameWnd::OnCreate(LPCREATESTRUCT lpcs)
{
    CWnd::OnCreate(lpcs);
    // 這裡應呼叫 OnCreateHelper() => OnCreateClient() => CreateView()
    // （詳見《深入淺出 MFC》2/e 繁體版 p464，簡體版 p346）
    // 最終在 CreateView() 中完成 CView 物件的建構與 view 視窗的誕生。
    // 為求簡化，MFCLite 跳過 OnCreateHelper() 和 OnCreateClient()
    // （兩者都和 UI 有關），直接在這裡呼叫 CreateView()。
    CCreateContext* pContext = (CCreateContext*)lpcs->lpCreateParams;

    // 只有當 pContext 不為 NULL，才需往下製造 CView object 及 view window.
    // 例如，來自 CMYWinApp::InitInstance() 的呼叫只指定第一引數，導致
    // pContext 為預設值 NULL；是的，因為當時要的是 mainFrame 而非 docFrame
    if (pContext != NULL)
        CreateView(pContext);

    delete lpcs->lpCreateParams;
    delete lpcs; // 這是跟著 msg 而來的一個 heap object。必須清除。
    return 0; // success
}
```

```
CWnd* CFrameWnd::CreateView(CCreateContext* pContext)
{
    CWnd* pView = (CWnd*)pContext->m_pNewViewClass->CreateObject();
    // 動態生成 CView object
    pView->Create(pContext); // 產生 view 視窗，進而產生 WM_CREATE，
    // 進而喚起 CView::OnCreate()
    JJSetActiveView(pView->m_hWnd); // MFCLite 獨特（簡化）的視窗管理
    return pView;
}
```

上述呼叫過程中，你看到諸如 `CCreateContext` 或 `CREATESTRUCT` 等資料型別，用來傳遞大量資訊。前者可記錄動態生成需要的各種資訊，後者是 Windows API programming 所使用的資料結構：

```
// 記錄產生一個視窗時所需的所有資訊。本例簡化為只剩一個欄位。
typedef struct tagCREATESTRUCTA {
    LPVOID      lpCreateParams;
} CREATESTRUCT, *LPCREATESTRUCT;
```

MFC(Lite) 正是利用 Windows API 提供的這麼一個夾帶資訊的機會，成功地將動態生成所需資訊夾帶給 `CFrameWnd::OnCreate()`，使後者得以動態生成出 `CView` object。整個故事得回溯到視窗的誕生過程（6.6.4 節），彼時曾經提過，所有視窗最終都是藉由 `CWnd::CreateEx()` → `::CreateWindowEx()` 而誕生，前者接受的引數便是來自 `Create()` 的一個 `CCreateContext*`，並被轉為 `void*`：

```
BOOL CWnd::Create(/*...*/ CCreateContext* pContext)
{
    return CreateEx((LPVOID)pContext);
    // 喚起 non-virtual CWnd::CreateEx()
}

BOOL CFrameWnd::Create(/*...*/ CCreateContext* pContext)
{
    CreateEx((LPVOID)pContext); // 喚起 non-virtual CWnd::CreateEx()
    return TRUE;
}

BOOL CWnd::CreateEx(/*...*/ LPVOID lpParam)
{
    // ...
    m_hWnd = ::CreateWindowEx(lpParam);
}
```

```
HWND CreateWindowEx(/*...*/ LPVOID lpParam)
{
    // ...
    // 模擬 WM_CREATE 訊息誕生（並進入 msg queue）。Windows 規定：
    // WM_CREATE lpParam 應放置 CREATESTRUCT（代表視窗生成的所有資訊）位址。
    // 本例簡化 CREATESTRUCT，使它只含一個成員 lpCreateParams，
    // 並令其值為本函式接獲的最後一個引數：lpParam。
    CREATESTRUCT* pcs = new CREATESTRUCT;
    if (lpParam == NULL)
        pcs->lpCreateParams = lpParam;
```

```

else
    pcs->lpCreateParams =
        new CCreateContext(*(CCreateContext*)lpParam);
MSG msg;
msg.hWnd = hWnd;
msg.nMsg = WM_CREATE;
msg.wParam = 0;
msg.lParam = (LPARAM)pcs;    // 接受此訊息者，應該負責 clean-up
g_msgQueue.push(msg);        // 將 WM_CREATE 訊息推入訊息佇列
return hWnd;
}

```

深入下探了這麼多細節之後，讓我們再回到制高點。目前的情況是，先前所列的 `CMultiDocTemplate::OpenDocumentFile()` 內執行了 `CreateNewDocument()` 和 `CreateNewFrame()`，於是動態生成一個 `document`、一個 `docFrame` 和一個 `view`。再來便要執行 `pDocument->OnNewDocument()`。新開一份文件並不需要任何準備動作（因為新文件內沒有內容），只要將某些記錄設定妥當即可：

```

BOOL CDocument::OnNewDocument()
{
    m_strPathName.empty(); // 開一個新文件，不需給予檔名
    return TRUE;
}

```

接下來，文件的內容有賴「`view` 視窗接受使用者輸入」而完成，詳見 6.9.4 節。

圖 6-35 是 [File/New]（開新檔案）的完整程式流程。

文件生成之二：開啓舊檔 [File/Open]

如果使用者按下的是 [File/Open]（開啓舊檔），根據先前揭示的 `message map`，會被攔截至以下函式進行處理，後續動作亦分別描述於後。

```

void CWinApp::OnFileOpen()
{
    m_pDocManager->OnFileOpen();
}

```

```

void CDocManager::OnFileOpen()
{
    string newName;
}

```

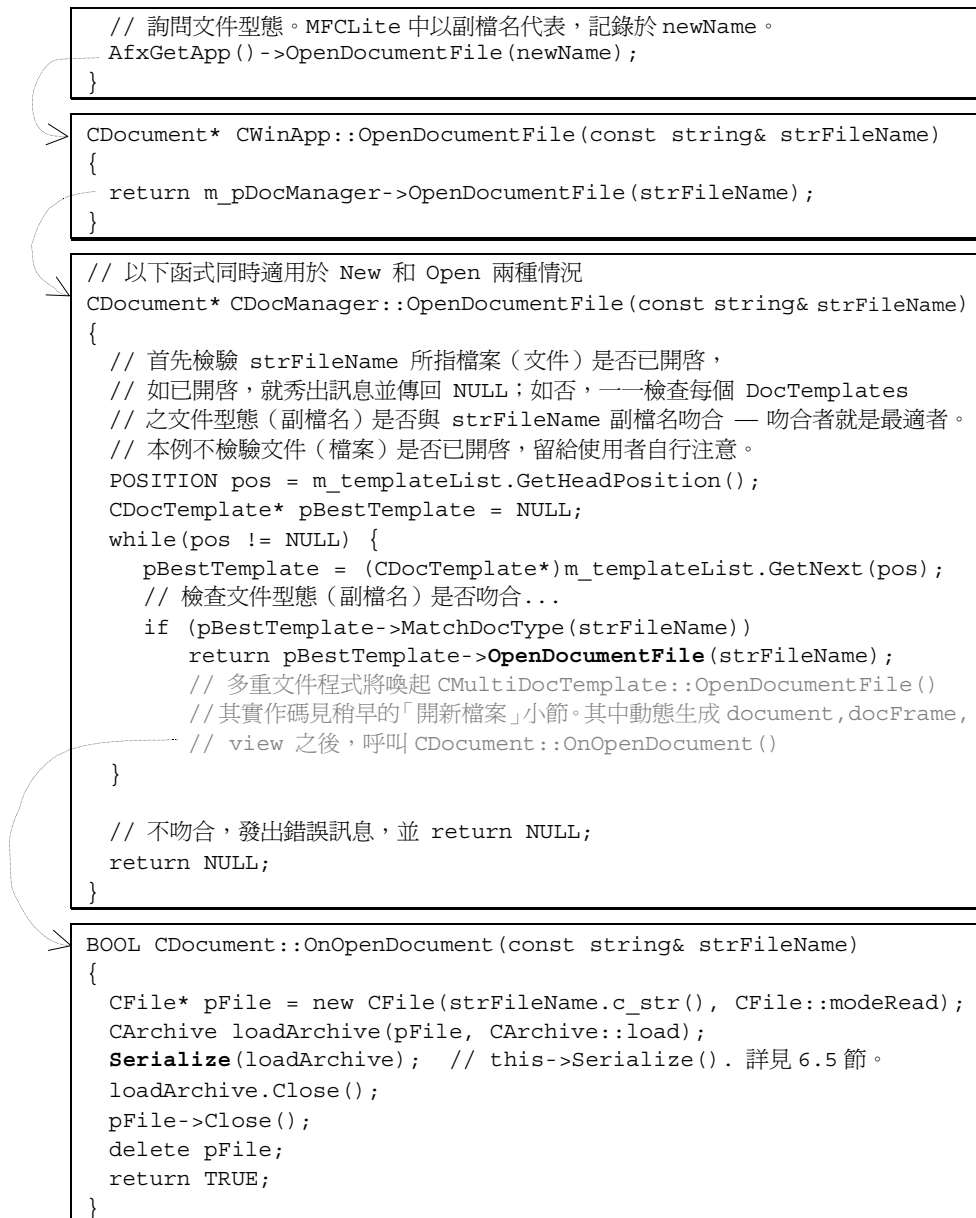


圖 6-35 是 [File/Open] (開啓舊檔) 的完整程式流程。

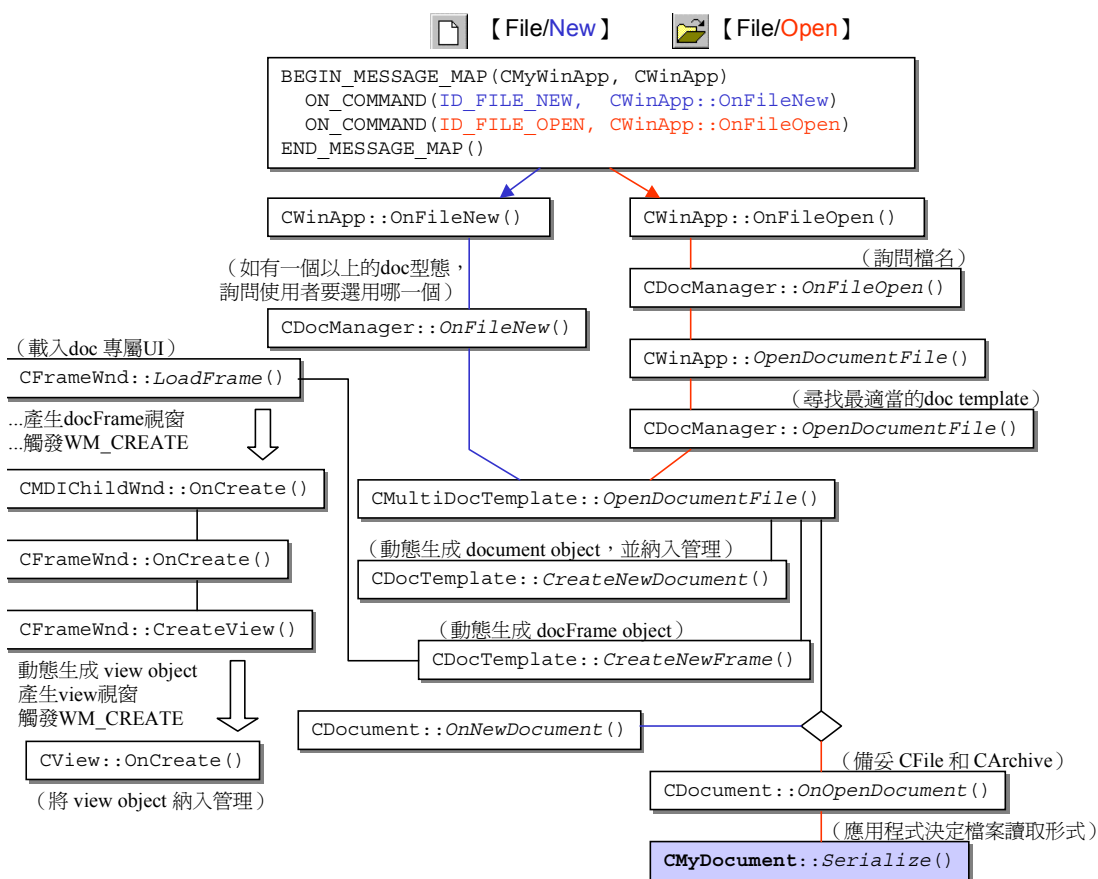


圖 6-35 [File/New] (開新檔案) 和 [File/Open] (開啓舊檔) 的完整程式流程。圖中斜體字表示虛擬函式，如果應用程式覆寫了它們，執行流程就會流往應用程式。

文件儲存之二：儲存檔案 [File/Save]

[File/Save] (儲存檔案) 和 [File/New] (另存新檔) 兩個命令訊息被 MFC(Lite) 安排在這裡處理：

```

BEGIN_MESSAGE_MAP(CDocument, CCmdTarget)
ON_COMMAND(ID_FILE_SAVE_AS, OnFileSaveAs)
ON_COMMAND(ID_FILE_SAVE, OnFileSave)
END_MESSAGE_MAP()
  
```

後續呼叫動作分別描述於下。以下皆為 MFCLite 源碼和虛擬碼；MFC 必須考慮 GUI 及更多面向，所以某些地方更複雜些。

```
void CDocument::OnFileSave()
{
    DoFileSave();
}
```

```
BOOL CDocument::DoFileSave()
{
    DoSave(m_strPathName);
    return TRUE;
}
```

```
BOOL CDocument::DoSave(const string& strFileName)
// 本函式同時適用於 Save 和 SaveAs
{
    string newName = strFileName;

    // 以下對 MFC 略作修改，以新增之 GetDocType() 取得目前文件之副檔名
    string extName = m_pDocTemplate->GetDocType();

    if (newName.length() == 0) { // 'SaveAs'
        // ... 給予提示，要求輸入檔案名稱，並接受輸入至 newName
    }

    // 檢查副檔名是否與文件型態 (CDocTemplate's m_nIDResource 所記錄) 吻合。
    // 如不吻合，給予使用者錯誤訊息，並 return false;
    if (!m_pDocTemplate->MatchDocType(newName)) {
        TRACE0("extention file name error!\n");
        return FALSE;
    }

    OnSaveDocument(newName);
    return TRUE;
}
```

```
BOOL CDocument::OnSaveDocument(const string& strFileName) //5
{
    CFile* pFile = new CFile(strFileName.c_str(), CFile::modeWrite);
    CArchive saveArchive(pFile, CArchive::store);
    Serialize(saveArchive); // this->Serialize()
    saveArchive.Close();
    pFile->Close();
    delete pFile;
    return TRUE;
}
```

很明顯可以看出來，文件的儲存比文件的創建簡單多了，主要是因為不再需要動態生成 document, docFrame, view。圖 6-36 是 [File/Save]（儲存檔案）的完整程式流程。

文件儲存之二：另存新檔 [File/SaveAs]

如果使用者按下的是 [File/SaveAs]（另存新檔），根據先前揭示的 message map，會被攔截至以下函式進行處理。

```
void CDocument::OnFileSaveAs()
{
    DoSave(string());
}
```

CDocument::DoSave() 的實作碼及其後續動作已於上一頁揭示。圖 6-36 是 [File/SaveAs]（另存新檔）的完整程式流程。

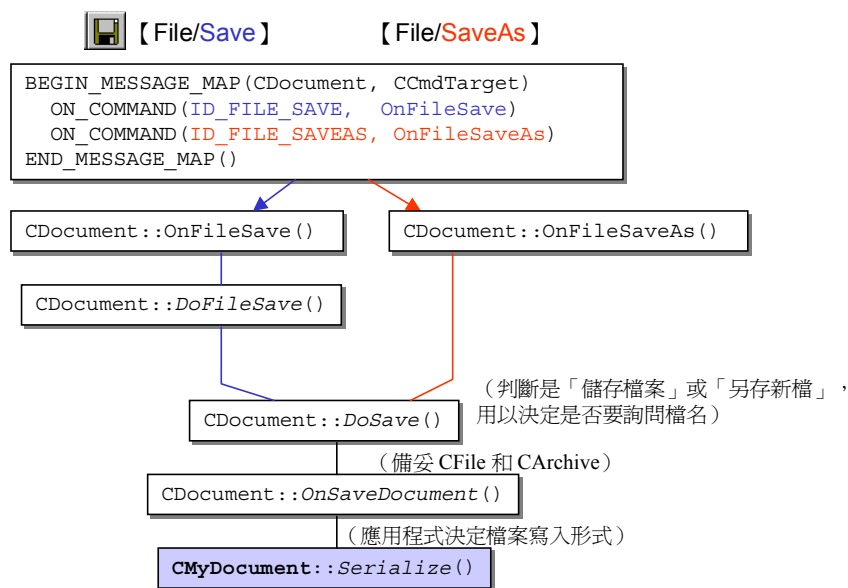


圖 6-36 [File/Save]（儲存檔案）和 [File/SaveAs]（另存新檔）的完整程式流程。

斜體字表示虛擬函式，如果應用程式覆寫了它們，執行流程就會流往應用程式。

6.9.4 視圖，CView

MFC(Lite) 以 document-view 模型將文件區分為資料本體和外顯形式，這正是 MVC 模型的兩大特質之一（另一特質是 **subscribe-notify** 協定，見 6.9.7 節）。同一份資料可以有不同的顯現方式（或相同的顯現方式但不同的「觀景窗」），因此同一個 document 理應允許多個相應的 views，這可從圖 6-33 清楚看出：document 有一個資料成員 `m_viewList`，用來管理所有相應的 views。當框架模組根據需求動態生成 document-、docFrame-、view- objects 時（如前一小節所述），隨著 `WM_CREATE` 訊息被喚起的 `CView::OnCreate()` 函式便會將 view object 加入上述 list 內：

```
int CView::OnCreate(LPCREATESTRUCT lpcs)
{
    CWnd::OnCreate(lpcs); // jjhou: just for default behavior?

    CCreateContext* pContext = (CCreateContext*)lpcs->lpCreateParams;
    pContext->m_pCurrentDoc->AddView(dynamic_cast<CView*>(this));
    ...
    return 0; // success
}
```

做為一個專門負責「表現」document 內容的類別，CView-derived classes 應該覆寫 `OnDraw()` 函式，這在 CView 中是個純虛擬函式，因此你也不能不覆寫它。關於這一部分，我將在稍後 6.9.7 節詳述。

View 的另一功能是做為使用者輸入介面。為此，你的 `CMyView` 必須攔截滑鼠訊息。下面是典型作法：

```
// 應用模組
BEGIN_MESSAGE_MAP(CMyView, CView)
    ON_WM_LBUTTONDOWN() // 滑鼠左鍵按下
    ON_WM_LBUTTONUP()   // 滑鼠左鍵放開
    ON_WM_MOUSEMOVE()    // 滑鼠移動
END_MESSAGE_MAP()
```

6.10 節所展示的 MFCLite 應用程式中，便以熱鍵 'u' 代表 `WM_LBUTTONUP`，進而觸發一連串預設的文件內容充填動作：

```
void CMyView::OnLButtonUp(UINT nFlags, CPoint point)
// CPoint 代表滑鼠位置。MFCLite 不支援 GUI，所以此參數無用，但仍保留其介面。
```



```

{
    CMyDocument* pDoc = GetDocument();
    ... // 充填文件內容
    pDoc->UpdateAllViews(this); // 詳見 6.9.7 節
}

```

透過圖 6-33，這些動作一目瞭然。

「面對同一份 document 產生第二、第三…個 views」的作法，MFC 支援了許多種。MFCLite 簡化為一種：透過表單命令 [Window/New] 完成。這個命令是標準的 Windows 程式功能。詳見 6.9.6 節。

6.9.5 文件外框視窗 (docFrame)，CChildFrame

所謂文件外框視窗，其實就是一般的 frame 視窗，應該繼承自 CFrameWnd。如果希望這個外框視窗更有 MDI 功能，就得令它繼承自 CMDIChildWnd。但我早已強調，docFrame 視窗採用 MDI 風格將過於複雜，對我所要討論的核心技術無益，所以 MFCLite 不支援這個功能。但是為求儘可能保留 MFC 介面，我還是這麼做：

```

class CChildFrame : public CMDIChildWnd { ... };

pDocTemplate = new CMultiDocTemplate(
    IDR_MYDOCTYPE,           // UI
    RUNTIME_CLASS(CMyDocument), // document
    RUNTIME_CLASS(CChildFrame), // docFrame
    RUNTIME_CLASS(CMyView)    // view
);

```

然後簡化 CMDIChildWnd，使它的種種行為就像 CFrameWnd 一樣。換句話說在 MFCLite 中，上述的 document template 和以下寫法所產生的效果完全相同：

```

pDocTemplate = new CMultiDocTemplate(
    IDR_MYDOCTYPE,           // UI
    RUNTIME_CLASS(CMyDocument), // document
    RUNTIME_CLASS(CFrameWnd),  // docFrame
    RUNTIME_CLASS(CMyView)    // view
);

```

一旦我們在這裡指定了 docFrame 的類別，當程式必須產生文件（詳見 6.9.3 節）並因而需要動態生成一個 docFrame 視窗時，就會取用這個被指定的類別。

6.9.6 在目前作用的 document 增加一個 view

先前已經說過，MFCLite 唯一支援的一種「為 document 產生一個新的 view」的作法就是透過 Windows 程式的標準表單命令 [Window/New] (開新視窗)。既曰標準表單命令，表示它在框架模組 (而非應用模組) 中處理。命令可流經整個訊息映射表，所以 MFC(Lite) 可以在任何一個 classes 攔截它，但是就此命令訊息而言，比較理想的地點還是在 `CMDIFrameWnd`，因為此一命令的處理過程中需要得知 active document，而 `CMDIFrameWnd` 可以比較方便地獲得它。

```
BEGIN_MESSAGE_MAP(CMDIFrameWnd, CFrameWnd)
    ON_COMMAND(ID_WINDOW_NEW, OnWindowNew)
END_MESSAGE_MAP()

void CMDIFrameWnd::OnWindowNew()
{
    CMDIChildWnd* pActiveChild = MDIGetActive();
    CDocument* pDocument;
    if (pActiveChild == NULL ||
        (pDocument = pActiveChild->GetActiveDocument()) == NULL)
    {
        TRACE0("Warning: No active document.\n");
        return;    // command failed
    }

    // 現在，準備產生一個 new docFrame!
    CDocTemplate* pTemplate = pDocument->GetDocTemplate();
    CFrameWnd* pFrame = pTemplate->CreateNewFrame(pDocument,
                                                pActiveChild);

    if (pFrame == NULL)
    {
        TRACE0("Warning: failed to create new frame.\n");
        return;    // command failed
    }

    // InitialUpdateFrame... (MFCLite 略)
}
```

以上實作碼中產生一個新的 docFrame，兼而也就產生了一個 view (詳見圖 6-35 左下部分)。是的，docFrame 和 view 是須與不可分離的雙生體☺。

但是，透過這種方式產生的新視圖，由於使用相同的 view class，也就有著相同的文件表現手法 (亦即同一個 `OnDraw()`)。如果我們希望新產生的 view 有不一樣的文件表現手法，該當如何？正規作法 (MFC 內) 是以靜態分裂視窗 (static splitter)

在具備 MDI 性質的 `CChildFrame` 視窗中分別產生多個子視窗(或曰窗口, `panes`)，並令各個窗口擁有各自的 `view class` (如圖 6-32a。技術細節詳見《深入淺出 MFC》2e，第 13 章)。由於 `MFCLite` 不支援 `CChildFrame` 的 MDI 性質，所以此法無從施行於 `MFCLite` 身上。

另一個(非正規)作法是從上述的 `CMDIFrameWnd::OnWindowNew()` 獲得靈感，依樣畫葫蘆地撰寫一個 `CMainFrame::OnWindowNew()`，並利用訊息映射表，將 `[Window/New]` 命令訊息牽引到這個新函式身上，然後我們再在其中做點手腳。此法劍走偏鋒，從應用模組(而非框架模組)的角度去解決，雖然不是正規作法，卻能夠讓我們更加了解圖 6-33 的架構，因此我將在 6.10.1 節實現之。

6.9.7 文件與視窗之間的 subscribe-notify 協定

subscribe-notify 協定是 MVC 模型的一個重要特徵，其意義是，由於文件（資料本體）和其顯現方式被切割開來，因此必須建立一種協定，俾當文件內容有變時得以透過這個協定通知所有的 views 進行重繪動作（達到資料圖像的即時更新）。

MFC(Lite) document-view 模型之間的 subscribe-notify 協定是以下列方式達成（請參考圖 6-33）：

```
// 以下在框架模組 (application framework) 端
class CDocument : public CCmdTarget
{
public:
    // subscribe-notify 協定中的通知動作。
    void UpdateAllViews(CView* pSender, LPARAM lHint = 0L,
                       CObject* pHint = NULL);
    ...
};

class CView : public CWnd
{
protected:
    // subscribe-notify 協定中的收受函式，代表 document 內容有變。
    virtual void OnUpdate(CView* pSender, LPARAM lHint,
                        CObject* pHint);
    // 接收「資料有變」的通知後，輾轉喚起以下函式。
    void OnPaint();
    virtual void OnDraw(/* CDC* pDC */) = 0;
};
```

首先請你注意，CDocument::UpdateAllViews() 是爲了讓外界呼叫並進而啓動 subscribe-notify 協定，因此它的存取層級是 public。CView::OnUpdate() 是爲了接收通知，是系統內部動作，不準備讓外界呼叫，所以它的存取層級是 protected，但它可以被身爲 friend 的 CDocument object 呼叫。

下面是 subscribe-notify 協定的來龍去脈：

```
void CDocument::UpdateAllViews(CView* pSender,
                               LPARAM lHint /* 0L */,
                               CObject* pHint /* NULL */)
{
    // 巡訪所有的 views，一一呼叫其 OnUpdate()。
```

```

POSITION pos = m_viewList.GetHeadPosition();
while (pos != NULL)
{
    CView* pView = (CView*)m_viewList.GetNext(pos);
    if (pView != pSender) // 自己不通知自己（從常理看，沒有必要）
        pView->OnUpdate(pSender, lHint, pHint);
}

```

```

void CView::OnUpdate(CView* pSender, LPARAM, CObject*)
{
    // 令重繪
    Invalidate(TRUE);
}

```

/*
 你也可以在 CMyView 中覆寫虛擬函式 OnUpdate()，在其中對參數 lHint 和 pHint 做點處理，那兩個參數主要是為了強化繪圖效率（例如框圈出重繪範圍，避免全部重繪）而設計。如果你的確覆寫了 OnUpdate()，請依上述標準行為，呼叫 Invalidate()。
 */

```

void CWnd::Invalidate(BOOL bErase /* TRUE */)
{
    ::InvalidateRect(m_hWnd); // ref afxwin2.inl
}

```

```

BOOL InvalidateRect(HWND hWnd) // 模擬 Win32 API
{
    //...
    PostMessage(hWnd, WM_PAINT, 0, 0);
    return TRUE;
}

```

```

BEGIN_MESSAGE_MAP(CView, CWnd)
    ON_WM_PAINT()
END_MESSAGE_MAP()

```

```

void CView::OnPaint()
{
    // ...準備繪圖工具。MFCLite 略。
    OnDraw(/* &dc */); // this->OnDraw()
}

```

```

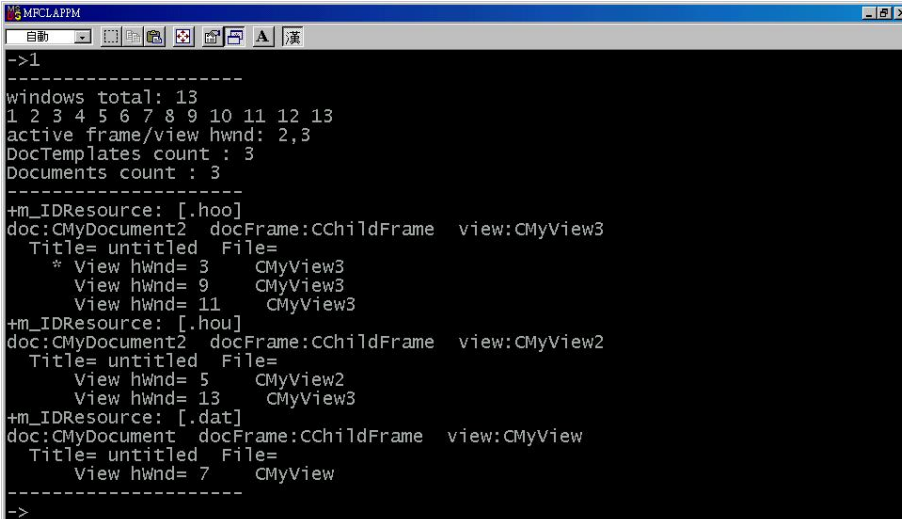
void CMyView::OnDraw(/* CDC* */)
{
    CMyDocument* pDoc = GetDocument();
    ... // 顯示文件內容
}

```

6.10 一個完整的測試程式

MFCLite3 的所有核心技術，已經在前 9 個小節中解說完畢。當然，還有更多技術細節，在書附源碼中等待你去體會。現在，測試的時候到了。

本節設計的這個 MFCLite 測試程式（亦為書附源碼一部分）幾乎完全相同於一般 MFC 應用程式。程式支援多重文件，分別為 .dat 和 .hou 兩種型態。並支援以下表單命令和訊息：開啓舊檔（'o'）、開新檔案（'N'）、儲存檔案（'s'）、另存新檔（'A'）、開新視窗（'w'）、關於程式（'B'）、觀察程式現狀（'o'）、切換作用視窗（'l'）、輸出除錯檔（'g'）、求助（'H'）、送出重繪訊息 WM_PAINT（'p'）、滑鼠左鍵按下（'d'）、滑鼠左鍵放開（'u'）、滑鼠移動（'m'）。任何按鍵動作都會被記錄於螢幕上。程式將按鍵所引發的訊息處理完畢後，便停下來等待下一次輸入。除錯檔（詳見 6.11 節）固定名為 "debug.txt" — 每當程式啟動時便被清空，一旦使用者按下 'g'，便將「類別型錄網」的所有元素（CRuntimeClass objects）和訊息映射表的所有內容輸出到 "debug.txt"。下面是這個名為 mfclApp.exe 的程式的執行畫面（此程式不涉及任何圖形介面，可於任何平台的 console 環境內執行）：



```

MFCLAPP
自動
->1
-----
windows total: 13
1 2 3 4 5 6 7 8 9 10 11 12 13
active frame/view hwnd: 2,3
DocTemplates count : 3
Documents count : 3
-----
+m_IDResource: [.hoo]
doc:CMYDocument2 docFrame:CChildFrame view:CMYView3
  Title= untitled File=
    * View hwnd= 3 CMYView3
    View hwnd= 9 CMYView3
    View hwnd= 11 CMYView3
+m_IDResource: [.hou]
doc:CMYDocument2 docFrame:CChildFrame view:CMYView2
  Title= untitled File=
    View hwnd= 5 CMYView2
    View hwnd= 13 CMYView3
+m_IDResource: [.dat]
doc:CMYDocument docFrame:CChildFrame view:CMYView
  Title= untitled File=
    View hwnd= 7 CMYView
-----
->
  
```

下面是執行流程與執行結果的解說（-> 是輸入提示符號）：

D:\pic2\mfclite>mfclappm 在 Windows console (DOS box) 中執行程式。
一開始便如同幾乎任何 Windows 程式一樣地開啓一份空白文件。本程式支援兩種文件型態 (.hou 和 .dat)，所以會詢問使用者欲開啓何種型態的文件。

```
(1) .hou (2) .dat
select docType (1 or 2 or 3...) : 2      詢問文件型態。選擇題。
->N                                     [File/New] 開新檔 (新文件)
(1) .hou (2) .dat
select docType (1 or 2 or 3...) : 1      詢問文件型態。選擇題。
->W                                     [Window/New] 開新視圖 (根據目前文件)。
->W                                     [Window/New] 開新視圖 (根據目前文件)。
->O                                     [App/Status] 顯示目前狀態。
-----
windows total: 9                        目前總共的視窗數量
1 2 3 4 5 6 7 8 9                      所有視窗。
active frame/view hwnd: 8,9             作用中的 docFrame/view 視窗
DocTemplates count : 2                  兩種文件型態。
Documents count : 2                     兩份文件。
-----                                以下為各類文件現狀
+m_IDResource: [.hou]                   文件型態 .hou
doc:CMyDocument2 docFrame:CChildFrame view:CMyView2
  Title= untitled File=                 目前無檔名
    View hwnd= 5   CMyView2             視圖視窗一
    View hwnd= 7   CMyView2             視圖視窗二
    * View hwnd= 9   CMyView2            視圖視窗三，active
+m_IDResource: [.dat]                   文件型態 .dat
doc:CMyDocument docFrame:CChildFrame view:CMyView
  Title= untitled File=                 目前無檔名
    View hwnd= 3   CMyView               視圖視窗一
-----
->1                                     熱鍵，切換 active 視窗
-----                                切換後自動顯示目前狀態
windows total: 9
1 2 3 4 5 6 7 8 9
active frame/view hwnd: 2,3             作用中的 docFrame/view 視窗
DocTemplates count : 2
Documents count : 2
-----
+m_IDResource: [.hou]
doc:CMyDocument2 docFrame:CChildFrame view:CMyView2
  Title= untitled File=
    View hwnd= 5   CMyView2
    View hwnd= 7   CMyView2
    View hwnd= 9   CMyView2
+m_IDResource: [.dat]
doc:CMyDocument docFrame:CChildFrame view:CMyView
  Title= untitled File=
    * View hwnd= 3   CMyView             視圖視窗，active
-----
->W [Window/New] 開新視圖 (根據目前的 document)
```

```

->u      [LButtonUp] 滑鼠左鍵放開。本例視同 UI 輸入完畢，MFCLite 會強迫
          發出 WM_PAINT，要求視窗重繪。以下之文字輸出用以模擬視窗繪圖。
CEllipse: x=3, y=3, r1=7, r2=21
CCircle: x=5, y=5, r=7
CTriangle: x1=0, y1=0 x2=1, y2=0 x3=0, y3=1
CRect: x=5.6, y=6.8, height=3, width=9
CSquare: x=3.2, y=4.3, width=6
CStroke(7): 1 2 3 4 5 6 7
CTriangle: x1=0, y1=0 x2=3.5, y2=0 x3=0, y3=5.3
CRect: x=9.9, y=9.9, height=4.8, width=9.8
->S      [File/Save] 存檔
filename (with ext-name such as 'file.dat'): temp.dat 詢問檔名
CMyDocument::Serialize() 寫入檔案（物件永續）
->1      [ChangeActiveWnd] 循環切換作用視窗。
... 畫面略 切換後自動顯示目前狀態。
->1      [ChangeActiveWnd] 循環切換作用視窗。
          切換後自動顯示目前狀態。
-----
windows total: 11
1 2 3 4 5 6 7 8 9 10 11 所有視窗
active frame/view hwnd: 4,5 作用中的 docFrame/view 視窗
DocTemplates count : 2 兩種文件型態
Documents count : 2 兩份文件
-----
+m_IDResource: [.hou] 文件型態 .hou
doc:CMyDocument2 docFrame:CChildFrame view:CMyView2
  Title= untitled File= 目前無檔名
    * View hwnd= 5 CMyView2 視圖視窗一，active
      View hwnd= 7 CMyView2 視圖視窗二
      View hwnd= 9 CMyView2 視圖視窗三
+m_IDResource: [.dat]
doc:CMyDocument docFrame:CChildFrame view:CMyView
  Title= temp.dat File= temp.dat 存檔後，此文件被賦予檔名
    View hwnd= 3 CMyView 視圖視窗一
    View hwnd= 11 CMyView 視圖視窗二
-----
->u      [LButtonUp] 滑鼠左鍵放開，本例視同 UI 輸入完畢
CMyView2::OnLButtonUp() 目前有三個 views，所以自動通知另兩個 views
CMyView2::OnDraw() 自動通知繪圖
9 7 5 3 1 繪出內容
CMyView2::OnDraw() 自動通知繪圖
9 7 5 3 1 繪出內容
->A      [File/SaveAs] 存檔
filename (with ext-name such as 'file.hou'): temp.hou 詢問檔名
CMyDocument2::Serialize() 寫入檔案（物件永續）
->X      結束程式
          （後續還有清理動作與提示，見 6.12.3 節）

```


執行完畢後，以傾印工具觀察 `temp.dat` 內容，結果如下（內容解說見 6.2.5 節。請注意，6.12.2 節對於文件格式有更好更完整的設計）：

```
Turbo Dump Version 3.1 Copyright (c) 1988, 1992 Borland International
Display of File TEMP.DAT (245 bytes)

000000: 08 00 00 00 08 00 43 45 6C 6C 69 70 73 65 00 00 .....CEllipse..
000010: 40 40 00 00 40 40 00 00 E0 40 00 00 A8 41 00 00 @@...@...@...A..
000020: 07 00 43 43 69 72 63 6C 65 00 00 A0 40 00 00 A0 ..CCircle...@...
000030: 40 00 00 E0 40 00 00 09 00 43 54 72 69 61 6E 67 @...@....CTriang
000040: 6C 65 00 00 00 00 00 00 00 00 00 80 3F 00 00 le.....?...
000050: 00 00 00 00 00 00 00 00 80 3F 00 00 05 00 43 52 .....?....CR
000060: 65 63 74 33 33 B3 40 9A 99 D9 40 00 00 40 40 00 ect33. @...@...@.
000070: 00 10 41 00 00 07 00 43 53 71 75 61 72 65 CD CC ..A....CSquare..
000080: 4C 40 9A 99 89 40 00 00 C0 40 00 00 07 00 43 53 L@...@...@....CS
000090: 74 72 6F 6B 65 07 00 01 00 00 00 02 00 00 00 03 troke.....
0000A0: 00 00 00 04 00 00 00 05 00 00 00 06 00 00 00 07 .....
0000B0: 00 00 00 00 00 09 00 43 54 72 69 61 6E 67 6C 65 .....CTriangle
0000C0: 00 00 00 00 00 00 00 00 00 00 60 40 00 00 00 00 .....`@....
0000D0: 00 00 00 00 9A 99 A9 40 00 00 05 00 43 52 65 63 .....@....CRec
0000E0: 74 66 66 1E 41 66 66 1E 41 9A 99 99 40 CD CC 1C tff.Aff.A...@...
0000F0: 41 00 00 00 00 .....A.....
```

以傾印工具觀察 `temp.hou` 內容，結果如下（內容解說見 6.2.5 節。請注意，6.12.2 節對於文件格式有更好更完整的設計）：

```
Turbo Dump Version 3.1 Copyright (c) 1988, 1992 Borland International
Display of File TEMP.HOU (37 bytes)

000000: 01 00 0B 00 43 44 57 6F 72 64 41 72 72 61 79 05 ....CDWordArray.
000010: 00 09 00 00 00 07 00 00 00 05 00 00 00 03 00 00 .....
000020: 00 01 00 00 00 .....

```

現在，再次執行測試程式，將上述 `temp.dat` 和 `temp.hou` 讀入並顯示，驗證無誤：

```
D:\pic2\mfclite>mfclappm
(1) .hou (2) .dat
select docType (1 or 2 or 3...) : 2          詢問文件型態。選擇題。
->0          [File/Open] 開舊文件
filename (with ext-name such as 'file.xxx'): temp.dat 詢問型態
CMyDocument::Serialize()          從檔案讀出
->p          強迫發出 WM_PAINT，要求視窗重繪
CMyView::OnDraw()          通知 view 重繪。以下輸出模擬視窗繪圖
CEllipse: x=3, y=3, r1=7, r2=21
CCircle: x=5, y=5, r=7
CTriangle: x1=0, y1=0 x2=1, y2=0 x3=0, y3=1
CRect: x=5.6, y=6.8, height=3, width=9
```

```

CSquare: x=3.2, y=4.3, width=6
CStroke(7): 1 2 3 4 5 6 7
CTriangle: x1=0, y1=0 x2=3.5, y2=0 x3=0, y3=5.3
CRect: x=9.9, y=9.9, height=4.8, width=9.8
->O [File/Open] 開舊文件
filename (with ext-name such as 'file.xxx'): temp.hou 詢問型態
CMyDocument2::Serialize() 從檔案讀出
->p 強迫發出 WM_PAINT, 要求視窗重繪
CMyView2::OnDraw() 通知 view 重繪。以下文字輸出模擬視窗繪圖
9 7 5 3 1 繪出內容
->X 結束程式
(後續還有清理動作與提示, 見 6.12.3 節)

```

重要的不在此刻這個範例程式達成了什麼功能，而在它展現了什麼雛型！MFC(Lite) 所扮演的 application framework 角色，使應⽤程式的初始規模十分規範，因而就技術上而言「程式碼自動產生器」垂手可得（諸君要不要試著在 MFCLite 的基礎上寫個類似 VC++ AppWizard 的工具？好玩的題目）。程式碼自動產生器所產生出來的應用程式（或曰骨幹應用程式），初始規模即具備了所有的（application framework 所設定的）基本功能，程式員不必再花費大量心思去打造這些不在商業競爭範圍內的標準設施。此外，由於物件導向技術和 Template Method 的大量運用，應用程式員只需覆寫某幾個關鍵性的虛擬函式（最典型的例如 CView::OnDraw(), CDocument::Serialize()），便可改變框架模組的行為，把流程牽引到應用模組身上來。這種規律的工⽤作模式使得「輔助開發工具」就技術上而言也垂手可得。事實上，application framework 問世之後，軟體開發產業已經極大規模地依賴它們了。

6.10.1 新增一個顯示方法不用的 view

你已經清楚看到了，在上述測試程式中，以 [Window/New]（熱鍵 'w'）獲得的新視圖（view），總是採用「當時的 active docFrame 視窗所對應的 document 所對應的 docTemplate」內登錄的那個 view class（請參考 6.9.6 節）。這一點可從該命令之處理函式 `CMDIFrameWnd::OnWindowNew()` 的動作清楚獲得驗證：

```
void CMDIFrameWnd::OnWindowNew()
{
    CMDIChildWnd* pActiveChild = MDIGetActive();
    CDocument* pDocument;
    if (pActiveChild == NULL ||
        (pDocument = pActiveChild->GetActiveDocument()) == NULL)
        ...
    // 現在，準備產生一個 new docFrame!
    CDocTemplate* pTemplate = pDocument->GetDocTemplate();
    CFrameWnd* pFrame = pTemplate->CreateNewFrame(pDocument,
                                                pActiveChild);
    ...
}
```

這種工作模式，也正是圖 6-33 的繪圖依據。

6.9.6 節末尾給了一個訴求，希望突破上述限制，讓新產生的 view 有不一樣的文件表現手法。正規作法（MFC 內）是以靜態分裂視窗（static splitter）在具備 MDI 性質的 `CChildFrame` 視窗中分別產生多個子視窗（或曰窗口，panes），並令各個窗口擁有各自的 view class（如圖 6-32a。技術細節詳見《深入淺出 MFC》2e，第 13 章）。但 `MFCLite` 不支援 `CChildFrame` 的 MDI 性質，所以此法無從施行於 `MFCLite` 身上。

另一個（非正規）作法是，依樣畫葫蘆地撰寫一個 `CMainFrame::OnWindowNew()`，並利用訊息映射表，將 [Window/New] 命令訊息牽引到新函式身上（圖 6-37）。新函式完全模仿 `CMDIFrameWnd::OnWindowNew()`，只不過取 document template 時，不再取「active docFrame 視窗所對應的 document 所對應的 docTemplate」，而是自行準備另一個 document template，並在其中設定你想要的 view class。

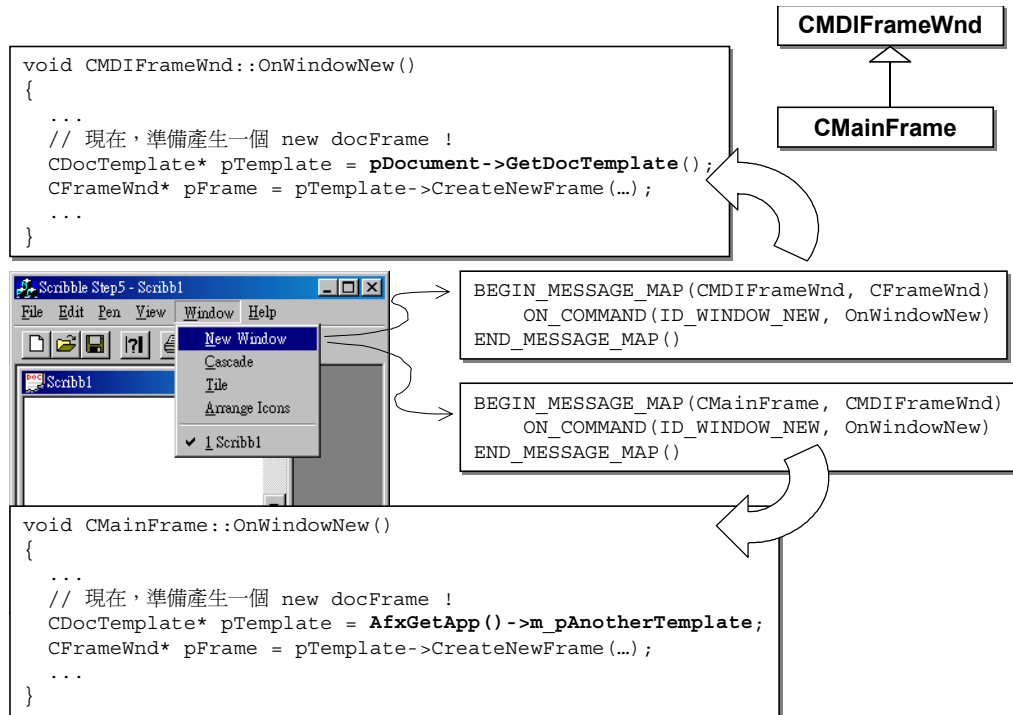


圖 6-37 由於 CMainFrame 衍生自 CMDIFrameWnd，
如果兩者的訊息映射表都攔截同一個命令訊息，前者會攔截到。

以下是本節測試程式的修改步驟（全部都在應用模組端，也就是在本例的 `mfclapp.h` 和 `mfclapp.cpp`）：

- 新增一個 `CMyView3`，令它成為 `CMyDocument2` 的 `friend`，並令其 `OnDraw()` 函式的輸出手法為：針對 `CMyDocument2` 內的 `pDArray` 的每個元素，包裹一個中括號 ('[]') 然後才輸出。
- 令 `CMyView3::OnLButtonUp()` 函式的元素添加方式為：每次添加 8,6,4,2,0 等 5 個元素，以別於 `CMyView2::OnLButtonUp()` 的元素添加方式：每次添加 9,7,5,3,1 等 5 個元素。這麼做純粹只是為了在執行時特別彰顯不同罷了。

- 令 `CMyView3::GetDocument()` 函式為：

```
ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CMyDocument2)));
return (CMyDocument2*)m_pDocument;
```

- 添加一個新的 resource：

```
#define IDR_MYDOCTYPE3 string(".hoo")
```

- 為 `CMyWinApp` 添加一個 `public` 成員：

```
CMultiDocTemplate* m_pDocTemplate3;
```

並在 `CMyWinApp::InitInstance()` 中添加一個 `docTemplate` 如下：

```
m_pDocTemplate3 = new CMultiDocTemplate(
    IDR_MYDOCTYPE3,          // .hoo
    RUNTIME_CLASS(CMyDocument2),
    RUNTIME_CLASS(CChildFrame),
    RUNTIME_CLASS(CMyView3)
);
AddDocTemplate(m_pDocTemplate3);
// 注意，以上將 CMyDocument2 和 CMyView3 綁在一起。
```

- 在表單中增加一個 [Window/New-with-CMyView3] 命令，令其熱鍵為 '3'。並令 `CMainFrame` 攔截之。
- 在 `CMainFrame` 中增加一個 `OnWindowNew()` 函式。注意，此非虛擬函式，與 `CMDIFrameWnd` 中的同名函式之間並非覆寫（`override`）關係，而是遮掩（`hide`）關係，見 2.x 節。`CMainFrame::OnWindowNew()` 主要內容如下：

```
void CMainFrame::OnWindowNew()
{
    CMDIChildWnd* pActiveChild = MDIGetActive();
    CDocument* pDocument;
    if (pActiveChild == NULL ||
        (pDocument = pActiveChild->GetActiveDocument()) == NULL)
        ...
    // 現在，準備產生一個 new docFrame !
    CDocTemplate* pTemplate =
```

```

        ((CMyWinApp*)AfxGetApp())->m_pDocTemplate3; // 注意這行
CFrameWnd* pFrame = pTemplate->CreateNewFrame(pDocument,
                                                pActiveChild
        ...
    }

```

下面是新程式的執行結果：

```

D:\pic2\mfclite>mfclappm
(1) .hoo (2) .hou (3) .dat
select docType (1 or 2 or 3...) : 2          詢問文件型態。選擇題。
->W          [Window/New] 開新視圖（根據目前文件）
->3          [Window/New-with-CMyView3] 開新視圖
->u          [LButtonUp] 滑鼠左鍵放開。本例視同 UI 輸入完畢
CMyView3::OnLButtonUp()
CMyView2::OnDraw()          目前有三個 views，通知所有 non-active views 重繪
8 6 4 2 0                  non-active view 1 重繪結果
CMyView2::OnDraw()
8 6 4 2 0                  non-active view 2 重繪結果
->P          強迫發出 WM_PAINT，要求 active view 重繪
CMyView3::OnDraw()
[8] [6] [4] [2] [0]        active view 重繪結果
->0          [App/Status] 顯示目前狀態。
-----
windows total: 7
1 2 3 4 5 6 7
active frame/view hwnd: 6,7
DocTemplates count : 3
Documents count : 1
-----
+m_IDResource: [.hoo]
doc:CMyDocument2 docFrame:CChildFrame view:CMyView3
+m_IDResource: [.hou]
doc:CMyDocument2 docFrame:CChildFrame view:CMyView2
  Title= untitled File=
    View hwnd= 3    CMyView2
    View hwnd= 5    CMyView2
  * View hwnd= 7    CMyView3
+m_IDResource: [.dat]
doc:CMyDocument docFrame:CChildFrame view:CMyView
-----
->S          [File/Save] 存檔
filename (with ext-name such as 'file.hou'): temp1.hou 詢問檔名
CMyDocument2::Serialize() 寫入檔案（物件永續）
->X          結束程式
          （後續還有清理動作與提示，見 6.12.3 節）

```

利用這個機會，我們也等於複習了在諸如 MFC(Lite) 這樣的 application framework 控制之下，應用程式的開發模式。

上述作法在 MFCLite 和 MFC 中都可行，但由於 MFCLite 將 GUI 完全簡化掉了，上述修改因而出現⁷ – 個危機：按說，表單命令 [Window/New-with-CMyView3] 應該只在「active document 為 CMyDocument2」的情況下才能起作用，而我們對此卻沒有任何防護措施。假設使用者執行上述程式時，在「active document 為 CMyDocument」的情況下按下熱鍵 '3'，程式不會阻攔；稍後當按下熱鍵 'p' 打算重繪文件內容時，會出現：

> Assertion failed:
m_pDocument->IsKindOf((&CMyDocument2::classCMyDocument2))

這是因為上述的 WM_PAINT 喚起 CMyView3::OnDraw()，幸好後者對其所取得的文件，利用 IsKindOf() 做了型態上的檢驗：

```
CMyDocument2* CMyView3::GetDocument() const
{
    ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CMyDocument2)));
    return (CMyDocument2*)m_pDocument;
}
```

事前預防當然比事後修補好的好，更比程式不正常結束然後請使用者寄來一份錯誤報告好得多（別找罵捱了吧☹）。在正式的 application framework 中，每當針對一個 document 產生一個 docFrame 視窗，也就是 CDocTemplate::CreateNewFrame() 被喚起時，其內呼叫 CFrameWnd::LoadFrame()，後者便會載入相應的 GUI，包括一套表單。於是，不同的文件型態有不同的表單，也就可以避免張冠李戴的情況。MFCLite 也可以模擬這一點，留給讀者做練習。

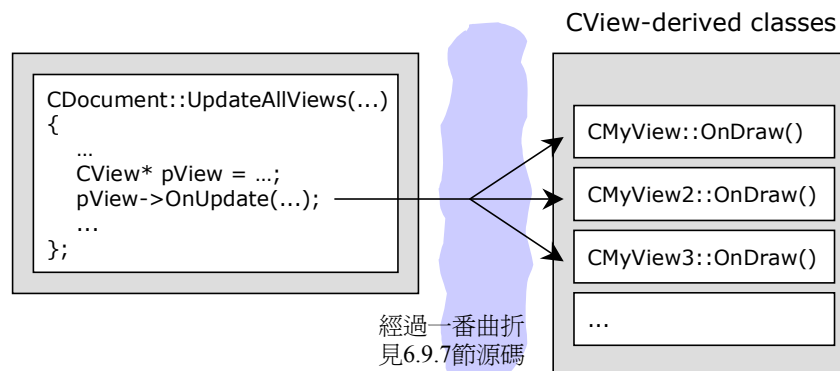
最後一個問題是，從圖 6-33 的架構來看，如果（依照上述手法）為某個 document 產生 view 時，竟然可以採用該 document 所隸屬之 docTemplate 以外的 docTemplate，那麼是不是每一個 view 身上必須記錄著對應的 view class？否則 view 如何知道其所對應的 view class 呢（顯然已經不能由 view→document→docTemplate 的方式來推導了）？唔，我知道你正在想「subscribe-notify 協定」的問題。這點不勞操心，因為，如同 6.9.7 節的源碼顯示：

```

void CDocument::UpdateAllViews(CView* pSender,
                               LPARAM lHint /* 0L */,
                               CObject* pHint /* NULL */)
{
    // 巡訪所有的 views，一一呼叫其 OnUpdate()。
    POSITION pos = m_viewList.GetHeadPosition();
    while (pos != NULL)
    {
        CView* pView = (CView*)m_viewList.GetNext(pos);
        if (pView != pSender) // 自己不通知自己（從常理看，沒有必要）
            pView->OnUpdate(pSender, lHint, pHint);
    }
}

```

`pView->OnUpdate(...)` 會根據 `pView` 實際所指的 `CView`-derived class object，喚起相應的 `OnUpdate()`，進而喚起相應的 `OnDraw()`。`CView`-derived class 通常沒有必要覆寫虛擬函式 `OnUpdate()`，只需覆寫最後終點站 `OnDraw()` 即可。這又是典型的 **Template Method** 手法：



6.1.1 除錯機制

程式開發過程中，錯誤不可能不發生。application framework 如此這般的複雜機制，更不可能在開發過程中從頭順利到尾。是的，即使有 MFC 源碼在旁輔助，我也在開發 MFCLite 的過程中一再出錯，一再絞盡腦汁，一再修正。

除錯，不能夠漫無章法地頭痛醫頭腳痛醫腳。一個好的程式開發人員，必須在架構程式的同時，就想好除錯機制。以 MFC 為例，它就提供了一個名為 `afxDump` 的 global `CDumpContext` object，你可以將任何物件往它身上招呼，像這樣：

```
afxDump.SetDepth(1); // 0 => this object, 1 => children objects.
CDocManager* pDocManager = AfxGetApp()->m_pDocManager;
afxDump << *pDocManager;
```

這便會在 VC++ 整合環境的除錯視窗中顯示應用程式的 `m_pDocManager`（如圖 6-33）的完整內容，包括文件型態個數及各文件型態所記錄的重要資訊，以及根據每種文件型態所開啓的文件個數、對應的 `views`... 等等。諸如此類的除錯機制所提供的寶貴資訊，不僅對應用程式員極為有用，對於 MFC application framework 開發人員也極具價值。

MFCLite 所賴以生存的，無非是穩固無誤的三大基礎設施，以及穩固無誤的訊息映射表和訊息繞行機制。爲了確保這些東西的確能夠如預期般地組織起來，把它們列印出來是最理想的辦法。爲此，MFCLite 提供了一個除錯熱鍵 'G'，按下它就會輸出一個 "debug.txt"，記錄整個訊息映射表和所有 `CRuntimeClass` objects。

6.1.1.1 追蹤訊息映射表與訊息繞行路線

訊息映射表，一如圖 6-27 所示，其實不是個「表」，而是棵「樹」，只不過實作上並非以正規的「樹」來實現。

爲了走訪整個訊息映射表，並印證走訪路徑一如圖 6-26 所示規則，最好的辦法就是讓除錯熱鍵觸發一個除錯專用的 `WM_COMMAND`（這樣才有可能全「樹」走透透；唯有命令訊息才有繞行現象，一般訊息只是直線上溯），並且不在任何地點攔截它。當此特殊命令被訊息繞行機制推動流過所有路徑，我們便在訊息比對動作中

於察知此一除錯命令時，將訊息映射表的每一筆項目輸出到除錯檔。下面是相關程式碼的節錄，灰色網底部分，與目前的討論有關：

```
ofstream g_debugfile("debug.txt");
// 注意，由於這會自動開啓，所以如果某次執行時未曾按下 debug 命令，
// 原有的（以前執行所留下的）debug.txt 會被清空。

BOOL GetMessage(MSG* pMsg)
{
    ...
    switch (c) {
        case 'G' : // debu'G'
            // 注意，此鍵在程式過程中可被按多次，因為第一次按下完成任務後，
            // 檔案會被關閉。下次再按已無效果。
            pMsg->hWnd = JJGetMainFrame(); // 命令必來自 MainFrame
            pMsg->nMsg = WM_COMMAND;
            pMsg->wParam = ID_MFCLITE_DEBUG;
            return TRUE;
    }
}

const AFX_MSGMAP_ENTRY* AfxFindMessageEntry(
    const AFX_MSGMAP_ENTRY* lpEntry,
    UINT nMsg, UINT nCode, UINT nID)
{
    while (lpEntry->nSig != AfxSig_end) // 走訪整個 _messageEntries[]
    {
        if (nID == ID_MFCLITE_DEBUG)
        {
            g_debugfile << hex;
            g_debugfile << lpEntry->nMessage << '\t'
                << lpEntry->nCode << '\t'
                << lpEntry->nID << '\t'
                << lpEntry->nLastID << '\t'
                << lpEntry->nSig << '\t'
                // << lpEntry->pfn << '\t'
                // CB5: 'operator<<' not implemented in type 'ostream'
                // for arguments of type 'void (CCmdTarget::*)()'
                << endl;
        }
        if (lpEntry->nMessage == nMsg && lpEntry->nCode == nCode &&
            nID >= lpEntry->nID && nID <= lpEntry->nLastID)
            return lpEntry; // 找到了

        lpEntry++; // 下一筆。array 才能允許如此寫法。
    }

    if (nID == ID_MFCLITE_DEBUG) {
        // 輸出 _messageEntries[] 的最後一筆 (AfxSig_end)
        g_debugfile << hex;
    }
}
```

```

    a debugfile << lpEntry->nMessage      << '\t'
               << lpEntry->nCode         << '\t'
               << lpEntry->nID           << '\t'
               << lpEntry->nLastID       << '\t'
               << lpEntry->nSig          << '\t'
               // << lpEntry->pfn         << '\t'
               // CB5: 'operator<<' not implemented in type 'ostream'
               // for arguments of type 'void (CCmdTarget::*)()'
               << endl;
    }
    return NULL;    // not found
}

```

這個除錯命令未被攔阻地歷經所有繞送途徑，最終到達 `::DefWindowProc()`。那裡會將 `CRuntimeClass` objects 一併寫入除錯檔（詳見 6.11.2 節），然後關閉檔案，結束整個除錯任務。

為什麼我們不能像（下一節所討論的）對付 `CRuntimeClass` objects tree 一樣的作法，在某個定點取訊息映射表的「起點」來走訪整個映射構造呢？因為 `CRuntimeClass` objects tree 是以 linked-list 完成，並維護一個起頭（head），而訊息映射表卻真正是個 tree，我們沒有現成的走訪機制。而且我們所擁有的訊息映射表「起點」，只是其根節點（root），缺乏向下推動的連結（links）；如果要由下往上推，起點在於四個可能的葉節點（分別是 view, document, frame, app），而那卻是 MFCLite 無法獲得的（因為葉節點在應用模組端，不在框架模組端）。因此，以訊息繞送機制做為現成的走訪機制，是最理想的辦法了。

請注意，6.8.4 節說過，加入 MDI 介面之後，MFC(Lite) 的訊息繞送途徑變得複雜許多；`CMDIFrameWnd::OnCommand()` 和 `CMDIFrameWnd::OnCmdMsg()` 聯手打造的三套唧送（pumping）策略中，有不少重複路徑，這些重複路徑也都忠實地被記錄於 "debug.txt"。以下便是一份摘自 "debug.txt" 的訊息繞行完整歷程，搭配圖 6-38 便可驗證訊息繞行的正確性。請注意第 6 欄位 pfn 是個 pointer to member function，C++ 標準輸出裝置 cout 並未定義 pointer to member function 的輸出行為。某些編譯器（例如 VC6 和 GCC）會讓它過關，但輸出結果（如下）並不正確。下表是 VC++ 版本的執行結果，允許 pointer to member function 輸出到 cout，但結果皆為 0 或 1。

nMsg	nCode	nID	nLastID	nSig	pfn	補充說明
201	0	0	0	31	1	CMyView
202	0	0	0	31	1	
200	0	0	0	31	1	
0	0	0	0	0	0	
f	0	0	0	c	1	CView
1	0	0	0	9	1	
0	0	0	0	0	0	
1	0	0	0	9	1	CWnd
0	0	0	0	0	0	
0	0	0	0	0	0	
0	0	0	0	0	0	CCmdTarget
111	0	e121	e121	c	1	CMyDocument
0	0	0	0	0	0	
111	0	e104	e104	c	1	
111	0	e103	e103	c	1	CDocument
0	0	0	0	0	0	
0	0	0	0	0	0	
0	0	0	0	0	0	CCmdTarget
0	0	0	0	0	0	CCmdTarget
0	0	0	0	0	0	CChildFrame
1	0	0	0	9	1	CMDIChildWnd
0	0	0	0	0	0	
0	0	0	0	0	0	
1	0	0	0	9	1	CFrameWnd
0	0	0	0	0	0	
0	0	0	0	0	0	
1	0	0	0	9	1	CWnd
0	0	0	0	0	0	
0	0	0	0	0	0	
0	0	0	0	0	0	CCmdTarget
111	0	e100	e100	c	1	CMyWinApp
111	0	e101	e101	c	1	
111	0	e140	e140	c	1	
111	0	e122	e122	c	1	
111	0	e123	e123	c	1	
0	0	0	0	0	0	
0	0	0	0	0	0	CWinApp
0	0	0	0	0	0	CCmdTarget
以上為第一條唧送 (pump) 路線						
201	0	0	0	31	1	CMyView
202	0	0	0	31	1	
200	0	0	0	31	1	
0	0	0	0	0	0	
f	0	0	0	c	1	CView
1	0	0	0	9	1	
0	0	0	0	0	0	
1	0	0	0	9	1	CWnd
0	0	0	0	0	0	
0	0	0	0	0	0	
0	0	0	0	0	0	CCmdTarget
111	0	e121	e121	c	1	CMyDocument
0	0	0	0	0	0	
111	0	e104	e104	c	1	

111	0	e103	e103	c	1	
0	0	0	0	0	0	
0	0	0	0	0	0	CCmdTarget
0	0	0	0	0	0	CCmdTarget
0	0	0	0	0	0	CChildFrame
1	0	0	0	9	1	CMDIChildWnd
0	0	0	0	0	0	
1	0	0	0	9	1	CFrameWnd
0	0	0	0	0	0	
1	0	0	0	9	1	CWnd
0	0	0	0	0	0	
0	0	0	0	0	0	CCmdTarget
111	0	e100	e100	c	1	CMyWinApp
111	0	e101	e101	c	1	
111	0	e140	e140	c	1	
111	0	e122	e122	c	1	
111	0	e123	e123	c	1	
0	0	0	0	0	0	
0	0	0	0	0	0	CWinApp
0	0	0	0	0	0	CCmdTarget
以上為第二條唧送 (pump) 路線						
1	0	0	0	9	1	CMainFrame
0	0	0	0	0	0	
111	0	e130	e130	c	1	CMDIFrameWnd
0	0	0	0	0	0	
1	0	0	0	9	1	CFrameWnd
0	0	0	0	0	0	
1	0	0	0	9	1	CWnd
0	0	0	0	0	0	
0	0	0	0	0	0	CCmdTarget
111	0	e100	e100	c	1	CMyWinApp
111	0	e101	e101	c	1	
111	0	e140	e140	c	1	
111	0	e122	e122	c	1	
111	0	e123	e123	c	1	
0	0	0	0	0	0	
0	0	0	0	0	0	CWinApp
0	0	0	0	0	0	CCmdTarget
以上為第三條唧送 (pump) 路線						

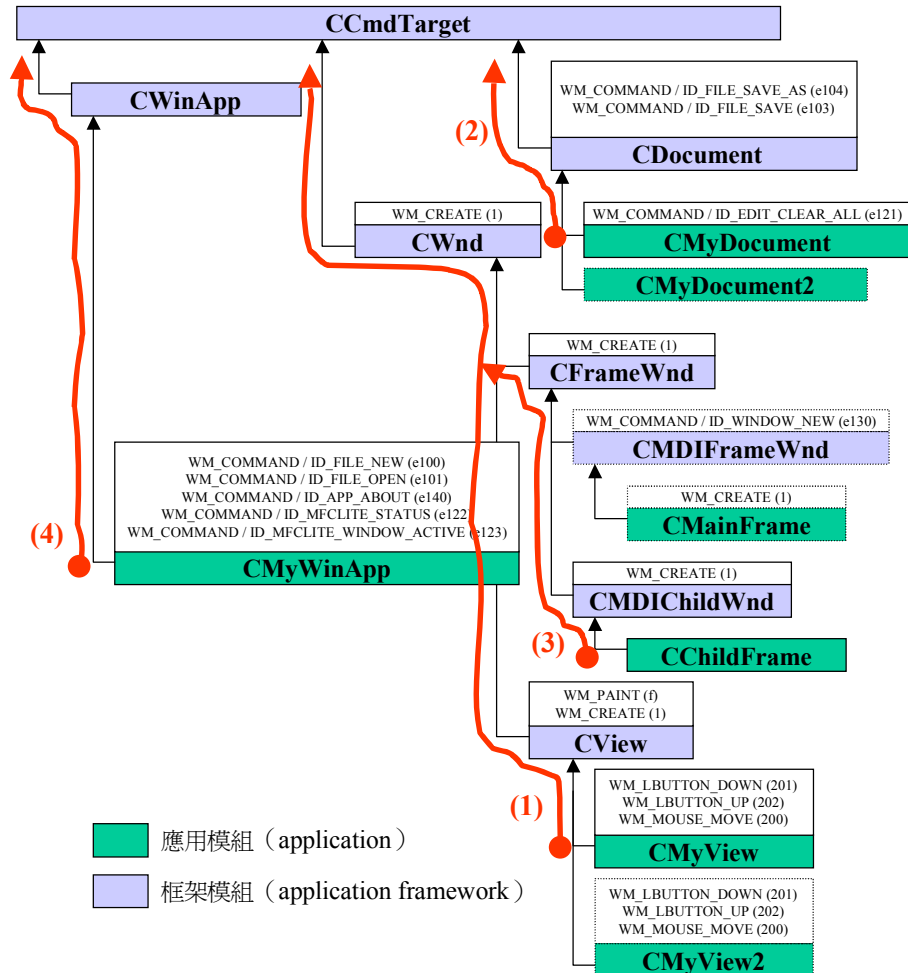


圖 6-38a MFCLApp + MFCLite 共同組成的訊息映射「樹」。每個 classes 名稱上方標示的便是本例之中該 class 的訊息映射條目，並附實際定義之數值。紅線代表訊息繞行路線。在 `CMDIFrameWnd::OnCommand()` 和 `CMDIFrameWnd::OnCmdMsg()` 聯手打造的三套唧送 (pumping) 策略中，以本例而言，前兩套策略所形成的繞行路線完全相同，分別按圖中(1),(2),(3),(4) 的次序流動。

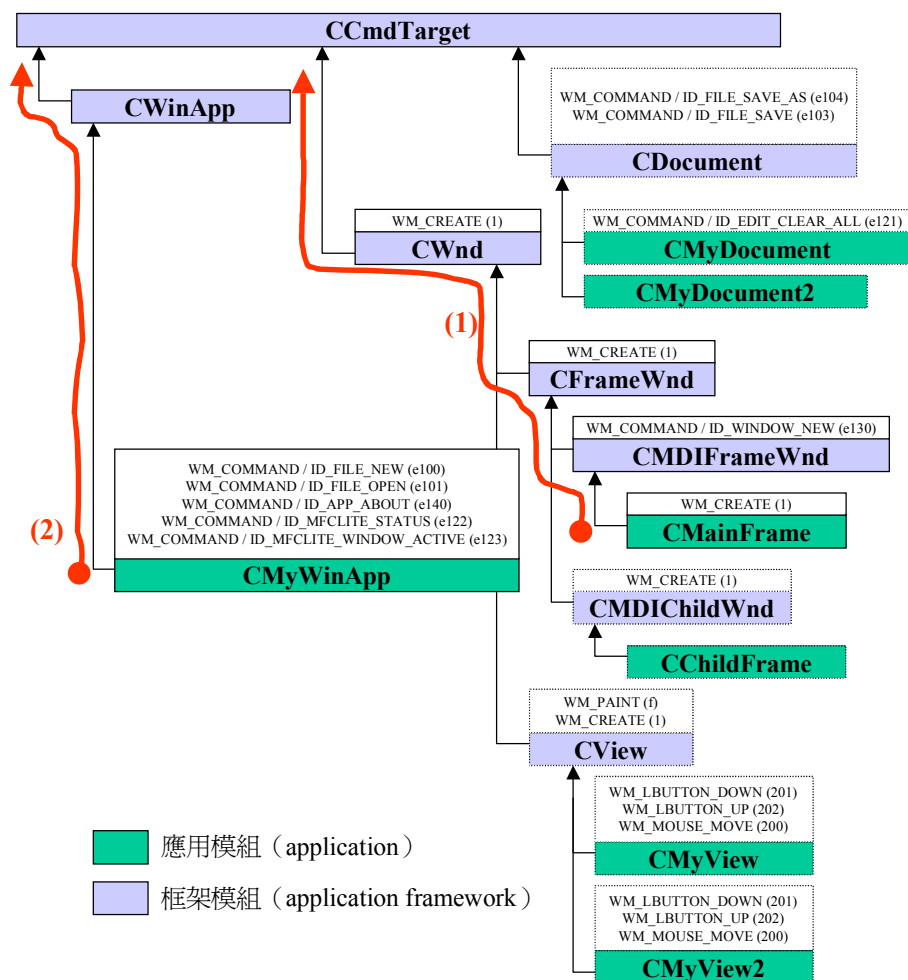


圖 6-38b 紅線代表本例第三套唧送 (pumping) 策略。按圖中(1),(2) 次序流動。

6.11.2 追蹤 CRuntimeClass 類別型錄網

走訪 CRuntimeClass objects tree 非常簡單，因為它是以 linked-list 完成，並維護一個起點。只要取得起點，走訪 linked-list 是件輕鬆事兒。我把這個任務交給除錯命令最終一定會到達的地點：DefWindowProc()，見以下灰色部分。請注意，這個函式必須傳回 0 或 1，告訴其呼叫者，流入的訊息是否已被處理完全。傳回值嚴重關係到整個訊息機制的表現是否正常，各種判斷及註解直接列於函式之內。

```
LRESULT DefWindowProc(HWND hWnd, UINT nMsg,
                      WPARAM wParam, LPARAM lParam)
{
    // WM_MFCLITE_DEBUG 最終會到這裡，此時 debug.txt 已有
    // MsgMap 的完整記錄，應將 CRuntimeClass tree 也輸出到檔案去。
    if (wParam == ID_MFCLITE_DEBUG && hWnd == 1) {
        // 一定要加第二個判斷式
        // hWnd!=1 意味尚非來自 MainFrame，那麼就要讓 debug 訊息經歷
        // CMDIFrameWnd::OnCmdMsg() 的 2nd & 3rd pumping，否則不完整
        // （未能走完所有可能的繞行路線）。
        CRuntimeClass* p = CRuntimeClass::pFirstClass;
        while(p) {
            g_debugfile << p->m_lpszClassName          << '\t'
                        << p->m_nObjectSize            << '\t'
                        << p->m_wSchema                << '\t'
                        << p->m_pfnCreateObject         << '\t'
                        << p->m_pBaseClass              << '\t'
                        << p->m_pNextClass              << '\t'
                        << endl;
            p = p->m_pNextClass;
        }

        // 除錯檔是全域物件，程式開始時便開啓，程式結束後會自動關閉。
        // 這裡強迫關閉
        g_debugfile.close();
        cout << "output the debug info. to \"debug.txt\" OK. " << endl;
    }

    // 一旦訊息流入此地，只有二種情況才代表處理完畢（處理完全）：
    // (1) 訊息不為 WM_COMMAND
    // (2) 訊息是 WM_COMMAND|ID_MFCLITE_DEBUG，且來自 MainFrame
    //     （表示它已走完所有路線）
    if (nMsg!=WM_COMMAND || (wParam==ID_MFCLITE_DEBUG && hWnd==1))
        return 1;
    else
        return 0;
}
```


下面便是摘自 "debug.txt" 的完整 CRuntimeClass objects 類別型錄網。這份資料顯示 6.10 節測試程式內被納入類別型錄網的所有 classes (含 MFCLite classes, 以藍色標示), 形如圖 6-6。只要曾經使用三大基礎建設巨集, 就會被納入類別型錄網。採用第二層巨集 (動態生成) 者, pfn 會有非零值, 採用第三層巨集 (次第檔案讀寫) 者, 序號 (scheme no.) 會有 ffff (-1) 以外的值。

Class 名稱	大小	序號	pfn	base	next
CanotherShapeList	14	0	00415EF1	0043B160	0043B230
CStroke	18	0	00415E28	0043B160	0043B210
CTriangle	1c	0	00415D53	0043B160	0043B1E8
CEllipse	14	0	00415C82	0043B1C0	0043B1C0
CCircle	10	0	00415BB3	0043B160	0043B1A0
CRect	14	0	00415AE2	0043B180	0043B180
CSquare	10	0	00415A13	0043B160	0043B160
CShape	4	ffff	0041597E	0043A048	0043A048
CObject	4	ffff	00000000	00000000	0043A028
CFile	30	ffff	00000000	0043A048	0043A008
CDocManager	18	ffff	00000000	0043A048	00439FE0
CPtrList	14	ffff	00000000	0043A048	00439FB8
CDWordArray	14	1	00408440	0043A048	00439F90
CObList	10	1	00408377	0043A048	00439F70
CView	c	ffff	00000000	00439ED8	00439F50
CMDIChildWnd	c	ffff	004082B4	00439F00	00439F28
CMDIFrameWnd	c	ffff	0040821F	00439F00	00439F00
CFrameWnd	c	ffff	0040818A	00439ED8	00439ED8
CWnd	8	ffff	004080F5	00439DF0	00439EB8
CDocument	3c	ffff	00000000	00439DF0	00439E90
CMultiDocTemplate	34	ffff	00000000	00439E60	00439E60
CDocTemplate	20	ffff	00000000	00439DF0	00439E38
CWinApp	c	ffff	00000000	00439E18	00439E18
CWinThread	4	ffff	00000000	00439DF0	00439DF0
CCmdTarget	4	ffff	00000000	0043A048	00439260
CMyDocument2	40	ffff	004013A6	00439EB8	00439238
CMyView2	c	ffff	00401311	00439F70	00439210
CChildFrame	c	ffff	0040127C	00439F50	004391E8
CMainFrame	c	ffff	004011E7	00439F28	004391C0
CMyDocument	4c	ffff	00401152	00439EB8	00439198
CMyView	c	ffff	004010BD	00439F70	00000000

6.11.3 追蹤 document-view 現狀

由於 MFCLite 不支援 GUI，程式執行時心中有視窗而實際螢幕上並無視窗，因此必須提供使用者隨時掌握程式現狀的功能。MFCLite 以熱鍵 '0' 啟動現狀觀察功能。下面是相關程式碼。

```
BEGIN_MESSAGE_MAP(CWinApp, CCmdTarget)
    ON_COMMAND(ID_MFCLITE_STATUS, OnAppShowStatus)
    ON_COMMAND(ID_MFCLITE_CHANGE_ACTIVE, OnAppChangeActive)
    ON_COMMAND(ID_APP_EXIT, OnAppExit)
END_MESSAGE_MAP()
```

```
void CWinApp::OnAppShowStatus()    //hotkey:0
// 此函式顯示程式的整個視窗狀態，包括 mainFrame/Docs/docFrames/Views。
// 非 MFC 正常範圍內的動作，但對「MFCLite 缺乏 GUI」之事實而言，乃為必要。
{
    JJShowStatus();
}
```

```
void CWinApp::OnAppChangeActive() //hotkey:1
// 此函式模擬 active window 的切換。非 MFC 正常範圍內的動作，
// 但對「MFCLite 缺乏 GUI」之事實而言，乃為必要。
// 此函式將「用來儲存目前所有 doc-frame 視窗和 view 視窗」之 map 視為
// 環狀，次第切換。
{
    JJSetNextActive();
    OnAppShowStatus(); // to display
}
```

按說熱鍵 '0' 和熱鍵 '1' 不該出現於應用模組端的資源描述檔 (.rc)，因為它是 MFCLite 內建功能，出現於表單 (menu) 實在突兀。但訊息迴圈取得的不是命令訊息就是視窗訊息，而我可不想為這兩個功能另設計兩個對應的自定訊息，更不想再為它們設計兩個對應的 ON_WM_XXX() 巨集 (6.7.3 節)。將它們視為命令訊息 (WM_COMMAND) 是為了圖個方便，可使用現成的巨集加入訊息映射表中。實際攫取訊息時，還是應該以對待視窗訊息的方式來攫取它們（而非如 6.6.3 節一般在表單識別碼對映表格 g_MenuTable 中辨識它們）：

```
BOOL GetMessage(MSG* pMsg)
{
    ...
    switch (c) {
        case '0' : // System HotKey '0' : show Status
```

```
pMsg->hWnd = JJGetMainFrame(); // 命令必來自 MainFrame
pMsg->nMsg = WM_COMMAND;
pMsg->wParam = ID_MFCLITE_STATUS;
break;

case '1' : // System HotKey '1' : Change Active docFrame and View
pMsg->hWnd = JJGetMainFrame(); // 命令必來自 MainFrame
pMsg->nMsg = WM_COMMAND;
pMsg->wParam = ID_MFCLITE_CHANGE_ACTIVE;
break;

...
}
```

爲什麼要額外在框架模組端設計出一個 `CStatus` 呢？爲什麼不在以上訊息處理常式內直接取出所有 `document-view` 現狀呢？原因是，許多資料被設計爲各個 `MFCLite` classes 的 `private` 或 `protected` 成員，而 `MFCLite` 又未提供相關函式以供取得這些成員內容。這麼一來外界就不可能取得這些資料。所以我讓 `CStatus` 成爲所有必要之 `MFCLite` classes 的 `friend`，於是其建構式便可取得任何想要的資訊。這種作法乃是將除錯（資料顯示）機制設計爲 `MFCLite` 的本質，和 `MFC` 的 `afxDump` 頗有相同的意味。

6.11.4 呼叫堆疊 (Call Stack)

任何除錯機制都應該包含 `call stack`，也就是「一個函式呼叫另一個函式…」的喚起程序。在沒有專業除錯器協助的情況下，MFCLite 唯一能夠顯現 `call stack` 的方法就是，在每個函式的進入點列印相關文字，例如：

```
#define TRACE1 printf          // 除錯用
#define TRACEmr printf        // 除錯用 for msg routing

BOOL CWnd::PreCreateWindow(CREATESTRUCT& cs)
{
    TRACE1("CWnd::PreCreateWindow() \n");
    ...
}

LRESULT CWnd::WindowProc(UINT nMsg, WPARAM wParam, LPARAM lParam)
{
    TRACEmr("CWnd::WindowProc() \n");
    ...
}
```

如果程式撰寫之初即能清楚規劃以不同的 `TRACEx()` 表現不同類型的函式，將來要隱藏或顯現特定某一類函式的 `call stack`，就很方便。例如我令所有與訊息繞行有關的函式皆使用 `TRACEmr()`，除錯時便可輕易將所有與訊息繞行有關的函式的 `call stack` 開啓或關閉（令 `TRACEmr()` 有作用或註解掉就行）。

6.11.5 土法鍊鋼與現代化器械

再怎麼說土法鍊鋼都沒有辦法和現代化機械的效率和功能相比。上述的除錯裝置都只是一種設計思考上的啓發，真正商業應用軟體的除錯，還是要得力於專業除錯器。6.1.2 節的「整合開發環境」一節，已經告訴你如何將 MFCLite 置入 VC++，這麼一來不但擁有步進 (step by step) 功能，也可以隨時觀看變數現值、觀看 `call stack`。整合開發環境的另一個好處是，我們可以輕鬆瀏覽程式內的所有 `classes` 的所有 `members`，也能夠輕鬆觀察其所對應的源碼。對於程式開發的流暢度以及整個 `classes` 階層體系的掌握度，確實有非常大的幫助。你唯一會抱怨的是，資料太多，螢幕不夠大☺。

6.12 強化 MFCLite

MFCLite3 已經達到了一個 application framework 的核心目標。

還可以更好！

請試著修改 MFCLite，讓它更完善。有些題目與 application framework 的核心技術無關，但是可以讓你練習操作、改寫 MFCLite；有些題目可以讓 MFCLite 更具彈性；還有些題目關係到更深一層的視窗管理技術。

- mfclApp.cpp 並未實現 [Edit|Clear All] 功能，試完成之。
- 試阻止使用者於執行過程中重複按下 ‘G’ 鍵（那會造成除錯檔內容重複，無意義又影響閱讀）。
- 目前的 Help 寫死於框架模組中。試模仿 VC 整合環境，讓應用程式員將 Help 資訊寫於 resource.rc 中，於程式執行期讀入。resource.rc 的格式既然擴充，rc.cpp 亦必須對應修改。
- 找出共同規則，使 MFCLite 接受熱鍵輸入時，不必以 switch-case 一一寫出它所支援的每一個訊息熱鍵，而以更好的編程方式取代。
- 目前版本只接受一份表單（menu），換言之並未實現「一種 document 有一套對應而專屬的 UI」。因此在一個支援多重文件（multiple documents）的程式中，針對 document A 而設的 menu items，就可能被終端使用者誤用於 document B 身上（如 6.10.1 節所述）。解決之道是落實「一種 document 對應一套專屬的 UI（menu）」。首先應從 document 的資源識別碼 IDR_xxx 著手，其次必須允許 resource.rc 出現多份表單；RC.CPP 必須對應修改。最後，如何載入表單並在適當時機切換，是個大工程，考驗你對整個 MFCLite 的掌握度。
- 目前版本只允許 mainFrame 有子視窗，不允許 docFrame（childFrame）再有子視窗。試模擬 MFC 多層子視窗的架構。首先你必須了解 MDI 的底層機制（什麼是 MDI-Frame, MDI-Client, MDI-Children），然後閱讀 MFC 相關源碼，包括 CMDIFrameWnd, CMDIChildWnd, CSplitterWnd，然後才有動手（並成功😊）的可能¹³。

¹³ 《深入淺出 MFC》2e 第 13 章可以為此題目帶來一些幫助。

- 熱鍵輸入時，試找出不同編譯平台中適當的 CRT (C Runtime) 函式，讓使用者在輸入過程時感到舒服。目前版本的缺點，在 Visual C++ 和 C++Builder 版本中，熱鍵輸入時不必按下<Enter> (很好)，但該熱鍵字元會被保留於鍵盤緩衝區中，影響接下來的文字輸入 (例如檔案名稱的輸入)。至於目前的 GCC 版本，熱鍵輸入時必須按下<Enter> (不方便)，不過卻也因此清空了鍵盤緩衝區，不會影響接下來的文字輸入。
- MFCLite 應用程式執行時，需讀取其相應資源檔 resource.res。試模擬真正的 Windows RC.EXE (資源編譯器) 的第二項功能：將可執行檔 (.exe) 和資源檔 (.res) 結合為一個獨立的執行檔 (.exe)。這一點非常困難，涉及可執行檔格式。
- CPtrList 的實作手法十分精巧 (6.12.1 節詳述)，但若論及方便性與功能性，我們可以在 MFCLite 中以 C++ 標準程式庫所提供之 list<> 取而代之。後者 (以及 C++標準程式庫的所有容器) 的唯一缺點是不支援 persistence，但 MFC 的 CPtrList 本來也就不支援 persistence¹⁴，所以這一點無分軒輊。為什麼我不一開始就在 MFCLite3 中採用 list<> 來取代 CPtrList 呢？因為兩者提供的介面不同，而 MFCLite 延續 MFC 的風格，程式中處處使用 CPtrList 介面，所以乾脆還是完成一個 CPtrList 好了。你可以試試 **Adaptor** 樣式 (見 7.3 節)：仍舊寫出一個如 6.2.5 節的 CPtrList 介面，並在其中每一個 (或大部份) member functions 中轉呼叫 list 的 member functions，如此一來既維護了 CPtrList 使用者 (MFCLite) 的程式碼不變，又把實際機能轉由 list 承擔。當然，就技術而言，CPtrList 的實作手法 (6.12.1 節) 是一個很好的啟發和學習對象。
- 模擬特定視窗 (例如 active 視窗) 的關閉動作。提示：從增加一個熱鍵 (關閉現行視窗，active window) 想起；重點之一是清理視窗對應的 document。這個性質已實現於 6.12.3 節。
- 根據 x.x 節所說，儘量不要使用 #define 來定義常數值，應該改用 const。它所帶來的好處在我們使用除錯器協助對 MFCLite 除錯時，特別明顯。試將所有的 #define 常數值都改為 const 常數值。

¹⁴ MFC Collection classes 中凡涉及指標者 (亦即 class 名稱中帶有 Ptr，如 CPtrArray, CPtrList, CMapWordToPtr, CMapPtrToWord, CMapPtrToPtr, CmapStringToPtr)，皆不支援 Persistence，也就是皆不支持 Serialization。它們都只使用第一層巨集：DECLARE_DYNAMIC, IMPLEMENT_DYNAMIC。

接下來的兩小節，我將分別討論 CPtrList 精巧的 memory pool 機制和 garbage collection（垃圾回收）機制，以及 persistence 文件格式的改善和優化。其中所討論的作法都已實現於 MFCLite3。

6.12.1 CPtrList 的垃圾回收機制（Pool 技巧實現）

6.2.5 節已經展示過 CPtrList 的介面。由於其實作手法十分精巧，涵蓋了 Memory Pool 和 Garbage Collection（GC），頗值得一說，所以我利用這一節加以解釋。首先請看圖 6-39。

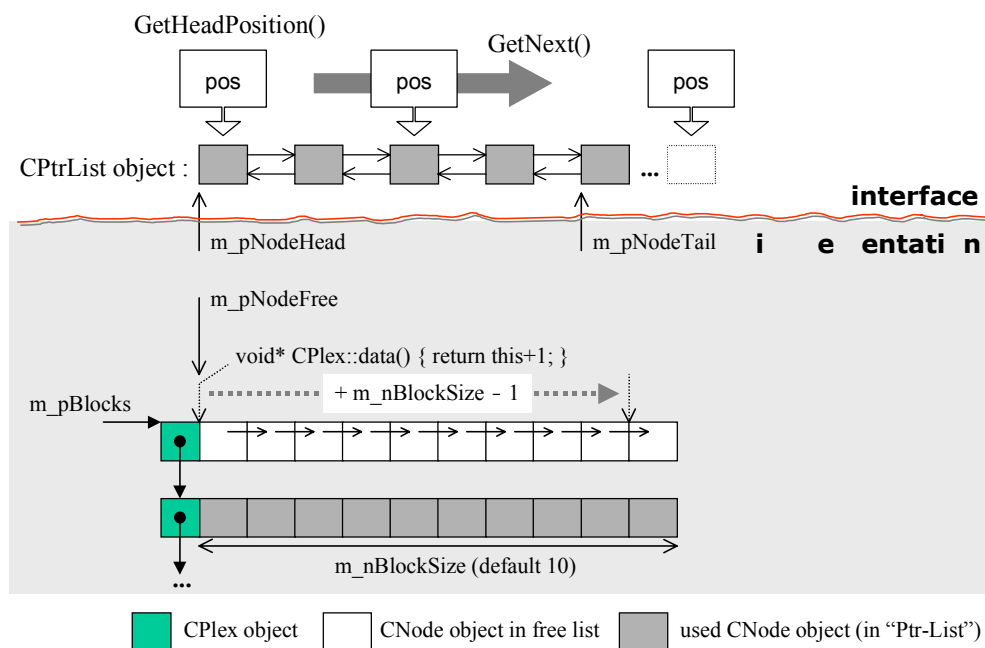


圖 6-39 CPtrList 的介面與實作示意。細節見內文說明。注意：圖上半部的雙向串列，每一個節點其實來自圖下半部的 memory pool（由 CPtrList objects 構成），只不過是以指標串接起來而已。圖上半部是從使用者角度看出去的邏輯顯示。只有在 CPtrList 從未釋放任何節點的情況下，圖下半部才會那麼「漂亮」，否則白色節點和灰色節點應該參差交錯。

CPtrList 是個雙向串列（doubly linked list），每個節點都是 CNode object。雙向串列的維護，在任何一本資料結構書籍中一定會談到，我不贅述。這裡特別之處

是 CPtrList 並非一個節點一個節點地成長，而是採用 memory pool 策略：一次配置多個節點空間，再逐次從中一個個取出節點來用。Memory pool 本身也有彈性，一次配置一個 Cplex object，內含的節點個數可於初始化時設定，預設為 10 個。

```
struct Cplex
{
    Cplex* pNext;
    void* data() { return this+1; } // 參見圖 6-39
    static Cplex* Create(Cplex*& head, UINT nMax, UINT cbElement);
    void FreeDataChain(); // free this one and links
};
```

```
Cplex* Cplex::Create(Cplex*& pHead, UINT nMax, UINT cbElement)
{
    ASSERT(nMax > 0 && cbElement > 0);
    Cplex* p = (Cplex*) new BYTE[sizeof(Cplex) + nMax * cbElement];
    // 參見圖 6-39。may throw exception
    p->pNext = pHead;
    pHead = p; // change head (adds in reverse order for simplicity)
    return p;
}
```

```
void Cplex::FreeDataChain() // free this one and links
{
    Cplex* p = this;
    while (p != NULL)
    {
        BYTE* bytes = (BYTE*) p;
        Cplex* pNext = p->pNext;
        delete[] bytes;
        p = pNext;
    }
}
```

CPtrList 的另一個特點在於其 garbage collection：以 m_pNodeFree 為頭端，指向一個單向串列（singly linked list，以下稱為 *free list*）——其中每個節點都是被釋放的 CNode object。當 CPtrList 要釋放節點時，並不真正呼叫 delete，而是將被釋放的節點納入 *free list*。每當 CPtrList 需要新節點，首先就到 *free list* 取廢棄節點來用；如果 *free list* 之中已經沒有任何廢棄節點了，才再配置一個 Cplex object 加入 memory pool，並加入 *free list*，然後從中取一個節點來用。

memory pool（由 Cplex objects 組成）會維持生命直到 CPtrList 的元素個數為 0

(亦即 CPtrList 的所有節點都被釋放光了)，才將記憶體整個釋放掉。因此在 CPtrList 的生命過程中，memory pool 的大小將接近 CPtrList 的曾經最大元素個數。

CPtrList 的宣告見 6.2.5 節，下面是其關鍵實作部分。

```
CPtrList::CPtrList(int nBlockSize /* =10 */)
{
    m_nCount = 0;
    m_pNodeHead = m_pNodeTail = m_pNodeFree = NULL;
    m_pBlocks = NULL;
    m_nBlockSize = nBlockSize;
}
```

```
CPtrList::~CPtrList()
{
    RemoveAll();
    ASSERT(m_nCount == 0);
}
```

```
POSITION CPtrList::AddTail(void* newElement)
{
    CNode* pNewNode = NewNode(m_pNodeTail, NULL);
    pNewNode->data = newElement;
    if (m_pNodeTail != NULL)
        m_pNodeTail->pNext = pNewNode;
    else
        m_pNodeHead = pNewNode;
    m_pNodeTail = pNewNode;
    return (POSITION) pNewNode;
}
```

```
POSITION CPtrList::Find(void* searchValue,
                        POSITION startAfter /* NULL */) const
{
    // 預設從頭端 HEAD 開始找 (第二參數預設值為 NULL)。
    // 如果沒找到就傳回 NULL。

    CNode* pNode = (CNode*) startAfter;
    if (pNode == NULL)
        pNode = m_pNodeHead; // start at head
    else
        // 這裡應該先檢查 pNode 是否為一個有效位址。本處略。
        pNode = pNode->pNext; // 從指定位置開始檢查。

    for (; pNode != NULL; pNode = pNode->pNext)
        if (pNode->data == searchValue)
```

```

        return (POSITION) pNode;
    return NULL;
}

```

```

void CPtrList::FreeNode(CPtrList::CNode* pNode)
{
    // 垃圾收集，把不要的元素串起來。直到最後才會一次總清。
    pNode->pNext = m_pNodeFree;
    m_pNodeFree = pNode;
    m_nCount--;
    ASSERT(m_nCount >= 0); // make sure we don't underflow

    // 如果不再有任何元素，就將垃圾完全清除
    if (m_nCount == 0)
        RemoveAll(); // 總清理
}

```

```

CPtrList::CNode*
CPtrList::NewNode(CPtrList::CNode* pPrev, CPtrList::CNode* pNext)
{
    if (m_pNodeFree == NULL)
    {
        // add another block
        CPlex* pNewBlock = CPlex::Create(m_pBlocks, m_nBlockSize,
                                          sizeof(CNode));

        // chain them into free list
        CNode* pNode = (CNode*) pNewBlock->data(); // 參見圖 6-39
        // free in reverse order to make it easier to debug
        pNode += m_nBlockSize - 1; // 參見圖 6-39
        for (int i = m_nBlockSize-1; i >= 0; i--, pNode--)
        {
            pNode->pNext = m_pNodeFree;
            m_pNodeFree = pNode;
        }
    }
    ASSERT(m_pNodeFree != NULL); // we must have something

    CPtrList::CNode* pNode = m_pNodeFree;
    m_pNodeFree = m_pNodeFree->pNext;
    pNode->pPrev = pPrev;
    pNode->pNext = pNext;
    m_nCount++;
    ASSERT(m_nCount > 0); // make sure we don't overflow

    pNode->data = 0; // start with zero
    return pNode;
}

```

```
void CPtrList::RemoveAll()
{
    // CPtrList 採用 garbage collection，最後才一次總清理。
    m_nCount = 0;
    m_pNodeHead = m_pNodeTail = m_pNodeFree = NULL;
    m_pBlocks->FreeDataChain(); // 這裡便是總清的地點。
    m_pBlocks = NULL;

    // 侯捷註：這裡應該一一釋放每個元素（指標）所指向的記憶體嗎？
    // 不必，因為使用者每次呼叫 RemoveAt() 後，如有必要，
    // 自己得做 delete 動作，見 CDocManager::~CDocManager()。
    // 如無必要，當然就不做了，見 CDocument::RemoveView()。
}
```

```
void CPtrList::RemoveAt(POSITION position)
{
    CNode* pOldNode = (CNode*) position;

    // 自 list 身上移除 pOldNode（調整前後指標關係，便可辦到）
    if (pOldNode == m_pNodeHead)
        m_pNodeHead = pOldNode->pNext;
    else
        pOldNode->pPrev->pNext = pOldNode->pNext;

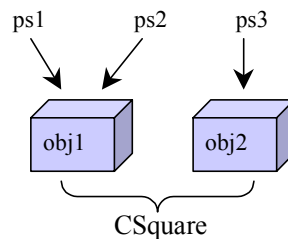
    if (pOldNode == m_pNodeTail)
        m_pNodeTail = pOldNode->pPrev;
    else
        pOldNode->pNext->pPrev = pOldNode->pPrev;

    FreeNode(pOldNode); // 將被移除的元素放進垃圾桶
}
```

6.12.2 Object Persistence 再探討

6.2.5 節展示的文件格式（圖 6-9、圖 6-10）以及 6.5.3 節展示的 CArchive 和 Object Persistence 實作手法，比較直觀，也比較簡化，雖然功能可用，卻有以下兩個缺點。我以實例揭示之：

```
CSquare *ps1 = new CSquare(1,2,3); // CSquare 見 6.2.6 節
CSquare *ps2 = ps1 ;                // ps1 和 ps2 指向同一個物件
CSquare *ps3 = new CSquare(3,2,1); // ps1,ps2 所指物件隸屬相同 class
assert (ps1 == ps2);                // 檢驗應該成功
{
    // persistence (store)
    CFile write("ptralias.out", CFile::modeWrite) ;
    CArchive store(&write, CArchive::store) ;
    store << ps1 << ps2 << ps3;
}
delete ps1;
delete ps3;
ps1 = ps2 = ps3 = NULL; // 刪除指標後清為 NULL，是好習慣。
{
    // persistence (load)
    CFile read("ptralias.out", CFile::modeRead) ;
    CArchive load(&read, CArchive::load) ;
    load >> ps1 >> ps2 >> ps3;
}
assert (ps1 == ps2);                // (A) 按理應該通過檢驗
```



- 如果 pointers 出現 alias 情況（亦即指向同一個 object，如上例之 ps1 和 ps2），將文件寫入再讀出便會造成錯誤¹⁵：原本指向同一個 object 的兩個 pointers，寫入再讀出後卻不再指向同一個 object。換句話說上例的 (A) 本應檢驗通過，但是截至目前我們所發展的文件格式（圖 6-9、圖 6-10）和 CArchive 實作手法（6.5.3 節）卻會使得上例 (A) 式無法過關。
- 造成資源浪費。上例 ps1 和 ps2 所指物件隸屬同一個 class，那麼該 class 的

¹⁵ 感謝肖翔先生在本章試閱活動中對此提出忠告。

相關資訊應該寫入文件檔一次就夠了。但是截至目前我們所發展的文件格式和 CArchive 實作手法，卻會產生重複的 class 資訊。一旦重複次數過多，就形成了巨大的浪費，不僅浪費文件空間，也浪費文件讀寫時間。在今天無處不上網的時代，也就連帶浪費了網絡頻寬並影響網絡效率。

為解決上述兩個問題，MFC 的作法是維護一個 cache（快取）裝置，其中放置已讀/寫的 objects 位址、已讀寫的 class 資訊（亦即 CRuntimeClass 位址）。每當讀寫文件檔，面對每一個 object，都參考並維護這份 cache。當然，還得設計出一套標籤（tags）系統，用來辨識各種可能的情况。

三種標籤（tags）的設計

讓我們謹慎分析所有可能情况，並觀察 MFC（以及稍後 MFCLite 要補強）的設計：

1. 初次遭遇某個 class，標籤為 `wNewClassTag (0xFFFF)`，亦稱為 **newclass tag**。
2. 非初次遭遇某個 class，但係初次遭遇某個 object，標籤設定為 `wClassTag | nClassIndex (0x800?)`。此亦稱為 **class tag**。
3. 非初次遭遇某個 object，標籤為 `nObIndex (000?)`。此亦稱為 **object tag**。

為此進行了以下定義：

```
#define wNullTag          ((WORD)0)
#define wNewClassTag      ((WORD)0xFFFF)
#define wClassTag         ((WORD)0x8000)
#define dwBigClassTag     ((DWORD)0x80000000)
#define wBigObjectTag     ((WORD)0x7FFF)
#define nMaxMapCount      ((DWORD)0x3FFFFFFE)
// 後三個定義用於 cache 成長至極大時，程式碼會用到，但枝微末節我就不解釋了
```

CArchive 介面增修

寫檔時以 `CMapPtrToPtr` 做為 cache，讀檔時則以 `CPtrArray` 做為 cache。這些機關都封裝於 CArchive 之內。下面列出以 6.5.3 節的 CArchive 為基礎，新增加的成員：

```
class CArchive    // 只列出本節新設計較之於 6.5.3 節添補之處。
{
...
public:
    // advanced object mapping (used for forced references)
    void MapObject(const CObject* pObj);
```

```

    // public for advacned use
    UINT m_nObjectSchema;

    // 觀察並除錯用。MFC 無，侯捷添加。
    void ShowLoadArray();
    void ShowStoreMap();

protected:
    // array/map for CObject* and CRuntimeClass* load/store
    UINT m_nMapCount;
    union
    {
        CPtrArray* m_pLoadArray;
        CMapPtrToPtr* m_pStoreMap;
    };
    // map to keep track of mismatched schemas
    CMapPtrToPtr* m_pSchemaMap;
};

```

一個 `archive` 要不用於讀檔，要不用於寫檔，不可能兩種功能兼俱，所以上述使用 `union` 是合理的。此處 `map` 以指標為鍵值 (*key*)，以某種索引為實值 (*value*) — 稍後我會討論為什麼這麼設計 — 其中身為 *key* 的指標有兩種類型 (稍後討論)，因此宣告時必須選擇 `void*` 才能通用。至於 *value* 可以是個 `WORD`。但 MFC 卻捨棄 `CMapPtrToWord` 而選用 `CMapPtrToPtr`。也因此 MFC 相關源碼常有將 `WORD` 轉換為 `void*` 以及將 `void*` 轉換為 `WORD` 的不自然行為，像這樣：

```

m_pStoreMap->SetAt(NULL, (void*)(DWORD)wNullTag); // 設定 value
(*m_pStoreMap)[(void*)pOb] = (void*)m_nMapCount++; // 設定 value
nClassIndex = (DWORD)(*m_pStoreMap)[(void*)pClassRef]); // 讀取 value

```

MFC 的這種選擇 (以及連帶行為) 令人費解。為了這個原因，也為了簡化實作，我決定在 MFCLite 中以 C++ 標準程式庫的 `map` 和 `vector` 取代上述兩個容器：

```

// in MFCLite.h
typedef map<void*,UINT> T_STOREMAP;
typedef vector<void*> T_LOADARRAY;
typedef map<void*,UINT> T_SCHEMAMAP;
// 為什麼要定義這些特別的型別呢？因為這些容器在 MFCLite.cpp 中需要
// 被巡訪 (觀察及除錯用)，而該處只需使用 T_STOREMAP::iterator 便可宣告
// 出 map 迭代器，不必寫為 map<void*,UINT>::iterator。這可使型別的撰寫
// 獨立於一處，有利維護。

// in MFCLite.cpp, CArchive
T_LOADARRAY m_LoadArray;
T_STOREMAP m_StoreMap;

```

```
T_SCHEMAMAP m_SchemaMap;
```

以 C++ 標準容器來取代 MFC collection classes，唯一可能形成功能障礙的便是，前者不支援 **persistence** 而後者支援（名稱中夾帶有 `Ptr` 者除外，例如 `CPtrArray`, `CPtrList`, `CMapWordToPtr`, `CMapPtrToWord`, `CMapPtrToPtr`, `CMapStringToPtr`）。目前的情況是，MFC 採用的兩個容器本來就不支援 **persistence**，而且它們用來做為 **cache**，也不需要 **persistence** 功能。因此以 C++ 標準程式庫的 `map` 和 `vector` 來取代它們是完全沒有問題的。

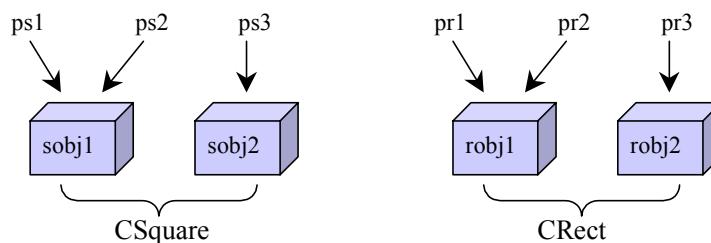
輸出至檔案

下面是將 document 內的每一個 object 輸出至檔案（所謂 *Store*）的邏輯步驟。這些步驟實現於 `CArchive::WriteObject()` 和 `CArchive::WriteClass()` 中：

1. 如果在 *StoreMap* 中找到 object 位址（視為 *key*），那麼就（只）將其 *value* 寫入檔案，此即所謂 **object tag**。*StoreMap* 維持不變。處理下一個 object。
2. 如果沒在 *StoreMap* 中找到 object 位址，但找到了其 class 相應的 `CRuntimeClass` 位址（被視為 *key*），那麼就先將 **class tag** 寫入檔案，再寫入 object 本身。同時並維護 *StoreMap*：以 object 位址為 *key*，**class tag** 中的 `nClassIndex` 為 *value*，寫入 *StoreMap*。處理下一個 object。
3. 如果沒在 *StoreMap* 中找到 object 位址，也沒找到其相應的 class 的 `CRuntimeClass` 位址，表示面對的是個嶄新的 class。先將 **new class tag** 寫入檔案，再寫入 class 資訊（包含 object 版本號碼 — 亦即 *schema no.*，以及 class 名稱 — 採用 `LString` 格式），然後再寫入 object 本身。同時並維護 *StoreMap*：以 `CRuntimeClass` 位址為 *key*，以累積之索引為 *value*，寫入 *StoreMap*。處理下一個 object。

假設我們有這些 objects：

```
CSquare *ps1 = new CSquare(1,2,3);
CSquare *ps2 = ps1 ;    // ps1 和 ps2 指向同一個 object
CSquare *ps3 = new CSquare(3,2,1);
CRect    *pr1 = new CRect(1,2,3,4);
CRect    *pr2 = pr1 ;    // pr1 和 pr2 指向同一個 object
CRect    *pr3 = new CRect(4,3,2,1);
```



經過這樣的寫檔動作：

```
CFile write("ptralias.out", CFile::modeWrite) ;
CArchive store(&write, CArchive::store) ;
store << ps1 << ps2 << ps3 << pr1 << pr2 << pr3;

delete ps1; delete ps3; delete pr1; delete pr3;
ps1 = ps2 = ps3 = pr1 = pr2 = pr3 = NULL;
```

獲得的結果是：

```
Turbo Dump Version 3.1 Copyright (c) 1988, 1992 Borland International
Display of File ptralias.out (88 bytes)

000000: FF FF 00 00 07 00 43 53 71 75 61 72 65 00 00 80 .....CSquare...
000010: 3F 00 00 00 40 00 00 40 40 02 00 01 80 00 00 40 ?...@...@@.....@
000020: 40 00 00 00 40 00 00 80 3F FF FF 00 00 05 00 43 @...@...?.....C
000030: 52 65 63 74 00 00 80 3F 00 00 00 40 00 00 40 40 Rect...?...@...@
000040: 00 00 80 40 05 00 04 80 00 00 80 40 00 00 40 40 ...@.....@...@
000050: 00 00 00 40 00 00 80 3F ...@...?.....
```

StoreMap 的最後狀態如下(前為 *key*, 後為 *value*)，這是 `CArchive::ShowStoreMap()`

的輸出成果：

```
[00000000, 0] // null pointer
[0043F720, 1] // &CSquare::classCSquare
[007A1240, 2] // &sobj1 (見稍早圖示), ps1。
[007A1080, 3] // &sobj2 (見稍早圖示), ps2。
[0043F740, 4] // &CRect::classCRect
[007A1060, 5] // &robj1 (見稍早圖示), pr1。
[007A1040, 6] // &robj1 (見稍早圖示), pr2。
```

圖 6-40a 可以清楚告訴你各種 tags 和 *StoreMap* 之間的關係。

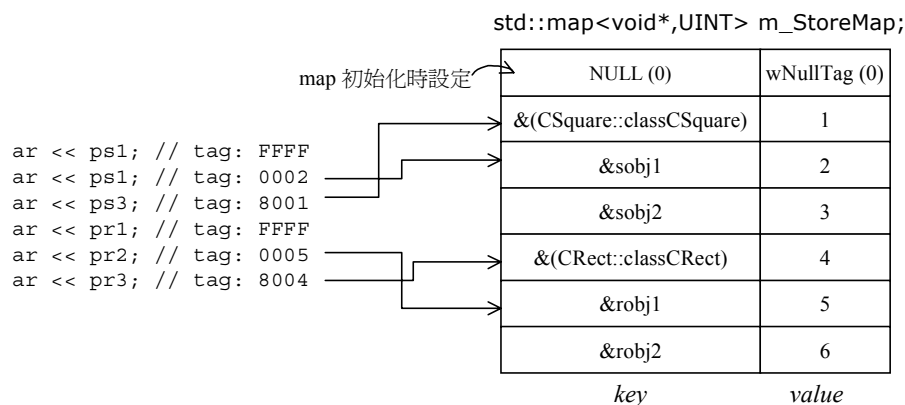


圖 6-40a 經過精心設計，三種不同的 tags（new class tags, class tags, object tags）扮演不同的角色，可用來指向 *StoreMap* 內的元素。

從檔案輸入

下面是將檔案內的 object 讀出置於 document（所謂 *Load*）的邏輯步驟。這些步驟實現於 `CArchive::ReadObject()` 和 `CArchive::ReadClass()` 中。每次讀取資料，首先讀到的是 tag：

1. 如果讀到的是個 **new class tag**，繼續讀取 class 資訊（包含 object 版本號碼，亦即 schema no.，以及採用 `LString` 格式的 class 名稱），然後據此找出對應的 `CRuntimeClass`，再據以產生（new）一個 object，並從檔案中繼續讀取資料做為 object 內容。同時並維護 *LoadArray*：將 `CRuntimeClass` 位址和 object 位址依序寫入 *LoadArray* 的尾端。處理下一個 object。
2. 如果讀到的是個 **object tag**，就以 **object tag** 為索引直接從 *LoadArray* 中獲得 object 位址。處理下一個 object。
3. 如果讀到的是個 **class tag**，就以 **class tag** 內的 `nClassIndex` 為索引，直接從 *LoadArray* 中獲得 `CRuntimeClass` 位址，再據以產生（new）一個 object，並從檔案中繼續讀取資料做為 object 內容。同時並維護 *LoadArray*：將 object 位址寫入 *LoadArray* 的尾端。

現在，延續稍早的例子，經過這樣的讀檔動作：

```
// 注意，在此之前，所有指標已被 deleted，並被設為 NULL。見稍早所示。
CFile read("ptralias.out", CFile::modeRead) ;
CArchive load(&read, CArchive::load) ;
load >> ps1 >> ps2 >> ps3 >> pr1 >> pr2 >> pr3;
```

將獲得正確的結果。*LoadArray* 的最後狀態如下，是 `CArchive::ShowLoadArray()` 的輸出成果（依元素次序）：

```
00000000          // NULL pointer
0043F720          // &CSquare::classCSquare。與 StoreMap 內相同。
007A1040          // &sobj1（見稍早圖示），ps1。
007A1860          // &sobj2（見稍早圖示），ps2。
0043F740          // &CRect::classCRect。與 StoreMap 內相同。
007A1020          // &robj1（見稍早圖示），pr1。
007A1810          // &robj1（見稍早圖示），pr2。
```

圖 6-40b 可以清楚告訴你各種 tags 和 *LoadArray* 之間的關係。

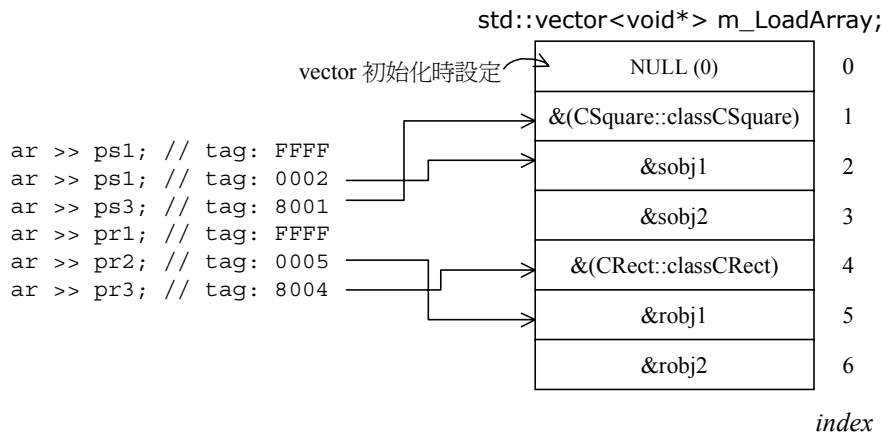


圖 6-40b 經過精心設計，三種不同的 tags（new class tags, class tags, object tags）扮演不同的角色，可用來指向 *LoadArray* 內的元素。

LoadArray 內的 `CRuntimeClass` 位址和 *StoreMap* 內的相同，可以理解：此兩例是同一個程式生命中的寫檔讀檔動作，由於是同一個程式生命，所以 `CRuntimeClass` 的位址並沒有改變。*LoadArray* 內的 object 位址和 *StoreMap* 內的不同，也是可以理解的：object 被寫入檔案後，經過刪除（delete）動作，才又以相同的指標接納重新讀入的 object，所以兩階段中的 object 位址非常有可能不同。

除了 *StoreMap* 和 *LoadArray*，`CArchive` 還內含第三個容器 *SchemaMap*，用來輔助物件版本號碼（schema no.）的比對查證。枝微末節，就不提了。

以上討論雖然實作上需要費一番手腳，但邏輯層面很是清楚。程式碼集中於 CArchive 的 ReadObject(), ReadClass(), WriteObject(), WriteClass() 以及 MapObject() 身上。

將 MFCLite 的 object persistence 改為本節所討論的作法，然後重新執行 6.10 節的完整範例，檔案輸出如下：

```
Turbo Dump Version 3.1 Copyright (c) 1988, 1992 Borland International
      Display of File TEMP.HOU (39 bytes)

000000: FF FF 01 00 0B 00 43 44  57 6F 72 64 41 72 72 61  ....CDWordArra
000010: 79 05 00 09 00 00 00 07  00 00 00 05 00 00 00 03  y.....
000020: 00 00 00 01 00 00 00      .....
```

```
Turbo Dump Version 3.1 Copyright (c) 1988, 1992 Borland International
      Display of File TEMP.DAT (235 bytes)

000000: 08 00 FF FF 00 00 08 00  43 45 6C 6C 69 70 73 65  ....CEllipse
000010: 00 00 40 40 00 00 40 40  00 00 E0 40 00 00 A8 41  ..@...@...A
000020: FF FF 00 00 07 00 43 43  69 72 63 6C 65 00 00 A0  ....CCircle...
000030: 40 00 00 A0 40 00 00 E0  40 FF FF 00 00 09 00 43  @...@...@.....C
000040: 54 72 69 61 6E 67 6C 65  00 00 00 00 00 00 00 00  Triangle.....
000050: 00 00 80 3F 00 00 00 00  00 00 00 00 00 00 80 3F  ...?.....?
000060: FF FF 00 00 05 00 43 52  65 63 74 33 33 B3 40 9A  ....CRect33.@.
000070: 99 D9 40 00 00 40 40 00  00 10 41 FF FF 00 00 07  ..@...@...A....
000080: 00 43 53 71 75 61 72 65  CD CC 4C 40 9A 99 89 40  .CSquare..L@...@
000090: 00 00 C0 40 FF FF 00 00  07 00 43 53 74 72 6F 6B  ...@.....CStrok
0000A0: 65 07 00 01 00 00 00 02  00 00 00 03 00 00 00 04  e.....
0000B0: 00 00 00 05 00 00 00 06  00 00 00 07 00 00 00 05  .....
0000C0: 80 00 00 00 00 00 00 00  00 00 00 60 40 00 00 00  .....`@...
0000D0: 00 00 00 00 00 9A 99 A9  40 07 80 66 66 1E 41 66  .....@..ff.Af
0000E0: 66 1E 41 9A 99 99 40 CD  CC 1C 41      f.A...@...A.....
```

6.12.3 文件清理和視窗關閉

原先我對 MFCLite 的構想中並沒有納入「視窗關閉」模擬計劃。各位可從 6.10 節的測試程式執行結果中看出來，使用者並未在程式結束前一一關閉視窗，程式也沒有要求這麼做。但是資源（尤其是記憶體）總是必須回收，否則會造成洩漏（leak）問題。為此，我在未模擬「視窗關閉」的情況下做了一些處理，卻犯了一些錯誤，養出一隻大臭蟲。現在我終於要來面對隻臭蟲¹⁶。這個錯誤經驗頗值得記錄下來供大家引為鏡鑑。

正確而完善的作法是，每個視窗都提供 [File|Close] 表單命令，並在處理其相應訊息（WM_CLOSE）時判斷視窗內的 document 內容是否曾被修改。如果曾被修改，就提示使用者做存檔動作，像這樣（取自 Word 執行畫面）：



然後將視窗摧毀，刪除其內的 document 和連帶的每一個 views，並維護 DocTemplate（如圖 6-33）。如果使用者直接將程式結束（[File|Exit]），程式應該通知每一個視窗，進行上述動作。

錯誤的設計 — 挖東補西 鋸箭療傷

由於 MFCLite 的焦點放在 application framework 主軸身上，並未模擬真正的圖像視窗，所以當初也就沒有打算實現上述 [File|Close] 表單命令。我並未在我的 MFCLite 範例程式 mfclapp.cpp 中撰寫任何「得以編修 document 內容」的功能，只以 WM_LBUTTONDOWN 觸發初值設定（見 6.10 節程式執行過程中的熱鍵 'u'）。就算你決定在你的 MFCLite 應用程式中編修 document 內容，我也認為將所有文件內容先儲存起來再結束程式，是測試者的責任；MFCLite 意在模擬最重要的框架主軸

¹⁶ 感謝肖翔先生在本章試閱活動中給了我一些刺激。

技術，枝微末節引不起我的興趣 ☺。

但是即便如此，清理工作總一定得做，因為每一份 document（及其相應的一切資源）都是動態配置而來，如果缺乏適當的清理，會造成嚴重的資源洩漏。為此，我將清理工作放在 `CMultiDocTemplate::~CMultiDocTemplate()` 中，因為它是程式結束前肯定會被喚起的一個函式，就責任歸屬而言頗為合理：

```
; CallStack (未實現[File|Close]的情況下):
CMultiDocTemplate::~CMultiDocTemplate()
CDocManager::~CDocManager()
CWinApp::~CWinApp()
```

```
CWinApp::~CWinApp()
{
    // 釋放 doc manager
    if (m_pDocManager != NULL)
        delete m_pDocManager;
}
```

```
CDocManager::~CDocManager()
{
    // cleanup - 刪除所有的 document templates
    POSITION pos = m_templateList.GetHeadPosition();
    while (pos != NULL)
    {
        POSITION posTemplate = pos;
        CDocTemplate* pTemplate =
            (CDocTemplate*)m_templateList.GetNext(pos);
        m_templateList.RemoveAt(posTemplate); // 移除元素（指標）
        delete (CDocTemplate*)pTemplate;      // 刪除元素（指標）所指物件
    }
}
```

```
CMultiDocTemplate::~CMultiDocTemplate()
{
    // 在 MFC 源碼中，這裡除了在 DEBUG 模式下檢查 m_docList.IsEmpty()，
    // 以及在 non-DEBUG 模式下檢查某些 GUI 資源外，再無其他動作。

    // jjhou 第一個疑問：難道不該在此將 MultiDocTemplate 管理的所有資源釋放？
}
```

這就很奇怪了。我的第一個疑問是：MFC 難道不該將 `MultiDocTemplate` 管理的所有 document/frame/view 統統刪除（釋放）嗎？有了這樣的想法，我做出以下設計取而代之：

```

CMultiDocTemplate::~CMultiDocTemplate()
{
    // 以下為 jjhou 添加，用以清除 MultiDocTemplate 管轄的所有 documents
    POSITION pos = m_docList.GetHeadPosition();
    while (pos != NULL)
    {
        POSITION posDoc = pos;
        CDocument* pDoc = (CDocument*)m_docList.GetNext(pos);
        m_docList.RemoveAt(posDoc);    // 移除元素（指標）
        delete (CDocument*)pDoc;    // (1) 刪除元素（指標）所指物件。Crash!
    }
}

```

上述最後一個動作會引發 CDocument 解構式：

```

CDocument::~CDocument()
{
    // 不應該留下任何 views。
    DisconnectViews();
    ASSERT(m_viewList.IsEmpty());

    // jjhou 第二個疑問：難道此處不需將 document 所關連的 frame 視窗關閉，
    // 並將 frame 物件摧毀嗎？
    delete m_pFrame;    // (2) jjhou 添加

    if (m_pDocTemplate != NULL)
        m_pDocTemplate->RemoveDocument(this);
    // 撤除 document (this 所指) 和 doc template 之間的關係
    ASSERT(m_pDocTemplate == NULL);    // 必須被解除關係 (detached)
}

```

```

void CMultiDocTemplate::RemoveDocument(CDocument* pDoc)
{
    CDocTemplate::RemoveDocument(pDoc);
    m_docList.RemoveAt(m_docList.Find(pDoc));
}

```

```

void CDocTemplate::RemoveDocument(CDocument* pDoc)
{
    // 撤除 document (pDoc 所指) 和 doc template 之間的關係
    ASSERT(pDoc->m_pDocTemplate == this);    // must be attached to us
    pDoc->m_pDocTemplate = NULL;
}

```

上述灰色區域的第二個疑問，答案應該是肯定的，但為什麼沒有那樣的動作？難道 MFC 處處存在洩漏問題嗎？為此我又加了一個 delete 動作，刪除 document 所對應（所棲身）之 docFrame 視窗。其中的 m_pFrame 並非 MFC 原始設計，而是

我為求方便在 CDocument 身上添加的成員。沒有它，要迂迴找出 document 對應的 docFrame 視窗倒也不是辦不到，MFC 在必要場合中就是這麼做的：根據圖 6-33 所示，從 document 中取其所轄的 views，再取後者的父視窗，即為 docFrame 視窗。但 MFCLite 未曾模擬實像視窗，也就未曾管理各個視窗¹⁷，也就不能以此法獲得父視窗。

上述 (1)，(2) 兩處補強看似合理，執行時 (1) 卻造成程式崩潰，令我百思不解。無計可施之下只好相當阿 Q 地將 (1) 標註起來，反正執行至此，記憶體洩漏也於事無損¹⁸。

但這終究是個疙瘩。什麼原因造成崩潰？由於某些機緣，我終於發現崩潰的原因。上述 delete 動作引發 CDocument 解構式，CallStack 如下：

```
; CallStack (未實現 [File|Close] 的情況下):
CMultiDocTemplate::RemoveDocument()          // (B)
CDocument::~~CDocument()
CMultiDocTemplate::~~CMultiDocTemplate()      // (A)
CDocManager::~~CDocManager()
CWinApp::~~CWinApp()
```

(A) 已針對 pDoc 呼叫 m_docList.RemoveAt()，而 (B) 又做以下動作：

```
m_docList.RemoveAt(m_docList.Find(pDoc));
```

此時 pDoc 已在 (A) 中被移除，於是 Find() 傳回 NULL，於是 RemoveAt(NULL) 造成崩潰。為了讓程式正常運行，我決定在 RemoveAt() 函式起始處增加一行判斷：

```
void CPtrList::RemoveAt(POSITION position)
{
    if (position == NULL) return;    // (3) jjhou 添加
    ...
}
```

終於讓整個程式順利執行。

¹⁷ 真正的視窗系統（例如 MS Windows）中，每個視窗的從屬、父子關係，都會被記錄並維護。

¹⁸ 以 MFCLite 的設計而言，到了這一步，已是程式即將結束的前一刻。一個程式即使有記憶體洩漏問題，結束之後一了百了，不會影響其他程式，因為現代化作業系統十分聰明，能夠在某個行程（process）結束時回收其所有記憶體。

這樣的分析真像太陽底下的影子那麼明顯，但是當局者迷，花了我不少時間。此外，就像一個謊話需要三個謊話來圓一樣，一個補釘引發了三個補釘，誰知道還有什麼漏洞是我沒有看到或測到的呢！(2)，(3)皆由(1)而來，為什麼 MFC 本尊既不需要(2)也不需要(3)？這是否意味(1)是不良的設計？

正是如此。一個不良的設計往往引發無數的麻煩。這個問題的良好設計應該是讓每一個 docFrame 視窗實現 [File|Close] 命令，讓 mainFrame 視窗也實現 [File|Close] 命令，並且讓每一個 frame 視窗處理 WM_CLOSE 訊息。現在，我決定把前述的(1)，(2)，(3)都拿掉（MFC 中本來就沒有它們），重新回到正軌。

MFC 的視窗關閉應對措施（CallStack 的觀察）

過去以來 MFCLite 的模擬全都借重於我個人對 MFC 的了解，以丁字鎬鶴嘴鋤畚箕扁擔的方式進行：一堆源碼，一個文字編輯器，一個文字搜尋器（grep utility），再加上一顆耐煩的腦袋瓜。這次我要借助重機具，示範如何快速掌握程式流程與源碼。我已經知道問題出在「不該由 CMultiDocTemplate::~~CMultiDocTemplate() 喚起 CDocument::~~CDocument()」，因此決定先找個 MFC 應用程式，以除錯模式編譯，並在 CDocument::~~CDocument() 身上設立中斷點（breakpoint），然後在程式執行過程中觀察各種視窗關閉情況下的函式呼叫次序（CallStack）。

- 使用者點選任何一個 docFrame 視窗的 [system|Close]，意思是要關閉該視窗。皮之不存毛將附焉，所以必須連帶關閉所轄文件，文件所轄的 views 亦應一併關閉；其標準提示文句為（見視窗最下）：Close the active window and prompts to save the documents。這個動作等同於雙擊（double click）視窗左上角的圖示。稍後我將在 MFCLite 中以熱鍵 'c' 模擬之。

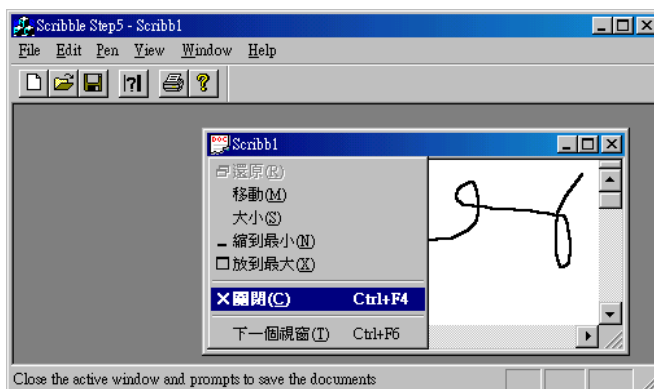


圖 6-41a

下面是執行至 `CDocument::~~CDocument()` 為止的呼叫堆疊 (CallStack)：

```
; CallStack (for MFC App. Scribble)
CDocument::~~CDocument()
CScribbleDoc::~~CScribbleDoc()
CDocument::OnCloseDocument()
CFrameWnd::OnClose()
CWnd::OnWndMsg(WM_CLOSE,...)
CWnd::WindowProc(WM_CLOSE,...)
AfxCallWndProc(...)
AfxWndProc(...)
AfxWndProcBase(...)
KERNEL32! bff7363b()
KERNEL32! bff945af()
```

- 使用者點選 `mainFrame` 視窗的 [File|Close]，意思是要清理 active document (現行文件) 並關閉其 `docFrame` 視窗。Document 所轄的 views 亦應一併關閉；其標準提示文句為 (見視窗最下)：Close the active document。稍後我將在 MFCLite 中以熱鍵 'C' 模擬之。



圖 6-41b

下面是執行至 `CDocument::~~CDocument()` 為止的呼叫堆疊 (CallStack)：

```
; CallStack (for MFC App. Scribble)
CDocument::~~CDocument()
CScribbleDoc::~~CScribbleDoc()
CDocument::OnCloseDocument()
CDocument::OnFileClose()
_AfxDispatchCmdMsg(CCmdTarget * 0x00781380 {CScribbleDoc},
    ID_FILE_CLOSE, int 0,
    void (void) * 0x5f441f01 CDocument::OnFileClose(void),
    void * 0x00000000, unsigned int 12,
    AFX_CMDHANDLERINFO * 0x00000000)
CCmdTarget::OnCmdMsg(ID_FILE_CLOSE,...)
```

```

CDocument::OnCmdMsg(ID_FILE_CLOSE,...)
CView::OnCmdMsg(ID_FILE_CLOSE,...)
CFrameWnd::OnCmdMsg(ID_FILE_CLOSE,...)
CWnd::OnCommand(ID_FILE_CLOSE,...)
CFrameWnd::OnCommand(ID_FILE_CLOSE,...)
CWnd::OnWndMsg(WM_CLOSE,ID_FILE_CLOSE,...)
CWnd::WindowProc(WM_CLOSE,ID_FILE_CLOSE,...)
AfxCallWndProc(...)
CMDIFrameWnd::OnCommand(ID_FILE_CLOSE,...)
CWnd::OnWndMsg(WM_CLOSE,ID_FILE_CLOSE,...)
CWnd::WindowProc(WM_CLOSE,ID_FILE_CLOSE,...)
AfxCallWndProc(...)
AfxWndProc(...)
AfxWndProcBase(...)
KERNEL32! bff7363b()
KERNEL32! bff945af()

```

- 使用者點選 mainFrame 視窗的[system|Close]，意思是要關閉程式 — 連帶所有 documents（及其所轄的 docFrame 視窗和 views）都必須先關閉；其標準提示文句為（見視窗最下）：Close the active window and prompts to save the documents。稍後我將在 MFCLite 中以熱鍵 'x' 模擬之。



圖 6-41c

下面是執行至 CDocument::~~CDocument() 為止的呼叫堆疊（CallStack）：

```

; CallStack (for MFC App. Scribble)
CDocument::~~CDocument()
CScribbleDoc::~~CScribbleDoc()
CDocument::OnCloseDocument()
CDocTemplate::CloseAllDocuments(int 0)
CDocManager::CloseAllDocuments(int 0)
CWinApp::CloseAllDocuments(int 0)
CFrameWnd::OnClose()
CWnd::OnWndMsg(WM_CLOSE,...)
CWnd::WindowProc(WM_CLOSE,...)

```

```

AfxCallWndProc(...)
AfxWndProc(...)
AfxWndProcBase(...)
KERNEL32! bff7363b()
KERNEL32! bff945af()

```

- 使用者點選 mainFrame 視窗的[File|Exit]，意思是要結束程式 — 連帶所有 documents（及其所轄的 docFrame 視窗和 views）都必須先關閉；其標準提示文句為（見視窗最下）：Quit the application; prompts to save documents。稍後我將在 MFCLite 中以熱鍵 'X' 模擬之。



圖 6-41d

下面是執行至 CDocument::~CDocument() 為止的呼叫堆疊（CallStack）：

```

; CallStack (for MFC App. Scribble)
CDocument::~~CDocument()
CScribbleDoc::~~CScribbleDoc()
CDocument::OnCloseDocument()
CDocTemplate::CloseAllDocuments(int 0)
CDocManager::CloseAllDocuments(int 0)
CWinApp::CloseAllDocuments(int 0)
CFrameWnd::OnClose()
CWnd::OnWndMsg(WM_CLOSE, ...)
CWnd::WindowProc(WM_CLOSE, unsigned int 0, long 0)
AfxCallWndProc(...)
AfxWndProc(...)
AfxWndProcBase(...)
KERNEL32! bff7363b()
KERNEL32! bff945af()

```

上述這些 CallStack 資訊提供了很好的線索，讓我們得知 MFC 中哪些 class member functions 和本節主題有關。此外我不必再使用可憐的文字搜尋器在茫茫碼海中尋

找字串（以求從搜尋結果得知某個函式位於哪個源碼檔案內，然後從中學習、修剪，模擬於 MFCLite 之中），只需在 VC 除錯器的 CallStack 視窗上點選列出的任何函式，VC 除錯器就會立刻打開對應的源碼檔案，並在編輯視窗中顯示出該函式的源碼。真是太方便了。

細說從頭一視窗關閉流程（APIs 層面和訊息層面）

正式動工之前，讓我先說明 Microsoft Windows 的視窗關閉流程，包括相關訊息、文件關閉與視窗關閉的關連性、程式結束與文件關閉的關連性。6.6.3 節已經說明了 Windows 系統的訊息流動機制。圖 6-42 係從 Windows APIs 層次來看視窗關閉的訊息和相應 Win32 APIs 動作。

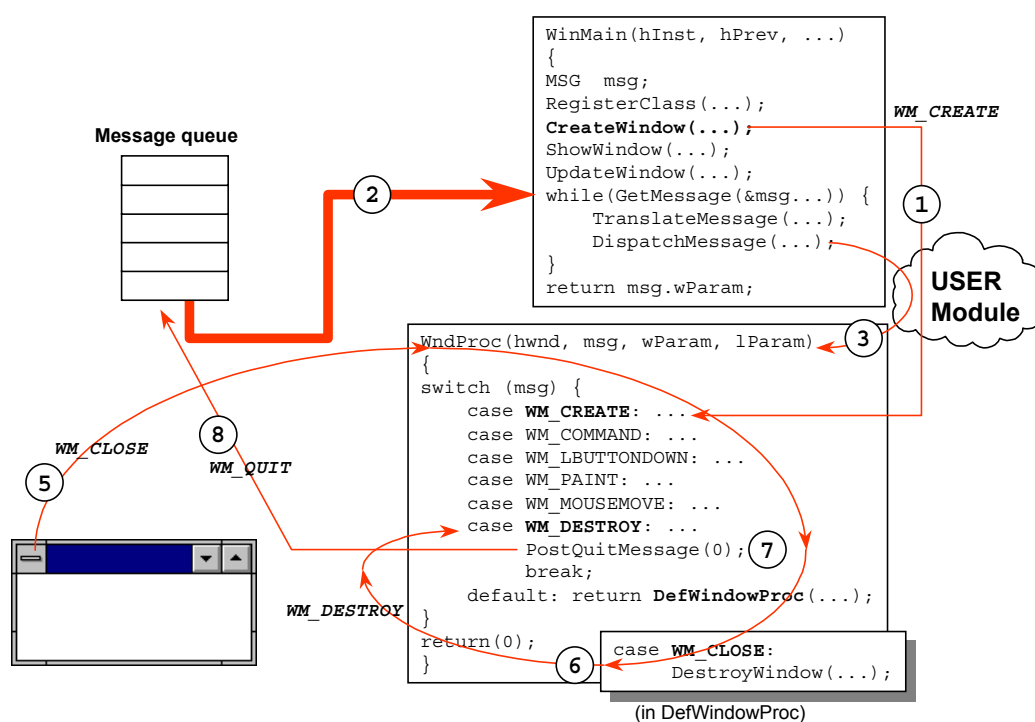


圖 6-42 Windows APIs 層面上的視窗關閉相關訊息和動作。

本圖參考《深入淺出 MFC》2e, 圖 1-5。

圖 6-42 的✎與視窗誕生有關，無關本節主題。✎,✎是訊息派送（dispatching）。視窗關閉的程序是●~() ¹⁹：

● 當[system|Close] 命令項被點選或視窗圖示被雙擊（double click），系統送出 WM_CLOSE。如果視窗函式不攔截此一訊息，就由系統提供的::DefWindowProc() 處理。

✎ ::DefWindowProc 收到 WM_CLOSE 後會呼叫::DestroyWindow() 將視窗清除掉。::DestroyWindow() 本身會送出 WM_DESTROY 訊息。

☞ 程式對 WM_DESTROY 的標準回應是呼叫::PostQuitMessage()。

() ::PostQuitMessage() 的唯一動作就是送出 WM_QUIT 訊息，準備讓訊息迴圈中的::GetMessage() 取得，如行為✎，結束訊息迴圈 — 因為::GetMessage()取得 WM_QUIT 訊息後會傳回 0，導致 while 迴圈結束。

由此我們知道，要讓 MFCLite 有能力關閉視窗，必須為它加上 WM_CLOSE 訊息的處理能力；要讓 MFCLite 有能力關閉文件（而非視窗），必須為它加上[File|Close] 命令項的處理能力。同時我們還必須判斷，當視窗被關閉時，被關閉的視窗究竟是 mainFrame 或 docFrame（前者表示要結束整個程式），以此判斷是否應該採取更多的回應。此外，在上述一般應用程式關注的訊息和採取的應對措施之外，MFC 還運用了一個罕見的訊息 WM_NCDESTROY ²⁰，稍後我有進一步說明。

¹⁹ 此處是 SDI 程式的表現。MDI 程式略有不同，但主要動作和意義都一樣，此處表述仍然具有代表性。

²⁰ WM_NCDESTROY 的 NC 是指 Non-Client，也就是視窗的 client-area 以外的區域，如標題、狀態欄、視窗圖示、表單（如果有的話）等等。client-area 是應用程式的工作區，由應用程式負責，non-client area 則由系統全權負責。

MFCLite 的簡易視窗管理

雖然 6.6.4 節對於 MFCLite 的視窗管理曾有簡單的敘述，但具體手法並未明示。這裡要提出比較更具體的說明。

真正的視窗作業系統裡頭，視窗管理是一門大學問。視窗的從屬關係包括 parent/child/sibling，如圖 6-43a²¹，其間又有 ownership 的概念（圖中未畫出）。

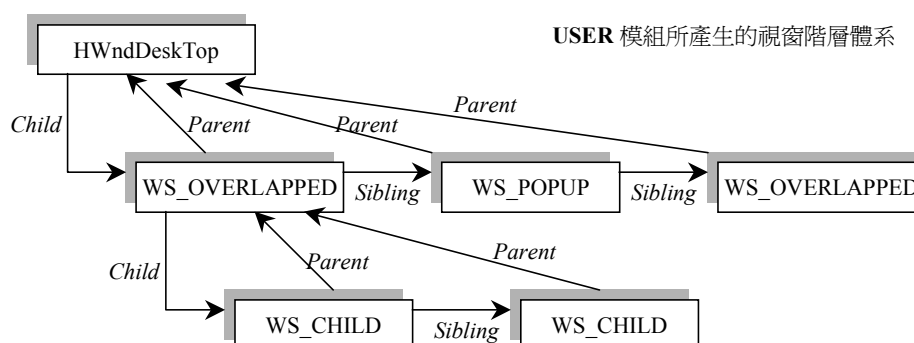


圖 6-43a Microsoft Windows USER 模組所產生的視窗階層體系

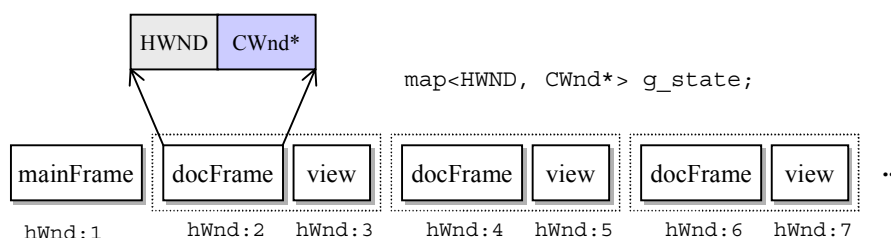


圖 6-43b MFCLite 的視窗管理

MFCLite 的關心焦點不在視窗身上，所以只以最簡化作法來管理虛擬視窗（徒具概念而無實體的視窗）。在 MFCLite 的視窗哲學中，第一個視窗一定是 mainFrame，然後是一組一組的 docFrame/view，每個 docFrame 視窗只能擁有一個 view（雖然 docFrame 是個 CMDIFrameWnd object，理應可有許多 child 視窗，但 MFCLite 簡化

²¹ 參考《Windows Internals》by Matt Pietrek, 1993, Addison Wesley.

了這情況，不讓你這麼做）。視窗的 `hwnd` 均以流水號碼遞增，被摧毀後不再重複使用。因此 `hwnd` 的規則是：`mainFrame` 永遠為 1，`docFrame` 為偶數，`view` 為奇數；所有視窗以一個 `g_state map` 統籌管理，個數永遠為奇數。如圖 6-43b。

MFCLite 的視窗無實際形體，因此，使用者唯一能操弄視窗者，唯熱鍵一途。當熱鍵 'I'（切換 active 視窗）被按下，如果 `map` 之中只有一個視窗，表示此時無任何 `docFrame/view` 視窗，切換無效；否則便從圖 6-43b 所示的 `g_state map` 中尋找下一組 `docFrame/view` 做為 active 視窗；如果已到「盡頭」²²，就調頭從 `hwnd2` 重新找起（`hwnd1` 為 `mainFrame`，不該加入切換行列）。

視窗被刪除時（稍後數小節所介紹的熱鍵 'C', 'c', 'x', 'X' 都有此功能），MFCLite 視窗管理系統只需把握以下原則，依序檢驗執行：

- 如果 `map` 之中只有一個視窗（必為 `mainFrame`），刪除後應導致程式結束。
- 如果 `map` 之中有三個視窗（必為 `mainFrame`、`docFrame` 和 `view`），刪除後應只剩 `mainFrame` 視窗，此時應將 active `docFrame/view` 設為 0/0 以為標示。
- 如果欲刪除之視窗是 active 視窗，那麼就先切換 active 視窗，再進行刪除（從 `g_state map` 中移除）。
- 如果欲刪除之視窗不是 active 視窗，那麼不必切換，直接刪除（從 `g_state map` 中移除）。

為什麼沒有討論「`g_state map` 內有多個視窗，而即將被刪除者為 `mainFrame`」的情形呢（此為稍後小節所討論的熱鍵 'x' 或 'X'）？因為一旦發生這種情況，MFCLite 會先摧毀 `g_state map` 內的每一個 `docFrame/view`，最後只剩下一個 `mainFrame`，才摧毀 `mainFrame`。

MFCLite 的視窗管理工作，全部都由全域物件 `g_state` 和以下全域函式所提供的服務完成：`JJGetMainFrame()`、`JJGetActiveFrame()`、`JJGetActiveView()`、`JJSetActiveFrame()`、`JJSetActiveView()`、`JJSetNextActive()`、`JJDelElemAndSetNextActive()`。

²² 這裡所謂「下一組」和「盡頭」，都是「把 `map` 想像為一個根據 `key` 而排列的線性空間」下的概念。

擴充 MFCLite，增加物件關閉功能、視窗關閉功能

欲在 MFCLite 中處理[File|Close] 命令訊息，誰是最適當的候選人？根據訊息映射表我們知道，任何一個 CCmdTarget-derived classes 都有機會攔截命令訊息。不過既然這個命令用來處理 document，當然由 CDocument 來攔截最是適合。至於視窗訊息 WM_CLOSE，只會落到視窗相關類別（CWnd-derived classes）去，MFCLite 提供的諸如 CWnd, CFrameWnd, CMDIFrameWnd, CMDIChildWnd, CView...等類別都是候選人，但只要底層的 CWnd, CFrameWnd 做掉任務，上層類別就不需操心了。

下面便是為 MFCLite 加上「視窗關閉暨文件清理」功能的步驟。這份清單同時也可做為日後擴充 MFCLite 時的一份參考。

1. 增加熱鍵 'C', 'c', 'X', 'x'，代表的意義分別如圖 6-41 所示。首先修改 resource.rc，為表單增加命令項：

```
POPUP "&File"
    MENUITEM "&Close",    ID_FILE_CLOSE
    MENUITEM "e&Xit",     ID_APP_EXIT
    ...
BEGIN
```

2. 在 .h 檔中定義命令和訊息識別碼。如果和框架模組有關，定義於 afxres.h，如果和應用模組有關，定義於 resource.h。本例定義於 afxres.h：

```
#define ID_FILE_CLOSE    0xE102    // decimal:57602
#define ID_APP_EXIT      0xE141
#define WM_DESTROY       0x0002
#define WM_CLOSE         0x0010    // decimal:16
#define WM_QUIT          0x0012    // decimal:18
#define WM_NCDESTROY     0x0082    // decimal:130
```

3. 在熱鍵求助函式 CMyWinApp::OnAppHotKeyHelp() 中增加四行文字輸出：

```
"C - File|Close, Close the active document"
"X - File|eXit, Quit the application; prompts to save documents"
"c - WM_CLOSE, Close the active window and prompts to save documents"
"x - WM_CLOSE, Quit the application; prompts to save documents"
```

4. 增加以下命令映射項以處理 [File|Close]、[File|Exit] 命令：

```
BEGIN_MESSAGE_MAP(CDocument, CCmdTarget)
    ON_COMMAND(ID_FILE_CLOSE, OnFileClose)
END_MESSAGE_MAP()

BEGIN_MESSAGE_MAP(CWinApp, CCmdTarget)
```



```
ON_COMMAND(ID_APP_EXIT, OnAppExit)
END_MESSAGE_MAP()
```

並宣告 `CDocument::OnFileClose()` 和 `CWinApp::OnAppExit()` 函式。唔，程式結束命令由 `CWinApp` 處理，文件關閉命令由 `CDocument` 處理，很合理。

```
void CDocument::OnFileClose();
void CWinApp::OnAppExit();
```

5. 在 `afxmsg.h` 中增加 `WM_ON_CLOSE()` 巨集如下：

```
#define ON_WM_CLOSE() \
    { WM_CLOSE, 0, 0, 0, AfxSig_vv, \
      (AFX_PMSG) (AFX_PMSGW) (void (CWnd::*) (void)) &OnClose },
```

6. 增加以下訊息映射項以處理 `WM_CLOSE` 訊息，此訊息必由 `CWnd-driven` 處理。

```
BEGIN_MESSAGE_MAP(CWnd, CCmdTarget)
    ON_WM_CLOSE()
END_MESSAGE_MAP()

BEGIN_MESSAGE_MAP(CFrameWnd, CWnd)
    ON_WM_CLOSE()
END_MESSAGE_MAP()
```

並宣告 `CWnd::OnClose()` 和 `CFrameWnd::OnClose()` 函式：

```
void CWnd::OnClose() { DEFAULT(); }; // 啥也沒做。
void CFrameWnd::OnClose(); // 這裡才是主角
```

更深層類別如 `CMDIFrameWnd`，`CMDIChildWnd` 無需處理 `WM_CLOSE`，因為它們並不在視窗關閉前有什麼特徵行為（倒是在視窗被關閉後——也就是 `WM_DESTROY` 發生時有特徵行為，稍後詳述）。`CView` 也無需處理 `WM_CLOSE`，因為視窗關閉前它也無需任何特徵行為。

7. 在 `mfclite.cpp` 的全域函式 `GetMessage()` 中增加判斷以處理「非命令」熱鍵：

```
switch (c) {
    case 'c' : // WM_CLOSE. 模擬 docFrame 視窗的 [system|Close] 命令
        pMsg->hWnd = JJGetActiveFrame();
        pMsg->nMsg = WM_CLOSE;
        break;

    case 'x' : // App-Exit. 模擬 mainFrame 視窗的 [system|Close] 命令
        pMsg->hWnd = JJGetMainFrame();
        pMsg->nMsg = WM_CLOSE;
        break;
```

不同的熱鍵所引發的 `WM_CLOSE` 訊息來自不同的 frame 視窗。

8. 思考前述出現的三個函式的邏輯：

```
void CDocument::OnFileClose(); // 文件關閉
```

```
void CWinApp::OnAppExit(); // 程式結束 (相當於 mainFrame 視窗關閉)  
void CFrameWnd::OnClose(); // 視窗關閉
```

其中相對簡單的是文件的關閉(因為不必考慮 docFrame 視窗關閉相關事務)，就讓我們從它開始。

關閉文件：CDocument::OnFileClose() 熱鍵 'C'

欲關閉一份文件，下面是幾個思考重點²³（整個流程概觀請見圖 6-44）：

- 如果文件內容已遭修改，應詢問是否要存檔 — 如果文件有名稱，便喚起 [Save] 對話盒，如無名稱就喚起 [Save As] 對話盒。判斷文件內容是否被修改，並非我撰寫 MFCLite 的興趣所在（我只在乎 application framework 主軸），況且目前版本之 MFCLite 也未曾提供文件內容編撰功能。所以，MFCLite 一律詢問使用者是否要儲存檔案。

```
// MFCLite code
void CDocument::OnFileClose()
{
    if (!SaveModified()) // 判斷內容是否修改，詢問是否存檔
        return;
    // 以上 SaveModified() 會詢問存檔否，並在使用者回答 Yes 的情況下
    // 喚起 CDocument::OnFileSave()，見 6.9.3 節。

    OnCloseDocument();
    // 此將摧毀這個 document
}
```

- 走訪 document 所擁有的 m_viewList，找出每一個 views，並找出其父視窗，亦即 docFrame。分別摧毀這 view 視窗和 docFrame 視窗。一旦所有的 views 和 docFrames 都被摧毀，便摧毀 document 本身，並在解構式中解除 views 對 document 的連結 — 如果此刻還有殘餘的 views（一般而言應該是沒有了）。請注意，MFC 允許多個 views 安置於一個 docFrame 視窗內，但 MFCLite 為求簡化，並不允許如此，只許一個 docFrame 視窗容納一個 view（6.9.5 節早已提過這一點）。以下為 MFCLite 程式碼。

```
// MFCLite code
void CDocument::OnCloseDocument()
{
    BOOL bAutoDelete = m_bAutoDelete;
    m_bAutoDelete = FALSE; // 關閉 views 的過程中不摧毀 document
    while (!m_viewList.IsEmpty())
    {
        CView* pView = (CView*)m_viewList.GetHead();
        CFrameWnd* pFrame = pView->GetParentFrame();
    }
}
```

²³ 圖 6-33 對於 docManager, docTemplates, documents, views, docFrames 彼此之間的從屬關係，有很好的示意圖。思考本節的視窗關閉系統，一定需要圖 6-33 的協助。

```

        // 將 frame 關閉
        PreCloseFrame(pFrame); // 目前版本什麼也不做。
        pFrame->DestroyWindow();
        // 此動作會一併刪除 view
        // 侯捷註：這裡引發的動作極為複雜。見稍後說明。
    }
    m_bAutoDelete = bAutoDelete;

    // 摧毀 document 之前先清理文件內容。(這是個 template method)
    DeleteContents(); // 應用程式應該覆寫虛擬函式 DeleteContents()

    // 如果必要，刪除 document
    if (m_bAutoDelete) {
        delete this;
    }
}

```

```

// MFCLite code
CDocument::~CDocument()
{
    // 不應該留下任何 views。
    DisconnectViews();
    ASSERT(m_viewList.IsEmpty());

    if (m_pDocTemplate != NULL)
        m_pDocTemplate->RemoveDocument(this);
    // 撤除 document (this 所指) 和 doc template 之間的關係
    ASSERT(m_pDocTemplate == NULL); // 必須被解除關係 (detached)
}

```

- 每當 view 視窗被摧毀，CDocument::m_viewList 也應有所維護（減 1）。這個動作極為重要，但極隱晦。稍後另有說明。
- 由於 docFrame 是個 MDI 視窗，所以父視窗被摧毀時，子視窗（可能多個）應連帶被摧毀。MFC 的作法是呼叫 docFrame 的 DestroyWindow() 成員函式來摧毀 docFrame 視窗，並在 CMDIChildFrame::DestroyWindow() 函式中發出 WM_MDIDESTROY 訊息給 docFrame 視窗（它是個 MDI-frame 視窗），以期直接利用 Windows MDI 視窗管理系統本身能力。此後就由 Win32-MDI 層面接管，事情相對簡單：

```

// MFC code
BOOL CMDIChildWnd::DestroyWindow()
{
    ...
    CMDIFrameWnd* pFrameWnd = GetMDIFrame();
}

```

```

        HWND hWndFrame = pFrameWnd->m_hWnd;
        ...
        MDIDestroy();
        // 其內執行 ::SendMessage(..., WM_MDIDESTROY, ...);
        ...
    }

```

- Windows MDI 視窗管理系統會在摧毀 MDI-frame 視窗、MDI-client 視窗、MDI-child 視窗的同時，發給每個視窗的視窗函式（**window procedure**，或謂 **window function**）一個 WM_DESTROY 訊息。為此，MFC 分別在兩個函式 CMDIChildWnd::OnDestroy() 和 CView::OnDestroy() 承受了它們，前者服務 docFrame 視窗，後者服務 view 視窗。至於 MDI-client 視窗，一般而言隱藏不現²⁴。

```

void CMDIChildWnd::OnDestroy()
{
    ...
    CFrameWnd::OnDestroy();
}

```

```

void CFrameWnd::OnDestroy()
{
    ...
    CWnd::OnDestroy();
}

```

```

void CView::OnDestroy()
{
    ...
    CWnd::OnDestroy();
}

```

```

void CWnd::OnDestroy()
{
    ...
    Default();
}

```

- 雖然 MFC 把 MDI 視窗及其子視窗的摧毀動作交給 Win32-MDI 層面接管，但最終還需取回控制權，因為它還得維護文件的 m_viewList，也還得刪除

²⁴ 如果你曾經在 MFC 程式中用過 CSplitterWnd，你也許會知道，它就是在 MDI-client 視窗產生之際被「貼」在 MDI-client 視窗上。請參考《深入淺出 MFC》by 侯捷，第 13 章：「多重文件與多重顯示」。

各視窗對應的 `CWnd` object。為此，MFC 的 `CWnd` 攔截了 `WM_NCDESTROY` 訊息。此訊息一旦出現，代表視窗的 `non-client area` 已經被摧毀，是視窗毀滅的最後一個步驟。該訊息的處理常式 `CWnd::OnNcDestroy()` 的任務是喚起後繼動作：`this->PostNcDestroy()`²⁵；`CFrameWnd` 和 `CView` 覆寫了這個虛擬函式，為的是在 `frame` 視窗和 `view` 視窗被摧毀後，能自動將對應的 `CWnd*` object 刪除，以免發生記憶體洩漏（memory leak）。C++ objects 的刪除會引發解構式，我們正好可在 `CView::~CView()` 之中維護 `m_viewList`。

- MFCLite 並無上述精良的 MDI 管理系統（那自然，MDI 管理系統是 Windows USER 模組的一個大工程☺），權宜之計是在視窗實際被摧毀的地點（`::DestroyWindow()`）直接呼叫上述兩個 `OnNcDestroy()` 函式²⁶：

```
// MFCLite code
BOOL DestroyWindow(HWND hWnd) // 模擬 Win32 API
{
    // 摧毀視窗27。
    // 以下摧毀視窗。文字模式中無實相視窗，從視窗管理系統中除名即代表摧毀之。
    // 被刪除者若為 docFrame，必為 active，應更換 active docFrame。
    // 被刪除者若為 mainFrame，則最後會引發送出 WM_QUIT，結束程式。
    // 注意，MFCLite 假設一個 docFrame 只有一個 view
    CWnd* pFrame = ...; // get active docFrame
    CView* pView = ...; // get active view
    ...

    // 以下模擬 WM_NCDESTROY 效果（運用該訊息清理 C++ 相關物件）
    pFrame->OnNcDestroy(); // 針對 frame 送出 (sends) WM_NCDESTROY
    if (pView)
        pView->OnNcDestroy(); // 針對 view 送出 (sends) WM_NCDESTROY

    return TRUE;
}
```

- 任何需要在 `WM_NCDESTROY` 訊息發生之「後」（也就是視窗被摧毀之後）做

²⁵ 注意其函式名稱的意義：`PostNcDestroy()` 表示在 `WM_NCDESTROY` 訊息發生之「後」所做的動作。

²⁶ 如果只是為了釋出 `WM_NCDESTROY`，利用 MFCLite 已模擬完成之 `::PostMessage()` 也行。但 MFC 用的是 `::SendMessage()`，而它有同步（Sync.）效果，MFCLite 無法模擬出來。唯一能夠模擬同步效果的，就是直接呼叫這裡所列的兩個函式。

²⁷ 任何一個視窗被摧毀，MFCLite 都應維護其自身的一個簡易視窗管理系統 `g_state`，其中記錄有 `mainFrame`, `docFrames`, `views` 視窗的 `hWnd` 和 `CWnd*`。

善後清理工作的 CWnd-derived classes，都可以（並且應該）覆寫虛擬函式 PostNcDestroy()。下面是 CFrameWnd 和 CView 的實現：

```
void CFrameWnd::PostNcDestroy()
{
    // frame window 總是從 heap 中配置而來，因此預設的 post-cleanup
    // 動作就是 'delete this'。千萬不要對著 CFrameWnd 呼叫 delete。
    // 應以 DestroyWindow 取而代之。
    delete this;
    // 無論 this 是 CMDIFrameWnd* 或 CMDIChildWnd*，這個 delete 動作最終
    // 總是會喚起 CFrameWnd::~~CFrameWnd()。參考 2.x 節。
}
```

```
CFrameWnd::~~CFrameWnd()
{
    /*
    // 以下是 MFC 源碼。MFCLite 簡化了視窗管理，所以不需要。
    RemoveFrameWnd();
    if (m_phWndDisable != NULL)
        delete[] (void*)m_phWndDisable;
    */
}
```

```
void CView::PostNcDestroy()
{
    // views 總是從 heap 中配置而來，因此預設的 post-cleanup 動作
    // 就是 'delete this'。千萬不要對著 view 呼叫 delete。
    delete this;
    // 無論 this 是 CMyView1* 或 CMyView2*，這個 delete 動作最終
    // 總是會喚起 CView::~~CView()。參考 2.x 節。
}
```

```
CView::~~CView()
{
    if (m_pDocument != NULL)
        m_pDocument->RemoveView(this); //維護 CDocument::m_viewList
}
```

上述整個思考與實作流程於圖 6-44 有綜合性的示意圖，其中圖(a) 解釋 MFC 的行為，圖(b) 解釋 MFCLite 的行為。

MFC 文件關閉系統

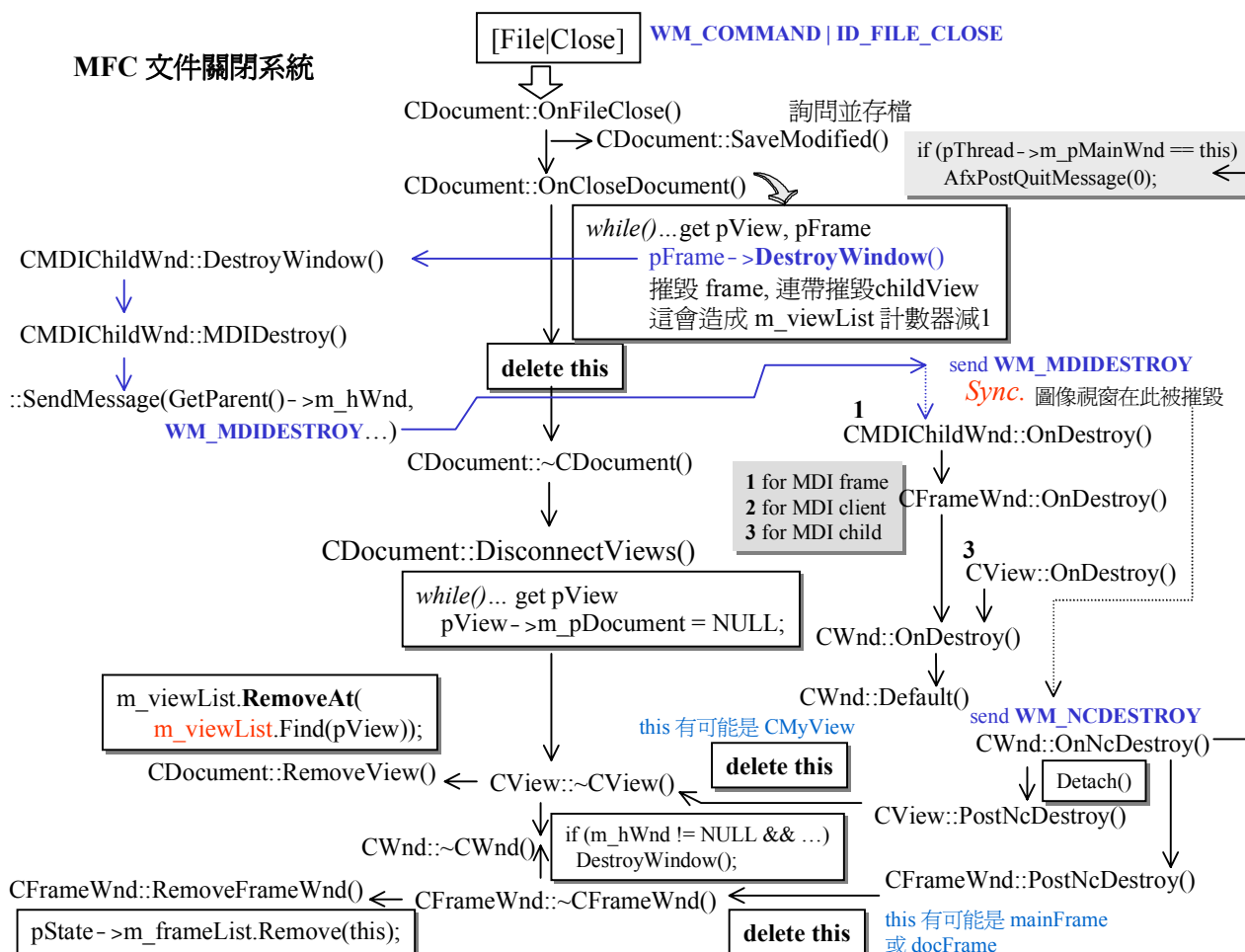


圖 6-44a MFC 文件關閉系統

MFCLite 文件關閉系統

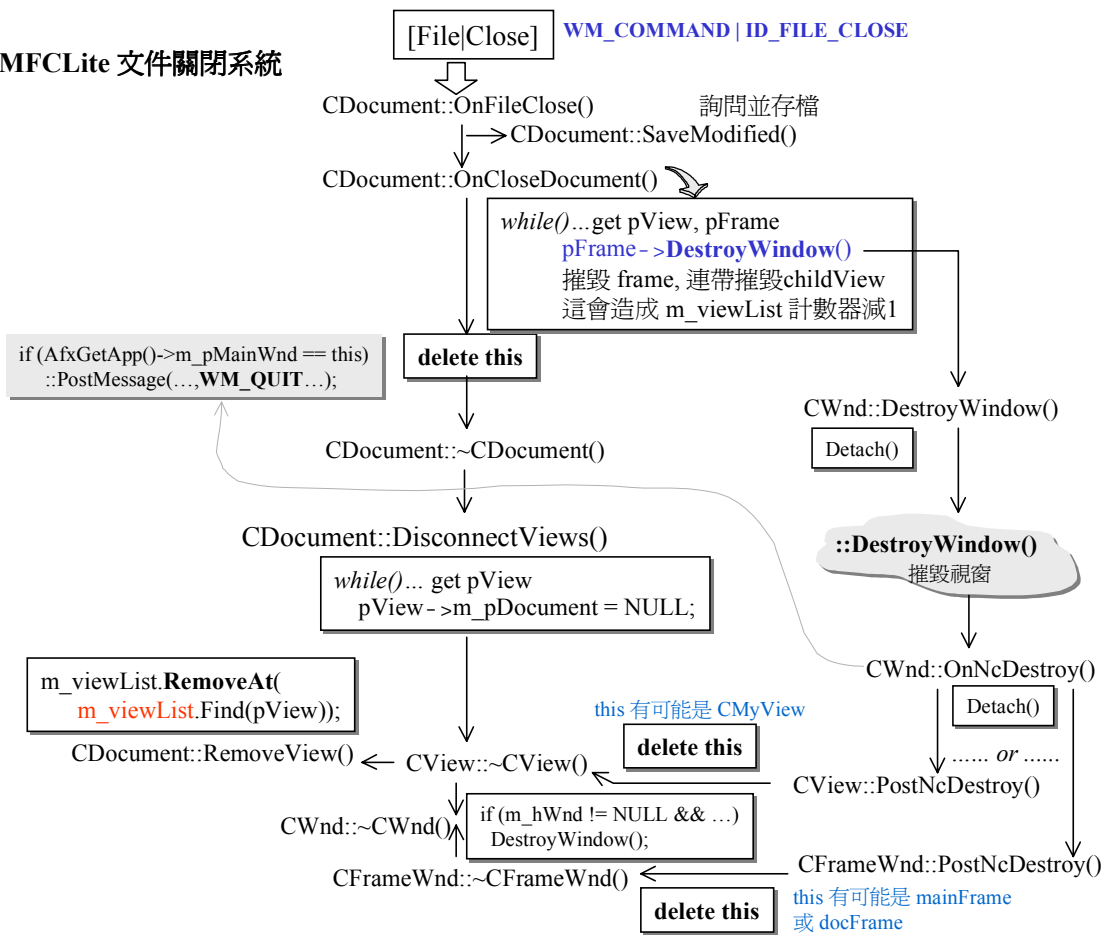


圖 6-44b MFCLite 文件關閉系統

關閉視窗：CFrameWnd::OnClose() 熱鍵 'c' 或 'x'

使用者按下熱鍵 'c' 是爲了關閉 active docFrame 視窗（連帶關閉相應文件），按下 'x' 是爲了關閉 mainFrame 視窗（連帶關閉所有文件）。這兩個熱鍵既然都是爲了關閉視窗，都釋出 WM_CLOSE 訊息，理應都由 CFrameWnd::OnClose() 負責²⁸。由於收到 WM_CLOSE 訊息的視窗可能是 docFrame（熱鍵 'c'）也可能是 mainFrame（熱鍵 'x'），所以在此函式中，this 指標的型別可能是 CMDIChildWnd* 也可能是 CMDIFrameWnd*。

CFrameWnd::OnClose() 的主要任務是處理（儲存、關閉）視窗內的文件，然後摧毀自己（視窗）。下面是思考重點：

- 如果 this object 不是 mainFrame（則必爲 docFrame），必須詢問並於獲得肯定答案後將相應文件儲存起來。詢問並儲存文件的動作已於稍早「關閉文件」小節中實現出來，現在只需喚起它即可。見圖 6-45a 的綠線流程。
- 如果 this object 是 mainFrame，必須設法將所有 docFrames 內的相應文件儲存起來（當然應該先詢問使用者的意願）。見圖 6-45a 的紅線線條。
- 如果 this object 是 mainFrame，還必須關閉其所管轄的所有文件。文件關閉動作已於稍早「關閉文件」小節中實現出來（其中包含 docFrame 視窗的關閉和 view 視窗的關閉），現在只需喚起它即可。見圖 6-45a 的黑線流程。
- 最後，無論自身視窗是 docFrame 或 mainFrame，都應該關閉自己。見圖 6-45a 的藍線流程。

這樣的設計，能夠充份運用先前針對「文件關閉」所做的努力成果。圖 6-45a 表現十分清楚，我就不再書上擺放源碼了。

如果上述的 this object 是個 mainFrame 視窗，上述最後一個步驟會喚起 CWnd::OnNcDestroy()，其中會釋出 WM_QUIT，如下：

```
void CWnd::OnNcDestroy()
{
```

²⁸ 之所以不分別設計 CMDIFrameWnd::OnClose() 和 CMDIChildWnd::OnClose()，是因為在這兩個 classes 內對於 WM_CLOSE 訊息並無太多特徵行爲。唯一的差別在於針對 CMDIFrameWnd 時意味著結束程式，而這項差別在 CFrameWnd::OnClose() 中透過一些動作即可測知。

```

if (this == AfxGetApp()->m_pMainWnd) // 被摧毀的是 mainFrame。
    ::PostMessage(this->m_hWnd, WM_QUIT, 0, 0); // 設法結束程式。
// 以上與 MFC 處理型式略有不同。MFC 判斷的是 thread 的 m_pMainWnd，
// 呼叫的是 AfxPostQuitMessage(0)。基本上與上述型式的意義相同。
}

```

進而結束訊息迴圈（見 6.6.3 節及圖 6-19），進而結束 main() 或 WinMain()，進而結束全域物件 CMyWinApp theApp 的生命，進而喚起其解構式，那正是釋放資源的好時機²⁹。此時的 documents, views, docFrames 都已釋放殆盡，剩下需要釋放的資源是 CDocManager object 和 CDocTemplate object。圖 6-45b 展示其間作法。

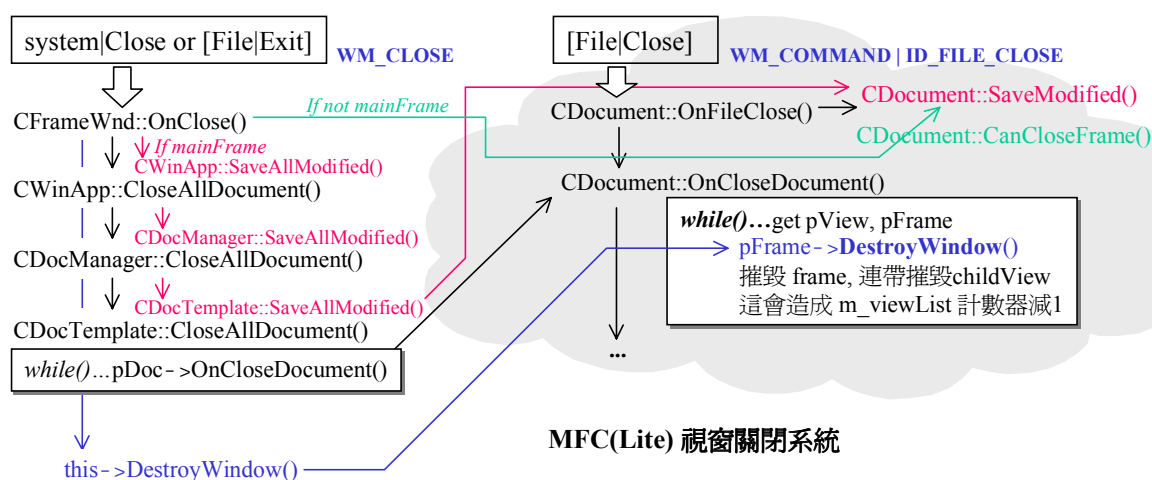


圖 6-45a MFC(Lite) 視窗關閉系統

²⁹ 6.12.3 節一開始討論的 MFCLite 原先版本的錯誤作法，便是在此處釋放 documents, views, docFrames。那不是好的設計。

當訊息迴圈結束，程式即將離開 MFCLite 的 `main()` 或 MFC 的 `WinMain()`，於是全域物件 `CMyWinApp theApp` 的解構式被喚起，於是引發資源釋放：

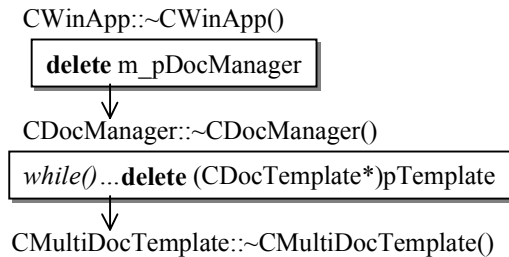


圖 6-45b MFC(Lite) 視窗關閉系統

結束 程式： `CWinApp::OnAppExit()` 熱鍵 ‘X’

這個函式一旦被喚起，表示使用者決定結束程式。既然 `CFrameWnd::OnClose()` 中已經考量了程式結束的處理情況，此處只要直接將 `WM_CLOSE` 訊息丟給 `mainFrame` 視窗就好了：

```

void CWinApp::OnAppExit()    // [File|Exit] handler
{
    // same as double-clicking on main window close box
    ASSERT(m_pMainWnd != NULL);

    // m_pMainWnd->SendMessage(WM_CLOSE);
    // MFC 使用上述的 SendMessage()。
    // MFCLite 只支援 PostMessage()，尚可。
    ::PostMessage(this->m_pMainWnd->m_hWnd, WM_CLOSE, 0, 0);
}
  
```

本節（6.12）為原先的 MFCLite 再添兩大性質，一是更完善的物件永續（Object Persistence）設計，一是視窗 / 文件關閉系統。都是大傢伙。

