《深入浅出MFC》2/e

电子书开放自由下载 声明

致亲爱的大陆读者

我是侯捷（侯俊杰）。自从华中理工大学于1998/04 出版了我的《深入浅出MFC》1/e 简体版（易名《深入浅出Windows MFC 程序设计》）之后，陆陆续续我收到了许多许多的大陆读者来函。其中对我的赞美、感谢、关怀、殷殷垂询，让我非常感动。

《深入浅出MFC》2/e 早已于1998/05 于台湾出版。之所以迟迟没有授权给大陆进行简体翻译，原因我曾于回复读者的时候说过很多遍。我在此再说一次。

1998 年中，本书之发行公司松岗（UNALIS）即希望我授权简体版，然因当时我已在构思3/e，预判3/e 繁体版出版时，2/e 简体版恐怕还未能完成。老是让大陆读者慢一步看到我的书，令我至感难过，所以便请松岗公司不要进行2/e 简体版之授权，直接等3/e 出版后再动作。没想到一拖经年，我的3/e 写作计划并没有如期完成，致使大陆读者反而没有《深入浅出MFC》2/e 简体版可看。

《深入浅出MFC》3/e 没有如期完成的原因是，MFC 本体架构并没有什么大改变。《深入浅出MFC》2/e 书中所论之工具及程序代码虽采用VC5+MFC42，仍适用于目前的VC6+MFC421（唯，工具之画面或功能可能有些微变化）。

由于《深入浅出MFC》2/e 并无简体版，因此我时时收到大陆读者来信询问购买繁体版之管道。一来我不知道是否台湾出版公司有提供海外邮购或电购，二来即使有，想必带给大家很大的麻烦，三来两岸消费水平之差异带给大陆读者的负担，亦令我深感不安。

因此，此书虽已出版两年，鉴于仍具阅读与技术上的价值，鉴于繁简转译制作上的费时费工，鉴于我对同胞的感情，我决定开放此书内容，供各位免费阅读。

我已为《深入浅出MFC》2/e 制作了PDF 格式之电子文件， 放在 http://www.jjhou.com 供自由下载。北京http://expert.csdn.net/jjhou 有侯捷网站的一个GBK mirror，各位也可试着自该处下载。

我所做的这份电子书是繁体版，我没有精力与时间将它转为简体。这已是我能为各位尽力的极限。如果（万一）您看不到文件内容，可能与字形的安装有关-虽然我已尝试内嵌字形。anyway，阅读方面的问题我亦没有精力与时间为您解决。请各位自行开辟讨论区，彼此交换阅读此电子书的solution。请热心的读者告诉我您阅读成功与否，以及网上讨论区（如有的话）在哪里。

曾有读者告诉我，《深入浅出MFC》1/e 简体版在大陆被扫描上网。亦有读者告诉我，大陆某些书籍明显对本书侵权（详细情况我不清楚）。这种不尊重作者的行为，我虽感遗憾，并没有太大的震惊或难过。一个社会的进化，终究是一步一步衍化而来。台湾也曾经走过相同的阶段。但盼所有华人，尤其是我们从事智能财产行为者，都能够尽快走过灰暗的一面。

在现代科技的协助下，文件影印、文件复制如此方便，智财权之尊重有如「君子不欺暗室」。没有人知道我们私下的行为，只有我们自己心知肚明。《深入浅出MFC》2/e 虽免费供大家阅读，但此种作法实非长久之计。为计久长，我们应该尊重作家、尊重智财，以良好（至少不差）的环境培养有实力的优秀技术作家，如此才有源源不断的好书可看。

我的近况，我的作品，我的计划，各位可从前述两个网址获得。欢迎各位写信给我（jjhou@ccca.nctu.edu.tw）。虽然不一定能够每封来函都回复，但是我乐于知道读者的任何点点滴滴。

关于《深入浅出 MFC》2/e 电子书

《深入浅出 MFC》2/e 电子书共有五个档案：

档名	內容	大小 bytes
dissecting MFC 2/e part1.pdf	chap1~chap3	3,384,209
dissecting MFC 2/e part2.pdf	chap4	2,448,990
dissecting MFC 2/e part3.pdf	chap5~chap7	2,158,594
dissecting MFC 2/e part4.pdf	chap8~chap16	5,171,266
dissecting MFC 2/e part5.pdf	appendix A,B,C,D	1,527,111

每个档案都可个别阅读。每个档案都有书签（亦即目录连接）。每个档案都不需密码即可打开、选择文字、打印。

请告诉我您的资料

每一位下载此份电子书的朋友，我希望您写一封 email 給我（jjhou@ccca.nctu.edu.tw），告诉我您的以下資料，俾让我对我的读者有一些基本瞭解，谢谢。

姓名：
现职：
毕业学校科系：
年龄：
性别：
居住省份（如是台湾读者，请写县市）：
对侯捷的建议：

-- the end

浅出 MFC 程序设计

第 5 章

总观 Application Framework

带艺术气息的软件创作行为将在Application Framework出现后逐渐成为工匠技术，
而我们都将成为软件IC装配厂里的男工女工。
但，不是亨利福特，我们又如何能够享受大众化的汽车？
或许以后会出现「纯手工精制」的软件，可我自己从来不嫌机器馒头难吃。

什么是 Application Framework？

还没有学习任何一套Application Framework 的使用之前，就给你近乎学术性的定义，我可以想象对你而言绝对是「形而上的」（超物质的无形哲理），尤其如果你对对象导向（Object Oriented）也还没有深刻体会的话。形而上者谓之道，形而下者谓之器，我想能够舍器而直接近道者，几稀！但是，「定义」这种东西又似乎宜开宗明义摆在前头。我诚挚地希望你在阅读后续的技术章节时能够时而回来看看这些形而上的叙述。当你有所感受，技术面应该也进入某个层次了。

侯捷怎么说

首先我们看看侯捷在其无责任书评中是怎么说的：

演化(revolution)永远在进行，但这个世界却不是每天都有革命性(revolution)的事物发生。动不动宣称自己（或自己的产品）是划时代的革命性的，带来的影响就像时下满街跑的大师一样使我们渐渐无动于衷（大师不可能满街跑）！但是Application Framework 的的确确在我们软件界称得上具有革命精神。

什么是Application Framework？Framework 这个字眼有组织、框架、体制的意思，Application Framework不仅是一般性的泛称，它其实还是对象导向领域中的一个专有名词。

基本上你可以说,Application Framework是一个完整的程序模型，具备标准应用软件所需的一切基本功能，像是文件存取、打印预视、资料交换...，以及这些功能的使用接口（工具栏、状态列、菜单、对话框）。如果更以术语来说,Application Framework 就是由一整组合作无间的「对象」架构起来的大模型。喔不不，当它还没有与你的程序产生火花的时候，它还只是有形无体，应该说是一组合作无间的「类别」架构起来的大模型。这带来什么好处呢？程序员只要带个购物袋到「类别超级市场」采买，随你要买MDI 或OLE 或ODBC 或Printing Preview，回家后就可以轻易拼凑出一个色香味俱全的大餐。「类别超级市场」就是C++ 类别库，以产品而言，在Microsoft 是MFC，在Borland 是OWL，在IBM 则是OpenClass。这个类别库不只是类别库而已，传统的函数库（C Runtime 或Windows API）乃至于一般类别库提供的是生鲜超市中的一条鱼一支葱一颗大白菜，彼此之间没有什么关联，主掌中馈的你必须自己选材自己调理。能够称得上Application Framework者，提供的是火锅拼盘（就是那种带回家通通丢下锅就好的那种），依你要的是白菜火锅鱼头火锅或是麻辣火锅，菜色带调理包都给你配好。当然这样的火锅拼盘是不能够就地吃的，你得给它加点能量。放把火烧它吧，这火就是所谓的application object（在MFC 程序中就是衍生自CWinApp的一个全域性对象）。是这

个对象引起了连锁反应（一连串的'new'），使每一个形（类别）有了真正的体（对象），把应用程序以及Application Framework 整个带动起来。一切因缘全由是起。

Application Framework 带来的革命精神是，程序模型已经存在，程序员只要依个人需求加料就好：在衍生类别中改写虚拟函数，或在衍生类别中加上新的成员函数。这很像你在火锅拼盘中依个人口味加盐添醋。

由于程序代码的初期规模十分一致（什么样风格的程序应该使用什么类别，是一成不变的），而修改程序以符合私人需要的基本动作也很一致（我是指像「开辟一个空的骨干函数」这种事情），你动不了Application Framework 的大架构，也不需要动。这是福利不是约束。

应用程序代码骨干一致化的结果，使优越的软件开发工具如CASE（Computer Aid Software Engineering）tool 容易开发出来。你的程序代码大架构掌握在Application Framework 设计者手上，于是他们就有能力制作出整合开发环境（Integrated Development Environment，IDE）了。这也是为什么Microsoft、Borland、Symantec、Watcom、IBM 等公司的整合开发环境进步得如此令人咋舌的原因了。

有人说工学院中唯一保有人文气息的只剩建筑系，我总觉得信息系也勉强可以算上。带艺术气息的软件创作行为（我一直是这么认为的）将在Application Framework 出现后逐渐成为工匠技术，而我们都将只是软件IC 装配厂里的男工女工。其实也没什么好顾影自怜，功成名就的冠冕从来也不曾落在程序员头上；我们可能像纽约街头的普普（POP）工作者，自认为艺术家，可别人怎么看呢？不得而知！话说回来，把开发软件这件事情从艺术降格到工技，对人类只有好处没有坏处。不是亨利福特，我们又如何能够享受大众化的汽车？或许以后会出现「纯手工精制」的软件，谁感兴趣不得而知，我自己嘛...唔...倒是从来不嫌机器馒头难吃。

如果要三言两语点出Application Framework 的特质，我会这么说：我们挖出别人早写好的一整套模块（MFC 或OWL 或OpenClass）之中的一部份，给个引子（application object）使它们一一具象化动起来，并被允许修改其中某些零件使这程序更符合私人需

求，如是而已。

我怎么说

侯捷的这一段话实在已经点出Application Framework 的精神。凝聚性强、组织化强的类别库就是Application Framework。一组合作无间的对象，彼此藉消息的流动而沟通，并且互相调用对方的函数以求完成任务，这就是Application Framework。对象存在哪里？在MFC 中?! 这样的说法不是十分完善，因为MFC 的各个类别只是「对象属性（行为）的定义」而已，我们不能够说MFC 中有实际的对象存在。唯有当程序被application object（这是一个衍生自MFC *CWinApp* 的全域对象）引爆了，才将我们选用的类别一一具象化起来，产生实体并开始动作。图5-1 是一个说明。

这样子说吧，静态情况下MFC 是一组类别库，但在程序执行时期它就生出了一群有活动力的对象组。最重要的一点是，这些对象之间的关系早已建立好，不必我们（程序员）操心。好比说当使用者按下菜单的【File/Open】项，开文件对话框就会打开；使用者选好档名后，Application Framework 就开始对着你的资料类别，唤起一个名为*Serialize* 的特殊函数。这整个机制都埋好了，你只要把心力放在那个叫作*Serialize* 的函数上即可。

选用标准的类别，做出来的产品当然就没有什么特色，因为别人的零件和你的相同，兜起来的成品也就一样。我指的是使用者接口（UI）对象。但你要知道，软件工业发展到现阶段这个世代，着重的已不再是UI 的争奇斗艳，取巧哗众；UI 已经渐渐走上标准化了。软件一决胜负的关键在资料的处理。事实上，在「真正做事」这一点，整个application framework 是无能为力的，也就是说对于数据结构的安排，数据的处理，数据的显示，Application Framework 所能提供的，无一不是单单一个空壳而已-- 在C++ 语言来讲就是个虚拟函数。软件开发人员必须想办法改造（override）这些虚拟函数，才能符合个人所需。基于C++ 语言的特性，我们很容易继承既有之类别并加上自己的特色，这就是物件导向程序设计的主要精神。也因此，C++ 语言中有关于「继承」性质的份量，在MFC 程序设计里头占有很重的比例，在学习使用MFC 的同时，你应该对C++ 的继承性质和虚拟函数有相当的认识。第 2 章有我个人对C++ 这两个性质的心得。

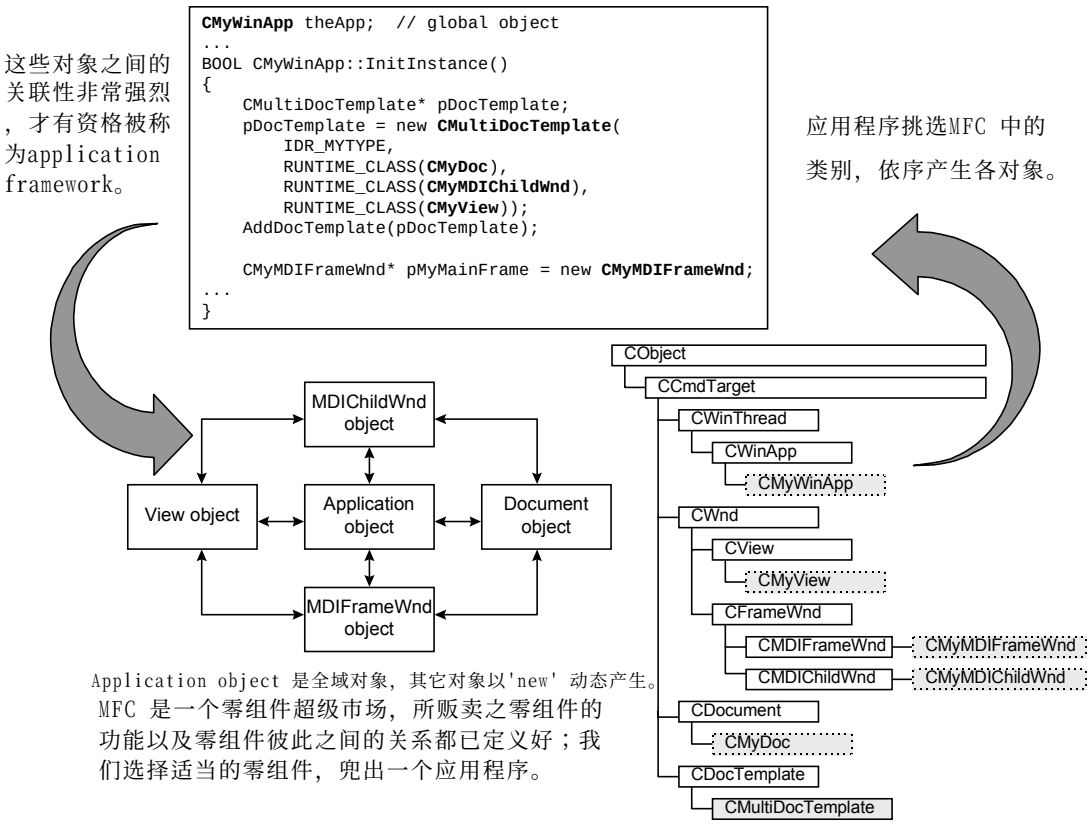


图5-1 MFC 是一个零组件超级市场，所贩卖的零组件功能以及零组件彼此之间的关系都已定义好；我们选择自己喜欢的零件，兜出一个应用程序。

Application Framework 究竟能提供我们多么实用的类别呢？或者我们这么问：哪些工作已经被处理掉了，哪些工作必须由程序员扛下来？比方说一个标准的MFC 程序应该有一个可以读写文件资料的功能，然而应用程序本身有它独特的数据结构，从MFC 得来的零件可能与我的个人需求搭配吗？

我以另一个比喻做回答。

假设你买了一个整厂整线计划，包括仓贮、物料、MIS、生管，各个部门之间的搭配也都建立起来了，包含在整厂整线计划内。这个厂原先是为了生产葡萄酒，现在改变主意要生产白兰地，难道整个厂都不能用了吗？不！只要把进货原料改一改、发酵程序改一改、瓶装程序改一改、整个厂的其它设备以及设备与设备之间的联机配合（这是顶顶重要的）都可以再利用。物料之后到生管，装瓶之后到仓贮，仓贮之后到出货，再加上MIS监控全厂，这些程序都是不必改变的。

一个整厂整线计划，每一单元之间的联机沟通，合作关系，都已经建立起来，是一种建构好的运作模式。抽换某个单元的性质或部份性质，并不影响整体操作。「整厂整线」最重要最有价值的是各单元之间的流程与控制。

反映到对象导向程序设计里头，Application Framework 就是「整厂整线规划」，Application Framework 提供的类别就是上述工厂中一个一个的单元。最具价值的就是各类别之间的交互运作模式。

虽然文件读写（此动作在MFC 称为Serialize）单元必须改写以符合个人所需，但是单元与单元之间的关系依然存在而且极富价值。当使用者在程序中选按【File/Open】或【File/Save】，主框窗口自动通知Document 对象（内存资料），引发Serialization 动作；而你为了个人的需求，改写了这个*Serialize* 虚拟函数。你对MFC 的改写范围与程度，视你的程序有多么特异而定。

可是如果酿酒厂想改装为炼钢厂？那可就无能为力了。这种情况不会出现在软件开发上，因为软件的必备功能使它们具有相当的相似性（尤其Windows 又强调接口一致性)。好比说程序想支持MDI 接口，想支持OLE，这基本上是超越程序之专业应用领域之外的一种大格局，一种大架构，最适合Application Framework 发挥。

很明显，Application Framework 是一组超级的类别库。能够被称为Framework 者必须其中的类别性质紧密咬合，互相呼应。因此你也就可以想象，Framework 所提供的类别是一伙的，不是单片包装的。你把这伙东西放入程序里，你也就得乖乖遵循一种特定的（Application Framework 所规定的）程序风格来进进程序设计工作。但是侯捷也告诉我

们，这是福利不是约束。

别人怎么说

其它人又怎么看Application Framework？我将胪列数篇文章中的相关定义。若将原文译为中文，我恐怕力有未逮辞不达意，所以列出原文供你参考。

1985 年，Apple 公司的MacApp 严格而系统化地定义出做为一个商业化Application Framework 所需要的关键理念：

> The key ideas of a commercial application framework : a generic app on steroids that provides a large amount of general-purpose functionality within a well-planned, well-tested, cohesive structure.

*cohesive 的意思是强而有力、有凝聚力的。

*steroid 是类固醇。自从加拿大100 公尺名将班强生在汉城奥运吃了这药物而夺得金牌并打破世界记录，相信世人对这个名称不会陌生（当然强生的这块金牌和他的世界记录后来是被取消的）。类固醇俗称美国仙丹，是一种以胆固醇结构为基础，衍生而来的荷尔蒙，对于发炎红肿等症状有极速疗效。然而因为它是透过抑制人类免疫系统而得到疗效，如果使用不当，会带来极不良的副作用。运动员用于短时间内增强身体机能的雄性激素就是类固醇的一种，会影响脂肪代谢，服用过量会导至极大的副作用。

基本上MacApp 以类固醇来比拟Application Framework 虽是妙喻，但类固醇会对人体产生不好的副作用而Application Framework 不会对软件开发产生副作用-- 除非你认为不能随心所欲写你的码也算是一种副作用。

Apple 更一步更明确地定义一个Application Framework是：

> an extended collection of classes that cooperate to support a complete application architecture or application model, providing more complete application development support than a simple set of class libraries.

* 这里所指的support 并不只是视觉性UI 组件如menu、dialog、listbox...，还包括一个应用程序所需要的其它功能设备，像是Document，View，Printing，Debugging。

另一个相关定义出现在Ray Valdes 于1992 年10 月发表于Dr. Dobb's Journal 的"Sizing up Application Frameworks and Class Libraries" 一文之中：

> An application framework is an integrated object-oriented software system that offers all the application-level classes（documents, views, and commands）needed by a generic application.
>
> An application framework is meant to be used in its entirety, and fosters both design reuse and code reuse. An application framework embodies a particular philosophy for structuring an application, and in return for a large mass of prebuilt functionality, the programmer gives up control over many architectural-design decisions.

Donald G. Firesmith 在一篇名为"Frameworks : The Golden path of the object Nirvana" 的文章中对Application Framework 有如下定义：

> What are frameworks？They are significant collections of collaborating classes that capture both the small-scale patterns and major mechanisms that, in turn, implement the common requirements and design in a specific application domain.

* Nirvana 是涅盘、最高境界的意思。

Bjarne Stroustrup（C++ 原创者）在他的The C++ Programming Language 一书中对于 Application Framework 也有如下叙述：

> Libraries build out of the kinds of classes described above support design and re-use of code by supplying building blocks and ways of combining them; the application builder designs a framework into which these common building blocks are fitted. An alternative, and sometimes more ambitious, approach to the support of design and re-use is to provide code that establishes a common framework into which the application builder fits application-specific code as building blocks. Such an approach is often called an application framework. The classes establishing such a framework often have such fat interfaces that they are hardly types in the traditional sense. They approximate the ideal of being complete applications, except that they don't do anything. The specific actions are supplied by the application programmer.

Kaare Christian 在1994/02/08 的PC Magazine 中有一篇"C++ Application Frameworks" 文章，其中有下列叙述（节录）：

两年前我在纽约北边的乡村盖了一栋post-and-beam 房子。在我到达之前我的木匠已经把每一根梁的外形设计好并制作好，把一根根的粗糙木材变成一块块锯得漂漂亮亮的零件，一切准备就线程只待安装。(注：所谓post-and-beam 应是指那种梁柱都已规格化，可以邮购回来自己动手盖的DIY ¡ V Do It Yourself -- 房子）。

使用Application Framework建造一个 Windows应用程序也有类似的过程。你使用一组早已做好的零件，它使你行进快速。由于这些零件坚强耐用而且稳固，后面的工作就简单多了。但最重要的是，不论你使用规格化的梁柱框架来盖一栋房子，或是使用Application Framework来建立一个Windows程序，工作类型已然改变，出现了一种完全崭新的做事方法。在我的post-and-beam 房子中，工作类型的改变并不总是带来帮助；贸易商在预制梁柱的技巧上可能会遭遇适应上的困扰。同样的事情最初也发生在Windows 身上，因为你原已具备的某些以C 语言写Windows 程序的能力，现在在以

C++ 和Application Framework 开发程序的过程中无用武之地。时间过去之后，Windows程序设计的类型移转终于带来了伟大的利益与方便。Application Framework本身把message loops 和其它Windows 的苦役都做掉了，它促进一个比较秩序井然的程序结构。

Application Framework --建立Windows应用软件所用的C++类别库--如今已行之有年，因为对象导向程序设计已经快速地获得了接受度。Windows API 是程序性的，Application Framework则让你写对象导向式的Windows程序.它们提供预先写好的机能（以C++ 类别型式呈现出来），可以加速应用软件的开发。

Application Framework提供数种优点.或许最重要的,是它们在对象导向程序设计模式下对Windows 程序设计过程的影响。你可以使用Framework 来减轻例行但繁复的琐事,目前的Application Framework可以在图形、对话框、打印、求助、OCX控制组件、剪贴簿、OLE 等各方面帮助我们，它也可以产生漂亮的UI 接口如工具栏和状态列。

借着Application Framework的帮助写出来的码往往比较容易组织化,因为Framework改变了Windows 管理消息的方法。也许有一天Framework 还可以帮你维护单一一套码以应付不同的执行平台。

你必须对Application Framework有很好的知识,才能够修改由它附带的软件开发工具制作出来的骨干程序。它们并不像Visual Basic 那么容易使用。但是对Application Framework 专家而言，这些程序代码产生器可以省下大量时间。

使用Application Framework 的主要缺点是,没有单一一套产品广被所有的C++ 编译器支持。所以当你选定一套Framework，在某个范围来说，你也等于是选择了一个编译器。

为什么使用Application Framework

虽然Application Framework 并不是新观念，它们却在最近数年才成为PC 平台上软件开发的主流工具。对象导向语言是具体实现Application Framework 的理想载具，而C++ 编译器在PC 平台上的出现与普及终于允许主流PC 程序员能够享受Application Framework 带来的利益。

从八十年代早期到九十年代初始，C++ 大都存在于UNIX 系统和研究人员的工作站中，不在PC 以及商业产品上。C++ 以及其它的对象导向语言（例如Smalltalk-80）使一些大学和研究计划生产出现今商业化Application Framework 的鼻祖。但是这些早期产品并没有明显区隔出应用程序与Application Framework 之间的界线。

今天应用软件的功能愈来愈复杂，建造它们的工具亦复如此。Application Framework、Class Library 和GUI toolkits 是三大类型的软件开发工具（请见方块说明），这三类工具虽然以不同的技术方式逼近目标，它们却一致追求相同而基本的软件开发关键利益：降低写程序代码所花的精力、加速开发效率、加强可维护性、增加强固性（robustness）、为组合式的软件机能提供杠杆支点（有了这个支点，再大的软件我也举得起来）。

当我们面临软件工业革命，我们的第一个考量点是：我的软件开发技术要从哪一个技术面切入？从raw API 还是从高阶一点的工具？如果答案是后者，第二个考量点是我使用哪一层级的工具？GUI toolkits 还是Class Library 还是Application Framework？如果答案又是后者，第三个考量点是我使用哪一套产品？MFC 或OWL 或Open Class Library？（目前PC 上还没有第四套随编译器附赠的Application Framework 产品）

别认为这是领导者的事情不是我（工程师）的事情，有这种想法你就永远当不成领导者。也别认为这是工程师的事情不是我（学生）的事情，学生的下一步就是工程师；及早想点工业界的激烈竞争，对你在学生阶段规划人生将有莫大助益。

我相信，Application Framework 是最好的杠杆支点。

Application Framework，Class Library，GUI toolkit

一般而言，Class Library 和GUI toolkit 比Application Framework 的规模小，定位也没那么高阶宏观。Class Library 可以定义为「一组具备对象导向性质的类别，它们使应用程序的某些功能实现起来容易一些，这些功能包括数值运算与数据结构、绘图、内存管理等等等；这些类别可以一片一片毫无瓜葛地并入应用程序内」。
请特别注意这个定义中所强调的「一片一片毫无瓜葛」，而不像Application Framework 是大伙儿一并加入。因此，你尽可以随意使用Class Library，它并不会强迫你遵循任何特定的程序架构。Class Library 通常提供的不只是UI 功能、也包括一般性质的机能，像数据结构的处理、日期与时间的转换等等。
GUI toolkit 提供的服务类似Class Library，但它的程序接口是程序导向而非对象导向。而且它的功能大都集中在图形与UI 接口上。GUI toolkit 的发展历史早在对象导向语言之前，某些极为成功的产品甚至是以汇编语言（assembly）写成。不要必然地把GUI 联想到Windows，GUI toolkit 也有DOS 版本。我用过的Chatter Box 就是DOS 环境下的GUI 工具（是一个函数库）。

使用Application Framework 的最直接原因是，我们受够了日益暴增的Windows API。把MFC 想象为第四代语言，单单一个类别就帮我们做掉原先要以一大堆APIs 才能完成的事情。

但更深入地想，Application Framework 绝不只是为了降低我们花在浩瀚无涯的Windows API 的时间而已；它所带来的对象导向程序设计观念与方法，使我们能够站在一群优秀工程师（MFC 或OWL 的创造者）的努力心血上，继承其成果而开发自己之所需。同时，因为Application Framework 特殊的工作类型，整体开发工具更容易制作，也制作的更完美。在我们决定使用Application Framework 的同时，我们也获得了这些整合性软件开发环境的支持。在软件开发过程中，这些开发工具角色之吃重不亚于Application Framework 本身。

Application Framework 将成为软件技术中最重要的一环。如果你不知道它是什么，赶快学习它；如果你还没有使用它，赶快开始用。机会之窗不会永远为你打开，在你的竞争者把它关闭之前赶快进入！如果你认为改朝换代还早得很，请注意两件事情。第一，江山什么时候变色可谁也料不准，当你埋首工作时，外面的世界进步尤其飞快；第二，物件导向和Application Framework 可不是那么容易学的，花多少时间才能登堂入室可还得凭各人资质和基础呢。

浩瀚无涯的Windows API

Windows 版本	推出日期	API 个数	消息个数
1.0	1985.11	379	？
2.0	1987.11	458	？
3.0	1990.05	578	？
Multimedia Ex.	1991.12	120	？
3.1	1992.04	973	271
Win32s	1993.08	838	287
Win32	1993.08	1449（持续增加当中）	291（持续增加当中）

Microsoft Foundation Classes（MFC）

PC 世界里出了三套C++ Application Frameworks，并且有愈多愈多的趋势。这三套是Microsoft 的MFC （ Microsoft Foundation Classes ） ， Borland 的OWL （ Object WindowLibrary），以及IBM VisualAge C++ 的Open Class Library。至于其它C++ 编译器厂商如Watcom、Symantec、Metaware，只是供应整合开发环境（Integraded Development Environment，IDE），其Application Framework 都是采用微软公司的MFC。

> Delphi（Pascal 语言），依我之见，也称得上是一套Application Framework。Java 语言本身内建一套标准类别库，依我之见，也够得上资格被称为Application Framework。
> Delphi 和Visual Basic，又被称为是一种应用程序快速开发工具（RAD，Rapid Application Development）。它们采用PME（Properties-Method-Event）架构，写程序的过程像是在一张画布上拼凑一个个现成的组件（components）：设定它们的属性（properties）、指定它们应该「有所感」的外来刺激（events），并决定它们面对此刺激时在预设行为之外的行为（methods）。所有动作都以拖拉、设定数值的方式完成，非常简单。只有在设定组件与组件之间的互动关系时才牵涉到程序代码的写作（这一小段码也因此成为顺利成功的关键）。
> Borland 公司于1997 年三月推出的C++ Builder 也属于PME 架构，提供一套Visual Component Library（VCL），内有许许多多的组件。因此C++ Builder 也算得上是一套RAD（应用程序快速开发工具）。

早初，开发Windows 应用程序必须使用微软的SDK（Software Development Kit），直接调用Windows API 函数，向Windows 操作系统提出各种要求，例如配置内存、开启窗口、输出图形...。

> 所谓API（Application Programming Interface），就是开放给应用程序调用的系统功能。

数以千计的Windows APIs，每个看起来都好象比重相若（至少你从手册上看不出来孰轻孰重）。有些APIs 彼此虽有群组关系，却没有相近或组织化的函数名称。星罗棋布，雾列星驰；又似雪球一般愈滚愈多，愈滚愈大。撰写Windows 应用程序需要大量的耐力与毅力，以及大量的小心谨慎！

MFC 帮助我们把这些浩繁的APIs，利用对象导向的原理，逻辑地组织起来，使它们具备抽象化、封装化、继承性、多态性、模块化的性质。

1989 年微软公司成立Application Framework 技术团队，名为AFX 小组，用以开发C++ 对象导向工具给Windows 应用程序开发人员使用。AFX 的"X" 其实没有什么意义，只是为了凑成一个响亮好念的名字。

这个小组最初的「宪章」，根据记载，是要"utilize the latest in object oriented technology to provide tools and libraries for developers writing the most advanced GUI applications on the market"，其中并未画地自限与Windows 操作系统有关。果然，其第一个原型产品，有自己的窗口系统、自己的绘图系统、自己的对象数据库、乃至于自己的内存管理系统。当小组成员以此产品开发应用程序，他们发现实在是太复杂，又悖离公司的主流系统--Windows -- 太遥远。于是他们修改宪章变成"deliver the power of object-oriented solutions to programmers to enable them to build world-class Windows based applications in C++." 这差不多正是Windows 3.0 异军崛起的时候。

C++ 是一个复杂的语言，AFX 小组预期MFC 的使用者不可能人人皆为C++ 专家，所以他们并没有采用所有的C++ 高阶性质（例如多重继承）。许多「麻烦」但「几乎一成不变」的Windows 程序动作都被隐藏在MFC 类别之中， 例如*WinMain* 、*RegisterClass*、Window Procedure 等等等。

> 虽说这些被隐藏的Windows 程序动作几乎是一成不变的，但它们透露了Windows 程序的原型奥秘，这也是为什么我要在本书之中锲而不舍地挖出它们的原因。

为了让MFC 尽可能地小，尽可能地快，AFX 小组不得不舍弃高度的抽象（导至过多的虚拟函数），而引进他们自己发明的机制，尝试在对象导向领域中解决Windows 消息的处理问题。这也就是本书第 9 章深入探讨的Message Mapping 和Message routing 机制。注意，他们并没有改变C++ 语言本身，也没有扩大语言的功能。他们只是设计了一些令人拍案叫绝的宏，而这些宏背后隐藏着巨大的机制。

了解这些宏（以及它们背后所代表的机制）的意义，以及隐藏在MFC 类别之中的那些足以曝露原型机密的「麻烦事儿」，正是我认为掌握MFC 这套Application Framework 的重要手段。

就如同前面那些形而上的定义，MFC 是一组凝聚性强、组织性强的类别库。如果你要利用MFC 发展你的应用程序，必须同时引用数个必要类别，互相搭配奥援。图5-3 是一个标准的MFC 程序外貌。隐藏在精致画面背后更重要的是，就如我在前面说过，对象与对象之间的关系已经存在，消息的流动程序也都已设定。当你要为这个程序设计真正的应用功能，不必在意诸如「我如何得知使用者按左键？左键按下后我如何激活某一个函数？参数如何传递过去...」等琐事，只要专注在左键之后真正要做的功能动作就好。

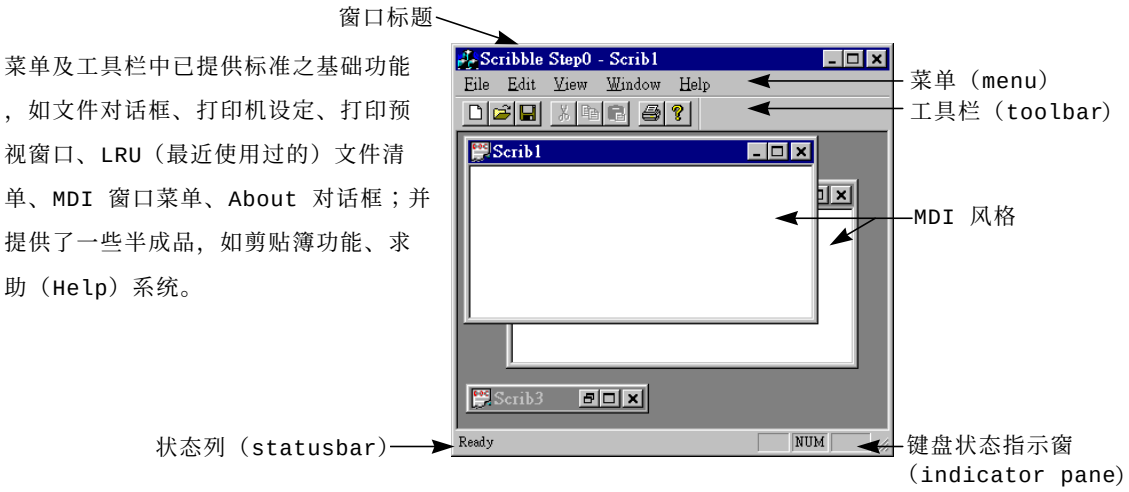


图5-3 标准MFC 程序的风貌。

白头宫女话天宝：Visual C++ 与 MFC

微软公司于1992/04 推出C/C++ 7.0 产品时初次向世人介绍了MFC 1.0，这个初试啼声的产品包含了20,000 行C++ 源代码，60 个以上的Windows 相关类别，以及其它的一般类别如时间、数据处理、文件、内存、诊断、字符串等等等。它所提供的，其实是一个"thin and efficient C++ transformation of the Windows API"。其32 位版亦在1992/07 随着Win32 SDK 推出。

MFC 1.0 获得的回响带给AFX 小组不少鼓舞。他们的下一个目标放在：

- 更高阶的架构支持
- 罐装组件（尤其在使用者接口上）

前者成就了Document/View 架构，后者成就了工具栏、状态列、打印、预览等极受欢迎的UI 性质。当然，他们并没有忘记兼容性与移植性。虽然AFX 小组并未承诺MFC 可以跨不同操作系统如UNIX XWindow、OS/2 PM、Mac System 7，但在其本家（Windows 产品线）身上，在16 位Windows 3.x 和32 位Windows 95 与Windows NT 之间的移植性是无庸置疑的。虽然其16 位产品和32 位产品是分别包装销售，你的原始码通常只需重新编译联结即可。

Visual C++ 1.0（也就是C/C++ 8.0）搭配MFC 2.0 于1993/03 推出，这是针对Windows 3.x 的16 位产品。接下来又在1993/08 推出在Windows NT 上的Visual C++ 1.1 for Windows NT，搭配的是MFC 2.1。这两个版本有着相同的基本性质。MFC 2.0 内含近60,000 行C++ 程序代码，分散在100 个以上的类别中。Visual C++ 整合环境的数个重要工具（大家熟知的Wizards）本身即以MFC 2.0 设计完成，它们的出现对于软件生产效率的提升有极大贡献。

微软在1993/12 又推出了16 位的Visual C++ 1.5，搭配MFC 2.5。这个版本最大的进步是多了OLE2 和ODBC 两组类别。整合环境也为了支持这两组类别而做了些微改变。

1994/09，微软推出Visual C++ 2.0，搭配MFC 3.0，这个32 位版本主要的特征在于配合目标操作系统（Windows NT 和Windows 95），支持多执行线程。所有类别都是thread-safe。UI 对象方面，加入了属性表（Property Sheet）、miniframe 窗口、可随处停驻的工具栏。MFC collections 类别改良为template-based。联结器有重大突破，原使用的Segmented Executable Linker 改为Incremental Linker，这种联结器在对OBJ 档做联结时，并不每次从头到尾重新来过，而只是把新资料往后加，旧资料加记作废。想当然耳，EXE 档会累积许多不用的垃圾，那没关系, 透过Win32 memory-mapped file, 操作系统（Windows NT 及Windows 95）只把欲使用的部份加载，丝毫不影响执行速度。必要时程序员也可选用传统方式联结，这些垃圾自然就不见了。对我们这些终日受制于editbuild-run-debug 轮回的程序员, Incremental Linker 可真是个好礼物。

1995/01，微软又加上了MAPI（Messaging API）和WinSock 支持，推出MFC 3.1（32 位元版），并供应13 个通用控制组件，也就是Windows 95 所提供的tree、tooltip、spin、slider、progress、RTF edit 等等控制组件。

1995/07，MFC 有了3.2 版，那是不值一提的小改版。

然后就是1995/09 的32 位MFC 4.0。这个版本纳入了DAO 数据库类别、多执行线程同步控制类别，并允许制作OCX containers。搭配推出的Visual C++ 4.0 编译器，也终于支持了template、RTTI 等C++ 语言特性。IDE 整合环境有重大的改头换面行动，Class View、Resource View、File View 都使得项目的管理更直觉更轻松，Wizardbar 则活脱脱是一个简化的ClassWizard。此外，多了一个极好用的Components Gallery，并允许程序员订制AppWizard。

1996 年上半年又推出了MFC 4.1，最大的焦点在ISAPI（Internet Server API）的支持，提供五个新类别， 分别是*CHttpServer* 、*CHttpFilter* 、*CHttpServerContext* 、*CHttpFilterContext*、*CHtmlStream*，用以建立交互式Web 应用程序。整合环境方面也对应地提供了一个ISAPI Extension Wizard。在附加价值上，Visual C++ 4.1 提供了Game SDK，帮助开发Windows 95 上的高效率游戏软件。Visual C++ 4.1 还提供不少个由协力

公司完成的OLE 控制组件（OCXs），这些OLE 控制组件技术很快就要全面由桌上跃到网上，称为ActiveX 控制组件。不过，遗憾的是，Visual C++ 4.1 的编译器有些臭虫，不能够制作VxD（虚拟装置驱动程序）。

1996 年下半年推出的MFC 4.2，提供对ActiveX 更多的技术支持，并整合Standard C++ Library。它封包一组新的Win32 Internet 类别（统称为WinInet），使Internet 上的程式开发更容易。它提供22 个新类别和40 个以上的新成员函数。它也提供一些控制元件，可以绑定（binding）近端和远程的资料源（data sources）。整合环境方面，Visual C++ 4.2 提供新的Wizard 给ActiveX 程序开发使用，改善了影像编辑器，使它能够处理在Web 服务器上的两个标准图档格式：GIF 和JPEG。

1997 年五月推出的Visual C++ 5.0，主要诉求在编译器的速度改善，并将Visual C++ 合并到微软整个Visual Tools 的终极管理软件Visual Studio 97 之中。所有的微软虚拟开发工具，包括Visual C++、Visual Basic、Visual J++、Visual InterDev、Visual FoxPro、都在Visual Studio 97 的整合之下有更密切的彼此奥援。至于程序设计方面，MFC 本身没有什么变化（4.21 版），但附了一个ATL（Active Template Library）2.1 版，使ActiveX 控制组件的开发更轻松些。

我想你会发现，微软正不断地为「为什么要使用MFC」加上各式各样的强烈理由，并强烈导引它成为Windows 程序设计的C++ 标准接口。你会看到愈来愈多的MFC/C++ 程式码。对于绝大多数的技术人员而言，Application Framework 的抉择之道无它，「MFC 是微软公司钦定产品」，这个理由就很呛人了。

纵览 MFC

MFC 非常巨大（其它application framework 也不差），在下一章正式使用它之前，让我们先做个浏览。

> 请同时参考书后所附之MFC 架构图

MFC 类别主要可分为下列数大群组：

- General Purpose classes - 提供字符串类别、数据处理类别（如数组与串行），异常情况处理类别、文件类别...等等。
- Windows API classes - 用来封包Windows API，例如窗口类别、对话框类别、DC 类别...等等。
- Application framework classes - 组成应用程序骨干者，　即此组类别，　包括Document/View、消息邦浦、消息映射、消息绕行、动态生成、文件读写等等。
- high level abstractions - 包括工具栏、状态列、分裂窗口、卷动窗口等等。
- operation system extensions - 包括OLE、ODBC、DAO、MAPI、WinSock、ISAPI等等。

General Purpose classes

也许你使用MFC 的第一个目标是为了写Windows 程序，但并不是整个MFC 都只为此目的而活。下面这些类别适用于Windows，也适用于DOS。

CObject

绝大部份类别库，往往以一个或两个类别，做为其它绝大部份类别的基础。MFC 亦复如此。*CObject* 是万类之首，凡类别衍生自*CObject* 者，得以继承数个对象导向重要性质，包括RTTI（执行时期型别鉴识）、Persistence（对象保存）、Dynamic Creation（动态生成）、Diagnostic（错误诊断）。本书第 3 章对于这些技术已有了一份DOS 环境下的模拟，第 8 章另有MFC 相关源代码的探讨。其中，「对象保存」又牵扯到*CArchive*，「诊断」又牵扯到*CDumpContext*，「执行时期型别鉴识」以及「动态生成」又牵扯到*CRuntimeClass*。

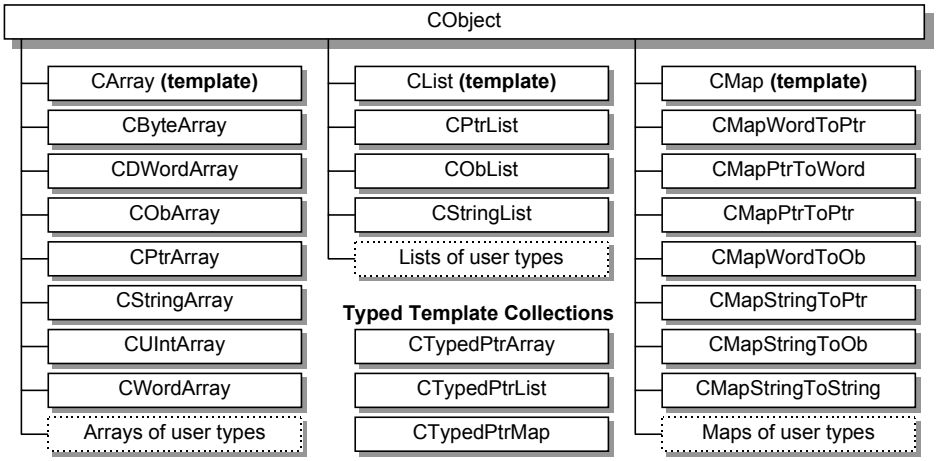
数据处理类别（collection classes）

所谓collection，意指用来管理一「群」对象或标准类型的资料。这些类别像是Array 或

List 或Map 等等，都内含针对元素的「加入」或「删除」或「巡访」等成员函数。Array（数组）和List（串行）是数据结构这门课程的重头戏，大家比较熟知，Map（可视之为表格）则是由成双成对的两两对象所构成，使你很容易由某一对象得知成对的另一物件；换句话说一个对象是另一个对象的键值（key）。例如，你可以使用String-to-String Map，管理一个「电话-人名」数据库；或者使用Word-to-Ptr Map，以16 位数值做为一个指针的键值。

最令人侧目的是，由于这些类别都支持Serialization，一整个数组或串行或表格可以单一一进程序代码就写到文件中（或从文件读出）。第 8 章的Scribble Step1 范例程序中你就会看到它的便利。

MFC 支持的collection classes 有：



杂项类别

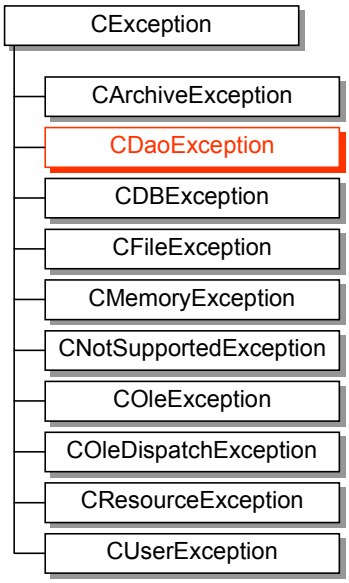
- CRect - 封装Windows 的*RECT* 结构。这个类别在Windows 环境中特别有用，因为*CRect* 常常被用作MFC 类别成员函数的参数。
- *CSize* - 封装Windows 的*SIZE* 结构。
- *CPoint* - 封装Windows 的*POINT* 结构。这个类别在Windows 环境中特别有用，

因为*CPoint* 常常被用作MFC 类别成员函数的参数。

- *CTime* - 表现绝对时间，提供许多成员函数，包括取得目前时间（*static GetCurrentTime*）、将时间资料格式化、抽取特定字段（时、分、秒）等等。它对于+、-、+=、-+ 等运算子都做了多载动作。
- *CTimeSpan* - 以秒数表现时间，通常用于计时码表。提供许多成员函数，包括把秒数转换为日、时、分、秒等等。
- *CString* - 用来处理字符串。支持标准的运算子如=、+=、< 和>。

异常处理类别（exception handling classes）

所谓异常情况（exception），是发生在你的程序执行时期的不正常情况，像是文件打不开、内存不足、写入失败等等等。我曾经在第 2 章最后面介绍过异常处理的观念及相关的MFC 类别，并在第 4 章「Exception Handling」一节介绍过一个简单的例子。与「异常处理」有关的MFC 类别一共有以下11 种：



Windows API classes

这是MFC 声名最著的一群类别。如果你去看看源代码，就会看到这些类别的成员函数所对应的各个Windows API 函数。

- *CWinThread* - 代表MFC 程序中的一个执行线程。自从3.0 版之后，所有的MFC 类别就都已经是thread-safe 了。SDK 程序中标准的消息循环已经被封装在此一类别之中（你会在第 6 章看到我如何把这一部份开膛剖肚）。
- *CWinApp* - 代表你的整个MFC 应用程序。此类别衍生自*CWinThread*；要知道，任何32 位Windows 程序至少由一个执行线程构成。*CWinApp* 内含有用的成员变量如*m_szExeName*，放置执行档档名，以及有用的成员函数如*ProcessShellCommand*，处理命令列选项。
- *CWnd* - 所有窗口，不论是主框窗口、子框窗口、对话框、控制组件、view 视窗，都有一个对应的C++ 类别，你可以想象「窗口handle」和「C++ 对象」结盟。这些C++ 类别统统衍生自*CWnd*，也就是说，凡衍生自*CWnd* 之类别才能收到*WM_* 窗口消息（*WM_COMMAND* 除外）。
  所谓「窗口handle」和「C++ 对象」结盟，实际上是*CWnd* 对象有一个成员变数*m_hWnd*，就放着对应的窗口handle。所以，只要你手上有一个*CWnd* 对象或*CWnd* 对象指针，就可以轻易获得其窗口handle：

  ```
  HWND hWnd = pWnd->m_hWnd;
  ```

- *CCmdTarget* - *CWnd* 的父类别。衍生自它，类别才能够处理命令消息*WM_COMMAND*。这个类别是消息映射以及命令消息绕行的大部份关键，我将在第 9 章推敲这两大神秘技术。
- GDI 类别、DC 类别、Menu 类别。

Application framework classes

这一部份最为人认知的便是Document/View，这也是使MFC 跻身application framework 的关键。Document/View 的观念是希望把资料的本体，和资料的显像分开处理。由于文件产生之际，必须动态生成Document/View/Frame 三种对象，所以又必须有所谓的Document Template 管理之。

- *CDocTemplate*、*CSingleDocTemplate*、*CMultiDocTemplate* - Document Template 扮演黏胶的角色，把Document 和View 和其Frame（外框窗口）胶黏在一块儿。
- *CSingleDocTemplate* 一次只支持一种文件类型，*CMultiDocTemplate* 可同时支持多种文件类型。注意，这和MDI 程序或SDI 程序无关，换句话说，MDI 程序也可以使用*CSingleDocTemplate*，SDI 程序也可以使用*CMultiDocTemplate*。

> 但是，逐渐地，MDI 这个字眼与它原来的意义有了一些出入（要知道，这个字眼早在SDK 时代即有了）。因此，你可能会看到有些书籍这么说：MDI 程序使用*CMultiDocTemplate*，SDI 程序使用*CSingleDocTemplate*。

- *CDocument* - 当你为自己的程序由*CDocument* 衍生出一个子类别后，应该在其中加上成员变量，以容纳文件资料；并加上成员函数，负责修改文件内容以及读写档。读写文件由虚拟函数*Serialize* 负责。第 8 章的Scribble Step1 范例程序有极佳的示范。
- *CView* - 此类别负责将文件内容呈现到显示装置上：也许是屏幕，也许是打印机。文件内容的呈现由虚拟函数*OnDraw* 负责。由于这个类别实际上就是你在屏幕上所看到的窗口（外再罩一个外框窗口），所以它也负责使用者输入的第一线服务。例如第 8 章的Scribble Step1 范例，其View 类别便处理了鼠标的按键动作。

High level abstractions

视觉性UI 对象属于此类，例如工具栏*CToolBar*、状态列*CStatusBar*、对话框列*CDialogBar*。加强型的View 也属此类，如可卷动的*ScrollView*、以对话框为基础的

CFormView、小型文字编辑器*CEditView*、树状结构的*CTreeView*，支持RTF 文件格式的*CRichEditView* 等等。

Afx 全域函数

还记得吧，C++ 并不是纯种的对象导向语言（SmallTalk 和Java 才是）。所以，MFC 之中得以存在有不属于任何类别的全域函数，它们统统在函数名称开头冠以*Afx*。

下面是几个常见的Afx 全域函数：

函数名称	说明
AfxWinInit	被*WinMain*（由MFC 提供）调用的一个函数，用做MFC GUI 程序初始化的一部份，请看第 6 章的「*AfxWinInit* - AFX 内部初始化动作」一节。如果你写一个MFC console 程序，就得自行调用此函数（请参考Visual C++ 所附之Tear 范例程序）。
AfxBeginThread	开始一个新的执行线程（请看第14 章，# 756 页）。
AfxEndThread	结束一个旧的执行线程（请看第14 章，# 756 页）。
AfxFormatString1	类似*printf* 一般地将字符串格式化。
AfxFormatString2	类似*printf* 一般地将字符串格式化。
AfxMessageBox	类似Windows API 函数*MessageBox*。
AfxOutputDebugString	将字符串输往除错装置（请参考附录D，# 924 页）。
AfxGetApp	取得application object（*CWinApp* 衍生对象）的指针。
AfxGetMainWnd	取得程序主窗口的指针。
AfxGetInstance	取得程序的instance handle。
AfxRegisterClass	以自定的*WNDCLASS* 注册窗口类别（如果MFC 提供的数个窗口类别不能满足你的话）。

MFC 宏（macros）

CObject 和*CRuntimeClass* 之中封装了数个所谓的object services，包括「取得执行时期的类别信息」（RTTI）、Serialization（文件读写）、动态产生对象...等等。所有衍生自*CObject*

的类别，都继承这些机能。我想你对这些名词及其代表的意义已经不再陌生-- 如果你没有错过第 3 章的「MFC 六大技术仿真」的话。

- 取得执行时期的类别信息（RTTI），使你能够决定一个执行时期的对象的类别信息，这样的能力在你需要对函数参数做一些额外的类型检验，或是当你要针对对象属于某种类别而做特别的动作时，份外有用。
- Serialization 是指将对象内容写到文件中，或从文件中读出。如此一来对象的生命就可以在程序结束之后还延续下去，而在程序重新激活之后，再被读入。这样的对象可说是"persistent"（永续存在）。
- 所谓动态的对象生成（Dynamic object creation），使你得以在执行时期产生一个特定的对象。例如document、view、和frame 对象就都必须支持动态对象生成，因为framework 需要在执行时期产生它们（第 8 章有更详细的说明）。

此外，OLE 常常需要在执行时期做对象的动态生成动作。例如一个OLE server 程序必须能够动态产生OLE items，用以反应OLE client 的需求。

MFC 针对上述这些机能，准备了一些宏，让程序能够很方便地继承并实作出上述四大机能。这些宏包括：

宏名称	提供机能	出现章节
DECLARE_DYNAMIC	执行时期类别信息	第 3 章、第 8 章
IMPLEMENT_DYNAMIC	执行时期类别信息	第 3 章、第 8 章
DECLARE_DYNCREATE	动态生成	第 3 章、第 8 章
IMPLEMENT_DYNCREATE	动态生成	第 3 章、第 8 章
DECLARE_SERIAL	对象内容的文件读写	第 3 章、第 8 章
IMPLEMENT_SERIAL	对象内容的文件读写	第 3 章、第 8 章
DECLARE_OLECREATE OLE	对象的动态生成	不在本书范围之内
IMPLEMENT_OLECREATE OLE	对象的动态生成	不在本书范围之内

我也已经在第３章提过MFC 的消息映射（Message Mapping）与命令绕行（Command Routing）两个特性。这两个性质系由以下这些MFC 宏完成：

宏名称	提供机能	出现章节
DECLARE_MESSAGE_MAP	声明消息映射表数据结构	第３章、第９章
BEGIN_MESSAGE_MAP	开始消息映射表的建置	第３章、第９章
ON_COMMAND	增加消息映射表中的项目	第３章、第９章
ON_CONTROL	增加消息映射表中的项目	本书未举例
ON_MESSAGE	增加消息映射表中的项目	???
ON_OLECMD	增加消息映射表中的项目	本书未举例
ON_REGISTERED_MESSAGE	增加消息映射表中的项目	本书未举例
ON_REGISTERED_THREAD_MESSAGE	增加消息映射表中的项目	本书未举例
ON_THREAD_MESSAGE	增加消息映射表中的项目	本书未举例
ON_UPDATE_COMMAND_UI	增加消息映射表中的项目	第３章、第９章
END_MESSAGE_MAP	结束消息映射表的建置	第３章、第９章

事实上，与其它MFC Programming 书籍相比较，本书最大的一个特色就是，要把上述这些MFC 宏的来龙去脉交待得非常清楚。我认为这对于撰写MFC 程序是非常重要的一件事。

MFC 数据类型（data types）

下面所列的这些数据类型，常常出现在MFC 之中。其中的绝大部份都和一般的Win32 程序（SDK 程序）所用的相同。

下面这些是和Win32 程序（SDK 程序）共同使用的数据类型：

数据类型	意义
BOOL	Boolean 值（布尔值，不是TRUE 就是FALSE）
BSTR	32-bit 字符指针
BYTE	8-bit 整数，未带正负号
COLORREF	32-bit 数值，代表一个颜色值
DWORD	32-bit 整数，未带正负号
LONG	32-bit 整数，带正负号
LPARAM	32-bit 数值，做为窗口函数或callback 函数的一个参数
LPCSTR	32-bit 指针，指向一个常数字符串
LPSTR	32-bit 指针，指向一个字符串
LPCTSTR	32-bit 指针，指向一个常数字符串。此字符串可移植到Unicode 和DBCS（双字节字集）
LPTSTR	32-bit 指针，指向一个字符串。此字符串可移植到Unicode 和DBCS（双位组字集）
LPVOID	32-bit 指针，指向一个未指定类型的资料
LPRESULT	32-bit 数值，做为窗口函数或callback 函数的回返值
UINT	在Win16 中是一个16-bit 未带正负号整数，在Win32 中是一个32-bit 未带正负号整数。
WNDPROC	32-bit 指针，指向一个窗口函数
WORD	16-bit 整数，未带正负号
WPARAM	窗口函数的callback 函数的一个参数。在Win16 中是16 bits，在Win32 中是32 bits。

下面这些是MFC 独特的数据类型：

数据类型	意义
POSITION	一个数值，代表collection 对象（例如数组或串行）中的元素位置。常使用于MFC collection classes。
LPCRECT	32-bit 指针，指向一个不变的RECT 结构。

前面所说那些MFC 数据类型与C++ 语言数据类型之间的对应，定义于WINDEF.H 中。我列出其中一部份，并且将不符合(_MSC_VER >= 800) 条件式的部份略去。

```
#define NULL    0

#define far             //侯俊杰注：Win32 不再有far 或near memory model,
#define near            //而是使用所谓的flat model。pascall 函数调用习惯
#define pascal __stdcall  //也被stdcall 函数调用习惯取而代之。

#define cdecl _cdecl
#define CDECL _cdecl

#define CALLBACK    __stdcall  // 侯俊杰注：在Windows programming演化过程中
#define WINAPI      __stdcall  // 曾经出现的PASCAL、CALLBACK、WINAPI
#define WINAPIV     __cdecl    //APIENTRY，现在都代表相同的意义，就是stdcall
#define APIENTRY    WINAPI     // 函数调用习惯。
#define APIPRIVATE  __stdcall
#define PASCAL      __stdcall

#define FAR         far
#define NEAR        near
#define CONST       const

typedef unsigned long         DWORD;
typedef int                   BOOL;
typedef unsigned char         BYTE;
typedef unsigned short        WORD;
typedef float                 FLOAT;
typedef FLOAT                 *PFLOAT;
typedef BOOL near             *PBOOL;
typedef BOOL far              *LPBOOL;
typedef BYTE near             *PBYTE;
typedef BYTE far              *LPBYTE;
```

```
typedef int near         *PINT;
typedef int far          *LPINT;
typedef WORD near        *PWORD;
typedef WORD far         *LPWORD;
typedef long far         *LPLONG;
typedef DWORD near        *PDWORD;
typedef DWORD far        *LPDWORD;
typedef void far         *LPVOID;
typedef CONST void far   *LPCVOID;

typedef int              INT;
typedef unsigned int     UINT;
typedef unsigned int     *PUINT;

/* Types use for passing & returning polymorphic values */
typedef UINT WPARAM;
typedef LONG LPARAM;
typedef LONG LRESULT;

typedef DWORD   COLORREF;
typedef DWORD   *LPCOLORREF;

typedef struct tagRECT
{
   LONG   left;
   LONG   top;
   LONG   right;
   LONG   bottom;
} RECT, *PRECT, NEAR *NPRECT, FAR *LPRECT;

typedef const RECT FAR* LPCRECT;

typedef struct tagPOINT
{
   LONG  x;
   LONG  y;
} POINT, *PPOINT, NEAR *NPPOINT, FAR *LPPOINT;

typedef struct tagSIZE
{
   LONG       cx;
   LONG       cy;
} SIZE, *PSIZE, *LPSIZE;
```

第 6 章 MFC 程序设计导论

MFC 程序的生死因果

理想如果不向实际做点妥协，理想就会归于尘土。
中华民国还得十次革命才得建立，对象导向怎能把一切传统都抛开。

以传统的C/SDK 撰写Windows 程序，最大的好处是可以清楚看见整个程序的来龙去脉和消息动向，然而这些重要的动线在MFC 应用程序中却隐晦不明，因为它们被Application Framework 包起来了。这一章主要目的除了解释MFC 应用程序的长像，也要从MFC 源代码中检验出一个Windows 程序原本该有的程序进入点（*WinMain*）、视窗类别注册（*RegisterClass*）、窗口产生（*CreateWindow*）、消息循环（Message Loop）、窗口函数（Window Procedure）等等动作，抽丝剥茧彻底了解一个MFC 程序的诞生与结束，以及生命过程。

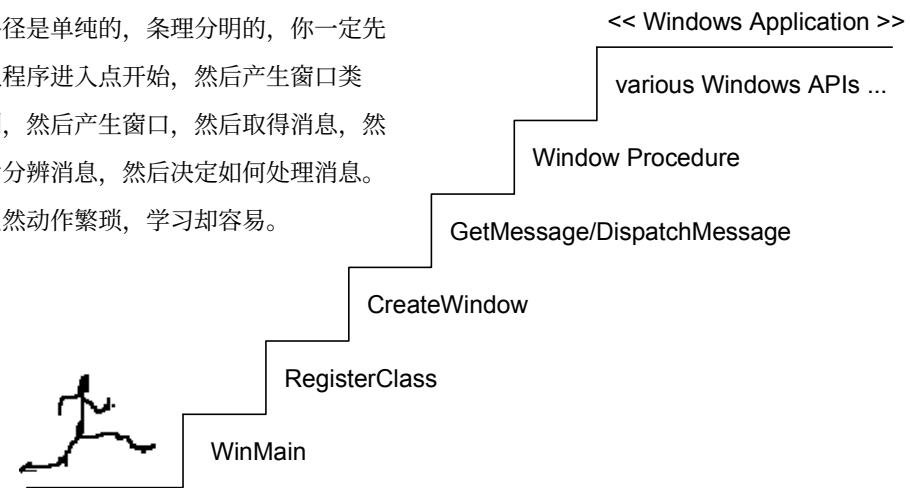
为什么要安排这一章？了解MFC 内部构造是必要的吗？看电视需要知道映射管的原理吗？开汽车需要知道传动轴与变速箱的原理吗？学习MFC 不就是要一举超越烦琐的Windows API？啊，厂商（不管是哪一家）广告给我们的印象就是，藉由可视化的工具我们可以一步登天，基本上这个论点正确，只是有个但书：你得学会操控Application Framework。

想象你拥有一部保时捷，风驰电挚风光得很，但是引擎盖打开来全傻了眼。如果你懂汽车内部运作原理，那么至少开车时「脚不要老是含着离合器，以免来令片磨损」这个道理背后的原理你就懂了，「踩煞车时绝不可以同时踩离合器，以免失去引擎煞车力」这个道理背后的原理你也懂了，甚至你的保时捷要保养维修时或也可以不假外力自己来。

不要把自己想象成这场游戏中的后座车主，事实上作为这本技术书籍的读者的你，应该是车厂师傅。

好，这个比喻不见得面面俱到，但起码你知道了自己的身份。
题外话：我的朋友曾铭源（现在纽约工作）写信给我说：『最近项目的压力大，人员纷纷离职。接连一个多礼拜，天天有人上门面谈。人事部门不知从哪里找来这些阿哥，号称有三年的SDK/MFC 经验，结果对起话来是鸡同鸭讲，*WinMain* 和Windows Procedure 都搞不清楚。问他什么是message handler？只会在ClassWizard 上click、click、click !!! 拜 Wizard 之赐，人力市场上多出了好几倍的VC/MFC 程序员，但这些「Wizard 通」我们可不敢要』。

以raw Windows API 开发程序，学习的路径是单纯的，条理分明的，你一定先从程序进入点开始，然后产生窗口类别，然后产生窗口，然后取得消息，然后分辨消息，然后决定如何处理消息。虽然动作繁琐，学习却容易。

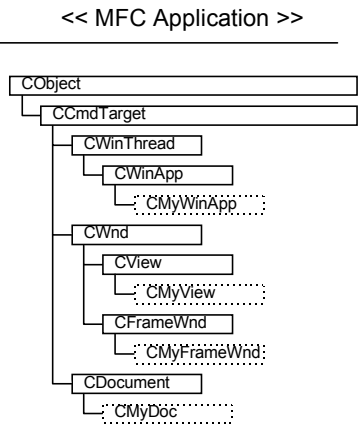


以MFC开发程序，一开始很快速，因为开发工具会为你产生一个骨干程序，一般该有的各种接口一应俱全。但是MFC的学习曲线十分陡峭，程序员从骨干程式出发一直到有能力修改程序代码以符合个人的需要，是一段不易攀登的峭壁。

一个 MFC 骨干程序

Visual C++ 各种工具之使用

近乎垂直的学习曲线



如果我们了解 Windows 程序的基本运作原理，并了解 MFC 如何把这些基础动作整合起来，我们就能够使 MFC 学习曲线的陡峭程度缓和下来。因此能够迅速接受MFC，进而使用MFC。呵，一条似远实近的道路！

MFC 骨干程序

Visual C++ 各种工具之使用

WinMain 由 MFC 提供

RegisterClass 包装于 AfxWinInit 中

CreateWindow 应该在 CWinApp::InitInstance 中调用

GetMessage/DispatchMessage
包装在 CWinApp::Run 中

Window Procedure 由 MFC 提供

Windows APIs
包装在MFC 各类别中

Message Mapping,
Message Routing.

Dynamic Creation,
Serialization.

<< MFC Application >>

一个

我希望你了解，本书之所以在各个主题中不厌其烦地挖MFC 内部动作，解释骨干程序的每一条指令，每一个环节，是为了让你踏实地接受MFC，进而有能力役使MFC。你以为这是一条远路？呵呵，似远实近！

不二法门：熟记MFC 类别的阶层架构

MFC 在1.0 版时期的诉求是「一组将SDK API 包装得更好用的类别库」，从2.0 版开始更进一步诉求是一个「Application Framework」，拥有重要的Document-View 架构；随后又在更新版本上增加了OLE 架构、DAO 架构...。为了让你有一个最轻松的起点，我把第一个程序简化到最小程度，舍弃Document-View 架构，使你能够尽快掌握C++/MFC 程序的面貌。这个程序并不以AppWizard 制作出来，也不以ClassWizard 管理维护，而是纯手工打造。毕竟Wizards 做出来的程序代码有一大堆批注，某些批注对Wizards 有特殊意义，不能随便删除，却可能会混淆初学者的视听焦点；而且Wizards 所产生的程序骨干已具备Document-View 架构，又有许多奇奇怪怪的宏，初学者暂避为妙。我们目前最想知道的是一个最阳春的MFC 程序以什么面貌呈现，以及它如何开始运作，如何结束生命。

SDK 程序设计的第一要务是了解最重要的数个API 函数的意义和用法， 像是*RegisterClass*、*CreateWindow*、*GetMessage*、*DispatchMessage*，以及消息的获得与分配。MFC 程序设计的第一要务则是熟记MFC 的类别阶层架构，并清楚知晓其中几个一定会用到的类别。本书最后面有一张MFC 4.2 架构图，叠床架屋，令人畏惧，我将挑出单单两个类别，组合成一个"Hello MFC" 程序。这两个类别在MFC 的地位如图6-1 所示。

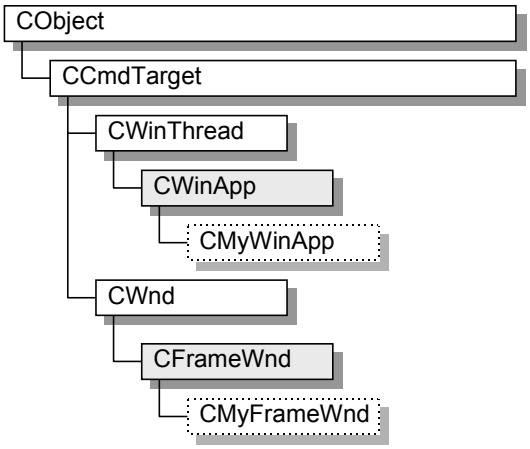


图6-1 本章范例程序所用到的MFC 类别

需要什么函数库？

开始写码之前，我们得先了解程序代码以外的外围环境。第一个必须知道的是，MFC 程序需要什么函数库？SDK 程序联结时期所需的函数库已在第一章显示，MFC 程序一样需要它们：

■ Windows C Runtime函数库（VC++ 5.0）

文件名称	文件大小	说明
LIBC.LIB	898826	C Runtime 函数库的静态联结版本
MSVCRT.LIB	510000	C Runtime 函数库的动态联结版本
MSVCRTD.LIB	803418	'D' 表示使用于Debug 模式

＊ 这些函数库不再区分Large/Medium/Small 内存模式，因为32 位操作系统不再有记忆体模式之分。这些函数库的多线程版本，请参考本书#38 页。

■ DLL Import 函数库（VC++ 5.0）

文件名称	文件大小	说明
GDI32.LIB	307520	for GDI32.DLL（136704 bytes in Win95）
USER32.LIB	517018	for USER32.DLL（45568 bytes in Win95）
KERNEL32.LIB	635638	for KERNEL32 DLL（413696 bytes in Win95）
. . .		

此外，应用程序还需要联结一个所谓的MFC 函数库，或称为AFX 函数库，它也就是MFC 这个application framework 的本体。你可以静态联结之，也可以动态联结之，AppWizard 给你选择权。本例使用动态联结方式，所以需要一个对应的MFC import 函数库：

■ MFC 函数库（AFX 函数库)（VC++ 5.0，MFC 4.2）

文件名称	文件大小	说明
MFC42.LIB	4200034	MFC42.DLL（941840 bytes）的 import 函数库。
MFC42D.LIB	3003766	MFC42D.DLL（1393152 bytes）的 import 函数库。
MFCS42.LIB	168364	
MFCS42D.LIB	169284	
MFCN42D.LIB	91134	
MFCD42D.LIB	486334	
MFCO42D.LIB	2173082	
. . .		

我们如何在联结器（link.exe） 中设定选项，把这些函数库都联结起来？ 稍后在 HELLO.MAK 中可以一窥全貌。

如果在Visual C++ 整合环境中工作，这些设定不劳你自己动手，整合环境会根据我们圈选的项目自动做出一个合适的makefile。这些makefile 的内容看起来非常诘屈聱牙，事实上我们也不必太在意它，因为那是整合环境的工作。这一章我不打算依赖任何开发工具，一切自己来，你会在稍后看到一个简洁清爽的makefile。

需要什么头文件？

SDK 程序只要包含WINDOWS.H 就好，所有API 的函数声明、消息定义、常数定义、宏定义、都在WINDOWS.H 档中。除非程序另调用了操作系统提供的新模块（如CommDlg、ToolHelp、DDEML...），才需要再各别包含对应的.H 档。

> WINDOWS.H 过去是一个巨大文件，大约在5000 行上下。现在已拆分内容为数十个较小的.H 档，再由WINDOWS.H 包含进来。也就是说它变成一个"Master included file for Windows applications"。

MFC 程序不这么单纯，下面是它常常需要面对的另外一些.H 档：

- STDAFX.H - 这个文件用来做为Precompiled header file（请看稍后的方块说明）, 其内只是包含其它的MFC 头文件。应用程序通常会准备自己的
- STDAFX.H，例如本章的Hello 程序就在STDAFX.H 中包含AFXWIN.H。
- AFXWIN.H - 每一个Windows MFC 程序都必须包含它，因为它以及它所包含的文件声明了所有的MFC 类别。此档内含AFX.H，后者又包含AFXVER_.H，后者又包含AFXV_W32.H，后者又包含WINDOWS.H（啊呼，终于现身）。
- AFXEXT.H - 凡使用工具栏、状态列之程序必须包含这个文件。
- AFXDLGS.H - 凡使用通用型对话框（Common Dialog）之MFC 程序需包含此档，其内部包含COMMDLG.H。
- AFXCMN.H - 凡使用Windows 95 新增之通用型控制组件（Common Control）之MFC 程序需包含此文件。
- AFXCOLL.H - 凡使用Collections Classes（用以处理数据结构如数组、串行）之程序必须包含此文件。
- AFXDLLX.H - 凡MFC extension DLLs 均需包含此档。
- AFXRES.H - MFC 程序的RC 文件必须包含此档。MFC 对于标准资源（例如File、Edit 等菜单项目）的ID 都有默认值，定义于此文件中，例如：

```
// File commands
#define ID_FILE_NEW        0xE100
#define ID_FILE_OPEN       0xE101
#define ID_FILE_CLOSE      0xE102
#define ID_FILE_SAVE       0xE103
#define ID_FILE_SAVE_AS    0xE104
...
// Edit commands
#define ID_EDIT_COPY       0xE122
#define ID_EDIT_CUT        0xE123
...
```

这些菜单项目都有预设的说明文字（将出现在状态列中），但说明文字并不会事先定义于此文件，AppWizard 为我们制作骨干程序时才把说明文字加到应用程序的RC 文件中。第４章的骨干程序Scribble step0 的RC 档中就有这样的字符串表格：

```
STRINGTABLE DISCARDABLE
BEGIN
  ID_FILE_NEW     "Create a new document"
  ID_FILE_OPEN    "Open an existing document"
  ID_FILE_CLOSE   "Close the active document"
  ID_FILE_SAVE    "Save the active document"
  ID_FILE_SAVE_AS  "Save the active document with a new name"
  ...
  ID_EDIT_COPY "Copy the selection and puts it on the Clipboard"
  ID_EDIT_CUT  "Cut the selection and puts it on the Clipboard"
  ...
END
```

所有MFC 头文件均置于\MSVC\MFC\INCLUDE 中。这些文件连同Windows SDK 的包含档WINDOWS.H、COMMDLG.H、TOOLHELP.H、DDEML.H... 每每在编译过程中耗费大量的时间，因此你绝对有必要设定Precompiled header。

Precompiled Header
一个应用程序在发展过程中常需要不断地编译。Windows 程序包含的标准.H 文件非常巨大但内容不变，编译器浪费在这上面的时间非常多。Precompiled header 就是将.H 档第一次编译后的结果贮存起来，第二次再编译时就可以直接从磁盘中取出来用。这种观念在Borland C/C++ 早已行之，Microsoft 这边则是一直到Visual C++ 1.0 才具备。

简化的MFC 程序架构－以Hello MFC 为例

现在我们正式进入MFC 程序设计。由于Document/View 架构复杂，不适合初学者，所以我先把它略去。这里所提的程序观念是一般的MFC Application Framework 的子集合。本章程序名为Hello，执行时会在窗口中从天而降"Hello, MFC" 字样。Hello 是一个非常简单而具代表性的程序，它的代表性在于：

- 每一个MFC 程序都想从MFC 中衍生出适当的类别来用（ 不然又何必以MFC 写程序呢），其中两个不可或缺的类别*CWinApp* 和*CFrameWnd* 在Hello 程序中会表现出来，它们的意义如图6-2。
- MFC 类别中某些函数一定得被应用程序改写（例如*CWinApp*::*InitInstance*），这在Hello 程序中也看得到。
- 菜单和对话框，Hello 也都具备。

图6-3 是Hello 源文件的组成。第一次接触MFC 程序，我们常常因为不熟悉MFC 的类别分类、类别命名规则，以至于不能在脑中形成具体印象，于是细部讨论时各种信息及说明彷如过眼云烟。相信我，你必须多看几次，并且用心熟记MFC 命名规则。

图6-3 之后是Hello 程序的源代码。由于MFC 已经把Windows API 都包装起来了，源代码再也不能够「说明一切」。你会发现MFC 程序很有点见林不见树的味道：

■ 看不到*WinMain*，因此不知程序从哪里开始执行。

■ 看不到*RegisterClass* 和*CreateWindow*，那么窗口是如何做出来的呢？

■ 看不到Message Loop（*GetMessage*/*DispatchMessage*），那么程序如何推动？

■ 看不到Window Procedure，那么窗口如何运作？

我的目的就在铲除这些困惑。

Hello 程序源代码

■ HELLO.MAK - makefile

■ RESOURCE.H - 所有资源ID 都在这里定义。本例只定义一个IDM_ABOUT。

■ JJHOUR.ICO - 图标文件，用于主窗口和对话框。

■ HELLO.RC - 资源描述档。本例有一份菜单、一个图标、和一个对话框。

■ STDAFX.H - 包含AFXWIN.H。

■ STDAFX.CPP - 包含STDAFX.H，为的是制造出Precompiled header。

■ HELLO.H - 声明*CMyWinApp* 和*CMyFrameWnd*。

■ HELLO.CPP - 定义*CMyWinApp* 和*CMyFrameWnd*。

注意：没有模块定义文件.DEF？是的，如果你不指定模块定义文件，联结器就使用默认值。

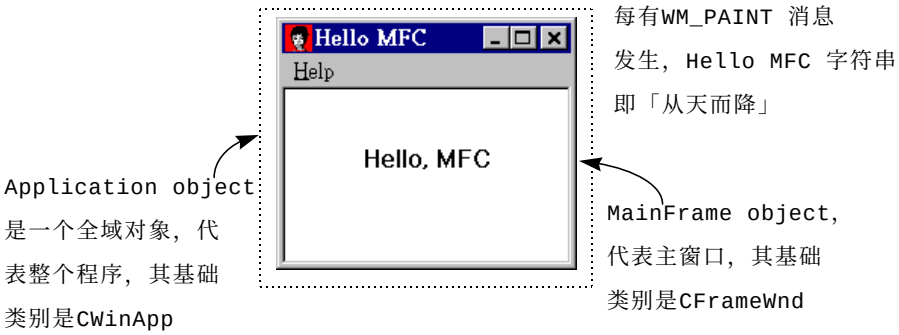


图6-2 Hello 程序中的两个对象

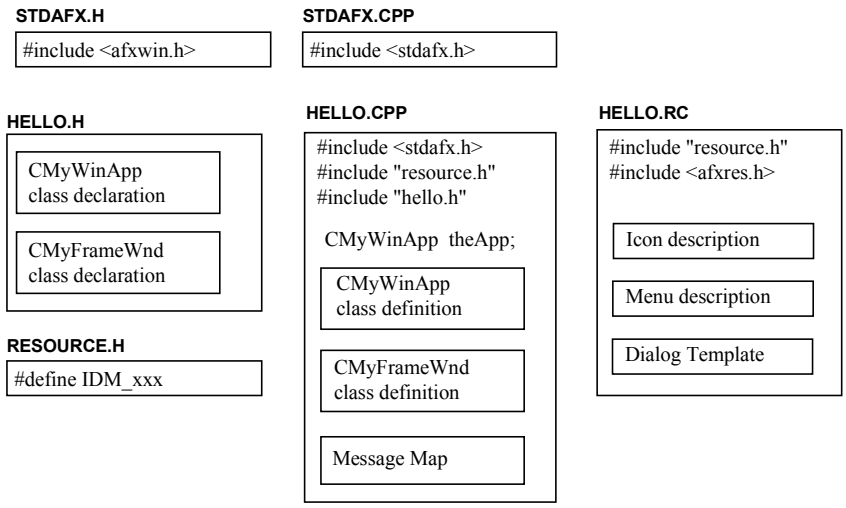


图6-3 Hello 程序的基本文件架构。一般习惯为每个类别准备一个.H（声明）和一个.CPP（实作），本例把两类别集中在一起是为了简化。

HELLO.MAK （请在DOS 窗口中执行nmake hello.mak。环境设定请参考p.224）

```
#0001  # filename : hello.mak
#0002  # make file for hello.exe (MFC 4.0 Application)
#0003  # usage : nmake hello.mak (Visual C++ 5.0)
#0004
#0005  Hello.exe : StdAfx.obj Hello.obj Hello.res
#0006    link.exe /nologo /subsystem:windows /incremental:no \
#0007            /machine:I386 /out:"Hello.exe" \
#0008            Hello.obj StdAfx.obj Hello.res \
#0009            msvcrt.lib kernel32.lib user32.lib gdi32.lib mfc42.lib
#0010
#0011  StdAfx.obj : StdAfx.cpp StdAfx.h
#0012    cl.exe /nologo /MD /W3 /GX /O2 /D "WIN32" /D "NDEBUG" /D "_WINDOWS" \
#0013          /D "_AFXDLL" /D "_MBCS" /Fp"Hello.pch" /Yc"stdafx.h"  \
#0014          /c StdAfx.cpp
#0015
#0016  Hello.obj : Hello.cpp Hello.h StdAfx.h
#0017    cl.exe /nologo /MD /W3 /GX /O2 /D "WIN32" /D "NDEBUG" /D "_WINDOWS" \
#0018          /D "_AFXDLL" /D "_MBCS" /Fp"Hello.pch" /Yu"stdafx.h"  \
#0019          /c Hello.cpp
#0020
#0021  Hello.res : Hello.rc Hello.ico jjhour.ico
#0022    rc.exe /l 0x404 /Fo"Hello.res" /D "NDEBUG" /D "_AFXDLL" Hello.rc
```

RESOURCE.H

```
#0001  // resource.h
#0002  #define IDM_ABOUT   100
```

HELLO.RC

```
#0001  // hello.rc
#0002  #include "resource.h"
#0003  #include "afxres.h"
#0004
#0005  JJHouRIcon        ICON  DISCARDABLE  "JJHOUR.ICO"
#0006  AFX_IDI_STD_FRAME  ICON  DISCARDABLE  "JJHOUR.ICO"
#0007
#0008  MainMenu MENU DISCARDABLE
#0009  {
#0010      POPUP "&Help"
#0011      {
#0012        MENUITEM "&About HelloMFC...", IDM_ABOUT
#0013      }
#0014  }
#0015
#0016  AboutBox DIALOG DISCARDABLE  34, 22, 147, 55
#0017  STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
#0018  CAPTION "About Hello"
#0019  {
#0020    ICON         "JJHouRIcon",IDC_STATIC,11,17,18,20
#0021    LTEXT        "Hello MFC 4.0",IDC_STATIC,40,10,52,8
#0022    LTEXT        "Copyright 1996 Top Studio",IDC_STATIC,40,25,100,8
#0023    LTEXT        "J.J.Hou",IDC_STATIC,40,40,100,8
#0024    DEFPUSHBUTTON "OK",IDOK,105,7,32,14,WS_GROUP
#0025  }
```

STDAFX.H

```
#0001  // stdafx.h : include file for standard system include files,
#0002  // or project specific include files that are used frequently,
#0003  // but are changed infrequently
#0004
#0005  #include <afxwin.h>   // MFC core and standard components
```

STDAFX.CPP

```
#0001  // stdafx.cpp : source file that includes just the standard includes
#0002  //     Hello.pch will be the pre-compiled header
#0003  //     stdafx.obj will contain the pre-compiled type information
#0004
#0005  #include "stdafx.h"
```

HELLO.H

```
#0001  //----------------------------------------------------------------
#0002  //            MFC 4.0  Hello Sample Program
#0003  //       Copyright (c) 1996   Top Studio * J.J.Hou
#0004  // 档名: hello.h
#0005  //
#0006  // 作者: 侯俊杰
#0007  // 编译联结: 请参考hello.mak
#0008  //
#0009  // 声明Hello 程序的两个类别: CMyWinApp 和CMyFrameWnd
#0010  //----------------------------------------------------------------
#0011
#0012  class CMyWinApp : public CWinApp
#0013  {
#0014  public:
#0015     BOOL InitInstance();  // 每一个应用程序都应该改写此函数
#0016  };
#0017
#0018  //----------------------------------------------------------------
#0019  class CMyFrameWnd : public CFrameWnd
#0020  {
#0021  public:
#0022     CMyFrameWnd();          // constructor
#0023     afx_msg void OnPaint();   // for WM_PAINT
#0024     afx_msg void OnAbout();   // for WM_COMMAND (IDM_ABOUT)
#0025
#0026  private:
#0027     DECLARE_MESSAGE_MAP()    // Declare Message Map
#0028     static VOID CALLBACK LineDDACallback(int,int,LPARAM);
#0029     //注意: callback 函数必须是"static", 才能去除隐藏的'this' 指针。
#0030  };
```

HELLO.CPP

```
#0001  //-----------------------------------------------------------------
#0002  //             MFC 4.0  Hello sample program
#0003  //       Copyright (c) 1996   Top Studio * J.J.Hou
#0004  //  档名: hello.cpp
#0005  //
#0006  //  作者: 侯俊杰
#0007  //  编译联结: 请参考hello.mak
#0008  //
#0009  // 本例示范最简单之MFC 应用程序, 不含Document/View 架构。程序每收到
#0010  // WM_PAINT 即利用GDI 函数LineDDA() 让"Hello, MFC" 字符串从天而降。
#0011  //-----------------------------------------------------------------
#0012  #include "Stdafx.h"
#0013  #include "Hello.h"
#0014  #include "Resource.h"
#0015
#0016  CMyWinApp theApp;   // application object
#0017
#0018  //-----------------------------------------------------------------
#0019  // CMyWinApp's member
#0020  //-----------------------------------------------------------------
#0021  BOOL CMyWinApp::InitInstance()
#0022  {
#0023     m_pMainWnd = new CMyFrameWnd();
#0024     m_pMainWnd->ShowWindow(m_nCmdShow);
#0025     m_pMainWnd->UpdateWindow();
#0026     return TRUE;
#0027  }
#0028  //-----------------------------------------------------------------
#0029  // CMyFrameWnd's member
#0030  //-----------------------------------------------------------------
#0031  CMyFrameWnd::CMyFrameWnd()
#0032  {
#0033     Create(NULL, "Hello MFC", WS_OVERLAPPEDWINDOW, rectDefault,
#0034           NULL, "MainMenu");     // "MainMenu" 定义于 RC 档
#0035  }
#0036  //-----------------------------------------------------------------
#0037  BEGIN_MESSAGE_MAP(CMyFrameWnd, CFrameWnd)
#0038     ON_COMMAND(IDM_ABOUT, OnAbout)
#0039     ON_WM_PAINT()
#0040  END_MESSAGE_MAP()
#0041  //-----------------------------------------------------------------
#0042  void CMyFrameWnd::OnPaint()
#0043  {
#0044  CPaintDC dc(this);
```

```
#0045  CRect rect;
#0046
#0047    GetClientRect(rect);
#0048
#0049    dc.SetTextAlign(TA_BOTTOM | TA_CENTER);
#0050
#0051    ::LineDDA(rect.right/2, 0, rect.right/2, rect.bottom/2,
#0052       (LINEDDAPROC) LineDDACallback, (LPARAM) (LPVOID) &dc);
#0053  }
#0054  //---------------------------------------------------------------
#0055  VOID CALLBACK CMyFrameWnd::LineDDACallback(int x, int y, LPARAM lpdc)
#0056  {
#0057  static char szText[] = "Hello, MFC";
#0058
#0059    ((CDC*)lpdc)->TextOut(x, y, szText, sizeof(szText)-1);
#0060    for(int i=1; i<50000; i++);  // 纯粹是为了延迟下降速度，以利观察
#0061  }
#0062  //---------------------------------------------------------------
#0063  void CMyFrameWnd::OnAbout()
#0064  {
#0065    CDialog about("AboutBox", this);  //"AboutBox" 定义于RC 档
#0066    about.DoModal();
#0067  }
```

上面这些程序代码中，你看到了一些MFC 类别如*CWinApp* 和*CFrameWnd*，一些MFC 数据类型如*BOOL* 和*VOID*，一些MFC 宏如*DECLARE_MESSAGE_MAP* 和*BEGIN_MESSAGE_END* 和*END_MESSAGE_MAP*。这些都曾经在第5章的「纵览MFC」一节中露过脸。但是单纯从C++ 语言的角度来看，还有一些是我们不能理解的，如HELLO.H 中的*afx_msg*（#23 行）和*CALLBACK*（#28 行）。

你可以在WINDEF.H 中发现*CALLBACK* 的意义：

```
#define CALLBACK __stdcall // 一种函数调用习惯
```

可以在AFXWIN.H 中发现*afx_msg* 的意义：

```
#define afx_msg // intentional placeholder
                // 故意安排的一个空位置。也许以后版本会用到。
```

MFC 程序的来龙去脉（causal relations）

让我们从第1章的C/SDK 观念出发，看看MFC 程序如何运作。

第一件事情就是找出MFC 程序的进入点。MFC 程序也是Windows 程序，所以它应该也有一个*WinMain*，但是我们在Hello 程序看不到它的踪影。是的，但先别急，在程序进入点之前，更有一个（而且仅有一个）全域对象（本例名为*theApp*），这是所谓的application object，当操作系统将程序加载并激活，这个全域对象获得配置，其构造式会先执行，比*WinMain* 更早。所以以时间顺序来说，我们先看看这个application object。

我只借用两个类别：CWinApp 和 CFrameWnd

你已经看过了图6-2，作为一个最最粗浅的MFC 程序，Hello 是如此单纯，只有一个视窗。回想第一章Generic 程序的写法，其主体在于*WinMain* 和*WndProc*，而这两个部份其实都有相当程度的不变性。好极了，MFC 就把有着相当固定行为之*WinMain* 内部动作包装在*CWinApp* 中，把有着相当固定行为之*WndProc* 内部动作包装在*CFrameWnd* 中。也就是说：

- *CWinApp* 代表程序本体
- *CFrameWnd* 代表一个主框窗口（Frame Window）

但虽然我说，*WinMain* 内部动作和*WndProc* 内部动作都有着相当程度的固定行为，它们毕竟需要面对不同应用程序而有某种变化。所以，你必须以这两个类别为基础，衍生自己的类别，并改写其中一部份成员函数。

```
class CMyWinApp : public CWinApp
{
...
};
class CMyFrameWnd : public CFrameWnd
{
...
};
```

本章对衍生类别的命名规则是：在基础类别名称的前面加上"*My*"。这种规则真正上战场时不见得适用，大型程序可能会自同一个基础类别衍生出许多自己的类别。不过以教学目的而言，这种命名方式使我们从字面就知道类别之间的从属关系，颇为理想（根据我的经验，初学者会被类别的命名搞得头昏脑胀）。

CWinApp－取代 WinMain 的物件

CWinApp 的衍生对象被称为application object，可以想见，*CWinApp* 本身就代表一个程式本体。一个程序的本体是什么？回想第 1 章的SDK 程序，与程序本身有关而不与视窗有关的资料或动作有些什么？ 系统传进来的四个*WinMain* 参数算不算？*InitApplication* 和*InitInstance* 算不算？消息循环算不算？都算，是的，以下是MFC 4.x 的*CWinApp* 声明（节录自AFXWIN.H）：

```
class CWinApp : public CWinThread
{
// Attributes
    // Startup args (do not change)
    HINSTANCE m_hInstance;
    HINSTANCE m_hPrevInstance;
    LPTSTR m_lpCmdLine;
    int m_nCmdShow;

    // Running args (can be changed in InitInstance)
    LPCTSTR m_pszAppName;  // human readable name
    LPCTSTR m_pszRegistryKey;   // used for registry entries

public:  // set in constructor to override default
    LPCTSTR m_pszExeName;      // executable name (no spaces)
    LPCTSTR m_pszHelpFilePath;  // default based on module path
    LPCTSTR m_pszProfileName;   // default based on app name

public:
    // hooks for your initialization code
    virtual BOOL InitApplication();

    // overrides for implementation
    virtual BOOL InitInstance();
    virtual int ExitInstance();
    virtual int Run();
    virtual BOOL OnIdle(LONG lCount);
...
};
```

几乎可以说*CWinApp* 用来取代*WinMain* 在SDK 程序中的地位。这并不是说MFC 程序没有*WinMain*（稍后我会解释），而是说传统上SDK 程序的*WinMain* 所完成的工作现在由*CWinApp* 的三个函数完成：

```
virtual BOOL InitApplication();
virtual BOOL InitInstance();
virtual int Run();
```

WinMain 只是扮演役使它们的角色。

会不会觉得*CWinApp* 的成员变量中少了点什么东西？是不是应该有个成员变量记录主窗口的handle（或是主窗口对应之C++ 对象）？的确，在MFC 2.5 中的确有*m_pMainWnd* 这么个成员变量（以下节录自MFC 2.5 的AFXWIN.H）：

```
class CWinApp : public CCmdTarget
{
// Attributes
   // Startup args (do not change)
   HINSTANCE m_hInstance;
   HINSTANCE m_hPrevInstance;
   LPSTR m_lpCmdLine;
   int m_nCmdShow;

   // Running args (can be changed in InitInstance)
   CWnd* m_pMainWnd;         // main window (optional)
   CWnd* m_pActiveWnd;       // active main window (may not be m_pMainWnd)
   const char* m_pszAppName;   // human readable name

public:  // set in constructor to override default
   const char* m_pszExeName;      // executable name (no spaces)
   const char* m_pszHelpFilePath;  // default based on module path
   const char* m_pszProfileName;   // default based on app name

public:
   // hooks for your initialization code
   virtual BOOL InitApplication();
   virtual BOOL InitInstance();

   // running and idle processing
   virtual int Run();
   virtual BOOL OnIdle(LONG lCount);
```

```
    // exiting
    virtual int ExitInstance();
...
};
```

但从MFC 4.x 开始，*m_pMainWnd* 已经被移往*CWinThread* 中了（它是*CWinApp* 的父类别）。以下内容节录自MFC 4.x 的AFXWIN.H：

```
class CWinThread : public CCmdTarget
{
// Attributes
    CWnd* m_pMainWnd;     // main window (usually same AfxGetApp()->m_pMainWnd)
    CWnd* m_pActiveWnd;   // active main window (may not be m_pMainWnd)

    // only valid while running
    HANDLE m_hThread;      // this thread's HANDLE
    DWORD m_nThreadID;     // this thread's ID

    int GetThreadPriority();
    BOOL SetThreadPriority(int nPriority);

// Operations
    DWORD SuspendThread();
    DWORD ResumeThread();

// Overridables
    // thread initialization
    virtual BOOL InitInstance();

    // running and idle processing
    virtual int Run();
    virtual BOOL PreTranslateMessage(MSG* pMsg);
    virtual BOOL PumpMessage();    // low level message pump
    virtual BOOL OnIdle(LONG lCount); // return TRUE if more idle processing

public:
    // valid after construction
    AFX_THREADPROC m_pfnThreadProc;
...
};
```

熟悉Win32 的朋友，看到*CWinThread* 类别之中的*SuspendThread* 和*ResumeThread* 成员函数，可能会发出会心微笑。

CFrameWnd－取代 WndProc 的地位

CFrameWnd 主要用来掌握一个窗口，几乎你可以说它是用来取代SDK 程序中的窗口函式的地位。传统的SDK 窗口函数写法是：

```
long FAR PASCAL WndProc(HWND hWnd, UNIT msg, WORD wParam, LONG lParam)
{
    switch(msg) {
        case WM_COMMAND :
            switch(wParam) {
               case IDM_ABOUT :
                    OnAbout(hWnd, wParam, lParam);
                    break;
            }
            break;
        case WM_PAINT :
            OnPaint(hWnd, wParam, lParam);
            break;
        default :
            DefWindowProc(hWnd, msg, wParam, lParam);
    }
}
```

MFC 程序有新的作法，我们在Hello 程序中也为*CMyFrameWnd* 准备了两个消息处理例程，声明如下：

```
class CMyFrameWnd : public CFrameWnd
{
public:
       CMyFrameWnd();
       afx_msg void OnPaint();
       afx_msg void OnAbout();
       DECLARE_MESSAGE_MAP()
};
```

OnPaint 处理什么消息？*OnAbout* 又是处理什么消息？我想你很容易猜到，前者处理*WM_PAINT*，后者处理*WM_COMMAND* 的*IDM_ABOUT*。这看起来十分俐落，但让人搞不懂来龙去脉。程序中是不是应该有「把消息和处理函数关联在一起」的设定动作？是的，这些设定在HELLO.CPP 才看得到。但让我先着一鞭：*DECLARE_MESSAGE_MAP* 宏与此有关。

这种写法非常奇特，原因是MFC 内建了一个所谓的Message Map 机制，会把消息自动送到「与消息对映之特定函数」去；消息与处理函数之间的对映关系由程序员指定。*DECLARE_MESSAGE_MAP* 另搭配其它宏，就可以很便利地将消息与其处理函数关联在一起：

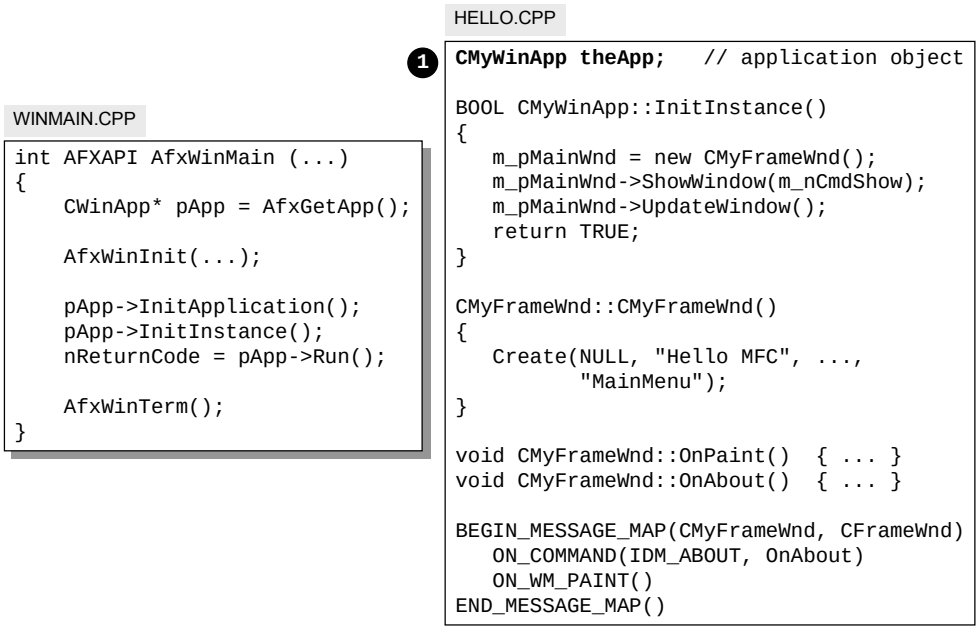
```
BEGIN_MESSAGE_MAP(CMyFrameWnd, CFrameWnd)
  ON_WM_PAINT()
  ON_COMMAND(IDM_ABOUT, OnAbout)
END_MESSAGE_MAP()
```

稍后我就来探讨这些神秘的宏。

引爆器－Application object

我们已经看过HELLO.H 声明的两个类别，现在把目光转到HELLO.CPP 身上。这个档案将两个类别实作出来，并产生一个所谓的application object。故事就从这里展开。

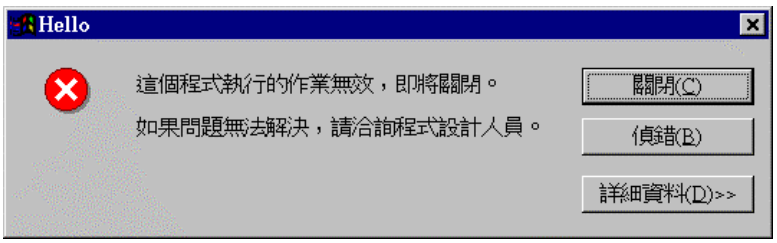
下面这张图包括右半部的Hello 源代码与左半部的MFC 源代码。从这一节以降，我将以此图解释MFC 程序的激活、运行、与结束。不同小节的图将标示出当时的程序进行状况。



上图的*theApp* 就是Hello 程序的application object，每一个MFC 应用程序都有一个，而且也只有这么一个。当你执行Hello，这个全域对象产生，于是构造式执行起来。我们并没有定义*CMyWinApp* 构造式；至于其父类别*CWinApp* 的构造式内容摘要如下（摘录自APPCORE.CPP）：

```
CWinApp::CWinApp(LPCTSTR lpszAppName)
{
    m_pszAppName = lpszAppName;

    // initialize CWinThread state
    AFX_MODULE_THREAD_STATE* pThreadState = AfxGetModuleThreadState();
    pThreadState->m_pCurrentWinThread = this;
    m_hThread = ::GetCurrentThread();
    m_nThreadID = ::GetCurrentThreadId();

    // initialize CWinApp state
    AFX_MODULE_STATE* pModuleState = AfxGetModuleState();
    pModuleState->m_pCurrentWinApp = this;

    // in non-running state until WinMain
    m_hInstance = NULL;
    m_pszHelpFilePath = NULL;
    m_pszProfileName = NULL;
    m_pszRegistryKey = NULL;
    m_pszExeName = NULL;
    m_lpCmdLine = NULL;
    m_pCmdInfo = NULL;
    ...
}
```

CWinApp 之中的成员变量将因为*theApp* 这个全域对象的诞生而获得配置与初值。如果程序中没有*theApp* 存在，编译联结还是可以顺利通过，但执行时会出现系统错误消息:



隱晦不明的 WinMain

HELLO.CPP

```
CMyWinApp theApp;   // application object

BOOL CMyWinApp::InitInstance()
{
   m_pMainWnd = new CMyFrameWnd();
   m_pMainWnd->ShowWindow(m_nCmdShow);
   m_pMainWnd->UpdateWindow();
   return TRUE;
}

CMyFrameWnd::CMyFrameWnd()
{
   Create(NULL, "Hello MFC", ...,
          "MainMenu");
}

void CMyFrameWnd::OnPaint()  { ... }
void CMyFrameWnd::OnAbout()  { ... }

BEGIN_MESSAGE_MAP(CMyFrameWnd, CFrameWnd)
   ON_COMMAND(IDM_ABOUT, OnAbout)
   ON_WM_PAINT()
END_MESSAGE_MAP()
```

WINMAIN.CPP

```
int AFXAPI AfxWinMain (...)
{
    CWinApp* pApp = AfxGetApp();

    AfxWinInit(...);

    pApp->InitApplication();
    pApp->InitInstance();
    nReturnCode = pApp->Run();

    AfxWinTerm();
}
```

theApp 配置完成后，*WinMain* 登场。我们并未撰写*WinMain* 程序代码，这是MFC 早已准备好并由联结器直接加到应用程序代码中的，其源代码列于图6-4。_tWinMain 函数的 ¡ §-*t*¡¯ 是为了支持Unicode 而准备的一个宏。

```
// in APPMODUL.CPP
extern "C" int WINAPI
_tWinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
          LPTSTR lpCmdLine, int nCmdShow)
{
       // call shared/exported WinMain
       return AfxWinMain(hInstance, hPrevInstance, lpCmdLine, nCmdShow);
}
```

此外，在DLLMODUL.CPP 中有一个*DllMain* 函数。本书并未涵盖DLL 程序设计。

```
// in WINMAIN.CPP
#0001 ////////////////////////////////////////////////////////////
#0002 // Standard WinMain implementation
#0003 //  Can be replaced as long as 'AfxWinInit' is called first
#0004
#0005 int AFXAPI AfxWinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
#0006         LPTSTR lpCmdLine, int nCmdShow)
#0007 {
#0008     ASSERT(hPrevInstance == NULL);
#0009
#0010     int nReturnCode = -1;
#0011     CWinApp* pApp = AfxGetApp();
#0012
#0013     // AFX internal initialization
#0014     if (!AfxWinInit(hInstance, hPrevInstance, lpCmdLine, nCmdShow))
#0015            goto InitFailure;
#0016
#0017     // App global initializations (rare)
#0018     ASSERT_VALID(pApp);
#0019     if (!pApp->InitApplication())
#0020            goto InitFailure;
#0021     ASSERT_VALID(pApp);
#0022
#0023     // Perform specific initializations
#0024     if (!pApp->InitInstance())
#0025     {
#0026         if (pApp->m_pMainWnd != NULL)
#0027         {
#0028             TRACE0("Warning: Destroying non-NULL m_pMainWnd\n");
#0029             pApp->m_pMainWnd->DestroyWindow();
#0030         }
#0031         nReturnCode = pApp->ExitInstance();
#0032         goto InitFailure;
#0033     }
#0034     ASSERT_VALID(pApp);
#0035
#0036     nReturnCode = pApp->Run();
#0037     ASSERT_VALID(pApp);
#0038
#0039 InitFailure:
#0040
#0041     AfxWinTerm();
#0042     return nReturnCode;
#0043 }
```

图6-4 Windows 程序进入点。源代码可从MFC 的WINMAIN.CPP 中获得。

稍加整理去芜存菁，就可以看到这个「程序进入点」主要做些什么事：

```
int AFXAPI AfxWinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                       LPTSTR lpCmdLine, int nCmdShow)
{
    int nReturnCode = -1;
    CWinApp* pApp = AfxGetApp();

    AfxWinInit(hInstance, hPrevInstance, lpCmdLine, nCmdShow);

    pApp->InitApplication();
    pApp->InitInstance();
    nReturnCode = pApp->Run();

    AfxWinTerm();
    return nReturnCode;
}
```

其中，*AfxGetApp* 是一个全域函数，定义于AFXWIN1.INL 中：

```
_AFXWIN_INLINE CWinApp* AFXAPI AfxGetApp()
       { return afxCurrentWinApp; }
```

而*afxCurrentWinApp* 又定义于AFXWIN.H 中：

```
#define afxCurrentWinApp  AfxGetModuleState()->m_pCurrentWinApp
```

再根据稍早所述*CWinApp*::*CWinApp* 中的动作，我们于是知道，*AfxGetApp* 其实就是取得*CMyWinApp* 对象指针。所以，*AfxWinMain* 中这样的动作：

```
CWinApp* pApp = AfxGetApp();
pApp->InitApplication();
pApp->InitInstance();
nReturnCode = pApp->Run();
```

其实就相当于调用：

```
CMyWinApp::InitApplication();
CMyWinApp::InitInstance();
CMyWinApp::Run();
```

因而导至调用：

```
CWinApp::InitApplication();  //因为 CMyWinApp 并没有改写InitApplication
CMyWinApp::InitInstance();   //因为 CMyWinApp 改写了 InitInstance
CWinApp::Run();              //因为 CMyWinApp 并没有改写Run
```

根据第 1 章SDK 程序设计的经验推测，*InitApplication* 应该是注册窗口类别的场所？*InitInstance* 应该是产生窗口并显示窗口的场所？*Run* 应该是攫取消息并分派消息的场所？有对有错！以下数节我将实际带你看看MFC 的源代码，如此一来就可以了解隐藏在MFC 背后的玄妙了。我的终极目标并不在MFC 源代码（虽然那的确是学习设计一个 application framework 的好教材），我只是想拿把刀子把MFC 看似朦胧的内部运作来个大解剖，挑出其经脉；有这种扎实的根基，使用MFC 才能知其然并知其所以然。下面小节分别讨论*AfxWinMain* 的四个主要动作以及引发的行为。

AfxWinInit－AFX 內部初始化动作

HELLO.CPP

```
CMyWinApp theApp;   // application object

BOOL CMyWinApp::InitInstance()
{
   m_pMainWnd = new CMyFrameWnd();
   m_pMainWnd->ShowWindow(m_nCmdShow);
   m_pMainWnd->UpdateWindow();
   return TRUE;
}

CMyFrameWnd::CMyFrameWnd()
{
   Create(NULL, "Hello MFC", ...,
          "MainMenu");
}

void CMyFrameWnd::OnPaint()  { ... }
void CMyFrameWnd::OnAbout()  { ... }

BEGIN_MESSAGE_MAP(CMyFrameWnd, CFrameWnd)
   ON_COMMAND(IDM_ABOUT, OnAbout)
   ON_WM_PAINT()
END_MESSAGE_MAP()
```

WINMAIN.CPP

```
int AFXAPI AfxWinMain (...)
{
    CWinApp* pApp = AfxGetApp();

    AfxWinInit(...);

    pApp->InitApplication();
    pApp->InitInstance();
    nReturnCode = pApp->Run();

    AfxWinTerm();
}
```

我想你已经清楚看到了，*AfxWinInit* 是继*CWinApp* 构造式之后的第一个动作。以下是它的动作摘要（节录自APPINIT.CPP）：

```
BOOL AFXAPI AfxWinInit(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                       LPTSTR lpCmdLine, int nCmdShow)
{
   ASSERT(hPrevInstance == NULL);

   // set resource handles
   AFX_MODULE_STATE* pState = AfxGetModuleState();
   pState->m_hCurrentInstanceHandle = hInstance;
   pState->m_hCurrentResourceHandle = hInstance;

   // fill in the initial state for the application
   CWinApp* pApp = AfxGetApp();
   if (pApp != NULL)
   {
      // Windows specific initialization (not done if no CWinApp)
```

```
        pApp->m_hInstance = hInstance;
        pApp->m_hPrevInstance = hPrevInstance;
        pApp->m_lpCmdLine = lpCmdLine;
        pApp->m_nCmdShow = nCmdShow;
        pApp->SetCurrentHandles();
    }

    // initialize thread specific data (for main thread)
    if (!afxContextIsDLL)
        AfxInitThread();

    return TRUE;
}
```

其中调用的*AfxInitThread* 函数的动作摘要如下（节录自THRDCORE.CPP）：

```
void AFXAPI AfxInitThread()
{
    if (!afxContextIsDLL)
    {
        // attempt to make the message queue bigger
        for (int cMsg = 96; !SetMessageQueue(cMsg) && (cMsg -= 8); )
            ;

        // set message filter proc
        _AFX_THREAD_STATE* pThreadState = AfxGetThreadState();
        ASSERT(pThreadState->m_hHookOldMsgFilter == NULL);
        pThreadState->m_hHookOldMsgFilter = ::SetWindowsHookEx(WH_MSGFILTER,
            _AfxMsgFilterHook, NULL, ::GetCurrentThreadId());

        // intialize CTL3D for this thread
        _AFX_CTL3D_STATE* pCtl3dState = _afxCtl3dState;
        if (pCtl3dState->m_pfnAutoSubclass != NULL)
            (*pCtl3dState->m_pfnAutoSubclass)(AfxGetInstanceHandle());

        // allocate thread local _AFX_CTL3D_THREAD just for automatic termination
        _AFX_CTL3D_THREAD* pTemp = _afxCtl3dThread;
    }
}
```

如果你曾经看过本书前身Visual C++ **对象导向 MFC 程序设计**，我想你可能对这句话印象深刻：「*WinMain* 一开始即调用*AfxWinInit*，注册四个窗口类别」。这是一个已成昨日黄花的事实。MFC 的确会为我们注册四个窗口类别，但不再是在*AfxWinInit* 中完成。稍后我会把注册动作挖出来，那将是窗口诞生前一刻的行为。

CWinApp::InitApplication

HELLO.CPP

```
1  CMyWinApp theApp;   // application object

   BOOL CMyWinApp::InitInstance()
   {
      m_pMainWnd = new CMyFrameWnd();
      m_pMainWnd->ShowWindow(m_nCmdShow);
      m_pMainWnd->UpdateWindow();
      return TRUE;
   }

   CMyFrameWnd::CMyFrameWnd()
   {
      Create(NULL, "Hello MFC", ...,
             "MainMenu");
   }

   void CMyFrameWnd::OnPaint()  { ... }
   void CMyFrameWnd::OnAbout()  { ... }

   BEGIN_MESSAGE_MAP(CMyFrameWnd, CFrameWnd)
      ON_COMMAND(IDM_ABOUT, OnAbout)
      ON_WM_PAINT()
   END_MESSAGE_MAP()
```

WINMAIN.CPP

```
int AFXAPI AfxWinMain (...)
{
    CWinApp* pApp = AfxGetApp();

2   AfxWinInit(...);

3   pApp->InitApplication();
    pApp->InitInstance();
    nReturnCode = pApp->Run();

    AfxWinTerm();
}
```

AfxWinInit 之后的动作是*pApp->InitApplication*。稍早我说过了，*pApp* 指向*CMyWinApp* 对象（也就是本例的*theApp*），所以，当程序调用：

```
pApp->InitApplication();
```

相当于调用：

```
CMyWinApp::InitApplication();
```

但是你要知道，*CMyWinApp* 继承自*CWinApp*，而*InitApplication* 又是*CWinApp* 的一个虚拟函数；我们并没有改写它（大部份情况下不需改写它），所以上述动作相当于调用：

```
CWinApp::InitApplication();
```

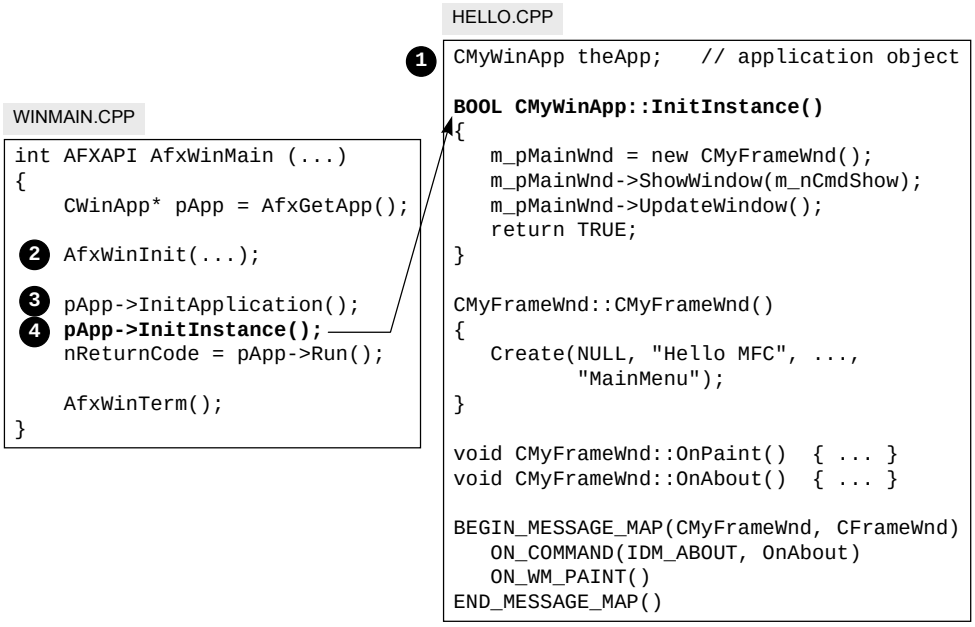
此函数之源代码出现在APPCORE.CPP 中：

```
BOOL CWinApp::InitApplication()
{
    if (CDocManager::pStaticDocManager != NULL)
    {
        if (m_pDocManager == NULL)
            m_pDocManager = CDocManager::pStaticDocManager;
        CDocManager::pStaticDocManager = NULL;
    }

    if (m_pDocManager != NULL)
        m_pDocManager->AddDocTemplate(NULL);
    else
        CDocManager::bStaticInit = FALSE;

    return TRUE;
}
```

这些动作都是MFC 为了内部管理而做的。

关于Document Template 和*CDocManager*，第 7 章和第 8 章另有说明。

CMyWinApp::InitInstance



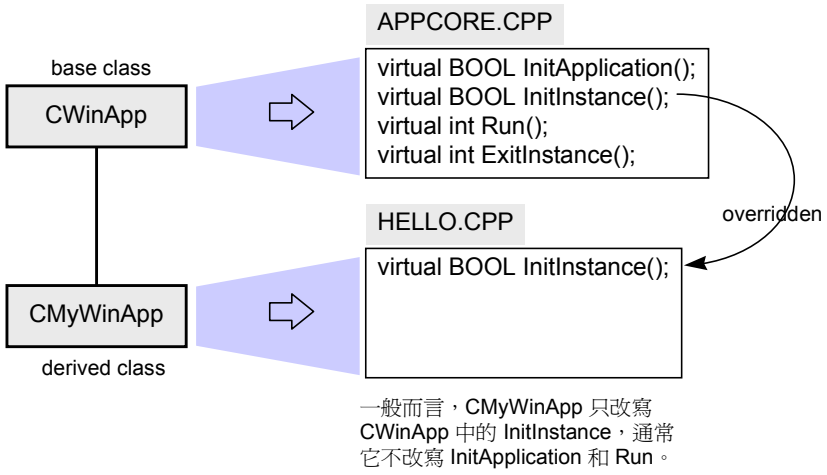
继*InitApplication* 之后，*AfxWinMain* 调用*pApp->InitInstance*。稍早我说过了，*pApp* 指向 *CMyWinApp* 对象（也就是本例的*theApp*），所以，当程序调用：

```
pApp->InitInstance();
```

相当于调用

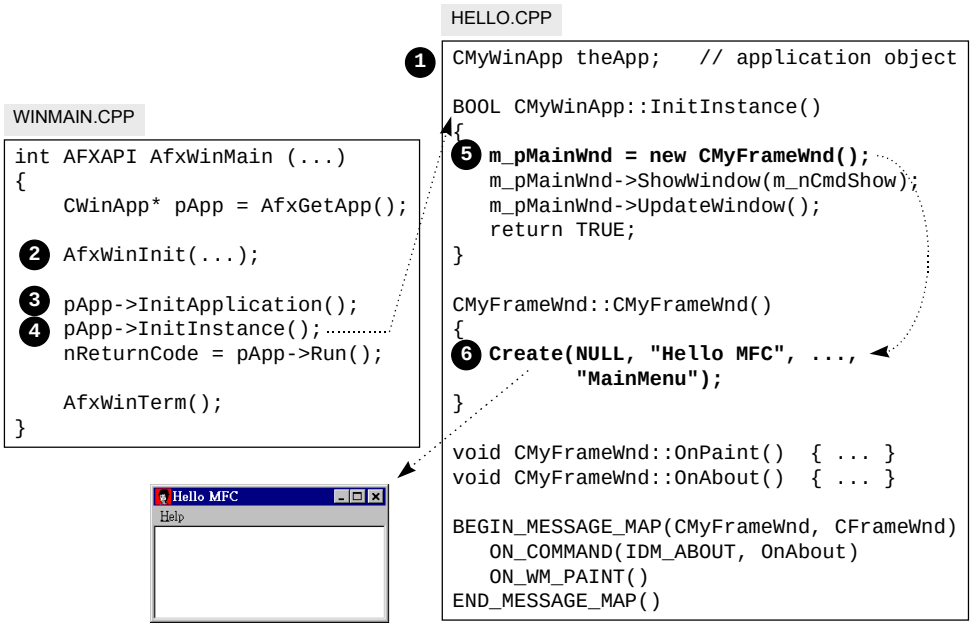
```
CMyWinApp::InitInstance();
```

但是你要知道，*CMyWinApp* 继承自*CWinApp*，而*InitInstance* 又是*CWinApp* 的一个虚拟函数。由于我们改写了它，所以上述动作的的确确就是调用我们自己（*CMyWinApp*）的这个*InitInstance* 函数。我们将在该处展开我们的主窗口生命。



注意：应用程序一定要改写虚拟函数*InitInstance*，因为它在*CWinApp* 中只是个空函数，没有任何内建（预设）动作。

CFrameWnd::Create 产生主窗口（并先注册窗口类别）



CMyWinApp::InitInstance 一开始*new* 了一个*CMyFrameWnd* 对象，准备用作主框窗口的C++ 对象。*new* 会引发构造式：

```
CMyFrameWnd::CMyFrameWnd
{
    Create(NULL, "Hello MFC", WS_OVERLAPPEDWINDOW, rectDefault, NULL,
           "MainMenu");
}
```

其中*Create* 是*CFrameWnd* 的成员函数，它将产生一个窗口。但，使用哪一个窗口类别呢？

> 这里所谓的「窗口类别」是由*RegisterClass* 所注册的一份数据结构，不是C++ 类别。

根据*CFrameWnd*::*Create* 的规格：

```
BOOL Create( LPCTSTR lpszClassName,
             LPCTSTR lpszWindowName,
             DWORD dwStyle = WS_OVERLAPPEDWINDOW,
             const RECT& rect = rectDefault,
             CWnd* pParentWnd = NULL,
             LPCTSTR lpszMenuName = NULL,
             DWORD dwExStyle = 0,
             CCreateContext* pContext = NULL );
```

八个参数中的后六个参数都有默认值，只有前两个参数必须指定。**第一个参数** *lpszClassName* 指定*WNDCLASS* 窗口类别，我们放置*NULL* 究竟代表什么意思？意思是要以MFC 内建的窗口类别产生一个标准的外框窗口。但，此时此刻Hello 程序中根本不存在任何窗口类别呀！噢，*Create* 函数在产生窗口之前会引发窗口类别的注册动作，稍后再解释。

第二个参数 *lpszWindowName* 指定窗口标题，本例指定"Hello MFC"。**第三个参数** *dwStyle* 指定窗口风格，预设是*WS_OVERLAPPEDWINDOW*，也正是最常用的一种，它被定义为（在WINDOWS.H 之中）：

```
#define WS_OVERLAPPEDWINDOW (WS_OVERLAPPED | WS_CAPTION |
                             WS_SYSMENU | WS_THICKFRAME |
                             WS_MINIMIZEBOX | WS_MAXIMIZEBOX)
```

因此如果你不想要窗口右上角的极大极小钮，就得这么做：

```
Create(NULL,
       "Hello MFC",
       WS_OVERLAPPED | WS_CAPTION | WS_SYSMENU | WS_THICKFRAME
       rectDefault,
       NULL,
       "MainMenu");
```

如果你希望窗口有垂直滚动条，就得在第三个参数上再加增*WS_VSCROLL* 风格。

除了上述标准的窗口风格，另有所谓的扩充风格，可以在*Create* 的**第七个参数** *dwExStyle* 指定之。扩充风格唯有以::*CreateWindowEx*（而非::*CreateWindow*）函数才能完成。事实上稍后你就会发现，*CFrameWnd*::*Create* 最终调用的正是::*CreateWindowEx*。Windows 3.1 提供五种窗口扩充风格：

```
WS_EX_DLGMODALFRAME
WS_EX_NOPARENTNOTIFY
WS_EX_TOPMOST
WS_EX_ACCEPTFILES
WS_EX_TRANSPARENT
```

Windows 95 有更多选择，包括*WS_EX_WINDOWEDGE* 和*WS_EX_CLIENTEDGE*，让窗口更具3D 立体感。Framework 已经自动为我们指定了这两个扩充风格。

Create 的**第四个参数***rect* 指定窗口的位置与大小。默认值*rectDefault* 是*CFrameWnd* 的一个static 成员变量，告诉Windows 以预设方式指定窗口位置与大小，就好象在SDK 程序中以*CW_USEDEFAULT* 指定给*CreateWindow* 函数一样。如果你很有主见，希望窗口在特定位置有特定大小，可以这么做：

```
Create(NULL,
       "Hello MFC",
       WS_OVERLAPPEDWINDOW,
       CRect(40, 60, 240, 460),  // 起始位置 (40,60)，寬 200，高 400）
       NULL,
       "MainMenu");
```

第五个参数*pParentWnd* 指定父窗口。对于一个top-level 窗口而言，此值应为*NULL*，表示没有父窗口（其实是有的，父窗口就是desktop 窗口）。

第六个参数*lpszMenuName* 指定菜单。本例使用一份在RC 中准备好的菜单¡§ MainMenu¡¨ 。**第八个参数***pContext* 是一个指向*CCreateContext* 结构的指针，framework 利用它，在具备Document/View 架构的程序中初始化外框窗口（第８章的「CDocTemplate 管理CDocument / CView / CFrameWnd」一节中将谈到此一主题）。本例不具备Document/View 架构，所以不必指定*pContext* 参数，默认值为*NULL*。

前面提过，*CFrameWnd*::*Create* 在产生窗口之前，会先引发窗口类别的注册动作。让我再扮一次MFC 向导，带你寻幽访胜。你会看到MFC 为我们注册的窗口类别名称，及注册动作。

◆ WINFRM.CPP

```
BOOL CFrameWnd::Create(LPCTSTR lpszClassName,
                       LPCTSTR lpszWindowName,
                       DWORD dwStyle,
                       const RECT& rect,
                       CWnd* pParentWnd,
                       LPCTSTR lpszMenuName,
                       DWORD dwExStyle,
                       CCreateContext* pContext)
{
   HMENU hMenu = NULL;
   if (lpszMenuName != NULL)
   {
      // load in a menu that will get destroyed when window gets destroyed
      HINSTANCE hInst = AfxFindResourceHandle(lpszMenuName, RT_MENU);
      hMenu = ::LoadMenu(hInst, lpszMenuName);
   }

   m_strTitle = lpszWindowName;    // save title for later

   CreateEx(dwExStyle, lpszClassName, lpszWindowName, dwStyle,
      rect.left, rect.top, rect.right - rect.left, rect.bottom - rect.top,
      pParentWnd->GetSafeHwnd(), hMenu, (LPVOID)pContext);

   return TRUE;
}
```

函数中调用*CreateEx*。注意，*CWnd* 有成员函数*CreateEx*，但其衍生类别*CFrameWnd* 并无，所以这里虽然调用的是*CFrameWnd*::*CreateEx*，其实乃是从父类别继承下来的*CWnd* 之*CreateEx*。

◆ WINCORE.CPP

```
BOOL CWnd::CreateEx(DWORD dwExStyle, LPCTSTR lpszClassName,
                    LPCTSTR lpszWindowName, DWORD dwStyle,
                    int x, int y, int nWidth, int nHeight,
                    HWND hWndParent, HMENU nIDorHMenu, LPVOID lpParam)
{
   // allow modification of several common create parameters
   CREATESTRUCT cs;
   cs.dwExStyle = dwExStyle;
   cs.lpszClass = lpszClassName;
```

```
    cs.lpszName = lpszWindowName;
    cs.style = dwStyle;
    cs.x = x;
    cs.y = y;
    cs.cx = nWidth;
    cs.cy = nHeight;
    cs.hwndParent = hWndParent;
    cs.hMenu = nIDorHMenu;
    cs.hInstance = AfxGetInstanceHandle();
    cs.lpCreateParams = lpParam;

    PreCreateWindow(cs);
    AfxHookWindowCreate(this);  //此动作将在第9章探讨。
    HWND hWnd = ::CreateWindowEx(cs.dwExStyle, cs.lpszClass,
                  cs.lpszName, cs.style, cs.x, cs.y, cs.cx, cs.cy,
                  cs.hwndParent, cs.hMenu, cs.hInstance, cs.lpCreateParams);
    ...
}
```

函数中调用的*PreCreateWindow* 是虚拟函数，*CWnd* 和*CFrameWnd* 之中都有定义。由于*this* 指针所指对象的缘故，这里应该调用的是*CFrameWnd*::*PreCreateWindow*（还记得第 2 章我说过虚拟函数常见的那种行为模式吗？）

◆ WINFRM.CPP

```
// CFrameWnd second phase creation
BOOL CFrameWnd::PreCreateWindow(CREATESTRUCT& cs)
{
    if (cs.lpszClass == NULL)
    {
        AfxDeferRegisterClass(AFX_WNDFRAMEORVIEW_REG);
        cs.lpszClass = _afxWndFrameOrView;  // COLOR_WINDOW background
    }
    ...
}
```

其中*AfxDeferRegisterClass* 是一个定义于AFXIMPL.H 中的宏。

◆ AFXIMPL.H

```
#define AfxDeferRegisterClass(fClass) \
  ((afxRegisteredClasses & fClass) ? TRUE : AfxEndDeferRegisterClass(fClass))
```

这个宏表示，如果变量*afxRegisteredClasses* 的值显示系统已经注册了*fClass* 这种视窗类别，MFC 就啥也不做；否则就调用*AfxEndDeferRegisterClass*(fClass)，准备注册之。*afxRegisteredClasses* 定义于AFXWIN.H，是一个旗标变量，用来记录已经注册了哪些视窗类别：

```
// in AFXWIN.H
#define afxRegisteredClasses  AfxGetModuleState()->m_fRegisteredClasses
```

◆ WINCORE.CPP：

```
#0001  BOOL AFXAPI AfxEndDeferRegisterClass(short fClass)
#0002  {
#0003      BOOL bResult = FALSE;
#0004
#0005      // common initialization
#0006      WNDCLASS wndcls;
#0007      memset(&wndcls, 0, sizeof(WNDCLASS));   // start with NULL defaults
#0008      wndcls.lpfnWndProc = DefWindowProc;
#0009      wndcls.hInstance = AfxGetInstanceHandle();
#0010      wndcls.hCursor = afxData.hcurArrow;
#0011
#0012      AFX_MODULE_STATE* pModuleState = AfxGetModuleState();
#0013      if (fClass & AFX_WND_REG)
#0014      {
#0015          // Child windows - no brush, no icon, safest default class styles
#0016          wndcls.style = CS_DBLCLKS | CS_HREDRAW | CS_VREDRAW;
#0017          wndcls.lpszClassName = _afxWnd;
#0018          bResult =  AfxRegisterClass(&wndcls);
#0019          if (bResult)
#0020              pModuleState->m_fRegisteredClasses |= AFX_WND_REG;
#0021      }
#0022      else if (fClass & AFX_WNDOLECONTROL_REG)
#0023      {
#0024          // OLE Control windows - use parent DC for speed
#0025          wndcls.style |= CS_PARENTDC | CS_DBLCLKS | CS_HREDRAW | CS_VREDRAW;
#0026          wndcls.lpszClassName = _afxWndOleControl;
#0027          bResult =  AfxRegisterClass(&wndcls);
#0028          if (bResult)
```

```
#0029            pModuleState->m_fRegisteredClasses |= AFX_WNDOLECONTROL_REG;
#0030        }
#0031        else if (fClass & AFX_WNDCONTROLBAR_REG)
#0032        {
#0033            // Control bar windows
#0034            wndcls.style = 0;   // control bars don't handle double click
#0035            wndcls.lpszClassName = _afxWndControlBar;
#0036            wndcls.hbrBackground = (HBRUSH)(COLOR_BTNFACE + 1);
#0037            bResult =  AfxRegisterClass(&wndcls);
#0038            if (bResult)
#0039                pModuleState->m_fRegisteredClasses |= AFX_WNDCONTROLBAR_REG;
#0040        }
#0041        else if (fClass & AFX_WNDMDIFRAME_REG)
#0042        {
#0043            // MDI Frame window (also used for splitter window)
#0044            wndcls.style = CS_DBLCLKS;
#0045            wndcls.hbrBackground = NULL;
#0046            bResult = RegisterWithIcon(&wndcls, _afxWndMDIFrame,
                                            AFX_IDI_STD_MDIFRAME);
#0047            if (bResult)
#0048                pModuleState->m_fRegisteredClasses |= AFX_WNDMDIFRAME_REG;
#0049        }
#0050        else if (fClass & AFX_WNDFRAMEORVIEW_REG)
#0051        {
#0052            // SDI Frame or MDI Child windows or views - normal colors
#0053            wndcls.style = CS_DBLCLKS | CS_HREDRAW | CS_VREDRAW;
#0054            wndcls.hbrBackground = (HBRUSH) (COLOR_WINDOW + 1);
#0055            bResult = RegisterWithIcon(&wndcls, _afxWndFrameOrView,
                                            AFX_IDI_STD_FRAME);
#0056            if (bResult)
#0057                pModuleState->m_fRegisteredClasses |= AFX_WNDFRAMEORVIEW_REG;
#0058        }
#0059        else if (fClass & AFX_WNDCOMMCTLS_REG)
#0060        {
#0061            InitCommonControls();
#0062            bResult = TRUE;
#0063            pModuleState->m_fRegisteredClasses |= AFX_WNDCOMMCTLS_REG;
#0064        }
#0065
#0066        return bResult;
#0067    }
```

出现在上述函数中的六个窗口类别卷标代码，分别定义于AFXIMPL.H 中：

```
#define AFX_WND_REG             (0x0001)
#define AFX_WNDCONTROLBAR_REG   (0x0002)
#define AFX_WNDMDIFRAME_REG     (0x0004)
#define AFX_WNDFRAMEORVIEW_REG  (0x0008)
#define AFX_WNDCOMMCTLS_REG     (0x0010)
#define AFX_WNDOLECONTROL_REG   (0x0020)
```

出现在上述函数中的五个窗口类别名称，分别定义于WINCORE.CPP 中：

```
const TCHAR _afxWnd[] = AFX_WND;
const TCHAR _afxWndControlBar[] = AFX_WNDCONTROLBAR;
const TCHAR _afxWndMDIFrame[] = AFX_WNDMDIFRAME;
const TCHAR _afxWndFrameOrView[] = AFX_WNDFRAMEORVIEW;
const TCHAR _afxWndOleControl[] = AFX_WNDOLECONTROL;
```

而等号右手边的那些AFX_ 常数又定义于AFXIMPL.H 中：

```
#ifndef _UNICODE
#define _UNICODE_SUFFIX
#else
#define _UNICODE_SUFFIX _T("u")
#endif

#ifndef _DEBUG
#define _DEBUG_SUFFIX
#else
#define _DEBUG_SUFFIX _T("d")
#endif

#ifdef _AFXDLL
#define _STATIC_SUFFIX
#else
#define _STATIC_SUFFIX _T("s")
#endif

#define AFX_WNDCLASS(s) \
  _T("Afx") _T(s) _T("42") _STATIC_SUFFIX _UNICODE_SUFFIX _DEBUG_SUFFIX

#define AFX_WND                 AFX_WNDCLASS("Wnd")
#define AFX_WNDCONTROLBAR       AFX_WNDCLASS("ControlBar")
#define AFX_WNDMDIFRAME         AFX_WNDCLASS("MDIFrame")
#define AFX_WNDFRAMEORVIEW      AFX_WNDCLASS("FrameOrView")
#define AFX_WNDOLECONTROL       AFX_WNDCLASS("OleControl")
```

所以，如果在Windows 95（non-Unicode）中使用MFC 动态联结版和除错版，五个窗口类别的名称将是：

```
"AfxWnd42d"
"AfxControlBar42d"
"AfxMDIFrame42d"
"AfxFrameOrView42d"
"AfxOleControl42d"
```

如果在Windows NT（Unicode 环境）中使用MFC 静态联结版和除错版，五个窗口类别的名称将是：

```
"AfxWnd42sud"
"AfxControlBar42sud"
"AfxMDIFrame42sud"
"AfxFrameOrView42sud"
"AfxOleControl42sud"
```

这五个窗口类别的使用时机为何？稍后再来一探究竟。

让我们再回顾*AfxEndDeferRegisterClass* 的动作。它调用两个函数完成实际的窗口类别注册动作，一个是*RegisterWithIcon*，一个是*AfxRegisterClass*：

```
static BOOL AFXAPI RegisterWithIcon(WNDCLASS* pWndCls,
       LPCTSTR lpszClassName, UINT nIDIcon)
{
   pWndCls->lpszClassName = lpszClassName;
   HINSTANCE hInst = AfxFindResourceHandle(
          MAKEINTRESOURCE(nIDIcon), RT_GROUP_ICON);
   if ((pWndCls->hIcon = ::LoadIcon(hInst, MAKEINTRESOURCE(nIDIcon))) == NULL)
   {
      // use default icon
      pWndCls->hIcon = ::LoadIcon(NULL, IDI_APPLICATION);
   }
   return AfxRegisterClass(pWndCls);
}
```

```
BOOL AFXAPI AfxRegisterClass(WNDCLASS* lpWndClass)
{
   WNDCLASS wndcls;
```

```
    if (GetClassInfo(lpWndClass->hInstance,
                     lpWndClass->lpszClassName, &wndcls))
    {
        // class already registered
        return TRUE;
    }

    ::RegisterClass(lpWndClass);
    ...
    return TRUE;
}
```

注意，不同类别的*PreCreateWindow* 成员函数都是在窗口产生之前一刻被调用，准备用来注册窗口类别。如果我们指定的窗口类别是*NULL*，那么就使用系统预设类别。从*CWnd* 及其各个衍生类别的*PreCreateWindow* 成员函数可以看出，整个Framework 针对不同功能的窗口使用了哪些窗口类别：

```
// in WINCORE.CPP
BOOL CWnd::PreCreateWindow(CREATESTRUCT& cs)
{

    {
        AfxDeferRegisterClass(AFX_WND_REG);
        ...
        cs.lpszClass = _afxWnd;          (这表示CWnd 使用的窗口类别是_afxWnd)
    }
    return TRUE;
}

// in WINFRM.CPP
BOOL CFrameWnd::PreCreateWindow(CREATESTRUCT& cs)
{
    if (cs.lpszClass == NULL)
    {
        AfxDeferRegisterClass(AFX_WNDFRAMEORVIEW_REG);
        ...
        cs.lpszClass = _afxWndFrameOrView;     (这表示CFrameWnd 使用的窗口
    }                                          类别是_afxWndFrameOrView)
    ...
}

// in WINMDI.CPP
BOOL CMDIFrameWnd::PreCreateWindow(CREATESTRUCT& cs)
```

```
{
    if (cs.lpszClass == NULL)
    {
        AfxDeferRegisterClass(AFX_WNDMDIFRAME_REG);
        ...
        cs.lpszClass = _afxWndMDIFrame;       (这表示CMDIFrameWnd 使用的窗口
    }                                          类别是_afxWndMDIFrame)
    return TRUE;
}

// in WINMDI.CPP
BOOL CMDIChildWnd::PreCreateWindow(CREATESTRUCT& cs)
{
    ...
    return CFrameWnd::PreCreateWindow(cs);   (这表示CMDIChildWnd 使用的窗口
}                                             类别是_afxWndFrameOrView)

// in VIEWCORE.CPP
BOOL CView::PreCreateWindow(CREATESTRUCT & cs)
{
    if (cs.lpszClass == NULL)
    {
        AfxDeferRegisterClass(AFX_WNDFRAMEORVIEW_REG);
        ...
        cs.lpszClass = _afxWndFrameOrView;     (这表示CView 使用的窗口
    }                                           类别是_afxWndFrameOrView)
    ...
}
```

题外话：「*Create* 是一个比较粗糙的函数，不提供我们对图标（icon）或鼠标光标的设定，所以在*Create* 函数中我们看不到相关参数」。这样的说法对吗？虽然「不能够让我们指定窗口图标以及鼠标光标」是事实，但这本来就与*Create* 无关。回忆SDK 程序，指定图标和光标形状实为*RegisterClass* 的责任而非*CreateWindow* 的责任！

MFC 程序的*RegisterClass* 动作并非由程序员自己来做，因此似乎难以改变图标。不过，MFC 还是开放了一个窗口，我们可以在HELLO.RC 这么设定图标：

```
AFX_IDI_STD_FRAME ICON DISCARDABLE "HELLO.ICO"
```

你可以从*AfxEndDeferRegisterClass* 的第55 行看出，当它调用*RegisterWithIcon* 时，指定的icon 正是AFX_IDI_STD_FRAME。

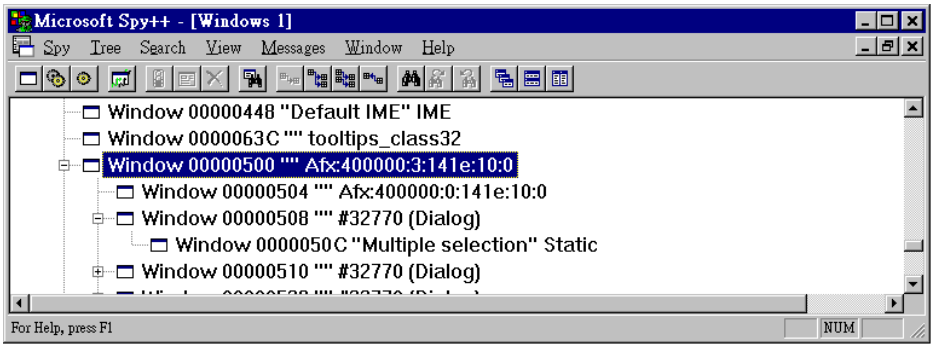
鼠标光标的设定就比较麻烦了。要改变光标形状，我们必须调用*AfxRegisterWndClass*（其中有¡¨ Cursor¡ 参数）注册自己的窗口类别；然后再将其传回值（一个字符串）做为*Create* 的第一个参数。

奇怪的窗口类别名称 Afx:b:14ae:6:3e8f

当应用程序调用*CFrameWnd::Create*（或*CMDIFrameWnd*::*LoadFrame*，第 7 章）准备产生窗口时，MFC 才会在*Create* 或*LoadFrame* 内部所调用的*PreCreateWindow* 虚拟函式中为你产生适当的窗口类别。你已经在上一节看到了，这些窗口类别的名称分别是（假设在Win95 中使用MFC 4.2 动态联结版和除错版）：

```
"AfxWnd42d"
"AfxControlBar42d"
"AfxMDIFrame42d"
"AfxFrameOrView42d"
"AfxOleControl42d"
```

然而，当我们以Spy++（VC++ 所附的一个工具）观察窗口类别的名称，却发现：



窗口类别名称怎么会变成像Afx:b:14ae:6:3e8f 这副奇怪模样呢？原来是Application Framework 玩了一些把戏，它把这些窗口类别名称转换为Afx:x:y:z:w 的型式，成为独一无二的窗口类别名称：

x: 窗口风格（window style）的hex

y: 窗口鼠标光标的hex 值　值

z: 窗口背景颜色的hex 值

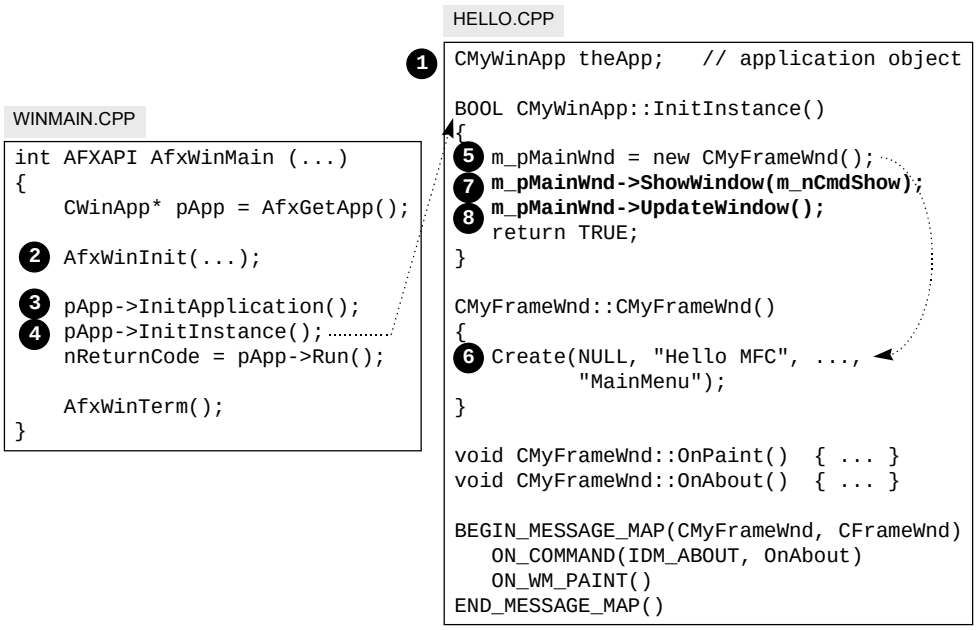
w: 窗口图标（icon）的hex 值

如果你要使用原来的（MFC 预设的）那些个窗口类别，但又希望拥有自己定义的一个有意义的类别名称，你可以改写*PreCreateWindow* 虚拟函数（因为*Create* 和*LoadFrame* 的内部都会调用它），在其中先利用API 函数*GetClassInfo* 获得该类别的一个副本，更改其类别结构中的*lpszClassName* 字段（ 甚至更改其*hIcon* 字段），再以*AfxRegisterClass* 重新注册之，例如：

```
#0000  #define MY_CLASSNAME  "MyClassName"
#0001
#0002  BOOL CMainFrame::PreCreateWindow(CREATESTRUCT& cs)
#0003  {
#0004      static LPCSTR className = NULL;
#0005
#0006      if (!CFrameWnd::PreCreateWindow(cs))
#0007          return FALSE;
#0008
#0009      if (className==NULL) {
#0010          // One-time class registration
#0011          // The only purpose is to make the class name something
#0012          // meaningful instead of "Afx:0x4d:27:32:hup1hup:hike!"
#0013          //
#0014          WNDCLASS wndcls;
#0015          ::GetClassInfo(AfxGetInstanceHandle(), cs.lpszClass, &wndcls);
#0016          wndcls.lpszClassName = MY_CLASSNAME;
#0017          wndcls.hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
#0018          VERIFY(AfxRegisterClass(&wndcls));
#0019          className=TRACEWND_CLASSNAME;
#0020      }
#0021      cs.lpszClass = className;
#0022
#0023      return TRUE;
#0024  }
```

本书附录D「以MFC 重建Debug Window（DBWIN）」会运用到这个技巧。

窗口显示与更新



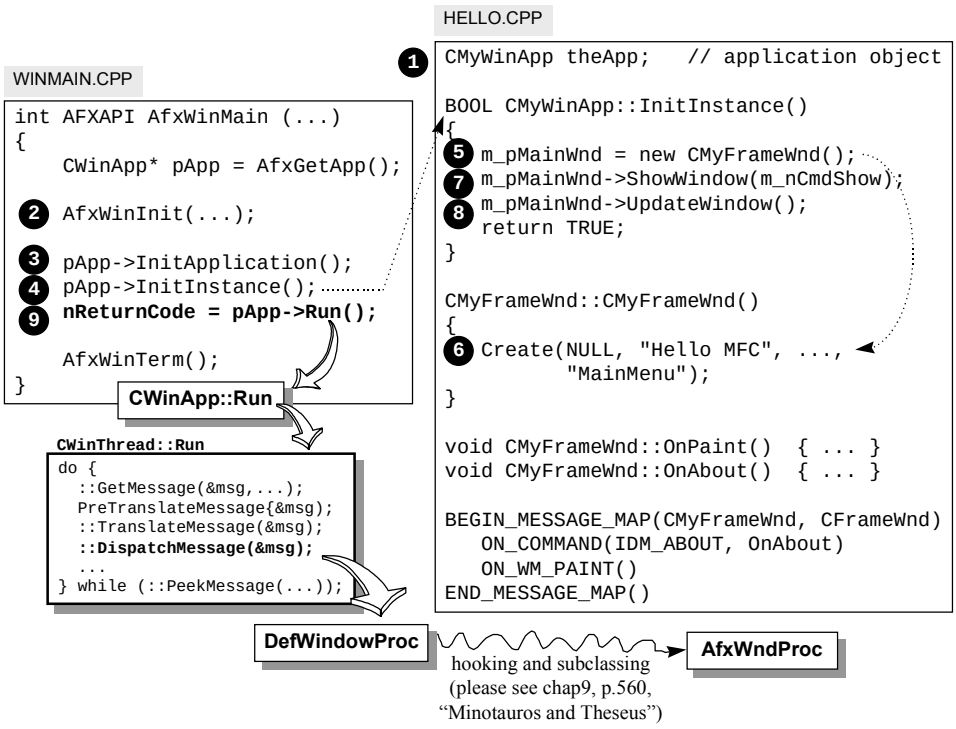
CMyFrameWnd::CMyFrameWnd 结束后， 窗口已经诞生出来； 程序流程又回到 *CMyWinApp*::*InitInstance* ， 于是调用*ShowWindow* 函数令窗口显示出来， 并调用 *UpdateWindow* 函数令Hello 程序送出*WM_PAINT* 消息。

我们很关心这个*WM_PAINT* 消息如何送到窗口函数的手中。而且，窗口函数又在哪里？

MFC 程序是不是也像SDK 程序一样，有一个*GetMessage*/*DispatchMesage* 循环？是否每个窗口也都有一个窗口函数，并以某种方式进行消息的判断与处理？

两者都是肯定的。我们马上来寻找证据。

CWinApp::Run - 程序生命的活水源头



Hello 程序进行到这里，窗口类别注册好了，窗口诞生并显示出来了，*UpdateWindow* 被调用，使得消息队列中出现了一个*WM_PAINT* 消息，等待被处理。现在，执行的脚步到达*pApp->Run*。

稍早我说过了，*pApp* 指向*CMyWinApp* 对象（也就是本例的*theApp*），所以，当程序调用：

```
pApp->Run();
```

相当于调用：

```
CMyWinApp::Run();
```

要知道，*CMyWinApp* 继承自*CWinApp*，而*Run* 又是*CWinApp* 的一个虚拟函数。我们并没有改写它（大部份情况下不需改写它），所以上述动作相当于调用：

```
CWinApp::Run();
```

其源代码出现在APPCORE.CPP 中：

```
int CWinApp::Run()
{
    if (m_pMainWnd == NULL && AfxOleGetUserCtrl())
    {
        // Not launched /Embedding or /Automation, but has no main window!
        TRACE0("Warning: m_pMainWnd is NULL in CWinApp::Run - quitting
                application.\n");
        AfxPostQuitMessage(0);
    }
    return CWinThread::Run();
}
```

32 位MFC 与16 位MFC 的巨大差异在于*CWinApp* 与*CCmdTarget* 之间多出了一个*CWinThread*，事情变得稍微复杂一些。*CWinThread* 定义于THRDCORE.CPP：

```
int CWinThread::Run()
{
    // for tracking the idle time state
    BOOL bIdle = TRUE;
    LONG lIdleCount = 0;

    // acquire and dispatch messages until a WM_QUIT message is received.
    for (;;)
    {
        // phase1: check to see if we can do idle work
        while (bIdle &&
               !::PeekMessage(&m_msgCur, NULL, NULL, NULL, PM_NOREMOVE))
        {
            // call OnIdle while in bIdle state
            if (!OnIdle(lIdleCount++))
                bIdle = FALSE; // assume "no idle" state
        }

        // phase2: pump messages while available
        do
        {
            // pump message, but quit on WM_QUIT
```

```
            if (!PumpMessage())
                return ExitInstance();

            // reset "no idle" state after pumping "normal" message
            if (IsIdleMessage(&m_msgCur))
            {
                bIdle = TRUE;
                lIdleCount = 0;
            }

        } while (::PeekMessage(&m_msgCur, NULL, NULL, NULL, PM_NOREMOVE));
    }

    ASSERT(FALSE);  // not reachable
}

BOOL CWinThread::PumpMessage()
{
    if (!::GetMessage(&m_msgCur, NULL, NULL, NULL))
    {
        return FALSE;
    }

    // process this message
    if (m_msgCur.message != WM_KICKIDLE && !PreTranslateMessage(&m_msgCur))
    {
        ::TranslateMessage(&m_msgCur);
        ::DispatchMessage(&m_msgCur);
    }
    return TRUE;
}
```

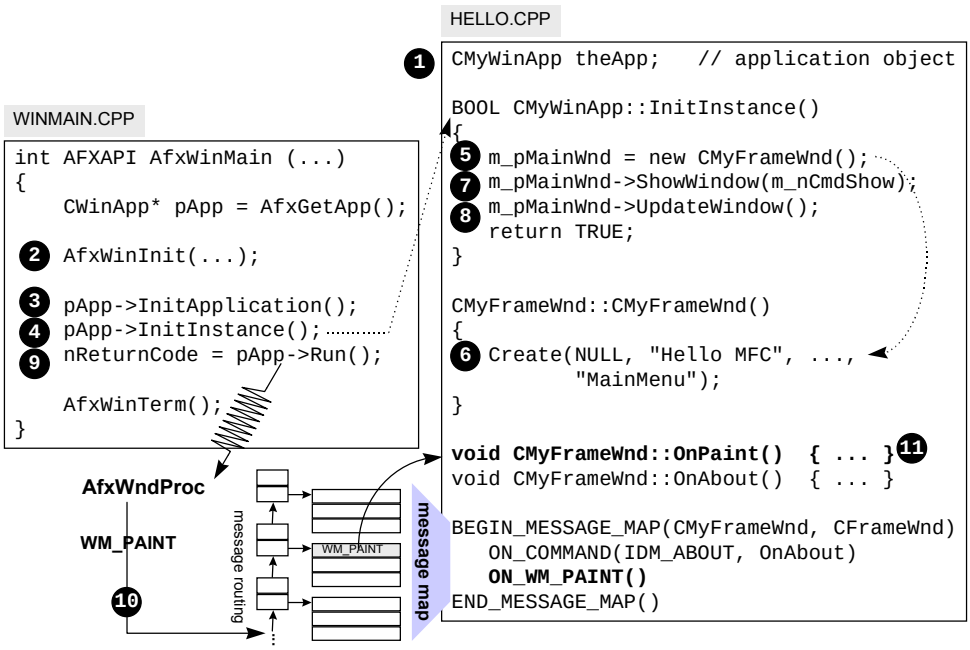
获得的消息如何交给适当的例程去处理呢？SDK 程序的作法是调用*DispatchMessage*，把消息丢给窗口函数；MFC 也是如此。但我们并未在Hello 程序中提供任何窗口函数，是的，窗口函数事实上由MFC 提供。回头看看前面*AfxEndDeferRegisterClass* 源代码，它在注册四种窗口类别之前已经指定窗口函数为：

```
wndcls.lpfnWndProc = DefWindowProc;
```

注意，虽然窗口函数被指定为*DefWindowProc* 成员函数，但事实上消息并不是被唧往该处，而是一个名为*AfxWndProc* 的全域函数去。这其中牵扯到MFC 暗中做了大挪移的手脚（利用hook 和subclassing），我将在第 9 章详细讨论这个「乾坤大挪移」。

你看，*WinMain* 已由MFC 提供，窗口类别已由MFC 注册完成、连窗口函数也都由MFC 提供。那么我们（程序员）如何为特定的消息设计特定的处理例程？MFC 应用程序对讯息的辨识与判别是采用所谓的「Message Map 机制」。

把消息与处理函数串接在一起：Message Map 机制



基本上Message Map 机制是为了提供更方便的程序接口（例如宏或表格），让程序员很方便就可以建立起消息与处理例程的对应关系。这并不是什么新发明，我在第 1 章示范了一种风格简明的SDK 程序写法，就已经展现出这种精神。

MFC 提供给应用程序使用的「很方便的接口」是两组宏。以Hello 的主窗口为例，第一个动作是在HELLO.H 的*CMyFrameWnd* 加上*DECLARE_MESSAGE_MAP*：

```
class CMyFrameWnd : public CFrameWnd
{
public:
       CMyFrameWnd();
       afx_msg void OnPaint();
       afx_msg void OnAbout();
       DECLARE_MESSAGE_MAP()
};
```

第二个动作是在HELLO.CPP 的任何位置（当然不能在函数之内）使用宏如下：

```
BEGIN_MESSAGE_MAP(CMyFrameWnd, CFrameWnd)
    ON_WM_PAINT()
    ON_COMMAND(IDM_ABOUT, OnAbout)
END_MESSAGE_MAP()
```

这么一来就把消息*WM_PAINT* 导到*OnPaint* 函数， 把*WM_COMMAND*（*IDM_ABOUT*）导到OnAbout 函数去了。但是，单凭一个*ON_WM_PAINT* 宏，没有任何参数，如何使*WM_PAINT* 流到*OnPaint* 函数呢？

MFC 把消息主要分为三大类，Message Map 机制中对于消息与函数间的对映关系也明定以下三种:

■ 标准Windows 消息（WM_xxx）的对映规则：

宏名称	对映消息	消息处理函数（名称已由系统预设）
ON_WM_CHAR	WM_CHAR	OnChar
ON_WM_CLOSE	WM_CLOSE	OnClose
ON_WM_CREATE	WM_CREATE	OnCreate
ON_WM_DESTROY	WM_DESTROY	OnDestroy
ON_WM_LBUTTONDOWN	WM_LBUTTONDOWN	OnLButtonDown
ON_WM_LBUTTONUP	WM_LBUTTONUP	OnLButtonUp
ON_WM_MOUSEMOVE	WM_MOUSEMOVE	OnMouseMove
ON_WM_PAINT	WM_PAINT	OnPaint
...		

■　命令消息（*WM_COMMAND*）的一般性对映规则是：

```
ON_COMMAND(<id>,<memberFxn>)
```

例如：

```
ON_COMMAND(IDM_ABOUT, OnAbout)
ON_COMMAND(IDM_FILENEW, OnFileNew)
ON_COMMAND(IDM_FILEOPEN, OnFileOpen)
ON_COMMAND(IDM_FILESAVE, OnFileSave)
```

■　「Notification 消息」（由控制组件产生，例如*BN_xxx*）的对映机制的宏分为好几种（因为控制组件本就分为好几种），以下各举一例做代表：

控制组件	宏名称	消息处理函数
Button	ON_BN_CLICKED(<id>,<memberFxn>)	memberFxn
ComboBox	ON_CBN_DBLCLK(<id>,<memberFxn>)	memberFxn
Edit	ON_EN_SETFOCUS(<id>,<memberFxn>)	memberFxn
ListBox	ON_LBN_DBLCLK(<id>,<memberFxn>)	memberFxn

各个消息处理函数均应以afx_msg void 为函数型式。

为什么经过这样的宏之后，消息就会自动流往指定的函数去呢？谜底在于Message Map 的结构设计。如果你把第 3 章的Message Map 仿真程序好好研究过，现在应该已是成竹在胸。我将在第 9 章再讨论MFC 的Message Map。

好奇心摆两旁，还是先把实用上的问题放中间吧。如果某个消息在Message Map 中找不到对映记录，消息何去何从？答案是它会往基础类别流窜，这个消息流窜动作称为「Message Routing」。如果一直窜到最基础的类别仍找不到对映的处理例程，自会有预设函数来处理，就像SDK 中的*DefWindowProc* 一样。

MFC 的*CCmdTarget* 所衍生下来的每一个类别都可以设定自己的Message Map，因为它们都可能（可以）收到消息。

消息流动是个颇为复杂的机制，它和Document/View、动态生成（Dynamic Creation），文件读写（Serialization）一样，都是需要特别留心的地方。

来龙去脉总整理

前面各节的目的就是如何将表面上看来不知所以然的MFC 程序对映到我们在SDK 程序设计中学习到的消息流动观念，从而清楚地掌握MFC 程序的诞生与死亡。让我对MFC 程序的来龙去脉再做一次总整理。

程序的诞生：

- Application object 产生，内存于是获得配置，初值亦设立了。
- *Afx WinMain* 执行*AfxWinInit*，后者又调用*AfxInitThread*，把消息队列尽量加大到96。
- *Afx WinMain* 执行*InitApplication*。这是*CWinApp* 的虚拟函数，但我们通常不改写它。
- *AfxWinMain* 执行*InitInstance*。这是*CWinApp* 的虚拟函数，我们必须改写它。
- *CMyWinApp*::*InitInstance 'new'* 了一个*CMyFrameWnd* 对象。
- *CMyFrameWnd* 构造式调用*Create*，产生主窗口。我们在*Create* 参数中指定的窗口类别是*NULL*， 于是MFC 根据窗口种类， 自行为我们注册一个名为"AfxFrameOrView42d" 的窗口类别。
- 回到*InitInstance* 中继续执行*ShowWindow*，显示窗口。
- 执行*UpdateWindow*，于是发出*WM_PAINT*。
- 回到*AfxWinMain*，执行*Run*，进入消息循环。

程序开始运作：

- 程序获得*WM_PAINT* 消息（藉由*CWinApp*::*Run* 中的::*GetMessage* 循环）。
- *WM_PAINT* 经由::*DispatchMessage* 送到窗口函数*CWnd*::*DefWindowProc* 中。

- *CWnd*::*DefWindowProc* 将消息绕行过消息映射表格（Message Map）。
- 绕行过程中发现有吻合项目，于是调用项目中对应的函数。此函数是应用程序利用*BEGIN_MESSAGE_MAP* 和*END_MESSAGE_MAP* 之间的宏设立起来的。
- 标准消息的处理例程亦有标准命名，例如*WM_PAINT* 必然由*OnPaint* 处理。

以下是程序的死亡：

- 使用者选按【File/Close】，于是发出*WM_CLOSE*。
- *CMyFrameWnd* 并没有设置*WM_CLOSE* 处理例程，于是交给预设之处理例程。
- 预设函数对于*WM_CLOSE* 的处理方式是调用::*DestroyWindow*，并因而发出*WM_DESTROY*。
- 预设之*WM_DESTROY* 处理方式是调用::*PostQuitMessage*，因此发出*WM_QUIT*。
- *CWinApp*::*Run* 收到*WM_QUIT* 后会结束其内部之消息循环，然后调用*ExitInstance*，这是*CWinApp* 的一个虚拟函数。
- 如果*CMyWinApp* 改写了*ExitInstance*，那么*CWinApp*::*Run* 所调用的就是*CMyWinApp*::*ExitInstance*，否则就是*CWinApp*::*ExitInstance*。
- 最后回到*AfxWinMain*，执行*AfxWinTerm*，结束程序。

Callback 函数

Hello 的*OnPaint* 在程序收到*WM_PAINT* 之后开始运作。为了让"Hello, MFC" 字样从天而降并有动画效果，程序采用*LineDDA* API 函数。我的目的一方面是为了示范消息的处理，一方面也为了示范MFC 程序如何调用Windows API 函数。许多人可能不熟悉*LineDDA*，所以我也一并介绍这个有趣的函数。

首先介绍*LineDDA*：

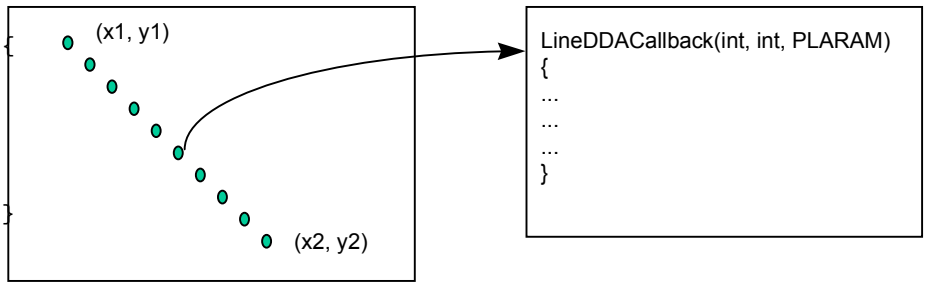
```
void WINAPI LineDDA(int, int, int, int, LINEDDAPROC, LPARAM);
```

这个函数用来做动画十分方便，你可以利用前四个参数指定屏幕上任意两点的(x,y)座标，此函数将以Bresenham 算法（注) 计算出通过两点之直线中的每一个屏幕图素座标；每计算出一个坐标，就通知由*LineDDA* 第五个参数所指定的callback 函数。这个callback 函数的型式必须是：

```
typedef void (CALLBACK* LINEDDAPROC)(int, int, LPARAM);
```

通常我们在这个callback 函数中设计绘图动作。玩过Windows 的接龙游戏吗？接龙成功后扑克牌的跳动效果就可以利用*LineDDA* 完成。虽然扑克牌的跳动路径是一条曲线，但将曲线拆成数条直线并不困难。*LineDDA* 的第六个（最后一个）参数可以视应用程序的需要传递一个32 位指针，本例中Hello 传的是一个Device Context。

> Bresenham 算法是计算机图学中为了「显示器（屏幕或打印机）系由图素构成」的这个特性而设计出来的算法，使得求直线各点的过程中全部以整数来运算，因而大幅提升计算速度。



你可以指定两个坐标点, `LineDDA` 将以 `Bresenham` 算法计算出通过两点之直线中每一个屏幕图素的坐标。每计算出一个坐标，就以该坐标为参数，调用你所指定的`callback` 函数。

图6-6 `LineDDA` 函数说明

LineDDA 并不属于任何一个MFC 类别，因此调用它必须使用C++ 的"scope operator"（也就是::）：

```
void CMyFrameWnd::OnPaint()
{
CPaintDC dc(this);
CRect rect;

   GetClientRect(rect);

   dc.SetTextAlign(TA_BOTTOM | TA_CENTER);

   ::LineDDA(rect.right/2, 0, rect.right/2, rect.bottom/2,
      (LINEDDAPROC) LineDDACallback, (LPARAM) (LPVOID) &dc);
}
```

其中*LineDDACallback* 是我们准备的callback 函数，必须在类别中先有声明：

```
class CMyFrameWnd : public CFrameWnd
{
...
private:
   static VOID CALLBACK LineDDACallback(int,int,LPARAM);
};
```

请注意，如果类别的成员函数是一个callback 函数， 你必须声明它为"static"，才能把C++ 编译器加诸于函数的一个隐藏参数*this* 去掉（请看方块批注)。

以类别的成员函数作为 Windows callback 函数

虽然现在来讲这个题目，对初学者而言恐怕是过于艰深，但我想毕竟还是个好机会--- 我可以在介绍如何使用callback 函数的场合，顺便介绍一些C++ 的重要观念。

首先我要很快地解释一下什么是callback 函数。凡是由你设计而却由Windows 系统调用的函数，统称为callback 函数。这些函数都有一定的类型，以配合Windows 的调用动作。

某些Windows API 函数会要求以callback 函数作为其参数之一，这些API 例如

SetTimer、*LineDDA*、*EnumObjects*。通常这种API 会在进行某种行为之后或满足某种状态之时调用该callback 函数。图6-6 已解释过*LineDDA*调用callback 函数的时机；下面即将示范的*EnumObjects* 则是在发现某个Device Context 的GDI object 符合我们的指定类型时，调用callback 函数。

好，现在我们要讨论的是，什么函数有资格在C++ 程序中做为callback 函数？这个问题的背后是：C++ 程序中的callback 函数有什么特别的吗？为什么要特别提出讨论？

是的，特别之处在于，C++ 编译器为类别成员函数多准备了一个隐藏参数（程序代码中看不到），这使得函数类型与Windows callback 函数的预设类型不符。

假设我们有一个*CMyclass* 如下：

```
class CMyclass {
  private :
    int nCount;
    int CALLBACK _export
        EnumObjectsProc(LPSTR lpLogObject, LPSTR lpData);
  public :
    void enumIt(CDC& dc);
}
void CMyclass::enumIt(CDC& dc)
{
  // 注册callback 函数
  dc.EnumObjects(OBJ_BRUSH, EnumObjectsProc, NULL);
}
```

C++ 编译器针对*CMyclass*::*enumIt* 实际做出来的码相当于：

```
void CMyclass::enumIt(CDC& dc)
{
  // 注册callback 函数
  CDC::EnumObjects(OBJ_BRUSH, EnumObjectsProc,
                   NULL, (CDC *)&dc);
}
```

你所看到的最后一个参数，(CDC *)&dc，其实就是*this* 指针。类别成员函数靠着*this*

指针才得以抓到正确对象的资料。你要知道，内存中只会有一份类别成员函数，但却可能有许多份类别成员变量--- 每个对象拥有一份。

C++ 以隐晦的*this* 指针指出正确的对象。当你这么做：

```
nCount = 0;
```

其实是：

```
this->nCount = 0;
```

基于相同的道理，上例中的*EnumObjectsProc* 既然是一个成员函数，C++ 编译器也会为它多准备一个隐藏参数。

好，问题就出在这个隐藏参数。callback 函数是给Windows 调用用的，Windows 并不经由任何对象调用这个函数，也就无由传递*this* 指针给callback 函数，于是导至堆栈中有一个随机变量会成为*this* 指针，而其结果当然是程序的崩溃了。

要把某个函数用作callback 函数，就必须告诉C++ 编译器，不要放*this* 指针作为该函数的最后一个参数。两个方法可以做到这一点：

1. 不要使用类别的成员函数（也就是说，要使用全域函数）做为callback 函数。
2. 使用static 成员函数。也就是在函数前面加上static 修饰词。

第一种作法相当于在C 语言中使用callback 函数。第二种作法比较接近OO 的精神。

我想更进一步提醒你的是，C++ 中的static 成员函数特性是，即使对象还没有产生，static 成员也已经存在（函数或变量都如此)。换句话说对象还没有产生之前你已经可以调用类别的static 函数或使用类别的static 变量了。请参阅第二章。

也就是说，凡声明为static 的东西（不管函数或变量）都并不和对象结合在一起，它们是类别的一部份，不属于对象。

空闲时间（idle time）的处理：OnIdle

为了让Hello 程序更具体而微地表现一个MFC 应用程序的水准，我打算为它加上空闲时间（idle time）的处理。

我已经在第 1 章介绍过了空闲时间，也简介了Win32 程序如何以*PeekMessage*「偷闲」。Microsoft 业已把这个观念落实到*CWinApp*（不，应该是*CWinThread*）中。请你回头看看本章的稍早的「*CWinApp*::*Run* - 程序生命的活水源头」一节，那一节已经揭露了MFC 消息循环的秘密：

```
int CWinThread::Run()
{
    ...
    for (;;)
    {
        while (bIdle &&
               !::PeekMessage(&m_msgCur, NULL, NULL, NULL, PM_NOREMOVE))
        {
            // call OnIdle while in bIdle state
            if (!OnIdle(lIdleCount++))
                bIdle = FALSE; // assume "no idle" state
        }
        ... // msg loop
    }
}
```

CThread::*OnIdle* 做些什么事情呢？*CWinApp* 改写了*OnIdle* 函数，*CWinApp*::*OnIdle* 又做些什么事情呢？你可以从THRDCORE.CPP 和APPCORE.CPP 中找到这两个函数的源代码，源代码可以说明一切。当然基本上我们可以猜测*OnIdle* 函数中大概是做一些系统（指的是MFC 本身）的维护工作。这一部份的功能可以说日趋式微，因为低优先权的执行线程可以替代其角色。

如果你的MFC 程序也想处理idle time，只要改写*CWinApp* 衍生类别的*OnIdle* 函数即可。这个函数的类型如下：

```
virtual BOOL OnIdle(LONG lCount);
```

lCount 是系统传进来的一个值，表示自从上次有消息进来，到现在，*OnIdle* 已经被调用了多少次。稍后我将改写Hello 程序，把这个值输出到窗口上，你就可以知道空闲时间是多么地频繁。*lCount* 会持续累增，直到*CWinThread*::*Run* 的消息循环又获得了一个讯息，此值才重置为0。

> 注意：Jeff Prosise 在他的Programming Windows 95 with MFC 一书第 7 章谈到*OnIdle* 函数时，曾经说过有几个消息并不会重置*lCount* 为0，包括鼠标消息、*WM_SYSTIMER*、*WM_PAINT*。不过根据我实测的结果，至少鼠标消息是会的。稍后你可在新版的Hello 程序移动鼠标，看看*lCount* 会不会重设为0。

我如何改写Hello 呢？下面是几个步骤：

1. 在*CMyWinApp* 中增加*OnIdle* 函数的声明：

```
class CMyWinApp : public CWinApp
{
public:
 virtual BOOL InitInstance(); // 每一个应用程序都应该改写此函数
 virtual BOOL OnIdle(LONG lCount); // OnIdle 用来处理空闲时间（idle time）
};
```

2. 在*CMyFrameWnd* 中增加一个*IdleTimeHandler* 函数声明。这么做是因为我希望在窗口中显示*lCount* 值， 所以最好的作法就是在*OnIdle* 中调用*CMyFrameWnd* 成员函数，这样才容易获得绘图所需的DC。

```
class CMyFrameWnd : public CFrameWnd
{
public:
   CMyFrameWnd();          // constructor
   afx_msg void OnPaint();   // for WM_PAINT
   afx_msg void OnAbout();   // for WM_COMMAND (IDM_ABOUT)
   void IdleTimeHandler(LONG lCount);  // we want it call by CMyWinApp::OnIdle
   ...
};
```

3. 在HELLO.CPP 中定义*CMyWinApp*::*OnIdle* 函数如下：

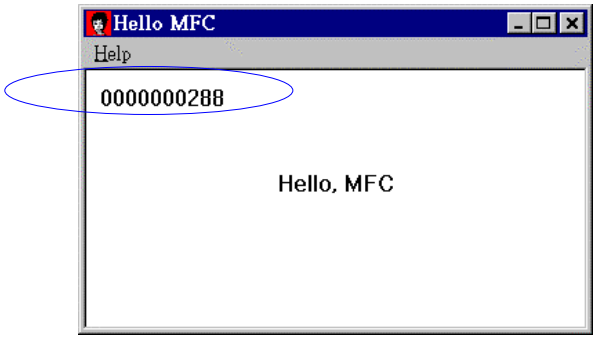
```
BOOL CMyWinApp::OnIdle(LONG lCount)
{
   CMyFrameWnd* pWnd = (CMyFrameWnd*)m_pMainWnd;
   pWnd->IdleTimeHandler(lCount);

   return TRUE;
}
```

4. 在HELLO.CPP 中定义*CMyFrameWnd*::*IdleTimeHandler* 函数如下：

```
void CMyFrameWnd::IdleTimeHandler(LONG lCount)
{
  CString str;
  CRect rect(10,10,200,30);
  CDC* pDC = new CClientDC(this);

  str.Format("%010d", lCount);
  pDC->DrawText(str, &rect, DT_LEFT | DT_TOP);
}
```

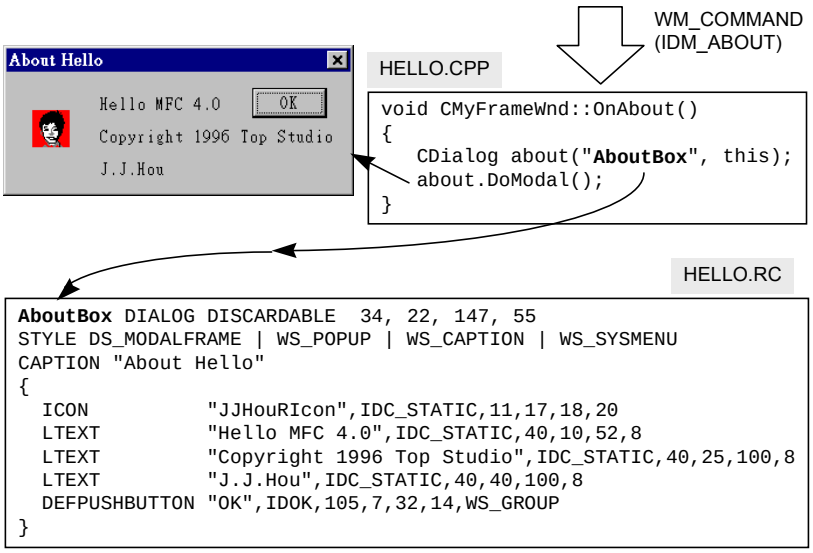
为了输出*lCount*，我又动用了三个MFC 类别：*CString*、*CRect* 和*CDC*。前两者非常简单，只是字符串与四方形结构的一层C++ 包装而且，后者是在Windows 系统中绘图所必须的DC（Device Context）的一个包装。

新版Hello 执行结果如下。左上角的*lCount* 以飞快的速度更迭。移动鼠标看看，看*lCount* 会不会重置为0。



Dialog 与 Control

回忆SDK 程序中的对话框作法：RC 文件中要准备一个对话框的Template，C 程序中要设计一个对话框函数。MFC 提供的*CDialog* 已经把对话框的窗口函数设计好了，因此在MFC 程序中使用对话框非常地简单：



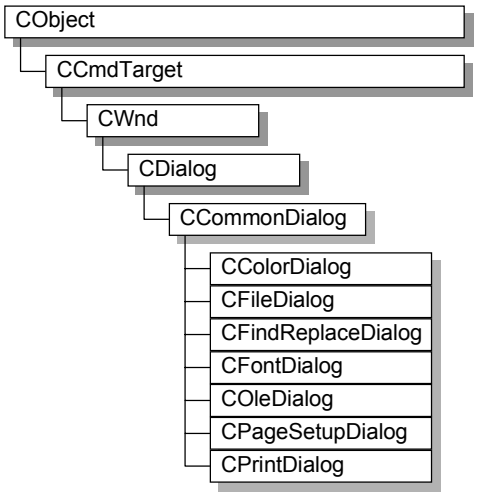
当使用者按下【File/About】菜单， 根据Message Map 的设定，*WM_COMMAND*（*IDM_ABOUT*）被送到*OnAbout* 函数去。我们首先在*OnAbout* 中产生一个*CDialog* 物件，名为*about*。*CDialog* 构造式容许两个参数，第一个参数是对话框的模板资源，第二个参数是*about* 对象的主人。由于我们的"About" 对话框是如此地简单，不需要改写*CDialog* 中的对话框函数，所以接下来直接调用*CDialog*::*DoModal*，对话框就开始运作了。

通用对话框（Common Dialogs）

有些对话框，例如【File Open】或【Save As】对话框，出现在每一个程序中的频率是如此之高，使微软公司不得不面对此一事实。于是，自从Windows 3.1 之后，Windows API 多了一组通用对话框（ Common Dialogs ） API 函数， 系统也多了一个对应的 COMMDLG.DLL（32 位版则为COMDLG32.DLL）。

MFC 也支持通用对话框，下面是其类别与其类型：

类别	类型
CCommonDialog	以下各类别的父类别
CFileDialog	File 对话框（Open 或Save As）
CPrintDialog	Print 对话框
CFindReplaceDialog	Find and Replace 对话框
CColorDialog	Color 对话框
CFontDialog	Font 对话框
CPageSetupDialog	Page Setup 对话框（MFC 4.0 新增）
COleDialog	Ole 相关对话框



在C/SDK 程序中，　使用通用对话框的方式是，　首先填充一块特定的结构如*OPENFILENAME*，然后调用API 函数如*GetOpenFileName*。当函数回返，结构中的某些字段便持有了使用者输入的值。

MFC 通用对话框类别，使用之简易性亦不输Windows API。下面这段码可以激活【Open】对话框并最后获得文件完整路径：

```
char szFileters[] = "Text fiels (*.txt)|*.txt|All files (*.*)|*.*||"
CFileDialog opendlg (TRUE, "txt", "*.txt",
                     OFN_FILEMUSTEXIST | OFN_HIDEREADONLY, szFilters, this);
if (opendlg.DoModal() == IDOK) {
         filename = opendlg.GetPathName();
}
```

opendlg 构造式的第一个参数被指定为*TRUE*，表示我们要的是一个【Open】对话框而不是【Save As】对话框。第二参数"txt" 指定预设扩展名；如果使用者输入的文件没有扩展名，就自动加上此一扩展名。第三个参数"*.txt" 出现在一开始的【file name】字段中。*OFN_* 参数指定文件的属性。第五个参数*szFilters* 指定使用者可以选择的文件型态，最后一个参数是父窗口。

当*DoModal* 回返，我们可以利用*CFileDialog* 的成员函数*GetPathName* 取得完整的档案路径。也可以使用另一个成员函数*GetFileName* 取其不含路径的文件名称，或*GetFileTitle* 取得既不含路径亦不含扩展名的文件名称。

这便是MFC 通用对话框类别的使用。你几乎不必再从其中衍生出子类别，直接用就好了。

本章回顾

乍看MFC 应用程序代码，实在很难推想程序的进行。一开始是一个衍生自*CWinApp* 的全域对象application object，然后是一个隐藏的*WinMain* 函数，调用application object 的*InitInstance* 函数，将程序初始化。初始化动作包括构造一个窗口对象（*CFrameWnd* 物件），而其构造式又调用*CFrameWnd*::*Create* 产生真正的窗口（并在产生之前要求MFC 注册窗口类别）。窗口产生后*WinMain* 又调用*Run* 激活消息循环，将*WM_COMMAND*（*IDM_ABOUT*）和*WM_PAINT* 分别交给成员函数*OnAbout* 和*OnPaint* 处理。

虽然刨根究底不易，但是我们都同意，MFC 应用程序代码的确比SDK 应用程序代码精简许多。事实上，MFC 并不打算让应用程序代码比较容易理解，毕竟raw Windows API 才是最直接了当的动作。许许多多细碎动作被包装在MFC 类别之中，降低了你写程序的负担，当然，这必须建立在一个事实之上：你永远可以改变MFC 的预设行为。这一点是无庸置疑的，因为所有你可能需要改变的性质，都被设计为MFC 类别中的虚拟函数了，你可以从MFC 衍生出自己的类别，并改写那些虚拟函数。

MFC 的好处在更精巧更复杂的应用程序中显露无遗。至于复杂如OLE 者，那就更是非MFC 不为功了。本章的Hello 程序还欠缺许多Windows 程序完整功能，但它毕竟是一个好起点，有点晦涩但不太难。下一章范例将运用MDI、Document/View、各式各样的UI 对象...。

第７章

简单而完整：MFC 骨干程序

当技术愈来愈复杂，
入门愈来愈困难，
我们的困惑愈来愈深，
犹豫愈来愈多。

上一章的Hello 范例，对于MFC 程序设计导入很适合。但它只发挥了MFC 的一小部份特性，只用了三个MFC 类别（*CWinApp*、*CFrameWnd* 和*CDialog*）。这一章我们要看一个完整的MFC 应用程序骨干（注），其中包括丰富的UI 对象（如工具栏、状态列）的生成，以及很重要的Document/View 架构观念。

> 注：我所谓的MFC 应用程序骨干，指的是由AppWizard 产生出来的MFC 程序，也就是像第４章所产生的Scribble step0 那样的程序。

不二法门：熟记MFC 类别阶层架构

我还是要重复这一句话：MFC 程序设计的第一要务是熟记各类别的阶层架构，并清楚了解其中几个一定会用到的类别。一个MFC 骨干程序（不含ODBC 或OLE 支持）运用到的类别如图7-1 所示，请与图6-1 做个比较。

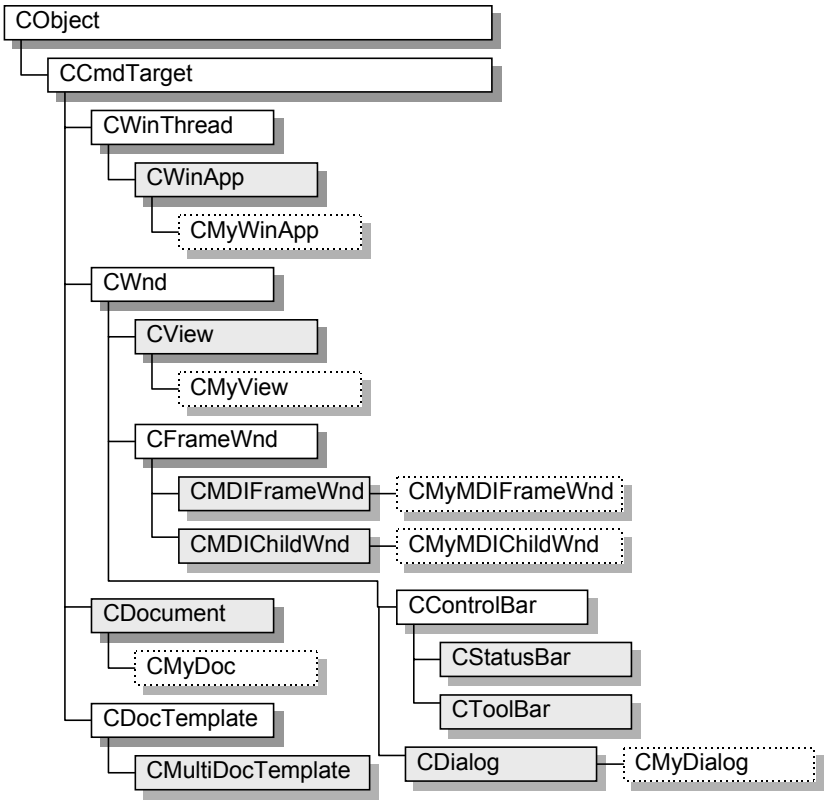


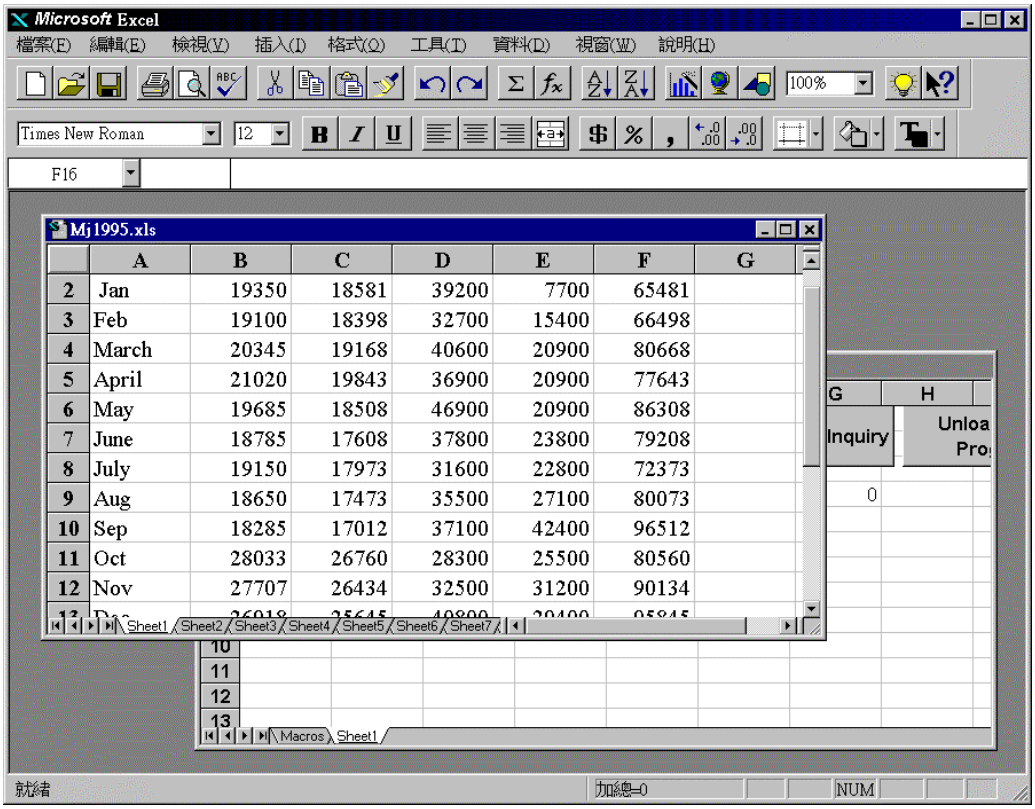
图7-1 本章范例程序所使用的MFC 类别。请与图6-1 做比较。

MFC 程序的UI 新风貌

一套好软件少不得一幅漂亮的使用者接口。图7-2 是信手拈来的几个知名Windows 软体，它们一致具备了工具栏和状态列等视觉对象，并拥有MDI 风格。利用MFC，我们很轻易就能够做出同等级的UI 接口。

第 7 章简单而完整：MFC 骨干程序

图7-2a Microsoft Word for Windows，允许使用者同时编辑多份文件，每一份文件就是所谓的document，这些document 窗口總不会脱Word 主窗口的管辖。



	A	B	C	D	E	F	G
2	Jan	19350	18581	39200	7700	65481	
3	Feb	19100	18398	32700	15400	66498	
4	March	20345	19168	40600	20900	80668	
5	April	21020	19843	36900	20900	77643	
6	May	19685	18508	46900	20900	86308	
7	June	18785	17608	37800	23800	79208	
8	July	19150	17973	31600	22800	72373	
9	Aug	18650	17473	35500	27100	80073	
10	Sep	18285	17012	37100	42400	96512	
11	Oct	28033	26760	28300	25500	80560	
12	Nov	27707	26434	32500	31200	90134	

图7-2b Microsoft Excel，允许同时制作多份报表。每一份报表就是一份 document。

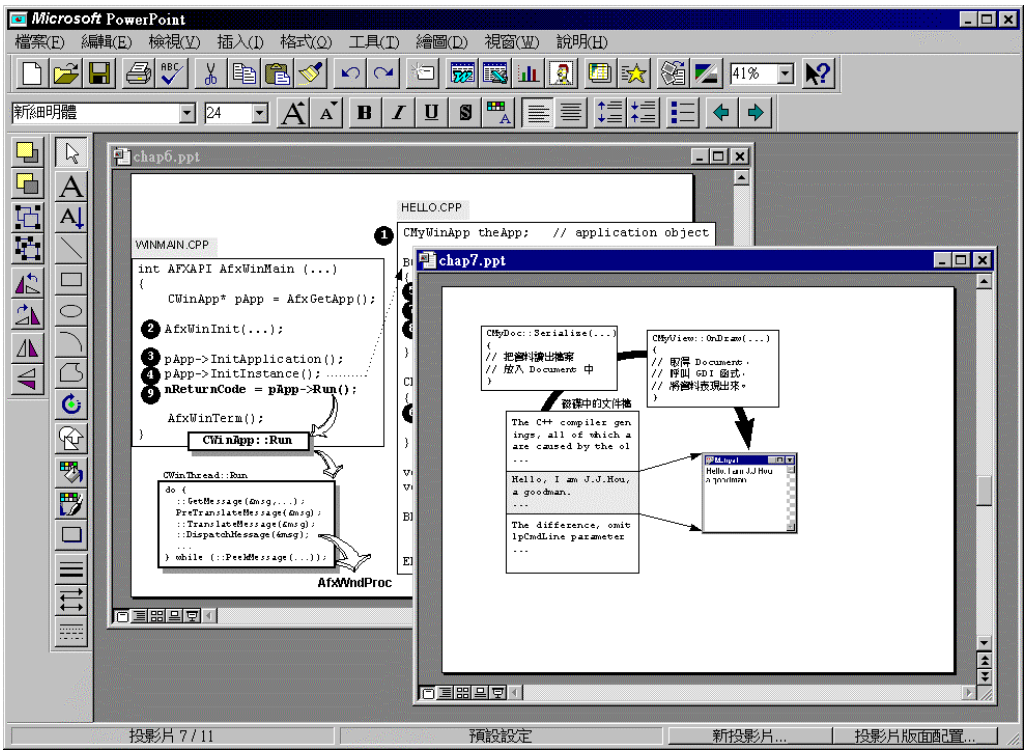
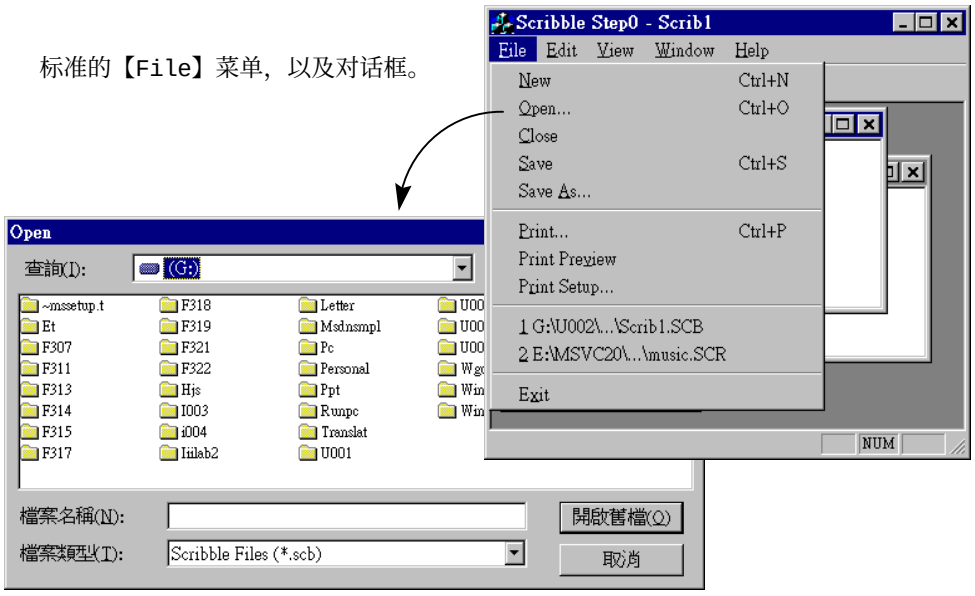


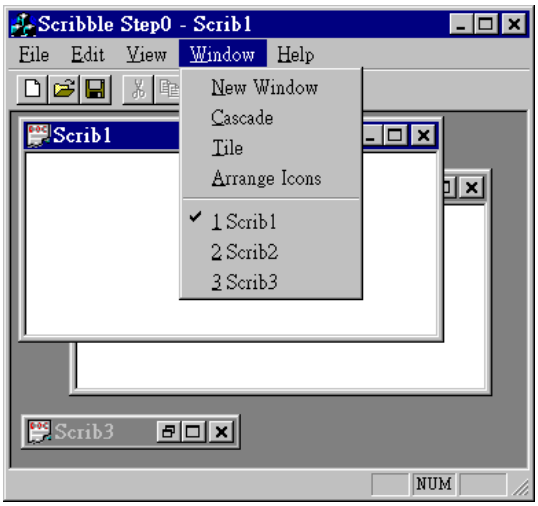
图7-2c Microsoft PowerPoint 允许同时制作多份演示文稿资料，每一份演示文稿就是一份document

撰写MFC 程序，我们一定要放弃传统的「纯手工打造」方式，改用Visual C++ 提供的各种开发工具。AppWizard 可以为我们制作出MFC 程序骨干；只要选择某些按钮，不费吹灰之力你就可以获得一个很漂亮的程序。这个全自动生产线做出来的程序虽不具备任何特殊功能（那正是我们程序员的任务），但已经拥有以下的特征：

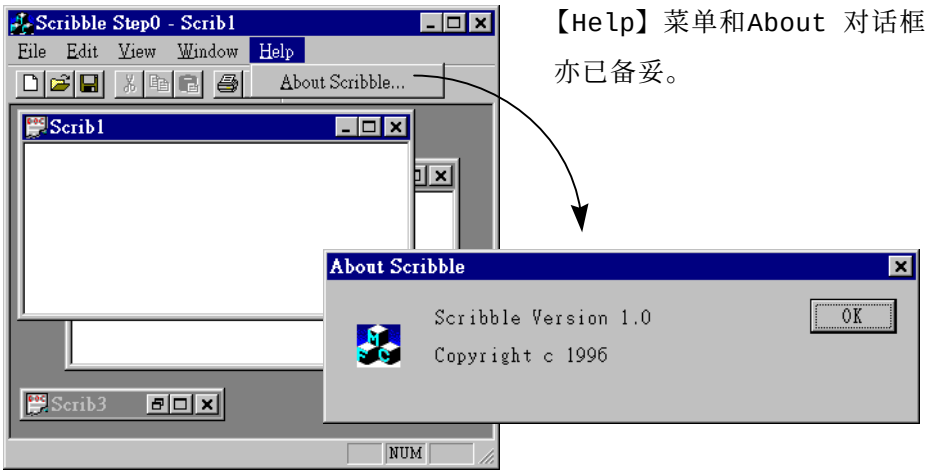


標準的【File】菜單，以及对话框。

標準的【Edit】菜单（剪贴簿功能）。这份菜单是否一开始就有功效，必须视你选用哪一种View 而定，例如CEditView 就内建有剪贴簿功能。



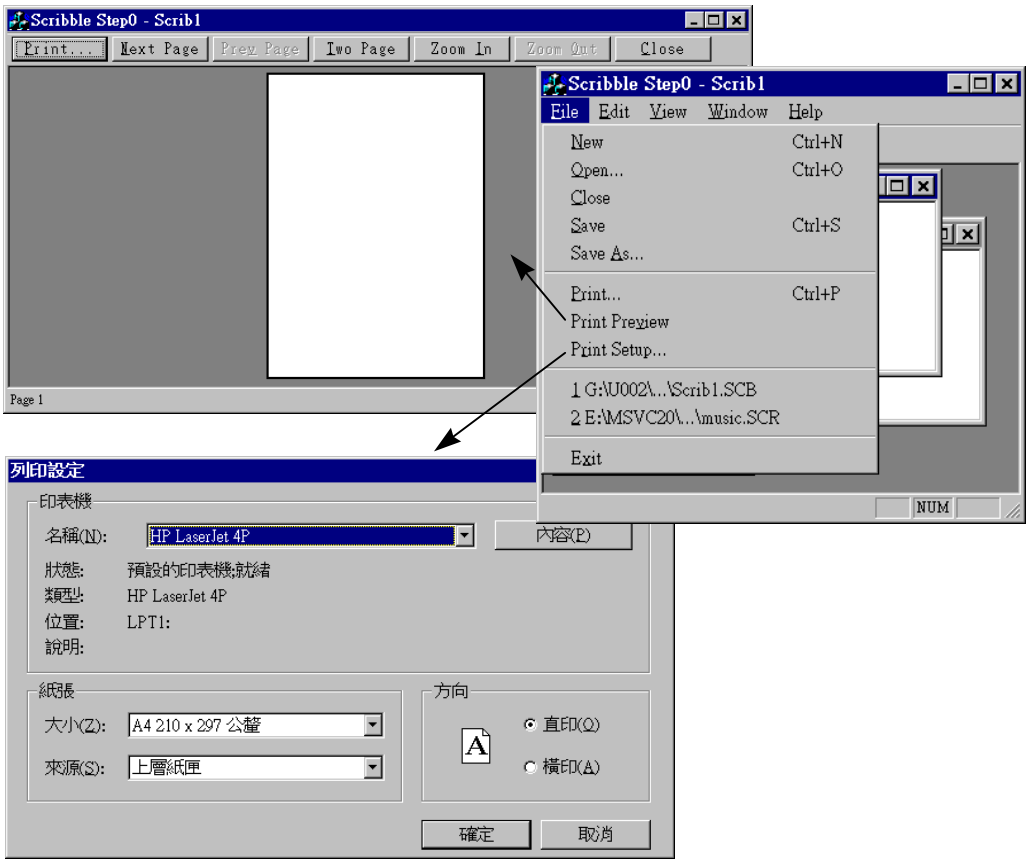
标准MDI 程序应该具备的【Window】菜单



【Help】菜单和About 对话框亦已备妥。

此外，标准的工具栏和状态列也已备妥，并与菜单内容建立起映射关系。所谓工具栏，是将某几个常用的菜单项目以按钮型式呈现出来，有一点热键的味道。这个工具栏可以随处停驻（dockable）。所谓状态列，是主窗口最下方的文字显示区；只要菜单拉下，状态列就会显示鼠标座落的菜单项目的说明文字。状态列右侧有三个小窗口（可扩充个数），用来显示一些特殊按键的状态。

打印与预视功能也已是半成品。【File】菜单拉下来可以看到【Print...】和【Print Preview】两项目：



骨干程序的Document 和View 目前都还是白纸一张，需要我们加工，所以一开始看不出打印与预视的真正功能。但如果我们在AppWizard 中选用的View 类别是*CEditView*（如同第 4 章292 页），使用者就可以打印其编辑成果，并可以在打印之前预视。也就是说，一进程序代码都不必写，我们就获得了一个可以同时编辑多份文件的文字编辑软件。

Document/View 支撑你的应用程序

我已经多次强调，Document/View 是MFC 进化为Application Framework 的灵魂。这个特征表现于程序设计技术上远多于表现在使用者接口上，因此使用者可能感觉不到什么是Document/View。程序员呢？程序员将因陌生而有一段阵痛期，然后开始享受它带来的便利。

我们在OLE 中看到各对象（注）的集合称为一份Document；在MDI 中看到子窗口所掌握的资料称为一个Document；现在在MFC 又看到Document。"Document" 如今处处可见，再过不多久八成也要和"Object" 一样地泛滥了。

> OLE 对象指的是PaintBrush 完成的一张bitmap、SoundRecorder 完成的一段Wave 声音、Excel 完成的一份电子表格、Word 完成的一份文字等等等。为了恐怕与C++ 的「对象」混淆，有些书籍将OLE object 称为OLE item。

在MFC 之中，你可以把Document 简单想作是「资料」。是的，只是资料，那么MFC 的*CDocument* 简单地说就是负责处理资料的类别。

问题是，一个预先写好的类别怎么可能管理未知的资料呢？MFC 设计之际那些伟大的天才们并不知道我们的数据结构，不是吗?! 他怎么知道我的程序要处理的资料是简单如：

```
char  name[20];
char  address[30];
int   age;
bool  sex;
```

或是复杂如：

```
struct dbllistnode
      {
        struct dbllistnode *next, *prev;
        struct info_t
              {
               int left;
               int top;
```

```
            int width;
            int height;
            void (*cursor)();
           } *item;
};
```

的确，预先处理未知的资料根本是不可能的。*CDocument* 只是把空壳做好，等君入瓮。它可以内嵌其它对象（用来处理基层数据类型如串行、数组等等），所以程序员可以在 Document 中拼拼凑凑出实际想要表达的文件完整格式。下一章进入Scribble 程序的实际设计时，你就能够感受这一点。

CDocument 的另一价值在于它搭配了另一个重要的类别：*CView*。

不论什么型式，数据总是有体有面。实际的资料数值就是体，显示在屏幕上（甚而打印机上）的画面就是面（图7-3a）。「数值的处理」应该使用字节、整数、浮点数、串列、数组等数据结构，而「数值的表现」应该使用绘图工具如坐标系统、笔刷颜色、点线圆弧、字形...。*CView* 就是为了资料的表现而设计的。

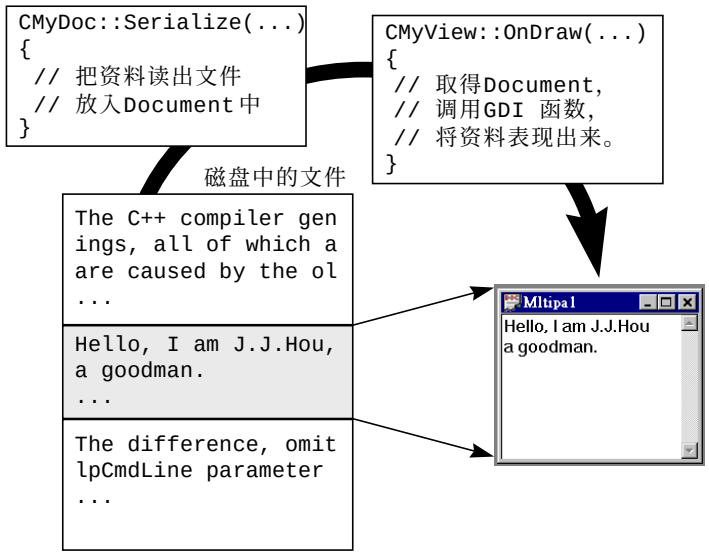


图7-3a Document 是资料的体， View 是资料的面。

}

除了负责显示，View 还负责程序与使用者之间的交谈接口。使用者对资料的编辑、修改都需仰赖窗口上的鼠标与键盘动作才得完成，这些消息都将由View 接受后再通知 Document (图7-3b)。

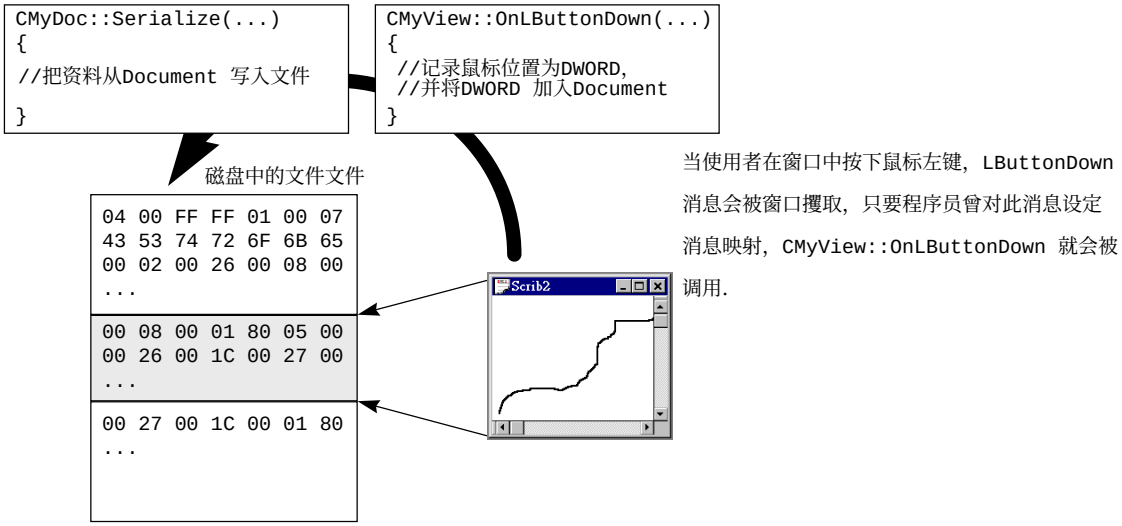


图7-3b View 是Document 的第一线， 负责与使用者接触。

Document/View 的价值在于，这些MFC 类别已经把一个应用程序所需的「数据处理与显示」的函数空壳都设计好了，这些函数都是虚拟函数，所以你可以（也应该）在衍生类别中改写它们。有关文件读写的动作在*CDocument* 的*Serialize* 函数进行，有关画面显示的动作在*CView* 的*OnDraw* 或*OnPaint* 函数进行。当我为自己衍生两个类别*CMyDoc* 和*CMyView*，我只要把全付心思花在*CMyDoc*::*Serialize* 和*CMyView*::*OnDraw* 身上，其它琐事一概不必管，整个程序自动会运作得好好的。

什么叫做「整个程序会自动运作良好」？以下是三个例子：

- 如果按下【File/Open】，Application Framework 会激活对话框让你指定文件名，然后自动调用*CMyDoc*::*Serialize* 读档。Application Framework 还会调用 *CMyView*::*OnDraw*，把资料显示出来。
- 如果屏幕状态改变，产生了*WM_PAINT*，Framework 会自动调用你的 *CMyView*::*OnDraw*，传一个Display DC 让你重新绘制窗口内容。
- 如果按下【File/Print...】，Framework 会自动调用你的*CMyView*::*OnDraw*，这次传进去的是个Printer DC，因此绘图动作的输出对象就成了打印机。

MFC 已经把程序大架构完成了，模块与模块间的消息流动路径以及各函数的功能职司都已确定好（这是MFC 之所以够格称为一个Framework 的原因），所以我们写程序的焦点就放在那些必须改写的虚拟函数身上即可。软件界当初发展GUI 系统时，目的也是希望把程序员的心力导引到应用软件的真正目标去，而不必花在使用者接口上。MFC 的 Document/View 架构希望更把程序员的心力导引到真正的数据结构设计以及真正的数据显示动作上，而不要花在模块的沟通或消息的流动传递上。今天，程序员都对GUI 称便，Document/View 也即将广泛地证明它的贡献。

Application Framework 使我们的程序写作犹如做填充题；Visual C++ 的软件开发工具则使我们的程序写作犹如做选择题。我们先做选择题，再在骨干程序中做填充题。的确，程序员的生活愈来愈像侯捷所言「只是软件IC 装配厂里的男工女工」了。

现在让我们展开MFC 深度之旅，彻底把MFC 骨干程序的每一行都搞清楚。你应该已经从上一章具体了解了MFC 程序从激活到结束的生命过程，这一章的例子虽然比较复杂，程序的生命过程是一样的。我们看看新添了什么内容，以及它们如何运作。我将以 AppWizard 完成的Scribble Step0（第 4 章）为解说对象，一行不改。然后我会做一点点修改，使它成为一个多窗口文字编辑器。

利用Visual C++ 工具完成Scribble step0

我已经在第 4 章示范过AppWizard 的使用方法，并实际制作出Scribble Step0 程序，这里就不再重复说明了。完整的骨干程序源代码亦已列于第 4 章。

这些由「生产线」做出来的程序代码其实对初学者并不十分合适，原因之一是容易眼花撩乱，有许多`#if...#endif`、批注、奇奇怪怪的符号（例如`//{` 和`//}`）；原因之二是每一个类别有自己的.H 档和.CPP 文件，整个程序因而幅员辽阔（六个.CPP 档和六个.H 档）。

图7-4 是Scribble step0 程序中各类别的相关资料。

类别名称	基础类别	类别声明于	类别定义于
CScribbleApp	*CWinApp*	Scribble.h	Scribble.cpp
CMainFrame	*CMDIFrameWnd*	Mainfrm.h	Mainfrm.cpp
CChildFrame	*CMDIChildWnd*	Childfrm.h	Childfrm.cpp
CScribbleDoc	*CDocument*	ScribbleDoc.h	ScribbleDoc.cpp
CScribbleView	*CView*	ScribbleView.h	ScribbleView.cpp
CAboutDlg	*CDialog*	Scribble.cpp	Scribble.cpp

图7-4 Scribble 骨干程序中的重要组成份子

骨干程序使用哪些MFC 类别？

对，你看到的Scribble step0 就是一个完整的MFC 应用程序，而我保证你一定昏头转向茫无头线程。没有关系，我们才刚启航。

如果把标准图形接口（工具栏和状态列）以及Document/View 考虑在内，一个标准的MFC MDI 程序使用这些类别：

MFC 类别名称	我的类别名称	功能
CWinApp	*CScribbleApp*	application object
CMDIFrameWnd	*CMainFrame*	MDI 主窗口
CMultiDocTemplate	直接使用	管理 Document/View
CDocument	*CScribbleDoc*	Document，负责数据结构与文件动作
CView	*CScribbleView*	View，负责资料的显示与打印
CMDIChildWnd	*CChildFrame*	MDI 子窗口
CToolBar	直接使用	工具栏
CStatusBar	直接使用	状态列
CDialog	*CAboutDlg*	About 对话框

应用程序各显身手的地方只是各个可被改写的虚拟函数。这九个类别在MFC 的地位请看图7-1。下一节开始我会逐项解释每一个对象的产生时机及其重要性质。

Document/View 不只应用在MDI 程序，也应用在SDI 程序上。你可以在AppWizard 的「Options 对话框」（图4-2b）选择SDI 风格。本书以MDI 程序为讨论对象。

为了对标准的MFC 程序有一个大局观，图7-4 显示Scribble step0 中各重要组成份子（类别），这些组成份子在执行时期的意义与主从关系显示于图7-5。

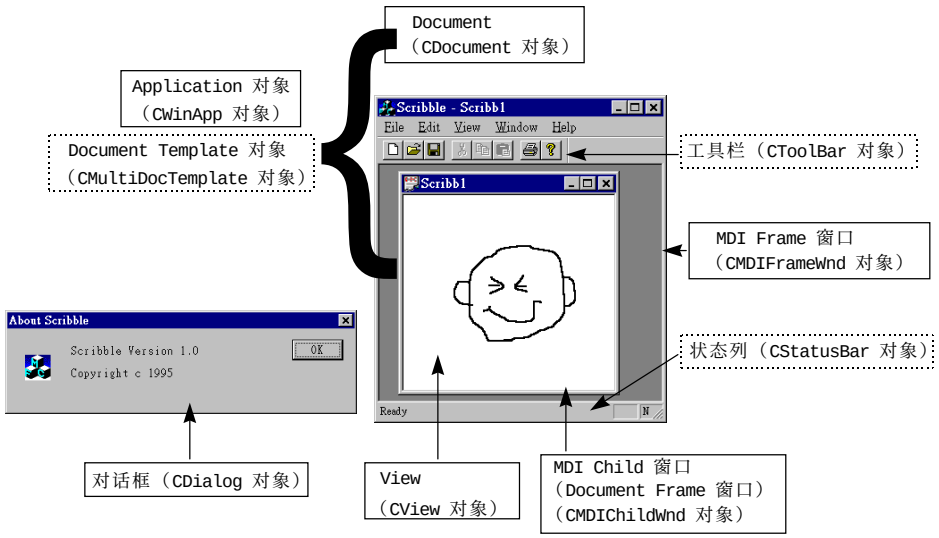


图7-5 Scribble step0 程序中的九个对象（几乎每个MFC MDI 程序都如此）。

图7-6 是Scribble step0 程序缩影，我把执行时序标上去，对于整体概念的形成将有帮助。

```
#0001 class CScribbleApp : public CWinApp
#0002 {
#0003    virtual BOOL InitInstance();   //注意：第６章的HelloMFC 程序是在
#0004    afx_msg void OnAppAbout();     //类别中处理"About" 命令，这里的
#0005    DECLARE_MESSAGE_MAP()          //程序却在CWinApp 衍生类别中处理之。到底，
#0006 };                                //一个消息可以（或应该）在哪里被处理才是合理？
#0007                                   // 第９章「消息映射与命令绕行」可以解决这个疑惑。
#0008 class CMainFrame : public CMDIFrameWnd
#0009 {
#0010    DECLARE_DYNAMIC(CMainFrame)
#0011    CStatusBar  m_wndStatusBar;
#0012    CToolBar    m_wndToolBar;
#0013    afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
#0014    DECLARE_MESSAGE_MAP()
#0015 };
#0016
#0017 class CChildFrame : public CMDIChildWnd
#0018 {
```

```
#0019    DECLARE_DYNCREATE(CChildFrame)
#0020    DECLARE_MESSAGE_MAP()
#0021 };
#0022
#0023 class CScribbleDoc : public CDocument
#0024 {
#0025    DECLARE_DYNCREATE(CScribbleDoc)
#0026   virtual void Serialize(CArchive& ar);
#0027    DECLARE_MESSAGE_MAP()
#0028 };
#0029
#0030 class CScribbleView : public CView
#0031 {
#0032    DECLARE_DYNCREATE(CScribbleView)
#0033    CScribbleDoc* GetDocument();
#0034
#0035    virtual void OnDraw(CDC* pDC);
#0036   DECLARE_MESSAGE_MAP()
#0037 };
#0038
#0039 class CAboutDlg : public CDialog
#0040 {
#0041       DECLARE_MESSAGE_MAP()
#0042 };
#0043
#0044 //----------------------------------
#0045
#0046 CScribbleApp theApp; ①
#0047
#0048 BEGIN_MESSAGE_MAP(CScribbleApp, CWinApp)
#0049    ON_COMMAND(ID_APP_ABOUT, OnAppAbout) ❹
#0050    ON_COMMAND(ID_FILE_NEW, CWinApp::OnFileNew)
#0051    ON_COMMAND(ID_FILE_OPEN, CWinApp::OnFileOpen) ❷
#0052    ON_COMMAND(ID_FILE_PRINT_SETUP, CWinApp::OnFilePrintSetup)
#0053 END_MESSAGE_MAP()
#0054
#0055 BOOL CScribbleApp::InitInstance() ⑤
#0056 {
#0057    ...
#0058    CMultiDocTemplate* pDocTemplate;
#0059    pDocTemplate = new CMultiDocTemplate( ⑥
#0060           IDR_SCRIBTYPE,
#0061           RUNTIME_CLASS(CScribbleDoc),
#0062           RUNTIME_CLASS(CChildFrame),
#0063           RUNTIME_CLASS(CScribbleView));
#0064    AddDocTemplate(pDocTemplate);
```

```
// MFC 內部
Int AfxAPI AfxWinMain(¡K
{
  CWinApp* pApp = AfxGetApp();
  AfxWinInit(¡K;                    ②
  pApp->InitApplication();          ③
  pApp->InitInstance();             ④
  nReturnCode = pApp->Run();        ❶
```

```
#0065
#0066    CMainFrame* pMainFrame = new CMainFrame;  ⑦
#0067     pMainFrame->LoadFrame(IDR_MAINFRAME);    ⑧
#0068     m_pMainWnd = pMainFrame;
#0069     ¡ K
#0070     pMainFrame->ShowWindow(m_nCmdShow);  ⑩
#0071     pMainFrame->UpdateWindow();
#0072     return TRUE;
#0073 }
#0074
#0075 BEGIN_MESSAGE_MAP(CAboutDlg, CDialog)
#0076 END_MESSAGE_MAP()
#0077
#0078 void CScribbleApp::OnAppAbout()  ❺
#0079 {
#0080     CAboutDlg aboutDlg;
#0081     aboutDlg.DoModal();
#0082 }
#0083
#0084 IMPLEMENT_DYNAMIC(CMainFrame, CMDIFrameWnd)
#0085
#0086 BEGIN_MESSAGE_MAP(CMainFrame, CMDIFrameWnd)
#0087     ON_WM_CREATE()
#0088 END_MESSAGE_MAP()
#0089
#0090 static UINT indicators[] =
#0091 {
#0092     ID_SEPARATOR,
#0093     ID_INDICATOR_CAPS,
#0094     ID_INDICATOR_NUM,
#0095     ID_INDICATOR_SCRL,
#0096 };
#0097
#0098 int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)  ⑨
#0099 {
#0100     m_wndToolBar.Create(this);
#0101     m_wndToolBar.LoadToolBar(IDR_MAINFRAME);
#0102     m_wndStatusBar.Create(this);
#0103     m_wndStatusBar.SetIndicators(indicators,
#0104              sizeof(indicators)/sizeof(UINT));
#0105
#0106     m_wndToolBar.SetBarStyle(m_wndToolBar.GetBarStyle() |
#0107            CBRS_TOOLTIPS | CBRS_FLYBY | CBRS_SIZE_DYNAMIC);
#0108
#0109     m_wndToolBar.EnableDocking(CBRS_ALIGN_ANY);
```

```
// MFC 內部
CFrameWnd::Create
        ▼
CWnd::CreateEx
        ▼
::CreateWindowEx
```

WM_CREATE

```
#0110    EnableDocking(CBRS_ALIGN_ANY);
#0111    DockControlBar(&m_wndToolBar);
#0112    return 0;
#0113 }
#0114
#0115 IMPLEMENT_DYNCREATE(CChildFrame, CMDIChildWnd)
#0116
#0117 BEGIN_MESSAGE_MAP(CChildFrame, CMDIChildWnd)
#0118 END_MESSAGE_MAP()
#0119
#0120 IMPLEMENT_DYNCREATE(CScribbleDoc, CDocument)
#0121
#0122 BEGIN_MESSAGE_MAP(CScribbleDoc, CDocument)
#0123 END_MESSAGE_MAP()
#0124
#0125 void CScribbleDoc::Serialize(CArchive& ar)  ❸
#0126 {
#0127    if (ar.IsStoring())
#0128    {
#0129           // TODO: add storing code here
#0130    }
#0131    else
#0132    {
#0133           // TODO: add loading code here
#0134    }
#0135 }
#0136
#0137 IMPLEMENT_DYNCREATE(CScribbleView, CView)
#0138
#0139 BEGIN_MESSAGE_MAP(CScribbleView, CView)
#0140    ON_COMMAND(ID_FILE_PRINT, CView::OnFilePrint)
#0141    ON_COMMAND(ID_FILE_PRINT_DIRECT, CView::OnFilePrint)
#0142    ON_COMMAND(ID_FILE_PRINT_PREVIEW, CView::OnFilePrintPreview)
#0143 END_MESSAGE_MAP()
#0144
#0145 void CScribbleView::OnDraw(CDC* pDC)
#0146 {
#0147    CScribbleDoc* pDoc = GetDocument();
#0148    // TODO: add draw code for native data here
#0149 }
```

图7-6 Scribble step0 执行时序。这是一张简图，有一些次要动作（例如鼠标拉曳功能、设定对话框底色）并未列出，但是在稍后的细部讨论中会提到。

以下是图7-6 程序流程之说明：

①~④ 动作与流程和前一章的Hello 程序如出一辙。

⑤ 我们改写*InitInstance* 这个虚拟函数。

⑥ *new* 一个*CMultiDocTemplate* 对象，此对象规划Document、View 以及Document Frame 窗口三者之关系。

⑦ *new* 一个*CMyMDIFrameWnd* 对象，做为主窗口对象。

⑧ 调用*LoadFrame*，产生主窗口并加挂菜单等诸元，并指定窗口标题、文件标题、文件档扩展名等（关键在IDR_MAINFRAME 常数）。*LoadFrame* 内部将调用*Create*，后者将调用*CreateWindowEx*，于是触发*WM_CREATE* 消息。

⑨ 由于我们曾于*CMainFrame* 之中拦截*WM_CREATE*（利用*ON_WM_CREATE* 宏），所以*WM_CREATE* 产生之际Framework 会调用*OnCreate*。我们在此为主窗口挂上工具列和状态列。

⑩ 回到*InitInstance*，执行*ShowWindow* 显示窗口。

❶ *InitInstance* 结束，回到*AfxWinMain*，执行*Run*，进入消息循环。其间的黑盒子已在上一章的Hello 范例中挖掘过。

❷ 消息经由Message Routing 机制，在各类别的Message Map 中寻求其处理例程。*WM_COMMAND*/*ID_FILE_OPEN* 消息将由*CWinApp*::*OnFileOpen* 函数处理。此函数由MFC 提供，它在显示过【File Open】对话框后调用*Serialize* 函数。

❸ 我们改写*Serialize* 函数以进行我们自己的文件读写动作。

❹ *WM_COMMAND*/*ID_APP_ABOUT* 消息将由*OnAppAbout* 函数处理。

❺ *OnAppAbout* 函数利用*CDialog* 的性质很方便地产生一个对话框。

Document Template 的意义

Document Template 是一个全新的观念。

稍早我已提过Document/View 的概念，它们互为表里。View 本身虽然已经是一个窗口，其外围却必须再包装一个外框窗口做为舞台。这样的切割其实是为了让View 可以非常独立地放置于「MDI Document Frame 窗口」或「SDI Document Frame 窗口」或「OLE Document Frame 窗口」等各种应用之中。也可以说，Document Frame 窗口是View 窗口的一个容器。资料的内容、资料的表象、以及「容纳资料表象之外框窗口」三者是一体的，换言之，程序每打开一份文件（资料），就应该产生三份对象：

1. 一份Document 对象,
2. 一份View 对象,
3. 一份*CMDIChildWnd* 对象（做为外框窗口）

这三份对象由一个所谓的Document Template 对象来管理。让这三份对象产生关系的关键在于*CMultiDocTemplate*：

```
BOOL CScribbleApp::InitInstance()
{
    ...
    CMultiDocTemplate* pDocTemplate;
    pDocTemplate = new CMultiDocTemplate(
           IDR_SCRIBTYPE,
           RUNTIME_CLASS(CScribbleDoc),
           RUNTIME_CLASS(CChildFrame),
           RUNTIME_CLASS(CScribbleView));
    AddDocTemplate(pDocTemplate);
    ...
}
```

如果程序支持不同的资料格式（例如一为TEXT 一为BITMAP），那么就需要不同的Document Template：

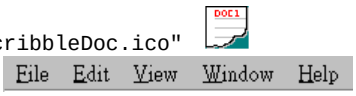
```
BOOL CMyWinApp::InitInstance()
{
  ...
  CMultiDocTemplate* pDocTemplate;

  pDocTemplate = new CMultiDocTemplate(
          IDR_TEXTTYPE,
          RUNTIME_CLASS(CTextDoc),
          RUNTIME_CLASS(CChildFrame),
          RUNTIME_CLASS(CTextView));
  AddDocTemplate(pDocTemplate);

  pDocTemplate = new CMultiDocTemplate(
          IDR_BMPTYPE,
          RUNTIME_CLASS(CBmpDoc),
          RUNTIME_CLASS(CChildFrame),
          RUNTIME_CLASS(CBmpView));
  AddDocTemplate(pDocTemplate);
  ...
}
```

这其中有许多值得讨论的地方，而*CMultiDocTemplate* 的构造式参数透露了一些端倪：

```
CMultiDocTemplate::CMultiDocTemplate(UINT nIDResource,
                                     CRuntimeClass* pDocClass,
                                     CRuntimeClass* pFrameClass,
                                     CRuntimeClass* pViewClass);
```

1. *nIDResource*：这是一个资源ID，表示此一文件类型（文件格式）所使用的资源。本例为*IDR_SCRIBTYPE*，在RC 档中代表多种资源（不同种类的资源可使用相同的ID）：

```
IDR_SCRIBTYPE ICON DISCARDABLE  "res\\ScribbleDoc.ico"
IDR_SCRIBTYPE MENU PRELOAD DISCARDABLE
{ ... }
STRINGTABLE PRELOAD DISCARDABLE
BEGIN
   IDR_MAINFRAME "Scribble Step0"
   IDR_SCRIBTYPE "\nScrib\nScrib\nScribble Files (*.scb)\n.SCB\n
                  Scribble.Document\nScrib Document"
END
```



原码太长,我把它截为两半,实际上是一整行.有七个子字符串各以'\n'分隔.这些子字符串完整描述文件的类型.第一个子字符串于MDI程序中用不着，故本例省略（成为空字符串）。

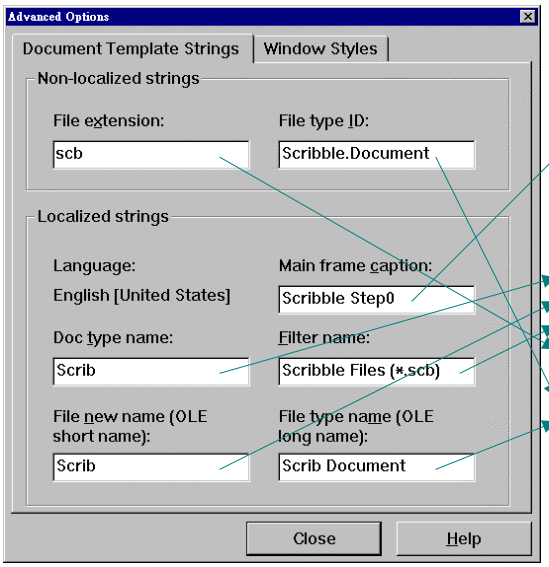
其中的ICON 是文件窗口被最小化之后的图标；MENU 是当程序存在有任何文件窗口时所使用的菜单（如果没有开启任何文件窗口，菜单将是另外一套，稍后再述）。至于字符串表格（STRINGTABLE）中的字符串，稍后我有更进一步的说明。

2. *pDocClass*。这是一个指针，指向Document 类别（ 衍生自*CDocument*）之「*CRuntimeClass* 对象」。
3. *pFrameClass*。这是一个指针，指向Child Frame 类别（衍生自*CMDIChildWnd*）之「*CRuntimeClass* 对象」。
4. *pViewClass*。这是一个指针，指向View 类别（衍生自*CView*）之「*CRuntimeClass* 对象」。

CRuntimeClass
我曾经在第 3 章「自制RTTI」一节解释过什么是*CRuntimeClass*。它就是「类别型录网」串行中的元素类型。任何一个类别只要在声明时使用*DECLARE_DYNAMIC* 或*DECLARE_DYNCREATE* 或*DECLARE_SERIAL* 宏，就会拥有一个静态的（static）*CRuntimeClass* 内嵌对象。

好，你看，Document Template 接受了三种类别的*CRuntimeClass* 指针，于是每当使用者打开一份文件，Document Template 就能够根据「类别型录网」（第 3 章所述），动态生成出三个对象（document、view、document frame window）。如果你不记得MFC 的动态生成是怎么一回事儿，现在正是复习第 3 章的时候。我将在第 8 章带你实际看看Document Template 的内部动作。

前面曾提到，我们在*CMultiDocTemplate* 构造式的第一个参数置入*IDR_SCRIBTYPE*，代表RC 文件中的菜单（MENU）、图标（ICON）、字符串（STRING）三种资源，其中又以字符串资源大有学问。这个字符串以'\n' 分隔为七个子字符串，用以完整描述文件类型。七个子字符串可以在AppWizard 的步骤四的【Advanced Options】对话框中指定：



RC 文件中的字符串资源

IDR_MAINFRAME：

```
"Scribble Step0"
```

IDR_SCRIBTYPE（7 个子字串）：

```
\n
Scrib\n
Scrib\n
Scribble Files (*.scb)\n
.SCB\n
Scribble.Document\n
Scrib Document
```

每一个子字符串都可以在程序进行过程中取得，只要调用*CDocTemplate*::*GetDocString* 并在其第二参数中指定索引值（1~7）即可，但最好是以*CDocTemplate* 所定义的七个常数代替没有字面意义的索引值。下面就是*CDocTemplate* 的 7 个常数定义：

```
// in AFXWIN.H
class CDocTemplate : public CCmdTarget
{
  ...
  enum DocStringIndex
  {
    windowTitle,  // default window title
    docName,      // user visible name for default document
    fileNewName,  // user visible name for FileNew

    // for file based documents:
    filterName,   // user visible name for FileOpen
    filterExt,    // user visible extension for FileOpen

    // for file based documents with Shell open support:
    regFileTypeId,   // REGEDIT visible registered file type identifier
    regFileTypeName  // Shell visible registered file type name
  };
  ...
};
```

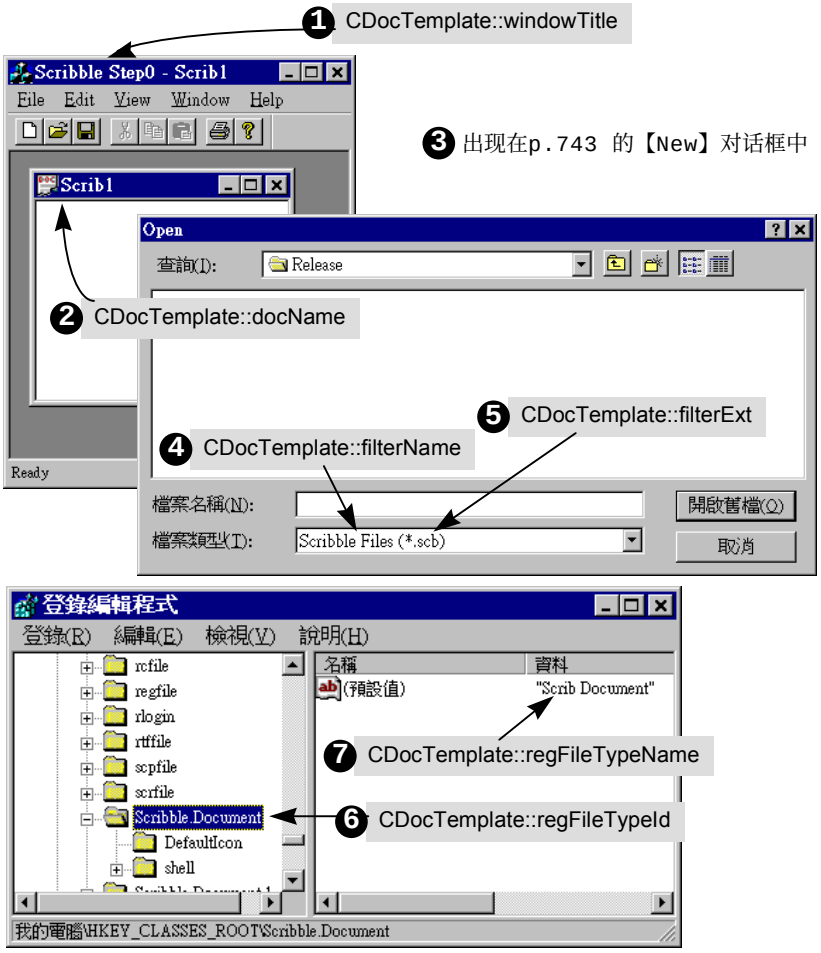
所以，你可以这么做：

```
CString strDefExt, strDocName;
pDocTemplate->GetDocString(strDefExt, CDocTemplate::filterExt);
pDocTemplate->GetDocString(strDocName, CDocTemplate::docName);
```

七个子字符串意义如下：

index	意义
1. *CDocTemplate::windowTitle*	主窗口标题栏上的字符串。SDI 程序才需要指定它，MDI 程式不需要指定，将以*IDR_MAINFRAME* 字符串为默认值。
2. *CDocTemplate::docName*	文件基底名称（本例为"Scrib"）。这个名称再加上一个流水号码，即成为新文件的名称（例如"Scrib1"）。如果此字符串未被指定，文件预设名称为"Untitled"。
3. *CDocTemplate::fileNewName*	文件类型名称。如果一个程序支持多种文件，此字符串将显示在【File/New】对话框中。如果没有指明，就不能够在【File/New】对话框中处理此种文件。本例只支持一种文件类型，所以当你选按【File/New】，并不会出现对话框。第13 章将示范「一个程序支持多种文件」的作法（#743 页）。
4. *CDocTemplate::filterName*	文件类型以及一个适用于此类型之万用过滤字符串(wildcard filter string)。本例为"Scribble(*.scb)"。这个字符串将出现在【File Open】对话框中的【List Files Of Type】列示盒中。
5. *CDocTemplate::filterExt*	文件档之扩展名，例如"scb"。如果没有指明，就不能够在【File Open】对话框中处理此种文件档。
6. *CDocTemplate::regFileTypeId*	如果你以::RegisterShellFileTypes 对系统的登录数据库（Registry）注册文件类型，此值会出现在HKEY_CLASSES_ROOT 之下成为其子机码（subkey）并仅供Windows 内部使用。如果未指定，此种文件类型就无法注册，鼠标拖放(drag and drop) 功能就会受影响。
7. *CDocTemplate::regFileTypeName*	这也是储存在登录数据库（Registry）中的文件类型名称，并且是给人（而非只给系统）看的。它也会显示于程序中用以处理登录数据库之对话框内。

以下是Scribble 范例中各个字符串出现的位置：



我必须再强调一次，AppWizard 早已帮我们产生出这些字符串。把这些来龙去脉弄清楚，只是为了知其所以然。当然，同时也为了万一你不喜欢AppWizard 准备的字符串内容，你知道如何去改变它。

Scribble 的Document/View 设计

用最简单的一句话描述，Document 就是资料的体，View 就是资料的面。我们藉 *CDocument* 管理资料，藉Collections Classes（MFC 中的一组专门用来处理资料的类别）处理实际的资料数据；我们藉*CView* 负责资料的显示，藉*CDC* 和*CGdiObject* 实际绘图。人们常说一体两面一体两面，在MFC 中一体可以多面：同一份资料可以文字描述之，可以长条图描述之，亦可以曲线图描述之。

Document/View 之间的关系可以图7-3 说明。View 就像一个观景器（我避免使用「窗口」这个字眼，以免引起不必要的联想），使用者透过View 看到Document，也透过View 改变Document。View 是Document 的外显接口，但它并不能完全独立，它必须依存在一个所谓的Document Frame 窗口内。

一份Document 可以映射给许多个Views 显示，不同的Views 可以对映到同一份巨大Document 的不同区域。总之，请把View 想象是一个镜头，可以观看大画布上的任何区域（我们可以选用*CScrollView* 使之具备滚动条）；在镜头上加特殊的偏光镜、柔光镜、十字镜，我们就会看到不同的影像-- 虽然观察的对象完全相同。

资料的管理动作有哪些？读档和写档都是必要的，文件存取动作称为Serialization，由 *Serialize* 函数负责。我们可以（而且也应该）在*CMyDoc* 中改写*Serialize* 函数，使它符合个人需求。资料格式的建立以及文件读写功能将在Scribble step1 中加入，本例（step0）的*CScribbleDoc* 中并没有什么成员变量（也就是说容纳不了什么数据），*Serialize* 则简直是个空函数：

```
void CScribbleDoc::Serialize(CArchive& ar)
{
    if (ar.IsStoring())
    {
           // TODO: add storing code here
    }
    else
    {
           // TODO: add loading code here
    }
}
```

这也是我老说骨干程序啥大事也没做的原因。

除了文件读写，资料的显示也是必要的动作，资料的接受编辑也是必要的动作。两者都由View负责。使用者对Document 的任何编辑动作都必须透过Document Frame 窗口，消息随后传到*CView*。我们来想想我们的View 应该改写哪些函数？

1. 当Document Frame 窗口收到*WM_PAINT*，窗口内的View 的*OnPaint* 函数会被呼叫，*OnPaint* 又调用*OnDraw*。所以为了显示资料，我们必须改写*OnDraw*。至于*OnPaint*，主要是做「只输出到屏幕而不到打印机」的动作。有关打印机，我将在第12 章提到。
2. 为了接受编辑动作，我们必须在View 类别中接受鼠标或键盘消息并处理之。如果要接受鼠标左键，我们应该改写View 类别中的三个虚拟函数：

```
afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
afx_msg void OnLButtonUp(UINT nFlags, CPoint point);
afx_msg void OnMouseMove(UINT nFlags, CPoint point);
```

上述两个动作在Scribble step0 都看不到，因为它是个啥也没做的程序：

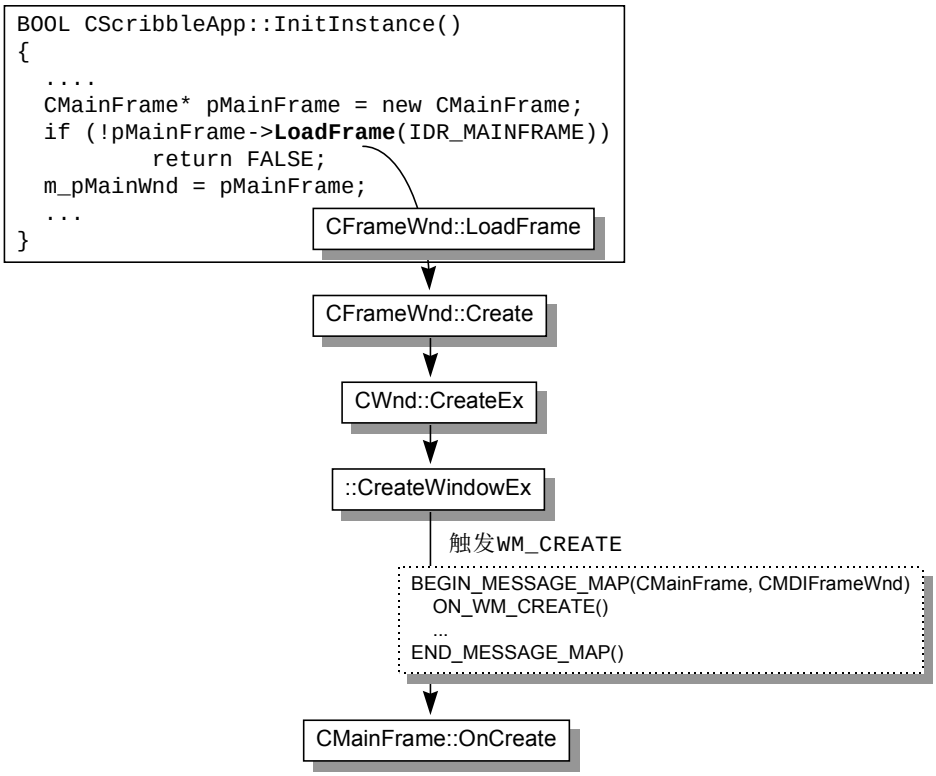
```
void CScribbleView::OnDraw(CDC* pDC)
{
       CScribbleDoc* pDoc = GetDocument();
       ASSERT_VALID(pDoc);

       // TODO: add draw code for native data here
}
```

主窗口的诞生

上一章那个极为简单的Hello 程序，主窗口采用*CFrameWnd* 类别。本例是MDI 风格，将采用*CMDIFrameWnd* 类别。

构造MDI 主窗口，有两个步骤。第一个步骤是*new* 一个*CMDIFrameWnd* 对象，第二个步骤是调用其*LoadFrame* 函数。此函数内容如下：

```
// in WNDFRM.CPP
BOOL CFrameWnd::LoadFrame(UINT nIDResource, DWORD dwDefaultStyle,
                          CWnd* pParentWnd, CCreateContext* pContext)
{
    CString strFullString;
    if (strFullString.LoadString(nIDResource))
        AfxExtractSubString(m_strTitle, strFullString, 0);  // first sub-string

    if (!AfxDeferRegisterClass(AFX_WNDFRAMEORVIEW_REG))
        return FALSE;

    // attempt to create the window
    LPCTSTR lpszClass = GetIconWndClass(dwDefaultStyle, nIDResource);
    LPCTSTR lpszTitle = m_strTitle;
    if (!Create(lpszClass, lpszTitle, dwDefaultStyle, rectDefault,
                pParentWnd, MAKEINTRESOURCE(nIDResource), 0L, pContext))
    {
           return FALSE;   // will self destruct on failure normally
    }

    // save the default menu handle
    m_hMenuDefault = ::GetMenu(m_hWnd);

    // load accelerator resource
    LoadAccelTable(MAKEINTRESOURCE(nIDResource));

    if (pContext == NULL)   // send initial update
           SendMessageToDescendants(WM_INITIALUPDATE, 0, 0, TRUE, TRUE);

    return TRUE;
}
```



窗口产生之际会发出*WM_CREATE* 消息，因此*CMainFrame*::*OnCreate* 会被执行起来，那里将进行工具栏和状态列的建立工作（稍后描述）。*LoadFrame* 函数的参数（本例为 *IDR_MAINFRAME*） 用来设定窗口所使用的各种资源， 你可以从前一页的 *CFrameWnd*::*LoadFrame* 源代码中清楚看出。这些同名的资源包括：

```
// defined in SCRIBBLE.RC
IDR_MAINFRAME ICON DISCARDABLE   "res\\Scribble.ico"
IDR_MAINFRAME MENU PRELOAD DISCARDABLE  { ... }
IDR_MAINFRAME ACCELERATORS PRELOAD MOVEABLE PURE  { ... }
STRINGTABLE PRELOAD DISCARDABLE
BEGIN
   IDR_MAINFRAME  "Scribble Step0" (这个字符串将成为主窗口的标题)
   ..
END
```

这种作法（使用*LoadFrame* 函数）与第 6 章的作法（使用*Create* 函数）不相同，请注意。

工具栏和状态列的诞生 (Toolbar & Status bar)

工具栏和状态列分别由*CToolBar* 和*CStatusBar* 掌管。两个对象隶属于主窗口，所以我们在*CMainFrame* 中以两个变量（事实上是两个对象）表示之：

```
class CMainFrame : public CMDIFrameWnd
{
protected:  // control bar embedded members
       CStatusBar  m_wndStatusBar;
       CToolBar    m_wndToolBar;
...
};
```

主窗口产生之际立刻会发出*WM_CREATE*，我们应该利用这时机把工具栏和状态列建立起来。为了拦截*WM_CREATE*，首先需在Message Map 中设定「映射项目」：

```
BEGIN_MESSAGE_MAP(CMyMDIFrameWnd, CMDIFrameWnd)
       ON_WM_CREATE()
END_MESSAGE_MAP()
```

ON_WM_CREATE 这个宏表示，只要*WM_CREATE* 发生，我的*OnCreate* 函数就应该被调用。下面是由AppWizard 产生的*OnCreate* 标准动作：

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
   if (CMDIFrameWnd::OnCreate(lpCreateStruct) == -1)
          return -1;

   if (!m_wndToolBar.Create(this) ||
          !m_wndToolBar.LoadToolBar(IDR_MAINFRAME))
   {
          TRACE0("Failed to create toolbar\n");
          return -1;     // fail to create
   }

   if (!m_wndStatusBar.Create(this) ||
          !m_wndStatusBar.SetIndicators(indicators,
            sizeof(indicators)/sizeof(UINT)))
   {
          TRACE0("Failed to create status bar\n");
          return -1;     // fail to create
   }
```

```
    // TODO: Remove this if you don't want tool tips or a resizeable toolbar
    m_wndToolBar.SetBarStyle(m_wndToolBar.GetBarStyle() |
           CBRS_TOOLTIPS | CBRS_FLYBY | CBRS_SIZE_DYNAMIC);

    // TODO: Delete these three lines if you don't want the toolbar to
    //  be dockable
    m_wndToolBar.EnableDocking(CBRS_ALIGN_ANY);
    EnableDocking(CBRS_ALIGN_ANY);
    DockControlBar(&m_wndToolBar);

    return 0;
}
```

其中有四个动作与工具栏和状态列的产生及设定有关：

- m_wndToolBar.Create(this) 表示要产生一个隶属于*this*（也就是目前这个对象，也就是主窗口）的工具栏。
- m_wndToolBar.LoadToolBar(IDR_MAINFRAME) 将RC 档中的工具栏资源载入。*IDR_MAINFRAME* 在RC 档中代表两种与工具栏有关的资源：

```
IDR_MAINFRAME BITMAP MOVEABLE PURE "RES\\TOOLBAR.BMP"
```

```
IDR_MAINFRAME TOOLBAR DISCARDABLE  16, 15
BEGIN
    BUTTON      ID_FILE_NEW
    BUTTON      ID_FILE_OPEN
    BUTTON      ID_FILE_SAVE
        SEPARATOR
    BUTTON      ID_EDIT_CUT
    BUTTON      ID_EDIT_COPY
    BUTTON      ID_EDIT_PASTE
        SEPARATOR
    BUTTON      ID_FILE_PRINT
    BUTTON      ID_APP_ABOUT
END
```

LoadToolBar 函数一举取代了前一版的*LoadBitmap* + *SetButtons* 两个动作。*LoadToolBar* 知道如何把BITMAP 资源和TOOLBAR 资源搭配起来，完成工具栏的设定。当然啦，如果你不是使用VC++资源工具来编辑工具栏，BITMAP 资源和TOOLBAR 资源就可能格数不符，那是不被允许的。TOOLBAR 资源中的各ID 值就

是菜单项目的子集合，因为所谓工具栏就是把比较常用的菜单项目集合起来以按钮方式提供给使用者。

- m_wndStatusBar.Create(this) 表示要产生一个隶属于*this* 对象（也就是目前这个对象，也就是主窗口）的状态列。
- m_wndStatusBar.SetIndicators(,...) 的第一个参数是个数组；第二个参数是数组元素个数。所谓Indicator 是状态列最右侧的「指示窗口」，用来表示大写键、数字键等的On/Off 状态。AFXRES.H 中定义有七种indicators：

```
#define ID_INDICATOR_EXT   0xE700  // extended selection indicator
#define ID_INDICATOR_CAPS  0xE701  // cap lock indicator
#define ID_INDICATOR_NUM   0xE702  // num lock indicator
#define ID_INDICATOR_SCRL  0xE703  // scroll lock indicator
#define ID_INDICATOR_OVR   0xE704  // overtype mode indicator
#define ID_INDICATOR_REC   0xE705  // record mode indicator
#define ID_INDICATOR_KANA  0xE706  // kana lock indicator
```

本例使用其中三种：

```
static UINT indicators[] =
{
        ID_SEPARATOR, // status line indicator
        ID_INDICATOR_CAPS,
        ID_INDICATOR_NUM,
        ID_INDICATOR_SCRL,
};
```

鼠标拖放 (Drag and Drop)

MFC 程序很容易拥有Drag and Drop 功能。意思是，你可以从Shell（例如Windows 95 的文件总管）中以鼠标拉动一个文件，拖到你的程序中，你的程序因而打开此文件并读其内容，将内容放到一个Document Frame 窗口中。甚至，使用者在Shell 中以鼠标对某个文件文件（你的应用程序的文件文件）快按两下，也能激活你这个程序，并自动完成开档，读档，显示等动作。

在SDK 程序中要做到Drag and Drop，并不算太难，这里简单提一下它的原理以及作法。当使用者从Shell 中拖放一个文件到程序A，Shell 就配置一块全域内存，填入被拖曳的文件名称（包含路径），然后发出*WM_DROPFILES* 传到程序A的消息队列。程式A取得此消息后，应该把内存的内容取出，再想办法开档读档。

> 并不是张三和李四都可以收到*WM_DROPFILES*，只有具备*WS_EX_ACCEPTFILES* 风格的窗口才能收到此一消息。欲让窗口具备此一风格，必须使用*CreateWindowEx*（而不是传统的*CreateWindow*），并指定第一个参数为*WS_EX_ACCEPTFILES*。

剩下的事情就简单了：想办法把内存中的文件名和其它信息取出（内存handle 放在*WM_DROPFILES* 消息的*wParam* 中）。这件事情有*DragQueryFile* 和*DragQueryPoint* 两个API 函数可以帮助我们完成。

SDK 的方法真的不难，但是MFC 程序更简单：

```
BOOL CScribbleApp::InitInstance()
{
    ...
    // Enable drag/drop open
    m_pMainWnd->DragAcceptFiles();

    // Enable DDE Execute open
    EnableShellOpen();
    RegisterShellFileTypes(TRUE);
    ...
}
```

这三个函数的用途如下：

■ CWnd::DragAcceptFile(BOOL bAccept=TRUE); 参数*TRUE* 表示你的主窗口以及每一个子窗口（文件窗口）都愿意接受来自Shell 的拖放文件。*CFrameWnd* 内有一个*OnDropFiles* 成员函数，负责对*WM_DROPFIELS* 消息做出反应，它会通知application 对象的*OnOpenDocument*（此函数将在第８章介绍），并夹带被拖放的文件的名称。

- CWinApp::EnableShellOpen(); 当使用者在Shell 中对着本程序的文件文件快按两下时，本程序能够打开文件并读内容。如果当时本程序已执行，Framework 不会再执行起程序的另一副本，而只是以DDE（Dynamic Data Exchange，动态资料交换）通知程序把文件（文件）读进来。DDE 处理例程内建在*CDocManager* 之中（第 8 章会谈到这个类别）。也由于DDE 的能力，你才能够很方便地把文件图标拖放到打印机图标上，将文件打印出来。

  通常此函数后面跟随着*RegisterShellFileTypes*。

- CWinApp::RegisterShellFileTypes(); 此函数将向Shell 注册本程序的文件型态。有了这样的注册动作，使用者在Shell 的双击动作才有着力点。这个函数搜寻Document Template 串行中的每一种文件类型，然后把它加到系统所维护的registry（登录数据库）中。

  在传统的Windows 程序中，对Registry 的注册动作不外乎两种作法，一是准备一个.reg 档，由使用者利用Windows 提供的一个小工具regedit.exe，将.reg 合并到系统的Registry 中。第二种方法是利用::*RegCreateKey*、::*RegSetValue* 等Win32 函数，直接编辑Registry。MFC 程序的作法最简单，只要调用 *CWinApp*::*RegisterShellFileTypes* 即可。

必须注意的是，如果某一种文件类型已经有其对应的应用程序（例如.txt 对应 Notepad，.bmp 对应PBrush，.ppt 对应PowerPoint，.xls 对应Excel），那么你的程序就不能够横刀夺爱。如果本例Scribble 的文件档扩展名为.txt，使用者在Shell 中双击这种文件，激活的将是Notepad 而不是Scribble。

另一个要注意的是，拖放动作可以把任何类型的文件文件拉到你的窗口中，并不只限于你所注册的文件类型。你可以把.bmp 文件从Shell 拉到Scribble 窗口，Scribble 程序一样会读它并为它准备一个窗口。想当然耳，那会是个无言的结局：

消息映射（Message Map）

每一个衍生自*CCmdTarget* 的类别都可以有自己的Message Map 以处理消息。首先你应该在类别声明处加上*DECLARE_MESSAGE_MAP* 宏，　然后在.CPP 档中使用*BEGIN_MESSAGE_MAP* 和*END_MESSAGE_MAP* 两个宏，宏中间夹带的就是「讯息与函数对映关系」的一笔笔记录。

你可以从图7-6 那个浓缩的Scribble 源代码中看到各类别的Message Map。

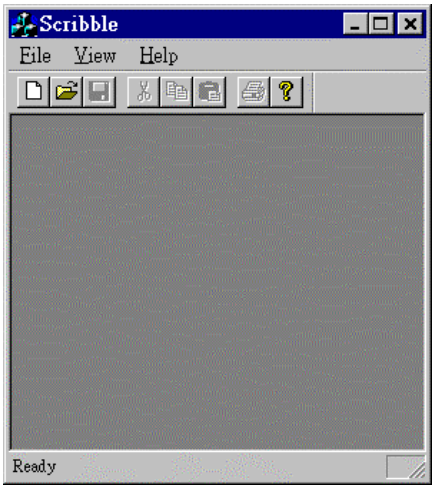
本例*CScribbleApp* 类别接受四个*WM_COMMAND* 消息：

```
BEGIN_MESSAGE_MAP(CScribbleApp, CWinApp)
  ON_COMMAND(ID_APP_ABOUT, OnAppAbout)
  ON_COMMAND(ID_FILE_NEW, CWinApp::OnFileNew)
  ON_COMMAND(ID_FILE_OPEN, CWinApp::OnFileOpen)
  ON_COMMAND(ID_FILE_PRINT_SETUP, CWinApp::OnFilePrintSetup)
END_MESSAGE_MAP()
```

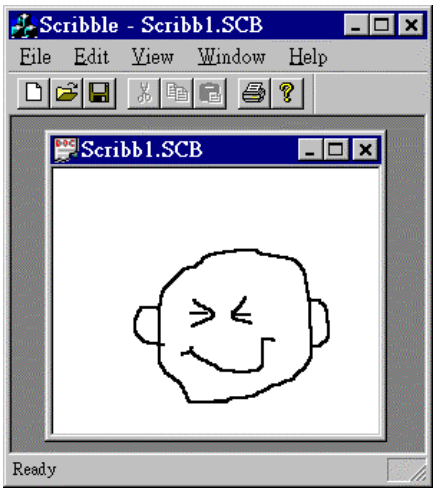
除了*ID_APP_ABOUT* 是由我们自己设计一个*OnAppAbout* 函数处理之，其它三个讯息都交给*CWinApp* 成员函数去处理，因为那些动作十分制式，没什么好改写的。到底有哪些制式动作呢？看下一节！

标准菜单File / Edit / View / Window / Help

仔细观察你所能搜集到的各种MDI 程序，你会发现它们几乎都有两组菜单。一组是当没有任何子窗口（文件窗口）存在时出现（本例代码是*IDR_MAINFRAME*）：



另一组则是当有任何子窗口（文件窗口）存在时出现（本例代码是*IDR_SCRIBTYPE*）：



前者多半只有【File】、【View】、【Help】等选项，后者就复杂了，程序所有的功能都在上面。本例的*IDR_MAINFRAME* 和*IDR_SCRIBTYPE* 就代表RC 档中的两组菜单。
当使用者打开一份文件文件，程序应该把主窗口上的菜单换掉，这个动作在SDK 程序中由程序员负责，在MFC 程序中则由Framework 代劳了。

拉下这些菜单仔细瞧瞧，你会发现Framework 真的已经为我们做了不少琐事。凡是菜单项目会引起对话框的，像是Open 对话框、Save As 对话框、Print 对话框、Print Setup 对话盒、Find 对话框、Replace 对话框，都已经恭候差遣；Edit 菜单上的每一项功能都已经可以应用在由*CEditView* 掌控的文字编辑器上；File 菜单最下方记录着最近使用过的（所谓LRU）四个文件名称（个数可在Appwizard 中更改），以方便再开启；View 选单允许你把工具栏和状态列设为可见或隐藏；Window 菜单提供重新排列子窗口图标的能力，以及对子窗口的排列管理，包括卡片式（Cascade）或拼贴式（Tile）。

下表是预设之菜单命令项及其处理例程的摘要整理。最后一个字段「是否预有关联」如果是Yes，意指只要你的程序菜单中有此命令项，当它被选按，自然就会引发命令处理例程，应用程序不需要在任何类别的Message Map 中拦截此命令消息。但如果是No，表示你必须在应用程序中拦截此消息。

菜单内容	命令项ID	预设的处理函数	预有关联
File			
New	*ID_FILE_NEW*	*CWinApp::OnFileNew*	No
Open	*ID_FILE_OPEN*	*CWinApp::OnFileOpen*	No
Close	*ID_FILE_CLOSE*	*CDocument::OnFileClose*	Yes
Save	*ID_FILE_SAVE*	*CDocument::OnFileSave*	Yes
Save As	*ID_FILE_SAVEAS*	*CDocument::OnFileSaveAs*	Yes
Print	*ID_FILE_PRINT*	*CView::OnFilePrint*	No
Print Pre&view	*ID_FILE_PRINT_PREVIEW*	*CView::OnFilePrintPreview*	No
Print Setup	*ID_FILE_PRINT_SETUP*	*CWinApp::OnFilePrintSetup*	No
"Recent File Name"	*ID_FILE_MRU_FILE1~4*	*CWinApp::OnOpenRecentFile*	Yes

菜单内容	命令项ID	预设的处理函数	预有关联
Exit	*ID_APP_EXIT*	*CWinApp::OnFileExit*	Yes
Edit			
Undo	*ID_EDIT_UNDO*	None	
Cut	*ID_EDIT_CUT*	None	
Copy	*ID_EDIT_COPY*	None	
Paste	*ID_EDIT_PASTE*	None	
View			
Toolbar	*ID_VIEW_TOOLBAR*	*FrameWnd::OnBarCheck*	Yes
Status Bar	*ID_VIEW_STATUS_BAR*	*FrameWnd::OnBarCheck*	Yes
Window（MDI only）			
New Window	*ID_WINDOW_NEW*	*MDIFrameWnd::OnWindowNew*	Yes
Cascade	*ID_WINDOW_CASCADE*	*MDIFrameWnd::OnWindowCmd*	Yes
Tile	*ID_WINDOW_TILE_HORZ*	*MDIFrameWnd::OnWindowCmd*	Yes
Arrange Icons	*ID_WINDOW_ARRANGE*	*MDIFrameWnd::OnWindowCmd*	Yes
Help			
About AppName	*ID_APP_ABOUT*	None	

上表的最后一字段为No 者有五笔，表示虽然那些命令项有预设的处理例程，但你必须在自己的Message Map 中设定映射项目，它们才会起作用。噢，AppWizard 此时又表现出了它的善体人意，自动为我们做出了这些码：

```
BEGIN_MESSAGE_MAP(CScribbleApp, CWinApp)
        ...
        ON_COMMAND(ID_FILE_NEW, CWinApp::OnFileNew)
        ON_COMMAND(ID_FILE_OPEN, CWinApp::OnFileOpen)
        ON_COMMAND(ID_FILE_PRINT_SETUP, CWinApp::OnFilePrintSetup)
END_MESSAGE_MAP()

BEGIN_MESSAGE_MAP(CScribbleView, CView)
        ...
        ON_COMMAND(ID_FILE_PRINT, CView::OnFilePrint)
        ON_COMMAND(ID_FILE_PRINT_PREVIEW, CView::OnFilePrintPreview)
END_MESSAGE_MAP()
```

对话框

Scribble 可以激活许多对话框，前一节提了许多。唯一要程序员自己动手（我的意思是出现在我们的程序代码中）的只有About 对话框。

为了拦截*WM_COMMAND* 的*ID_APP_ABOUT* 项目，首先我们必须设定其Message Map：

```
BEGIN_MESSAGE_MAP(CScribbleApp, CWinApp)
    ON_COMMAND(ID_APP_ABOUT, OnAppAbout)
    ON_COMMAND(ID_FILE_NEW, CWinApp::OnFileNew)
    ON_COMMAND(ID_FILE_OPEN, CWinApp::OnFileOpen)
    ON_COMMAND(ID_FILE_PRINT_SETUP, CWinApp::OnFilePrintSetup)
END_MESSAGE_MAP()
```

当消息送来，就由*OnAppAbout* 处理：

```
void CScribbleApp::OnAppAbout()
{
       CAboutDlg aboutDlg;
       aboutDlg.DoModal();
}
```

其中*CAboutDlg* 是*CDialog* 的衍生类别：

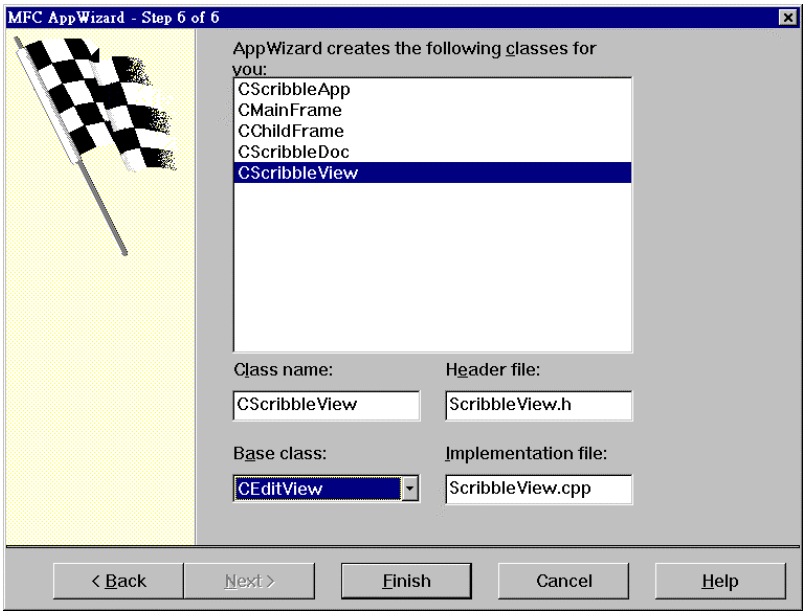
```
class CAboutDlg : public CDialog
{
    enum { IDD = IDD_ABOUTBOX }; // IDD_ABOUTBOX 是RC 文件中的对话框模板资源
    ...
    DECLARE_MESSAGE_MAP()
};
```

比之于SDK 程序中的对话框，这真是方便太多了。传统SDK 程序要在RC 文件中定义对话框模板（dialog template，也就是其外形），在C 程序中设计对话框函数。现在只需从*CDialog* 衍生出一个类别，然后产生该类别之对象，并指定RC 文件中的对话框面板资源，再调用对话框对象的*DoModal* 成员函数即可。

第10 章一整章将讨论所谓的对话框数据交换（DDX）与对话框数据确认（DDV）。

改用CEditView

Scribble step0 除了把一个应用程序的空壳做好，不能再贡献些什么。如果我们在AppWizard 步骤六中把*CScribbleView* 的基础类别从*CView* 改为*CEditView*，那可就有大妙用了：

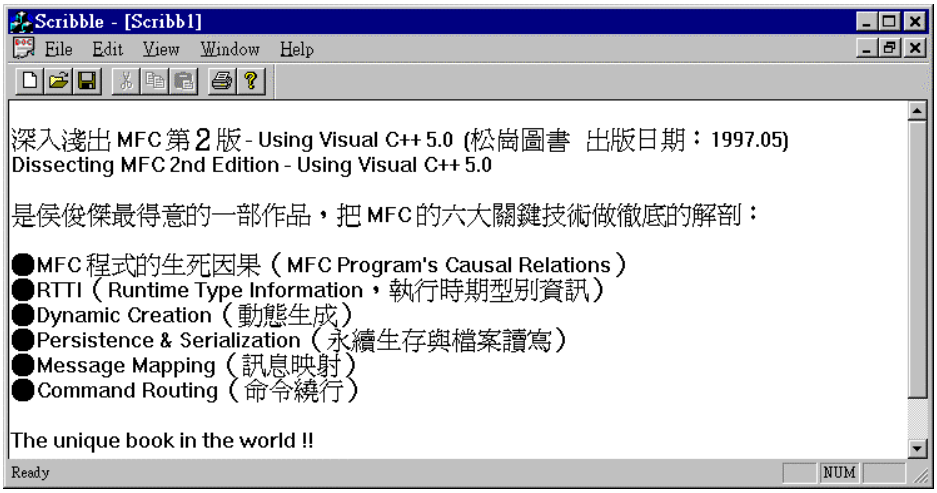


CEditView 是一个已具备文字编辑能力的类别，它所使用的窗口是Windows 的标准控制组件之一Edit，其*SerializeRaw* 成员函数可以把Edit 控制组件中的raw text（而非「对象」所持有的资料）写到文件中。当我们在AppWizard 步骤六选择了它，程序代码中所有的*CView* 统统变成*CEditView*，而最重要的两个虚拟函数则变成：

```
void CScribbleDoc::Serialize(CArchive& ar)
{
    // CEditView contains an edit control which handles all serialization
    ((CEditView*)m_viewList.GetHead())->SerializeRaw(ar);
}

void CScribbleView::OnDraw(CDC* pDC)
{
```

```
    CScribbleDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);

    // TODO: add draw code for native data here
}
```

就这样，我们不费吹灰之力获得了一个多窗口的文字编辑器：



并拥有读写档能力以及预视能力：