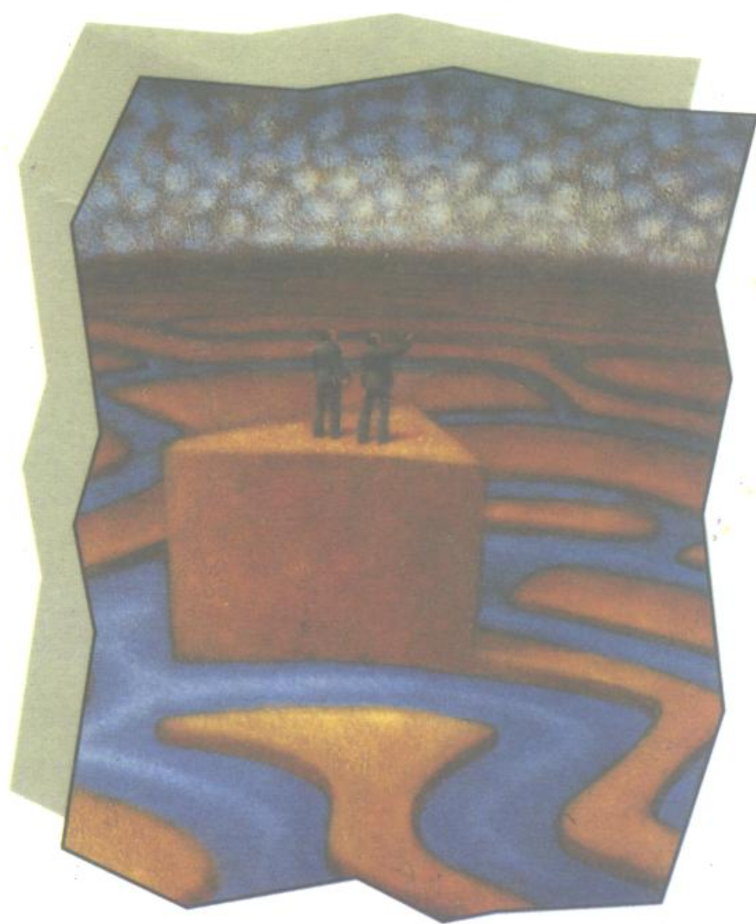
Windows NT
开发指南

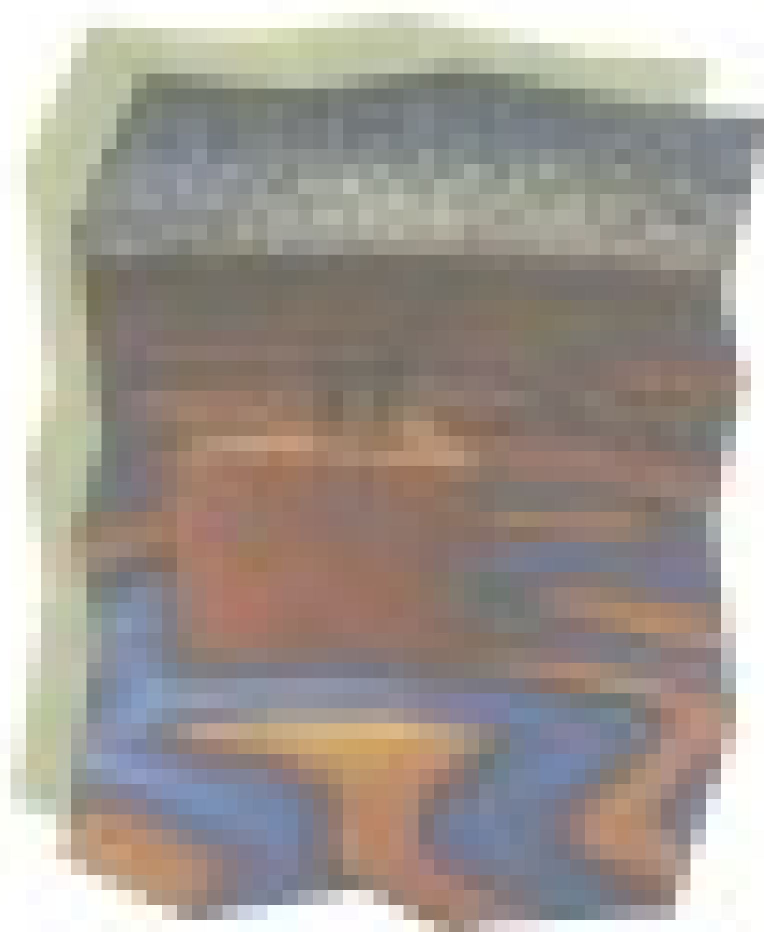


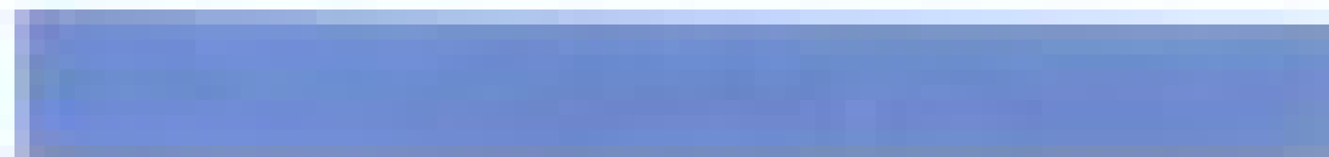
〔美〕K. J. 古德曼 著

.86
/1

科学出版社
龍門書局

Windows NT 开发指南





TP316.86
GDM/1

Windows NT 开发指南

〔美〕 K. 古德曼 著
赵立伟 等 译
燕卫华 校

科学出版社
龍門書局
1997

044297

（京）新登字 092 号

内容简介

Microsoft 公司推出的 Windows NT 超越了 Windows 3.x 作为 DOS 扩展平台的局限，本书可帮助用户了解和掌握 Windows NT，并迅速从 Windows 3.1 程序设计调整到 Windows NT 程序设计。

本书详细介绍了有关线程、32 位程序设计、Windows NT 内存管理、Windows NT 安全性系统结构以及在设计程序时应用这些特性的方法。另外，本书还介绍了 Windows NT 的新特性，如增加的 GDI 函数、寄存器及远程过程调用。此外，还介绍了怎样把程序移植到 Windows NT 中，怎样在 Windows NT 中编写基于文本的程序等等。

本书特别适合于有一些 Windows 经验的软件工程师、程序设计人员、高校计算机专业的研究生、本科生。

版权声明

本书英文版名为：Windwos NT：A Developer's Guide，由 M&T Books 公司出版，版权归 M&T Books 公司所有。本书中文版由 Far East Books 公司授权出版。未经出版者书面许可，本书的任何部分不得以任何形式或手段复制或传播。

Windows NT 开发指南

〔美〕K. 古德曼 著
赵立伟 等 译
燕卫华 校
责任编辑：汪亚文
科学出版社
龙门书局 出版
北京东黄城根北街 16 号
邮政编码：100717
兰空印刷厂印刷
新华书店北京发行所发行 各地新华书店经销
1997 年 2 月第 一 版 开本：787×1092 1/16
1997 年 2 月第一次印刷 印张：18 1/2
印数：1—5000 字数：427 000
ISBN7-5077-005350-8/TP・590
定价：29.90 元

目　录

引言 …… (1)

第一章　进程的生命周期 …… (15)

1.1　仔细研究 CreateProcess …… (15)

1.1.1　子进程能继承些什么 …… (16)

1.1.2　什么不能被继承 …… (17)

1.1.3　决定优先类和进程类型的创建标志 …… (18)

1.2　线程 …… (20)

1.2.1　何时使用附加线程 …… (20)

1.2.2　创建线程 …… (22)

1.2.3　挂起线程的执行 …… (23)

1.2.4　设置线程优先级 …… (24)

1.2.5　使用线程局部存储区 …… (25)

1.2.6　终止线程 …… (27)

1.2.7　什么时候不使用线程 …… (27)

1.3　覆盖的 I/O …… (27)

1.4　结构和 CreateProcess API …… (28)

1.4.1　环境变量 …… (28)

1.4.2　PROCESS_INFORMATION 结构 …… (29)

1.4.3　终止 GUI 进程 …… (29)

1.5　同步 …… (30)

1.5.1　Win16 与 Win32 的差别 …… (30)

1.5.2　同步对象 …… (30)

1.5.3　临界区对象 …… (31)

1.5.4　事件对象 …… (32)

1.5.5　互斥对象 …… (33)

1.5.6　信号量对象 …… (33)

1.5.7　同步和 GDI 对象 …… (42)

第二章　32 位程序设计 …… (44)

2.1　32 位计算的优点 …… (44)

2.1.1　线性程序设计模式 …… (44)

2.1.2　可以得到的更多的数据和地址空间 …… (45)

2.1.3　并非必要的结构复杂的编译器支持 …… (46)

2.1.4　适用于 32 位模式的处理器 …… (47)

2.2　了解目标平台 …… (48)

2.2.1　扩展基本类型 …… (50)

2.2.2　Intel 386/486 寄存器集 …… (51)

2.2.3　弄清参数调用 …… (53)

2.2.4　MIPS R4000 和 R4400 处理器 …… (56)

试读结束，需要全本PDF请购买 www.ertongbook.com

2.2.5 DEC ALpha AXP …… (62)
2.3 小结…… (62)
如何阅读一个 MAP 文件 …… (62)
第三章 关于应用程序移植到 WIN32 API 中的问题 …… (65)
3.1 两个程序库:Win16 的库和 Win32 的库 …… (65)
3.1.1 处理 makefile 问题 …… (66)
3.1.2 数据类型长度的改变 …… (67)
3.1.3 Windows 消息的变动 …… (70)
3.1.4 API 调用的改变 …… (72)
3.1.5 段式存储体系问题 …… (73)
3.1.6 直接访问硬件 …… (73)
3.1.7 直接访问 WIN.INI 与 SYSTEM.INI …… (74)
3.1.8 虚拟设备驱动程序(VxD) …… (75)
3.1.9 对 hprevInstance 的依赖性 …… (75)
3.1.10 输入状态的变动 …… (75)
3.1.11 对 DOS 的依赖 …… (75)
3.1.12 移植汇编语言 …… (78)
3.2 一个代码库:与 Win16 和 Win32 兼容 …… (79)
3.2.1 C 语言途径 …… (79)
3.2.2 GET 移植宏 …… (83)
3.2.3 移植用于控制的宏 …… (83)
3.2.4 轻松的途径:使用应用程序框架 …… (84)
3.2.5 有关 MFC 的一些注意事项 …… (89)
参加一个移植实验室 …… (92)
第四章 高级 GDI 功能 …… (94)
4.1 GDI 的"客户/服务器"性质简介 …… (94)
4.1.1 GDI API 的一些功能变动 …… (95)
4.1.2 被删除的 GDI 功能 …… (96)
4.2 对 GDI 的改进 …… (97)
4.2.1 对直线的改进 …… (97)
4.2.2 对曲线的改进 …… (97)
4.2.3 对弧线的改进 …… (99)
4.2.4 对路径的改进 …… (102)
4.2.5 对位图的改进 …… (107)
4.2.6 增强的元文件 …… (112)
4.3 改进 GDI 作图效果 …… (115)
4.4 为 Win32 应用程序增加三维效果 …… (117)
4.5 小结 …… (119)
第五章 Windows NT 的安全体系 …… (120)
5.1 单个用户验证 …… (121)
5.2 安全标识符 …… (121)

5.3 存取令牌 …… (122)
5.4 安全对象 …… (123)
5.5 控制存取权限 …… (124)
5.6 应用程序示例:SecureView …… (128)
5.7 授予特权 …… (143)
5.8 应用程序示例:ExitWindows …… (144)
5.9 安全体系对应用程序的影响 …… (151)
5.9.1 独立工作站 …… (151)
5.9.2 单域和多域网络 …… (152)
5.10 模仿 …… (152)
5.11 安全性和屏幕保护程序 …… (153)
5.11.1 NT 安全体系 …… (154)
第六章 注册区 …… (155)
6.1 为什么在 NT 中注册是必要的 …… (155)
6.2 注册区的基本结构 …… (157)
6.2.1 关键字和值入口 …… (157)
6.2.2 预定义句柄 …… (158)
6.3 确保 Windows NT 能启动的控制集 …… (161)
6.4 INI 文件映射 …… (162)
6.5 设置环境变量 …… (164)
6.6 如何结构化应用程序数据 …… (165)
6.7 注册区编程 …… (166)
6.7.1 查询注册区 …… (168)
6.7.2 枚举关键字和值 …… (170)
6.7.3 写到注册区 …… (171)
6.7.4 安全注册区入口 …… (171)
6.7.5 保存和从磁盘文件中恢复注册区 …… (171)
6.7.6 连接到远程机器上 …… (172)
6.7.7 卸下 SDK …… (173)
6.7.8 性能数据 …… (174)
6.7.9 获取计数器数据 …… (178)
6.8 示例程序:Registrar …… (179)
6.9 小结 …… (187)
从注册区中抽取的典型项目 …… (187)
第七章 远程过程调用 …… (190)
7.1 什么是远程过程调用 …… (190)
7.1.1 NT RPC 与 OSF/DCE …… (191)
7.1.2 RPC 和数据转换 …… (191)
7.2 开发 RPC 应用程序 …… (192)
7.2.1 利用 Microsoft 接口定义语言定义接口 …… (192)

7.2.2 GUID 结构 …… (193)
7.2.3 使用类型属性自定义类型 …… (195)
7.2.4 RPC 客户 …… (197)
7.2.5 客户 API 联编函数 …… (199)
7.2.6 RPC 服务器 …… (202)
7.2.7 本地远程过程调用 …… (205)
7.3 示例程序:分布式 MAKE …… (205)
7.4 小结 …… (207)
命名习惯 …… (207)
第八章 Win32s …… (209)
8.1 Win32s 是如何工作的 …… (210)
8.1.1 Win32s.EXE:建立一个任务数据库 …… (211)
8.1.2 16 位与 32 位模块间的转换程序 …… (211)
8.1.3 组成 Win32s 系统的文件 …… (211)
8.1.4 可移植可执行文件格式 …… (212)
8.2 确定在什么情况下 Win32s 适合应用程序 …… (214)
8.2.1 用 Win16 代替 Win32s …… (214)
8.2.2 使用 Win32 代替 Win32s …… (215)
8.3 通用转换程序 …… (215)
8.3.1 注册转换程序 …… (215)
8.3.2 注消转换程序 …… (217)
8.3.3 转换指针 …… (217)
8.3.4 示例程序:Exit Windows 32s …… (218)
8.3.5 开发 Win16 程序 …… (228)
8.3.6 从 Windows 3.x 应用程序中调用 Win32 DLL …… (229)
8.3.7 与 Win16 应用程序并存 …… (229)
8.3.8 NotifyRegister 与 Win32s …… (230)
8.3.9 NotifyRegister for Win32 的一个选择 …… (231)
8.4 调试 Win32s 应用程序 …… (231)
8.5 小结 …… (235)
第九章 内存管理 …… (236)
9.1 Windows NT 如何管理虚拟内存 …… (236)
9.2 内存 API …… (238)
9.2.1 虚拟 API …… (238)
9.2.2 VirtualFree …… (239)
9.2.3 监视页 …… (240)
9.2.4 零碎内存解决办法 …… (241)
9.2.5 VirtualQuery …… (241)
9.2.6 VirtualLock …… (242)
9.3 Win32 中的 Global 和 LocalAlloc 函数 …… (242)
9.4 用标准 C 语言库管理内存 …… (243)

9.5 在 Win32 的下新堆 API …… (244)
9.5.1 用于分配更大内存的 HeapRealloc …… (245)
9.5.2 撤消堆的 HeapDestroy …… (246)
9.5.3 MFC C++ new 及 delete 函数 …… (246)
结构异常处理 …… (246)
9.6 管理内存交叉进程 …… (247)
9.6.1 使用内存映射文件进行交叉进程操作 …… (248)
9.6.2 访问共享数据的其它进程 …… (250)
9.6.3 共享内存 DLLs …… (251)
9.6.4 传送只读数据的 WM_COPYDATA …… (252)
9.6.5 ReadProcessMemory 和 WriteProcessMemory …… (253)
9.7 应用程序示例:读写内存 …… (254)
9.8 小结 …… (269)
第十章 编写控制台应用程序 …… (271)
10.1 控制台应用程序中不支持的功能 …… (273)
10.1.1 控制台应用程序中的钩子 …… (273)
10.1.2 定时器和控制台应用程序 …… (273)
10.1.3 控制台与其它窗口的相互作用 …… (274)
10.1.4 分类和控制台应用程序 …… (274)
10.2 控制台应用程序的特性 …… (274)
10.2.1 控制台句柄 …… (275)
10.3 C 运行库对控制台输入和输出的支持 …… (278)
10.3.1 GetLastError 与 errno …… (279)
10.3.2 GreateThread 和_beginthread …… (281)
10.4 在控制台应用程序中使用图形用户界面(GUI)功能 …… (282)
10.5 与图形应用程序相结合的控制台窗口 …… (284)
10.5.1 向控制台窗口的直接输出 …… (284)
10.5.2 取得控制台输入 …… (286)
10.5.3 重新定向标准输入/输出 …… (287)
10.6 检测事件 …… (293)
10.7 使用定时器 …… (295)
10.8 独立进程 …… (296)
10.9 从控制台应用程序打印 …… (297)
10.9.1 运行时确定控制台应用程序 …… (301)
10.10 小结 …… (303)

引　言

NT的意思是"new technology(新技术)",不过,在Windows NT的后面隐藏着一段历史。我一直希望在1987年推出的是Windows NT而不是OS/2。计算机需要新一代操作系统,其原因是明显的。现在需要一种新的操作系统,我并不是知道这一点的唯一的人。

微软公司知道几百万台计算机上的DOS操作系统已经到达其顶峰了。DOS的局限性是明显的。为8088芯片设计的DOS受限于实模式和单线程。对应用程序和DOS本身而言什么情况都可能发生并会导致死锁。由于以前把视频内存的段址定为A000h,所以DOS将应用程序的地址空间限制为640KB。

微软公司在1987年以前曾试图延长DOS的寿命,不过这些努力仅仅取得了有限的成功。所以在1987年,微软公司与IBM公司联手推出了OS/2操作系统。同时,IBM也推出了PS/2计算机。PS/2是IBM的新一代个人计算机,PS/2使用专有总线,所以不再与AT总线PC机上的板卡兼容。同样,OS/2在设计上也存在同样的缺陷。PS/2需要用户放弃现有的硬件资源,而OS/2 1.0版需要用户放弃现有的软件资源。所以尽管OS/2提供了与DOS兼容的模块,但是由于操作系统是基于Intel 80286芯片的,所以用户能分别运行OS/2或DOS应用程序,但是却不能同时运行它们。

OS/2 1.0使用了显示管理(PM)应用程序设计接口(API),这与Windows 1.x API不兼容。更糟糕的是,Windows 2.0保留了与Windows 1.x一致的API,由于实际情况的限制,微软公司决定不在Windows上采用PM API。对于那些刚把应用程序从Macintosh转向Windows的独立软件商,微软公司不能抽掉他们脚下的地毯。因为没有哪家公司愿意为同一个操作系统两次完全重写应用程序。所以Windows 2.0只能对Windows 1.x API集进行扩展。这么一来,用户将不能在OS/2中运行自己喜欢的DOS应用程序了,因为其字符模式与DOS API不相容,而其图形用户接口(GUI)API又与Windows API不相容。尤其糟糕的是,一些开发商和最终用户把两种产品名字中的/2混为一谈,很多人以为只有PS/2才能运行OS/2,当然这是错误的。OS/2 1.x只要求80286(或以上)计算机,它并不只局限于IBM 80286计算机。

为什么微软这样的公司会错过这个机会呢?对初学者来说,PM API比Windows API要好学得多。用户很难从Windows API的函数名中分辨出该函数是哪一类函数,Windows 3.1主要由三个动态链接库组成:KERNEL.EXE,GDI.EXE和USER.EXE,每个库都是函数集。Windows API的函数名字和其所属的模块之间没有明显的联系。函数名也不是分类排列的,因而在手册中很难寻找出某个函数,也很难得到有关函数的帮助信息。大多数的微软开发者指南都需要读者事先找出该函数应归为哪一类型。

除了API命名的缺陷外,Windows 1.0的设计者是值得赞扬的。1.0 API设计者的工作受到相当大的限制,因为在实模式下模拟保护模式和切换程序段是一件令人可畏的工作。出于性能的考虑,Windows 1.0不得不独立出会话进程、窗口进程和按钮,不得不独立出只读代码段。但是,要是用户处于PM API之中,那就可以不受限于实模式了。我想,读者是会改

动它的。

PM API 的设计者对这些缺陷已经做了修改。第一，把函数名字按逻辑顺序分组，例如 PM 的底层 API 都加上了 DOS 前缀，包括 DosOpen 和 DosClose；而所有的窗口管理函数都加上了 WIN 前缀，包括 WinGetMsg 和 WinCreat。第二，去掉了键盘管理、按钮、对话框等。OS/2 中只有窗口，对话框是窗口，按钮也是窗口。OS/2 的设计者力求使软件开发者喜爱这种新的 API，并希望他们愿意把所有 DOS/Windows 代码改写为 OS/2 代码。这是一项繁琐的工程，所以只有少数应用程序转换为 OS/2 代码，OS/2 并没有象盖茨预言的那样卖出去成千套，而 Windows 3.0 就不一样了。

因为 Windows 3.0 是第一个可以运行 MS-DOS 程序的图形环境，所以 Windows 3.0 取得了非常大的成功。采用 Intel 80386 处理器增强模式的优点，DOS 程序可以在 Windows 桌面的一个窗口中运行。另外，Windows 直接运行在 DOS 上面，这也消除了用户在选择 OS/2 还是 DOS 时的忧虑，并且 Windows 3.0 API 是 Windows 2.x API 的超集，因而 2.x 应用程序都可以直接在 3.0 中运行。所以，Windows 3.0 在市场上取得了最大的成功。

尽管这样，Windows 3.0 还是含有不可恢复的应用程序错误(UAE)，很让人恼火(UAE 用于保护系统的崩溃，但是 Windows 3.0 中的 UAE 有时会破坏其它应用程序和 Windows 本身)。除此之外，Windows 3.0 可说是最好的了，所以程序开发者广泛接受了它。然后到了 IBM 与微软的分裂，如果你不是在“铁幕”后面或者说没有投身 UNIX 之中的话，那么应该知道两个公司的分歧最终解散了他们之间联手开发软件的联盟。IBM 不再联合开发下一版的 OS/2，而是决定单独开发操作系统(OS)。而微软公司继续开发其 Windows 并取得了称为“可移植的 OS/2”或“NTOS/2”的 OS/2 未来版本的控制权。

IBM 的 OS/2 版本 2.0 紧接着进入了 PC 操作系统领域。OS/2 2.0 是 32 位操作系统，至少需要 Intel 386 或 486 计算机。PM1.x 应用程序能够在 OS/2 2.0 下运行，但是没有在 API 一级的兼容。IBM 推出 OS/2 2.0 时，另外五千个 Windows 3.0 应用程序已经写成。那些想把应用程序移植到 PM API 的开发者也看到了 OS/2 2.0 的局限性。尽管底层的 OS 是抢占式的，PM 仍然是非抢占式的。多线程的 PM 应用程序必须时刻注意，以免被挂起来。例如，因为非消息队列线程不能将消息发给消息队列线程，所以开发者不得不时常注意输入状态。设计于 386 及 486 平台上的 OS/2 2.0 自称是 32 位 OS。不过，为了尽快出台，其设备驱动程序和图形系统仍然是 16 位的。

许多开发者会问到这个问题：

- 如果高档 486 PC 机上执行应用程序不够快，那么转移到多 486 芯片的 PC 机上会怎么样呢？
- 这些应用程序能转移到其它类型芯片的机器上吗？

不幸的是，如果不重写这些应用程序，以上两个问题的答案是“可能不行”。OS/2 2.0 是用 x86 平台的汇编语言写成的。多处理器的支持必须在 OS 中设计到，它不能够轻易地附加上去(OS/2 的 LANMAN1.x 可以在第二个处理器上运行)。不过，比起转换到新的平台，多处理器的支持要容易得多，所以在 OS/2 能运行于其它类型的处理器之前，没准 OS/2 已经能支持多 486s 的处理了。

正当这些问题还在困扰着未来的 OS/2 2.0 开发者时，微软公司又推出了 Windows 3.1。Windows 新版本的 API 支持多媒体和笔记本计算机。此外，新增的参数有效检测使得

试读结束，需要全本PDF请购买 www.ertongbook.com

应用程序要弄垮 Windows 极其困难。微软公司在 Windows 3.1 中增加了参数有效检测，以尽量减少在内核、GDI 和用户 DLL 中的异常。参数有效检测确保不会出现假句柄和错误指针，所以 Windows 本身不会崩溃。不过，要按照 API 建立一个错误指针的方式来放弃错误指针是很困难的。当发生一般保护(GP)错误时，Windows 3.1 不再显示 UAE 框，而是弹出指令的代码段址和偏移址。

我一直没有把 Windows 称为一个操作系统。尽管微软公司坚持称 Windows 是 DOS 扩展器，但它并不是操作系统。DOS 扩展器在实方式下从 DOS 命令启动，并自动切换到保护方式。Windows 只能称作最好的 DOS 扩展器，但却不能冠之以操作系统。当开发者越来越熟悉 Windows 3.1 之后，他们会发现它只不过是 DOS 上面的一层漂亮的装饰，而 DOS 是我们首先要踢掉的操作系统。此外，Windows 3.1 还有一些设计上的缺陷，除了与 DOS 有关的缺陷之外，还有 OS/2 2.0 也同样具有的缺陷，如单线程的输入模式、非抢占式以及只能在一个 x86 处理器上运行等。

下面在程序 0-1 中列出的代码段称为终端程序(Terminater)，它说明了要冻结 Windows 3.1 是件很容易的事。如果是 Windows 开发者，或许你的代码已经对 Windows 造成过大破坏了。这段代码是为了显示人的弱点，诸如忘记关掉捕获程序，或者忘记让步，这些都会导致 Windows 的崩溃。读者用不着亲自在 Windows 3.1 下运行这个程序，如果执行终端程序，必须按 Ctrl-Alt-Del 来结束它。否则，Windows 只会处于冻结状态，按 Ctrl-Esc 不管用，Alt-Tab 不管用，Alt-Esc 也不灵。

程序 0-1　终端程序(Terminater)

```
//

#include <afxwin.h>
#include "resource.h"

#include "term.h"
    CString s = "You are Terminated!";
////////////////////////////////////////////////////////////////////////////
// theApp:
// Just creating this application object runs the whole application.
//
CTheApp theApp;
////////////////////////////////////////////////////////////////////////////
// CMainWindow constructor:
// Create the window with the appropriate style, size, menu, etc.
//
CMainWindow::CMainWindow()
{
    LoadAccelTable( "MainAccelTable" );
    Create(NULL, "The Terminator",
           WS_OVERLAPPEDWINDOW, rectDefault, NULL, "MainMenu" );
}
```

```
// OnPaint:
// This routine draws the string "Hello, Windows!" in the center of the
// client area. It is called whenever Windows sends a WM_PAINT message.
// Note that creating a CPaintDC automatically does a BeginPaint and
// an EndPaint call is done when it is destroyed at the end of this
// function. CPaintDC's constructor needs the window (this).
//
void CMainWindow::OnPaint()

{
    CPaintDC dc( this );
    CRect rect;
    GetClientRect( rect );
    dc.SetTextAlign( TA_BASELINE | TA_CENTER );
    dc.SetTextColor( ::GetSysColor( COLOR_WINDOWTEXT ) );
    dc.SetBkMode(TRANSPARENT);
    dc.TextOut( ( rect.right / 2 ), ( rect.bottom / 2 ),
                s, s.GetLength() );
}

// OnAbout:
// This member function is called when a WM_COMMAND message with an
// IDM_ABOUT code is received by the CMainWindow class object. The
// message map below is responsible for this routing.
//
// We create a CModalDialog object using the "AboutBox" resource (see
// term.rc), and invoke it.
//
void CMainWindow::OnAbout()
{
    CModalDialog about( "AboutBox", this );
    about.DoModal();
}
// CMainWindow message map:
// Associate messages with member functions.
//
// It is implied that the ON_WM_PAINT macro expects a member function
// "void OnPaint()".
//
// It is implied that members connected with the ON_COMMAND macro
// receive no arguments and are void of return type, e.g., "void OnAbout()".
//
```

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

////////////////////////////////////////////////////////////////////////////////
// CTheApp

// InitInstance:
// When any CTheApp object is created, this member function is automatically
// called.  Any data may be set up at this point.
//
// Also, the main window of the application should be created and shown here.
// Return TRUE if the initialization is successful.
//
BOOL CTheApp::InitInstance()
{

    TRACE( "HELLO WORLD\n" );

    m_pMainWnd = new CMainWindow();
    m_pMainWnd->ShowWindow( m_nCmdShow );
    m_pMainWnd->UpdateWindow();
    m_pMainWnd->Invalidate(TRUE);
    m_pMainWnd->UpdateWindow();

    m_pMainWnd->SetCapture();
    for (;;);   // The user intended to have a loop that did some lengthy
                // rpocessing so he turned the capture on to get stray keystrokes
                // Unfortunately a stray semicolon will cause this to loop forever

    .
    .     // Some long processing goes here.
    .
    return TRUE;
}
// term.h :Declares the class interfaces for the application.
//         Term is a simple program which consists of a main window
//         and an "About" dialog which can be invoked by a menu choice.
//         Due to a bug in program logic Term accidently goes into a loop
//         after setting the capture on.
#ifndef _TERM_H_
#define _TERM_H_
```

```
/////////////////////////////////////////////////////////////////////////////

// CMainWindow:
// See term.cpp for the code to the member functions and the message map.
//
class CMainWindow : public CFrameWnd

{
public:
    CMainWindow();

    afx_msg void OnPaint();
    afx_msg void OnAbout();

    DECLARE_MESSAGE_MAP()
};

/////////////////////////////////////////////////////////////////////////////

// CTheApp:
// See term.cpp for the code to the InitInstance member function.
//
class CTheApp : public CWinApp
{
public:
    BOOL InitInstance();
};
/////////////////////////////////////////////////////////////////////////////
#endif // _TERM_H_
```

我从GENERIC开始，只增加了一些函数调用就冻结了Windows。我想要是其它应用程序都这样的话，那是卖不出去很多拷贝的，这种例外仅仅是偶发的。请留心这段程序，因为在讲述了NT之后我们还会遇到它。

这一切引来了Windows NT。Windows NT实际上就是我先前提到的可移植的OS/2。读者认为NT是基于哪个API呢？基于OS/2 API还是基于一个月销售百万份拷贝的API呢？当微软公司与IBM公司分裂时，微软公司就把Windows NT的发展重点从基于OS/2的API转移到一种新的API上。这种新的操作系统和新的API使得微软公司能建立于Windows的成功之上，进一步说是赎了先前的罪过。微软公司不希望两次犯同一种错误，也不想放弃忠诚的开发者和用户群。这样读者就会理解这一点，即在本书中给出的新的API函数，当然其名字并不是重要的。当读者看到那些独立的对话框进程和窗口进程例子时，就会理解这一切都是为了兼容性。

0.1　Windows NT 总览

在了解了 NT 的发展历史以后，让我们来看看 NT 的特性。下面讨论了 NT 与 Win16 的兼容性，NT 的多线程处理结构以及独立地址空间的使用和 NT 的稳定性。

0.1.1　NT 与 Win16 的兼容性

"兼容"是一件说起来容易做起来难的事情，NT 看上去很像 Windows 3.1，所以 NT 的正式名称是"Windows NT 3.1"。微软公司责无旁贷要保护用户的投资，而且相当多的公司都以 Windows 为标准，那种认为他们会愿意转换到另外的用户接口（UI）的想法是很愚蠢的。而 NT 不仅仅看上去像 Windows 3.1，而且 NT 也能运行 Windows 应用程序。此外，NT 还能运行 DOS、LANMAN、基于字符的 POSIX 以及基于字符的 16 位 OS/2 应用程序。用户不需要在 NT 与其它的 OS 之间选来选去。这是微软公司的市场策略，OS/2 只能运行在 Intel 平台上，而 NT 可以在任何平台上支持 DOS、LANMAN 和 POSIX。微软公司不需要用户为这些操作系统开发新的应用程序，相反，由于 NT 支持 DOS、OS/2 及 POSIX，因而用户可以用所喜爱的工具和编辑器来编写新的 NT 应用程序。那就是说，如果想用到非常喜爱的 DOS 或 OS/2 应用工具，那么在 NT 版应用工具推出之前，用户可以自己填补这个空白。

除了考虑到其它公司对 Windows 的投资外，微软公司的设计人员还要保护独立软件商（ISV）的投资，因为他们中的大部分人都采用了 Windows API。微软创造了一种新的称作 Win32 的 API，使得 ISV 能容易地把应用程序移植到 NT 中。Win32 基本来说是 Windows 3.x API 的 32 位版本，称为 Win32，微软称 Windows 3.x API 为 Win16。Win32 的主要目标是与 Win16 的向上兼容，用户需要做的工作仅仅是重新编译代码。比起把应用程序从 Win16 移植到 PM 中，显然从 Win16 移植到 Win32 要容易得多。Win16 到 Win32 几乎不用做什么语法上的改变，最大的改变是把许多参数从 16 位扩增至 32 位。此外，Win32 中的所有 API 都返回一错误状态。使用 GetLastError 可得到扩展的错误支持，这使得一些 API 与 Win16 有所改变，同时这也意味着如果代码检测了返回状态，那么就不会使用无效句柄。在第三章"把应用程序移植到 Win32 API 中"，解释了把应用程序从 Win16 移植到 Win32 可能涉及到的问题。

兼容性是重要的，但是如果一个操作系统没有新的特征，那么没有人会买它。Win32 引入了 Win16 中所不具备的特征和函数。Win32 可防止应用程序间的破坏，所以它是真正在底层的操作系统。其盾牌功能使得 NT 明显有别于 DOS 或 OS/2。正如我前面提到的，NT 是基于 32 位的，对于习惯了基于 Intel 系统 16 位段址的程序员，这一扩展会给他们带来些不适。不管怎样，段的概念是完全消失了，偏移地址是 32 位的，因此 16 位段址对开发者来说就失去了意义，当然段寄存器也失去了意义。那就是说，不用再去关心 CS、DS 和 SS 段寄存器的值，也不用在编辑时写这些代码。回想当初在 DOS 下编程的日子，微模式下代码、数据、堆栈段连起来形成一个 64KB 的块。而在 NT 中，微模式就是 4GB 长。只要愿意，可以定址超过 64KB 的内存。操作系统并不需要绕过一道弯才给程序分配地址。如果用户愿意，可以在 C 运行库中使用 malloc 来获得 20MB 的内存，这在 Windows 3.x 中是不可能的。第二章"32 位程序设计"，详细介绍了 32 位的程序设计方法。

0.1.2 NT的多线程进程结构

Windows 3.x 中另一个主要的改进是NT的多线程进程结构。在Windows 3.x中，每段可执行进程对应了一个可执行的线程，因而每个可执行进程只有一个线程，所以“线程”和“进程”是同义的并可互换。只有拥有了多个进程，才能有多个线程。Windows 3.x应用程序有时会生成几个进程，而每个进程作为独立的可执行段来运行，但是Windows 3.x只有一个地址空间，所以进程可以获取其它进程的句柄并与它们共享内存。在Windows NT中，每个进程有多个线程，并且每个线程由NT的规划器来抢占式地安排。OS/2中每个进程也对应于多个线程，这在概念上与NT是一致的，差别在于NT中的进程运行在独立地址空间中，而OS/2的进程则运行在同一地址空间中。在第一章“进程的生命周期”中，我们会详细讨论进程、线程和同步。

0.1.3 独立地址空间

千万不要被独立地址空间及保护地址空间这些字眼给唬住，NT不允许相互交叉的进程操作。如果非做不可的话，那么必须事先给出安全保证。在第五章“Windows NT的安全体系”中，我会告诉读者怎样获得安全保证。具有优先级别的线程可以从其它进程中读或写信息。子系统也能控制客户进程的虚机器，并能读写客户的地址空间。例如，POSIX子系统可以挂起新一段进程，然后为进程找出栈，再改变栈。POSIX子系统用这种方式来传递信号。既然NT不再依靠MS-DOS来提供底层函数，如创建文件等，所以API代替了DOS和Windows 3.x的旧INT 21H函数调用。本书中，我会经常用到几个这些API调用的例子。

0.1.4 稳定性

对UAE和系统暂停有哪些改进呢？如果操作系统允许犯错误的应用程序破坏整个系统，那么所有的优点和32位的API就都毫无意义了。为了解决这个问题时，NT系统采用了客户/服务器模型以确保没有进程能够偶然或故意地破坏操作系统。Win32是子系统，所有的子系统都在用户模式下工作。所谓用户模式，是指核心服务器(称作核心模式)的客户。NT在第一版本提供了几个子系统，Win32是其中之一。其它还有POSIX和OS/2。DOS和Windows 3.x应用程序运行在WOW(Windows on Windows)之中。WOW并不是真正的子系统，只是用户模式的应用程序。NT也使用一种异步输入模型，那样如果某个应用程序取得控制权并且不响应其它应用程序的请求(没有调用PeekMessage或GetMessage)，那么最坏的情况是把自己挂起来，而其它应用程序根本就不会关注这个犯了错误的应用程序，它们会继续正常运行下去。这样，任何产生一般保护(GP)错误的应用程序挂起了，而系统正常运转。所以当用户在NT中试验终端应用程序时，将得到不同的结果。由于完全的抢占式，所以无论一个应用程序做了什么动作，其它的应用程序都能获得自己的时间片。

直到想运行第二个16位应用程序，NT才通知终端程序被停止了。因为所有的Win16应用程序在同一地址空间运行，所以当运行第二个16位应用程序时，屏幕显示如图0-1所示的对话框，表明Win16子系统失去反应。

试读结束，需要全本PDF请购买 www.ertongbook.com