1.这个文档是从侯捷网站提供的繁体板简体化过来的。
2.由于排版问题，有些繁体说法在换行时候没有被替换，所以遇到问题大家可以对照原文比较一下。
3.附录、无责任书评那个文件没有转(估计看到那个地方的时候，你手里也该有一本纸板的了)。

《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

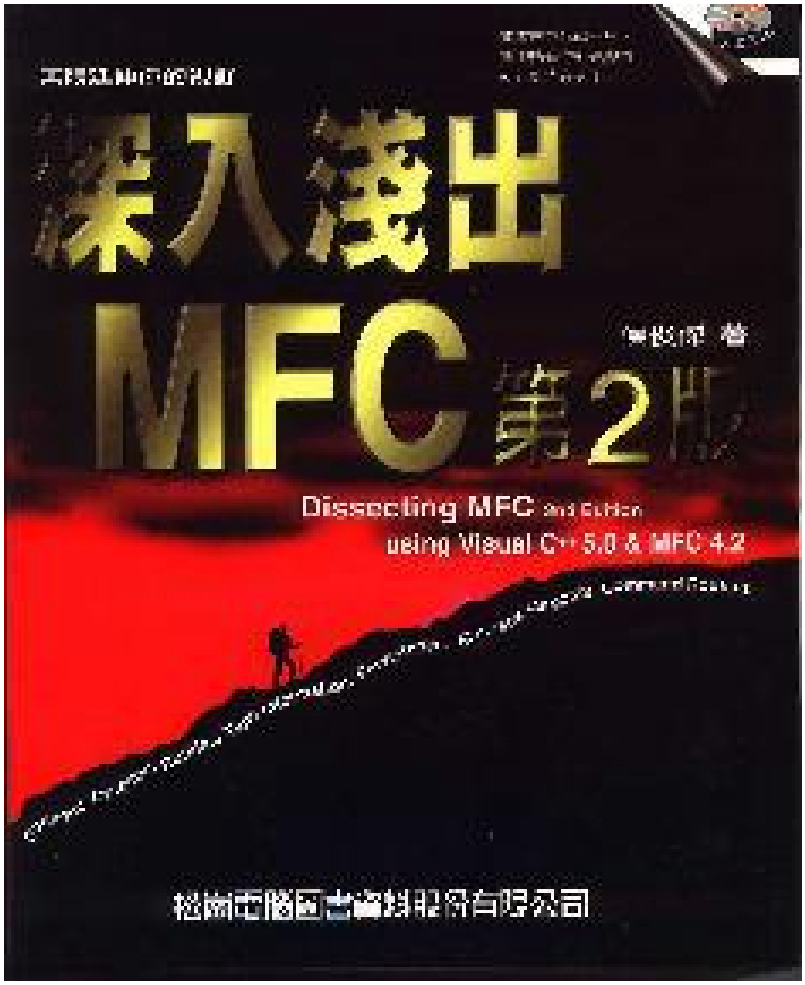
每个档案都可个别阅读。每个档案都有书签（亦即目录连接）。每个档案都不需密码即可打开、选择文字、打印。

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


深入淺出
MFC
第2版
Dissecting MFC 2nd Edition
using Visual C++ 5.0 & MFC 4.2
松崗電腦圖書資訊股份有限公司

山高月小　水落石出

深入淺出 MFC

（第二版　使用 Visual C++ 5.0 & MFC 4.2）

Dissecting MFC

（Second Edition　Using Visual C++ 5.0 & MFC 4.2）

侯俊傑 著

松崗電腦圖資料股份有限公司 印行

Pioneer is the one that an arrow on his back

读者来函

新竹市. 高翠路. 刘嘉均

1996 年11 月，我在书店看到了深入浅出MFC 这本书，让我想起自己曾经暗暗告诉过自己：Application Framework 真是一个好东西。我在书店驻足察看这本书五分钟之后，我便知道这本书是一定要买下的。适巧我工作上的项目进度也到了一个即将完成的阶度，所以我便一口气将这本书给读完了，而且是彻彻底底读了两遍。

我个人特别喜欢第 3 章：MFC 六大关键技术之仿真。这章内容的设计的确在MFC 丛林中，大刀阔斧地披露出最重要的筋络，我相信这正是所有学习MFC 的人所需要的一种表明方式。对我而言，以往遗留的许多疑惑，在此都一一得到了解答。最重要的是，您曾经说过，学习MFC 的过程中最重要的莫过于自我审视MFC 程序代码的能力。很高兴地，在我看完本书之后，我确实比以前更有能力来看MFC 源代码了。总之，我为自己能够更深入了解MFC 而要向您说声谢谢。谢谢您为我们写了深入浅出MFC 这本书。我受益匪浅。

加拿大. 温哥华. 陈宗泰

阁下之书，尚有人性，因此我参而再参，虽不悟，也是enjoyable。看阁下之书的附带效果是，重燃我追求知识的热情。知也无涯，定慧谈何容易。向阁下致敬：『Kruglinski 的Inside Visual C++ 和Hou 的Dissecting MFC 是通往MFC Programming 的皇家大道』。

香港. lnlo@hkstart.com

我是你的一位读者，住在香港。我刚买了你翻译的Inside Visual C++ 4.0（中文版）。在此之前我买了你的另一本书深入浅出MFC。在读了深入浅出MFC 前面50~70 页之后，我想我错买了一本很艰深的书籍。我需要的是一本教我如何利用MFC 来产生一个程序的书，而不是一本教我如何设计一套MFC 的书。但是在我又读了30~40 页之后，我想这本书真是棒，它告诉了我许多过去我不甚清楚的事情，像是virtual function、template、exception...。

Lung Feng <h6003@kl.ntou.edu.tw>

我是您的忠实读友，从您1992 年出版的「Windows 程序设计」到现在的深入浅出MFC，我已花不少银子买下数本您的大作。虽然您的作品比其它国内出版的作品价格稍为高了一点，但我觉得很值得，因为我可以感受出您真的非常用心在撰写，初阅读您的作品时，有时不知其然，但只要用心品尝，总是入味七分。有些书教人一边看书一边上机实作，会是一个比较好的学习曲线，但我是一个从基隆到台北通车的上班族，花很多时间在车上，在车上拜读您的大作真是让人愉快的事情（我回到家已晚，也只有在车上能有时间充实自己）。这段时间内，我无法上机，却能从中受益。而且一次再一次阅读，常会有新的收获，真如古人所说温故而知新。

Asing Chang <asing@ms4.hinet.net>

今天抱着「无论如何一定要」的心情, 把Dissecting MFC 拿出来开封。序还看没完就被深深地感动。这是一本太好的书，我想，我们是一群幸运的读者。虽然我们没有Petzold 或Pietrek, 但是我们一样能拥有最好的阅读水准。

Jaguar <simon830@ms7.hinet.net>

有个问题想问您，为何在台湾要做基础的事（R & D）总是很难如愿。为何只有做近利的事才能被认可为成功之道。希望您能多出版一些像深入浅出MFC 的书，让我们这些想要真正好书的人能大快朵颐一番。谢谢。

Shieh <lmy64621@mail.seeder.net.tw>

我是您忠实的读者，您所写的书一向是我在学习新的东西时很大的帮助。除此之外，我也十分喜欢看你为新书所写的序，带有哲学家的感觉！最近我想学MFC，在市面上找到你的大作深入浅出MFC，拜读后甚为兴奋，也十分谢谢您写书的用心。

台北县. 土城. 邱子文文（资策会技研处）

（深入浅出MFC 是我所读过之MFC 书籍中最精采的一本。您大概不知道，我们人手一本深入浅出MFC 吧。深入Visual C++ 4.0 的情况也差不了多少！

台北市yrleu@netrd.iii.org.tw

买了您的译作深入Visual C++ 4.0 后，才认识您这在Windows Programming 著作方面的头号人物。十分佩服您的文笔，译得通顺流畅，可谓信达雅。之后偶然翻阅您另外的作品Windows 95 系统程序设计大奥秘与深入浅出MFC，更是对您五体投地，立刻将这两本书买下来，准备好好享受一下。对于深入浅出MFC，我给予极高的评价，因为它完全满足我的需要。我去年才从台大电机博士班计算器科学组毕业，目前在资策会信息技术处服国防役。先前作的纯是理论研究，现在才开始接触Windows Programming。您的深入浅出MFC 对我而言是圣经。

真的很感谢您为知识传授所作的努力！

台中Fox Wang

自从阅读深入浅出MFC 之后，我便成了您的忠实读者。我并不是一位专业程序设计师，而是一个对Windows Programming 有浓厚兴趣（和毅力）的大学生。在漫长而陡峭的学习曲线中，有几本书对于我们而言就像茫茫大海中的一盏明灯，为我们指引明确的航道！我说的是Charles Petzold 的*Programming Windows 95*、David J.Kruglinski 的*Inside Visual C++*，还有，当然，您的深入浅出MFC！

印尼. 雅加达robin.hood@ibm.net

对您的书总是捧读再三，即使翻烂了也值得。这本深入浅出MFC，不但具有学习价值，亦极具参考价值。

我买您的第一本书，好象是「内存管理与多任务」。还记得当时热中突破640KB 内存，发现该书如获至宝。数月前购买了深入浅出MFC，并利用闲暇时间翻阅学习（包括如厕时间... ）。

我的学习曲线比较不同，我比较倾向于了解事情的因，而不是该如何做事情。比方说，「应该使用MFC 的哪个类别」或「要改写哪个虚拟函数」，对我而言还不如「*CWinApp* 何时何地调用了我的什么函数」或「*CDocManager* 到底做了什么」来得有趣（嗯，虽说是一样重要啦）。这些「事情的因」在您的书中有大量详细的介绍。

新庄. 辅大skyman@tpts4.seed.net.tw

拜读您的大作深入浅出MFC 令我感到无比兴奋，对于您对计算机技术的专研如此深入，感到真是中国人之光。系上同学对于您的书籍爱恨交加，爱是如此清晰明了，恨是恨自己不成材呀！许多学长、同学、学弟都很喜爱您的作品，有些同学还拜您为偶像。因此想请您来演说，让我们更深入认识程序语言的奥秘。大四学长知道要邀请您来，都非常高兴，相当期待您的到来。

Rusty（枫桥驿站CompBook 版）

深入浅出MFC 我读好几遍了，讲一句实在话，这本书给我的帮助真的很多！毕竟这样深入挖MFC 运作原理的书难找！要学MFC 又没有Windows SDK 经验者，建议跟Programming Windows 95 with MFC 一起看，学起MFC 会比较扎实。

若单纯就「买了会不会后悔」来判断一本书到底好不好，这本书我觉得物超所值！

内坊. 元智Richard

刚才又把深入浅出MFC step0~step1 的程序看了一次，真的感触良多。酒越陈越香，看老师您的书，真的是越看越「爽」，而且一定要晚上10:00 以后看，哇，那种感觉真是过瘾。

桃园Shelly

在书局看到您多本书籍，实在忍不住想告诉您我的想法！我是来谢谢您的。怎么说呢？姑且不论英文能力，看原文书总是没有看中文书来得直接啊！您也知晓的，许多翻译书中的每个中文字都看得懂，但是整段落就是不知他到底在说啥！因此看到书的作者是您，感觉上就是一个品质上的保证，必定二话不说，抱回家啰！虽然眼前用不到，但是翻翻看，大致了解一下，待有空时或是工作上需要时再好好细读。

网络书局的盛行，让我也开始上网买些书。但是我只敢买像您的书！有品质嘛！其它的可就不敢直接买啰，总是必须到书局翻翻看，确定一下內容，才可能考虑。

台北市Jedi

Your books is already 100 times better than any translation on the market. I won't think of to get a Chinese computer book unless you wrote it or translated it.

shiowli@ms13.hinet.net

1997/11 月我看见了深入浅出MFC。仔细研读后我知道这是我在MFC 及Windows 程序设计领域中的大卫之星。您的书一直都是我的良师，不但奠定了我的根基，也使我对Windows 程序设计兴趣大增。国内外介绍MFC 程序设计的书很多，但看过范例后仍有一种被当成 puppet 的感觉。感谢侯先生毫不保留地攻坚MFC，使我得到了豁然开朗的喜悦。

侯先生的文笔及胸襟令我佩服。有着Charles Petzold 慈父般的讲解，也有着David J.Kruglinski 特有的风趣。您自述中说「先生不知何许人也」，嗯，我愿意，我愿意做一个默默祝福的人，好叫我在往后岁月里能有更多喝采和大叫eureka 的机会！

"anchor" <hcy89@mozart.ee.ncku.edu.tw>

I am a student of NCKU EE Department, I am also a reader of you books. Your Book give me a lot of help on my research.

北投z22356@ms13.hinet.net

选到这本书之前，我还在书架前犹豫：「又是一本厚厚大大却一堆废话的烂书，到底有没有比较能让我了解的书呀？」但是读了您的深入浅出MFC 之后，把我80% 的疑虑通通消除掉了。想永远做您的读者的读者敬上。

David david@mail.u-p-c.com

我是您的读者，虽然我尚未看完这本书，只看到第三篇的第 5 章，但我忍不住要把心得告诉您。去年我因为想写Windows 程序而买您的书，说老实话，我实在看不懂您所写的文字（我真的懂C++ 语法，也用Visual Basic 写过Windows 程序），故放弃改买其它书籍来学习。虽把那些书全部看完了，也利用MFC 来写简单程序，但心里仍搞不懂这些程序的How、What、Why。后来整理书架，发现这本书，就停下来拿来看，结果越看越搞懂这些Windows 程序到底如何How、What、Why。正如您的序所说「只用一样东西，不明白它的道理，实在不高明」，感谢您能不计代价去做些不求近利的深耕工作，让我们一群读者能少走冤枉路。谢谢您!! 祝您身体健康!!

Chengwei chengwei@accton.com.tw

最近正拜读大作深入浅出MFC，明显感受到作者写书之负责用心（不论内容、排版设计、甚至用字遣词乃至众多的图标）。虽然刚开始会觉得有些艰涩，但在反复阅读之后便能深深感受其中奥秘。谢谢你的努力! 请再接再厉让我们有更多好书可看!

Fox Wang <wych@ms10.hinet.net>

身为您的忠实读者，总是希望能让您听到我们的声音：记得您总是常在作品中强调读者们寄给你的信对你具有很大的意义，但我要说，您可能不知道您带给了某些读者多大的正面意义！就我自己而言，从几乎放弃对信息科学的兴趣，到留下来继续努力，从排拒原文书到阅读原文书成为习惯，从害怕阅读原文期刊到慢慢发现阅读它们的乐趣，然后，我打算往信息方面的研究所前进。我想，不管将来我是否能将工作和兴趣互相结合，您都丰富了我追求资讯科学这个兴趣时的生活，非常感谢您！

当然，身为一位读者，还是忍不住要自私地说，希望在很久很久以后，还可以看到您仍然在写作！当然，身子也要好好照顾。

上海wuwei akira <hakira@hotmail.com>

I'm your reader in Shanghai JiaoTong University（按：上海交通大学）in mainland. Your <Programming WINDOWS MFC> （按：深入浅出MFC 简体版）is a very good book that I wanted to have for years. Thank you very much. So I want to know if there are another your book that I can buy in mainland? I hope to read your new books.

hon.bbs@bbs.ee.ncu.edu.tw
我非常喜欢你的书，不管是著作或是翻译，给人的感觉真的是"深入浅出"，我喜欢你用浅近的比喻说明，来解释一些比较深入和抽象的东西，读你的书，总让我有突然"顿悟" 的感觉，欣喜自己能在迷时找到良师。

武汉"wking" <wking@telekbird.com.cn>
Microsoft Developer Studio 与MFC（Microsoft Foundation Classes）相配合，构成了一个强大的利用C++ 进行32 位Windows 程序开发的工具，但是由于MFC 系统相当庞大，内容繁杂，并且夹杂着大量令初学者莫明其妙的macros，更加大了学习上的难度。
当今市面上有不少讲解C++ 和VC++ 程序设计的书籍，但C++ 书籍单纯只讲C++，从C++ 过渡到VC++ 却是初学者的一大难关；大多数讲解VC++ 的书都将重点放在如何使用Microsoft Developer Studio，很少有对MFC 进行深入而有系统的讲解。而将C++ 与VC++ 相联系，从C++ 的角度来剖析MFC 的运作，深入其设计原理与内部机制的书，更是凤毛麟角。本人在市面上找了将近四个月，才发现这样的一本，这就是由蜚声海峡两岸的著名电脑专家侯俊杰先生所着之《深入浅出WINDOWS MFC 程序设计》（按：深入浅出MFC 简体版）。
本人在一月前购得此书，仔细研究月余，自我感觉比以前大有长进，其间由于印刷错误等原因，发现多处错误，于是向先生去信求教，得先生热情支持和辅导。当先生得知本书（简体本）未附光盘，且书中有多处误印，深恐贻误读者，于是将原书光盘所附之源程序和执行文件email 一份给我，嘱我广为散发，以惠大众。

EricYang <robe@FreeBSD.csie.NCTU.edu.tw>
这真是本值得好好阅读，好好保存的好书

cview.bbs@cis.nctu.edu.tw News / BBS 论坛programming
深入浅出MFC，侯sir 自评为MFC 四大天王之一，的确是杰作...

"lishyhan" <lishyhan@ms14.hinet.net>
我听别人介绍，买了深入浅出MFC 第二版，的确是很适合我，之前买的书都太笼统了。

美国dengqi@glocom-us.com
侯俊杰先生：您好！从学校出来的七年间，我大多从事embedded system software 的设计。在大陆，主要从事交换机系统软件的设计，到了美国，主要从事卫星通信地面站系统软件的设计。程序设计主要结合C 和Assembly。在大陆，embedded system 多采用Intel 的processor，在美国，embedded system 多采用Motorola 的processor。所以，我对Intel 8086, 8051 系列及Motorola 68000 系列的assembly 语言比较熟悉，而对framework 这样的软件制造思想和手段一直并不熟悉。近来偶有机会加入一个project，要生成在Win95 下运行的代码，因此，想尝试一下使用framework 构造软件。很幸运，我找到了您的书。讲VC++ MFC 的书很多，但能像您这样做到「深入浅出」的，实在很少。看您的书，是享受。我手里这本是简体版，华中理工大学出版社出版。

News / BBS 论坛（CompBook and/or programming）

请问，在MFC 书籍之中，哪一本比较容易懂，因为我是初学MFC，所以我需要的是比较基础且容易了解的，想请大家推荐一下适合的书。

ob9@bbs.ee.ntu.edu.tw：反正不管你是不是初学，只要你要继续学，就应该看看深入浅出MFC 啦！

os2.bbs@titan.cc.ntu.edu.tw：侯俊杰的书就对了!!

openwin.bbs@cis.nctu.edu.tw：等你对MFC 有一个程度的了解后再去看侯sir 写的深入浅出MFC... 保证让你功力大增~~

Rosario.bbs@bbs.ntu.edu.tw：深入浅出MFC 这本比较好~~~不过之前最好买侯老师的多型与虚拟拟，把C++ 弄清楚。最后看起深入Visual C++ 就会吸收很快。

请问，想要从DOS 跨足到Windows 程序设计有哪些书值得推荐呢?

hschin.bbs@bbs.cs.nthu.edu.tw：建议你看侯俊杰的深入浅出MFC，里面除了对窗口程序的架构作基础性的说明，让你了解一些基础概论，也说了不少窗口程序设计的课题，是非常不错的一本书。

News / BBS 论坛（CompBook and/or programming）

请问VISUAL C++ 初学者适合的好书?

wayne.bbs@bbs.ee.ncu：侯俊杰的深入Visual C++（*Inside visual C++* 中译本）不错，适合初学者对MFC 做初步的认识与应用。深入浅出MFC 这一本原理讲的较多。

Sagitta.bbs@firebird.cs.ccu.edu.tw：*Inside Visual C++ 4.0* 不是初学者用的书，因为它未从最基本观念讲起。深入浅出MFC 前半本都在描述(或说仿真) MFC 的内部技术，甚至挖出MFC 部份原始程序代码来说明，透过这本书来学MFC 会学得很扎实，不过自己要先对Windows 这个操作系统的运作方式有一程度的了解，不然会看不懂，以某方面来说，也不是初学者用的书。基本上侯俊杰写的书不论文笔或是内容都相当的好，相当有购买的价值，不过你别期望会是「初学用书」。

刚学MFC 程序，是否可以推荐几本你认为很好的工具书或者是参考书，原文的也没关系，重要的是讲的详细。谢谢各位

dickg.bbs@csie.nctu.edu.tw：我个人认为侯俊杰先生所着的深入浅出MFC 第二版不错。这是一本受大众推崇的好书，值得一再阅读。但它的内容在某方面有些难度，so...需有耐心地一再翻阅，再辅以on-line help 和其它VC 书籍，如此定能收获不少

Rusty (Rusty)：我推荐*Programming Windows 95 with MFC*（Jeff Prosise / Microsoft Press）。*Inside Visual C++* 这本广度够，不过MFC 初学者可能会看不懂；读完了上一本之后再读这本，你会活得快乐些 。中文书嘛，大同小异的一大堆，不过侯俊杰的深入浅出MFC 非常独特，值得一读，很棒的一本书！

News / BBS 论坛（CompBook and/or programming）

请推荐几本Visual C++ 的书

kuhoung.bbs@csie.nctu.edu.tw： (1) *Inside Visual C++ 5.0* (2) *MFC Professional 5.0* (3) Mr. 侯俊杰Any Books

"howard" <lm3@ms22.hinet.net>： 先读一点SDK 著作，再读深入浅出MFC，就够了。剩下就多看MSDN 吧。

我是一个刚学VC 不久的人，想写Windows 程序，却发现一大堆看不懂的函数或类别。查help，都是英文，难懂其中意思。请问一下有没有关于这方面的函数用法及意义的书籍呢? 有没有这方面的初学书籍。我逛了几间书店，是有买几本MFC 书籍，不过还是看不懂。

"apexsoft" <lishyhan@ms14.hinet.net>：如果要说书的话，侯俊杰先生翻译的深入Visual C++ 和他所写的深入浅出MFC 两本应该是够用了。不然就再加上一本SDK 书籍，这样子应该是可以打个基础了。

CCA.bbs@cis.nctu.edu.tw：函数名称可以查help，重要的是C++ 的观念。另外就是要了解MFC 里的Document/View/Frame，以及Dynamic Creation, Message mapping 等等。深入浅出MFC 第二版对这些部份都有很深入的探讨，把MFC 里的一些机制直接trace code 加以说明。

News / BBS 论坛（CompBook and/or programming）

我想请问以下宏的意义及其使用时机和作用： DECLARE_DYNCREATE, DECLARE_DYNAMIC, IMPLEMENT_DYNAMIC, IMPLEMENT_DYNCREATE, DECLARE_MESSAGE_MAP, BEGIN_MESSAGE_MAP, END_MESSAGE_MAP。感激不尽，因为我常搞不清楚。

titoni：可参考侯俊杰着的深入浅出MFC 2/e 第三章，第八章及第九章，书上的讲解可以让你有很大的收获。

好象世界末日：最近买了深入浅出MFC。我一页一页仔细地阅读。第一章...第二章... 勉强有点概念，但是到了第三章，感觉好象世界末日了。MFC 六大技术的仿真...好象很难懂，读起来非常吃力 是不是有其它书讲得比较简单的？我不是计算机科系学生，只是对计算机程序设计有兴趣，一路由basic -> FORTRAN -> C -> C++ 走来...

szu.bbs@bbs.es.ncku.edu.tw：是的，第三章也许是世界末日，当初我看的时候也是跳过不看，不然就是看完frame1 后就说再见了。但是你只要很努力地慢慢看，一步一步地看，你就会发现后面的章节是那么清楚明了... 慢慢来吧，这第三章我也是看了三遍才弄懂了一次。我也非计算机科系学生，与你相同的路子走来，有点SDK 概念和一点Data structure 概念，对第三章会很容易懂的，加油。

轶名：我看第三章的时候也很辛苦，但懂了之后，后面的章节可是用飙的喔。

武汉bin_zhou

I am your reader of 《深入浅出WINDOWS MFC 程序设计》（编按：深入浅出MFC 简体版）. I'm leaving in HUBEI__WUHAN（编按：湖北武汉）. Now, I have already get the book in HUA_ZHONG_LI_GONG_DA_XUE（编按：华中理工大学）. And I am interested in this book very much.

屏东a8756001@mail.npust.edu.tw

侯先生,您好，我是屏科大的学生，想要用MFC 写一个可以新增、修改、删除资料等动作的程序，日前老师借了我您的书深入浅出MFC 第二版，我读了很快乐，对于Visual C++ 的IDE 环境更为了解，对于MFC 整个架构，有了比较明朗的感觉。

大陆Mike Dong <mikedong@online.sh.cn>

尊敬的侯俊杰先生：我叫董旬。我对C/C++ 非常有兴趣。畅读了您写的书《深入浅出WINDOWS MFC 程序设计》（编按：深入浅出MFC 简体版），对我有非常大的帮助。在此，先感谢您。现在我感到对C++ 语言本身和MFC 框架十分了解，但在编程过程中仍然感到生疏，主要是函数的运用和函数的参数十分复杂。我对WINDOWS SDK 编程较少，是否应该要熟悉WINDOWS API 函数后，结合MFC 框架编程？

侯俊杰回复：的确如此。MFC 其实就是把Windows API 做了一层薄薄包装，包装于各个设计良好的classes 而已。所以，掌握了MFC framework 架构组织之后，接下来在programming 实务方面，就是去了解并运用各个classes，而各个classes 的member functions 有许多就是Windows APIs 的一对一化身。

左营luke@orchid.ee.ntu.edu.tw

侯先生你好：我现在是一名信息预官，还在左营受训。因为受训的关系所以偶然间有机会读到你写的深入浅出MFC 第二版。本以为这么大一本书，一定很难K，但从第一眼开始我就深深的被其中优雅且适当的文辞所吸引。尤其当阅读第三章时，那些表格让我回忆起以前修过advanced compiler 去trace java compiler 的那段过程，不禁发出会心一笑。

由于我本身学的是电机，所以不同于一般信息人员所着重的应用层面。从大二时因为想充实自己的计算机实力，努力学写程序开始，就在浩翰的书海中发现你独特的风格。尤其现今电书籍多是翻译居多其中品质良莠不齐，你的作品尤其难能可贵。现今我仍然有时会去阅读专业期刊或者杂志，但碍于毕竟不是信息教育训练出身，有时会抓不住重点，甚者不求甚解。这是我觉得遗憾之处。但读你的作品让我在质量之间都获得了相当的进步，且读来相当轻松自然。你的序言中提到欢迎读者的反应，这也是这封mail 的动机。我想好的作家需要我们的鼓励，当然也希望能从你处获得更多的新知。谢谢。

大陆"BaiLu" <jinyang@public1.wx.js.cn>

侯先生：您好！以前我一直是用DELPHI 和PB 主要做调制解调器的，近日在看您编写的《深入浅出WINDOWS MFC 程序设计》（编按：深入浅出MFC 简体版），收益非浅，很佩服您的写作水平，讲得非常好。在大陆还是很少有您这般水准写C++ 的书。在此表示感谢。

北京"Zhang Yongzhong" <yongzhongz@263.net>

尊敬的侯俊杰先生：您好！我是北京的一名计算器工作者，也是您的忠实读者。有幸读到您的一本非常优秀的著作《深入浅出WINDOWS MFC 程序设计》，非常兴奋，自感受益匪浅，觉得是一本难得的好书。

深入浅出MFC

二版五刷感言

我很开心地获知，深入浅出MFC 第二版即将进行第五刷。如果把第一版算进去，那就累积印制9150 本了（不含简体版）。也就是说，这本书拥有几近一万人（次）的读者群（不含简体版读者）。

对一本如此高阶又如此高价的技术书籍而言，诚不易也。我有许多感触！

先从技术面谈起。我阅读过的VC++ 或MFC 书籍不算少，因此，我很有信心地说，这本书的内容有其独步全球之处。本书企图引领读者进入MFC 这个十分庞大并在软件工具市场上极端重要之application framework 的核心；我尝试剖析其中美好的对象导向性质（注1）的实作方式，亦尝试剖析其中与Windows 程序设计模型（注2）息息相关之特殊性质（注3）的实作方式。

注1：此指runtime type information、dynamic creation、persistence、document/view¡ K。

注2：此指message based、event driven 之programming model。

注3：此指message mapping、command routing ¡ K。

在技术层次上，唯MFC Internals 堪与本书比拟（本书附录A附有MFC Internals 简介）。

但是MFC Internals 与Dissecting MFC（本书之英文名称）不仅在内容上各擅胜场，在诉求上亦截然不同。这本书并不是为精通MFC programming 的老手而写（虽然它通常亦带给这样的读者不少帮助），而是为初窥MFC programming 的新手所写。MFC Internals 可以说是为技术而技术，探讨深入，取材范围极广；Dissecting MFC 却是为生活而技术，探讨深入，但谨守主轴份际。所有我所铺陈的核心层面的知识，都是为了建立起一份扎扎实实的programming 基础，让你彻底了解MFC 为你铺陈的骨干背后，隐藏了多少巧妙机关，做掉了多少烦琐事务。

有了这份基础，你才有轻松驾驭MFC 的本钱。

唯有这份基础，才能使你胸中自有丘壑。

如果够用心，你还可以附带地从本书概略学习到一个application framework 的设计蓝图。虽然，99.99999% 的programmer 终其一生不会设计一个application framework，这样的蓝图仍可以为你的对象导向观念带来许多面向的帮助。

我一直希望，能够为此书发行英文国际版。囿于个人的语文能力以及时间，终未能行。但是看到来自世界各地的华人读者的信函（加拿大、纽西兰、越南、印尼、香港、中国大陆、美国...），也是另一种安慰。在BBS 及Internet News 看到各界对此书的评介，以及对此书内容的探讨，亦让我感到十分欣喜。

这本书（第二版）所使用的开发环境是Visual C++ 5.0 & MFC 4.21。就在第五刷即将印行的今天，Visual C++ 6.0 也已问世；其中的programming 关键，也就是MFC，在主干上没有什么变化，因此我不打算为了Visual C++ 6.0 而改版。

在此新刷中，我继续修正了一些笔误，并加上新的读者来函。

未来，本书第三版，你会看到很大的变化。

侯俊杰台湾.新竹1998.09.11

jjhou@ccca.nctu.edu.tw

FAX 886-3-5733976

第二版序

任何人看到前面的读者来函，都会感动于一本计算机书籍的读者与作者竟然能够产生如此深厚的共鸣，以及似无若有的长期情感。

何况，身为这本书的作者的我！

我写的是一本技术书籍，但是赢得未曾谋面的朋友们的信赖与感情。我知道，那是因为这本书里面也有许多我自己的感情。每当收到读友们针对这本书寄来的信件（纸张的或电子的），我总是怀着感恩的心情仔细阅读。好几位读友，针对书中的可疑处或是可以更好的地方，不吝啬地拨出时间，写满一大张一大张的信纸，一一向我指正。我们谈的不只是技术，还包括用词遣字的意境。新竹刘嘉均先生和加拿大陈宗泰先生给我非常宝贵的技术上的意见。陈先生甚至在一个月内来了五封航空信。

这些，怎不教我心怀感谢，并且更加戒慎恐惧！

感谢所有读者促成这本书的更精致化。Visual C++ 5.0 面世了，MFC 则停留在4.2，程序设计的主轴没有什么大改变。对于新读者，本书乃全新产品自不待言，您可以从目录中细细琢磨所有的主题。对于老读者，本书所带给您的，是更精致的制作，以及数章新增的内容（请看第０章「与前版本之差异」）。

最后，我要说，我知道，这本书真的带给许多人很扎实的东西。而我所以愿意不计代价去做些不求近利的深耕工作，除了这是身为专业作家的责任，以及个人的兴趣之外，是的，我自己是工程师，我最清楚工程师在学习MFC 时想知道什么、在哪里触礁。

所有出自我笔下的东西，我自己受益最丰。

感谢你们。

侯俊杰台湾.新竹1997.04.15

jjhou@ccca.nctu.edu.tw

FAX 886-3-5733976

第一版序

有一种软件名曰version control，用来记录程序开发过程中的各种版本，以应不时之需，可以随时反省、检查、回复过去努力的轨迹。

遗憾的是人的大脑没有version control 的能力。学习过程的彷徨犹豫、挫折困顿、在日积月累的渐悟或　x那之间的顿悟之后，彷佛都成了遥远模糊的回忆；而屡起屡仆、大惑不解的地方，学成之后看起来则是那么「理所当然」。

学习过往的艰辛，模糊而明亮，是学成冠冕上闪亮的宝石。过程愈艰辛，宝石愈璀璨。作为私人「想当年」的绝佳话题可矣，对于后学则无甚帮助。的确，谁会在一再跌倒的地方做上记号，永志不忘？谁会把推敲再三的心得殷实详尽地记录下来，为后学铺一条红地毯？也许，没有version control 正是人类的本能，空出更多的脑力心力与精力，追求更新的事物。

但是，作为信息教育体系一员的我，不能不有version control。事实上我亦从来没有忘记初学MFC 的痛苦：C++ 语言本身的技术问题是其一，MFC 庞大类别库的命名规则是其二，熟知的Windows 程序基本动作统统不见了是其三，对象导向的观念与application framework 的包装是其四。初学MFC programming 时，我的脑袋犹如网目过大的筛子，什么东西都留不住；各个类别及其代表意义，过眼即忘。

初初接触MFC 时，我对Windows 操作系统以及SDK 程序设计技术的掌握，实已处在众人金字塔的顶端，困顿犹复如斯。实在是因为，对传统程序员而言，application framework 和MFC 的运作机制太让人陌生了。

目前市面上有不少讲解MFC 程序设计观念的书籍，其中不乏很好的作品，包括*Programming Windows 95 with MFC*（Jeff Prosise 着，Microsoft Press 出版），以及我曾经翻译过的*Inside Visual C++ 4.0*（David J.Kruglinski 着，Microsoft Press 出版）。深入浅出MFC 的宗旨与以上二书，以及全世界所有的MFC 或Visual C++ 书籍，都不相同。全世界（呵，我的确敢这么说）所有与MFC 相关的书籍的重点，都放在如何使用各式各样的MFC 类别上，并供应各式各样的应用实例，我却意不在此。我希望提供的是对MFC 应用程序基本架构的每一个技术环节的深入探讨，其中牵扯到MFC 本身的设计原理、对象导向的观念、以及C++ 语言的高级议题。有了基础面的全盘掌握，各个MFC 类别之使用对我们而言只不过是手册查阅的功夫罢了。

本书书名已经自我说明了，这是一本既深又浅的书。深与浅是悖离的两条射线，理不应同时存在。然而，没有深入如何浅出？不入虎穴焉得虎子？

唯有把MFC 骨干程序的每一个基础动作弄懂，甚至观察其源代码，才能实实在在掌握MFC 这一套application framework 的内涵，及其对象导向的精神。我向来服膺一句名言：源代码说明一切，所以，我挖MFC 源代码给你看。

这是我所谓的深入。

唯有掌握住MFC 的内涵，对于各式各样的MFC 应用才能够如履平地，面对庞大的application framework 也才能够胸中自有丘壑。

这是我所谓的浅出。

本书分为四大篇。第一篇提出学习MFC 程序设计之前的必要基础，包括Windows 程序的基本观念以及C++ 的高阶议题。「学前基础」是相当主观的认定，不过，基于我个人的学习经验以及教学经验，我的挑选应该颇具说服力。第二篇介绍Visual C++ 整合环境开发工具。本篇只不过是提纲挈领而已，并不企图取代Visual C++ 使用手册。然而对于软件使用的老手，此篇或已足以让您掌握Visual C++ 整合环境。工具的使用虽然谈不上学问，但在可视化软件开发过程中扮演极重角色，切莫小觑它。

第三篇介绍application framework 的观念，以及MFC 骨干程序。所谓骨干程序，是指Visual C++ 的工具AppWizard 所产生出来的程序代码。当然，AppWizard 会根据你的选项做出不同的程序代码，我所据以解说的，是大众化选项下的产品。

第四篇以微软公司附于Visual C++ 光盘片上的一个范例程序Scribble 为主轴，一步一步加上新的功能。并在其间深入介绍Runtime Type Information（RTTI）、Dynamic Creation、Persistence（Serialization）、Message Mapping、Command Routing 等核心技术。这些技术正是其它书籍最缺乏的部份。此篇之最后数章则脱离Scribble 程序，另成一格。

本书前身，1994/08 出版的Visual C++ 对象导向MFC 程序设计基础篇以及1995/04 年出版的应用篇篇，序言之中我曾经这么说，全世界没有任何书籍文章，能够把MFC 谈得这么深，又表现得这么浅。这些话已有一半成为昨日黄花：Microsoft Systems Journal 1995/07 的一篇由Paul Dilascia 所撰的文章*Meandering Through the Maze of MFC Message and Command Routing*，以及Addison Wesley 于1996/06 出版的*MFC Internals* 一书，也有了相当程度的核心涉猎，即连前面提及的*Programming Windows 95 with MFC* 以及*Inside Visual C++ 4.0* 两本书，也都多多少少开始涉及MFC 核心。我有一种「德不孤必有邻」的喜悦。

为了维护本书更多的唯一性，也由于我自己又钻研获得了新的心得，本书增加了前版未有的Runtime Type Information、Dynamic Creation 等主题，对于Message Mapping 与Command Routing 的讨论也更详细得多，填补了上一版的缝隙。更值得一提的是， 我把这些在MFC 中极神秘而又极重要的机制，以简化到不能再简化的方式，在DOS 程序中仿真出来，并且补充一章专论C++ 的高阶技术。至此，整个MFC 的基础架构已经完全曝露在你的掌握之中，再没有任何神秘咒语了。

本书从MFC 的运用，钻入MFC 的内部运作，进而application framework 的原理，再至物件导向的精神，然后回到MFC 的运用。这会是一条迢迢远路吗？

似远实近！

许多朋友曾经与我讨论过，对于MFC 这类application framework，应该挖掘其内部机制到什么程度？探究源代码，岂不有违「黑盒子」初衷？但是，没有办法，他们也同意，不把那些奇奇怪怪的宏和指令搞清楚，只能生产出玩具来。对付MFC 内部机制，态度不必像对付MFC 类别一样；你只需好好走过那么一回，有个印象，足矣。至于庞大繁复的整个application framework 技术的铺陈串接，不必人人都痛苦一次，我做这么一次也就够了 。

林语堂先生在朱门一书中说过的一句话，适足作为我写作本书的心境，同时也对我与朋友之间的讨论做个总结：

「只用一样东西，不明白它的道理，实在不高明」。

祝各位胸中丘壑自成！

侯俊杰　　新竹1996.08.15

P.S. 愈来愈多的朋友在网络上与我打招呼，闲聊谈心。有医师、盲生、北京的作家、香港的读者、从国中到研究所的各级学生。学生的科系范围广到令我惊讶，年龄的范围也大到令我惊讶。对于深居简出的作家而言，读者群只是一个想象空间，哦，我真有这么多读者吗?! 呵呵，喜欢这种感觉。回信虽然是一种压力，不过这是个甜蜜的负担。

你们常常感谢我带给你们帮助。你们一定不知道，没有你们细心研读我的心血，并且热心写信给我，我无法忍受写作的漫漫孤寂！我可以花三天的时间写一篇序，也可以花一个上午设计一张图。是的，我愿意！我对拥有一群可爱可敬的读者感到骄傲。

目 錄

（* 表示本版新增內容）

第 0 章

你一定要知道（导读）

这本书适合谁

深入浅出 MFC 是一本介绍MFC（Microsoft Foundation Classes）程序设计技术的书籍。对于Windows 应用软件的开发感到兴趣，并欲使用Visual C++ 整合环境的视觉开发工具，以MFC 为程序基础的人，都可以从此书获得最根本最重要的知识与实例。

如果你是一位对Application Framework 和对象导向（Object Oriented）观念感兴趣的技术狂热份子，想知道神秘的Runtime Type Information、Dynamic Creation、Persistence、Message Mapping 以及Command Routing 如何实作，本书能够充分满足你。事实上，依我之见，这些核心技术与彻底学会操控MFC 乃同一件事情。

全书分为四篇：

第一篇【勿在浮砂筑高台】提供进入MFC 核心技术以及应用技术之前的所有技术基础，包括：

- Win32 程序观念：message based, event driven, multitasking, multithreading, console programming。
- C++ 重要技术：类别与对象、this 指针与继承、静态成员、虚拟函数与多态、

模板（template）类别、异常处理（exception handling）。

■ MFC 六大技术之简化仿真（Console 程序）

第二篇【欲善工事先利其器】提供给对Visual C++ 整合环境全然陌生的朋友一个导引。这一篇当然不能取代Visual C++ User's Guide 的地位，但对整个软件开发环境有全盘以及概观性的介绍，可以让初学者迅速了解手上掌握的工具，以及它们的主要功能。

第三篇【浅出MFC 程序设计】介绍一个MFC 程序的生死因果。已经有MFC 程序经验的朋友，不见得不会对本篇感到惊艳。根据我的了解，太多人使用MFC 是「只知道这么做，不知道为什么」；本篇详细解释MFC 程序之来龙去脉，为初入MFC 领域的读者奠定扎实的基础。说不定本篇会让你有醍醐灌顶之感。

第四篇【深入MFC 程序设计】介绍各式各样MFC 技术。「只知其然不知其所以然」的不良副作用，在程序设计的企图进一步开展之后，愈来愈严重，最终会行不得也！那些最困扰我们的MFC 宏、MFC 常数定义，不得一窥堂奥的MFC 黑箱操作，在本篇陆续曝光。本篇将使您高喊：Eureka！

> 阿基米德在洗澡时发现浮力原理，高兴得来不及穿上裤子，跑到街上大喊：Eureka（我找到了）。

范例程序方面，第三章有数个Console 程序（DOS-like 程序，在Windows 系统的DOS Box 中执行），仿真并简化Application Framework 六大核心技术。另外，全书以一个循序渐进的Scribble 程序（Visual C++ 所附范例），从第七章开始，分章探讨每一个MFC 应用技术主题。第13 章另有三个程序，示范Multi-View 和Multi-Document 的情况。14 章~16 章是第二版新增内容，主题分别是MFC 多线程程序设计、Custom AppWizard、以及如何使用Component Gallery 提供的ActiveX controls 和components。

你需要什么技术基础

从什么技术层面切入Windows 软件开发领域？C/SDK？抑或C++/MFC？这一直是个引起争议的论题。就我个人观点，C++/MFC 程序设计必须跨越四大技术障碍：

1. 对象导向观念与C++ 语言。
2. Windows 程序基本观念（程序进入点、消息流动、窗口函数、callback...）。
3. Microsoft Foundation Classes（MFC）本身。
4. Visual C++ 整合环境与各种开发工具（难度不高，但需熟练）。

换言之，如果你从未接触C++，千万不要阅读本书，那只会打击你学习新技术的信心而已。如果已接触过C++ 但不十分熟悉，你可以一边复习C++ 一边学习MFC，这也是我所鼓励的方式（很多人是为了使用MFC 而去学习C++ 的）。C++ 语言的继承（inheritance）特性对于我们使用MFC 尤为重要，因为使用MFC 就是要继承各个类别并为己用。所以，你应该对C++ 的继承特质（以及虚拟函数，当然）多加体会。我在第 2 章安排了一些C++ 的必要基础。我所挑选的题目都是本书会用到的技术，而其深度你不见得能够在一般C++ 书籍中发现。

如果你有C++ 语言基础，但从未接触过Win16 或Win32 程序设计，只在DOS 环境下开发过软件，我在第 1 章为你安排了一些Win32 程序设计基础。这个基础至为重要，只会在各个Wizards 上按来按去，却不懂所谓message loop 与window procedure 的人，不可能搞定Windows 程序设计-- 不管你用的是MFC 或OWL 或Open Class Library，不管你用的是Visual C++ 或Borland C++ 或VisualAge C++。

你需要什么软硬件环境

一套Windows 95（或Windows NT）操作系统当然是必须的，中英文皆可。此外，你需要一套Visual C++ 32 位版。目前的最新版本是Visual C++ 5.0，也是我使用的版本。

硬件方面，只要能跑上述两种操作系统就算过关。内存（RAM）是影响运作速度的主因，多多益善。厂商宣称16MB RAM 是一个能够使你工作舒适的数字，但我因此怀疑「舒适」这个字眼的定义。写作本书时我的软硬件环境是：

- Pentium 133
- 96M RAM
- 2GB 硬盘
- 17 寸显示器。别以为显示器和程序设计没有关系。大尺寸屏幕使我们一次看多一点东西，不必在Visual C++ 整合环境所提供的密密麻麻的画面上卷来卷去。
- Windows 95（中文版）
- Visual C++ 5.0

让我们使用同一种语言

要在计算机书籍不可或缺的英文术语与流利顺畅的中文解说之间保持一个平衡，是多么不容易的一件事。我曾经以为我通过了最大的考验，但每次总有新鲜事儿发生。是该叫class好呢？还是叫「类别」好？该叫object 好呢？还是叫「对象」好？framework 难道该译为框架吗？Document 译为「文件」，可也，可View 是什么碗糕？我很伤脑筋耶。考虑了这本书的潜在读者所具备的技术基础与教育背景之后，原谅我，不喜欢在中文书中看到太多英文字的朋友，你终究还是在这本书上看到了不少原文名词。只有几已统一化、没有异议、可以望文生义的中文名词，我才使用。
虽然许多名词已经耳熟能详，我想我还是有必要把它们界定一下：

API - Application Programming Interface。系统开放出来，给程序员使用的接口，就是API。一般人的观念中API 是指像C 函数那样的东西，不尽然！DOS 的中断向量（interrupt vector）也可以说是一种API，OLE Interface（以C++ 类别的形式呈现）也可以说是一种API。不是有人这么说吗：MFC 势将成为Windows 环境上标准的C++ API（我个人认为这句话已成为事实）。

SDK - Software Development Kit，原指软件开发工具。每一套环境都可能有自己的SDK，例如Phar Lap 的386|DOS Extender 也有自己的SDK。在Windows 这一领域，SDK 原是指Microsoft 的软件开发工具，但现在已经变成一个一般性名词。凡以Windows raw API 撰写的程序我们通常也称为SDK 程序。也有人把Windows API 称为SDK API。Borland 公司的C++ 编译器也支持相同的SDK API（那当然，因为Windows只有一套）。本书如果出现「SDK 程序」这样的名词，指的就是以Windows raw API 完成的程序。

MFC - Microsoft Foundation Classes 的缩写，这是一个架构在Windows API 之上的C++ 类别库（C++ Class Library），意图使Windows 程序设计过程更有效率，更符合物件导向的精神。MFC 在争取成为「Windows 类别库标准」的路上声势浩大。Symantec C++以及WATCOM C/C++ 已向微软取得授权，在它的软件开发平台上供应MFC。Borland C++ 也可以吃进MFC 程序代码-- 啊，OWL 的地位益形尴尬了。

OWL - Object Windows Library 的缩写，这也是一个具备Application Framework 架势的C++ 类别库，附含在Borland C++ 之中。

Application Framework - 在对象导向领域中，这是一个专有名词。关于它的意义，本书第5章有不少介绍。基本上它可以说是一个更有凝聚力，关联性更强的类别库。并不是每一套C++ 类别库都有资格称为Application Framework，不过MFC 和OWL 都可入列，IBM 的Open Class Library 也是。Application Framework 当然不一定得是C++ 类别库，Java 和Delphi 应该也都称得上。

为使全书文字流畅精简，我用了一些缩写字：

API - Application Programming Interface
DLL - Dynamic Link Library
GUI - Graphics User Interface
MDI - Multiple Document Interface

MFC - Microsoft Foundation Class

OLE - Object Linking & Embedded

OWL - Object Windows Library

SDK - Software Development Kit

SDI - Single Document Interface

UI - User Interface

WinApp : Windows Application

以下是本书使用之中英文名词对照表：

control	控制组件，如 Edit、ListBox、Button...。
drag & drop	拖放（鼠标左键按下，选中图标后拖动，然后放开）
Icon	图标（窗口缩小化后的小图样）
linked-list	串行
listbox	列表框、列表清单
notification	通告消息（发生于控制组件）
preemptive	强制性、先占式、优先权式
process	进程（一个执行起来的程序）
queue	队列
template	template C++ 有所谓的class template，一般译为类别模板； Windows 有所谓的dialog template，我把它译为对话框模板； MFC 有所谓的Document Template，我没有译它（其义请见第 7 章和第 8 章）
window class	窗口类别（不是一种 C++ 类别）
window focus	窗口焦点（拥有焦点之窗口，将可获得键盘输入）

类别	class
对象	object
构造式	constructor
析构式	destructor
运算子	operator
改写	override
多载	overloading，亦有他书译为「过荷」
封装	Encapsulation
继承	Inheritance
动态绑定	Dynamic Binding，亦即后期绑定（late binding）
虚拟函数	virtual function
多态	Polymorphism，亦有他书译为「同名异式」
成员函数	member function
成员变量	data member，亦有他书译为「数据成员」
基础类别	Base Class，亦即父类别
衍生类别	Derived Class，亦即子类别

另有一些名词很难说用什么中文字眼才好。例如"double click"，有时候我写「双击」，有时候我写「以鼠标快按两下」；而"click"，我可能用「选按」「选择」「以鼠标按一下」等字眼，完全视上下文而定。虽没有统一，但您在文字中一定会了解我的意思。我期盼写出一本读起来很顺又绝对不会让你误解意思的中文计算机书。还有些名词在某些场合使用中文而在某些场合使用原文，例如Class（类别）和Object（对象）和Menu（菜单），为的也是使上下文阅读起来舒服一些。这些文字的使用都肇基于我个人对文字的认知以及习惯，如果与您的风格不符，深感抱歉。我已尽力在一个处处需要英文名词的领域中写一本尽可能阅读顺畅的中文技术书籍。

本书符号习惯

斜体字表示函数、常数、变量、语言保留字、宏、识别码等等，例如：

CreateWindow	这是Win32 函数
strtok	这是C Runtime 函数库的函数
WM_CREATE	这是Windows 消息
ID_FILE_OPEN	这是资源识别码（ID）
CDocument::Serialize	这是MFC 类别的成员函数
m_pNewViewClass	这是MFC 类别的成员变量
BEGIN_MESSAGE_MAP	这是MFC 宏
public	这是C++ 语言保留字

当我解释程序操作步骤时，如果使用中括号，例如【File/New】，表示选按File 菜单中的New 命令项。或者用来表示一个对话窗，例如我写：【New Project】对话窗。

磁盘内容与安装

本书光盘片内含书中所有的范例程序，包括源代码与EXE 档。中介文件（如.OBJ 和.RES 等）并未放入。所有程序都可以在Visual C++ 5.0 整合环境中制作出来。安装方式很简单（根本没有什么安装方式）：利用DOS 外部指令，XCOPY，把整个光盘片拷贝到你的硬盘上即是了。

范例程序说明

- Generic（第 1 章）：这是一个Win32 程序，主要用意在让大家了解Win32 程式的基本架构。
- Jbackup（第 1 章）：这是一个Win32 console 程序，主要用意在让大家了解

Visual C++ 整合环境中也可以做很单纯的DOS-like 程序，而且又能够使用Win32 API。

- MFCcon（第１章）：这是一个很简单的MFC console 程序，主要用意在让大家了解Visual C++ 整合环境中也可以做很单纯的DOS-like 程序，而且又能够使用MFC classes。

- MltiThrd（第１章）：这是一个Win32 多线程程序，示范如何以*CreateThread* 做出多个执行线程，并设定其虚悬状态、优先权、重新激活状态、睡眠状态。

- Frame1~8（第３章）：这些都是console 程序（所谓DOS-like 程序），仿真并简化Application Framework 的六大核心技术。只有¡° Persistence¡ 技术未仿真出来，因为那牵扯太广。

  - ☐ Frame1：仿真MFC 阶层架构以及application object
  - ☐ Frame2：仿真MFC 的*WinMain* 四大动作流程
  - ☐ Frame3：仿真*CRuntimeClass* 以及*DYNAMIC* 宏，组织起所谓的类别型录网
  - ☐ Frame4：仿真*IsKindOf*（执行时期对象类别的鉴识能力，也就是所谓的RTTI）
  - ☐ Frame5：仿真Dynamic Creation（MFC 2.5 的作法）（在本新版中已拿掉）
  - ☐ Frame6：仿真Dynamic Creation（MFC 4.x 的作法）
  - ☐ Frame7：仿真Message Map
  - ☐ Frame8：仿真Command Routing

- Hello 范例程序（第６章）：首先以最小量（两个）MFC 类别，完成一个最最简单的MFC 程序。没有Document/View -- 事实上这正是MFC 1.0 版的应用程序风貌。本例除了提供你对MFC 程序的第一印象，也对类别的静态成员函式应用于callback 函数做了一个示范。每有窗口异动（产生*WM_PAINT*），就有一个"Hello MFC" 字符串从天而降。此外，也示范了空闲时间（idle time）的处理。

■ Scribble Step0~Step5 ： ¡¨ Scribble¡¨ 范例之于MFC 程序设计， 几乎相当于¡§ Generic¡¨ 范例之于SDK 程序设计。微软的「官方手册」Visual C++ Class Library User's Guide 全书即以本例为主轴，介绍这个可以让你在窗口中以鼠标左键绘图的程序。Scribble 程序共有Step1~Step7，七个阶段的所有源代码都可以在Visual C++ 5.0 的\DEVSTUDIO\VC\MFC\SAMPLES\SCRIBBLE 目录中找到。本书只采用Step1~Step5，并增列Step0。Step6 是OnLine Help 的制作，Step7 是OLE Server 的制作，这两个主题本书从缺。

■ Scribble Step0－由MFC AppWizard 做出来的空壳程序，也就是所谓的MFC 骨干程序。完整源代码列于第 4 章「东圈西点完成程序骨干」一节。完整解说出现在第 7 章。

■ Scribble Step1－具备Document/View 架构（第 8 章）：本例主旨在加上资料处理与显示的能力。这一版的窗口没有卷动能力。同一文件的两个显示窗口也没有能够做到实时更新的效果。当你在窗口甲改变文件内容，对映至同一文件的窗口乙并不会实时修正内容，必须等*WM_PAINT* 产生（例如拉大窗口）。

这个版本已具备打印与预视能力，但并非「所见即所得」（What You See Is What You Get），打印结果明显缩小，这是因为映射模式采用*MM_TEXT*。15寸监视器的 640 个图素换到300dpi 上才不过两英寸多一点。

我们可以在这个版本中学习以AppWizard 制作骨干，并大量运用ClassWizard 为我们增添消息处理函数；也可以学习如何设计Document，如何改写*CView*::*OnDraw* 和 *CDocument*::*Serialize*，这是两个极端重要之虚拟函数。

■ Scribble Step2－修改使用者接口（第 9 章）：这个版本变化了菜单，使程序多了笔宽设定功能。由于菜单的变化，也带动了工具栏与状态列的变化。

从这个版本中我们可以学习如何使用资源编辑器，制作各式各样的程序资源。为了把菜单命令处理函数放置在适当的类别之中，我们需要深入了解所谓的Message

Mapping 和Command Routing。

■ Scribble Step3－增加「笔划宽度对话框」（第10 章）：这个版本做出「画笔宽度对话框」，使用者可以在其中设定细笔宽度和粗笔宽度。预设的细笔为两个图素（pixel）宽，粗笔为五个图素宽。

从这个版本中可以学习如何以对话框编辑器设计对话框模板，以ClassWizard 增设对话盒处理函数，以及如何以MFC 提供的DDX/DDV 机制做出对话框控制组件（control）的内容传递与内容查核。DDX（Dialog Data eXchange）的目的在简化应用程序取得控制组件内容的过程，DDV（Dialog Data Validation）的目的则在加强应用程序对控制组件内容之数值合理化检查。

■ Scribble Step4－加强显示能力- 滚动条与分裂窗口（第11 章）：Scribble 可以对同一份Document 产生一个以上的Views，但有一个缺点亟待克服，那就是你在窗口Ａ的绘图动作不能实时影响窗口Ｂ -- 即使它们是同一份资料的一体两面!

Step4 解决上述问题。主要关键在于我们必须想办法通知所有同血源（同一份Document）的兄弟（各个Views），让它们一起行动。但因此却必须多考虑一个情况：当使用者的一个鼠标动作可能引发许多许多程序绘图动作时，绘图效率就变得非常重要。因此在考量如何加强显示能力时，我们就得设计所谓的「必要绘图区」，也就是所谓的Invalidate Region（不再适用的区域）。事实上每当使用者开始绘图（增加新的线条），程序可以设定「必要绘图区」为：该线条之最小外围四方形。为了记录这项数据，Step1 所设计并延续至今的Document 数据结构必须改变。

Step1 的View 窗口有一个缺点：没有滚动条。新版本加上了垂直和水平滚动条，此外它也示范一种所谓的分裂窗口（Splitter）。

■ Scribble Step5－打印与预视（第12 章）：Step1 已有打印和预视能力，这当然归功于*CScribbleView::OnDraw*。现在要加强的是更细致的打印能力，包括表

头、表尾、页码、映射模式等等。坐标系统（也就是映射模式，Mapping Mode）的选择，关系到是否能够「所见即所得」。为了这个目的，必须使用能够反应真实世界之尺寸（如英寸、公分）的映像模式，本例使用 MM_LOENGLISH，每个逻辑单位0.01 英寸。

我们也在此版中学习如何设定文件的大小。有了大小，才能够在打印时做分页动作。

- Graph 范例程序（第13 章）：这个程序示范如何在静态分裂窗口的不同窗口中，以不同的方式（本例为长条图、点状图和文字形式）显示同一份资料。
- Text 范例程序（第13 章）：这个程序示范如何在同一份Document 的各个「同源view 窗口」中，以不同的显示方法表现同一份资料，做到一体数面。
- Graph2 范例程序（第13 章）：这个程序示范如何为程序加上第二个Document 类型。其间关系到新的Document，新的View，新的UI。
- MltiThrd 范例程序（第14 章）：这是第 1 章的同名程序的MFC 版。我只示范MFC 多线程程序的架构，原Mltithrd 程序的绘图部份留给读者练习。
- Top 范例程序（第15 章）：示范如何量身定做一个属于自己的AppWizard。我的这个Top Studio AppWizard 架在系统的MFC AppWizard 之上，增加一个开发步骤，询问程序员名称及其简单声明，然后就会在每一个产生出来的原始码文件最前端加上一段固定格式的说明文字。
- ComTest 范例程序（第16 章）：此程序示范使用Component Gallery 中的三个components：Splash Screen、SysInfo、Tip Of The Day。
- OcxTest 范例程序（第16 章）：此程序示范使用Component Gallery 中的Grid ActiveX control。

与前版本之差异

深入浅出MFC 第二版与前一版本之重大差异在于：

1. 软件工具由Visual C++ 4.0 改为Visual C++ 5.0，影响所及，第４章「Visual C++ - 整合性软件开发环境」之内容改变极大。全书之中有关于MFC 内部动作逻辑及其源代码的变动不多，因为Visual C++ 5.0 中的MFC 版本还维持在4.2。
2. 第１章增加Console 程序设计，以及Win32 多线程程序实例Mltithrd。
3. 第２章增加「四种不同的对象生存方式」一节。
4. 第３章去除原有之Frame5 程序（该程序以MFC 2.5 的技术仿真Dynamic Creation）。
5. 第４章全部改为Visual C++ 5.0 使用画面，并在最后增加一节「Console 程序的项目管理」。
6. 第６章增加「奇怪的窗口类别名称Afx:x:y:z:w」一节，以及增加Hello 程序对idle time 的处理。
7. 增加14~16 三章。
8. 附录A增加<无责任书评/侯捷先生> 的「MFC 四大天王」一文。
9. 附录D由原先之「OWL 程序设计一览」，改为「以MFC 重建DBWIN」。

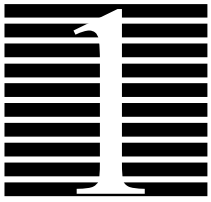
本书第一版之Scribble 程序自step1（加了*CStroke*）之后，即无法在Visual C++ 4.2 和Visual C++ 5.0 上顺利编译。原因出在VC++ 4.2 和VC++ 5.0 似乎未能支持"forward declaration of data structure class"（但是我怀疑VC++ 怎么会走退步？是不是有什么选项可以设定）。无论如何，只要将*CStroke* 的声明搬移到SCRIBBLEDOC.H 的最前面，然后再接续*CScribbleDoc* 的声明，即可顺利编译。请阅读本书第８章「*CScribbleDoc* 的修改」一节之中于SCRIBBLEDOC.H 源代码列表后的一段说明（#477 页）。

如何联络作者

我非常乐意和本书的所有读者沟通，接受您对本书以及对我的指正和建议。请将沟通内容局限在对书籍、对知识的看法，以及对本书误谬之指正和建议上面，请勿要求我为您解决技术问题（例如您的程序臭虫或您的项目瓶颈）。如果只是单纯地想和我交个朋友聊聊天，我更倍感荣幸。

我的Email 地址是jjhou@ccca.nctu.edu.tw

我的永久通讯址是新竹市建中一路39 号13 楼之二（FAX：03-5733976）



勿在浮砂筑高台

第 1 章

Win32 基本程序观念

程序设计领域里，每一个人都想飞。

但是，还没学会走之前，连跑都别想！

虽然这是一本深入讲解MFC 程序设计的书，我仍坚持要安排这第一章，介绍Win32 的基本程序设计原理（也就是所谓的SDK 程序设计原理）。

从来不曾学习过在「事件驱动（event driven）系统」中撰写「以消息为基础（message based）之应用程序」者，能否一步跨入MFC 领域，直接以application framework 开发Windows 程序，我一直抱持怀疑的态度。虽然有了MFC（或任何其它的application framework），你可以继承一整组类别，从而快速得到一个颇具规模的程序，但是Windows 程序的运作本质（Message Based，Event Driven）从来不曾也不会改变。如果你不能了解其髓，空有其皮其肉或其骨，是不可能有所精进的，即使能够操控wizard，充其量却也只是个puppet，对于手上的程序代码，没有自主权。

我认为学习MFC 之前，必要的基础是，对于Windows 程序的事件驱动特性的了解（包括消息的产生、获得、分派、判断、处理），以及对C++ 多态（polymorphism）的精确体会。本章所提出的，是我对第一项必要基础的探讨，你可以从中获得关于Windows 程序的诞生与死亡，以及多任务环境下程序之间共存的观念。至于第二项基础，将由第二章为你夯实。

让我再强调一遍，本章就是我认为Windows 程序设计者一定要知道的基础知识。一个连这些基础都不清楚的人，不能要求自己冒冒然就开始用Visual C++、用MFC、用对象导向的方式去设计一个你根本就不懂其运作原理的程序。

还没学会走之前，不要跑！

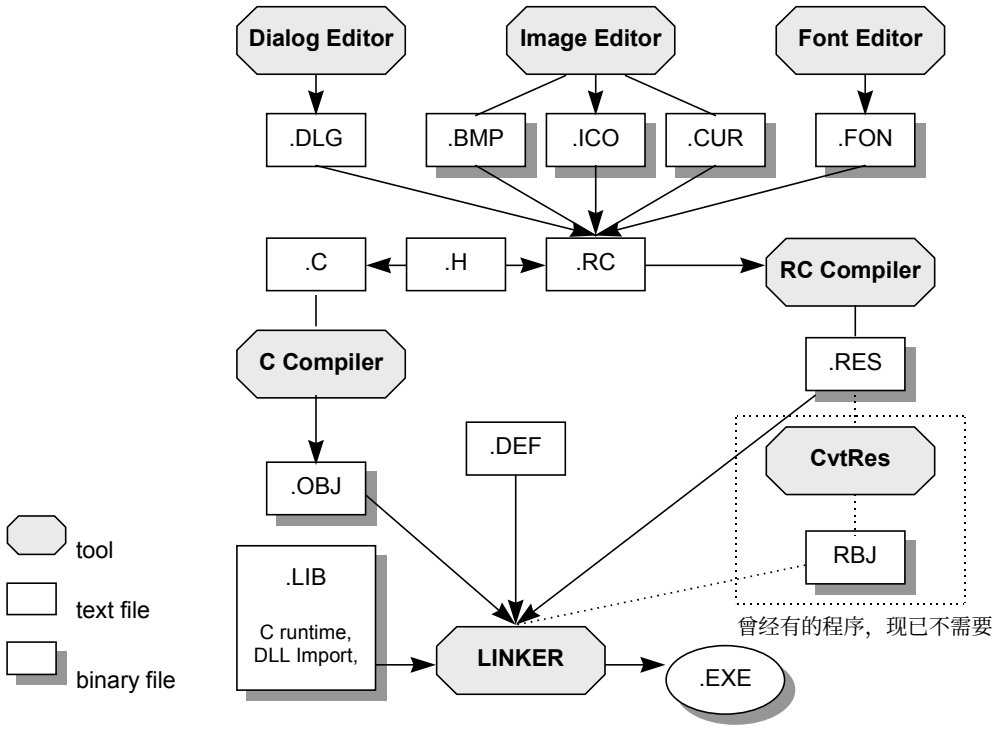


图 1-1 一个32位Windows SDK 程序的开发流程

Win32 程序开发流程

Windows 程序分为「程序代码」和「UI（User Interface）资源」两大部份，两部份最后以RC编译器整合为一个完整的EXE 文件（图1-1）。所谓UI 资源是指功能菜单、对话框外貌、程序图标、光标形状等等东西。这些UI 资源的实际内容（二进制代码）系借助各种工具产生，并以各种扩展名存在，如.ico、.bmp、.cur 等等。程序员必须在一个所谓的资源描述档（.rc）中描述它们。RC 编译器（RC.EXE）读取RC 档的描述后将所有UI 资源档集中制作出一个.RES 档，再与程序代码结合在一起，这才是一个完整的Windows 可执行档。

需要什么函数库（.LIB）

众所周知Windows 支持动态联结。换句话说，应用程序所调用的Windows API 函数是在「执行时期」才联结上的。那么，「联结时期」所需的函数库做什么用？有哪些？并不是延伸档名为.dll 者才是动态联结函数库（DLL，Dynamic Link Library），事实上.exe、.dll、.fon、.mod、.drv、.ocx 都是所谓的动态联结函数库。

Windows 程序调用的函数可分为C Runtimes 以及Windows API 两大部份。早期的C Runtimes 并不支持动态联结，但Visual C++ 4.0 之后已支持，并且在32 位操作系统中已不再有small/medium/large 等内存模式之分。以下是它们的命名规则与使用时机:

- LIBC.LIB - 这是C Runtime 函数库的静态联结版本。
- MSVCRT.LIB - 这是C Runtime 函数库动态联结版本（MSVCRT40.DLL）的 import 函数库。如果联结此一函数库，你的程序执行时必须有MSVCRT40.DLL 在场。

另一组函数，Windows API，由操作系统本身（主要是Windows 三大模块GDI32.DLL 和USER32.DLL 和KERNEL32.DLL）提供（注）。虽说动态联结是在执行时期才发生「联

结」事实，但在联结时期，联结器仍需先为调用者（应用程序本身）准备一些适当的信息，才能够在执行时期顺利「跳」到DLL 执行。如果该API 所属之函数库尚未加载，系统也才因此知道要先行加载该函数库。这些适当的信息放在所谓的「import 函数库」中。32 位Windows 的三大模块所对应的import 函数库分别为GDI32.LIB 和USER32.LIB 和KERNEL32.LIB。

> 注：谁都知道，Windows 95 是16/32 位的混合体，所以旗下除了32 位的GDI32.DLL、USER32.DLL 和KERNEL32.DLL， 又有16 位的GDI.EXE 、USER.EXE 和KRNL386.EXE。32 位和16 位两组DLLs 之间以所谓的thunking layer 沟通。站在纯粹APIs 使用者的立场，目前我们不必太搭理这个事实。

Windows 发展至今，逐渐加上的一些新的API 函数（例如Common Dialog、ToolHelp）并不放在GDI 和USER 和KERNEL 三大模块中，而是放在诸如COMMDLG.DLL、TOOLHELP.DLL 之中。如果要使用这些APIs，联结时还得加上这些DLLs 所对应的import 函数库，诸如COMDLG32.LIB 和TH32.LIB。

很快地，在稍后的范例程序¡ §Generic¡ ¨ 的makefile 中，你就可以清楚看到联结时期所需的各式各样函数库（以及各种联结器选项）。

需要什么头文件（.H）

所有Windows 程序都必须包含WINDOWS.H。早期这是一个巨大的头文件，大约有5000 行左右，Visual C++ 4.0 已把它切割为各个较小的文件，但还以WINDOWS.H 总括之。除非你十分清楚什么API 动作需要什么头文件，否则为求便利，单单一个WINDOWS.H 也就是了。

不过，WINDOWS.H 只照顾三大模块所提供的API 函数，如果你用到其它system DLLs，例如COMMDLG.DLL 或MAPI.DLL 或TAPI.DLL 等等，就得包含对应的头文件，例如COMMDLG.H 或MAPI.H 或TAPI.H 等等。

以消息为基础，以事件驱动之（message based， event driven）

Windows 程序的进行系依靠外部发生的事件来驱动。换句话说，程序不断等待（利用一个while 回路），等待任何可能的输入，然后做判断，然后再做适当的处理。上述的「输入」是由操作系统捕捉到之后，以消息形式（一种数据结构）进入程序之中。操作系统如何捕捉外围设备（如键盘和鼠标）所发生的事件呢？噢，USER 模块掌管各个外围的驱动程序，它们各有侦测回路。

如果把应用程序获得的各种「输入」分类，可以分为由硬件装置所产生的消息（如鼠标移动或键盘被按下），放在系统队列（system queue）中，以及由Windows 系统或其它Windows 程序传送过来的消息，放在程序队列（application queue）中。以应用程序的眼光来看，消息就是消息，来自哪里或放在哪里其实并没有太大区别，反正程序调用 *GetMessage* API 就取得一个消息，程序的生命靠它来推动。所有的GUI 系统，包括UNIX 的X Window 以及OS/2 的Presentation Manager，都像这样，是以消息为基础的事件驱动系统。

可想而知，每一个Windows 程序都应该有一个回路如下：

```
MSG msg;
while (GetMessage(&msg, NULL, NULL, NULL)) {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
}
// 以上出现的函数都是Windows API 函数
```

消息，也就是上面出现的*MSG* 结构，其实是Windows 内定的一种资料格式:

```
/* Queued message structure */
typedef struct tagMSG
{
    HWND hwnd;
    UINT message; // WM_xxx，例如WM_MOUSEMOVE，WM_SIZE...
    WPARAM wParam;
    LPARAM lParam;
    DWORD time;
    POINT pt;
} MSG;
```

接受并处理消息的主角就是窗口。每一个窗口都应该有一个函数负责处理消息，程序员必须负责设计这个所谓的「窗口函数」（window procedure，或称为window function）。如果窗口获得一个消息，这个窗口函数必须判断消息的类别，决定处理的方式。

以上就是Windows 程序设计最重要的观念。至于窗口的产生与显示，十分简单，有专门的API 函数负责。稍后我们就会看到Windows 程序如何把这消息的取得、分派、处理动作表现出来。

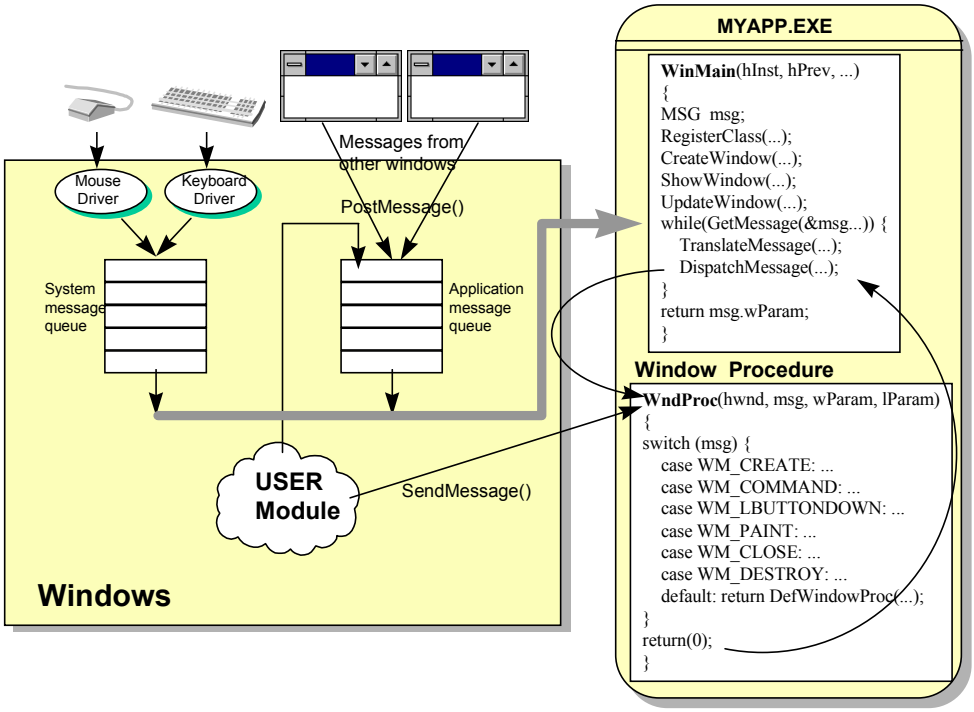


图1-2 Windows 程序的本体与操作系统之间的关系。

一个具体而微的Win32 程序

许多相关书籍或文章尝试以各种方式简化Windows 程序的第一步，因为单单一个Hello 程序就要上百行，怕把大家吓坏了。我却宁愿各位早一点接触正统写法，早一点看到全貌。Windows 的东西又多又杂，早一点一窥全貌是很有必要的。而且你会发现，经过有条理的解释之后，程序代码的多寡其实构不成什么威胁（否则无字天书最适合程序员阅读）。再说，上百行程序代码哪算得了什么！

你可以从图1-2 得窥Win32 应用程序的本体与操作系统之间的关系。Win32 程序中最具代表意义的动作已经在该图显示出来，完整的程序代码展示于后。本章后续讨论都围绕着此一程序。

稍后会出现一个makefile。关于makefile 的语法，可能已经不再为大家所熟悉了。我想我有必要做个说明。

所谓makefile，就是让你能够设定某个文件和某个文件相比-- 比较其产生日期。由其比较结果来决定要不要做某些你所指定的动作。例如：

```
generic.res : generic.rc generic.h
rc generic.rc
```

意思就是拿冒号（:）左边的generic.res 和冒号右边的generic.rc 和generic.h 的文件日期相比。只要右边任一文件比左边的文件更新，就执行下一行所指定的动作。这动作可以是任何命令列动作，本例为rc generic.rc。

因此，我们就可以把不同文件间的依存关系做一个整理，以makefile 语法描述，以产生必要的编译、联结动作。makefile 必须以NMAKE.EXE（Microsoft 工具）或MAKE.EXE（Borland 工具）处理之，或其它编译器套件所附的同等工具（可能也叫做MAKE.EXE）处理之。

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

```
#0001  # filename : generic.mak
#0002  # make file for generic.exe (Generic Windows Application)
#0003  # usage : nmake generic.mak (Microsoft C/C++ 9.00) (Visual C++ 2.x)
#0004  # usage : nmake generic.mak (Microsoft C/C++ 10.00) (Visual C++ 4.0)
#0005
#0006  all: generic.exe
#0007
#0008  generic.res : generic.rc generic.h
#0009      rc generic.rc
#0010
#0011  generic.obj : generic.c generic.h
#0012      cl -c -W3 -Gz -D_X86_ -DWIN32 generic.c
#0013
#0014  generic.exe : generic.obj generic.res
#0015      link /MACHINE:I386 -subsystem:windows generic.res generic.obj \
#0016           libc.lib kernel32.lib user32.lib gdi32.lib
```

Generic.h

```
#0001  //--------------------------------------------------------------------
#0002  // 档名 : generic.h
#0003  //--------------------------------------------------------------------
#0004  BOOL InitApplication(HANDLE);
#0005  BOOL InitInstance(HANDLE, int);
#0006  LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
#0007  LRESULT CALLBACK About(HWND, UINT, WPARAM, LPARAM);
```

Generic.c （粗体代表Windows API 函数或宏）

```
#0001  //--------------------------------------------------------------------
#0002 // Generic - Win32 程序的基础写法
#0003 // Top Studio * J.J.Hou
#0004 // 档名: generic.c
#0005 // 作者: 侯俊杰
#0006 // 编译联结: 请参考generic.mak
#0007  //--------------------------------------------------------------------
#0008
#0009  #include <windows.h>   // 每一个 Windows 程序都需要包含此档
#0010  #include "resource.h"  // 內含各个 resource IDs
#0011  #include "generic.h"   // 本程序之含入档
#0012
#0013  HINSTANCE _hInst;     // Instance handle
#0014  HWND      _hWnd;
```

```
#0015
#0016  char _szAppName[] = "Generic";    //程序名称
#0017  char _szTitle[]   = "Generic Sample Application"; //窗口标题
#0018
#0019  //--------------------------------------------------------------------
#0020  // WinMain - 程序进入点
#0021  //--------------------------------------------------------------------
#0022  int CALLBACK WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
#0023                       LPSTR lpCmdLine,    int nCmdShow)
#0024  {
#0025  MSG msg;
#0026
#0027    UNREFERENCED_PARAMETER(lpCmdLine);     // 避免编译时的警告
#0028
#0029    if (!hPrevInstance)
#0030       if (!InitApplication(hInstance))
#0031          return (FALSE);
#0032
#0033    if (!InitInstance(hInstance, nCmdShow))
#0034       return (FALSE);
#0035
#0036    while (GetMessage(&msg, NULL, 0, 0)) {
#0037       TranslateMessage(&msg);
#0038       DispatchMessage(&msg);
#0039    }
#0040
#0041    return (msg.wParam); //传回 PostQuitMessage 的参数
#0042  }
#0043  //--------------------------------------------------------------------
#0044  // InitApplication - 注册窗口类别
#0045  //--------------------------------------------------------------------
#0046  BOOL InitApplication(HINSTANCE hInstance)
#0047  {
#0048  WNDCLASS  wc;
#0049
#0050    wc.style        = CS_HREDRAW | CS_VREDRAW;
#0051    wc.lpfnWndProc   = (WNDPROC)WndProc;     //窗口函数
#0052    wc.cbClsExtra    = 0;
#0053    wc.cbWndExtra    = 0;
#0054    wc.hInstance     = hInstance;
#0055    wc.hIcon        = LoadIcon(hInstance, "jjhouricon");
#0056    wc.hCursor       = LoadCursor(NULL, IDC_ARROW);
#0057    wc.hbrBackground = GetStockObject(WHITE_BRUSH);  // 窗口背景颜色
#0058    wc.lpszMenuName  = "GenericMenu";              // .RC 所定义的菜单
#0059    wc.lpszClassName = _szAppName;
#0060
```

```
#0061   return (RegisterClass(&wc));
#0062 }
#0063 //--------------------------------------------------------------------
#0064 // InitInstance - 产生窗口
#0065 //--------------------------------------------------------------------
#0066 BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
#0067 {
#0068   _hInst = hInstance; // 存储为全域变量, 方便使用
#0069
#0070   _hWnd = CreateWindow(
#0071                     _szAppName,
#0072                     _szTitle,
#0073                     WS_OVERLAPPEDWINDOW,
#0074                     CW_USEDEFAULT,
#0075                     CW_USEDEFAULT,
#0076                     CW_USEDEFAULT,
#0077                     CW_USEDEFAULT,
#0078                     NULL,
#0079                     NULL,
#0080                     hInstance,
#0081                     NULL
#0082                    );
#0083
#0084   if (!_hWnd)
#0085      return (FALSE);
#0086
#0087   ShowWindow(_hWnd, nCmdShow); // 显示窗口
#0088   UpdateWindow(_hWnd);        // 送出 WM_PAINT
#0089   return (TRUE);
#0090 }
#0091 //--------------------------------------------------------------------
#0092 // WndProc - 窗口函数
#0093 //--------------------------------------------------------------------
#0094 LRESULT CALLBACK WndProc(HWND hWnd,    UINT message,
#0095                         WPARAM wParam, LPARAM lParam)
#0096 {
#0097 int wmId, wmEvent;
#0098
#0099   switch (message) {
#0100     case WM_COMMAND:
#0101
#0102         wmId    = LOWORD(wParam);
#0103         wmEvent = HIWORD(wParam);
#0104
#0105         switch (wmId) {
#0106           case IDM_ABOUT:
```

```
#0107                DialogBox(_hInst,          // Winmain第一个参数, 应用程序的实例句柄
#0108                          "AboutBox",     // 对话框资源名称
#0109                          hWnd,          // 父窗口
#0110                          (DLGPROC)About  //对话框函数名称
#0111                         );
#0112               break;
#0113
#0114           case IDM_EXIT:
#0115               // 使用者想结束程序。处理方式与 WM_CLOSE 相同
#0116               DestroyWindow (hWnd);
#0117               break;
#0118
#0119           default:
#0120               return (DefWindowProc(hWnd, message, wParam, lParam));
#0121         }
#0122         break;
#0123
#0124     case WM_DESTROY:  // 窗口已经被摧毁（程序即将结束）。
#0125         PostQuitMessage(0);
#0126         break;
#0127
#0128     default:
#0129         return (DefWindowProc(hWnd, message, wParam, lParam));
#0130   }
#0131   return (0);
#0132 }
#0133 //--------------------------------------------------------------------
#0134 // About -对话框函数
#0135 //--------------------------------------------------------------------
#0136 LRESULT CALLBACK About(HWND hDlg,     UINT message,
#0137                         WPARAM wParam, LPARAM lParam)
#0138 {
#0139   UNREFERENCED_PARAMETER(lParam);      // 避免編译时的警告
#0140
#0141   switch (message) {
#0142     case WM_INITDIALOG:
#0143          return (TRUE);     // TRUE 表示我已处理过这个消息
#0144
#0145     case WM_COMMAND:
#0146          if (LOWORD(wParam) == IDOK
#0147               || LOWORD(wParam) == IDCANCEL) {
#0148             EndDialog(hDlg, TRUE);
#0149             return (TRUE);  // TRUE 表示我已处理过这个消息
#0150          }
#0151          break;
#0152   }
```

```
#0153   return (FALSE); // FALSE 表示我没有处理这个消息
#0154 }
```

Generic.rc

```
#0001 //-----------------------------------------------------------------
#0002 // 档名 : generic.rc
#0003 //-----------------------------------------------------------------
#0004 #include "windows.h"
#0005 #include "resource.h"
#0006
#0007 jjhouricon ICON    DISCARDABLE     "jjhour.ico"
#0008
#0009 GenericMenu MENU DISCARDABLE
#0010 BEGIN
#0011     POPUP "&File"
#0012     BEGIN
#0013         MENUITEM "&New",             IDM_NEW, GRAYED
#0014         MENUITEM "&Open...",         IDM_OPEN, GRAYED
#0015         MENUITEM "&Save",            IDM_SAVE, GRAYED
#0016         MENUITEM "Save &As...",      IDM_SAVEAS, GRAYED
#0017         MENUITEM SEPARATOR
#0018         MENUITEM "&Print...",        IDM_PRINT, GRAYED
#0019         MENUITEM "P&rint Setup...",   IDM_PRINTSETUP, GRAYED
#0020         MENUITEM SEPARATOR
#0021         MENUITEM "E&xit",            IDM_EXIT
#0022     END
#0023     POPUP "&Edit"
#0024     BEGIN
#0025         MENUITEM "&Undo\tCtrl+Z",     IDM_UNDO, GRAYED
#0026         MENUITEM SEPARATOR
#0027         MENUITEM "Cu&t\tCtrl+X",      IDM_CUT, GRAYED
#0028         MENUITEM "&Copy\tCtrl+C",     IDM_COPY, GRAYED
#0029         MENUITEM "&Paste\tCtrl+V",    IDM_PASTE, GRAYED
#0030         MENUITEM "Paste &Link",       IDM_LINK, GRAYED
#0031         MENUITEM SEPARATOR
#0032         MENUITEM "Lin&ks...",         IDM_LINKS, GRAYED
#0033     END
#0034     POPUP "&Help"
#0035     BEGIN
#0036         MENUITEM "&Contents",              IDM_HELPCONTENTS, GRAYED
#0037         MENUITEM "&Search for Help On...",  IDM_HELPSEARCH, GRAYED
#0038         MENUITEM "&How to Use Help",        IDM_HELPHELP, GRAYED
#0039         MENUITEM SEPARATOR
#0040         MENUITEM "&About Generic...",       IDM_ABOUT
#0041     END
```

```
#0042 END
#0043
#0044 AboutBox DIALOG DISCARDABLE 22, 17, 144, 75
#0045 STYLE DS_MODALFRAME | WS_CAPTION | WS_SYSMENU
#0046 CAPTION "About Generic"
#0047 BEGIN
#0048     CTEXT "Windows 95", -1,0, 5,144,8
#0049     CTEXT "Generic Application",-1,0,14,144,8
#0050     CTEXT "Version 1.0", -1,0,34,144,8
#0051     DEFPUSHBUTTON "OK", IDOK,53,59,32,14,WS_GROUP
#0052 END
```

程序进入点 WinMain

main 是一般C 程序的进入点：

```
int main(int argc, char *argv[ ], char *envp[ ]);
{
...
}
```

WinMain 则是Windows 程序的进入点：

```
int CALLBACK WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                     LPSTR lpCmdLine, int nCmdShow)
{
...
}
// 在Win32 中CALLBACK 被定义为__stdcall，是一种函数调用习惯，关系到
// 参数挤压到堆栈的次序，以及处理堆栈的责任归属。其它的函数调用习惯还有
// _pascal 和_cdecl
```

当Windows 的「外壳」（shell，例如Windows 3.1 的程序管理员或Windows 95 的文件总管）侦测到使用者意欲执行一个Windows 程序，于是调用加载器把该程序加载，然后调用C startup code，后者再调用*WinMain*，开始执进程序。*WinMain* 的四个参数由操作系统传递进来。

窗口类别之注册与窗口之诞生

一开始，Windows 程序必须做些初始化工作，为的是产生应用程序的工作舞台：窗口。这没有什么困难，因为API 函数*CreateWindow* 完全包办了整个巨大的工程。但是窗口产生之前，其属性必须先设定好。所谓属性包括窗口的「外貌」和「行为」，一个窗口的边框、颜色、标题、位置等等就是其外貌，而窗口接收消息后的反应就是其行为（具体地说就是指窗口函数本身）。程序必须在产生窗口之前先利用API 函数*RegisterClass* 设定属性（我们称此动作为注册窗口类别）。*RegisterClass* 需要一个大型数据结构 *WNDCLASS* 做为参数，*CreateWindow* 则另需要11 个参数。

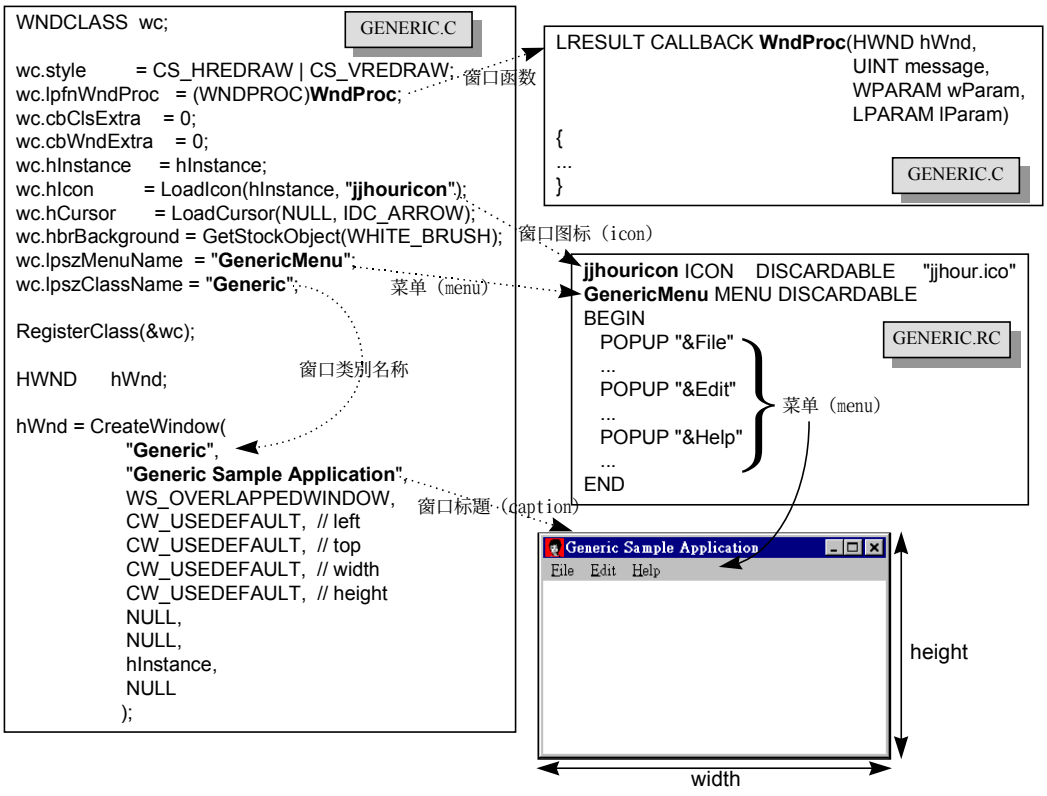


图 1-3 RegisterClass 与 CreateWindow

从图1-3 可以清楚看出一个窗口类别牵扯的范围多么广泛，其中*wc.lpfnWndProc* 所指定的函数就是窗口的行为中枢，也就是所谓的窗口函数。注意，*CreateWindow* 只产生窗口，并不显示窗口，所以稍后我们必须再利用*ShowWindow* 将之显示在屏幕上。又，我们希望先传送个*WM_PAINT* 给窗口， 以驱动窗口的绘图动作， 所以调用*UpdateWindow*。消息传递的观念暂且不表，稍后再提。

请注意，在Generic 程序中，*RegisterClass* 被我包装在*InitApplication* 函数之中，*CreateWindow* 则被我包装在*InitInstance* 函数之中。这种安排虽非强制，却很普遍：

```
int CALLBACK WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
LPSTR lpCmdLine, int nCmdShow)
{
  if (!hPrevInstance)
    if (!InitApplication(hInstance))
        return (FALSE);
   if (!InitInstance(hInstance, nCmdShow))
       return (FALSE);
...
}
//--------------------------------------------------
BOOL InitApplication(HINSTANCE hInstance)
{
 WNDCLASS wc;
...
  return (RegisterClass(&wc));
}
//--------------------------------------------------
BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
{
_hWnd = CreateWindow(...);
...
}
```

两个函数（*InitApplication* 和*InitInstance*）的名称别具意义：

■ 在Windows 3.x 时代，窗口类别只需注册一次，即可供同一程序的后续每一个执行实例（instance）使用（之所以能够如此，是因为所有进程共在一个地址空间中），所以我们把*RegisterClass* 这个动作安排在「只有第一个执行个体才会

进入」的*InitApplication* 函数中。至于此一进程是否是某个程序的第一个执行实例，可由*WinMain* 的参数*hPrevInstance* 判断之；其值由系统传入。

■ 产生窗口， 是每一个执行实例（ instance ） 都得做的动作， 所以我们把*CreateWindow* 这个动作安排在「任何执行实例都会进入」的*InitInstance* 函数中。

以上情况在Windows NT 和Windows 95 中略有变化。由于Win32 程序的每一个执行实例（instance）有自己的地址空间，共享同一窗口类别已不可能。但是由于Win32 系统令*hPrevInstance* 永远为0，所以我们仍然得以把*RegisterClass* 和*CreateWindow* 按旧习惯安排。既符合了新环境的要求，又兼顾到了旧源代码的兼容。

InitApplication 和*InitInstance* 只不过是两个自定函数，为什么我要对此振振有词呢？原因是MFC 把这两个函数包装成*CWinApp* 的两个虚拟成员函数。第 6 章「MFC 程序的生与死」对此有详细解释。

消息循环

初始化工作完成后，*WinMain* 进入所谓的消息循环：

```
while (GetMessage(&msg,...)) {
     TranslateMessage(&msg); // 转换键盘消息
     DispatchMessage(&msg); // 分派消息
}
```

其中的*TranslateMessage* 是为了将键盘消息转化，*DispatchMessage* 会将消息传给窗口函数去处理。没有指定函数名称，却可以将消息传送过去，岂不是很玄？这是因为消息发生之时，操作系统已根据当时状态，为它标明了所属窗口，而窗口所属之窗口类别又已经明白标示了窗口函数（也就是*wc.lpfnWndProc* 所指定的函数），所以*DispatchMessage* 自有脉络可寻。请注意图1-2 所示，*DispatchMessage* 经过USER 模块的协助，才把消息交到窗口函数手中。

消息循环中的*GetMessage* 是Windows 3.x 非强制性（non-preemptive）多任务的关键。应用程序藉由此动作，提供了释放控制权的机会：如果消息队列上没有属于我的消息，我就把机会让给别人。透过程序之间彼此协调让步的方式，达到多任务能力。Windows 95 和

Windows NT 具备强制性（preemptive）多任务能力，不再非靠*GetMessage* 释放CPU 控制权不可，但程序写法依然不变，因为应用程序仍然需要靠消息推动。它还是需要抓消息！

窗口的生命中枢：窗口函数

消息循环中的*DispatchMessage* 把消息分配到哪里呢？它透过USER 模块的协助，送到该窗口的窗口函数去了。窗口函数通常利用*switch*/*case* 方式判断消息种类，以决定处置方式。由于它是被Windows 系统所调用的（我们并没有在应用程序任何地方调用此函数），所以这是一种call back 函数，意思是指「在你的程序中，被Windows 系统调用」的函数。这些函数虽然由你设计，但是永远不会也不该被你调用，它们是为Windows 系统准备的。

程序进行过程中，消息由输入装置，经由消息循环的抓取，源源传送给窗口并进而送到窗口函数去。窗口函数的体积可能很庞大，也可能很精简，依该窗口感兴趣的消息数量多寡而定。至于窗口函数的形式，相当一致，必然是：

```
LRESULT CALLBACK WndProc(HWND hWnd,
                         UINT message,
                         WPARAM wParam,
                         LPARAM lParam)
```

注意，不论什么消息，都必须被处理，所以*switch*/*case* 指令中的*default*: 处必须调用*DefWindowProc*，这是Windows 内部预设的消息处理函数。

窗口函数的*wParam* 和*lParam* 的意义，因消息之不同而异。*wParam* 在16 位环境中是16 位，在32 位环境中是32 位。因此，参数内容（格式）在不同操作环境中就有了变化。

我想很多人都会问这个问题：为什么Windows Programming Modal 要把窗口函数设计为一个call back 函数？为什么不让程序在抓到消息（*GetMessage*）之后直接调用它就好了？原因是，除了你需要调用它，有很多时候操作系统也要调用你的窗口函数（例如当

某个消息产生或某个事件发生）。窗口函数设计为callback 形式，才能开放出一个接口给操作系统叫用。

消息映射（Message Map）的雏形

有没有可能把窗口函数的内容设计得更模块化、更一般化些？下面是一种作法。请注意，以下作法是MFC「消息映射表格」（第９章）的雏形，我所采用的结构名称和变量名称，都与MFC 相同，藉此让你先有个暖身。

首先，定义一个*MSGMAP_ENTRY* 结构和一个*dim* 宏：

```
struct MSGMAP_ENTRY {
                    UINT nMessage;
                    LONG (*pfn)(HWND, UINT, WPARAM, LPARAM);
                     };
#define dim(x) (sizeof(x) / sizeof(x[0]))
```

请注意*MSGMAP_ENTRY* 的第二元素*pfn* 是一个函数指针，我准备以此指针所指之函数处理*nMessage* 消息。这正是对象导向观念中把「资料」和「处理资料的方法」封装起来的一种具体实现，只不过我们用的不是C++ 语言。

接下来，组织两个数组*_messageEntries*[ ] 和*_commandEntries*[ ]，把程序中欲处理的消息以及消息处理例程的关联性建立起来：

```
// 消息与处理例程之对照表格
     struct MSGMAP_ENTRY _messageEntries[] =
     {
          WM_CREATE, OnCreate,
          WM_PAINT, OnPaint,
          WM_SIZE, OnSize,
          WM_COMMAND, OnCommand,
          WM_SETFOCUS, OnSetFocus,
          WM_CLOSE, OnClose,
          WM_DESTROY, OnDestroy,
     } ;      ↑          ↑
          这是消息     这是消息处理例程
```

```
    // Command-ID 与处理例程之对照表格
   struct MSGMAP_ENTRY _commandEntries =
   {
      IDM_ABOUT,      OnAbout,
      IDM_FILEOPEN,   OnFileOpen,
      IDM_SAVEAS,     OnSaveAs,
   } ;    ↑                ↑
    这是WM_COMMAND 命令项这是命令处理例程
```

于是窗口函数可以这么设计：

```
//------------------------------------------------------------------------
// 窗口函数
//------------------------------------------------------------------------
LRESULT CALLBACK WndProc(HWND hWnd,     UINT message,
                         WPARAM wParam, LPARAM lParam)
{
int i;

  for(i=0; i < dim(_messageEntries); i++) {  // 消息对照表
     if (message == _messageEntries[i].nMessage)
        return((*_messageEntries[i].pfn)(hWnd, message, wParam, lParam));
  }
  return(DefWindowProc(hWnd, message, wParam, lParam));
}
//------------------------------------------------------------------------
// OnCommand --专门处理 WM_COMMAND
//------------------------------------------------------------------------
LONG OnCommand(HWND hWnd, UINT message,
               WPARAM wParam, LPARAM lParam)
{
int i;

  for(i=0; i < dim(_commandEntries); i++) {  // 命令项目对照表
     if (LOWORD(wParam) == _commandEntries[i].nMessage)
        return((*_commandEntries[i].pfn)(hWnd, message, wParam, lParam));
  }
  return(DefWindowProc(hWnd, message, wParam, lParam));
}
//------------------------------------------------------------------------
LONG OnCreate(HWND hWnd, UINT wMsg, UINT wParam, LONG lParam)
{
   ...
}
```

```
//--------------------------------------------------------------------
LONG OnAbout(HWND hWnd, UINT wMsg, UINT wParam, LONG lParam)
{
...
}
//--------------------------------------------------------------------
```

这么一来，*WndProc* 和*OnCommand* 永远不必改变，每有新要处理的消息，只要在*_messageEntries*[ ] 和*_commandEntries*[ ] 两个数组中加上新元素，并针对新消息撰写新的处理例程即可。

这种观念以及作法就是MFC 的Message Map 的雏形。MFC 把其中的动作包装得更好更精致（当然因此也就更复杂得多），成为一张庞大的消息地图；程序一旦获得消息，就可以按图上溯，直到被处理为止。我将在第 3 章简单仿真MFC 的Message Map，并在第 9 章「消息映射与绕行」中详细探索其完整内容。

对话框的运作

Windows 的对话框依其与父窗口的关系，分为两类：

1. 「令其父窗口除能，直到对话框结束」，这种称为modal 对话框。

2. 「父窗口与对话框共同运行」，这种称为modeless 对话框。

比较常用的是modal 对话框。我就以Generic 的¡ §About¡ ¨ 对话框做为说明范例。

为了做出一个对话框，程序员必须准备两样东西：

1. 对话框模板（dialog template）。这是在RC 文件中定义的一个对话框外貌，以各种方式决定对话框的大小、字形、内部有哪些控制组件、各在什么位置...等等。

2. 对话框函数（dialog procedure）。其类型非常类似窗口函数，但是它通常只处理*WM_INITDIALOG* 和*WM_COMMAND* 两个消息。对话框中的各个控制组件也都是小小窗口，各有自己的窗口函数，它们以消息与其管理者（父窗口，也就是对话框）沟通。而所有的控制组件传来的消息都是*WM_COMMAND*，再由其参数分辨哪一种控制组件以及哪一种通告（notification）。

Modal 对话框的激活与结束，靠的是*DialogBox* 和*EndDialog* 两个API 函数。请看图1-4。

对话框处理过消息之后，应该传回*TRUE*；如果未处理消息，则应该传回*FALSE*。这是因为你的对话框函数之上层还有一个系统提供的预设对话框函数。如果你传回*FALSE*，该预设对话框函数就会接手处理。

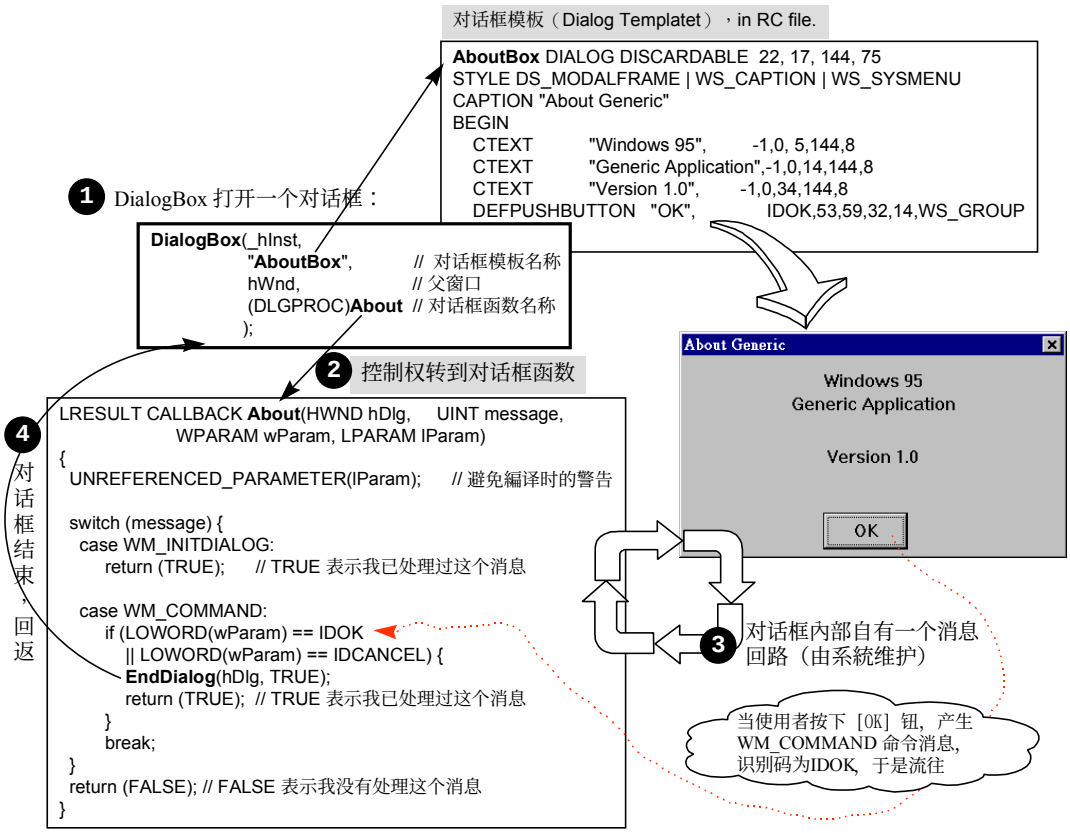


图 1-4 对话框的诞生、运作、结束

模块定义文件（.DEF）

Windows 程序需要一个模块定义文件，将模块名称、程序节区和资料节区的内存特性、模块堆积（heap）大小、堆栈（stack）大小、所有callback 函数名称...等等登记下来。下面是个实例：

```
NAME         Generic
DESCRIPTION 'Generic Sample'
EXETYPE      WINDOWS
STUB        'WINSTUB.EXE'
CODE        PRELOAD DISCARDABLE
DATA        PRELOAD MOVEABLE MULTIPLE
HEAPSIZE    4096
STACKSIZE   10240
EXPORTS
            MainWndProc @1
            AboutBox @2
```

在Visual C++ 整合环境中开发程序，不再需要特别准备.DEF 文件，因为模块定义文件中的设定都有默认值。模块定义文件中的STUB 指令用来指定所谓的stub 程序（埋在Windows 程序中的一个DOS 程序，你所看到的This Program Requires Microsoft Windows 或This Program Can Not Run in DOS mode 就是此程序发出来的），Win16 允许程序员自设一个stub 程序，但Win32 不允许，换句话说在Win32 之中Stub 指令已经失效。

资源描述档（.RC）

RC 文件是一个以文字描述资源的地方。常用的资源有九项之多，分别是ICON、CURSOR、BITMAP、FONT、DIALOG、MENU、ACCELERATOR、STRING、VERSIONINFO。还可能有新的资源不断加入，例如Visual C++ 4.0 就多了一种名为TOOLBAR 的资源。这些文字描述需经过RC 编译器，才产生可使用的二进制代码。本例Generic 示范ICON、MENU 和DIALOG 三种资源。

Windows 程序的生与死

我想你已经了解Windows 程序的架构以及它与Windows 系统之间的关系。对Windows 消息种类以及发生时机的透彻了解，正是程序设计的关键。现在我以窗口的诞生和死亡，说明消息的发生与传递，以及应用程序的兴起与结束，请看图1-5 及图1-6。

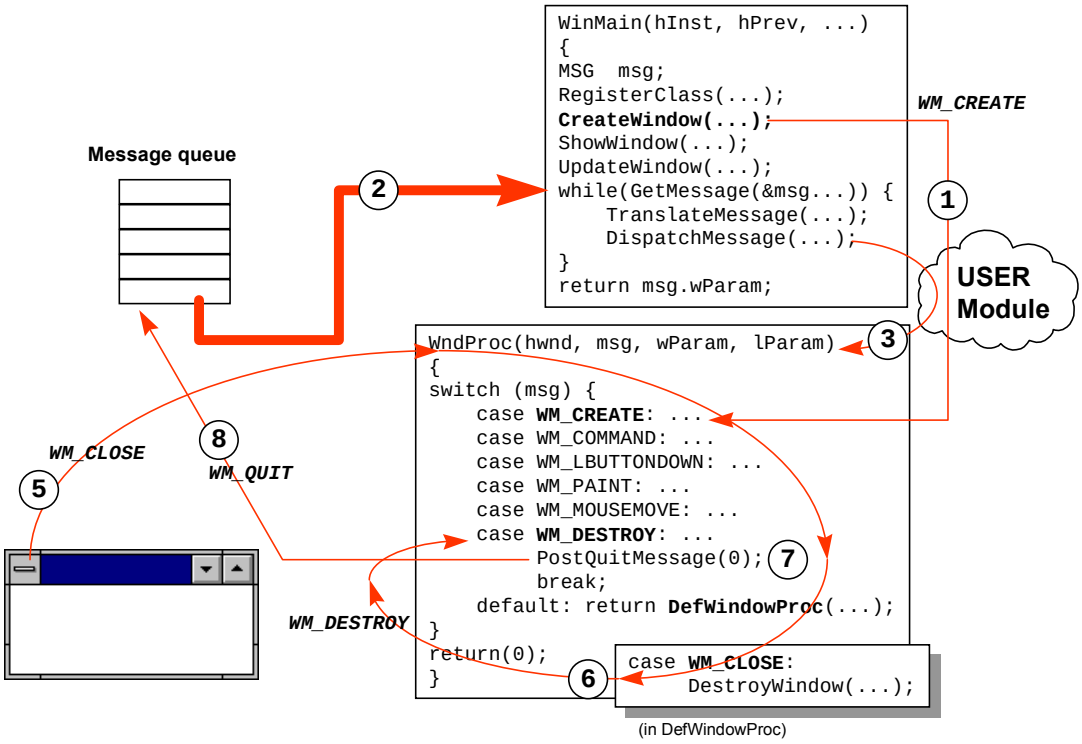


图 1-5 窗口的生命周期（详细说明请看图 1-6）

1. 程序初始化过程中调用*CreateWindow*，为程序建立了一个窗口，做为程序的萤幕舞台。*CreateWindow* 产生窗口之后会送出*WM_CREATE* 直接给窗口函数，后者于是可以在此时机做些初始化动作（例如配置内存、开文件、读初始资料...）。

2. 程序活着的过程中，不断以*GetMessage* 从消息贮列中抓取消息。如果这个消息是*WM_QUIT*，*GetMessage* 会传回0 而结束*while* 循环，进而结束整个程序。

3. *DispatchMessage* 透过Windows USER 模块的协助与监督，把消息分派至窗口函数。消息将在该处被判别并处理。

4. 程序不断进行**2.** 和**3.** 的动作。

5. 当使用者按下系统菜单中的Close 命令项，系统送出*WM_CLOSE*。通常程序的窗口函数不栏截此消息，于是*DefWindowProc* 处理它。

6. *DefWindowProc* 收到*WM_CLOSE* 后， 调用*DestroyWindow* 把窗口清除。*DestroyWindow* 本身又会送出*WM_DESTROY*。

7. 程序对*WM_DESTROY* 的标准反应是调用*PostQuitMessage*。

8. *PostQuitMessage* 没什么其它动作，就只送出*WM_QUIT* 消息，准备让消息循环中的*GetMessage* 取得，如步骤**2**，结束消息循环。

图1-6 窗口的生命周期（请对照图1-5）

为什么结束一个程序复杂如斯？因为操作系统与应用程序职司不同，二者是互相合作的关系，所以必需各做各的份内事，并互以消息通知对方。如果不依据这个游戏规则，可能就会有麻烦产生。你可以作一个小实验，在窗口函数中拦截*WM_DESTROY*，但不调*ÓÃPostQuitMessage*。你会发现当选择系统菜单中的Close 时，屏幕上这个窗口消失了，（因为窗口摧毁及数据结构的释放是*DefWindowProc* 调用*DestroyWindow* 完成的），但是应用程序本身并没有结束（因为消息循环结束不了），它还留存在内存中。

空闲时间的处理：OnIdle

所谓空闲时间（idle time），是指「系统中没有任何消息等待处理」的时间。举个例子，没有任何程序使用定时器（timer，它会定时送来*WM_TIMER*），使用者也没有碰触键盘和鼠标或任何外围，那么，系统就处于所谓的空闲时间。

空闲时间常常发生。不要认为你移动鼠标时产生一大堆的*WM_MOUSEMOVE*，事实上夹杂在每一个*WM_MOUSEMOVE* 之间就可能存在许多空闲时间。毕竟，计算机速度超乎想像。

背景工作最适宜在空闲时间完成。传统的SDK 程序如果要处理空闲时间，可以以下列循环取代*WinMain* 中传统的消息循环：

```
while (TRUE) {
  if (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE) {
      if (msg.message == WM_QUIT)
          break;
      TranslateMessage(&msg);
      DispatchMessage(&msg);
  }
  else {
    OnIdle();
  }
}
```

原因是*PeekMessage* 和*GetMessage* 的性质不同。它们都是到消息队列中抓消息，如果抓不到，程序的主执行线程（primary thread，是一个UI 执行线程）会被操作系统虚悬住。当操作系统再次回来照顾此一执行线程，而发现消息队列中仍然是空的，这时候两个API 函数的行为就有不同了：

- *GetMessage* 会过门不入，于是操作系统再去照顾其它人。
- *PeekMessage* 会取回控制权，使程序得以执行一段时间。于是上述消息循环进入*OnIdle* 函数中。

第 6 章的HelloMFC 将示范如何在MFC 程序中处理所谓的idle time（p.403）。

Console 程序

说到Windows 程序，一定得有*WinMain*、消息循环、窗口函数。即使你只产生一个对话窗（Dialog Box）或消息窗（Message Box），也有隐藏在Windows API（*DialogBox* 和 *MessageBox*）内里的消息循环和窗口函数。

过去那种单单纯纯纯的C/C++ 程序，有着简单的*main* 和*printf* 的好时光到哪里去了？夏天在阴凉的树荫下嬉戏，冬天在温暖的炉火边看书，啊，Where did the good times go？

其实说到Win32 程序，并不是每个都如Windows GUI 程序那么复杂可怖。是的，你可以在Visual C++ 中写一个"DOS-like" 程序，而且仍然可以调用部份的、不牵扯到图形使用者接口（GUI）的Win32 API。这种程序称为console 程序。甚至你还可以在console 程序中使用部份的MFC 类别（同样必须是与GUI 没有关连的），例如处理数组、串行等数据结构的collection classes（*CArray*、*CList*、*CMap*）、与文件有关的*CFile*、*CStdioFile*。

我在BBS 论坛上看到很多程序设计初学者，还没有学习C/C++，就想直接学习Visual C++。并不是他们好高骛远，而是他们以为Visual C++ 是一种特殊的C++ 语言。吃过苦头的过来人以为初学所说的Visual C++ programming 是指MFC programming，所以大吃一惊（没有一点C++ 基础就要学习MFC programming，当然是大吃一惊）。

在Visual C++ 中写纯种的C/C++ 程序？当然可以！不牵扯任何窗口、对话窗、控制组件，那就是console 程序　虽然我这本书没有打算照顾 C++ 初学者，然而我还是决定把console 程序设计的一些相关心得放上来，同时也是因为我打算以console 程序完成稍后的多线程程序范例。第 3 章的MFC 六大技术仿真程序也都是console 程序。

其实，除了"DOS-like"，console 程序还另有妙用。如果你的程序和使用者之间是以巨量文字来互动，或许你会选择使用edit 控制组件（或MFC 的*CEditView*）。但是你知道，计算机在一个纯粹的「文字窗口」（也就是console 窗口）中处理文字的显现与卷动比较

快，你的程序动作也比较简单。所以，你也可以在Windows 程序中产生console 窗口，独立出来操作。

这也许不是你所认知的console 程序。总之，有这种混合式的东西存在。

这一节将以我自己的一个极简易的个人备份软件JBACKUP 为实例，说明Win32 console 程序的撰写，以及如何在其中使用Win32 API（其实直接调用就是了）。再以另一个极小的程序MFCCON 示范MFC console 程序（用到了MFC 的*CStudioFile* 和*CString*）。对于这么小的程序而言，实在不需动用到整合环境下的什么项目管理。至于复杂一点的程序，就请参考第 4 章最后一节「Console 程序的项目管理」。

Console 程序与DOS 程序的差别

不少人把DOS 程序和console 程序混为一谈，这是不对的。以下是各方面的比较。

制造方式

在Windows 环境下的DOS Box 中，或是在Windows 版本的各种C++ 编译器套件的整合环境（IDE）中（第 4 章「Console 程序项目管理」），利用Windows 编译器、联结器做出来的程序，都是所谓Win32 程序。如果程序是以*main* 为进入点，调用C runtime 函数和「不牵扯GUI」的Win32 API 函数，那么就是一个console 程序，console 窗口将成为其标准输入和输出装置（*cin* 和*cout*）。

过去在DOS 环境下开发的程序，称为DOS 程序，它也是以*main* 为程序进入点，可以调用C runtime 函数。但，当然，不可能调用Win32 API 函数。

程序能力

过去的DOS 程序仍然可以在Windows 的DOS Box 中跑（Win95 的兼容性极高，WinNT 的兼容性稍差）。

Console 程序当然更没有问题。由于console 程序可以调用部份的Win32 API（尤其是KERNEL32.DLL 模块所提供的那一部份），所以它可以使用Windows 提供的各种高级功能。它可以产生进程（processes），产生执行线程（threads）、取得虚拟内存的信息、刺探操作系统的各种资料。但是它不能够有华丽的外表-- 因为它不能够调用与GUI 有关的各种API 函数。

DOS 程序和console 程序两者都可以做*printf* 输出和*cout* 输出，也都可以做*scanf* 输入和*cin* 输入。

可执行档格式

DOS 程序是所谓的MZ 格式（MZ 是Mark Zbikowski 的缩写，他是DOS 系统的一位主要构造者）。Console 程序的格式则和所有的Win32 程序一样，是所谓的PE（Portable Executable）格式，意思是它可以被拿到任何Win32 平台上执行。

Visual C++ 附有一个DUMPBIN 工具软件，可以观察PE 文件格式。拿它来观察本节的JBACKUP 程序和MFCCON 程序（以及第 3 章的所有程序），得到这样的结果：

```
H:\u004\prog\jbackup.01>dumpbin /summary jbackup.exe
Microsoft (R) COFF Binary File Dumper Version 5.00.7022
Copyright (C) Microsoft Corp 1992-1997. All rights reserved.
  Dump of file jbackup.exe
  File Type: EXECUTABLE IMAGE
        Summary
          5000 .data
          1000 .idata
          1000 .rdata
          5000 .text
```

拿它来观察DOS 程序，则得到这样的结果：

```
C:\UTILITY>dumpbin /summary dsize.exe
Microsoft (R) COFF Binary File Dumper Version 5.00.7022
Copyright (C) Microsoft Corp 1992-1997. All rights reserved.

Dump of file dsize.exe
DUMPBIN : warning LNK4094: "dsize.exe" is an MS-DOS executable;
use EXEHDR to dump it

    Summary
```

Console 程序的编译联结

你可以写一个makefile，编译时指定常数/D_CONSOLE，联结时指定subsystem 为 console，如下：

```
#0001  # filename : pedump.mak
#0002  # make file for pedump.exe
#0003  # usage : nmake pedump.msc (Visual C++ 5.0)
#0004
#0005  all : pedump.exe
#0006
#0007  pedump.exe: pedump.obj exedump.obj objdump.obj common.obj
#0008     link /subsystem:console /incremental:yes \
#0009          /machine:i386 /out:pedump.exe \
#0010          pedump.obj common.obj exedump.obj objdump.obj \
#0011          kernel32.lib user32.lib
#0012
#0013  pedump.obj : pedump.c
#0014     cl /W3 /GX /Zi /YX /Od /DWIN32 /D_CONSOLE /FR /c pedump.c
#0015
#0016  common.obj : common.c
#0017     cl /W3 /GX /Zi /YX /Od /DWIN32 /D_CONSOLE /FR /c common.c
#0018
#0019  exedump.obj : exedump.c
#0020     cl /W3 /GX /Zi /YX /Od /DWIN32 /D_CONSOLE /FR /c exedump.c
#0021
#0022  objdump.obj : objdump.c
#0023     cl /W3 /GX /Zi /YX /Od /DWIN32 /D_CONSOLE /FR /c objdump.c
```

如果是很简单的情况，例如本节的 JBACKUP 只有一个 C 原代码，那么这样也行（在命令列之下）：

```
cl jbackup.c <ENTER>    ← 将获得 jbackup.exe
```

注意，环境变量要先设定好（请参考本章稍早的「如何产生Generic.exe」一节）。

第 3 章的Frame_ 程序则是这样完成的：

```
cl my.cpp mfc.cpp <ENTER>  将获得my.exe
```

至于到底该联结哪些链接库，全让CL.EXE 去伤脑筋就好了。

JBACKUP：Win32 Console 程序设计

撰写console 程序，有几个重点请注意：

1. 进入点为*main*。

2. 可以使用*printf*、*scanf*、*cin*、*cout* 等标准输出入装置。

3. 可以调用和GUI 无关的Win32 API。

我的这个JBACKUP 程序可以有一个或两个参数，用法如下：

C:\SomeoneDir>JBACKUP SrcDir [DstDir]
例如JBACKUP g: k:
将磁盘目录SrcDir 中的新文件拷贝到磁盘目录DstDir,
并将DstDir 的赘余文件杀掉。
如果没有指定DstDir，预设为k:（那是我的可写入光驱-- MO -- 的代码啦）
并将k: 的磁盘目录设定与SrcDir 相同。
例如JBACKUP g:
而目前g: 是g:\u002\doc
那么相当于把g:\u002\doc 备份到k:\u002\doc 中，并杀掉k:\u002\doc 的赘余文件。

JBACK 检查SrcDir 中所有的文件和DstDir 中所有的文件，把比较新的文件从SrcDir 中拷贝到DstDir 去，并把DstDir 中多出来的文件删除，使ScrDir 和DstDir 的文件保

持完全相同。之所以不做xcopy 完全拷贝动作，为的是节省拷贝时间（做为备份装置，通常是软盘或磁带或可擦写光盘MO，读写速度并不快）。

JBACKUP 没有能力处理SrcDir 底下的子目录文件。如果要处理子目录，漂亮的作法是使用递归（recursive），但是有点伤脑筋，这一部份留给你了。我的打字速度还算快，多切换几次磁盘目录不是问题，呵呵呵。

JBACKUP 使用以下数个Win32 APIs：

```
GetCurrentDirectory
FindFirstFile
FindNextFile
CompareFileTime
CopyFile
DeleteFile
```

在处理完毕命令列参数中的SrcDir 和DstDir 之后，JBACKUP 先把SrcDir 中的所有文件（不含子目录文件）搜寻一遍，储存在一个数组srcFiles[ ] 中，每个数组元素是一个我自定的SRCFILE 数据结构：

```
typedef struct _SRCFILE
{
    WIN32_FIND_DATA fd;
    BOOL bIsNew;
} SRCFILE;
SRCFILE srcFiles[FILEMAX];
WIN32_FIND_DATA fd;
    // prepare srcFiles[]...
    bRet = TRUE;
    iSrcFiles = 0;
    hFile = FindFirstFile(SrcDir, &fd);
    while (hFile != INVALID_HANDLE_VALUE && bRet)
    {
        if (fd.dwFileAttributes == FILE_ATTRIBUTE_ARCHIVE) {
            srcFiles[iSrcFiles].fd = fd;
            srcFiles[iSrcFiles].bIsNew = FALSE;
            iSrcFiles++;
        }
```

```
    bRet = FindNextFile(hFile, &fd);
  }
```

再把DstDir 中的所有文件（不含子目录文件）搜寻一遍，储存在一个destFiles[ ] 数组中，每个数组元素是一个我自定的DESTFILE 数据结构：

```
  typedef struct _DESTFILE
  {
      WIN32_FIND_DATA fd;
      BOOL bMatch;
  } DESTFILE;
  DESTFILE destFiles[FILEMAX];
  WIN32_FIND_DATA fd;
    bRet = TRUE;
    iDestFiles = 0;
    hFile = FindFirstFile(DstDir, &fd);
    while (hFile != INVALID_HANDLE_VALUE && bRet)
    {
        if (fd.dwFileAttributes == FILE_ATTRIBUTE_ARCHIVE) {
            destFiles[iDestFiles].fd = fd;
            destFiles[iDestFiles].bMatch = FALSE;
            iDestFiles++;
        }
        bRet = FindNextFile(hFile, &fd);
  }
```

然后比对srcFiles[ ] 和destFiles[ ] 之中的所有文件名称以及建档日期，找出scrFiles[ ] 中的哪些文件比desFiles[ ] 中的文件更新，然后将其bIsNew 字段设为TRUE。同时也对存在于desFiles[ ] 中而不存在于srcFiles[ ] 中的文件，令其bMatch 字段为FALSE。最后，检查srcFiles[ ] 中的所有文件，将bIsNew 字段为TRUE 者，拷贝到DstDir 去。并检查destFiles[ ] 中的所有文件，将bMatch 字段为FALSE 者统统杀掉。

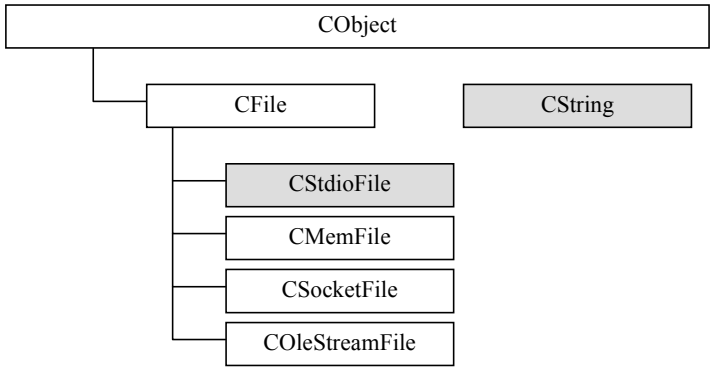
JBACKUP 的源代码与可执行文件放在书附盘片的Jbackup.01 子目录中。

MFCCON：MFC Console 程序设计

当你的进度还在第 1 章的Win32 基本程序观念，我却开始讲如何设计一个MFC console 程序，是否有点时地不宜？

是有一点！所以我挑一个最单纯而无与别人攀缠纠葛的MFC 类别，写一个40 行的小程序。目标纯粹是为了做一个导入，并与Win32 console 程序做一比较。

我所挑选的两个单纯的MFC 类别是*CStdioFile* 和*CString*：



在MFC 之中，*CFile* 用来处理正常的文件I/O 动作。*CStdioFile* 衍生自*CFile*，一个*CStdioFile* 对象代表以C runtime 函数*fopen* 所开启的一个stream 文件。Stream 文件有缓冲区，可以文字模式（预设情况）或二进制模式开启。

CString 对象代表一个字符串，是一个完全独立的类别。

我的例子用来计算小于100 的所有费伯纳契数列（Fabonacci sequence）。费伯纳契数列的计算方式是：

1. 头两个数为1。

2. 接下来的每一个数是前两个数的和。

以下便是 MFCCON.CPP 內容：

```
#0001  // File   : MFCCON.CPP
#0002  // Author : J.J.Hou / Top Studio
#0003  // Date   : 1997.04.06
#0004  // Goal   : Fibonacci sequencee, less than 100
#0005  // Build  : cl /MT mfccon.cpp  (/MT means Multithreading)
#0006
#0007  #include <afx.h>
#0008  #include <stdio.h>
#0009
#0010  int main()
#0011  {
#0012  int lo, hi;
#0013  CString str;
#0014  CStdioFile fFibo;
#0015
#0016     fFibo.Open("FIBO.DAT", CFile::modeWrite |
#0017               CFile::modeCreate | CFile::typeText);
#0018
#0019     str.Format("%s\n", "Fibonacci sequencee, less than 100 :");
#0020     printf("%s", (LPCTSTR) str);
#0021     fFibo.WriteString(str);
#0022
#0023     lo = hi = 1;
#0024
#0025     str.Format("%d\n", lo);
#0026     printf("%s", (LPCTSTR) str);
#0027     fFibo.WriteString(str);
#0028
#0029     while (hi < 100)
#0030     {
#0031        str.Format("%d\n", hi);
#0032        printf("%s", (LPCTSTR) str);
#0033        fFibo.WriteString(str);
#0034        hi = lo + hi;
#0035        lo = hi - lo;
#0036     }
#0037
#0038     fFibo.Close();
#0039     return 0;
#0040  }
```

以下是执行结果（在 console 窗口和 FIBO.DAT 档案中，结果都一样）：

```
Fibonacci sequencee, less than 100 :
1 1 2 3 5 8
13
21
34
55
89
```

这么简单的例子中，我们看到MFC Console 程序的几个重点：

1. 程序进入点仍为*main*

2. 需包含所使用之类别的头文件（本例为AFX.H）

3. 可直接使用与GUI 无关的MFC 类别（本例为*CStdioFile* 和*CString*）

4. 编辑时需指定/MT，表示使用多执行线程版本的C runtime 函数库。

第 4 点需要多做说明。在MFC console 程序中一定要指定多线程版的C runtime 函数库，所以必须使用/MT 选项。如果不做这项设定，会出现这样的联结错误：

```
Microsoft (R) 32-Bit Incremental Linker Version 5.00.7022
Copyright (C) Microsoft Corp 1992-1997. All rights reserved.
/out:mfccon.exe
mfccon.obj
nafxcw.lib(thrdcore.obj) : error LNK2001: unresolved external symbol __endthreadex
nafxcw.lib(thrdcore.obj) : error LNK2001: unresolved external symbol __beginthreadex
mfccon.exe : fatal error LNK1120: 2 unresolved externals
```

表示它找不到*__beginthreadex* 和*__endthreadex*。怪了，我们的程序有调用它们吗？没有，但是MFC 有！这两个函数将在稍后与执行线程有关的小节中讨论。

MFCCON 的源代码与可执行文件放在书附盘片的mfccon.01 子目录中。

什么是C Runtime函数库的多线程版本

当C runtime 函数库于1970s 年代产生出来时，PC 的内存容量还很小，多任务是个新奇观念，更别提什么多执行线程了。因此以当时产品为基础所演化的C runtime 函数库在多线程（multithreaded）的表现上有严重问题，无法被多线程程序使用。

利用各种同步机制（synchronous mechanism）如critical section、mutex、semaphore、event，可以重新开发一套支持多执行线程的runtime 函数库。问题是，加上这样的能力，可能导至程序代码大小和执行效率都遭受不良波及-- 即使你只激活了一个执行线程。

Visual C++ 的折衷方案是提供两种版本的C runtime 函数库。一种版本给单线程程序使用，一种版本给多线程程序使用。多线程版本的重大改变是，第一，变量如errno 者现在变成每个执行线程各拥有一个。第二，多线程版中的数据结构以同步机制加以保护。

Visual C++ 一共有六个C runtime 函数库产品供你选择：

- ◆Single-Threaded（static） libc.lib 898,826
- ◆Multithreaded（static） libcmt.lib 951,142
- ◆ Multithreaded DLL msvcrt.lib 5,510,000
- ◆ Debug Single-Threaded（static） libcd.lib 2,374,542
- ◆ Debug Multithreaded（static）libcmtd.lib 2,949,190
- ◆ Debug Multithreaded DLL msvcrtd.lib 803,418

Visual C++ 编译器提供下列选项，让我们决定使用哪一个C runtime 函数库：

- ◆ /ML Single-Threaded（static）
- ◆ /MT Multithreaded（static）
- ◆ /MD Multithreaded DLL（dynamic import library）
- ◆ /MLd Debug Single-Threaded（static）
- ◆ /MTd Debug Multithreaded（static）
- ◆ /MDd Debug Multithreaded DLL（dynamic import library）

进程与执行线程（Process and Thread）

OS/2、Windows NT 以及Windows 95 都支持多执行线程，这带给PC 程序员一股令人兴奋的气氛。然而它带来的不全然是利多，如果不谨慎小心地处理执行线程的同步问题，程序的错误以及除错所花的时间可能使你发誓再也不碰「执行线程」这种东西。

我们习惯以进程（process）表示一个执行中的程序，并且以为它是CPU 排程单位。事实上执行线程才是排程单位。

核心对象

首先让我解释什么叫作「核心对象」（kernel object）。「GDI 对象」是大家比较熟悉的东西，我们利用GDI 函数所产生的一支笔（Pen）或一支刷（Brush）都是所谓的「GDI 对象」。但什么又是「核心对象」呢?

你可以说核心对象是系统的一种资源（噢，这说法对GDI 对象也适用），系统对象一旦产生，任何应用程序都可以开启并使用该对象。系统给予核心对象一个计数值（usage count）做为管理之用。核心对象包括下列数种：

核心对象	产生方法
event	*CreateEvent*
mutex	*CreateMutex*
semaphore	*CreateSemaphore*
file	*CreateFile*
file-mapping	*CreateFileMapping*
process	*CreateProcess*
thread	*CreateThread*

前三者用于执行线程的同步化：file-mapping 对象用于内存映射文件（memory mapping file），process 和thread 对象则是本节的主角。这些核心对象的产生方式（也就是我们

所使用的API）不同，但都会获得一个handle 做为识别；每被使用一次，其对应的计数值就加1。核心对象的结束方式相当一致，调用*CloseHandle* 即可。

「process 对象」究竟做什么用呢？它并不如你想象中用来「执进程序代码」；不，程序代码的执行是执行线程的工作，「process 对象」只是一个数据结构，系统用它来管理进程。

一个进程的诞生与死亡

执行一个程序，必然就产生一个进程（process）。最直接的程序执行方式就是在shell （如Win95 的文件总管或Windows 3.x 的文件管理员）中以鼠标双击某一个可执行文件图标（假设其为App.exe），执行起来的App 进程其实是shell 调用*CreateProcess* 激活的。

让我们看看整个流程：

1. shell 调用*CreateProcess* 激活App.exe。
2. 系统产生一个「进程核心对象」，计数值为1。
3. 系统为此进程建立一个4GB 地址空间。
4. 加载器将必要的码加载到上述地址空间中，包括App.exe 的程序、资料，以及所需的动态联结函数库（DLLs）。加载器如何知道要加载哪些DLLs 呢？它们被记录在可执行文件（PE 文件格式）的.idata section 中。
5. 系统为此进程建立一个执行线程，称为主执行线程（primary thread）。执行线程才是CPU 时间的分配对象。
6. 系统调用C runtime 函数库的Startup code。
7. Startup code 调用App 程序的*WinMain* 函数。
8. App 程序开始运作。
9. 使用者关闭App 主窗口，使*WinMain* 中的消息循环结束掉，于是*WinMain* 结束。
10. 回到Startup code。
11. 回到系统，系统调用*ExitProcess* 结束进程。

可以说，透过这种方式执行起来的所有Windows 程序，都是shell 的子进程。本来，母进程与子进程之间可以有某些关系存在，但shell 在调用*CreateProcess* 时已经把母子之间的脐带关系剪断了，因此它们事实上是独立实例。稍后我会提到如何剪断子进程的脐带。

产生子进程

你可以写一个程序， 专门用来激活其它的程序。关键就在于你会不会使用*CreateProcess*。这个API 函数有众多参数：

```
CreateProcess(
              LPCSTR lpApplicationName,
              LPSTR lpCommandLine,
              LPSECURITY_ATTRIBUTES lpProcessAttributes,
              LPSECURITY_ATTRIBUTES lpThreadAttributes,
              BOOL bInheritHandles,
              DWORD dwCreationFlags,
              LPVOID lpEnvironment,
              LPCSTR lpCurrentDirectory,
              LPSTARTUPINFO lpStartupInfo,
              LPPROCESS_INFORMATION lpProcessInformation
             );
```

第一个参数*lpApplicationName* 指定可执行档档名。第二个参数*lpCommandLine* 指定欲传给新进程的命令列（command line）参数。如果你指定了*lpApplicationName*，但没有扩展名，系统并不会主动为你加上.EXE 扩展名；如果没有指定完整路径，系统就只在目前工作目录中寻找。但如果你指定*lpApplicationName* 为*NULL* 的话，系统会以*lpCommandLine* 的第一个「段落」（我的意思其实是术语中所谓的token）做为可执行档档名；如果这个档名没有指定扩展名，就采用预设的".EXE" 扩展名；如果没有指定路径，Windows 就依照五个搜寻路径来寻找可执行文件，分别是：

1. 调用者的可执行文件所在目录

2. 调用者的目前工作目录

3. Windows 目录

4. Windows System 目录

5. 环境变量中的path 所设定的各目录

让我们看看实例：

```
CreateProcess("E:\\CWIN95\\NOTEPAD.EXE", "README.TXT",...);
```

系统将执行E:\CWIN95\NOTEPAD.EXE，命令列参数是"README.TXT"。如果我们这样子调用：

```
CreateProcess(NULL, "NOTEPAD README.TXT",...);
```

系统将依照搜寻次序，将第一个被找到的NOTEPAD.EXE 执行起来，并转送命令列参数"README.TXT" 给它。

建立新进程之前，系统必须做出两个核心对象，也就是「进程对象」和「执行线程对象」。*CreateProcess* 的第三个参数和第四个参数分别指定这两个核心对象的安全属性。至于第五个参数（*TRUE* 或*FALSE*）则用来设定这些安全属性是否要被继承。关于安全属性及其可被继承的性质，碍于本章的定位，我不打算在此介绍。

第六个参数*dwCreationFlags* 可以是许多常数的组合，会影响到进程的建立过程。这些常数中比较常用的是*CREATE_SUSPENDED*，它会使得子进程产生之后，其主执行线程立刻被暂停执行。

第七个参数*lpEnvironment* 可以指定进程所使用的环境变量区。通常我们会让子进程继承父进程的环境变量，那么这里要指定*NULL*。

第八个参数*lpCurrentDirectory* 用来设定子进程的工作目录与工作磁盘。如果指定*NULL*，子进程就会使用父进程的工作目录与工作磁盘。

第九个参数*lpStartupInfo* 是一个指向*STARTUPINFO* 结构的指针。这是一个庞大的结构，可以用来设定窗口的标题、位置与大小，详情请看API 使用手册。

最后一个参数是一个指向*PROCESS_INFORMATION* 结构的指针：

```
typedef struct _PROCESS_INFORMATION {
        HANDLE hProcess;
        HANDLE hThread;
        DWORD dwProcessId;
        DWORD dwThreadId;
} PROCESS_INFORMATION;
```

当系统为我们产生「进程对象」和「执行线程对象」，它会把两个对象的handle 填入此结构的相关字段中，应用程序可以从这里获得这些handles。

如果一个进程想结束自己的生命，只要调用：

```
VOID ExitProcess(UINT fuExitCode);
```

就可以了。如果进程想结束另一个进程的生命，可以使用：

```
BOOL TerminateProcess(HANDLE hProcess, UINT fuExitCode);
```

很显然，只要你有某个进程的handle，就可以结束它的生命。*TerminateProcess* 并不被建议使用，倒不是因为它的权力太大，而是因为一般进程结束时，系统会通知该进程所开启（所使用）的所有DLLs，但如果你以*TerminateProcess* 结束一个进程，系统不会做这件事，而这恐怕不是你所希望的。

前面我曾说过所谓割断脐带这件事情，只要你把子进程以*CloseHandle* 关闭，就达到了目的。下面是个例子：

```
PROCESS_INFORMATION ProcInfo;
BOOL fSuccess;

fSuccess = CreateProcess(...,&ProcInfo);
if (fSuccess) {
        CloseHandle(ProcInfo.hThread);
        CloseHandle(ProcInfo.hProcess);
}
```

一个执行线程的诞生与死亡

程序代码的执行，是执行线程的工作。当一个进程建立起来，主执行线程也产生。所以每一个Windows 程序一开始就有了一个执行线程。我们可以调用*CreateThread* 产生额外的执行线程，系统会帮我们完成下列事情：

1. 配置「执行线程对象」，其handle 将成为*CreateThread* 的传回值。

2. 设定计数值为1。

3. 配置执行线程的context。

4. 保留执行线程的堆栈。

5. 将context 中的堆栈指针缓存器（SS）和指令指针缓存器（IP）设定妥当。

看看上面的态势，的确可以显示出执行线程是CPU 分配时间的单位。所谓工作切换（context switch）其实就是对执行线程的context 的切换。

程序若欲产生一个新执行线程，调用*CreateThread* 即可办到：

```
CreateThread(LPSECURITY_ATTRIBUTES lpThreadAttributes,
             DWORD dwStackSize,
             LPTHREAD_START_ROUTINE lpStartAddress,
             LPVOID lpParameter,
             DWORD dwCreationFlags,
             LPDWORD lpThreadId
             );
```

第一个参数表示安全属性的设定以及继承，请参考API 手册。Windows 95 忽略此一参数。第二个参数设定堆栈的大小。第三个参数设定「执行线程函数」名称，而该函数的参数则在这里的第四个参数设定。第五个参数如果是0，表示让执行线程立刻开始执行，如果是*CREATE_SUSPENDED*，则是要求执行线程暂停执行（那么我们必须调用*ResumeThread* 才能令其重新开始）。最后一个参数是个指向*DWORD* 的指针，系统会把执行线程的ID 放在这里。

上面我所说的「执行线程函数」是什么？让我们看个实例：

```
VOID ReadTime(VOID);
HANDLE hThread;
DWORD ThreadID;

hThread = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)ReadTime,
                       NULL, 0, &ThreadID);
...
//---------------------------------------------------------------------
// thread 函数。
// 不断利用 GetSystemTime 取系统时间，
// 并将结果显示在对话框 _hwndDlg 的 IDE_TIMER 栏位上。
//---------------------------------------------------------------------
VOID ReadTime(VOID)
{
char str[50];
SYSTEMTIME st;

   while(1) {
      GetSystemTime(&st);
      sprintf(str,"%u:%u:%u", st.wHour, st.wMinute, st.wSecond);
      SetDlgItemText (_hWndDlg, IDE_TIMER, str);
      Sleep (1000);  // 延迟一秒。
   }
}
```

当*CreateThread* 成功，系统为我们把一个执行线程该有的东西都准备好。执行线程的主体在哪里呢？就在所谓的执行线程函数。执行线程与执行线程之间，不必考虑控制权释放的问题，因为Win32 操作系统是强制性多任务。

执行线程的结束有两种情况，一种是寿终正寝，一种是未得善终。前者是执行线程函数正常结束退出，那么执行线程也就自然而然终结了。这时候系统会调用*ExitThread* 做些善后清理工作（其实执行线程中也可以自行调用此函数以结束自己）。但是像上面那个例子，执行线程根本是个无穷循环，如何终结？一者是进程结束（自然也就导至执行线程的结束），二者是别的执行线程强制以*TerminateThread* 将它终结掉。不过，*TerminateThread* 太过毒辣，非必要还是少用为妙（请参考API 手册）。

以_beginthreadex 取代CreateThread

别忘了Windows 程序除了调用Win32 API，通常也很难避免调用任何一个C runtime 函数。为了保证多线程情况下的安全，C runtime 函数库必须为每一个执行线程做一些簿记工作。没有这些工作，C runtime 函数库就不知道要为每一个执行线程配置一块新的内存，做为执行线程的区域变量用。因此，*CreateThread* 有一个名为*_beginthreadex* 的外包函数，负责额外的簿记工作。

请注意函数名称的底线符号。它必须存在，因为这不是个标准的ANSI C runtime 函数。*_beginthreadex* 的参数和*CreateThread* 的参数其实完全相同，不过其型别已经被「净化」了，不再有Win32 型别包装。这原本是为了要让这个函数能够移植到其它操作系统，因为微软希望*_beginthreadex* 能够被实作于其它平台，不需要和Windows 有关、不需要包含windows.h。但实际情况是，你还是得调用*CloseHandle* 以关闭执行线程， 而*CloseHandle* 却是个Win32 API，所以你还是需要包含windows.h、还是和Windows 脱离不了关系。微软空有一个好主意，却没能落实它。

把*_beginthreadex* 视为*CreateThread* 的一个看起来比较有趣的版本，就对了：

```
unsigned long _beginthreadex(
                  void *security,
                  unsigned stack_size,
                  unsigned (__stdcall *start_address)(void *),
                  void *arglist,
                  unsigned initflag,
                  unsigned* thrdaddr
                  );
```

_beginthreadex 所传回的unsigned long 事实上就是一个Win32 HANDLE，指向新执行线程。换句话说传回值和*CreateThread* 相同，但*_beginthreadex* 另外还设立了errno 和doserrno。

下面是一个最简单的使用范例：

```
#0001 #include <windows.h>
#0002  #include <process.h>
```

```
#0003  unsigned __stdcall myfunc(void* p);
#0004
#0005  void main()
#0006  {
#0007      unsigned long thd;
#0008      unsigned tid;
#0009
#0010      thd = _beginthreadex(NULL,
#0011                           0,
#0012                           myfunc,
#0013                           0,
#0014                           0,
#0015                           &tid );
#0016      if (thd != NULL)
#0017      {
#0018          CloseHandle(thd);
#0019      }
#0020  }
#0021
#0022  unsigned __stdcall myfunc(void* p)
#0023  {
#0024      // do your job...
#0025  }
```

针对Win32 API *ExitThread*，也有一个对应的C runtime 函数：*_endthreadex*。它只需要一个参数，就是由*_beginthreadex* 第 6 个参数传回来的ID 值。

关于*_beginthreadex* 和*_endthreadex*，以及执行线程的其它各种理论基础、程序技术、使用技巧，可参考由Jim Beveridge & Robert Wiener 合着，Addison Wesley 出版的 ***Multithreading Applications in Win32*** 一书（Win32 多线程程序设计/ 侯俊杰译/ 峰出版）。

执行线程优先权（Priority）

优先权是排程的重要依据。优先权高的执行线程，永远先获得CPU 的青睐。当然啦，操作系统会视情况调整各个执行线程的优先权。例如前景执行线程的优先权应该调高一些，背景执行线程的优先权应该调低一些。

执行线程的优先权范围从0（最低）到31（最高）。当你产生执行线程，并不是直接以数值指定其优先权，而是采用两个步骤。第一个步骤是指定「优先权等级（Priority Class）」给进程，第二步骤是指定「相对优先权」给该进程所拥有的执行线程。图1-7 是优先权等级的描述，其中的代码在*CreateProcess* 的*dwCreationFlags* 参数中指定。如果你不指定， 系统预设给的是*NORMAL_PRIORITY_CLASS* -- 除非父进程是*IDLE_PRIORITY_CLASS*（那么子进程也会是*IDLE_PRIORITY_CLASS*）。

等级	代码	优先权值
idle	IDLE_PRIORITY_CLASS	4
normal	NORMAL_PRIORITY_CLASS	9（前景）或 7（背景）
high	HIGH_PRIORITY_CLASS	13
realtime	REALTIME_PRIORITY_CLASS	24

图 1-7　Win32 执行线程的优先权等级划分

- "idle" 等级只有在CPU 时间将被浪费掉时（也就是前一节所说的空闲时间）才执行。此等级最适合于系统监视软件，或屏幕保护软件。
- "normal" 是预设等级。系统可以动态改变优先权，但只限于"normal" 等级。当进程变成前景，执行线程优先权提升为9，当进程变成背景，优先权降低为7。
- "high" 等级是为了立即反应的需要，例如使用者按下Ctrl+Esc 时立刻把工作管理器（task manager）带出场。
- "realtime" 等级几乎不会被一般应用程序使用。就连系统中控制鼠标、键盘、

磁盘状态重新扫描、Ctrl+Alt+Del 等的执行线程都比"realtime" 优先权还低。这种等级使用在「如果不在某个时间范围内被执行的话，资料就要遗失」的情况。这个等级一定得在正确评估之下使用之，如果你把这样的等级指定给一般的（并不会常常被阻塞的）执行线程，多任务环境恐怕会瘫痪，因为这个执行线程有如此高的优先权，其它执行线程再没有机会被执行。

上述四种等级，每一个等级又映射到某一范围的优先权值。*IDLE_* 最低，*NORMAL_* 次之，*HIGH_* 又次之，*REALTIME_* 最高。在每一个等级之中，你可以使用*SetThreadPriority* 设定精确的优先权，并且可以稍高或稍低于该等级的正常值（范围是两个点数）。你可以把*SetThreadPriority* 想象是一种微调动作。

SetThreadPriority 的参数	微调幅度
THREAD_PRIORITY_LOWEST	-2
THREAD_PRIORITY_BELOW_NORMAL	-1
THREAD_PRIORITY_NORMAL	不变
THREAD_PRIORITY_ABOVE_NORMAL	+1
THREAD_PRIORITY_HIGHEST	+2

除了以上五种微调，另外还可以指定两种微调常数：

SetThreadPriority 的参数	面对任何等级的调整結果：	面对"realtime"等级的调整結果：
THREAD_PRIORITY_IDLE	1	16
THREAD_PRIORITY_TIME_CRITICAL	15	31

这些情况可以以图1-8 作为总结。

优先权等级	idle	lowest	below normal	normal	above normal	highest	time critical
idle	1	2	3	4	5	6	15
normal (背景)	1	5	6	7	8	9	15
normal (前景)	1	7	8	9	10	11	15
high	1	11	12	13	14	15	15
realtime	16	22	23	24	25	26	31

图1-8 Win32执行线程优先权

多线程程序设计实例

我设计了一个MltiThrd 程序，放在书附盘片的MltiThrd.01 子目录中。这个程序一开始产生五个执行线程，优先权分别微调-2、-1、0、+1、+2，并且虚悬不执行：

```
HANDLE _hThread[5];  // global variable
...
LONG APIENTRY MainWndProc (HWND hWnd, UINT message,
                           UINT wParam, LONG lParam)
{
    DWORD  ThreadID[5];
    static DWORD  ThreadArg[5] = {HIGHEST_THREAD,    // 0x00
                                  ABOVE_AVE_THREAD,  // 0x3F
                                  NORMAL_THREAD,     // 0x7F
                                  BELOW_AVE_THREAD,  // 0xBF
                                  LOWEST_THREAD      // 0xFF
                                  };    // 用来调整四方形顏色
    ...
    for(i=0; i<5; i++)   // 产生 5 个 threads
        _hThread[i] = CreateThread(NULL,
                                   0,
                                   (LPTHREAD_START_ROUTINE)ThreadProc,
                                   &ThreadArg[i],
                                   CREATE_SUSPENDED,
                                   &ThreadID[i]);

    // 設定 thread priorities
    SetThreadPriority(_hThread[0], THREAD_PRIORITY_HIGHEST);
    SetThreadPriority(_hThread[1], THREAD_PRIORITY_ABOVE_NORMAL);
```

```
    SetThreadPriority(_hThread[2], THREAD_PRIORITY_NORMAL);
    SetThreadPriority(_hThread[3], THREAD_PRIORITY_BELOW_NORMAL);
    SetThreadPriority(_hThread[4], THREAD_PRIORITY_LOWEST);
    ...
}
```

当使用者按下【Resume Threads】菜单项目后，五个执行线程如猛虎出柙，同时冲出来。这五个执行线程使用同一个执行线程函数*ThreadProc*。我在*ThreadProc* 中以不断的*Rectangle* 动作表示执行线程的进行。所以我们可以从画面上观察执行线程的进度。我并且设计了两种延迟方式，以利观察。第一种方式是在每一次循环之中使用*Sleep*(10)，意思是先睡10 个毫秒，之后再醒来；这段期间，CPU 可以给别人使用。第二种方式是以空循环30000 次做延迟；空循环期间CPU 不能给别人使用（事实上CPU 正忙碌于那30000 次空转）。

```
UINT   _uDelayType=NODELAY;  // global variable
...
VOID ThreadProc(DWORD *ThreadArg)
{
RECT rect;
HDC  hDC;
HANDLE hBrush, hOldBrush;
DWORD dwThreadHits = 0;
int   iThreadNo, i;
   ...
   do
   {
     dwThreadHits++;      // 计数器

     // 画出四方形，代表 thread 的进行
     Rectangle(hDC, *(ThreadArg), rect.bottom-(dwThreadHits/10),
               *(ThreadArg)+0x40, rect.bottom);

     // 延迟...
     if (_uDelayType == SLEEPDELAY)
        Sleep(10);
     else if (_uDelayType == FORLOOPDELAY)
        for (i=0; i<30000; i++);
     else // _uDelayType == NODELAY)
        {  }
   } while (dwThreadHits < 1000);     // 循环  1000 次
   ...
}
```

图1-9 是执行画面。注意，先选择延迟方式（"for loop delay" 或"sleep delay"），再按下【Resume Thread】。如果你选择¡§for loop delay¡¨（图1-9a），你会看到执行线程0（优先权最高）几乎一路冲到底，然后才是执行线程1（优先权次之），然后是执行线程2（优先权再次之）...。但如果你选择的¡§sleep delay¡¨（图1-9b），所有执行线程不分优先权高低，同时行动。关于执行线程的排程问题，我将在第14 章做更多的讨论。

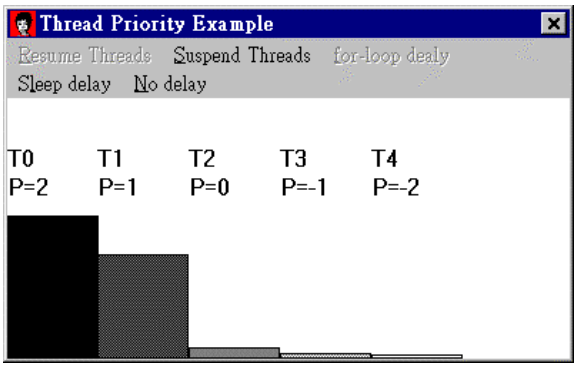


图1-9a MltiThrd.exe的执行画面（¡¨for loop delay¡¨）

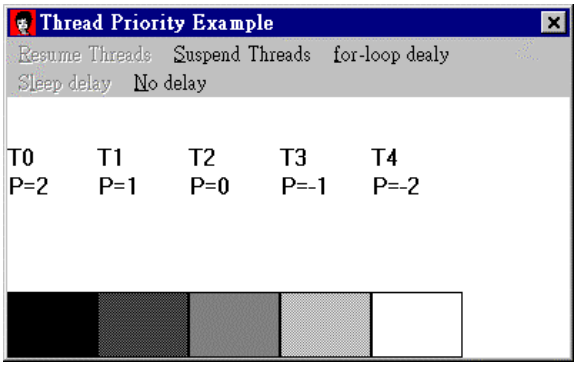


图1-9b MltiThrd.exe的执行画面（¡¨sleep delay¡¨）

注意：为什么图1-9a 中执行线程1尚未完成，执行线程2~4 竟然也有机会偷得一点点CPU时间呢？这是排程器的巧妙设计，动态调整执行线程的优先权。是啊，总不能让优先权低的执行线程直到天荒地老，没有一点点获得。关于执行线程排程问题，第14章有更多的讨论。

图1-10 是以Process Viewer（Visual C++ 5.0 所附工具）观察Mltithrd.exe 的执行结果。图上方出现目前所有的进程，点选其中的MLTITHRD.EXE，果然在窗口下方出现六个执行线程，其中包括主执行线程（优先权已被调整为10）。

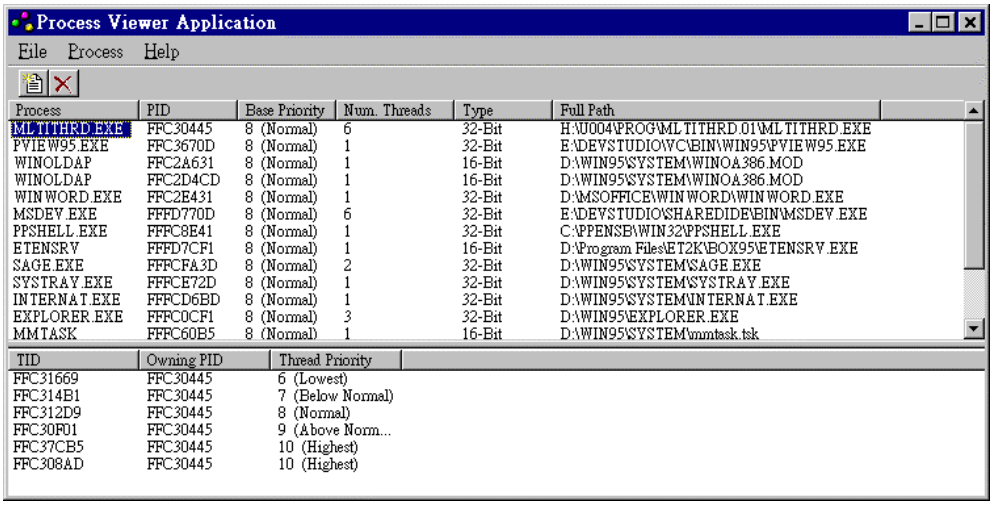


图1-10 利用Process Viewer（PVIEW95.EXE ）观察MLTITHRD.EXE 的执行，的确发现有六个执行线程，其中包括一个主执行线程。

第 2 章

C++的重要性质

C++ 是一种扭转程序员思维模式的语言。

一个人思维模式的扭转，不可能轻而易举一蹴而成。

迄来「对象导向」一词席卷了整个软件界。对象导向程序设计（Object Oriented Programming）其实是一种观念，用什么语言实现它都可以。但，当然，对象导向程序语言（Object Oriented Programming Language）是专门为对象导向观念而发展出来的，以之完成对象导向的封装、继承、多态等特性自是最为便利。

C++ 是最重要的对象导向语言，因为它站在C 语言的肩膀上，而C 语言拥有绝对优势的使用者。C++ 并非纯然的对象导向程序语言，不过有时候混血并不是坏事，纯种也不见得就多好。

所谓纯对象导向语言，是指不管什么东西，都应该存在于对象之中。JAVA 和Small Talk 都是纯对象导向语言。

如果你是C++ 的初学者，本章不适合你（事实上整本书都不适合你），你的当务之急是去买一本C++ 专书。一位专精Basic 和Assembly 语言的朋友问我，有没有可能不会C++ 而学会MFC？答案是当然没有可能。

如果你对C++ 一知半解，语法大约都懂了，语意大约都不懂，本章是我能够给你的最好礼物。我将从类别与对象的关系开始，逐步解释封装、继承、多态、虚拟函数、动态绑定。不只解释其操作方式，更要点出其意义与应用，也就是，为什么需要这些性质。

C++ 语言范围何其广大，这一章的主题挑选完全是以MFC Programming 所需技术为前提。下一章，我们就把这里学到的C++ 技术和OO 观念应用到application framework 的仿真上，那是一个DOS 程序，不牵扯Windows。

类别及其成员- 谈封装（encapsulation）

让我们把世界看成是一个由对象（object）所组成的大环境。对象是什么？白一点说，「东西」是也！任何实际的物体你都可以说它是对象。为了描述对象，我们应该先把对象的属性描述出来。好，给「对象的属性」一个比较学术的名词，就是「类别」（class）。对象的属性有两大成员，一是资料，一是行为。在对象导向的术语中，前者常被称为property（Java 语言则称之为field），后者常被称为method。另有一双比较像程序设计领域的术语，名为member variable（或data member）和member function。为求统一，本书使用第二组术语，也就是member variable（成员变量）和member function（成员函数）。一般而言，成员变量通常由成员函数处理之。

如果我以*CSquare* 代表「四方形」这种类别，四方形有*color*，四方形可以*display*。好，*color* 就是一种成员变量，*display* 就是一种成员函数：

```
CSquare square; // 声明square 是一个四方形。
square.color = RED; // 设定成员变量。RED 代表一个颜色值。
square.display(); // 调用成员函数。
```

下面是C++ 语言对于*CSquare* 的描述：

```
class CSquare // 常常我们以C 作为类别名称的开头
{
private:
    int m_color; // 通常我们以m_ 作为成员变量的名称开头
public:
    void display() { ... }
    void setcolor(int color) { m_color = color; }
};
```

成员变量可以只在类别内被处理，也可以开放给外界处理。以资料封装的目的而言，自

然是前者较为妥当，但有时候也不得不开放。为此，C++ 提供了*private*、*public* 和 *protected* 三种修饰词。一般而言成员变量尽量声明为*private*，成员函数则通常声明为 *public*。上例的*m_color* 既然声明为*private*，我们势必得准备一个成员函数*setcolor*，供外界设定颜色用。

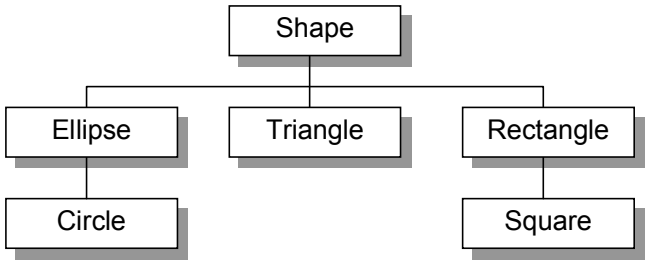
把资料声明为*private*，不允许外界随意存取，只能透过特定的接口来操作，这就是对象导向的封装（encapsulation）特性。

基础类别与衍生类别：谈继承(Inheritance)

其它语言欲完成封装性质，并不太难。以C 为例，在结构（*struct*）之中放置资料，以及处理资料的函数的指针（function pointer），就可得到某种程度的封装精神。

C++ 神秘而特有的性质其实在于继承。

矩形是形，椭圆形是形，三角形也是形。苍蝇是昆虫，蜜蜂是昆虫，蚂蚁也是昆虫。是的，人类习惯把相同的性质抽取出来，成立一个基础类别（base class），再从中衍化出衍生类别（derived class）。所以，关于形状，我们就有了这样的类别阶层：



注意：衍生类别与基础类别的关系是"IsKindOf" 的关系。也就是说，
Circle「是一种」Ellipse，Ellipse「是一种」Shape；
Square「是一种」Rectangle，Rectangle「是一种」Shape。

```
#0001  class CShape       // 形状
#0002  {
#0003  private:
#0004    int m_color;
#0005
#0006  public:
#0007    void setcolor(int color) { m_color = color; }
#0008  };
#0009
#0010  class CRect : public CShape       // 矩形是一种形状
#0011  {                                 //  它会继承 m_color 和 setcolor()
#0012  public:
#0013    void display() { ... }
#0014  };
#0015
#0016  class CEllipse : public CShape    // 椭圆形是一种形状
#0017  {                                 //  它会继承 m_color 和 setcolor()
#0018  public:
#0019    void display() { ... }
#0020  };
#0021
#0022  class CTriangle : public CShape   // 三角形是一种形状
#0023  {                                 //  它会继承 m_color 和 setcolor()
#0024  public:
#0025    void display() { ... }
#0026  };
#0027
#0028  class CSquare : public CRect      // 四方形是一种矩形
#0029
#0030  public:
#0031    void display() { ... }
#0032  };
#0033
#0034  class CCircle : public CEllipse   // 圆形是一种椭圆形
#0035  {
#0036  public:
#0037    void display() { ... }
#0038  };
```

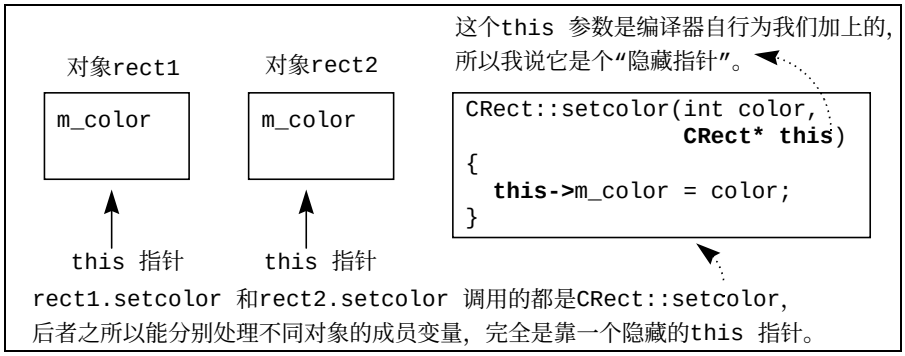
于是你可以这么动作：

```
CSquare square;
CRect   rect1, rect2;
CCircle circle;

  square.setcolor(1);   // 令 square.m_color = 1;
```

```
square.display(); // 调用CSquare::display

rect1.setcolor(2); // 于是rect1.m_color = 2
rect1.display(); // 调用CRect::display

rect2.setcolor(3); // 于是rect2.m_color = 3
rect2.display(); // 调用CRect::display

circle.setcolor(4); // 于是circle.m_color = 4
circle.display(); // 调用CCircle::display
```

注意以下这些事实与问题：

1. 所有类别都由*CShape* 衍生下来，所以它们都自然而然继承了*CShape* 的成员，包括变量和函数。也就是说，所有的形状类别都「暗自」具备了*m_color* 变量和*setcolor* 函数。我所谓暗自（implicit），意思是无法从各衍生类别的声明中直接看出来。
2. 两个矩形对象*rect1* 和*rect2* 各有自己的*m_color*，但关于*setcolor* 函数却是共享相同的*CRect::setcolor*（其实更应该说是*CShape::setcolor*）。我用这张图表示其间的关系：



让我替你问一个问题：同一个函数如何处理不同的资料？为什么*rect1.setcolor* 和*rect2.setcolor* 明明都是调用*CRect*::*setcolor*（其实也就是*CShape*::*setcolor*），却能够有条不紊地分别处理*rect1.m_color* 和*rect2.m_color*？答案在于所谓的*this* 指针。下一节我就会提到它。

3. 既然所有类别都有*display* 动作，把它提升到老祖宗*CShape* 去，然后再继承之，好吗？不好，因为*display* 函数应该因不同的形状而动作不同。

4. 如果*display* 不能提升到基础类别去，我们就不能够以一个*for* 循环或*while* 循环干净漂亮地完成下列动作（此种动作模式在对象导向程序方法中重要无比）：

```
CShape shapes[5];
... // 令5 个shapes 各为矩形、四方形、椭圆形、圆形、三角形
for (int i=0; i<5; i++)
{
shapes[i].display;
}
```

5. Shape 只是一种抽象意念，世界上并没有「形状」这种东西！你可以在一个C++ 程序中做以下动作，但是不符合生活法则：

```
CShape shape; // 世界上没有「形状」这种东西,
shape.setcolor(); // 所以这个动作就有点奇怪。
```

这同时也说出了第三点的另一个否定理由：按理你不能够把一个抽象的「形状」显示出来，不是吗?!

如果语法允许你产生一个不应该有的抽象对象，或如果语法不支持「把所有形状（不管什么形状）都display 出来」的一般化动作作，这就是个失败的语言。C++ 是成功的，自然有它的整治方式。

记住，「对象导向」观念是描绘现实世界用的。所以，你可以以真实生活中的经验去思考程序设计的逻辑。

this 指针

刚刚我才说过，两个矩形对象*rect1* 和*rect2* 各有自己的*m_color* 成员变量，但*rect1.setcolor* 和*rect2.setcolor* 却都通往唯一的*CRect*::*setcolor* 成员函数。那么*CRect*::*setcolor* 如何处理不同对象中的*m_color*？答案是：成员函数有一个隐藏参数，名为*this* 指针。当你调用：

```
rect1.setcolor(2); // rect1 是CRect 对象
rect2.setcolor(3); // rect2 是CRect 对象
```

编译器实际上为你做出来的码是：

```
CRect::setcolor(2, (CRect*)&rect1);
CRect::setcolor(3, (CRect*)&rect2);
```

不过，由于*CRect* 本身并没有声明*setcolor*，它是从*CShape* 继承来的，所以编译器实际上产生的码是：

```
CShape::setcolor(2, (CRect*)&rect1);
CShape::setcolor(3, (CRect*)&rect2);
```

多出来的参数，就是所谓的*this* 指针。至于类别之中，成员函数的定义：

```
class CShape
{
...
public:
    void setcolor(int color) { m_color = color; }
};
```

被编译器整治过后，其实是：

```
class CShape
{
...
public:
    void setcolor(int color, (CShape*)this) { this->m_color = color; }
};
```

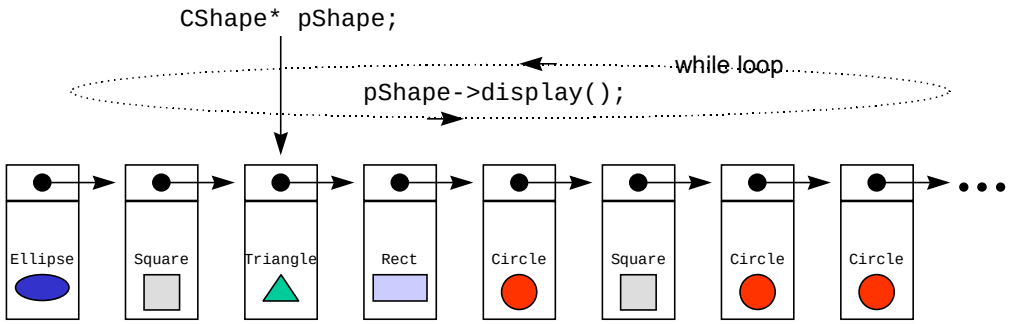
我们拨开了第一道疑云。

虚拟函数与多态(Polymorphism)

我曾经说过，前一个例子没有办法完成这样的动作：

```
CShape shapes[5];
... // 令5 个shapes 各为矩形、四方形、椭圆形、圆形、三角形
for (int i=0; i<5; i++)
{
   shapes[i].display;
}
```

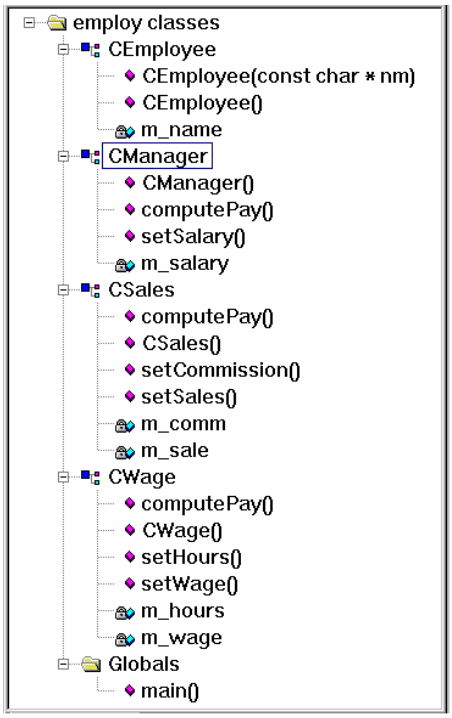
可是这种所谓对象操作的一般化动作在application framework 中非常重要。作为framework 设计者的我，总是希望能够准备一个*display* 函数，给我的使用者调用；不管他根据我的这一大堆形状类别衍生出其它什么奇形怪状的类别，只要他想*display*，像下面那么做就行。

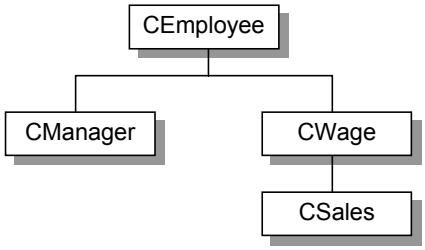


为了支持这种能力，C++ 提供了所谓的虚拟函数（virtual function）。

虚拟+ 函数?! 听起来很恐怖的样子。如果你了解汽车的离合器踩下去代表汽车空档，空档表示失去引擎本身的牵制力，你就会了解「高速行驶间煞车绝不能踩离合器」的道理并矢志遵行。好，如果你真的了解为什么需要虚拟函数以及什么情况下需要它，你就能够掌握它的灵魂与内涵，真正了解它的设计原理，并且发现认为它非常人性。并且，真正知道怎么用它。

让我用另一个例子来展开我的说明。这个范例灵感得自Visual C++ 手册之一：*Introdoction to C++*。假设你的类别种类如下：





本图以Visual C++ 之「Class Info 窗口」获得

程序代码实作如下:

```
#0001  #include <string.h>
#0002
#0003  //---------------------------------------------------------------
#0004  class CEmployee   // 职员
#0005  {
#0006  private:
#0007  char m_name[30];
#0008
#0009  public:
#0010    CEmployee();
#0011    CEmployee(const char* nm) { strcpy(m_name, nm); }
#0012  };
```

```
#0013  //-----------------------------------------------------------------
#0014  class CWage : public CEmployee     // 时薪职员是一种职员
#0015  {
#0016  private :
#0017    float m_wage;
#0018    float m_hours;
#0019
#0020  public :
#0021    CWage(const char* nm)  : CEmployee(nm) { m_wage = 250.0; m_hours = 40.0; }
#0022    void setWage(float wg)  { m_wage = wg; }
#0023    void setHours(float hrs)  { m_hours = hrs; }
#0024    float computePay();
#0025  };
#0026  //-----------------------------------------------------------------
#0027  class CSales : public CWage    // 销售员是一种时薪职员
#0028  {
#0029  private :
#0030    float m_comm;
#0031    float m_sale;
#0032
#0033  public :
#0034    CSales(const char* nm) : CWage(nm)  { m_comm = m_sale = 0.0; }
#0035    void setCommission(float comm)     { m_comm = comm; }
#0036    void setSales(float sale)         { m_sale = sale; }
#0037    float computePay();
#0038  };
#0039  //-----------------------------------------------------------------
#0040  class CManager : public CEmployee   // 经理也是一种职员
#0041  {
#0042  private :
#0043    float m_salary;
#0044  public :
#0045    CManager(const char* nm) : CEmployee(nm)  { m_salary = 15000.0; }
#0046    void setSalary(float salary)            { m_salary = salary; }
#0047    float computePay();
#0048  };
#0049  //-----------------------------------------------------------------
#0050  void main()
#0051  {
#0052  CManager aManager("陈美静");
#0053  CSales   aSales("侯俊杰");
#0054  CWage    aWager("曾铭源");
#0055  }
#0056  //-----------------------------------------------------------------
#0057  //虽然各类别的 computePay函数都没有定义，但因为程序也没有调用之，所以无妨。
```

如此一来，*CWage* 继承了*CEmployee* 所有的成员（包括资料与函数），*CSales* 又继承了*CWage* 所有的成员（包括资料与函数）。在意义上， 相当于*CSales* 拥有资料如下：

```
// private data of CEmployee
char m_name[30];

// private data of CWage
float m_wage;
float m_hours;

// private data of CSales
float m_comm;
float m_sale;
```

以及函数如下：

```
void setWage(float wg);
void setHours(float hrs);
void setCommission(float comm);
void setSale(float sales);
void computePay();
```

从Visual C++ 的除错器中，我们可以看到，上例的*main* 执行之后，程序拥有三个对象，内容（我是指成员变量）分别为：

从薪水说起

虚拟函数的故事要从薪水的计算说起。根据不同职员的计薪方式，我设计*computePay* 函数如下：

```
float CManager::computePay()
{
     return m_salary; // 经理以「固定周薪」计薪。
}
float CWage::computePay()
{
     return (m_wage * m_hours); // 时薪职员以「钟点费* 每周工时」计薪。
}
float CSales::computePay()
{
// 销售员以「钟点费* 每周工时」再加上「佣金* 销售额」计薪。
     return (m_wage * m_hours + m_comm * m_sale); // 语法错误。
}
```

但是*CSales* 对象不能够直接取用*CWage* 的*m_wage* 和*m_hours*，因为它们是*private* 成员变量。所以是不是应该改为这样：

```
float CSales::computePay()
{
     return computePay() + m_comm * m_sale;
}
```

这也不好，我们应该指明函数中所调用的*computePay* 究归谁属-- 编译器没有厉害到能够自行判断而保证不出错。正确写法应该是：

```
float CSales::computePay()
{
     return CWage::computePay() + m_comm * m_sale;
}
```

这就合乎逻辑了：销售员是一般职员的一种，他的薪水应该是以时薪职员的计薪方式作为底薪，再加上额外的销售佣金。我们看看实际情况，如果有一个销售员：

```
CSales aSales("侯俊杰");
```

那么侯俊杰的底薪应该是:

```
aSales.CWage::computePay(); // 这是销售员的底薪。注意语法。
```

而侯俊杰的全薪应该是:

```
aSales.computePay(); // 这是销售员的全薪
```

结论是：要调用父类别的函数，你必须使用scope resolution operator（::）明白指出。

接下来我要触及对象类型的转换，这关系到指针的运用，更直接关系到为什么需要虚拟函数。了解它，对于application framework 如MFC 者的运用十分十分重要。

假设我们有两个对象：

```
CWage aWager;
CSales aSales("侯俊杰");
```

销售员是时薪职员之一，因此这样做是合理的：

```
aWager = aSales; // 合理，销售员必定是时薪职员。
```

这样就不合理：

```
aSales = aWager; // 错误，时薪职员未必是销售员。
```

如果你一定要转换，必须使用指针，并且明显地做型别转换（cast）动作:

```
CWage* pWager;
CSales* pSales;
CSales aSales("侯俊杰");
pWager = &aSales; // 把一个「基础类别指针」指向衍生类别之对象，合理且自然。
pSales = (CSales *)pWager; // 强迫转型。语法上可以，但不符合现实生活。
```

真实世界中某些时候我们会以「一种动物」来总称猫啊、狗啊、兔子猴子等等。为了某种便利（这个便利稍后即可看到），我们也会想以「一个通用的指针」表示所有可能的职员类型。无论如何，销售员、时薪职员、经理，都是职员，所以下面动作合情合理：

```
CEmployee* pEmployee;
CWage aWager("曾铭源");
CSales aSales("侯俊杰");
CManager aManager("陈美静");
pEmpolyee = &aWager; // 合理，因为时薪职员必是职员
pEmpolyee = &aSales; // 合理，因为销售员必是职员
pEmpolyee = &aManager; // 合理，因为经理必是职员
```

也就是说，你可以把一个「职员指针」指向任何一种职员。这带来的好处是程序设计的巨大弹性，譬如说你设计一个串行（linked list），各个元素都是职员（哪一种职员都可以），你的*add* 函数可能因此希望有一个「职员指针」作为参数：

```
add(CEmployee* pEmp); // pEmp 可以指向任何一种职员
```

晴天霹雳

我们渐渐接触问题的核心。上述C++ 性质使真实生活经验的确在计算机语言中仿真了出来，但是万里无云的日子里却出现了一个晴天霹雳：如果你以一个「基础类别之指针」指向一个「衍生类别之对象」，那么经由此指针，你就只能够调用基础类别（而不是衍生类别）所定义的函数。因此：

```
CSales aSales("侯俊杰")；
CSales* pSales;
CWage* pWager;

pSales = &aSales;
pWager = &aSales; // 以「基础类别之指针」指向「衍生类别之对象」

pWager->setSales(800.0); // 错误（编译器会检测出来),
// 因为CWage 并没有定义setSales 函数。
pSales->setSales(800.0); // 正确，调用CSales::setSales 函数。
```

虽然*pSales* 和*pWager* 指向同一个对象，但却因指针的原始类型而使两者之间有了差异。

延续此例，我们看另一种情况：

```
pWager->computePay(); // 调用CWage::computePay()
pSales->computePay(); // 调用CSales::computePay()
```

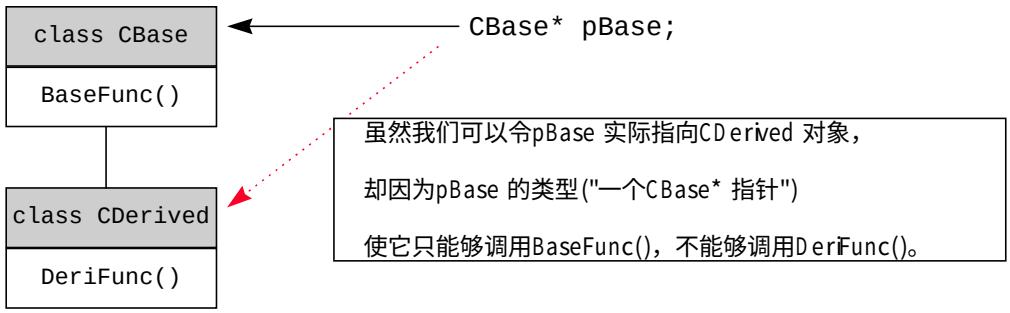
虽然*pSales* 和*pWager* 实际上都指向*CSales* 对象，但是两者调用的*computePay* 却不

相同。到底调用到哪个函数，必须视指针的原始类型而定，与指针实际所指之对象无关。

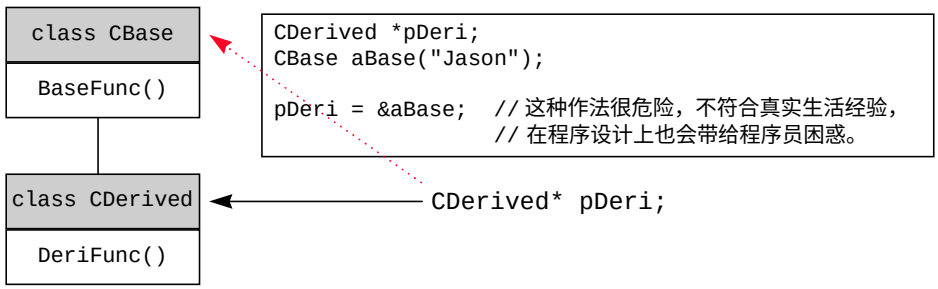
三个结论

我们得到了三个结论：

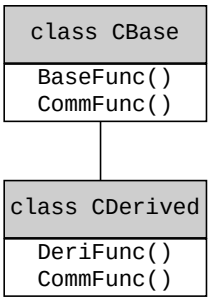
1. 如果你以一个「基础类别之指针」指向「衍生类别之对象」，那么经由该指针你只能够调用基础类别所定义的函数。



2. 如果你以一个「衍生类别之指针」指向一个「基础类别之对象」，你必须先做明显的转型动作（explicit cast）。这种作法很危险，不符合真实生活经验，在程序设计上也会带给程序员困惑。



3. 如果基础类别和衍生类别都定义了「相同名称之成员函数」，那么透过对象指针调用成员函数时，到底调用到哪一个函数，必须视该指针的原始型别而定，而不是视指针实际所指之对象的型别而定。这与第 1 点其实意义相通。




```
CBase* pBase;
CDerived* pDeri;
```

不论你把这两个指针指向何方，由于它们的原始类型，
使它们在调用同名的Com m Func() 时有着无可改变的宿命:

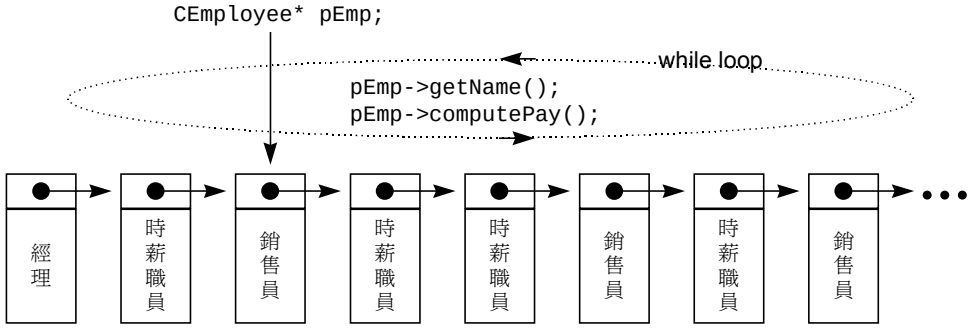
- pBase->CommFunc() 永远是指CBase::CommFunc
- pDeri->CommFunc() 永远是指CDerived::CommFunc

得到这些结论后，看看什么事情会困扰我们。前面我曾提到一个由职员组成的串行，如果我想写一个*printNames* 函数走访串行中的每一个元素并印出职员的名字，我们可以在*CEmployee*（最基础类别）中多加一个*getName* 函数，然后再设计一个*while* 循环如下：

```
int count = 0;
CEmployee* pEmp;
...
while (pEmp = anIter.getNext())
{
   count++;
   cout << count << ' ' << pEmp->getName() << endl;
}
```

你可以把*anIter.getNext* 想象是一个可以走访串行的函数，它传回*CEmPloyee**，也因此每一次获得的指针才可以调用定义于*CEmployee* 中的*getName*。

计薪循环图



但是，由于函数的调用是依赖指针的原始类型而不管它实际上指向何方（何种对象），因此如果上述*while* 循环中调用的是*pEmp->computePay*，那么*while* 循环所执行的将总是相同的运算，也就是*CEmployee*::*computePay*，这就糟了（销售员领到经理的薪水还不糟吗）。更糟的是，我们根本没有定义*CEmployee*::*computePay*，因为*CEmployee* 只是个抽象概念（一个抽象类别）。指针必须落实到具象类型上如*CWage* 或*CManager* 或*CSales*，才有薪资计算公式。

虚拟函数与一般化

我想你可以体会，上述的*while* 循环其实就是把动作「一般化」。「一般化」之所以重要，在于它可以把现在的、未来的情况统统纳入考量。将来即使有另一种名曰「顾问」的职员，上述计薪循环应该仍然能够正常运作。当然啦，「顾问」的*computePay* 必须设计好。

「一般化」是如此重要，解决上述问题因此也就迫切起来。我们需要的是什么呢？是能够「依旧以*CEmpolyee* 指针代表每一种职员」，而又能够在「实际指向不同种类之职员」时，「调用到不同版本（不同类别中）之*computePay*」这种能力。

这种性质就是多态（polymorphism），靠虚拟函数来完成。

再次看看那张计薪循环图：

- 当*pEmp* 指向经理，我希望*pEmp->computePay* 是经理的薪水计算式，也就是*CManager*::*computePay*。
- 当*pEmp* 指向销售员，我希望*pEmp->computePay* 是销售员的薪水计算式，也就是*CSales*::*computePay*。
- 当*pEmp* 指向时薪职员，我希望*pEmp->computePay* 是时薪职员的薪水计算式，也就是*CWage*::*computePay*。

虚拟函数正是为了对「如果你以一个基础类别之指针指向一个衍生类别之对象，那么透过该指针你就只能够调用基础类别所定义之成员函数」这条规则反其道而行的设计。

不必设计复杂的串行函数如*add* 或*getNext* 才能验证这件事，我们看看下面这个简单例子。如果我把职员一例中所有四个类别的*computePay* 函数前面都加上*virtual* 保留字，使它们成为虚拟函数，那么：

```
CEmployee* pEmp;
CWage     aWager("曾銘源");
CSales    aSales("侯俊傑");
CManager   aManager("陳美靜");

pEmp = &aWager;
cout << pEmp->computePay();  //  调用的是 CWage::computePay
pEmp = &aSales;
cout << pEmp->computePay();  //  调用的是 CSales::computePay
pEmp = &aManager;
cout << pEmp->computePay();  //  调用的是 CManager::computePay
```

现在重新回到Shape 例子，我打算让*display* 成为虚拟函数：

```
#0001  #include <iostream.h>
#0002  class CShape
#0003  {
#0004    public:
#0005    virtual void display() { cout << "Shape \n"; }
#0006  };
#0007  //------------------------------------------------
#0008  class CEllipse : public CShape
#0009  {
#0010    public:
#0011    virtual void display() { cout << "Ellipse \n"; }
#0012  };
#0013  //------------------------------------------------
#0014  class CCircle : public CEllipse
#0015  {
#0016    public:
#0017    virtual void display() { cout << "Circle \n"; }
#0018  };
#0019  //------------------------------------------------
#0020  class CTriangle : public CShape
#0021  {
#0022    public:
#0023    virtual void display() { cout << "Triangle \n"; }
#0024  };
```

```
#0025  //-----------------------------------------------
#0026  class CRect : public CShape
#0027  {
#0028    public:
#0029    virtual void display() { cout << "Rectangle \n"; }
#0030  };
#0031  //-----------------------------------------------
#0032  class CSquare : public CRect
#0033  {
#0034    public:
#0035    virtual void display() { cout << "Square \n"; }
#0036  };
#0037  //-----------------------------------------------
#0038  void main()
#0039  {
#0040  CShape       aShape;
#0041  CEllipse     aEllipse;
#0042  CCircle      aCircle;
#0043  CTriangle    aTriangle;
#0044  CRect        aRect;
#0045  CSquare      aSquare;
#0046  CShape* pShape[6] = { &aShape,
#0047                       &aEllipse,
#0048                       &aCircle,
#0049                       &aTriangle,
#0050                       &aRect,
#0051                       &aSquare };
#0052
#0053    for (int i=0; i< 6; i++)
#0054         pShape[i]->display();
#0055  }
#0056  //-----------------------------------------------
```

得到的結果是：

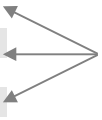
```
Shape
Ellipse
Circle
Triangle
Rectangle
Square
```

如果把所有类别中的virtual 保留字拿掉，执行结果变成：

```
Shape
Shape
Shape
Shape
Shape
Shape
```

综合Employee 和Shape 两例，第一个例子是：

```
pEmp = &aWager;
cout << pEmp->computePay();
pEmp = &aSales;
cout << pEmp->computePay();
pEmp = &aBoss;
cout << pEmp->computePay();
```



第二个例子是：

```
CShape* pShape[6];
for (int i=0; i< 6; i++)
    pShape[i]->display();  // 此进程序代码执行了6 次。
```

我们看到了一种奇特现象：程序代码完全一样（因为一般化了），执行结果却不相同。这就是虚拟函数的妙用。

如果没有虚拟函数这种东西，你还是可以使用scope resolution operator（::）明白指出调用哪一个函数，但程序就不再那么优雅与弹性了。

从操作型定义来看，什么是虚拟函数呢？如果你预期衍生类别有可能重新定义某一个成员函数，那么你就在基础类别中把此函数设为*virtual*。MFC 有两个十分十分重要的虚拟函数：与document 有关的*Serialize* 函数和与view 有关的*OnDraw* 函数。你应该在自己的*CMyDoc* 和*CMyView* 中改写这两个虚拟函数。

多态（Polymorphism）

你看，我们以相同的指令却唤起了不同的函数，这种性质称为Polymorphism，意思是"the ability to assume many forms"（多态）。编译器无法在编译时期判断*pEmp->computePay* 到底是调用哪一个函数，必须在执行时期才能评估之，这称为后期绑定late binding 或动态绑定dynamic binding。至于C 函数或C++ 的non-virtual 函数，在编译时期就转换为一个固定地址的调用了，这称为前期绑定early binding 或静态绑定static binding。

Polymorphism 的目的，就是要让处理「基础类别之对象」的程序代码，能够完全透通地继续适当处理「衍生类别之对象」。

可以说，虚拟函数是了解多态（Polymorphism）以及动态绑定的关键。同时，它也是了解如何使用MFC 的关键。

让我再次提示你，当你设计一套类别，你并不知道使用者会衍生什么新的子类别出来。如果动物世界中出现了新品种名曰雅虎，类别使用者势必在*CAnimal* 之下衍生一个*CYahoo*。饶是如此，身为基础类别设计者的你，可以利用虚拟函数的特性，将所有动物必定会有的行为（例如哮叫*roar*），规划为虚拟函数，并且规划一些一般化动作（例如「让每一种动物发出一声哮叫」）。那么，虽然，你在设计基础类别以及这个一般化动作时，无法掌握使用者自行衍生的子类别，但只要他改写了*roar* 这个虚拟函数，你的一般化对象操作动作自然就可以调用到该函数。

再次回到前述的Shape 例子。我们说*CShape* 是抽象的，所以它根本不该有*display* 这个动作。但为了在各具象衍生类别中绘图，我们又不得不在基础类别*CShape* 加上*display* 虚拟函数。你可以定义它什么也不做（空函数）：

```
class CShape
{
public:
   virtual void display() { }
};
```

或只是给个消息：

```
class CShape
{
public:
   virtual void display() { cout << "Shape \n"; }
};
```

这两种作法都不高明，因为这个函数根本就不应该被调用（*CShape* 是抽象的），我们根本就不应该定义它。不定义但又必须保留一块空间（spaceholder）给它，于是C++ 提供了所谓的纯虚拟函数：

```
class CShape
{
public:
   virtual void display() = 0; // 注意"= 0"
};
```

纯虚拟函数不需定义其实际动作，它的存在只是为了在衍生类别中被重新定义，只是为了提供一个多态接口。只要是拥有纯虚拟函数的类别，就是一种抽象类别，它是不能够被具象化（instantiate）的，也就是说，你不能根据它产生一个对象（你怎能说一种形状为'Shape' 的物体呢）。如果硬要强渡关山，会换来这样的编译消息：

```
error : illegal attempt to instantiate abstract class.
```

关于抽象类别，我还有一点补充。*CCircle* 继承了*CShape* 之后，如果没有改写*CShape* 中的纯虚拟函数，那么*CCircle* 本身也就成为一个拥有纯虚拟函数的类别，于是它也是一个抽象类别。

是对虚拟函数做结论的时候了：

- 如果你期望衍生类别重新定义一个成员函数，那么你应该在基础类别中把此函数设为*virtual*。
- 以单一指令唤起不同函数，这种性质称为Polymorphism，意思是"the ability to assume many forms"，也就是多态。
- 虚拟函数是C++ 语言的Polymorphism 性质以及动态绑定的关键。

- 既然抽象类别中的虚拟函数不打算被调用，我们就不应该定义它，应该把它设为纯虚拟函数（在函数声明之后加上"=0" 即可）。
- 我们可以说，拥有纯虚拟函数者为抽象类别（abstract Class），以别于所谓的具象类别（concrete class)。
- 抽象类别不能产生出对象实体，但是我们可以拥有指向抽象类别之指针，以便于操作抽象类别的各个衍生类别。
- 虚拟函数衍生下去仍为虚拟函数，而且可以省略*virtual* 关键词。

类别与对象大解剖

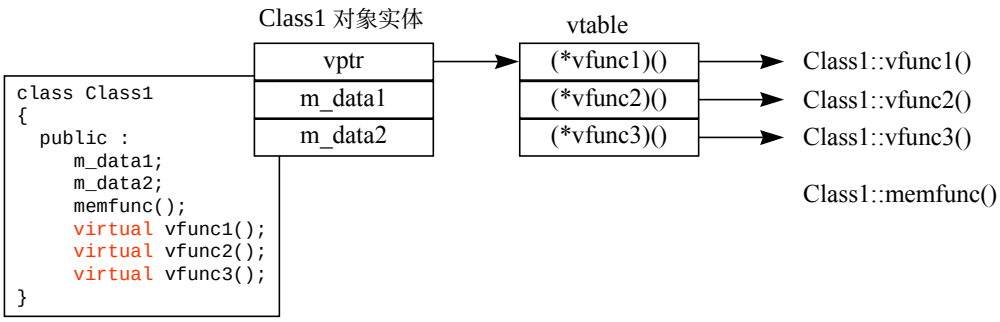
你一定很想知道虚拟函数是怎么做出来的，对不对？

如果能够了解C++ 编译器对于虚拟函数的实现方式，我们就能够知道为什么虚拟函数可以做到动态绑定。

为了达到动态绑定（后期绑定）的目的，C++ 编译器透过某个表格，在执行时期「间接」调用实际上欲绑定的函数（注意「间接」这个字眼）。这样的表格称为虚拟函数表（常被称为vtable）。每一个「内含虚拟函数的类别」，编译器都会为它做出一个虚拟函数表，表中的每一笔元素都指向一个虚拟函数的地址。此外，编译器当然也会为类别加上一项成员变量，是一个指向该虚拟函数表的指针（常被称为vptr）。举个例：

```
class Class1 {
      public :
      data1;
      data2;
      memfunc();
      virtual vfunc1();
      virtual vfunc2();
      virtual vfunc3();
};
```

Class1 对象实体在内存中占据这样的空间：

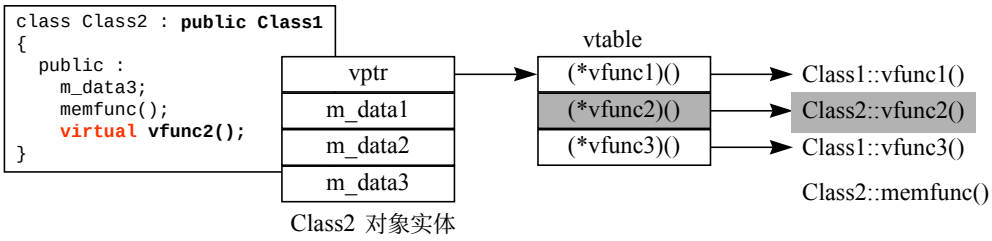


C++ 类别的成员函数，你可以想象就是C 语言中的函数。它只是被编译器改过名称，并增加一个参数（this 指针），因而可以处理调用者（C++ 对象）中的成员变量。所以，你并没有在Class1 对象的内存区块中看到任何与成员函数有关的任何东西。

每一个由此类别衍生出来的对象，都有这么一个vptr。当我们透过这个对象调用虚拟函数，事实上是透过vptr 找到虚拟函数表，再找出虚拟函数的真正地址。

奥妙在于这个虚拟函数表以及这种间接调用方式。虚拟函数表的内容是依据类别中的虚拟函数声明次序，一一填入函数指针。衍生类别会继承基础类别的虚拟函数表（以及所有其它可以继承的成员），当我们在衍生类别中改写虚拟函数时，虚拟函数表就受了影响：表中元素所指的函数地址将不再是基础类别的函数地址，而是衍生类别的函数地址。看看这个例子：

```
class Class2 : public Class1 {
  public :
    data3;
    memfunc();
    virtual vfunc2();
};
```



于是，一个「指向*Class1* 所生对象」的指针，所调用的*vfunc2* 就是*Class1*::*vfunc2*，而一个「指向*Class2* 所生对象」的指针，所调用的*vfunc2* 就是*Class2*::*vfunc2*。

动态绑定机制，在执行时期，根据虚拟函数表，做出了正确的选择。

我们解开了第二道神秘。

口说无凭，何不看点实际。观其地址，物焉 C哉，下面是一个测试程序：

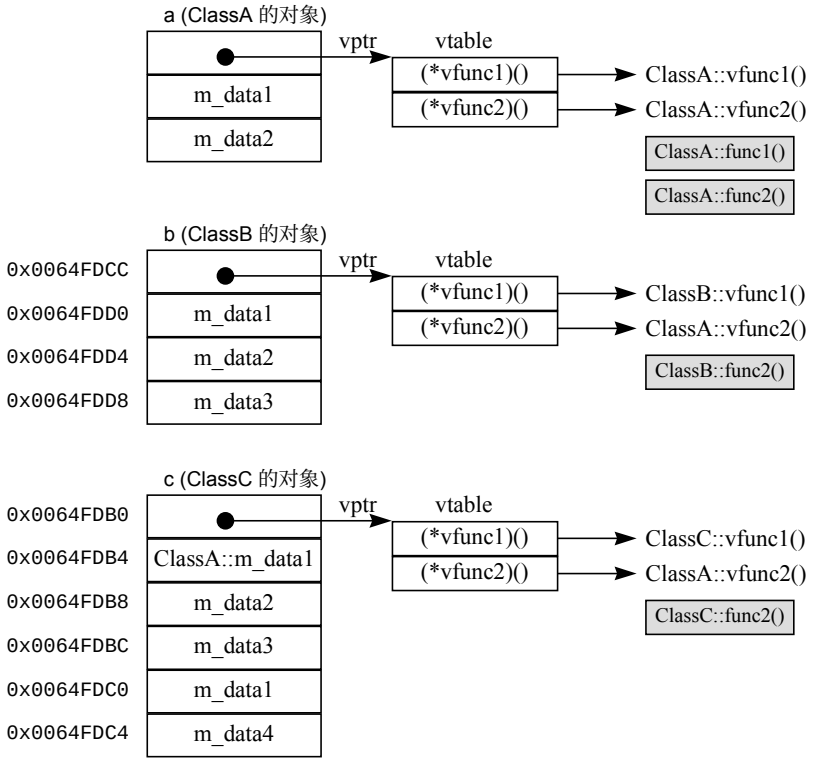
```
#0001  #include <iostream.h>
#0002  #include <stdio.h>
#0003
#0004  class ClassA
#0005  {
#0006  public:
#0007  int m_data1;
#0008  int m_data2;
#0009  void func1() {   }
#0010  void func2() {   }
#0011  virtual void vfunc1() {    }
#0012  virtual void vfunc2() {    }
#0013  };
#0014
#0015  class ClassB : public ClassA
#0016  {
#0017  public:
#0018  int m_data3;
#0019  void func2() {  }
#0020  virtual void vfunc1() {   }
#0021  };
#0022
#0023  class ClassC : public ClassB
#0024  {
```

```
#0025  public:
#0026  int m_data1;
#0027  int m_data4;
#0028  void func2() {  }
#0029  virtual void vfunc1() {    }
#0030  };
#0031
#0032  void main()
#0033  {
#0034    cout << sizeof(ClassA) << endl;
#0035    cout << sizeof(ClassB) << endl;
#0036    cout << sizeof(ClassC) << endl;
#0037
#0038    ClassA a;
#0039    ClassB b;
#0040    ClassC c;
#0041
#0042    b.m_data1 = 1;
#0043    b.m_data2 = 2;
#0044    b.m_data3 = 3;
#0045    c.m_data1 = 11;
#0046    c.m_data2 = 22;
#0047    c.m_data3 = 33;
#0048    c.m_data4 = 44;
#0049    c.ClassA::m_data1 = 111;
#0050
#0051    cout << b.m_data1 << endl;
#0052    cout << b.m_data2 << endl;
#0053    cout << b.m_data3 << endl;
#0054    cout << c.m_data1 << endl;
#0055    cout << c.m_data2 << endl;
#0056    cout << c.m_data3 << endl;
#0057    cout << c.m_data4 << endl;
#0058    cout << c.ClassA::m_data1 << endl;
#0059
#0060    cout << &b << endl;
#0061    cout << &(b.m_data1) << endl;
#0062    cout << &(b.m_data2) << endl;
#0063    cout << &(b.m_data3) << endl;
#0064    cout << &c << endl;
#0065    cout << &(c.m_data1) << endl;
#0066    cout << &(c.m_data2) << endl;
#0067    cout << &(c.m_data3) << endl;
#0068    cout << &(c.m_data4) << endl;
#0069    cout << &(c.ClassA::m_data1) << endl;
#0070  }
```

执行结果与分析如下：

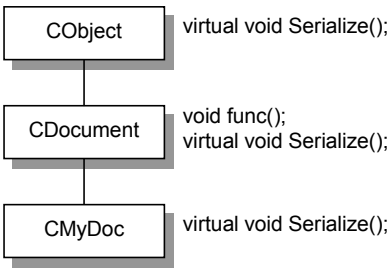
执行結果	意义	说明
12	Sizeof (ClassA)	2 個 *int* 加上一個 vptr
16	Sizeof (ClassB)	继承自 *ClassA*，再加上 1 个 *int*
24	Sizeof (ClassC)	继承自 *ClassB*，再加上 2 个 *int*
1	b.m_data1 的內容	
2	b.m_data2 的內容	
3	b.m_data3 的內容	
11	c.m_data1 的內容	
22	c.m_data2 的內容	
33	c.m_data3 的內容	
44	c.m_data4 的內容	
111	c.ClassA::m_data1 的內容	
0x0064FDCC	b 对象的起始地址	这个地址中的內容就是 vptr
0x0064FDD0	b.m_data1 的地址	
0x0064FDD4	b.m_data2 的地址	
0x0064FDD8	b.m_data3 的地址	
0x0064FDB0	c 对象的起始地址	这个地址中的內容就是 vptr
0x0064FDC0	c.m_data1 的地址	
0x0064FDB8	c.m_data2 的地址	
0x0064FDBC	c.m_data3 的地址	
0x0064FDC4	c.m_data4 的地址	
0x0064FDB4	c.ClassA::m_data1 的地址	

a、b、c 对象的内容图示如下：



Object slicing 与虚拟函数

我要在这里说明虚拟函数另一个极重要的行为模式。假设有三个类别，阶层关系如下：



以程序表现如下：

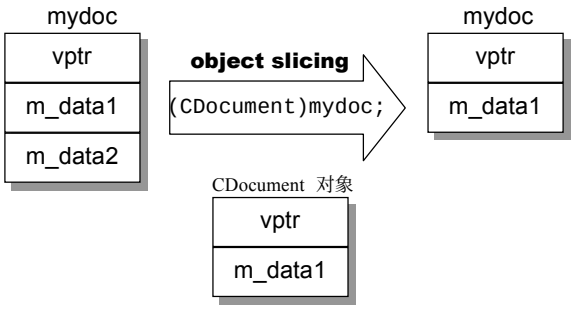
```
#0001  #include <iostream.h>
#0002
#0003  class CObject
#0004  {
#0005  public:
#0006    virtual void Serialize() { cout << "CObject::Serialize() \n\n"; }
#0007  };
#0008
#0009  class CDocument : public CObject
#0010  {
#0011  public:
#0012    int m_data1;
#0013    void func() { cout << "CDocument::func()" << endl;
#0014                  Serialize();
#0015                }
#0016
#0017    virtual void Serialize() { cout << "CDocument::Serialize() \n\n"; }
#0018  };
#0019
#0020  class CMyDoc : public CDocument
#0021  {
#0022  public:
#0023    int m_data2;
#0024    virtual void Serialize() { cout << "CMyDoc::Serialize() \n\n"; }
#0025  };
#0026  //---------------------------------------------------------------
#0027  void main()
#0028  {
#0029  CMyDoc  mydoc;
#0030  CMyDoc* pmydoc = new CMyDoc;
#0031
#0032    cout << "#1 testing" << endl;
#0033    mydoc.func();
#0034
#0035    cout << "#2 testing" << endl;
#0036    ((CDocument*)(&mydoc))->func();
#0037
#0038    cout << "#3 testing" << endl;
#0039    pmydoc->func();
#0040
#0041    cout << "#4 testing" << endl;
#0042    ((CDocument)mydoc).func();
#0043  }
```

由于*CMyDoc* 自己没有*func* 函数，而它继承了*CDocument* 的所有成员，所以*main* 之中的四个调用动作毫无问题都是调用*CDocument*::*func*。但，*CDocument*::*func* 中所调用的*Serialize* 是哪一个类别的成员函数呢？如果它是一般（non-virtual）函数，毫无问题应该是*CDocument*::*Serialize*。但因为这是个虚拟函数，情况便有不同。以下是执行结果：

```
#1 testing
CDocument::func()
CMyDoc::Serialize()

#2 testing
CDocument::func()
CMyDoc::Serialize()

#3 testing
CDocument::func()
CMyDoc::Serialize()

#4 testing
CDocument::func()
CDocument::Serialize()  <-- 注意
```

前三个测试都符合我们对虚拟函数的期望：既然衍生类别已经改写了虚拟函数*Serialize*，那么理当调用衍生类别之*Serialize* 函数。这种行为模式非常频繁地出现在application framework 身上。后续当我追踪MFC 源代码时，遇此情况会再次提醒你。

第四项测试结果则有点出乎意料之外。你知道，衍生对象通常都比基础对象大（我是指内存空间），因为衍生对象不但继承其基础类别的成员，又有自己的成员。那么所谓的upcasting（向上强制转型）：(CDocument)mydoc，将会造成对象的内容被切割（object slicing）：



当我们调用：

```
((CDocument)mydoc).func();
```

mydoc **已经是一个被切割得剩下半条命的对象，而***func* **内部调用虚拟函数***Serialize***；后者将使用的「***mydoc* **的虚拟函数指针」虽然存在，它的值是什么呢？你是不是隐隐觉得有什么大灾难要发生？**

幸运的是，由于`((CDocument)mydoc).func()` **是个传值而非传址动作，编译器以所谓的拷贝构造式（**copy constructor**）把***CDocument* **对象内容复制了一份，使得***mydoc* **的** vtable **内容与***CDocument* **对象的**vtable **相同。本例虽没有明显做出一个拷贝构造式，编译器会自动为你合成一个。**

说这么多，总结就是，经过所谓的data slicing**，本例的***mydoc* **真正变成了一个完完全全**

静态成员（变量与函数）

我想你已经很清楚了，如果你依据一个类别产生出三个对象，每一个对象将各有一份成员变量。有时候这并不是你要的。假设你有一个类别，专门用来处理存款帐户，它至少应该要有存户的姓名、地址、存款额、利率等成员变量：

```
class SavingAccount
{
private:
  char m_name[40];  // 存戶姓名
  char m_addr[60];  // 存戶地址
  double m_total;   // 存款額
  double m_rate;    // 利率
...
};
```

这家行库采用浮动利率，每个帐户的利息都是根据当天的挂牌利率来计算。这时候 *m_rate* 就不适合成为每个帐户对象中的一笔资料，否则每天一开市，光把所有帐户内容

叫出来，修改*m_rate* 的值，就花掉不少时间。*m_rate* 应该独立在各对象之外，成为类别独一无二的资料。怎么做？在*m_rate* 前面加上*static* 修饰词即可：

```
class SavingAccount
{
private:
char m_name[40]; // 存户姓名
char m_addr[60]; // 存户地址
double m_total; // 存款额
static double m_rate; // 利率
...
};
```

static 成员变量不属于对象的一部份，而是类别的一部份，所以程序可以在还没有诞生任何对象的时候就处理此种成员变量。但首先你必须初始化它。

不要把*static* 成员变量的初始化动作安排在类别的构造式中，因为构造式可能一再被调用，而变量的初值却只应该设定一次。也不要把初始化动作安排在头文件中，因为它可能会被包含许多地方，因此也就可能被执行许多次。你应该在实作档中且类别以外的任何位置设定其初值。例如在*main* 之中，或全域函数中，或任何函数之外：

```
double SavingAccount::m_rate = 0.0075; // 设立static 成员变量的初值
void main() { ... }
```

这么做可曾考虑到*m_rate* 是个*private* 资料？没关系，设定*static* 成员变量初值时，不受任何存取权限的束缚。请注意，*static* 成员变量的型别也出现在初值设定句中，因为这是一个初值设定动作，不是一个数量指定（assignment）动作。事实上，*static* 成员变量是在这时候（而不是在类别声明中）才定义出来的。如果你没有做这个初始化动作，会产生联结错误：

```
error LNK2001: unresolved external symbol "private: static double
                SavingAccount::m_rate"(?m_rate@SavingAccount@@2HA)
```

关于static 成员的使用实例，第 6 章的HelloMFC 有一个，附录D的「自制DBWIN 工具（MFC 版）」也有一个。第 3 章的「RTTI（执行时期型别辨识）」一节仿真MFC 的*CRuntimeClass*，也有一个static 应用实例。

下面是存取*static* 成员变量的一种方式，注意，此刻还没有诞生任何对象实体：

```
// 第一种存取方式
void main()
{
  SavingAccount::m_rate = 0.0075; // 欲此行成立，须把 m_rate 改为public
}
```

下面这种情况则是产生一个对象后，透过对象来处理*static* 成员变量：

```
// 第二种存取方式
void main()
{
  SavingAccount myAccount;
  myAccount.m_rate = 0.0075;  // 欲此行成立，须把 m_rate 改为public
}
```

你得搞清楚一个观念，*static* 成员变量并不是因为对象的实现而才得以实现，它本来就存在，你可以想象它是一个全域变量。因此，第一种处理方式在意义上比较不会给人错误的印象。

只要access level 允许，任何函数（包括全域函数或成员函数，static 或non-static）都可以存取static 成员变量。但如果你希望在产生任何object 之前就存取其class 的private static 成员变量，则必须设计一个static 成员函数（例如以下的s*etRate*）：

```
class SavingAccount
{
private:
  char m_name[40];  // 存户姓名
  char m_addr[60];  // 存户地址
  double m_total;   // 存款额
  static double m_rate;    // 利率
  ...
public:
  static void setRate(double newRate) { m_rate = newRate; }
  ...
};

double SavingAccount::m_rate = 0.0075;  // 设置 static 成员变量的初值

void main()
```

```
{
  SavingAccount::setRate(0.0074);  // 直接调用类别的 static 成员函数

  SavingAccount myAccount;
  myAccount.setRate(0.0074);       // 通过对象调用 stati  c 成员函数
}
```

由于*static* 成员函数不需要借助任何对象，就可以被调用执行，所以编译器不会为它暗加一个*this* 指针。也因为如此，*static* 成员函数无法处理类别之中的non-static 成员变量。还记得吗，我在前面说过，成员函数之所以能够以单一一份函数码处理各个对象的资料而不紊乱，完全靠的是*this* 指针的指示。

static 成员函数「没有*this* 参数」的这种性质，正是我们的MFC 应用程序在准备callback 函数时所需要的。第 6 章的Hello World 例中我就会举这样一个实例。

C++ 程序的生与死：兼谈构造式与析构式

C++ 的*new* 运算子和C 的*malloc* 函数都是为了配置内存，但前者比之后者的优点是，*new* 不但配置对象所需的内存空间时，同时会引发构造式的执行。

所谓构造式（constructor），就是对象诞生后第一个执行（并且是自动执行）的函数，它的函数名称必定要与类别名称相同。

相对于构造式，自然就有个析构式（destructor），也就是在对象行将毁灭但未毁灭之前一刻，最后执行（并且是自动执行）的函数，它的函数名称必定要与类别名称相同，再在最前面加一个~ 符号。

一个有着阶层架构的类别群组，当衍生类别的对象诞生之时，构造式的执行是由最基础类别（most based）至最尾端衍生类别（most derived）；当对象要毁灭之前，析构式的执行则是反其道而行。第 3 章的frame1 程序对此有所示范。

我以实例展示不同种类之对象的构造式执行时机。程序代码中的编号请对照执行结果。

```
#0001  #include <iostream.h>
#0002  #include <string.h>
#0003
#0004  class CDemo
#0005  {
#0006  public:
#0007        CDemo(const char* str);
#0008        ~CDemo();
#0009  private:
#0010        char name[20];
#0011  };
#0012
#0013  CDemo::CDemo(const char* str)    // 构造函数
#0014  {
#0015    strncpy(name, str, 20);
#0016    cout << "Constructor called for " << name << '\n';
#0017  }
#0018
#0019  CDemo::~CDemo()    // 析构函数
#0020  {
#0021    cout << "Destructor called for " << name << '\n';
#0022  }
#0023
#0024  void func()
#0025  {
#0026    CDemo LocalObjectInFunc("LocalObjectInFunc");  // in stack ❺
#0027    static CDemo StaticObject("StaticObject");    // local static ❻
#0028    CDemo* pHeapObjectInFunc = new CDemo("HeapObjectInFunc"); // in heap ❼
#0029
#0030    cout << "Inside func" << endl; ❽
#0031
#0032  } ❾
#0033
#0034  CDemo GlobalObject("GlobalObject");  // global static ❶
#0035
#0036  void main()
#0037  {
#0038    CDemo LocalObjectInMain("LocalObjectInMain"); // in stack ❷
#0039    CDemo* pHeapObjectInMain = new CDemo("HeapObjectInMain"); // in heap ❸
#0040
#0041    cout << "In main, before calling func\n"; ❹
#0042    func();
#0043    cout << "In main, after calling func\n"; ❿
#0044
#0045  } ①  ②  ③
```

以下是执行结果：

```
❶ Constructor called for GlobalObject
❷ Constructor called for LocalObjectInMain
❸ Constructor called for HeapObjectInMain
❹ In main, before calling func
❺ Constructor called for LocalObjectInFunc
❻ Constructor called for StaticObject
❼ Constructor called for HeapObjectInFunc
❽ Inside func
❾ Destructor called for LocalObjectInFunc
❿ In main, after calling func
① Destructor called for LocalObjectInMain
② Destructor called for StaticObject
③ Destructor called for GlobalObject
```

我的结论是：

- 对于全域对象（如本例之*GlobalObject*），程序一开始，其构造式就先被执行（比程序进入点更早）；程序即将结束前其析构式被执行。MFC 程序就有这样一个全域对象，通常以application object 称呼之，你将在第 6 章看到它。
- 对于区域对象，当对象诞生时，其构造式被执行；当程序流程将离开该对象的存活范围（以至于对象将毁灭），其析构式被执行。
- 对于静态（static）对象，当对象诞生时其构造式被执行；当程序将结束时（此对象因而将遭致毁灭）其析构式才被执行，但比全域对象的析构式早一步执行。
- 对于以*new* 方式产生出来的区域对象，当对象诞生时其构造式被执行。析构式则在对象被*delete* 时执行（上例程序未示范）。

四种不同的对象生存方式（in stack、in heap、global、local static）

既然谈到了static 对象，就让我把所有可能的对象生存方式及其构造式调用时机做个整理。所有作法你都已经在前一节的小程序中看过。

在C++ 中，有四种方法可以产生一个对象。第一种方法是在堆栈（stack）之中产生它：

```
void MyFunc()
{
   CFoo foo; // 在堆栈（stack）中产生foo 对象
   ...
}
```

第二种方法是在堆积（heap）之中产生它：

```
void MyFunc()
{
   ...
   CFoo* pFoo = new CFoo(); // 在堆（heap）中产生对象
}
```

第三种方法是产生一个全域对象（同时也必然是个静态对象）：

```
CFoo foo; // 在任何函数范围之外做此动作
```

第四种方法是产生一个区域静态对象：

```
void MyFunc()
{
   static CFoo foo; // 在函数范围（scope）之内的一个静态对象
...
}
```

不论任何一种作法，C++ 都会产生一个针对*CFoo* 构造式的调用动作。前两种情况，C++ 在配置内存-- 来自堆栈（stack）或堆积（heap）-- 之后立刻产生一个隐藏的（你的原代码中看不出来的）构造式调用。第三种情况，由于对象实现于任何「函数活动范围（function scope）」之外，显然没有地方来安置这样一个构造式调用动作。

是的，第三种情况（静态全域对象）的构造式调用动作必须靠startup 码帮忙。startup 码是什么？是更早于程序进入点（*main* 或*WinMain*）执行起来的码，由C++ 编译器提供，被联结到你的程序中。startup 码可能做些像函数库初始化、进程信息设立、I/O stream 产生等等动作，以及对static 对象的初始化动作（也就是调用其构造式）。

当编译器编译你的程序，发现一个静态对象，它会把这个对象加到一个串行之中。更精

确地说则是，编译器不只是加上此静态对象，它还加上一个指针，指向对象之构造式及其参数（如果有的话）。把控制权交给程序进入点（*main* 或*WinMain*）之前，startup 码会快速在该串行上移动，调用所有登记有案的构造式并使用登记有案的参数，于是就初始化了你的静态对象。

第四种情况（区域静态对象）相当类似C 语言中的静态区域变量，只会有一个实体（instance）产生，而且在固定的内存上（既不是stack 也不是heap）。它的构造式在控制权第一次移转到其声明处（也就是在*MyFunc* 第一次被调用）时被调用。

所谓 〝Unwinding〞

C++ 对象依其生存空间，适当地依照一定的顺序被析构（destructed）。但是如果发生异常情况（exception），而程序设计了异常情况处理程序（exception handling），控制权就会截弯取直地「直接跳」到你所设定的处理例程去，这时候堆栈中的C++ 对象有没有机会被析构？这得视编译器而定。如果编译器有支持unwinding 功能，就会在一个异常情况发生时，将堆栈中的所有对象都析构掉。

关于异常情况（exception）及异常处理（exception handling），稍后有一节讨论之。

执行时期型别信息（RTTI）

我们有可能在程序执行过程中知道某个对象是属于哪一种类别吗？这种在C++ 中称为执行时期型别信息（Runtime Type Information，RTTI）的能力，晚近较先进的编译器如Visual C++ 4.0 和Borland C++ 5.0 才开始广泛支持。以下是一个实例：

```
#0001  // RTTI.CPP - built by   C:\> cl.exe -GR rtti.cpp <ENTER>
#0002  #include <typeinfo.h>
#0003  #include <iostream.h>
#0004  #include <string.h>
#0005
#0006  class graphicImage
```

```
#0007  {
#0008  protected:
#0009      char name[80];
#0010
#0011  public:
#0012      graphicImage()
#0013      {
#0014          strcpy(name,"graphicImage");
#0015      }
#0016
#0017      virtual void display()
#0018      {
#0019          cout << "Display a generic image." << endl;
#0020      }
#0021
#0022      char* getName()
#0023      {
#0024          return name;
#0025      }
#0026  };
#0027  //---------------------------------------------------------------
#0028  class GIFimage : public graphicImage
#0029  {
#0030  public:
#0031      GIFimage()
#0032      {
#0033          strcpy(name,"GIFimage");
#0034      }
#0035
#0036      void display()
#0037      {
#0038          cout << "Display a GIF file." << endl;
#0039      }
#0040  };
#0041
#0042  class PICTimage : public graphicImage
#0043  {
#0044  public:
#0045      PICTimage()
#0046      {
#0047          strcpy(name,"PICTimage");
#0048      }
#0049
#0050      void display()
#0051      {
#0052          cout << "Display a PICT file." << endl;
```

```
#0053     }
#0054  };
#0055  //------------------------------------------------------------------
#0056  void processFile(graphicImage *type)
#0057  {
#0058     if (typeid(GIFimage) == typeid(*type))
#0059     {
#0060        ((GIFimage *)type)->display();
#0061     }
#0062     else if (typeid(PICTimage) == typeid(*type))
#0063     {
#0064        ((PICTimage *)type)->display();
#0065     }
#0066     else
#0067        cout << "Unknown type! " << (typeid(*type)).name() << endl;
#0068  }
#0069
#0070  void main()
#0071  {
#0072     graphicImage *gImage = new GIFimage();
#0073     graphicImage *pImage = new PICTimage();
#0074
#0075     processFile(gImage);
#0076     processFile(pImage);
#0077  }
```

执行结果如下：

```
Display a GIF file.
Display a PICT file.
```

这个程序与RTTI 相关的地方有三个：

1. 编译时需选用/GR 选项（/GR 的意思是enable C++ RTTI）
2. 包含typeinfo.h
3. 新的*typeid* 运算子。这是一个多载（overloading）运算子，多载的意思就是拥有一个以上的型式，你可以想象那是一种静态的多态（Polymorphism）。*typeid* 的参数可以是类别名称（如本例#58 左），也可以是对象指针（如本例#58 右）。它传回一个*type_info*&。*type_info* 是一个类别，定义于typeinfo.h 中：

```
class type_info {
```

```
public:
       virtual ~type_info();
       int operator==(const type_info& rhs) const;
       int operator!=(const type_info& rhs) const;
       int before(const type_info& rhs) const;
       const char* name() const;
       const char* raw_name() const;
private:
       ...
};
```

虽然Visual C++ 编译器自从4.0 版已经支持RTTI，但MFC 4.x 并未使用编译器的能力完成其对RTTI 的支持。MFC 有自己一套沿用已久的办法（从1.0 版就开始了）。喔，不要因为MFC 的作法特殊而非难它，想想看它的悠久历史。

MFC 的RTTI 能力牵扯到一组非常神秘的宏（ *DECLARE_DYNAMIC* 、*IMPLEMENT_DYNAMIC*）和一个非常神秘的类别（*CRuntimeClass*）。MFC 程序员都知道怎么用它，却没几个人懂得其运作原理。大道不过三两行，说穿不值一文钱，下一章我就仿真出一个RTTI 的DOS 版本给你看。

动态生成（Dynamic Creation)

对象导向术语中有一个名为persistence，意思是永续存留。放在RAM 中的东西，生命受到电力的左右，不可能永续存留；唯一的办法是把它写到文件去。MFC 的一个术语Serialize，就是做有关文件读写的永续存留动作，并且实做作出一个虚拟函数，就叫作*Serialize*。

看起来永续存留与本节的主题「动态生成」似乎没有什么干连。有！你把你的资料储存到文件，这些资料很可能（通常是）对象中的成员变量 我把它读出来后，势必要依据文件上的记载，重新*new* 出那些个对象来。问题在于，即使我的程序有那些类别定义（就算我的程序和你的程序有一样的内容好了），我能够这么做吗：

```
char className[30] = getClassName(); // 从文件（或使用者输入）获得一个类别名称
CObject* obj = new classname; // 这一行行不通
```

首先，`new classname` 这个动作就过不了关。其次，就算过得了关，*new* 出来的对象究竟该是什么类别类型？虽然以一个指向MFC 类别老祖宗（*CObject*）的对象指针来容纳它绝对没有问题，但终不好总是如此吧！不见得这样子就能够满足你的程序需求啊。显然，你能够以*Serialize* 函数写档，我能够以*Serialize* 函数读档，但我就是没办法恢复你原来的状态-- 除非我的程序能够「动态生成」。

MFC 支持动态生成， 靠的是一组非常神秘的宏（ *DECLARE_DYNCREATE* 、*IMPLEMENT_DYNCREATE*）和一个非常神秘的类别（*CRuntimeClass*）。第 3 章中我将把它抽丝剥茧，以一个DOS 程序仿真出来。

异常处理（Exception Handling)

Exception（异常情况）是一个颇为新鲜的C++ 语言特征，可以帮助你管理执行时期的错误，特别是那些发生在深度巢状（nested）函数调用之中的错误。Watcom C++ 是最早支持ANSI C++ 异常情况的编译器，Borland C++ 4.0 随后跟进，然后是Microsoft Visual C++ 和Symantec C++。现在，这已成为C++ 编译器必需支持的项目。

C++ 的exception 基本上是与C 的*setjmp* 和*longjmp* 函数对等的东西，但它增加了一些功能，以处理C++ 程序的特别需求。从深度巢状的例程调用中直接以一条快捷方式撤回到异常情况处理例程（exception handler），这种「错误管理方式」远比结构化程序中经过层层的例程传回一系列的错误状态来的好。事实上exception handling 是MFC 和OWL 两个application frameworks 的防弹中心。

C++ 导入了三个新的exception 保留字：

1. *try*。之后跟随一段以{ } 圈出来的程序代码，exception 可能在其中发生。
2. *catch*。之后跟随一段以{ } 圈出来的程序代码，那是exception 处理例程之所在。*catch* 应该紧跟在*try* 之后。
3. *throw*。这是一个指令，用来产生（抛出）一个exception。

下面是个实例 ：

```
try {
     // try block.
}
catch (char *p) {
     printf("Caught a char* exception, value %s\n",p);
}
catch (double d) {
     printf("Caught a numeric exception, value %g\n",d);
}
catch (...) {  // catch anything
     printf("Caught an unknown exception\n");
}
```

MFC 早就支持exception，不过早期它用的是非标准语法。Visual C++ 4.0 编译器本身支持完整的C++ exceptions，MFC 也因此有了两个exception 版本：你可以使用语言本身提供的性能，也可以沿用MFC 古老的方法（以宏形式出现）。人们曾经因为MFC 的方案不同于ANSI 标准而非难它，但是不要忘记它已经运作了多少年。

MFC 的exceptions 机制是以宏和exception types 为基础。这些宏类似C++ 的exception 保留字，动作也满像。MFC 以下列宏仿真C++ exception handling：

```
TRY
CATCH(type,object)
AND_CATCH(type,object)
END_CATCH
CATCH_ALL(object)
AND_CATCH_ALL(object)
END_CATCH_ALL
END_TRY
THROW()
THROW_LAST()
```

MFC 所使用的语法与日渐浮现的标准稍微不同，不过其间差异微不足道。为了以MFC 捕捉exceptions，你应该建立一个*TRY* 区块，下面接着*CATCH* 区块：

```
TRY {
     // try block.
}
CATCH (CMemoryException, e) {
```

```
        printf("Caught a memory exception.\n");
    }
    AND_CATCH_ALL (e) {
        printf("Caught an exception.\n");
    }
    END_CATCH_ALL
```

THROW 宏相当于C++ 语言中的*throw* 指令；你以什么类型做为*THROW* 的参数，就会有一个相对应的*AfxThrow_* 函数被调用（这是台面下的行为）：

MFC Exception Type	MFC Throw Function	DOS support	Windows support
CException		v	v
CMemoryException	AfxThrowMemoryException	v	v
CFileException	AfxThrowFileException	v	v
CArchiveException	AfxThrowArchiveException	v	v
CNotSupportedException	AfxThrowNotSupportedException	v	v
CResourceException	AfxThrowResourceException		v
COleException	AfxThrowOleException		v
COleDispatchException	AfxThrowOleDispatchException		v
CDBException	AfxThrowDBException		v
CDaoException	AfxThrowDaoException		v
CUserException	AfxThrowUserException		v

以下是MFC 4.x 的exceptions 宏定义:

```
// in AFX.H
/////////////////////////////////////////////////////////////////////////
// Exception macros using try, catch and throw
//  (for backward compatibility to previous versions of MFC)

#ifndef _AFX_OLD_EXCEPTIONS

#define TRY { AFX_EXCEPTION_LINK _afxExceptionLink; try {
```

```
#define CATCH(class, e) } catch (class* e) \
       { ASSERT(e->IsKindOf(RUNTIME_CLASS(class))); \
              _afxExceptionLink.m_pException = e;

#define AND_CATCH(class, e) } catch (class* e) \
       { ASSERT(e->IsKindOf(RUNTIME_CLASS(class))); \
              _afxExceptionLink.m_pException = e;

#define END_CATCH } }

#define THROW(e) throw e
#define THROW_LAST() (AfxThrowLastCleanup(), throw)

// Advanced macros for smaller code
#define CATCH_ALL(e) } catch (CException* e) \
       { { ASSERT(e->IsKindOf(RUNTIME_CLASS(CException))); \
              _afxExceptionLink.m_pException = e;

#define AND_CATCH_ALL(e) } catch (CException* e) \
       { { ASSERT(e->IsKindOf(RUNTIME_CLASS(CException))); \
              _afxExceptionLink.m_pException = e;

#define END_CATCH_ALL } } }

#define END_TRY } catch (CException* e) \
       { ASSERT(e->IsKindOf(RUNTIME_CLASS(CException))); \
              _afxExceptionLink.m_pException = e; } }

#else //_AFX_OLD_EXCEPTIONS

/////////////////////////////////////////////////////////////////////////////
// Exception macros using setjmp and longjmp
//  (for portability to compilers with no support for C++ exception handling)

#define TRY \
       { AFX_EXCEPTION_LINK _afxExceptionLink; \
       if (::setjmp(_afxExceptionLink.m_jumpBuf) == 0)

#define CATCH(class, e) \
       else if (::AfxCatchProc(RUNTIME_CLASS(class))) \
       { class* e = (class*)_afxExceptionLink.m_pException;

#define AND_CATCH(class, e) \
       } else if (::AfxCatchProc(RUNTIME_CLASS(class))) \
       { class* e = (class*)_afxExceptionLink.m_pException;
```

```
#define END_CATCH \
       } else { ::AfxThrow(NULL); } }

#define THROW(e) AfxThrow(e)
#define THROW_LAST() AfxThrow(NULL)

// Advanced macros for smaller code
#define CATCH_ALL(e) \
       else { CException* e = _afxExceptionLink.m_pException;

#define AND_CATCH_ALL(e) \
       } else { CException* e = _afxExceptionLink.m_pException;

#define END_CATCH_ALL } }

#define END_TRY }

#endif //_AFX_OLD_EXCEPTIONS
```

Template

这并不是一本C++ 书籍，我也并不打算介绍太多距离「运用MFC」主题太远的C++ 论题。Template 虽然很重要，但它与「运用MFC」有什么关系？有！第８章当我们开始设计Scribble 程序时，需要用到MFC 的collection classes，而这一组类别自从MFC 3.0 以来就有了template 版本（因为Visual C++ 编译器从2.0 版开始支持C++ template）。运用之前，我们总该了解一下新的语法、精神、以及应用。

好，到底什么是template？重要性如何？Kaare Christian 在1994/01/25 的PC-Magazine 上有一篇文章，说得很好：

> 无性生殖并不只是存在于遗传工程上，对程序员而言它也是一个由来已久的动作。过去，我们只不过是以一个简单而基本的工具，也就是一个文字编辑器，重制我们的程序代码。今天，C++ 提供给我们一个更好的繁殖方法：template。
> 复制一段既有程序代码的一个最平常的理由就是为了改变数据类型。举个例子，假设你写了一个绘图函数，使用整数x, y 坐标；突然之间你需要相同的程序代码，但坐标值改采

long。你当然可以使用一个文字编辑器把这段码拷贝一份，然后把其中的数据类型改变过来。有了C++，你甚至可以使用多载（overloaded）函数，那么你就可以仍旧使用相同的函数名称。函数的多载的确使我们有比较清爽的程序代码，但它们意味着你还是必须在你的程序的许多地方维护完全相同的算法。

C 语言对此问题的解答是：使用宏。虽然你因此对于相同的算法只需写一次程序代码，但宏有它自己的缺点。第一，它只适用于简单的功能。第二个缺点比较严重：宏不提供资料型别检验，因此牺牲了C++ 的一个主要效益。第三个缺点是：宏并非函数，程序中任何调用宏的地方都会被编译器前置处理器原原本本地插入宏所定义的那一段码，而非只是一个函数调用，因此你每使用一次宏，你的执行文件就会膨胀一点。

Templates 提供比较好的解决方案，它把「一般性的算法」和其「对资料型别的实作部份」区分开来。你可以先写算法的程序代码，稍后在使用时再填入实际资料型别。新的C++ 语法使「资料型别」也以参数的姿态出现。有了template，你可以拥有宏「只写一次」的优点，以及多载函数「类型检验」的优点。

C++ 的template 有两种，一种针对function，另一种针对class。

Template Functions

假设我们需要一个计算数值幂次方的函数，名曰*power*。我们只接受正幂次方数，如果是负幂次方，就让结果为0。

对于整数，我们的函数应该是这样：

```
#0001  int power(int base, int exponent)
#0002  {
#0003      int result = base;
#0004      if (exponent == 0)  return (int)1;
#0005      if (exponent < 0)   return (int)0;
#0006      while (--exponent)  result *= base;
#0007      return result;
#0008  }
```

对于长整数，函数应该是这样：

```
#0001  long power(long base, int exponent)
#0002  {
#0003      long result = base;
#0004      if (exponent == 0)  return (long)1;
#0005      if (exponent < 0)   return (long)0;
#0006      while (--exponent)  result *= base;
#0007      return result;
#0008  }
```

对于浮点数，我们应该...，对于复数，我们应该...。喔喔，为什么不能够把资料型别也变成参数之一，在使用时指定呢？是的，这就是template 的妙用：

```
template <class T> T power(T base, int exponent);
```

写成两行或许比较清楚：

```
template <class T>
T power(T base, int exponent);
```

这样的函数声明是以一个特殊的*template* 前缀开始，后面紧跟着一个参数列（本例只一个参数）。容易让人迷惑的是其中的"class" 字眼，它其实并不一定表示C++ 的class，它也可以是一个普通的数据类型。<class T> 只不过是表示：T 是一种类型，而此一类型将在调用此函数时才给予。

下面就是*power* 函数的template 版本：

```
#0001  template <class T>
#0002  T power(T base, int exponent)
#0003  {
#0004      T result = base;
#0005      if (exponent == 0)  return (T)1;
#0006      if (exponent < 0)   return (T)0;
#0007      while (--exponent)  result *= base;
#0008      return result;
#0009  }
```

传回值必须确保为类型T，以吻合template 函数的声明。

下面是template 函数的调用方法：

```
#0001  #include <iostream.h>
#0002  void main()
#0003  {
#0004     int i = power(5, 4);
#0005     long l = power(1000L, 3);
#0006     long double d = power((long double)1e5, 2);
#0007
#0008     cout << "i= " << i << endl;
#0009     cout << "l= " << l << endl;
#0010     cout << "d= " << d << endl;
#0011  }
```

执行结果如下：

```
i= 625
l= 1000000000
d= 1e+010
```

在第一次调用中，T 变成*int*，在第二次调用中，T 变成*long*。而在第三次调用中，T 又成为了一个*long double*。但如果调用时候把数据类型混乱掉了，像这样：

```
int i = power(1000L, 4); // 基值是个long，传回值却是个int。错误示范！
```

编译时就会出错。

template 函数的资料型别参数T 究竟可以适应多少种类型？我要说，几乎「任何资料型态」都可以，但函数中对该类型数值的任何运算动作，都必须支持-- 否则编译器就不知道该怎么办了。以*power* 函数为例，它对于*result* 和*base* 两个数值的运算动作有：

1. T result = base;
2. return (T)1;
3. return (T)0;
4. result *= base;
5. return result;

C++ 所有内建数据类型如*int* 或*long* 都支持上述运算动作。但如果你为某个C++ 类别产生一个*power* 函数，那么这个C++ 类别必须包含适当的成员函数以支持上述动作。

如果你打算在template 函数中以C++ 类别代替class T，你必须清楚知道哪些运算动作曾被使用于此一函数中，然后在你的C++ 类别中把它们全部实作出来。否则，出现的

错误耐人寻味。

Template Classes

我们也可以建立template classes，使它们能够神奇地操作任何类型的资料。下面这个例子是让*CThree* 类别储存三个成员变量，成员函数*Min* 传回其中的最小值，成员函数*Max* 则传回其中的最大值。我们把它设计为template class，以便这个类别能适用于各式各样的数据类型：

```
#0001  template <class T>
#0002  class CThree
#0003  {
#0004  public :
#0005      CThree(T t1, T t2, T t3);
#0006      T Min();
#0007      T Max();
#0008  private:
#0009      T a, b, c;
#0010  };
```

语法还不至于太稀奇古怪，把T 看成是大家熟悉的*int* 或*float* 也就是了。下面是成员函数的定义：

```
#0001  template <class T>
#0002  T CThree<T>::Min()
#0003  {
#0004      T minab = a < b ? a : b;
#0005      return minab < c ? minab : c;
#0006  }
#0007
#0008  template <class T>
#0009  T CThree<T>::Max()
#0010  {
#0011      T maxab = a < b ? b : a;
#0012      return maxab < c ? c : maxab;
#0013  }
#0014
#0015  template <class T>
#0016  CThree<T>::CThree(T t1, T t2, T t3) :
#0017      a(t1), b(t2), c(t3)
#0018  {
```

```
#0019 return;
#0020 }
```

这里就得多注意些了。每一个成员函数前都要加上template <class T>，而且类别名称应该使用CThree<T>。

以下是template class 的使用方式：

```
#0001 #include <iostream.h>
#0002 void main()
#0003 {
#0004 CThree<int> obj1(2, 5, 4);
#0005 cout << obj1.Min() << endl;
#0006 cout << obj1.Max() << endl;
#0007
#0008 CThree<float> obj2(8.52, -6.75, 4.54);
#0009 cout << obj2.Min() << endl;
#0010 cout << obj2.Max() << endl;
#0011
#0012 CThree<long> obj3(646600L, 437847L, 364873L);
#0013 cout << obj3.Min() << endl;
#0014 cout << obj3.Max() << endl;
#0015 }
```

执行结果如下：

```
        2 5
        -6.75
        8.52
        364873
        646600
```

稍早我曾说过，只有当template 函数对于资料型别T 支持所有必要的运算动作时，T 才得被视为有效。此一限制对于template classes 亦属实。为了针对某些类别产生一个*CThree*，该类别必须提供copy 构造式以及operator<，因为它们是*Min* 和*Max* 成员函数中对T 的运算动作。

但是如果你用的是别人template classes，你又如何知道什么样的运算动作是必须的呢？唔，该template classes 的说明文件中应该有所说明。如果没有，只有源代码才能揭露秘密。C++ 内建资料型别如*int* 和*float* 等不需要在意这份要求，因为所有内建的资料类型都支持所有的标准运算动作。

Templates　的编译与联结

对程序员而言C++ templates 可说是十分容易设计与使用，但对于编译器和联结器而言却是一大挑战。编译器遇到一个template 时，不能够立刻为它产生机器码，它必须等待，直到template 被指定某种类型。从程序员的观点来看，这意味着template function 或template class 的完整定义将出现在template 被使用的每一个角落，否则，编译器就没有足够的信息可以帮助产生目的码。当多个源文件使用同一个template 时，事情更趋复杂。

随着编译器的不同，掌握这种复杂度的技术也不同。有一个常用的技术，Borland 称之为Smart，应该算是最容易的：每一个使用Template 的程序代码的目的档中都存在有template码，联结器负责复制和删除。

假设我们有一个程序，包含两个源文件A.CPP 和B.CPP，以及一个THREE.H（其内定义了一个template 类别，名为*CThree*）。A.CPP 和B.CPP 都包含THREE.H。如果A.CPP 以*int* 和*double* 使用这个template 类别，编译器将在A.OBJ 中产生*int* 和*double* 两种版本的template 类别可执行码。如果B.CPP 以*int* 和*float* 使用这个template 类别，编译器将在B.OBJ 中产生*int* 和*float* 两种版本的template 类别可执行码。即使虽然A.OBJ 中已经有一个*int* 版了，编译器没有办法知道。

然后，在联结过程中，所有重复的部份将被删除。请看图2-1。

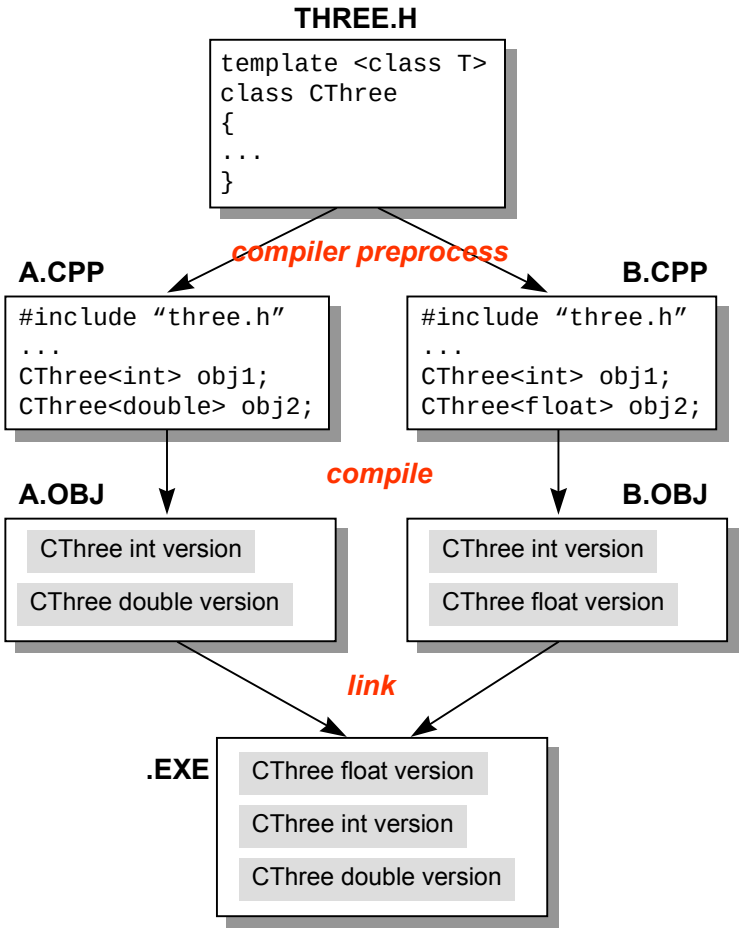


图2-1 联结器会把所有赘余的template 码剔除。这在Borland 联结器里头称为smart 技术。其它联结器亦使用类似的技术。

第 3 章

MFC 六大关键技术之仿真

演化（evolution）永远在进行，
这个世界却不是每天都有革命（revolution）发生。
Application Framework 在软件界确实称得上具有革命精神。

仿真MFC？有必要吗？意义何在？如何仿真？

我已经在序言以及导读开宗明义说过了，这本书除了教导你使用MFC，另一个重要的功能是让你认识一个application framework 的内部运作。以MFC 为教学载具，我既可以让你领略application framework 的设计方式，更可以让你熟悉MFC 类别，将来运用时得心应手。呵，双效合一。

整个MFC 4.0 多达189 个类别，源代码达252 个实作档，58 个头文件，共10 MB 之多。

MFC 4.2 又多加了29 个类别。这么庞大的对象，当然不是每一个类别每一个数据结构都是我的仿真目标。我只挑选最神秘又最重要，与应用程序主干息息相关的题目，包括：

- MFC 程序的初始化过程
- RTTI（Runtime Type Information）执行时期型别信息
- Dynamic Creation 动态生成
- Persistence 永续留存
- Message Mapping 消息映射
- Message Routing 消息绕行

MFC 本身的设计在Application Framework 之中不见得最好，敌视者甚至认为它是个Minotaur（注）！但无论如何，这是当今软件霸主微软公司的产品，从探究application framework 设计的角度来说，实为一个重要参考；而如果从选择一套application framework作为软件开发工具的角度来说，单就就业市场的需求，我对MFC 的推荐再加10 分！

> 注：Minotaur 是希腊神话中的牛头人身怪物，居住在迷宫之中。进入迷宫的人如果走不出来，就会被一口吃掉

另一个问题是，为什么要仿真？第三篇第四篇各章节不是还要挖MFC 源代码来看吗？原因是MFC 太过庞大，我必须撇开枝节，把唯一重点突显出来，才容易收到教育效果。而且，仿真才能实证嘛！

如何仿真？我采用文字模式，也就是所谓的Console 程序，这样可以把程序结构的负荷降到最低。但是像消息映射和消息绕行怎么办？消息的流动是Windows 程序才有的特征啊！唔，看了你就知道。

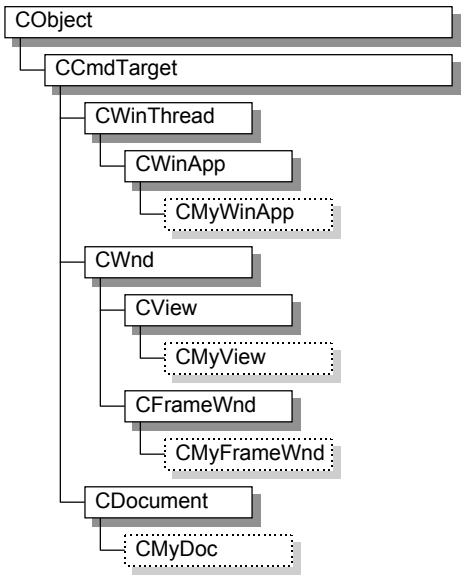
我的最高原则是：简化再简化，简化到不能再简化。

请注意，以下所有程序的类别阶层架构、类别名称、变量名称、结构名称、函数名称、函数行为，都以MFC 为仿真对象，具体而微。也可以说，我从数以万行计的MFC 原代码中，「偷」了一些出来，砍掉旁枝末节，只露出重点。

在文件的安排上，我把仿真MFC 的类别都集中在MFC.H 和MFC.CPP 中，把自己衍生的类别集中在MY.H 和MY.CPP 中。对于自定类别，我的命名方式是在父类别的名称前面加一个"*My*"，例如衍生自*CWinApp* 者，名为*CMyWinApp*，衍生自*CDocument* 者，名为*CMyDoc*。

MFC类别阶层

首先我以一个极简单的程序Frame1，把MFC 数个最重要类别的阶层关系仿真出来：



这个实例仿真MFC 的类别阶层。后续数节中，我会继续在这个类别阶层上开发新的能力。在这些名为Frame? 的各范例中，我以MFC 源代码为蓝本，尽量仿真MFC 的内部行为，并且使用完全相同的类别名称、函数名称、变量名称。这样的仿真对于我们在第三篇以及第四篇中深入探讨MFC 时将有莫大助益。相信我，这是真的。

Frame1　范例程序

MFC.H

```
#0001  #include <iostream.h>
#0002
#0003  class CObject
#0004  {
#0005  public:
```

```
#0006   CObject::CObject()  {  cout << "CObject Constructor \n";  }
#0007   CObject::~CObject() {  cout << "CObject Destructor \n";   }
#0008 };
#0009
#0010 class CCmdTarget : public CObject
#0011 {
#0012 public:
#0013   CCmdTarget::CCmdTarget()  {  cout << "CCmdTarget Constructor \n"; }
#0014   CCmdTarget::~CCmdTarget() {  cout << "CCmdTarget Destructor \n";  }
#0015 };
#0016
#0017 class CWinThread : public CCmdTarget
#0018 {
#0019 public:
#0020   CWinThread::CWinThread()  {  cout << "CWinThread Constructor \n"; }
#0021   CWinThread::~CWinThread() {  cout << "CWinThread Destructor \n";  }
#0022 };
#0023
#0024 class CWinApp : public CWinThread
#0025 {
#0026 public:
#0027   CWinApp* m_pCurrentWinApp;
#0028
#0029 public:
#0030   CWinApp::CWinApp()  {   m_pCurrentWinApp = this;
                              cout << "CWinApp Constructor \n";  }
#0031   CWinApp::~CWinApp() {  cout << "CWinApp Destructor \n";   }
#0032 };
#0033
#0034 class CDocument : public CCmdTarget
#0035 {
#0036 public:
#0037   CDocument::CDocument()   {  cout << "CDocument Constructor \n"; }
#0038   CDocument::~CDocument()  {  cout << "CDocument Destructor \n";  }
#0039 };
#0040
#0041
#0042 class CWnd : public CCmdTarget
#0043 {
#0044 public:
#0045   CWnd::CWnd()   {  cout << "CWnd Constructor \n"; }
#0046   CWnd::~CWnd()  {  cout << "CWnd Destructor \n";  }
#0047 };
#0048
#0049 class CFrameWnd : public CWnd
#0050 {
```

```
#0051  public:
#0052    CFrameWnd::CFrameWnd()   {  cout << "CFrameWnd Constructor \n"; }
#0053    CFrameWnd::~CFrameWnd()  {  cout << "CFrameWnd Destructor \n";  }
#0054  };
#0055
#0056  class CView : public CWnd
#0057  {
#0058  public:
#0059    CView::CView()   {  cout << "CView Constructor \n"; }
#0060    CView::~CView()  {  cout << "CView Destructor \n";  }
#0061  };
#0062
#0063
#0064  // global function
#0065
#0066  CWinApp* AfxGetApp();
```

MFC.CPP

```
#0001 #include "my.h" // 原本包含mfc.h 就好, 但为了CMyWinApp 的定义, 所以...
#0002
#0003  extern CMyWinApp theApp;
#0004
#0005  CWinApp* AfxGetApp()
#0006  {
#0007    return theApp.m_pCurrentWinApp;
#0008  }
```

MY.H

```
#0001  #include <iostream.h>
#0002  #include "mfc.h"
#0003
#0004  class CMyWinApp : public CWinApp
#0005  {
#0006  public:
#0007    CMyWinApp::CMyWinApp()   {  cout << "CMyWinApp Constructor \n"; }
#0008    CMyWinApp::~CMyWinApp()  {  cout << "CMyWinApp Destructor \n";  }
#0009  };
#0010
#0011  class CMyFrameWnd : public CFrameWnd
#0012  {
#0013  public:
#0014    CMyFrameWnd()   {  cout << "CMyFrameWnd Constructor \n";  }
#0015    ~CMyFrameWnd()  {  cout << "CMyFrameWnd Destructor \n";   }
```

```
#0016 };
```

MY.CPP

```
#0001 #include "my.h"
#0002
#0003 CMyWinApp theApp;  // global object
#0004
#0005 //--------------------------------------------------------------------
#0006 // main
#0007 //--------------------------------------------------------------------
#0008 void main()
#0009 {
#0010
#0011   CWinApp* pApp = AfxGetApp();
#0012
#0013 }
```

Frame1 的命令列编译联结动作是（环境变量必须先设定好，请参考第 4 章的「安装与设定」一节）：

```
cl my.cpp mfc.cpp  <Enter>
```

Frame1 的执行结果是：

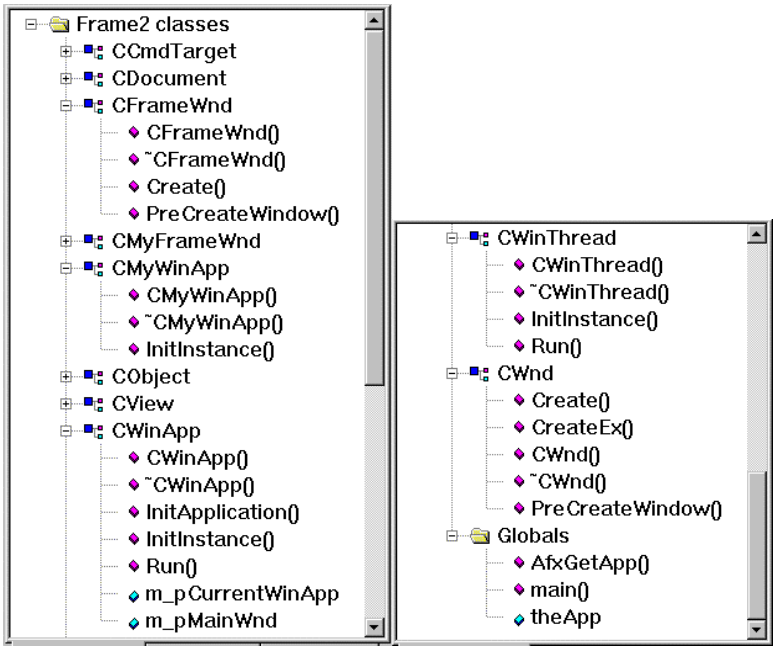
```
CObject Constructor
CCmdTarget Constructor
CWinThread Constructor
CWinApp Constructor
CMyWinApp Constructor

CMyWinApp Destructor
CWinApp Destructor
CWinThread Destructor
CCmdTarget Destructor
CObject Destructor
```

好，你看到了，Frame1 并没有*new* 任何对象，反倒是有一个全域对象*theApp* 存在。C++ 规定，全域对象的构造将比程序进入点（在DOS 环境为*main*，在Windows 环境为*WinMain*）更早。所以*theApp* 的构造式将更早于*main*。换句话说你所看到的执行结果中的那些构造式输出动作全都是在*main* 函数之前完成的。

main 函数调用全域函数*AfxGetApp* 以取得*theApp* 的对象指针。这完全是仿真MFC 程序的手法。

MFC 程序的初始化过程

MFC 程序也是个Windows 程序，它的内部一定也像第 1 章所述一样，有窗口注册动作，有窗口产生动作，有消息循环动作，也有窗口函数。此刻我并不打算做出Windows 程序，只是想交待给你一个程序流程，这个流程正是任何MFC 程序的初始化过程的简化。以下是Frame2 范例程序的类别阶层及其成员。对于那些「除了构造式与析构式之外没有其它成员」的类别，我就不在图中展开他们了：



（本图从Visual C++ 的「Class View 窗口」中获得）

就如我曾在第 1 章解释过的，*InitApplication* 和*InitInstance* 现在成了MFC 的*CWinApp* 的两个虚拟函数。前者负责「每一个程序只做一次」的动作，后者负责「每一个执行个

体都得做一次」的动作。通常，系统会（并且有能力）为你注册一些标准的窗口类别（当然也就准备好了一些标准的窗口函数），你（应用程序设计者）应该在你的*CMyWinApp* 中改写*InitInstance*，并在其中把窗口产生出来-- 这样你才有机会在标准的窗口类别中指定自己的窗口标题和菜单。下面就是我们新的*main* 函数：

```
// MY.CPP
CMyWinApp theApp;
void main()
{
  CWinApp* pApp = AfxGetApp();
  pApp->InitApplication();
  pApp->InitInstance();
  pApp->Run();
}
```

其中*pApp* 指向*theApp* 全域对象。在这里我们开始看到了虚拟函数的妙用（还不熟练者请快复习第 2 章）：

- `pApp->InitApplication()` 调用的是*CWinApp*::*InitApplication*,
- `pApp->InitInstance()` 调用的是*CMyWinApp*::*InitInstance*（因为*CMyWinApp* 改写它了），
- `pApp->Run()` 调用的是*CWinApp*::*Run*,

好，请注意以下*CMyWinApp*::*InitInstance* 的动作，以及它所引发的行为：

```
BOOL CMyWinApp::InitInstance()
{
    cout << "CMyWinApp::InitInstance \n";
    m_pMainWnd = new CMyFrameWnd; // 引发CMyFrameWnd::CMyFrameWnd 构造式
    return TRUE;
}
CMyFrameWnd::CMyFrameWnd()
{
    Create(); // Create 是虚拟函数，但CMyFrameWnd 未改写它，所以引发父类别的
              // CFrameWnd::Create
}
BOOL CFrameWnd::Create()
{
```

```
  cout << "CFrameWnd::Create \n";
  CreateEx(); // CreateEx 是虚拟函数，但CFrameWnd 未改写之，所以引发
              // CWnd::CreateEx
  return TRUE;
}

BOOL CWnd::CreateEx()
{
  cout << "CWnd::CreateEx \n";
 PreCreateWindow(); // 这是一个虚拟函数，CWnd 中有定义，CFrameWnd 也改写了
                    // 它。那么你说这里到底是调用CWnd::PreCreateWindow 还是
                    // CFrameWnd::PreCreateWindow 呢？
  return TRUE;
}
```

答案是CFrameWnd::PreCreateWindow。
這便是我在第 2 章的「Object slicing 与虚拟函数」一节所说提「虚拟函数的一个极重要的行为模式」。

```
BOOL CFrameWnd::PreCreateWindow()
{
  cout << "CFrameWnd::PreCreateWindow \n";
  return TRUE;
}
```

你看到了，这些函数什么正经事儿也没做，光只输出一个标识符串。我主要的目的是在让你先熟悉MFC 程序的执行流程。

Frame2 的命令列编译联结动作是（环境变量必须先设定好，请参考第 4 章的「安装与设定」一节）：

```
cl my.cpp mfc.cpp  <Enter>
```

以下就是Frame2 的执行结果：

```
CWinApp::InitApplication
CMyWinApp::InitInstance
CMyFrameWnd::CMyFrameWnd
CFrameWnd::Create
CWnd::CreateEx
CFrameWnd::PreCreateWindow
CWinApp::Run
CWinThread::Run
```

Frame2 范例程序

MFC.H

```
#0001  #define BOOL int
#0002  #define TRUE 1
#0003  #define FALSE 0
#0004
#0005  #include <iostream.h>
#0006
#0007  class CObject
#0008  {
#0009  public:
#0010    CObject::CObject()  {   }
#0011    CObject::~CObject() {   }
#0012  };
#0013
#0014  class CCmdTarget : public CObject
#0015  {
#0016  public:
#0017    CCmdTarget::CCmdTarget()  {   }
#0018    CCmdTarget::~CCmdTarget() {   }
#0019  };
#0020
#0021  class CWinThread : public CCmdTarget
#0022  {
#0023  public:
#0024    CWinThread::CWinThread()  {   }
#0025    CWinThread::~CWinThread() {   }
#0026
#0027    virtual BOOL InitInstance() {
#0028                                cout << "CWinThread::InitInstance \n";
#0029                                return TRUE;
#0030                                }
#0031    virtual int Run() {
#0032                       cout << "CWinThread::Run \n";
#0033                       return 1;
#0034                      }
#0035  };
#0036
#0037  class CWnd;
#0038
#0039  class CWinApp : public CWinThread
#0040  {
#0041  public:
```

```
#0042   CWinApp* m_pCurrentWinApp;
#0043   CWnd* m_pMainWnd;
#0044
#0045  public:
#0046   CWinApp::CWinApp()  { m_pCurrentWinApp = this; }
#0047   CWinApp::~CWinApp() {   }
#0048
#0049   virtual BOOL InitApplication() {
#0050                                cout << "CWinApp::InitApplication \n";
#0051                                return TRUE;
#0052                                   }
#0053   virtual BOOL InitInstance()   {
#0054                                    cout << "CWinApp::InitInstance \n";
#0055                                    return TRUE;
#0056                                   }
#0057   virtual int Run() {
#0058                     cout << "CWinApp::Run \n";
#0059                     return CWinThread::Run();
#0060                    }
#0061  };
#0062
#0063
#0064  class CDocument : public CCmdTarget
#0065  {
#0066  public:
#0067    CDocument::CDocument()   {   }
#0068    CDocument::~CDocument()  {   }
#0069  };
#0070
#0071
#0072  class CWnd : public CCmdTarget
#0073  {
#0074  public:
#0075    CWnd::CWnd()   {   }
#0076    CWnd::~CWnd()  {   }
#0077
#0078   virtual BOOL Create();
#0079   BOOL CreateEx();
#0080   virtual BOOL PreCreateWindow();
#0081  };
#0082
#0083  class CFrameWnd : public CWnd
#0084  {
#0085  public:
#0086    CFrameWnd::CFrameWnd()   {   }
#0087    CFrameWnd::~CFrameWnd()  {   }
```

```
#0088   BOOL Create();
#0089   virtual BOOL PreCreateWindow();
#0090 };
#0091
#0092 class CView : public CWnd
#0093 {
#0094 public:
#0095   CView::CView()   {   }
#0096   CView::~CView()  {   }
#0097 };
#0098
#0099
#0100 // global function
#0101 CWinApp* AfxGetApp();
```

MFC.CPP

```
#0001 #include "my.h"  // 原该包含mfc.h 就好, 但为了CMyWinApp 的定义, 所以...
#0002
#0003 extern CMyWinApp theApp;  // external global object
#0004
#0005 BOOL CWnd::Create()
#0006 {
#0007   cout << "CWnd::Create \n";
#0008   return TRUE;
#0009 }
#0010
#0011 BOOL CWnd::CreateEx()
#0012 {
#0013   cout << "CWnd::CreateEx \n";
#0014   PreCreateWindow();
#0015   return TRUE;
#0016 }
#0017
#0018 BOOL CWnd::PreCreateWindow()
#0019 {
#0020   cout << "CWnd::PreCreateWindow \n";
#0021   return TRUE;
#0022 }
#0023
#0024 BOOL CFrameWnd::Create()
#0025 {
#0026   cout << "CFrameWnd::Create \n";
#0027   CreateEx();
#0028   return TRUE;
#0029 }
```

```
#0030
#0031  BOOL CFrameWnd::PreCreateWindow()
#0032  {
#0033    cout << "CFrameWnd::PreCreateWindow \n";
#0034    return TRUE;
#0035  }
#0036
#0037
#0038  CWinApp* AfxGetApp()
#0039  {
#0040    return theApp.m_pCurrentWinApp;
#0041  }
```

MY.H

```
#0001  #include <iostream.h>
#0002  #include "mfc.h"
#0003
#0004  class CMyWinApp : public CWinApp
#0005  {
#0006  public:
#0007    CMyWinApp::CMyWinApp()   {   }
#0008    CMyWinApp::~CMyWinApp()  {   }
#0009
#0010    virtual BOOL InitInstance();
#0011  };
#0012
#0013  class CMyFrameWnd : public CFrameWnd
#0014  {
#0015  public:
#0016    CMyFrameWnd();
#0017    ~CMyFrameWnd()  {   }
#0018  };
```

MY.CPP

```
#0001  #include "my.h"
#0002
#0003  CMyWinApp theApp;  // global object
#0004
#0005  BOOL CMyWinApp::InitInstance()
#0006  {
#0007     cout << "CMyWinApp::InitInstance \n";
#0008     m_pMainWnd = new CMyFrameWnd;
#0009     return TRUE;
```

```
#0010  }
#0011
#0012  CMyFrameWnd::CMyFrameWnd()
#0013  {
#0014      cout << "CMyFrameWnd::CMyFrameWnd \n";
#0015      Create();
#0016  }
#0017
#0018  //--------------------------------------------------------------------
#0019  // main
#0020  //--------------------------------------------------------------------
#0021  void main()
#0022  {
#0023
#0024      CWinApp* pApp = AfxGetApp();
#0025
#0026      pApp->InitApplication();
#0027      pApp->InitInstance();
#0028      pApp->Run();
#0029  }
```

RTTI（执行时期型别辨识）

你已经在第 2 章看到，Visual C++ 4.0 支持RTTI，重点不外乎是：

1. 编译时需选用/GR 选项（/GR 的意思是enable C++ RTTI）
2. 包含typeinfo.h
3. 使用新的*typeid* 运算子。

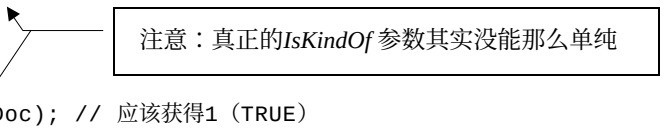
> RTTI 亦有称为Runtime Type Identification 者。

MFC 早在编译器支持RTTI 之前，就有了这项能力。我们现在要以相同的手法，在DOS 程序中仿真出来。我希望我的类别库具备*IsKindOf* 的能力，能在执行时期侦测某个对象是否「属于某种类别」，并传回*TRUE* 或*FALSE*。以前一章的Shape 为例，我希望：

```
CSquare* pSquare = new CSquare;
cout << pSquare->IsKindOf(CSquare); // 应该获得1（TRUE）
cout << pSquare->IsKindOf(CRect); // 应该获得1（TRUE）
cout << pSquare->IsKindOf(CShape); // 应该获得1（TRUE）
```

```
cout << pSquare->IsKindOf(CCircle); // 应该获得0（FALSE）
```

以MFC 的类别阶层来说，我希望：



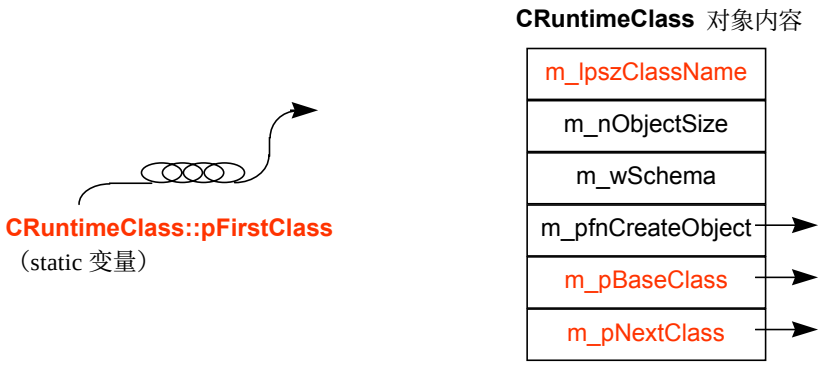

```
CMyDoc* pMyDoc = new CMyDoc;
cout << pMyDoc->IsKindOf(CMyDoc); // 应该获得1（TRUE）
cout << pMyDoc->IsKindOf(CDocument); // 应该获得1（TRUE）
cout << pMyDoc->IsKindOf(CCmdTarget); // 应该获得1（TRUE）
cout << pMyDoc->IsKindOf(CWnd); // 应该获得0（FALSE）
```

类别型录网与CRuntimeClass

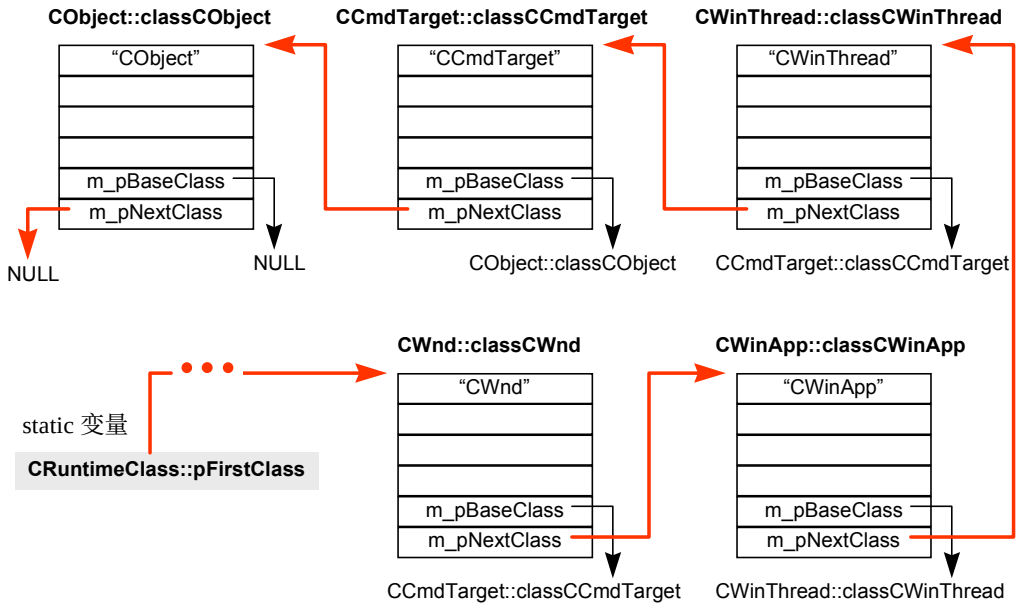
怎么设计RTTI 呢？让我们想想，当你手上握有一种色泽，想知道它的RGB 成份比，不查色表行吗？当你持有一种产品，想知道它的型号，不查型录行吗？要达到RTTI 的能力，我们（类别库的设计者）一定要在类别构造起来的时候，记录必要的信息，以建立型录。型录中的类别信息，最好以串行（linked list）方式串接起来，将来方便一一比对。

我们这份「类别型录」的串行元素将以*CRuntimeClass* 描述之，那是一个结构，内中至少需有类别名称、串行的Next 指针，以及串行的First 指针。由于First 指针属于全域变量，一份就好，所以它应该以static 修饰之。除此之外你所看到的其它*CRuntimeClass* 成员都是为了其它目的而准备，陆陆续续我会介绍出来。

```
// in MFC.H
struct CRuntimeClass
{
// Attributes
   LPCSTR m_lpszClassName;
   int m_nObjectSize;
   UINT m_wSchema; // schema number of the loaded class
   CObject* (PASCAL* m_pfnCreateObject)(); // NULL => abstract class
   CRuntimeClass* m_pBaseClass;

   // CRuntimeClass objects linked together in simple list
   static CRuntimeClass* pFirstClass; // start of class list
   CRuntimeClass* m_pNextClass; // linked list of registered classes
};
```



我希望，每一个类别都能拥有这样一个*CRuntimeClass* 成员变量，并且最好有一定的命名规则（例如在类别名称之前冠以"*class*" 作为它的名称），然后，经由某种手段将整个类别库构造好之后，「类别型录」能呈现类似这样的风貌：



DECLARE_DYNAMIC / IMPLEMENT_DYNAMIC宏

为了神不知鬼不觉把*CRuntimeClass* 对象塞到类别之中，并声明一个可以抓到该对象地址的函数，我们定义*DECLARE_DYNAMIC* 宏如下：

```
#define DECLARE_DYNAMIC(class_name) \
public: \
       static CRuntimeClass class##class_name; \
       virtual CRuntimeClass* GetRuntimeClass() const;
```

出现在宏定义之中的##，用来告诉编译器，把两个字符串系在一起。如果你这么使用此宏：

```
DECLARE_DYNAMIC(CView)
```

编译器前置处理器为你做出的码是：

```
public:
       static CRuntimeClass classCView;
       virtual CRuntimeClass* GetRuntimeClass() const;
```

这下子，只要在声明类别时放入*DECLARE_DYNAMIC* 宏即万事OK 喽。

不，还没有OK，类别型录（也就是各个*CRuntimeClass* 对象）的内容指定以及串接工作最好也能够神不知鬼不觉，我们于是再定义*IMPLEMENT_DYNAMIC* 宏：

```
#define IMPLEMENT_DYNAMIC(class_name, base_class_name) \
       _IMPLEMENT_RUNTIMECLASS(class_name, base_class_name, 0xFFFF, NULL)
```

其中的*_IMPLEMENT_RUNTIMECLASS* 又是一个宏。这样区分是为了此一宏在「动态生成」（下一节主题）时还会用到。

```
#define _IMPLEMENT_RUNTIMECLASS(class_name, base_class_name,wSchema,pfnNew) \
      static char _lpsz##class_name[] = #class_name; \
      CRuntimeClass class_name::class##class_name = { \
             _lpsz##class_name, sizeof(class_name), wSchema, pfnNew, \
                    RUNTIME_CLASS(base_class_name), NULL }; \
      static AFX_CLASSINIT _init_##class_name(&class_name::class##class_name); \
      CRuntimeClass* class_name::GetRuntimeClass() const \
             { return &class_name::class##class_name; } \
```

其中又有*RUNTIME_CLASS* 宏，定义如下：

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

看起来整个*IMPLEMENT_DYNAMIC* 内容好象只是指定初值，不然，其曼妙处在于它所使用的一个struct AFX_CLASSINIT，定义如下：

```
struct AFX_CLASSINIT
        { AFX_CLASSINIT(CRuntimeClass* pNewClass); };
```

这表示它有一个构造式（别惊讶，C++ 的struct 与class 都有构造式），定义如下：

```
AFX_CLASSINIT::AFX_CLASSINIT(CRuntimeClass* pNewClass)
{
        pNewClass->m_pNextClass = CRuntimeClass::pFirstClass;
        CRuntimeClass::pFirstClass = pNewClass;
}
```

很明显，此构造式负责linked list 的串接工作。

整组宏看起来有点吓人，其实也没有什么，文字代换而已。现在看看这个实例：

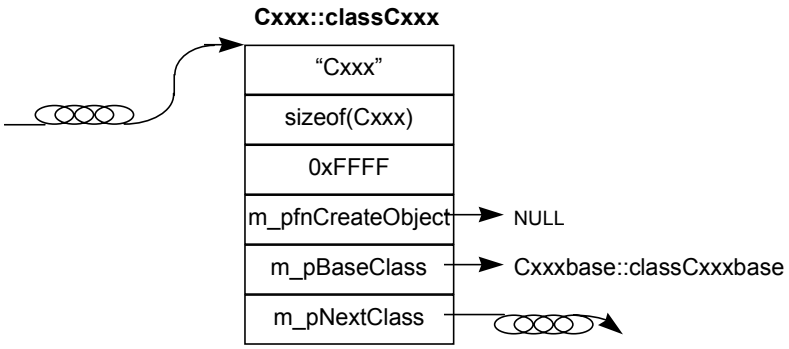
```
// in header file
class CView : public CWnd
{
        DECLARE_DYNAMIC(CView)
        ...
};

// in implementation file
IMPLEMENT_DYNAMIC(CView, CWnd)
```

上述的码展开来成为：

```
// in header file
class CView : public CWnd
{
public:
        static CRuntimeClass classCView; \
        virtual CRuntimeClass* GetRuntimeClass() const;
        ...
};
```

```
// in implementation file
static char _lpszCView[] = "CView";
CRuntimeClass CView::classCView = {
       _lpszCView, sizeof(CView), 0xFFFF, NULL,
              &CWnd::classCWnd, NULL };
static AFX_CLASSINIT _init_CView(&CView::classCView);
CRuntimeClass* CView::GetRuntimeClass() const
       { return &CView::classCView; }
```

于是乎， 程序中只需要简简单单的两个宏*DECLARE_DYNAMIC(Cxxx)* 和 *IMPLEMENT_DYNAMIC(Cxxx, Cxxxbase)*，就完成了构造资料并加入串行的工作：



可是你知道，串行的头，总是需要特别费心处理，不能够套用一般的串行行为模式。我们的类别根源*CObject*， 不能套用现成的宏*DECLARE_DYNAMIC* 和 *IMPLEMENT_DYNAMIC*，必须特别设计如下：

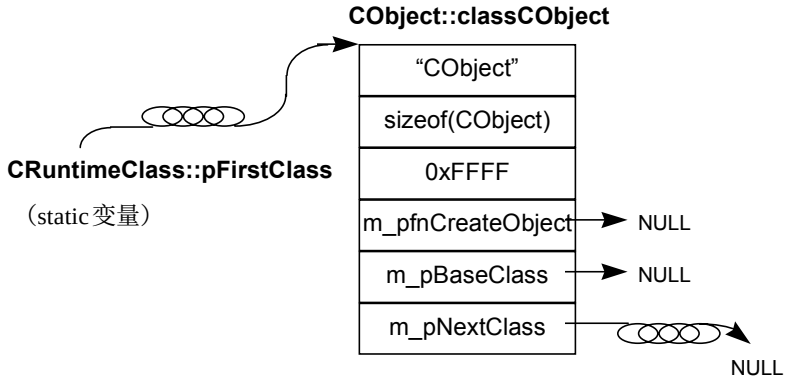
```
// in header file
class CObject
{
public:
  virtual CRuntimeClass* GetRuntimeClass() const;
  ...
public:
  static CRuntimeClass classCObject;
};
```

```
// in implementation file
static char szCObject[] = "CObject";
struct CRuntimeClass CObject::classCObject =
       { szCObject, sizeof(CObject), 0xffff, NULL, NULL };
static AFX_CLASSINIT _init_CObject(&CObject::classCObject);

CRuntimeClass* CObject::GetRuntimeClass() const
{
       return &CObject::classCObject;
}
```

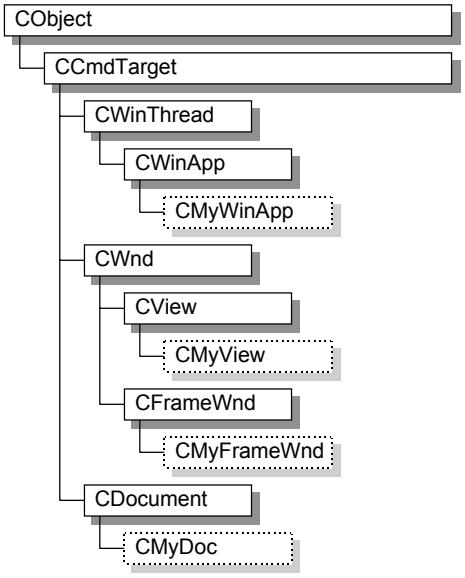
并且，*CRuntimeClass* 中的*static* 成员变量应该要初始化（如果你忘记了，赶快复习第2章的「静态成员（变量与函数）」一节）：

```
// in implementation file
CRuntimeClass* CRuntimeClass::pFirstClass = NULL;
```

终于，整个「类别型录」串行的头部就这样形成了：



范例程序Frame3 在.h 档中有这些类别声明：




```
class CObject
{
...
};
class CCmdTarget : public CObject
{
     DECLARE_DYNAMIC(CCmdTarget)
...
};
class CWinThread : public CCmdTarget
{
     DECLARE_DYNAMIC(CWinThread)
...
};
class CWinApp : public CWinThread
{
     DECLARE_DYNAMIC(CWinApp)
...
};
class CDocument : public CCmdTarget
{
     DECLARE_DYNAMIC(CDocument)
```

```
...
};
class CWnd : public CCmdTarget
{
     DECLARE_DYNAMIC(CWnd)  // 其实在MFC 中是DECLARE_DYNCREATE(), 见下节。
...
};
class CFrameWnd : public CWnd
{
    DECLARE_DYNAMIC(CFrameWnd) // 其实在MFC 中是DECLARE_DYNCREATE(), 见下节。
...
};
class CView : public CWnd
{
     DECLARE_DYNAMIC(CView)
...
};
class CMyWinApp : public CWinApp
{
...
};
class CMyFrameWnd : public CFrameWnd
{
  ... // 其实在MFC 应用程序中这里也有DECLARE_DYNCREATE(), 见下节。
};
class CMyDoc : public CDocument
{
  ... // 其实在MFC 应用程序中这里也有DECLARE_DYNCREATE(), 见下节。
};
class CMyView : public CView
{
  ... // 其实在MFC 应用程序中这里也有DECLARE_DYNCREATE(), 见下节。
};
```

范例程序Frame3 在.cpp 档中有这些动作：

```
IMPLEMENT_DYNAMIC(CCmdTarget, CObject)
IMPLEMENT_DYNAMIC(CWinThread, CCmdTarget)
IMPLEMENT_DYNAMIC(CWinApp, CWinThread)
IMPLEMENT_DYNAMIC(CWnd, CCmdTarget) // 其实在MFC 中它是IMPLEMENT_DYNCREATE(), 见下节。
IMPLEMENT_DYNAMIC(CFrameWnd, CWnd) // 其实在MFC 中它是IMPLEMENT_DYNCREATE(), 见下节。
IMPLEMENT_DYNAMIC(CDocument, CCmdTarget)
IMPLEMENT_DYNAMIC(CView, CWnd)
```

于是组织出图3-1 这样一个大网。

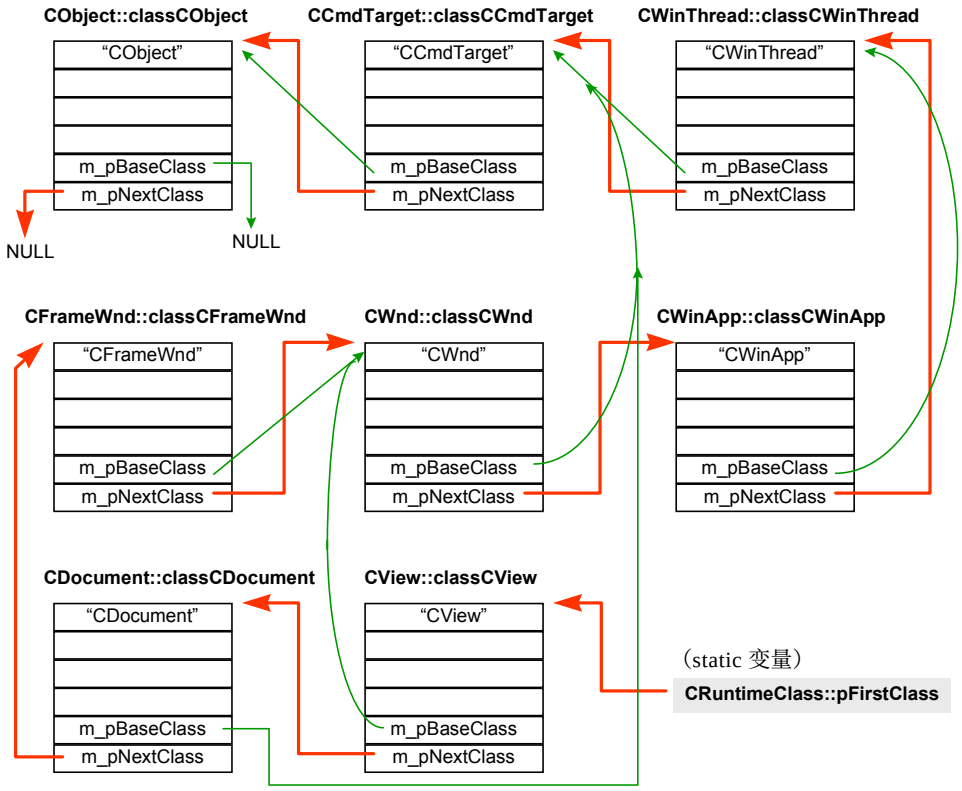


图3-1 CRuntimeClass 对象构成的类别型录网。本图只列出与RTTI 有关系的成员。

为了实证整个类别型录网的存在，我在*main* 函数中调用*PrintAllClasses*，把串行中的每一个元素的类别名称、对象大小、以及schema no. 印出来：

```
void PrintAllClasses()
{
    CRuntimeClass* pClass;

    // just walk through the simple list of registered classes
    for (pClass = CRuntimeClass::pFirstClass; pClass != NULL;
           pClass = pClass->m_pNextClass)
    {
        cout << pClass->m_lpszClassName << "\n";
```

```
        cout << pClass->m_nObjectSize << "\n";
        cout << pClass->m_wSchema << "\n";
    }
}
```

Frame3 的命令列编译联结动作是（环境变量必须先设定好，请参考第 4 章的「安装与设定」一节）：

```
cl my.cpp mfc.cpp  <Enter>
```

Frame3 的执行结果如下：

```
CView
4
65535
CDocument
4
65535
CFrameWnd
4
65535
CWnd
4
65535
CWinApp
12
65535
CWinThread
4
65535
CCmdTarget
4
65535
CObject
4
65535
```

Frame3 范例程序

MFC.H

```
#0001  #define BOOL int
#0002  #define TRUE 1
#0003  #define FALSE 0
```

```
#0004  #define LPCSTR  LPSTR
#0005  typedef char*   LPSTR;
#0006  #define UINT int
#0007  #define PASCAL _stdcall
#0008
#0009  #include <iostream.h>
#0010
#0011  class CObject;
#0012
#0013  struct CRuntimeClass
#0014  {
#0015  // Attributes
#0016         LPCSTR m_lpszClassName;
#0017         int m_nObjectSize;
#0018         UINT m_wSchema; // schema number of the loaded class
#0019         CObject* (PASCAL* m_pfnCreateObject)(); // NULL => abstract class
#0020         CRuntimeClass* m_pBaseClass;
#0021
#0022         // CRuntimeClass objects linked together in simple list
#0023         static CRuntimeClass* pFirstClass; // start of class list
#0024         CRuntimeClass* m_pNextClass; // linked list of registered classes
#0025  };
#0026
#0027  struct AFX_CLASSINIT
#0028         { AFX_CLASSINIT(CRuntimeClass* pNewClass); };
#0029
#0030  #define RUNTIME_CLASS(class_name) \
#0031         (&class_name::class##class_name)
#0032
#0033  #define DECLARE_DYNAMIC(class_name) \
#0034  public: \
#0035         static CRuntimeClass class##class_name; \
#0036         virtual CRuntimeClass* GetRuntimeClass() const;
#0037
#0038  #define _IMPLEMENT_RUNTIMECLASS(class_name, base_class_name, wSchema, pfnNew) \
#0039         static char _lpsz##class_name[] = #class_name; \
#0040         CRuntimeClass class_name::class##class_name = { \
#0041                _lpsz##class_name, sizeof(class_name), wSchema, pfnNew, \
#0042                       RUNTIME_CLASS(base_class_name), NULL }; \
#0043       static AFX_CLASSINIT _init_##class_name(&class_name::class##class_name); \
#0044         CRuntimeClass* class_name::GetRuntimeClass() const \
#0045                { return &class_name::class##class_name; } \
#0046
#0047  #define IMPLEMENT_DYNAMIC(class_name, base_class_name) \
#0048         _IMPLEMENT_RUNTIMECLASS(class_name, base_class_name, 0xFFFF, NULL)
#0049
```

```
#0050  class CObject
#0051  {
#0052  public:
#0053    CObject::CObject()  {
#0054                        }
#0055    CObject::~CObject() {
#0056                        }
#0057
#0058    virtual CRuntimeClass* GetRuntimeClass() const;
#0059
#0060  public:
#0061    static CRuntimeClass classCObject;
#0062  };
#0063
#0064  class CCmdTarget : public CObject
#0065  {
#0066         DECLARE_DYNAMIC(CCmdTarget)
#0067  public:
#0068    CCmdTarget::CCmdTarget()  {
#0069                              }
#0070    CCmdTarget::~CCmdTarget() {
#0071                              }
#0072  };
#0073
#0074  class CWinThread : public CCmdTarget
#0075  {
#0076         DECLARE_DYNAMIC(CWinThread)
#0077  public:
#0078    CWinThread::CWinThread()  {
#0079                              }
#0080    CWinThread::~CWinThread() {
#0081                              }
#0082
#0083    virtual BOOL InitInstance() {
#0084                                 return TRUE;
#0085                               }
#0086    virtual int Run() {
#0087                       return 1;
#0088                     }
#0089  };
#0090
#0091  class CWnd;
#0092
#0093  class CWinApp : public CWinThread
#0094  {
#0095         DECLARE_DYNAMIC(CWinApp)
```

```
#0096  public:
#0097    CWinApp* m_pCurrentWinApp;
#0098    CWnd* m_pMainWnd;
#0099
#0100  public:
#0101    CWinApp::CWinApp()  {
#0102                          m_pCurrentWinApp = this;
#0103                         }
#0104    CWinApp::~CWinApp() {
#0105                         }
#0106
#0107    virtual BOOL InitApplication() {
#0108                                    return TRUE;
#0109                                    }
#0110    virtual BOOL InitInstance()    {
#0111                                    return TRUE;
#0112                                   }
#0113    virtual int Run() {
#0114                       return CWinThread::Run();
#0115                      }
#0116  };
#0117
#0118  class CDocument : public CCmdTarget
#0119  {
#0120         DECLARE_DYNAMIC(CDocument)
#0121  public:
#0122    CDocument::CDocument()   {
#0123                              }
#0124    CDocument::~CDocument()  {
#0125                              }
#0126  };
#0127
#0128  class CWnd : public CCmdTarget
#0129  {
#0130         DECLARE_DYNAMIC(CWnd)
#0131  public:
#0132    CWnd::CWnd()   {
#0133                   }
#0134    CWnd::~CWnd()  {
#0135                   }
#0136
#0137    virtual BOOL Create();
#0138    BOOL CreateEx();
#0139    virtual BOOL PreCreateWindow();
#0140  };
#0141
```

```
#0142  class CFrameWnd : public CWnd
#0143  {
#0144         DECLARE_DYNAMIC(CFrameWnd)
#0145  public:
#0146    CFrameWnd::CFrameWnd()   {
#0147                             }
#0148    CFrameWnd::~CFrameWnd()  {
#0149                             }
#0150    BOOL Create();
#0151    virtual BOOL PreCreateWindow();
#0152  };
#0153
#0154  class CView : public CWnd
#0155  {
#0156         DECLARE_DYNAMIC(CView)
#0157  public:
#0158    CView::CView()   {
#0159                     }
#0160    CView::~CView()  {
#0161                     }
#0162  };
#0163
#0164
#0165  // global function
#0166  CWinApp* AfxGetApp();
```

MFC.CPP

```
#0001  #include "my.h"  //原该包含mfc.h 就好，但为了CMyWinApp 的定义，所以...
#0002
#0003  extern CMyWinApp theApp;
#0004
#0005  static char szCObject[] = "CObject";
#0006  struct CRuntimeClass CObject::classCObject =
#0007         { szCObject, sizeof(CObject), 0xffff, NULL, NULL };
#0008  static AFX_CLASSINIT _init_CObject(&CObject::classCObject);
#0009
#0010  CRuntimeClass* CRuntimeClass::pFirstClass = NULL;
#0011
#0012  AFX_CLASSINIT::AFX_CLASSINIT(CRuntimeClass* pNewClass)
#0013  {
#0014         pNewClass->m_pNextClass = CRuntimeClass::pFirstClass;
#0015         CRuntimeClass::pFirstClass = pNewClass;
#0016  }
#0017
#0018  CRuntimeClass* CObject::GetRuntimeClass() const
```

```
#0019  {
#0020         return &CObject::classCObject;
#0021  }
#0022
#0023  BOOL CWnd::Create()
#0024  {
#0025    return TRUE;
#0026  }
#0027
#0028  BOOL CWnd::CreateEx()
#0029  {
#0030    PreCreateWindow();
#0031    return TRUE;
#0032  }
#0033
#0034  BOOL CWnd::PreCreateWindow()
#0035  {
#0036    return TRUE;
#0037  }
#0038
#0039  BOOL CFrameWnd::Create()
#0040  {
#0041    CreateEx();
#0042    return TRUE;
#0043  }
#0044
#0045  BOOL CFrameWnd::PreCreateWindow()
#0046  {
#0047    return TRUE;
#0048  }
#0049
#0050  IMPLEMENT_DYNAMIC(CCmdTarget, CObject)
#0051  IMPLEMENT_DYNAMIC(CWinThread, CCmdTarget)
#0052  IMPLEMENT_DYNAMIC(CWinApp, CWinThread)
#0053  IMPLEMENT_DYNAMIC(CWnd, CCmdTarget)
#0054  IMPLEMENT_DYNAMIC(CFrameWnd, CWnd)
#0055  IMPLEMENT_DYNAMIC(CDocument, CCmdTarget)
#0056  IMPLEMENT_DYNAMIC(CView, CWnd)
#0057
#0058  // global function
#0059  CWinApp* AfxGetApp()
#0060  {
#0061    return theApp.m_pCurrentWinApp;
#0062  }
```

MY.H

```
#0001  #include <iostream.h>
#0002  #include "mfc.h"
#0003
#0004  class CMyWinApp : public CWinApp
#0005  {
#0006  public:
#0007    CMyWinApp::CMyWinApp()   {
#0008                             }
#0009    CMyWinApp::~CMyWinApp()  {
#0010                             }
#0011
#0012    virtual BOOL InitInstance();
#0013  };
#0014
#0015  class CMyFrameWnd : public CFrameWnd
#0016  {
#0017  public:
#0018    CMyFrameWnd();
#0019    ~CMyFrameWnd()  {
#0020                    }
#0021  };
#0022
#0023
#0024  class CMyDoc : public CDocument
#0025  {
#0026  public:
#0027    CMyDoc::CMyDoc()  {
#0028                      }
#0029    CMyDoc::~CMyDoc() {
#0030                      }
#0031  };
#0032
#0033  class CMyView : public CView
#0034  {
#0035  public:
#0036    CMyView::CMyView()   {
#0037                         }
#0038    CMyView::~CMyView()  {
#0039                         }
#0040  };
#0041
#0042  // global function
#0043  void PrintAllClasses();
```

MY.CPP

```
#0001  #include "my.h"
#0002
#0003  CMyWinApp theApp;
#0004
#0005  BOOL CMyWinApp::InitInstance()
#0006  {
#0007      m_pMainWnd = new CMyFrameWnd;
#0008      return TRUE;
#0009  }
#0010
#0011  CMyFrameWnd::CMyFrameWnd()
#0012  {
#0013      Create();
#0014  }
#0015
#0016  void PrintAllClasses()
#0017  {
#0018      CRuntimeClass* pClass;
#0019
#0020      // just walk through the simple list of registered classes
#0021      for (pClass = CRuntimeClass::pFirstClass; pClass != NULL;
#0022             pClass = pClass->m_pNextClass)
#0023      {
#0024          cout << pClass->m_lpszClassName << "\n";
#0025          cout << pClass->m_nObjectSize << "\n";
#0026          cout << pClass->m_wSchema << "\n";
#0027      }
#0028  }
#0029  //--------------------------------------------------------------------
#0030  // main
#0031  //--------------------------------------------------------------------
#0032  void main()
#0033  {
#0034    CWinApp* pApp = AfxGetApp();
#0035
#0036    pApp->InitApplication();
#0037    pApp->InitInstance();
#0038    pApp->Run();
#0039
#0040    PrintAllClasses();
#0041  }
```

IsKindOf　（型别辨识）

有了图3-1 这张「类别型录」网，要实现*IsKindOf* 功能，再轻松不过了：

1. 为*CObject* 加上一个*IsKindOf* 函数，于是此函数将被所有类别继承。它将把参数所指定的某个*CRuntimeClass* 对象拿来与类别型录中的元素一一比对。比对成功（在型录中有发现），就传回*TRUE*，否则传回*FALSE*：

```
// in header file
class CObject
{
public:
  ...
  BOOL IsKindOf(const CRuntimeClass* pClass) const;
};

// in implementation file
BOOL CObject::IsKindOf(const CRuntimeClass* pClass) const
{
       CRuntimeClass* pClassThis = GetRuntimeClass();
       while (pClassThis != NULL)
       {
              if (pClassThis == pClass)
                     return TRUE;
              pClassThis = pClassThis->m_pBaseClass;
       }
       return FALSE;      // walked to the top, no match
}
```

注意，*while* 循环中所追踪的是「同宗」路线，也就是凭借着*m_pBaseClass* 而非*m_pNextClass*。假设我们的调用是：

```
CView* pView = new CView;
pView->IsKindOf(RUNTIME_CLASS(CWinApp));
```

IsKindOf 的参数其实就是&*CWinApp*::*classCWinApp*。函数内利用*GetRuntimeClass* 先取得&*CView*::*classCView*，然后循线而上（从图3-1 来看，所谓循线分别是指*CView*、*CWnd*、*CCmdTarget*、*CObject*），每获得一个*CRuntimeClass* 对象指针，就拿来和*CView*::*classCView* 的指针比对。靠这个土方法，完成了*IsKindOf* 能力。

2. *IsKindOf* 的使用方式如下：

```
CMyDoc* pMyDoc = new CMyDoc;
CMyView* pMyView = new CMyView;

cout << pMyDoc->IsKindOf(RUNTIME_CLASS(CMyDoc));      // 應該獲得 TRUE
cout << pMyDoc->IsKindOf(RUNTIME_CLASS(CDocument));   // 應該獲得 TRUE
cout << pMyDoc->IsKindOf(RUNTIME_CLASS(CCmdTarget));  // 應該獲得 TRUE
cout << pMyDoc->IsKindOf(RUNTIME_CLASS(CObject));     // 應該獲得 TRUE
cout << pMyDoc->IsKindOf(RUNTIME_CLASS(CWinApp));     // 應該獲得 FALSE
cout << pMyDoc->IsKindOf(RUNTIME_CLASS(CView));       // 應該獲得 FALSE

cout << pMyView->IsKindOf(RUNTIME_CLASS(CView));      // 應該獲得 TRUE
cout << pMyView->IsKindOf(RUNTIME_CLASS(CObject));    // 應該獲得 TRUE
cout << pMyView->IsKindOf(RUNTIME_CLASS(CWnd));       // 應該獲得 TRUE
cout << pMyView->IsKindOf(RUNTIME_CLASS(CFrameWnd)); // 应该获得FALSE
```

IsKindOf 的完整范例放在Frame4 中。

Frame4　范例程序

Frame4 与Frame3 大同小异，唯一不同的就是前面所说的，在*CObject* 中加上*IsKindOf* 函数的声明与定义，并将私有类别（non-MFC 类别）也挂到「类别型录网」中：

```
// in header file
class CMyFrameWnd : public CFrameWnd
{
DECLARE_DYNAMIC(CMyFrameWnd) // 在MFC 程序中这里其实是DECLARE_DYNCREATE()
... // 稍后我便会仿真DECLARE_DYNCREATE() 给你看
};
class CMyDoc : public CDocument
{
DECLARE_DYNAMIC(CMyDoc) // 在MFC 程序中这里其实是DECLARE_DYNCREATE()
... // 稍后我便会仿真DECLARE_DYNCREATE() 给你看
};
class CMyView : public CView
{
DECLARE_DYNAMIC(CMyView) // 在MFC 程序中这里其实是DECLARE_DYNCREATE()
... // 稍后我便会仿真DECLARE_DYNCREATE() 给你看
};
```

```
// in implementation file
...
 IMPLEMENT_DYNAMIC(CMyFrameWnd, CFrameWnd) //在MFC 程序中这里其实是IMPLEMENT_DYNCREATE()
                                           // 稍后我便会仿真IMPLEMENT_DYNCREATE() 给你看
...
IMPLEMENT_DYNAMIC(CMyDoc, CDocument) //在MFC 程序中这里其实是IMPLEMENT_DYNCREATE()
                                     // 稍后我便会仿真IMPLEMENT_DYNCREATE() 给你看
...
IMPLEMENT_DYNAMIC(CMyView, CView)    // 在MFC 程序中这里其实是IMPLEMENT_DYNCREATE()
                                     // 稍后我便会仿真IMPLEMENT_DYNCREATE() 给你看
```

我不在此列出Frame4 的源代码，你可以在书附光盘片中找到完整的文件。Frame4 的命令列编译联结动作是（环境变量必须先设定好，请参考第 4 章的「安装与设定」一节）：

```
cl my.cpp mfc.cpp <Enter>
```

以下即是Frame4 的执行结果：

```
pMyDoc->IsKindOf(RUNTIME_CLASS(CMyDoc))        1
pMyDoc->IsKindOf(RUNTIME_CLASS(CDocument))     1
pMyDoc->IsKindOf(RUNTIME_CLASS(CCmdTarget))    1
pMyDoc->IsKindOf(RUNTIME_CLASS(CObject))       1
pMyDoc->IsKindOf(RUNTIME_CLASS(CWinApp))       0
pMyDoc->IsKindOf(RUNTIME_CLASS(CView))         0

pMyView->IsKindOf(RUNTIME_CLASS(CView))        1
pMyView->IsKindOf(RUNTIME_CLASS(CObject))      1
pMyView->IsKindOf(RUNTIME_CLASS(CWnd))         1
pMyView->IsKindOf(RUNTIME_CLASS(CFrameWnd))    0

pMyWnd->IsKindOf(RUNTIME_CLASS(CFrameWnd))     1
pMyWnd->IsKindOf(RUNTIME_CLASS(CWnd))          1
pMyWnd->IsKindOf(RUNTIME_CLASS(CObject))       1
pMyWnd->IsKindOf(RUNTIME_CLASS(CDocument))     0
```

Dynamic Creation (动态生成）

基础有了，做什么都好。同样地，有了上述的「类别型录网」，各种应用纷至沓来。其中一个应用就是解决棘手的动态生成问题。

我已经在第二章描述过动态生成的困难点：你没有办法在程序执行期间，根据动态获得的一个类别名称（通常来自读档，但我将以屏幕输入为例），要求程序产生一个对象。上述的「类别型录网」虽然透露出解决此一问题的些微曙光，但是技术上还得加把劲儿。

如果我能够把类别的大小记录在类别型录中，把构造函数（注意，这里并非指C++ 构造式，而是指即将出现的*CRuntimeClass::CreateObject*）也记录在类别型录中，当程序在执行时期获得一个类别名称，它就可以在「类别型录网」中找出对应的元素，然后调用其构造函数（这里并非指C++ 构造式），产生出对象。

好主意！

类别型录网的元素型式*CRuntimeClass* 于是有了变化：

```
// in MFC.H
struct CRuntimeClass
{
// Attributes
       LPCSTR m_lpszClassName;
       int m_nObjectSize;
       UINT m_wSchema; // schema number of the loaded class
       CObject* (PASCAL* m_pfnCreateObject)(); // NULL => abstract class
       CRuntimeClass* m_pBaseClass;

       CObject* CreateObject();
       static CRuntimeClass* PASCAL Load();

       // CRuntimeClass objects linked together in simple list
       static CRuntimeClass* pFirstClass; // start of class list
       CRuntimeClass* m_pNextClass;      // linked list of registered classes
};
```

DECLARE_DYNCREATE/IMPLEMENT_DYNCREATE宏

为了因应*CRuntimeClass* 中新增的成员变量， 我们再添两个宏，

DECLARE_DYNCREATE 和*IMPLEMENT_DYNCREATE*：

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

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

于是，以*CFrameWnd* 为例，下列程序代码：

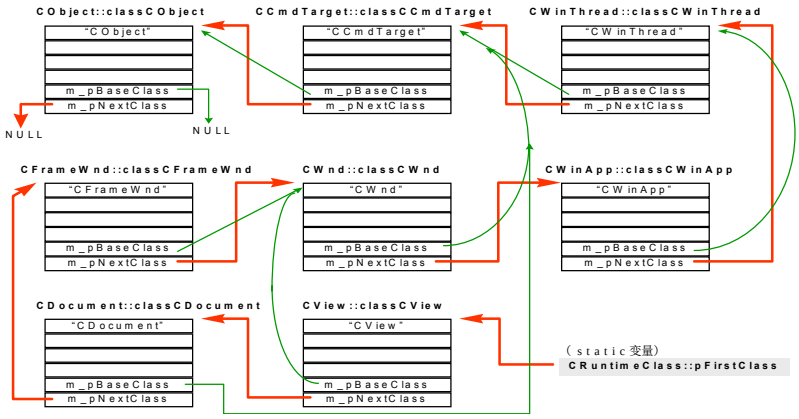
```
// in header file
class CFrameWnd : public CWnd
{
       DECLARE_DYNCREATE(CFrameWnd)
   ...
};

// in implementation file
IMPLEMENT_DYNCREATE(CFrameWnd, CWnd)
```

就被展开如下（注意，编译器选项/P 可得前置处理结果）：

```
// in header file
class CFrameWnd : public CWnd
{
public:
       static CRuntimeClass classCFrameWnd;
       virtual CRuntimeClass* GetRuntimeClass() const;
       static CObject* PASCAL CreateObject();
   ...
};

// in implementation file
CObject* PASCAL CFrameWnd::CreateObject()
       { return new CFrameWnd; }
```

```
static char _lpszCFrameWnd[] = "CFrameWnd";
CRuntimeClass CFrameWnd::classCFrameWnd = {
       _lpszCFrameWnd, sizeof(CFrameWnd), 0xFFFF, CFrameWnd::CreateObject,
              RUNTIME_CLASS(CWnd), NULL };
static AFX_CLASSINIT _init_CFrameWnd(&CFrameWnd::classCFrameWnd);
CRuntimeClass* CFrameWnd::GetRuntimeClass() const
       { return &CFrameWnd::classCFrameWnd; }
```

图示如下：



「对象生成器」*CreateObject* 函数很简单，只要说*new* 就好。

从宏的定义我们很清楚可以看出，拥有动态生成（Dynamic Creation）能力的类别库，必然亦拥有执行时期类型识别（RTTI）能力，因为*_DYNCREATE* 宏涵盖了*_DYNAMIC* 宏。

注意：以下范例直接跳到Frame6。本书第一版有一个Frame5 程序，用以仿真MFC 2.5 对动态生成的作法。往事已矣，读者曾经来函表示没有必要提过去的东西，徒增脑力负荷。我想也是，况且MFC 4.x 的作法更好更容易了解，所以我把Frame5 拿掉了，但仍保留着序号。

范例程序Frame6 在.h 档中有这些类别声明：

```
class CObject
{
...
};
class CCmdTarget : public CObject
{
       DECLARE_DYNAMIC(CCmdTarget)
...
};
class CWinThread : public CCmdTarget
{
       DECLARE_DYNAMIC(CWinThread)
...
};
class CWinApp : public CWinThread
{
       DECLARE_DYNAMIC(CWinApp)
...
};
class CDocument : public CCmdTarget
{
       DECLARE_DYNAMIC(CDocument)
...
};
class CWnd : public CCmdTarget
{
       DECLARE_DYNCREATE(CWnd)
...
};
class CFrameWnd : public CWnd
{
       DECLARE_DYNCREATE(CFrameWnd)
...
};
class CView : public CWnd
{
       DECLARE_DYNAMIC(CView)
...
};
class CMyWinApp : public CWinApp
{
...
};
class CMyFrameWnd : public CFrameWnd
```

```
{
       DECLARE_DYNCREATE(CMyFrameWnd)
...
};
class CMyDoc : public CDocument
{
       DECLARE_DYNCREATE(CMyDoc)
...
};
class CMyView : public CView
{
       DECLARE_DYNCREATE(CMyView)
...
};
```

在.cpp 档中又有这些动作：

```
IMPLEMENT_DYNAMIC(CCmdTarget, CObject)
IMPLEMENT_DYNAMIC(CWinThread, CCmdTarget)
IMPLEMENT_DYNAMIC(CWinApp, CWinThread)
IMPLEMENT_DYNCREATE(CWnd, CCmdTarget)
IMPLEMENT_DYNCREATE(CFrameWnd, CWnd)
IMPLEMENT_DYNAMIC(CDocument, CCmdTarget)
IMPLEMENT_DYNAMIC(CView, CWnd)
IMPLEMENT_DYNCREATE(CMyFrameWnd, CFrameWnd)
IMPLEMENT_DYNCREATE(CMyDoc, CDocument)
IMPLEMENT_DYNCREATE(CMyView, CView)
```

于是组织出图3-2 这样一个大网。

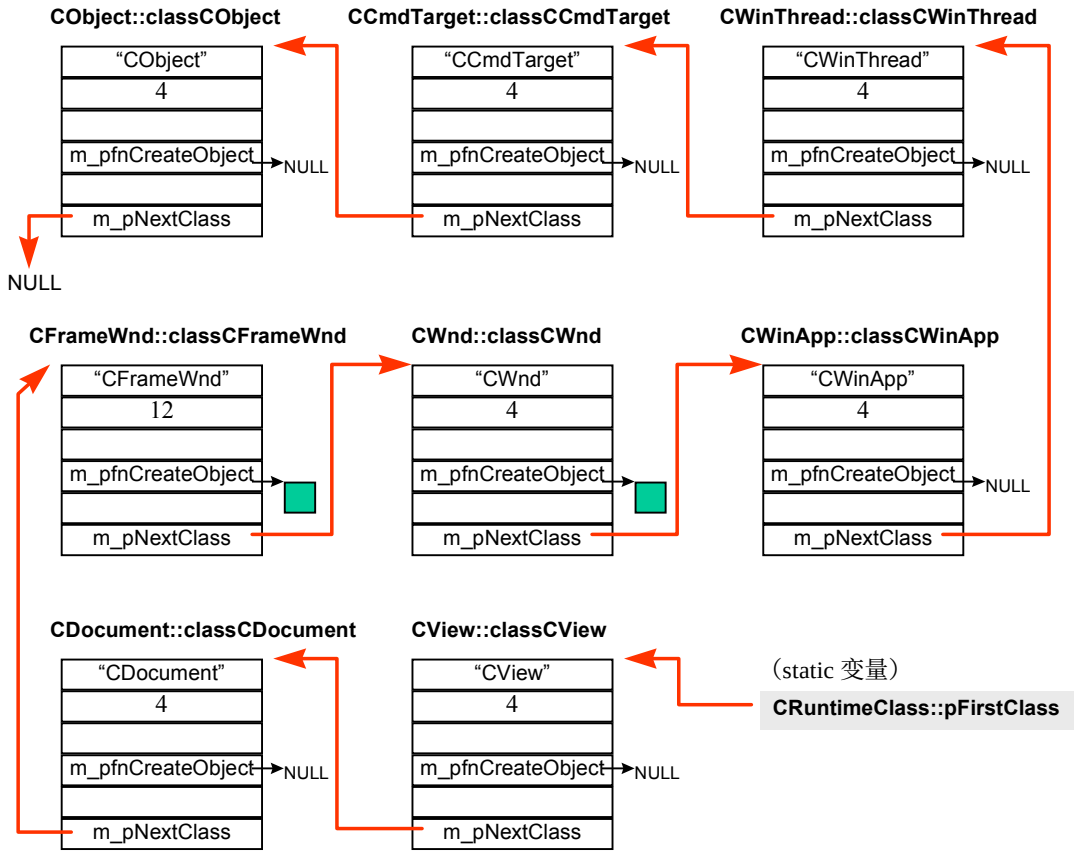


图3-2 以CRuntimeClass 对象构成的「类别型录网」。本图只列出与动态生成（Dynamic Creation）有关系的成员。凡是m_pfnCreateObject不为NULL 者， 即可动态生成。

现在，我们开始仿真动态生成。首先在*main* 函数中加上这一段码：

```
void main()
{
  ...
  //Test Dynamic Creation
  CRuntimeClass* pClassRef;
  CObject* pOb;
  while(1)
  {
```

```
   if ((pClassRef = CRuntimeClass::Load()) == NULL)
      break;

   pOb = pClassRef->CreateObject();
   if (pOb != NULL)
      pOb->SayHello();
 }
}
```

并设计 *CRuntimeClass*::*CreateObject* 和 *CRuntimeClass*::*Load* 如下：

```
// in implementation file
CObject* CRuntimeClass::CreateObject()
{
       if (m_pfnCreateObject == NULL)
       {
              TRACE1("Error: Trying to create object which is not "
                     "DECLARE_DYNCREATE \nor DECLARE_SERIAL: %hs.\n",
                      m_lpszClassName);
              return NULL;
       }

       CObject* pObject = NULL;
       pObject = (*m_pfnCreateObject)();

       return pObject;
}

CRuntimeClass* PASCAL CRuntimeClass::Load()
{
       char szClassName[64];
       CRuntimeClass* pClass;
       // JJHOU : instead of Load from file, we Load from cin.
       cout << "enter a class name...  ";
       cin >> szClassName;

       for (pClass = pFirstClass; pClass != NULL; pClass = pClass->m_pNextClass)
       {
              if (strcmp(szClassName, pClass->m_lpszClassName) == 0)
                     return pClass;
       }

       TRACE1("Error: Class not found: %s \n", szClassName);
       return NULL; // not found
}
```

然后，为了验证这样的动态生成机制的确有效（也就是对象的确被产生了），我让许多个类别的构造式都输出一段文字，而且在取得对象指针后，真的去调用该对象的一个成员函数*SayHello*。我把*SayHello* 设计为虚拟函数，所以根据不同的对象类型，会调用到不同的*SayHello* 函数，出现不同的输出字符串。

请注意，*main* 函数中的*while* 循环必须等到*CRuntimeClass*::*Load* 传回*NULL* 才会停止，而*CRuntimeClass*::*Load* 是在它从整个「类别型录网」中找不到它要找的那个类别名称时，才传回*NULL*。这些都是我为了仿真与示范，所采取的权宜设计。

Frame6 的命令列编译联结动作是（环境变量必须先设定好，请参考第 4 章的「安装与设定」一节）：

```
cl my.cpp mfc.cpp <Enter>
```

下面是Frame6 的执行结果。粗体表示我（程序执行者）在屏幕上输入的类别名称：

```
enter a class name...  CObject
Error: Trying to create object which is not DECLARE_DYNCREATE
or DECLARE_SERIAL: CObject.

enter a class name...  CView
Error: Trying to create object which is not DECLARE_DYNCREATE
or DECLARE_SERIAL: CView.

enter a class name...  CMyView
CWnd Constructor
CMyView Constructor
Hello CMyView

enter a class name...  CMyFrameWnd
CWnd Constructor
CFrameWnd Constructor
CMyFrameWnd Constructor
Hello CMyFrameWnd

enter a class name...  CMyDoc
CMyDoc Constructor
Hello CMyDoc
```

```
enter a class name...  CWinApp
Error: Trying to create object which is not DECLARE_DYNCREATE
or DECLARE_SERIAL: CWinApp.
enter a class name... CJjhou （故意输入一个不在「类别型录网」中的类别名称）
Error: Class not found: CJjhou （程序结束）
```

Frame6　范例程序

MFC.H

```
#0001  #define BOOL int
#0002  #define TRUE 1
#0003  #define FALSE 0
#0004  #define LPCSTR  LPSTR
#0005  typedef char*   LPSTR;
#0006  #define UINT int
#0007  #define PASCAL _stdcall
#0008  #define TRACE1 printf
#0009
#0010  #include <iostream.h>
#0011  #include <stdio.h>
#0012  #include <string.h>
#0013
#0014  class CObject;
#0015
#0016  struct CRuntimeClass
#0017  {
#0018  // Attributes
#0019     LPCSTR m_lpszClassName;
#0020     int m_nObjectSize;
#0021     UINT m_wSchema; // schema number of the loaded class
#0022     CObject* (PASCAL* m_pfnCreateObject)(); // NULL => abstract class
#0023     CRuntimeClass* m_pBaseClass;
#0024
#0025     CObject* CreateObject();
#0026     static CRuntimeClass* PASCAL Load();
#0027
#0028     // CRuntimeClass objects linked together in simple list
#0029     static CRuntimeClass* pFirstClass; // start of class list
#0030     CRuntimeClass* m_pNextClass;    // linked list of registered classes
#0031  };
#0032
#0033  struct AFX_CLASSINIT
```

```
#0034        { AFX_CLASSINIT(CRuntimeClass* pNewClass); };
#0035
#0036  #define RUNTIME_CLASS(class_name) \
#0037        (&class_name::class##class_name)
#0038
#0039  #define DECLARE_DYNAMIC(class_name) \
#0040  public: \
#0041        static CRuntimeClass class##class_name; \
#0042        virtual CRuntimeClass* GetRuntimeClass() const;
#0043
#0044  #define DECLARE_DYNCREATE(class_name) \
#0045        DECLARE_DYNAMIC(class_name) \
#0046        static CObject* PASCAL CreateObject();
#0047
#0048  #define _IMPLEMENT_RUNTIMECLASS(class_name, base_class_name, wSchema, pfnNew) \
#0049        static char _lpsz##class_name[] = #class_name; \
#0050        CRuntimeClass class_name::class##class_name = { \
#0051               _lpsz##class_name, sizeof(class_name), wSchema, pfnNew, \
#0052                      RUNTIME_CLASS(base_class_name), NULL }; \
#0053        static AFX_CLASSINIT _init_##class_name(&class_name::class##class_name); \
#0054        CRuntimeClass* class_name::GetRuntimeClass() const \
#0055               { return &class_name::class##class_name; } \
#0056
#0057  #define IMPLEMENT_DYNAMIC(class_name, base_class_name) \
#0058        _IMPLEMENT_RUNTIMECLASS(class_name, base_class_name, 0xFFFF, NULL)
#0059
#0060  #define IMPLEMENT_DYNCREATE(class_name, base_class_name) \
#0061        CObject* PASCAL class_name::CreateObject() \
#0062               { return new class_name; } \
#0063        _IMPLEMENT_RUNTIMECLASS(class_name, base_class_name, 0xFFFF, \
#0064               class_name::CreateObject)
#0065
#0066  class CObject
#0067  {
#0068  public:
#0069    CObject::CObject()  {
#0070                        }
#0071    CObject::~CObject() {
#0072                         }
#0073
#0074    virtual CRuntimeClass* GetRuntimeClass() const;
#0075    BOOL IsKindOf(const CRuntimeClass* pClass) const;
#0076
#0077  public:
#0078    static CRuntimeClass classCObject;
#0079    virtual void SayHello() { cout << "Hello CObject \n"; }
```

```
#0080 };
#0081
#0082 class CCmdTarget : public CObject
#0083 {
#0084        DECLARE_DYNAMIC(CCmdTarget)
#0085 public:
#0086   CCmdTarget::CCmdTarget()  {
#0087                             }
#0088   CCmdTarget::~CCmdTarget() {
#0089                             }
#0090 };
#0091
#0092 class CWinThread : public CCmdTarget
#0093 {
#0094        DECLARE_DYNAMIC(CWinThread)
#0095 public:
#0096   CWinThread::CWinThread()  {
#0097                             }
#0098   CWinThread::~CWinThread() {
#0099                             }
#0100
#0101   virtual BOOL InitInstance() {
#0102                               return TRUE;
#0103                              }
#0104   virtual int Run() {
#0105                     return 1;
#0106                    }
#0107 };
#0108
#0109 class CWnd;
#0110
#0111 class CWinApp : public CWinThread
#0112 {
#0113        DECLARE_DYNAMIC(CWinApp)
#0114 public:
#0115   CWinApp* m_pCurrentWinApp;
#0116   CWnd* m_pMainWnd;
#0117
#0118 public:
#0119   CWinApp::CWinApp()  {
#0120                       m_pCurrentWinApp = this;
#0121                      }
#0122   CWinApp::~CWinApp() {
#0123                       }
#0124
#0125   virtual BOOL InitApplication() {
```

```
#0126                                  return TRUE;
#0127                                 }
#0128   virtual BOOL InitInstance()    {
#0129                                  return TRUE;
#0130                                 }
#0131   virtual int Run() {
#0132                       return CWinThread::Run();
#0133                      }
#0134 };
#0135
#0136
#0137 class CDocument : public CCmdTarget
#0138 {
#0139        DECLARE_DYNAMIC(CDocument)
#0140 public:
#0141   CDocument::CDocument()   {
#0142                            }
#0143   CDocument::~CDocument()  {
#0144                            }
#0145 };
#0146
#0147 class CWnd : public CCmdTarget
#0148 {
#0149        DECLARE_DYNCREATE(CWnd)
#0150 public:
#0151   CWnd::CWnd()   {
#0152                   cout << "CWnd Constructor \n";
#0153                  }
#0154   CWnd::~CWnd()  {
#0155                  }
#0156
#0157   virtual BOOL Create();
#0158   BOOL CreateEx();
#0159   virtual BOOL PreCreateWindow();
#0160   void SayHello() { cout << "Hello CWnd \n"; }
#0161 };
#0162
#0163 class CFrameWnd : public CWnd
#0164 {
#0165        DECLARE_DYNCREATE(CFrameWnd)
#0166 public:
#0167   CFrameWnd::CFrameWnd()   {
#0168                             cout << "CFrameWnd Constructor \n";
#0169                            }
#0170   CFrameWnd::~CFrameWnd()  {
#0171                            }
```

```
#0172   BOOL Create();
#0173   virtual BOOL PreCreateWindow();
#0174   void SayHello() { cout << "Hello CFrameWnd \n"; }
#0175 };
#0176
#0177 class CView : public CWnd
#0178 {
#0179       DECLARE_DYNAMIC(CView)
#0180 public:
#0181   CView::CView()   {
#0182                    }
#0183   CView::~CView()  {
#0184                    }
#0185 };
#0186
#0187 // global function
#0188 CWinApp* AfxGetApp();
```

MFC.CPP

```
#0001 #include "my.h"  // it should be mfc.h, but for CMyWinApp definition, so...
#0002
#0003 extern CMyWinApp theApp;
#0004
#0005 static char szCObject[] = "CObject";
#0006 struct CRuntimeClass CObject::classCObject =
#0007       { szCObject, sizeof(CObject), 0xffff, NULL, NULL };
#0008 static AFX_CLASSINIT _init_CObject(&CObject::classCObject);
#0009
#0010 CRuntimeClass* CRuntimeClass::pFirstClass = NULL;
#0011
#0012 AFX_CLASSINIT::AFX_CLASSINIT(CRuntimeClass* pNewClass)
#0013 {
#0014       pNewClass->m_pNextClass = CRuntimeClass::pFirstClass;
#0015       CRuntimeClass::pFirstClass = pNewClass;
#0016 }
#0017
#0018 CObject* CRuntimeClass::CreateObject()
#0019 {
#0020       if (m_pfnCreateObject == NULL)
#0021       {
#0022             TRACE1("Error: Trying to create object which is not "
#0023                   "DECLARE_DYNCREATE \nor DECLARE_SERIAL: %hs.\n",
#0024                   m_lpszClassName);
#0025             return NULL;
#0026       }
```

```
#0027
#0028        CObject* pObject = NULL;
#0029        pObject = (*m_pfnCreateObject)();
#0030
#0031        return pObject;
#0032  }
#0033
#0034  CRuntimeClass* PASCAL CRuntimeClass::Load()
#0035  {
#0036        char szClassName[64];
#0037        CRuntimeClass* pClass;
#0038
#0039        // JJHOU : instead of Load from file, we Load from cin.
#0040        cout << "enter a class name...  ";
#0041        cin >> szClassName;
#0042
#0043        for (pClass = pFirstClass; pClass != NULL; pClass = pClass->m_pNextClass)
#0044        {
#0045              if (strcmp(szClassName, pClass->m_lpszClassName) == 0)
#0046                    return pClass;
#0047        }
#0048
#0049        TRACE1("Error: Class not found: %s \n", szClassName);
#0050        return NULL; // not found
#0051  }
#0052
#0053  CRuntimeClass* CObject::GetRuntimeClass() const
#0054  {
#0055        return &CObject::classCObject;
#0056  }
#0057
#0058  BOOL CObject::IsKindOf(const CRuntimeClass* pClass) const
#0059  {
#0060        CRuntimeClass* pClassThis = GetRuntimeClass();
#0061        while (pClassThis != NULL)
#0062        {
#0063              if (pClassThis == pClass)
#0064                    return TRUE;
#0065              pClassThis = pClassThis->m_pBaseClass;
#0066        }
#0067        return FALSE;      // walked to the top, no match
#0068  }
#0069
#0070  BOOL CWnd::Create()
#0071  {
#0072    return TRUE;
```

```
#0073  }
#0074
#0075  BOOL CWnd::CreateEx()
#0076  {
#0077    PreCreateWindow();
#0078    return TRUE;
#0079  }
#0080
#0081  BOOL CWnd::PreCreateWindow()
#0082  {
#0083    return TRUE;
#0084  }
#0085
#0086  BOOL CFrameWnd::Create()
#0087  {
#0088    CreateEx();
#0089    return TRUE;
#0090  }
#0091
#0092  BOOL CFrameWnd::PreCreateWindow()
#0093  {
#0094    return TRUE;
#0095  }
#0096
#0097  CWinApp* AfxGetApp()
#0098  {
#0099    return theApp.m_pCurrentWinApp;
#0100  }
#0101
#0102  IMPLEMENT_DYNAMIC(CCmdTarget, CObject)
#0103  IMPLEMENT_DYNAMIC(CWinThread, CCmdTarget)
#0104  IMPLEMENT_DYNAMIC(CWinApp, CWinThread)
#0105  IMPLEMENT_DYNAMIC(CDocument, CCmdTarget)
#0106  IMPLEMENT_DYNCREATE(CWnd, CCmdTarget)
#0107  IMPLEMENT_DYNAMIC(CView, CWnd)
#0108  IMPLEMENT_DYNCREATE(CFrameWnd, CWnd)
```

MY.H

```
#0001  #include <iostream.h>
#0002  #include "mfc.h"
#0003
#0004  class CMyWinApp : public CWinApp
#0005  {
#0006  public:
#0007    CMyWinApp::CMyWinApp()   {
```

```
#0008                              }
#0009   CMyWinApp::~CMyWinApp()  {
#0010                              }
#0011
#0012   virtual BOOL InitInstance();
#0013 };
#0014
#0015 class CMyFrameWnd : public CFrameWnd
#0016 {
#0017         DECLARE_DYNCREATE(CMyFrameWnd)
#0018 public:
#0019   CMyFrameWnd();
#0020   ~CMyFrameWnd()  {
#0021                    }
#0022   void SayHello() { cout << "Hello CMyFrameWnd \n"; }
#0023 };
#0024
#0025 class CMyDoc : public CDocument
#0026 {
#0027         DECLARE_DYNCREATE(CMyDoc)
#0028 public:
#0029   CMyDoc::CMyDoc()  {
#0030                      cout << "CMyDoc Constructor \n";
#0031                     }
#0032   CMyDoc::~CMyDoc() {
#0033                     }
#0034   void SayHello() { cout << "Hello CMyDoc \n"; }
#0035 };
#0036
#0037 class CMyView : public CView
#0038 {
#0039         DECLARE_DYNCREATE(CMyView)
#0040 public:
#0041   CMyView::CMyView()   {
#0042                        cout << "CMyView Constructor \n";
#0043                       }
#0044   CMyView::~CMyView()  {
#0045                        }
#0046   void SayHello() { cout << "Hello CMyView \n"; }
#0047 };
#0048
#0049 // global function
#0050 void AfxPrintAllClasses();
```

MY.CPP

```
#0001  #include "my.h"
#0002
#0003  CMyWinApp theApp;
#0004
#0005  BOOL CMyWinApp::InitInstance()
#0006  {
#0007      m_pMainWnd = new CMyFrameWnd;
#0008      return TRUE;
#0009  }
#0010
#0011  CMyFrameWnd::CMyFrameWnd()
#0012  {
#0013      cout << "CMyFrameWnd Constructor \n";
#0014      Create();
#0015  }
#0016
#0017  IMPLEMENT_DYNCREATE(CMyFrameWnd, CFrameWnd)
#0018  IMPLEMENT_DYNCREATE(CMyDoc, CDocument)
#0019  IMPLEMENT_DYNCREATE(CMyView, CView)
#0020
#0021  void PrintAllClasses()
#0022  {
#0023      CRuntimeClass* pClass;
#0024
#0025      // just walk through the simple list of registered classes
#0026      for (pClass = CRuntimeClass::pFirstClass; pClass != NULL;
#0027             pClass = pClass->m_pNextClass)
#0028      {
#0029          cout << pClass->m_lpszClassName << "\n";
#0030          cout << pClass->m_nObjectSize << "\n";
#0031          cout << pClass->m_wSchema << "\n";
#0032      }
#0033  }
#0034  //--------------------------------------------------------------------
#0035  // main
#0036  //--------------------------------------------------------------------
#0037  void main()
#0038  {
#0039    CWinApp* pApp = AfxGetApp();
#0040
#0041    pApp->InitApplication();
#0042    pApp->InitInstance();
#0043    pApp->Run();
#0044
```

```
#0045   //Test Dynamic Creation
#0046   CRuntimeClass* pClassRef;
#0047   CObject* pOb;
#0048   while(1)
#0049   {
#0050     if ((pClassRef = CRuntimeClass::Load()) == NULL)
#0051        break;
#0052
#0053     pOb = pClassRef->CreateObject();
#0054     if (pOb != NULL)
#0055        pOb->SayHello();
#0056   }
#0057 }
```

Persistence（永续生存）机制

对象导向有一个术语：Persistence，意思就是把对象永久保留下来。Power 一关，啥都没有，对象又如何能够永续存留？

当然是写到文件去 ！

把资料写到文件，很简单。在Document/View 架构中，资料都放在一份document（文件）里头，我们只要把其中的成员变量依续写进文件即可。成员变量很可能是个对象，而面对对象，我们首先应该记载其类别名称，然后才是对象中的资料。

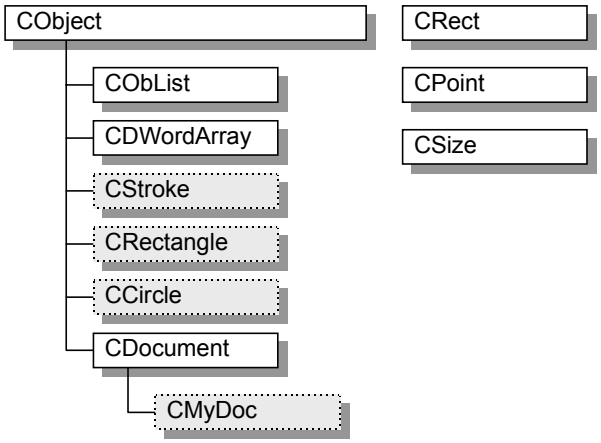
读档就有点麻烦了。当程序从文件中读到一个类别名称，它如何实现（instantiate）一个对象？呵，这不就是动态生成的技术吗？我们在前一章已经解决掉了。

MFC 有一套Serialize 机制，目的在于把档名的选择、文件的开关、缓冲区的建立、资料的读写、萃取运算子（>>）和嵌入运算子（<<）的多载（overload）、对象的动态生成都包装起来。 ...

上述Serialize 的各部份工作，除了资料的读写和对象的动态生成，其余都是支节。动态生成的技术已经解决，让我们集中火力，分析资料的读写动作。

Serialize（资料读写）

假设我有一份文件，用以记录一张图形。图形只有三种基本元素：线条（Stroke）、圆形、矩形。我打算用以下类别，组织这份文件：



其中*CObList* 和*CDWordArray* 是MFC 提供的类别，前者是一个串行，可放置任何从*CObject* 衍生下来的对象，后者是一个数组，每一个元素都是"double word"。另外三个类别：*CStroke* 和*CRectangle* 和*CCircle*，是我从*CObject* 中衍生下来的类别。

```
class CMyDoc : public CDocument
{
   CObList m_graphList;
   CSize   m_sizeDoc;
   ...
};
class CStroke : public CObject
{
   CDWordArray m_ptArray;   // series of connected points
   ...
};
class CRectangle : public CObject
{
   CRect m_rect;
   ...
};
class CCircle : public CObject
```

```
{
   CPoint  m_center;
   UINT    m_radius;
   ...
};
```

假设现有一份文件，内容如图3-3，如果你是Serialize 机制的设计者，你希望怎么做呢？把图3-3 写成这样的文件内容好吗：

```
06 00                         ;CObList elements count

07 00                         ;class name string length
43 53 74 72 6F 6B 65          ;"CStroke"
02 00                         ;DWordArray size
28 00 13 00                   ;point
28 00 13 00                   ;point

0A 00                         ;class name string length
43 52 65 63 74 61 6E 67 6C 65 ;"CRectangle"
11 00 22 00 33 00 44 00       ;CRect

07 00                         ;class name string length
43 43 69 72 63 6C 65          ;"CCircle"
55 00 66 00 77 00             ;CPoint & radius

07 00                         ;class name string length
43 53 74 72 6F 6B 65          ;"CStroke"
02 00                         ;DWordArray size
28 00 35 00                   ;point
28 00 35 00                   ;point

0A 00                         ;class name string length
43 52 65 63 74 61 6E 67 6C 65 ;"CRectangle"
11 00 22 00 33 00 44 00       ;CRect

07 00                          ;class name string length
43 43 69 72 63 6C 65           ;"CCircle"
55 00 66 00 77 00              ;CPoint & radius
```

还算堪用。但如果考虑到屏幕卷动的问题，以及打印输出的问题，应该在最前端增加「文件大小」。另外，如果这份文件有100 条线条，50 个圆形，80 个矩形，难不成我们要记录230 个类别名称？应该有更好的方法才是。

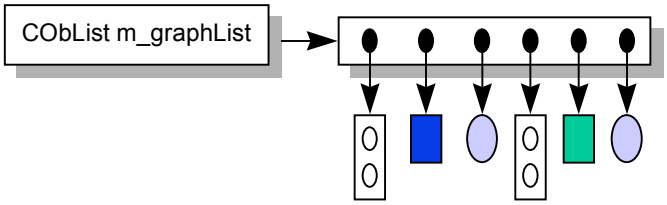


图3-3 一个串行， 内含三种基本图形： 线条、圆形、矩形。

我们可以在每次记录对象内容的时候，先写入一个代码，表示此对象之类别是否曾在档案中记录过了。如果是新类别，乖乖地记录其类别名称；如果是旧类别，则以代码表示。这样可以节省文件大小以及程序用于解析的时间。啊，不要看到文件大小就想到硬盘很便宜，桌上的一切都将被带到网上，你得想想网络频宽这回事。

还有一个问题。文件的「版本」如何控制？旧版程序读取新版文件，新版程序读取旧版文件，都可能出状况。为了防弊，最好把版本号码记录上去。最好是每个类别有自己的版本号码。

下面是新的构想，也就是Serialization 的目标：

```
20 03 84 03                ;Document Size
06 00                      ;CObList elements count

FF FF                      ;new class tag
02 00                      ;schema
07 00                      ;class name string length
43 53 74 72 6F 6B 65       ;"CStroke"
02 00                      ;DWordArray size
28 00 13 00                ;point
28 00 13 00                ;point

FF FF                          ;new class tag
01 00                          ;schema
0A 00                          ;class name string length
43 52 65 63 74 61 6E 67 6C 65  ;"CRectangle"
11 00 22 00 33 00 44 00        ;CRect

FF FF                          ;new class tag
01 00                          ;schema
07 00                          ;class name string length
43 43 69 72 63 6C 65           ;"CCircle"
55 00 66 00 77 00              ;CPoint & radius

01 80                          ;old class tag
02 00                          ;DWordArray size
28 00 35 00                    ;point
28 00 35 00                    ;point

03 80                          ;old class tag
11 00 22 00 33 00 44 00        ;CRect

05 80                          ;old class tag
55 00 66 00 77 00              ;CPoint & radius
```

我希望有一个专门负责Serialization 的函数，就叫作*Serialize* 好了。假设现在我的Document 类别名称为*CScribDoc*，我希望有这么便利的程序方法（请仔细琢磨琢磨其便利性）：

```
void CScribDoc::Serialize(CArchive& ar)
{
   if (ar.IsStoring())
      ar << m_sizeDoc;
   else
      ar >> m_sizeDoc;
   m_graphList.Serialize(ar);
}

void CObList::Serialize(CArchive& ar)
{
   if (ar.IsStoring()) {
```

```
        ar << (WORD) m_nCount;
        for (CNode* pNode = m_pNodeHead; pNode != NULL; pNode = pNode->pNext)
            ar << pNode->data;
    }
    else {
        WORD nNewCount;
        ar >> nNewCount;
        while (nNewCount--) {
            CObject* newData;
            ar >> newData;
            AddTail(newData);
        }
    }
}

void CStroke::Serialize(CArchive& ar)
{
    m_ptArray.Serialize(ar);
}

void CDWordArray::Serialize(CArchive& ar)
{
    if (ar.IsStoring()) {
        ar << (WORD) m_nSize;
        for (int i = 0; i < m_nSize; i++)
            ar << m_pData[i];
    }
    else {
        WORD nOldSize;
        ar >> nOldSize;
        for (int i = 0; i < m_nSize; i++)
            ar >> m_pData[i];
    }
}

void CRectangle::Serialize(CArchive& ar)
{
    if (ar.IsStoring())
        ar << m_rect;
    else
        ar >> m_rect;
}

void CCircle::Serialize(CArchive& ar)
{
    if (ar.IsStoring()) {
```

```
        ar << (WORD)m_center.x;
        ar << (WORD)m_center.y;
        ar << (WORD)m_radius;
    }
    else {
        ar >> (WORD&)m_center.x;
        ar >> (WORD&)m_center.y;
        ar >> (WORD&)m_radius;
    }
}
```

每一个可写到文件或可从文件中读出的类别，都应该有它自己的*Serailize* 函数，负责它自己的资料读写文件动作。此类别并且应该改写<< 运算子和>> 运算子，把资料导流到archive 中。archive 是什么？是一个与文件息息相关的缓冲区，暂时你可以想象它就是文件的化身。当图3-3 的文件写入文件时，*Serialize* 函数的调用次序如图3-4。

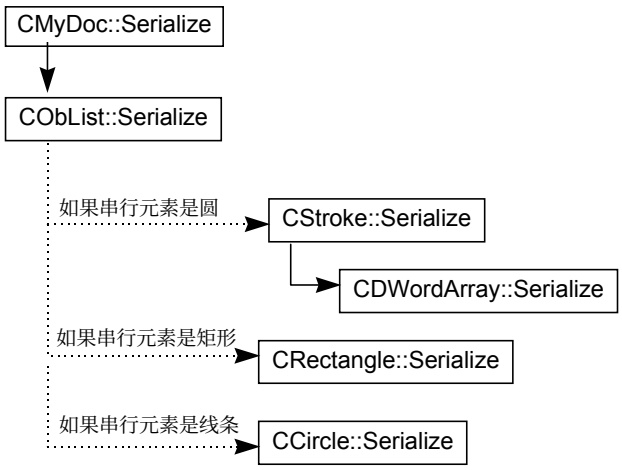


图3-4 图3-3 的文件内容写入文件时，Serialize 函数的调用次序。

DECLARE_SERIAL／ IMPLEMENT_SERIAL 宏

要将<< 和>> 两个运算子多载化，还要让*Serialize* 函数神不知鬼不觉地放入类别声明之中，最好的作法仍然是使用宏。

类别之能够进行文件读写动作，前提是拥有动态生成的能力，所以，MFC 设计了两个宏*DECLARE_SERIAL* 和*IMPLEMENT_SERIAL*：

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

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

为了在每一个对象被处理（读或写）之前，能够处理琐屑的工作，诸如判断是否第一次出现、记录版本号码、记录文件名等工作，*CRuntimeClass* 需要两个函数*Load* 和*Store*：

```
struct CRuntimeClass
{
// Attributes
       LPCSTR m_lpszClassName;
       int m_nObjectSize;
       UINT m_wSchema; // schema number of the loaded class
       CObject* (PASCAL* m_pfnCreateObject)(); // NULL => abstract class
       CRuntimeClass* m_pBaseClass;

       CObject* CreateObject();
       void Store(CArchive& ar) const;
       static CRuntimeClass* PASCAL Load(CArchive& ar, UINT* pwSchemaNum);

       // CRuntimeClass objects linked together in simple list
       static CRuntimeClass* pFirstClass; // start of class list
       CRuntimeClass* m_pNextClass;     // linked list of registered classes
};
```

你已经在上一节看过*Load* 函数，当时为了简化，我把它的参数拿掉，改为由屏幕上获得类别名称，事实上它应该是从文件中读一个类别名称。至于*Store* 函数，是把类别名称写入文件中：

```
// Runtime class serialization code
CRuntimeClass* PASCAL CRuntimeClass::Load(CArchive& ar, UINT* pwSchemaNum)
{
    WORD nLen;
    char szClassName[64];
    CRuntimeClass* pClass;

    ar >> (WORD&)(*pwSchemaNum) >> nLen;

    if (nLen >= sizeof(szClassName) || ar.Read(szClassName, nLen) != nLen)
           return NULL;
    szClassName[nLen] = '\0';

    for (pClass = pFirstClass; pClass != NULL; pClass = pClass->m_pNextClass)
    {
        if (lstrcmp(szClassName, pClass->m_lpszClassName) == 0)
            return pClass;
    }
    return NULL; // not found
}

void CRuntimeClass::Store(CArchive& ar) const
       // stores a runtime class description
{
       WORD nLen = (WORD)lstrlenA(m_lpszClassName);
       ar << (WORD)m_wSchema << nLen;
       ar.Write(m_lpszClassName, nLen*sizeof(char));
}
```

图3-4 的例子中，为了让整个Serialization 机制运作起来，我们必须做这样的类别声明：

```
class CScribDoc : public CDocument
{
       DECLARE_DYNCREATE(CScribDoc)
...
};
class CStroke : public CObject
{
       DECLARE_SERIAL(CStroke)
public:
```

```
        void Serialize(CArchive&);
...
};
class CRectangle : public CObject
{
        DECLARE_SERIAL(CRectangle)
public:
        void Serialize(CArchive&);
...
};
class CCircle : public CObject
{
        DECLARE_SERIAL(CCircle)
public:
        void Serialize(CArchive&);
...
};
```

以及在.CPP 档中做这样的动作：

```
IMPLEMENT_DYNCREATE(CScribDoc, CDocument)
IMPLEMENT_SERIAL(CStroke, CObject, 2)
IMPLEMENT_SERIAL(CRectangle, CObject, 1)
IMPLEMENT_SERIAL(CCircle, CObject, 1)
```

然后呢？分头设计*CStroke*、*CRectangle* 和*CCircle* 的*Serialize* 函数吧。

当然，毫不令人意外地，MFC 源代码中的*CObList* 和*CDWordArray* 有这样的内容：

```
// in header files
class CDWordArray : public CObject
{
        DECLARE_SERIAL(CDWordArray)
public:
        void Serialize(CArchive&);
...
};
class CObList : public CObject
{
        DECLARE_SERIAL(CObList)
public:
        void Serialize(CArchive&);
...
};

// in implementation files
```

```
IMPLEMENT_SERIAL(CObList, CObject, 0)
IMPLEMENT_SERIAL(CDWordArray, CObject, 0)
```

而*CObject* 也多了一个虚拟函数*Serialize*：

```
class CObject
{
public:
       virtual void Serialize(CArchive& ar);
...
}
```

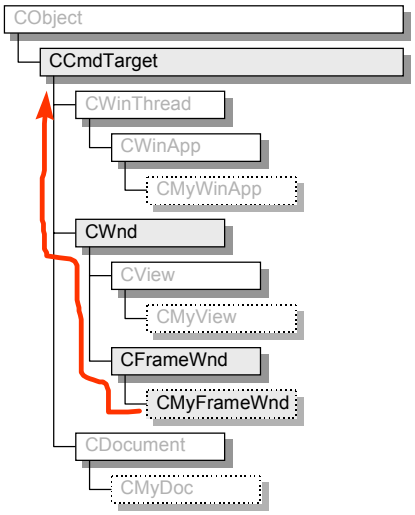
没有范例程序

抱歉，我没有准备DOS 版的Serialization 范例程序给你。你看到了，很多东西需要仿真：*CFile*、*CArchive*、*CObList*、*CDWordArray*、*CRect*、*CPoint*、运算子多载、*Serialize* 函数...。我干脆在本书第８章直接为你解释MFC 的作法，更好。

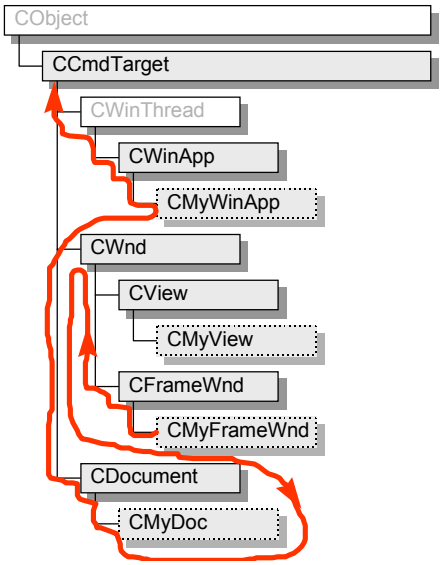
Message Mapping（消息映射）

Windows 程序靠消息的流动而维护生命。你已经在第一章看过了消息的一般处理方式，也就是在窗口函数中借着一个大大的*switch*/*case* 比对动作，判别消息再调用对应的处理例程。为了让大大的*switch*/*case* 比对动作简化，也让程序代码更模块化一些，我在第１章提供了一个简易的消息映射表作法，把消息和其处理例程关联起来。

当我们的类别库成立，如果其中与消息有关的类别（姑且叫作「消息标的类别」好了，在MFC 之中就是*CCmdTarget*）都是一条鞭式地继承，我们应该为每一个「消息标的类别」准备一个消息映射表，并且将基础类别与衍生类别之消息映射表串接起来。然后，当窗口函数做消息的比对时，我们就可以想办法导引它沿着这条路走过去：



但是，MFC 之中用来处理消息的C++ 类别，并不呈单鞭发展。作为application framework 的重要架构之一的document/view，也具有处理消息的能力（你现在可能还不清楚什么是 document/view，没有关系）。因此，消息藉以攀爬的路线应该有横流的机会：



消息如何流动，我们暂时先不管。是直线前进，或是中途换跑道，我们都暂时不管，本节先把这个攀爬路线网建立起来再说。这整个攀爬路线网就是所谓的消息映射表（Message Map）；说它是一张地图，当然也没有错。将消息与表格中的元素比对，然后调用对应的处理例程，这种动作我们也称之为消息映射（Message Mapping）。

为了尽量降低对正常（一般）类别声明和定义的影响，我们希望，最好能够像RTTI 和 Dynamic Creation 一样，用一两个宏就完成这巨大蜘蛛网的构造。最好能够像 *DECLARE_DYNAMIC* 和*IMPLEMENT_DYNAMIC* 宏那么方便。
首先定义一个数据结构：

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

其中的 AFX_MSGMAP_ENTRY 又是另一个数据结构：

```
struct AFX_MSGMAP_ENTRY  // MFC 4.0 format
{
       UINT nMessage;      // windows message
       UINT nCode;         // control code or WM_NOTIFY code
       UINT nID;           // control ID (or 0 for windows messages)
       UINT nLastID;       // used for entries specifying a range of control id's
       UINT nSig;          // signature type (action) or pointer to message #
       AFX_PMSG pfn;       // routine to call (or special value)
};
```

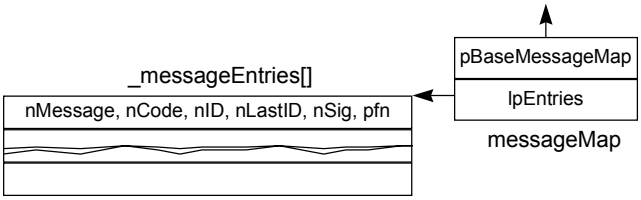
其中的AFX_PMSG 定义为函数指针：

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

然后我们定义一个宏：

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

于是，*DECLARE_MESSAGE_MAP* 就相当于声明了这样一个数据结构：



这个数据结构的内容填塞工作由三个宏完成：

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

#define ON_COMMAND(id, memberFxn) \
       { WM_COMMAND, 0, (WORD)id, (WORD)id, AfxSig_vv, (AFX_PMSG)memberFxn },

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

其中的AfxSig_end 定义为：

```
enum AfxSig
{
       AfxSig_end = 0,    // [marks end of message map]
       AfxSig_vv,
};
```

> AfxSig_xx 用来描述消息处理例程memberFxn 的类型（参数与回返值）。本例纯为仿真与简化，所以不在这上面作文章。真正讲到MFC 时（第四篇p580），我会再解释它。

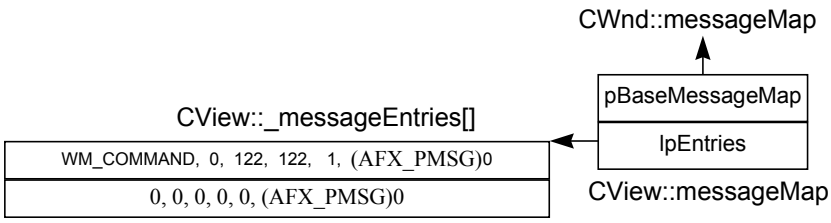
于是，以*CView*为例，下面的源代码：

```
// in header file
class CView : public CWnd
{
public:
  ...
  DECLARE_MESSAGE_MAP()
};

// in implementation file
#define CViewid 122
...
BEGIN_MESSAGE_MAP(CView, CWnd)
  ON_COMMAND(CViewid, 0)
END_MESSAGE_MAP()
```

就被展开成为：

```
// in header file
class CView : public CWnd
{
public:
  ...
  static AFX_MSGMAP_ENTRY _messageEntries[];
  static AFX_MSGMAP messageMap;
  virtual AFX_MSGMAP* GetMessageMap() const;
};

// in implementation file
AFX_MSGMAP* CView::GetMessageMap() const
       { return &CView::messageMap; }
AFX_MSGMAP CView::messageMap =
{ &(CWnd::messageMap),
       (AFX_MSGMAP_ENTRY*) &(CView::_messageEntries) };
AFX_MSGMAP_ENTRY CView::_messageEntries[] =
{
  { WM_COMMAND, 0, (WORD)122, (WORD)122, 1, (AFX_PMSG)0 },
  { 0, 0, 0, 0, 0, (AFX_PMSG)0 }
};
```

以图表示则为：



我们还可以定义各种类似*ON_COMMAND* 这样的宏，把各式各样的消息与特定的处理例程关联起来。MFC 里头就有名为*ON_WM_PAINT*、*ON_WM_CREATE*、*ON_WM_SIZE*...等等的宏。

我在Frame7 范例程序中为*CCmdTarget* 的每一衍生类别都产生类似上图的消息映射表：

```
// in header files
class CObject
{
    ... // 注意：CObject 并不属于消息流动网的一份子。
};
class CCmdTarget : public CObject
{
...
  DECLARE_MESSAGE_MAP()
};
class CWinThread : public CCmdTarget
{
 ... // 注意：CWinThread 并不属于消息流动网的一份子。
};
class CWinApp : public CWinThread
{
...
  DECLARE_MESSAGE_MAP()
};
class CDocument : public CCmdTarget
{
...
  DECLARE_MESSAGE_MAP()
};
class CWnd : public CCmdTarget
{
...
  DECLARE_MESSAGE_MAP()
```

```
};
class CFrameWnd : public CWnd
{
...
  DECLARE_MESSAGE_MAP()
};
class CView : public CWnd
{
...
  DECLARE_MESSAGE_MAP()
};
class CMyWinApp : public CWinApp
{
...
  DECLARE_MESSAGE_MAP()
};
class CMyFrameWnd : public CFrameWnd
{
...
  DECLARE_MESSAGE_MAP()
};
class CMyDoc : public CDocument
{
...
  DECLARE_MESSAGE_MAP()
};
class CMyView : public CView
{
...
  DECLARE_MESSAGE_MAP()
};
```

并且把各消息映射表的关联性架设起来，给予初值（每一个映射表都只有*ON_COMMAND*一个项目）：

```
// in implementation files
BEGIN_MESSAGE_MAP(CWnd, CCmdTarget)
ON_COMMAND(CWndid, 0)
END_MESSAGE_MAP()

BEGIN_MESSAGE_MAP(CFrameWnd, CWnd)
ON_COMMAND(CFrameWndid, 0)
END_MESSAGE_MAP()

BEGIN_MESSAGE_MAP(CDocument, CCmdTarget)
```

```
ON_COMMAND(CDocumentid, 0)
END_MESSAGE_MAP()

BEGIN_MESSAGE_MAP(CView, CWnd)
ON_COMMAND(CViewid, 0)
END_MESSAGE_MAP()

BEGIN_MESSAGE_MAP(CWinApp, CCmdTarget)
ON_COMMAND(CWinAppid, 0)
END_MESSAGE_MAP()

BEGIN_MESSAGE_MAP(CMyWinApp, CWinApp)
ON_COMMAND(CMyWinAppid, 0)
END_MESSAGE_MAP()

BEGIN_MESSAGE_MAP(CMyFrameWnd, CFrameWnd)
ON_COMMAND(CMyFrameWndid, 0)
END_MESSAGE_MAP()

BEGIN_MESSAGE_MAP(CMyDoc, CDocument)
ON_COMMAND(CMyDocid, 0)
END_MESSAGE_MAP()

BEGIN_MESSAGE_MAP(CMyView, CView)
ON_COMMAND(CMyViewid, 0)
END_MESSAGE_MAP()
```

同时也设定了消息的终极镖靶*CCmdTarget* 的映射表内容：

```
AFX_MSGMAP CCmdTarget::messageMap =
{
       NULL,
       &CCmdTarget::_messageEntries[0]
};

AFX_MSGMAP_ENTRY CCmdTarget::_messageEntries[] =
{
       { 0, 0, CCmdTargetid, 0, AfxSig_end, 0 }
};
```

于是，整个消息流动网就隐然成形了（图3-5）。

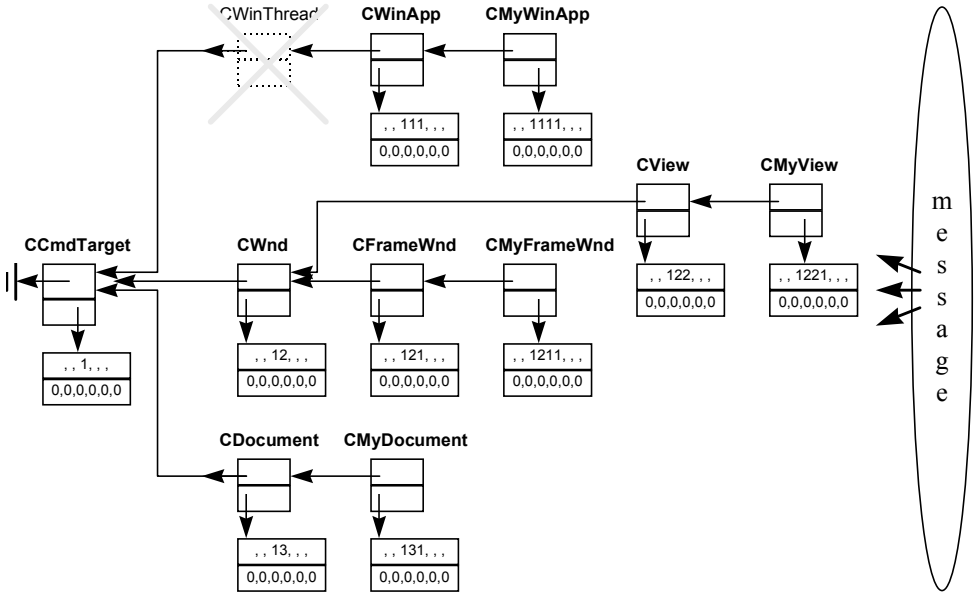


图3-5 Frame7 程序所架构起来的消息流动网（也就是Message Map）。

为了验证整个消息映射表，我必须在映射表中做点记号，等全部构造完成之后，再一一追踪把记号显示出来。我将为每一个类别的消息映射表加上这个项目：

```
ON_COMMAND(Classid, 0)
```

这样就可以把Classid 嵌到映射表中当作记号。正式用途（于MFC 中）当然不是这样，这只不过是权宜之计。

在*main* 函数中，我先产生四个对象（分别是*CMyWinApp*、*CMyFrameWnd*、*CMyDoc*、*CMyView* 对象）：

```
CMyWinApp theApp; // theApp 是CMyWinApp 对象
void main()
{
    CWinApp* pApp = AfxGetApp();
    pApp->InitApplication();
    pApp->InitInstance(); // 产生CMyFrameWnd 对象
    pApp->Run();
```

```
    CMyDoc* pMyDoc = new CMyDoc; // 产生CMyDoc 对象
    CMyView* pMyView = new CMyView; // 产生CMyView 对象
    CFrameWnd* pMyFrame = (CFrameWnd*)pApp->m_pMainWnd;
    ...
}
```

然后分别取其消息映射表，一路追踪上去，把每一个消息映射表中的类别记号打印出来：

```
void main()
{
...
    AFX_MSGMAP* pMessageMap = pMyView->GetMessageMap();
    cout << endl << "CMyView Message Map : " << endl;
    MsgMapPrinting(pMessageMap);

    pMessageMap = pMyDoc->GetMessageMap();
    cout << endl << "CMyDoc Message Map : " << endl;
    MsgMapPrinting(pMessageMap);

    pMessageMap = pMyFrame->GetMessageMap();
    cout << endl << "CMyFrameWnd Message Map : " << endl;
    MsgMapPrinting(pMessageMap);

    pMessageMap = pApp->GetMessageMap();
    cout << endl << "CMyWinApp Message Map : " << endl;
    MsgMapPrinting(pMessageMap);
}
```

下面这个函数追踪并打印消息映射表中的classid 记号：

```
void MsgMapPrinting(AFX_MSGMAP* pMessageMap)
{
    for(; pMessageMap != NULL; pMessageMap = pMessageMap->pBaseMessageMap)
{
        AFX_MSGMAP_ENTRY* lpEntry = pMessageMap->lpEntries;
        printlpEntries(lpEntry);
    }
}

void  printlpEntries(AFX_MSGMAP_ENTRY* lpEntry)
{
struct {
  int classid;
  char* classname;
} classinfo[] = {
                 CCmdTargetid,  "CCmdTarget",
                 CWinThreadid,  "CWinThread",
```

```
                CWinAppid,     "CWinApp",
                CMyWinAppid,   "CMyWinApp",
                CWndid,        "CWnd",
                CFrameWndid,   "CFrameWnd",
                CMyFrameWndid, "CMyFrameWnd",
                CViewid,     "CView",
                CMyViewid,   "CMyView",
                CDocumentid, "CDocument",
                CMyDocid,    "CMyDoc",
                0,           "      "
            };

  for (int i=0; classinfo[i].classid != 0; i++)
  {
     if (classinfo[i].classid == lpEntry->nID)
     {
        cout << lpEntry->nID << "    ";
        cout << classinfo[i].classname << endl;
        break;
     }
  }
}
```

Frame7 的命令列编译联结动作是（环境变量必须先设定好，请参考第 4 章的「安装与设定」一节）：

```
cl my.cpp mfc.cpp  <Enter>
```

Frame7 的执行结果是：

```
CMyView Message Map :
1221    CMyView
122    CView
12    CWnd
1    CCmdTarget

CMyDoc Message Map :
131    CMyDoc
13    CDocument
1    CCmdTarget

CMyFrameWnd Message Map :
1211    CMyFrameWnd
121    CFrameWnd
12    CWnd
```

```
1   CCmdTarget

CMyWinApp Message Map :
1111   CMyWinApp
111   CWinApp
1   CCmdTarget
```

Frame7　范例程序

MFC.H

```
#0001  #define TRUE 1
#0002  #define FALSE 0
#0003
#0004  typedef char* LPSTR;
#0005  typedef const char* LPCSTR;
#0006
#0007  typedef unsigned long  DWORD;
#0008  typedef int          BOOL;
#0009  typedef unsigned char  BYTE;
#0010  typedef unsigned short WORD;
#0011  typedef int            INT;
#0012  typedef unsigned int   UINT;
#0013  typedef long           LONG;
#0014
#0015  #define WM_COMMAND           0x0111
#0016  #define CObjectid            0xffff
#0017  #define   CCmdTargetid        1
#0018  #define    CWinThreadid       11
#0019  #define      CWinAppid       111
#0020  #define        CMyWinAppid    1111
#0021  #define    CWndid            12
#0022  #define      CFrameWndid      121
#0023  #define        CMyFrameWndid  1211
#0024  #define      CViewid         122
#0025  #define        CMyViewid      1221
#0026  #define    CDocumentid        13
#0027  #define      CMyDocid        131
#0028
#0029  #include <iostream.h>
#0030
#0031  ////////////////////////////////////////////////////////////////
#0032  // Window message map handling
#0033
```

```
#0034  struct AFX_MSGMAP_ENTRY;      // declared below after CWnd
#0035
#0036  struct AFX_MSGMAP
#0037  {
#0038         AFX_MSGMAP* pBaseMessageMap;
#0039         AFX_MSGMAP_ENTRY* lpEntries;
#0040  };
#0041
#0042  #define DECLARE_MESSAGE_MAP() \
#0043         static AFX_MSGMAP_ENTRY _messageEntries[]; \
#0044         static AFX_MSGMAP messageMap; \
#0045         virtual AFX_MSGMAP* GetMessageMap() const;
#0046
#0047  #define BEGIN_MESSAGE_MAP(theClass, baseClass) \
#0048         AFX_MSGMAP* theClass::GetMessageMap() const \
#0049                { return &theClass::messageMap; } \
#0050         AFX_MSGMAP theClass::messageMap = \
#0051         { &(baseClass::messageMap), \
#0052                (AFX_MSGMAP_ENTRY*) &(theClass::_messageEntries) }; \
#0053         AFX_MSGMAP_ENTRY theClass::_messageEntries[] = \
#0054         {
#0055
#0056  #define END_MESSAGE_MAP() \
#0057         { 0, 0, 0, 0, AfxSig_end, (AFX_PMSG)0 } \
#0058         };
#0059
#0060  // Message map signature values and macros in separate header
#0061  #include "afxmsg_.h"
#0062
#0063  class CObject
#0064  {
#0065  public:
#0066    CObject::CObject()  {
#0067                         }
#0068    CObject::~CObject() {
#0069                         }
#0070  };
#0071
#0072  class CCmdTarget : public CObject
#0073  {
#0074  public:
#0075    CCmdTarget::CCmdTarget()  {
#0076                               }
#0077    CCmdTarget::~CCmdTarget() {
#0078                               }
#0079    DECLARE_MESSAGE_MAP()      // base class - no {{ }} macros
```

```
#0080 };
#0081
#0082 typedef void (CCmdTarget::*AFX_PMSG)(void);
#0083
#0084 struct AFX_MSGMAP_ENTRY  // MFC 4.0
#0085 {
#0086    UINT nMessage; // windows message
#0087    UINT nCode;    // control code or WM_NOTIFY code
#0088    UINT nID;     // control ID (or 0 for windows messages)
#0089    UINT nLastID;  // used for entries specifying a range of control id's
#0090    UINT nSig;    // signature type (action) or pointer to message #
#0091    AFX_PMSG pfn;  // routine to call (or special value)
#0092 };
#0093
#0094 class CWinThread : public CCmdTarget
#0095 {
#0096 public:
#0097   CWinThread::CWinThread()  {
#0098                              }
#0099   CWinThread::~CWinThread() {
#0100                              }
#0101
#0102   virtual BOOL InitInstance() {
#0103                               cout << "CWinThread::InitInstance \n";
#0104                               return TRUE;
#0105                              }
#0106   virtual int Run() {
#0107                       cout << "CWinThread::Run \n";
#0108                       return 1;
#0109                     }
#0110 };
#0111
#0112 class CWnd;
#0113
#0114 class CWinApp : public CWinThread
#0115 {
#0116 public:
#0117   CWinApp* m_pCurrentWinApp;
#0118   CWnd* m_pMainWnd;
#0119
#0120 public:
#0121   CWinApp::CWinApp()  {
#0122                        m_pCurrentWinApp = this;
#0123                       }
#0124   CWinApp::~CWinApp() {
#0125                        }
```

```
#0126
#0127   virtual BOOL InitApplication() {
#0128                                cout << "CWinApp::InitApplication \n";
#0129                                return TRUE;
#0130                                   }
#0131   virtual BOOL InitInstance()  {
#0132                                  cout << "CWinApp::InitInstance \n";
#0133                                  return TRUE;
#0134                                 }
#0135   virtual int Run() {
#0136                       cout << "CWinApp::Run \n";
#0137                       return CWinThread::Run();
#0138                     }
#0139
#0140   DECLARE_MESSAGE_MAP()
#0141 };
#0142
#0143 typedef void (CWnd::*AFX_PMSGW)(void);
#0144        // like 'AFX_PMSG' but for CWnd derived classes only
#0145
#0146 class CDocument : public CCmdTarget
#0147 {
#0148 public:
#0149   CDocument::CDocument()   {
#0150                            }
#0151   CDocument::~CDocument()  {
#0152                            }
#0153   DECLARE_MESSAGE_MAP()
#0154 };
#0155
#0156 class CWnd : public CCmdTarget
#0157 {
#0158 public:
#0159   CWnd::CWnd()   {
#0160                  }
#0161   CWnd::~CWnd()  {
#0162                  }
#0163
#0164   virtual BOOL Create();
#0165   BOOL CreateEx();
#0166   virtual BOOL PreCreateWindow();
#0167
#0168   DECLARE_MESSAGE_MAP()
#0169 };
#0170
#0171 class CFrameWnd : public CWnd
```

```
#0172  {
#0173  public:
#0174    CFrameWnd::CFrameWnd()   {
#0175                             }
#0176    CFrameWnd::~CFrameWnd()  {
#0177                             }
#0178    BOOL Create();
#0179    virtual BOOL PreCreateWindow();
#0180
#0181    DECLARE_MESSAGE_MAP()
#0182  };
#0183
#0184  class CView : public CWnd
#0185  {
#0186  public:
#0187    CView::CView()   {
#0188                     }
#0189    CView::~CView()  {
#0190                     }
#0191    DECLARE_MESSAGE_MAP()
#0192  };
#0193
#0194  // global function
#0195  CWinApp* AfxGetApp();
```

AFXMSG_.H

```
#0001  enum AfxSig
#0002  {
#0003         AfxSig_end = 0,     // [marks end of message map]
#0004         AfxSig_vv,
#0005  };
#0006
#0007  #define ON_COMMAND(id, memberFxn) \
#0008      { WM_COMMAND, 0, (WORD)id, (WORD)id, AfxSig_vv, (AFX_PMSG)memberFxn },
```

MFC.CPP

```
#0001  #include "my.h"  // 原该包含 mfc.h 就好, 但为了 CMyWinApp 的定义, 所以...
#0002
#0003  extern CMyWinApp theApp;
#0004
#0005  BOOL CWnd::Create()
#0006  {
#0007    cout << "CWnd::Create \n";
```

```
#0008   return TRUE;
#0009  }
#0010
#0011  BOOL CWnd::CreateEx()
#0012  {
#0013   cout << "CWnd::CreateEx \n";
#0014   PreCreateWindow();
#0015   return TRUE;
#0016  }
#0017
#0018  BOOL CWnd::PreCreateWindow()
#0019  {
#0020   cout << "CWnd::PreCreateWindow \n";
#0021   return TRUE;
#0022  }
#0023
#0024  BOOL CFrameWnd::Create()
#0025  {
#0026   cout << "CFrameWnd::Create \n";
#0027   CreateEx();
#0028   return TRUE;
#0029  }
#0030
#0031  BOOL CFrameWnd::PreCreateWindow()
#0032  {
#0033   cout << "CFrameWnd::PreCreateWindow \n";
#0034   return TRUE;
#0035  }
#0036
#0037  CWinApp* AfxGetApp()
#0038  {
#0039   return theApp.m_pCurrentWinApp;
#0040  }
#0041
#0042  AFX_MSGMAP* CCmdTarget::GetMessageMap() const
#0043  {
#0044         return &CCmdTarget::messageMap;
#0045  }
#0046
#0047  AFX_MSGMAP CCmdTarget::messageMap =
#0048  {
#0049         NULL,
#0050         &CCmdTarget::_messageEntries[0]
#0051  };
#0052
#0053  AFX_MSGMAP_ENTRY CCmdTarget::_messageEntries[] =
```

```
#0054  {
#0055      // { 0, 0, 0, 0, AfxSig_end, 0 }    // nothing here
#0056      { 0, 0, CCmdTargetid, 0, AfxSig_end, 0 }
#0057
#0058  };
#0059
#0060  BEGIN_MESSAGE_MAP(CWnd, CCmdTarget)
#0061  ON_COMMAND(CWndid, 0)
#0062  END_MESSAGE_MAP()
#0063
#0064  BEGIN_MESSAGE_MAP(CFrameWnd, CWnd)
#0065  ON_COMMAND(CFrameWndid, 0)
#0066  END_MESSAGE_MAP()
#0067
#0068  BEGIN_MESSAGE_MAP(CDocument, CCmdTarget)
#0069  ON_COMMAND(CDocumentid, 0)
#0070  END_MESSAGE_MAP()
#0071
#0072  BEGIN_MESSAGE_MAP(CView, CWnd)
#0073  ON_COMMAND(CViewid, 0)
#0074  END_MESSAGE_MAP()
#0075
#0076  BEGIN_MESSAGE_MAP(CWinApp, CCmdTarget)
#0077  ON_COMMAND(CWinAppid, 0)
#0078  END_MESSAGE_MAP()
```

MY.H

```
#0001  #include <iostream.h>
#0002  #include "mfc.h"
#0003
#0004  class CMyWinApp : public CWinApp
#0005  {
#0006  public:
#0007    CMyWinApp::CMyWinApp()   {
#0008                             }
#0009    CMyWinApp::~CMyWinApp()  {
#0010                             }
#0011
#0012    virtual BOOL InitInstance();
#0013    DECLARE_MESSAGE_MAP()
#0014  };
#0015
#0016  class CMyFrameWnd : public CFrameWnd
#0017  {
#0018  public:
```

```
#0019   CMyFrameWnd();
#0020   ~CMyFrameWnd()  {
#0021                    }
#0022   DECLARE_MESSAGE_MAP()
#0023 };
#0024
#0025 class CMyDoc : public CDocument
#0026 {
#0027 public:
#0028   CMyDoc::CMyDoc()  {
#0029                     }
#0030   CMyDoc::~CMyDoc() {
#0031                     }
#0032   DECLARE_MESSAGE_MAP()
#0033 };
#0034
#0035 class CMyView : public CView
#0036 {
#0037 public:
#0038   CMyView::CMyView()   {
#0039                       }
#0040   CMyView::~CMyView()  {
#0041                        }
#0042   DECLARE_MESSAGE_MAP()
#0043 };
```

MY.CPP

```
#0001 #include "my.h"
#0002
#0003 CMyWinApp theApp;
#0004
#0005 BOOL CMyWinApp::InitInstance()
#0006 {
#0007    cout << "CMyWinApp::InitInstance \n";
#0008    m_pMainWnd = new CMyFrameWnd;
#0009    return TRUE;
#0010 }
#0011
#0012 CMyFrameWnd::CMyFrameWnd()
#0013 {
#0014    Create();
#0015 }
#0016
#0017 BEGIN_MESSAGE_MAP(CMyWinApp, CWinApp)
#0018 ON_COMMAND(CMyWinAppid, 0)
```

```
#0019  END_MESSAGE_MAP()
#0020
#0021  BEGIN_MESSAGE_MAP(CMyFrameWnd, CFrameWnd)
#0022  ON_COMMAND(CMyFrameWndid, 0)
#0023  END_MESSAGE_MAP()
#0024
#0025  BEGIN_MESSAGE_MAP(CMyDoc, CDocument)
#0026  ON_COMMAND(CMyDocid, 0)
#0027  END_MESSAGE_MAP()
#0028
#0029  BEGIN_MESSAGE_MAP(CMyView, CView)
#0030  ON_COMMAND(CMyViewid, 0)
#0031  END_MESSAGE_MAP()
#0032
#0033  void  printlpEntries(AFX_MSGMAP_ENTRY* lpEntry)
#0034  {
#0035  struct {
#0036    int classid;
#0037    char* classname;
#0038  } classinfo[] = {
#0039                   CCmdTargetid,  "CCmdTarget",
#0040                   CWinThreadid,  "CWinThread",
#0041                   CWinAppid,     "CWinApp",
#0042                   CMyWinAppid,    "CMyWinApp",
#0043                   CWndid,         "CWnd",
#0044                   CFrameWndid,    "CFrameWnd",
#0045                   CMyFrameWndid,  "CMyFrameWnd",
#0046                   CViewid,        "CView",
#0047                   CMyViewid,    "CMyView",
#0048                   CDocumentid,  "CDocument",
#0049                   CMyDocid,    "CMyDoc",
#0050                   0,            "     "
#0051               };
#0052
#0053    for (int i=0; classinfo[i].classid != 0; i++)
#0054    {
#0055       if (classinfo[i].classid == lpEntry->nID)
#0056       {
#0057          cout << lpEntry->nID << "    ";
#0058          cout << classinfo[i].classname << endl;
#0059          break;
#0060       }
#0061    }
#0062  }
#0063
#0064  void MsgMapPrinting(AFX_MSGMAP* pMessageMap)
```

```
#0065 {
#0066     for(; pMessageMap != NULL; pMessageMap = pMessageMap->pBaseMessageMap) {
#0067         AFX_MSGMAP_ENTRY* lpEntry = pMessageMap->lpEntries;
#0068         printlpEntries(lpEntry);
#0069     }
#0070 }
#0071
#0072 //--------------------------------------------------------------
#0073 // main
#0074 //--------------------------------------------------------------
#0075 void main()
#0076 {
#0077
#0078     CWinApp* pApp = AfxGetApp();
#0079
#0080     pApp->InitApplication();
#0081     pApp->InitInstance();
#0082     pApp->Run();
#0083
#0084     CMyDoc* pMyDoc = new CMyDoc;
#0085     CMyView* pMyView = new CMyView;
#0086     CFrameWnd* pMyFrame = (CFrameWnd*)pApp->m_pMainWnd;
#0087
#0088     // output Message Map construction
#0089     AFX_MSGMAP* pMessageMap = pMyView->GetMessageMap();
#0090     cout << endl << "CMyView Message Map : " << endl;
#0091     MsgMapPrinting(pMessageMap);
#0092
#0093     pMessageMap = pMyDoc->GetMessageMap();
#0094     cout << endl << "CMyDoc Message Map : " << endl;
#0095     MsgMapPrinting(pMessageMap);
#0096
#0097     pMessageMap = pMyFrame->GetMessageMap();
#0098     cout << endl << "CMyFrameWnd Message Map : " << endl;
#0099     MsgMapPrinting(pMessageMap);
#0100
#0101     pMessageMap = pApp->GetMessageMap();
#0102     cout << endl << "CMyWinApp Message Map : " << endl;
#0103     MsgMapPrinting(pMessageMap);
#0104 }
```

Command Routing（命令绕行）

我们已经在上一节把整个消息流动网架设起来了。当消息进来，会有一个邦浦推动它前进。消息如何进来，以及邦浦函数如何推动，都是属于Windows 程序设计的范畴，暂时不管。我现在要仿真出消息的流动绕行路线-- 我常喜欢称之为消息的「二万五千里长征」。

消息如果是从子类别流向父类别（纵向流动），那么事情再简单不过，整个Message Map 消息映射表已规划出十分明确的路线。但是正如上一节一开始我说的，MFC 之中用来处理消息的C++ 类别并不呈单鞭发展，作为application framework 的重要架构之一的 document/view，也具有处理消息的能力（你现在可能还不清楚什么是document/view，没有关系）；因此，消息应该有横向流动的机会。MFC 对于消息绕行的规定是：

- 如果是一般的Windows 消息（*WM_xxx*），一定是由衍生类别流向基础类别，没有旁流的可能。
- 如果是命令消息*WM_COMMAND*，就有奇特的路线了：

命令消息接收物的类型	处理次序
Frame 窗口	1. View 2. Frame 窗口本身 3. CWinApp 对象
View	1. View 本身 2. Document
Document	1. Document 本身 2. Document Template ❖

❖ 目前我们还不知道什么是 Document Template，但是没有关系

❖ 第 9 章将解开虚线跳跃之谜

图3-6 MFC 对于命令消息WM_COMMAND 的特殊处理顺序。

不管这个规则是怎么定下来的，现在我要设计一个推动引擎，把它仿真出来。以下这些函数名称以及函数内容，完全仿真MFC 内部。有些函数似乎赘余，那是因为我删掉了许多主题以外的动作。不把看似赘余的函数拿掉或合并，是为了留下MFC 的足迹。此外，为了追踪调用过程（call stack），我在各函数的第一行输出一串识别文字。

首先我把新增加的一些成员函数做个列表：

类别	与消息绕行有关的成员函数	注意
none	*AfxWndProc*	global
none	*AfxCallWndProc*	global
CCmdTarget	*OnCmdMsg*	virtual
CDocument	*OnCmdMsg*	virtual
CWnd	*WindowProc*	virtual
	OnCommand	virtual
	DefWindowProc	virtual
CFrameWnd	*OnCommand*	virtual
	OnCmdMsg	virtual
CView	*OnCmdMsg*	virtual

全域函数*AfxWndProc* 就是我所谓的推动引擎的起始点。它本来应该是在*CWinThread*::*Run* 中被调用，但为了实验目的，我在*main* 中调用它，每调用一次便推送一个消息。这个函数在MFC 中有四个参数，为了方便，我加上第五个，用以表示是谁获得消息（成为绕行的起点）。例如：

```
AfxWndProc(0, WM_CREATE, 0, 0, pMyFrame);
```

表示*pMyFrame* 获得了一个*WM_CREATE*，而：

```
AfxWndProc(0, WM_COMMAND, 0, 0, pMyView);
```

表示*pMyView* 获得了一个*WM_COMMAND*。

下面是消息流动的过程：

```
LRESULT AfxWndProc(HWND hWnd, UINT nMsg, WPARAM wParam, LPARAM lParam,
                CWnd *pWnd)  // last param. pWnd is added by JJHou.
{
  cout << "AfxWndProc()" << endl;
  return AfxCallWndProc(pWnd, hWnd, nMsg, wParam, lParam);
}

LRESULT AfxCallWndProc(CWnd* pWnd, HWND hWnd, UINT nMsg,
                    WPARAM wParam, LPARAM lParam)
{
  cout << "AfxCallWndProc()" << endl;
  LRESULT lResult = pWnd->WindowProc(nMsg, wParam, lParam);
  return lResult;
}
```

pWnd->WindowProc 究竟是调用哪一个函数？不一定，得视*pWnd* 到底指向何种类别之对象而定-- 别忘了*WindowProc* 是虚拟函数。这正是虚拟函数发挥它功效的地方呀：

- 如果*pWnd* 指向*CMyFrameWnd* 对象，那么调用的是*CFrameWnd*::*WindowProc*。
  而因为*CFrameWnd* 并没有改写*WindowProc* ， 所以调用的其实是
  CWnd::*WindowProc*。
- 如果*pWnd* 指向*CMyView* 对象，那么调用的是*CView*::*WindowProc*。而因为*CView* 并没有改写*WindowProc*，所以调用的其实是*CWnd*::*WindowProc*。
  虽然殊途同归，意义上是不相同的。切记！切记！

CWnd::*WindowProc* 首先判断消息是否为*WM_COMMAND*。如果不是，事情最单纯，就把消息往父类别推去，父类别再往祖父类别推去。每到一个类别的消息映射表，原本应该比对*AFX_MSGMAP_ENTRY* 的每一个元素，比对成功就调用对应的处理例程。不过在这里我不作比对，只是把*AFX_MSGMAP_ENTRY* 中的类别识别代码印出来（就像上一节的Frame7 程序一样），以表示「到此一游」：

```
LRESULT CWnd::WindowProc(UINT nMsg, WPARAM wParam, LPARAM lParam)
{
    AFX_MSGMAP* pMessageMap;
    AFX_MSGMAP_ENTRY* lpEntry;

    if (nMsg == WM_COMMAND) // special case for commands
    {
        if (OnCommand(wParam, lParam))   ❶
            return 1L; // command handled
        else
            return (LRESULT)DefWindowProc(nMsg, wParam, lParam);  ❾
    }

    pMessageMap = GetMessageMap();

    for (; pMessageMap != NULL;
         pMessageMap = pMessageMap->pBaseMessageMap)
    {
            lpEntry = pMessageMap->lpEntries;
            printlpEntries(lpEntry);
    }
    return 0;  // J.J.Hou: if find, should call lpEntry->pfn,
               //  otherwise should call DefWindowProc.
               // for simplification, we just return 0.
}
```

如果消息是*WM_COMMAND*，*CWnd*::*WindowProc* 调用 ❶ *OnCommand*。好，注意了，这又是一个*CWnd* 的虚拟函数：

1. 如果*this* 指向*CMyFrameWnd* 对象，那么调用的是*CFrameWnd*::*OnCommand*。
2. 如果*this* 指向*CMyView* 对象，那么调用的是*CView*::*OnCommand*。而因为*CView* 并没有改写*OnCommand*，所以调用的其实是*CWnd*::*OnCommand*。

这次可就没有殊途同归了。

我们以第一种情况为例，再往下看：

```
BOOL CFrameWnd::OnCommand(WPARAM wParam, LPARAM lParam)
{
    cout << "CFrameWnd::OnCommand()" << endl;
    // ...
    // route as normal command
    return CWnd::OnCommand(wParam, lParam);  ❷
}

BOOL CWnd::OnCommand(WPARAM wParam, LPARAM lParam)
{
    cout << "CWnd::OnCommand()" << endl;
    // ...
    return OnCmdMsg(0, 0);  ❸
}
```

又一次遭遇虛拟函数。经过前两次的分析，相信你对此很有经验了。❸ *OnCmdMsg* 是 *CCmdTarget* 的虛拟函数，所以：

1. 如果*this* 指向*CMyFrameWnd* 对象，那么调用的是*CFrameWnd*::*OnCmdMsg*。
2. 如果*this* 指向*CMyView* 对象，那么调用的是*CView*::*OnCmdMsg*。
3. 如果*this* 指向*CMyDoc* 对象，那么调用的是*CDocument*::*OnCmdMsg*。
4. 如果*this* 指向*CMyWinApp* 对象，那么调用的是*CWinApp*::*OnCmdMsg*。而因为 *CWinApp* 并没有改写*OnCmdMsg*，所以调用的其实是*CCmdTarget*::*OnCmdMsg*。

目前的情况是第一种，于是调用CFrameWnd::OnCmdMsg：

```
BOOL CFrameWnd::OnCmdMsg(UINT nID, int nCode)
{
    cout << "CFrameWnd::OnCmdMsg()" << endl;
    // pump through current view FIRST
    CView* pView = GetActiveView();
    if (pView->OnCmdMsg(nID, nCode))  ❹
           return TRUE;

    // then pump through frame
    if (CWnd::OnCmdMsg(nID, nCode))  ❼
           return TRUE;

    // last but not least, pump through app
    CWinApp* pApp = AfxGetApp();
```

```
    if (pApp->OnCmdMsg(nID, nCode))  ❽
           return TRUE;

    return FALSE;
}
```

这个函数反应出图3-6 Frame 窗口处理*WM_COMMAND* 的次序。最先调用的是

❹ *pView->OnCmdMsg*，于是：

```
BOOL CView::OnCmdMsg(UINT nID, int nCode)
{
    cout << "CView::OnCmdMsg()" << endl;
    if (CWnd::OnCmdMsg(nID, nCode))  ❺
       return TRUE;

    BOOL bHandled = FALSE;
    bHandled = m_pDocument->OnCmdMsg(nID, nCode);  ❻
    return bHandled;
}
```

这又反应出图3-6 View 窗口处理*WM_COMMAND* 的次序。最先调用的是

❺ *CWnd*::*OnCmdMsg* ， 而*CWnd* 并未改写*OnCmdMsg* ， 所以其实就是调用

CCmdTarget::*OnCmdMsg*：

```
{
BOOL CCmdTarget::OnCmdMsg(UINT nID, int nCode)
{
    cout << "CCmdTarget::OnCmdMsg()" << endl;
    // Now look through message map to see if it applies to us
    AFX_MSGMAP* pMessageMap;
    AFX_MSGMAP_ENTRY* lpEntry;
    for (pMessageMap = GetMessageMap(); pMessageMap != NULL;
        pMessageMap = pMessageMap->pBaseMessageMap)
    {
           lpEntry = pMessageMap->lpEntries;
           printlpEntries(lpEntry);
    }

     return FALSE; // not handled
}
```

这是一个走访消息映射表的动作。注意，*GetMessageMap* 也是个虚拟函数（隐藏在 *DECLARE_MESSAGE_MAP* 宏定义中），所以它所得到的消息映射表将是*this*（以目前而言是*pMyView*）所指对象的映射表。于是我们得到了这个结果：

```
pMyFrame received a WM_COMMAND, routing path and call stack:
AfxWndProc()
AfxCallWndProc()
CWnd::WindowProc()
CFrameWnd::OnCommand()
CWnd::OnCommand()
CFrameWnd::OnCmdMsg()
CFrameWnd::GetActiveView()
CView::OnCmdMsg()
CCmdTarget::OnCmdMsg()
1221    CMyView
122    CView
12    CWnd
1    CCmdTarget
```

如果在映射表中找到了对应的消息，就调用对应的处理例程，然后也就结束了二万五千里长征。如果没找到，长征还没有结束，这时候退守回到*CView*::*OnCmdMsg*，调用❻*CDocument*::*OnCmdMsg*：

```
BOOL CDocument::OnCmdMsg(UINT nID, int nCode)
{
    cout << "CDocument::OnCmdMsg()" << endl;
    if (CCmdTarget::OnCmdMsg(nID, nCode))
        return TRUE;

    return FALSE;
}
```

于是得到这个结果：

```
CDocument::OnCmdMsg()
CCmdTarget::OnCmdMsg()
131    CMyDoc
13    CDocument
1    CCmdTarget
```

如果在映射表中还是没找到对应消息，二万五千里长征还是未能结束，这时候退守回到*CFrameWnd*::*OnCmdMsg*，调用❼*CWnd*::*OnCmdMsg*（也就是*CCmdTarget*::*OnCmdMsg*），得到这个结果：

```
CCmdTarget::OnCmdMsg()
1211    CMyFrameWnd
121    CFrameWnd
12    CWnd
1    CCmdTarget
```

如果在映射表中还是没找到对应消息，二万五千里长征还是未能结束，再退回到*CFrameWnd*::*OnCmdMsg*，调用❽*CWinApp*::*OnCmdMsg*（亦即*CCmdTarget*::*OnCmdMsg*），得到这个结果：

```
1111    CMyWinApp
111    CWinApp
1    CCmdTarget
```

万一还是没找到对应的消息，二万五千里长征可也穷途末路了，退回到*CWnd::WindowProc*，调用❾*CWnd*::*DefWindowProc*。你可以想象，在真正的MFC 中这个成员函数必是调用Windows API 函数::*DefWindowProc*。为了简化，我让它在Frame8 中是个空函数。

故事结束！

我以图3-7 表示这二万五千里长征的调用次序（call stack），图3-8 表示这二万五千里长征的消息流动路线。

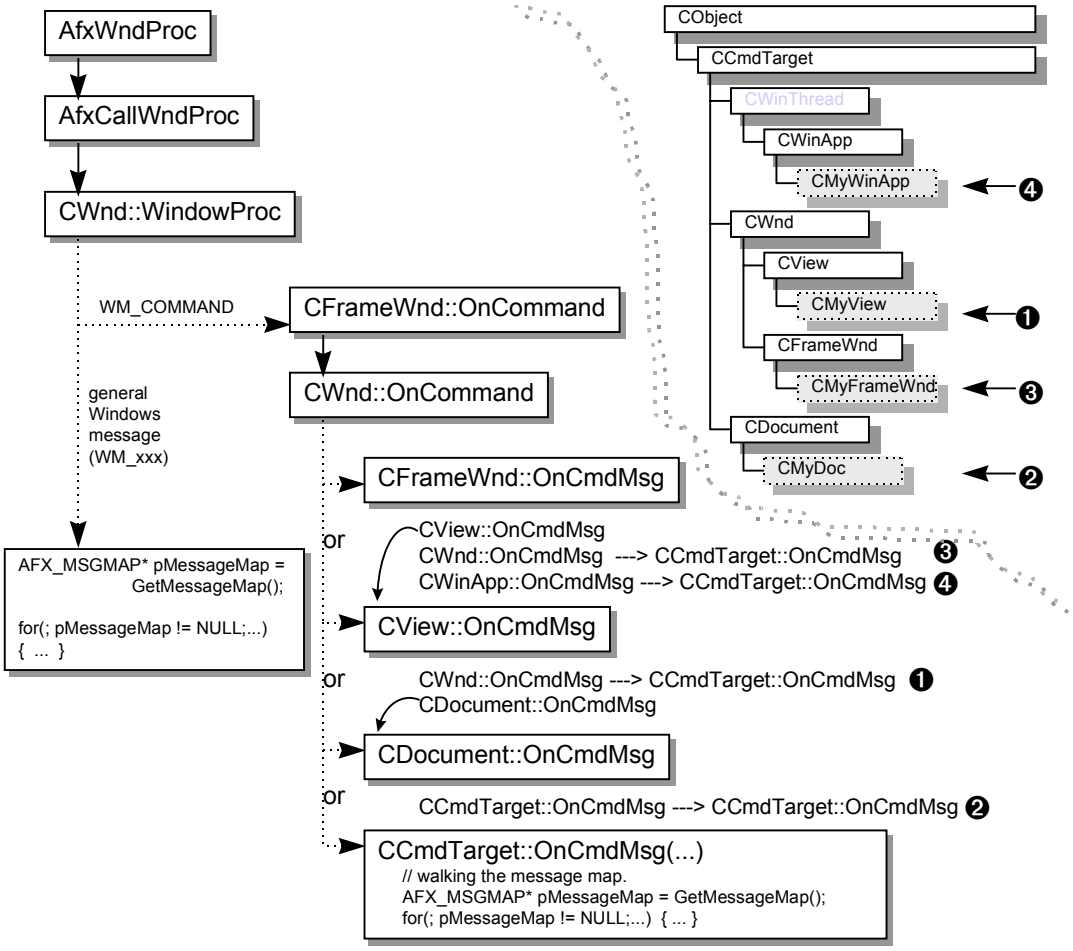


图3-7 当CMyFrameWnd 对象获得一个WM_COMMAND，所引起的Frame8 函数调用次序。

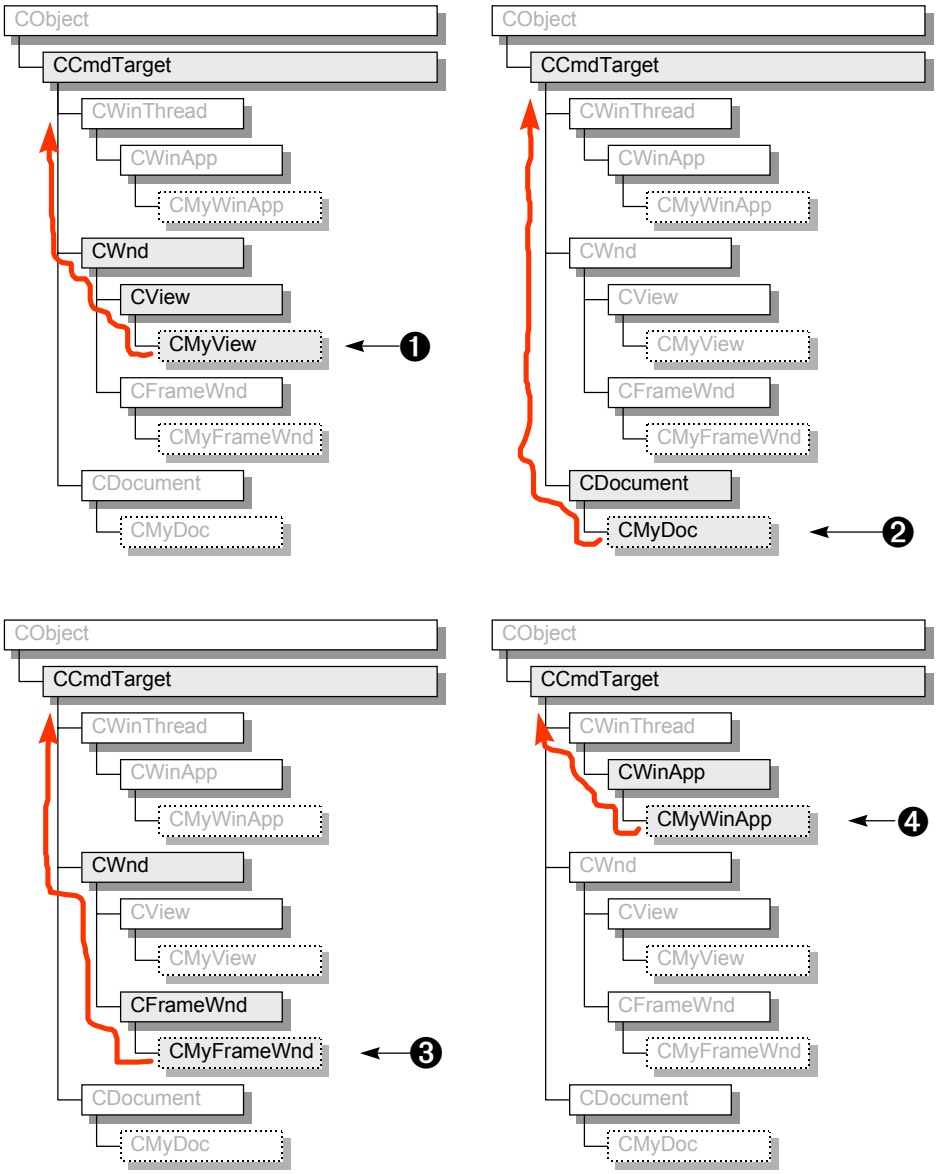


图3-8 当CMyFrameWnd 对象获得一个WM_COMMAND，所引起的消息流动路线。

Frame8 测试四种情况：分别从frame 对象和view 对象中推动消息，消息分一般 Windows 消息和*WM_COMMAND* 两种：

```
// test Message Routing
AfxWndProc(0, WM_CREATE, 0, 0, pMyFrame);
AfxWndProc(0, WM_PAINT, 0, 0, pMyView);
AfxWndProc(0, WM_COMMAND, 0, 0, pMyView);
AfxWndProc(0, WM_COMMAND, 0, 0, pMyFrame);
```

Frame8 的命令列编译联结动作是（环境变量必须先设定好，请参考第 4 章的「安装与设定」一节）：

```
cl my.cpp mfc.cpp  <Enter>
```

以下是Frame8 的执行结果：

```
CWinApp::InitApplication
CMyWinApp::InitInstance
CFrameWnd::Create
CWnd::CreateEx
CFrameWnd::PreCreateWindow
CWinApp::Run
CWinThread::Run

pMyFrame received a WM_CREATE, routing path and call stack:
AfxWndProc()
AfxCallWndProc()
CWnd::WindowProc()
1211    CMyFrameWnd
121    CFrameWnd
12    CWnd
1    CCmdTarget

pMyView received a WM_PAINT, routing path and call stack:
AfxWndProc()
AfxCallWndProc()
CWnd::WindowProc()
1221    CMyView
122    CView
12    CWnd
1    CCmdTarget

pMyView received a WM_COMMAND, routing path and call stack:
AfxWndProc()
```

```
AfxCallWndProc()
CWnd::WindowProc()
CWnd::OnCommand()
CView::OnCmdMsg()
CCmdTarget::OnCmdMsg()
1221   CMyView
122   CView
12   CWnd
1   CCmdTarget
CDocument::OnCmdMsg()
CCmdTarget::OnCmdMsg()
131   CMyDoc
13   CDocument
1   CCmdTarget
CWnd::DefWindowProc()

pMyFrame received a WM_COMMAND, routing path and call stack:
AfxWndProc()
AfxCallWndProc()
CWnd::WindowProc()
CFrameWnd::OnCommand()
CWnd::OnCommand()
CFrameWnd::OnCmdMsg()
CFrameWnd::GetActiveView()
CView::OnCmdMsg()
CCmdTarget::OnCmdMsg()
1221   CMyView
122   CView
12   CWnd
1   CCmdTarget
CDocument::OnCmdMsg()
CCmdTarget::OnCmdMsg()
131   CMyDoc
13   CDocument
1   CCmdTarget
CCmdTarget::OnCmdMsg()
1211   CMyFrameWnd
121   CFrameWnd
12   CWnd
1   CCmdTarget
CCmdTarget::OnCmdMsg()
1111   CMyWinApp
111   CWinApp
1   CCmdTarget
CWnd::DefWindowProc()
```

Frame8 范例程序

MFC.H

```
#0001  #define TRUE 1
#0002  #define FALSE 0
#0003
#0004  typedef char* LPSTR;
#0005  typedef const char* LPCSTR;
#0006
#0007  typedef unsigned long  DWORD;
#0008  typedef int            BOOL;
#0009  typedef unsigned char  BYTE;
#0010  typedef unsigned short WORD;
#0011  typedef int            INT;
#0012  typedef unsigned int   UINT;
#0013  typedef long           LONG;
#0014
#0015  typedef UINT         WPARAM;
#0016  typedef LONG         LPARAM;
#0017  typedef LONG         LRESULT;
#0018  typedef int          HWND;
#0019
#0020  #define WM_COMMAND           0x0111
#0021  #define WM_CREATE            0x0001
#0022  #define WM_PAINT             0x000F
#0023  #define WM_NOTIFY            0x004E
#0024
#0025  #define CObjectid            0xffff
#0026  #define  CCmdTargetid        1
#0027  #define    CWinThreadid      11
#0028  #define      CWinAppid       111
#0029  #define        CMyWinAppid   1111
#0030  #define    CWndid           12
#0031  #define      CFrameWndid     121
#0032  #define        CMyFrameWndid 1211
#0033  #define      CViewid         122
#0034  #define        CMyViewid     1221
#0035  #define    CDocumentid       13
#0036  #define      CMyDocid        131
#0037
#0038  #include <iostream.h>
#0039
#0040  ////////////////////////////////////////////////////////////////
```

```
#0041  // Window message map handling
#0042
#0043  struct AFX_MSGMAP_ENTRY;      // declared below after CWnd
#0044
#0045  struct AFX_MSGMAP
#0046  {
#0047         AFX_MSGMAP* pBaseMessageMap;
#0048         AFX_MSGMAP_ENTRY* lpEntries;
#0049  };
#0050
#0051  #define DECLARE_MESSAGE_MAP() \
#0052         static AFX_MSGMAP_ENTRY _messageEntries[]; \
#0053         static AFX_MSGMAP messageMap; \
#0054         virtual AFX_MSGMAP* GetMessageMap() const;
#0055
#0056  #define BEGIN_MESSAGE_MAP(theClass, baseClass) \
#0057         AFX_MSGMAP* theClass::GetMessageMap() const \
#0058                { return &theClass::messageMap; } \
#0059         AFX_MSGMAP theClass::messageMap = \
#0060         { &(baseClass::messageMap), \
#0061                (AFX_MSGMAP_ENTRY*) &(theClass::_messageEntries) }; \
#0062         AFX_MSGMAP_ENTRY theClass::_messageEntries[] = \
#0063         {
#0064
#0065  #define END_MESSAGE_MAP() \
#0066         { 0, 0, 0, 0, AfxSig_end, (AFX_PMSG)0 } \
#0067         };
#0068
#0069  // Message map signature values and macros in separate header
#0070  #include "afxmsg_.h"
#0071
#0072  class CObject
#0073  {
#0074  public:
#0075    CObject::CObject()  {
#0076                        }
#0077    CObject::~CObject() {
#0078                        }
#0079  };
#0080
#0081  class CCmdTarget : public CObject
#0082  {
#0083  public:
#0084    CCmdTarget::CCmdTarget()  {
#0085                               }
#0086    CCmdTarget::~CCmdTarget() {
```

```
#0087                              }
#0088
#0089   virtual BOOL OnCmdMsg(UINT nID, int nCode);
#0090
#0091   DECLARE_MESSAGE_MAP()      // base class - no {{ }} macros
#0092 };
#0093
#0094 typedef void (CCmdTarget::*AFX_PMSG)(void);
#0095
#0096 struct AFX_MSGMAP_ENTRY  // MFC 4.0
#0097 {
#0098    UINT nMessage; // windows message
#0099    UINT nCode;    // control code or WM_NOTIFY code
#0100    UINT nID;     // control ID (or 0 for windows messages)
#0101    UINT nLastID;  // used for entries specifying a range of control id's
#0102    UINT nSig;    // signature type (action) or pointer to message #
#0103    AFX_PMSG pfn;  // routine to call (or special value)
#0104 };
#0105
#0106 class CWinThread : public CCmdTarget
#0107 {
#0108 public:
#0109   CWinThread::CWinThread()  {
#0110                              }
#0111   CWinThread::~CWinThread() {
#0112                              }
#0113
#0114   virtual BOOL InitInstance() {
#0115                               cout << "CWinThread::InitInstance \n";
#0116                               return TRUE;
#0117                              }
#0118   virtual int Run() {
#0119                      cout << "CWinThread::Run \n";
#0120                      // AfxWndProc(...);
#0121                      return 1;
#0122                    }
#0123 };
#0124
#0125 class CWnd;
#0126
#0127 class CWinApp : public CWinThread
#0128 {
#0129 public:
#0130   CWinApp* m_pCurrentWinApp;
#0131   CWnd* m_pMainWnd;
#0132
```

```
#0133  public:
#0134    CWinApp::CWinApp()  {
#0135                          m_pCurrentWinApp = this;
#0136                        }
#0137    CWinApp::~CWinApp() {
#0138                        }
#0139
#0140    virtual BOOL InitApplication() {
#0141                                    cout << "CWinApp::InitApplication \n";
#0142                                    return TRUE;
#0143                                     }
#0144    virtual BOOL InitInstance()   {
#0145                                     cout << "CWinApp::InitInstance \n";
#0146                                     return TRUE;
#0147                                   }
#0148    virtual int Run() {
#0149                         cout << "CWinApp::Run \n";
#0150                         return CWinThread::Run();
#0151                       }
#0152
#0153    DECLARE_MESSAGE_MAP()
#0154  };
#0155
#0156  typedef void (CWnd::*AFX_PMSGW)(void);
#0157                // like 'AFX_PMSG' but for CWnd derived classes only
#0158
#0159  class CDocument : public CCmdTarget
#0160  {
#0161  public:
#0162    CDocument::CDocument()   {
#0163                             }
#0164    CDocument::~CDocument()  {
#0165                             }
#0166
#0167    virtual BOOL OnCmdMsg(UINT nID, int nCode);
#0168
#0169    DECLARE_MESSAGE_MAP()
#0170  };
#0171
#0172  class CWnd : public CCmdTarget
#0173  {
#0174  public:
#0175    CWnd::CWnd()   {
#0176                    }
#0177    CWnd::~CWnd()  {
#0178                    }
```

```
#0179
#0180   virtual BOOL Create();
#0181   BOOL CreateEx();
#0182   virtual BOOL PreCreateWindow();
#0183   virtual LRESULT WindowProc(UINT nMsg, WPARAM wParam, LPARAM lParam);
#0184   virtual LRESULT DefWindowProc(UINT message, WPARAM wParam, LPARAM lParam);
#0185   virtual BOOL OnCommand(WPARAM wParam, LPARAM lParam);
#0186
#0187   DECLARE_MESSAGE_MAP()
#0188 };
#0189
#0190 class CView;
#0191
#0192 class CFrameWnd : public CWnd
#0193 {
#0194 public:
#0195   CView* m_pViewActive;      // current active view
#0196
#0197 public:
#0198   CFrameWnd::CFrameWnd()   {
#0199                            }
#0200   CFrameWnd::~CFrameWnd()  {
#0201                            }
#0202   BOOL Create();
#0203   CView* GetActiveView() const;
#0204   virtual BOOL PreCreateWindow();
#0205   virtual BOOL OnCommand(WPARAM wParam, LPARAM lParam);
#0206   virtual BOOL OnCmdMsg(UINT nID, int nCode);
#0207
#0208   DECLARE_MESSAGE_MAP()
#0209
#0210   friend CView;
#0211 };
#0212
#0213 class CView : public CWnd
#0214 {
#0215 public:
#0216   CDocument* m_pDocument;
#0217
#0218 public:
#0219   CView::CView()   {
#0220                    }
#0221   CView::~CView()  {
#0222                    }
#0223
#0224   virtual BOOL OnCmdMsg(UINT nID, int nCode);
```

```
#0225
#0226   DECLARE_MESSAGE_MAP()
#0227
#0228   friend CFrameWnd;
#0229 };
#0230
#0231 // global function
#0232 CWinApp* AfxGetApp();
#0233 LRESULT AfxWndProc(HWND hWnd, UINT nMsg, WPARAM wParam, LPARAM lParam,
#0234                    CWnd* pWnd); // last param. pWnd is added by JJHOU.
#0235 LRESULT AfxCallWndProc(CWnd* pWnd, HWND hWnd, UINT nMsg, WPARAM wParam,
#0236                        LPARAM lParam);
```

AFXMSG_.H

```
#0001 enum AfxSig
#0002 {
#0003        AfxSig_end = 0,     // [marks end of message map]
#0004        AfxSig_vv,
#0005 };
#0006
#0007 #define ON_COMMAND(id, memberFxn) \
#0008    { WM_COMMAND, 0, (WORD)id, (WORD)id, AfxSig_vv, (AFX_PMSG)memberFxn },
```

MFC.CPP

```
#0001 #include "my.h" // 原该包含mfc.h 就好, 但为了extern CMyWinApp 所以...
#0002
#0003 extern CMyWinApp theApp;
#0004 extern void printlpEntries(AFX_MSGMAP_ENTRY* lpEntry);
#0005
#0006 BOOL CCmdTarget::OnCmdMsg(UINT nID, int nCode)
#0007 {
#0008     // Now look through message map to see if it applies to us
#0009     AFX_MSGMAP* pMessageMap;
#0010     AFX_MSGMAP_ENTRY* lpEntry;
#0011     for (pMessageMap = GetMessageMap(); pMessageMap != NULL;
#0012          pMessageMap = pMessageMap->pBaseMessageMap)
#0013     {
#0014            lpEntry = pMessageMap->lpEntries;
#0015            printlpEntries(lpEntry);
#0016     }
#0017
#0018     return FALSE;   // not handled
#0019 }
```

```
#0020
#0021  BOOL CWnd::Create()
#0022  {
#0023      cout << "CWnd::Create \n";
#0024      return TRUE;
#0025  }
#0026
#0027  BOOL CWnd::CreateEx()
#0028  {
#0029      cout << "CWnd::CreateEx \n";
#0030      PreCreateWindow();
#0031      return TRUE;
#0032  }
#0033
#0034  BOOL CWnd::PreCreateWindow()
#0035  {
#0036    cout << "CWnd::PreCreateWindow \n";
#0037    return TRUE;
#0038  }
#0039
#0040  LRESULT CWnd::WindowProc(UINT nMsg, WPARAM wParam, LPARAM lParam)
#0041  {
#0042      AFX_MSGMAP* pMessageMap;
#0043      AFX_MSGMAP_ENTRY* lpEntry;
#0044
#0045      if (nMsg == WM_COMMAND) // special case for commands
#0046      {
#0047          if (OnCommand(wParam, lParam))
#0048              return 1L; // command handled
#0049          else
#0050              return (LRESULT)DefWindowProc(nMsg, wParam, lParam);
#0051      }
#0052
#0053      pMessageMap = GetMessageMap();
#0054
#0055      for (; pMessageMap != NULL;
#0056           pMessageMap = pMessageMap->pBaseMessageMap)
#0057      {
#0058              lpEntry = pMessageMap->lpEntries;
#0059              printlpEntries(lpEntry);
#0060      }
#0061      return 0;  // add by JJHou. if find, should call lpEntry->pfn,
#0062                 //            otherwise should call DefWindowProc.
#0063  }
#0064
#0065  LRESULT CWnd::DefWindowProc(UINT message, WPARAM wParam, LPARAM lParam)
```

```
#0066  {
#0067      return TRUE;
#0068  }
#0069
#0070  BOOL CWnd::OnCommand(WPARAM wParam, LPARAM lParam)
#0071  {
#0072      // ...
#0073      return OnCmdMsg(0, 0);
#0074  }
#0075
#0076  BOOL CFrameWnd::OnCommand(WPARAM wParam, LPARAM lParam)
#0077  {
#0078      // ...
#0079      // route as normal command
#0080      return CWnd::OnCommand(wParam, lParam);
#0081  }
#0082
#0083  BOOL CFrameWnd::Create()
#0084  {
#0085      cout << "CFrameWnd::Create \n";
#0086      CreateEx();
#0087      return TRUE;
#0088  }
#0089
#0090  BOOL CFrameWnd::PreCreateWindow()
#0091  {
#0092      cout << "CFrameWnd::PreCreateWindow \n";
#0093      return TRUE;
#0094  }
#0095
#0096  CView* CFrameWnd::GetActiveView() const
#0097  {
#0098      return m_pViewActive;
#0099  }
#0100
#0101  BOOL CFrameWnd::OnCmdMsg(UINT nID, int nCode)
#0102  {
#0103      // pump through current view FIRST
#0104      CView* pView = GetActiveView();
#0105      if (pView->OnCmdMsg(nID, nCode))
#0106            return TRUE;
#0107
#0108      // then pump through frame
#0109      if (CWnd::OnCmdMsg(nID, nCode))
#0110            return TRUE;
#0111
```

```
#0112     // last but not least, pump through app
#0113     CWinApp* pApp = AfxGetApp();
#0114     if (pApp->OnCmdMsg(nID, nCode))
#0115            return TRUE;
#0116
#0117     return FALSE;
#0118  }
#0119
#0120  BOOL CDocument::OnCmdMsg(UINT nID, int nCode)
#0121  {
#0122     if (CCmdTarget::OnCmdMsg(nID, nCode))
#0123        return TRUE;
#0124
#0125     return FALSE;
#0126  }
#0127
#0128  BOOL CView::OnCmdMsg(UINT nID, int nCode)
#0129  {
#0130     if (CWnd::OnCmdMsg(nID, nCode))
#0131        return TRUE;
#0132
#0133     BOOL bHandled = FALSE;
#0134     bHandled = m_pDocument->OnCmdMsg(nID, nCode);
#0135     return bHandled;
#0136  }
#0137
#0138  AFX_MSGMAP* CCmdTarget::GetMessageMap() const
#0139  {
#0140     return &CCmdTarget::messageMap;
#0141  }
#0142
#0143  AFX_MSGMAP CCmdTarget::messageMap =
#0144  {
#0145     NULL,
#0146     &CCmdTarget::_messageEntries[0]
#0147  };
#0148
#0149  AFX_MSGMAP_ENTRY CCmdTarget::_messageEntries[] =
#0150  {
#0151
#0152        { 0, 0, CCmdTargetid, 0, AfxSig_end, 0 }
#0153  };
#0154
#0155  BEGIN_MESSAGE_MAP(CWnd, CCmdTarget)
#0156  ON_COMMAND(CWndid, 0)
#0157  END_MESSAGE_MAP()
```

```
#0158
#0159  BEGIN_MESSAGE_MAP(CFrameWnd, CWnd)
#0160  ON_COMMAND(CFrameWndid, 0)
#0161  END_MESSAGE_MAP()
#0162
#0163  BEGIN_MESSAGE_MAP(CDocument, CCmdTarget)
#0164  ON_COMMAND(CDocumentid, 0)
#0165  END_MESSAGE_MAP()
#0166
#0167  BEGIN_MESSAGE_MAP(CView, CWnd)
#0168  ON_COMMAND(CViewid, 0)
#0169  END_MESSAGE_MAP()
#0170
#0171  BEGIN_MESSAGE_MAP(CWinApp, CCmdTarget)
#0172  ON_COMMAND(CWinAppid, 0)
#0173  END_MESSAGE_MAP()
#0174
#0175  CWinApp* AfxGetApp()
#0176  {
#0177    return theApp.m_pCurrentWinApp;
#0178  }
#0179
#0180  LRESULT AfxWndProc(HWND hWnd, UINT nMsg, WPARAM wParam, LPARAM lParam,
#0181                  CWnd *pWnd)  // last parameter pWnd is added by JJHou.
#0182  {
#0183    //...
#0184    return AfxCallWndProc(pWnd, hWnd, nMsg, wParam, lParam);
#0185  }
#0186
#0187  LRESULT AfxCallWndProc(CWnd* pWnd, HWND hWnd, UINT nMsg,
                            WPARAM wParam, LPARAM lParam)
#0188  {
#0189    LRESULT lResult = pWnd->WindowProc(nMsg, wParam, lParam);
#0190    return lResult;
#0191  }
```

MY.H

```
#0001  #include <iostream.h>
#0002  #include "mfc.h"
#0003
#0004  class CMyWinApp : public CWinApp
#0005  {
#0006  public:
#0007    CMyWinApp::CMyWinApp()   {
#0008                            }
```

```
#0009   CMyWinApp::~CMyWinApp()  {
#0010                          }
#0011   virtual BOOL InitInstance();
#0012   DECLARE_MESSAGE_MAP()
#0013 };
#0014
#0015 class CMyFrameWnd : public CFrameWnd
#0016 {
#0017 public:
#0018   CMyFrameWnd();
#0019   ~CMyFrameWnd()  {
#0020                 }
#0021   DECLARE_MESSAGE_MAP()
#0022 };
#0023
#0024 class CMyDoc : public CDocument
#0025 {
#0026 public:
#0027   CMyDoc::CMyDoc()  {
#0028                     }
#0029   CMyDoc::~CMyDoc() {
#0030                     }
#0031   DECLARE_MESSAGE_MAP()
#0032 };
#0033
#0034 class CMyView : public CView
#0035 {
#0036 public:
#0037   CMyView::CMyView()   {
#0038                        }
#0039   CMyView::~CMyView()  {
#0040                        }
#0041   DECLARE_MESSAGE_MAP()
#0042 };
```

MY.CPP

```
#0001 #include "my.h"
#0002
#0003 CMyWinApp theApp;  // global object
#0004
#0005 BOOL CMyWinApp::InitInstance()
#0006 {
#0007     cout << "CMyWinApp::InitInstance \n";
#0008     m_pMainWnd = new CMyFrameWnd;
#0009     return TRUE;
```

```
#0010  }
#0011
#0012  CMyFrameWnd::CMyFrameWnd()
#0013  {
#0014     Create();
#0015  }
#0016
#0017  BEGIN_MESSAGE_MAP(CMyWinApp, CWinApp)
#0018  ON_COMMAND(CMyWinAppid, 0)
#0019  END_MESSAGE_MAP()
#0020
#0021  BEGIN_MESSAGE_MAP(CMyFrameWnd, CFrameWnd)
#0022  ON_COMMAND(CMyFrameWndid, 0)
#0023  END_MESSAGE_MAP()
#0024
#0025  BEGIN_MESSAGE_MAP(CMyDoc, CDocument)
#0026  ON_COMMAND(CMyDocid, 0)
#0027  END_MESSAGE_MAP()
#0028
#0029  BEGIN_MESSAGE_MAP(CMyView, CView)
#0030  ON_COMMAND(CMyViewid, 0)
#0031  END_MESSAGE_MAP()
#0032
#0033  void  printlpEntries(AFX_MSGMAP_ENTRY* lpEntry)
#0034  {
#0035  struct {
#0036    int classid;
#0037    char* classname;
#0038  } classinfo[] = {
#0039                   CCmdTargetid ,  "CCmdTarget   ",
#0040                   CWinThreadid ,  "CWinThread   ",
#0041                   CWinAppid    ,  "CWinApp     ",
#0042                   CMyWinAppid  ,  "CMyWinApp    ",
#0043                   CWndid       ,  "CWnd        ",
#0044                   CFrameWndid  ,  "CFrameWnd    ",
#0045                   CMyFrameWndid,  "CMyFrameWnd  ",
#0046                   CViewid      ,  "CView       ",
#0047                   CMyViewid    ,  "CMyView     ",
#0048                   CDocumentid  ,  "CDocument    ",
#0049                   CMyDocid     ,  "CMyDoc      ",
#0050                   0            ,  "           "
#0051               };
#0052
#0053     for (int i=0; classinfo[i].classid != 0; i++)
#0054     {
#0055        if (classinfo[i].classid == lpEntry->nID)
```

```
#0056        {
#0057            cout << lpEntry->nID << "    ";
#0058            cout << classinfo[i].classname << endl;
#0059            break;
#0060        }
#0061    }
#0062 }
#0063 //--------------------------------------------------------------------
#0064 // main
#0065 //--------------------------------------------------------------------
#0066 void main()
#0067 {
#0068     CWinApp* pApp = AfxGetApp();
#0069
#0070     pApp->InitApplication();
#0071     pApp->InitInstance();
#0072     pApp->Run();
#0073
#0074     CMyDoc* pMyDoc = new CMyDoc;
#0075     CMyView* pMyView = new CMyView;
#0076     CFrameWnd* pMyFrame = (CFrameWnd*)pApp->m_pMainWnd;
#0077     pMyFrame->m_pViewActive = pMyView;
#0078     pMyView->m_pDocument = pMyDoc;
#0079
#0080     // test Message Routing
#0081     cout << endl << "pMyFrame received a WM_CREATE, routing path :" << endl;
#0082     AfxWndProc(0, WM_CREATE, 0, 0, pMyFrame);
#0083
#0084     cout << endl << "pMyView received a WM_PAINT, routing path :" << endl;
#0085     AfxWndProc(0, WM_PAINT, 0, 0, pMyView);
#0086
#0087     cout << endl << "pMyView received a WM_COMMAND, routing path :" << endl;
#0088     AfxWndProc(0, WM_COMMAND, 0, 0, pMyView);
#0089
#0090     cout << endl << "pMyFrame received a WM_COMMAND, routing path :" << endl;
#0091     AfxWndProc(0, WM_COMMAND, 0, 0, pMyFrame);
#0092 }
```

本章回顾

像外科手术一样精准，我们拿起锋利的刀子，划开MFC 坚韧的皮肤，再一刀下去，剖开它的肌理。掏出它的内脏，反复观察研究。终于，借着从MFC 掏挖出来的源代码清洗整理后完成的几个小小的C++ console 程序，我们彻底了解了所谓Runtime Class、Runtime Time Information、Dynamic Creation、Message Mapping、Command Routing 的内部机制。

咱们并不是要学着做一套application framework，但是这样的学习过程确实有必要。因为，「只用一样东西，不明白它的道理，实在不高明」。况且，有什么比光靠三五个一两百行小程序，就搞定对象导向领域中的高明技术，更值得的事？有什么比欣赏那些由Runtime Class 所构成的「类别型录网」示意图、消息的实际流动图、消息映射表的架构图，更令人心旷神怡？

把Frame1~Frame8 好好研究一遍，你已经对MFC 的架构成竹在胸。再来，就是MFC 类别的实际运用，以及Visual C++ 工具的熟练　！